Bipartite Cardinality Matching
Worst Case Generators
and
An Improved Implementation of the ABMP-algorithm

February 2, 2000

0.1 Summary

We give generators that force bipartite matching algorithms into their worst case. The generators are robust in the sense that the worst case running time is assumed on a random representation of the generator. We also improve the implementation of the ABMP-algorithm in two respects. We give a rational choice for the constants required by the algorithm (in the book we had set the constants to rather arbitrary values) and we add a heuristic to the algorithm which removes all nodes that cannot lie on augmenting paths anymore. The heuristic increases running time at most by a few percent and sometimes leads to dramatic improvements.

⟨*additions to h-files*⟩≡

```
// the following declarations should go into the appropriate h-files

void permute(graph& G);

void mcb_worst_case_gen(graph& G, int n, int m, list<node>& A, list<node>& B);

list<edge> MAX_CARD_BIPARTITE_MATCHING_ABMP_NEW(graph& G,
                              const list<node>& A,
                              const list<node>& B,
                              node_array<bool>& NC,
                              bool use_pruning = true,
                              bool use_heuristic = true);
```

TODO; make these additions.

0.2 Worst Case Generators

The goal of this section is to design a class of graphs which force our bipartite matching algorithms into their worst case behavior. More precisely, we will define for each n and m a graph G_{nm} with $O(n)$ nodes and $O(m)$ edges such that the running time of our algorithms on G_{nm} is $\Omega(nm)$ and $\Omega(\sqrt{n}m)$, respectively. The graph will consist of a complete bipartite graph and a set of chains and therefore we will first study the behavior of our matching algorithms on complete bipartite graphs and chains. The claim about the running time on G_{nm} will then follow easily.

The running time of graph and network algorithms depends on the representation of a graph; reordering adjacency lists and renumbering nodes may have a dramatic effect on the running time. It may even be the case that the average running time on a graph averaged over all representations of the graph is much smaller than the worst case running time on a representation of the graph. We call a family G_{nm} of graphs a *robust* worst-case example if the average running time on the represenations of G_{nm} is $\Omega(nm)$ and $\Omega(\sqrt{n}m)$, respectively. We will first construct a non-robust worst-case example and then make it robust.

A Worst-Case Representation: We begin our study with the complete bipartite graph G_C with $\sqrt{cm}$ nodes on each side and cm edges for some still to be determined constant c. We observe first that it is trivial to find a perfect matching. The greedy heuristic, the Ford-Fulkerson algorithm with breadth-first search both find a perfect matching in time $O(m)$, the Hopcroft-Karp algorithm and the ABMP-algorithm find a perfect matching by considering only augmenting path of length one and hence also determine a perfect matching in linear time $O(m)$. It seems that we are not really heading towards a worst-case example, but wait. Assume that we have determined a perfect matching in G_C. An important observation about G_C is the fact that every edge is contained in a short alternating path. More precisely, let b_0 be an arbitrary node on the B-side. Then every edge is contained in an alternating path of length at most four starting in b_0. This is easy to see. We may assume that the nodes are numbered such that node a_i is matched with node b_i. The edge (b_0, a_0) is an alternating path by itself, an edge (a_0, b_i) with $i > 0$ lies on an alternating path (b_0, a_0, b_i) of length two, an edge (b_i, a_i) with $i > 0$ lies on an alternating path (b_0, a_0, b_i, a_i) of length three, and an edge (a_i, b_j) with $i > 0$ and $j \neq i$ lies on an alternating path $(b_0, a_0, b_i, a_i, b_j)$ of length four.

We will exploit this fact as follows. We make the complete graph G_C part of $G_{n,m}$. Consider an arbitrary node a of $G_{n,m}$, assume that we have the edge (a, b_0) and that the current matching is such that the shortest augmenting path from a has length more than five. Then a search for an augmenting path from a using either breadth-first search or the search method in the ABMP-algorithm will explore all of G_C and hence will have cost $\Omega(m)$. Similarly, if all augmenting paths with respect to the current matching have length more than five then the construction of the layered network in the HK-algorithm will take time $\Omega(m)$.

The following program constructs a complete bipartite graph with m edges and returns a node on its B-side. The nodes on both sides are added to the lists A and B, respectively.

⟨*mcb_generator.c*⟩+≡

```
node mcb_worst_case_gen_GC(graph& G, int m, list<node>& A, list<node>& B)
{ int n = (int) sqrt(m);
  array<node> a(n),b(n); int i, j;
  for (i = 0; i < n; i++)
    { A.append(a[i] = G.new_node()); B.append(b[i] = G.new_node()); }
```

```
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) G.new_edge(a[i],b[j]);
  return b[0];
}
```

The second ingredient to our construction are chains. Let C_k be a chain consisting of $2k$ nodes, k on each side. Let us agree that the edges in the chain are (a_i, b_i) for $0 \leq i < k$ and (a_i, b_{i-1}) for $1 \leq i < k$. If we put the edge (a_i, b_{i-1}) before the edge (a_i, b_i) in the adjacency list of a_i and consider the nodes on the A-side in the order $a_{k-1}, a_{k-2}, \ldots, a_0$ then all our algorithms will first pair the "wrong" pairs in the chain, i.e., will leave the endpoints a_0 and b_{k-1} unmatched and will pair a_i and b_{i-1} for $1 \leq i < k$. It will require an additional search from a_0 to find the right matching. The additional search constructs an augmenting path of length $2k - 1$.

Thus, if $k \geq 4$ and we add the edge (a_0, z), where z is an arbitrary node on the B-side of G_C, the search will explore all of G_C and hence will have cost $\Omega(m)$.

The program below constructs a chain of length $2k$ and connects all nodes on the A-side of the chain to a given node z. It will become clear below why we connect all nodes on the A-side to z and not just a_0. Observe that the node a_0 is appended last to the list A and hence will be considered last in the search for augmenting paths.

⟨*mcb_generator.c*⟩+≡

```
void mcb_worst_case_gen_C(graph& G, int k,
                          list<node>& A, list<node>& B, node z)
{ array<node> a(k), b(k); int i;
  for (i = 1; i < k; i++)
  { A.append(a[i] = G.new_node()); B.append(b[i] = G.new_node()); }
  A.append(a[0] = G.new_node()); B.append(b[0] = G.new_node());
  for (i = 1; i < k; i++)
  { G.new_edge(a[i],b[i-1]);
    G.new_edge(a[i],z);
    G.new_edge(a[i],b[i]);
  }
  G.new_edge(a[0],b[0]);
  G.new_edge(a[0],z);
}
```

We can finally put together our worst case graph $G_{n,m}$. It consists of:

- A complete graph G_C with $\sqrt{m'}$ nodes on each side and m' edges. We will determine m' below such that the total graph has about m edges. Let z be an arbitrary node on the B-side of G_C.
- $n/8$ chains with 4 nodes on each side. This makes a total of $n/2$ nodes on each side and a total of $7n/8 + n/2$ edges. Recall that all nodes on the A-side will be connected to z.
- A single chain with i nodes on each side for $5 \leq i < \sqrt{n}$. This makes a total of about $n/2$ nodes on each side and about $3n/2$ edges.

The total number of edges of our graph is about $m' + 7n/8 + 2n$ and therefore we set $m' = \max(1, m - 23n/8)$; we want m' to be positive in order to guarantee the existance of the node z. The number of nodes will be about $n + \sqrt{m'}$ which is certainly less than $2n$. The following program constructs the worst case graph.

⟨*mcb_generator.c*⟩+≡

```
void mcb_worst_case_gen(graph& G, int n, int m,
                        list<node>& A, list<node>& B)
{ int m1 = m -  23*n/8; // to account for the edges in the chains.
  if (m1 < 0) m1 = 2;
  node z = mcb_worst_case_gen_GC(G,m1,A,B);
  int N = 0;
  int k = 4;
  for (int j = 0; j < n/8; j++) mcb_worst_case_gen_C(G,k,A,B,z);
  while ( N < n/2 )
    { mcb_worst_case_gen_C(G,++k,A,B,z); N += k; }
}
```

What is the running time of our bipartite matching algorithms on the graph $G_{n,m}$? We leave it to the exercises to discuss the running time of our matching algorithms on a single chain and on a collection of chains.

- The Ford-and-Fulkerson algorithm with either depth-first or breadth-first search takes time $\Omega(m)$ for each chain and hence a total time of $\Omega(n/2 \cdot m')$. For breadth-first search this follows from the fact that all our chains have length at least four and for depth-first search this follows from the fact that we put the edge (a_0, z) before the edge (a_0, b_0) for the endpoint a_0 of each chain and hence depth-first search will explore the graph G_C before it explores the chain.

- The Hopcroft-Karp algorithm will construct layered graphs with $2k$ layers for k from 2 to $\sqrt{n}$. The construction of each layered graph will explore all of G_C and hence a running time of $\Omega(\sqrt{n}m)$ results.

- The ABMP algorithm works in two phases. Assume that it switches from phase one to phase two when the maximal layer is equal to L. Then only augmenting path of length at most L were used and hence there are still $\sqrt{n} - L$ chains without a perfect matching. For all such chains all of G_C will be explored in the second phase and hence the running time will be $\Omega((\sqrt{n} - L)m)$. In the first phase we will lift the node a_0 of some chain to level L and hence all of G_C at least to level $L - 5$. Thus the running time of the first phase is $\Omega(Lm)$. We conclude that the running time of the ABMP-algorithm is $\Theta(\sqrt{n}m)$.

Table 0.1 shows the running times of bipartite matching algorithms on the graphs generated by *mcb_worst_case_gen*.

A Robust Worst-Case Graph: In the arguments of the preceding section we exploited the fact that our algorithms are fed a particular represenation of the graph $G_{n,m}$? The repre-

n	m	m'	FF	HK	ABMP	ABMP*
2000	60000	58084	5.36	1.96	1.19	0.13
2000	120000	118084	11.6	3.99	2.55	0.23
2000	240000	238084	24.3	8.11	5.26	0.42
4000	60000	56167	9.06	2.72	1.12	0.17
4000	120000	116167	21.21	5.66	2.45	0.26
4000	240000	236167	46.04	11.48	5.14	0.46
8000	60000	52334	13.22	3.81	1.22	0.26
8000	120000	112334	36.76	8.06	3.08	0.35
8000	240000	232334	86.78	16.71	7.08	0.56

Table 0.1 The running times of bipartite matching algorithms on the graphs generated by *mcb_worst_case_gen*. The first three columns give the values of n, m, and m', where m' is as defined above. The next four columns show the running times of the Ford-Fulkerson with breadth-first search and the greedy heuristic, of the Hopcroft-Karp algorithm with the greedy heuristic, of the ABMP-algorithm, and of the ABMP with the pruning heuristic. The latter algorithm will be discussed in Section 0.3.
The asymptotic running time of the first algorithm is $\Theta(nm)$. This is clearly visible: if either n or m is doubled, the running time doubles. The asymptotic running time of the second and the third algorithm is $\Theta(\sqrt{n}m)$. Again this is clearly visible: if n is quadrupled or m is doubled, the running time doubles. The running time of the last algorithm is $\Theta(\sqrt{n}n + m)$. Again this is somewhat visible: for fixed n and varying m, the running time grows linearly in m (observe that $(T(n, 4m) - T(n, 2m))/(T(n, 2m) - T(n, m)) \approx 2$ for each of the three values of n) and for fixed m like $n^{3/2}$ in n (observe that $(T(4n, m) - T(2n, m))/(T(2n, m) - T(n, m)) \approx 2.8 = ((4n)^{3/2} - (2n)^{3/2})/((2n)^{3/2} - n^{3/2})$, e.g. for $m = 120000$, we have $T(8000, m) - T(4000, m) = 0.09$ and $T(4000, m) - T(2000, m) = 0.03$ and similarly for the other values of m. See Table 0.3 for a related experiment.

sentation guarantees that the algorithms first construct the "false" matching on the chains contained in $G_{n,m}$. The matching leaves the endpoints of the chains exposed and hence a long augmenting path is needed to obtain the optimal matching of the chain. We have chosen different lengths for the chains to guarantee a large running time.

What will change when we run our algorithms on a random representation of $G_{n,m}$, i.e., we use

```
mcb_worst_case_gen(G,n,m,A,B);
permute(G);  A.permute();
```

to generate G. Here *permute*(G) is a function that randomly permutes the representation of G, i.e., the list of nodes, the list of edges, and all adjacency lists, and *A.permute*( ) randomly permutes A.

n	m	m'	FF	HK	ABMP	ABMP*
2000	60000	58084	3.14	1.657	1.584	1.754
2000	120000	118084	8.125	3.717	4.369	3.34
2000	240000	238084	16.85	7.7	8.014	8.497
4000	60000	56167	5.603	2.539	2.184	1.598
4000	120000	116167	12.15	5.518	3.899	2.659
4000	240000	236167	35.08	11.33	8.231	7.25
8000	60000	52334	11.43	3.331	1.729	2.338
8000	120000	112334	26.46	7.329	5.432	6.438
8000	240000	232334	82.72	15.96	12.15	9.118

Table 0.2 The running times of bipartite matching algorithms on a random representation of the graphs generated by *mcb_worst_case_gen*. All times are the averages of ten runs. The first three columns give the values of n, m, and m', where m' is as defined above. The next four columns show the running times of the Ford-Fulkerson with breadth-first search and the greedy heuristic, of the Hopcroft-Karp algorithm with the greedy heuristic, of the ABMP-algorithm, and of the ABMP with the pruning heuristic. The latter algorithm will be discussed in Section 0.3.
The asymptotic running times of the first algorithm is $\Theta(nm)$. This is clearly visible: if either n or m is doubled, the running time doubles. The asymptotic running time of the other three algorithms is $\Theta(\sqrt{n}m)$. Again this is clearly visible: if n is quadrupled or m is doubled, the running time doubles.

$\langle$*random_representation.c*$\rangle{+}\equiv$

```
void permute(graph& G)
{
  list<edge> E = G.all_edges();
  E.permute();
  G.sort_edges(E);
  list<node> V = G.all_nodes();
  V.permute();
  G.sort_nodes(V);
}
```

Table 0.2 shows the running times of our matching algorithms on random representations of our worst case graphs $G_{n,m}$. We observe the same asymptotic running times as in Table 0.1 for the first three algorithms. For *ABMP** the running has increased from $O(n^{3/2}+m)$ to $\Theta(\sqrt{n}m)$.

When we permute the representation of the graph and also the list of nodes in A, we should expect an initial matching of the following structure. One of the nodes in the chain

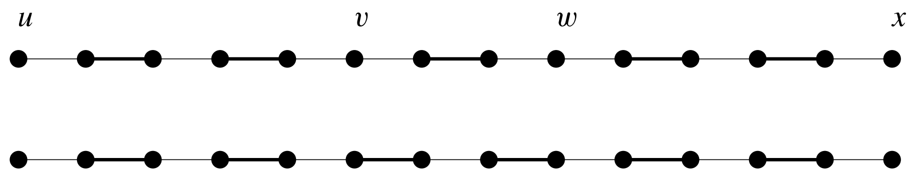

Figure 0.1 In the top chain there are three augmenting paths, namely with endpoints u and v, v and w, and w and x, respectively. Using the path from v to w for augmentation, merges the three paths into one. If the endpoint of an augmenting path is an endpoint of the containing chain, only two paths are merged.

is matched[1] with z, the special node on the B-side of G_C, we will have a near-perfect matching on G_C with one free node on the A-side, and we will have some matching on the chains. The matching decomposes the chains into augmenting paths. Each subsequent augmentation will remove one of those augmenting paths. The effect of this is to merge (in generally three) augmenting paths into one, see Figure 0.1. When the augmenting path has length more than five, we will still have the effect that all of G_C will be explored in the search. Here we use the fact that we connected all nodes on the A-sides in the chains to the special node on the B-side in G_C. We conclude that the asymptotic running times of our implementations do not change.

We discuss the behavior of *ABMP*[*] at the end of Section 0.3.

0.3 The Alt-Blum-Mehlhorn-Paul Algorithm, Revisited

We describe two improvements to our implementation of the algorithm of Alt, Blum, Mehlhorn, and Paul. The first improvement concerns the choice of the constants which govern the switch from phase one to phase two of the algorithm and the second improvement incorporates a heuristic which prunes the graph from all nodes that cannot lie on an augmenting path.

In the book we write:

> 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).

This sentence is a bit too simpleminded. We now give a rational choice for the constants.

The ABMP-algorithm works in two phases. In the first phase it maintains a layered graph and in the second phase it uses breadth-first search to match the remaining free nodes. When is the right time to switch from phase one to phase two? We should try to balance the cost

[1] Observe that we have many more nodes in the chains than in G_C and hence the nodes in the chains are considered first.

of the two phases. We do so by counting inspections of edges. The running time of both phases is bounded by a constant times the number of edge inspections. In the first phase we can count edge inspections inside the current edge function. We charge one inspection for the call to *ce* and one inspection for each execution of the inner loop.

⟨*current edge function*⟩≡

```
inline edge ce(const node v, const graph& G,
               const node_array<int>& layer, node_array<edge>& ce_it,
               const node_array<node_status>& status, double& EdgeInsp)
{
  edge e = ce_it[v]; EdgeInsp++;
  node w;
  if ( e == nil ) e = G.first_adj_edge(v);
  while (e && (layer[w = G.target(e)] != layer[v] - 1 || status[w] == DEAD))
  { e = G.adj_succ(e); EdgeInsp++; }
  ce_it[v] = e;
  return e;
}
```

In the second phase we have at most m edge inspections for each search for an augmenting path. The number of searches for augmenting paths in the second phase is bounded by $1+f$ edges, where f denotes the number of nodes (in A) which are matched in the second phase. We can bound f in different ways:

- by the number of free nodes as the begining of the second phase,
- by n/L since there must be f node-disjoint augmenting path each of length at least L.
- by $\min_{0 \le k \le Lmax} N_k/(k+1)$, where N_k is the number of nodes on layers zero to k, since there must be f node-disjoint augmenting paths and hence the number of nodes on layers at most k must be at least $f(k+1)$.

In our implementation we are only using the first two bounds. We experimented with the third bound and concluded that it is not worth using.

We switch from phase one to phase two as soon as the number of edge inspections in phase one exceeds $\min(n/L, \mathit{free\_in\_A.size}(\,))m/32$. The division by 32 accounts for the fact that the edge inspections in phase two are cheaper than the inspections in case one and that only a part of the graph is usually explored in the search for an augmenting path in phase two. We arrived at the number 32 by experiments.

Let us argue that this definition of the switching point guarantees a running time of $O(\sqrt{n}m)$. Let T_1 be the running time of phases one, let L_0 be the value of L when we switch from phase one to phase two, and let $B_2 = O(\min(n/L_0, \mathit{free\_in\_A.size}(\,))m) = O(nm/L_0)$ be our bound for the maximal running time of phase two. We also know that $T_1 = O(L_0 m)$. The definition of switching time guarantees that $T_1 = B_2 \pm O(m)$. We conclude that $T_1 + B_2 = O(\min(L_0 m, nm/L_0)) = O(\sqrt{n}m)$.

The overall structure of the implementation of the algorithm does not change.

⟨*_ABMP_matching.c*⟩+≡

```
enum node_status { UNKNOWN, ALIVE, DEAD };
⟨pruning heuristic⟩
⟨standard search for augmenting path⟩
⟨current edge function⟩
list<edge> MAX_CARD_BIPARTITE_MATCHING_ABMP_NEW(graph& G,
                            const list<node>& A, const list<node>& B,
                            node_array<bool>& NC,
                            bool use_pruning,
                            bool use_heuristic)
{ node v; edge e;
  //check that all edges are directed from A to B
  forall(v,B) assert(G.outdeg(v) == 0);
  ⟨ABMP: initialization⟩
  // additional initializations
  node_array<node_status> status(G,ALIVE);
  double EdgeInsp1 = 0, EdgeInsp2 = 0;
  int n = G.number_of_nodes(), m = G.number_of_edges();
  node_array<edge> cur_edge(G,nil); // current edge iterator
  b_stack<edge> p(G.number_of_nodes());
  while ( (EdgeInsp1 + EdgeInsp2)/m < leda_min(n/L,free_in_A.size())/32 )
  { node v = free_in_A.pop();
    L = layer[v];
    if ( EdgeInsp2 > 10*m && use_pruning )
    { EdgeInsp1 += EdgeInsp2; EdgeInsp2 = 0;
      PRUNE(G,free_in_A,B,free,status);
    }
    ⟨search for an augmenting path from v⟩
  }
  if ( use_pruning ) PRUNE(G,free_in_A,B,free,status);
  ⟨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,status) )
    number_of_augmentations++;
}
```

Observe that we have given *find_aug_path_by_bfs* an additional parameter *status*. The chunks ⟨*ABMP: initialization*⟩, ⟨*search for an augmenting path from v*⟩, ⟨*MCB: greedy heuristic*⟩, and ⟨*MCB: prepare result and node cover and restore orientations*⟩ are basically as in the book and hence we do not give them here. The only change is that the call of the current edge function *ce*(w, G, *layer*, *cur_edge*, *status*, *EdgeInsp2*) in the second chunk gets the additional parameters *status* and *EdgeInsp2*.

We complete the discussion of the first change with two small remarks. We count the number of edge inspections with two counters *EdgeInsp1* and *EdgeInsp2*. When *use_pruning* is false, the first counter stays zero and only the second counter is used. Also, we label all nodes as *ALIVE* initially. The status of nodes is only changed in the function *PRUNE* and hence, if *use_pruning* is false, all nodes will be alive all the time. Hence *status* may be ignored in this case.

We come to our second change, the pruning heuristic. It removes all nodes from the graph that can no longer lie on an augmenting path by labeling those nodes as DEAD. We will show that the heuristic can be implemented to run in linear time $O(m)$. In fact, the heuristic will inspect every edge of the graph once. We call the heuristic after $10m$ edge inspections and hence the pruning heuristic can increase the total running time by at most a small constant factor (we should inspect an increase of at most 10%). How are we using the pruning? We use the pruning in the current edge function and in the search for augmenting paths by breadth-first search by skipping over all edges that lead to dead nodes. Thus

⟨*standard search for augmenting path*⟩≡

```
static int number_of_augmentations;
static bool find_aug_path_by_bfs(graph& G,
                                  node a,
                   node_array<bool>& free,
                   node_array<edge>& pred,
                   node_array<int>&  mark,
                   const node_array<node_status>& status)
{ 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 || status[w] == DEAD) 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;
}
```

Observe that the only change is the test whether w is alive.

We come to the implementation of the pruning procedure. A node is alive if it lies on some augmenting path. If a node is dead at some point, it will always be dead. As the result of an augmentation more nodes may die. The procedure makes use of an auxiliary label *UNKNOWN*.

The logic underlying the procedure is simple. We first compute the strongly connected components of the graph (by depth-first search) and then run depth-first search on the shrunken graph starting from all free nodes in A. The reachable nodes in B are alive and if a node has an edge leading to a live node then v is alive.

We gain efficiency by combining the two depth-first searches into one. We start by labeling all non-dead nodes as unknown and all free nodes in B as alive. We then propagate liveness along tree- and cross-edges. The crucial observation is that this will give the roots of strongly connected components their correct label (the root of a strongly connected component is the node with the smallest dfs-number in the component; see the Section on strongly connected components). When the root is correctly labelled, we can extend the label to the other nodes in the component. After having labeled the live nodes, we label all nodes that have still the label *UNKNOWN* as dead.

As in the strongly connnected components algorithm we have a recursive procedure (here called *scc_dfs*) and a master procedure (here called *PRUNE*). The master procedure iterates over all free nodes in A and calls the recursive procedure for every node that was not reached by a previous call.

⟨*pruning heuristic*⟩≡

```
static void scc_dfs(const graph& G, node v, node_array<int>& dfsnum,
                                           node_array<node_status>& status,
                                           node_list& unfinished,
                                           list<node>& roots,
                                           int& count1);

void PRUNE(const graph& G, const list<node>& free_in_A,
                           const list<node>& B,
                           const node_array<bool>& free,
                           node_array<node_status>& status)
{
```

```
  list<node>      roots;
  node_list       unfinished;
  node_array<int> dfsnum(G,-1);

  int count1 = 0;

  node v;
  forall_nodes(v,G) if (status[v] != DEAD) status[v] = UNKNOWN;
  forall(v,B) if (free[v]) status[v] = ALIVE;

  forall(v,free_in_A)
    if (dfsnum[v] == -1 && status[v] != DEAD)
       scc_dfs(G,v,dfsnum,status,unfinished,roots,count1);
}
```

The recursive procedure maintains two stacks. The stack of *unfinished* nodes and the stack of *roots*. The two stacks are manipulated exactly as in the strongly connected components algorithm. In addition we propagate alive-labels across edges. When the root of a scc is found, all nodes in the component are given the label of the root. If the label of the root is still un*UNKNOWN*, the nodes in the component are declared dead.

⟨*pruning heuristic*⟩+≡

```
static void scc_dfs(const graph& G, node v, node_array<int>& dfsnum,
                                            node_array<node_status>& status,
                                            node_list& unfinished,
                                            list<node>& roots,
                                            int& count1)
{ node w;

  dfsnum[v] = ++count1;
  unfinished.push(v);
  roots.push(v);

  forall_adj_nodes(w,v)
  { if (status[w] == DEAD) continue;
    if (dfsnum[w] == -1)
       scc_dfs(G,w,dfsnum,status,unfinished,roots,count1);
    else
      if ( unfinished(w) )
        while (dfsnum[roots.head()] > dfsnum[w]) roots.pop();

    if (status[w] == ALIVE) status[v] = ALIVE;
  }

  if (v == roots.head())
  { node_status comp_status = status[v];
    if (comp_status == UNKNOWN) comp_status = DEAD;
    do { w = unfinished.pop(); status[w] = comp_status;
           /* w is an element of the scc with root v */
       } while (v != w);
    roots.pop();
  }
}
```

What is the effect of the pruning heuristic on our worst case graphs?

n	FF	HK	ABMP	ABMP*
10000	0.16	0.23	0.17	0.25
20000	0.31	0.44	0.36	0.49
40000	0.61	0.88	0.74	1.01
10000	0.237	1.712	0.309	0.367
20000	0.594	5.543	0.794	0.953
40000	1.293	15.91	1.836	2.426
10000	0.2	4.21	0.26	0.33
20000	0.43	12.61	0.6	0.74
40000	0.89	36.22	1.35	1.63
10000	0.247	1.548	0.335	0.405
20000	0.532	4.671	0.786	0.929
40000	1.099	13.7	1.761	2.335

Table 0.3 The running times of bipartite matching algorithms on a single chain with n nodes on each side (top half) and a set of chains with k nodes, $1 \le k \le \sqrt{2n}$, nodes on each side (bottom half). Each half consists of two parts. In the first part we report the runnings time on the chain representation given in Section 0.2 and and in the second part we report the running times on a random representation. The times for random representations are the averages of ten runs.

The greedy heuristic (or the consideration of all augmenting paths of length one) determines a perfect matching in G_C. No node in G_C can lie on an augmenting paths afterwards and hence the first call of *PRUNE* will effectively remove G_C from G. Thus the running time becomes $O(m)$ plus whatever it takes to compute a perfect matching on a collection of chains. It is shown in the exercises that the latter time is $O(n^{3/2})$. Thus the total running time of *ABMP** on $G_{n,m}$ is $\Theta(m + n^{3/2})$. Can we verify this by measurements? For $T(n,m) = m + n^{3/2}$ we have

$$(T(n,4m) - T(n,2m))/(T(n,2m) - T(n,m)) = (4m - 2m)/(2m - m) = 2$$

and

$$\begin{aligned}(T(4n,m) - T(2n,m))/(T(2n,m) - T(n,m)) &= ((4n)^{3/2} - (2n)^{3/2})/((2n)^{3/2} - n^{3/2}) \\ &= 2^{3/2} \approx 2.8.\end{aligned}$$

If a random representation of $G_{n,m}$ is used, the running time seems to increase to $\Theta(\sqrt{n}m)$. Why is this so? Why is the graph G_C not pruned away early in the execution? We attempt an explanation. After running the greedy heuristic, the special node z on the B-side of G_C

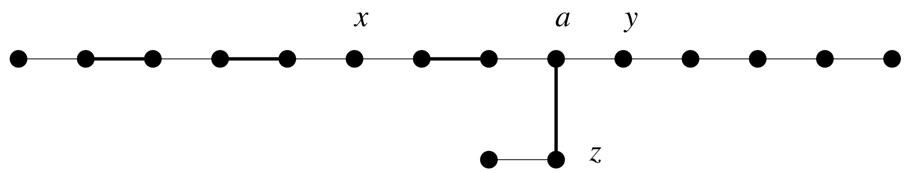

Figure 0.2 The situation produced by the greedy matching.

will most likely be matched to a random node in one of the chains, because the chains contains many more nodes than G_C. Also the mate of z is most likely to belong to a long chain. Observe that we have $n/2$ nodes in chains of length 7 and $2k$ nodes in chains of length $2k-1$ for $5 \leq k \leq \sqrt{n}$ and hence the expected length of the chain containing a random node is

$$(n/2 \cdot 7 + \sum_{5 \leq k \leq \sqrt{n}} 2k \cdot (2k-1))/n = \Theta((\sqrt{n})^3/n) = \Theta(n^{1/2}),$$

since $\sum_{1 \leq k \leq K} k^2 \approx \int_{1 \leq k \leq K} k^2 = \Theta(K^3/3)$.

So far our argument was solid. From now on, it becomes a plausibility argument. Consider Figure 0.2. The node a of some chain is matched with the special node z on the B-side of G_C. We will match z with the free node in G_C once we use the augmenting path containing (a, z) for augmentation. This will be the case once it is a shortest augmenting path. However, it is unlikely that a specific path is the shortest and hence the length of the augmenting path finally doing it will be a fraction of the length of the chain containing a.

Table 0.4 shows the running time of our matching algorithms on random group graphs.

Exercises for 0.3

1 What is the running time of bipartite matching algorithms on a single path C_n with n nodes on each side? Discuss two cases: (1) the graph representation is such that the algorithms first construct a matching which leaves only the endpoints unmatched and (2) the graph representation is random. Predict the asymptotic running time. The top half of Table 0.3 shows measured running times.

2 What is the running time of bipartite matching algorithms on a graph consisting of a collection of paths C_k with $1 \leq k \leq \sqrt{n}$. Discuss the same cases as in the preceding exercise. Predict the asymptotic running times. The bottom half of Table 0.3 shows measured running times.

3 In Table 0.3 the difference between the last two columns is striking. The use of the pruning heuristic increases the running time by about 50%; this contradicts the claim made in the text that the use of the pruning heuristic should increase running time by at most 10%. Try to find an explanation. We have only a partial explanation. The function *permute*(G) which we use to construct a random representation of a graph permutes adjacency lists and also the list of nodes. In the function *PRUNE* we iterate over the list of nodes. This iteration will be slow for the random representation as it incurs a

n	m	k	FF-	FF+	HK-	HK+	AB-	AB+	AB*-	AB*+	Check
4	8	1	2.34	2.01	3.34	3.1	4.33	4.27	3.21	2.83	0.17
4	8	100	1.83	1.27	2.19	2.09	2.72	2.68	2.53	2.27	0.15
4	8	10000	0.55	0.53	2.05	1.89	2.45	2.4	2.27	2.17	0.14
4	12	1	91.5	30.86	4.8	4.6	6.22	6.24	4.46	4.36	0.21
4	12	100	45.32	28.79	3.65	3.5	3.69	3.68	3.65	3.03	0.16
4	12	10000	0.71	0.65	4.1	4.05	2.76	2.75	2.19	2.09	0.16
4	16	1	29.08	11.64	5.15	4.76	5.25	5.19	5.28	5.27	0.26
4	16	100	33.58	26.75	3.95	3.77	3.14	3.13	3.55	3.32	0.19
4	16	10000	0.82	0.78	7.32	7.05	2.2	2.19	2.55	2.52	0.17
8	16	1	6.35	6.86	7.16	6.71	12.23	12.07	8.5	8.59	0.37
8	16	100	5.19	4.13	4.56	4.23	8.17	8	6.78	6	0.28
8	16	10000	1.25	1.15	4.87	4.48	6.78	6.65	5.91	5.61	0.28
8	24	1	373.5	113.6	10.33	9.79	17.93	17.67	12.88	10.11	0.46
8	24	100	215.9	139.2	7.95	7.64	11.14	11.11	7.53	7.46	0.35
8	24	10000	3.79	3.05	9.9	9.84	8.18	8.15	5.88	5.74	0.3201
8	32	1	99.54	31.34	12.14	11.3	19.17	19.12	11.55	11.54	0.55
8	32	100	167.6	134.2	9.48	9.13	11.93	11.71	8.23	7.71	0.3701
8	32	10000	15	13.52	16.74	16.24	7.69	7.44	8.16	8.06	0.3501

Table 0.4 The running times of the bipartite matching algorithms *FF*, *HK*, *ABMP*, 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.

large number of cache faults. Change the function *permute* such that it only permutes adjacency lists. Repeat the experiment.