Contents

7

Graph Algorithms

LEDA offers a wide variety of graph algorithms. Starting in the third section of this chapter we discuss depth-first and breadth-first search, algorithms to compute graph decompositions, and algorithms for shortest paths, matchings in bipartite and general graphs, maximum flows, and minimum cuts. For each class of algorithms we first discuss their functionality and then discuss implementations. In many cases we also derive a checker of correctness.

The first two sections of this chapter are orthogonal to the other sections of the chapter. They deal with general considerations for algorithms on weighted graphs. In Section 7.1 we discuss the use of template functions for such algorithms and in Section 7.2 we discuss the requirements on the underlying arithmetic. Both sections can be skipped on first reading.

7.1 Templates for Network Algorithms

Many graph algorithms operate on graphs whose nodes or edges have an associated weight from some number type. For example, the single-source shortest-path algorithm operates on an edge-weighted graph and computes for each node its distance from the source. The algorithm works for any linearly ordered number type. It is natural to formulate it as a template function.

```
template <class NT>
bool DIJKSTRA_T(const graph& G, node s, const edge_array<NT>& c,
                node_array<NT>& dist, node_array<edge>& pred);
```

The template parameter *NT* can be instantiated with any number type[1]. The most frequent instantiations are with the built-in number types *int* and *double* and the LEDA number types *integer* and *real*. It is desirable that:

- the most frequent instantiations are pre-compiled, as this reduces the compilation time of application programs and allows us to distribute object code instead of source code to all those users, who do not need instantiations with other number types, and that
- the pre-instantiated versions can be used side by side with the template version.

We describe our mechanism to achieve these goals. We use the shortest-path algorithm as our running example. We write three files: dijkstra.h, dijkstra.t, and _dijkstra.c, which are contained in the directories LEDAROOT/incl/LEDA, LEDAROOT/incl/LEDA/templates, and LEDAROOT/src, respectively.

The file dijkstra.h contains the prototypes of all functions. We distinguish the template version and the pre-instantiated versions of a function by the suffix _T in the function name. Thus

⟨*dijkstra.h*⟩≡

```
#ifndef DIJKSTRA_H
#define DIJKSTRA_H
#include <LEDA/graph.h>
template <class NT>
void DIJKSTRA_T(const graph& G, node s, const edge_array<NT>& c,
                node_array<NT>& dist, node_array<edge>& pred);
/* next come the pre-instantiated versions */
void DIJKSTRA(const graph& G, node s, const edge_array<int>& c,
              node_array<int>& dist, node_array<edge>& pred);
// and, similarly, for double, ...
#endif
```

The file dijkstra.t contains the definition of the template function.

⟨*dijkstra.t*⟩≡

```
#include <LEDA/dijkstra.h>
template <class NT>
void DIJKSTRA_T(const graph& G, node s, const edge_array<NT>& c,
                node_array<NT>& dist, node_array<edge>& pred)
{
  /* implementation of DIJKSTRA_T */
}
```

[1] The number type must, of course, satisfy certain syntactic and semantic requirements, e.g., there must be a linear ordering defined on it and addition must be monotone.

The file _dijkstra.c contains the implementations of the instantiations in terms of the template function.

⟨*_dijkstra.c*⟩≡

```
#include <LEDA/templates/dijkstra.t>
void DIJKSTRA(const graph& G, node s, const edge_array<int>& c,
              node_array<int>& dist, node_array<edge>& pred)
{
  DIJKSTRA_T(G,s,c,dist,pred);
}
// and, similarly, for double ...
```

Observe the include statement. As mentioned already, all files containing definitions of template functions are collected in the subdirectory *templates* of the LEDA include directory.

The file _dijkstra.c is pre-compiled into the object file _dijkstra.o, which is included in one of the object libraries of the LEDA system.

We next discuss how to use the pre-instantiated and the template versions of the shortest-path algorithm.

In order to use one of the pre-instantiated versions, one includes dijkstra.h into the application program, for example,

⟨*foo.c*⟩≡

```
#include <LEDA/dijkstra.h>
// define G, s, c, dist, pred with number type int
DIJKSTRA(G,s,c,dist,pred);
```

In order to use the template version, one includes templates/dijkstra.t into the application program, as, for example, in

⟨*foo.c*⟩+≡

```
#include <LEDA/templates/dijkstra.t>
// define G, s, c, dist, pred for any number type NT
DIJKSTRA_T(G,s,c,dist,pred);
// define G, s, c, dist, pred for number type int
// and use template version
DIJKSTRA_T(G,s,c,dist,pred);
// use pre-instantiated version
DIJKSTRA(G,s,c,dist,pred);
```

Observe that there is no problem to use one of the pre-instantiated versions and the template version side by side in an application program such as foo.c.

We nevertheless recommend a different strategy. We suggest that the t-files are not included directly into application programs, as t-files may contain the definitions of auxiliary functions which might clobber the name space of the application program. We rather recommend to define intermediate files as shown next.

In order to instantiate DIJKSTRA_T for a particular number type, say the LEDA number type *real*, we recommend defining files

⟨*real_dijkstra.h*⟩≡

```
#include <LEDA/real.h>
void DIJKSTRA(const graph& G, node s, const edge_array<real>& c,
              node_array<real>& dist, node_array<edge>& pred)
```

and

⟨*real_dijkstra.c*⟩≡

```
#include "real_dijkstra.h"
#include <LEDA/templates/dijkstra.t>
void DIJKSTRA(const graph& G, node s, const edge_array<real>& c,
              node_array<real>& dist, node_array<edge>& pred)
{
  DIJKSTRA_T(G,s,c,dist,pred);
}
```

to include the former in application programs, to pre-compile the latter, and to add the object file real_dijkstra.o to the set of objects for the linker. The alternative strategy has the advantage of introducing no extraneous names into application programs.

We summarize: functions whose name ends with _T are function templates. In order to use them one must include a file LEDA/templates/X.t. The pre-instantiated functions have the same name except for the _T. In order to use them one needs to include a file LEDA/X.h.

7.2 Algorithms on Weighted Graphs and Arithmetic Demand

Many algorithms of this chapter operate on weighted graphs and work for any number type *NT*. The algorithms use additions, subtractions, comparisons, and in rare cases multiplication and division. The correctness proofs of the algorithms rely on the laws of arithmetic and hence the algorithms are only correct if the implementation of the number type obeys the laws of arithmetic.

The two most commonly used number types are *int* and *double*. Unfortunately, both types do not guarantee that the basic arithmetic operations obey their mathematical laws. For example, *int*-arithmetic may overflow and wrap around[2] and *double*-arithmetic incurs rounding error, see Chapter 4. It is therefore not at all obvious that an instantiation of a network algorithm with types *int* or *double* will work correctly. Sections 4.1 and 7.10.5 contain examples of what can go wrong.

[2] Execute `cout << MAXINT + MAXINT;`

We use the following two-step approach to guarantee correctness.

Step 1: We analyze the arithmetic demand of our algorithms. We state clearly which operations must be supported by the number type (that's easy to do, since a simple inspection of the code suffices) and we prove theorems of the following form: if all input weights are integers whose absolute value is bounded by B, then all numbers handled by the algorithm are integers whose absolute value is bounded by $f \cdot B$. We call such an algorithm *f-bounded*. For example, we will show that the maximum weight bipartite matching algorithm is 3-bounded and that the maximum weight assignment algorithm is $4n$-bounded, where n is the number of nodes of the bipartite graph.

Step 2 for type int: In the instantiation of a network algorithm for type *int*, we check that all input weights w satisfy $f \cdot w \leq MAXINT$. If not, we write an appropriate message to diagnostic output. If yes, step 1 guarantees correctness of the computation.

We give an example. We mentioned already that the maximum weight bipartite matching algorithm is 3-bounded. The instantiation is therefore as follows:

⟨*instantiation for ints*⟩≡

```
list<edge> MAX_WEIGHT_BIPARTITE_MATCHING(graph& G,
                         const edge_array<int>& c, node_array<int>& pot)
{ int W = MAXINT/3;
  check_weights(G,c,-W,W,"MWBM<int>");
  return MAX_WEIGHT_BIPARTITE_MATCHING_T(G,c,pot);
}
```

where

⟨*scale_weights.h*⟩+≡

```
inline bool check_weights(const graph& G, const edge_array<int>& c,
                          int lb, int ub, string inf)
{ edge e;
  bool all_edges_ok = true;
  forall_edges(e,G)
    if ( c[e] < lb || c[e] > ub ) all_edges_ok = false;
  if ( !all_edges_ok ) cerr << inf << ": danger of overflow.\n";
  return all_edges_ok;
}
```

There is a similar function for node arrays.

Step 2 for type double: The problem with *double*-arithmetic is round-off error. Round-off errors invalidate the correctness and termination proof and hence a "naive" instantiation of a network algorithm with the number type *double* may run forever, terminate with a run-time error, terminate with an incorrect result, or terminate with the correct result.

It would be nice if we could guarantee that no rounding occurs during a computation, as this will guarantee termination and the absence of run-time errors. It does not guarantee by itself that the result produced has any relationship to the correct result. We come back to this point below.

We can avoid rounding by scaling the input weights appropriately. We replace any input weight w by $sign(w) \cdot \lfloor |w| \cdot S \rfloor / S$, where the *scaling parameter* $S = 2^s$ is a suitable power of two. We use the *same* scaling parameter for all input weights. This has the effect that, after scaling, all input weights are of the form $w' \cdot 2^{-s}$, where w' is an integer. Hence floating point arithmetic will incur no rounding error as long as all intermediate results are of the form $z \cdot 2^{-s}$, where z is an integer that fits into the mantissa of a floating point number. It remains to choose s.

Let C be the maximum absolute value of any input weight. Since the division by 2^s effects only the exponent of a floating point number, we may as well assume that every input weight w is replaced by $sign(w)\lfloor |w| \cdot S \rfloor$. This will turn all inputs into integers and hence step 1 guarantees that the absolute value of all intermediate results is bounded by $f \cdot \lfloor C \cdot S \rfloor$ in the case of an f-bounded algorithm. If we choose s such that all intermediate results can be represented exactly as a double precision floating point number then the computation will incur no rounding error. This is the case if

$$f \cdot \lfloor C \cdot S \rfloor < 2^{53},$$

since double precision floating point arithmetic can represent all integers in the range $[-(2^{53}-1) .. 2^{53}-1]$. Observe that double precision floating point arithmetic uses a 52-bit mantissa and that a floating point number with mantissa $m_1 m_2 \ldots m_{52}$ and exponent 52 represents the integer

$$(1 + \sum_{1 \le i \le 52} m_i 2^{-i}) \cdot 2^{52}.$$

The inequality $f \cdot \lfloor C \cdot S \rfloor < 2^{53}$ is certainly satisfied if

$$f \cdot C \cdot S < 2^{53}$$

or

$$s < 53 - \log(f \cdot C).$$

We summarize:

Lemma 1 *Consider an f-bounded algorithm, let C be the maximum absolute value of any input weight, and let S be a power of two such that $f \cdot C \cdot S < 2^{53}$. If every input weight w is replaced by $sign(w)\lfloor |w| \cdot S \rfloor$, then the algorithm will incur no rounding error in a computation with doubles and hence computes the correct result for the scaled inputs weights.*

What is the relationship between the result for the scaled input weights and the result for the original input weights? We can make no general claim. However, there are many situations where one can claim that the result for the scaled inputs is a good approximation for the result on the unscaled inputs. For all but one network problem considered in this chapter, namely min-cost flow, the objective value is a sum of input weights; for example, the cost of a shortest path is a sum of edge weights, the cost of a matching is a sum of edge weights, and the maximum flow in a network is the minimum capacity of a cut and hence a sum of edge weights. Assume that the objective value is the sum of at most L weights. For any set of at most L weights the sum of the scaled weights and the sum of the unscaled weights differs by at most L/S, since for any individual weight the difference is at most $1/S$. If S is chosen as the largest power of two such that $S < 2^{53}/(f \cdot C)$, then $S \geq 2^{52}/(f \cdot C)$ and hence the maximum absolute error in the objective function is at most $L \cdot f \cdot C \cdot 2^{-52}$. We summarize in:

Lemma 2 *Under the hypothesis of the preceding lemma and the additional assumption that the algorithm computes an objective value, which is the sum of at most L input weights, the maximum absolute error in the objective function is at most* $L \cdot f \cdot C \cdot 2^{-52}$.

Let us give an example. Consider the maximum weighted matching algorithm for bipartite graphs. This algorithm is 3-bounded and the value of a matching is the sum of at most n edges, where n is the number of nodes of the graph. The maximum absolute error is therefore at most $3 \cdot C \cdot 2^{-52}$.

Observe that Lemma 2 bounds the absolute error in the objective function, but not the relative error. We can make no general claims about the relative error. It must be studied individually for each algorithm.

In order to compute s and to scale the input weights, we use the functions *frexp*, *ldexp*, and *floor* from the math-library. Let $x = f \cdot C$.

```
double frexp(double x, int* exp);
```

returns a double y such that y is a double with magnitude in the interval [1/2, 1) or 0, and x equals y times 2 raised to the power *exp* (more precisely, $*exp$). If x is 0, both parts of the result are 0.

Thus, if x is non-zero, then $\log|x| = exp - \epsilon$ where $0 < \epsilon \leq 1$ and hence

$$53 - \log(f \cdot C) = 53 - exp + \epsilon.$$

We therefore choose s as

$$s = 53 - exp.$$

If $C = 0$ and hence $x = 0$, the choice of s is arbitrary. We will set s to 53 in this case. The following procedures implement the computation of s and S. We also compute $1/S$, as it will be convenient to have it around.

⟨*scale_weights.h*⟩+≡

```
#include <math.h>
inline int compute_s(double f, double C)
{
  int exp;
  double x = frexp(f*C,&exp);
  return 53 - exp;
}
```

```
inline double compute_S(double f, double C, double& one_over_S)
{
  int exp;
  double x = frexp(f*C,&exp);
  one_over_S = ldexp(1,exp - 53);
  return ldexp(1,53 - exp);
}
```

where

```
double ldexp(double x, int exp);
```

computes the quantity $x \cdot 2^{exp}$.

How can we compute $w' = sign(w) \cdot \lfloor |w| \cdot S \rfloor / S$? We use

```
double floor(double x);
```

which computes the largest integral value not greater than x.

⟨*scale_weights.h*⟩+≡

```
inline double scale_weight(double w, double S, double one_over_S)
{
  if ( w == 0 ) return 0;
  int sign_w = +1;
  if ( w < 0 ) { sign_w = -1; w = -w; }
  return sign_w * floor(w * S) * one_over_S;
}
```

Let us see scaling at work. We use again the weighted matching algorithm for bipartite graphs. The instantiation for number type *double* is as follows.

⟨*instantiation for double*⟩≡

```
list<edge> MAX_WEIGHT_BIPARTITE_MATCHING(graph& G,
                    const edge_array<double>& c, node_array<double>& pot)
{ edge_array<double> c1(G);
  scale_weights(G,c,c1,3.0,"MWBM<double>");
  return MAX_WEIGHT_BIPARTITE_MATCHING_T(G,c1,pot);
}
```

where

⟨*scale_weights.h*⟩+≡

```
inline bool scale_weights(const graph& G, const edge_array<double>& c,
                          edge_array<double>& c1, double f)
{ edge e;
  double C = 0;
  forall_edges(e,G) C = leda_max(C,fabs(c[e]));
  double one_over_S;
  double S = compute_S(f,C,one_over_S);
  bool no_scaling = true;
```

```
  forall_edges(e,G)
  { c1[e] = scale_weight(c[e],S,one_over_S);
    if ( c[e] != c1[e] ) no_scaling = false;
  }
  return no_scaling;
}
inline bool scale_weights(const graph& G, const edge_array<double>& c,
                          edge_array<double>& c1, double f, string inf)
{ bool no_scaling = scale_weights(G,c,c1,f);
  if ( no_scaling == false ) cerr << inf << ": scaling was required";
  return no_scaling;
}
```

We also offer a function that replaces a weight vector by its scaled version.

⟨*scale_weights.h*⟩+≡

```
inline bool scale_weights(const graph& G, edge_array<double>& c,
                          double f)
{ edge_array<double> c0 = c;
  return scale_weights(G,c0,c,f);
}
```

There are also analogous functions for node arrays.

How does scaling interact with program checking? We showed in Lemmas 1 and 2 that a computation with doubles computes the exact result for the scaled weights and that the result for the scaled weights is frequently a good approximation of the result for the unscaled weights. We should not expect them to be equal. It is therefore nonsense to check whether a double computation produced the correct result for the unscaled weights if scaling took place.

For example, in the program

```
list<edge> M = MAX_WEIGHT_BIPARTITE_MATCHING(G,c,pot);
CHECK_MWBM(G,c,M,pot);
```

the call of CHECK_MWBM may fail. Indeed, it is very likely to fail if scaling took place in the computation of the maximum weight matching.

We recommend the following strategy of using program checking together with a computation with doubles. *The scaling should be done on the level of the user program.* To this end, each network algorithm comes with a function that replaces all input weights by their scaled versions.

For example, in <*mwb_matching.h*> we also define a function

```
bool MWBM_SCALE_WEIGHTS(const graph& G, edge_array<double>& c)
{
  return scale_weights(G,c,3.0);
}
```

that replaces the cost vector c by a scaled version. One may then write

```
MWBM_SCALE_WEIGHTS(G,c);
list<edge> M = MAX_WEIGHT_BIPARTITE_MATCHING(G,c,pot);
CHECK_MWBM(G,c,M,pot);
```

and checking will work.

The remainder of this section may be skipped. It is worthwhile to study in more detail what it means to replace w by $w' = sign(w) \cdot \lfloor |w| \cdot S \rfloor / S$. Clearly, if $w = 0$ then $w' = 0$. So assume $w \neq 0$. By symmetry, it suffices to study the case $w > 0$.

Lemma 3 *Let* $0 < w = x \cdot 2^e$ *with* $1/2 \leq |x| < 1$, *e integral, and let* $w_1 w_2 \ldots w_{52}$ *be the mantissa of the floating point representation of* w. *Let* s *be an integer, let* $S = 2^s$, *and let* $w' = \lfloor w \cdot S \rfloor / S$. *If* $e + s \leq 0$ *then* $w' = 0$. *If* $e + s > 0$ *then* w' *is obtained from* w *by replacing the mantissa by* $w_1 \ldots w_{e+s-1} 0 \ldots 0$.

Proof We have $w = x \cdot 2^e$ with $1/2 \leq |x| < 1$. If $e + s \leq 0$ then $w' = 0$. So assume $e + s > 0$. We have $2 \cdot x = 1 + \sum_{1 \leq i \leq 52} w_i 2^{-i}$ and hence

$$\begin{aligned}
\lfloor w \cdot S \rfloor &= \lfloor x \cdot 2^{e+s} \rfloor = \lfloor 2 \cdot x \cdot 2^{e+s-1} \rfloor \\
&= \lfloor (1 + \sum_{1 \leq i \leq 52} w_i 2^{-i}) \cdot 2^{e+s-1} \rfloor \leq (1 + \sum_{1 \leq i \leq e+s-1} w_i 2^{-i}) \cdot 2^{e+s-1} \\
&= (1 + \sum_{1 \leq i \leq e+s-1} w_i 2^{-i}) / (2 \cdot 2^e \cdot 2^s)
\end{aligned}$$

and hence

$$w' = \lfloor w \cdot S \rfloor / S = (1 + \sum_{1 \leq i \leq e+s-1} w_i 2^{-i}) / (2 \cdot 2^e),$$

i.e., w' has the same exponent as w and mantissa $w_1 \ldots w_{e+s-1} 0 \ldots 0$. □

Let us consider two special cases.

If all input weights are integers, then the scaling will not change any input as long as $f \cdot C < 2^{53}$. This is as for *int*s, but with *MAXINT* replaced by $2^{53} - 1$.

For the second case we assume that all input weights are less than one. We may assume w.l.o.g. that $1/2 \leq C < 1$. Then $s = 53 - k$ where $k = \lfloor \log f \rfloor + 1$ or $k = \lfloor \log f \rfloor$. If w is any input weight and w has binary representation

$$0.w w_1 w_2 \ldots$$

then w' has binary representation

$$0.w_1 w_2 \ldots w_{53-k} 000 \ldots ,$$

i.e., the binary representation is truncated after the $(53 - k)$-th bit. In this way the scaled weights leave k bits of the mantissa unused. The unused bits can be used to compute intermediate results without rounding error.

7.3 Depth-First Search and Breadth-First Search

Depth-first search and breadth-first search are two powerful methods to explore a graph in a systematic way. Both methods start at some node v of a directed graph G and visit all nodes that can be reached from v. They differ in the order in which they visit the nodes.

Depth-first search always explores edges out of the node most recently reached by the search. When it has exhausted all edges out of a node it backtracks to the node from which the node was reached.

Depth-first search is most easily formulated as a recursive procedure *dfs* that takes a node v as an argument (and additional arguments depending on the application of depth-first search). A call $dfs(v, \ldots)$ first labels v as reached and then makes recursive calls for all nodes w such that (v, w) is an edge out of v and node w is not yet reached. A depth-first search on a graph G induces two numberings of the vertices of G, one in the order in which the nodes are reached by the search and one in the order in which the calls to *dfs* are completed. The two numbers associated with a node are usually called its *depth-first search number* and its *completion number*. Depth-first search can also be used to partition the edges of G into so-called *tree*, *forward*, *backward*, and *cross* edges.

In the program below we use node arrays *dfsnum* and *compnum* to record the two numberings and we use a list T to collect tree edges. The sets of forward, backward, and cross edges are determined implicitly, as we will discuss later. We define two procedures, a recursive procedure $dfs(v, dfsnum, compnum, T)$ and a master $DFS\_NUM(G, dfsnum, compnum)$. A call $dfs(v, \ldots)$ visits and numbers all vertices reachable from v that were not reached previously. We maintain the invariant that $dfsnum[v] = -1$ iff v was not visited yet. The master procedure *DFS_NUM* initializes the variables and then iterates over all nodes. For every node v that was not reached yet it calls $dfs(v, \ldots)$. The call $dfs(v, \ldots)$ sets $dfsnum[v]$ to the current value of *dfsnum_counter*, and then iterates over all edges out of v. Each edge (v, w) to an unreached node w is added to T and leads to a recursive call $dfs(w, \ldots)$. When the edges out of v are exhausted $compnum[v]$ is set to the current value of *compnum_counter*.

$\langle dfs \rangle{+}\!\equiv$

```
static int dfsnum_counter;
static int compnum_counter;
static void dfs(node v, node_array<int>& dfsnum, node_array<int>& compnum,
                list<edge>& T )
{ dfsnum[v] = ++dfsnum_counter;
  edge e;
  forall_adj_edges(e,v)
    { node w = target(e);
      if (dfsnum[w] == -1)
       { T.append(e);
         dfs(w,dfsnum,compnum,T);
       }
    }
  compnum[v] = ++compnum_counter;
}
```

```
list<edge> DFS_NUM(const graph& G, node_array<int>& dfsnum,
                                   node_array<int>& compnum)
{
  list<edge> T;
  dfsnum_counter = compnum_counter = 0;
  dfsnum.init(G,-1);  // declares all nodes unreached
  node v;
  forall_nodes(v,G)
    if (dfsnum[v] == -1) dfs(v,dfsnum,compnum,T);
  return T;
}
```

Figure 7.1 shows the result of a run of *DFS_NUM*. A call *DFS_NUM*$(G, \ldots)$ partitions the edges of G into four classes in a natural way; the four classes are also shown in Figure 7.1. An edge $e = (v, w)$ is called a *tree edge* if *dfs*$(w, \ldots)$ is called when the edge e is scanned in *dfs*$(v, \ldots)$; we use T to denote the set of tree edges. The tree T is the call tree of procedure *dfs*. An edge $e = (v, w)$ is called a *forward edge* if it is parallel to a path of tree edges, but is not a tree edge, i.e., $v \xrightarrow[T]{+} w$ and $e \notin T$; it is called a *backward edge* (or back edge) if it is anti-parallel to a path of tree edges, i.e., $w \xrightarrow[T]{*} v$; and it is called a *cross edge* in all other cases. The two numberings of the vertices can be used to classify the edges[3]. An edge (v, w) is a :

- tree or forward edge iff $dfsnum[v] < dfsnum[w]$ and $compnum[v] > compnum[w]$,
- backward edge iff $dfsnum[v] \geq dfsnum[w]$ and $compnum[v] \leq compnum[w]$,
- cross edge iff $dfsnum[v] > dfsnum[w]$ and $compnum[v] > compnum[w]$.

Let us see why this is true. We only give an intuitive argument and refer the reader to [Meh84, IV.5] and [CLR90, chapter 23] for more detailed discussions.

If two calls C and D of *dfs* are nested within one another, say D is nested within C, then C starts before D and ends after D, i.e., the dfs-number of the node corresponding to C is smaller than the dfs-number of the node corresponding to D and the completion-number of the node corresponding to C is larger than the completion number of the node corresponding to D. This explains the characterization of tree, forward, and backward edges.

If two calls C and D are not nested within one another and, say C starts after D, then C starts after the completion of D and hence the dfs-number of the node corresponding to C is larger than the dfs-number of the node corresponding to D and the same holds for completion-numbers. This fact together with the observation that a cross edge always runs from a node reached later to a node reached earlier explains the characterization of cross edges.

Depth-first search considers every edge of the graph G exactly once and hence runs in linear time $O(n + m)$, where $n = |V|$ and $m = |E|$.

[3] There is no standard convention concerning self-loops. We classify self-loops as back edges.

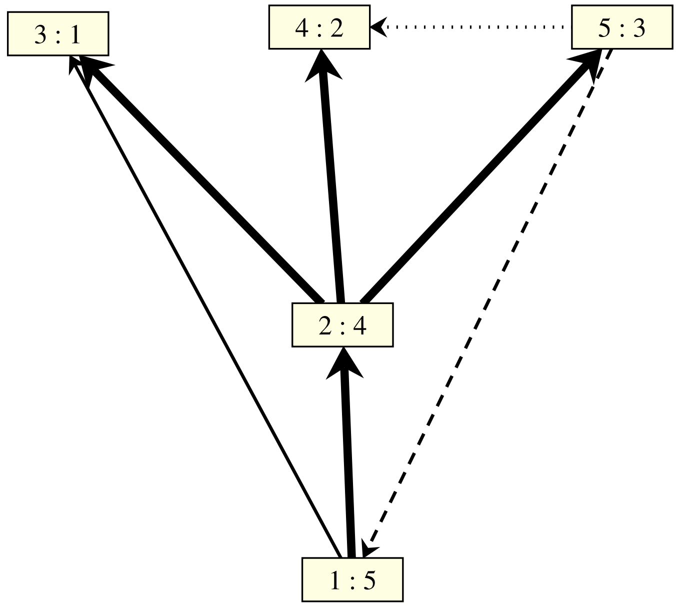

Figure 7.1 Depth-first search: The search started at the bottom-most node. For each node the dfs- and the completion-number are shown inside the node. Tree edges are shown as thick solid edges, forward edges are shown as thin solid edges, backward edges are shown as dashed edges, and cross edges are shown a dotted edges. It is customary to draw dfs-trees such that tree edges are directed upwards and cross edges are directed from right to left. Observe how dfs-numbers increase along every tree path and how completion-numbers decrease. Also observe that cross edges go from nodes with higher dfs- and completion-number to nodes with lower dfs- and completion-number. You may generate your own figures by calling the xlman-demo gw_dfs.

Why should one be interested in the classification of the edges into tree, forward, backward, and cross edges? Here is one reason. A depth-first search on an acyclic graph does not find any backward edges. Thus $compnum[v] > compnum[w]$ for any edge (v, w), i.e., all edges go from higher to lower completion numbers. In other words, *compnum* is a topological numbering of the graph.

We turn to breadth-first search. It explores the edges in the order in which their source vertex is reached. It uses a queue Q to store the vertices in the order in which they are reached and always explores edges out of the first node of the queue. When all edges out of the first node are scanned, the first node is popped from the queue and exploration from the new first node is started. BFS can be used to label the vertices with their distance from a particular node s, i.e., to compute a *node_array*<*int*> *dist* such that $dist[w] = d$ iff there is a path from s to w of length d and d is the smallest integer with this property.

$\langle bfs \rangle \equiv$
```
void BFS(const graph& G, node s, node_array<int>& dist)
{ queue<node> Q;
  node v,w;
```

```
    forall_nodes(w,G) dist[w] = -1;
    dist[s] = 0;
    Q.append(s);
    while (!Q.empty())
    { v = Q.pop();
      forall_adj_nodes(w,v)
        if (dist[w] < 0)
        { Q.append(w);
          dist[w] = dist[v] + 1;
        }
    }
  }
```

The correctness of BFS is easy to establish. Clearly, if $dist[w] = d$ then there is a path of length d from s to w. On the other hand, if $s = v_0, v_1, \ldots, v_l = w$ is a path from s to w of length l then $dist[v_i] \leq i$ for all i, $1 \leq i \leq l$.

Exercises for 7.3

1 Why can there be no edge (v, w) in a depth-first search with $dfsnum[v] < dfsnum[w]$ and $compnum[v] < compnum[w]$?

2 Write a procedure based on depth-first search that tests a graph for acyclicity. If the graph is acyclic it should also compute a so-called topological numbering of the vertices of G, i.e., a labeling of the nodes of G such that for all edges of G the label of the source node is smaller than the label of the target node.

3 Use the program LEDAROOT/demo/xlman/gw_dfs.c as the basis of a program that illustrates BFS.

7.4 Reachability and Components

We start with an overview of the algorithms that compute reachability information and simple structural information of directed and undirected graphs: transitive closure, connected and biconnected components, and strongly connected components. Then we discuss the details of the strongly connected components algorithm, and finally we describe an animation of this algorithm.

7.4.1 *Functionality*

We deal with basic problems concerning reachability in directed and undirected graphs. We first consider directed graphs and later turn to undirected graphs.

Let $G = (V, E)$ be a directed graph and let v and w be two vertices of G. Recall that w is *reachable* from v if there is a path in G from v to w, i.e., if either $v = w$ or there is a sequence $e_1, \ldots, e_k$ of edges of G with $k \geq 1$, $v = source(e_1)$, $w = target(e_k)$, and $target(e_i) = source(e_{i+1})$ for all i, $1 \leq i < k$.

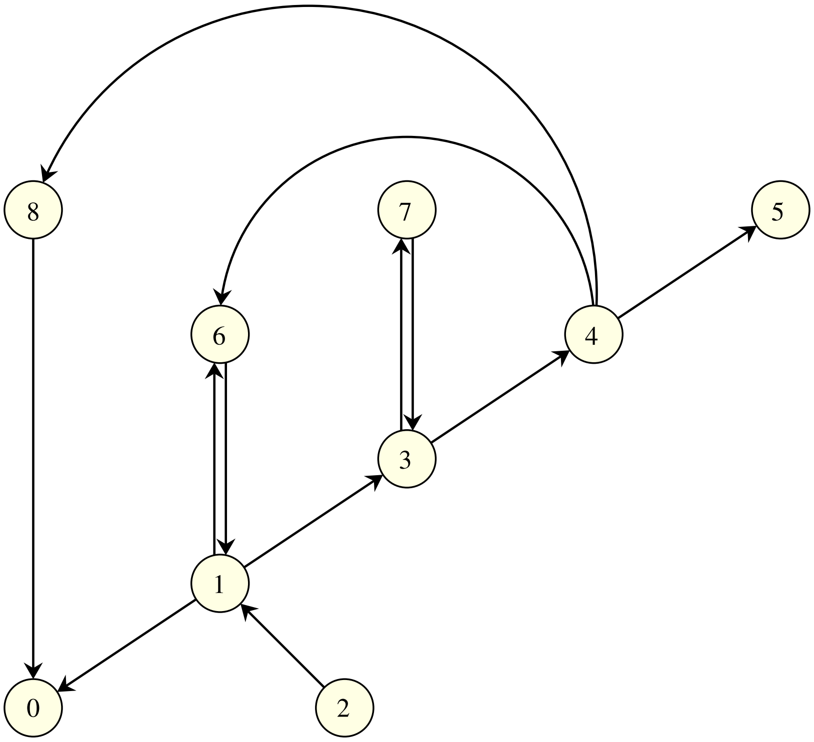

Figure 7.2 A graph with five strongly connected components. The five components are induced by the node sets $C_0 = \{8\}$, $C_1 = \{5\}$, $C_2 = \{1, 3, 4, 6, 7\}$, $C_3 = \{0\}$, and $C_4 = \{1\}$. The xlman-demo gw_scc_anim illustrates strongly connected components.

The graph $G^* = (V, E^*)$ where $E^* = \{(v, w);\, w \text{ is reachable from } v\}$ is called the *reflexive transitive closure* of G. The procedure

```
graph TRANSITIVE_CLOSURE(const graph& G);
```

computes G^* from G in time $O(n^2 + m_{red} \cdot n)$ where $n = |V|$ and m_{red} is the number of edges in a transitive reduction of G. A *transitive reduction* of G is a minimal (with respect to set inclusion of edges) subgraph of G with the same transitive closure as G. In an acyclic graph, m_{red} is the number of edges (v, w) of G such that there is no path of length two or more from v to w in G. For random graphs in the $G_{n,p}$-model and arbitrary value of p, $E(m_{red}) = O(n)$ and hence the expected running time of the transitive closure algorithm is $O(n^2)$, see [Meh84, IV.3].

A directed graph G is called *strongly connected* if from any node of G there is a path to any other node of G. A *strongly connected component* (scc) of a graph G is a maximal strongly connected subgraph. Figure 7.2 shows a graph with five strongly connected components. Shrinking the strongly connected components of a graph to single nodes gives rise to an acyclic graph $G_s = (V_s, E_s)$ with

$$V_s = \{C;\, C \text{ is an scc of } G\}$$

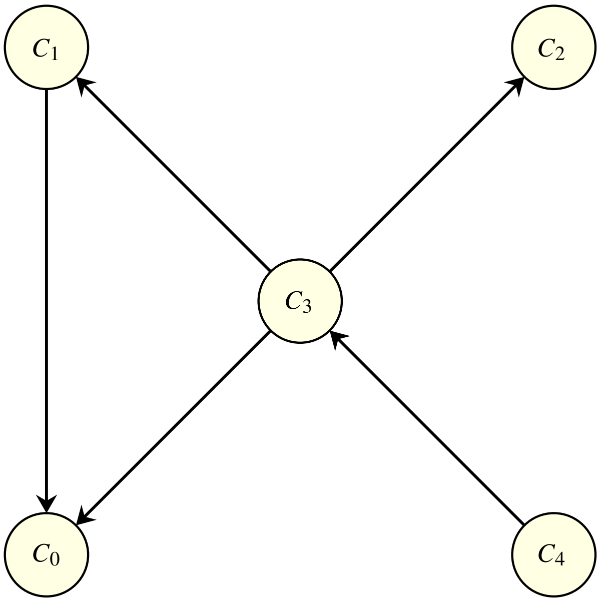

Figure 7.3 The graph obtained by shrinking the sccs of the graph in Figure 7.2 to single nodes. The given numbering of the sccs will be obtained if a first depth-first search is started in node 0 (it will only reach 0) and a second depth-first search is started in node 2.

and

$$E_s = \{(C, D);\ C, D \in V_s \text{ and there exists } (v, w) \in E \text{ with } v \in C \text{ and } w \in D\}$$

Figure 7.3 shows the shrunken graph obtained from the graph of Figure 7.2.

The procedure

```
int STRONG_COMPONENTS(const graph& G, node_array<int>& comp_num)
```

returns the number of strongly connected components of G and computes a *node_array*<*int*> *comp_num* with the following properties[4]:

- For all nodes v of G: $0 \le comp\_num[v] <$ number of sccs of G.
- $comp\_num[v] = comp\_num[w]$ iff the vertices v and w belong to the same strongly connected component.
- If (v, w) is an edge of G then $comp\_num[v] \ge comp\_num[w]$.

In other words, the array *comp_num* encodes the strongly connected components of G and moreover induces a topological ordering of the shrunken graph. The scc demo illustrates the strongly connected components algorithm. The demo allows one to construct a graph interactively. After every edit step the strongly connected components are recomputed and highlighted by a color and numbering code. Procedure *STRONG_COMPONENTS* runs in

[4] Observe that *comp_num* stands for component number and *compnum* stands for completion number.

linear time $O(n+m)$, where $n = |V|$ and $m = |E|$; its implementation is given in the next section.

The transitive closure algorithm uses the strongly connected components algorithm as a subroutine: it first computes the sccs, then the shrunken graph, then the transitive closure of the shrunken graph, and finally the transitive closure of the full graph. We give the simple procedure for computing the shrunken graph *SG* corresponding to a graph G. We first call the strong components algorithm for G and give *SG* one vertex for each scc of G. We then iterate over the edges of G and add an edge to *SG* for each edge (v, w) of G where v and w belong to distinct sccs. Finally, we remove parallel edges by calling *Make_Simple*(*SG*).

⟨*shrunken_graph*⟩≡
```
graph SHRUNKEN_GRAPH(const graph& G)
{ node_array<int> comp_num(G);
  int N = STRONG_COMPONENTS(G, comp_num);
  graph SG;
  array<node> V(N);
  for (int i = 0; i < N; i++) V[i] = SG.new_node();
  edge e;
  forall_edges(e,G)
  { node v = G.source(e); node w = G.target(e);
    if (comp_num[v] > comp_num[w] )
      SG.new_edge(V[comp_num[v]],V[comp_num[w]]);
  }
  Make_Simple(SG);
  return SG;
}
```

We turn to undirected graphs. The data type *ugraph* represents undirected graphs. Alternatively, directed graphs may be interpreted as undirected graphs, see Section 6.7. In the early versions of LEDA we used *ugraphs* as the argument of all graph algorithms that operate on undirected graphs. We now prefer to use *graphs* and to let the algorithms interpret them as undirected graphs. In the discussion of the algorithms we talk about undirected graphs, of course.

Let $G = (V, E)$ be an undirected graph. It is called *connected* if for any two vertices v and w there is a path from v to w in G, i.e., either $v = w$ or there is a sequence $v_1, \ldots, v_k$ of vertices such that $v = v_1$, $w = v_k$, and $\{v_i, v_{i+1}\}$ is an edge of G for all i, $1 \le i < k$. A component of G is a maximal connected subgraph of G. The procedure

```
int COMPONENTS(const graph& G, node_array<int>& comp_num)
```

computes the number of connected components, say N, of G and an array *comp_num* such that $0 \le comp\_num[v] < N$ for all vertices v and $comp\_num[v] = comp\_num[w]$ iff the vertices v and w belong to the same connected component of G. It runs in linear time $O(n+m)$.

A connected undirected graph $G = (V, E)$ is called *biconnected* if $G - v$ is connected

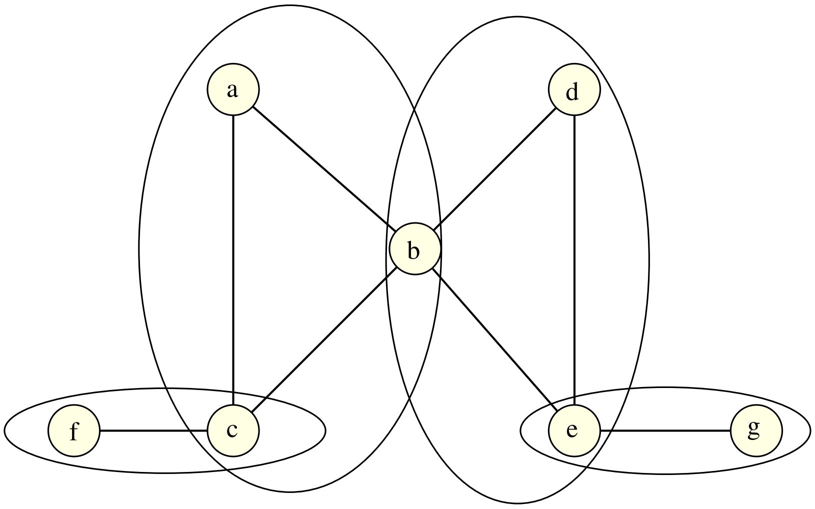

Figure 7.4 A graph with four bccs. The bccs are indicated by ovals. They have edge sets $\{\{f, c\}\}$, $\{\{e, g\}\}$,$\{\{a, b\}, \{b, c\}, \{a, c\}\}$, and $\{\{b, d\}, \{b, e\}, \{e, d\}\}$, respectively. The articulation points are the nodes b, c, and e.

for every $v \in V$. Here

$$G - v = (V - v, \{e; e \in E \text{ and } v \notin e\})$$

is the graph obtained by removing the vertex v and all edges incident to v from G. For graphs with at least three nodes the following alternative definition is useful: G is biconnected if for any distinct vertices v and w there are two vertex-disjoint paths connecting v and w. A *biconnected component* (bcc) is a maximal biconnected subgraph. A vertex a is called an *articulation point* of G if $G - a$ is not connected. Figure 7.4 shows a graph with four biconnected components.

Let G be an undirected graph and let $G_1 = (V_1, E_1), \ldots, G_m = (V_m, E_m)$ be the biconnected components of G. We claim that $E = E_1 \cup \ldots \cup E_m$ and $|V_i \cap V_j| \leq 1$ and $E_i \cap E_j = \emptyset$ for $i \neq j$. To see this, note first that for each edge $\{v, w\} \in E$ the graph consisting of vertices v and w and the single edge $\{v, w\}$ is biconnected, and hence contained in one of the biconnected components of G. It remains to show that any two distinct bccs share at most one vertex (this also implies that they can share no edge). Assume otherwise, i.e., we have distinct bccs G_i and G_j and a pair $\{v, w\}$ of nodes belonging to both. Since G_i and G_j are maximal biconnected subgraphs, the subgraph $G' = (V_i \cup V_j, E_i \cup E_j)$ is not biconnected and hence has an articulation point, say a. Let x and y be vertices in different components of $G' - a$. Since a is neither an articulation point in G_i nor in G_j, the graphs $G_i - a$ and $G_j - a$ are connected and hence x and y cannot both be vertices in the same graph G_i or G_j. We may assume w.l.o.g. that $x \in V_i$ and $y \in V_j$. Since a cannot be equal to both v and w we may assume $v \neq a$. Since $G_i - a$ and $G_j - a$ are connected, a path exists from x to v in $G_i - a$ and from y to v in $G_j - a$. Hence a path exists from x to y in

$G' - a$ and we have reached a contradiction. We conclude that the bccs of a graph partition the edges.

The procedure

```
int BICONNECTED_COMPONENTS(const graph& G, edge_array<int>& comp_num)
```

returns the number of bccs of the undirected version of G and computes an edge array *comp_num* such that $comp\_num[e] = comp\_num[f]$ iff the edges e and f belong to the same biconnected component of G. The running time is $O(n + m)$.

We give more details. Let c be the number of biconnected components and let c' be the number of biconnected components containing at least one edge; $c - c'$ is the number of isolated nodes in G, i.e., the number of nodes v that are not connected to a node different from v. The function returns c and labels each edge of G (which is not a self-loop) by an integer in $[0 \,..\, c' - 1]$. Two edges receive the same label iff they belong to the same biconnected component. The edge labels are returned in *comp_num*. Be aware that self-loops receive no label since self-loops are ignored when interpreting a graph as an undirected graph.

The nodes of a biconnected graph can be numbered in a special way which is useful for many algorithms on biconnected graphs. Imagine the following physical experiment. G is a biconnected graph and s and t are any two nodes of G that are connected by an edge. We replace all edges of G by rubber bands and then pull s and t apart. Since G has no articulation point, this will exert force on every node of G and order the nodes of G along the line from s to t. We number the nodes from 1 to n starting with s and proceeding towards t. Every node v of G, except for s and t, will have a smaller numbered and a higher numbered neighbor. Such a numbering is called an *st-numbering* of G. The function

```
void ST_NUMBERING(graph& G, node_array<int>& stnum, list<node>& stlist)
```

numbers the nodes of G with the integers 1 to n (the number of any node v is returned in $stnum[v]$ and the ordered list of nodes is returned in *stlist*) such that every node v with $1 < stnum[v] < n$ is connected to a node with smaller number and to a node with higher number, and such that the nodes with numbers 1 and n are connected by an edge. The running time is $O(n + m)$. We will see an application of st-numbering in Section 8.7.

7.4.2 *Strongly Connected Components: An Implementation*

We give a program to compute the strongly connected components of a directed graph. An animation of this program is available as the xlman-demo gw_scc_anim. The algorithm is an extension of depth-first search and was first described in [CM96]; alternative algorithms are described in [Tar72] and [Sha81].

Consider a depth-first search on G and use $G_c = (V_c, E_c)$ to denote the subgraph already explored, i.e., V_c is the set of nodes v for which $dfs(v, \ldots)$ has been called and E_c consists of all edges e which have been explored in one of the calls of *dfs*. The algorithm maintains the strongly connected components of G_c. In order to derive the algorithm we first introduce some notation and then state some properties of G_c.

We call a vertex $v \in V$ *completed* if the call $dfs(v, \ldots)$ has been completed, *unreached* if

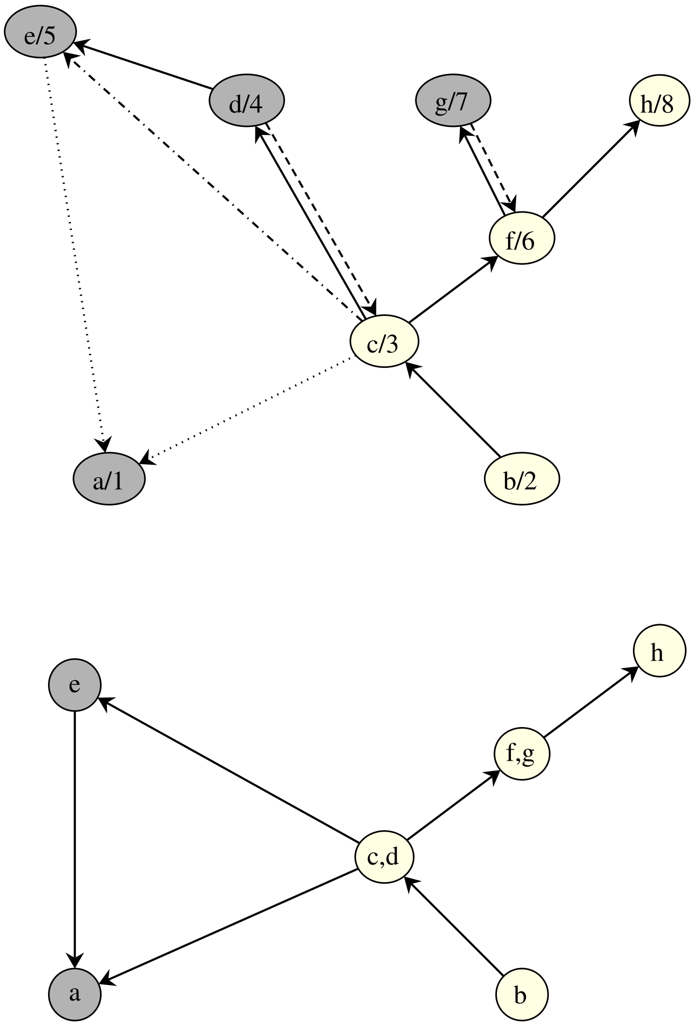

Figure 7.5 A snapshot of depth-first search on the graph of Figure 7.2 and the shrunken graph corresponding to it.
A first dfs was started at node a and a second dfs was started at node b. The upper part shows the snapshot of dfs; it is assumed that the search has just reached node h and is starting to explore the edges out of h. The edge (h, i) and the node i have not been seen yet and the depth-first search numbers of the nodes are indicated. The node h is the current node. Completed nodes are shown shaded.
The shrunken graph is shown in the lower part of the figure. The components $\{a\}$ and $\{e\}$ are permanent and all other components are tentative. The permanent components are shown shaded. The tentative components form a path P in the shrunken graph and h belongs to the last component of P. The roots of the tentative components are the vertices b, c, f, and h. They lie on a common tree path of the depth-first search tree of G.

the call $\mathit{dfs}(v, \ldots)$ has not been started yet, and *active* otherwise, i.e., if the call has already been started but not yet completed. All active nodes lie on a single path in G and this path corresponds to the recursion stack of depth-first search. We call the last node of this path the *current node*. We call an scc of G_c *permanent* if all its vertices are completed and we call it *tentative* if this is not the case. The *root* of an scc is the node in the scc with the smallest depth-first search number. Figure 7.5 illustrates these concepts. In this example the shrunken graph of G_c exhibits considerable structure:

(1) There is no edge $(v, w) \in E$ with v belonging to a permanent scc and w not belonging to a permanent scc. In particular, all vertices reachable from a vertex in a permanent scc are completed.

(2) The tentative sccs form a path P in the shrunken graph and the current node is contained in the last scc of this path.

(3) If C and C' are distinct tentative sccs with C preceding C' on P then all vertices in C have smaller dfs-number than all vertices in C'.

(4) Let C be a tentative scc of G_c and let r be its root. Then all vertices in C and all nodes in all successors of C on P are tree descendants of r in the depth-first search tree, i.e., the name root is justified.

We will show below that all four properties hold true generally and not only for our running example. The four properties will be invariants of the algorithm to be developed. The first invariant implies that the permanent sccs of G_c are actually sccs of G, i.e., it is justified to call them permanent. This observation is so important that it deserves to be stated as a lemma.

Lemma 4 *A permanent scc of G_c is an scc of G.*

Proof Let v be a vertex in a permanent scc of G_c and let w be a node of G such that v and w belong to the same scc of G. Thus there is a cycle C in G passing through v and w. If v and w do not belong to the same scc of G_c, one of the edges of C does not belong to G_c. The source node of this edge cannot be completed and hence does not lie in a permanent component. Since v lies in a permanent component, there must be an edge (x, y) on C such that x lies in a permanent component, but y does not. This is a contradiction to our first invariant. □

Invariants (2) to (4) suggest a simple method to represent the tentative sccs of G_c. We simply keep a sequence *unfinished* of all vertices in tentative sccs in increasing order of dfs-number and a sequence *roots* of all roots of tentative sccs. In our example *unfinished* is b, c, d, f, g, h, and *roots* is b, c, f, h. For both sequences the data type *stack<node>* is appropriate.

We can now start to write code. As already mentioned the program is an extension of depth-first search and has the same global structure. As in Section 7.3 we define two procedures: *STRONG_COMPONENTS* is the main procedure and *SCC_DFS* is an auxiliary procedure. Both procedures make use of the stacks *unfinished* and *roots* and the node arrays *dfsnum* and *comp_num*: *dfsnum*$[v]$ is the dfs-number of v for all reached nodes and is -1 for all unreached nodes; *comp_num*$[v]$ is the number of the scc containing v for all nodes belonging to permanent sccs and is -1 for all other nodes. The variables *dfscount* and *comp_count* keep track of the used dfs-numbers and component numbers, respectively.

STRONG_COMPONENTS defines and initializes all variables and then iterates over all nodes of G. It calls *SCC_DFS*$(v, \ldots)$ for each unreached node v. A call *SCC_DFS*$(v, \ldots)$ assigns the next dfs-number to v and makes v a tentative scc of its own. It then explores all edges out of v. Finally, it returns from the call.

⟨*SCC*⟩≡
```
void SCC_DFS(node v, const graph& G, node_array<int>& dfsnum,
             node_array<int>& comp_num, stack<node>& unfinished,
             stack<node>& roots, int& dfscount, int& comp_count)
{ dfsnum[v] = dfscount++;
  ⟨make v a tentative scc of its own⟩
  node w;
  forall_adj_nodes(w,v){ ⟨explore edge (v,w)⟩ }
  ⟨return from the call for node v⟩
}
int STRONG_COMPONENTS(const graph& G, node_array<int>& comp_num)
{ stack<node> unfinished;
  stack<node> roots;
  node_array<int> dfsnum(G, - 1);
  node v;
  forall_nodes(v,G) comp_num[v] = - 1;
  int dfscount = 0;
  int comp_count = 0;
  forall_nodes(v,G)
    if (dfsnum[v] == -1)
      SCC_DFS(v,G,dfsnum,comp_num,unfinished,roots,dfscount,comp_count);
  return comp_count;
}
```

A call *SCC_DFS*$(v, \ldots)$ makes v a tentative scc of its own since G_c contains no edges out of v yet. This amounts to adding v to the top of *unfinished* and *roots*. Thus

⟨*make v a tentative scc of its own*⟩≡
```
unfinished.push(v);
roots.push(v);
```

It is easy to check that all invariants are maintained.

We come to the exploration of an edge $e = (v, w)$. If e is a tree edge (this is the case iff *dfsnum*$[w] = -1$) we simply initiate a recursive call. If e is a non-tree edge and w belongs to a permanent scc (this will be the case if *dfsnum*$[w] \geq 0$ and *comp_num*$[w] \geq 0$), then, by Lemma 4, no action is required to maintain the invariants. If e is a non-tree edge and w belongs to a tentative scc (this will be the case if *dfsnum*$[w] \geq 0$ and *comp_num*$[w] = -1$)

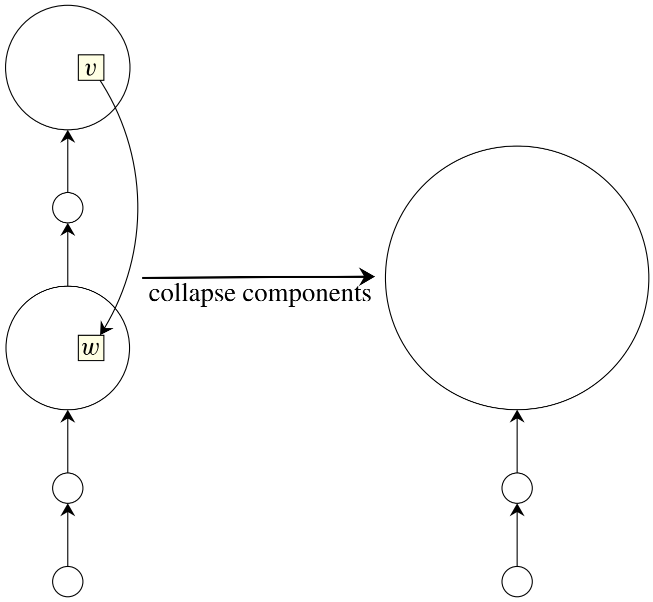

Figure 7.6 The path of tentative sccs and the effect of exploring an edge (v, w), where w belongs to a tentative scc. All tentative sccs on the path from the tentative scc containing w to the tentative scc containing v are collapsed into a single scc.

then some final segment of the path of tentative sccs collapses to a single scc (cf. Figure 7.6). Thus

⟨*explore edge (v,w)*⟩≡
```
if (dfsnum[w] == - 1)
   SCC_DFS(w,G,dfsnum,comp_num,unfinished,roots,dfscount,comp_count);
else if (comp_num[w] == - 1) { ⟨merge sccs⟩ }
```

We give the details of merging sccs. Assume that w belongs to a tentative scc with root r. Then r is the topmost root in *roots* with $dfsnum[r] \le dfsnum[w]$ (by invariant (3)). Any root r' above r ceases to be a root since $v \longrightarrow w \overset{*}{\longrightarrow} r \overset{*}{\longrightarrow} r' \overset{*}{\longrightarrow} v$. Note that $w \overset{*}{\longrightarrow} r$ since w and r belong to the same scc, and $r \overset{*}{\longrightarrow} r' \overset{*}{\longrightarrow} v$ since the shrunken graph of tentative sccs is a path. Thus

⟨*merge sccs*⟩≡
```
while (dfsnum[roots.top()] > dfsnum[w]) roots.pop();
```

What do we have to do when we return from a call, say for node v? The completion of v completes an scc if v is a root (by invariant (4)) and v is a root iff $v = roots.top(\,)$ (since the call for the topmost root is completed before the call of any other root contained in *roots*, again by invariant (4)). If v is a root the scc of v consists of all nodes in *unfinished* whose *dfsnum* is at least as large as v's *dfsnum* (by invariant (3)). We simply pop these nodes from *unfinished* and define their *comp_num*. Lemma 4 tells us that this scc is also an scc of the final graph.

⟨*return from the call for node v*⟩≡

```
if (v == roots.top())
{ do
  { w = unfinished.pop();
    comp_num[w] = comp_count;
  } while ( w != v);
  comp_count++;
  roots.pop();
}
```

Invariants (2), (3), and (4) are clearly maintained. For invariant (1) this can be seen as follows. Let C be the scc with root v. Then C is the last scc of the path P of tentative sccs and hence all other tentative sccs are predecessors of C on P. Thus there can be no edge in E_c from a vertex in C to a vertex in any other tentative scc. Since all nodes in C are completed, all edges $(x, y) \in E$ with $x \in C$ are also edges in E_c and invariant (1) holds.

7.4.3 *Strongly Connected Components: An Animation*

We describe an animation of the algorithm of the preceding section. The animation is available as the xlman-demo gw_scc_anim . The animation consists of two parts. In the first part the user can interactively construct a directed graph G; after every edit operation of the user the strongly connected components of G are recomputed and shown in number and color code, i.e., nodes belonging to the same scc are shown in the same color and with the same integer label. In the second part the execution of our scc-algorithm on the graph constructed in the first section is animated. Figure 7.7 shows a screen-shot. The overall structure of the program is as follows:

⟨*gw_scc_anim.c*⟩≡

```
#include <LEDA/graph_alg.h>
#include <LEDA/graphwin.h>
⟨display functions for part one⟩
⟨display functions for part two⟩
⟨help panels⟩
int main(){
GraphWin gw("SCC Animation Demo");
gw.display();          // open display
gw.set_directed(true);
int h_menu = gw.get_menu("Help");
gw_add_simple_call(gw,about_scc_anim1, "About SCC: phase 1",h_menu);
gw_add_simple_call(gw,about_scc_anim2, "About SCC: phase 2",h_menu);
gw_add_simple_call(gw,about_scc_anim_basics, "About SCC: basics",h_menu);
gw_add_simple_call(gw,about_scc_anim_data_structures,
                                    "About SCC: data structures",h_menu);
⟨part one of demo⟩
```

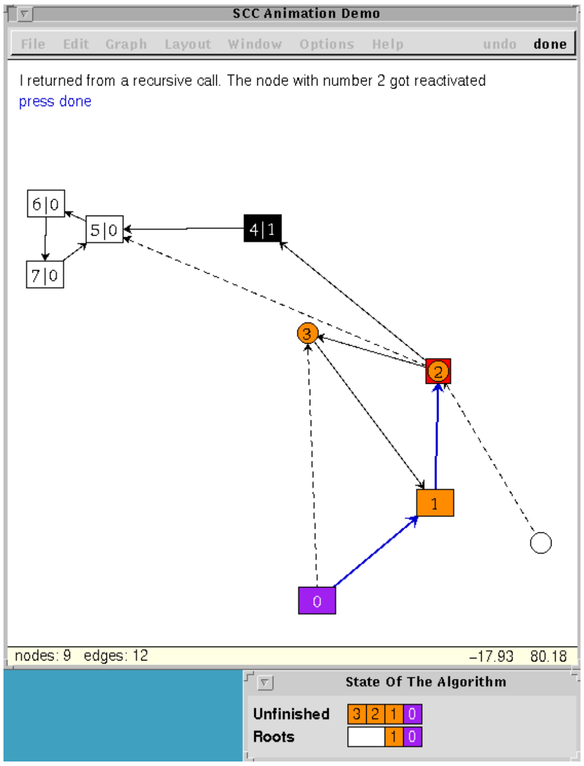

Figure 7.7 A screen-shot of the second part of gw_scc_anim. Explored nodes are labeled with their depth-first search number and nodes in permanent sccs are labeled with their depth-first search number and the number of the scc containing them. Explored edges are drawn solid and unexplored edges are drawn dashed. The nodes in permanent sccs are shown in the left half of the window and the other nodes are shown in the right half of the screen. The node with depth-first number 2 is the currently active node and there are two tentative components, one consisting of node 0 and the other one consisting of nodes 1, 2, and 3. There is one unreached node. The stacks *unfinished* and *roots* are indicated at the bottom of the screen-shot. The text at the top of the window explains the actions of the algorithm.

```
⟨part two of demo⟩
return 0;
}
```

The animation is based on the data type *GraphWin*; this data type is a combination of graphs and windows and is discussed in Chapter **??**. Most of the current section can be appreciated without knowledge of GraphWins, as we explain the used features of GraphWin as we go along. However, the explanations of GraphWin will be kept short and hence readers without knowledge of GraphWin will miss some of the fine points. We hope that all readers will enjoy the demo so much that they will also study GraphWin.

In *main* we first define a *GraphWin* *gw* and then inform *gw* that we are dealing with directed graphs. We then set up the help menu. *GraphWin* has already a predefined help menu. We get its number and add three buttons to it. The corresponding functions *about_scc_anim1*, ... are defined in the program chunk ⟨*help panels*⟩. We do not show it as the help buttons of gw_scc_anim should give sufficient information. Having set up the help buttons we start part one of the demo. It makes use of the display functions defined in the corresponding chunk. The same holds true for the second part of the demo.

We come to the first part of the demo. Thanks to the powerful *GraphWin* data type it is extremely simple to write. A *GraphWin* always has an associated graph and moreover it maintains information about how to display the constituents of this graph: for example, for a node it maintains the position of the node, the color of the node, and the shape of the node (circle, square, rectangle, or ellipse), and for an edge it maintains the style of the edge (solid or dashed or dotted) and the color and the width of the edge. The display information can be modified.

In *display_scc* we first get the current graph G from *gw* and then compute the strongly connected components of G. We then set for each node v of G the color of v to the component number of v modulo 16 (as we rely only on the availability of 16 different colors) and we set the so-called user label[5] of v to the component number of v. We also inform *gw* that we want the user label to be displayed with each node.

We want *display_scc* to be called whenever the graph associated with *gw* is modified. This is easy to achieve. It is possible to associate functions with a *GraphWin* (so-called handlers) that are called whenever a node or edge is added or deleted. For example, *gw.set_del_edge_handler*(*display_scc*) informs *gw* that the function *display_scc* is to be called whenever an edge is deleted. The handlers for the addition of a node or edge are syntactically required to have a second argument which is a node or edge, respectively. We therefore need to wrap *display_scc* accordingly before defining the new edge and the new node handler.

After having set the handlers we open the display, show the help information for phase one, and put *gw* into edit mode. The call *gw.edit*( ) is terminated by a click on the done-button of *gw*.

⟨*display functions for part one*⟩≡

```
void display_scc(GraphWin& gw)
{ graph& G = gw.get_graph();
  node_array<int> comp_num(G);
  int N = STRONG_COMPONENTS(G,comp_num);
  node v;
  forall_nodes(v,G)
  { gw.set_color(v,comp_num[v]%16);
    gw.set_user_label(v,string("%d",comp_num[v]));
    gw.set_label_type(v,user_label);
  }
}
void new_edge_handler(GraphWin& gw, edge)  { display_scc(gw); }
```

[5] In *GraphWin* each node has a number of predefined labels; one of them is called the user label.

```
void new_node_handler(GraphWin& gw, node)  { display_scc(gw); }
```

⟨*part one of demo*⟩≡

```
gw.set_init_graph_handler(display_scc);
gw.set_del_edge_handler(display_scc);
gw.set_del_node_handler(display_scc);
gw.set_new_node_handler(new_node_handler);
gw.set_new_edge_handler(new_edge_handler);
about_scc_anim1(gw);  // inform user about phase 1
gw.message("\\blue Construct or load a graph and press done.");
wait(1.75);
gw.message("");
gw.edit();            // enter edit mode
```

We come to part two of the demo. The goal of part two is to animate the strongly connected components algorithm of the preceding section. The idea behind the animation is as follows. We use a split design for the main window. The right half of the window shows all tentative components of G_c and all unexplored nodes and the left half of the screen shows all permanent components. Also, unreached nodes are shown as white empty circles and unexplored edges are shown dashed. The code below sets up the initial configuration of this design and also displays some textual information for the user (which we do not show here to save space).

We first get the coordinates of the window boundaries and then move the contents of *gw* to the right half of the screen. We then create the initial drawing of the demo. For each node v we set the color to white, state that the node is to be drawn as a circle of radius *small_width* (*small_width* is defined in program chunk ⟨*display functions for part two*⟩), state that the displayed information is the user label, set the user label to the empty string, and compute the position to which v is moved once it belongs to a permanent component. We also set the style of all edges to dashed. We then call the *STRONG_COMPONENTS* function of the preceding section; of course, this function needs to be augmented by display actions and therefore needs additional arguments, namely, *gw* and *perm_pos*.

⟨*part two of demo*⟩≡

```
gw.disable_calls();     // disable buttons
about_scc_anim2(gw);
graph&  G = gw.get_graph();
window& W = gw.get_window();
node_array<point> perm_pos(G);
double xmin = gw.get_xmin() + W.pix_to_real(20);
           // coordinate of left boundary plus 20 pixels
double xmax = gw.get_xmax() - W.pix_to_real(20);
double ymin = gw.get_ymin() + W.pix_to_real(30);
double ymax = gw.get_ymax() - W.pix_to_real(20);
double dx = xmax - xmin;
double dy = ymax - ymin;
```

```
gw.place_into_box(xmin+dx/2,ymin,xmax,ymax-dy/5);
          // move everything to right half of screen
gw.set_flush(false);  // changes are accumulated
node v;
forall_nodes(v,G)
{ gw.set_color(v,white);
  gw.set_label(v,user_label); gw.set_user_label(v,"");
  gw.set_shape(v,circle_node);
  gw.set_node_width(small_width);
  double xcoord = gw.get_position(v).xcoord();
  double ycoord = gw.get_position(v).ycoord();
  perm_pos[v] = point(xcoord-dx/2,ycoord);
}
edge e;
forall_edges(e,G) gw.set_style(e,dashed_edge);
gw.redraw();        // all changes are performed now
gw.set_flush(true);
```
⟨*more information about part two*⟩
```
node_array<int> comp_num(G);
STRONG_COMPONENTS(G,comp_num,gw,perm_pos);
gw.message("\\bf Wasn\'t this a nice demo ?");
wait(1);
gw.message("");
gw.fill_window();
gw.enable_calls();         // enable buttons
gw.edit();
```

We come to the display functions used for part two. We display nodes in two sizes: roots and nodes in permanent components are shown as large rectangles and all other nodes are shown as small circles. All nodes in the same strongly connected components are colored with the same color. For permanent components we use the color corresponding to the component number and for tentative components we use the height of the root of the component in the *roots*-stack. In order to keep the colors for permanent and tentative components separate (or at least approximately so) we add an integer *color_shift* to all colors of tentative components.

The demo can be run in either of two modes. In step mode the next action is triggered by a click on the done-button and in continuous mode the animation is run to completion without user interaction. The choice of mode is controlled by the variable *step* and the procedure *message* which we use to write messages *msg* into *gw*. If *step* is true, *msg* is displayed until the done-button is pressed. If *step* is true and the exit button is pressed, *step* is set to false and the demo runs to completion (since *message* has no effect when *step* is false).

We define a window *state_win* (in addition to the window associated with *gw*) and use it to display state information. The state information is generated by the function *state_info*.

It draws the stacks[6] *unfinished* and *roots* as sequences of rectangles into *state_win*. Each rectangle is labeled with the dfs-number of the node it represents. The stacks *unfinished* and *roots* are displayed in a way that equal elements are aligned (recall that *roots* is a subsequence of *unfinished*).

⟨*display functions for part two*⟩≡

```
static int  small_width = 20;
static int  large_width = 36;
static int  color_shift = 5;

static bool step = true;

void message(GraphWin& gw, string msg)
{ msg += "\\5 \\blue press done \\black";
  if (step && !gw.wait(msg)) step = false;
}

static window state_win(320,60,"State Of The Algorithm");

static void state_redraw(window* wp) { wp->flush_buffer(); }

static color text_color(color col)
{ if (col==black || col==red || col==blue || col==violet ||
     col==brown || col==pink || col==blue2 || col==grey3)
     return white;
  else
     return black;
}

void state_info(GraphWin& gw, const list<node>& unfinished,
                              const list<node>& roots,
                              const node_array<int>& dfsnum,
                              node cur_v)
{
  if (!state_win.is_open())
  { state_win.set_bg_color(grey1);
    state_win.set_redraw(state_redraw);
    state_win.display(-gw.get_window().xpos()+8,0);
    state_win.init(0,320,0);
    state_win.start_buffering();
   }

  state_win.clear();

  double th = state_win.text_height("H");

  double x0 = state_win.text_width("Unfinished") + 2*th;
  double y1 = state_win.ymax() - 1.75*th;
  double y2 = state_win.ymax() - 3.20*th;

  double d = 18;

  state_win.draw_text(5,y1+(d+th)/2,"Unfinished");
  state_win.draw_text(5,y2+(d+th)/2,"Roots");

  list_item r_it = roots.first();

  double x = x0;
```

[6] In contrast to the preceding section we realize both stacks as lists, the reason being that we need to iterate over all elements in both stacks and that stacks do not support iteration over their elements (they probably should).

```
  list_item u_it;
  forall_items(u_it, unfinished)
  { node v = unfinished[u_it];
    color col = gw.get_color(v);
    int   dn  = dfsnum[v];
    state_win.draw_box(x,y1,x+d,y1+d,col);
    state_win.draw_rectangle(x,y1-1,x+d,y1+d,black);
    state_win.draw_ctext(x+d/2,y1+d/2,string("%d",dn),text_color(col));
    if ( v == roots[r_it] )
    { state_win.draw_box(x,y2,x+d,y2+d,col);
      state_win.draw_rectangle(x,y2-1,x+d,y2+d,black);
      state_win.draw_ctext(x+d/2,y2+d/2,string("%d",dn),
                                                      text_color(col));
      r_it = roots.succ(r_it);
     }
    else
      state_win.draw_box(x+1,y2,x+d,y2+d,white);
    x += d;
   }
  state_win.draw_rectangle(x0,y1-1,x,y1+d,black);
  state_win.draw_rectangle(x0,y2-1,x,y2+d,black);
  state_win.flush_buffer();
}
```

The functions *STRONG_COMPONENTS* and *SCC_DFS* have the same overall structure as in the preceding section, but are augmented by display actions. At the beginning of a call *SCC_DFS*$(v, \ldots)$ we call *gw.select*(v) to highlight v and at the end of the call we call *gw.deselect*(v) to unhighlight v. In the *forall_adj_edges_loop* we color the edge explored red and make it solid.

⟨*display functions for part two*⟩$+\equiv$

```
void SCC_DFS(node v, const graph& G, node_array<int>& dfsnum,
             node_array<int>& comp_num, list<node>& unfinished,
             list<node>& roots, int& dfscount, int& comp_count,
             GraphWin& gw, const node_array<point>& perm_pos)
{ gw.select(v);
  ⟨new node v was reached⟩
  node w; edge e;
  forall_adj_edges(e,v)
  { w = G.target(e);
    gw.set_style(e,solid_edge);
    gw.set_color(e,red);
    string msg = "I am exploring the red edge.\\3 ";
    if (dfsnum[w] == - 1) { ⟨tree edge and recursive call⟩ }
    else if (comp_num[w] == - 1)
           { ⟨non-tree edge into tentative component⟩ }
         else
           { ⟨non-tree edge into permanent component⟩ }
  }
```

```
  if (v == roots.head()) { ⟨v is a root⟩ }
  gw.deselect(v);
}
```

In *STRONG_COMPONENTS* we inform the user about every new call of *SCC_DFS* except for the first.

⟨*display functions for part two*⟩+≡

```
int STRONG_COMPONENTS(const graph& G, node_array<int>& comp_num,
                GraphWin& gw, const node_array<point>& perm_pos)
{ list<node> unfinished;
  list<node> roots;
  node_array<int> dfsnum(G,-1);
  node v;
  forall_nodes(v,G) comp_num[v] = -1;
  int dfscount = 0;
  int comp_count = 0;
  forall_nodes(v,G)
    if (dfsnum[v] == -1)
    { SCC_DFS(v,G,dfsnum,comp_num,unfinished,roots,dfscount,
                                   comp_count,gw,perm_pos);
      message(gw,"This was a return from an outermost call\\3
                I am looking for an unreached node and \\n\
                (if successful) start a new search from it.");
    }
  return comp_count;
}
```

When a new node is reached it is given a dfs-number and is pushed on *unfinished* and *roots*. The new node forms a tentative strongly connected component of its own. We set the color of v to the size of the *roots*-stack (shifted by *color_shift* so as to avoid too much overlap with the colors used for permanent components), we set the user label of v to its dfs-number, and we set the shape and width of v to a large rectangular shape (so as to indicate that v is a root). We build up a string to explain our actions, hand it to *message* to display it, and call *state_info* to update the state information.

⟨*new node v was reached*⟩≡

```
dfsnum[v] = dfscount++;
unfinished.push(v);
roots.push(v);
gw.set_color(v,(color_shift + roots.size())%16);
gw.set_user_label(v,string("%d",dfsnum[v]));
gw.set_shape(v,rectangle_node);
gw.set_width(v,large_width);
string msg;
msg += "A new node has been reached.\\3 ";
msg += "It got the dfs-number ";
msg += string("%d ",dfsnum[v]);
msg += "and it is the new current node.\\3 ";
```

```
msg += "It is the root of a new tentative component.";
state_info(gw,unfinished,roots,dfsnum,v);
message(gw,msg);
```

A tree edge $e = (v, w)$ leads to a recursive call. We inform the reader about this fact by textual output, we unhighlight v as it ceases to be a current node, and we emphasize the edge e (by increasing its width and setting its color to blue); in this way the tree path to the current node is always shown as a path of thick blue edges. Then we make the recursive call. After the return from the recursive call, we de-emphasize e and highlight v (again), and we inform the reader that we just returned from a recursive call and that v became active again.

⟨*tree edge and recursive call*⟩≡

```
msg += "It's a tree edge and I am making a recursive call.";
message(gw,msg);
state_info(gw,unfinished,roots,dfsnum,v);
gw.deselect(v);
gw.set_color(e,blue);
gw.set_width(e,2);
SCC_DFS(w,G,dfsnum,comp_num,unfinished,roots,dfscount,
                                    comp_count,gw, perm_pos);
gw.set_width(e,1);
gw.set_color(e,black);
gw.select(v);
state_info(gw,unfinished,roots,dfsnum,0);
message(gw,"I returned from a recursive call. The node with \
number " + string("%d ",dfsnum[v]) + " got reactivated");
```

A non-tree edge $e = (v, w)$ into a tentative component may close a cycle involving several tentative components. These components are merged into one. More precisely, all components whose root has a dfs-number larger than *dfsnum*$[w]$ cease to exist. We inform the user about this fact by textual output and then start popping *roots*. Whenever a node is popped from *roots* its shape and width are changed to a small circle. We put a *wait*(0.25) statement into the loop that pops from *roots* so that different roots are visibly popped one after the other. Once all roots are popped we recolor the nodes in the newly formed scc and give state information. Finally, we change the color of e back to black.

⟨*non-tree edge into tentative component*⟩≡

```
msg += "It's a non-tree edge into a tentative component. This edge may \
        merge several components into one.\\n More precisely: all \
        components whose root is larger than " + string("%d ",dfsnum[w]);
msg += "cease to exist and are merged into the component \
        containing the node with dfs-number " + string("%d. ",dfsnum[w]);
msg += "Algorithmically, this amounts to removing all roots \
        larger than " + string("%d ",dfsnum[w]);
msg += "from the stack of roots. I do so one by one. Removal of a node \
```

```
          from the stack of roots turns its shape from rectangular \
          to circular.";
  message(gw,msg);
  state_info(gw,unfinished,roots,dfsnum,v);
  while (dfsnum[roots.head()] > dfsnum[w])
  { node z = roots.pop();
    gw.set_shape(z,circle_node);
    gw.set_width(z,small_width);
    state_info(gw,unfinished,roots,dfsnum,v);
    wait(0.25);
  }
  node u;
  forall(u,unfinished)
    if (dfsnum[u] >= dfsnum[roots.head()] )
      gw.set_color(u,(color_shift + roots.size())%16);
  state_info(gw,unfinished,roots,dfsnum,0);
  message(gw,string("Now all roots are removed and the newly formed \
                     component has been recolored. The current \
                     node is still: %d.", dfsnum[v]));
  gw.set_color(e,black);
```

A non-tree edge *e* into a permanent component requires no action. We inform the user and change the color of *e* back to black.

⟨*non-tree edge into permanent component*⟩≡

```
  msg += "It's a non-tree edge into a permanent component. I do nothing.";
  message(gw,msg);
  state_info(gw,unfinished,roots,dfsnum,v);
  gw.set_color(e,black);
```

When a call *SCC_DFS*$(v, \ldots)$ for a root v is completed a permanent component has been found. We inform the reader accordingly. All nodes in the permanent component are moved to the left half of the window (by setting their position as given by *perm_pos*, the shape and width is changed to a large rectangular shape, the user label is set to a pair consisting of dfs-number and component number, and the color is set to the color corresponding to the component number.

⟨*v is a root*⟩≡

```
  string msg = "Node " + string("%d",dfsnum[v]) + " has been \
         completed. It is a root and hence we have identified \
         a permanent component. \\3 \
         The permanent component consists of all nodes  in \
         unfinished whose dfs-number is at least as large as "
         + string("%d",dfsnum[v]) + ". \\3 \
         I move all nodes in the component to the left and \
         indicate their dfs-number and their component number.";
  state_info(gw,unfinished,roots,dfsnum,0);
```

```
message(gw,msg);
do { w = unfinished.pop();
     if (v == w) roots.pop();
     comp_num[w] = comp_count;
     gw.set_shape(w,rectangle_node);
     gw.set_width(w,large_width);
     gw.set_color(w,comp_count%16);
     gw.set_user_label(w,string("%d | %d", dfsnum[w],comp_num[w]));
     state_info(gw,unfinished,roots,dfsnum,0);
     gw.set_position(w,perm_pos[w]);
} while ( w != v);
comp_count++;
```

Enjoy the animation.

Exercises for 7.4

1 Modify the algorithm for the computation of strongly connected components to compute biconnected components of undirected graphs. Hint: Define the root of a biconnected component as the node in the component with the second largest dfs-number. Then proceed as for strongly connected components.

2 Part one of the animation of strongly connected components is unsatisfactory as color changes are not "local". It would be desirable to have the following behavior: after the addition or deletion of a node or edge only the colors of those nodes change whose containing strongly connected component has changed. Modify the animation to achieve this behavior.

3 Animate the biconnected components algorithm of the first item.

4 Extend the first part of the animation of strongly connected components so that the shrunken graph is also visualized. A reasonable approach seems to represent each vertex of the shrunken graph by the convex hull of the vertices of the corresponding strongly connected component.

5 Define the shrunken graph of an undirected graph with respect to its biconnected components as follows. There is a vertex for each biconnected component and for each articulation point. A vertex standing for a component is connected to a vertex representing an articulation point if the articulation point is contained in the component. Show that the shrunken graph is a tree and give a program that computes it.

7.5 Shortest Paths

We introduce the shortest-path problem and describe the functionality of our various shortest-path programs. We discuss a checker for the single-source shortest-path problem and derive a generic shortest-path algorithm. We give algorithms and their implementations for acyclic networks, for the single-source problem with arbitrary edge costs, for the single-source single-sink problem, for the all-pairs problem, and for the minimum cost to profit

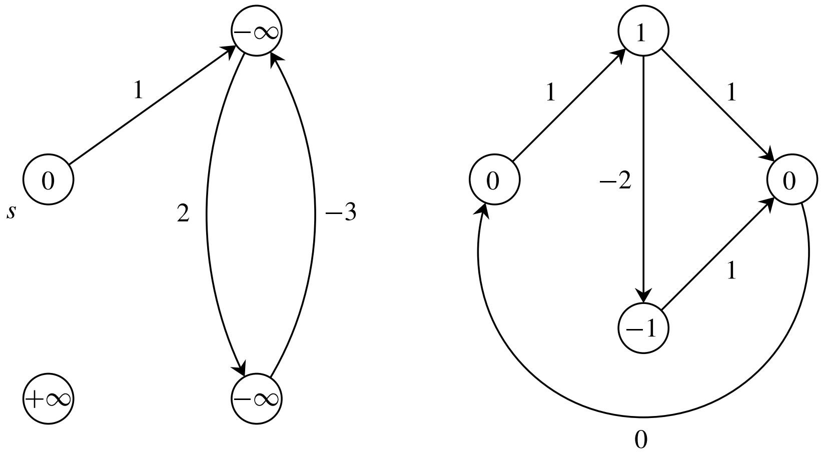

Figure 7.8 The node labels indicate $\mu(s, .)$. The graph on the left contains a negative cycle and also a node that is not reachable from s. Therefore there are node labels equal to $\pm\infty$. The graph on the right contains no negative cycle and all nodes are reachable from s. Therefore all node labels are finite.

ratio cycle problem; an algorithm for the single-source problem with non-negative edge costs was already given in Section 6.6. We also give experimental results about the running times of the various implementations.

7.5.1 *Functionality*

Let $G = (V, E)$ be a directed graph and let $c : E \longrightarrow \mathbb{R}$ be a *cost* function on the edges of G. We will also say *length* instead of cost. We extend the cost function to *paths* in the natural way: the cost (or length) of a path is the sum of the costs of its constituent edges, i.e., if $p = [e_1, e_2, \ldots, e_k]$ is a path then $c(p) = \sum_{1 \le i \le k} c(e_i)$. We will abuse notation and write $c(u, v)$ instead of $c(e)$ for $e = (u, v)$. For every vertex $v \in G$ the trivial path consisting of no edge is a path from v to v; its cost is zero. A *cycle* is a non-trivial path from v to v for some node v. A *negative cycle* is a cycle whose cost is negative.

For two vertices v and w we use $\mu(v, w)$ to denote the minimal cost of a path from v to w, i.e.,

$$\mu(v, w) = \inf\{c(p) \; ; \; p \text{ is a path from } v \text{ to } w\}\,.$$

The infimum of the empty set is defined as $+\infty$, i.e., $\mu(v, w) = +\infty$ if w is not reachable from v. Figure 7.8 illustrates this definition. The set of paths from v to w is in general an infinite set and hence it is not clear whether $\mu(v, w)$ is actually achieved by a path from v to w. The following lemma gives information about the existence of shortest paths.

Lemma 5

(a) *If w is not reachable from v then $\mu(v, w) = +\infty$.*
(b) *If there is a path from v to w containing a negative cycle then $\mu(v, w) = -\infty$.*

(c) *If w is reachable from v and there is no path from v to w passing through a negative cycle then $-\infty < \mu(v, w) < +\infty$ and $\mu(v, w)$ is the length of a simple path from v to w.*

(d) *If $\mu(v, w) = -\infty$ then there is a path from v to w containing a negative cycle.*

Proof Part (a) is true by definition.

For part (b), we observe that if there is a path from v to w containing a negative cycle then by going around the cycle sufficiently often a path from v to w whose cost is below any prescribed number is obtained. Thus $\mu(v, w) = -\infty$.

For part (c) consider any path p from v to w. If p contains a cycle let p' be obtained by removing a cycle from p. Since p contains no negative cycle we have $c(p') \le c(p)$. Continuing in this way we obtain a simple path from v to w whose cost is at most the cost of p. Thus

$$\mu(v, w) = \inf\{c(p)\ ;\ p \text{ is a simple path from } v \text{ to } w\}.$$

The number of simple paths from v to w is finite and hence $\mu(v, w) = c(p)$ for some simple path p.

We turn to part (d). If $\mu(v, w) = -\infty$ then w is reachable from v. If there is no path from v to w containing a negative cycle then $\mu(v, w) > -\infty$ by part (c). □

We distinguish between the *single-source single-sink shortest-path problem*, the *single-source shortest-path problem*, and the *all-pairs shortest-path problem*. The first problem asks for the computation of $\mu(s, t)$ for two specified nodes s and t and will be discussed in Section 7.5.6. The second problem asks to compute $\mu(s, v)$ for a specified node s and all v and the third problem asks to compute $\mu(s, v)$ for all nodes s and v. The single-source problem is the basis for the solutions to the other two problems and hence we discuss it first.

In our discussion of the single-source problem we use s to denote the source and we write $\mu(v)$ instead of $\mu(s, v)$. The following characterization of the function μ is extremely useful for the correctness proofs of shortest-path algorithms[7].

Lemma 6

(a) *We have*

$$\mu(s) = \min(0, \min\{\mu(u) + c(e)\ ;\ e = (u, s) \in E\})$$

and

$$\mu(v) = \min\{\mu(u) + c(e)\ ;\ e = (u, v) \in E\}$$

for $v \neq s$.

(b) *If d is a function from V to $\mathbb{R} \cup \{-\infty, +\infty\}$ with*

- *$d(v) \ge \mu(v)$ for all $v \in V$,*

[7] In this characterization and for the remainder of the section we use the following definitions for the arithmetic and order on $\mathbb{R} \cup \{-\infty, +\infty\}$: $-\infty < x < +\infty$, $+\infty + x = +\infty$, and $-\infty + x = -\infty$ for all $x \in \mathbb{R}$.

- $d(s) \le 0$, *and*
- $d(v) \le d(u) + c(u, v)$ *for all* $e = (u, v) \in E$

then $d(v) = \mu(v)$ *for all* $v \in V$.

Proof For part (a) we consider only the case $v \ne s$ and leave the case $v = s$ to the reader. Any path p from s to v consists of a path from s to some node u plus an edge from u to v. Thus

$$\begin{aligned} \mu(v) &= \inf\{c(p)\ ;\ p \text{ is a path from } s \text{ to } v\} \\ &= \min_u \inf\left\{c(p') + c(e)\ ;\ p' \text{ is a path from } s \text{ to } u \text{ and } e = (u, v) \in E\right\} \\ &= \min\{\mu(u) + c(e)\ ;\ e = (u, v) \in E\}. \end{aligned}$$

For part (b) we assume for the sake of a contradiction that $d(v) > \mu(v)$ for some v. Then $\mu(v) < +\infty$. We distinguish cases.

If $\mu(v) > -\infty$, let $[s = v_0, v_1, \ldots, v_k = v]$ be a shortest path from s to v. We have $\mu(s) = 0 = d(s)$, $\mu(v_i) = \mu(v_{i-1}) + c(v_{i-1}, v_i)$ for $i > 0$, and $\mu(v) < d(v)$. Thus, there is a least $i > 0$ with $\mu(v_i) < d(v_i)$ and hence

$$d(v_i) > \mu(v_i) = \mu(v_{i-1}) + c(v_i, v_{i-1}) = d(v_{i-1}) + c(v_i, v_{i-1}),$$

a contradiction.

If $\mu(v) = -\infty$, let $[s = v_0, v_1, \ldots, v_i, \ldots, v_j, \ldots, v_k = v]$ be a path from s to v containing a negative cycle. Such a path exists by Lemma 5. Assume that the subpath from v_i to v_j is a negative cycle. If $d(v) > \mu(v)$ then $d(v) > -\infty$ and hence $d(v_l) > -\infty$ for all l, $0 \le l \le k$. Thus,

$$\begin{aligned} d(v_i) &= d(v_j) && \text{since } v_i = v_j \\ &\le d(v_{j-1}) + c(v_{j-1}, v_j) \\ &\le d(v_{j-2}) + c(v_{j-2}, v_{j-1}) + c(v_{j-1}, v_j) \\ &\vdots \\ &\le d(v_i) + \textstyle\sum_{l=i}^{j-1} c(v_l, v_{l+1}), \end{aligned}$$

and hence $\sum_{l=i}^{j-1} c(v_l, v_{l+1}) \ge 0$, a contradiction to the fact that the subpath from v_i to v_j is a negative cycle. □

We split the set of vertices of G into three sets:

$$\begin{aligned} V^- &= \{v \in V\ ;\ \mu(v) = -\infty\}, \\ V^f &= \{v \in V\ ;\ -\infty < \mu(v) < +\infty\}, \text{ and} \\ V^+ &= \{v \in V\ ;\ \mu(v) = +\infty\}. \end{aligned}$$

The vertex s belongs to V^f if there is no negative cycle passing through s and it belongs to V^- otherwise; in the latter case V^f is empty. The set V^+ consists of all vertices that are not

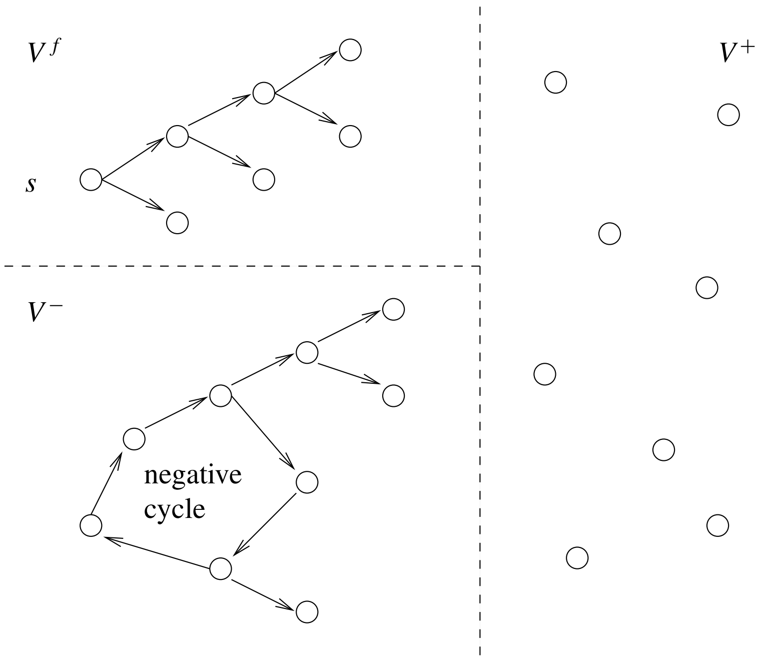

Figure 7.9 A solution to a single-source problem: It consists of a shortest-path tree on V^f, a collection of negative cycles plus trees emanating from them on V^-, a set V^+ of isolated nodes, and the values $\mu(v)$ for $v \in V^f$.

reachable from s. A *shortest-path tree* with respect to s is a tree defined on V^f such that for any $v \in V^f$ the tree path from s to v is a shortest path from s to v.

We next define the output convention for the single-source shortest-path problem. What do we want to know? Certainly, $\mu(v)$ for all nodes v. However, knowing $\mu(v)$ is usually not enough. If $v \in V^f$, it is useful to know a shortest path from s to v and if $v \in V^-$, it is useful to know the negative cycle that "puts" v into V^-. Our algorithms therefore also produce a shortest-path tree on V^f and a collection of negative cycles plus trees emanating from them on V^-, see Figure 7.9. The exact definition is as follows[8]:

The solution to a single-source shortest-path problem (G, s, c) is a pair (*dist*, *pred*), where *dist* is a *node_array*<*NT*> and *pred* is a *node_array*<*edge*>. Let

$$P = \{pred[v]\ ;\ v \in V \text{ and } pred[v] \neq nil\}\,.$$

The pair must have the following properties:

- $s \in V^f$ iff $pred[s] = nil$ and $s \in V^-$ iff $pred[s] \neq nil$.
- For $v \neq s$: $v \in V^+$ iff $pred[v] = nil$ and $v \in V^f \cup V^-$ iff $pred[v] \neq nil$.

[8] We further comment on our output convention after its definition.

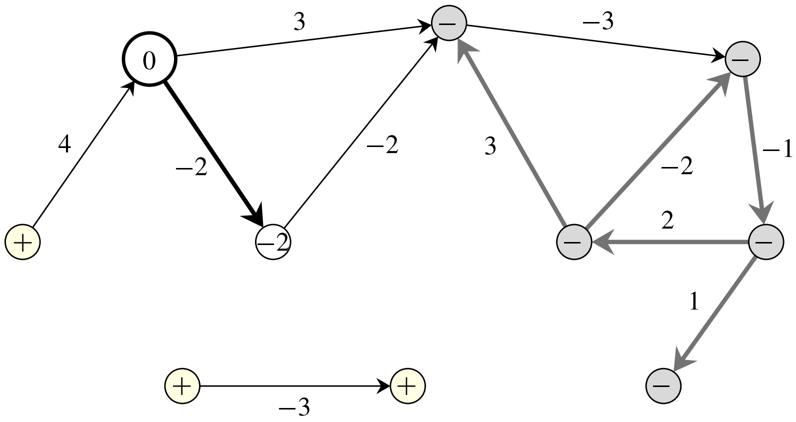

Figure 7.10 The output of a single-source shortest-path problem. The source node s is shown bigger than all other nodes. Its *dist*-label is zero. Edge costs are indicated. For every node v with $pred[v] \neq nil$ the edge $pred[v]$ is shown in bold. For the nodes in V^f the *dist*-value is shown inside the node. For nodes $v \in V^+ \cup V^-$ the set containing v is indicated by a $+$ or $-$. V^+ consists of all nodes $v \neq s$ with $pred[v] = nil$, V^f consists of all nodes that are reachable from s by a P-path, and V^- consists of all nodes that lie on a P-cycle or are reachable from a P-cycle by the P-path. All P-cycles have negative cost. You may generate your own figures with the xlman-demo gw_shortest_path.

- $v \in V^f$ if v is reachable from s by a P-path[9] and $s \in V^f$. P restricted to V^f forms a shortest-path tree and $dist[v] = \mu(v)$ for $v \in V^f$.
- All P-cycles have negative cost and $v \in V^-$ iff v lies on a P-cycle or is reachable from a P-cycle by a P-path.

Figure 7.10 shows an example. Observe that our output convention leaves the value of $dist[v]$ unspecified for $v \in V^+ \cup V^-$. We have made this choice because most number types have no representation for $+\infty$ and $-\infty$. In the *absence of negative cycles* our output convention simplifies to the following:

- For $v \neq s$: $v \in V^f$ iff $pred[v] \neq nil$ and $v \in V^+$ otherwise.
- $pred[s] = nil$.
- P is a shortest-path tree on V^f and $dist[v] = \mu(v)$ for $v \in V^f$.

Our output convention for the single-source shortest-path problem is non-standard. Most papers on the shortest-path problem do not define precisely how negative cycles are reported and this was also true for early versions of LEDA. We have defined our output convention such that:

- the return value of a single-source algorithm consists of a pair (*dist*, *pred*), as is customary for single-source algorithms. We played with the idea to add an output

[9] A P-path is a path all of whose edges belong to P and a P-cycle is a cycle all of whose edges belong to P.

parameter, which indicates for every node its membership to the sets V^+, V^f, and V^-. We decided against it, because we wanted to stick with the traditional interface of shortest-path algorithms,

- it can be checked in linear time whether a pair (*dist*, *pred*) is a solution to the shortest-path problem (G, s, c), see Section 7.5.2,
- shortest-path algorithms can satisfy it with little additional effort.

We turn to algorithms. All algorithms are function templates that work for an arbitrary number type *NT*. We use the convention that names of function templates for graph algorithms end with *_T*. In order to use the templates one must include LEDA/templates/shortest_path.t. LEDA also contains pre-compiled instantiations for the number types *int* and *double*. The function names for the instantiated versions are *without* the suffix *_T*. In order to use the instantiated versions one must include LEDA/graph_alg.h. Section 7.1 discusses the relationship between templates and instantiated versions in more detail.

Acyclic Graphs:

```
void ACYCLIC_SHORTEST_PATH_T(const graph& G, node s,
                             const edge_array<NT>& c,
                             node_array<NT>& dist,
                             node_array<edge>& pred)
```

solves the problem in time $O(n + m)$ for acyclic graphs, see Section 7.5.4. As always, we use n to denote the number of nodes of G and m to denote the number of edges of G.

Non-Negative Edge Costs:

```
void DIJKSTRA_T(const graph& G, node s, const edge_array<NT>& c,
                node_array<NT>& dist, node_array<edge>& pred)
```

solves the problem in time $O(m + n \log n)$ if all edge costs are non-negative. We have discussed this function already in Section 6.6. If all edge costs are equal to one then breadth-first search, see Section 7.3, solves the problem in linear time.

General Edge Costs:

```
bool BELLMAN_FORD_T(const graph& G, node s, const edge_array<NT>& c,
                    node_array<NT>& dist, node_array<edge>& pred)
```

solves the problem in time $O(n \cdot m)$ for arbitrary edge costs. It returns false if $\mu(v) = -\infty$ for some vertex v. Otherwise, it returns *true*.

We also have a procedure

```
bool SHORTEST_PATH_T(const graph& G, node s, const edge_array<NT>& c,
                     node_array<NT>& dist, node_array<edge>& pred)
```

that tests whether one of the two special cases applies and, if so, applies the efficient procedure applicable to the special case. If none of the special cases applies, *BELLMAN_FORD_T* is called. The implementation of *SHORTEST_PATH_T* is simple.

⟨*SP.t*⟩≡

```
template <class NT>
bool SHORTEST_PATH_T(const graph& G, node s, const edge_array<NT>& c,
                     node_array<NT>& dist, node_array<edge>& pred )
{ if ( Is_Acyclic(G) )
  { ACYCLIC_SHORTEST_PATH_T(G,s,c,dist,pred);
    return true;
  }
  bool non_negative = true;
  edge e;
  forall_edges(e,G) if (c[e] < 0) non_negative = false;
  if (non_negative) { DIJKSTRA_T(G,s,c,dist,pred);
                      return true;
                    }
  return BELLMAN_FORD_T(G,s,c,dist,pred);
}
```

The Single-Sink Problem: The single-source single-sink shortest-path problem asks for the computation of a shortest path from a specified node s, the source, to a specified node t, the sink.

```
NT DIJKSTRA_T(const graph& G, node s, node t,
              const edge_array<NT>& c, node_array<edge>& pred)
```

computes a shortest path from s to t and returns its length. The cost of all edges must be non-negative. The return value is unspecified if there is no path from s to t. The array *pred* allows one to trace a shortest path from s to t in reverse order, i.e., $pred[t]$ is the last edge on the path. If there is no path from s to t or if $s = t$ then $pred[t] = nil$. The worst case running time is $O(m + n \log n)$, but frequently much better. The implementation is discussed in Section 7.5.6.

The All-Pairs Problem: The all-pairs shortest-path problem asks for the computation of the complete distance function μ.

```
bool ALL_PAIRS_SHORTEST_PATHS_T(graph& G, edge_array<NT> c,
                                node_matrix<NT> DIST)
```

returns *true* if G has no negative cycle and returns *false* otherwise. In the latter case all values returned in *DIST* are unspecified. In the former case we have for all v and w: if $\mu(v, w) < \infty$ then $DIST(v, w) = \mu(v, w)$ and if $\mu(v, w) = \infty$, the value of $DIST(v, w)$ is unspecified. The procedure runs in time $O(nm + n^2 \log n)$.

Our output convention for the all-pairs problem is somewhat unsatisfactory. It is dictated by the fact that many number types have no representation of $+\infty$. An alternative solution is to also return a node matrix *PRED* of edges in analogy to the single-source problem.

7.5.2 *A Checker for Single-Source Shortest-Path Algorithms*

We develop a program *CHECK_SP_T*$(G, s, c, dist, pred)$ that checks whether $(dist, pred)$ is a correct solution to the shortest-path problem (G, s, c). If not, the program aborts (with the error message "assertion failed") and if so, the program returns a *node_array<int> label* with $label[v] < 0$ if $v \in V^-$, $label[v] = 0$ if $v \in V^f$, and $label[v] > 0$ if $v \in V^+$.

Let $P = \{pred[v]\,; pred[v] \neq nil\}$ be the set of edges defined by the *pred*-array and define

$$
\begin{aligned}
U^+ &= \{v\,;\, v \neq s \text{ and } pred[v] = nil\}\,,\\
U^f &= \emptyset, \text{ if } pred[s] \neq nil,\\
U^f &= \{v\,;\, v \text{ is reachable from } s \text{ by a } P\text{-path}\}\,, \text{ if } pred[s] = nil,\\
U^- &= \{v\,;\, v \text{ lies on a } P\text{-cycle or is reachable from a } P\text{-cycle by a } P\text{-path}\}\,.
\end{aligned}
$$

We perform the following checks:

(1) $v \in U^+$ iff v is not reachable from s in G.
(2) All P-cycles have negative cost.
(3) There is no edge $(v, w) \in E$ with $v \in U^-$ and $w \in U^f$.
(4) For all $e = (v, w) \in E$: if $v \in U^f$ and $w \in U^f$ then $dist[v] + c[e] \geq dist[w]$.
(5) For all $v \in U^f$: if $v = s$ then $dist[v] = 0$ and if $v \neq s$ then $dist[v] = dist[u] + c[pred[v]]$ where u is the source of $pred[v]$.

Lemma 7 *If* $(dist, pred)$ *satisfies the five conditions above then it is a solution to the shortest-path problem* (G, s, c).

Proof Observe first that $v \in V^+$ iff v is not reachable from s. Thus (1) implies that $U^+ = V^+$ and hence $U^f \cup U^- = V^f \cup V^-$. We next show that $U^- \subseteq V^-$. Consider any $v \in U^-$. By definition of U^- there is a P-cycle, call it C, from which v is reachable. Moreover, the cost of C is negative by (2) and there is a node on C that is reachable from s by (1). Thus $\mu(s, v) = -\infty$ and hence $v \in V^-$. Thus $U^- \subseteq V^-$ and therefore $U^f \supseteq V^f$. Assume for the sake of a contradiction that the latter inclusion is proper and let $v \in U^f \setminus V^f$ be arbitrary. Then $v \in V^-$ and hence there is a path p from s to v containing a negative cycle, say C. By (3) there is no edge (x, y) with $x \in U^-$ and $y \in U^f$. We conclude that all vertices on p belong to U^f. This implies that (4) holds for all edges of C. Let $e_0, e_1, \ldots, e_{k-1}$ with $e_i = (v_i, v_{i+1})$ be the edges of C. Then $v_0 = v_k$. We have

$$dist[v_0] + c(C) = dist[v_0] + \sum_{0 \leq i < k} c[e_i]$$

$$\begin{aligned} &\geq \; dist[v_1] + \sum_{1 \leq i < k} c[e_i] \;\geq\; \ldots \;\geq\; dist[v_k] \\ &= \; dist[v_0] \end{aligned}$$

by repeated application of (4). Thus $c(C) \geq 0$, a contradiction. We have now shown that $U^+ = V^+$, $U^f = V^f$, and $U^- = V^-$. We still need to show that P restricted to V^f is a shortest-path tree. Consider any $v \in V^f$. Condition (5) implies that $dist[v]$ is the length of the P-path from s to v and (4) implies that the length of any path from s to v is at least $dist[v]$. Thus P is a shortest-path tree. □

We come to the implementation. We start with condition (1). We use depth-first search to determine all nodes reachable from s and we check whether for all nodes v different from s: $pred[v] = nil$ iff v is not reachable from s. We give all nodes that are not reachable from s the label *PLUS*; *PLUS* is an element of an enumeration type that we use to classify nodes. All nodes start with the label *UNKNOWN*. The other members of the enumeration type will be explained below.

⟨*condition one*⟩≡

```
enum{ NEG_CYCLE = -2,  ATT_TO_CYCLE = -1, FINITE = 0, PLUS = 1,
      CYCLE = 2, ON_CUR_PATH = 3, UNKNOWN = 4  };
node_array<int> label(G,UNKNOWN);
node_array<bool> reachable(G,false);
DFS(G,s,reachable);
node v;
forall_nodes(v,G)
{ if (v != s)
  { assert( (pred[v] == nil) == (reachable[v] == false));
    if (reachable[v] == false) label[v] = PLUS;
  }
}
```

Next we compute the sets U^f and U^-. Consider any node $v \notin U^+$. Tracing the path $[v, source(pred[v]), source(pred[source(pred[v])]), \ldots]$ until either a node is encountered twice or until the path cannot be extended further (it must end in s in the latter case because s is the only node outside U^+ which may have no incoming P-edge) allows us to classify all nodes on the path. In the former case v and all nodes on the path belong to U^- and in the latter case all of them belong to U^f. For the sequel it is useful to have a finer classification of the nodes in U^- into nodes lying on a P-cycle (label *CYCLE*) and nodes attached to a cycle by a P-path (label *ATT_TO_CYCLE*) and so we will compute the finer classification.

Of course, we do not want to trace the same path several times. We therefore stop tracing a path once a node is reached whose label is known (more precisely, is different from *UNKNOWN*). As we trace a path all nodes on the path are put onto a stack S and are given the label *ON_CUR_PATH*.

We initialize the classification step by giving s the label *FINITE* if its *pred*-value is *nil*.

⟨*classification of nodes*⟩≡

```
if (pred[s] == nil) label[s] = FINITE;
forall_nodes(v,G)
{ if ( label[v] == UNKNOWN )
  { stack<node> S;
    node w = v;
    while ( label[w] == UNKNOWN )
    { label[w] = ON_CUR_PATH;
      S.push(w);
      w = G.source(pred[w]);
    }
    ⟨label all nodes on current path⟩
  }
}
```

When a node w is encountered whose label is different from *UNKNOWN* we distinguish cases: if w is labeled *FINITE*, i.e., $v \in U^f$, then all nodes on the path belong to U^f, and if w is labeled *CYCLE* or *ATT_TO_CYCLE*, i.e., $v \in U^-$, then all nodes on the path (except for w) are attached to a cycle but do not lie on a cycle themselves, and if w belongs to the current path then the situation is as shown in Figure 7.11. This leads to the following code.

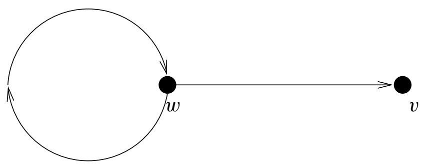

Figure 7.11 A cycle and a path emanating from it. The search started in v and w is the first node encountered twice.

⟨*label all nodes on current path*⟩≡

```
int t = label[w];
if ( t == ON_CUR_PATH )
{ node z;
  do { z = S.pop();
       label[z] = CYCLE;
     }
  while ( z != w );
  while ( !S.empty() ) label[S.pop()] = ATT_TO_CYCLE;
}
else // t is CYCLE, ATT_TO_CYCLE, or FINITE
{ if ( t == CYCLE ) t = ATT_TO_CYCLE;
  while ( !S.empty() ) label[S.pop()] = t;
}
```

We next check that all P-cycles have negative cost. Given our classification of nodes this is fairly simple. For every cycle node we trace the cycle containing it and compute its cost. We assert that the cost is negative. If so, we promote all nodes on the cycle to label *NEG_CYCLE*; this guarantees that every cycle is traced only once.

⟨*condition two*⟩≡

```
forall_nodes(v,G)
{ if ( label[v] == CYCLE )
  { node w = v;
    NT cycle_length = 0;
    do
    { cycle_length += c[pred[w]];
      label[w] = NEG_CYCLE;
      w = G.source(pred[w]);
    } while (w != v);
    assert(cycle_length < 0);
  }
}
```

Conditions (3), (4), and (5) are trivial to check.

⟨*conditions three, four, and five*⟩≡

```
if ( label[s] == FINITE ) assert(dist[s] == 0);
edge e;
forall_edges(e,G)
{ node v = G.source(e);
  node w = G.target(e);
  if ( label[w] == FINITE )
  { assert( label[v] == FINITE || label[v] == PLUS);
    if ( label[v] == FINITE )
    { assert( dist[v] + c[e] >= dist[w] );
      if ( e == pred[w] ) assert( dist[v] + c[e] == dist[w] );
    }
  }
}
```

Putting it all together we obtain:

⟨*check_sp.t*⟩+≡

```
template <class NT>
node_array<int> CHECK_SP_T(const graph& G, node s,
                           const edge_array<NT>& c,
                           const node_array<NT>& dist,
                           const node_array<edge>& pred)
{ ⟨condition one⟩
  ⟨classification of nodes⟩
  ⟨condition two⟩
  ⟨conditions three, four, and five⟩
  return label;
}
```

7.5.3 *A Generic Single-Source Shortest-Path Algorithm*

We derive a generic shortest-path algorithm. All our implementations for the single-source problem will be instances of the generic algorithm and the correctness proofs and running time claims of our implementations will be consequences of the lemmas derived in this section.

In Lemma 6 we gave a characterization of shortest-path distances.

Let $d : V \longrightarrow \mathbb{R} \cup \{-\infty, \infty\}$ be a function with

$$\begin{array}{llcll} (1) & d(v) & \geq & \mu(v) & \text{for all } v \in V \\ (2) & d(s) & \leq & 0 & \\ (3) & d(v) & \leq & d(u) + c(u,v) & \text{for all } e = (u,v) \in E \end{array}$$

Then $d(v) = \mu(v)$ for all $v \in V$.

The generic algorithm maintains a function d satisfying (1) and (2) and aims at establishing (3). We call $d(v)$ the *tentative distance label* of v.

$d(s) = 0$; $d(v) = \infty$ for $v \neq s$;
$\pi(v) = nil$ for all $v \in V$;
while there is an edge $e = (u,v) \in E$ with $d(v) > d(u) + c(e)$
{ $d(v) = d(u) + c(e)$;
  $\pi(v) = e$;
}

We will refer to the body of the while-loop as *relaxing*[10] *edge* e. Besides the tentative distance labels the generic algorithm maintains for each node v the edge $\pi(v)$ that defined $d(v)$.

It is easy to see that (1) and (2) are invariants of the algorithm. We only have to observe that $d(v)$ never increases (and hence $d(s) \leq 0$ always) and that $d(v) < +\infty$ implies that $d(v)$ is the length of some path from s to v (and hence $d(v) \geq \mu(v)$ always). When the algorithm terminates we also have (3). Thus, $d(v) = \mu(v)$ for all $v \in V$ when the algorithm terminates. A lot more can be said about the generic algorithm.

Lemma 8 *The following is true at any time during the execution of the generic algorithm*[11]. *Let*

$$P = \{e\, ;\, e = \pi(v) \in E \text{ for some } v \in V\}.$$

(a) $d(s) = 0$ *iff* $\pi(s) = nil$ *and* $d(v) < \infty$ *iff* $\pi(v) \neq nil$ *for* $v \neq s$.
(b) *If* $\pi(v) = e = (u,v)$ *then* $d(v) \geq d(u) + c(e)$.
(c) *If* $\pi(v) \neq nil$ *then* v *either lies on a* P*-cycle, or is reachable from a* P*-cycle by a* P*-path, or is reachable from* s *by a* P*-path. If* $\pi(s) \neq nil$ *then* s *lies on a* P*-cycle.*

[10] Think of $e = (u,v)$ as a rubber band that wants to keep v within distance $c(e)$ of u. If $d(v) > d(u) + c(e)$ the rubber band is under tension. Setting $d(v)$ to $d(u) + c(e)$ relaxes it.

[11] Observe the similarity of items (a), (d), (e), (f), and (g) with the four bullets in the definition of our output convention.

(d) *P-cycles have negative cost.*
(e) *If v lies on a P-cycle or is reachable from a P-cycle then $\mu(v) = -\infty$.*
(f) *If $v \in V^f$ and $d(v) = \mu(v)$ then there is a P-path from s to v and this path has cost $\mu(v)$.*
(g) *If $d(v) = \mu(v)$ for all $v \in V^f$ then P defines a shortest-path tree on V^f.*

Proof (a) We start with $d(s) = 0$, $d(v) = \infty$ for all v with $v \neq s$, and $\pi(v) = nil$ for all v. When $d(v)$ is decreased, $\pi(v)$ is set, and when $\pi(v)$ is set, $d(v)$ is decreased.

(b) Consider the moment of time when $\pi(v)$ was set most recently. At this moment we had $d(v) = d(u) + c(u, v)$, $d(v)$ has not changed since then, and $d(u)$ can only have decreased.

(c) Consider any node u, $u \neq s$, with $(u, v) \in P$ for some v. Then $\pi(v) = (u, v)$ and hence $d(v) < \infty$ by part (a). Then $d(u) + c(u, v) \leq d(v)$ by part (b) and hence $d(u) < \infty$. Thus, $\pi(u) \neq nil$ by part (a). We conclude that any node u, $u \neq s$, with an outgoing P-edge has also an incoming P-edge. Thus s is the only node which may have outgoing P-edges but no incoming P-edge.

(d) Let $[e_0, \ldots, e_{k-1}]$ with $e_i = (v_i, v_{i+1})$ and $v_0 = v_k$ be a P-cycle. We may assume w.l.o.g. that $\pi(v_k) = e_{k-1}$ is the edge in the cycle that was added to P last. Just prior to the addition of e_{k-1} we have

$$d(v_{i+1}) \geq d(v_i) + c(e_i) \text{ for all } i, 0 \leq i \leq k-2$$

by part (b) and

$$d(v_k) > d(v_{k-1}) + c(e_{k-1}).$$

Summation yields

$$\sum_{0 \leq i < k} d(v_{i+1}) > \sum_{0 \leq i < k} (d(v_i) + c(e_i))$$

and hence (since $v_k = v_0$ and thus $d(v_k) = d(v_0)$)

$$\sum_{0 \leq i < k} c(e_i) < 0.$$

(e) Any node v with $\pi(v) \neq nil$ has $d(v) < \infty$ and is hence reachable from s in G. Any P-cycle has negative cost. Thus $\mu(v) = -\infty$ for any node v lying on a P-cycle or being reachable from a P-cycle.

(f) Assume $V^f \neq \emptyset$ and consider any node $v \in V^f$ with $d(v) = \mu(v)$. For $v = s$ there is nothing to show. For $v \neq s$, $d(v) = \mu(v) < \infty$ implies $\pi(v) \neq nil$. From (c) and (e) we conclude that v is reachable from s by a P-path $p = [e_0, \ldots, e_{k-1}]$ with $e_i = (v_i, v_{i+1})$, $v_0 = s$, and $v_k = v$. From (b) we conclude

$$d(v_{i+1}) \geq d(v_i) + c(e_i) \text{ for } i, 0 \leq i < k$$

and hence

$$d(v_k) \geq d(v_0) + c(p) = c(p),$$

where the last equality follows from $v_0 = s \in V^f$ and hence $d(s) = \mu(s) = 0$ by (1) and (2). Thus, $c(p) \le d(v) = \mu(v)$ and we must have equality since no path from s to v can be shorter than $\mu(v)$.

(g) This follows immediately from part (f). □

There are two major problems with the generic algorithm:

- In the presence of negative cycles it will never terminate (since the d-values are always the length of some path and hence cannot reach $-\infty$).
- Even in the absence of negative cycles the running time can be exponential, see [Meh84, page40] for an example.

We address the second problem in the remainder of this section and deal with the first problem in Section 7.5.7. When we decrease the distance label $d(v)$ of a node v in the generic algorithm this may create additional violations of (3), namely for the edges out of v. This suggests maintaining a set U of nodes with

$$U \supseteq \{u\ ;\ d(u) < \infty \text{ and } \exists (u, v) \in E \text{ with } d(u) + c(u, v) < d(v)\}$$

and to rewrite the algorithm[12] as:

```
d(s) = 0; d(v) = ∞ for v ≠ s;
U = {s};
while  U ≠ ∅
{ select u ∈ U and remove it;
  forall  edges e = (u, v)
  { if  d(u) + c(e) < d(v)
    { add v to U;
      d(v) = d(u) + c(e);
      π(v) = e;
    }
  }
}
```

We are left with the decision of which node u to select from U. There is always an optimal choice.

Lemma 9 *(Existence of optimal choice)*

(a) *As long as $d(v) > \mu(v)$ for some $v \in V^f$: for any $v \in V^f$ with $d(v) > \mu(v)$ there is a $u \in U$ with $d(u) = \mu(u)$ and lying on a shortest path from s to v.*

(b) *When a node u is removed from U with $d(u) = \mu(u)$ then it is never added to U again.*

[12] We reuse the name generic shortest-path algorithm for the modified version of the algorithm.

Proof (a) Let $[s = v_0, v_1, \ldots, v_k = v]$ be a shortest path from s to v. Then $\mu(s) = 0 = d(s)$ and $d(v_k) > \mu(v_k)$. Let i be minimal such that $d(v_i) > \mu(v_i)$. Then $i > 0$, $d(v_{i-1}) = \mu(v_{i-1})$ and

$$d(v_i) > \mu(v_i) = \mu(v_{i-1}) + c(v_{i-1}, v_i) = d(v_{i-1}) + c(v_{i-1}, v_i).$$

Thus, $v_{i-1} \in U$.

(b) We have $d(u) \geq \mu(u)$ always. Also, when u is added to U then $d(u)$ is decreased. Thus, if a node u is removed from U with $d(u) = \mu(u)$ it will never be added to U at a later time. □

There are two important special cases of the single-source problem where the existence claim of an optimal choice can be made algorithmic. Both cases deal with graphs where the structure of the graphs excludes negative cycles: graphs with non-negative edge costs and acyclic graphs.

Lemma 10 *(Algorithmic optimal choice)*

(a) *If $c(e) \geq 0$ for all $e \in E$ then $d(u) = \mu(u)$ for the node $u \in U$ with minimal $d(u)$.*

(b) *If G is acyclic and $u_0, u_1, \ldots, u_{n-1}$ is a topological order of the nodes of G, i.e., if $(u_i, u_j) \in E$ then $i < j$, then $d(u) = \mu(u)$ for the node $u = u_i \in U$ with i minimal.*

Proof Assume $d(u) > \mu(u)$ for the node chosen in either part (a) or (b). By the preceding lemma there is a node $z \in U$ lying on a shortest path from s to u with $d(z) = \mu(z)$. We now distinguish cases.

In part (a) we have $\mu(z) \leq \mu(u)$ since all edge costs are assumed to be non-negative. Thus, $d(z) < d(u)$, contradicting the choice of u.

In part (b) we have $z = u_j$ for some $j < i$, contradicting the choice of u. □

Part (a) of the lemma above is the basis of Dijkstra's algorithm, see Section 6.6, and part (b) is the basis of a linear time algorithm for acyclic graphs, which we will discuss in the next section.

In our shortest-path programs we use a *node_array*<*NT*> *dist* to represent the function d and a *node_array*<*edge*> *pred* for the function π. Since most number types have no representation of $+\infty$ we will not be able to maintain equality between d and *dist*. We exploit the fact that the equivalence

$$d(v) = +\infty \text{ iff } v \neq s \text{ and } \pi(v) = nil$$

holds in the generic algorithm and use it for the representation of $+\infty$. We maintain the following relationship between (d, π) and (*dist*, *pred*): for all nodes v:

- *pred*$[v] = \pi(v)$ and
- *dist*$[v] = d(v)$, if $d(v) < \infty$, and *dist*$[v]$ arbitrary, if $d(v) = +\infty$.

With this representation a comparison $d < d(v)$ with $d \in \mathbb{R}$ can be realized as:

```
(pred[v] == nil && v != s) || d < dist[v]
```

We remark that the alternative

```
(v != s && pred[v] == nil) || d < dist[v]
```

is less efficient. All but one node v is different from s and hence the test `v != s` evaluates to true most of the time; thus the test `pred[v] == nil` will also be performed most of the time in the second line. In the first line, the test `pred[v] == nil` evaluates to true only when the first edge into v is considered (since $d(v) < +\infty$ afterwards) and hence evaluates to false in the majority of the cases (at least if the average indegree is larger than two). Thus the test `v != s` will not be made in the majority of the cases.

The general rule is that in a conjunction of tests one should start with the test that evaluates to false most often and that in a disjunction of tests one should start with the test that evaluates to true most often. Please, do not use this rule blindly since interchanging the order of tests may change the semantics (since C++ evaluates a test from left to right and terminates the evaluation once the value of the test is known). In the example above, it would be unwise (why?) to change the expression into

```
d < dist[v] || (pred[v] == nil && v != s)
```

7.5.4 *Acyclic Graphs*

We show how topological sorting can be used to solve the single-source shortest-path problem in acyclic graphs in linear time $O(n + m)$. Let G be an acyclic graph and assume that $v_1, v_2, \ldots, v_n$ is an ordering of the nodes such that $(v_i, v_j) \in E$ implies $i \leq j$. Such an ordering is easy to compute.

$\langle$*acyclic graphs: establish topological order*$\rangle \equiv$

```
node_array<int> top_ord(G);
TOPSORT(G,top_ord);  // top_ord is now a topological ordering of G
int n = G.number_of_nodes();
array<node> v(1,n);
node w;
forall_nodes(w,G) v[top_ord[w]] = w;  // top_ord[v[i]] == i for all i
```

The call TOPSORT(...) numbers the nodes of G with the integers 1 to n such that all edges go from lower numbered to higher numbered nodes. In the forall_node-loop we store the node with number i in $v[i]$.

It is now easy to implement the generic single-source algorithm. Let $k = top\_ord[s]$. Nodes v_i with $i < k$ are not reachable from s. We step through the nodes in the order $v_k, v_{k+1}, \ldots$ and maintain the set U implicitly. Assume we have reached node i. Then U consists of all nodes v_j with $j \geq i$ and $dist(v_j) < +\infty$. For $j > k$ the latter condition is equivalent to $pred[v_j] \neq nil$. If $v[i]$ is equal to s or has a defined predecessor edge we propagate $dist[v[i]]$ over all edges out of $v[i]$ and proceed to the next node.

⟨*acyclic_sp.t*⟩+≡

```
template <class NT>
void ACYCLIC_SHORTEST_PATH_T(const graph& G, node s,
                             const edge_array<NT>& c,
                             node_array<NT>& dist,
                             node_array<edge>& pred)
{
  ⟨acyclic graphs: establish topological order⟩
  forall_nodes(w,G) pred[w] = nil;
  dist[s] = 0;
  for(int i = top_ord[s]; i <= n; i++)
  { node u = v[i];
    if ( pred[u] == nil && u != s ) continue;
    edge e;
    NT du = dist[u];
    forall_adj_edges(e,u)
    { node w = G.target(e);
      if ( pred[w] == nil || du + c[e] < dist[w])
      { pred[w] = e;
        dist[w] = du + c[e];
      }
    }
  }
}
```

The correctness follows immediately from the remarks preceding the program and Lemma 10. The running time is $O(n+m)$ since each node and each edge is considered at most once.

7.5.5 *Non-Negative Edge Costs*

Dijkstra's algorithm for the shortest-path problem with non-negative edge costs was already treated in Section 6.6.

7.5.6 *The Single-Source Single-Sink Problem*

The single-source single-sink shortest-path problem is probably the most natural shortest-path problem. The goal is to find a shortest path from a given source node s to a given sink node t.

We describe the so-called *bidirectional search algorithm* (an alternative approach is discussed in the exercises). The algorithm assumes that edge costs are non-negative. The worst case running time of the algorithm is $O(m + m\log n)$; the observed running time is frequently much better.

The bidirectional search algorithm runs two instances of Dijkstra's algorithm (see Section 6.6) concurrently, one to find shortest-path distances from s and one to find shortest-path distances to t. The first instance is simply Dijkstra's algorithm and the second instance

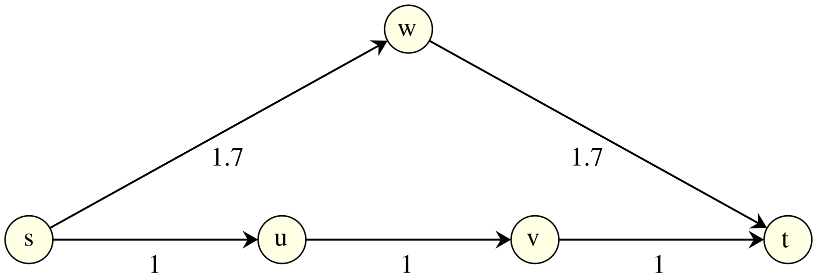

Figure 7.12 Termination of the bidirectional shortest-path algorithm: In our implementation we alternately add nodes to K_s and K_t. In the example we add s to K_s, t to K_t, u to K_s, v to K_t, w to K_s, w to K_t, and v to K_s. The algorithm terminates when v is added to K_s. It does not terminate when w is added to K_t, although $w \in K_s \cap K_t$ at this point of time. Observe that $d_t(v) = 1$ after adding t to K_t and $d_s(v) = 2$ after adding u to K_s. Thus $D = 3$ after adding u to K_s and hence w does not realize D when it is added to $K_s \cap K_t$.

is a symmetric version of Dijkstra's algorithm, where the search starts at t and shortest-path distances are propagated across the edges *into* a node instead of the edges out of a node.

We use $d_s(v)$ to denote the tentative distance from s to v and $d_t(v)$ to denote the tentative distance from v to t. Initially, $d_s(s) = d_t(t) = 0$, $d_s(v) = \infty$ for $v \neq s$, and $d_t(v) = \infty$ for $v \neq t$. The algorithm maintains

$$D = \min_v (d_s(v) + d_t(v))$$

which is the shortest known length of a path from s to t.

Let K_s and K_t be the set of nodes that were removed from the priority queue in the shortest-path calculations from s and t, respectively. We know from Section 7.5.3 that

$$\begin{aligned}
d_s(v) &= \mu(s, v) \text{ for } v \in K_s \\
d_s(v) &= \min\{\mu(s, u) + c(u, v); u \in K_s\} \text{ for } v \notin K_s \\
d_t(v) &= \mu(v, t) \text{ for } v \in K_t \\
d_t(v) &= \min\{c(v, u) + \mu(u, t); u \in K_t\} \text{ for } v \notin K_t
\end{aligned}$$

The bidirectional algorithm terminates when D is realized by a node in $K_s \cap K_t$ or when both queues become empty. In the former case D is the shortest-path distance from s to t, and in the latter case there is no path from s to t. Figure 7.12 illustrates the termination condition.

Theorem 1 *The bidirectional search algorithm is correct.*

Proof If there is no path from s to t then there is never a node in $K_s \cap K_t$ and hence the algorithm terminates when both queues become empty. Thus the algorithm is correct if there is no path from s to t.

So let us assume that there is a path from s to t. Let $p = [s = v_0, v_1, \ldots, v_{k-1}, v_k = t]$ be a shortest path from s to t.

We argue first that the event that D is realized by a node in $K_s \cap K_t$ will occur. This is easy to see. Observe first that all nodes on p are reachable from s as well as t. When a node v_h on p is added to $K_s \cap K_t$, we have

$$\mu(s,t) \le D \le d_s(v_h) + d_t(v_h) = \mu(s, v_h) + \mu(v_h, t) = \mu(s,t),$$

and hence the event that D is realized by a node in $K_s \cap K_t$ will occur at the latest when a node on p is added to $K_s \cap K_t$.

It remains to show that $D = c(p)$ when the event actually occurs. Assume otherwise, i.e., $c(p) < D$ when the algorithm terminates. Then there is no node of p in $K_s \cap K_t$ at the time of termination.

Consider the time of termination, let $w \in K_s \cap K_t$ be the node with $D = d_s(w) + d_t(w)$, let i be minimal with $v_{i+1} \notin K_s$, and let j be maximal with $v_{j-1} \notin K_t$. Both indices exist since $v_0 = s$ is the first node to be added to K_s and $v_k = t$ is the first node to be added to K_t. We have $i < j$ by our assumption that no node of p is added to $K_s \cap K_t$ and hence $d_s(w) \le d_s(v_{i+1})$, since $w \in K_s$ and $v_{j-1} \notin K_s$, and $d_t(w) \le d_t(v_{j-1})$, since $w \in K_t$ and $v_{j-1} \notin K_t$. If $i + 1 \le j - 1$, we have

$$c(p) \ge \mu(s, v_{i+1}) + \mu(v_{j-1}, t) = d_s(v_{i+1}) + d_t(v_{j-1}) \ge d_s(w) + d_t(w) = D,$$

and if $i + 1 = j$, we have with $v = v_{i+1} = v_j$

$$c(p) = \mu(s,v) + \mu(v,t) = d_s(v) + d_t(v) \ge D.$$

□

We turn to the implementation. We distinguish the two versions of Dijkstra's algorithm by indices 0 and 1 and provide two copies of the required data structures in arrays with index set {0, 1}.

⟨*single sink: data structures*⟩≡

```
array<node> terminal(2);
terminal[0] = s; terminal[1] = t;
array<node_pq<NT>* >  PQ(2);
PQ[0] = new node_pq<NT>(G);
PQ[1] = new node_pq<NT>(G);
PQ[0]->insert(terminal[0],0);
PQ[1]->insert(terminal[1],0);
array<node_array<NT> > dist(2);
dist[0] = dist[1] = node_array<NT>(G);
dist[0][s] = dist[1][t] = 0;
array<node_array<edge> > Pred(2);
Pred[0] = Pred[1] = node_array<edge>(G,nil);
bool D_equals_infinity = (s != t? true : false);
NT D = 0;
```

We store the tentative distances $d_s(v)$ and $d_t(v)$ in *dist*[0][v] and *dist*[1][v], respectively, we use *PQ*[0] and *PQ*[1] as the priority queue in the search from s and t, respectively, we use *Pred*[0][v] to record the edge into v that defines $d_s(v)$, and we use *Pred*[1][v] to record the edge out of v that defines $d_t(v)$.

We initialize D to infinity if $s \neq t$, and to zero otherwise. Since we cannot assume that the number type *NT* provides the value $+\infty$ we use a boolean flag to indicate this special value.

A remark is in order about the declarations above. We declared *dist* as an array of node arrays and *PQ* as an array of pointers to priority queues. Why did we make this distinction? In order to declare an *array*<T> for some type T, T must provide a default constructor, a copy constructor, and some other operations, e.g., the input and output operators $\ll$ and $\gg$, see Section 2.8. Node arrays provide all required functions except for the input and output operators and those are easily defined in the current file, since the missing functions are non-member functions of node arrays. The situation is different for node priority queues; they define only a few of the required functions and, in particular, a member function is missing. We cannot add the member function in this file. Moreover, in the case of *dist* it is more important to have an array of node arrays instead of an array of pointers to node arrays, since having an array of pointers to node arrays would force us to write either `(*dist[i])[v]` or `dist[i]->operator[](v)` instead of `dist[i][v]`.

⟨*dijkstra_single_sink.t*⟩≡

```
template <class T>
ostream& operator<<(ostream& o,const node_array<T>&) { return o; }
template <class T>
istream& operator>>(istream& i,node_array<T>&)       { return i; }
```

The structure of the single-source single-sink program is as described above. We run both instances of Dijkstra's algorithm concurrently, and terminate when either both queues become empty or when we encounter a node $u \in K_s \cap K_t$ with $D = d_s(u) + d_t(u)$. In the former case there is no path from s to t. According to our output convention for the single-source single-sink problem this fact is recorded by having *pred*[t] $=$ *nil* in the return values.

⟨*dijkstra_single_sink.t*⟩+≡

```
template<class NT>
NT DIJKSTRA_T(const graph& G, node s, node t,
              const edge_array<NT>& cost, node_array<edge>& pred)
{
  ⟨single sink: data structures⟩
  while ( !PQ[0]->empty() || !PQ[1]->empty() )
  { for (int i = 0; i < 2; i++)
    { if ( PQ[i]->empty() ) continue;
      node u = PQ[i]->del_min();
      ⟨return if u is in Ks and Kt and D = d_s(u) + d_t(u)⟩
      ⟨relax edges out of u, if i = 0, or into u, if i = 1⟩
```

```
    }
  }
  pred[t] = nil; // no path from s to t
  return D;
}
```

The relaxation of edges is copied from Section 6.6 with two small modifications.

In the search for shortest paths from s we iterate over the edges out of u, and in the search for shortest paths to t we iterate over the edges into u.

Whenever the dist-value of a node is improved we check whether this leads to an improvement of D.

⟨*relax edges out of u, if i = 0, or into u, if i = 1*⟩≡

```
for ( edge e = (i == 0? G.first_adj_edge(u): G.first_in_edge(u));
      e != nil;
      e = (i == 0? G.adj_succ(e): G.in_succ(e)) )
{ node v = (i == 0? G.target(e) : G.source(e) );
  NT c = dist[i][u] + cost[e];
  if ( Pred[i][v] == nil && v != terminal[i] )
     PQ[i]->insert(v,c); // first path to v
  else if (c < dist[i][v]) PQ[i]->decrease_p(v,c); // better path
       else continue;
  dist[i][v] = c;
  Pred[i][v] = e;
  if ( ( v == terminal[1-i] || Pred[1-i][v] != nil )
       // dist[1-i][v] is defined iff true
     && ( D_equals_infinity || dist[0][v] + dist[1][v] < D ))
  { D_equals_infinity = false;
    D = dist[0][v] + dist[1][v];
  }
}
```

How can we check whether $u \in K_s \cap K_t$? Assume w.l.o.g. that $i = 0$. Then $u \in K_s$ since we have just removed it from $PQ[0]$. Also, we have $u \in K_t$ if u has been in $PQ[1]$, but is not there anymore. u has been or still is in $PQ[1]$ if either $u = t$ or $Pred[1][u]$ is defined, and u is not in $PQ[1]$ if $PQ[1] \rightarrow member(u)$ returns false.

If $u \in K_s \cap K_t$ and $D = d_s(u) + d_t(u)$ we terminate the computation, record the path in the predecessor array, and return D as the length of the shortest path from s to t. In order to record the path in the *pred*-array we trace the two "half paths" from u to s and from u to t, respectively. When tracing the latter path we observe that $Pred[1]$ stores out-edges and not in-edges.

⟨*return if u is in Ks and Kt and D = d_s(u) + d_t(u)*⟩≡

```
if ( (u == terminal[1-i] || Pred[1-i][u] != nil) &&
     !PQ[1-i]->member(u) && dist[0][u] + dist[1][u] == D )
{ // have found shortest path from s to t.
  // trace path from u to s
  node z = u;
```

n	m	Single-sink	Dijkstra
10000	500000	0.118	0.736

Table 7.1 A comparison of the running time of the single-sink algorithm with the running time of the standard version of Dijkstra's algorithm. The standard version computes the distance from the source to all other vertices and then extracts the distance value of the sink.

```
  while ( z != s ) z = G.source(pred[z] = Pred[0][z]);
  // trace path from u to t
  z = u;
  edge e;
  while ( (e = Pred[1][z] ) != nil) { pred[z = G.target(e)] = e; }
  return D;
}
```

Table 7.1 compares the running times of the single-source single-sink algorithm presented in this section and the standard version of Dijkstra's algorithm.

7.5.7 *General Networks: The Bellman–Ford Algorithm*

We derive and implement a single-source shortest-path algorithm for arbitrary edge costs. The algorithm is due to Bellman [Bel58] and Ford. We will refer to the algorithm as the basic Bellman–Ford algorithm[13]. In Section 7.5.3 we studied a generic shortest-path algorithm. Let us recall what we know:

- The algorithm maintains a set U containing all nodes u for which there is an edge (u, v) with $d(u) + c(u, v) < d(v)$. U may also contain other nodes.
- In each iteration the algorithm selects some node in U and relaxes all edges out of it.
- As long as $d(v) > \mu(v)$ for some $v \in V^f$, there is a node $u \in U$ with $d(u) = \mu(u)$ (Lemma 9). We use the phrase that not all finite distance values are determined to mean that $d(v) > \mu(v)$ for some $v \in V^f$.
- When a node u is removed from U with $d(u) = \mu(u)$ it will never be added to U again at a later stage.
- Let $P = \{e\, ; e = \pi(v) \in E$ for some $v \in V\}$. All P-cycles are negative and if $d(v) = \mu(v)$ for all $v \in V^f$ then P defines a shortest-path tree on V^f.

What is a good strategy for selecting from U? We know that U contains a perfect choice (at least as long as not all finite distance values are determined), but we do not know which node in U is the perfect choice. In order to play it safe we should therefore not discriminate against any node in U. A way to achieve fairness is to organize the computation in phases.

[13] We will study a refined version in the next section.

Let U_i be the set U at the beginning of phase i, $i \geq 0$; U_0 is equal to $\{s\}$. In phase i we remove all vertices in U_i from U. Newly added vertices are inserted into U_{i+1}. In this way we guarantee that at least one finite distance value is determined in each phase (if there is one that is still to be determined) and hence all finite distance values are determined after at most n phases.

In the program below we realize the set U by a queue Q. During phase i all nodes in U_i are at the front of the queue and all nodes in U_{i+1} are at the rear of the queue. We separate U_i and U_{i+1} by the marker *nil*. We count the number of phases in *phase_count*. Whenever the marker appears at the front of Q we increment *phase_count*. In order to avoid putting nodes several times into Q we keep a *node_array*<*bool*> *in_Q* with *in_Q*$[v] = $ *true* iff $v \in Q$.

We terminate the algorithm when Q becomes empty or when phase n is reached. In the former case we have $d(v) = \mu(v)$ for all v and in the latter case we have $d(v) = \mu(v)$ for all $v \in V^+ \cup V^f$. We will deal with the nodes in V^- in a postprocessing step.

⟨*bellman_ford_basic.t*⟩≡

```
#include <LEDA/graph_alg.h>
#include <LEDA/b_queue.h>
⟨BF: helper⟩
template <class NT>
bool BELLMAN_FORD_B_T(const graph& G, node s, const edge_array<NT>& c,
                      node_array<NT>& dist, node_array<edge>& pred )
{ int n = G.number_of_nodes();
  int phase_count = 0;
  b_queue<node> Q(n+1);
  node_array<bool> in_Q(G,false);
  node u,v;
  edge e;
  forall_nodes(v,G) pred[v] = nil;
  dist[s] = 0;
  Q.append(s); in_Q[s] = true;
  Q.append((node) nil); // end marker
  while( phase_count < n )
  { u = Q.pop();
    if ( u == nil)
    { phase_count++;
      if ( Q.empty() ) return true;
      Q.append((node) nil);
      continue;
    }
    else in_Q[u] = false;
    NT du = dist[u];
    forall_adj_edges(e,u)
    { v = G.opposite(u,e);  // makes it also work for ugraphs
      NT d = du + c[e];
      if ( (pred[v] == nil && v != s) || d < dist[v] )
      { dist[v] = d; pred[v] = e;
        if ( !in_Q[v] ) { Q.append(v); in_Q[v] = true; }
```

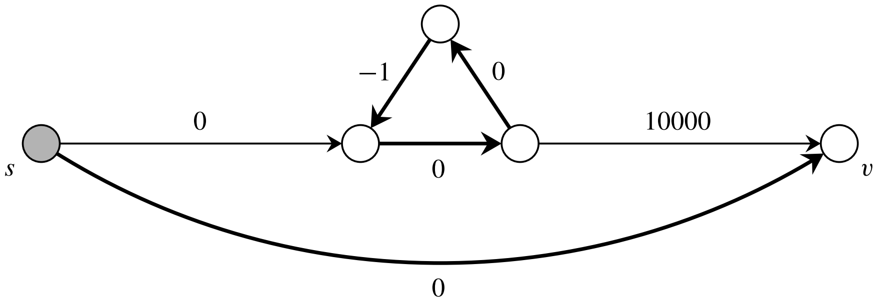

Figure 7.13 The situation at the beginning of phase $n = |V| = 5$. Note that v belongs to V^-, that P contains a negative cycle from which v is reachable in G, but that v is not reachable from this cycle by a P-path. Running the algorithm for another 10000 phases will establish the output convention; the quantity 10000 reflects the fact that traversing the cycle 10000 times creates a path of cost -10000.

```
      }
    }
  }
  ⟨BF: postprocessing⟩
  return false;
}
```

We turn to the postprocessing step required when U is non-empty after n phases. Figure 7.13 shows that our output convention is not automatically satisfied. As the figure shows there may be nodes in V^- that are not reachable yet from a P-cycle by a P-path.

How can we establish our output convention that all nodes in V^- are reachable from a P-cycle by a P-path? We could run the algorithm for more phases until a path containing a negative cycle has been discovered for all nodes in V^-. This may take very long as Figure 7.13 shows. We need a better method. In the following lemma we show that Figure 7.14 describes the situation at the beginning of phase n. The argument is with respect to the generic algorithm with the selection rule of the Bellman–Ford algorithm.

For an integer k, $k \geq 0$, let

$$\mu_k(v) = \min\{c(p)\ ;\ p \text{ is a path from } s \text{ to } v \text{ consisting of at most } k \text{ edges}\}.$$

Lemma 11 *After n phases:*

(a) *$d(v) \leq \mu_n(v)$ and if $v \in U$ then $d(v) < \mu_{n-1}(v)$.*
(b) *$s \in V^f$ iff $\pi(s) = nil$.*
(c) *Every $u \in U$ lies either on a P-cycle or on a P-path emanating from a P-cycle.*
(d) *Every $v \in V^-$ is reachable in G from a $u \in U$.*
(e) *If $\pi(s) \neq nil$ then the output convention is already satisfied.*

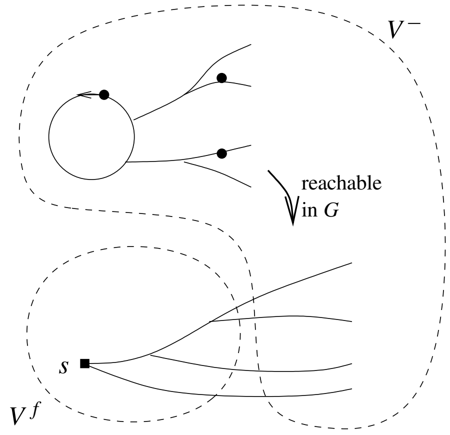

Figure 7.14 The situation at the beginning of phase n: Some nodes in V^- are still reachable from s by a P-path and some are already contained in a P-cycle or lie on a P-path emanating from a P-cycle. *All* nodes in U (nodes in U are shown as solid circles) belong to the latter category by part (d) of Lemma 11. All nodes in V^- are reachable in G from a node in U by part(e) of Lemma 11.

Proof (a) Let $p = [s = v_0, v_1, \ldots, v_k = v]$ be any path starting in s. Then $d(v_i) \le \sum_{0<j\le i} c(v_{j-1}, v_j)$ at the beginning of phase i and hence $d(v) \le \mu_{n-1}(v)$ at the beginning of phase $n-1$ and $d(v) \le \mu_n(v)$ at the beginning of phase n. If v is added to U in phase $n-1$, $d(v)$ is decreased and hence $d(v) < \mu_{n-1}(v)$ at the beginning of phase n.

(b) If $s \in V^f$ then $d(s) = 0$ and hence $\pi(s) = nil$. If $s \notin V^f$ then there is a negative cycle passing through s and hence $\mu_n(s) < 0$. Thus, $d(s) < 0$ and hence $\pi(s) \ne nil$.

(c) If $\pi(s) \ne nil$ part (c) follows from Lemma 8, part (c). So assume $s \in V^f$ and assume that there is a $u \in U$ that is reachable from s by a P-path, say p. Then $d(u) \ge c(p) \ge \mu_{n-1}(u)$, a contradiction to part (a).

(d) Let $v \in V^-$ be arbitrary. Since $\mu(v) = -\infty$ there must be a path p from s to v with $c(p) < d(v)$. Let p_i be the path consisting of the first i edges of p and let v_i be the target node of p_i. Let k be minimal such that $c(p_k) < d(v_k)$. Then $k > 0$ since $c(p_0) = 0$ and $d(v_0) = d(s) \le 0$ and hence $c(p_{k-1}) \ge d(v_{k-1})$. Thus,

$$d(v_k) > c(p_k) = c(p_{k-1}) + c(v_{k-1}, v_k) \ge d(v_{k-1}) + c(v_{k-1}, v_k)$$

and hence $v_{k-1} \in U$.

(e) If $\pi(s) \ne nil$ then part (c) of Lemma 8 tells us that every node reachable from s lies either on a P-cycle or a P-path emanating from a P-cycle. □

Parts (a), (d), and (e) of the lemma above are the key for the postprocessing step. If $\pi(s) \ne nil$ we are done. So assume $\pi(s) = nil$, i.e., $V^f \ne \emptyset$, and let R be the set of nodes that are reachable from s by a P-path. Then $R \supseteq V^f$ but this inclusion may be proper, see Figure 7.14. All nodes in R that are reachable from a node $u \in U$ belong to V^- and hence their π-values have to be changed. We can do so by performing a depth-first search from each node $u \in U$. Whenever a node in R is reached we change its π-value to the edge which led to the node. In this way we connect all nodes in $R \cap V^-$ to the nodes in U and hence, by part (c), make them reachable from P-cycles by P-paths.

How can we determine the nodes in R? We simply perform a depth-first search from s on the subgraph defined by P. This can be done by hiding all edges not in P, performing a depth-first search, and restoring (= unhiding) all edges in P. In the program below the nodes in R are labeled true in the node array *in_R*.

In the program chunk below the cast ((*graph*∗) &G) turns G from a const-object to a non-const-object. The cast is required since *hide_edge* and *restore_all_edges* modify the graph and the cast is safe since *restore_all_edges* restores the original situation.

⟨*BF: postprocessing*⟩≡
```
if (pred[s] != nil) return false;
node_array<bool> in_R(G,false);
forall_edges(e,G)
  if (e != pred[G.target(e)]) ((graph*) &G)->hide_edge(e);
DFS(G,s,in_R); // sets in_R[v] = true for v in R
((graph*) &G)->restore_all_edges();
node_array<bool> reached_from_node_in_U(G,false);
forall_nodes(v,G)
  if (in_Q[v] && !reached_from_node_in_U[v])
    Update_pred(G,v,in_R,reached_from_node_in_U,pred);
```

where

⟨*BF: helper*⟩≡
```
inline void Update_pred(const graph& G, node v,
                 const node_array<bool>& in_R,
                 node_array<bool>& reached_from_node_in_U,
                 node_array<edge>& pred)
{ reached_from_node_in_U[v] = true;
  edge e;
  forall_adj_edges(e,v)
    { node w = G.target(e);
      if ( !reached_from_node_in_U[w] )
        { if ( in_R[w] ) pred[w] = e;
            Update_pred(G,w,in_R,reached_from_node_in_U,pred);
        }
    }
}
```

The running time of the Bellman–Ford algorithm is $O(nm)$. This can be seen as follows. There are at most n phases and the running time of each phase is proportional to the sum of the outdegrees of the nodes removed from Q in the phase. This implies that the cost of any one phase is $O(m)$ and the bound follows.

A somewhat tighter analysis is as follows. Let D be the maximal number of edges on any shortest path. We have $D < n$ if V^- is empty and $D = \infty$ otherwise. Then Q is empty after phase D and hence the running time is $O(\min(D, n) \cdot m)$.

For many graphs D is much smaller than n. Examples are complete graphs with edge costs chosen uniformly at random from $[0 .. 1]$. In this case $D = O(\log^2 n)$ with high probability [CFMP97]; the expected running time is therefore $O(n^2 \log^2 n)$ for complete graphs with random edge costs. More generally, it is an experimental fact that the Bellman–Ford algorithm is efficient for almost any kind of random graph.

However, there are also graphs where the worst case running time is actually achieved. We give one example in the next section and one now.

A first example are graphs with negative cycles. If V^- is non-empty then the algorithm always uses n phases and a high running time results. We will show in the next but one section how negative cycles can frequently be recognized earlier.

7.5.8 *A Difficult Graph*

The goal of this section is to construct a graph with non-negative edge costs that forces the algorithm of the preceding section into its worst case running time.

The running time analysis given above tells us that a running time of $\Omega(nm)$ results if a fixed fraction of the nodes is removed and added to the queue in each iteration. The Bellman–Ford algorithm uses a breadth-first scanning strategy, i.e., essentially explores paths in the order of their number of edges. Thus if we ensure that paths consisting of more edges have smaller cost we will ensure that every node is added to the queue many times.

We will define the graph in two steps. In the first step we will allow edges of negative cost and in the second step we will remove them. Figure 7.15 shows our worst case example. The graph has nodes $0, \ldots, L-1, L, \ldots, L+K-2$ where $L = 2^l$ is a power of two. We will fix K and L later.

On nodes $L-1$ to $L+K-2$ we have the complete graph in which all edge costs are zero. This makes $(K-1)^2$ edges. On the first L nodes we have the edge $(0, L-1)$, the edges $(L - L/2^j, L - L/2^{j+1})$ and $(L - L/2^{j+1}, L-1)$ for all j, $0 \le j < l-1$, and the L edges $(i, i+1)$ for all i, $0 \le i < L-1$. This makes for no more than $2L$ edges.

We claim that for any r, $1 \le r < L$, there is exactly one path from node 0 to node $L-1$ consisting of r edges. This is certainly true for $r = 1$. So assume that $r > 1$. We construct the path as follows. If $r > L/2$ we use $L/2$ edges to go from 0 to $L/2$ and if $r \le L/2$ we use one edge. In either case we are left with the task of constructing a path from $L/2$ to $L-1$ consisting of r' edges, where $1 \le r' < L/2$. This path is constructed by applying the argument recursively.

How do we assign edge costs to the edges (i, j) with $0 \le i < j < L$? We want an assignment which favors paths with more edges. This suggests assigning cost -1 to every edge as this makes sure that the cost of a path consisting of k edges is equal to $-k$. Thus paths with more edges are shorter than paths with fewer edges. We said at the beginning that we will construct a graph with non-negative edge costs and now we have set the cost

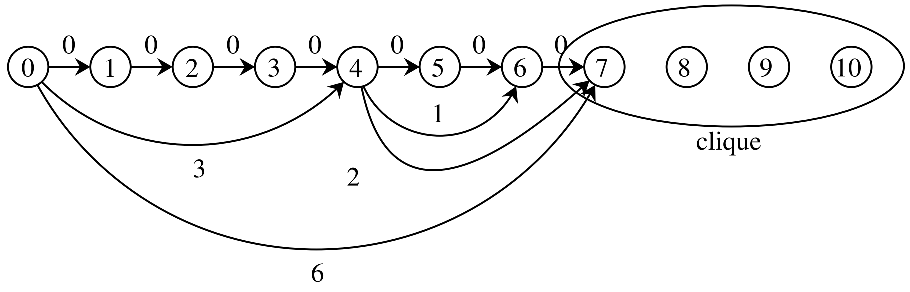

Figure 7.15 The graph generated by *BF_GEN* for $L = 8$ and $K = 4$. The K nodes labeled $L-1$ to $L+K-2$ form a complete directed graph in which all edge costs are zero. The edges in this clique are not shown.

of some edges to -1. This is easily corrected. We set the cost of edge (i, j) to $j-i-1$. Then all edges have non-negative cost and the cost of a path from 0 to $L-1$ consisting of k edges has cost $L-1-k$. Thus we are again favoring paths with more edges over paths with fewer edges.

The total number of edges in our graph is certainly less than $2L + K^2$ and the number of nodes is $L+K-1$. With $K = \lfloor\sqrt{m/2}\rfloor$ and L the largest power of two no larger than $n/2$, we get a graph with at most $n + m/2$ edges and $n/2 + \sqrt{m/2}$ nodes. This is less than m and n, respectively, if $m \geq 2n$ and $m \leq n^2/2$.

The following procedure *BF_GEN* realizes the construction just outlined. For the edge costs there is the choice between non-negative and arbitrary edge costs. If $m \geq 2n$ and $m \leq n^2/2$ then the constructed graph has at most n nodes and at most m edges.

⟨*_BF_GEN.c*⟩≡

```
#include <LEDA/array.h>
#include <LEDA/graph_alg.h>
void BF_GEN(GRAPH<int,int>& G, int n, int m,
                    bool non_negative)
{ G.clear();
  int K = 1; while ( (K+1)*(K+1) <= m/2 ) K++;
  int l = 0; int L = 1;
  while ( 2*L <= n/2 ) {l++; L = 2*L; }
  array<node> V(n);
  int i, j;
  for (i = 0; i < n; i++) V[i] = G.new_node(i);
  for (i = L - 1; i < L - 1 + K; i++)
    for (j = L - 1; j < L - 1 + K; j++)
      if ( j != i ) G.new_edge(V[i], V[j], 0);
  for (i = 0; i < L - 1; i++) G.new_edge(V[i], V[i+1], 0);
  G.new_edge(V[0],V[L-1],(non_negative? L-1-1  :-1));
  int powj = 1;
  for (j = 0; j < l-1; j++)
  { int x = L - L/powj;
```

```
      int y = L - L/(2*powj);
      G.new_edge(V[x],V[y],  (non_negative? y-x-1 :   -1));
      G.new_edge(V[y],V[L-1],(non_negative? L-1-y-1 : -1));
      powj *= 2;
    }
  }
```

How does our algorithm of the previous section do on the graphs generated by *BF_GEN*? There will be L phases and in each phase the K nodes $L-1, \ldots, L+K-2$ will be removed from the queue and hence K^2 edges will be scanned in each phase. Since $L \geq n/4$ and $K^2 \geq m/4$ the running time is $\Omega(nm)$.

Table 7.2 shows the running times of the basic and the refined version of the Bellman–Ford algorithm (the refined version is the subject of the next section), the time for checking the output, and, if applicable, the running time of Dijkstra's algorithm. We observe that the basic version beats the refined version for random inputs and that both of them are almost competitive with Dijkstra's algorithm for random inputs with non-negative edge costs. The situation changes completely for graphs with negative cycles and graphs generated by *BF_GEN*.

For random graphs with negative cycles the running time of the basic version explodes because it always executes n phases on such graphs. The refined version behaves much better.

For graphs generated by *BF_GEN* the basic version shows the claimed $\Omega(nm)$ behavior. Doubling n (more than) quadruples the running time; the fact that the running time more than quadruples is due to cache effects. Again, the refined version behaves much better. Its running time seems to less than triple if n is doubled. We will explain this effect at the end of Section 7.5.9. Dijkstra's algorithm performs much better than either version of the Bellman–Ford algorithm.

In all cases the time needed to verify the computation is no larger than the time required to compute the result.

There are more shortest-path algorithms than the ones treated in this book, see [AMO93], and some of them have an edge over the algorithms in LEDA in certain situations. The papers [CG96, CGR94, MCN91] contain extensive experimental comparisons of various shortest-path algorithms. The algorithms that we have selected for LEDA are the asymptotically most efficient and also exhibit excellent actual running times.

7.5.9 *A Refined Bellman–Ford Algorithm*

We describe a variant of the Bellman–Ford algorithm due to Tarjan [Tar81]. The worst case running time of the variant is also $O(nm)$. However, the algorithm is frequently much faster than the basic Bellman–Ford algorithm, as Table 7.2 shows, and the algorithm is never much slower. It is available as *BELLMAN_FORD_T*[14].

[14] This is clearly a misnomer. However, we want to keep the name *BELLMAN_FORD_T* for our currently best implementation for the single-source problem with arbitrary edge costs.

Instance	*BF_Basic*	*Bellman_Ford*	*Dijkstra*	*Checking*
n, $n = 10000$	0.3	0.57	0.22	0.31
n, $n = 20000$	0.69	1.36	0.57	0.69
n, $n = 40000$	1.98	3.59	1.47	1.69
c, $n = 10000$	0.3	0.63	—	0.3
c, $n = 20000$	0.81	1.63	—	0.7
c, $n = 40000$	2.02	3.72	—	1.68
r, $n = 2000$	20.2	0.08	—	0.03
r, $n = 4000$	73.15	0.17	—	0.08
r, $n = 8000$	462.5	0.54	—	0.18
g, $n = 4000$	7.52	0.42	0.01001	0.04999
g, $n = 8000$	30.66	1.17	0.04004	0.07996
g, $n = 16000$	131.5	3.24	0.07001	0.19

Table 7.2 Running times of different shortest-path algorithms. We used four different kinds of graphs. Random graphs (generated by *random_graph*(G, n, m)) with random non-negative edge costs in $[0 .. 1000]$, random graphs with arbitrary edge costs but no negative cycles (we chose for each node v a random node potential $pot[v] \in [0 .. 1000]$ and for each edge $e = (v, w)$ a random cost $c[e] \in [0 .. 1000]$ and then set the cost of e to $pot[v] + c[e] - pot[w]$; this generates arbitrary edge costs but no negative cycles as the potentials cancel along any cycle, see Section 7.5.10.), random graphs with random edge costs in $[-100 .. 1000]$, and graphs generated by *BF_GEN*. In the table above the four types of graphs are indicated by the labels n, c, r, and g, respectively. For each type we generated graphs with three different values of n and $m = 8n$. Observe that the graphs in the top half of the table are much larger than the graphs in the lower half of the table. The column *BF_Basic* stands for the basic version of the Bellman–Ford algorithm. You may generate your own version of this table by calling shortest_path_time in the demo-directory.

The variant maintains the shortest-path tree[15] not only implicitly in the form of the *pred*-array but also explicitly. We use T to denote the shortest-path tree. The algorithm uses T to overcome two weaknesses of the basic Bellman–Ford algorithm. Consider the scanning of an edge $e = (v, w)$ and assume that it reduces $dist[w]$ to $dist[v] + c[e]$. In the basic algorithm the only action is to add w to Q (if it is not already there). In the variant we do more:

- The fact that a shorter path to w has been discovered implies that shorter paths exist for all nodes in T_w (= the subtree of T rooted at w). Thus there is no need to propagate the current distance labels of these nodes any further (as smaller distance labels will be

[15] We ignore the possibility of a negative cycle for the moment.

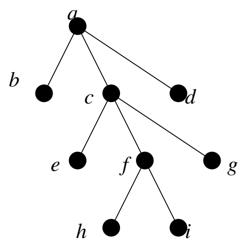

Figure 7.16 The pre-order traversal of the tree shown yields the sequence $a, b, c, e, f, h, i, g, d$.

propagated sometime in the future) and hence all nodes in T_w can be removed from Q and T. Upon removal of T_w from Q and T, w is added to Q and made a child of v in T. This modification introduces a distance related component into the otherwise purely breadth-first scanning strategy of the Bellman–Ford algorithm.

- If w is an ancestor of v or, equivalently, v is a descendant of w then a negative cycle has been detected and all nodes reachable from v can be added to V^-. This modification replaces the indirect way of recognizing negative cycles used in the basic algorithm ("more than n phases") by a direct method.

We come to the details. We use T to denote the current shortest-path tree. It is rooted at s and if w is a child of v in T then $pred[w] = (v, w)$. Conversely, if $pred[w] \neq nil$ then w was already added to T at least once; it may or may not belong to T currently. The tree T is represented by its list of vertices in pre-order traversal, see Figure 7.16, i.e., a single node tree is represented by that node and a tree with root r and subtrees $T_1, \ldots, T_k$ is represented by r, followed by the list for $T_1, \ldots$, followed by the list for T_k. We use a *list<node>* T to represent the shortest-path tree, a *node_array<int> t_degree* to store the degree of each node, and a *node_array<list_item> pos_in_T* to store the position of each node in the list T. For nodes $v \notin T$ we have $t\_degree[v] = 0$ and $pos\_in\_T[v] = nil$ and for nodes $v \in T$ we have $T[pos\_in\_T[v]] = v$.

The queue Q is also realized as a list of nodes. Every node knows its position in Q. We use a *node_array<list_item> pos_in_Q* for that purpose. If a node v belongs to Q then $pos\_in\_Q[v]$ is its position in Q and if a node v does not belong to Q then $pos\_in\_Q[v] = nil$.

We use $w\_item = pos\_in\_T[w]$ to denote the item corresponding to node w in T. We define a procedure *delete_subtree*$(w\_item, \ldots)$ that deletes all nodes in the subtree T_w from T and Q and returns the item following T_w in T. In Figure 7.16 a call *delete_subtree*$(f\_item, \ldots)$ would delete the subtree T_f and return the item corresponding to g.

If w has no children ($t\_degree[w] = 0$), we simply delete w from T and maybe also from Q. If w has children, the idea is to remove the subtrees of the children by recursive calls. The first child is easy to find; it is the node immediately after w in the list T. The second child (if the degree of w is more than one) is the first node after the sublist representing the first subtree of T_w. This is precisely the node returned by the first recursive call of *delete_subtree* and hence a simple loop removes all subtrees of T_w.

The procedure *delete_subtree* uses Q, T, *pos_in_Q*, *pos_in_T* and *t_degree*. We make them parameters. We will initialize them below.

⟨*BF: auxiliary functions*⟩≡

```
inline list_item BF_delete_subtree(list_item w_item, list<node>& Q,
          list<node>& T, node_array<int>& t_degree,
          node_array<list_item>& pos_in_Q,
          node_array<list_item>& pos_in_T)
{ list_item child = T.succ(w_item);
  node w = T[w_item];
  while (t_degree[w] > 0)
  { t_degree[w]--;
    child = BF_delete_subtree(child,Q,T,t_degree,pos_in_Q,pos_in_T);
  }
  pos_in_T[w] = nil;
  T.del_item(w_item);
  if ( pos_in_Q[w] )
  { Q.del_item(pos_in_Q[w]);
    pos_in_Q[w] = nil;
  }
  return child;
}
```

As in the basic algorithm we operate in phases. For the zeroth phase we initialize Q and T with s.

⟨*BF: initialize T, Q, dist, and pred*⟩≡

```
node_array<list_item> pos_in_Q(G,nil);
node_array<int>       t_degree(G,0);
node_array<list_item> pos_in_T(G,nil);
node v;
forall_nodes(v,G) pred[v] = nil;
dist[s] = 0;
list<node> Q;  pos_in_Q[s] = Q.append(s);
list<node> T;  pos_in_T[s] = T.append(s);
```

During the k-th phase, $k \geq 0$, we maintain the following invariants. They refine the invariants of the basic algorithm. We use $\mu_k(v)$ to denote the length of a shortest path from s to v consisting of at most k edges.

(1) For every node v, *dist*$[v]$ is the cost of some path from s to v, and if v belongs to T then *dist*$[v]$ is the cost of the tree path from s to v and *pred*$[v]$ is the tree edge ending in v.

(2) If v has been in T at least once, but is not in T now, then $\mu(v) <$ *dist*$[v]$, i.e, its current distance label is not its true distance label.

(3) Only leaves of T belong to Q, and these leaves have depth k or $k+1$ in T. The nodes of depth k precede the nodes of depth $k+1$ in Q.

(4) The algorithm maintains a *node_array*<*bool*> *in_Vm* such that the following items hold for every node v:

(a) $in\_Vm[v] = true$ implies $v \in V^-$.
(b) If every path defining $\mu_k(v)$ contains a negative cycle then $in\_Vm[v] = true$.
(c) If $in\_Vm[v] = true$ and w is reachable from v in G then $in\_Vm[w] = true$.

(5) If v is a node in $T \setminus Q$ then $dist[v] + c[e] \geq dist[w]$ for all edges $e = (v, w)$ with $in\_Vm[w] = false$, i.e., if v is in T but not in Q then its outgoing edges are relaxed. Observe that $in\_Vm[w] = true$ implies $\mu(w) = -\infty$ and hence may be interpreted as "$dist[w] = -\infty$".

(6) For every node v with $in\_Vm[v] = false$, $dist(v) \leq \mu_k(v)$.

Phase k ends when Q contains no node of depth k anymore[16] and the algorithm terminates when Q is empty.

Let v be the first node in Q and let k be its depth in T. The goal is to remove v from Q without violating the invariants. We explain the required actions first and then give the code. We suggest that the code is read in parallel to the explanation.

We scan all edges $e = (v, w)$ out of v. If $in\_Vm[w] = true$ then there is nothing to do (by invariants (5) and (6)). So assume otherwise. We compare $dist[v] + c[e]$ and $dist[w]$. There are two cases to consider.

If $dist[w] \leq dist[v] + c[e]$ then there is nothing to do, i.e, all invariants hold already. This is obvious if we have inequality or $w \in T$. So assume that we have equality and w does not belong to T. Don't we have to add w to T? No! Observe that $dist[w] = dist[v] + c[e]$ implies $dist[w] < \infty$. Thus w has been in T at least once, and hence (2) implies $\mu(w) < dist[w]$. Thus the invariants also hold in this case.

If $dist[v] + c[e] < dist[w]$ then $\mu(z) < dist[z]$ for all nodes z in T_w. Thus, we may remove w and all its descendants from T and Q, set $dist[w]$ to $dist[v] + c[e]$ and $pred[w]$ to e.

If v was not in T_w and hence v is still in T at this point we make w a child of v and add w to Q. This maintains all invariants. In order to make w a child of v, we simply insert it immediately after v into the list T and increment the degree of v.

If v belonged to T_w then we discovered a negative cycle consisting of the tree path from w to v followed by the edge e. We move all nodes reachable from v in G to V^-.

⟨*bellman_ford.t*⟩≡

```
#include <assert.h>
⟨BF: auxiliary functions⟩
template <class NT>
bool BELLMAN_FORD_T(const graph& G, node s,
                       const edge_array<NT> & c,
                       node_array<NT> & dist,
                       node_array<edge>& pred)
{ ⟨BF: initialize T, Q, dist, and pred⟩
  node_array<bool> in_Vm(G,false); // for V_minus
  bool no_negative_cycle = true;
```

[16] The algorithm does not keep track of node depths and phase numbers; we only use them in the correctness proof.

```
  while (!Q.empty())
  { // select a node v from Q
    node v = Q.pop(); pos_in_Q[v] = nil;
    edge e;
    forall_adj_edges(e,v)
    { node w = G.target(e);
      if ( in_Vm[w] ) continue;
      NT d = dist[v] + c[e];
      if ( ( pred[w] == nil && w != s ) || d < dist[w])
      { dist[w] = d;
        // remove the subtree rooted at w from T and Q
        // if w has a parent, decrease its degree
        if (pos_in_T[w])
        { BF_delete_subtree(pos_in_T[w],Q,T,t_degree,
                                    pos_in_Q,pos_in_T);
          if (pred[w] != nil) t_degree[G.source(pred[w])]--;
        }
        pred[w] = e;
        if (pos_in_T[v] == nil) // v belonged to T_w
        { no_negative_cycle = false;
          ⟨move v and all nodes reachable from it to Vm⟩
        }
        else
        { // make w a child of v and add w to Q
          pos_in_T[w] = T.insert(w,pos_in_T[v],after);
          t_degree[v]++;
          pos_in_Q[w] = Q.append(w);
        }
      }
    }
  }
#ifndef LEDA_CHECKING_OFF
CHECK_SP_T(G,s,c,dist,pred);
#endif
  return no_negative_cycle;
}
```

We still need to complete the case that a negative cycle is detected. When v belonged to T_w we discovered a negative cycle. After setting $pred[w] = e = (v, w)$ this negative cycle is already recorded in the *pred*-array. What remains is to add all nodes that are reachable from v to V^- and to set their *pred*-values accordingly. We want to do so without destroying the negative cycle just found.

This is readily achieved. We first set *in_Vm* to *true* for all nodes on the cycle and then in a second pass over the cycle call $add\_to\_Vm(G, z, \ldots)$ for all nodes z of the cycle. In $add\_to\_Vm(G, z, \ldots)$ we scan all edges out of z. For each edge $e = (z, w)$, where w does not belong to V^- yet, we remove all nodes in T_w from T and Q, we add w to V^-, set $pred[w]$ to e, and make a recursive call.

⟨*move v and all nodes reachable from it to Vm*⟩≡

```
node z = v;
do
{ in_Vm[z] = true;
  z = G.source(pred[z]);
} while (z != v);
do
{ BF_add_to_Vm(G,z,in_Vm,pred,Q,T,t_degree,pos_in_Q,pos_in_T);
  z = G.source(pred[z]);
} while (z != v);
```

where

⟨*BF: auxiliary functions*⟩+≡

```
inline void BF_add_to_Vm(const graph& G, node z,
               node_array<bool>& in_Vm,
               node_array<edge>& pred,
               list<node>& Q, list<node>& T,
               node_array<int>& t_degree,
               node_array<list_item>& pos_in_Q,
               node_array<list_item>& pos_in_T)
{ edge e;
  forall_adj_edges(e,z)
    { node w = G.target(e);
      if ( !in_Vm[w] )
        { if (pos_in_T[w])
          { BF_delete_subtree(pos_in_T[w],Q,T,t_degree,
                                     pos_in_Q,pos_in_T);
            if (pred[w] != nil) t_degree[G.source(pred[w])]--;
          }
          pred[w] = e;
          in_Vm[w] = true;
          BF_add_to_Vm(G,w,in_Vm,pred,
                       Q,T,t_degree,pos_in_Q,pos_in_T);
        }
    }
}
```

This completes the description of the algorithm. We still have to complete the correctness proof and establish the $O(nm)$ running time.

Lemma 12 *The refined Bellman–Ford algorithm solves the single-source shortest-path problem in time $O(nm)$.*

Proof The nodes in V^+ are never reached and hence are treated correctly.

Next consider the nodes in V^-. Invariant (4) tells us that *in_Vm* is set to true only for nodes in V^-. We need to show that *in_Vm* is set to true for all nodes in V^- at some point during the execution. Let $v \in V^-$ be arbitrary. If every path defining $\mu_n(v)$ contains a negative cycle or if v is reachable from such a node then *in_V*$[v]$ is set to true by invariant

(4). We need to show that this is indeed the case. If $v \in V^-$ then there must be an integer $N > n$ and a path $[v_0 = s, v_1, \ldots, v_N = v]$ from s to v such that this path is shorter than any path from s to v with less than N edges. The prefix consisting of the first i edges of this path is a path to v_i that is shorter than any path to v_i with less than i edges. In particular, $\mu_n(v_n) < \mu_{n-1}(v_n)$ and hence any path to v_n defining $\mu_n(v_n)$ contains a negative cycle.

Finally, consider a node in V^f and assume that $\mu(v) = \mu_k(v)$. Then $dist[v] = \mu(v)$ after phase k, the tree path from s to v has cost $\mu(v)$, and the tree path is recorded in the *pred*-array by invariants (1), (2), and (6).

The two preceding paragraphs establish that there are at most $n + 1$ phases. Since each node is removed from Q at most once in each phase the running time is $O(nm)$. □

Table 7.2 shows the running times of the refined Bellman–Ford algorithm on the graphs generated by *BF_GEN*. The running time seems to triple if n is doubled. This can be explained as follows. At the beginning of each phase the nodes L to $L - K - 2$ are children of node $L - 1$ in the shortest-path tree and the nodes $L - 1$ to $L - K - 2$ (and some nodes smaller than $L - 1$) are in Q. In the basic algorithm all nodes $L - 1$ to $L - K - 2$ are removed from the queue and their outgoing edges are scanned. This results in $\Omega(m)$ edge scans per phase. In the refined algorithm the discovery of a better path to node $L - 1$ causes the nodes L to $L - K - 2$ to be removed from Q and T without(!!) scanning their edges. When the edges out of node $L - 1$ are scanned they are again added to Q and T. In this way only the edges out of node $L - 1$ are scanned in each phase. Thus only $\Theta(K) = \Theta(\sqrt{m})$ edges are scanned in each phase and the total running time is therefore $\Theta(n\sqrt{m})$. In particular, for $m = 8n$ as in Table 7.2, the running time grows like $n^{3/2}$ and hence about triples when n is doubled[17].

7.5.10 ***The All-Pairs Problem***

The all-pairs shortest-path problem is the task to compute $\mu(v, w)$ for all pairs of nodes v and w. This could be solved by solving the single-source problem with respect to each v. We describe a better method based on so-called *node potentials*; the improved method applies whenever G has no negative cycles. We will see further uses of the node potential method in the section on matchings.

A node potential assigns a number $pot(v)$ to each vertex v. The *transformed* or *reduced* edge costs $\bar{c}$ with respect to a potential function *pot* are defined by

$$\bar{c}(e) = pot(v) + c(e) - pot(w)$$

for each edge $e = (v, w) \in E$. Consider a path $p = [e_0, \ldots, e_{k-1}]$ and let $e_i = (v_i, v_{i+1})$. Then

$$\bar{c}(p) \quad = \quad \sum_{0 \le i < k} \bar{c}(e_i) \quad = \quad \sum_{0 \le i < k} (pot(v_i) + c(e_i) - pot(v_{i+1}))$$

[17] The authors initially assumed that the running time of the refined algorithm would also grow like nm on the *BF_GEN*-examples and were surprised to learn from the experiments that this is not the case. It took us some time to understand why not.

$$= \quad pot(v_0) + \sum_{0 \le i < k} c(e_i) - pot(v_k) \quad = \quad pot(v_0) + c(p) - pot(v_k),$$

i.e., the cost of p with respect to $\bar{c}$ is the cost of p with respect to c plus the potential difference between the source and the target of the path. This difference is independent(!!) of the particular path p and only depends on the endpoints of the path. Thus for any two paths p and q with the same source and the same target, $\bar{c}(p) \le \bar{c}(q)$ iff $c(p) \le c(q)$, i.e., the relative order of path costs is not changed by the transformation.

Assume now that G has no negative cycles and that all nodes of G are reachable from some node s. We claim that $pot(v) = \mu(s, v)$ is a node potential such that all reduced costs with respect to it are non-negative. This is easily seen. Observe first that $\mu(s, v)$ is finite for all v if G has no negative cycles and all nodes are reachable from s. The reduced costs are therefore well defined. Observe next that for any edge $e = (v, w)$ we have $\mu(s, v) + c(e) \ge \mu(s, w)$ and hence

$$\bar{c}(e) = \mu(s, v) + c(e) - \mu(s, w) \ge 0.$$

The observations above suggest the following strategy to solve the all-pairs problem. We first solve the single-source problem with respect to some node s from which all nodes of G are reachable. If G has a negative cycle, we stop. Otherwise we use the distances from s to transform the edge costs into non-negative ones and solve the single-source problem for each node v of G. Finally, we translate the computed distances back to the original edge costs, i.e., for each pair (v, w) we set

$$dist(v, w) = dist1(v, w) + pot(w) - pot(v),$$

where *dist* and *dist1* denote the distances with respect to the original and the transformed distance function.

How do we choose s? We add a new vertex s to G and add edges (s, v) of length 0 for all vertices of G. Observe that this does not create any additional cycles; in particular, it does not create any negative cycles. We use the distances $\mu(s, v)$ as our potential function.

$\langle$*all_pairs.t*$\rangle\equiv$

```
#include <LEDA/graph_alg.h>
template <class NT>
bool ALL_PAIRS_SHORTEST_PATHS_T(graph&G, const edge_array<NT>& c,
                                node_matrix<NT>& DIST)
{ edge e;
  node v,w;
  node s = G.new_node();
  forall_nodes(v,G) if ( v != s ) G.new_edge(s,v);
  edge_array<NT>    c1(G);
  forall_edges(e,G) c1[e] = (G.source(e) == s? 0 : c[e]);
  node_array<NT>    dist1(G);
  node_array<edge>  pred(G);
  if (!BELLMAN_FORD_T(G,s,c1,dist1,pred)) return false;
  G.del_node(s);
```

```
    forall_edges(e,G)
      c1[e] = dist1[G.source(e)] + c[e] - dist1[G.target(e)];
    // (G,c1) is a non-negative network; for every node v
    // compute row DIST[v] of the distance matrix DIST
    // by a call of DIJKSTRA_T(G,v,c1,DIST[v])
    forall_nodes(v,G) DIJKSTRA_T(G,v,c1,DIST[v],pred);
    // correct the entries of DIST
    forall_nodes(v,G)
    { NT dv = dist1[v];
      forall_nodes(w,G) DIST(v,w) += (dist1[w] - dv);
    }
   return true;
  }
```

7.5.11 *Minimum Cost to Profit Ratio Cycles*

We consider a graph G with two weight functions defined on its edges: a function p that assigns a profit to each edge and a function c that assigns a cost to each edge. For a cycle C we use

$$p(C) = \sum_{e \in C} p(e), \qquad c(C) = \sum_{e \in C} c(e), \qquad \lambda(C) = c(C)/p(C)$$

to denote the profit, the cost, and cost to profit ratio of the cycle, respectively. Our goal is to find a cycle that minimizes the cost to profit ratio[18]. We use λ^* and C^* to denote the minimum ratio and a cycle realizing it, respectively, i.e.,

$$\lambda^* = \lambda(C^*) = \min\{\lambda(C)\ ;\ C \text{ is a cycle}\}\,.$$

Figure 7.17 shows an example. We will define a function

```
rational MINIMUM_RATIO_CYCLE(graph& G,
                             const edge_array<int>& c,
                             const edge_array<int>& p,
                             list<edge>& C_opt);
```

that returns the ratio and the list of edges (in *C_opt*) of a minimum cost to profit ratio cycle. The program returns zero if there is no cycle in G; also the empty list is returned in *C_opt* in this case. The procedure runs in time $O(nm\log(n \cdot C \cdot P))$ where C and P are the maximum cost and profit of any edge, respectively. Observe that edge costs and profits are assumed to be integral. We assume that there are no cycles of cost zero or less with respect to either c or p.

Lawler [Law66] has shown that λ^* and C^* can be found by binary search and repeated shortest-path calculations.

Let λ be a real parameter and consider the cost function c_λ defined by

$$c_\lambda(e) = c(e) - \lambda \cdot p(e)$$

[18] For some readers it may seem more natural to maximize the ratio $p(C)/c(C)$. However, maximizing $p(C)/c(C)$ is the same as minimizing $c(C)/p(C)$ if the cost and profit of all cycles are positive.

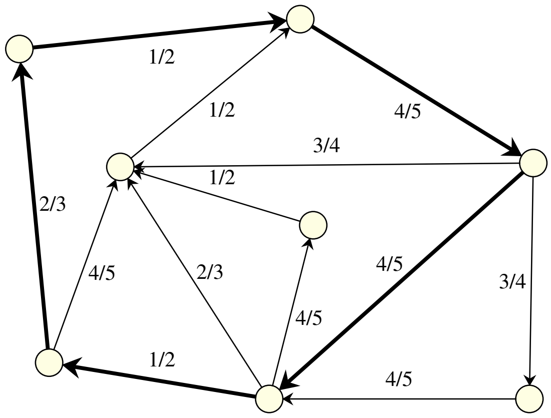

Figure 7.17 An example of a minimum cost to profit ratio cycle. Edge labels are of the form "cost/profix". The optimal cycle is shown in bold. It has cost 12 and profit 17. This figure was generated with the xlman-demo gw_minimum_ratio_cycle. The program minimum_ratio_cycle in LEDAROOT/demo/book/Graph illustrates the execution of *MINIMUM_RATIO_CYCLE*.

for all edges e. We can compare λ with the unknown λ^* by solving a shortest-path problem with cost function c_λ.

If $\lambda > \lambda^*$ then

$$c_\lambda(C^*) = c(C^*) - \lambda \cdot p(C^*) = (\lambda(C^*) - \lambda) \cdot p(C^*) < 0,$$

i.e., there is a negative cycle.

If $\lambda \leq \lambda^*$ and C is any cycle then

$$c_\lambda(C) = c(C) - \lambda \cdot p(C) = (\lambda(C) - \lambda) \cdot p(C) \geq (\lambda^* - \lambda) \cdot p(C) \geq 0,$$

i.e., there is no negative cycle.

We capture this argument in the following procedure. It takes a rational *lambda* and returns true if *lambda* is greater than λ^*. The implementation is simple. It assumes that s is a node from which all other nodes of G are reachable. We set up the cost function c_λ and then test for a negative cycle. It is important that all nodes are reachable from s (otherwise, a negative cycle could hide in a part of the graph that is unreachable from s).

We have performed one optimization. The costs $c_\lambda(e)$ for $e \in E$ are rational numbers, all with the same denominator. We therefore multiply all costs with their common denominator and work in integers.

⟨*minimum ratio cycle: compare*⟩≡

```
bool greater_than_lambda_star(const graph& G, node s,
                              const edge_array<int>& c,
                              const edge_array<int>& p,
                              rational lambda)
```

```
{ edge_array<integer> cost(G);
  edge e;
  integer num = lambda.numerator();
  integer denom = lambda.denominator();
  forall_edges(e,G) cost[e] = denom*c[e] - num*p[e];
  node_array<integer> dist(G);
  node_array<edge> pred(G);
  return !BELLMAN_FORD_T(G,s,cost,dist,pred);
}
```

We next show how to use the compare function above in a binary search for λ^*. Let P_{max} and C_{max} be the maximum profit and cost of any edge, respectively. Then

$$p(C) \in [1 \mathrel{..} n \cdot P_{max}] \text{ and } c(C) \in [1 \mathrel{..} n \cdot C_{max}].$$

Thus $\lambda(C)$ is a rational number whose denominator is in the former range and whose numerator is in the latter range. If C_1 and C_2 are cycles with $\lambda(C_1) = a/b \neq c/d = \lambda(C_2)$ then

$$|\lambda(C_1) - \lambda(C_2)| = |a/b - c/d| = |ad - cb|/(bd) \geq 1/(bd) \geq 1/(n \cdot P_{max})^2.$$

Let $\delta = 1/(n \cdot P_{max})^2$. We now have all the ingredients for a binary search. We start with the half-open interval $[\lambda_{min} \mathrel{..} \lambda_{max}) = [0 \mathrel{..} 1 + n \cdot C_{max})$ (it is convenient to maintain the invariant $\lambda_{min} \leq \lambda^* < \lambda_{max}$) and then repeatedly compare $\lambda = (\lambda_{min} + \lambda_{max})/2$ with λ^*. If $\lambda > \lambda^*$ we set λ_{max} to λ and if $\lambda \leq \lambda^*$ we set λ_{min} to λ. In this way we maintain the invariant $\lambda_{min} \leq \lambda^* < \lambda_{max}$. We continue until $\lambda_{max} - \lambda_{min} \leq \delta$. Then $\lambda_{min} \leq \lambda^* < \lambda_{min} + \delta$ and hence there is no cycle C with $\lambda^* < \lambda(C) < \lambda_{max}$. We will use this observation below to extract C^* and λ^*.

The following procedure summarizes the discussion. We first add a new node s and edges (s, v) for all $v \in V$ to our graph (this makes all nodes reachable from s) and then perform the binary search. Whenever a midpoint is computed in the binary search we normalize its representation, i.e., cancel out common factors of numerator and denominator. This is important to keep the representations of the rationals small.

⟨*_minimum_ratio_cycle.c*⟩≡

```
#include <LEDA/templates/shortest_path.t>
#include <LEDA/rational.h>
⟨minimum ratio cycle: compare⟩
rational MINIMUM_RATIO_CYCLE(graph& G,
                             const edge_array<int>& c,
                             const edge_array<int>& p,
                             list<edge>& C_opt)
{ node v; edge e;
  ⟨additional variables for demos⟩ // for minimum ratio cycle demo
  C_opt.clear();
  if ( Is_Acyclic(G) ) return rational(0);
  node s = G.new_node();
```

```
  forall_nodes(v,G) if (v != s) G.new_edge(s,v);
  edge_array<int> c1(G);
  edge_array<int> p1(G);
  int Cmax = 0; int Pmax = 0;
  forall_edges(e,G)
  { if (G.source(e) == s) { c1[e] = p1[e] = 0; }
    else
    { c1[e] = c[e]; p1[e] = p[e];
      Cmax = Max(Cmax,c[e]);
      Pmax = Max(Pmax,p[e]);
    }
  }
  int n = G.number_of_nodes();
  ⟨minimum ratio cycle: check precondition⟩
  integer int_n(n);
  integer int_Pmax(Pmax);
  rational lambda_min(integer(0));
  rational lambda_max(int_n * integer(Cmax) + integer(1));
  rational delta(1,int_n * int_n * int_Pmax * int_Pmax);
  while (lambda_max - lambda_min > delta)
  { rational lambda = (lambda_max + lambda_min)/2;
    lambda.normalize();   // important
    ⟨report progress in demos⟩
    if ( greater_than_lambda_star(G,s,c1,p1,lambda) )
     lambda_max = lambda;
    else
     lambda_min = lambda;
  }
  rational lambda_opt;
  { ⟨minimum ratio cycle: determine lambda_opt and C_opt⟩ }
  G.del_node(s);
  return lambda_opt;
}
```

When the binary search terminates we have

$$\lambda_{max} - \lambda_{min} \le \delta \text{ and } \lambda_{min} \le \lambda^* < \lambda_{max}$$

and hence there can be no cycle C with $\lambda^* < \lambda(C) < \lambda_{max}$. Let $\lambda = \lambda_{max}$. Since $\lambda^* < \lambda_{max}$, there is a negative cycle with respect to c_λ. Let C be any negative cycle with respect to c_λ. Then $\lambda(C) < \lambda = \lambda_{max}$ and hence $\lambda(C) = \lambda^*$. We conclude that any negative cycle with respect to c_λ is an optimal cycle.

A negative cycle with respect to c_λ is easy to find. We set up the cost function c_λ and run BELLMAN_FORD_T. We then run CHECK_SP_T on the output. It labels all nodes lying on a negative cycle by -2. We pick any such node and trace the cycle containing it.

⟨*minimum ratio cycle: determine lambda_opt and C_opt*⟩≡
```
edge_array<integer> cost(G);
node v; edge e;
integer num = lambda_max.numerator();
integer denom = lambda_max.denominator();
forall_edges(e,G) cost[e] = denom*c1[e] - num*p1[e];
node_array<integer> dist(G);
node_array<edge> pred(G);
BELLMAN_FORD_T(G,s,cost,dist,pred);
node_array<int> label = CHECK_SP_T(G,s,cost,dist,pred);
forall_nodes(v,G) if (label[v] == -2) break;
int P = 0; int C = 0;
node z = v;
do { P += p[pred[z]]; C += c[pred[z]];
     C_opt.append(pred[z]);
     z = G.source(pred[z]);
   } while ( z != v);
lambda_opt = rational(C)/rational(P);
```

We still need to show how to check the precondition $p(C) > 0$ and $c(C) > 0$ for all cycles C. We discuss the latter condition. Consider the cost function c_λ defined by $c'(e) = c(e) - 1/n$ for all edges e. Clearly, if there is no negative cycle with respect to c' then there is no cycle of length zero or less with respect to c. Conversely, if $c(C) > 0$ and hence $c(C) \geq 1$ for all C then $c'(C) = c(C) - |C|/n \geq 1 - n/n \geq 0$ and there is no negative cycle with respect to c'.

We can therefore misuse our comparison function to check the precondition.

⟨*minimum ratio cycle: check precondition*⟩≡
```
edge_array<int> unit_cost(G,1);
rational one_over_n(integer(1),integer(n));
if (greater_than_lambda_star(G,s,c1,unit_cost,one_over_n))
   error_handler(1,"cycle of cost zero or less wrt c");
if (greater_than_lambda_star(G,s,p1,unit_cost,one_over_n))
   error_handler(1,"cycle of cost zero or less wrt p");
```

The running time of the algorithm is $O(nm\log(n \cdot P_{max} \cdot C_{max}))$. This can be seen as follows. The binary search starts with an interval of length $nC_{max} + 1$ and ends with an interval of length $1/(n \cdot P_{max})^2$. The length of the interval is halved in each iteration and hence the number of iterations is $O(\log(n \cdot P_{max} \cdot C_{max}))$. Each iteration takes time $O(nm)$.

The technique used in our program for the minimum ratio cycle problem is called *parametric search*. Parametric search is applicable in the following situation:

- One searches for the threshold value λ^* of a monotone predicate $P(\lambda)$ of one real argument λ. A predicate P is monotone if

$$\lambda_1 < \lambda_2 \text{ and } P(\lambda_1) \text{ imply } P(\lambda_2),$$

and the threshold value of P is

$$\lambda^* = \inf\{\lambda \; ; \; P(\lambda)\} \, .$$

In the problem of this section $P(\lambda)$ holds if there is a negative cycle with respect to the cost function c_λ.

- There is a decision procedure for $P(\lambda)$.
- There is a master procedure that drives the search for λ^*. We used binary search as the master procedure in this section.

We refer the reader to [Meg83] and [AST94] for further applications of parametric search.

Parametric search has high demands on the underlying arithmetic. You can get an impression of the arithmetic demand of the minimum ratio cycle procedure by calling the program minimum_ratio_cycle in LEDAROOT/demo/book/Graph. The paper [SSS97] discusses an application of the number class *real* to parametric search.

Exercises for 7.5

1 (Single-pair shortest-path problem) Let s and t be distinct nodes in a directed graph with non-negative edge costs. The goal is to compute a shortest path from s to t. Assume that there is heuristic information available which gives, for any node v, a *lower bound* $lb(v)$ for the length of a shortest path from v to t. Modify Dijkstra's algorithm such that $dist(v) + lb(v)$ is used as the priority of node v.

2 Show that the condition $d(v) \geq \mu(v)$ for all v in part (b) of Lemma 6 is essential, i.e., the claim does not hold without it.

3 Investigate the following shortest-path algorithm. Split the input graph G into G^- consisting of all edges of negative cost and $G^{\geq 0}$ consisting of all edges of non-negative cost. What can you say when G^- is not acyclic? If G^- is acyclic then run alternately the acyclic shortest-path algorithm on G^- and Dijkstra's algorithm on $G^{\geq 0}$. In each case the distance labels output by the preceding run must be taken as the initial distance labels for the next run. Modify the programs accordingly.

4 Consider the following version of the Bellman–Ford algorithm. It iterates over all edges on the graph n times. Whenever an edge $e = (v, w)$ is considered, $d(w)$ is set to the minimum of $d(w)$ and $d(v) + c(e)$.

```
dist[s] = 0;
forall_nodes(v,G) pred[v] = nil;

for(int i = 0; i < n; i++)
  forall_edges(e,G)
  { node v = G.source(e);
    node w = G.target(e);
    if ( v != s && pred[v] == nil) continue;
    // dist[v] is finite
    d = dist[v] + cost[e];
    if ( pred[w] == nil && v != s || d < dist[w] )
    { dist[w] = d; pred[w] = e; }
  }
```

Show that the algorithm computes all finite distances correctly (Hint: show that $d(v)$ is bounded above by the length of a shortest path consisting of at most k edges after the k-th iteration.). Modify the algorithm so that it satisfies our output convention. Implement the algorithm and compare its running time to the implementations of the Bellman–Ford algorithm given in the text. What is best case running time of the algorithm?

5 In all our algorithms we implemented the test $c < d(w)$ in a somewhat clumsy way due to the fact that $d(w)$ may be $+\infty$ and that most number types have no representation for $+\infty$. Show that nC where C is the largest cost of any edge can be taken as an approximation of $+\infty$. Modify the algorithms accordingly and time them in comparison to the algorithms in the text.

6 Our algorithm for determining minimum ratio cycles uses binary search. It starts with an interval of length $nC_{max} + 1$ and stops as soon as the length of the interval becomes $1/(nP_{max})^2$ or less. Thus there are $\log(n^3 P_{max}^2 C_{max})$ iterations and hence the algorithm handles rational numbers with denominator as large as $n^3 P_{max}^2 C_{max}$. This is unnecessarily large since the $\lambda(C)$ are rational numbers whose denominator is bounded by nP_{max}. Explore the possibility that the values of λ are restricted to rational numbers whose denominator is bounded by nP_{max}. This requires us to write a function that "rounds" a rational number to the closest rational number whose denominator is bounded by some prescribed integer. Inspect the function *small_rational_near* of class *rational* to see how such a function can be realized.

7 Define a number class *NT_star*. The definition is with respect to a fixed graph G with integral weight functions c and p. Let λ^* be the minimum cost to profit ratio of a cycle in G. Each number of this class is represented by a pair of integers. Addition is componentwise and there is no multiplication. Zero has both its components equal to zero. A pair (a, b) is less than (equal to, larger than) a pair (c, d) if $a + \lambda^* b < (=, >) c + \lambda^* d$. Implement the compare function as follows. Let $\lambda = (c - a)/(b - d)$ and use the comparison between λ and λ^* (realized by a shortest-path computation as in the text). The number type maintains an interval $[\lambda_{min} .. \lambda_{max}]$ containing λ^*. Whenever a comparison is performed this interval is updated. Use the number type in a shortest-path computation on the graph G. What will the final interval be?

7.6 Bipartite Cardinality Matching

We start with the problem definition and the functionality of the bipartite matching algorithms. We describe a checker and then lay the foundations of matching algorithms. In the bulk of the section we discuss the implementations of several matching algorithms and derive some general implementation principles. We close with an experimental comparison of our implementations.

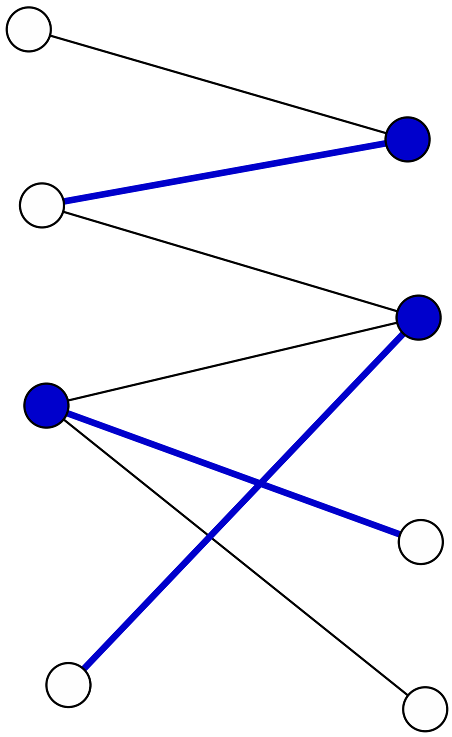

Figure 7.18 A graph and a maximum matching: The bold edges form a matching of cardinality three. The filled nodes form a node cover of cardinality three; a node cover is a set of nodes containing at least one endpoint of every edge. The node cover proves the optimality of the matching. This figure was generated with the xlman-demo gw_mcb_matching.

7.6.1 *Concepts and Functionality*

Let $G = (V, E)$ be a graph. A *matching* M is a subset of the edges no two of which share an endpoint, see Figure 7.18. The cardinality $|M|$ of a matching M is the number of edges in M.

A node v is called *matched* with respect to a matching M if there is an edge in M incident to v and it is called *free* or *unmatched* otherwise. An edge $e \in M$ is called a *matching* edge. A matching is called *perfect* if all nodes of G are matched. For a matched node v the unique node w connected to v by a matching edge is called the *mate* of v.

In this section we assume that G is *bipartite*, i.e., that there is a partition $V = A \dot{\cup} B$ of the nodes of G such that every edge of G has one endpoint in A and one endpoint in B. Matchings in general graphs are the topic of Section 7.7. The procedure

```
bool Is_Bipartite(const graph& G, list<node>& A, list<node>& B)
```

tests whether G is bipartite and if so computes an appropriate partition of the nodes in lists A and B. It runs in time $O(n + m)$.

The procedure

```
list<edge> MAX_CARD_BIPARTITE_MATCHING(graph& G);
```

returns a maximum cardinality matching; the graph G must be bipartite. The worst case and average case running time of the algorithm are $O(\sqrt{n} \cdot m)$ and $O(m \log n)$, respectively. The variant

```
list<edge> MAX_CARD_BIPARTITE_MATCHING(graph& G, node_array<bool>& NC);
```

returns in addition a proof of optimality in the form of a node cover NC.

A *node cover* is a set U of nodes such that for every edge (v, w) of G at least one of the endpoints is in U.

Lemma 13 *Let M be a matching and let U be a node cover. Then $|M| \le |U|$. If $|M| = |U|$ then M is a maximum cardinality matching and U is a minimum cardinality node cover.*

Proof Since U is a node cover, each edge $e \in M$ has at least one endpoint in U. We assign an endpoint in U to each edge in M; for an edge in M having both endpoints in U the choice of the endpoint is arbitrary. Each node is assigned at most once since every node v has at most one edge in M incident to it. Hence, $|M| \le |U|$.

If $|M| = |U|$ then M is a maximum cardinality matching, since no matching can have cardinality larger than $|U|$, and U is a minimum cardinality node cover, since no node cover can have cardinality smaller than $|M|$. □

We will later show that in bipartite graphs there is always a node cover and a matching of the same cardinality. Lemma 13 is the basis for a checker for maximum cardinality matchings in bipartite graphs. The checker takes a set M of edges and a set NC of nodes, and checks that M is a matching, NC is a node cover, and that the cardinality of M is equal to the cardinality of NC.

⟨*_mcb_matching*⟩≡

```
static bool False(string s)
{ cerr << "CHECK_MCB: " + s +"\n"; return false; }
bool CHECK_MCB(const graph& G,const list<edge>& M,
               const node_array<bool>& NC)
{ node v; edge e;
  // check that M is a matching
  node_array<int> deg_in_M(G,0);
  forall(e,M)
  { deg_in_M[G.source(e)]++;
    deg_in_M[G.target(e)]++;
  }
  forall_nodes(v,G)
   if ( deg_in_M[v] > 1 ) return False("M is not a matching");
  // check size(M) = size(NC)
  int K = 0;
  forall_nodes(v,G) if (NC[v]) K++;
```

```
  if ( K != M.size() ) return False("M is smaller than node cover");
  // check that NC is a node cover
  forall_edges(e,G)
    if ( ! (NC[G.source(e)] || NC[G.target(e)]) )
      return False("NC is not a node cover");
  return true;
}
```

7.6.2 *Concepts for Maximum Matching Algorithms*

We introduce the concepts of alternating and augmenting paths that are crucial for all matching algorithms. A large part of the section applies not only to bipartite graphs but to all graphs. We will clearly state when we restrict attention to bipartite graphs.

A simple path $p = [e_0, e_1, \ldots, e_{k-1}]$ from v to w in G is called an *alternating* path with respect to a matching M if:

- the edges in p are alternately in M and not in M,
- exactly one of e_0 and e_{k-1} is a matching edge if $v = w$,
- either e_0 is a matching edge or v is free and either e_{k-1} is a matching edge or w is free if $v \neq w$.

Figure 7.19 shows examples. The importance of alternating paths stems from:

Lemma 14 *If p is an alternating path with respect to M then $M' = M \oplus p = (M \setminus p) \cup (p \setminus M)$ is also a matching.*

Proof Consider any node z. We need to show that at most one edge of M' is incident to z. This is obvious if z does not lie on p or if z is not an endpoint of p or if p is a cycle. So assume that z is an endpoint of p and p is not a cycle, say $z = v \neq w$. Since p is simple, it contains only one edge incident to v, namely e_0. Moreover, if $e_0 \notin M$ then v is free with respect to M. Thus at most one edge of M' is incident to v. □

If p is alternating with respect to M then $M \oplus p$ has cardinality one larger than M if both endpoints of p are free, has the same cardinality as M if exactly one endpoint is free, and has cardinality one smaller than M if no endpoint is free.

An alternating path p is called *augmenting* if both endpoints of p are free. For an augmenting path the cardinality of the matching $M \oplus p$ is one larger than the cardinality of M. If M does not have maximum cardinality then there is always an augmenting path, as the next lemma shows; if M is "far" from optimality there are many augmenting paths (even short ones).

Lemma 15 *Let M and M' be matchings in a graph G. We have the following:*

- *$M \oplus M'$ consists of alternating paths and alternating cycles.*

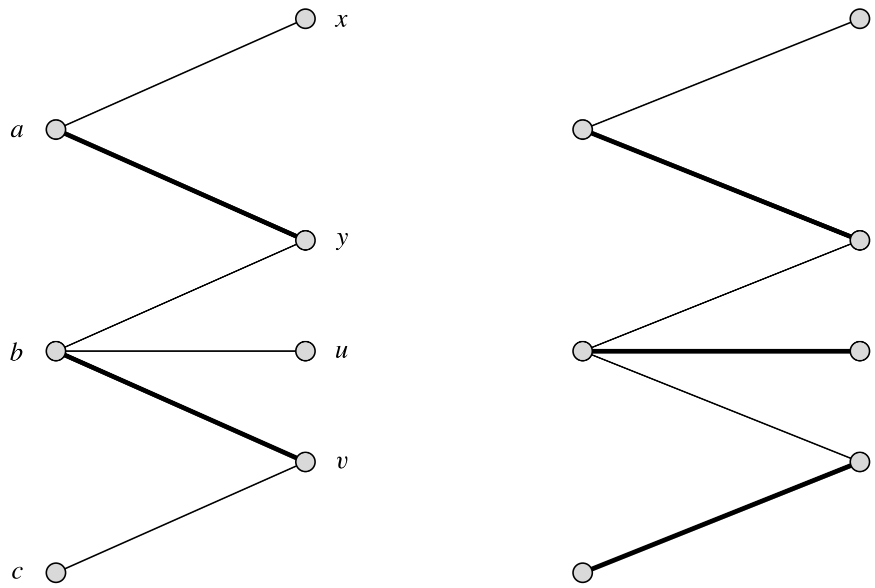

Figure 7.19 Alternating paths: The edges of a matching M are shown in bold. The paths $p_1 = [a, y, b, v]$, $p_2 = [u, b, v]$, and $p_3 = [u, b, v, c]$ are alternating with respect to M, but the path $p_4 = [a, y, b]$ is not. Augmenting M by p_1 decreases the size of the matching (as both endpoints of p_1 are matched), augmenting by p_2 leaves the size of the matching unchanged (as exactly one of the endpoints of p_2 is matched), and augmentation by p_3 increases the size of the matching by one (as both endpoints of p_3 are free). The right half of the figure shows the matching obtained by augmenting by p_3.

- *If $|M| < |M'|$ then there is at least one augmenting path in G with respect to M.*
- *Let $d = |M'| - |M|$. Then there is at least one augmenting path of length at most n/d and there are at least $d/2$ augmenting paths of length at most $2n/d$.*

Proof Consider the graph with edge set $M \oplus M'$. In this graph each node has degree zero, one, or two, and hence the graph consists of paths, cycles, and isolated nodes. Since M and M' are matchings, the edges of M and M' alternate on every path and cycle.

An alternating cycle contains the same number of edges of M and M'. Thus, if $|M| < |M'|$, then there must be at least one path in $M \oplus M'$ which contains more edges of M' than of M. Such a path contains one more edge of M' than of M and hence the first and the last edge of the path belong to M'. Thus the path is augmenting with respect to M.

The argument in the previous paragraph actually shows that there must be d paths in $M \oplus M'$ which contain more edges of M' than of M. Thus there are d augmenting paths with respect to M. The paths are node-disjoint and hence contain at most n edges in total. Thus their average length is at most n/d and there are at least $d/2$ paths whose length is at most $2n/d$. □

It is worthwhile looking at a numerical example. Assume that M is empty and that G

allows for a perfect matching. Taking M' as a perfect matching we have $d = n/2$ and hence there are at least $n/4$ augmenting paths of length at most 2.

Corollary 2 *Let M be a matching in a graph G. M is a maximum cardinality matching in G iff there is no augmenting path in G with respect to M.*

Proof Clearly, if there is an augmenting path p with respect to M then M is not a maximum cardinality matching.

Assume conversely, that M is not a maximum cardinality matching. Then there is a matching M' such that $|M| < |M'|$. Lemma 15 implies the existence of an augmenting path with respect to M. □

Corollary 2 immediately suggests an algorithm for finding maximum matchings.

M = some matching;
while there is an augmenting path p with respect to M
{ augment M by p; }

In the remainder of this section we concentrate on bipartite graphs. In a bipartite graph $G = (A \dot\cup B, E)$ there is a particularly simple method for finding augmenting paths. We direct all free edges from A to B and all matching edges from B to A. The existence of an augmenting path is then tantamount to the existence of a path from a free node in A to a free node in B. Also, augmentation by a path p is trivial. One simply reverses the direction of all edges on the path. Observe that this correctly records that the endpoints of p are now matched and that M was replaced by $M \oplus p$, see Figure 7.20. *We will use this "directed" view in all our implementations of bipartite matching algorithms.*

Before we turn to implementations we make the observation that it suffices to search for augmenting paths only from vertices in A and from each vertex only once, i.e., the algorithm above can be modified to:

M = some matching;
forall nodes v in A
{ **if** there is an augmenting path p with respect to M starting in v
  { augment M by p; }
}

We prove that the modified algorithm is correct. We observe first that the set of nodes in A that are matched in $M \oplus p$ are exactly the nodes that are matched in M plus the source node of p.

Let M_0 be the initial matching, let A_0 be the nodes in A that are matched in M_0 and let $v_1, v_2, \ldots, v_k$ be the vertices in $A \setminus A_0$ in the order in which they are considered. For all i, $i \geq 1$, let M_i be equal to M_{i-1} if there is no augmenting path p_i with respect to M_{i-1}

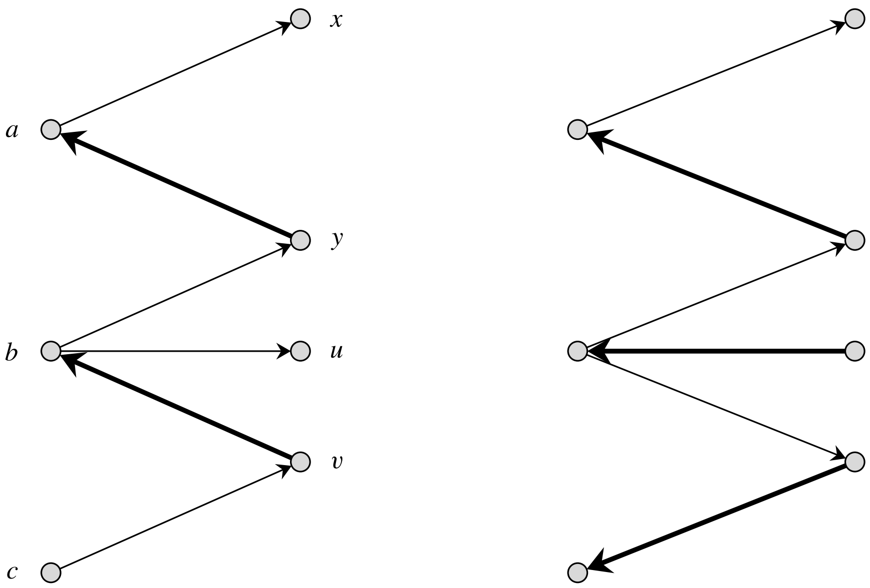

Figure 7.20 The edges of a matching M are shown in bold. Matching edges are directed from right to left and non-matching edges are directed from left to right. The path $p = [c, v, b, u]$ is an augmenting path with respect to M. Augmenting M by p yields the matching $M \oplus p$ shown in the right half of the figure.

starting in v_i and let it be $M_{i-1} \oplus p_i$ otherwise. Let $A_i = A_0 \cup \{v_1, \ldots, v_i\}$ and let G_i be the subgraph spanned by $V_i = A_i \dot{\cup} B$.

Lemma 16 *For all i: M_i is a maximum cardinality matching in G_i.*

Proof The claim is certainly true for $i = 0$ as all nodes in A_0 are matched. So consider $i \geq 1$ and assume that the claim is true for M_{i-1}. Let k_i be the maximum cardinality of a matching in the subgraph spanned by V_i. If $k_i = k_{i-1}$ then the claim clearly holds for i. So assume that $k_i > k_{i-1}$ and let M^* be an optimal matching in G_i. Then v_i must be matched in M^* (otherwise there would be a matching of cardinality k_i in G_{i-1}, a contradiction to the optimality of M_{i-1}) and hence $M_{i-1} \oplus M^*$ contains a path p starting in v_i. The path starts with an edge in M^* and is alternating with respect to M_{i-1}; we consider the maximal length path of this form. If p also ends with an edge in M^* then p is augmenting with respect to M_{i-1} and hence the cardinality of M_i is one larger than the cardinality of M_{i-1}. Thus M_i is optimal. If p ends with an edge in M_{i-1} then $M^* \oplus p$ has the same cardinality as M^* and does not match v_i. Thus there is a matching of cardinality k_i in G_{i-1}, a contradiction to the optimality of M_{i-1}. $\square$

7.6.3 *Translating between the Directed and the Undirected View*

We stated in the previous section that augmenting paths in bipartite graphs are particularly easy to find if one adopts a directed view: all matching edges are directed from B to A, all non-matching edges are directed from A to B, and augmentation by a path p means to reverse all its edges. We take this directed view in all our implementations of bipartite matching algorithms. However, we do not want to impose this directed view on the users of matching algorithm. For them an "undirected" view is more appropriate. In this section we discuss how to translate between the two views.

We postulate the following common interface for all our implementations:

- The node set is partitioned into disjoint sets A and B (given as lists of nodes).
- All edges are directed from A to B.
- The implementations are allowed to modify the graph in two ways: they may reorder adjacency lists and they may change the orientation of edges during execution. At termination, all edges must again[19] be directed from A to B. However, the ordering of the adjacency lists may be arbitrary.

In this section we show how to prepare this input format and how to restore the original graph.

We determine a bipartition $V = A \dot\cup B$ of V by calling *Is_Bipartite*(G, A, B). This call will return true iff G is bipartite and compute A and B if G is bipartite. We then orient all edges from A to B. Having oriented all edges from A to B we compute a maximum matching by calling one of our matching algorithms. After returning from the matching algorithm we restore the original orientation of all edges and the original order of all adjacency lists.

We give more details. We deal with the edge orientations first. We collect all edges out of nodes in B in a list *edges_out_of_B* and reverse the orientation of all of them (operation *rev_edge*). After return from the matching algorithm we again reverse all edges in *edges_out_of_B* and thus restore their original orientation.

We come to the orderings of the adjacency lists. We number all edges according to their original order and use *sort_edges* to restore the original order.

Among our implementations of matching algorithms the algorithm by Alt, Blum, Mehlhorn, and Paul seems to be the best, see Section 7.6.7 for an experimental comparison of all implementations. We therefore use it as our default implementation.

⟨*_mcb_matching*⟩+≡

```
list<edge> MAX_CARD_BIPARTITE_MATCHING(graph& G, node_array<bool>& NC)
{ list<node> A,B;
  node v; edge e;
  if ( !Is_Bipartite(G,A,B) )
    error_handler(1,"MAX_CARD_BIPARTITE_MATCHING: G is not bipartite");
  edge_array<int> edge_number(G); int i = 0;
  forall_nodes(v,G)
```

[19] We would not make this requirement anymore if we could start from scratch.

```
    forall_adj_edges(e,v) edge_number[e] = i++;
  list<edge> edges_out_of_B;
  forall(v,B)
  { list<edge> outedges = G.adj_edges(v);
    edges_out_of_B.conc(outedges);
  }
  forall(e,edges_out_of_B) G.rev_edge(e);
  list<edge> result = MAX_CARD_BIPARTITE_MATCHING_ABMP(G,A,B,NC);
  forall(e,edges_out_of_B) G.rev_edge(e);
  G.sort_edges(edge_number);
#ifndef LEDA_CHECKING_OFF
CHECK_MCB(G,result,NC);
#endif
  return result;
}
```

7.6.4 *The Ford and Fulkerson Algorithm*

In this section $G = (V, E)$ is a bipartite graph with $V = A \dot{\cup} B$. All edges have one endpoint in A and one endpoint in B and all edges are directed from A to B. Our goal is to compute a matching of maximum cardinality. We are allowed to reorder adjacency lists and to reorient edges but we must at the end again orient all edges from A to B.

We will give several implementations of the Ford and Fulkerson algorithm [FF63] already derived in Section 7.6.2.

M = some matching;
forall nodes v in A
{ **if** there is an augmenting path p with respect to M starting in v
  { augment M by p; }
}

The implementations differ:

- in the strategy used to search for augmenting paths (we will study depth-first and breadth-first search),
- in the choice of the initial matching (we will either use the empty matching or the matching produced by the so-called greedy heuristic),
- in the data structures used.

All implementations have a worst case running time of $O(nm)$. They have different best case behaviors and different average case behaviors and they behave drastically differently in practice.

A First Implementation: We implement the algorithm above and call the resulting procedure MAX_CARD_BIPARTITE_MATCHING_FFB; FFB stands for basic version of the Ford and Fulkerson algorithm. It starts by declaring all nodes as free and then iterates over all nodes in A. For each node v in A it tries to find an augmenting path starting in v by calling *find_aug_path_by_dfs*$(G, f,$ *free*, *reached*$)$ for the edges f out of v. A call *find_aug_path_by_dfs*$(G, f, \ldots)$ returns true if there is an augmenting path starting with f and returns false otherwise. In the former case it also augments the current matching by the path (by reversing all its edges) and labels the endpoint in B of the path as non-free. In either case it labels all visited nodes (by setting *reached*$[w]$ to true for each visited node w). If an augmenting path starting with a particular edge f is found, v is made non-free and the next node in A is considered.

When all nodes in A have been considered the result list is prepared, all edges are directed from A to B (as this is required by our interface convention), and a node cover is computed.

⟨*_FFB_matching*⟩≡

```
⟨FFB: dfs⟩
list<edge> MAX_CARD_BIPARTITE_MATCHING_FFB(graph& G,
                                 const list<node>& A, const list<node>& B,
                                 node_array<bool>& NC)
{ node v; edge e;
  node_array<bool> free(G,true);
  // check that all edges are directed from A to B
  forall(v,B) assert(G.outdeg(v) == 0);
  forall(v,A)
  { edge f;
    node_array<bool> reached(G,false);
    forall_adj_edges(f,v)
    { if (find_aug_path_by_dfs(G,f,free,reached))
      { free[v] = false;
        break;
      }
    }
  }
  ⟨MCB: prepare result and node cover and restore orientations⟩
}
```

We give the details of *find_aug_path_by_dfs*$(G, f,$ *free*, *reached*$)$. It is a variant of depth-first search; later in the section we will also consider breadth-first search. In a general call, f is some edge and the recursion stack contains a path p starting at a free node in A and ending in f. In the procedure we distinguish cases according to whether the target node of f is free or not.

If the target node w of f is free, we have found an augmenting path. We label w as non-free and then reverse all edges in p. This can be done by unwinding the recursion stack and reversing all edges contained in it. More precisely, we reverse f and return true. The enclosing call receives true and knows that an augmenting path has been found. It reverses its argument and returns true. In this way all edges on the path are reversed.

If w is not free, we try to extend the path. Let $e = (w, z)$ be any edge out of w. If z was already reached then there is no need to explore e as we know already that no free node in B can be reached from z. If z was not reached yet we make a recursive call for e.

⟨*FFB: dfs*⟩≡

```
static bool find_aug_path_by_dfs(graph& G, edge f,
                       node_array<bool>& free, node_array<bool>& reached)
{ node w = G.target(f);
  reached[w] = true;
  if (free[w])
  { free[w] = false;
    G.rev_edge(f);
    return true;
  }
  edge e;
  forall_adj_edges(e,w)
  { node z = G.target(e);
    if ( reached[z] ) continue;
    if ( find_aug_path_by_dfs(G,e,free,reached) )
    { G.rev_edge(f);
      return true;
    }
  }
  return false;
}
```

We complete the description of our first matching algorithm by discussing how to produce the matching, the node cover, and how to orient all edges from A to B. The matching M consists of all edges that are directed from B to A. Their directions need to be reversed.

How can we find a node cover *NC*? We claim that the following rule determines a node cover. For each matched edge we select the endpoint in B, if this endpoint can be reached from a free node in A, and the endpoint in A otherwise, see Figure 7.21.

Clearly, each matching edge is incident to a node in *NC*. We now consider a non-matching edge $e = (v, w)$ with $v \in A$ and $w \in B$. If v is free then w must be matched (by optimality of M), and w was selected according to the rule above. If v is matched and was not selected then there must be a matching edge $f = (v, w')$ with w' selected. This means that w' can be reached from a free node in A. Extend this path by f and e to see that w is selected according to the rule above.

⟨*MCB: prepare result and node cover and restore orientations*⟩≡

```
list<edge> result;
forall(v,B)
 forall_adj_edges(e,v) result.append(e);
forall_nodes(v,G) NC[v] = false;
node_array<bool> reachable(G,false);
```

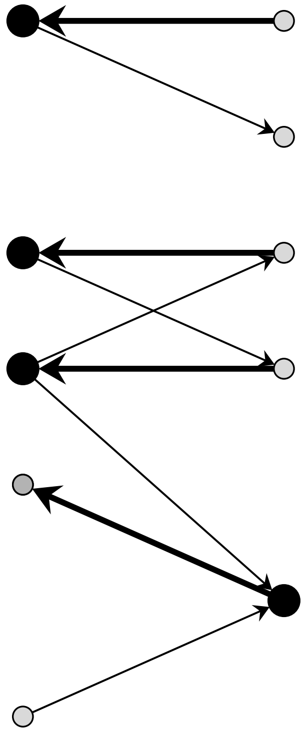

Figure 7.21 The edges of a matching M are shown in bold. Matching edges are directed from right to left and non-matching edges are directed from left to right. The edge e_0 is the only matching edge whose endpoint in B is reachable from a free node in A. The node cover is shown as large solid circles.

```
forall(v,A)
  if (free[v]) DFS(G,v,reachable);
forall(e,result)
  if ( reachable[G.source(e)] )
    NC[G.source(e)] = true;
  else
    NC[G.target(e)] = true;
forall(e,result) G.rev_edge(e);
return result;
```

What is the time complexity of our implementation? The worst case complexity is $O(nm)$ since we search at most n times for an augmenting path and since each search takes time $O(m)$ in the worst case. On many graphs the running time is smaller. However, the running time of the implementation above is never better than $\Omega(n^2)$. *This is due to very poor algorithmics* which lets each search for an augmenting path take time $\Omega(n)$. The culprit is the innocent looking statement

```
node_array<bool> reached(G,false);
```

which consumes $\Theta(n)$ time and is executed in each of the n phases. We will next describe two improvements. None of them improves the worst case running time, but both of them improve the running time dramatically for many inputs.

Improving the Best Case: We show how to improve the best case from $\Omega(n^2)$ to $O(m)$. We will see that the optimization has a dramatic effect on the observed running time of our implementation.

Consider the first search for an augmenting path when the current matching is still empty. At this point any edge is an augmenting path and hence the first call of *find_aug_path_by_dfs* returns with success immediately. However, in the implementation above the search will take time $\Omega(n)$ since the initialization of the *node_array*<*bool*> *reached* takes linear time. We aim for a design where the cost for reinitializing *reached* is proportional to the number of nodes that were actually reached in the previous search and not proportional to the total number of nodes. We call this the principle of

paying only for what we actually touched
and not
for what we could have conceivably touched.

We describe three ways to realize the principle.

The first method uses a *stack*<*node*> *reached_stack* in addition to the boolean array *reached*. Whenever *reached*$[w]$ is set to true for a node w we also push w onto *reached_stack* and after a successful augmentation we use *reached_stack* to reset *reached* to false for all nodes on the stack. In this way reinitialization takes time proportional to the number of elements reached. We obtain the following code. In *find_aug_path_by_dfs* we write

```
reached[w] = true; reached_stack.push(w);
```

and in the body of MAX_CARD_BIPARTITE_MATCHING_FFB we write

```
node_array<bool> reached(G,false);
stack<node> reached_stack;

forall(v,A)
{ edge f;
  forall_adj_edges(f,v)
  { if (find_aug_path_by_dfs(G,f,free,reached))
    { free[v] = false;
      while ( !reached_stack.empty() )
        reached[reached_stack.pop()] = false;
      break;
    }
  }
}
```

The second method uses the data type *node_slist*. This data type offers the functions *member*, *push*, *pop*, and *empty* and hence combines the functionality of a boolean array with a stack. We leave it to the reader to rewrite the algorithm so that a *node_slist* is used instead of *reached* and *reached_stack*.

The third method uses a counter *number_of_augmentations* and a *node_array*<*int*> *mark* instead of *reached*. The counter is increased whenever an augmentation occurs[20] and the mark *number_of_augmentations* is assigned to all nodes reached in the current search for an augmenting path. The test whether a node w has already been reached in the current search amounts to *mark*$[w] ==$ *number_of_augmentations*. We obtain the following code. In this code we have also made provisions for our second improvement in form of the program chunk ⟨*MCB: greedy heuristic*⟩.

⟨*_FF_DFS_matching*⟩≡

```
⟨FF: dfs⟩
list<edge> MAX_CARD_BIPARTITE_MATCHING_FF_DFS(graph& G,
                               const list<node>& A, const list<node>& B,
                               node_array<bool>& NC)
{ node v; edge e;
  node_array<bool> free(G,true);
  node_array<int> mark(G,-1);
  // check that all edges are directed from A to B
  forall(v,B) assert(G.outdeg(v) == 0);
  ⟨MCB: greedy heuristic⟩
  number_of_augmentations = 0;
  forall(v,A)
  { if ( !free[v] ) continue;
    edge f;
    forall_adj_edges(f,v)
    { if (find_aug_path_by_dfs(G,f,free,mark))
      { free[v] = false;
        number_of_augmentations++ ;
        break;
      }
    }
  }
  ⟨MCB: prepare result and node cover and restore orientations⟩
}
```

where

⟨*FF: dfs*⟩≡

```
static int number_of_augmentations;
static bool find_aug_path_by_dfs(graph& G, edge f,
             node_array<bool>& free, node_array<int>& mark)
{ node w = G.target(f);
  mark[w] = number_of_augmentations;
  if (free[w])
  { free[w] = false;
```

[20] There are 2^{32} numbers of type *int* and hence this counter will never overflow.

```
    G.rev_edge(f);
    return true;
  }
  edge e;
  forall_adj_edges(e,w)
  { node z = G.target(e);
    if ( mark[z] == number_of_augmentations ) continue;
    if ( find_aug_path_by_dfs(G,e,free,mark))
    { G.rev_edge(f);
      return true;
    }
  }
  return false;
}
```

The third method has an interesting side effect (which we did not intend). Suppose that we searched for an augmenting path from a and did not succeed. Then all nodes reached by this search are marked (and stay marked) and hence the search from the next free node in A will not explore them. In this way the worst case time between successive augmentations is $O(m)$.

Table 7.3 compares the running times of the implementations FFB and FF in columns FFB- and FF- on random bipartite graphs; the other columns will be explained in the next section. Observe that FF is much faster than FFB. We conclude that the principle of

paying only for what we actually touched
and not
for what we could have conceivably touched

is worth being observed.

The Greedy Heuristic: We come to our second improvement. In our considerations at the beginning of the section we started the matching algorithm with the line

```
M = some matching;
```

So far, we have chosen the empty matching as out initial matching. We will now do something more clever and use the so-called *greedy heuristic* to find an initial matching. The greedy heuristic considers all edges in turn and adds an edge to the current matching if both of its endpoints are free.

⟨*MCB: greedy heuristic*⟩≡

```
forall_edges(e,G)
{ node v = G.source(e);
  node w = G.target(e);
  if ( free[v] && free[w] )
  { free[v] = free[w] = false;
    G.rev_edge(e);
  }
}
```

n	m	FFB-	FFB+	FF-	FF+	Check
1000	2000	1.17	0.32	0.04	0.03	0
1000	4000	1.26	0.3	0.11	0.08	0.01
1000	8000	1.2	0.18	0.08	0.1	0.01
2000	4000	4.57	1.22	0.09	0.07	0
2000	8000	5.04	1.2	0.27	0.25	0.01
2000	16000	4.67	0.57	0.21	0.25	0.01
4000	8000	18.32	4.51	0.29	0.18	0.009998
4000	16000	20.57	4.82	0.97	0.51	0.02
4000	32000	18.47	2.09	0.64	0.7	0.04
8000	16000	72.05	18.1	0.67	0.46	0.04001
8000	32000	82	19.82	2.79	1.47	0.04999
8000	64000	74.05	7.63	1.78	1.54	0.07999

Table 7.3 The running times of four versions of the basic bipartite matching algorithm. FFB and FF refer to the two programs above, a minus sign indicates that no heuristic was used to find an initial matching and a plus sign indicates that the greedy heuristic was used. The last column shows the time required to check the results. The programs were run on random bipartite graphs with n nodes on each side and m edges (generated by *random_bigraph*(G, n, n, m, A, B)). FFB and FF use depth-first search to find augmenting paths. You may perform your own experiments by calling FF_matching_time in the demo directory.

The greedy heuristic is frequently highly effective. We support this statement by analysis and also by experimental evidence.

For the analysis we consider random graphs where $|A| = |B| = n$ and each node in A has d incident edges for some integer d. The edges go to random destinations, e.g., for each edge the endpoint in B is chosen uniformly at random from the nodes in B.

Let us consider the case $d = 1$ first. We consider the nodes in A one by one. When the node v is considered and its incident edge is $e = (v, w)$ we add e to the matching if w is free and we discard e if w is already matched. This shows that every node in B which has degree at least one will be matched by the greedy heuristic. The probability that a node w in B has degree zero is $(1 - 1/n)^n \approx e^{-1} \approx 0.37$ since the probability that the edge starting in any particular node in A does not end in w is $(n - 1)/n = 1 - 1/n$ and hence the probability that none of the n edges starting in a node in A ends in w is $(1 - 1/n)^n$. Thus about $(1 - e^{-1})n \approx 0.63n$ nodes will be matched by the greedy heuristic in the case $d = 1$. Of course, even more nodes will be matched on average for larger d. We give a plausibility argument of what to expect; the remainder of this paragraph is not rigorous.

Consider $d = 2$. About $e^{-1}n$ nodes in A will not be matched by only considering the first edge incident to any node. For these nodes the second incident edge will be considered and hence a total number of about $n + n/e$ edges will be considered. The probability that a node in B stays unmatched reduces to $(1 - 1/n)^{n+n/e} \approx e^{-(1+1/e)} \approx 0.25$.

We turn to experiments. Table 7.4 shows the effect and the cost of the greedy heuristic. We used the program below. The effect of the heuristic is as predicted by our analysis, i.e, for $m = n$ about 63% of the nodes are matched by the heuristic and for $m = 2n$ about 75% of the nodes are matched by the heuristic. The running time of the heuristic is insignificant. Even for the graphs with $m = 10n$ the running time of the heuristic is less than 10 times the time required to initialize the node array *free* and the time to check that all edges are directed from A to B.

⟨*mcb: effect of heuristic*⟩≡
```
double MCB_EFFECT_OF_HEURISTIC(graph& G,
                               const list<node>& A, const list<node>& B)
{ node v; edge e;
  node_array<bool> free(G,true);
  forall(v,B) assert(G.outdeg(v) == 0);
  if (use_heuristic == 0) return 0;
  ⟨MCB: greedy heuristic⟩
  int n = 0;
  forall(v,A) if (!free[v]) n++;
  return double(n)/A.size();
}
```

Table 7.3 shows the running time of four variants of our basic algorithm. The table indicates that both refinements have a tremendous impact on running time at least for random graphs. The greedy heuristic finds a large initial matching and hence saves many searches for augmenting paths and the refined implementation of the set of reached nodes keeps the cost of searching for augmenting paths low. Observe that the running time of both versions of FFB is quadratic in n. FFB+ (that is, FFB with greedy heuristic) has a smaller constant in the n^2 term in the running time since the expensive search for augmenting paths is only started from those nodes in A that are left free by the greedy heuristic. Also FFB+ runs faster for denser graphs since the matching found by the greedy heuristic is larger for denser graphs. FF is always much better than FFB and the time to check the output of our algorithms is negligible compared to the running times of the algorithms.

We summarize the findings of this section:

- The use of a heuristic to find a good initial solution can speed up graph algorithms tremendously. *We recommend exploring the use of a heuristic always.* The value of a heuristic is usually the highest for the least sophisticated algorithm.

- If graph exploration, e.g., a depth-first or a breadth-first search or a shortest-path computation, is used as a subroutine in a graph algorithm, the initialization of the data

n	m	No heuristic		Greedy heuristic	
		%	time	%	time
10000	10000	0	0.02	0.632	0.07
10000	20000	0	0.03	0.764	0.08
10000	30000	0	0.02	0.823	0.1
10000	40000	0	0.02	0.858	0.11
10000	50000	0	0.03	0.881	0.11
10000	60000	0	0.03	0.9	0.12
10000	70000	0	0.02	0.912	0.13
10000	80000	0	0.03	0.927	0.14
10000	90000	0	0.02	0.931	0.14
10000	100000	0	0.03	0.937	0.14

Table 7.4 Percentage of nodes matched by the greedy heuristic and cost of the greedy heuristic. The experiments were performed on random bigraphs with n nodes on each side and m edges (generated by *random_bigraph*(G, n, n, m, A, B)). You can perform your own experiments by calling mcb_effect_of_heuristic in the demo directory.

structures should be performed outside the subroutine. Only those parts of the data structure which are actually touched inside the subroutine should be reinitialized.

Breadth-First versus Depth-First Search: In the previous section we used depth-first search for finding augmenting paths. In this section we will investigate the use of breadth-first search. We will see that breadth-first search is more effective than depth-first search in finding augmenting paths.

Before we give the code we briefly argue that this should be the case. Assume that a is a free node in A, that the shortest augmenting path starting in a consists of k edges, and that the outdegree of all nodes in A is bounded by d. When breadth-first search from a is used in a search for an augmenting path then only nodes in distance at most $k+1$ from a are visited in the search. The number of such nodes is bounded by $d^{(k+1)/2}$. Observe that we have fan-out only at the nodes in A since nodes in B have at most outgoing edge. Actually, the stronger bound $d(d-1)^{(k+1)/2-1}$ holds since each of the nodes in A reachable from a must have one matching edge incident to it and hence there are only $d-1$ outgoing edges left. For example for $d = 3$ and $k = 9$ the number of nodes visited is bounded by $3 \cdot 2^4 = 48$.

How will depth-first search do? Well, it might explore a large fraction of the graph in the worst case. Even, if there is an augmenting path of length one, it might explore the entire graph.

We turn to the implementation of breadth-first search. Let a be any free node in A. We start a breadth-first search from a. We maintain a queue Q that contains all nodes in A reached by the search from which we have not yet explored the outgoing edges. Initially, Q contains only a. A node (in A or B) has been reached by the search iff $mark[v] ==$ *number_of_augmentations* and for a reached node v, $pred[v]$ contains the edge through which v was reached. When the procedure finds an augmenting path it augments the path and returns true, otherwise it returns false.

The procedure starts by putting a into the queue and marking a. As long as the queue is not empty, the first node is removed from Q. Call the node v; v is a node in A. We explore all edges out of v. Let $e = (v, w)$ be any such edge. If w has been reached before, we do nothing. Otherwise we set $pred[w]$ to e and mark w. If w is free, we augment by the path from a to w and return *true*. The path can be found by tracing edges as given by *pred*. If w is not free, let $f = (w, x)$ be the matching edge incident to w; note that f is the only edge out of w. We set $pred[x]$ to f, mark x, and append x to Q.

⟨*FF: bfs*⟩≡

```
#include <LEDA/queue.h>
static bool find_aug_path_by_bfs(graph& G, node a,
                           node_array<bool>& free, node_array<edge>& pred,
                           node_array<int>&  mark)
{ queue<node> Q;
  Q.append(a); mark[a] = number_of_augmentations;
  edge e;
  while ( !Q.empty() )
  { node v = Q.pop();  // v is a node in A
    forall_adj_edges(e,v)
    { node w = G.target(e); // w is a node in B
      if (mark[w] == number_of_augmentations) continue;
      // w has not been reached before in this search
      pred[w] = e; mark[w] = number_of_augmentations;
      if (free[w])
      { // augment path from a to w
        free[w] = free[a] = false;
        while ( w != a)
        { e = pred[w];
          w = G.source(e);
          G.rev_edge(e);
        }
        return true;
      }
      // w is not free
      edge f = G.first_adj_edge(w);
      node x = G.target(f);
      pred[x] = f; mark[x] = number_of_augmentations;
      Q.append(x);
    }
```

```
  }
  return false;
}
```

The matching algorithm is as we already know it. We use either breadth-first or depth-first search for finding augmenting paths. The choice is made by the variable *use_bfs*. In both methods we declare all nodes unreached (by increasing *number_of_augmentations*) whenever an augmenting path has been found.

⟨*_FF_matching*⟩≡

```
⟨FF: dfs⟩
⟨FF: bfs⟩
list<edge> MAX_CARD_BIPARTITE_MATCHING_FF(graph& G,
                              const list<node>& A, const list<node>& B,
                              node_array<bool>& NC,
                              bool use_heuristic, bool use_bfs)
{ node v; edge e;
  node_array<bool> free(G,true);
  node_array<int>  mark(G,-1);
  node_array<edge> pred(G);
  number_of_augmentations = 0;
  // check that all edges are directed from A to B
  forall(v,B) assert(G.outdeg(v) == 0);
  if (use_heuristic) ⟨MCB: greedy heuristic⟩
  forall(v,A)
  { if ( !free[v] ) continue;
    if (use_bfs)
    { if (find_aug_path_by_bfs(G,v,free,pred,mark) )
         number_of_augmentations++ ;
    }
    else
    { edge f;
      forall_adj_edges(f,v)
      { if (find_aug_path_by_dfs(G,f,free,mark))
        { free[v] = false;
          number_of_augmentations++ ;
          break;
        }
      }
    }
  }
  ⟨MCB: prepare result and node cover and restore orientations⟩
}
```

Table 7.5 shows the running time of the procedure above on random bipartite graphs. The table shows that breadth-first search is almost always superior to depth-first search (as we already argued above). It also shows that breadth-first search is not helped at all by the greedy heuristic. We explain this observation. The greedy heuristic considers augmenting

n	m	k	dfs-	dfs+	bfs-	bfs+
10000	15000	1	0.26	0.26	0.28	0.27
10000	15000	10	0.25	0.24	0.26	0.25
10000	15000	100	0.24	0.23	0.24	0.25
10000	15000	1000	0.24	0.23	0.24	0.25
10000	15000	10000	0.23	0.23	0.25	0.24
10000	25000	1	8.46	3.56	2.89	2.91
10000	25000	10	5.44	3.11	2.34	2.33
10000	25000	100	5.34	3.11	2.54	2.53
10000	25000	1000	2.04	2.19	1.92	1.92
10000	25000	10000	0.31	0.29	0.29	0.28
10000	35000	1	5.38	2.28	2.51	2.52
10000	35000	10	7.62	2.55	2.75	2.76
10000	35000	100	22.78	2.24	2.37	2.37
10000	35000	1000	17.91	2.21	2.09	2.09
10000	35000	10000	2.15	1.12	0.92	0.93

Table 7.5 Depth-first versus breadth-first search. The table shows the running time of MAX_CARD_BIPARTITE_MATCHING_FF. Either no heuristic (indicated by a minus sign) or the greedy heuristic (indicated by a plus sign) is used to find an initial matching. To complete the matching, a search for an augmenting path is started from each free node in A that was not matched by the heuristic. Either breadth-first or depth-first search is used to find an augmenting path. The programs were run on random bipartite group graphs with n nodes on each side and m edges (generated by *random_bigraph*(G, n, n, m, A, B, k)). The nodes on either side are divided into k groups and the nodes in the i-th group are connected to nodes in groups $i-1$ and $i+1$ on the other side. The generator is described in detail in Section 7.6.7. You may perform your own experiments by calling mcb_dfs_vs_bfs in the demo directory.

paths of length one. It finds an augmenting path of length one by inspecting all the edges incident to a node. Breadth-first search does exactly the same when an augmenting path of length one exists.

7.6.5 *The Algorithm of Hopcroft and Karp*

In this and the next section we give algorithms whose worst case running time is $O(\sqrt{n}m)$.

The first such algorithm is due to Hopcroft and Karp [HK73]. They suggested organizing the execution into phases, restricting augmentation to shortest augmenting paths, and augmenting a maximal number of node disjoint augmenting paths in each phase. Observe that Lemma 15 guarantees the existence of many short augmenting paths when the current matching is still far from optimality.

The overall structure of the program is the same as for our previous algorithms. The differences are that we maintain some additional data structures, in particular a list of the free nodes in A, and that the search for augmenting paths is organized differently.

⟨_HK_matching⟩≡

```
⟨HK: bfs⟩
⟨HK: dfs⟩
list<edge> MAX_CARD_BIPARTITE_MATCHING_HK(graph& G,
                              const list<node>& A, const list<node>& B,
                              node_array<bool>& NC, bool use_heuristic)
{ node v;
  edge e;
  node_array<bool> free(G,true);
  //check that all edges are directed from A to B
  forall(v,B) assert(G.outdeg(v) == 0);
  if (use_heuristic) { ⟨MCB: greedy heuristic⟩ }
  node_list free_in_A;
  forall(v,A) if (free[v]) free_in_A.append(v);
  ⟨HK: data structures⟩
  while ( ⟨there is an augmenting path⟩ )
  { ⟨find a maximal set and augment⟩ }
  ⟨MCB: prepare result and node cover and restore orientations⟩
}
```

We now give the details of how the Hopcroft and Karp algorithm searches for augmenting paths.

The length (= number of edges) of the shortest augmenting path can be found by breadth-first search. The search starts from all free nodes in A. We give a variant of breadth-first search which does a bit more. It constructs a so-called *layered network*. In a layered network the nodes of a graph are partitioned into *layers* according to their distance with respect to the starting layer, i.e., a node v belongs to layer k if there is a path from the starting layer to v consisting of k edges and there is no path with fewer edges. For any edge in a layered network the distance of the target node is at most one more than the distance of the source node. Only edges that connect different layers can be contained in shortest augmenting paths and hence we mark them *useful* in the program below; the mark is an integer *phase_number* in which we count the number of phases executed[21]. The construction of the layered network starts by putting all free nodes in A into the zeroth layer, then proceeds by standard breadth-first search, and stops as soon as the first layer is completed that contains free nodes in B. We achieve the latter goal by stopping to put nodes into the queue as soon as the first free node in B has been removed from the queue.

[21] Observe that we are reusing the marking technique introduced in section 7.6.4. Incrementing *phase_counter* will unmark all edges.

The program returns *true* if there is an augmenting path and returns *false* otherwise.

⟨*HK: data structures*⟩≡

```
edge_array<int> useful(G,0);
node_array<int> dist(G);
node_array<int> reached(G,0);
phase_number = 1;
```

and

⟨*HK: bfs*⟩≡

```
#include <LEDA/b_queue.h>
#include <LEDA/node_list.h>
static int phase_number;
static bool bfs(graph& G, const node_list& free_in_A,
                const node_array<bool>& free, edge_array<int>& useful,
                node_array<int>& dist, node_array<int>& reached)
{
  list<node> Q;
  node v,w;
  edge e;
  forall(v,free_in_A)
  { Q.append(v);
    dist[v] = 0;  reached[v] = phase_number;
  }
  bool augmenting_path_found = false;
  while (!Q.empty())
  { v = Q.pop();
    int dv = dist[v];
    forall_adj_edges(e,v)
    { w = target(e);
      if (reached[w] != phase_number )
      { dist[w] = dv + 1; reached[w] = phase_number;
        if (free[w]) augmenting_path_found = true;
        if (!augmenting_path_found) Q.append(w);
      }
      if (dist[w] == dv + 1) useful[e] = phase_number;
    }
  }
  return augmenting_path_found;
}
```

With this procedure we can refine the test for the existence of an augmenting path in the main loop.

⟨*there is an augmenting path*⟩≡

```
bfs(G,free_in_A,free,useful,dist,reached)
```

The layered graph contains all augmenting paths of shortest length. We determine a maximal set P of augmenting paths. Distinct paths in P will be node disjoint and P is maximal in the sense that no augmenting path can be added to P without violating the disjointness property. We find P by a variant of depth-first search. The procedure *find_aug_path*$(G, f, \mathit{free}, \mathit{pred}, \mathit{useful})$ attempts to find a path in the layered network starting with the edge f, ending in a free vertex in B, and being node-disjoint from all previously constructed paths. In the main loop we will call this procedure for all edges out of free nodes in A. The call returns the last edge on the path if it succeeds and returns *nil* otherwise. It also records, for each node, the first edge through which the node was reached in a *node_array<edge> pred*.

The details of *find_aug_path*$(G, f, \ldots)$ are simple. Let w be the endpoint of f. We set $\mathit{pred}[w]$ to f and then distinguish cases. If w is a free node (it is necessarily in B then), we return f. If w is not a free node, we scan through all edges $e = (w, z)$ out of w. If e does not belong to the layered network or we have already tried to construct a path out of z, we ignore e. Otherwise, we recurse. The recursive call either returns *nil* or a proper edge. In the latter case we know that a new augmenting path has been found and forward the edge to the enclosing call.

⟨*HK: dfs*⟩≡

```
static edge find_aug_path(graph& G, edge f, const node_array<bool>& free,
                      node_array<edge>& pred, const edge_array<int>& useful)
{ node w = G.target(f);
  pred[w] = f;
  if (free[w]) return f;
  edge e;
  forall_adj_edges(e,w)
  { node z = G.target(e);
    if ( pred[z] != nil || useful[e] != phase_number ) continue;
    edge g  = find_aug_path(G,e,free,pred,useful);
    if ( g ) return g;
  }
  return nil;
}
```

In the main loop we call *find_aug_path* for all edges out of free nodes in A that belong to the layered network and where the target node of the edge has not been reached by a previous search and collect the (terminal edges of the) paths found in a list *EL*. We then augment all paths. Let *e* be an arbitrary edge in *EL*. We trace the path ending in *e* by means of the *pred*-array and for each path reverse all edges on the path. We complete the phase by incrementing *phase_number*.

⟨*find a maximal set and augment*⟩≡

```
node_array<edge> pred(G,nil);
list<edge> EL;
forall(v,free_in_A)
{ forall_adj_edges(e,v)
    if (pred[G.target(e)] == nil && useful[e] == phase_number)
```

```
    { edge f = find_aug_path(G,e,free,pred,useful);
      if ( f )  { EL.append(f); break; }
    }
}
while (!EL.empty())
{ edge e = EL.pop();
  free[G.target(e)] = false;
  node z;
  while (e)
  { G.rev_edge(e);
    z = G.target(e);
    e = pred[z];
  }
  free[z] = false;
  free_in_A.del(z);
}
// prepare for next phase
phase_number++;
```

We close our discussion of the Hopcroft–Karp matching algorithm with a word on running time. Each phase of the algorithm takes time $O(m)$ for the breadth-first and depth-first search and the augmentation and hence the total running time is $O(Dm)$ where D is the number of phases. It can be shown (see for example [HK73] or [AMO93, section 8.2] or [Meh84, IV.9.2]) that the number of phases is $O(\sqrt{n})$. On many graphs the number of phases is much smaller. In particular, Motwani [Mot94] has shown that the number of phases is $O(\log n)$ for random graphs.

7.6.6 *The Algorithm of Alt, Blum, Mehlhorn, and Paul*

We discuss a variant of the Hopcroft–Karp algorithm due to Alt, Blum, Mehlhorn, and Paul [ABMP91]. It uses ideas first propagated for flow algorithms [AO89, GT88] to integrate the breadth-first and depth-first search used in the Hopcroft–Karp algorithm. The resulting algorithm is usually faster.

As above, we direct all edges in the current matching from B to A and all other edges from A to B. In this directed graph every path is an alternating path. For each node $v \in V$ we maintain a distance label *layer*$[v]$. Nodes in B will occupy even layers, and all free nodes in B will be in layer zero. Nodes in A will occupy odd layers, and all free nodes in A will be in two adjacent layers L and $L + 2$, for some L. Observe that this layering is "opposite" to the layering used in the Hopcroft and Karp algorithm. Now free nodes in B are in the bottom layer (= layer zero) and free nodes in A are in the two topmost layers (= layers L and $L+2$). Initially, we put all nodes in B into layer zero, all nodes in A into layer one, direct all edges from A to B, and set L to one.

⟨*ABMP: initialization*⟩≡

```
node_array<bool> free(G,true);
node_array<int> layer(G);
if (use_heuristic) {⟨MCB: greedy heuristic⟩}
```

```
list<node> free_in_A;
forall(v,B) layer[v] = 0;
forall(v,A)
{ layer[v] = 1;
  if (free[v]) free_in_A.append(v);
}
int L = 1;
```

In *free_in_A* we collect all free nodes in A. We maintain the invariant that the free nodes in level L precede the free nodes in level $L + 2$. In this way L is always the layer of the first node in *free_in_A*.

We maintain the "layered graph invariant" that no edge reaches downwards by two or more layers, i.e.,

$$\text{for all edges } e = (v, w)\text{: } \mathit{layer}[v] \le \mathit{layer}[w] + 1.$$

It follows that $\mathit{layer}[v]$ is a lower bound on the length of an alternating path starting in v and ending in a free node in B. Call an edge $e = (v, w)$ *eligible*, if $\mathit{layer}[v] = \mathit{layer}[w] + 1$, and let $ce(v)$ be a function which returns an eligible edge starting in v, if there is one, and *nil* otherwise. We call *ce* the current edge function. Its implementation will be discussed at the end of the section.

We search for augmenting paths as follows: starting from a free node v in layer L we construct a path p of eligible edges. Let w be the last node of p. There are three cases to distinguish:

Case 1 (breakthrough): w is a free node in layer zero:
Then p is an augmenting path with respect to the current matching. We augment the current matching by reversing all edges of p and terminate the search.

Case 2 (advance): w is not a free node in layer zero and $ce(w)$ exists:
We extend p by adding $ce(w)$.

Case 3 (retreat): w is not a free node in layer zero and $ce(w) = \mathit{nil}$:
We increase $\mathit{layer}[w]$ by two and remove the last edge from p. If there is no last edge in p, i.e., w is equal to the free node v from which we started the search for an augmenting path, we terminate the search and add w to the end of *free_in_A*. Observe that this maintains the invariant that the nodes on layer L precede the nodes on layer $L + 2$ in *free_in_A*.

The following program chunk realizes this strategy. The edges of the path are stored in a stack p of edges and w is the last node of the path. In the case of a breakthrough v and w are declared matched and all edges of p are reversed. In the case of an advance we push the current edge of w onto p and set w to the target node of the edge. In the case of a retreat we increase the layer of w by two and pop the last edge from p and set w to the source node of the edge popped. If there is no edge to be popped we terminate the search and add w to the rear end of *free_in_A*.

⟨*search for an augmenting path from v*⟩≡
```
node w = v;
while (true)
{ if ( free[w] && layer[w] == 0 )
  { // breakthrough
    free[w] = free[v] = false;
    while ( !p.empty() )
    {  e = p.pop();
       ⟨breakthrough: current edge function⟩
       G.rev_edge(e);
    }
    break;
  }
  else
  { if ( (e = ce(w,G,layer,cur_edge)) )
    { // advance
      p.push(e);
      w = G.target(e);
    }
  else
    { // retreat
      layer[w] += 2;
      ⟨relabel: current edge function⟩
      if (p.empty())
      { free_in_A.append(w);
        break;
      }
      w = G.source(p.pop());
    }
  }
}
```

After a breakthrough or a retreat, which leaves us with an empty path, we start the next search for an augmenting path. If there are no more free nodes in layer L, we increase L by two and repeat. In the program below this increase of L is implicit; L is simply the layer of the first node in *free_in_A*. In this way we proceed until L exceeds *Lmax* where *Lmax* is a parameter of the algorithm or until the number of free nodes is smaller than δL where δ is a parameter (which we set rather arbitrarily to 50 in our implementation). The parameter *Lmax* can either be set by the user or is set to $\gamma\sqrt{n}$ where γ is a parameter (which we set rather arbitrarily to 0.1 in our implementation). Once L exceeds *Lmax* or the number of free nodes in A has fallen below δL we determine the remaining augmenting paths by breadth-first search as in the Ford and Fulkerson algorithm.

⟨*_ABMP_matching*⟩≡
```
static int number_of_augmentations;
⟨FF: bfs⟩  // for the basic algorithm
edge ce(const node v, const graph& G,
        const node_array<int>& layer, node_array<edge>& cur_edge)
{ ⟨implementation of current edge function⟩ }
```

```
list<edge> MAX_CARD_BIPARTITE_MATCHING_ABMP(graph& G,
                            const list<node>& A, const list<node>& B,
                            node_array<bool>& NC,
                            bool use_heuristic, int Lmax)
{ node v; edge e;
  //check that all edges are directed from A to B
  forall(v,B) assert(G.outdeg(v) == 0);
  ⟨ABMP: initialization⟩
  node_array<edge> cur_edge(G,nil); // current edge iterator
  if (Lmax == -1) Lmax = (int)(0.1*sqrt(G.number_of_nodes()));
  b_stack<edge> p(G.number_of_nodes());
  while ( L <= Lmax && free_in_A.size() > 50 * L)
  { node v = free_in_A.pop();
    L = layer[v];
    ⟨search for an augmenting path from v⟩
  }
  ⟨complete by basic algorithm⟩
  ⟨MCB: prepare result and node cover and restore orientations⟩
}
```

where

⟨complete by basic algorithm⟩≡

```
node_array<int> mark(G,-1);
node_array<edge> pred(G);
number_of_augmentations = 0;
forall(v,free_in_A)
{ if ( find_aug_path_by_bfs(G,v,free,pred,mark) )
    number_of_augmentations++;
}
```

We establish correctness.

Lemma 17 *At all times during the execution of the algorithm, the following invariants hold:*

(I1) *For all edges (v, w): $layer[w] \geq layer[v] - 1$.*

(I2) *$layer[v]$ is even iff $v \in B$.*

(I3) *Let $p = [e_0, e_1, \ldots, e_{l-1}]$ with $e_i = (v_i, v_{i+1})$. Then p is a path in the current graph with $layer[v_i] = L - i$ for all i, $0 \leq i < l$, and v_0 is a free node in A.*

(I4) *All free nodes $v \in A$ are in layers L or $L + 2$.*

(I5) *The set M of edges that are directed from B to A forms a matching in G; furthermore $free[v]$ is true iff v is free with respect to M.*

Proof We use induction on the number of executions of the loop. All invariants hold initially. For the induction step we address the invariants in turn.

Only relabeling a node or reversing the direction of an edge may invalidate (I1). When a node v is relabeled there are no eligible edges out of v and hence $layer[w] \geq layer[v]$ for all $(v, w) \in E$. Since nodes in A live on odd layers and nodes in B live on even layers we even have $layer[w] > layer[v]$ for all $(v, w) \in E$. Hence increasing $layer[v]$ by two preserves (I1) for all edges $(v, w) \in E$. For edges $(w, v) \in E$ the invariant also stays true. Reversing the edges of the path p in the case of a breakthrough maintains (I1) as well, since all edges in p are eligible. Altogether, we have shown that (I1) is maintained.

Since layer labels are always increased by two, (I2) remains true.

The path p always starts at a free node in A in layer L and is only extended by eligible edges.

When a node is relabeled, it must be on the path p. Thus no free node in layer $L + 2$ can be relabeled by (I3). When L is increased by two, there is no free node v in layer L. Thus, (I4) is preserved.

In the case of a breakthrough, p is an alternating path from a free node $w \in A$ to a free node $v \in B$ by (I3) and the induction hypothesis, i.e., an augmenting path with respect to the current matching. Thus (I5) is preserved in the case of a breakthrough. □

The correctness of our algorithm is now established. Next we show that it is a derivative of the Hopcroft–Karp algorithm.

Lemma 18 *The algorithm always increases the matching along a shortest augmenting path.*

Proof Any augmenting path p found has length L. (I4) and (I5) imply that all free nodes in A are in layers L or $L + 2$, and those of B are in layer zero. Now the claim follows from (I1). □

Lemma 19 *Let M^* be a matching of maximum cardinality in G and M the matching computed by our algorithm when ⟨complete by basic algorithm⟩ is reached. Then $|M^*| - |M| \leq \max(\gamma Lmax, n/Lmax)$. Furthermore, ⟨complete by basic algorithm⟩ takes time $O(\max(\gamma Lmax, n/Lmax) \cdot m)$.*

Proof When ⟨*complete by basic algorithm*⟩ is reached then either $L > Lmax$ and there is no augmenting path with respect to the current matching M of length less than $Lmax$ or the number of free nodes in A is smaller than γL which in turn is smaller than $\gamma Lmax$. In the latter case we have established the claimed bound on $|M^*| - |M|$. In the former case we observe that $M^* \oplus M$ must contain $|M^*| - |M|$ node-disjoint augmenting paths with respect to M. The total length of these paths is at most n and each path has length at least $Lmax$. Thus $(|M^*| - |M|) \cdot Lmax \leq n$.

In ⟨*complete by basic algorithm*⟩ we need time $O(m)$ for each node in A which is still free. By the previous paragraph there are at most $\max(\gamma Lmax, n/Lmax)$ such nodes when the chunk is reached. □

The previous lemma suggests our choice of $Lmax$. In order to balance the contribution of the two choices we should set *Lmax* to $\Theta(\sqrt{n})$. Unfortunately, the theoretical analysis is not strong enough to suggest the "correct" factor of proportionality.

Lemma 20 *The total number of increases of layer labels and the total of number of calls to the eligible edge function ce is* $O(n \cdot Lmax)$.

Proof (I4) implies that the maximum layer of a node during an execution of the algorithm is $Lmax + 2$. Thus any node is relabeled at most $(Lmax + 2)/2$ times.

Each time the function *ce* returns an eligible edge (v, w), we extend the current path p by this edge. Either it still belongs to the path when p becomes augmenting for the next time, or $layer[w]$ is increased by two when (v, w) is deleted from p. Thus the number of calls to the function *ce* is bounded by the total number of increases of layer labels plus the total length of all augmenting paths. Since the length of an augmenting path is at most *Lmax*, because of (I4), and since there are at most n of them, the bound follows from the bound for the number of relabels. □

Lemma 20 implies that the total time spent outside ⟨*complete by basic algorithm*⟩ is $O(n \cdot Lmax)$ plus the time spent in calls to the current edge function. We now show how to implement the current edge function efficiently. We maintain for each node v an edge $cur\_edge[v]$ out of v such that all edges preceding $cur\_edge[v]$ in v's adjacency list are not eligible; when $cur\_edge[v]$ is *nil* all edges in v's adjacency list may be eligible. Recall that an edge (v, w) is eligible if the layer of w is one less than the layer of v and that no edge goes down more than one layer. Thus relabeling w cannot make (v, w) eligible and reversing an edge in an augmentation cannot make the edge eligible (because all edges in the augmenting path go from lower layers to higher layers after the augmentation). Only relabeling v can make an edge out of v eligible. With these observations it is easy to maintain the invariant that all edges preceding $cur\_edge[v]$ in v's adjacency list are not eligible:

When w is relabeled we set $cur\_edge[w]$ to nil.

When we search for a current edge we start searching at the current value of $cur\_edge[v]$ (at the first edge out of v if the current value is *nil*) until an eligible edge is found.

When an edge $e = (v, w)$ is reversed and e is the current value of $cur\_edge[v]$ we advance $cur\_edge[v]$ to the successor edge of v.

⟨*relabel: current edge function*⟩≡
```
cur_edge[w] = nil;
```

⟨*implementation of current edge function*⟩≡
```
edge e = cur_edge[v];
if ( e == nil ) e = G.first_adj_edge(v);
while (e && layer[G.target(e)] != layer[v] - 1) e = G.adj_succ(e);
cur_edge[v] = e;
return e;
```

⟨*breakthrough: current edge function*⟩≡
```
  if (e == cur_edge[G.source(e)])
    cur_edge[G.source(e)] = G.adj_succ(e);
```

In this way the time spent in calls $ce(v)$ between relabelings of v is $O(\text{number of calls} + outdeg(v))$. Since each node is relabeled at most *Lmax* times and since the total number of calls to *ce* is $O(n \cdot Lmax)$ we conclude that the total time spent in calls to the current edge function is $O(m \cdot Lmax)$.

We summarize in:

Theorem 3 *A maximum cardinality matching in a bipartite graph with n nodes and m edges can be computed in time $O(\sqrt{n}m)$.*

Proof This follows from the discussion above and the choice $Lmax = \Theta(\sqrt{n})$. □

7.6.7 *An Experimental Comparison*

We compare the algorithms *FF*, *HK*, and *ABMP* experimentally on bipartite graphs of the form shown in Figure 7.22. We call these graphs *bipartite group graphs*. They were suggested by [CGM+97].

The following program generates bipartite group graphs with *na* nodes in A and *nb* nodes in B. We divide both sides into $k + 1$ groups numbered 0 to k. For all i, $0 \le i \le k - 1$, the i-th group on side X contains nodes $i \cdot Kx$ to $(i + 1) \cdot Kx - 1$ where $Kx = \lfloor nx/k \rfloor$. The final group contains nodes $k \cdot Kx$ to $n - 1$; it is empty if k divides nx.

We generate the edges in two phases. In the first phase we generate $d = \lfloor m/na \rfloor$ edges for each node in groups 0 to $k - 1$ of A. For a node in the i-group the destination of these edges are random nodes in groups $i - 1 \bmod k$ and $i + 1 \bmod k$ of B. In the second phase we add $m - d \cdot k \cdot Ka$ random edges.

⟨*random_bigraph.c*⟩≡
```
  void random_bigraph(graph& G, int na, int nb, int m,
                      list<node>& A, list<node>& B, int k)
  {  G.clear();
     if ( na < 0 || nb < 0 || m < 0 )
       error_handler(1,"random_bigraph: one of na, nb, or m < 0");
     node* AV = new node[na];
     node* BV = new node[nb];

     A.clear();
     B.clear();

     int a, b;
     for(a = 0; a < na; a++)  A.append(AV[a] = G.new_node());
     for(b = 0; b < nb; b++)  B.append(BV[b] = G.new_node());

     if ( na == 0 || nb == 0 || m == 0 ) return;
     if ( k < 1) error_handler(1,"random_bigraph: k < 1");

     int  d = m/na;
     if (k > na) k = na; if (k > nb) k = nb;
```

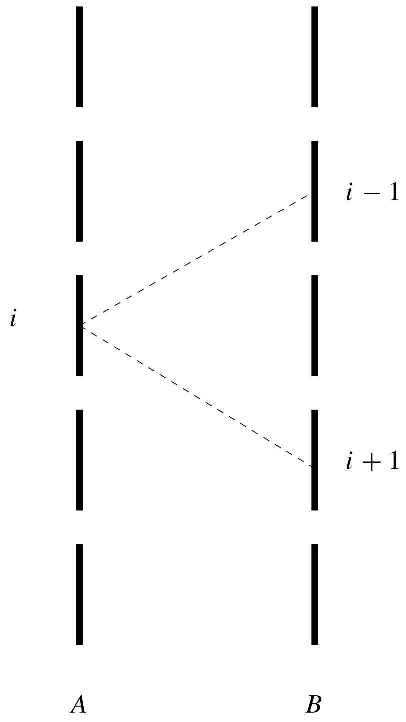

Figure 7.22 A bipartite graph with n nodes on each side. On each side the nodes are divided into k groups of size n/k each (this assumes that k divides n). Each node in A has degree $d = m/n$ and the edges out of a node in group i of A go to random nodes in groups $i+1$ and $i-1$ of B.

```
   int  Ka = na/k;    // group size in A
   int  Kb = nb/k;    // group size in B
   node v;
   int i;
   a = 0;
   forall(v,A)
   { int l = a/Ka;   // group of v
     if ( l == k) break;
     int base1 = (l == 0 ? (k-1)*Kb : (l-1)*Kb);
     int base2 = (l == k-1 ? 0 : (l+1)*Kb);
     for(i = 0; i < d; i++)
     { b = ( rand_int(0,1) == 0? base1 : base2 );
       G.new_edge(v,BV[b + rand_int(0,Kb-1)]);
     }
     a++;
   }
   int r = m - a*d;
   while (r--) G.new_edge(AV[rand_int(0,na-1)], BV[rand_int(0,nb-1)]);
   delete[] AV;
   delete[] BV;
}
```

n	m	k	FF-	FF+	HK-	HK+	AB-	AB+	Check
4	8	1	4.99	4.38	5.63	5.24	3.46	3.4	0.28
4	8	100	3.45	2.47	3.83	3.54	2.45	2.45	0.24
4	8	10000	1.11	1.04	3.76	3.51	2.16	2.16	0.22
4	12	1	155.7	50.02	8.37	7.95	4.91	4.95	0.36
4	12	100	69.07	44.09	5.94	5.78	3.19	3.1	0.26
4	12	10000	1.36	1.28	7.79	7.21	2.34	2.33	0.2599
4	16	1	42.75	21.34	9.71	9.16	4.95	5.33	0.43
4	16	100	48.75	41.59	6.99	6.57	3.02	3.37	0.29
4	16	10000	1.56	1.43	12.5	12.15	2.17	2.2	0.27
8	16	1	11.98	11.34	11.79	11.16	8.96	8.95	0.63
8	16	100	8.15	6.76	8.79	8.33	6.28	6.13	0.45
8	16	10000	2.33	2.15	7.83	7.29	5.42	5.44	0.46
8	24	1	611.6	188.6	19.49	18.56	12.28	12.35	0.77
8	24	100	349.8	221.4	13.14	12.69	8.33	8.36	0.54
8	24	10000	5.38	4.67	15.47	14.53	6.25	6.29	0.51
8	32	1	153.3	60.37	20.89	19.6	15.26	15.34	0.9099
8	32	100	247.1	208.2	13.9	13.22	9.73	9.76	0.6001
8	32	10000	13.58	12.46	26.38	25.96	6.75	6.71	0.5601

Table 7.6 The running times of the bipartite matching algorithms *FF*, *HK*, and *ABMP* on random bipartite group graphs with $n \cdot 10^4$ nodes on each side, $m \cdot 10^4$ edges and k groups (generated by *random_bigraph*(G, n, n, m, A, B, k)). The plus sign indicates the use of the greedy heuristic and the minus sign indicates that the algorithm started with the empty matching. The last column shows the time required to check the results. *FF* uses breadth-first search. You may perform your own experiments by calling mcb_matching_time in the demo directory.

Table 7.6 shows the outcome of our experiments. *FF* does very badly for some of the parameters and very well for others. It is always helped by the heuristic and frequently helped considerably. It shows the highest fluctuations of running time. *HK* and *ABMP* are more stable and *ABMP* is the fastest for most settings of the parameters. *HK* is always helped by the heuristic. For *ABMP* the effect of the heuristic is very small. If it is noticeable at all, it is negative. We have therefore chosen ABMP with the heuristic turned off as our default implementation. The time required for checking the result is negligible in all cases.

Exercises for 7.6

1 We described three methods to implement the principle of only paying for what is actually touched but gave the details of only two of them. Explore the third alternative. Rewrite MAX_BIPARTITE_CARD_MATCHING_FFB such that it uses a *node_slist* instead of *reached* and *reached_stack*.

2 In our implementations of matching algorithms we explicitly reverse the direction of matching edges by *rev_edge*. Explore the possibility of making the reversal only implicitly. Use a *node_array*<*edge*> *matching_edge* such that *matching_edge*[v] is *nil* if v is free and is the matching edge incident to v otherwise.

3 Rewrite the ABMP-implementation such that it uses depth-first search instead of breadth-first search in ⟨*complete by basic algorithm*⟩. Compare the running times.

4 Develop a strategy for choosing the parameter *Lmax* in the ABMP-algorithm (the authors have no good solution to this exercise).

5 Construct graphs where our maximum cardinality bipartite matching algorithms assume their worst case running time. Please inform the authors about your solution (as they can only partially solve this exercise).

7.7 Maximum Cardinality Matchings in General Graphs

A *matching* M in a graph G is a subset of the edges no two of which share an endpoint, see Figure 7.23. The cardinality $|M|$ of a matching M is the number of edges in M.

A node v is called *matched* with respect to a matching M if there is an edge in M incident to v and it is called *free* or *unmatched* otherwise. An edge e is called matching if $e \in M$. A matching is called *perfect* if all nodes of G are matched and is called *maximum* if it has maximum cardinality among all matchings.

The structure of this section is as follows. In Section 7.7.1 we discuss the functionality of our matching algorithms, in Section 7.7.2 we derive the so-called blossom shrinking algorithm for maximum matchings, and in Section 7.7.3 we give an implementation of it.

7.7.1 *Functionality*

The function

```
list<edge> MAX_CARD_MATCHING(const graph& G, int heur = 0)
```

returns a maximum matching in G. The underlying algorithm is the so-called blossom shrinking algorithm of Edmonds [Edm65b, Edm65a]. The worst case running time of the algorithm is $O(nm\alpha(m, n))$ ([Gab76]), the actual running time is usually much better. Table 7.7 contains some experimental data.

With *heur* $= 1$, the greedy heuristic is used to construct an initial matching which is then extended to a maximum matching by the blossom shrinking algorithm. As Table 7.7 shows, the influence of the greedy heuristic on the running time is small. It sometimes helps, it

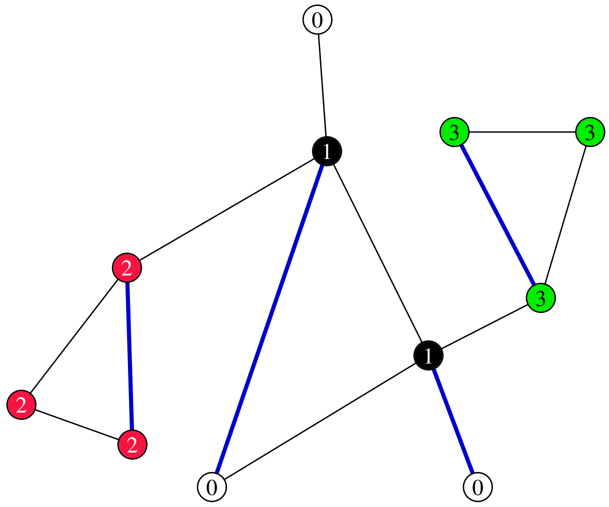

Figure 7.23 A maximum matching and a proof of optimality: The edges of the matching are shown in bold. The node labels prove the optimality of the matching. Observe that every edge is either incident to a node labeled 1 or connects two nodes that are labeled 2 or connects two nodes that are labeled 3. There are two nodes labeled 1, three nodes labeled 2, and three nodes labeled 3. Thus no matching can have more than $2 + \lfloor 3/2 \rfloor + \lfloor 3/2 \rfloor = 4$ edges. The matching shown has four edges and is hence optimal. You may generate similar figures with the xlman-demo gw_mc_matching.

sometimes harms, and it never causes a dramatic change. The cost of checking optimality is negligible in all cases.

In the remainder of this section we discuss the check of optimality. A labeling l of the nodes of G with non-negative integers is said to *cover* G (or to be a cover for G) if every edge of G (which is not a self-loop) is either incident to a node labeled 1 or connects two nodes labeled with the same i, for some $i \geq 2$. The *capacity* of l is defined as

$$cap(l) = n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor,$$

where n_i is the number of nodes labeled i. Observe that there may be nodes that are labeled zero. The capacity of a covering[22] is an upper bound on the cardinality of any matching.

Lemma 21 *If l covers G and M is any matching then $|M| \leq cap(l)$.*

Proof Since l covers every edge of G and hence every edge in M, each edge in M is either incident to a node labeled one or connects two nodes labeled i for some $i \geq 2$. There can be at most n_1 edges of the former kind and at most $\lfloor n_i/2 \rfloor$ edges of the second kind for any $i, i \geq 2$. Thus $|M| \leq cap(l)$. □

[22] In bipartite graphs only the labels zero and one are needed. The nodes labeled one form a node cover in the sense of Section 7.6.1.

n	m	MCM	MCM+	Check
10000	10000	0.287	0.223	0.024
20000	20000	0.905	0.717	0.074
40000	40000	2.178	1.758	0.184
80000	80000	4.857	3.934	0.413
10000	15000	1.049	1.03	0.027
20000	30000	3.799	3.862	0.102
40000	60000	11.45	11.9	0.262
80000	120000	30.51	33.57	0.583
10000	20000	1.247	1.304	0.04199
20000	40000	4.876	5.357	0.136
40000	80000	14.2	15.3	0.343
80000	160000	38.42	43.81	0.789
10000	25000	1.322	1.347	0.05099
20000	50000	4.761	4.782	0.169
40000	100000	13.95	14.22	0.422
80000	200000	35.2	37.3	0.959

Table 7.7 Running times of the general matching algorithm: The table shows the running time of the maximum cardinality matching algorithm without (MCM) and with the greedy heuristic (MCM+) and the time to check the result for random graphs with n nodes and m edges (generated by *random_graph*(G, n, m)). In all cases the time for checking the result is negligible compared to the time for computing the maximum matching. In each of the four blocks we used $n = 2^i \cdot 10^4$ for $i = 0, 1, 2, 3$ and a fixed relationship between n and m ($m/n = 1, 3/2, 2, 5/2$). The time to compute the maximum matching seems approximately to triple if n and m are doubled. Each entry is the average of ten runs. Except on the very sparse instances ($m \approx n$) it does not pay to use the greedy heuristic.

We will see in the next section that there is always a covering whose capacity is equal to the size of the maximum matching. The function

```
list<edge> MAX_CARD_MATCHING(const graph& G, node_array<int>& OSC,
                             int heur = 0)
```

returns a maximum matching M and a labeling OSC (OSC stands for odd set cover, a name to be explained in the next section) with:

- OSC covers G and

- $|M| = cap(OSC)$.

Thus *OSC* proves the optimality of M. Figure 7.23 shows an example. The additional running time for computing the proof of optimality is negligible.

The function

```
void CHECK_MAX_CARD_MATCHING(const graph& G, const list<edge>& M,
                             const node_array<int>& OSC)
```

checks whether *OSC* is a node labeling that covers G and whose capacity is equal to the cardinality of M. The function aborts if this is not the case. It runs in linear time.

The implementation of the checker is trivial. We determine for each i the number n_i of nodes with label i and then compute $S = n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor$. We assert that S is equal to the size of the matching.

We also check whether all edges are covered by the node labeling. Every edge must either be incident to a node labeled one or connect two nodes labeled i for some $i \geq 2$.

⟨*MCM: checker*⟩≡

```
static bool False(string s)
{ cerr << "CHECK_MAX_CARD_MATCHING: " << s << "\n";
  return false;
}
bool CHECK_MAX_CARD_MATCHING(const graph& G, const list<edge>& M,
                             const node_array<int>& OSC)
{ int n = Max(2,G.number_of_nodes());
  int K = 1;
  array<int> count(n);
  int i;
  for (i = 0; i < n; i++) count[i] = 0;
  node v; edge e;
  forall_nodes(v,G)
  { if ( OSC[v] < 0 || OSC[v] >= n )
      return False("negative label or label larger than n - 1");
    count[OSC[v]]++;
    if (OSC[v] > K) K = OSC[v];
  }
  int S = count[1];
  for (i = 2; i <= K; i++) S += count[i]/2;
  if ( S != M.length() )
    return False("OSC does not prove optimality");
  forall_edges(e,G)
  { node v = G.source(e); node w = G.target(e);
    if ( v == w || OSC[v] == 1 || OSC[w] == 1 ||
            ( OSC[v] == OSC[w] && OSC[v] >= 2) ) continue;
      return False("OSC is not a cover");
  }
  return true;
}
```

7.7.2 *The Blossom Shrinking Algorithm*

We derive the *blossom shrinking* algorithm of Edmonds [Edm65b, Edm65a] for maximum cardinality matching in non-bipartite graphs. In its original form the running time of the algorithm is $O(n^4)$. Gabow [Gab76] and Lawler [Law76] improved the running time to $O(n^3)$ and Gabow [Gab76] showed how to use the partition data structure of Section 5.5 to obtain a running time of $O(nm\alpha(m, n))$. Tarjan [Tar83] gave a very readable presentation of Edmond's algorithm and Gabow's improvement. Our presentation and our implementation is based on [Law76] and [Tar83].

The algorithm follows the general paradigm for matching algorithms: repeated augmentation by augmenting paths until a maximum matching is obtained. We assume familiarity with the paradigm, which can, for example, be obtained by reading Section 7.6.2. The natural way to search for an augmenting path starting in a node v is to grow a so-called *alternating tree* rooted at v.

The root of an alternating tree is a free node, the nodes on odd levels are reached by odd length alternating paths (and hence their incoming tree edge is a non-matching edge) and the nodes on even levels are reached by even length alternating paths (and hence their incoming tree edge is a matching edge). The root is even. All leaves in an alternating tree are even and odd nodes have exactly one child (namely their mate). Figure 7.24 shows an alternating tree. A node on an even level is called an *even* node and a node on an odd level is called an *odd* node. In the implementation an even node is labeled EVEN, an odd node is labeled ODD, and every node belonging to no alternating tree carries the label UNLABELED. This suggests calling a node *labeled* if it belongs to some alternating tree and calling it *unlabeled* otherwise.

We start the algorithm by making every free node the root of a trivial alternating tree (consisting only of the free node itself) and by labeling all free nodes even. We will maintain the following invariants:

- For each free node there is an alternating tree rooted at the free node.
- All nodes belonging to one of the alternating trees are labeled EVEN or ODD. Nodes on even levels are labeled EVEN and nodes on odd levels are labeled ODD.
- All nodes belonging to no alternating tree are unlabeled (= labeled UNLABELED).
- All unlabeled nodes are matched and if a node is unlabeled then its mate is also unlabeled.

An alternating tree is extended by exploring an edge $\{v, w\}$ incident to an even node v. It is a matter of implementation strategy which alternating tree is extended and which edge is chosen to extend it. There are four cases to be distinguished: w may be unlabeled, w may be odd, w may be even and in a different tree, and w may be even and in the same tree. The first three cases occur also in the bipartite case.

Case 1, w is unlabeled: We make w the child of v and the mate of w the child of w, see Figure 7.25. In this way, w becomes an odd node, its mate becomes an even node, and

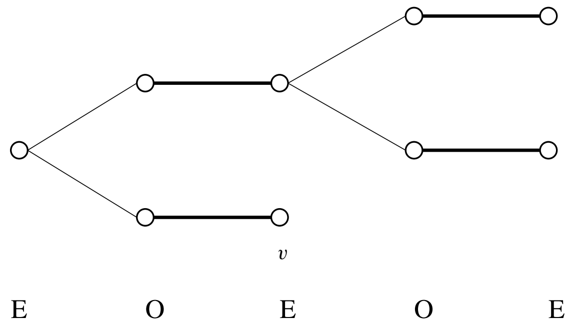

Figure 7.24 An alternating tree: It is rooted at a free node, nodes on odd levels (= odd nodes) are reached by odd length alternating paths, and nodes on even levels (= even nodes) are reached by even length alternating paths.

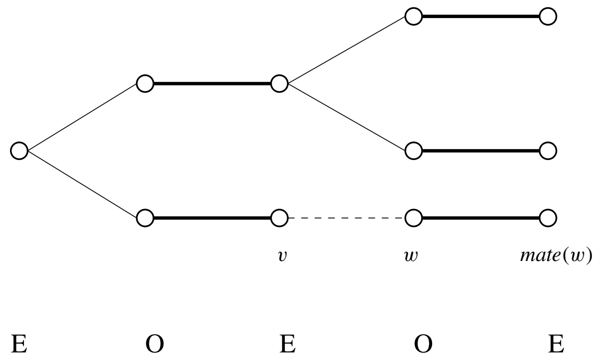

Figure 7.25 Growing an alternating tree: Exploration of the edge (v, w) turns w and its mate into labeled nodes, w becomes an odd node, and its mate becomes an even node.

both nodes become labeled. Observe that the growth action maintains the invariant that a matched node and its mate are either both labeled or both unlabeled.

Case 2, w is an odd node: We have discovered another odd length alternating path to w and do nothing.

Case 3, w is an even node in a different tree: We have discovered an augmenting path consisting of the edge $\{v, w\}$ and the tree paths from v and w to their respective roots, see Figure 7.26. We augment the matching by the augmenting path and unlabel all nodes in both trees. This makes all nodes in both trees matched (recall, that the root of an alternating tree is the only node in the tree that is unmatched) and destroys both trees. Observe that the remaining alternating trees, i.e., the ones whose roots are still free, are not affected by the augmentation. They are still augmenting trees with respect to the increased matching.

The three cases above also occur for bipartite graphs. The fourth and last case is new.

Case 4, w is an even node in the same tree as v: We have discovered a so-called *blossom*, see Figure 7.27. Let b be the lowest common ancestor of v and w, i.e., v and w

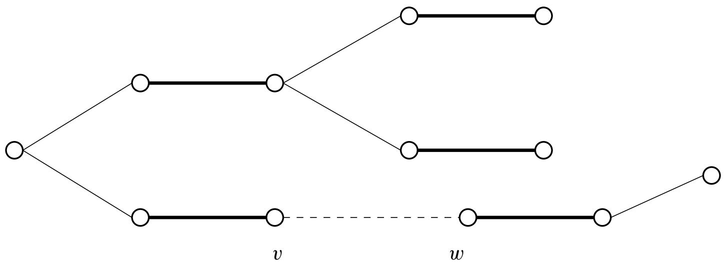

Figure 7.26 Discovery of an augmenting path: v and w are even nodes in distinct trees. The edge $\{v, w\}$ and the tree paths from v and w to their respective roots form an augmenting path.

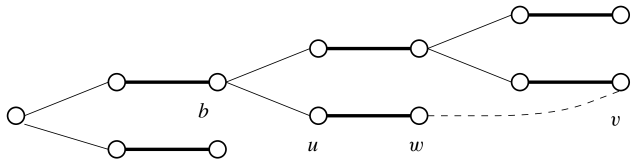

Figure 7.27 Discovery of a blossom: v and w are even nodes in the same tree. The node b is their lowest common ancestor. The blossom consists of the edge $\{v, w\}$ and the tree paths from b to v and w, respectively. The *stem* of the blossom consists of the tree path to b. The node b is the base of the blossom. The blossom consists of seven edges, three of which are matching. The even length alternating path to u follows the tree path to v, uses the edge $\{v, w\}$ and then proceeds down the tree to u.

are both descendants of b and there is no proper descendant of b with the same property. Since only even nodes can have more than one child, b is an even node. The blossom consists of the edge $\{v, w\}$ and the tree paths from b to v and w, respectively. The *stem* of the blossom consists of the tree path to b and b is called the *base* of the blossom. The stem is an even length alternating path ending in a matching edge; if the stem has length zero then b is free. The blossom is an odd length cycle of length $2k + 1$ containing k matching edges for some k, $k \geq 1$. All nodes in the blossom (except for the base) are reachable by an even and odd length alternating path from the root of the tree. For an even node u the even length path is simply the tree path to u and for an odd node u, say lying on the tree path from b to w, the even length path is the tree path to v followed by the edge $\{v, w\}$, followed by the path down the tree from w to u. For the odd length paths, the situation is reversed.

The action to take is to *shrink the blossom.* To shrink a blossom means to collapse all nodes of the blossom into the base of the blossom. This removes all edges from the graph which connect two nodes in the blossom and replaces any edge $\{u, z\}$ where u belongs to the blossom and z does not belong to the blossom by the edge $\{b, z\}$, see Figure 7.28. The node b is free after the shrinking iff it was free before the shrinking.

Lemma 22 *Let G' be obtained from G by shrinking a blossom with base b. If G' contains an augmenting path then so does G.*

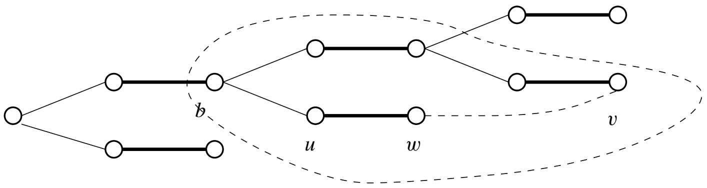

Figure 7.28 Shrinking a blossom: All nodes of the blossom are collapsed into the base of the blossom. After the shrinking, b stands for all the nodes enclosed by the dashed line.

Proof Suppose G' contains an augmenting path p. If p avoids b then p is an augmenting path in G and we are done. So let us assume that b lies on p. We break p at b into two pieces p_1 and p_2 and assume w.l.o.g that p_2 uses a non-matching edge e incident to b (in G'). The path p_1 is either empty (if b is free) or uses the matching edge incident to b. The edge $e = \{b, z\}$ in G' is induced by an edge $\{u, z\}$ in G where u is some node of the blossom. An augmenting path in G is obtained by first using p_1 then using the even length alternating path from b to u in the blossom, and then using p_2 (with its first edge replaced by $\{u, z\}$). □

We can now summarize the blossom shrinking algorithm. We grow alternating trees from the free nodes. Whenever a blossom is encountered it is shrunk. Whenever an augmenting path is discovered (this will in general happen after several shrinkings occurred), Lemma 22 is used to lift the augmenting path to the original graph. The matching is augmented by the augmenting path, the two trees involved are destroyed, all nodes in both trees are unlabeled, and the search for augmenting paths continues. The algorithm terminates when no alternating tree can be extended anymore. At this point the matching is maximum. Of course, this requires proof.

In order to show correctness we need the concept of an *odd-set cover*. It refines the notion of a covering introduced in Section 7.7.1.

For a subset N of an odd number of vertices of G we define the set of edges covered by N and the capacity of N as follows. If $|N| = 1$ then N covers all edges incident to the node in N and the capacity of N is equal to one. If $|N| = 2k + 1$ for some $k \geq 1$ then N covers all edges which have both endpoints in N and the capacity of N is k.

An *odd-set cover*[23] *OSC* of G is a family $\{N_1, \ldots, N_r\}$ of odd cardinality subsets of V such that each edge of G is covered by at least one of the sets in *OSC*. The capacity $c(OSC)$ of *OSC* is the sum of the capacities of the sets in *OSC*.

Lemma 23 *Let OSC be an odd-set cover in a graph G. Then the cardinality of any matching in G is at most* $c(OSC)$.

Proof Let M be any matching and let e be any edge in M. Then e must be covered by some

[23] An odd-set cover gives rise to an integer labeling of the nodes as follows: nodes that are contained in no set of the cover are labeled zero, nodes that are contained in a singleton set are labeled one, and nodes that are contained in an odd set of cardinality larger than one are labeled i for some $i > 1$. Distinct i's are used for distinct sets.

set in OSC. Moreover, the number of edges in M covered by any particular set in OSC is at most the capacity of the set. □

We are now ready for the correctness proof of the blossom shrinking algorithm. We will show that if the blossom shrinking algorithm does not find an augmenting path with respect to a matching M then there is an odd-set cover whose capacity is equal to the size of M, thus proving the optimality of M.

Let $G^{(0)} = G$ be our graph and let M be a matching in G. Suppose that the blossom shrinking algorithm does not discover an augmenting path. The blossom shrinking algorithm constructs a sequence $G^{(0)}, G^{(1)}, G^{(2)}, \ldots, G^{(h)}$ of graphs where for all i, $0 < i \leq h$, $G^{(i)}$ is obtained from $G^{(i-1)}$ by shrinking a blossom. Each node v of every $G^{(i)}$ stands for a set of nodes of G. In $G^{(0)}$ every node represents itself, and a node v in $G^{(i)}$ either stands for the same set as in $G^{(i-1)}$ or, if v is equal to the base node of the shrunken blossom, stands for all nodes represented by the nodes of $G^{(i-1)}$ collapsed into it.

Lemma 24 *For every i and every node v of $G^{(i)}$:*

- *v stands for an odd set of nodes in G,*
- *if v is odd or unlabeled then v stands for the singleton set consisting of v itself,*
- *if v stands for a set B of $2k + 1$ nodes in G for some $k \geq 1$ then the number of edges in M connecting nodes in B is equal to k.*

Proof The claim is certainly true for i equal to zero. When a blossom is shrunk an odd number of nodes is collapsed into a single node. By induction hypothesis each collapsed node represents an odd number of nodes of G. The sum of an odd number of odd numbers is odd.

The result of a shrinking operation is an even node. Thus odd and unlabeled nodes represent only themselves.

Consider a shrinking operation that collapses $2r + 1$ nodes into one. Out of these nodes, $r + 1$ were even before the shrinking (namely the base v and every even node on the two tree paths belonging to the blossom) and r were odd. Every odd node represents a single node of G and every even node stands for an odd set of nodes of G. Suppose that the i-th odd node represents a set B_i of $2k_i + 1$ nodes in G.

After the shrinking operation v stands for the r odd nodes and the union of the B_i's. Thus B consists of

$$r + \sum_{1 \leq i \leq r+1} (2k_i + 1) = 2(r + \sum_{1 \leq i \leq r+1} k_i) + 1$$

nodes and hence $k = r + \sum_{1 \leq i \leq r+1} k_i$. The number of edges in M running between nodes of B_i is k_i, and the number of edges of M belonging to the blossom is r. We conclude that k edges of M connect nodes in B. □

Consider now the graph $G^{(h)}$. In $G^{(h)}$ we have an alternating tree rooted at each free node and the tree growing process has come to a halt. Thus there cannot be an edge connecting two even nodes (because this would imply the existence of either an augmenting path or a blossom) and there cannot be an edge connecting an even node to an unlabeled node (as this would allow us to grow one of the alternating trees). Thus every edge either connects two nodes contained in the same blossom, or is incident to an odd node, or connects two unlabeled nodes. Every unlabeled node is matched to an unlabeled node (since a matched node and its mate are either both unlabeled or both matched) and hence the number of unlabeled nodes is even. We construct an odd-set cover *OSC* whose capacity is equal to M. *OSC* consists of:

- all odd nodes (interpreted as singleton sets),
- for each even node that stands for a set of cardinality at least three: the set represented by the node,
- no further set if there is no unlabeled node, a singleton set consisting of an arbitrary unlabeled node if there are exactly two unlabeled nodes, and a singleton set consisting of an arbitrary unlabeled node and a set consisting of the remaining unlabeled nodes if there are more than two unlabeled nodes.

Lemma 25 *The capacity of the odd-set cover OSC is equal to the cardinality of* M.

Proof The number of edges in M that still exist in $G^{(h)}$, i.e., have not been shrunken into a blossom in the course of the algorithm, is equal to the number of odd nodes plus half of the number of unlabeled nodes. For each even node v of $G^{(h)}$, representing a set B of $2r + 1$ nodes of G, the number of edges in M connecting nodes in B is equal to r by Lemma 24. This concludes the proof. □

Theorem 4 *The blossom shrinking algorithm is correct.*

Proof The algorithm terminates when it does not find an augmenting path. When this happens, there is, by Lemma 25, an odd-set cover whose capacity is equal to the size of M. Thus M is optimal. □

7.7.3 *The Implementation*

The goal of this section is to implement the blossom shrinking algorithm. Our implementation refines the implementation described in [Tar83] and is similar to the implementation given in [KP98]. The refinement does not change the worst case running time, but improves the best case running time from $\Omega(n^2)$ to $O(m)$. The observed behavior on random graphs with $m = O(n)$ seems to be much better than $O(n^2)$, see Table 7.7.

The overall structure of our implementation is given below. In the main loop we iterate over all nodes of G. Let $v_1, \ldots, v_n$ be an arbitrary ordering of the nodes of G. When $v = v_i$ is considered, every free node v_j with $j \geq i$ is the root of a trivial alternating tree, and the collection of alternating trees rooted at free nodes v_j with $j < i$ is *stable*. A collection $\mathcal{T}$

of alternating trees is stable if every edge $\{u, w\}$ incident to an even node u in $\mathcal{T}$ connects u to an odd node w in $\mathcal{T}$. In other words, every edge $\{u, w\}$ connecting a node u in $\mathcal{T}$ to a node outside $\mathcal{T}$ has u odd, and every edge connecting two nodes contained in $\mathcal{T}$ has at least one odd endpoint. It follows from our tree growing rules that the trees in $\mathcal{T}$ will not change in the future.

When $v = v_i$ is considered and v is already matched we do nothing. If v is still unmatched we grow the alternating tree T with root v until either an augmenting path is found or the growth comes to an end. We use a *node_list* Q to store all even nodes in T which have unexplored incident edges. We organize Q as a queue and hence grow the tree in breadth-first manner.

The growth process comes to an end when Q becomes empty. We claim that $\mathcal{T} \cup \{T\}$ is stable when Q becomes empty. Consider any edge $\{u, w\}$ with u an even node in T. Then w is odd, since otherwise the growth of T would not have come to an end. Moreover, w belongs to a tree in $\mathcal{T} \cup \{T\}$, since trees outside $\mathcal{T} \cup \{T\}$ are rooted at free nodes v_j, $j > i$, and consist only of a root and roots are even. Thus T can be added to our stable collection of alternating trees (this requires no action in the implementation) and the next free node can be considered.

When an augmenting path is found by exploring an edge $\{u, w\}$ with u an even node in T and w an even node in a tree different from T, w must be a free node v_j with $j > i$. Observe, that w cannot belong to T (since u and w are in distinct trees) and that w cannot belong to a tree in $\mathcal{T}$ (since $\mathcal{T}$ is stable). Thus w must belong to a tree rooted at some v_j, $j > i$, and hence must be equal to some v_j, $j > i$ (since the trees rooted at these nodes are trivial). When the matching is augmented by the augmenting path from v to w, all nodes in $T \cup w$ become matched and unlabeled. In order to be able to unlabel all nodes in $T \cup w$ in time proportional to the size of T we collect all nodes in T in a list of nodes (which we call T). We also set the variable *breakthrough* to *true* whenever an augmenting path is found in order to guarantee that we proceed to the next node in the main loop.

⟨*_mc_matching*⟩≡

```
enum LABEL {ODD, EVEN, UNLABELED};
⟨MCM: helpers⟩
list<edge> MAX_CARD_MATCHING(const graph& G,
                             node_array<int>& OSC, int heur)
{
  ⟨MCM: data structures⟩
  ⟨MCM: heuristics⟩
  node v; edge e;
  forall_nodes(v,G)
  { if ( mate[v] != nil ) continue;
    node_list Q; Q.append(v);
    list<node> T; T.append(v);
    bool breakthrough = false;
    while (!breakthrough && !Q.empty()) // grow tree rooted at v
    {
```

```
      node v = Q.pop();
      ⟨explore edges out of the even node v⟩
    }
  }
  list<edge> M;
  ⟨MCM: compute M⟩
  ⟨general checking: compute OSC⟩
  return M;
}
```

The Main Data Structures: We next discuss the main data structures used in the program. We use a *node_array<node> mate* to keep track of the current matching and we use a *node_partition base* to keep track of the blossoms.

⟨*MCM: data structures*⟩≡

```
node_array<node> mate(G,nil);
node_partition base(G);    // now base(v) = v for all nodes v
```

If two nodes v and w are matched then $mate[v] = w$ and $mate[w] = v$ and if a node v is free then $mate[v] = nil$. At the beginning, all nodes are free.

The node partition (see Section 6.8) *base* establishes the relationship between the current graph G' and the original graph G; recall that the current graph is obtained from the original graph by a sequence of shrinkings of blossoms, that a node partition partitions the nodes of a graph into disjoint sets called blocks, and that for a node v, $base(v)$ is the canonical representative of the block containing v. The relationship between G and G' is as follows:

- For any node v of G: if $base(v) = v$ then v is a node of G' and if $base(v) \neq v$ then v was collapsed into $base(v)$. Thus $\{base(v)\ ;\ v \in V\}$ is the set of nodes of G'.
- An edge $\{v, w\}$ represents the edge $\{base(v), base(w)\}$ of G'.

Every node is labeled as either EVEN, ODD, or UNLABELED. A node is labeled UNLABELED if it does not belong to any alternating tree and it is labeled EVEN or ODD otherwise. A node is labeled when it is added to an alternating tree. It retains its label when it is collapsed into another node. At the beginning all nodes are free and hence the root of an alternating tree. Thus all nodes are EVEN at the beginning. For an odd node v we use $pred[v]$ to store its parent node in the alternating tree. The pred value is set when a node is added to an alternating tree; it is not changed when the node is collapsed into another node.

⟨*MCM: data structures*⟩+≡

```
node_array<int>  label(G,EVEN);
node_array<node> pred(G,nil);
```

Figure 7.29 shows an example.

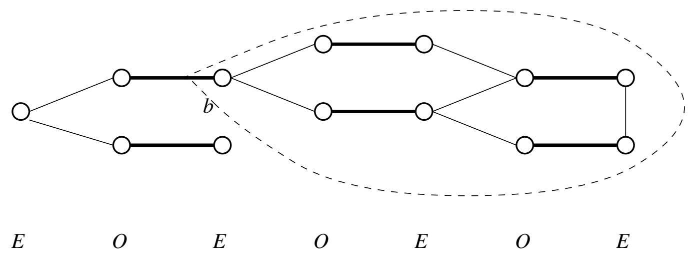

Figure 7.29 Snapshot of the data structure: The node labels are indicated by the labels "E" and "O". All nodes enclosed by the dashed line form a blossom and hence a block of the partition *base*. The canonical element of this block is b.

Exploring an Edge: Having defined most of the data structures we can give the details of exploring edges. Assume that v is an even node and let $e = \{v, w\}$ be an edge incident to v. Recall that e stands for the edge $\{base(v), base(w)\}$ in the current graph.

We do nothing if e is a self-loop or if $base(w)$ is ODD. If $base(w)$ is UNLABELED (this is equivalent to w being unlabeled) we grow the alternating tree containing v and if $base(w)$ is EVEN we have either discovered an augmenting path or a blossom.

⟨*explore edges out of the even node v*⟩≡

```
forall_inout_edges(e,v)
{ node w = G.opposite(v,e);
  if ( base(v) == base(w) || label[base(w)] == ODD )
     continue;   // do nothing
  if ( label[w] == UNLABELED )
    { ⟨grow tree⟩ }
  else  // base(w) is EVEN
    { ⟨augment or shrink blossom⟩ }
}
```

Growing the Tree: Let us first give the details of growing a tree. We label w as odd, make v the parent of w, label the mate of w as even, add the mate of w to Q, and add w and the mate of w to T.

⟨*grow tree*⟩≡

```
label[w] = ODD;             T.append(w);
pred[w] = v;
label[mate[w]] = EVEN;      T.append(mate[w]);
Q.append(mate[w]);
```

Discovery of a Blossom or an Augmenting Path: The node $base(w)$ is even. We have either found an augmenting path or a blossom. We have found an augmenting path if $base(v)$ and $base(w)$ belong to distinct trees and we have discovered a blossom if they belong to the same tree. We distinguish the two cases by tracing both tree paths in lock-step fashion until we either encounter a node that lies on both paths or reach both roots[24].

We discover a node lying on both paths as follows. We keep a counter *strue* which we increment in every execution of ⟨*augment or shrink blossom*⟩. Since there are at most n augmentations and at most n shrinkings between two augmentations the maximal value of the counter is bounded by n^2. It would therefore be unsafe to use type *int* for the counter, but type *double* is safe.

We use the counter as follows. As we trace the two tree paths we set *path1*[*hv*] to *strue* for all even nodes *hv* on the first path and *path2*[*hw*] to *strue* for all even nodes *hw* on the second path. The two paths meet iff *path1*[*hw*] or *path2*[*hv*] is equal to *strue* for some even *hw* on the second path or some even *hv* on the first path. The first node for which this is true is the base of the blossom. Recall that the base of a blossom is always even.

The cost of tracing the paths is proportional to the size of the blossom found, if a blossom is discovered, and is proportional to the length of the augmenting path found otherwise. Also observe that we define the arrays *path1* and *path2* outside the loop that searches for augmenting paths. Thus the cost for their initialization arises only once.

⟨*MCM: data structures*⟩+≡

```
  double strue = 0;
  node_array<double>  path1(G,0);
  node_array<double>  path2(G,0);
```

⟨*augment or shrink blossom*⟩≡

```
node hv = base(v);
node hw = base(w);
strue++;
path1[hv] = path2[hw] = strue;
while ((path1[hw] != strue && path2[hv] != strue) &&
       (mate[hv] != nil || mate[hw] != nil) )
{ if (mate[hv] != nil)
  { hv = base(pred[mate[hv]]);
    path1[hv] = strue;
  }
  if (mate[hw] != nil)
  { hw = base(pred[mate[hw]]);
    path2[hw] = strue;
  }
}
if (path1[hw] == strue || path2[hv] == strue)
  { ⟨shrink blossom⟩ }
else
  { ⟨augment path⟩ }
```

24 An alternative strategy is as follows: we have found an augmenting path if w is the root of a tree outside $\mathcal{T} \cup \{T\}$. We could, for each node, keep a bit to record this fact. The alternative simplifies the distinction between blossom shrinking and augmentations. However, it does not simplify the code overall, as all the information gathered in the program chunk ⟨*augment or shrink blossom*⟩ is needed in later steps of the algorithm.

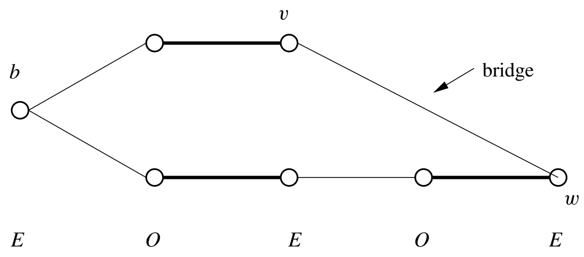

Figure 7.30 The bridge of a blossom: The edge $\{v, w\}$ closes a blossom with base b. For the odd nodes on the tree path from b to v we set *source_bridge* to v and *target_bridge* to w and for the odd nodes on the tree path from b to w we set *source_bridge* to w and *target_bridge* to v.

Shrinking a Blossom: Let us see how to shrink a blossom. The base b of the blossom[25] is either *hv* or *hw*. It is *hw* if *hw* also lies on the first path and it is *hv* otherwise. We shrink the blossom by shrinking the two paths that form the blossom.

The call *shrink_path*$(b, v, w, \ldots)$ collapses the path from v to b into b and the call *shrink_path*$(b, w, v, \ldots)$ collapses the path from w to b into b. Both calls also have the other end of the edge that closes the blossom as an argument.

⟨*shrink blossom*⟩≡

```
node b = (path1[hw] == strue) ? hw : hv;    // Base
shrink_path(b,v,w,base,mate,pred,source_bridge,target_bridge,Q);
shrink_path(b,w,v,base,mate,pred,source_bridge,target_bridge,Q);
```

Before we can give the details of the procedure *shrink_path* we need to introduce two more node labels. When an edge $\{v, w\}$ closes a blossom, all odd nodes in the blossom also get an even length alternating path to the root of their alternating tree. This path goes through the edge that closes the blossom. We call this edge the *bridge* of the blossom. The odd nodes on the tree path from v to b use the bridge in the direction from v to w and the odd nodes on the tree path from w to b use the bridge in the direction from w to v. We use the node arrays *source_bridge* and *target_bridge* to record for each odd node shrunken into a blossom the source node and the target node of its bridge (now viewed as a directed edge).

⟨*MCM: data structures*⟩+≡

```
node_array<node> source_bridge(G,nil);
node_array<node> target_bridge(G,nil);
```

The details of collapsing the tree path from v to b into b are now simple. For each node x on the path we perform *union_blocks*(x, b) to union the blocks containing x and b, for each odd node we set *source_bridge* to v and *target_bridge* to w, and we add all odd nodes to Q (because the edges out of the odd nodes now emanate from the even node b), see Figure 7.30.

[25] With the alternative case distinction between blossom shrinking and augmentation we would have to compute *hv* and *hw* at this point.

There is one subtle point. After a union operation the canonical element of the newly formed block is unspecified (it may be any element of the resulting block). It is important, however, that b stays the canonical element of the block containing it. We therefore explicitly make b the canonical element by *base.make_rep*(b).

$\langle$*MCM: helpers*$\rangle\equiv$

```
static void shrink_path(node b, node v, node w,
          node_partition& base, node_array<node>& mate,
          node_array<node>& pred, node_array<node>& source_bridge,
          node_array<node>& target_bridge, node_list& Q)
{ node x = base(v);
  while (x != b)
  {
    base.union_blocks(x,b);
    x = mate[x];
    base.union_blocks(x,b);
    base.make_rep(b);
    Q.append(x);
    source_bridge[x] = v;  target_bridge[x] = w;
    x = base(pred[x]);
  }
}
```

Augmentation: We treat the discovery of an augmenting path. The nodes v and w belong to distinct alternating trees with roots *hv* and *hw*, respectively. In fact, w is a root itself. The augmenting path consists of the edge $\{w, v\}$ plus the even length alternating path from v to its root *hv*.

For a node v let $p(v)$ be the even length alternating path from v to its root (if it exists). The path $p(v)$ can be defined inductively as follows:

If v is a root then $p(v)$ is the trivial path consisting solely of v.

If v is EVEN, $p(v)$ goes through the mate of v to the predecessor of the mate and then follows $p(\mathit{pred}[\mathit{mate}[v]])$.

If v is ODD, $p(v)$ consists of the alternating path from v to *source_bridge*$[v]$ concatenated with $p(\mathit{target\_bridge}[v])$.

Lemma 26 *The above characterization of* $p(v)$ *is correct.*

Proof The claim is certainly true when v is a root. So assume otherwise and consider the time when $p(v)$ is discovered in the course of the algorithm. For an even node this is the time when v is labeled EVEN and for an odd node this is the case when it becomes part of a blossom. In either case the characterization is correct. □

How can we find the alternating path from v to *source_bridge*$[v]$ when v is odd? The problem is that the *pred*-pointers are directed towards the roots of alternating trees and hence

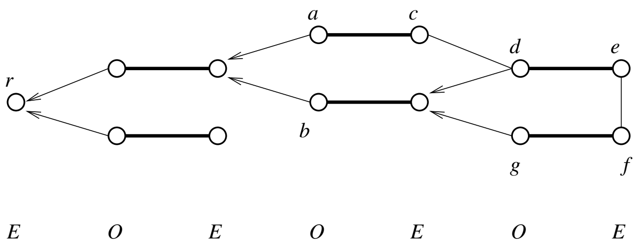

Figure 7.31 Tracing augmenting paths: The node labels are indicated by the labels "E" and "O". The predecessor pointer of the odd nodes are shown. When the bridge $\{e, f\}$ was explored we set *source_bridge*$[d]$ to e, *target_bridge*$[d]$ to f, *source_bridge*$[g]$ to f, and *target_bridge*$[g]$ to e, and when the bridge $\{c, d\}$ was explored we set *source_bridge*$[a]$ to c, *target_bridge*$[a]$ to d, *source_bridge*$[b]$ to d, and *target_bridge*$[b]$ to c.
The even length alternating path from b to its root r consists of the reversal of the path from $d =$ *source_bridge*$[b]$ to b followed by the even length alternating path from $c =$ *target_bridge*$[b]$ to r. The former path consists of the reversal of the alternating path from $e =$ *source_bridge*$[d]$ to d followed by the alternating path from $f =$ *target_bridge*$[d]$ to b.

there is no direct way to walk from v to *source_bridge*$[v]$. We walk from *source_bridge*$[v]$ to v instead and then take the reversal of the resulting path. The path from *source_bridge*$[v]$ to v is the prefix of $p($*source_bridge*$[v])$ ending in v, see Figure 7.31.

We cast this reasoning into a program by defining a procedure *find_path*$(P, x, y, \ldots)$ that takes two nodes x and y, such that y lies on $p(x)$ and such that the prefix of $p(x)$ ending in y has even length (the program would be slightly less elegant without the second assumption), and appends the prefix of $p(x)$ ending in y to the list P. *Find_path* distinguishes three cases:

If x is equal to y then the path consists of the single node x.

If $x \neq y$ and x is EVEN the path consists of x, *mate*$[x]$, followed by the path from *pred*$[$*mate*$[x]]$ to y.

If $x \neq y$ and x is ODD, let *P1* and *P2* be the paths from *target_bridge*$[x]$ to y and from *source_bridge*$[x]$ to *mate*$[x]$, respectively. Then path consists of x followed by the reversal of *P2* followed by *P1*.

⟨*MCM: helpers*⟩+≡
```
static void find_path(list<node>& P, node x, node y,
                 node_array<int>&  label, node_array<node>& pred,
                 node_array<node>& mate,
                 node_array<node>& source_bridge,
                 node_array<node>& target_bridge)
{ if ( x == y )
  {
    P.append(x);
    return;
  }
  if ( label[x] == EVEN )
  {
    P.append(x);
```

```
      P.append(mate[x]);
      find_path(P,pred[mate[x]],y,label,pred,mate,
                     source_bridge,target_bridge);
      return;
    }
    else // x is ODD
    {
      P.append(x);
      list<node> P2;
      find_path(P2,source_bridge[x],mate[x],label,pred,mate,
                               source_bridge,target_bridge);
      P2.reverse_items();
      P.conc(P2);
      find_path(P,target_bridge[x],y,label,pred,mate,
                        source_bridge,target_bridge);
      return;
    }
  }
```

Given *find_path*, it is trivial to construct the augmenting path. We construct the path from v to *hv* in P and append w to the front of the path. We augment the current matching by the path by walking along the path and changing *mate* accordingly.

It remains to prepare for the next search for an augmenting path. All nodes in $T \cup \{w\}$ are now matched. We unlabel all nodes in $T \cup \{w\}$ and split the blocks of *base* containing nodes of T. No action is required for the other alternating trees.

Finally, we set *breakthrough* to *true* and break from the forall-inout-edges loop. Setting *breakthrough* to *true* makes sure that we also leave the grow tree loop. The next action will therefore be to grow an alternating tree from the next free node.

⟨*augment path*⟩≡

```
list<node> P;
find_path(P,v,hv,label,pred,mate,source_bridge,target_bridge);
P.push(w);
while(! P.empty())
{ node a = P.pop();
  node b = P.pop();
  mate[a] = b;
  mate[b] = a;
}
T.append(w);
forall(v,T) label[v] = UNLABELED;
base.split(T);
breakthrough = true;
break;
```

Computing the Node Labeling *OSC***:** We compute the node labeling *OSC* as described in the paragraph preceding Lemma 25. We initialize *OSC*$[v]$ to -1 for all nodes v. This will allow us to recognize nodes without a proper *OSC*-label later. We then determine the number of unlabeled nodes (= nodes labeled *UNLABELED* and select an arbitrary unlabeled node. If there are unlabeled nodes, the selected unlabeled node is labeled one and all other unlabeled nodes are either labeled zero (if there are exactly two unlabeled nodes) or two (if there are more than two unlabeled nodes). We then set K to the smallest unused label larger than one.

Next we determine the number of sets of cardinality at least three and assign distinct labels to their representatives. We do so by iterating over all nodes. Every node v with $base(v) \neq v$ indicates a set of cardinality at least three. If its base is still unlabeled, we label it.

Finally, we label all other nodes. Nodes belonging to a set of cardinality at least two inherit the label of the base, and nodes that belong to sets of cardinality one (they satisfy `base(v) == v && OSC[base(v)] == -1`) are labeled one iff they are ODD and are labeled zero if they are EVEN.

⟨*general checking: compute OSC*⟩≡
```
forall_nodes(v,G) OSC[v] = -1;
int number_of_unlabeled = 0;
node arb_u_node;
forall_nodes(v,G)
 if ( label[v] == UNLABELED )
 { number_of_unlabeled++;
   arb_u_node = v;
 }
if ( number_of_unlabeled > 0 )
{ OSC[arb_u_node] = 1;
  int L = ( number_of_unlabeled == 2 ? 0 : 2 );
  forall_nodes(v,G)
    if ( label[v] == UNLABELED && v != arb_u_node ) OSC[v] = L;
}
int K = ( number_of_unlabeled <= 2 ? 2 : 3);
forall_nodes(v,G)
 if ( base(v) != v && OSC[base(v)] == -1 ) OSC[base(v)] = K++;
forall_nodes(v,G)
{ if ( base(v) == v && OSC[v] == -1 )
  { if ( label[v] == EVEN ) OSC[v] = 0;
    if ( label[v] == ODD  ) OSC[v] = 1;
  }
  if ( base(v) != v ) OSC[v] = OSC[base(v)];
}
```

Computing the List of Matching Edges: The list M of matching edges is readily constructed. We iterate over all edges. Whenever an edge is encountered whose endpoints are matched with each other, the edge is added to the matching. We also "unmate" the endpoints in order to avoid adding parallel edges to M.

⟨*MCM: compute M*⟩≡
```
forall_edges(e,G)
{ node v = source(e);
  node w = target(e);
  if ( v != w  &&  mate[v] == w )
  { M.append(e);
    mate[v] = v;
    mate[w] = w;
   }
 }
```

Heuristics: If $heur = 1$, the greedy heuristic is used to compute an initial matching. We iterate over all edges. If both endpoints of an edge are unmatched, we match the endpoints and declare both endpoints unlabeled. Recall that matched nodes that do not belong to an alternating tree are UNLABELED.

⟨*MCM: heuristics*⟩≡
```
switch (heur) {
case 0: break;
case 1: { edge e;
          forall_edges(e,G)
          { node v = G.source(e); node w = G.target(e);
            if ( v != w && mate[v] == nil && mate[w] == nil)
            { mate[v] = w; label[v] = UNLABELED;
              mate[w] = v; label[w] = UNLABELED;
            }
          }
          break;
        }
}
```

Summary: We summarize and complete the running time analysis. The algorithm computes a maximum matching in phases. In each phase an alternating tree T from a free node is grown to find an augmenting path. If the search for an augmenting path is successful, the matching is increased and all nodes in the alternating tree are unlabeled, and if the search is unsuccessful, the tree will stay around and will never be looked at again.

The running time of a phase is $O((n_T + m_T)\alpha(n_T, m_T))$, where n_T is the number of nodes included into T, m_T is the number of edges having at least one endpoint in T, and $\alpha(n, m_T)$ is the cost of m_T operations on a node partition of n nodes. This can be seen as follows. In a phase zero or more blossoms are shrunken. The search for a blossom (if successful) has cost proportional to the size of the blossom, and shrinking a blossom of size $2k + 1$ removes $2k$ nodes from the graph. Therefore the total size of all blossoms shrunk in a phase is $O(n_T)$. In each phase each edge is explored at most twice (once from each endpoint). Each exploration of an edge and each removal of a node involves a constant number of operations on the node partition *base*. We conclude that the total cost of a phase is $O((n_T + m_T)\alpha(n, m_T)) = O((n+m)\alpha(n, m)) = O(m\alpha(n, m))$, since $n_T \le n \le m$ and $m_T \le m$.

There are at most n phases and hence the total running time is $O(nm\alpha(n, m))$ in the worst case. One may hope that n_T is significantly smaller than n and m_T is significantly smaller than m for many phases. The running times reported in Section 7.7.1 show that the hope is justified in the case of random graphs. There are no analytical results concerning the average case behavior of general matching algorithms.

In an earlier implementation of the blossom shrinking algorithm we did not collect the nodes of the alternating tree grown into a set T. Rather, we iterated over all nodes at the beginning of a phase and labeled all free nodes EVEN and all matched nodes UNLABELED. With this implementation the running time is $\Omega(n^2)$. The implementation discussed in this section is significantly faster. It is superior for two reasons. Firstly, the cost of a phase is proportional to the size of the alternating tree grown in the phase and hence may be sublinear, and secondly, an alternating tree that does not lead to a breakthrough is not destroyed, but kept till the end of the execution.

Exercises for 7.7

1 Compare the running time of the general matching algorithm and the bipartite matching algorithm on bipartite graphs.

2 Exhibit a family of graphs where the running time of our matching algorithm is $\Omega(nm)$. Write a program to generate such graphs and provide it as an LEP.

7.8 Maximum Weight Bipartite Matching and the Assignment Problem

Throughout this section $G = (A \,\dot{\cup}\, B, E)$ denotes a bipartite graph and $c : E \mapsto I\!R$ denotes a *cost function* on the edges of G. We also say *weight* instead of cost. A *matching* M is a subset of E such that no two edges in M share an endpoint. The *cost of a matching* M is the sum of the cost of its edges, i.e.,

$$c(M) = \sum_{e \in M} c(e).$$

A node v is called *matched* with respect to a matching M if there is an edge in M incident to v and it is called *free* or *unmatched* otherwise. An edge e is called *matching* if $e \in M$. For a matched node v the unique node w connected to v by a matching edge is called the *mate* of v. A matching is called *perfect* or an *assignment* if all nodes of G are matched.

A matching is called:

- a *maximum weight matching* if its cost is at least as large as the cost of any other matching,
- a *maximum weight assignment* if it is a heaviest perfect matching,

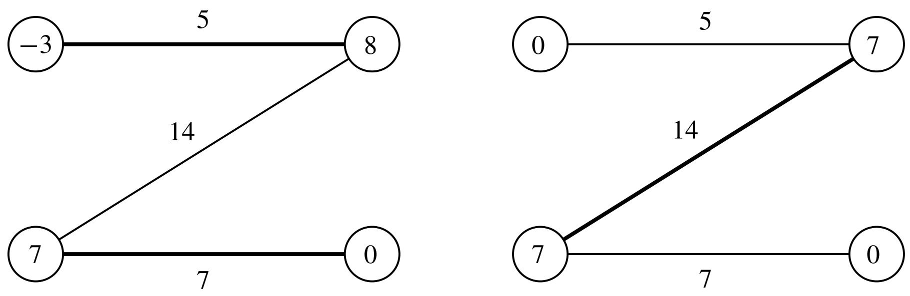

Figure 7.32 Maximum weight assignment and maximum weight matching. The matching on the left is a maximum weight perfect matching and the matching on the right is a maximum weight matching; the edges in the matchings are shown in bold in both cases. A potential function that proves the optimality of the matching is also given in both cases. The potential of each node and the cost of each edge is shown. For every edge the cost of the edge is bounded by the sum of the potentials of its endpoints. In an assignment every node is incident to exactly one edge of the assignment and hence the total cost of the assignment is bounded by the total potential. In the graph on the left the two quantities are equal and hence the assignment is optimal. In the graph on the right the potential function has the additional property that all potentials are non-negative and that all free nodes have potential zero. This implies (see Lemma 27) that the cost of any matching is bounded by the total potential. The two quantities are equal in the graph on the right and hence the matching is a maximum weight matching. The xlman-demo gw_mwb_matching allows the reader to experiment with weighted matchings in bipartite graphs.

- a *minimum weight assignment* if it is a lightest perfect matching,
- a *maximum weight maximum cardinality matching* if it is a heaviest matching among the matchings of maximum cardinality.

Figure 7.32 shows a a maximum weight assignment and a maximum weight matching. Clearly, a maximum or minimum weight assignment exists if and only if G contains a perfect matching.

In the next section we give the functionality of our algorithms and derive checkers of optimality. Sections 7.8.2 and 7.8.3 discuss an algorithm for maximum weight matchings and its implementation. In Sections 7.8.4 and 7.8.6 we modify our algorithms to compute assignments and maximum weight matchings of maximum cardinality. Finally, in Section 7.8.5 we show how to reduce the shortest path problem to the assignment problem.

7.8.1 *Functionality*

All functions in this section are function templates that work for an arbitrary number type *NT*. We use the convention that names of function templates for graph algorithms end with *_T*. In order to use the templates one must include <*LEDA/templates/mwb_matching.t*>. LEDA also contains pre-compiled instantiations for the number types *int* and *double*. The function names for the instantiated versions are *without* the suffix *_T*. In order to use the instantiated versions one must include <*LEDA/graph_alg.h*>. Section 7.1 discusses the relationship between templates and instantiated versions in more detail.

The function

```
list<edge> MAX_WEIGHT_BIPARTITE_MATCHING_T(graph& G,
                           const edge_array<NT>& c, node_array<NT>& pot)
```

returns a matching of maximal cost; the graph G is required to be bipartite. The worst case running time of the algorithm is $O(n \cdot (m + n \log n))$, the average case running time is much better. The function computes a proof of optimality in the form of the potential function *pot*. We discuss potential functions later in the section.

If a bipartition $V = A \dot\cup B$ is known and all edges are directed from A to B, the function

```
list<edge> MAX_WEIGHT_BIPARTITE_MATCHING_T(graph& G,
                           const list<node>& A, const list<node>& B,
                           const edge_array<NT>& c, node_array<NT>& pot)
```

can be used. If A and B have different sizes then it is advisable that A is the smaller set; in general, this leads to smaller running time.

The functions

```
list<edge> MAX_WEIGHT_ASSIGNMENT_T(graph& G,
                           const edge_array<NT>& c, node_array<NT>& pot);
list<edge> MIN_WEIGHT_ASSIGNMENT_T(graph& G,
                           const edge_array<NT>& c, node_array<NT>& pot);
```

return a maximum and minimum weight assignment, respectively. Both functions require that G is bipartite. If G does not contain a perfect matching the empty set of edges is returned.

All functions above are also available in the form where A and B are given as additional arguments and also without the argument *pot*.

The function

```
list<edge> MWMCB_MATCHING_T(graph& G,
                          const list<node>& A, const list<node>& B,
                          const edge_array<NT>& c, node_array<NT>& pot);
```

returns a maximum weight matching among the matchings of maximum cardinality. The potential function *pot* proves the optimality of the matching, see Section 7.8.6.

Potential Functions: We have mentioned the concept of a *potential function* several times already. It is time to define it. A function $\pi : V \mapsto \mathbb{R}$ is called a potential function. For an edge $e = (v, w)$ we call

$$\overline{c}(e) = \pi(v) + \pi(w) - c(e)$$

the *reduced cost* of e with respect to π. An edge is called *tight* iff its reduced cost is zero and the tight subgraph consists of all tight edges. For a subset U of the nodes we use $\pi(U)$ to denote $\sum_{v \in U} \pi(v)$. The following four properties of potential functions will play a role:

(1) Non-negativity of reduced costs, $\overline{c}(e) \geq 0$ for all $e \in E$.

(2) Tightness of matched edges, $\overline{c}(e) = 0$ for $e \in M$.

(3) Non-negativity of node potentials, $\pi(v) \geq 0$ for all $v \in V$.
(4) Tightness of free nodes, $\pi(v) = 0$ for all v that are free with respect to M.

The importance of potential functions stems from the following lemma.

Lemma 27 *Let M be any matching, let π be any potential function, and let F be the set of nodes that are free with respect to M.*

If all reduced costs are non-negative then $c(M) \leq \pi(V) - \pi(F)$. If, in addition, M is an assignment or all node potentials are non-negative then $c(M) \leq \pi(V)$.

If all reduced costs are non-negative and all matched edges have reduced cost zero then $c(M) = \pi(V) - \pi(F)$. If, in addition, M is an assignment or all free nodes have potential zero then $c(M) = \pi(V)$.

Proof If all reduced costs are non-negative then $c(e) \leq \pi(v) + \pi(w)$ for every edge $e = (v, w)$. Thus

$$\begin{aligned} c(M) &= \sum_{e \in M} c(e) \\ &\leq \sum_{e=(v,w) \in M} \pi(v) + \pi(w) \\ &= \sum_{v \in V;\, v \text{ is matched}} \pi(v) \;=\; \pi(V) - \pi(F), \end{aligned}$$

where the next to last equality follows from the fact that M is a matching and hence every matched node contributes exactly once to the sum on the second line and no free node contributes, and the last equality follows from the fact that the matched nodes are precisely the nodes that are not free. This establishes the first claim. For the third claim we observe that the inequality above becomes an equality if all matching edges have reduced cost zero.

The second and fourth claim follow from the first and third claim, respectively, and the additional observation that $\pi(F) \geq 0$ if node potentials are non-negative and that $\pi(F) = 0$ if M is an assignment or if the potential of all free nodes is zero. □

We call a potential function *feasible* if it satisfies (1), *non-negative* if it satisfies (3), and *tight* if it satisfies (1), (2), and (4). A tight non-negative potential function proves the optimality of a maximum weight matching and a tight potential function proves the optimality of a maximum weight assignment. Our algorithms return proofs of optimality in the form of tight potential functions.

The optimality conditions (1) to (4) are the basis for checkers of optimality. The function CHECK_MWBM_T takes a cost function c, a list of edges M, and a potential function *pot*, and checks that M is a matching and that the properties (1) to (4) above are satisfied.

⟨*mwb_matching.t*⟩+≡
```
bool False(const string s)
{ cerr << "CHECK_MWBM_T: " << s << "\n" << flush; return false;}
template <class NT>
```

```
bool CHECK_MWBM_T(const graph& G, const edge_array<NT>& c,
                  const list<edge>& M, const node_array<NT>& pot)
{ node v; edge e;
  // M is a matching
  node_array<int> deg_in_M(G,0);
  forall(e,M)
  { deg_in_M[G.source(e)]++;
    deg_in_M[G.target(e)]++;
  }
  forall_nodes(v,G)
    if ( deg_in_M[v] > 1) return False("M is not a matching");
  // node potentials are non-negative
  forall_nodes(v,G)
    if ( pot[v] < 0) return False("negative node potential");;
  // edges have non-negative reduced cost
  forall_edges(e,G)
  { node v = G.source(e); node w = G.target(e);
    if ( c[e] > pot[v] + pot[w])
      return False("negative reduced cost");
  }
  // edges in M have reduced cost equal to zero
  forall(e,M)
  { node v = G.source(e); node w = G.target(e);
    if ( c[e] != pot[v] + pot[w] )
     return False("non-tight matching edge");
  }
  // free nodes have potential equal to zero
  forall_nodes(v,G)
    if ( deg_in_M[v] == 0 && pot[v] != 0 )
      return False("free node with non-zero potential");
  return true;
}
```

The analogous functions

```
bool CHECK_MIN_WEIGHT_ASSIGNMENT_T(G,c,M,pot);
bool CHECK_MAX_WEIGHT_ASSIGNMENT_T(G,c,M,pot);
```

check minimum and maximum weight assignments, respectively. We do not give their implementations here. It is a good exercise to provide the implementations.

Potential Functions and Linear Programming Duality: We relate Lemma 27 to linear programming duality. Readers unfamiliar with linear programming may skip this material, although there is no harm in reading it anyway.

The maximum matching problem can be formulated as an integer program. We associate a variable $x(e)$ with every edge e, constrain it to the values 0 and 1, and consider the integer program

$$\max \sum_{e \in E} c(e)x(e)$$

subject to

$$\begin{aligned} \textstyle\sum_{e; e \text{ is incident to } v} x(e) &\le 1 && \text{for all } v \in V \\ x(e) &\in \{0, 1\} && \text{for all } e \in E. \end{aligned}$$

Let M be the set of edges e with $x(e) = 1$. The first constraint states that for each node v at most one of the incident edges belongs to M, i.e., it guarantees that M is a matching. The objective function states that we are looking for a matching of maximal weight. It was shown by Edmonds [Edm65b, Edm65a] that the integrality constraints $x(e) \in \{0, 1\}$ may be replaced by the linear constraints $x(e) \ge 0$ without changing the problem[26]. Assume that the integrality constraint $x(e) \in \{0, 1\}$ has been replaced by $x(e) \ge 0$. We now consider the dual linear program. The dual has one variable for each node and one constraint for each edge. We use $\pi(v)$ for the variable corresponding to node v and obtain

$$\min \sum_{v \in V} \pi(v)$$

subject to

$$\begin{aligned} c(e) &\le \pi(v) + \pi(w) && \text{for all } e = (v, w) \in E \\ \pi(v) &\ge 0 && \text{for all } v \in V. \end{aligned}$$

Linear programming duality states that the objective value of any feasible solution of the primal problem (= a matching) is no larger than the objective value of any feasible solution of the dual problem (= a potential function satisfying (1) and (3)) and that the value of the optimal solutions are equal. Complementary slackness implies in addition that the reduced cost of an edge in the matching must be zero and that the node potential of a free node must be zero. In fact, the proof of Lemma 27 is simply an adaption of the standard proofs of weak linear programming duality and complementary slackness to matchings.

26 We sketch a proof of this fact. We first observe that the non-negativity constraints $x(e) \ge 0$ together with the matching constraints $\sum_{e; e \text{ is incident to } v} x(e) \le 1$ guarantee $0 \le x(e) \le 1$. It therefore suffices to prove that the linear program has an optimal integral solution. The optimal solution to the linear program is given by a basic feasible solution, i.e., by the solution to a system $Bx = 1$ where B is a square submatrix of the constraint matrix and 1 is a vector of ones. Thus $x = B^{-1}1$. It therefore suffices to prove that B^{-1} is integral. By Cramer's rule, each entry of B^{-1} is the quotient of the determinant of a submatrix of B and the determinant of B. It therefore suffices to prove that the determinant of B is in $\{-1, 0, +1\}$. We prove more generally that the determinant of any square submatrix of the constraint matrix has determinant -1, 0, or $+1$, i.e., that the constraint matrix is a so-called *totally unimodular* matrix. Let B be any square submatrix. We need to compute the determinant of B. Each entry of B is either zero or one, each column of B corresponds to an edge of G, each row of B corresponds to a node of G, and each column contains at most two ones, one for each endpoint. As long as B contains a row or column with at most one one, we expand the determinant along this row or column. Each such reduction step reduces the dimension by one and yields a factor -1, 0, or $+1$. When no further reduction step applies, we have either reduced the dimension to zero and are done or reached a matrix B in which every row and column contains at least two ones. We will show that B is singular. Since a column contains at most two ones, we conclude that every column contains exactly two ones. Since B is square and since every row contains at least two ones we conclude that every row contains exactly two ones. In other words in the graph defined by B every node has degree two and thus the graph consists of a set of cycles. Each cycle has even length since G is bipartite (this is where we use the fact that G is bipartite). Let $v_1, v_2, \ldots, v_{2k}$ be any one of the cycles and consider the following linear combination formed by the rows corresponding to these nodes. Rows corresponding to nodes with odd index are multiplied by $+1$ and rows corresponding to nodes with even index are multiplied by -1. This linear combination yields the zero vector since, in each column corresponding to an edge of the cycle, one contribution is $+1$ and the other contribution is -1; this argument relies on the fact that the cycle has even length. Altogether we have now shown that the determinant of B is either -1, 0, or $+1$.

Arithmetic Demand: Special care should be taken when using the template functions with a number type *NT* that can incur rounding error, e.g., the type *double*. Section 7.2 contains a general discussion of this issue. The template functions are only guaranteed to perform correctly if all arithmetic performed is without rounding error. This is the case if all numerical values in the input are integers (albeit stored as a number of type *NT*) and if none of the intermediate results exceeds the maximal integer representable by the number type ($2^{53} - 1$ in the case of *doubles*). All intermediate results are sums and differences of input values, in particular, the algorithms do not use divisions and multiplications.

The algorithms have the following arithmetic demands. Let C be the maximal absolute value of any edge cost. If all weights are integral then all intermediate values are bounded by $3C$ in the case of maximum weight matchings and by $4nC$ in the case of the other matching algorithms. We will prove these bounds when we discuss the algorithms. For the sequel let $f = 3$ in the case of the maximum weight matchings and let $f = 4n$ in the other cases.

The pre-instantiations for number type *int* issue a warning if C is larger than $MAXINT/f$.

The pre-instantiations for number type *double* compute the optimal matching for a modified weight function *c1*, where for every edge e

$$c1[e] = sign(c[e]) \lfloor |c[e]| \cdot S \rfloor / S$$

and S is the largest power of two such that

$$S < 2^{53}/(f \cdot C).$$

The weight of the optimal matching for the modified weight function and the weight of the optimal matching for the original weight function differ by at most $n \cdot f \cdot C \cdot 2^{-52}$.

The weight modification can also be performed explicitly and we advise you to do so. The functions

```
bool MWBM_SCALE_WEIGHTS(const graph& G, edge_array<double>& c);
bool MWA_SCALE_WEIGHTS( const graph& G, edge_array<double>& c);
```

replace $c[e]$ by $c1[e]$ for every edge e, where $c1[e]$ was defined above and $f = 3$ for the first function and $f = 4n$ for the second function. The first scaling function is appropriate for the maximum weight matching algorithm and the second function is appropriate for all other matching algorithms. The functions return *false* if the scaling changed some weight, and return *true* otherwise.

7.8.2 *Maximum Weight Bipartite Matching: An Algorithm*

We describe an algorithm for maximum weight bipartite matching. The algorithm works iteratively. It starts with the empty matching and the graph spanned by B and the empty subset of A and then adds the nodes in A one by one. After each addition of a node it computes a new maximum weight matching and a new tight non-negative potential function.

Let $a_1, \ldots, a_n$ be an enumeration of the elements in A, let $A_i = \{a_1, \ldots, a_i\}$, let G_i be the subgraph spanned by $V_i = A_i \dot\cup B$, let M_i be a maximum weight matching in G_i, and let $\pi_i : V_i \mapsto I\!R_{\geq 0}$ be a non-negative potential function that is tight with respect to M_i. Our algorithm will construct M_i and π_i for $i = 0, 1, \ldots, n$. We assume that all matching edges are directed from B to A and all non-matching edges are directed from A to B.

M_0 and π_0 are trivial; M_0 is the empty matching and π_0 assigns zero to all nodes in B. Let us also construct M_1 and π_1. Let e be the heaviest edge incident to a_1. If e does not exist or has negative weight then M_1 is empty and π_1 assigns zero to all nodes in V_1. If e has non-negative weight then M_1 consists of e and π_1 assigns $c(e)$ to a_1 and zero to all nodes in B.

Assume now that we know M_{i-1} and π_{i-1} for some i, $i \geq 1$. We show how to construct M_i and π_i. An alternative interpretation of the construction will be given at the end of the section.

We start the construction of M_i and π_i by extending π_{i-1} to a feasible non-negative potential function $\overline{\pi}_i$ for V_i; this can be done by setting $\overline{\pi}_i(a_i)$ to any value that makes the reduced cost of all edges incident to a_i non-negative. Let $M = M_{i-1}$ and $\pi = \overline{\pi}_i$ and observe that M and π satisfy the optimality conditions (1), (2), and (3), and that $a = a_i$ is the only free node which violates (4). We now modify π (maintaining (1), (2), and (3), and (4) for all free nodes different from a) until there is an alternating path of tight edges from a either to a node a' in A having potential zero ($a = a'$ is possible) or to a free node in B. We set $M_i = M \oplus p$ and $\pi_i = \pi$. This re-establishes all four optimality conditions.

The potential function π is modified in phases. In each phase (except the last) we decrease $\pi(V_i)$ and we leave $\pi(V_i)$ unchanged in the last phase.

We now describe a phase. In each phase we determine the set R of nodes that are reachable from $a = a_i$ by tight edges and then distinguish three cases.

R contains a node in A of potential zero: Let v be a node in $A \cap R$ with $\pi(v) = 0$ and let p be a path of tight edges from a to v. We augment M by p, see Figure 7.33, and observe that π is tight with respect to $M \oplus p$. It is conceivable that $v = a$ and p is a path of length zero.

R contains a free node in B: Let w be a free node in $B \cap R$ and let p be a path of tight edges from a to w. We augment M by p, see Figure 7.34, and observe that π is tight with respect to $M \oplus p$.

Neither of the above: We define a value $\delta = \min(\alpha, \beta)$. Let α be the minimal value $\pi(v)$ for any node $v \in R \cap A$ and let β be the minimal value $\overline{c}(e)$ of any edge e leaving R. Then $\alpha > 0$ since R contains no node in A with potential zero and $\beta > 0$ since only non-tight edges can leave R, see Figure 7.35. We decrease the potential of all nodes in $R \cap A$ by δ, we increase the potential of all nodes in $R \cap B$ by δ, and recompute R. We continue in this fashion until one of the first two cases occurs.

The correctness of the method follows from the following lemma.

Lemma 28 *In the first two cases, π is tight with respect to $M \oplus p$. In the third case, the update of π preserves feasibility and non-negativity. The total potential decreases by δ. Moreover, all edges in M stay tight and a_i is the only free node whose potential can be*

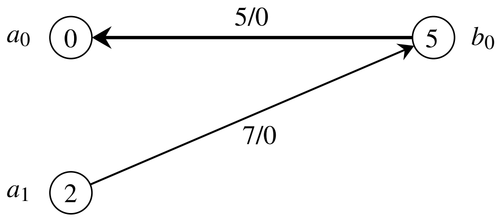

Figure 7.33 The edges of a matching M are shown in bold. The potential of each node is shown inside the node and the cost c and the reduced cost $\overline{c}$ of each edge is shown as $c(e)/\overline{c}(e)$. The path a_1, b_0, a_0 consists of tight edges and can be used for augmentation. The resulting matching has the same cardinality as the current matching.

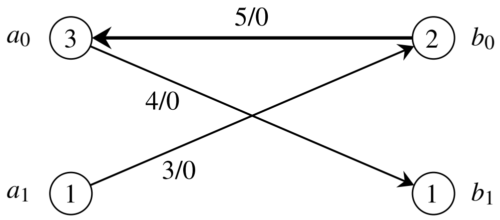

Figure 7.34 The edges of a matching M are shown in bold. The potential of each node is shown inside the node and the cost c and the reduced cost $\overline{c}$ of each edge is shown as $c(e)/\overline{c}(e)$. The path a_1, b_0, a_0, b_1 consists of tight edges and can be used for augmentation. The resulting matching has cardinality one larger than the current matching.

positive after the potential update. After the update there is either a node in $R \cap A$ whose potential is zero (if $\delta = \alpha$) or R grows (if $\delta = \beta$).

Proof In cases 1 and 2 we augment along a path of tight edges. Hence any edge in $M \oplus p$ is tight. Also, in case 1 we expose a node in A that has potential zero. Thus π is tight with respect to $M \oplus p$.

We turn to the third case. We start with a feasible potential function in which all edges in M are tight and in which all free nodes except for a_i have potential zero; a_i may or may not have potential zero. The set R contains one more node in A than in B since every node in R except a is matched and since for every matched edge either both endpoints or no endpoint is in R. Thus a potential update decreases the total potential by δ.

Let e be any edge. We show that the reduced cost of e stays non-negative. The reduced cost of e decreases only if one endpoint lies in $R \cap A$ and the other endpoint lies in $B \setminus R$. Then e is non-matching (since matching edges always have both or no endpoint in R) and hence the reduced cost of e before the potential update is at least β.

All edges in M stay tight since for any edge in M either both endpoints belong to R or neither endpoint does.

No free node in B can get a positive potential since there is no free node in $R \cap B$;

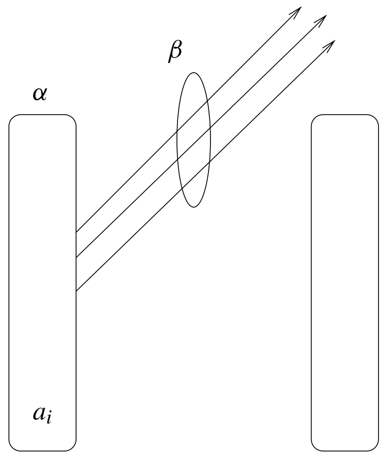

Figure 7.35 R is the set of nodes reachable from a_i by tight edges. The sets $A \cap R$ and $B \cap R$ are indicated as large ovals. The number of nodes in $A \cap R$ is one larger than the number of nodes in $B \cap R$, since each node in $A \cap R \setminus a_i$ is matched to a node in $B \cap R$ and vice versa. β is the minimum reduced cost of any edge leaving R (any such edge has its source node in A since all edges out of B are in M and hence tight) and α is the minimum potential of any node in $A \cap R$, and $\delta = \min(\alpha, \beta)$. We reduce the potential of all nodes in $A \cap R$ by δ and increase the potential of all nodes in $B \cap R$ by δ.

otherwise we would be in case 2. The potential of nodes in A does not increase and hence a_i can stay the only node with positive potential. □

At this point we have arrived at a first version of our algorithm.

M = the empty matching;
$pot(b) = 0$ for all b in B;

forall $a \in A$
{ set $pot(a)$ to some value that makes the reduced cost of all edges incident to a non-negative;

  while (true)
  { determine the set R of nodes reachable from a by tight edges.

    if R contains a node in A with potential zero or a free node in B
    { augment by a path of tight edges from a to this node;
      break;
    }
    compute α, β, and δ and adjust the potentials;
  }
}

We leave it to the reader to implement the basic version of the algorithm.

Let us take a closer look at the inner loop of this algorithm. It grows a set R until R contains either a free node in B or a node in A with potential zero. Let R_k be the set R in the k-th iteration of the loop for $k = 1, 2, \ldots, K + 1$, let δ_k be the value of δ determined in the k-th iteration, let $\Delta_k = \delta_1 + \ldots + \delta_k$, and let $\Delta = \Delta_K$ be the sum of all δ's. Then the total change of potential of the nodes in $R_k \setminus R_{k-1}$ is $\delta_k + \delta_{k+1} + \ldots = \Delta - (\delta_1 + \ldots + \delta_{k-1})$.

Also $\delta_k = \beta_k < \alpha_k$ for $k < K$ since $\delta_i = \alpha_i$ implies that case 1 occurs in the next iteration and hence $i = K$. Finally, $\delta_K = \alpha_K$ implies that $R_{K+1} = R_K$. We relate the growth of R to a shortest-path computation with source a_i.

Lemma 29 *Let w be any node and let $\mu(w)$ be the shortest-path distance of w from a_i with respect to the reduced costs defined by $\overline{\pi}_i$. Then w is added to R after a total potential change of $\mu(w)$.*

Proof Let w be any node and consider a shortest path p from a_i to w. Let $e_1, e_2, \ldots$ be the edges on p that are not tight initially in the order in which they occur on p. The source node of e_1 belongs to R_1. The reduced cost of e_1 is decreased by δ_1 in the first phase, by δ_2 in the second phase, and becomes zero at the end of some phase, say the $(l - 1)$-th, i.e., the original reduced cost of e_1 was equal to $\delta_1 + \ldots + \delta_{l-1}$. In phase l the source node of e_2 belongs to R and the next potential updates reduce the cost of e_2 to zero. In this way w is added to R after a total potential change of $\mu(w)$. $\square$

Lemma 30 *Let $\pi = \overline{\pi}_i$ and for any node w let $\mu(w)$ be the shortest-path distance of w from a_i with respect to the reduced costs defined by π. Let minA $= \min\{\mu(a) + \pi(a)\ ;\ a \in A\}$ and let minB $= \min\{\mu(b)\ ;\ b \in B \text{ and } b \text{ is free}\}$. Then $\Delta = \min(\mathit{minA}, \mathit{minB})$ and the total potential change for any node v is equal to $\max(0, \Delta - \mu(v))$.*

If $\Delta =$ minA, let z be the node that defines minA and if $\Delta =$ minB, let z be the node that defines minB (if minA $=$ minB, define z by either half-sentence). In either case let p be a path of length $\mu(z)$ from a_i to z. Then all edges of p are tight after the change of potential.

Proof Consider an arbitrary node $a \in A$. It is added to R when the total potential change is equal to $\mu(a)$. Subsequent potential changes decrease $\pi(a)$ and hence the total potential change cannot be more than $\mu(a) + \pi(a)$ (since node potentials always stay non-negative).

Consider a free node $b \in B$. It is added to R when the total potential change is equal to $\mu(b)$. Thus the total potential change cannot be more than $\mu(b)$.

We stop changing the potentials once a node in $A \cap R$ reaches potential zero or a free node in B is added to R. Thus the total potential change Δ is equal to $\min(\mathit{minA}, \mathit{minB})$.

A node v participates in potential changes after it has been added to R. Thus the total change of potential of v is equal to $\max(0, \Delta - \mu(v))$.

Let p be as defined in the statement of the lemma and let $e = (v, w)$ be any edge of p. Then $\mu(v) + \overline{c}(e) = \mu(w)$ since p is a shortest path. Also $\mu(v), \mu(w) \leq \Delta$ since p is a shortest path to the node that defines Δ. We show that e is tight after the potential change.

If e is matching and hence $v \in B$ and $w \in A$ and $\overline{c}(e) = 0$, we have $\mu(v) = \mu(w)$. Thus $\pi(v)$ is increased by $\Delta - \mu(v)$ and $\pi(w)$ is decreased by the same amount. Thus e stays tight.

If e is non-matching and hence $v \in A$ and $w \in B$, we have $\mu(v) + \overline{c}(e) = \mu(w)$. Thus $\pi(w)$ is increased by $\Delta - \mu(w) = \Delta - \mu(v) - \overline{c}(e)$ and $\pi(v)$ is decreased by $\Delta - \mu(v)$. The reduced cost of e is therefore reduced by $\overline{c}(e)$ and hence e becomes tight. □

Lemmas 29 and 30 allow us to refine our basic algorithm.

M = the empty matching;
pot(b) = 0 for all b in B;

forall $a \in A$
{ set *pot*(a) to some value that makes the reduced cost of all edges incident to a non-negative;

for any node v let *dist*(v) be the shortest-path distance of v from a and let

minA = min {*dist*(v) + *pot*(v) ; $v \in A$};
best_node_in_A = a node in A that defines *minA*;

minB = min {*dist*(v) ; $v \in B$ and free};
best_node_in_B = a node in B that defines *minB*;

Delta = min(*minA*, *minB*);

forall $v \in A$: *pot*(v) = *pot*(v) − max(0, *Delta* − *dist*(v));
forall $v \in B$: *pot*(v) = *pot*(v) + max(0, *Delta* − *dist*(v));

augment by the alternating path of tight edges from a to *best_node_in_A*, if *Delta* = *minA*, and from a to *best_node_in_B*, otherwise;
}

The description above suggests that it is necessary to compute *dist*[v] for all nodes v in each execution of the inner loop. This is not true. It is only necessary to compute *Delta* and the node defining it and to compute *dist*[v] for all nodes v with *dist*[v] < *Delta*. Given this information all potentials can be updated correctly and the augmentation can be made.

How can we compute *Delta* without computing *dist*[v] for all nodes v? We exploit the fact that Dijkstra's algorithm computes dist-values in increasing order. Let $v_0, v_1, \ldots$ with $v_0 = a$ be the order in which the nodes are reached by the shortest-path computation. Then $dist[v_0] \le dist[v_1] \le \ldots$ We observe:

(1) If

$$\min\{dist[v_i] + pot[v_i]\,;\, i < k \text{ and } v_i \in A\} \le dist[v_k]$$

then some v_i with $i < k$ and $v_i \in A$ defines *minA*. This follows from the fact that all node potentials are non-negative.

(2) *minB* is the dist-value of the first free node in B that is reached by the shortest-path computation. If no v_j with $j < k$ is a free node in B then $dist[v_k] \le minB$.

(3) If

$$\min\{dist[v_i] + pot[v_i]\,;\, i < k \text{ and } v_i \in A\} \le dist[v_k]$$

and no v_j with $j < k$ is a free node in B then $Delta = minA$. This follows from (1) and (2).

(4) If

$$\min\{dist[v_i] + pot[v_i]\,;\, i < k \text{ and } v_i \in A\} \ge dist[v_k]$$

and v_k is a free node in B then $Delta = minB$. This follows from (1) and (2).

(5) Let k be minimal such that either (3) or (4) holds. Then $Delta \le dist[v_j]$ for all $j > k$ and the potentials of all nodes v_j with $j > k$ are not changed.

We will use items (3) and (4) as the stopping criteria for the shortest-path computation in our implementation. Item (5) implies that only nodes that are reached by the shortest-path computation can be affected by the potential change.

7.8.3 *Maximum Weight Bipartite Matching: An Implementation*

After all this preparatory work we are ready for the implementation.

We start by declaring the data structures required by the algorithm, then use one of three heuristics to initialize the potential function and the matching, then call *augment*$(a, \ldots)$ for each node in A that is left unmatched by the heuristic, and finally restore the graph and prepare the list of edges comprising the matching.

The data structures used by the algorithm are two boolean arrays to keep track of the free nodes and the nodes in A and the data structures needed for the shortest-path computations (arrays *pred* and *dist*, and a node priority queue *PQ*).

We describe three heuristics. The simplest heuristic (called naive in the program below) sets the potential of all nodes in B equal to zero, the potential of all nodes in A equal to the maximal cost of all edges, and sets the matching to the empty matching. The other heuristics are described later in the section.

⟨*mwb_matching.t*⟩+≡

```
⟨mwb_matching: helpers⟩
static int which_heuristic = 2;
template <class NT>
list<edge> MAX_WEIGHT_BIPARTITE_MATCHING_T(graph& G,
                           const list<node>& A, const list<node>& B,
                           const edge_array<NT>& c, node_array<NT>& pot)
{ node a,b,v; edge e;
  list<edge> result;
  forall_nodes(v,G) pot[v] = 0;
  if (G.number_of_edges() == 0 ) return result;
  // check that all edges are directed from A to B
  forall(b,B) assert(G.outdeg(b) == 0);
  node_array<bool> free(G,true);
```

```
  node_array<edge> pred(G,nil);
  node_array<NT>   dist(G,0);
  node_pq<NT>      PQ(G);
  switch (which_heuristic)
  { case 0: { // naive heuristic
              NT C = 0;
              forall_edges(e,G) if (c[e] > C) C = c[e];
              forall(a,A)       pot[a] = C;
              break;
            }
    case 1: { // simple heuristic
              ⟨simple heuristic⟩
              break;
            }
    default: { // refined heuristic
              mwbm_heuristic( G, A, c, pot, free);
              break;
            }
  }
  forall(a,A)
    if (free[a]) augment(G,a,c,pot,free,pred,dist,PQ);
  forall(b,B)
    { forall_out_edges(e,b) result.append(e); }
  forall(e,result) G.rev_edge(e);
  return result;
}
```

We give the details of *augment*$(G, a, \ldots)$. It is a variant of Dijkstra's algorithm.

⟨*mwb_matching: helpers*⟩≡

```
⟨procedure augment_path_to⟩
template <class NT>
inline void augment(graph& G, node a, const edge_array<NT>& c,
                node_array<NT>& pot, node_array<bool>& free,
                node_array<edge>& pred, node_array<NT>& dist,
                node_pq<NT>& PQ)
{ ⟨augment: initialization⟩
  while ( true )
  { ⟨select from PQ the node b with minimal distance db⟩
    ⟨distinguish three cases⟩
  }
 ⟨augment: potential update and reinitialization⟩
}
```

We compute shortest paths starting in a. The priority queue *PQ* contains nodes in B (we will explain shortly why nodes in A are not put into the queue) together with their tentative distance from a, *minA* contains the minimum value of $\{\mu(v) + \pi(v) \; ; \; v \in A\}$ that we have seen so far, and *best_node_in_A* contains a node realizing *minA*. We use an array *dist* to record distances and an array *pred* to record predecessor edges in the shortest-path tree; this is as in Section 7.5.

Initially, the distance of a is zero, *minA* is equal to the potential of a, *best_node_in_A* is equal to a, and *PQ* contains all neighbors of a (recall that we store only nodes in B in the priority queue).

We do not define *PQ* within *augment* nor do we initialize *pred* within *augment*. This is absolutely vital for efficiency. We assume that *PQ* is empty and $pred[v] = nil$ for all v when *augment* is called. Within *augment* we collect, in stacks *RA* and *RB*, all nodes v (in A and B, respectively) that are added to *PQ* or for which $pred[v]$ is set. At the end of *augment* we use these stacks to reset *PQ* and *pred*. In this way augmentations can have sublinear running time.

⟨*augment: initialization*⟩≡

```
dist[a] = 0;

node best_node_in_A = a;
NT   minA           = pot[a];
NT   Delta;

stack<node> RA;  RA.push(a);
stack<node> RB;

node a1 = a; edge e;
⟨relax all edges out of a1⟩
```

where

⟨*relax all edges out of a1*⟩≡

```
forall_adj_edges(e,a1)
{ node b = G.target(e);
  NT db = dist[a1] + (pot[a1] + pot[b] - c[e]);
  if ( pred[b] == nil )
  { dist[b] = db; pred[b] = e; RB.push(b);
    PQ.insert(b,db);
  }
  else
  if ( db < dist[b] )
  { dist[b] = db; pred[b] = e;
    PQ.decrease_p(b,db);
  }
}
```

For each edge $e = (a1, b)$ we compute *db* as *dist*[*a1*] plus the reduced cost of e. If b is reached for the first time, we add it to *PQ* and to *RB*, and if w has been reached before but *db* is smaller than the current distance value of *b*, we update the distance value accordingly. We will reuse the program chunk above below and hence have formulated it for an arbitrary node $a1$ in A. In the main loop we remove the node with smallest distance from *PQ*. Let *b* be this node and let *db* be its distance; *b* is a node in B.

⟨*select from PQ the node b with minimal distance db*⟩≡

```
node b;
NT db;
if (PQ.empty()) b = nil;
else { b = PQ.del_min(); db = dist[b]; }
```

We distinguish three cases according to the discussion at the end of Section 7.8.2.

If b does not exist, i.e., PQ is empty, or $db \geq minA$, we augment by a path to node *best_node_in_A*. *Delta* is equal to *minA*.

If b exists, $db < minA$, and b is free, we augment by a path to b. *Delta* is equal to *db*.

If b exists, $db < minA$, and b is matched, we continue the shortest-path computation.

⟨*distinguish three cases*⟩≡

```
if ( b == nil || db >= minA )
{ Delta = minA;
  ⟨augmentation by path to best node in A⟩
}
else
{ if ( free[b] )
  { Delta = db;
    ⟨augmentation by path to b⟩
  }
  else
  { ⟨continue shortest-path computation⟩ }
}
```

Augmentation to the best node in A is done by *augment_path_to*(*best_node_in_A*, ...), which simply reverses the direction of all edges on the path from a to *best_node_in_A*. The path is given by the *pred*-array. We also declare a matched and *best_node_in_A* unmatched. It is important that we do the latter actions in this order, since a may be the best node in A, in which case we do not want to change the current matching.

⟨*augmentation by path to best node in A*⟩≡

```
augment_path_to(G,best_node_in_A,pred);
free[a] = false; free[best_node_in_A] = true; // order is important
break;
```

where

⟨*procedure augment_path_to*⟩≡

```
inline void augment_path_to(graph& G, node v,
                            const node_array<edge>& pred)
{ edge e = pred[v];
  while (e)
  { G.rev_edge(e);
    e = pred[G.target(e)]; // not source (!!!)
  }
}
```

Augmentation by a path to b is equally simple. We augment and declare a and b matched.

⟨*augmentation by path to b*⟩≡

```
augment_path_to(G,b,pred);
free[a] = free[b] = false;
break;
```

We come to the case where the shortest-path computation is to be continued. Then b is matched. Let e be the matching edge incident to b and consider the mate $a1$ of b. The mate has the same distance value as b and its predecessor edge is e.

If $db + pot[a1]$ is smaller than *minA* we update *minA* and *best_node_in_A*.

We also relax the edges out of $a1$. This may put more nodes in B into *PQ*. Observe that only nodes in B are put into *PQ*.

⟨*continue shortest-path computation*⟩≡

```
e = G.first_adj_edge(b);
node a1 = G.target(e);
pred[a1] = e; RA.push(a1);
dist[a1] = db;
if (db + pot[a1] < minA)
{ best_node_in_A = a1;
  minA = db + pot[a1];
}
⟨relax all edges out of a1⟩
```

This completes the description of the main loop.

We break from the main loop as soon as an augmenting path has been found. At this point $RA \cup RB$ contains all nodes that have been reached in the shortest-path computation and *Delta* contains the value required for the potential updates. For each node v in $RA \cup RB$ we reset $pred[v]$ to *nil*, remove v from the priority queue (only nodes in B can be in the queue), and update its potential. The potential change is $\max(0, Delta - dist[v])$. It is a decrease for the nodes in A and an increase for the nodes in B. For the nodes outside $RA \cup RB$ the potential does not change (by item (5) of the discussion at the end of Section 7.8.2).

⟨*augment: potential update and reinitialization*⟩≡

```
while (!RA.empty() )
{ node a = RA.pop();
  pred[a] = nil;
  NT pot_change = Delta - dist[a];
  if (pot_change <= 0 ) continue;
  pot[a] = pot[a] - pot_change;
}
while (!RB.empty() )
{ node b = RB.pop();
  pred[b] = nil;
  if (PQ.member(b)) PQ.del(b);
  NT pot_change = Delta - dist[b];
```

```
    if (pot_change <= 0 ) continue;
    pot[b] = pot[b] + pot_change;
  }
```

We come to the heuristics.

The simple heuristic sets *pot*[a] to the largest non-negative cost of any edge incident to a for every $a \in A$. This will make the heaviest edge incident to a tight (since the potential of all nodes in B is initially zero). The edge is added to the matching iff its endpoint in B is free.

⟨*simple heuristic*⟩≡

```
forall(a,A)
{ edge e_max = nil; NT C_max = 0;
  forall_adj_edges(e,a)
    if (c[e] >  C_max) { e_max = e; C_max = c[e]; }
  pot[a] = C_max;
  if ( e_max != nil && free[b = G.target(e_max)] )
  { G.rev_edge(e_max);
    free[a] = free[b] = false;
  }
}
```

The refined heuristic augments along paths of length one and length three. When it is called, the potential of all nodes in B is zero. It considers the nodes in A in turn. For each node $a \in A$ it determines the two incident edges with largest non-negative reduced cost. Call them *eb* and *e2*, respectively, and their reduced costs *max* and *max2*, respectively. If *e2* does not exist, then *max2* $= 0$, and if *eb* does not exist, then *max* $= 0$.

We then distinguish cases. If *eb* does not exist, we set *pot*[a] to zero. If *eb* exists, let b be the target of *eb*. If b is free, we add *eb* to the matching, record *e2* as the second best edge of a, and set *pot*[a] to *max2* and *pot*[b] to *max* - *max2*. This makes *eb* tight, and it makes *e2* tight if it leads to a free node in B. Finally, if b is not free we set *pot*[a] to *max* and consider the second best edge, say e, incident to the mate of b. If e exists and the target of e is free, we use the path of length three for augmentation.

⟨*mwb_matching: helpers*⟩+≡

```
template <class NT>
void mwbm_heuristic(graph& G, const list<node>& A,
                    const edge_array<NT>& c, node_array<NT>& pot,
                    node_array<bool>& free)
{
  node a, b;  edge e, e2, eb;
  node_array<edge> sec_edge(G,nil);
  forall( a, A )
  { NT max2 = 0; NT max = 0; eb = e2 = nil;
    // compute edges with largest and second largest slack
    forall_adj_edges( e, a )
    { NT we = c[e] - pot[target(e)];
```

```
        if ( we >= max2 )
        { if( we >= max )
          { max2 = max;  e2 = eb;
            max = we;    eb = e;
          }
          else
          { max2 = we;   e2 = e;
          }
        }
      }
      if( eb )
      { b = target(eb);
        if( free[b] )
        { // match eb and change pot[] to make slack of e2 zero
          sec_edge[a] = e2;
          pot[a] = max2;
          pot[b] = max-max2;
          G.rev_edge(eb);
          free[a] = free[b] = false;
        }
        else
        { // try to augment matching along
          // path of length 3 given by sec_edge[]
          pot[a] = max;
          e2 = G.first_adj_edge(b);
          e = sec_edge[target(e2)];
          if( e && G.outdeg(target(e)) == 0 )
          { free[a] = free[G.target(e)] = false;
            G.rev_edge(e); G.rev_edge(e2); G.rev_edge(eb);
          }
        }
      }
      else pot[a] = 0;
    }
  }
```

The worst case running time of our matching algorithm is n times the worst case running time of the shortest-path computation. The worst case running time of the shortest-path computation depends on the implementation of the priority queue. Priority queues are discussed in Section 5.4. With either the Fibonacci heap or the pairing heap implementation we obtain a worst case running time of $O(n(m + n \log n))$ and with the redistributive heap implementation we obtain a worst case running time of $O(n(m + n \log C))$ where C is the largest edge weight (edge weights are assumed to be integral for the latter time bound). The implementation given has worst case running time $O(n(m + n \log n))$. The average case running time seems to be much better as Table 7.8 shows.

Arithmetic Demand: How large are the numbers that are handled by the program above? Let us assume that all edge weights are integers whose absolute value is bounded by C.

We observe first that all node potentials are non-negative integers less than or equal to C.

This is clear for the nodes in A since their potential is initialized to a value less than or equal to C and is only decreased afterwards. For the nodes in B it follows from the observation that the potential of any matched node is at most C (since the reduced cost of a matched edge is zero) and that the potential of free nodes in B is zero.

The fact that node potentials are bounded by C implies that the reduced cost of any edge is bounded by $2C$. Thus the largest number handled in any of the shortest-path computations is at most $2\min(|A|, |B|) \cdot C$. This bound holds since matched edges have reduced cost zero and hence no simple path can contain more than $\min(|A|, |B|)$ edges of non-zero reduced cost.

We will next establish a much better bound. The quantity *minA* is always bounded by C, since it is initialized to the potential of a node in A and is only decreased afterwards. The shortest-path computation stops as soon as a distance value larger than *minA* is selected from the queue. Thus only distance values less than *minA* (and hence less than C) can lead to the insertion of additional distance values into the queue. We conclude that the maximal value ever put into the queue is bounded by C plus the maximal reduced cost of any edge and is hence bounded by $3C$. We summarize.

Lemma 31 *If all edge weights are integers whose absolute value is bounded by C then the largest number handled by the maximum weight bipartite matching algorithm is bounded by $3C$.*

Experimental Data: Table 7.8 contains some running times. We used random bipartite graphs with n nodes on each side and m edges, and three different kinds of edge weights:

- Uniform edge weights, i.e., all edge weights equal to one.
- Random edge weights in [1..1000].
- Large random edge weights in [10000..10005].

In all cases we also solved the corresponding unweighted matching problem.

The instances with random edge weights are by far the simplest, followed by the instances with large random edge weights, followed by the uniform instances. We expected that random edge weights from a large range lead to simple problems because heavy edges are much more favorable than light edges. We were surprised to find that the uniform problems are the hardest and have no explanation for it.

The density of the problem has a big influence on running time. For very sparse problems ($m = 2n$) the weighted matching algorithm is faster than the unweighted matching algorithm. This is due to the use of the potential function.

Consider the graph shown in Figure 7.36. It consists of a connected graph H which has a perfect matching and additional nodes $a_1, a_2, \ldots, a_k$. Each a_1 is connected to a node on the B-side of H. In the figure, all a_i are connected to the same node in B, but this is not essential. Assume that the perfect matching in H has already been constructed and that the nodes $a_1, a_2, \ldots, a_k$ are considered in turn. In the unweighted matching algorithm

C	n	m	No	Simple	Refined	Check	Unweighted
U	20000	40000	0.995	0.997	0.994	0.186	2.633
U	20000	60000	61.1	60.41	58.43	0.213	3.679
U	20000	80000	116.2	114.2	109.9	0.239	6.248
U	40000	80000	2.139	2.153	2.144	0.39	6.791
U	40000	120000	212.2	210.3	204.3	0.4539	9.61
U	40000	160000	410	402.8	387.8	0.5081	9.217
R	20000	40000	0.84	0.849	0.8467	0.1836	2.73
R	20000	60000	1.399	1.401	1.391	0.2189	3.811
R	20000	80000	2.635	2.509	2.578	0.2402	6.32
R	40000	80000	1.812	1.82	1.817	0.3922	7.056
R	40000	120000	3.001	2.941	2.973	0.4621	9.855
R	40000	160000	5.667	5.364	5.512	0.5168	9.532
L	20000	40000	1.293	1.31	1.307	0.1838	2.811
L	20000	60000	20.84	20.89	20.65	0.2305	3.922
L	20000	80000	41.6	40.69	41.05	0.2529	6.726
L	40000	80000	2.815	2.816	2.816	0.4213	7.222
L	40000	120000	57.06	56.9	54.67	0.4834	9.98
L	40000	160000	116.5	113.9	103.1	0.5283	9.595

Table 7.8 The running times of three versions of the weighted bipartite matching algorithm. The first three columns contain the running times of the algorithm above with the three different heuristics, the fourth column shows the time to verify the result and the last column shows the time required to solve the unweighted problem (by *MAX_CARD_BIPARTITE*). The graphs were generated by *random_bigraph*(G, n, n, m, A, B) and three kinds of edge weights were used: uniform edge weights (denoted U), i.e., all edge weights were set to one, random edge weights (denoted R) in [1 .. 1000] and random edge weights (denoted L) in [10000 .. 10005]. Each number is the average of ten runs. The function mwb_matching_time in the demo directory allows readers to perform their own experiments.

every search for an augmenting path will explore H in its entirety. Not so in the weighted matching algorithm. After the search from a_1, a_1 will have potential equal to zero (since it is free) and hence the node in B connected to it will have potential equal to one. Since H is assumed to be connected, every node in $H \cap B$ will have potential equal to one. Consider next a search for an augmenting path starting at a_i, $i \geq 2$. The node a_i is given potential one (since one is the largest cost of an edge incident to a_i), and hence all edges out of a_i have

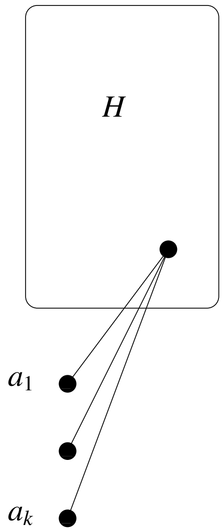

Figure 7.36 H has a perfect matching and each a_i, $1 \le i \le k$ is connected to some node on the B-side of H. After the search for an augmenting path from a_1 all nodes in B will have potential one. The searches from a_i, $i > 2$, take constant time.

reduced cost one. When the first neighbor of a_i is removed from the queue (with distance value one), the condition `dv >= minA` holds and hence the search for an augmenting path terminates. In this way, the fact that a_i cannot be matched is detected in time $O(1)$. We conclude that node potentials help tremendously in the example of Figure 7.36. Of course, this example is very special and hence we need to generalize the argument.

Our algorithm considers the nodes in A in turn. Let $A_{i-1} = \{a_1, \ldots, a_{i-1}\}$. After having considered the nodes in A_{i-1}, it has computed a maximal matching M_{i-1} in the subgraph G_{i-1} spanned by $A_{i-1} \cup B$ and a potential function π_{i-1} which proves the optimality of M_{i-1}. Observe now that a node in B which can be reached from a free node in A must have potential one (since free nodes have potential zero and hence their neighbors have potential one, and hence the neighbors of the neighbors have potential zero, ...).

Consider now the search for an augmenting path from a_i. We claim that it will not enter the subgraph H of G_{i-1} consisting of all nodes that can be reached from a free node in A_{i-1}. This is most easily seen for what we called the basic version of the algorithm in Section 7.8.2. We observe first that the node a_i is given potential one (since one is the largest cost of an edge incident to a_i) and hence an edge (a_i, b) will have reduced cost equal to zero or equal to one depending on whether the potential of b is zero or one. The edges connecting a_i to nodes in H will have reduced cost equal to one. The search will first explore all nodes that can be reached by tight edges. If a free node in B is reached, the matching will be increased. If no free node in B can be reached, a potential change will be made. The change reduces the potential of a_i to zero and hence no further search will be performed. We conclude that H is never entered.

For random edge weights the weighted matching algorithm is faster than the unweighted matching algorithm on the corresponding unweighted problem.

Alternative Interpretation: We close this section with an alternative interpretation of our algorithm. The alternative interpretation may be skipped.

We consider only the construction of M_i from M_{i-1} and π_{i-1}. For any alternating path starting in $a = a_i$ let $d(p)$ be the total cost of the edges in p that belong to M minus the total cost of the edges that do not belong to M, i.e.,

$$d(p) = \sum_{e \in p \cap M} c(e) - \sum_{p \setminus M} c(e).$$

Consider the matching $M \oplus p$ obtained by augmenting M by p. It has cost $c(M) - d(p)$ and hence $M \oplus p$ is "better" than M iff $d(p)$ is negative. This observation suggests the following definition. We call a path p improving with respect to M if $d(p)$ is negative. The observation also suggests the following algorithm for finding an improving path.

We orient all matching edges from B to A and all non-matching edges from A to B. We assign weight $c(e)$ to any matching edge and assign weight $-c(e)$ to any non-matching edge and search for a path of negative cost starting in a. If there is no such path then M is also a maximum cost matching in G_i. If there is such a path then let p be the most negative such path, i.e., the one with the most negative $d(p)$, and obtain M_i by augmenting M by p. A simple way to find p is to solve a single-source shortest-path problem with source a.

The previous paragraph leaves many questions unanswered. Why is M also a maximum cost matching in G_i if no path of negative cost exists, why is $M \oplus p$ a maximum cost matching in G_i if p is a most negative path, and why can there be no negative cycles?

In answering these questions the potential function $\pi = \pi_{i-1}$ comes handy. Recall that the first action in the construction of M_i is to extend π to a potential function on $A_i \cup B$ by setting $\pi(a)$ to any value that makes the reduced cost of every edge out of a non-negative. Consider any alternating path p with respect to M starting in a. Let $p = [e_1, \ldots, e_k]$ with $e_j = (v_{j-1}, v_j)$. Then v_0 is equal to a, $v_0, v_2, \ldots$ are nodes in A, $v_1, v_3, \ldots$ are nodes in B, $e_1, e_3, \ldots$ are edges not in M and $e_2, e_4, \ldots$ are edges in M, and if k is odd, then v_k is a free node in B. We have

$$d(p) = \sum_{j;\, j \text{ even}} c(e_j) - \sum_{j;\, j \text{ odd}} c(e_j).$$

Since π is tight with respect to M, we have $c(e_j) = \pi(v_{j-1}) + \pi(v_j)$ for all even j. Thus

$$\begin{aligned} d(p) &= \sum_{j;\, j \text{ even}} (\pi(v_{j-1}) + \pi(v_j)) - \sum_{j;\, j \text{ odd}} c(e_j) \\ &= -\pi(a) + \sum_{j;\, j \text{ odd}} (\pi(v_{j-1}) + \pi(v_j) - c(e_j)) + (-1)^k \pi(v_k) \\ &= -\pi(a) + \sum_{j;\, j \text{ odd}} \overline{c}(e_j) + (-1)^k \pi(v_k) \quad = \quad -\pi(a) + \sum_j \overline{c}(e_j) + (-1)^k \pi(v_k) \end{aligned}$$

$$= \quad -\pi(a) + \sum_j \overline{c}(e_j) + \pi(v_k).$$

This derivation deserves explanation. The first equality amounts to rearranging the sum. For example, if $k = 4$ then

$$\begin{aligned}&-c(e_1) + (\pi(v_1) + \pi(v_2)) - c(e_3) + (\pi(v_3) + \pi(v_4)) = \\ &\quad -\pi(a) + (\pi(v_0) - c(e_1) + \pi(v_1)) + (\pi(v_2) - c(e_3) + \pi(v_3)) + \pi(v_4)\end{aligned}$$

and if $k = 3$ then

$$\begin{aligned}&-c(e_1) + (\pi(v_1) + \pi(v_2)) - c(e_3) = \\ &\quad -\pi(a) + (\pi(v_0) - c(e_1) + \pi(v_1)) + (\pi(v_2) - c(e_3) + \pi(v_3)) - \pi(v_3).\end{aligned}$$

The second equality follows from $\overline{c}(e) = \pi(v) + \pi(w) - c(e)$ for any edge $e = (v, w)$, the third equality follows from the fact that $\overline{c}(e) = 0$ for any $e \in M$, and the last equality follows from the fact that $\pi(v_k) = 0$ if k is odd (since in this case v_k is a free node in B).

The derivation above is extremely powerful. It tells us that $d(p)$ is equal to the cost of p with respect to the reduced costs $\overline{c}$ plus the potential of the target node of p minus the potential of the source node of p. The source node of p is equal to a and hence the latter contribution is independent of p. In other words, searching for a path that minimizes $d(p)$ amounts to searching for a path that minimizes $\overline{c}(p) + \pi(v_k)$. For fixed v_k this amounts to searching for the path p from a to v_k that minimizes $\overline{c}(p)$. This problem is easily solved by Dijkstra's algorithm. For any node $v \in V_i$ let $\mu(v)$ be the minimum cost of a path from a to v with respect to the cost function $\overline{c}$. The iterative step from $M = M_{i-1}$ to M_i is then performed as follows:

Compute $\mu(v)$ for all v by Dijkstra's algorithm.

Let v be the node that minimizes $d = -\pi(a) + \mu(v) + \pi(v)$ and let p be a path from a to v that realizes $\mu(v)$.

If $d < 0$, augment M by p.

This completes our alternative derivation of the algorithm.

The first algorithm for the assignment problem was given by Kuhn [Kuh55]. In the early 60's, Jewell [Jew58], Iri [Iri60] and Busacker and Gowen [BG61] observed that the assignment problem can be solved by a sequence of shortest-path computations in general graphs. In the early 70's Tomizawa [Tom71] and Edmonds and Karp [EK72] showed that the use of node-potentials restricts the shortest-path computations to non-negative edge costs. Recent surveys of algorithms for the assignment problem can be found in an article by Galil [Gal86] and the book by Ahuja, Magnanti, and Orlin [AMO93]. In his master's thesis Markus Paul [Pau89] extended the algorithms to the maximum weight matching problem; he also implemented the algorithm for LEDA. His implementation always searched for augmenting paths from all nodes in A. Uli Finkler [Fin97] observed, in his PhD-thesis, that substantial improvements (not asymptotically but on average) can be obtained by considering the nodes in A one by one. The implementation given here follows his suggestion.

7.8.4 *The Assignment Problem*

The assignment problem asks for a perfect matching of maximum or minimum weight. A simple modification of the algorithm of the preceding section solves the maximum weight assignment problem.

We only need to change the way we search for augmenting paths. We insist that every augmentation increases the size of the matching and hence we continue our search for an augmenting path until a free node in B is found. When no free node in B is ever found, we return false to indicate that the graph has no perfect matching.

We obtain:

⟨*procedure augment for max weight assignment*⟩≡

```
#include <LEDA/stack.h>

template <class NT>
bool max_weight_assignment_augment(graph& G,
                    node a, const edge_array<NT>& c,
                    node_array<NT>& pot, node_array<bool>& free,
                    node_array<edge>& pred, node_array<NT>& dist,
                    node_pq<NT>& PQ)
{ ⟨augment: initialization⟩
  while ( true )
  { node b; NT db;
    if (PQ.empty()) { return false; }
    else { b = PQ.del_min(); db = dist[b]; }
    if ( free[b] )
    { Delta = db;
      ⟨augmentation by path to b⟩
    }
    else
    {⟨continue shortest-path computation⟩ }
  }
 ⟨augment: potential update and reinitialization⟩
 return true;
}
```

The minimum weight assignment problem is easily reduced to the maximum weight assignment problem. We only have to change the sign of all weights.

⟨*mwb_matching.t*⟩+≡

```
template <class NT>
list<edge> MIN_WEIGHT_ASSIGNMENT_T(graph& G,
                      const list<node>& A, const list<node>& B,
                      const edge_array<NT>& c, node_array<NT>& pot)
{ edge_array<NT> w(G);
  edge e;
  forall_edges(e,G) w[e] = - c[e];
  list<edge> M = MAX_WEIGHT_ASSIGNMENT_T(G,A,B,w,pot);
  node v;
```

```
  forall_nodes(v,G) pot[v] = -pot[v];
  return M;
}
```

The worst case running time of the maximum and minimum weight assignment algorithms is the same as for the maximum weight bipartite matching algorithm, namely $O(n(m + n \log n))$.

Arithmetic Demand: How large are the numbers that are handled by the assignment algorithms? We assume that all edge weights are integers whose absolute value is bounded by C. Let $k = |A| = |B|$.

We will first derive a bound on the node potentials. Let v be any node and consider a change[27] of $\pi(v)$. After a change of $\pi(v)$ there is an undirected path p of tight edges from a node $b \in B$ that was just matched to v. Let $p = [b = v_0, v_1, \ldots, v_s = v]$, where $s \leq 2k$. We claim that $\pi(v_i) \in [-iC \,..\, iC]$ for all i after the potential update. This is true for $i = 0$, since b was just matched and hence has potential equal to zero. For $i > 0$ the claim follows from the fact that the edge $\{v_i, v_{i-1}\}$ has reduced cost equal to zero and cost in $[-C \,..\, C]$. We conclude that $\pi(a) \in [-(2k-1)C \,..\, (2k-1)C]$ for $a \in A$ and $\pi(b) \in [-(2k-2)C \,..\, (2k-2)C]$ for $b \in B$ after a potential change. These bounds also hold before the first change of $\pi(v)$ since the potential of nodes in B is initialized to zero and since the potential of nodes in A is initialized such that there is a tight edge incident to the node.

The reduced cost of any edge is therefore bounded by $C + (2k-1)C + (2k-2)C \leq (4k-2)C$.

When we search for an augmenting path from a free node $a \in A$ we start a shortest-path computation from a. The computation stops when the first free node in B is encountered. Let p be an augmenting path from a to a free node in B. The maximal number handled in the shortest-path calculation is the cost of p (with respect to the reduced cost function) plus the maximal reduced cost of any edge. The cost of p is the difference between the old and the new potential of a and is therefore bounded by $4kC$. We conclude that the absolute value of all integers handled by the algorithm is bounded by $8kC$.

We summarize.

Lemma 32 *If all edge weights are integers whose absolute value is bounded by C then the absolute value of all numbers handled by the maximum and minimum weight assignment algorithm is bounded by $8kC = 4nC$, where $k = |A| = |B|$ and $n = 2k$.*

7.8.5 *Shortest Paths via Assignment*

Our algorithms for the maximum weight matching problem and the assignment problem use an algorithm for the shortest-path problem (for non-negative edge weights) as a subroutine. We show in this section that any algorithm for the assignment problem can be used to solve

[27] We will derive a bound on the initial value of $\pi(v)$ later in the section.

n	m	D	A	BF	A	BF	A
5000	50000	0.76	2.51	2.51	185.9	0.67	2.01

Table 7.9 A comparison of the running time of the shortest path via assignment algorithm (denoted A) with the shortest-path algorithms of Section 7.5. Columns three and four contain a comparison with Dijkstra's algorithm (D) and columns five and six and seven and eight contain a comparison with the Bellman–Ford algorithm (BF). We used random graphs with non-negative edge weights for the first comparison, random graphs with arbitrary edge weights but no negative cycle for the second comparison, and graphs generated by *BF_GEN* for the third comparison. The program shortest_path_via_assignment_time in the demo directory allows readers to perform their own experiments.

the shortest-path problem with arbitrary edge weights. This will give us an alternative to the algorithms in Section 7.5. The alternative is of considerable theoretical interest and has led to the asymptotically most efficient shortest-path algorithm for arbitrary edge costs, see [AMO93, sections 12.4 and 12.7]. We wrote this section to find out whether it also leads to efficient programs. At least in our implementation it does not, see Table 7.9.

Let $G = (V, E)$ be a directed graph. We construct a bipartite network $G' = (V' \dot\cup V'', E')$; see Figure 7.37 for an illustration. G' contains two copies of each node of G, one in V' and one in V''. For each node $v \in V$ we use v' to denote the copy in V' and v'' to denote the copy in V''. For each edge (v, w) there is an edge $\{v', w''\}$ of the same cost in E'. In addition, for each node $v \in V$ we have an edge $\{v', v''\}$ of cost zero in E'. Clearly, the set $\big\{\{v', v''\}\ ;\ v \in V\big\}$ is an assignment of cost zero. It is a minimum cost matching iff G has no negative cycle.

Lemma 33 *G' contains a perfect matching of negative cost iff G contains a negative cycle.*

Proof Let $C = [e_0, e_1, \ldots, e_{k-1}]$ with $e_i = (v_i, v_{i+1})$ and $v_k = v_0$ be a simple cycle of negative cost in G. We construct a perfect matching of the same cost in G'. It consists of the edges $\{v_i', v_{i+1}'\}$ for i, $0 \le i < k$, and the edges $\{v', v''\}$ for all nodes v that do not lie on C.

For the reverse direction consider any perfect matching M of negative cost in G'. We show that M corresponds to a set of cycles in G and that one of these cycles has negative cost. Consider any edge $\{v_0', v_1''\} \in M$ with $v_0 \neq v_1$; there must be at least one such edge since M has negative cost. The node v_1' must also be matched. Let v_2'' be its mate. Continuing in this fashion we construct a sequence of edges $\{v_0', v_1''\}, \{v_1', v_2''\}, \ldots, \{v_{k-1}', v_k''\}$ in M. We stop as soon as we encounter a node v_k'' such that v_k' appeared previously in the sequence. We must have $v_k = v_0$ since $v_k = v_j$ for some j, $j > 0$, implies that two matching edges are incident to v_k''. We conclude that $[v_0, v_1, \ldots, v_k]$ is a simple cycle in G. Thus M induces a set of simple cycles in G and the total cost of these cycles is equal to the cost of M. Hence, one of the cycles must have negative cost. □

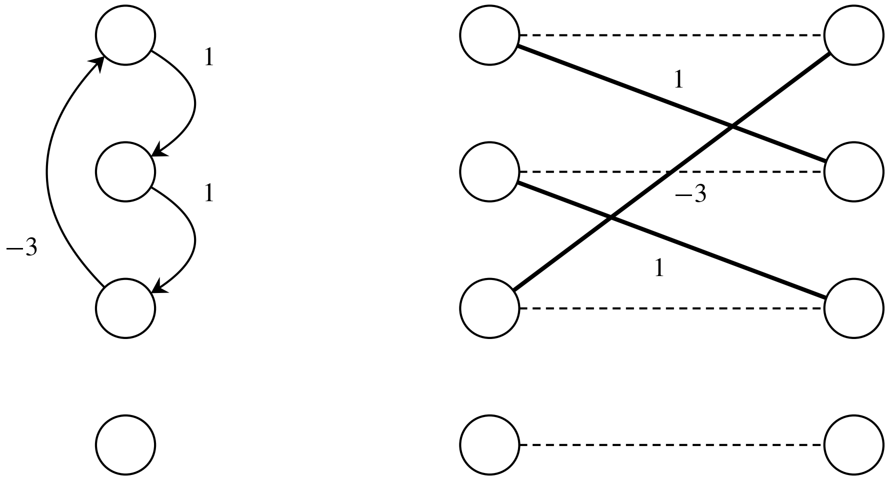

Figure 7.37 A directed graph and the derived bipartite graph. All dashed edges in the graph on the right have cost zero. The dashed edges define a perfect matching of cost zero. The solid edges together with the lowest dashed edge define a perfect matching of negative cost. It corresponds to the negative cycle in the graph on the left.

Assume now that G contains no negative cycle, let M be a minimum weight assignment in G' and let π' be a potential function that proves the optimality of M. We show that π' can be used to transform the cost function c into a non-negative cost function. M has cost zero[28] and hence $\sum_{v \in V} \pi'(v') + \pi'(v'') = 0$. Also $\pi'(v') + \pi'(v'') \le 0$ for all $v \in V$ and hence

$$\pi'(v') = -\pi'(v'') \text{ for all } v \in V.$$

We define a potential function π on V by

$$\pi(v) = \pi'(v'') \text{ for all } v \in V.$$

Consider any edge $e = (v, w)$ in G and let $\overline{c}(e) = \pi(v) + c(e) - \pi(w)$ be its reduced cost. We have:

$$\begin{aligned} \overline{c}(e) &= \pi(v) + c(e) - \pi(w) = \pi'(v'') + c(e) - \pi'(w'') \\ &= -\pi'(v') + c(e) - \pi'(w'') \ge 0, \end{aligned}$$

where the inequality follows from the fact that $c(e) \ge \pi'(v') + \pi'(w'')$ for all edges $e = \{v, w\}$.

We conclude that $\overline{c}$ is a non-negative cost function on G. The shortest-path problem with respect to $\overline{c}$ can be solved by Dijkstra's algorithm. Also, if $\mu(v)$ and $\overline{\mu}(v)$ are the shortest-path distances from s to v with respect to c and $\overline{c}$, respectively, then

$$\mu(v) = -\pi(s) + \overline{\mu}(v) + \pi(v),$$

[28] It is possible that one of the edges (v', v'') is not contained in M. How?

see Section 7.5.10.

The discussion above leads to the following program.

⟨*shortest_path_via_assignment.c*⟩≡

```
template <class NT>
bool shortest_path_via_assignment(const graph& G, node s,
                                  const edge_array<NT>& c,
                                  node_array<NT>& dist,
                                  node_array<edge>& pred)
{ node v,w; edge e;
  GRAPH<NT,NT> G1;
  list<node> A,B;
  node_array<node> left_copy(G), right_copy(G);
  forall_nodes(v,G)
  { A.append(left_copy[v]  = G1.new_node());
    B.append(right_copy[v] = G1.new_node());
    G1.new_edge(left_copy[v],right_copy[v],0);
  }
  forall_edges(e,G)
  { v = G.source(e); w = G.target(e);
    G1.new_edge(left_copy[v],right_copy[w],c[e]);
  }
  list<edge> M =
     MIN_WEIGHT_ASSIGNMENT_T(G1,A,B,G1.edge_data(),G1.node_data());
  NT sum = 0;
  forall_nodes(v,G1) sum += G1[v];
  if (sum < 0) return false;
  node_array<NT> pot(G);
  forall_nodes(v,G) pot[v] = G1[right_copy[v]];
  edge_array<NT> red_cost(G);
  forall_edges(e,G)
    red_cost[e] = pot[G.source(e)] + c[e] - pot[G.target(e)];
  DIJKSTRA_T(G,s,red_cost,dist,pred);
  forall_nodes(v,G) dist[v] += pot[v] - pot[s];
  return true;
}
```

7.8.6 *Maximum Weighted Matchings of Maximum Cardinality*

We show how to compute a matching of maximum weight among the matchings of maximum cardinality[29]. Let L be a real number and consider the weight function c_L defined by adding L to the weight of every edge, i.e.,

$$c_L(e) = c(e) + L \text{ for every } e \in E.$$

It is intuitively clear that larger values of L favor matchings of larger cardinality. We make this precise.

[29] For graphs that have a perfect matching this is the same as looking for a maximal weight perfect matching.

We observe first that $c_L(M) = c(M) + L|M|$ for any matching M. Thus, for two matchings M and N of the same cardinality the relative weight of the matchings does not change. Let C be the largest absolute value of any edge weight and let $k = \min(|A|, |B|)$. Then $|c(M)| \le kC$ for any matching M (since a matching consists of at most k edges) and hence $|c(N) - c(M)| \le 2kC$ for any two matchings M and N. We conclude that $|M| < |N|$ implies $c_L(M) < c_L(N)$ for $L > 2kC$. Thus in order to find a maximum weight matching of maximum cardinality we only have to find a maximum weight matching with respect to the cost function c_L where $L = 2kC + 1$.

$\langle$*mwb_matching.t*$\rangle{+}{\equiv}$

```
template <class NT>
list<edge> MWMCB_MATCHING_T(graph& G,
                           const list<node>& A, const list<node>& B,
                           const edge_array<NT>& c, node_array<NT>& pot)
{ NT C = 0;
  edge e;
  forall_edges(e,G)
  { if (c[e] > C) C = c[e];
    if (-c[e] > C) C = -c[e];
  }
  int k = Max(A.size(),B.size());
  C = 1 + 2*k*C;
  edge_array<NT> c_L(G);
  forall_edges(e,G) c_L[e] = c[e] + C;
  list<edge> M = MAX_WEIGHT_BIPARTITE_MATCHING_T(G,A,B,c_L,pot);
#ifndef LEDA_CHECKING_OFF
  if ( !CHECK_MWBM_T(G,c_L,M,pot) )
    error_handler(0,"check in MWMCB_MATCHING_T failed");
#endif
  return M;
}
```

Be aware that the computed potential function proves optimality with respect to the cost function c_L, where $L = 1 + 2kC$. The function has an arithmetic demand similar to the programs for the assignment problem. Recall that the maximum weight matching algorithm deals with numbers up to $3D$ when all edges costs are bounded by D in absolute value. We have $D = C + 1 + 2kC$ and hence the numbers handled by the algorithm may be as large as $3 + (6k + 3)C$. Since $C \ge 1$ and $k \ge 1$ we have $3 + (6k + 3)C \le 4nC$.

Exercises for 7.8

1 Write a checker for the maximum weight assignment problem.

2 Write a checker for the maximum weight assignment problem that takes only a matching M as input. Hint: Direct all edges in the matching from B to A, give each edge in the matching cost $c(e)$ and each edge outside the matching cost $-c(e)$. Show that the matching is optimal iff the resulting graph has no negative cycle.

3 Formulate Lemma 27 for the minimum weight assignment problem and write a checker for it.

4 Implement the basic version of the weighted bipartite matching algorithm.
5 Extend the function *shortest_path_via_assignment* such that it can also deal with graphs with negative cycles.
6 Show that the following strategy computes a maximum weight matching among the matchings of maximum cardinality: when searching for augmenting path from $a = a_i$ choose the shortest path to a free node in B (if there is one) and choose the path to the best node in A otherwise.
7 Write a program that computes a minimum weight matching among the matchings of maximal cardinality.
8 Write a program that computes a maximum weight matching of cardinality k, where k is a parameter of the algorithm. You may assume that the graph is connected.

7.9 Weighted Matchings in General Graphs

A *matching* M in a graph G is a subset of the edges no two of which share an endpoint, see Figure 7.38. The cardinality $|M|$ of a matching M is the number of edges in M. If w is a weight function on the edges of G then the weight $w(M)$ of a matching M is the sum of the weights of its edges, i.e.,

$$w(M) = \sum_{e \in M} w(e).$$

A node v is called *matched* with respect to a matching M if there is an edge in M incident to v and it is called *free* or *unmatched* otherwise. An edge e is called matching if $e \in M$. A matching is called a *maximum weight matching* if its weight is at least as large as the weight of any other matching. Figure 7.38 shows an example.

The function

```
list<edge> MAX_WEIGHT_MATCHING(const graph& G, const edge_array<int>& w)
```

returns a maximum weight matching in G with respect to the weight function w. Observe that the algorithm is only available for integer weights. The underlying algorithm is the so-called blossom shrinking algorithm of Edmonds[Edm65b, Edm65a]. Its worst case running time is $O(n^3)$ ([Law76]). The implementation is due to Thomas Ziegler [Zie95]. There are algorithms with better performance, both theorically [GMG86, Gal86] and practically [AC93]. At present the function cannot be asked to return a proof of optimality.

7.10 Maximum Flow

Let $G = (V, E)$ be a directed graph, let s and t be distinct vertices in G and let $cap : E \longrightarrow \mathbb{R}_{\geq 0}$ be a non-negative function on the edges of G. For an edge e, we call $cap(e)$

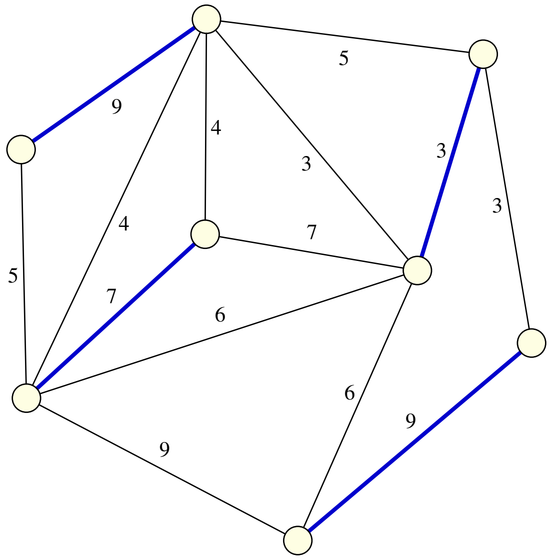

Figure 7.38 A maximum weight matching: The edges of the matching are shown in bold and the edge weights are indicated. We used the xlman-demo gw_mw_matching to generate this figure.

the *capacity* of e. An (s,t)-*flow* or simply *flow* is a function $f : E \longrightarrow \mathbb{R}_{\geq 0}$ satisfying the capacity constraints and the flow conservation constraints:

$$(1) \qquad 0 \leq f(e) \leq cap(e) \qquad \text{for every edge } e \in E$$

$$(2) \qquad \sum_{e;source(e)=v} f(e) = \sum_{e;target(e)=v} f(e) \qquad \text{for every node } v \in V \backslash \{s,t\}$$

The capacity constraints state that the flow across any edge is bounded by the capacity of the edge, and the flow conservation constraints state that for every node v different from s and t, the total flow out of the node is equal to the total flow into the node.

We call s and t the source and the sink of the flow problem, respectively, and we use V^+ to denote $V \backslash \{s,t\}$. For a node v, we call

$$excess(v) = \sum_{e;target(e)=v} f(e) - \sum_{e;source(e)=v} f(e)$$

the *excess* of v. Flow conservation states that all nodes except for s and t have zero excess.

The *value* of a flow f, denoted $|f|$, is the excess of the sink, i.e.,

$$|f| = excess(t).$$

A flow is called *maximum*, if its value is at least as large as the value of any other flow. Figure 7.39 shows an example.

In Section 7.10.1 we define the functionality of max flow algorithms and derive a checker,

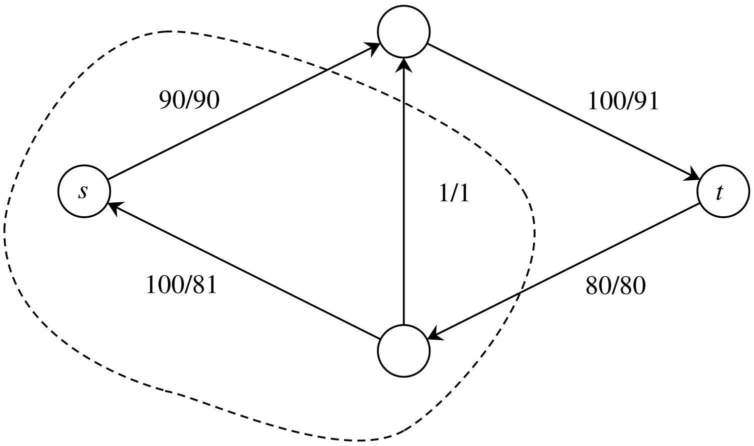

Figure 7.39 A maximum (s, t)-flow: For every edge e its capacity $cap(e)$ and the flow $f(e)$ across it are shown as $cap(e)/f(e)$. The value of the flow is equal 171. A saturated cut is indicated by the dashed line. It proves the maximality of the flow. The xlman-demo gw_max_flow visualizes maximum flows.

in Section 7.10.2 we discuss the generic preflow push algorithm, in Section 7.10.3 we give a first implementation of the preflow push algorithm, in Section 7.10.4 we describe several heuristic improvements, and in Section 7.10.5 we discuss the arithmetic demand of the algorithm and the danger of using the network flow algorithm with a number type that may incur rounding error.

7.10.1 *Functionality*

The function

```
NT MAX_FLOW_T(const graph& G, node s, node t
              const edge_array<NT>& cap, edge_array<NT>& f)
```

computes a maximum flow f in the network (G, s, t, cap) and returns the value of the flow. The function can be used with an arbitrary number type *NT*. There are pre-instantiated versions for the number types *int* and *double*. The function name of the pre-instantiated versions is MAX_FLOW, i.e., without the suffix _T. In order to use the pre-instantiated versions one must include <*LEDA/maxflow.h*>, and in order to use the template version, one must include <*LEDA/templates/maxflow.t*>.

Special care should be taken when using the template function with a number type *NT* that can incur rounding error, e.g., the type *double*. Section 7.2 contains a general discussion of this issue and Section 7.10.5 gives an example of what can go wrong in the computation of a maximum flow. The template function is only guaranteed to perform correctly if all arithmetic performed is without rounding error. This is the case if all numerical values in the input are integers (albeit stored as a number of type *NT*) and if none of the intermediate results exceeds the maximal integer representable by the number type ($2^{53} - 1$ in the case of *doubles*). All intermediate results are sums and differences of input values, in particular, the algorithms do not use divisions and multiplications.

The algorithm has the following arithmetic demand. Let C be the maximal absolute value of any edge capacity. If all capacities are integral then all intermediate values are bounded by $d \cdot C$, where d is the outdegree of the source.

The pre-instantiation for number type *int* issues a warning if C is larger than $MAXINT/d$.

The pre-instantiation for number type *double* computes the optimal matching for a modified capacity function *cap1*, where for every edge e

$$cap1[e] = sign(cap[e]) \lfloor |cap[e]| \cdot S \rfloor / S$$

and S is the largest power of two such that $S < 2^{53}/(d \cdot C)$. The value of the maximum flow for the modified capacity function and the value of the maximum flow for the original capacity function differ by at most $m \cdot d \cdot C \cdot 2^{-52}$.

The weight modification can also be performed explicitly and we advise you to do so. The function

```
bool MAX_FLOW_SCALE_CAPS(const graph& G, node s, edge_array<double>& cap)
```

replaces $cap[e]$ by $cap1[e]$ for every edge e, where $cap1[e]$ is as defined above. The function returns *false* if the scaling changed some weight, and returns *true* otherwise.

In the remainder of this section we discuss a check of optimality and derive the famous max-flow-min-cut theorem of Ford and Fulkerson [FF63]. We need a technical lemma that generalizes the notion of excess to a set of nodes.

Lemma 34 *Let $S \subseteq V$ and let $T = V \setminus S$. Then*

$$\sum_{u \in S} excess(u) = \sum_{e \in E \cap (T \times S)} f(e) - \sum_{e \in E \cap (S \times T)} f(e).$$

Proof We have

$$\sum_{u \in S} excess(u) = \sum_{u \in S} \left(\sum_{e;\, target(e)=u} f(e) - \sum_{e;\, source(e)=u} f(e) \right),$$

by definition of excess. We now observe that each edge $e \in E \cap (T \times S)$ contributes $f(e)$ to this sum, each edge $e \in E \cap (S \times T)$ contributes $-f(e)$ to this sum, and each edge $e \in E \cap (S \times S)$ contributes $f(e) - f(e)$ to this sum. □

We draw a quick consequence. An application with $S = V$ and hence $T = \emptyset$ yields

$$excess(s) + excess(t) = 0,$$

i.e., $excess(s) = -|f|$. This agrees with the intuition that the flow arriving at t must originate at s.

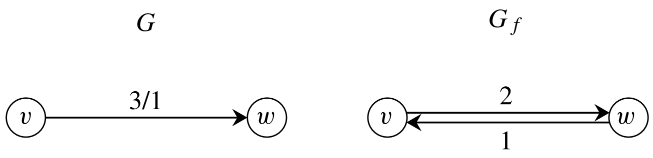

Figure 7.40 The residual network G_f: The left part shows an edge $e = (v, w)$ with capacity 3 and flow 1. It gives rise to two edges in the residual network shown on the right. The edge (v, w) has residual capacity 2 and the edge (w, v) has residual capacity 1.

An (s, t)-*cut* or simply *cut* is a set S of nodes with $s \in S$ and $t \notin S$. The *capacity* of a cut is the total capacity of the edges leaving the cut, i.e.,

$$cap(S) = \sum_{e \in E \cap (S \times T)} cap(e).$$

A cut S is called *saturated* if $f(e) = cap(e)$ for all $e \in E \cap (S \times T)$ and $f(e) = 0$ for all $e \in E \cap (T \times S)$.

The next lemma relates flows and cuts: the capacity of any (s, t)-cut is an upper bound for the value of any (s, t)-flow. Conversely, the value of any (s, t)-flow is a lower bound for the capacity of any (s, t)-cut.

Lemma 35 *Let f be any (s, t)-flow and let S be any (s, t)-cut. Then*

$$|f| \le cap(S).$$

If S is saturated then $|f| = cap(S)$.

Proof We have

$$\begin{aligned} |f| &= -excess(s) = -\sum_{u \in S} excess(u) \\ &= \sum_{e \in E \cap (S \times T)} f(e) - \sum_{e \in E \cap (T \times S)} f(e) \le \sum_{e \in E \cap (S \times T)} cap(e) \\ &= cap(S). \end{aligned}$$

For a saturated cut, the inequality is an equality. □

A saturated cut proves the maximality of f. A saturated cut is easily extracted from a maximum flow by means of the so-called residual network.

The *residual network* G_f with respect to a flow f has the same node set as G. Every edge of G_f is induced by an edge of G and has a so-called *residual capacity*. Let e be an arbitrary edge of G. If $f(e) < cap(e)$ then e is also an edge of G_f. Its residual capacity is $r(e) = cap(e) - f(e)$. If $f(e) > 0$ then e^{rev} is an edge of G_f. Its residual capacity is $r(e^{rev}) = f(e)$. Figure 7.40 shows an example.

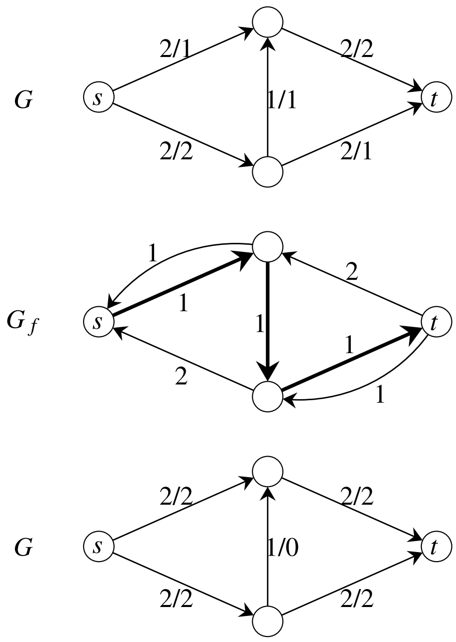

Figure 7.41 A path in the residual network and the resulting change of flow: A graph and an (s,t)-flow is shown at the top. The corresponding residual network is shown in the middle. A path p from s to t in the residual network is shown in bold. The flow obtained from augmentation by p is shown at the bottom.

Theorem 5 *Let f be an (s,t)-flow, let G_f be the residual network with respect to f, and let S be the set of nodes that are reachable from s in G_f.*

a) *If $t \in S$ then f is not maximum.*
b) *If $t \notin S$ then S is a saturated cut and f is maximum.*

Proof a) Let p be any simple path from s to t in G_f and let δ be the minimum residual capacity of any edge of p. Then $\delta > 0$. We construct a flow f' of value $|f| + \delta$. Let (see Figure 7.41)

$$f'(e) = \begin{cases} f(e) + \delta & \text{if } e \text{ is in } p \\ f(e) - \delta & \text{if } e^{rev} \text{ is in } p \\ f(e) & \text{if neither } e \text{ nor } e^{rev} \text{ belongs to } p. \end{cases}$$

Then f' is a flow and $|f'| = |f| + \delta$.

b) There is no edge (v,w) in G_f with $v \in S$ and $w \in T$. Hence, $f(e) = cap(e)$ for any e with $e \in E \cap (S \times T)$ and $f(e) = 0$ for any e with $e \in E \cap (T \times S)$, i.e., the cut S is saturated. Thus f is maximal. □

The function

```
bool CHECK_MAX_FLOW_T(const graph& G, node s, node t
                      const edge_array<NT>& cap, const edge_array<NT>& f)
```

checks whether f is a maximum (s,t)-flow. It returns *false* if this is not the case. The implementation is easy.

We check the capacity condition for each edge and compute the excess of all nodes. All nodes but s and t must have excess equal to zero. We then use breadth-first search to compute the set of nodes reachable from s in the residual graph; t must not be reachable.

⟨*max_flow_check*⟩≡

```
bool False_MF(string s)
{ cerr <<"\n\nCHECK_MAX_FLOW: " << s << "\n";
  return false;
}
template <class NT>
bool CHECK_MAX_FLOW_T(const graph& G, node s, node t,
                      const edge_array<NT>& cap, const edge_array<NT>& f)
{ node v; edge e;
  forall_edges(e,G)
    if ( f[e] < 0 && f[e] > cap[e] )
      return False_MF("illegal flow value");
  node_array<NT> excess(G,0);
  forall_edges(e,G)
  { node v = G.source(e); node w = G.target(e);
    excess[v] -= f[e]; excess[w] += f[e];
  }
  forall_nodes(v,G)
  { if ( v == s || v == t || excess[v] == 0 ) continue;
    return False_MF("node with non-zero excess");
  }
  node_array<bool> reached(G,false);
  queue<node> Q;
  Q.append(s); reached[s] = true;
  while ( !Q.empty() )
  { node v = Q.pop();
    forall_out_edges(e,v)
    { node w = G.target(e);
      if ( f[e] < cap[e] && !reached[w] )
      { reached[w] = true; Q.append(w); }
    }
    forall_in_edges(e,v)
    { node w = G.source(e);
      if ( f[e] > 0 && !reached[w] )
      { reached[w] = true; Q.append(w); }
    }
  }
  if ( reached[t] ) return False_MF("t is reachable in G_f");
  return true;
}
```

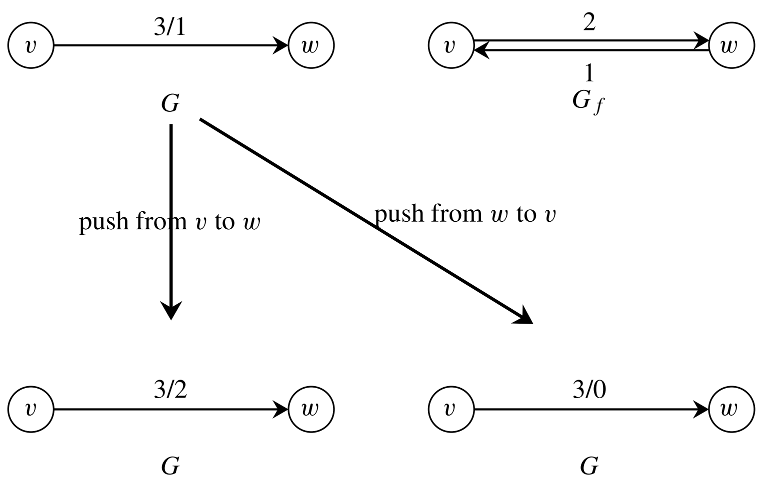

Figure 7.42 A push: The top left shows an edge $e = (v, w)$ in G with capacity three and flow one. This gives rise to two edges in the residual network shown on the right. A push of one unit of flow across e increases the flow across e by one and a push across e^{rev} decreases the flow across e by one.

7.10.2 *Algorithms*

The maximum flow problem is a widely studied problem and numerous algorithms have been proposed for it [FF63, EK72, Din70, Kar74, AO89, Gol85, GT88, CH95, CHM96, GR97].

Our implementations are based on the preflow-push method of Goldberg and Tarjan [GT88]. It manipulates a preflow that gradually evolves into a flow. Detailed computational studies of the preflow-push method can be found in [CG97, AKMO97] and in Section 7.10.4.

A *preflow* f is a function $f : E \longrightarrow \mathbb{R}_{\geq 0}$ with

$$\begin{array}{lll} (1) & 0 \leq f(e) \leq cap(e) & \text{for every edge } e \in E \text{ and} \\ (2) & excess(v) \geq 0 & \text{for every node } v \in V^+ \end{array}$$

i.e., the flow conservation constraint is replaced by the weaker constraint that no node in V^+ has negative excess. We call a node $v \in V^+$ *active* if its excess is positive. The residual network G_f with respect to a preflow f is defined as in the case of a flow.

The basic operation to manipulate a preflow is a *push*. Let v be an active node, let $e = (v, w)$ be a residual edge out of v, and let $\delta \leq \min(excess(v), r(e))$. A push of δ across e changes f as follows: it increases $f(e)$ by δ if e is an edge of G, and it decreases $f(e^{rev})$ by δ if e is the reversal of an edge of G, see Figure 7.42.

A push of δ across e increases $excess(w)$ by δ and decreases $excess(v)$ by δ. A push is called *saturating* if $\delta = r(e)$ and is called *non-saturating* otherwise. A saturating push across e removes e from the residual network and either kind of push adds e^{rev} to the residual network (if it is not already there).

The question is now which pushes to perform? Goldberg and Tarjan suggested to put the nodes of G (and hence G_f) onto layers with t on the bottom-most layer and to perform only pushes with transport excess to a lower layer. We use $d(v)$ to denote the (number of the) layer containing v. We call an edge $e = (v, w) \in G_f$ *eligible* if $d(w) < d(v)$.

Let us summarize: a push across an edge $e = (v, w) \in G_f$ can be performed if v is active and e is eligible. It moves $\delta \leq \min(excess(v), r(e))$ units of flow from v to w. If e is also an edge of G then $f(e)$ is increased by δ, and if e is the reversal of an edge of G then $f(e)$ is decreased by δ.

What are we going to do when v is active but there is no eligible edge out of v? In this situation v is *relabeled* by increasing $d(v)$ by one.

We are now ready for the generic preflow-push algorithm.

/* initialization */
set $f(e) = cap(e)$ for all edges with $source(e) = s$;
set $f(e) = 0$ for all other edges;
set $d(s) = n$ and $d(v) = 0$ for all other nodes;

/* main loop */
while there is an active node
{ let v be any active node;
  if there is an eligible edge $e = (v, w)$ in G_f
  { push δ across e for some $\delta \leq \min(excess(v), r(e))$; }
  else
  { relabel v; }
}

We will show that the algorithm terminates with a maximum flow (if it terminates). Call an edge $e = (v, w) \in G_f$ *steep* if $d(w) < d(v) - 1$, i.e., if it reaches down by two or more levels.

Lemma 36 *The algorithm maintains a preflow and does not generate steep edges. The nodes s and t stay on levels 0 and n, respectively.*

Proof The algorithm clearly maintains a preflow.

After the initialization, each edge in G_f either connects two nodes on level zero or connects a node on level zero to a node on level n. Thus, there are no steep edges (there are not even any eligible edges). A relabeling of a node v does not create a steep edge since a node is only relabeled if there are no eligible edges out of it. A push across an edge $e = (v, w) \in G_f$ may add the edge (w, v) to G_f. However, this edge is not even eligible.

Only active nodes are relabeled and only nodes different from s and t can be active. Thus, s and t stay on layers n and 0, respectively. □

The preceding lemma has an interesting interpretation. Since there are no steep edges, any path from v to t must have length (= number of edges) at least $d(v)$ and any path from

v to s must have length at least $d(v) - n$. Thus, $d(v)$ is a lower bound on the distance from v to t and $d(v) - n$ is a lower bound on the distance from v to s.

The next lemma shows that active nodes can always reach s in the residual network (since they must be able to send their excess back to s). It has the important consequence that d-labels are bounded by $2n - 1$.

Lemma 37 *If v is active then there is a path from v to s in G_f. No distance label ever reaches $2n$.*

Proof Let S be the set of nodes that are reachable from v in G_f and let $T = V \setminus S$. Then

$$\sum_{u \in S} excess(u) = \sum_{e \in E \cap (T \times S)} f(e) - \sum_{e \in E \cap (S \times T)} f(e),$$

by Lemma 34. Please convince yourself that this lemma holds for preflows and not only for flows.

There is no edge $(v, w) \in G_f$ with $v \in S$ and $w \notin S$. Thus, $f(e) = 0$ for every $e \in E \cap (T \times S)$. We conclude $\sum_{u \in S} excess(v) \leq 0$.

Since s is the only node whose excess may be negative and since $excess(v) > 0$ we must have $s \in S$.

Assume that a node v is moved to level $2n$. Since only active nodes are relabeled this implies the existence of a path (and hence simple path) in G_f from a node on level $2n$ to s (which is on level n). Such a path must contain a steep edge, a contradiction to Lemma 36. □

Theorem 6 *When the algorithm terminates, it terminates with a maximum flow.*

Proof When the algorithm terminates, there are no active nodes and hence the algorithm terminates with a flow. Call it f.

In G_f there can be no path from s to t since any such path must contain a steep edge (since s is on level n, t is on level 0). Thus, f is a maximum flow by Theorem 5. □

There is no guarantee that the generic preflow-push algorithm terminates, as it may choose to perform arbitrarily small pushes. However, it is fairly easy to bound the number of relabels and the number of saturating pushes.

Lemma 38 *There are at most $2n^2$ relabels and at most nm saturating pushes.*

Proof No distance label ever reaches $2n$ by Lemma 37 and hence each node is relabeled at most $2n$ times. The total number of relabels is therefore at most $2n^2$.

A saturating push across an edge $e = (v, w) \in G_f$ removes e from G_f. We claim that v has to be relabeled at least twice before the next push across e and hence there can be at most n saturating pushes across any edge. To see the claim, observe that only a push across e^{rev} can again add e to G_f. Since pushes occur only across eligible edges, w must be relabeled at least twice after the saturating push across e and before the next push across e^{rev}. Similarly, it takes two relabels of v before e becomes eligible again. □

It is more difficult to bound the number of non-saturating pushes. It depends heavily on which active node is selected for pushing, which edge is selected for pushing, and how much flow is pushed across the selected edge. In fact, without further assumptions, the number of non-saturating pushes is unbounded since we may choose to send only miniscule portions of flow. We make two assumptions for the remainder of the section:

***Maximality*:** Every push moves the maximal possible amount, i.e., when flow is pushed across an eligible edge $e = (v, w)$ out of an active node v, the amount pushed is

$$\delta = \min(excess(v), r(e)).$$

This rule guarantees that every non-saturating push makes the source of the push inactive.

***Persistence*:** When an active node v is selected, pushes out of v are performed until either v becomes inactive (because of a non-saturating push out of v) or until there are no eligible edges out of v anymore. In the latter case v is relabeled.

We study three rules for the selection of active nodes.

***Arbitrary*:** An arbitrary active node is selected. Goldberg and Tarjan have shown that the number of non-saturating pushes is $O(n^2 m)$ when the Arbitrary-rule is used. We will give their proof below.

***FIFO*:** The active nodes are kept in a queue and the first node in the queue is always selected. When a node is relabeled or activated the node is added to the rear of the queue. The number of non-saturating pushes is $O(n^3)$ when the FIFO-rule is used. This bound is due to Goldberg.

***Highest-Level*:** An active node on the highest level, i.e., with maximal dist-value, is selected. Observe that when a maximal level active node is relabeled it will be the unique maximal active node after the relabel. Thus, this rule guarantees that, when a node is selected, pushes out of the node will be performed until the node becomes inactive. The number of non-saturating pushes is $O(n^2\sqrt{m})$ when the highest-level-rule is used. This bound is due to Cheriyan and Maheshwari [CM89]. The proof given below is due to Cheriyan and Mehlhorn [CM99].

Lemma 39 *When the Arbitrary-rule is used, the number of non-saturating pushes is $O(n^2 m)$.*

Proof The proof makes use of a potential function argument. Consider the potential function

$$\Phi = \sum_{v;\, v \text{ is active}} d(v).$$

We will show:

(1) $\Phi \geq 0$ always, and $\Phi = 0$ initially.

(2) A non-saturating push decreases Φ by at least one.
(3) A relabeling increases Φ by one.
(4) A saturating push increases Φ by at most $2n$.

Suppose that we have shown (1) to (4). By (3) and (4) and Lemma 38 the total increase of Φ is at most $n^2 + nm2n = n^2(1 + 2m)$. By (1), the total decrease can be no larger than this. Thus, the number of non-saturating pushes can be at most $n^2(1 + 2m)$ by (3).

It remains to show (1) to (4). (1) is obvious. For (2) we observe that a non-saturating push deactivates a node. It may or may not activate a node at the level below. In either case, Φ decreases by at least one. For (3) we observe that a relabeling of v increases $d(v)$ by one, and for (4) we observe that a saturating push may activate a node and that all distance labels are bounded by $2n$. □

We turn to the FIFO-rule. Recall that it keeps the active nodes in a queue and always selects the head of the queue. Relabeled and activated nodes are added to the rear of the queue.

It is convenient to split the execution into phases. The first phase starts at the beginning of the execution and a phase ends when all nodes that were active at the beginning of the phase have been selected from the queue. In this way each node is selected at most once in each phase and hence the number of non-saturating pushes is at most n times the number of phases.

Lemma 40 *When the FIFO-rule is used, the number of non-saturating pushes is $O(n^3)$.*

Proof By the discussion preceding the lemma it suffices to show that the number of phases is $O(n^2)$.

We use a potential function argument. Consider

$$\Phi = \max \{d(v) \; ; \; v \text{ is active } \}.$$

We show:

(1) $\Phi \geq 0$ always, and $\Phi = 0$ initially.
(2) A phase containing no relabel operation decreases Φ by at least one.
(3) A phase containing a relabel operation increases Φ by at most one.

Suppose that we have shown (1) to (3). By (3) and Lemma 38, the total increase is bounded by $2n^2$. By (1), the total decrease can be no larger. Thus the number of phases containing no relabel operation is bounded by $2n^2$ by (3). The total number of phases is therefore bounded by $4n^2$.

It remains to show (1) to (3). (1) is obvious. For (2) we observe that if a phase contains no relabel operation then all nodes selected in the phase get rid of their excess and push it to a lower layer. Thus, Φ decreases by at least one (it can decrease by more than one if an active node on level $n + 1$ pushes its excess back to s). For (3), we observe that pushes move excess to a lower layer and that a relabeling of a node moves the node to one higher level. □

We turn to the highest-level selection rule. Recall that it always selects an active node with maximal distance label.

Lemma 41 *When the Highest-Level-rule is used, the number of non-saturating pushes is $O(n^2\sqrt{m})$.*

Proof We use a potential function argument. Let $K = \sqrt{m}$; this choice of K will become clear at the end of the proof. For a node v, let

$$d'(v) = |\{w; d(w) \le d(v)\}|/K$$

and consider

$$\Phi = \sum_{v;v \text{ is active}} d'(v).$$

We split the execution into phases. We define a phase to consist of all pushes between two consecutive changes of

$$d^* = \max\{d(v)\ ;\ v \text{ is active }\}$$

and call a phase *expensive* if it contains more than K non-saturating pushes, and *cheap* otherwise.

We show:

(1) The number of phases is at most $4n^2$.

(2) The number of non-saturating pushes in cheap phases is at most $4n^2K$.

(3) $\Phi \ge 0$ always, and $\Phi \le n^2/K$ initially.

(4) A relabeling or a saturating push increases Φ by at most n/K.

(5) A non-saturating push does not increase Φ.

(6) An expensive phase containing $Q \ge K$ non-saturating pushes decreases Φ by at least Q.

Suppose that we have shown (1) to (6). (4) and (5) imply that the total increase of Φ is at most $(2n^2 + mn)n/K$ and hence the total decrease can be at most this number plus n^2/K by (3). The number of non-saturating pushes in expensive phases is therefore bounded by $(2n^3 + n^2 + mn^2)/K$. Together with (2) we conclude that the total number of non-saturating pushes is at most

$$(2n^3 + n^2 + mn^2)/K + 4n^2K.$$

Observing that $n = O(m)$ and that the choice $K = \sqrt{m}$ balances the contributions from expensive and cheap phases, we obtain a bound of $O(n^2\sqrt{m})$.

It remains to prove (1) to (6). For (1) we observe that $d^* = 0$ initially, $d^* \ge 0$ always, and that only a relabel can increase d^*. Thus, d^* is increased at most $2n^2$ times, decreased no more than this, and hence changed at most $4n^2$ times. (2) follows immediately from (1) and the definition of a cheap phase. (3) is obvious. (4) follows from the observation that $d'(v) \le n/K$ for all v and at all times. For (5) observe that a non-saturating push across an edge (v, u) deactivates v, activates u (if it is not already active), and that $d'(u) \le d'(v)$.

For (6) consider an expensive phase containing $Q \geq K$ non-saturating pushes. By definition of a phase, d^* is constant during a phase, and hence all Q non-saturating pushes must be out of nodes at level d^*. The phase is finished either because level d^* becomes empty or because a node is moved from level d^* to level $d^* + 1$. In either case, we conclude that level d^* contains $Q \geq K$ nodes at all times during the phase. Thus, each non-saturating push in the phase decreases Φ by at least one (since $d'(u) \leq d'(v) - 1$ for a push from v to u). □

7.10.3 *A First Implementation*

We describe a first implementation of the generic preflow-push algorithm. The implementation is straightforward. We initialize a preflow, refine the flow into a flow, check that the computed flow is maximal, and return the value of the flow.

We want to execute the program with different rules for selection from the set of active nodes and therefore give the function two template parameters: the number type *NT* and the implementation of the set U of active nodes.

We want to count the number of pushes, the number of relabels, and the number of inspections of edges and therefore introduce appropriate parameters.

⟨*max_flow_basic*⟩≡

```
template<class NT, class SET>
NT MAX_FLOW_BASIC_T(const graph& G, node s, node t,
                    const edge_array<NT>& cap, edge_array<NT>& flow,
                    SET& U,
                    int& num_pushes, int& num_edge_inspections,
                    int& num_relabels)
{ if (s == t) error_handler(1,"MAXFLOW: source == sink");
  ⟨MF_BASIC: initialization⟩
  ⟨MF_BASIC: main loop⟩
#ifndef LEDA_CHECKING_OFF
  assert(CHECK_MAX_FLOW_T(G,s,t,cap,flow));
#endif
  return excess[t];
}
```

Initialization and Data Structures: We use the following data structures and variables: for each edge e we store the flow across e in *flow*$[e]$ and for each node v we store the level of v and the excess of v in *dist*$[v]$ and *excess*$[v]$, respectively. We store the active nodes in U.

We initialize the flow and the excess to zero, we put all nodes except for s on level zero, we put s on level n, we saturate all edges out of s, and initialize U with all nodes of positive excess. Thus

⟨*MF_BASIC: initialization*⟩≡
 ⟨*initialize flow and excess and saturate edges out of s*⟩
 ⟨*MF_BASIC: initialize dist and U*⟩
 ⟨*MF_BASIC: initialize counters*⟩

where

⟨*initialize flow and excess and saturate edges out of s*⟩≡
```
flow.init(G,0);
if (G.outdeg(s) == 0) return 0;
int n = G.number_of_nodes(); int max_level = 2*n - 1;
int m = G.number_of_edges();
node_array<NT>  excess(G,0);
// saturate all edges leaving s
edge e;
forall_out_edges(e,s)
{ NT c = cap[e];
  if (c == 0) continue;
  node v = target(e);
  flow[e] = c;
  excess[s] -= c;
  excess[v] += c;
 }
```

⟨*MF_BASIC: initialize dist and U*⟩≡
```
node_array<int> dist(G,0); dist[s] = n;
node v;
forall_nodes(v,G)
 if ( excess[v] > 0 ) U.insert(v,dist[v]);
```

⟨*MF_BASIC: initialize counters*⟩≡
```
num_relabels = num_pushes = num_edge_inspections = 0;
```

Implementations of the Set of Active Nodes: The implementation of U must support the following operations:

node U.del( ); delete a node from U and return it (return *nil* if U is empty).

U.insert(*node* v, *int* d); insert a node v with dist-value d. This version is to be used in the initialization phase and when a node is reinserted into the set of active nodes after a relabel.

U.insert0(*node* v, *int* d); insert a node v with dist-value d. This version is to be used when a node gets activated by a push into it.

bool U.empty( ); return true if U is empty.

U.clear( ); remove all elements from U.

Construction and Destruction.

We give three implementations:

The *FIFO implementation* keeps the nodes in U in a queue. Insertions add to the end of the queue, and deletions remove from the front of the queue.

⟨*FIFO implementation of SET*⟩≡

```
#include <LEDA/list.h>
class fifo_set{
  list<node> L;
public:
  fifo_set(){}
  node del() { if (!L.empty()) return L.pop(); else return nil; }
  void insert(node v, int d)  { L.append(v); }
  void insert0(node v, int d) { L.append(v); }
  bool empty() { return L.empty(); }
  void clear() { L.clear(); }
  ~fifo_set(){}
};
```

The *MFIFO (modified FIFO) implementation* keeps the nodes in U in a linear list and always selects the first node from the list. Nodes that are reinserted after a relabel operation are added to the front of the linear list, and nodes that get activated by a push into them are added to the rear of the list. In this way the same node is selected until all excess is removed from the node. The MFIFO implementation guarantees an $O(n^3)$ bound on the number of non-saturating pushes, see the exercises.

⟨*MFIFO implementation of SET*⟩≡

```
#include <LEDA/list.h>
class mfifo_set{
  list<node> L;
public:
  mfifo_set(){}
  node del() { if ( !L.empty() ) return L.pop(); else return nil; }
  void insert(node v, int d) { L.push(v); }
  void insert0(node v, int d){ L.append(v); }
  bool empty() { return L.empty(); }
  void clear() { L.clear(); }
  ~mfifo_set(){}
};
```

The *highest-level implementation* of U maintains an array A of linear lists with index range $[0\,..\,max\_level]$, where *max_level* is an argument of the constructor. The list $A[d]$ contains all nodes v that were inserted by *insert*(v, d) or *insert0*(v, d). The implementation maintains a variable *max* such that $A[d]$ is empty for $d > max$. In *insert0* we exploit the fact that it always inserts below the maximal level.

⟨*Highest level implementation of SET*⟩≡

```
#include <LEDA/list.h>
#include <LEDA/array.h>
class hl_set{
  int max, max_lev;
  array<list<node> > A;
public:
  hl_set(int max_level):A(max_level+1)
  { max = -1; max_lev = max_level;}
  node del()
  { while (max >= 0 && A[max].empty()) max--;
    if (max >= 0) return A[max].pop(); else return nil;
  }
  void insert(node v, int d)
  { A[d].push(v);
    if (d > max) max = d;
  }
  void insert0(node v, int d) { A[d].append(v); }
  bool empty()
  { while (max >= 0 && A[max].empty()) max--;
    return ( max < 0 );
  }
  ~hl_set(){}
  void clear()
  { for (int i = 0; i <= max_lev; i++) A[i].clear();
    max = -1;
  }
};
```

The Main Loop: In the main loop we select a node v from U. We call v the *current* node. If v does not exist, we break from the main loop, and if v is equal to t, we continue to the next iteration of the main loop. So assume otherwise. We try to push the excess of v to its neighbors in the residual graph. We inspect first the residual edges that correspond to edges out of v in G and then the residual edges that correspond to edges into v in G.

If v remains active after saturating all residual edges out of it, we relabel v and reinsert it into U.

⟨*MF_BASIC: main loop*⟩≡

```
for(;;)
{
  node v = U.del();
  if (v == nil) break;
  if (v == t) continue;
```

```
    NT  ev = excess[v]; // excess of v
    int dv = dist[v];   // level of v
    edge e;
    ⟨MF_BASIC: push across edges out of v⟩
    if ( ev > 0 )
    { ⟨MF_BASIC: push across edges into v⟩ }
    excess[v] = ev;
    if (ev > 0)
    { dist[v]++;
      num_relabels++;
      U.insert(v,dist[v]);
    }
  }
```

Pushing Excess Out of a Node: Let v be a node with positive excess. We want to push flow out of v along eligible edges. An edge $e \in G_f$ is either also an edge of G (and then *flow*$[e] <$ *cap*$[e]$) or the reversal of an edge of G (and then *flow*$[e^{rev}] > 0$). We therefore iterate over all edges out of v and all edges into v.

For each edge e out of v we push $\max($*excess*$[v]$, *cap*$[e] -$ *flow*$[e])$. If a push decreases the excess of v to zero we break from the loop.

⟨*MF_BASIC: push across edges out of v*⟩≡

```
  for (e = G.first_adj_edge(v); e; e = G.adj_succ(e))
  { num_edge_inspections++;
    NT& fe = flow[e];
    NT  rc = cap[e] - fe;
    if (rc == 0) continue;
    node w = target(e);
    int dw = dist[w];
    if ( dw < dv ) // equivalent to ( dw == dv - 1 )
    { num_pushes++;
      NT& ew = excess[w];
      if (ew == 0) U.insert0(w,dw);
      if (ev <= rc)
      { ew += ev; fe += ev;
        ev = 0;   // stop: excess[v] exhausted
        break;
      }
      else
      { ew += rc; fe += rc;
        ev -= rc;
      }
    }
  }
```

The code for the edges into v is symmetric.

⟨*MF_BASIC: push across edges into v*⟩≡

```
for (e = G.first_in_edge(v); e; e = G.in_succ(e))
{ num_edge_inspections++;
  NT& fe = flow[e];
  if (fe == 0) continue;
  node w = source(e);
  int dw = dist[w];
  if ( dw < dv ) // equivalent to ( dw == dv - 1 )
  { num_pushes++;
    NT& ew = excess[w];
    if (ew == 0) U.insert0(w,dw);
    if (ev <= fe)
    { fe -= ev; ew += ev;
      ev = 0;   // stop: excess[v] exhausted
      break;
    }
    else
    { ew += fe; ev -= fe;
      fe = 0;
    }
  }
}
```

Our first implementation is now complete. Let us see how it performs. We investigate the worst case complexity first and then give experimental data.

Worst Case Running Time: The running time of our implementation, not counting the time spent in the implementation of U, is proportional to the number of edge inspections. We bound the number of edge inspections first and then turn to the time spent in the implementation of U.

Consider an arbitrary iteration of the main loop and let v be the node selected in the iteration. In the iteration we inspect all edges incident to v, and either perform a push across an edge incident to v or relabel v. Thus the number of inspections of an edge e is bounded by the number of relabels of the endpoints of e plus the number of pushes out of the endpoints of e. No node is relabeled more than $2n$ times and hence the total number of edge inspections due to relabels is $O(nm)$. If P denotes the total number of pushes then the number of edge inspections due to pushes is $O(deg^* \cdot P)$, where deg^* is the maximal degree of any node. The number of pushes is $O(n^3)$ with the FIFO or MFIFO implementation for the set of active nodes and is $O(n^2\sqrt{m})$ with the highest-level implementation.

We turn to the time spent in maintaining the set of active nodes. For the FIFO and MFIFO implementation each operation on U takes constant time, and for the highest-level implementation each operation on U takes constant time plus the number of decreases of *max*. The number of decreases of *max* is bounded by the total increase of *max* and *max* is only increased by relabel operations. A relabel increases *max* by one. We conclude the total change of *max* is bounded by $O(n^2)$ by Lemma 38. The time spent in maintaining the set of active nodes is therefore $O(n^2)$ plus the number of operations on U. The number of operations on U is certainly bounded by the number of edge inspections.

We summarize in:

Theorem 7 *The worst case running time of our implementation is* $O(n^3 \cdot deg^*)$ *with the FIFO- or MFIFO-rule and is* $O(n^2\sqrt{m} \cdot deg^*)$ *with the highest-level-rule, where* deg^* *is the maximum degree of any node.*

The deg^*-factor in the running time is easily removed by means of the so-called *current edge data structure*. We used it already in Section 7.6. We found that the improvement is theoretical and does not show positively in the observed running times for all graphs where the average degree is bounded by 20. We therefore did not include the current edge data structure in our implementations.

We maintain for each node v a current out-edge *cur_out_edge*$[v]$ and a current in-edge *cur_in_edge*$[v]$ with the property that:

- no edge preceding *cur_out_edge*$[v]$ in the list of edges out of v is eligible and
- no edge preceding *cur_in_edge*$[v]$ in the list of edges into v is eligible.

When we push excess out of v we start searching for eligible edges at *cur_out_edge*$[v]$ and *cur_in_edge*$[v]$, respectively. When we relabel v we reset *cur_out_edge*$[v]$ and *cur_in_edge*$[v]$ to the first edge out of v and into v, respectively.

The implementation is correct since the only way a non-eligible edge $e = (v, w)$ can become eligible is through a relabeling of v.

The current edge implementation has the property that for any node v and between consecutive relabels of v the time spent in searching for eligible edges incident to v is proportional to the degree of v plus the number of pushes performed. The total time spent in searching for eligible edges is therefore bounded by $O(nm)$ plus the number of pushes.

Theorem 8 *The worst case running time of our implementation with the current edge data structure is* $O(n^3)$ *with the FIFO- or MFIFO-rule and is* $O(n^2\sqrt{m})$ *with the highest-level-rule.*

Four Generators: We describe four generators for max flow problems.

The first generator produces a graph with n nodes and $2n + m$ edges. It first produces a random graph with n nodes and m edges and makes s and t the first and the last node of G, respectively. It then adds edges (s, v) and (v, t) for all nodes v. The capacities are random numbers between 2 and 11 for all edges leaving s and between 1 and 10 for all other edges.

⟨_max_flow_gen.c⟩+≡

```
void max_flow_gen_rand(GRAPH<int,int>& G, node& s, node& t, int n, int m)
{ G.clear();
  random_graph(G,n,m);
  s = G.first_node(); t = G.last_node();
```

```
  node v; edge e;
  forall_nodes(v,G) { G.new_edge(s,v); G.new_edge(v,t); }
  forall_edges(e,G)
    G[e] = ( G.source(e) != s ? rand_int(1,10) : rand_int(2,11) );
}
```

The next two generators are due to Cherkassky and Goldberg [CG97]. For each integer k, $k \geq 1$, they generate the networks shown in Figure 7.43.

⟨*_max_flow_gen.c*⟩+≡

```
void max_flow_gen_CG1(GRAPH<int,int>& G, node& s, node& t, int n)
{ G.clear();
  if (n < 1)
    error_handler(1,"max_flow_gen_CG1: n must be at least one");
  array<node> V(2*n);
  int i;
  for(i = 0; i < 2*n; i++) V[i] = G.new_node();
  s = V[0]; t = V[2*n - 1];
  node v = V[n];
  for (i = 0; i < n; i++)
  { G.new_edge(V[i],V[i + 1], n - i);
    G.new_edge(V[i],v, 1);
  }
  G.new_edge(V[n - 1],V[2*n - 1], 1);
  G.new_edge(V[n - 1],V[n], 1);
  for (i = n; i <= 2*n - 2 ; i++ ) G.new_edge(V[i],V[i + 1],n);
}
void max_flow_gen_CG2(GRAPH<int,int>& G, node& s, node& t, int n)
{ G.clear();
  if (n < 1)
    error_handler(1,"max_flow_gen_CG2: n must be at least one");
  array<node> V(2*n);
  int i;
  for(i = 0; i < 2*n; i++) V[i] = G.new_node();
  s = V[0]; t = V[2*n-1];
  for (i = 0; i < n; i++ ) G.new_edge(V[i],V[2*n - 1 - i], 1);
  for (i = 0; i <= n - 1; i++ ) G.new_edge(V[i],V[i + 1], 2*n);
  for (i = n; i <= 2*n - 2; i++ ) G.new_edge(V[i],V[i + 1], n);
}
```

Observe the order in which we generate the edges out of node i: the edge from i to $2n-1-i$ precedes the edge to node $i+1$.

The fourth generator was suggested by Ahuja, Magnanti, and Orlin [AMO93]. The generated network is also shown in Figure 7.43.

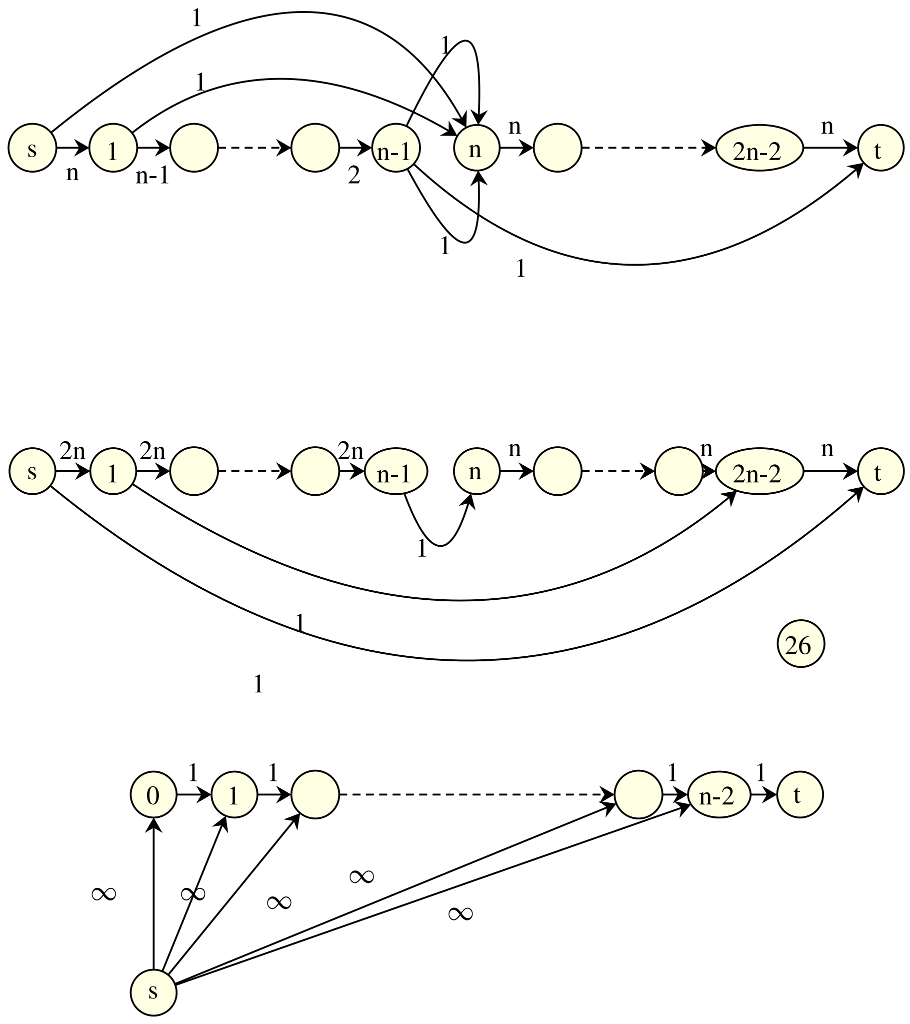

Figure 7.43 The generators *max_flow_gen_CG1*, *max_flow_gen_CG2*, and *max_flow_gen_AOM* generate the graphs shown. All three generators take the parameter *n* as an input.

⟨*_max_flow_gen.c*⟩+≡

```
void max_flow_gen_AMO(GRAPH<int,int>& G, node& s, node& t, int n)
{ G.clear();
  if (n < 1)
    error_handler(1,"max_flow_gen_AMO: n must be at least one");
  array<node> V(n);
  s = G.new_node();
  int i;
```

```
  for(i = 0; i < n; i++) V[i] = G.new_node();
  t = G.last_node();
  for (i = n - 2; i >= 0; i-- )
  { G.new_edge(s,V[i], 10000);
    G.new_edge(V[i],V[i + 1], 1);
  }
}
```

Running Times: Table 7.10 shows the behavior of our first implementation of the preflow-push method with three different selection rules and for four different kinds of graphs. For each of the four generators above we ran the cases $n = 500$ and $n = 1000$. For the random graph generator we used $m = 3n$. The number of pushes, the number of edge inspections, the number of relabels, and the running time quadruples or more than quadruples when n is doubled.

In the next section we will describe several optimizations which will lead to a dramatic improvement of observed running time. None of them improves the worst case behavior, however.

7.10.4 *Optimizations*

What is the best case running time of our implementation? The running time is $\Omega(n^2)$ if $\Omega(n)$ nodes need to be lifted above level n. This is usually the case. The best case behavior of the other parts of the algorithm is $O(m)$ and hence the cost of relabeling dominates the best case running time. In this section we will describe several heuristics that frequently reduce the time spent in relabeling nodes and as a side-effect reduce the time spent in all other operations. The heuristics will turn the preflow-push algorithm into a highly effective algorithm for solving flow problems.

Consider the example shown in Figure 7.44. We have nodes 0 to $n-1$, $s = 0$, $t = n-1$, and edges $(i, i+1)$ for all i, $0 \le i < n-1$. All edges have capacity two, except for edge $(n-2, n-1)$ which has capacity one.

Let us see what the preflow-push method does. In the initialization phase we saturate the edge (0, 1), put s on level n, and all other nodes on level 0. Node 1 has positive excess. We lift node 1 to level 1 and push its excess to node 2. We lift node 2 to level 1 and push its excess to node 3. Continuing in this way the excess is pushed to node $n-2$. Only one unit can be forwarded to t and one unit remains on node $n-2$. At this point the value of the maximum flow has been determined. There is one unit of flow into t and this is the maximum possible. However, the algorithm does not know this fact yet and it will take the algorithm a long time to discover it. We lift node $n-2$ to level 2 and push the unit back to node $n-3$. Continuing in this way we lift nodes $n-2, n-3, \ldots, 2$ to level 2 and push the excess back to node 1. Then we lift node 1 to level 2 and then level 3, and $\ldots$. Continuing in this way, we will invest $\Omega(n^2)$ relabels (and pushes) until nodes 1 to $n-1$ end up at level $n+1$. At this point we can push the excess back to s and the algorithm terminates.

Generator	Rule	Pushes	Inspections	Relabels	Time
rand	FIFO	1.764e+05	2.467e+06	2.34e+05	1.42
		6.831e+05	9.833e+06	9.28e+05	5.88
	HL	1.775e+05	2.672e+06	2.34e+05	1.47
		7.442e+05	1.073e+07	9.28e+05	6.04
	MFIFO	2.262e+05	2.566e+06	2.34e+05	1.28
		8.524e+05	1.018e+07	9.28e+05	5.25
CG1	FIFO	1.761e+05	9.63e+05	2.281e+05	0.81
		6.835e+05	4.121e+06	8.92e+05	3.94
	HL	1.875e+05	6.009e+06	1.885e+05	2.75
		7.5e+05	4.486e+07	7.52e+05	20.47
	MFIFO	1.682e+05	8.629e+05	2.207e+05	0.68
		6.713e+05	3.608e+06	8.801e+05	3.08
CG2	FIFO	2.864e+06	1.367e+07	2.751e+06	12.12
		1.149e+07	5.479e+07	1.1e+07	50.97
	HL	1.695e+06	1.226e+07	2.752e+06	11.33
		6.764e+06	4.902e+07	1.1e+07	43.17
	MFIFO	2.864e+06	1.367e+07	2.751e+06	11.02
		1.149e+07	5.479e+07	1.1e+07	45.14
AMO	FIFO	500	4.498e+06	1.5e+06	3.27
		1000	1.8e+07	6e+06	13.13
	HL	500	4.498e+06	1.5e+06	3.79
		1000	1.8e+07	6e+06	15.25
	MFIFO	500	4.498e+06	1.5e+06	2.74
		1000	1.8e+07	6e+06	11.13

Table 7.10 The basic implementation of the preflow-push algorithm. We show its behavior for four different kinds of graphs and three different selection rules. For each generator we ran the cases $n = 500$ and $n = 1000$. For the random graph generator we used $m = 3n$. The program max_flow_basic_time in the demo directory allows readers to make their own experiments.

We describe five optimizations. The first optimization is based on the observation that nodes on layer n and above can be treated more simply than nodes below level n. The

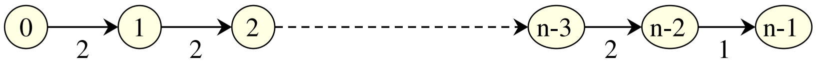

Figure 7.44 A network with nodes $0, \ldots, n-1$ and edges $(i, i+1)$ for all $i, 0 \leq i < n-1$. All edges have capacity two except for edge $(n-2, n-1)$ which has capacity one.

second and third optimizations increase distance labels more aggressively, the fourth optimization splits the execution into two phases (where a maximum preflow is computed in the first phase and the remaining excess is pushed back to s in the second phase), and the fifth optimization recognizes nodes that have no chance of forwarding their flow to t. The combined effect of the five heuristics is to reduce the running time dramatically for many instances of the max flow problem, see Table 7.16 on page 204.

Large Distance Labels: We call a node v *high* if $d(v) \geq n$ and *low* otherwise and show that high nodes can be treated simpler than low nodes.

What distinguishes high nodes from low nodes? There can never be a path of residual edges from a high node to t as any such path would necessarily contain a steep edge. All excess of active high nodes must therefore flow back to s. The situation is different for active low nodes. Some of their excess can be pushed to t and some of their excess must flow back to s.

How can we exploit the difference? All excess of active high nodes must flow back to s. The excess reaches the active high nodes through edges $e \in E$ with $f(e) > 0$. This suggests that it can be sent back through such edges.

We therefore define

$$E_f^* = \left\{ e^{rev} \,;\, e \in E \text{ and } f(e) > 0 \right\}$$

and use only edges in E_f^* when pushing out of high active nodes. We relabel a high active node when there are no eligible edges in E_f^* out of it.

/* initialization */
set $f(e) = cap(e)$ for all edges with $source(e) = s$;
set $f(e) = 0$ for all other edges;
set $d(s) = n$ and $d(v) = 0$ for all other nodes;

/* main loop */
while there is an active node
{ let v be any active node;
  if $d(v) < n$ and there is an eligible edge $e = (v, w) \in E_f$ or
    $d(v) \geq n$ and there is an eligible edge $e = (v, w) \in E_f^*$
  { push δ across e for $\delta = \min(excess(v), r(e))$; }
  else
  { relabel v; }
}

We need to show that the modified algorithm is correct. We adapt the correctness proof of the basic preflow-push algorithm. The modified algorithm may create steep edges. We show that no steep edge can end below level $n - 1$ and that every steep edge belongs to $E_f \setminus E_f^*$; this modifies Lemma 36.

Lemma 42 *Any residual edge $e = (v, w)$ that becomes steep in the modified algorithm satisfies $e \in E_f \setminus E_f^*$ and $d(w) \geq n - 1$.*

Proof A steep edge $e = (v, w)$ can only be created by a relabeling of v. A node v is only relabeled when there is no eligible edge $(v, w) \in E_f^*$. Thus only edges in $E_f \setminus E_f^*$ can become steep.

A node v with $d(v) < n$ is only relabeled when there is no eligible edge out of it. Thus a relabeling of v that creates a steep edge $e = (v, w)$ can only occur when $d(v) \geq n$. The edge e was not steep before the relabeling of v and hence $d(w) \geq n - 1$. □

We next show that every active node can reach s in G_f^*; this modifies Lemma 37. The proof carries over almost literally.

Lemma 43 *If v is active then there is a path from v to s in G_f^*. No distance label ever reaches $2n$.*

Proof Let S be the set of nodes that are reachable from v in G_f^* and let $T = V \setminus S$. Then

$$\sum_{u \in S} excess(u) = \sum_{e \in E \cap (T \times S)} f(e) - \sum_{e \in E \cap (S \times T)} f(e),$$

by Lemma 34.

There is no edge $(v, w) \in G_f^*$ with $v \in S$ and $w \notin S$. Thus, $f(e) = 0$ for every $e \in E \cap (T \times S)$. We conclude $\sum_{u \in S} excess(v) \leq 0$.

Since s is the only node whose excess may be negative and since $excess(v) > 0$ we must have $s \in S$.

Assume that a node u is moved to level $2n$. Since only active nodes are relabeled this implies the existence of a path (and hence simple path) in G_f^* from a node on level $2n$ to s (which is on level n). Such a path must contain a steep edge, a contradiction to Lemma 42. □

Theorem 9 *When the modified algorithm terminates it terminates with a maximum flow. All bounds on the number of relabels and the number of pushes shown for the basic algorithm hold also true for the modified algorithm.*

Proof When the algorithm terminates there are no active nodes and hence the algorithm terminates with a flow. Call it f.

Assume that there is a path p in G_f from s to t. Write $p = p_1 \odot p_2$ where p_1 ends in a node with level at least n and p_2 contains no node with level n or more. Then p_2 starts with a node on level $n-1$ and contains no steep edges. Both claims follow from Lemma 42. However, p_2 contains at most $n-1$ nodes (since it cannot contain s) and hence must contain a steep edge.

Thus there is no path from s to t in G_f and hence f is optimal by Theorem 5. □

The changes in the program are minor. We push across the edges out of v only when v lives on a layer less than n.

⟨*MF_LH: main loop*⟩≡

```
for(;;)
{
  node v = U.del();
  if (v == nil) break;
  if (v == t) continue;
  NT  ev = excess[v]; // excess of v
  int dv = dist[v];   // level of v
  edge e;
  if ( dist[v] < n )
  { ⟨MF_BASIC: push across edges out of v⟩ }
  if ( ev > 0 )
  { ⟨MF_BASIC: push across edges into v⟩ }
  excess[v] = ev;
  if (ev > 0)
  { dist[v]++;
    num_relabels++;
    U.insert(v,dist[v]);
  }
}
```

The procedure MAX_FLOW_LH_T results from MAX_FLOW_BASIC_T by replacing the main loop. Table 7.11 shows the effect of distinguishing between low and high nodes. The effect is small and significant savings are only observed for the CG2-generator.

The Local Relabeling Heuristic: The *local relabeling heuristic* applies whenever a node is relabeled. It increases the dist-value of v to

$$1 + \min\left\{d(w)\ ;\ (v, w) \in G_f\right\}.$$

Observe that v is active whenever it is relabeled and that an active node has at least one outgoing edge in G_f. The expression above is therefore well defined. When v is relabeled, none of the outgoing edges is eligible and hence $d(w) \geq d(v)$ for all $(v, w) \in G_f$. Thus, the local relabeling heuristic increases $d(v)$ by at least one. It may increase it by more than one.

The correctness of the heuristic follows from the following alternative description: when a node is relabeled, continue to relabel it until there is an eligible edge out of it.

The local relabeling heuristic is easily incorporated into our implementation. We maintain a variable *dmin*, which we initialize to MAXINT before we scan the edges incident

Generator	Rule	Pushes	Inspections	Relabels	Time
rand	FIFO	1.728e+05	2.426e+06	2.27e+05	1.51
		1.726e+05	2.422e+06	2.269e+05	1.54
	HL	1.811e+05	2.654e+06	2.27e+05	1.6
		1.81e+05	2.649e+06	2.269e+05	1.64
	MFIFO	2.164e+05	2.513e+06	2.27e+05	1.36
		2.16e+05	2.508e+06	2.268e+05	1.4
CG1	FIFO	1.761e+05	9.63e+05	2.281e+05	0.85
		1.761e+05	9.63e+05	2.281e+05	0.9
	HL	1.875e+05	6.009e+06	1.885e+05	2.83
		1.875e+05	6.009e+06	1.885e+05	2.88
	MFIFO	1.682e+05	8.629e+05	2.207e+05	0.73
		1.682e+05	8.629e+05	2.207e+05	0.89
CG2	FIFO	2.864e+06	1.367e+07	2.751e+06	12.82
		2.54e+06	1.221e+07	2.544e+06	11.98
	HL	1.695e+06	1.226e+07	2.752e+06	11.31
		1.57e+06	1.12e+07	2.627e+06	11.24
	MFIFO	2.864e+06	1.367e+07	2.751e+06	11.6
		2.54e+06	1.221e+07	2.544e+06	10.87

Table 7.11 Effect of low-high distinction. We show the behavior for three different kinds of graphs and three different selection rules. For each generator we ran the case $n = 500$. For the random graph generator we used $m = 3n$. For each case we give the running time of MAX_FLOW_BASIC_T (first line) and of MAX_FLOW_LH_T (second line). Use the program max_flow_lh_time in the demo directory to perform your own experiments.

to the current active node v. Let $e = (v, w)$ be a residual edge. If e is eligible, i.e., $d(w) < d(v)$, we push across e, and if e is not eligible, i.e, $d(w) \geq d(v)$, we set *dmin* to $\min(dmin, d(w))$. If v is still active after scanning all residual edges incident to it, we can set $d(v)$ to $1 + dmin$.

We obtain

⟨*push across edges out of v*⟩≡

```
for (e = G.first_adj_edge(v); e; e = G.adj_succ(e))
{ num_edge_inspections++;
  NT& fe = flow[e];
```

```
    NT  rc = cap[e] - fe;
    if (rc == 0) continue;
    node w = target(e);
    int dw = dist[w];
    if ( dw < dv ) // equivalent to ( dw == dv - 1 )
    { num_pushes++;
      NT& ew = excess[w];
      if (ew == 0) U.insert0(w,dw);
      if (ev <= rc)
      { ew += ev; fe += ev;
        ev = 0;   // stop: excess[v] exhausted
        break;
      }
      else
      { ew += rc; fe += rc;
        ev -= rc;
      }
    }
    else { if ( dw < dmin ) dmin = dw; }
  }
```

The code for the edges into v is symmetric.

⟨*push across edges into v*⟩≡

```
  for (e = G.first_in_edge(v); e; e = G.in_succ(e))
  { num_edge_inspections++;
    NT& fe = flow[e];
    if (fe == 0) continue;
    node w = source(e);
    int dw = dist[w];
    if ( dw < dv ) // equivalent to ( dw == dv - 1 )
    { num_pushes++;
      NT& ew = excess[w];
      if (ew == 0) U.insert0(w,dw);
      if (ev <= fe)
      { fe -= ev; ew += ev;
        ev = 0;   // stop: excess[v] exhausted
        break;
      }
      else
      { ew += fe; ev -= fe;
        fe = 0;
      }
    }
    else { if ( dw < dmin ) dmin = dw; }
  }
```

The main loop turns into

⟨*MF_LRH: main loop*⟩≡

```
for(;;)
{
  node v = U.del();
  if (v == nil) break;
  if (v == t) continue;
  NT  ev   = excess[v]; // excess of v
  int dv   = dist[v];   // level of v
  int dmin = MAXINT;     // for local relabeling heuristic
  edge e;
  if (dv < n)
  { ⟨push across edges out of v⟩ }
  if ( ev > 0 )
  { ⟨push across edges into v⟩ }
  excess[v] = ev;
  if (ev > 0)
  { dist[v] = 1 + dmin;
    num_relabels++;
    U.insert(v,dist[v]);
  }
}
```

The procedure MAX_FLOW_LRH_T results from MAX_FLOW_BASIC_T by replacing the main loop. Table 7.12 shows the combined effect of the local relabeling heuristic and the low-high distinction.

The Global Relabeling Heuristic: The *global relabeling heuristic* updates the dist-values of all nodes. It sets

$$d(v) = \begin{cases} \mu(v,t) & \text{if there is a path from } v \text{ to } t \text{ in } G_f \\ n + \mu^*(v,s) & \text{if there is a path from } v \text{ to } s \text{ in } G^*_f \text{ but no} \\ & \text{path from } v \text{ to } t \text{ in } G_f \\ 2n-1 & \text{otherwise} \end{cases}$$

Here $\mu(v,t)$ and $\mu^*(v,s)$ denote the lengths (= number of edges) of the shortest paths from v to t in G_f and from v to s in G^*_f, respectively. The reader should convince himself that the global relabeling heuristic does not generate any steep edges.

The global relabeling heuristic can be implemented by breadth-first search and requires time $O(m)$. It should therefore not be applied too frequently. We will apply it every $h \cdot m$ edge inspections for some suitable constant h. In this way $\Omega(m)$ time is spent between applications of the global relabel heuristic and hence the worst case running time is increased by at most a constant factor. The best case can improve significantly.

In our example from the beginning of the section, the global relabeling heuristic is highly effective. Assume that it is applied after the edge $(n-2, n-1)$ is saturated. It will put node i on level $n+i$ for all i, $1 \le i \le n-2$, and the excess on node $n-2$ will flow back to s in a series of n pushes. In this way the running time decreases from $\Omega(n^2)$ to $O(n)$.

Generator	Rule	Pushes	Inspections	Relabels	Time
rand	FIFO	1.878e+05	2.554e+06	2.349e+05	1.51
		1.945e+05	1.949e+06	1.498e+05	1.25
	HL	1.915e+05	2.768e+06	2.349e+05	1.6
		1.915e+05	2.04e+06	1.36e+05	1.27
	MFIFO	2.332e+05	2.644e+06	2.348e+05	1.39
		2.332e+05	1.986e+06	1.457e+05	1.17
CG1	FIFO	1.761e+05	9.63e+05	2.281e+05	0.85
		2.234e+05	7.007e+05	1.403e+05	0.68
	HL	1.875e+05	6.009e+06	1.885e+05	2.8
		1.875e+05	5.726e+06	9.438e+04	2.67
	MFIFO	1.682e+05	8.629e+05	2.207e+05	0.71
		1.682e+05	5.482e+05	1.16e+05	0.52
CG2	FIFO	2.54e+06	1.221e+07	2.544e+06	11.35
		2.216e+06	9.529e+06	1.82e+06	9.19
	HL	1.57e+06	1.12e+07	2.627e+06	10.35
		1.57e+06	7.51e+06	1.377e+06	7.41
	MFIFO	2.54e+06	1.221e+07	2.544e+06	10.35
		2.54e+06	9.996e+06	1.796e+06	8.99

Table 7.12 Effect of low-high distinction and local relabeling heuristic. We show the behavior for three different kinds of graphs and three different selection rules. For each generator we ran the case $n = 500$. For the random graph generator we used $m = 3n$. For each case we give the running time of MAX_FLOW_LH_T (first line) and of MAX_FLOW_LRH_T (second line). The local relabeling heuristic results in a considerable saving in all cases. Use max_flow_lrh_time in the demo directory to perform your own experiments.

We turn to the implementation.

We define two functions *compute_dist_t* and *compute_dist_s* that compute the distance to t and s, respectively. Both functions need access to the residual graph and hence have parameters G, *flow*, and *cap*. We also provide them with the node t and the node s, respectively. The functions store the computed distances in *dist*. It is assumed that $dist[v] \geq n$ for all nodes v prior to a call of *compute_dist_t* and that $dist[v] = 2 * n - 1$ for all nodes v that cannot reach t in G_f prior to a call of *compute_dist_s*; the latter function also assumes that nodes that can reach t in G_f have a distance value less than n.

The calls insert all active nodes with their new distance labels into U. It is assumed that U is empty prior to a call of *compute_dist_t* and that U contains all active nodes that can reach t in G_f prior to a call of *compute_dist_s*.

The functions are realized by breadth-first search and hence need a queue Q. We provide it as a parameter. It is assumed that the queue is empty prior to a call of both functions. Both functions leave Q empty when they terminate.

The function *compute_dist_t* also computes for each d, $0 \le d < n$, the number of nodes v with $dist[v] = d$ and stores the number in $count[d]$; this count will be needed in the so-called gap heuristic to be described later.

The details of both functions are fairly simple. In *compute_dist_t* we perform a "backward" breadth-first search starting at t. Whenever a new node w is reached, say from node v, we set $dist[w]$ to $1 + dist[v]$, we insert w into U if it is active, we increase $count[dist[w]]$, and we add w to the rear of Q. Since we are computing distances to t and s, respectively, all edges are considered in their reverse direction.

⟨*max_flow_dist_st*⟩+≡

```
template<class NT, class SET>
void compute_dist_t(const graph& G, node t, const edge_array<NT>& flow,
                    const edge_array<NT>& cap,
                    const node_array<NT>& excess, node_array<int>& dist,
                    SET& U, b_queue<node>& Q, array<int>& count)
{
  int n = G.number_of_nodes();
  Q.append(t);
  dist[t] = 0;
  count.init(0);
  count[0] = 1;
  while ( !Q.empty() )
  { node v = Q.pop();
    int  d = dist[v] + 1;
    edge e;
    for(e = G.first_adj_edge(v); e; e = G.adj_succ(e))
    { if ( flow[e] == 0 ) continue;
      node u = target(e);
      int& du = dist[u];
      if ( du >= n )
      { du = d;
        Q.append(u); count[d]++;
        if ( excess[u] > 0 ) U.insert(u,d);
       }
     }
    for(e = G.first_in_edge(v); e; e = G.in_succ(e))
    { if ( cap[e] == flow[e] ) continue;
      node u = source(e);
      int& du = dist[u];
      if ( du >= n )
      { du = d;
        Q.append(u); count[d]++;
```

```
        if (excess[u] > 0) U.insert(u,d);
      }
    }
  }
}
```

The "backward" breadth-first search from s is simpler because it only needs to consider edges in G_f^*.

⟨*max_flow_dist_st*⟩+≡

```
template<class NT, class SET>
void compute_dist_s(const graph& G, node s, const edge_array<NT>& flow,
                    const node_array<NT>& excess, node_array<int>& dist,
                    SET& U, b_queue<node>& Q)
{
  int n = G.number_of_nodes();
  int max_level = 2*n - 1;
  Q.append(s);
  dist[s] = n;
  while ( !Q.empty() )
  { node v = Q.pop();
    int  d = dist[v] + 1;
    edge e;
    for(e = G.first_adj_edge(v); e; e = G.adj_succ(e))
    { if ( flow[e] == 0 ) continue;
      node u = target(e);
      int& du = dist[u];
      if ( du == max_level )
      { du  = d;
        if (excess[u] > 0) U.insert(u,d);
        Q.append(u);
       }
     }
   }
}
```

Before we describe the required changes to the initialization phase and the main loop we describe one further optimization.

Two-Phase Approach: We partition the execution into two phases. The first phase ends when there is no active node at a level below n anymore. At this point of the execution the algorithm has determined a maximum preflow, i.e., a preflow which maximizes *excess*$[t]$. This follows from the observation that there can be no path in G_f from an active node to t at the end of phase one.

In the first phase we push only out of nodes with level below n and in the second phase we push only out of nodes with level at least n. Phase two ends when there are no active nodes anymore.

For the first phase we initialize *dist*$[v]$ with the distance from v to t (if v can reach t in

G_f where f is the flow obtained by saturating all edges out of s) and we initialize *dist*$[v]$ with n otherwise.

⟨*MF_GRH: initialize dist and U for first phase*⟩≡
```
node_array<int> dist(G);
dist.init(G,n);
compute_dist_t(G,t,flow,cap,excess,dist,U,Q,count);
```

The other initializations are as before:

⟨*MF_GRH: initialization*⟩≡
⟨*initialize flow and excess and saturate edges out of s*⟩
⟨*MF_GRH: additional data structures*⟩
⟨*MF_GRH: initialize dist and U for first phase*⟩
⟨*MF_GRH: initialize counters*⟩

⟨*MF_GRH: initialize counters*⟩≡
```
num_relabels = num_pushes = num_edge_inspections = 0;
num_global_relabels  = 0;
```

We need some additional data structures: the global distance calculations need a queue and we need to know which phase we are in. We also need to introduce the array *count*: *count*$[d]$ is to contain the number of nodes at level d for $0 \le d < n$. It will be required by the gap heuristic to be explained below.

⟨*MF_GRH: additional data structures*⟩≡
```
b_queue<node> Q(n);
int phase_number = 1;
array<int> count(n);
```

The main loop has the same structure as before.

⟨*MF_GRH: main loop*⟩≡
```
for(;;)
{
  ⟨MF_GRH: extract v from queue⟩
  NT  ev   = excess[v]; // excess of v
  int dv   = dist[v];   // level of v
  int dmin = MAXINT;
  edge e;
  if ( dist[v] < n )
  { ⟨push across edges out of v⟩ }
  if ( ev > 0 )
  { ⟨push across edges into v⟩ }
  excess[v] = ev;
  if (ev > 0)
  { ⟨MF_GRH: update distance label(s)⟩ }
}
```

We still need to describe how nodes are selected from the queue and how distance labels are updated.

Let v be the node selected from the set U of active nodes. If v does not exist and we are in the second phase, we break from the main loop. If v does not exist and we are in the first phase, we start the second phase. If v is equal to t, we ignore v. In all other cases, we proceed and attempt to push out of v.

How do we start the second phase? We need to initialize the distance labels and also the set of active nodes for the second phase. We first compute the set of nodes that can still reach t (none of them is active) and collect its complement in a set S. None of the nodes in S can reach t. We then compute the distance labels for all nodes in S by computing their distances to s in G_f^*.

⟨*MF_GRH: extract v from queue*⟩≡

```
node v = U.del();
if (v == nil)
{
  if ( phase_number == 2 ) break; // done
  dist.init(G,n);
  compute_dist_t(G,t,flow,cap,excess,dist,U,Q,count);
  node u;
  forall_nodes(u,G)
  { if (dist[u] == n)
    { S.append(u);
      dist[u] = max_level;
    }
  }
  phase_number = 2;
  compute_dist_s(G,s,flow,excess,dist,U,Q);
  continue;
}
if (v == t) continue;
```

The set S needs to be declared.

⟨*MF_GRH: additional data structures*⟩+≡

```
list<node> S;
```

It remains to describe how we update distance labels. We mentioned already that the global relabeling heuristic has a cost of $\Theta(m)$ and that we want to apply it every $h \cdot m$ edge inspections for some constant h.

We therefore introduce two integer variables *limit_heur* and *heuristic*, initialize *heuristic* to $h \cdot m$, increment *limit_heur* by *heuristic* whenever the global relabel heuristic is applied, and apply the global relabel heuristic whenever the number of edge inspections exceeds *limit_heur*. Thus

⟨*MF_GRH: update distance label(s)*⟩≡

```
if (num_edge_inspections <= limit_heur)
  { ⟨MF_GRH: update the distance label of v⟩ }
else
  { limit_heur += heuristic;
    num_global_relabels++;
    ⟨MF_GRH: global relabel⟩
  }
```

and

⟨*MF_GRH: additional data structures*⟩+≡

```
int heuristic  = (int) (h*m);
int limit_heur = heuristic;
```

In order to update the distance label of v we increment *dmin* and then distinguish cases. If we are in phase one and *dmin* is at least n, we set *dist*[v] to n and do not insert v into the set of active nodes (since v cannot reach t in G_f anymore). In all other cases, we set *dist*[v] to *dmin* and insert v into U.

⟨*MF_GRH: update the distance label of v*⟩≡

```
dmin++; num_relabels++;
if ( phase_number == 1 && dmin >= n) dist[v] = n;
else { dist[v] = dmin;
       U.insert(v,dmin);
     }
```

A global relabel operation clears U and then distinguishes cases. In phase two the distance to s is recomputed for all nodes in S; recall that the nodes in $V \setminus S$ can reach t in G_f and hence are irrelevant for phase two.

In phase one we compute the distance from v to t in G_f for all nodes v. For nodes that cannot reach t we set the distance label to n. If no active node can reach t, phase one ends. We set S to all nodes that cannot reach t and then proceed as described above for phase two.

⟨*MF_GRH: global relabel*⟩≡

```
U.clear();
if (phase_number == 1)
{ dist.init(G,n);
  compute_dist_t(G,t,flow,cap,excess,dist,U,Q,count);
  if ( U.empty() )
  { node u;
    forall_nodes(u,G)
    { if (dist[u] == n)
      { S.append(u);
        dist[u] = max_level;
      }
    }
    phase_number = 2;
```

```
    compute_dist_s(G,s,flow,excess,dist,U,Q);
  }
}
else
{ node u;
  forall(u,S) dist[u] = max_level;
  compute_dist_s(G,s,flow,excess,dist,U,Q);
}
```

The function MAX_FLOW_GRH_T incorporates the distinction between low and high nodes, the local and the global relabel heuristic, and the distinction between phases one and two.

⟨*max_flow_GRH*⟩≡

```
template<class NT, class SET>
NT MAX_FLOW_GRH_T(const graph& G, node s, node t,
                  const edge_array<NT>& cap, edge_array<NT>& flow,
                  SET& U, int& num_pushes, int& num_edge_inspections,
                  int& num_relabels, int& num_global_relabels, float h)
{ if (s == t) error_handler(1,"MAXFLOW: source == sink");
  ⟨MF_GRH: initialization⟩
  ⟨MF_GRH: main loop⟩
#ifndef LEDA_CHECKING_OFF
  assert(CHECK_MAX_FLOW_T(G,s,t,cap,flow));
#endif
  return excess[t];
}
```

Table 7.13 shows that the combined effect of the global relabel heuristic and the two-phase approach is dramatic. The running times decrease considerably for all generators and for all three selection rules.

The Gap Heuristic: We come to our last optimization.

Consider a relabeling of a node v in phase one and let *dv* be the layer of v before the relabeling. If the layer *dv* becomes empty by the relabeling of v, then v cannot reach t anymore in G_f after the relabeling, since any edge crossing the now empty layer would be steep.

If v cannot reach t in G_f then no node reachable from v in G_f can reach t. We may therefore move v and all nodes reachable from v to layer n whenever the old layer of v becomes empty by the relabeling of v. This is called the *gap heuristic*.

We realize the heuristic as follows. For each d, $0 \leq d < n$ we keep a count of the number of nodes in layer d. For this purpose we use the array *count* introduced in the previous section.

The array *count* is recomputed in *compute_dist_t* and is updated whenever a node is relabeled. When a node v is moved from a layer *dv* to a layer *dmin*, we decrement *count*[*dv*] and increment *count*[*dmin*] (if *dv* or *dmin* is smaller than n).

When *count*[*dv*] is decremented to zero we move v and all nodes reachable from v in G_f

Gen	Rule	Pushes	Inspections	Relabels	GR	Time
rand	FIFO	7.377e+05	7.354e+06	5.794e+05	—	4.8
		6978	5.181e+04	4119	2	0.06
	HL	7.254e+05	7.749e+06	5.32e+05	—	4.82
		5.412e+04	5.264e+05	4.2e+04	21	0.43
	MFIFO	8.907e+05	7.498e+06	5.631e+05	—	4.5
		8048	5.171e+04	3918	2	0.06
CG1	FIFO	8.908e+05	2.789e+06	5.581e+05	—	2.87
		5.02e+05	5.05e+05	994	6	0.91
	HL	7.5e+05	4.373e+07	3.763e+05	—	20.92
		5.015e+05	5.045e+05	988	12	1.22
	MFIFO	6.713e+05	2.352e+06	4.619e+05	—	2.3
		5.02e+05	5.05e+05	994	6	0.91
CG2	FIFO	8.851e+06	3.807e+07	7.277e+06	—	37.29
		9.793e+05	9.939e+05	4710	9	1.76
	HL	6.265e+06	3.002e+07	5.504e+06	—	29.81
		1.928e+04	5.53e+04	6518	1	0.17
	MFIFO	1.019e+07	4.012e+07	7.16e+06	—	36.53
		5.033e+05	5.085e+05	1992	9	0.98

Table 7.13 Effect of low-high distinction, the local relabeling heuristic, the global relabeling heuristic, and the two-phase approach. We show the behavior for three different kinds of graphs and three different selection rules. For each generator we ran the case $n = 1000$. For the random graph generator we used $m = 3n$. For each case we give the running time of MAX_FLOW_LRH_T (first line) and of MAX_FLOW_GRH_T (second line). The savings are dramatic in all cases. The column GR shows the number of times the global relabeling heuristic was applied. The parameter h of MAX_FLOW_GRH_T was set to 5. Use max_flow_grh_time in the demo directory to perform your own experiments.

to layer n. We find these nodes by a breadth-first search starting in v. We reuse the queue Q, which we introduced for the distance calculations, for the breadth-first search.

⟨*MF_GAP: update the distance label of v*⟩≡

```
num_relabels++;
if (phase_number == 1)
{ if ( --count[dv] == 0 || dmin >= n - 1)
  { // v cannot reach t anymore
```

```
    ⟨move all vertices reachable from v to level n⟩
  }
  else
  { dist[v] = ++dmin; count[dmin]++;
    U.insert(v,dmin);
  }
}
else // phase_number == 2
{ dist[v] = ++dmin;
  U.insert(v,dmin);
}
```

Let us see the details of the breadth-first search. The layer *dmin* is the highest layer containing a node reachable from v. If this layer is less than *n*, we start the breadth-first search from v. We visit all nodes that are reachable from v in G_f and that live on a layer less than n. We move all such nodes to layer n. We count the number of nodes moved by the gap heuristic in *num_gaps*.

⟨*move all vertices reachable from v to level n*⟩≡

```
dist[v] = n;
if ( dmin < n )
{ Q.append(v);
  node w,z;
  while ( !Q.empty() )
  { edge e;
    w = Q.pop(); num_gaps++;
    forall_out_edges(e,w)
    { if ( flow[e] < cap[e] && dist[z = G.target(e)] < n)
      { Q.append(z);
        count[dist[z]]--; dist[z] = n;
      }
    }
    forall_in_edges(e,w)
    { if ( flow[e] > 0 && dist[z = G.source(e)] < n)
      { Q.append(z);
        count[dist[z]]--; dist[z] = n;
      }
    }
  }
}
```

The main loop has the same structure as before and only one change is required. When the gap heuristic moves a node to layer *n* it does not remove it from the set of active nodes (which it should because the node should stay inactive till the beginning of phase two). We remedy the situation as follows. Whenever a node on level *n* is removed from the set of active nodes in phase one we ignore the node and continue to the next iteration.

⟨*MF_GAP: main loop*⟩≡

```
for(;;)
{
  ⟨MF_GRH: extract v from queue⟩
  if (dist[v] == n && phase_number == 1) continue;
  NT  ev   = excess[v]; // excess of v
  int dv   = dist[v];   // level of v
  int dmin = MAXINT;
  edge e;
  if ( dist[v] < n ) { ⟨push across edges out of v⟩ }
  if ( ev > 0 ) { ⟨push across edges into v⟩ }
  excess[v] = ev;
  if (ev > 0) { ⟨MF_GAP: update distance label(s)⟩ }
}
```

⟨*MF_GAP: update distance label(s)*⟩≡

```
if (num_edge_inspections <= limit_heur)
  { ⟨MF_GAP: update the distance label of v⟩ }
else
  { limit_heur += heuristic;
    num_global_relabels++;
    ⟨MF_GRH: global relabel⟩
  }
```

Finally, we give the function MAX_FLOW_GAP_T a further parameter *num_gaps*, in which we count the number of nodes that are moved by the gap heuristic.

⟨*max_flow_GAP*⟩≡

```
template<class NT, class SET>
NT MAX_FLOW_GAP_T(const graph& G, node s, node t,
                  const edge_array<NT>& cap, edge_array<NT>& flow,
                  SET& U, int& num_pushes, int& num_edge_inspections,
                  int& num_relabels, int& num_global_relabels,
                  int& num_gaps, float h)
{ if (s == t) error_handler(1,"MAXFLOW: source == sink");
  ⟨MF_GRH: initialization⟩
  num_gaps = 0;
  ⟨MF_GAP: main loop⟩
#ifndef LEDA_CHECKING_OFF
  assert(CHECK_MAX_FLOW_T(G,s,t,cap,flow));
#endif
  return excess[t];
}
```

Table 7.14 shows the combined effect of all heuristics.

Gen	Rule	Pushes	Inspections	Relabels	GR	Gaps	Time
rand	FIFO	1.394e+04	1.036e+05	8154	2	—	0.18
		1.39e+04	1.036e+05	8142	2	2	0.17
	HL	1.911e+05	1.929e+06	1.444e+05	38	—	1.59
		2.536e+04	1.959e+05	1.258e+04	3	934	0.27
	MFIFO	1.589e+04	1.033e+05	7674	2	—	0.15
		1.589e+04	1.033e+05	7672	2	11	0.15
CG1	FIFO	2.002e+06	2.008e+06	1988	12	—	4.49
		2.002e+06	2.008e+06	1988	12	0	4.05
	HL	2.003e+06	2.009e+06	1975	25	—	5.41
		2.003e+06	2.009e+06	1975	25	0	5.67
	MFIFO	2.004e+06	2.01e+06	1988	12	—	3.64
		2.004e+06	2.01e+06	1988	12	0	4.08
CG2	FIFO	3.951e+06	3.971e+06	6846	18	—	8.85
		3.982e+06	3.992e+06	3983	18	2015	7.88
	HL	3.852e+04	1.106e+05	1.302e+04	1	—	0.36
		1.599e+04	4.396e+04	4002	0	3995	0.28
	MFIFO	2.079e+06	2.098e+06	6684	18	—	3.93
		2.001e+06	2.012e+06	3983	18	2017	4.27

Table 7.14 Effect of low-high distinction, the local relabeling heuristic, the global relabeling heuristic, the two-phase approach, and the gap heuristic. We show the behavior for three different kinds of graphs and three different selection rules. For each generator we ran the case $n = 2000$. For the random graph generator we used $m = 3n$. For each case we give the running time of MAX_FLOW_GRH_T (first line) and of MAX_FLOW_GAP_T (second line). The effect of the gap heuristic is small. The column GR shows the number of global relabels and the column Gaps shows the number of nodes moved by the gap heuristic. Use max_flow_gap_time in the demo directory to perform your own experiments.

Choice of H: How often should the heuristics be applied? Table 7.15 shows the behavior for different values of h. The choice of h does not have a big influence on running time. We have chosen $h = 5$ as the default value of h.

Summary and Implementation History: Table 7.16 summarizes our experiments. It shows the running times of our different implementations for four different kinds of graphs, three selection rules, and two different graph sizes ($n = 1000$ and $n = 2000$). The heuristics

Gen	Rule	h	Pushes	Inspections	Relabels	GR	Gaps	Time
rand	FF	0.5	9988	6.362e+04	4850	3	9	0.14
		2.5	1.18e+04	8.356e+04	6562	2	4	0.14
		4.5	1.6e+04	1.236e+05	9753	2	7	0.19
		6.5	1.989e+04	1.636e+05	1.287e+04	2	9	0.22
	HL	0.5	1.425e+04	8.442e+04	5506	16	1333	0.36
		2.5	1.967e+04	1.403e+05	9113	5	280	0.26
		4.5	2.563e+04	1.998e+05	1.28e+04	4	811	0.28
		6.5	2.347e+04	1.812e+05	1.18e+04	2	1279	0.25
	MF	0.5	1.112e+04	5.376e+04	3592	10	17	0.2
		2.5	1.328e+04	7.814e+04	5729	3	0	0.15
		4.5	1.476e+04	9.33e+04	6992	2	0	0.15
		6.5	1.956e+04	1.333e+05	9943	2	0	0.18
CG1	FF	0.5	1.992e+06	1.998e+06	1970	30	0	4.23
		2.5	1.996e+06	2.002e+06	1985	15	0	4.11
		4.5	2e+06	2.006e+06	1990	10	0	4.06
		6.5	2.004e+06	2.01e+06	1993	7	0	4.05
	HL	0.5	2.003e+06	2.009e+06	1750	250	0	8.6
		2.5	2.003e+06	2.009e+06	1950	50	0	5.67
		4.5	2.003e+06	2.009e+06	1973	27	0	5.33
		6.5	2.003e+06	2.009e+06	1981	19	0	5.21
	MF	0.5	2.004e+06	2.01e+06	1874	126	0	5.08
		2.5	2.004e+06	2.01e+06	1975	25	0	4.19
		4.5	2.004e+06	2.01e+06	1986	14	0	4.11
		6.5	2.004e+06	2.01e+06	1991	9	0	4.06

Table 7.15 Effect of the choice of h. We show the behavior for two different kinds of graphs and three different selection rules. For each generator we ran the case $n = 2000$. For the random graph generator we used $m = 3n$. For each case we give the running time of MAX_FLOW_GAP_T for different values of h. FF stands for FIFO and MF stands for MFIFO.

lead to dramatic savings in all cases, the global relabeling heuristic being the main source

Gen	Rule	BASIC	HL	LRH	GRH	GAP	LEDA
rand	FF	5.84	6.02	4.75	0.07	0.07	—
		33.32	33.88	26.63	0.16	0.17	—
	HL	6.12	6.3	4.97	0.41	0.11	0.07
		27.03	27.61	22.22	1.14	0.22	0.16
	MF	5.36	5.51	4.57	0.06	0.07	—
		26.35	27.16	23.65	0.19	0.16	—
CG1	FF	3.46	3.62	2.87	0.9	1.01	—
		15.44	16.08	12.63	3.64	4.07	—
	HL	20.43	20.61	20.51	1.19	1.33	0.8
		192.8	191.5	193.7	4.87	5.34	3.28
	MF	3.01	3.16	2.3	0.89	1.01	—
		12.22	12.91	9.52	3.65	4.12	—
CG2	FF	50.06	47.12	37.58	1.76	1.96	—
		239	222.4	177.1	7.18	8	—
	HL	42.95	41.5	30.1	0.17	0.14	0.08002
		173.9	167.9	120.5	0.3599	0.28	0.1802
	MF	45.34	42.73	37.6	0.94	1.07	—
		198.2	186.8	165.7	4.11	4.55	—
AMO	FF	12.61	13.25	1.17	0.06	0.06	—
		55.74	58.31	5.01	0.1399	0.1301	—
	HL	15.14	15.8	1.49	0.13	0.13	0.07001
		62.15	65.3	6.99	0.26	0.26	0.1399
	MF	10.97	11.65	0.04999	0.06	0.06	—
		46.74	49.48	0.1099	0.1301	0.1399	—

Table 7.16 The effect of the different heuristics. We show the behavior for four different kinds of graphs and three selection rules. For each generator we ran the cases $n = 1000$ and $n = 2000$. The last column stands for the default implementation in LEDA. It uses one further optimiziation which we have not explained in the text.

of improvement. You may use the program max_flow_summary_time in the demo directory to perform your own experiments.

Gen	Rule	GRH			GAP			LEDA		
rand	FF	0.16	0.41	1.16	0.15	0.42	1.05	—	—	—
	HL	1.47	4.67	18.81	0.23	0.57	1.38	0.16	0.45	1.09
	MF	0.17	0.36	1.06	0.14	0.37	0.92	—	—	—
CG1	FF	3.6	16.06	69.3	3.62	16.97	71.29	—	—	—
	HL	4.27	20.4	77.5	4.6	20.54	80.99	2.64	12.13	48.52
	MF	3.55	15.97	68.45	3.66	16.5	70.23	—	—	—
CG2	FF	6.8	29.12	125.3	7.04	29.5	127.6	—	—	—
	HL	0.33	0.65	1.36	0.26	0.52	1.05	0.15	0.3	0.63
	MF	3.86	15.96	68.42	3.9	16.14	70.07	—	—	—
AMO	FF	0.12	0.22	0.48	0.11	0.24	0.49	—	—	—
	HL	0.25	0.48	0.99	0.24	0.48	0.99	0.12	0.24	0.52
	MF	0.11	0.24	0.5	0.11	0.24	0.48	—	—	—

Table 7.17 The asymptotic behavior of our implementations. We show the behavior for four different kinds of graphs and three selection rules. For each generator we ran the cases $n = 5000 \cdot 2^i$ for $i = 0$, 1, and 2. For the random graph generator we used $m = 3n$. FF stands for FIFO and MF stands for MFIFO. You may use the program max_flow_large_time in the demo directory to perform your own experiments. The program max_flow_time in the demo directory times the default implementation.

The FIFO and MFIFO selection rule are superiour to the HL-rule on three of our four generators, although never by a large margin. However, on the generator CG2 both rules do very badly compared to the HL-rule. Figure 7.17 shows this even more clearly. For generators rand and AMO the running time seems to grow linearly (or maybe slightly more) for all three selection rules, for generator CG1 the running time seems to grow quadratically for all three selection rules, and for generator CG2 the running time seems to grow quadratically for the FIFO and the MFIFO-rule and seems to grow linearly for the HL-rule.

We have chosen the HL-rule as the default selection rule for our max flow algorithm. This is also what other researchers recommend [CG97, AKMO97].

The worst case running time of our max flow algorithm is $O(mdeg \cdot n^2\sqrt{m})$, where $mdeg$ is the maximal degree of any node. This can be improved to $O(n^2\sqrt{m})$ with the current edge data structure. Theoretically more efficient algorithms are known. Goldberg and Tarjan [GT88] have shown that the so-called dynamic tree data structure can be used to improve the running time of the preflow-push method to $O(nm \log n)$. In [CH95, CHM96] this was further improved to $O(nm+n^2 \log n)$. The dynamic tree data structure is available in LEDA. Monika Humble [Hum96] has implemented the preflow-push algorithm with the dynamic tree data structure. The observed running time was not impressive. Recently, Goldberg and

Rao [GR97] improved the running time to $O(\min(n^{2/3}, m^{1/2}m \log(n^2/m) logU)$, where U is the largest capacity of any edge (the capacities must be integral for their algorithm). It remains to be seen whether the improved bound also leads to better observed running times. A first experimental evaluation can be found in [HST98].

The first implementation of the preflow-push algorithm for LEDA was done by Cheriyan and Näher in 1989. It used the FIFO selection rule, the distinction between low and high nodes, and the local and global relabeling heuristic. Stefan Näher refined the implementation over the years and added the highest-level selection rule. For the book we added the two-phase approach, the gap heuristic, and the possibility of choosing the selection rule.

7.10.5 *Network Flow and Floating Point Arithmetic*

The preflow-push algorithm computes the maximum flow iteratively (and so do all other maximum flow algorithms). It starts with a preflow which it gradually transforms into a flow. The flow across any single edge is changed by pushes across the edge. These pushes may be in forward and backward direction, i.e., the flow across an edge is changed by additions and subtractions: the final flow across an edge is a sum of flow portions and these flow portions may be positive and negative.

What happens when the algorithm is executed with an arithmetic which may incur rounding error, e.g., floating point arithmetic? Then there may be cancellation in forming this sum. As a consequence the correctness of the algorithm is no longer guaranteed. The algorithm may not terminate or compute a function f which is not a flow (because it violates one of the constraints) or is a flow but not a maximal flow. Figure 7.45 shows an example of the disastrous effect that rounding error may have.

The preflow-push algorithm uses only additions and subtractions to manipulate flow and determines the flow to be sent across an edge as the maximum of the available excess and the residual capacity of the edge. This implies that all flow values are integral when the capacities are integral. Also the maximum excess of any node is bounded by D, where D is the sum of the capacities of the edges out of s.

If the number type *double* is used and all edge capacities are integral, there will be no overflow as long as $D < 2^{53}$. If the number type *double* is used and the edge capacities are not integral, we replace the edge capacities by

$$cap1[e] = sign(cap[e]) \lfloor |cap[e]| \cdot S \rfloor / S,$$

where S is the largest power of two such that $S < 2^{53}/D$, and apply the results of Section 7.2. They guarantee that there is no rounding error in the computation of the maximum flow with respect to *cap1* and that the value of the maximum flows with respect to *cap* and *cap1*, respectively, differ by at most $m \cdot D \cdot 2^{-52}$. The bound follows from the fact that the value of the maximum flow is equal to the capacity of a minimum cut, that the capacity of a minimum cut is the sum of at most m edge capacities and that the choice of S guarantees that for each edge the difference between the orginal capacity and the modified capacity is at most $D \cdot 2^{-52}$.

The paragraph above bounds the absolute error in the value of the flow resulting from

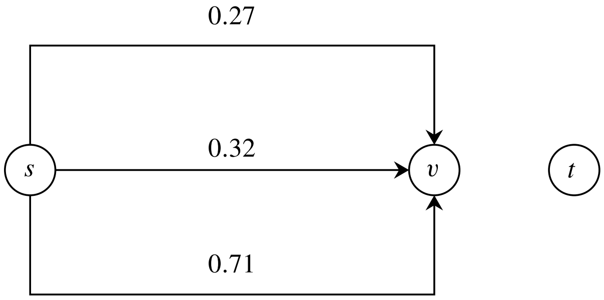

Figure 7.45 The effect of rounding error on the preflow-push algorithm: The capacities of the edges are as shown. The preflow-push algorithm starts by saturating all edges out of s. This will create an excess of $0.27 + 0.32 + 0.71 = 1.3$ in v. In the course of the execution, the algorithm will determine that none of this excess can be forwarded to t and hence the excess will be shipped back to s by sending 0.27, 0.32, and 0.71, respectively, across the three edges (v, s). The final excess in v is $1.3 - 0.27 - 0.32 - 0.71 = 0$.
Assume now that all calculations are carried out in a *floating point system with a mantissa of two decimal places and rounding by cut-off.* Then the excess in v after saturating all edges out of s will still be 1.3 as there is no cancellation in the summation. However, when the flow is pushed back to s the first subtraction $1.3 \ominus 0.29$ yields 1.1 as the last digit of 0.29 is dropped when the two summands are aligned for the subtraction; here $\ominus$ denotes floating point subtraction. The effect of this is that v ends up with an excess of 0.09, but no outgoing edge across which to push flow. This may put the algorithm into an infinite loop.

scaling. It does not bound the relative error. Observe that the quotient between D and the maximum flow may be arbitrarily large. Althaus and Mehlhorn [AM98] have shown that a slightly more elaborate scaling scheme can be used to bound the relative error. The idea is as follows. One modifies the edge capacities as described above and computes a maximum flow f with respect to them. Then

$$|val(f_{opt}) - val(f)| \leq m \cdot D \cdot 2^{-52},$$

where f_{opt} is a maximum flow with respect to the original edge capacities. One now distinguishes cases. If $m \cdot D \cdot 2^{-52} \ll val(f)$, the relative error in the value of the flow is small. Otherwise, let $B = val(f) + m \cdot D \cdot 2^{-52}$ and observe that $val(f_{opt}) \leq B$ and hence any capacity which is larger than B may be decreased to B without changing the maximum flow. Next they recompute D and S and repeat. After a smaller number of iterations the relative error will be small.

Exercises for 7.10

1 Let $G = (V, E)$ be a directed graph, let $cap : E \longrightarrow \mathbb{R}_{\geq 0}$ be a non-negative capacity function, and let $d : V \longrightarrow \mathbb{R}$ be a function with $\sum_{v \in V} d(v) = 0$. A node v with $d(v) > 0$ is called a *supply node*, a node v with $d(v) < 0$ is called a *demand node*, and d is called a demand function. A flow f is a function $f : E \longrightarrow \mathbb{R}_{\geq 0}$ satisfying the capacity constraints and the supply-demand constraints $excess(v) = d(v)$ for all $v \in V$. Design an algorithm that decides whether a flow exists and, if so, computes a flow. Hint:

Add two vertices s and t, an edge (s, v) with capacity $d(v)$ for every supply node, an edge (v, t) with capacity $-d(v)$ for every demand node, and compute a maximum (s, t)-flow.

2 The problem is as above but a lower bound $lb(e)$ on the flow across any edge e is also specified, i.e., for each edge two values $lb(e)$ and $ub(e)$ with $0 \leq lb(e) \leq ub(e)$are specified and the flow across any edge must lie between the lower and the upper bound. Hint: For any edge $e = (v, w)$ introduce two additional vertices a_e and b_e, replace e by the edges (v, a_e), (a_e, b_e), and (b_e, w), give a_e demand $-lb(e)$, give b_e supply $lb(e)$, and give (a_e, b_e) capacity $ub(e) - lb(e)$. Solve the problem above.

3 Show that the number of non-saturating pushes is $O(n^3)$ when the MFIFO-rule is used. Hint: Reuse the proof for the FIFO-rule.

4 Study alternative implementations of the highest-level-rule: *Insert*(v, d) and *insert0*(v, d) may add v to the front or the rear of the d-th list.

5 Incorporate the current edge data structure into our implementations.

6 Experiment with the global relabel heuristic but without the two-phase approach.

7.11 **Minimum Cost Flows**

The minimum cost maximum flow problem generalizes the maximum flow problem of the preceding section.

Let $G = (V, E)$ be a directed graph. For each edge $e \in E$ let $lcap(e)$ and $ucap(e)$ be lower and upper bounds for the flow across e (we assume $0 \leq lcap(e) \leq ucap(e)$) and let $cost(e)$ be the cost of shipping one unit of flow across e, and for each node v let $supply(v)$ be the supply or demand at node v. We talk about a supply if $supply(v) > 0$ and we talk about a demand if $supply(v) < 0$. We assume that the supplies and demands balance, i.e.,

$$\sum_{v \in V} supply(v) = 0.$$

A flow f is a function on the edges satisfying the capacity constraints and the mass balance conditions, i.e.,

$$lcap(e) \leq f(e) \leq ucap(e)$$

for every edge e and

$$supply(v) \quad = \sum_{e;\ source(e)=v} f(e) \quad - \sum_{e;\ target(e)=v} f(e)$$

for every node v.

For every edge e, $cost(e)$ is the cost of sending one unit of flow across the edge. The total cost of a flow f is therefore given by

$$cost(f) = \sum_{e \in E} f(e) \cdot cost(e).$$

A *minimum cost flow* is a flow of minimum cost. The function

```
bool MIN_COST_FLOW(graph& G, const edge_array<int>& lcap,
                             const edge_array<int>& ucap,
                             const edge_array<int>& cost,
                             const node_array<int>& supply,
                                   edge_array<int>& flow)
```

returns *true* if a flow exists and returns *false* otherwise. If a flow exists, it returns a minimum cost flow in *flow*. Observe that capacities and costs must be integers. The algorithm is based on capacity scaling and successive shortest-path computation (cf. [EK72] and [AMO93]) and has running time $O(m \log U(m + n \log n))$, where n is the number of nodes of G, m is the number of edges of G, and U is the largest absolute value of any capacity.

There is also a variant of this function where the lower bound on all flows is assumed to be zero.

```
bool MIN_COST_FLOW(graph& G, const edge_array<int>& cap,
                             const edge_array<int>& cost,
                             const node_array<int>& supply,
                                   edge_array<int>& flow);
```

The function

```
int MIN_COST_MAX_FLOW(graph& G, node s, node t,
                      const edge_array<int>& cap,
                      const edge_array<int>& cost,
                      edge_array<int>& flow)
```

computes a minimum cost maximal flow, i.e., it computes a maximal flow from s and t and among these flows a flow of minimum cost. The value of the flow is returned.

The xlman-demo gw_min_cost_flow illustrates minimum cost flows.

Exercises for 7.11

1 Consider an edge $e = (u, v)$ with $c = lcap(e) > 0$. Change the problem as follows: decrease $lcap(e)$ and $ucap(e)$ by c, decrease $supply(u)$ by c, and increase $supply(v)$ by c. Show that a solution to the modified problem yields a solution of the original problem.

2 Allow negative lower bounds. Describe a transformation that gets rid of negative lower bounds.

3 Assume that $lcap(e) = 0$ for all e. Introduce auxiliary nodes s and t and edges (s, v) with capacity $c = supply(v)$ for all nodes v with $supply(v) > 0$ and edges (u, t) with capacity $c = -supply(u)$ for all nodes u with $supply(u) < 0$. Show that there is a flow satisfying the capacity constraints and the bass balance constraints in the original network iff there is a flow from s to t in the modified network that saturates all edges out of s (and hence all edges into t). Based on this insight derive a necessary and sufficient condition for the existence of a flow satisfying the capacity constraints and the mass balance constraints.

4 Let f be a flow satisfying the capacity constraints and the mass balance constraints and let G_f be the residual network with respect to f. If $e = (v, w)$ is an edge in G with $f(e) < ucap(e)$ then there is an edge (v, w) in G_f with capacity $ucap(e) - f(e)$ and cost $cost(e)$ and if $e = (v, w)$ is an edge in G with $f(e) > lcap(e)$ then there is an edge

(w, v) in G_f with capacity $f(e) - lcap(e)$ and cost $-cost(e)$. Show that f is a minimum cost flow iff there is no negative cycle in G_f.

5 Derive a checker for minimum cost flows based on the preceding items.

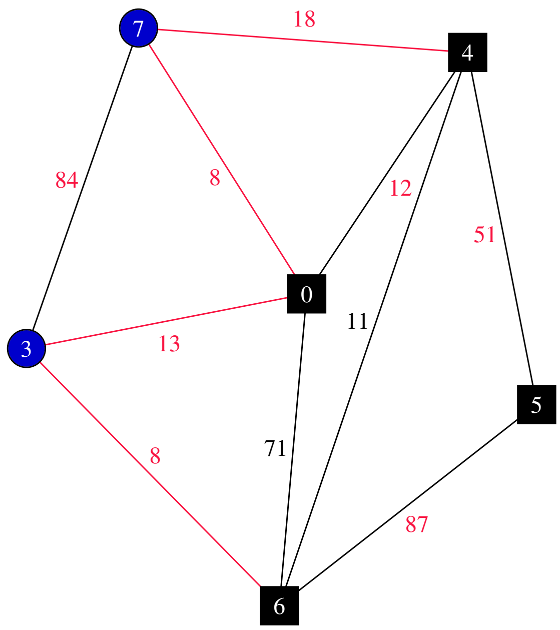

Figure 7.46 A minimum cut C in a graph. The nodes in C are shown as circles and the nodes outside C are shown as squares. The value of the cut is 47. You may generate your own figures with the xlman-demo gw_min_cut.

7.12 Minimum Cuts in Undirected Graphs

Let $G = (V, E)$ be an undirected graph (self-loops and parallel edges are allowed) and let $w : E \to \mathbb{R}_{\geq 0}$ be a *non-negative* weight function on the edges of G. A cut C of G is any subset of V with $\emptyset \neq C \neq V$. The weight of a cut is the total weight of the edges crossing the cut, i.e.,

$$w(C) = \sum_{e \in E; |e \cap C| = 1} w(e).$$

A *minimum cut* is a cut of minimum weight. Figure 7.46 shows an example. The function

```
int MIN_CUT(const graph& G,const edge_array<int>& weight,
            list<node>& C, bool use_heuristic = true)
```

	5000		10000		15000		20000	
	NOH	WH	NOH	WH	NOH	WH	NOH	WH
1000	9.22	3.52	17.11	17.11	27.86	29.36	38.88	39.46
2000	29.58	1.26	54.32	2.76	82.14	33.77	117.6	98.68
3000	62.51	3.71	107.2	3.64	145.6	8.76	191.1	85.17
4000	91.66	5.51	157	4.84	205.7	4.98	279.5	8.99
5000	144.2	15.62	213.5	11.8	273.8	11.7	378.6	18.22

Table 7.18 Running times of the minimum cut algorithms. We used random graphs with n nodes and m edges and random edge weights. The rows are indexed by n and the columns are indexed by m. For each combination of n and m we ran the algorithm without (NOH) and with the heuristic (WH). The use of the heuristic is the default.

takes a graph G and a *weight* function on the edges and computes a minimum cut. The value of the cut is returned and the nodes in the cut are assigned to C. The running time of the algorithm is $O(nm + n^2 \log n)$. The algorithm is due to [NI92, SW97]. The algorithm can be asked to use a heuristic. In some cases the heuristic improves the running time dramatically; it never seems to harm, see Table 7.18. There is also a version of the function where the cut C is the return value of the function.

```
list<node> MIN_CUT(const graph& G, const edge_array<int>& weight)
```

The function

```
int CUT_VALUE(const graph& G,const edge_array<int>& weight,
              const list<node>& C)
```

returns the value of the cut C.

We use a particularly simple and nevertheless efficient min-cut algorithm due to Nagamochi and Ibaraki [NI92] and later refined by Stoer and Wagner [SW97]. The algorithm runs in time $O(nm + n^2 \log n)$. Alternative minimum cut algorithms can be found in [PR90, HO92, KS96]. The papers [CGK$^+$97, JRT97] contain experimental comparisons of minimum cut algorithms.

We need the notion of an s-t cut. For a pair $\{s, t\}$ of distinct vertices of G a cut C is called an s-t cut if C contains exactly one of s and t.

The algorithm works in phases. In each phase it determines a pair of vertices s and t and a minimum s-t cut C. If there is a minimum cut of G separating s and t then C is a minimum cut of G. If not then any minimum cut of G has s and t on the same side and therefore the graph obtained from G by *combining* s and t has the same minimum cut as G. So a phase determines vertices s and t and a minimum s-t cut C and then combines s and t

into one node. After $n-1$ phases the graph is shrunk to a single node and one of the phases must have determined a minimum cut of G.

```
⟨min_cut⟩≡
  ⟨combine s and t⟩
  int MIN_CUT(const graph& G0, const edge_array<int>& weight,
              list<node>& C, bool use_heuristic)
  { node v; edge e;
    forall_edges(e,G0)
      if ( weight[e] < 0 )
        error_handler(1,"MIN_CUT: no negative weights");
    ⟨initialization⟩
    while ( G.number_of_nodes() >= 2 ) { ⟨a phase⟩ }
    return best_value;
  }
```

We call our input graph *G0* and our current Graph G. Every node of G represents a set of nodes of *G0*. This set is stored in a linear list pointed to by $G[v]$ and hence we use the type *GRAPH<list<node>∗, int>* for G. Every edge $e = \{v, w\}$ of G represents a set of edges of *G0*, namely $\{\{x, y\}\,;\, x \in G[v] \text{ and } y \in G[w]\}$. The total weight of these edges is stored in $G[e]$.

It is easy to initialize G. We simply make G a copy of *G0* (except for self-loops) and initialize $G[v]$ to the appropriate singleton set for every vertex v of G.

```
⟨initialization⟩≡
  typedef list<node>* nodelist_ptr;
  GRAPH<nodelist_ptr, int> G;
  G.make_undirected();
  node_array<node> partner(G0);
  forall_nodes(v,G0)
  { partner[v] = G.new_node(new list<node>);
    G[partner[v]]->append(v);
  }
  forall_edges(e, G0)
  if ( source(e) != target(e) )
    G.new_edge(partner[source(e)], partner[target(e)],weight[e]);
```

We also fix a particular node a of G and introduce variables to store the currently best cut.

```
⟨initialization⟩+≡
  node a = G.first_node();
  int best_value = MAXINT;
  int cut_weight = MAXINT;
```

We now come to the heart of the matter, a phase. A phase initializes a set A to the singleton set $\{a\}$ and then successively merges all other nodes of G into A. In each stage the node

$v \notin A$ which maximizes

$$w(v, A) = \sum_{e;\ e=\{v,y\} \text{ for some } y \in A} w(e)$$

is merged into A. Let s and t be the last two vertices added to A in a phase. The cut C computed by the phase is the cut consisting of node t only; in the graph *G0* this corresponds to the cut $G[t]$.

Lemma 44 *Let s and t be the last two nodes merged into A during a phase. Then $\{t\}$ is a minimum s-t cut.*

Proof Let C' be any s-t cut. We show that $w(C') \geq w(\{t\})$. Let $v_1, \ldots, v_n$ be the order in which the nodes are added to A. Then $v_1 = a$, $v_{n-1} = s$, and $v_n = t$.

Call a vertex $v = v_i$ critical if $i \geq 2$ and v_i and v_{i-1} belong to different sides of C'. Note that t is critical. Let k be the number of critical nodes and let i_1, i_2,..., i_k be the indices of the critical nodes. Then $i_k = n$. For integer i use A_i to denote the set $\{v_1, \ldots, v_i\}$. Then

$$w(\{t\}) = w(v_{i_k}, A_{i_k - 1})$$

and

$$w(C') \geq \sum_{j=1}^{k} w(v_{i_j}, A_{i_j - 1} \setminus A_{i_{j-1} - 1}),$$

since any edge counted on the right side is also counted on the left and edge costs are non-negative. We now show for all integers l, $1 \leq l \leq k$, that

$$w(v_{i_l}, A_{i_l - 1}) \leq \sum_{j=1}^{l} w(v_{i_j}, A_{i_j - 1} \setminus A_{i_{j-1} - 1}).$$

For $l = 1$ we have equality. So assume $l \geq 2$. We have

$$\begin{aligned}
w(v_{i_l}, A_{i_l - 1}) &= w(v_{i_l}, A_{i_{l-1} - 1}) + w(v_{i_l}, A_{i_l - 1} \setminus A_{i_{l-1} - 1}) \\
&\leq w(v_{i_{l-1}}, A_{i_{l-1} - 1}) + w(v_{i_l}, A_{i_l - 1} \setminus A_{i_{l-1} - 1}) \\
&\leq \sum_{j=1}^{l-1} w(v_{i_j}, A_{i_j - 1} \setminus A_{i_{j-1} - 1}) + w(v_{i_l}, A_{i_l - 1} \setminus A_{i_{l-1} - 1}) \\
&\leq \sum_{j=1}^{l} w(v_{i_j}, A_{i_j - 1} \setminus A_{i_{j-1} - 1}).
\end{aligned}$$

Here the first inequality follows from the fact that $v_{i_{l-1}}$ is added to $A_{i_{l-1} - 1}$ and not v_{i_l} and the second inequality uses the induction hypothesis. □

⟨*a phase*⟩≡

⟨*determine s and t and the value of the cut V-t,t*⟩;

```
bool new_best_cut = false;
if ( cut_weight < best_value )
```

```
{ C = *(G[t]);
  best_value = cut_weight;
  new_best_cut = true;
}
combine_s_and_t(G,s,t);
⟨heuristic⟩
```

How can we determine the order in which the vertices are merged into A? This can be done in a manner akin to Prim's minimum spanning tree algorithm. We keep the vertices v, $v \notin A$, in a priority queue ordered according to $w(v, A)$. In each stage we select the node, say u, with maximal $w(u, A)$ and add it to A. This increases $w(v, A)$ by $w(\{v, u\})$ for any vertex $v \notin A$ and $v \neq u$. Since LEDA priority queues select minimal values we store $-w(v, A)$ in the queue. The node added last to A is the vertex t. The value *cut_weight* is $w(t, A_t)$.

⟨*determine s and t and the value of the cut V-t,t*⟩≡

```
node t = a;
node s;
node_array<bool> in_PQ(G,false);
node_pq<int> PQ(G);
forall_nodes(v,G)
if (v != a)
{ PQ.insert(v,0);
  in_PQ[v] = true;
}
forall_adj_edges(e,a)
  PQ.decrease_inf(G.opposite(a,e),PQ.prio(G.opposite(a,e)) - G[e]);
while (!PQ.empty())
{ s = t;
  cut_weight =  -PQ.prio(PQ.find_min());
  t = PQ.del_min();
  in_PQ[t] = false;
  forall_adj_edges(e,t)
  { if (in_PQ[v = G.opposite(t,e)])
      PQ.decrease_p(v,PQ.prio(v) - G[e]);
  }
}
```

It remains to combine s and t. We do so by deleting t from G and moving all edges incident to t to s. More precisely, we need to do three things:

- Add $G[t]$ to $G[s]$ ($G[s] \rightarrow conc(*(G[t]))$).
- Increase $G[\{s, v\}]$ by $G[\{t, v\}]$ for all vertices v with $\{t, v\} \in E$ and $v \neq s$.
- Delete t and all its incident edges from G (*G.del_node*(t)).

The second step raises two difficulties: the edge $\{s, v\}$ might not exist and there is no simple way to go from the edge $\{t, v\}$ to the edge $\{s, v\}$. We overcome these problems by first recording the edge $\{s, v\}$ in *s_edge*$[v]$ for every neighbor v of s. We then go through the neighbors v of t: if v is connected to s then we simply increase $G[\{s, v\}]$ by $G[\{t, v\}]$, if v is not connected to s and different from s then we add a new edge $\{s, v\}$ with weight $G[\{t, v\}]$.

We formulate the piece of code to combine s and t as a procedure because we want to reuse it in the heuristic.

⟨*combine s and t*⟩≡

```
static void combine_s_and_t(GRAPH<list<node>*,int>& G, node s, node t)
{ G[s]->conc(*(G[t]));
  node_array<edge> s_edge(G,nil);
  edge e;
  forall_adj_edges(e,s)  s_edge[G.opposite(s,e)] =  e;
  forall_adj_edges(e,t)
  { node v = G.opposite(t,e);
    if ( v == s) continue;
    if (s_edge[v] == nil) G.new_edge(s,v,G[e]);
    else G[s_edge[v]] += G[e];
  }
  G.del_node(t);
}
```

This completes the description of the algorithm. The running time of our algorithm is clearly at most n times the running time of a phase. A phase takes time $O(m + n \log n)$ to merge all nodes into the set A (the argument is the same as for Prim's algorithm) and time $O(n)$ to record the cut computed and to merge s and t. The total running time is therefore $O(nm + n^2 \log n)$.

We next discuss a heuristic improvement. Clearly, any edge whose weight is at least *best_value* cannot cross a minimum cut whose value is smaller than *best_value*. We therefore might as well shrink any such edge.

Which edges might have weight at least as large as *best_value*? If *best_value* decreased in the current phase, then all edges of G are candidates, and if *best_value* stayed unchanged in the current phase, then all edges incident to s are candidates, because their weight may have increased .

⟨*heuristic*⟩≡

```
if ( use_heuristic )
{ bool one_more_round = true;
  while ( one_more_round )
  { one_more_round = false;
    forall_adj_edges(e,s)
    { node t = G.opposite(s,e);
      if ( G[e] >= best_value )
      { combine_s_and_t(G,s,t); one_more_round = true; break; }
    }
  }
  if ( new_best_cut )
```

```
  { bool one_more_round = true;
    while ( one_more_round )
    { one_more_round = false;
      forall_edges(e,G)
      { node s = G.source(e);
        node t = G.target(e);
        if ( G[e] >= best_value )
        { combine_s_and_t(G,s,t); one_more_round = true; break; }
      }
    }
  }
}
```

Table 7.18 shows that the heuristic can lead to dramatic improvements in running time. We will now argue that is does increase the asymptotic running time. If the phase did not decrease *best_value*, the running time of the heuristic is $O((1+k)n)$, where k is the number of edges shrunken by the heuristic. If the phase decreased *best_value*, the running time of the heuristic is $O((1+k)m)$, where k is the number of edges shrunken by the heuristic. In either case the asymptotic running time of our procedure is not increased, since a phase has cost $\Omega(m + n \log n)$.

We considered an alternative implementation of the heuristic. We kept the edges of G in a priority queue according to negative weight and at the end of each phase selected all edges from the queue which had weight at least as large as *best_value*. The alternative implementation was slower than the simple implementation described above.

Bibliography

[ABMP91] H. Alt, N. Blum, K. Mehlhorn, and M. Paul. Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}\sqrt{m/\log n})$. *Information Processing Letters*, 37(4):237–240, 1991.

[AC93] D. Applegate and W. Cook. Solving large-scale matching problems. In D. Johnson and C.C. McGeoch, editors, *Network Flows and Matchings*, volume 12 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 557–576. American Mathematical Society, 1993.

[AKMO97] R.K. Ahuja, M. Kodialam, A.K. Mishra, and J.B. Orlin. Computational investigation of maximum flow algorithms. *European Journal on Operational Research*, 97:509–542, 1997.

[AM98] E. Althaus and K. Mehlhorn. Maximum network flow with floating point arithmetic. *Information Processing Letters*, 66:109–113, 1998.

[AMO93] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows*. Prentice Hall, 1993.

[AO89] R.K. Ahuja and J.B. Orlin. A fast and simple algorithm for the maximum flow problem. *Operation Research*, 37:748–759, 1989.

[AST94] P.K. Agarwal, M. Sharir, and S. Toledo. Applications of parametric searching in geometric optimization. *Journal of Algorithms*, 17:292–318, 1994.

[Bel58] R.E. Bellman. On a routing problem. *Quart. Appl. Math.*, 16:87–90, 1958.

[BG61] R.G. Busacker and P.J. Gowen. A procedure for determining minimal-cost network flow patterns. Technical report, Operations Research Office, John Hopkins University, 1961.

[CFMP97] C. Cooper, A. Frieze, K. Mehlhorn, and V. Priebe. Average-case complexity of shortest-paths problems in the vertex-potential model. In José Rolim, editor, *Proceedings of the International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM'97)*, volume 1269 of *Lecture Notes in Computer Science*, pages 15–26. Springer, 1997.

[CG96] B.V. Cherkassky and A.V. Goldberg. Negative cycle detection algorithms. In *Proceedings of the 4th Annual European Symposium on Algorithms - ESA'96*, volume 1136 of *Lecture Notes in Computer Science*, pages 349–363, 1996.

[CG97] B.V. Cherkassky and A.V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.

[CGK+97] C.S. Chekuri, A.V. Goldberg, D.R. Karger, M.S. Levine, and C. Stein. Experimental study of minimum cut algorithms. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'97)*, pages 324–333, New Orleans, Louisiana, 5–7 January 1997.

[CGM+97] B. Cherkassky, A. Goldberg, P. Martin, J. Setubal, and J. Stolfi. Augment or relabel? A computational study of bipartite matching and unit capacity maximum flow algorithms. Technical Report TR 97-127, NEC Research Institute, 1997.

[CGR94] B.V. Cherkassky, A.V. Goldberg, and T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. In Daniel D. Sleator, editor, *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'94)*, pages 516–525. ACM Press, 1994.

[CH95] J. Cheriyan and T. Hagerup. A randomized maximum-flow algorithm. *SIAM Journal of Computing*, 24(2):203–226, April 1995.

[CHM96] J. Cheriyan, T. Hagerup, and K. Mehlhorn. An $o(n^3)$-time maximum flow algorithm. *SIAM Journal of Computing*, 25(6):1144–1170, 1996.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill Book Company, 1990.

[CM89] J. Cheriyan and A. Maheshwari. Analysis of preflow push algorithms for maximum network flow. *SIAM Journal of Computing*, 18:1057–1086, 1989.

[CM96] J. Cheriyan and K. Mehlhorn. Algorithms for dense graphs and networks on the random access computer. *Algorithmica*, 15(6):521–549, 1996.

[CM99] J. Cheriyan and K. Mehlhorn. An analysis of the highest-level selection rule in the preflow-push max-flow algorithm. *IPL*, 69:239–242, 1999. `www.mpi-sb.mpg.de/~mehlhorn/ftp/maxflow.ps`.

[Din70] E.A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Mathematics Doklady*, 11:1277–1280, 1970.

[Edm65a] J. Edmonds. Maximum matching and a polyhedron with 0,1 - vertices. *Journal of Research of the National Bureau of Standards*, 69B:125–130, 1965.

[Edm65b] J. Edmonds. Paths, trees, and flowers. *Canadian Journal on Mathematics*, pages 449–467, 1965.

[EK72] J. Edmonds and R.M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248–264, 1972.

[FF63] L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, 1963.

[Fin97] U. Finkler. *Design of Efficient and Correct Algorithms: Theoretical Results and Runtime Prediction of Programs in Practice*. PhD thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1997.

[Gab76] H.N. Gabow. An efficient implementation of Edmond's algorithm for maximum matching on graphs. *Journal of the ACM*, 23:221–234, 1976.

[Gal86] Z. Galil. Efficient algorithms for finding maximum matching in graphs. *ACM Computing Surveys*, 18(1):23–37, 1986.

[GMG86] Z. Galil, S. Micali, and H.N. Gabow. An $O(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs. *SIAM Journal of Computing*, 15:120–130, 1986.

[Gol85] A.V. Goldberg. A new max-flow algorithm. Technical Report MIT/LCS/TM-291, Lab. for Computer Science, MIT, Cambridge, Mass., 1985.

[GR97] A.V. Goldberg and S. Rao. Beyond the flow decomposition barrier. In *Proceedings of the 29th Annual ACM Symposium on the Theory of Computing (STOC'97)*, 1997.

[GT88] A.V. Goldberg and R.E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35:921–940, 1988.

[HK73] J.E. Hopcroft and R.M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal of Computing*, 2(4):225–231, 1973.

[HO92] J. Hao and J.B. Orlin. A faster algorithm for finding the minimum cut in a graph. In *Proceedings on the 3rd Annual Symposium on Discrete Algorithms (SODA'92)*, volume 3, pages 165–174. ACM/SIAM, 1992.

[HST98] T. Hagerup, P. Sanders, and J. Träff. An implementation of the binary blocking flow algorithm. In *Proceedings of the 2nd Workshop on Algorithm Engineering (WAE'98)*, pages 143–154. Max-Planck-Institut für Informatik, 1998.

[Hum96] M. Humble. Implementierung von Flußalgorithmen mit dynamischen Bäumen. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1996.

[Iri60] M. Iri. A new method for solving transportation-network problems. *Journal of the Operations Research Society of Japan*, 3:27–87, 1960.

[Jew58] W.S. Jewell. Optimal flow through networks. Technical report, Operations Research Center, MIT, 1958.

[JRT97] M. Jünger, G. Rinaldi, and S. Thienel. Practical performance of efficient minimum cut algorithms. Technical report, Institut für Informatik, Universität zu Köln, 1997.

[Kar74] A.V. Karzanov. Determining the maximum flow in a network by the method or preflows. *Soviet Mathematics Doklady*, 15:434–437, 1974.

[KP98] J.D. Kececioglu and J. Pecqueur. Computing maximum-cardinality matchings in sparse general graphs. In *Proceedings of the 2nd Workshop on Algorithm Engineering (WAE'98)*, pages 121–132. Max-Planck-Institut für Informatik, 1998.

[KS96] D.R. Karger and C. Stein. A new approach to the minimum cut problem. *Journal of the ACM*, 43(4):601–640, 1996.

[Kuh55] H.W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[Law66] E.L. Lawler. Optimal cycles in doubly weighted directed linear graphs. In *Theory of Graphs: International Symposium*, pages 209–213. Dunod, Paris, 1966.

[Law76] E.L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston, 1976.

[MCN91] J.-F. Mondou, T.G. Crainic, and S. Nguyen. Shortest path algorithms: A computational study with the C programming language. *Computers and Operations Research*, 18:767–786, 1991.

[Meg83] N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *Journal of the ACM*, 30:852–865, 1983.

[Meh84] K. Mehlhorn. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*. Springer, 1984.

[Mot94] R. Motwani. Average-case analysis of algorithms for matching and related problems. *Journal of the ACM*, 41(6):1329–1356, 1994.

[NI92] H. Nagamochi and T. Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM Journal on Discrete Mathematics*, 5(1):54–66, 1992.

[Pau89] M. Paul. Algorithmen für das Maximum Weight Matching Problem in bipartiten Graphen. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1989.

[PR90] M. Padberg and G. Rinaldi. An efficient algorithm for the minimum capacity cut problem. *Mathematical Programming*, 47:19–36, 1990.

[Sha81] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computational Mathematics with Applications*, 7(1):67–72, 1981.

[SSS97] S. Schirra, J. Schwerdt, and M. Smid. Computing the minimum diameter for moving points: An exact implementation using parametric search. In *Proceedings of the 13th Annual ACM Symposium on Computational Geometry (SCG'97)*, pages 466–468, 1997.

[SW97] M. Stoer and F. Wagner. A simple min-cut algorithm. *Journal of the ACM*, 44(4):585–591, July 1997.

[Tar72] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1:146–160, 1972.

[Tar81] R.E. Tarjan. Shortest paths. Technical report, AT&T Bell Laboratories, Murray Hill, New Jersey, 1981.

[Tar83] R.E. Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.

[Tom71] N. Tomizawa. On some techniques useful for solution of transportation network. *Networks*, 1:173–194, 1971.

[Zie95] T. Ziegler. Max-Weighted-Matching auf allgemeinen Graphen. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1995.

Index