0.1 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,
- 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 0.1 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 0.1.2 and 0.1.3 discuss an algorithm for maximum weight matchings and its implementation. In Sections 0.1.4 and 0.1.6 we modify our algorithms to compute assignments and maximum weight matchings of maximum cardinality. Finally, in Section 0.1.5 we show how to reduce the shortest path problem to the assignment problem.

0.1.1 ***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 **??** 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)
```

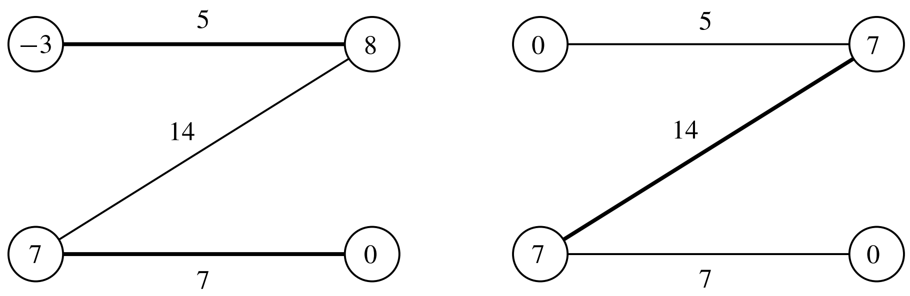

Figure 0.1 Maximum weight assignment and maximum weight matching. The matching on the left is a maximum weight perfect matching and the matching on the right is a maximum weight matching; the edges in the matchings are shown in bold in both cases. A potential function that proves the optimality of the 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 1) 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.

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 0.1.6.

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 1 *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 1 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

$$\begin{array}{rcll} \sum_{e;\, e \text{ is incident to } v} x(e) & \leq & 1 & \text{for all } v \in V \\ x(e) & \in & \{0,1\} & \text{for all } e \in E. \end{array}$$

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 $x(e) \geq 0$ without changing the problem[1]. Assume that

[1] We sketch a proof of this fact. We first observe that the non-negativity constraints $x(e) \geq 0$ together with the matching constraints $\sum_{e;e \text{ is incident to } v} x(e) \leq 1$ guarantee $0 \leq x(e) \leq 1$. 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

the integrality constraint $x(e) \in \{0, 1\}$ has been replaced by $x(e) \geq 0$. 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

$$\begin{array}{rcll} c(e) & \leq & \pi(v) + \pi(w) & \text{for all } e = (v, w) \in E \\ \pi(v) & \geq & 0 & \text{for all } v \in V. \end{array}$$

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 1 is simply an adaption of the standard proofs of weak linear programming duality and complementary slackness to matchings.

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 **??** 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

$$S < 2^{53}/(f \cdot C).$$

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 $+1$.

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.

0.1.2 ***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 $\pi_i : V_i \mapsto \mathbb{R}_{\geq 0}$ 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.

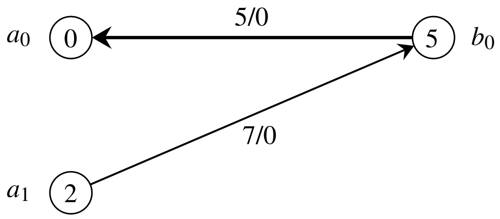

Figure 0.2 The edges of a matching M are shown in bold. The potential of each node is shown inside the node and the cost c and the reduced cost $\overline{c}$ of each edge is shown as $c(e)/\overline{c}(e)$. The path a_1, b_0, a_0 consists of tight edges and can be used for augmentation. The resulting matching has the same cardinality as the current matching.

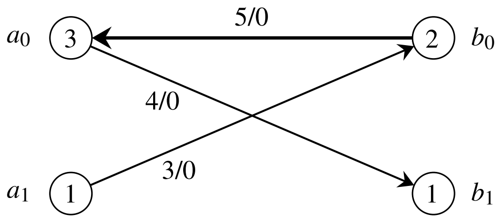

Figure 0.3 The edges of a matching M are shown in bold. The potential of each node is shown inside the node and the cost c and the reduced cost $\overline{c}$ of each edge is shown as $c(e)/\overline{c}(e)$. The path a_1, b_0, a_0, b_1 consists of tight edges and can be used for augmentation. The resulting matching has cardinality one larger than the current matching.

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 0.2, 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 0.3, 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 0.4. 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 2 *In the first two cases, π is tight with respect to $M \oplus p$. In the third case, the*

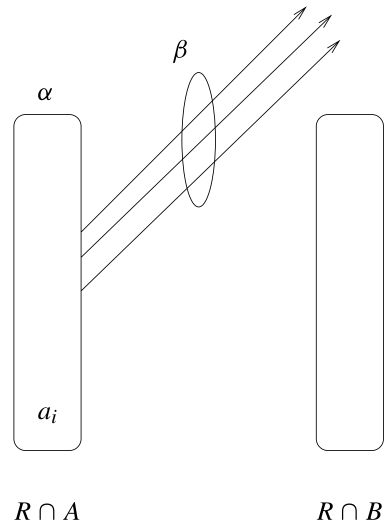

Figure 0.4 R is the set of nodes reachable from a_i by tight edges. The sets $A \cap R$ and $B \cap R$ are indicated as large ovals. The number of nodes in $A \cap R$ is one larger than the number of nodes in $B \cap R$, since each node in $A \cap R \setminus a_i$ is matched to a node in $B \cap R$ and 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.

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 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 $R \cap B$; 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 ∈ 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 3 *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 $\mu(w)$. □

Lemma 4 *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* $= \min\{\mu(b)\ ;\ b \in B$ *and b 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 3 and 4 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 $minA$;

$minB = \min\{dist(v)\ ;\ v \in B \text{ and free}\}$;
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.

0.1.3 *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 **??**.

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 0.1.2.

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 0.1.2).

⟨*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 **??**. 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 0.1 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 5 *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 0.1 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 0.5. 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 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 reduced cost one. When the first neighbor of a_i is removed from the queue (with distance

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 0.1 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.

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 0.5. 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

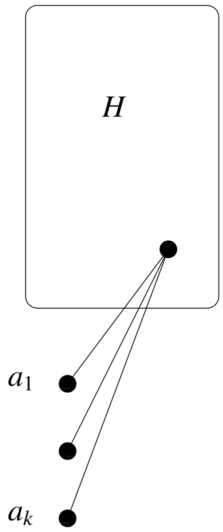

Figure 0.5 H has a perfect matching and each a_i, $1 \le i \le k$ is connected to some node on the B-side of H. After the search for an augmenting path from a_1 all nodes in B will have potential one. The searches from a_i, $i > 2$, take constant time.

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 0.1.2. 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

$$\begin{aligned}
d(p) &= \sum_{j;\, j \text{ even}} (\pi(v_{j-1}) + \pi(v_j)) - \sum_{j;\, j \text{ odd}} c(e_j) \\
&= -\pi(a) + \sum_{j;\, j \text{ odd}} (\pi(v_{j-1}) + \pi(v_j) - c(e_j)) + (-1)^k \pi(v_k) \\
&= -\pi(a) + \sum_{j;\, j \text{ odd}} \overline{c}(e_j) + (-1)^k \pi(v_k) \;=\; -\pi(a) + \sum_j \overline{c}(e_j) + (-1)^k \pi(v_k) \\
&= -\pi(a) + \sum_j \overline{c}(e_j) + \pi(v_k).
\end{aligned}$$

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

$$\begin{aligned}&-c(e_1) + (\pi(v_1) + \pi(v_2)) - c(e_3) = \\ &\qquad -\pi(a) + (\pi(v_0) - c(e_1) + \pi(v_1)) + (\pi(v_2) - c(e_3) + \pi(v_3)) - \pi(v_3).\end{aligned}$$

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 [Tom72] 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.

0.1.4 *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[2] 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 $s \le 2k$. We claim that $\pi(v_i) \in [-iC \mathinner{..} iC]$ 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 $[-C \mathinner{..} C]$. We conclude that $\pi(a) \in [-(2k-1)C \mathinner{..} (2k-1)C]$ for $a \in A$ and $\pi(b) \in [-(2k-2)C \mathinner{..} (2k-2)C]$ 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 $C + (2k-1)C + (2k-2)C \le (4k-2)C$.

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 6 *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 $n = 2k$.*

0.1.5 ***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 the shortest-path problem with arbitrary edge weights. This will give us an alternative to the algorithms in Section **??**. 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 0.2.

Let $G = (V, E)$ be a directed graph. We construct a bipartite network $G' = (V' \mathbin{\dot\cup} V'', E')$; see Figure 0.6 for an illustration. G' contains two copies of each node of G, one in

[2] 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 0.2 A comparison of the running time of the shortest path via assignment algorithm (denoted A) with the shortest-path algorithms of Section **??**. 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.

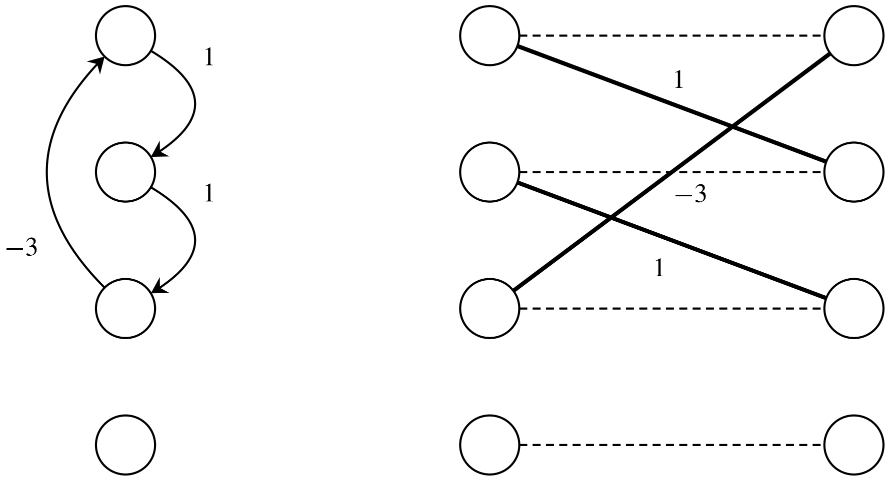

Figure 0.6 A directed graph and the derived bipartite graph. All dashed edges in the graph on the right have cost zero. The dashed edges define a perfect matching of cost zero. The solid edges together with the lowest dashed edge define a perfect matching of negative cost. It corresponds to the negative cycle in the graph on the left.

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 7 *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 $v_0 \ne v_1$; 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. □

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[3] 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

$$\mu(v) = -\pi(s) + \overline{\mu}(v) + \pi(v),$$

see Section **??**.

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,
```

[3] It is possible that one of the edges (v', v'') is not contained in M. How?

```
                                        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;
}
```

0.1.6 ***Maximum Weighted Matchings of Maximum Cardinality***

We show how to compute a matching of maximum weight among the matchings of maximum cardinality[4]. 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.

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

[4] For graphs that have a perfect matching this is the same as looking for a maximal weight perfect matching.

of maximum cardinality we only have to find a maximum weight matching with respect to the cost function c_L where $L = 2kC + 1$.

⟨*mwb_matching.t*⟩+≡
```
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 $C \geq 1$ and $k \geq 1$ we have $3 + (6k + 3)C \leq 4nC$.

Exercises for 0.1

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 1 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.

Bibliography

[AMO93] R. K. Ahuja, T. L. Magnanti, and J.B. Orlin. *Network Flows*. Prentice Hall, 1993.

[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.

[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 J. Math.*, 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.

[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.

[Gal86] Z. Galil. Efficient algorithms for finding maximum matching in graphs. *ACM Computer Surveys*, 18, 1:23–37, 1986.

[Iri60] M. Iri. A new method for solving transportation-network problems. *Journal of the Operations Research Society of Japan*, 2:27–87, 1960.

[Jew58] W.S. Jewell. Optimal flow through networks. Technical report, Operations Research Center, MIT, 1958.

[Kuh55] H.W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[Pau89] M. Paul. Algorithmen für das Maximum Weight Matching Problem in bipartiten Graphen. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1989.

[Tom72] N. Tomizawa. On some techniques useful for solution of network flow problems. *Networks*, 1:173–194, 1972.

Index