1

Algorithms for Dense Graphs and Networks

J. Cheriyan
School of Operations Research
Cornell University
Ithaca, New York
USA

K. Mehlhorn
Max–Planck–Institut für Informatik and
Universität des Saarlandes
66 Saarbrücken
Germany

September 2, 1991

Abstract

We improve upon the running time of several graph and network algorithms when applied to dense graphs. In particular, we show how to compute on a machine with word size λ a maximal matching in an n–vertex bipartite graph in time $O(n^2 + n^{2.5}/\lambda) = O(n^{2.5}/\log n)$, how to compute the transitive closure of a digraph with n vertices and m edges in time $O(nm/\lambda)$, how to solve the uncapacitated transportation problem with integer costs in the range $[0..C]$ and integer demands in the range $[-U..U]$ in time $O((n^3(\log\log n/\log n)^{1/2} + n^2 \log U)\log nC)$, and how to solve the assignment problem with integer costs in the range $[0..C]$ in time $O(n^{2.5}\log nC/(\log n/\log\log n)^{1/4})$.
Assuming a suitably compressed input, we also show how to do depth–first and breadth–first search and how to compute strongly connected components and biconnected components in time $O(n\lambda + n^2/\lambda)$, and how to solve the single source shortest path problem with integer costs in the range $[0..C]$ in time $O(n^2(\log C)/\log n)$.

Key words: graph, network, algorithm, dense graph, dense network

1 Introduction

We improve upon the running time of several graph and network algorithms when applied to dense graphs and networks by exploiting the parallelism on the word level available in Random Access Computers [?]. In particular, the bounds shown in Table 1 can be obtained for graphs and networks with n vertices and m edges on machines with word size $\lambda = \Omega(\log n)$. For several graph algorithms, we show that the previously best bounds can be improved by a factor λ on dense graphs; e.g., a maximum matching in a bipartite graph can be computed in time $O(n^2 + n^{2.5}/\lambda) = O(n^{2.5}/\log n)$. For problems on networks, e.g., the shortest path problem, the assignment problem, and the transportation problem, assuming that all the numeric parameters of the network are integers, we obtain improvements by a fractional power of $\log n$.

*When this research was carried out, both authors worked at the Dept. of Computer Science, Univ. d. Saarlandes, D-66 Saarbrücken. The research was partially supported by ESPRIT project no. 3075 ALCOM.

Problem	Running Time	Input	Previous Bound
Depth–first–search Breadth–first–search Strongly connected components Biconnected components	$O(n\lambda + n^2/\lambda)$	c	$O(n^2)$
Maximum matching in bipartite graphs	$O(n^2 + n^{2.5}/\lambda)$	s	$O(n^{2.5})$, [?]
Transitive closure	$O(nm/\lambda)$	s	$O(nm)$
Single source shortest paths with edge weights in $[0..C]$	$O(n^2 \frac{\log C}{\log n})$	c	$O(n^2)$, [?]
Assignment problem with edge weights in $[0..C]$	$O(n^{2.5}(\log nC) \cdot \beta)$	s	$O(n^{2.5} \log nC)$, [?], [?]
transportation problem with edge costs in [0..C] and demands in [0..U]	$O((n^3\gamma + n^2 \log U) \log nC)$	s	$O((n^3 + n^2 \log U) \log nC)$ [?]

Table 1: Survey of results: The first column specifies the problem, the second column states the running time obtained in this paper(β denotes $(\log\log n/\log n)^{1/4}$ and γ denotes $(\log\log n/\log n)^{1/2}$), the third column states whether the input is standard (s) or compressed (c), and the fourth column states the best previous bound. λ denotes the word size of the machine.

There is a simple common principle underlying all our improvements. This principle was introduced by Cheriyan, Hagerup, and Mehlhorn [?] in their $O(n^3/\log n)$ maximum flow algorithm. Alt et al [?] showed later that the technique can also be applied to the bipartite matching problem. They obtained a running time of $O(n^{2.5}/\sqrt{\log n})$. In this paper, we further exploit the principle and show that it can be applied to a large number of graph and network problems.

The technique is most easily described in the case of depth–first–search (DFS). DFS is a recursive procedure which explores a graph starting from a source s. Initially, all vertices are unlabeled and DFS(s) is called. A call DFS(v) labels v and then scans the edges (v, w) starting at v until an unlabeled vertex w is found. Then DFS(w) is called. The crucial observation is that, although up to n^2 edges are examined by DFS, only $n - 1$ of them lead to recursive calls. Suppose now that the adjacency matrix of the graph is available in "compressed form", i.e., t, $t \leq \lambda$, bits of the adjacency matrix are stored in a single computer word. Suppose also that we maintain the compressed bit–vector of unlabeled vertices. Then taking the component–wise AND of corresponding words of v's row of the adjacency matrix and the bit–vector of unlabeled vertices and testing the result for zero checks *simultaneously* for t vertices whether one of them is unlabeled and reachable from v by an edge. In this way, the adjacency lists of all vertices can be scanned in $O(n^2/t)$ time. Only n times will an edge leading to an unlabeled vertex be detected and a recursive call be required. This adds $O(nt)$ to the running time.

The details for DFS will be given in section 2.2. Breadth–first–search is discussed in section 2.3 , the computation of strongly connected and biconnected components in section 2.4, the matching problem in section 2.5, and the computation of transitive closures in section 2.6. We mention that Feder and Motwani [?] independently obtained an $O(n^{2.5}/\log n)$ bipartite matching algorithm. Their approach is completely different from ours. The algorithms for the computation of strongly and biconnected components given in section 2.4 are alternatives to the algorithms in [?]. We find that the correctness proofs in section 2.4 are more intuitive.

Chapter 3 is devoted to algorithms on networks; the shortest path problem is discussed in section 3.1, the transportation problem in 3.2, and the assignment problem in 3.3. For network algorithms, the compression technique requires the precomputation of tables and therefore typically the full word size cannot be exploited.

The machine model used in this paper is essentially the RAC (*Random Access Computer*) of Angluin and Valiant [?]. Let λ be an integer. A λ–RAC consists of $M = 2^\lambda$ registers, each of which can hold an integer in the range $[0..M - 1]$. The instruction set of a λ–RAC consists of arithmetic operations (addition, subtraction, multiplication and integer division (all modulo M)) and boolean operations (AND, OR, EXCLUSIVE–OR, Negation). For the boolean operations an integer is interpreted as a bitstring of length λ; all boolean operations work bit–wise, i.e., on all λ bits in parallel. In contrast to Angluin and Valiant [?] , we do not postulate that the word–size λ is logarithmic in the size of the input. Rather, we treat word size and length of input as independent quantities and only require that the word size is at least logarithmic in the size of the input. Following Kirkpatrick and Reisch [?] , we call an algorithm *conservative*, if it uses only a word size which is logarithmic in the size of the input (although the actual word size of the machine in use may be actually larger).

2 Graph Algorithms

2.1 Basics

For an integer t and a $0-1$ valued vector $L[0..n-1]$ the *t-compression* (or t–compressed version) $\diamond L$ of L is a vector $\diamond L[0..\lceil n/t\rceil-1]$ such that for $0 \le k < \lceil n/t \rceil$:

$$\diamond L[k] = \sum_{0 \le l < t} L[k \cdot t + l] 2^{t-l},$$

where $L[v] = 0$ for $v \ge n$ is assumed for simplicity. The entries of a t–compression take values in $T = [0..2^t - 1]$.

For integers $x \in T$ and $l, 0 \le l < t$, we use $(x)_l$ to denote the l-th bit of x, i.e., $x = \sum_{0 \le l < t} (x)_l \cdot 2^l$ and $(x)_l \in \{0,1\}$ for $0 \le l < t$. For $0 \le l < t$ let E_l denote the integer 2^l.

For $x \in T, x \ne 0$, $\lfloor \log x \rfloor$ is the index of the highest numbered non–zero bit in x. In our graph algorithms, we will frequently have to compute $\lfloor \log x \rfloor$. Let us assume that $\lfloor \log x \rfloor$ is not available as a machine instruction. The simplest algorithm is linear search.

$$l \leftarrow t-1; \textbf{while } (x \ AND \ E_l) = 0 \textbf{ do } l \leftarrow l-1 \textbf{ od}$$

It takes time $O(t)$ and needs no precomputation. A faster method is binary search. It takes $O(\log t)$ time and requires the precomputation of $O(t)$ masks. Finally, Fredman and Willard [**?**] have recently found a method which works in time $O(1)$ with $O(\lambda)$ precomputation. In the algorithms below we always state the time bounds in terms of linear search.

For a graph $G = (V, E)$ with n nodes we identify the vertices with the integers $0, 1, \ldots, n-1$ and we use E to also denote the adjacency matrix of G, i.e., $E[v,w] = 1$ iff $(v,w) \in E$, and $E[v,w] = 0$, otherwise. The t–compressed adjacency matrix $\diamond E$ is a matrix $\diamond E[0..n-1, 0..\lceil n/t\rceil - 1]$ such that the v-th row of $\diamond E$ is the t–compression of the v-th row of $E, 0 \le v < n$.

2.2 Depth First Search

Depth–First–Search (DFS) is a useful method for the systematic exploration of a graph. DFS visits the nodes of a graph in depth–first order, i.e., DFS always follows an unexplored edge (if any) out of the most recently reached vertex. Program 1 specifies DFS as a recursive procedure $dfs(node\ v)$. This program also computes two node labelings $dfsnum$ and $compnum$ and a list of tree edges. The labeling $dfsnum$ numbers the nodes by the time of the call of dfs, $compnum$ numbers the nodes by the time of the completion of the call of dfs, and $tree$ contains the set of edges whose exploration leads to recursive calls. DFS runs in time and space $O(n^2)$ on an n–vertex graph.

We now describe the compression technique. Let $\diamond E$ be the t–compressed adjacency matrix. We also store the bitvector *reached* in its t–compressed form $\diamond$*reached* and represent a node v by the pair (i,j) with $i = \lfloor v/t \rfloor$ and $j = v \bmod t$. The crucial observation is now that for $0 \le h \le \lceil n/t \rceil - 1$, $\diamond E[v,k] \wedge \neg \diamond reached[k] \ne 0$ iff some edge in $\{(v,w);\ kt \le w < (k+1)t\}$ leads to an unreached node, i.e., one operation checks t edges. The details are given in program 2.

```
(1)    proc dfs(node v)
(2)    begin reached[v] ← 1;
(3)          dfsnum[v] ← dfs_count1 ← dfs_count1 + 1;
(4)          for w ∈ V
(5)          do if E[v, w] and ¬reached[w]
(6)             then T.append((v, w));
(7)                  dfs(w)
(8)             fi
(9)          od;
(10)         compnum[v] ← dfs_count2 ← dfs_count2 + 1
(11)   end;
(12)   T ← empty list of edges
(13)   for v ∈ V do reached[v] ← 0 od;
(14)   dfs_count1 ← dfs_count2 ← −1;
(15)   for v ∈ V
(16)   do if ¬reached[v] then dfs(v) fi od
```

Program 1: Depth First Search

```
(1a)   proc dfs(integers i, j)
(1b)   begin v ← it + j;
(2a)         (reached[i])_j ← 1;
(3a)         dfsnum[v] ← dfs_count1 ← dfs_count1 + 1;
(4a)         for k ∈ [0..⌈n/t⌉ − 1]
(5a)         do X ← ◇E[v, k] ∧ ¬◇reached[k];
(5b)            while X ≠ 0
(5c)            do l ← t − 1; while (X)_l = 0 do l ← l − 1 od;
(6a)                T.append((v, kt + l));
(7a)                dfs(k, l);
(7b)                X ← ◇E[v, k] ∧ ¬◇reached[k]
(8a)            od
(9a)         od;
(10a)        compnum[v] ← dfs_count2 ← dfs_count2 + 1
(11a)  end;
(12a)  T ← empty list of edges
(13a)  for i ∈ [0..⌈n/t⌉ − 1] do ◇reached[i] ← 0 od;
(14a)  dfs_count1 ← dfs_count2 ← −1;
(15a)  for i ∈ [0..⌈n/t⌉ − 1]
(15b)  do for l ∈ [0..t − 1]
(16a)     do if ¬(reached[i])_l then dfs(i, l) fi  od
(17a)  od
```

Program 2: Depth First Search with compressed adjacency matrix

Lemma 1 *Given the t–compressed adjacency matrix $\diamond E$ of an n–vertex graph, $t \leq \lambda$, depth–first–search runs in time $O(n^2/t + nt)$ and space $O(n^2/t)$ on a λ–RAC.*

Proof: The time spent outside line (5c) is clearly $O(n^2/t)$. Also, a single execution of line (5c) takes time $O(t)$ and there are at most $n-1$ executions of it since there are at most $n-1$ calls to dfs in line (7a). This proves the time bound. The space bound is obvious. ∎

Remark: Line (5c) computes $\lfloor \log X \rfloor$ by linear search in time $O(t)$. In view of section 2.1, we may also use binary search or the constant time method of Fredman and Willard provided that we add $O(t)$ and $O(\lambda)$ preprocessing time respectively. This gives a running time of $O(n^2/t + n \log t + t)$ and $O(n^2/t + \lambda)$ respectively. We prefer to state our time bounds in terms of linear search because it uses the weakest machine model.

DFS can be used to partition the edges of a graph into tree–, forward–, back–, and cross–edges. The tree edges have already been collected in the list L in program 1. We now show how to construct the submatrices of $\diamond E$ corresponding to the three other classes of edges. All three classes can be characterized in terms of the two labelings $dfsnum$ and $compnum$, cf. [**?**] and [**?**, section IV.5], e.g., an edge (v,w) is a cross–edge if $dfsnum[v] > dfsnum[w]$ and $compnum[v] > compnum[w]$. We can therefore extract the submatrix of cross–edges by deleting all edges which violate one of the two defining conditions. The following simple strategy deletes for example all edges (v,w) with $dfsnum[v] \leq dfsnum[w]$. We step through the vertices in increasing order of dfsnumber and maintain the t–compression $\diamond smaller$ of a bit–vector $smaller$ with $smaller[w] = 1$ iff $dfsnum[w] < dfsnum[v]$. The AND of $\diamond E[v,*]$ and $\diamond smaller$ then deletes all edges (v,w) with $dfsnum[v] \leq dfsnum[w]$. The program follows:

for $v \in V$ **do** $ord[dfsnum[v]] \leftarrow v$ **od**;
for $i \in [0..\lceil n/t \rceil - 1]$ **do** $smaller[i] \leftarrow 0$ **od**;
for h **from** 0 **to** $n-1$
do $v \leftarrow ord[h]$; $i \leftarrow v$ div t; $j \leftarrow v$ mod t;
  for $k \in [0..\lceil n/t \rceil - 1]$
  do $\diamond C[v,k] \leftarrow \diamond E[v,k] \wedge \diamond smaller[k]$ **od**;
  $(\diamond smaller[i])_j \leftarrow 1$
od

Lemma 2 *Under the hypothesis of Lemma 1, the t–compressed adjacency matrices for the forward–, back–, and cross–edges can be computed in time $O(n^2/t + nt)$ on a λ–RAC.*

2.3 Breadth First Search

Breadth–First–Search computes the shortest distances of all nodes from a given set S of nodes, i.e., a node labeling d with $d(v) = \min\ \{k; \exists v_0, v_1, ..., v_k$ such that $v_0 \in S, v_k = v$, and $(v_i, v_{i+1}) \in E$ for $0 \leq i < k\}$, cf. Program 3. BFS takes time $O(n^2)$ on an n-vertex graph. With the methods of section 2.2 (the details are left to the reader) this can be improved to $O(n^2/t + nt)$ time on a λ–RAC, provided that $t \leq \lambda$ and the t–compressed adjacency matrix is given.

```
for all v ∈ V do Reached[w] ← 0 od;
Q ← empty queue;
for all s ∈ S do Q.append(s); Reached[s] ← 1; od; Q.append(#); d ← 0;
while Q ≠ ∅
do v ← Q.pop();
      if v = # and Q ≠ ∅
      then Q.append(#); d ← d + 1
      else d[v] ← d;
           for all w ∈ V
           do if E[v, w] and ¬Reached[w]
              then Reached[w] ← 1;
                   Q.append(w)
              fi
           od
      fi
od
```

Program 3: Breadth–First–Search; $Q.append(x)$ appends x to the rear of Q, $Q.pop()$ deletes the first element of Q and returns it.

The *layered subgraph* of a graph G consists of all edges (v, w) with $d[w] = d[v]+1$. It is needed in several applications of BFS, e.g. to matching or flow problems; cf. section 2.5. We now discuss how to construct the compressed adjacency matrix of the layered subgraph. For $k, 0 \le k < n$, let $\diamond L_k$ be the t–compression of the bit–vector L_k where $L_k[v] = 1$ iff $d[v] = k$ for all $v \in V$. These vectors can be computed in time $O(n^2/t + n)$; namely $O(n^2/t)$ time to initialize them to zero and $O(1)$ time for each vertex. Next observe that the v-th row of the t–compressed adjacency matrix $\diamond D$ of the layered subgraph is given by the AND of $\diamond E[v, *]$ and $L_{d[v]+1}$. We summarize in:

Lemma 3 *Given the t–compressed adjacency matrix of an n–vertex graph, $t \le \lambda$, and a subset S of the vertices, BFS and the construction of the layered subgraph of G take time $O(n^2/t + nt)$ on a λ–RAC.*

2.4 Strongly–Connected and Biconnected Components

A digraph $G = (V, E)$ is strongly connected if for any two vertices $v, w \in V$ there is a path from v to w. A strongly connected component (scc) is a maximal strongly connected subgraph. An undirected graph $G = (V, E)$ is biconnected if for any two edges e and e' there is a simple cycle containing e and e'. A biconnected component (bcc) is a maximal biconnected subgraph.

Tarjan [**?**] has given linear time algorithms for the computation of scc's and bcc's. Both algorithms are based on the computation of so–called lowpoints. Since lowpoint values can change $\Omega(m)$ times, his algorithms do not seem amenable to the techniques described in the previous sections. Another linear time algorithm for the computation of scc's was given by Sharir [**?**]. It uses DFS on G and G^{rev}, the graph obtained from G by reversal of all edges. Section 2.2 implies that Sharir's algorithm runs in time $O(n^2/t + nt)$ provided

Example: Assume that G_{cur} is as shown in Figure 1. We assume that DFS reaches the vertices in the order a, b, c, d, e, f, g, h that the calls $dfs(a), dfs(e), dfs(d), dfs(g)$ are completed, and that we are currently exploring edges out of vertex h. We have $unfinished = (b, c, d, f, g, h)$ and $roots = (b, c, f, h)$.

Figure 1: A graph G_{cur}; tree (back, cross, forward) edges are shown solid (dashed, wiggled, dash–dotted). Nodes for which the call of dfs is completed are shaded. The shrunken graph is also shown. Completed components are shaded.

that the t–compressed adjacency matrix of G and G^{rev} are given. Unfortunately, Sharir's algorithm cannot be used to compute bcc's of undirected graphs.

In this section we describe an $O(m)$ algorithm for the computation of scc's which is as fast as Tarjan's algorithm (Sharir's is slower by about a factor of two), is simple (maybe even simpler than Tarjan's algorithm), can be modified to compute bcc's, and can be made to run in time $O(n^2/t + nt)$ given the t–compressed adjacency matrix. Our algorithm is similar to an algorithm described by Dijkstra [?] which has however running time $\Omega(n^2)$.

Our algorithm is based on DFS and constructs the scc's of G incrementally. Let G_{cur} be the subgraph of G consisting of all vertices reached by DFS and all edges explored by DFS. A scc of G_{cur} is called *completed* if the call $dfs(v)$ is completed for all vertices v of the component, and *uncompleted*, otherwise. Let $unfinished = (v_1, v_2, ..., v_s)$ be the sequence of vertices of G_{cur} in uncompleted components of G_{cur} ordered according to increasing DFS–number. For each scc C call the node with the smallest DFS–number the *root* of C, and let $roots = (v_{i_1}, v_{i_2}, ..., v_{i_k})$ with $1 = i_1 < i_2 < ... < i_k$ be the subsequence of $unfinished$ consisting of the roots of the uncompleted components. We maintain the following three invariants.

- I1: There are no edges (x, y) of G_{cur} with x belonging to a completed component and y belonging to an uncompleted component.
- I2: The nodes in $roots$ lie on a single tree path, i.e., $v_{i_l} \xrightarrow[T]{}^* v_{i_l+1}$ for $1 \leq l < k$, and we are currently exploring edges out of v_p where $p \geq i_k$.
- I3: The nodes in the uncompleted scc with root v_{i_l} are the nodes $v_{i_l}, v_{i_l+1}, ..., v_{i_{l+1}-1}$ (with the convention $i_{k+1} = s + 1$). Moreover, all these nodes are tree descendants of the root v_{i_l}.

Let us now consider the exploration of edges and the completion of calls. If (v, w) is the edge to be explored, let $G'_{cur} = (V_{cur} \cup \{w\}, E_{cur} \cup \{(v, w)\})$ be the new graph spanned by the explored edges. Of course, $w \in V_{cur}$ if (v, w) is not a tree edge.

- Exploration of a tree edge (v, w):

In G'_{cur} the node w is a scc by itself and, of course, an uncompleted one; all other scc's stay the same. We can reflect this change by adding the node w at the end of sequences $unfinished$ and $roots$. Note that this preserves all our invariants. I3 is preserved since w is a scc by itself. I1 is preserved since the node v belongs to an uncompleted component according to I3; I2 is preserved since v is a tree descendant of the last element of sequence *roots* according to I2, I3, and the fact that (v, w) is a tree edge. Also, the sequences *unfinished* and *roots* are still ordered by DFS–number.

In Program 4, lines (3) and (4) implement the actions described above. The sequence *roots* and *unfinished* are realized as pushdown stores; in addition, *unfinished* is also represented as a boolean array *in_unfinished*.

```
(1)   procedure dfs(v : node);
(2)   count1 ← count1 + 1; dfsnum[v] ← count1; reached[v] ← true;
(3)   push v onto unfinished; in_unfinished[v] ← true;
(4)   push v onto roots;
(5)   for all w with (v, w) ∈ E
(6)   do if ¬reached[w]
(7)      then dfs(w)
(8)      else if in_unfinished[w]
(9)           then co we now merge components oc
(10)               while dfsnum[top(roots)] > dfsnum[w]
(11)               do pop(roots) od
(12)          fi
(13)     fi
(14)  od;
(15)  if v = top(roots)
(16)  then repeat w ← pop(unfinished); in_unfinished[w] ← false;
(17)               co w is an element of the scc with root v oc
(18)     until v = w;
(19)     pop(roots)
(20)  fi
(21)  end;

(22)  begin     co main program oc
(23)  unfinished ← roots ← empty_stack;
(24)  count1 ← 0;
(25)  for all v ∈ V do in_unfinished[v] ← false; reached[v] ← false; od;
(26)  for all v ∈ V do if ¬reached[v] then dfs(v) fi od
(27)  end.
```

Program 4: A scc algorithm

- Exploration of a non–tree edge (v, w):

  We have to distinguish two cases:

either w belongs to a completed component or it does not. The case distinction is made in line (8) of Program 4.

Case 1: w belongs to a completed component.

In this case no path exists from w to v, since v belongs to an uncompleted component of G_{cur} according to I2 and no edge exists from a node in a completed component to a node in an uncompleted component according to I1. Thus G'_{cur} and G_{cur} have the same scc's and no action is required. The three invariants are clearly preserved.

Case 2: w belongs to an uncompleted component.

Let $unfinished = (v_1, v_2, \ldots, v_s)$ and let $roots = (v_{i_1}, v_{i_2}, \ldots, v_{i_k})$, where $1 = i_1 < i_2 < \cdots < i_k$. Let $v = v_p$, where $p \geq i_k$ according to I2, and $w = v_q$ where $i_l \leq q < i_{l+1}$, i.e. v_{i_l} is the root of the scc containing w. Then the scc's of G'_{cur} can be obtained by merging the scc's of G_{cur} with roots $v_{i_l}, v_{i_{l+1}}, \ldots, v_{i_k}$ into a single scc with root v_{i_l} and leaving all other scc's unchanged. This can be seen as follows. Note first that completed scc's remain the same according to I1. Next consider any node z in an uncompleted component, i.e., $z = v_r$ for some r. If $r \geq i_l$, say $i_h \leq r < i_{h+1}$ with $l \leq h \leq k$, then

$$v_{i_l} \underset{E_{cur}}{\longrightarrow^*} v_{i_h} \underset{E_{cur}}{\longrightarrow^*} v_r \underset{E_{cur}}{\longrightarrow^*} v_h \underset{E_{cur}}{\longrightarrow^*} v_{i_k} \underset{E_{cur}}{\longrightarrow^*} v \longrightarrow w \underset{E_{cur}}{\longrightarrow^*} v_{i_l}$$

where the existence of the first, the fourth and the fifth path follows from I2 and I3, the existence of the second and third path follows from the fact that v_{i_h} and v_r belong to the same scc, and the existence of the seventh path follows from the fact that w and v_{i_l} belong to the same scc. Thus v_r and v_{i_l} belong to the same scc of G'_{cur} if $r \geq i_l$.

If $r < i_l$, say $i_h \leq r < i_{h+1}$ with $h < l$, then $v_r \underset{E_{cur}}{\longrightarrow^*} v_{i_h} \underset{E_{cur}}{\longrightarrow^*} v_{i_l} \underset{E_{cur}}{\longrightarrow^*} w$, since v_r and v_{i_h} (v_{i_l} and w respectively) belong to the same scc and $v_{i_h} \underset{E_{cur}}{\longrightarrow^*} v_{i_l}$ according to I2. Since $h < l$ no path exists from v_{i_l} to v_r in G_{cur}. If there were such a path in G'_{cur}, then it would have to use the edge (v, w) and hence there would have to be a path from w to v_r in G_{cur}. Thus w and v_r would belong to the same scc of G_{cur}, a contradiction. This shows that uncompleted scc's with roots v_{i_h}, $h < l$, remain unchanged.

We have now shown that the scc's of G'_{cur} can be obtained from the scc's of G_{cur} by merging the scc's with roots $v_{i_l}, \ldots, v_{i_k}$ into a single scc. The newly formed scc has root v_{i_l} and hence the merge can be achieved by simply deleting the roots $v_{i_{l+1}}, \ldots, v_{i_k}$ from *roots*. Next note that $i_l \leq q < i_{l+1} < \cdots < i_k$, where $w = v_q$ and hence $dfsnum[v_{i_l}] \leq dfsnum[w] < dfsnum[v_{i_{l+1}}] < \ldots < dfsnum[v_k]$ since *unfinished* and *roots* are ordered according to DFS–number. This shows that the merge can be achieved by popping all roots from *roots* which have a DFS–number larger than w. That is exactly what lines (10) and (11) of Program 4 do. The three invariants are preserved by the arguments above. This finishes the description of how edges are explored. We now turn to the completion of calls.

- Completion of a call $dfs(v)$:

According to I2 the node v is a tree descendant of the last vertex of *roots*, $i_k = top(roots)$. If it is a proper tree descendant, i.e., $v \neq top(roots)$, then the completion of $dfs(v)$ does not complete a scc. We return to $dfs(w)$ where w is the parent of v. Clearly, w is still a tree descendant of $top(roots)$ and also $w \underset{E_{cur}}{\longrightarrow} v \underset{E_{cur}}{\longrightarrow^*} top(roots)$ belong to the same scc This shows that I2 and I3 are preserved; I1 is also preserved since we do not complete a component.

If $v = top(roots)$ then we complete a component. According to I3 this component consists of exactly those nodes in *unfinished* which do not precede $top(roots)$ and hence these nodes are easily enumerated as shown in lines (16) through (18) of Program 4. Of course, $top(roots)$ ceases to be a root of an uncompleted scc and hence has to be deleted from *roots*; line (19). We still need to prove that the invariants are preserved. For I1 this follows from the fact that all edges leaving the just completed scc must terminate in previously completed scc's, since the uncompleted scc's form a path according to I2. The invariants I2 and I3 are also maintained by a similar argument as in the case $v \neq \mathit{top}(\mathit{roots})$.

We have now proved the correctness of Program 4 and summarize in:

Theorem 1 *Program 4 computes the strongly connected components of a digraph in time $O(n+m)$.*

Proof: Having already proved correctness, we still have to prove the time bound. The time bound follows directly from the linear time bound for DFS and the fact that every node is pushed onto and hence popped from *unfinished* and *roots* exactly once. This implies that the time spent in lines (11) and (16) is $O(n)$. The time spent in all other lines is $O(n+m)$. ∎

We next discuss a more efficient implementation of this algorithm for dense graphs. It is based on the observation that at most $2(n-1)$ edges lead to a recursive call or to a merge of existing components. Our goal is therefore to identify these edges quickly. For each root r of an uncompleted component let B_r be a bit–vector such that $B_r[v] = 1$ iff v belongs to the uncompleted component with root v. The exploration of an edge (v, w) leads to a recursive call $dfs(w)$ if $\mathit{reached}[w] = 0$ and it causes some components to be merged if $\mathit{in\_unfinished}[w] = 1$ and $B_{\mathit{top}(\mathit{roots})}[w] = 0$, i.e., if w lies in an uncompleted component which is not the component of $\mathit{top}(\mathit{roots})$. If all bit–vectors are stored in t–compressed form then this condition can be tested in time $O(1)$ for a block of t edges and hence the time spent on scanning adjacency lists is $O(n^2/t + nt)$. When a new vertex w is reached and $dfs(w)$ is called we create a new t–compressed vector $\diamond B_w$ and initialize it such that $\diamond B_w[x] = 1$ iff $w = x$. This takes time $O(n/t)$ for each call and hence $O(n^2/t)$ in total. When two components are merged, we also need to update the B–vectors, i.e. line (11) is changed into

$B \leftarrow B_{\mathit{top}(\mathit{roots})}$;
pop *roots*;
$B_{\mathit{top}(\mathit{roots})} \leftarrow B_{\mathit{top}(\mathit{roots})} \vee B$

This takes time $O(n/t)$ for each merge step and hence $O(n^2/t)$ in total. We summarize in:

Theorem 2 *Given the t–compressed adjacency matrix of an n–vertex graph, $t \leq \lambda$, the strongly connected components of G can be computed in time $O(n^2/t + nt)$ on a λ–RAC.*

We next turn to the computation of biconnected components of undirected graphs which we assume to be given by their (symmetric) adjacency matrix. For a bcc C let us call the vertex with the second smallest DFS–number the *center* of C, and for each vertex w let $\mathit{parent}[w]$ be the parent of w in the DFS–tree. A bcc C is called *completed* if the call $dfs(v)$ where v is the center of C is completed. As before, let *unfinished* denote the sequence of vertices belonging to uncompleted bcc's of G_{cur} in increasing order of DFS–number. Note that a vertex can belong to several bcc's; it stays in unfinished until all of them are completed. Finally, *centers* is the subsequence of centers in *unfinished*. The invariants are now:

- I1: For all edges (x, y) of G_{cur}, x and y belong to the same bcc of G_{cur}.

Let $unfinished = (v_1, v_2, ..., v_k)$ and $centers = (v_{i_1}, v_{i_2}, ..., v_{i_s})$, where $i_1 < i_2 < ... < i_k$.

- I2: The vertices in *centers* lie on a single tree path and we are currently exploring edges out of v_p where $p \geq i_k$.

- I3: The vertices in the uncompleted bcc with center v_{i_l} are the vertices $v_{i_l}, v_{i_l+1}, ..., v_{i_{l+1}}$ (with the convention $i_k = s$) together with the vertex $father[v_{i_l}]$. All but the vertex $father[v_{i_l}]$ are tree descendants of v_{i_l}.

In the program, one changes line (4) into

(4a) push v onto *centers*,

lines (10) and (11) into

(10a) **while** $dfsnum[father[top(centers)]] > dfsnum[w]$
(11a) **do** $pop(centers)$ **od**

and lines (15) to (20) into

(15a) **if** $v = top(centers)$
(16a) **then repeat** $w \leftarrow pop(unfinished); in\_unfinished[w] \leftarrow false$
(17a) **until** $w = v$;
(18a) $pop(centers)$;
(19a) (∗ $father[v]$ and the vertices just popped from the bcc with center c ∗)
(20a) **fi**.

Theorem 3 *The program above computes the biconnected components of an undirected graph in time $O(n + m)$. Given the t–compressed adjacency matrix, $t \leq \lambda$, it can be made to run in time $O(n^2/t + nt)$ on a λ–RAC.*

Proof: Analogous to the proofs of theorems 1 and 2. ∎

2.5 Maximum Bipartite Matching

The maximum bipartite matching problem (MPM–problem) is to find a maximum cardinality matching in a bipartite graph. An undirected graph $G = (V, E)$ is *bipartite* if there is a partition of the vertex set V into disjoint sets A and B such that every edge $e \in E$ has exactly one endpoint in each of the two sets. A *matching* M is a subset of E such that every vertex is incident to at most one edge in M.

Hopcroft and Karp [?] have shown how to solve the maximum bipartite matching problem in time $O(n^{1/2} \cdot m)$. We give an implementation of their algorithm which runs in time $O(n^{1/2}(n^2/\lambda + n\lambda))$ on a λ–RAC. Thus the MPM–problem can be solved in time $O(n^{2.5}/\log n)$ by a conservative algorithm. For dense graphs, this improves upon [?] and [?].

The algorithm of Hopcroft and Karp works in $O(\sqrt{n})$ Phases. In each phase, which takes $O(m)$ time, a maximal set (with respect to set inclusion) of shortest augmenting paths is determined by breadth–first and subsequent depth–first search.

An *augmenting path* with respect to a matching M is an alternating path connecting two *free* vertices in V, i.e. vertices which are not incident to an edge in M. An *alternating path* is a path in G which alternately uses edges in M and $E - M$. Interchanging the matching and non–matching edges of an augmenting path increases the cardinality of the matching by one.

We can now describe a phase of their algorithm in more detail. Let M be the matching at the beginning of the phase. Let $G_M = (V, E_M)$ be a directed graph with edge set $E_M = \{(v, w); \{v, w\} \in E \backslash M, v \in A, w \in B\} \cup \{(w, v); \{v, w\} \in M, v \in A, w \in B\}$, i.e., the edges in M are directed from B to A and the edges outside M are directed from A to B. Clearly, the paths from free vertices in A to free vertices in B are in one-to-one correspondence to the augmenting paths with respect to M. In each phase a BFS of G_M starting from the free vertices in A is carried out first. Let d be the minimal distance label of a free vertex in B, let G_L consist of the layers 0 through d of the layered subgraph of G_M, and let D be the adjacency matrix of G_L. Clearly, all shortest augmenting paths with respect to M can be found in G_L. A maximal set of vertex–disjoint augmenting paths can be determined by a variant of DFS, cf. Program 5. It maintains a set L of vertex–disjoint augmenting path (initially empty) and a set of reached vertices. A call search_path(v), where $v \in A$ is free, constructs an augmenting path from v to a free node in B (if any) and adds it to L. Also, all nodes visited by the search are added to the set of reached vertices. Having determined a maximal set L of vertex–disjoint augmenting paths, the matching M is updated by reversing the direction of all edges of all paths in L.

We now discuss how to implement a phase in time $O(n^2/\lambda + n\lambda)$ on a λ–RAC. Let us assume inductively that the λ–compressed adjacency matrix of G_M is available at the beginning of a phase. (For $M = \emptyset$, it takes time $O(n^2)$ to establish this assumption). We first construct the λ–compressed adjacency matrix $\diamond D$ of G_L using BFS as described in section 3.2 in time $O(n^2/\lambda + n\lambda)$, and then search for augmenting paths as described above. Since *search_path* is called at most once for each vertex and since the total length of the augmenting paths found in one phase is at most n, the time spent for the search is $O(n)$ except for the three lines marked by ($\diamond$).

```
L ← empty set of paths;
for all v ∈ V do reached[v] ← 0 od;
for all v ∈ A, v free
do (* L is a set of vertex–disjoint augmenting paths; reached[v] = 1
implies that either v lies on a path in L or there is no path
from v to a free vertex in B disjoint from the paths in L *)
   P ← search_path(v);
   if P ≠ nil then L.append(P) fi
od
where
   path procedure search_path(node v);
   (* when search_path(v) is called, the recursive stack contains a path
   from a free node in A to v which is disjoint from the paths in L. The
   call either finds a path from v to a free node in B and then returns
   this path or, otherwise, returns nil *)
   w ← 0; P ← nil;
   while P = nil and W < n    (◇)
   do if ¬reached[w] and D[v, w]    (◇)
      then if w is free
           then P ← ((v, w))
           else P ← search_path(w);
                if P ≠ nil then P.append((v, w)) fi
           fi
           reached [w] ← 1;
      fi
      w ← w + 1 (◇)
   od
   return P
   end
```

Program 5: Searching for augmenting paths

Replacing them by:

```
k ← 0; P ← nil;
while P ≠ nil and k < ⌈n/t⌉
do X ← ◇D[v, k] ∧ ¬Reached[k]
   if X ≠ 0
   then l ← 0; while (X)_l = 0 do l ← l + 1 od;
        w ← k · λ + l;
        ⋮
   else k ← k + 1
   fi
od
```

brings the cost of these lines down to $O(n^2/\lambda + n\lambda)$. Finally, reversing the direction of all edges of all paths in L takes time $O(n)$.

We summarize in:

Theorem 4

a) *On a λ–RAC a maximum matching in an n–vertex bipartite graph can be computed in time $O(n^{1/2} \cdot (n^2/\lambda + n\lambda))$.*

b) *The MPM–problem can be solved in time $O(n^{2.5}/\log n)$ by a conservative algorithm.*

2.6 Transitive Closure of Acyclic Graphs

In this section, we discuss the computation of transitive closures. We restrict ourselves to acyclic (directed) graphs because acyclicity makes the problem more difficult; for general graphs one can always compute the strongly connected components first and then shrink them to obtain an acyclic graph. We assume our graphs to be topologically sorted, i.e., $V = \{0, ...n-1\}$ and $(v, w) \in E$ implies $v < w$. The transitive closure E^* of E consists of all pairs (v, w) such that there is a path from v to w using only edges in E. The transitive reduction E_{red} of E consists of all edges $(v, w) \in E$ such that there is no path of length at least two from v to w. Let $m_{red} = |E_{red}|$.

Goralcikova and Koubek [?] have shown how to compute E^* in time $O(m_{red} \cdot n)$. The algorithm is quite simple. It steps through the vertices of G in *decreasing* order. When vertex v is considered it first initializes $E^*[v, v] = 1$ and $E^*[v, w] = 0$ for $w \neq v$, and then considers the edges $(v, w) \in E$ in *increasing* order of w. When $(v, w) \in E$ is considered, and $E^*[v, w] = 0$ at that time, then $E^*[v, *] \leftarrow E^*[v, *] \vee E^*[w, *]$ is performed.

The crucial observation is that the OR of the v-th and the w-th row of E^* is computed precisely for the edges $(v, w) \in E_{red}$; this implies the $O(m_{red} \cdot n)$ time bound. Also, if the λ–compression of E^* is computed instead, then the time bound reduces to $O(n^2 + m_{red} \cdot n/\lambda)$ where the n^2 term accounts for the computation of E^* from $\diamond E^*$.

Lemma 4 *On a λ–RAC the transitive closure of an n–vertex graph can be computed in time $O(n^2 + m_{red} n/\lambda)$.*

Let $\varepsilon, 0 < \varepsilon < 1$, be a fixed real number. A random acyclic digraph is defined as follows. For $v < w$, $prob((v, w) \in E) = \varepsilon$, and the different events $(v, w) \in E$ are independent.

Lemma 5 (Simon [?]) $E(m_{red}) \leq 2n \log n$ *for all* $\varepsilon, 0 < \varepsilon < 1$.

Theorem 5 *The transitive closure of an acyclic digraph can be computed by a deterministic and conservative algorithm whose expected running time on the class of random acyclic digraphs is $O(n^2)$.*

Proof: This is a direct consequence of the two preceding lemmas. ∎

3 Network Algorithms

3.1 An $O(n^2 \frac{\log C}{\log n})$ Shortest Path Algorithm

Let $N = (V, E, c, s)$ be an edge–weighted network with source s, i.e, (V, E) is a directed graph, $c : E \to \{0, ..., C\}$ is an integer–valued cost function on the edges and $s \in V$ is a distinguished vertex. The goal is to compute arrays *dist* and *pred*, where for all $v \in V$ $dist[v]$ is the length of a shortest path from s to v and $pred[v]$ is the predecessor of v on a shortest path.

We solve the shortest path problem in two phases; in the first phase we compute the *dist*–array and in the second phase we compute the *pred*–array. Both phases are based on Dijkstra's algorithm [?]. We use two data structures, namely an integer d and an array $Q : V \to \{-1, 0, ..., C, \infty\}$ that serves as a priority queue. During the execution, a vertex v is in one of two states: *scanned* ($Q[v] = -1$) or *unscanned* ($Q[v] \in \{0, 1, ..., C\} \cup \{\infty\}$). We maintain the invariant that the distance to every scanned vertex from s has been correctly computed, and that for every unscanned vertex v, $Q[v] + d$ is the length of a shortest path from s to v, subject to the restriction that every vertex in the path (except v) is scanned. In each iteration, an unscanned vertex v with minimal value $Q[v]$ is selected. Then $dist[v] = d + Q[v]$ according to the Invariant. For later use in Phase 2 the value $distmod[v] = dist[v] \bmod (C + 1)$ is also stored. Also, d is increased by $Q[v]$, $Q[w]$ is decreased by $Q[v]$ for all unscanned vertices w, and all edges emanating from v are inspected and cheaper paths are recorded. The details are given in Program 6. The proof of correctness is standard, see for example [?, section IV.7.2]. It is also easy to see that the algorithm can be made to run in time $O(n^2 \lceil \log C / \log n \rceil)$ by representing all Q– and *dist*–values in base $2^{\lfloor \log n \rfloor}$. Note that all values computed are bounded by $Cn = 2^{\log n + \log C}$ and hence require $O(\lceil \log C / \log n \rceil)$ digits in base $2^{\lfloor \log n \rfloor}$. In the remainder of the section we will show how to achieve running time $O(n^2 \log C / \log n)$. We may assume $\lceil \log(3 + C) \rceil \le \log n / 3$ because $\lceil \log C / \log n \rceil = O(\log C / \log n)$ otherwise and the claim is trivial. Let $b = \lceil \log(3 + C) \rceil$ and let $t \in \mathbb{N}$ be such that $t \cdot b \le \log n$. The exact value of t will be specified later. Also interpret c as a $V \times V$ matrix with entries in $\{0, ..., C\}$. We partition Q and each row of c into blocks of length t and represent each block as a single integer. More precisely, for $a \in D := \{-1, 0, ..., C, \infty\}$ let

$$\widehat{a} = \begin{cases} C + 2 & \text{if } a = -1 \\ C + 1 & \text{if } a = \infty \\ a & \text{if } 0 \le a \le C, \end{cases}$$

and for $a_0, ..., a_{t-1} \in D$ let $Comp(a_0, ..., a_{t-1}) = \sum_{0 \le i < t} \widehat{a}_i (C + 3)^i$. Then $0 \le Comp(a_0, ..., a_{t-1}) < (C + 3)^t \le 2^{bt} \le n$. The t–compressed representation of a sequence $a_0, ..., a_{n-1}$ of values in D, where for simplicity we assume t to be a divisor of n, is the sequence $Comp(a_0, ..., a_{t-1}), Comp(a_t, ..., a_{2t-1}), ...$. We assume from now on that c and Q are available in t–compressed form and show next how the row–scans implicit in lines (4), (6) and (7), and (9) to (11) can be done in time $O(n/t)$ each.

For integers $A, B, \widehat{a}$ with $0 \le A, B \le (C+3)^t - 1, 0 \le \widehat{a} \le C+1, A = Comp(a_0, ..., a_{t-1})$, and $B = Comp(b_0, ..., b_{t-1})$ let $select\_min(B) = (i, \widehat{b})$ where $0 \le i \le t - 1$ and $\widehat{b}_i = \min(\widehat{b}_0, \widehat{b}_1, ..., \widehat{b}_{t-1})$, $decrease(A, \widehat{a}) = A'$, where $A' = Comp(a_0 - a, ..., a_{t-1} - a)$ if $0 \le \widehat{a} \le C$, and $A' = A$ otherwise, and $componentwise\_min(A, B) = A'$ where $A' = Comp(a'_0, ..., a'_{t-1})$ and

$$a'_i = \begin{cases} b_i, & \text{if } b_i < a_i \le C + 1 \\ a_i & \text{otherwise.} \end{cases}$$

(1) $d \leftarrow Q[s] \leftarrow 0$;
(2) **for** all $v \neq s$ **do** $Q[v] \leftarrow \infty$ **od**;
(3) **while** $\exists$ *unscanned vertex*
(4) **do** let v be an unscanned vertex with minimal
    entry $Q[v]$;
(5) $d \leftarrow dist[v] \leftarrow d + Q[v]; distmod[v] \leftarrow d \bmod (C+1)$
(6)   **for** all unscanned w
(7)   **do** $Q[w] \leftarrow Q[w] - Q[v]$ **od** $(* \; \infty - Q[v] = \infty \; *)$
(8) $Q[v] \leftarrow -1$; $(*$ v is now scanned $*)$
(9)   **for** all unscanned w with $(v, w) \in E$
(10)   **do if** $c(v, w) < Q[w]$
(11)   **then** $Q[w] \leftarrow c(v, w)$
(12)   **fi**
(13)   **od**
(14) **od**

Program 6: shortest paths: Phase 1

Lemma 6

(a) *The function tables for functions select_min, decrease and componentwise_min can be computed in time $O(t \cdot 2^{2bt})$.*

(b) *Given tables for functions select_min, decrease and componentwise_min Phase 1 runs in time $O(n^2/t)$.*

Proof:

(a) The tables have at most $2^{bt}, 2^b \cdot 2^{bt}$, and 2^{2bt} entries respectively. Each entry can be computed in time $O(t)$.

(b) Each execution of lines (4), (6) to (7), and (9) to (11) is tantamount to n/t evaluations of functions *select_min*, *decrease*, and *componentwise_min* respectively and each evaluation takes constant time by table look–up. Finally, an execution of line (5) takes time $O(\log n / \log C) = O(\log n)$ since $0 \leq d \leq nC \leq 2^{2 \log n}$, and an execution of line (8) takes constant time using an appropriate table. Also, initialization takes linear time $O(n)$.

∎

Lemma 7 *Let $t = \lfloor (\log n)/3b \rfloor$. Given the distance matrix c in t–compressed form the shortest distances from s in an n–vertex graph can be computed in time $O(n^2(\log \max(2, C))/\log n)$.*

Proof: For $t = \lfloor (\log n)/3b \rfloor$ we have $t \cdot 2bt = O(n \log n)$ and $O(n^2/t) = O(n^2(\log \max(2, C))/\log n)$. The claim now follows from the preceding lemma. ∎

We next turn to the computation of the shortest path tree. In the standard implementation of Dijkstra's algorithm the *pred*–array is computed together with the *dist*–array by adding the assignment $pred[w] \leftarrow v$ in line (11). We cannot do that here because each such assignment requires to write $\log n$ bits and therefore several of these assignments cannot be compressed into a single assignment. We propose to compute the predecessor information in a second phase. The program for the second phase is the same as for the first phase except that line (7) is replaced by

(7a) $dmod \leftarrow (dmod + Q[v]) \bmod (C+1)$

and line (11) is replaced by

(11a) $Q[w] \leftarrow c(v,w)$
(11b) **if** $(dmod + Q[w]) \bmod (C+1) = distmod[w]$
(11c) **then** $pred[w] \leftarrow v$ **fi**

Lemma 8 *Phase 2 computes the pred–array correctly. Also, given a t–compressed c for $t = \lfloor \log n/3b \rfloor$ it can be made to run in time $O(n^2(\log \max(2,C))/\log n)$.*

Proof: We first prove correctness. Phase 1 computes $distmod[w] = dist[w] \bmod (C+1)$ for all vertices w. Also, $dist[w] \le d + Q[w]$ and $d \le dist[w]$ for all unscanned nodes and $dmod = d \bmod (C+1)$ throughout execution of Phase 2. Thus, whenever line (11a) is executed we have $d \le dist[w] \le d + Q[w] \le d + C$ and hence whenever line (11c) is executed we have $dist[w] = d + Q[w]$ and thus all assignments in line (11c) are valid. Next consider for any vertex $w \ne s$ the last assignment to $Q[w]$ in line (11a). At this point, we have $d + Q[w] = dist[w]$ and hence there will be an assignment to $pred[w]$. Finally observe that for any unscanned vertex w the value $d + Q[w]$ never increases, and decreases in every execution of line (11a). Thus there will be at most one assignment to $pred[w]$. This proves correctness and also that line (11c) is executed at most n times.

We next turn to the running time. For $0 \le A, B, D \le (C+3)^t - 1$, $A = Comp(a_o, ..., a_{t-1})$, $B = Comp(b_0, ..., b_{t-1})$, $D = Comp(d_0, ..., d_{t-1})$, and $0 \le d \le C$ let $Ext\_min(A,B,d,D) = (A', k)$ where $A' = Comp(a'_0, .., a'_{t-1})$ is defined as in the case of $componentwise\_min(A,B)$, $k = \sum_{0 \le i < t} k_i 2^i$, and

$$k_i = \begin{cases} 1 & \text{if } a'_i = b_i < a_i, \text{ and } d + b_i = d_i \bmod (C+1) \\ 0 & \text{otherwise} \end{cases}$$

We can use function *ext_min* in lines (11a) to (11c) as follows:
For t vertices $v_0, ..., v_{t-1}$ let $A = Comp(Q[v_0], ..., Q[v_{t-1}])$, $B = Comp(c[v, v_0], ..., c[v, v_{t-1}])$, and $D = Comp(distmod[v_0], ..., distmod[v_{t-1}])$. Then $ext\_min(A, B, dmod, D) = (A', k)$ where A' is the new content of Q and k codes all indices for which line (11c) has to be executed. Thus lines (11a) to (11c) can be executed for t vertices in time $O(1 + \#$ of executions of line (11c)), which amounts to $O(n^2/t + n)$ overall. Finally, a table for *ext_min* can be computed in time $O(t \cdot 2^b \cdot 2^{3bt}) = O(n(\log n)^2)$. ∎

Theorem 6 *Let $b = \lceil \log(3 + C) \rceil \le (\log n)/3$ and $t = \lfloor \log n/3b \rfloor$. Given the distance matrix c in t–compressed form the shortest path problem on an n–vertex graph with edge weights in $\{0, ..., C\} \cup \{\infty\}$ can be solved in $O(n^2(\log \max(2,C)/\log n)$ time.*

Proof: Obvious from Lemmas 7 and 8. ∎

3.2 The Uncapacitated Transportation Problem

Let (V, W, E) be a symmetric directed bipartite graph, i.e., $E \subseteq (V \times W) \cup (W \times V)$ and $(v, w) \in E$ iff $(w, v) \in E$, let $b : V \cup W \to [-U \dots U]$ be a supply–demand function on the vertices ($\sum_{x \in V \cup W} b(x) = 0$), and let $c : E \to [-C \dots C]$ be a cost function on the edges ($c(v, w) = -c(w, v)$ and $c(v, w) \geq 0$ for $(v, w) \in E \cap (V \times W)$). Let $cap : E \to \mathbb{R} \cup \{\infty\}$ with $cap(v, w) = \infty$ and $cap(w, v) = 0$ for $(v, w) \in E \cap (V \times W)$ be a capacity function on the edges. A solution for the transportation problem (V, W, E, b, c) is an integer valued function f on the edges such that

(1) $f(x, y) = -f(y, x)$ and $f(x, y) \leq cap(x, y)$ for all $(x, y) \in E$

(2) $b(x) = \sum_{(x,y) \in E} f(x, y)$ for all $x \in V \cup W$

(3) $cost(f) = \sum_{f(x,y) \geq 0} f(x, y) \cdot c(x, y)$ is minimized.

Ahuja et al. [?] have shown how to solve the uncapacitated transportation problem in time $O((nm + n^2 \log U) \log nC)$; see also [?].We take the latter paper as the basis of our exposition. We need the following definitions. A function f satisfying (1) is called a *pseudo–flow*. The *imbalance* of a vertex x with respect to a pseudo–flow f is given by $imb(x) = b(x) - \sum_{(x,y) \in E} f(x, y)$, and the *residual capacity* of an edge (x, y) is given by $rescap(x, y) = cap(x, y) - f(x, y)$. A dual function π is any function $\pi : V \cup W \to R$. A pseudo–flow f is *ε–optimal* with respect to real number $\varepsilon \geq 0$ and dual function π if $\bar{c}(x, y) := c(x, y) + \pi(x) - \pi(y) \geq -\varepsilon$ for all $(x, y) \in E$ with $rescap(x, y) > 0$; $\bar{c}(x, y)$ is called the *reduced cost* of edge (x, y). A pseudo–flow is a flow, if it satisfies (1) and (2).

Fact 1

(a) *Let $\varepsilon \leq 1/n$ and let f be a flow which is ε–optimal with respect to some dual function π. Then f is optimal.*

(b) *Let f be any flow and let $\pi(x) = 0$ for all $x \in V \cup W$. Then f is C–optimal with respect to π.*

Ahuja et al. solve the uncapacitated transportation problem by $\log nC$ iterations of a procedure *improve_approximation*. This procedure takes as input a dual function π' and an $\varepsilon > 0$ and returns a flow f and a dual function π such that f is $(\varepsilon/2)$–optimal wrt. π. The procedure requires the precondition that there is an ε–optimal flow f' with respect to π', although it need not to know this flow. For $\varepsilon = C$, the constant zero dual function has this property by part (b) of Fact 1. After $\log nC$ applications of improve_approximation an $(1/n)$–optimal flow is obtained. It is optimal according to part (a) of Fact 1. The procedure improve_approximation (cf. Program 7) starts with the pseudo–flow $f(x, y) = 0$ for all $(x, y) \in E$ and a dual function π such that f is $(\varepsilon/2)$–optimal wrt. π. It then turns f into a flow by successive augmentations along so–called *admissible paths*. An admissible path consists of admissible edges and leads from a vertex x with positive imbalance to

```
procedure improve_approximation(π′, ε);
    f(x,y) ← 0                    for all (x,y) ∈ E
    π(v)   ← π′(v)                for all v ∈ V
    π(w)   ← π′(w) − ε            for all w ∈ W
    Δ      ← 2^⌊log U⌋
    while ∃x ∈ V ∪ W with imb(x) ≠ 0
    do S(Δ) ← {x ∈ V ∪ W; imb(x) ≥ Δ}
       while S(Δ) ≠ ∅

       do   (* f is a (ε/2)–optimal pseudo–flow with respect to π and rescap(x,y)
            is a multiple of Δ for all (x,y) ∈ E *)
            select and delete a vertex x ∈ S(Δ);
            determine an admissible path P from x to some node y with imb(y) < 0;
            augment Δ units of flow along the path P and update f
       od;
       Δ ← Δ/2
    od
```

Program 7: going from ε–optimality to $(\varepsilon/2)$–optimality

a vertex y with negative imbalance. An edge is admissible, if its residual capacity is positive and its reduced cost is negative. An admissible path starting in a vertex x (with $imb(x) > 0$) is constructed iteratively. Suppose that we already have an admissible path from x to some other node y. If $imb(y) < 0$ then we are finished. If $imb(y) \geq 0$ and there is an admissible edge (y,z), then z is added to P (advance). If there is no such edge, then $\pi(y)$ is decreased by $\varepsilon/2$ and y (if different from x) is removed from P (retreat).

Fact 2 *Improve_approximation executes $O(n^2 \log U)$ advance and retreat steps and runs in time $O(n^2 \log U)$ plus the time needed to identify admissible edges. Moreover, $\pi(x)$, $x \in V \cup W$, is changed only $O(n)$ times within a call of Improve_Approximation.*

The $O(nm)$ term in the running time of the Ahuja et al. algorithm results from the fact that each adjacency list is scanned $O(n)$ times, once for each change of the dual value. We now discuss how to speed up the search for admissible edges in dense graphs.

Define the *truncated reduced cost* $\tilde{c}(x,y)$ of an edge $(x,y) \in E$ by:
$\tilde{c}(x,y) = \lfloor \bar{c}(x,y)/(\varepsilon/2) \rfloor \cdot (\varepsilon/2)$.

Lemma 9 *$(x,y) \in E$ is admissible iff $\tilde{c}(x,y) = -1$ and $rescap(x,y) > 0$.*

Proof:

"$\Leftarrow$ ": If $\tilde{c}(x,y) = -1$ then $\bar{c}(x,y) < 0$.

"$\Rightarrow$ ": If (x,y) is admissible then $\bar{c}(x,y) < 0$ and $rescap(x,y) > 0$. By $(\varepsilon/2)$–optimality, we also have $\bar{c}(x,y) \geq -\varepsilon/2$. Thus $\tilde{c}(x,y) = -1$.

∎

We maintain a matrix D which approximates $\widetilde{c}$. We have $D[x,y] \in [-z \ldots z] \cup \{-\infty, \infty\}$, where z is a constant to be fixed later, and $D[x,y] = -1$ iff $(x,y) \in E$ and $\widetilde{c}(x,y) = -1$. Let $b \geq \lceil \log(2z+3) \rceil$, i.e., b bits suffice to encode an entry of D, and let $t \in \mathbb{N}$ be such that $t^2 b \leq \log n$. We partition D into $t \times t$–submatrices and store each submatrix in a single RAM–word.

The invariant $D[x,y] = -1$ iff $\widetilde{c}(x,y) = -1$ is maintained by updating the entries in row and column x after every $z/2$ changes of x's dual value, i.e., $D[x,y]$ is set to $\widetilde{c}(x,y)$ if $(x,y) \in E$ and $\widetilde{c}(x,y) \in [-z, z]$, and to $+\infty$ or $-\infty$ otherwise. This takes time $O(n)$ for each update and hence time $O(n^3/z)$ totally.

For the search for admissible edges we also maintain a (compressed) matrix R with $R[x,y] \in \{0,1\}$ and $R[x,y] = 1$ iff $rescap(x,y) > 0$. $R[x,y]$ can be updated in time $O(1)$ per change of the preflow f and hence in time $O(n^2 \log U)$ totally. Also, given appropriate tables, a scan of an adjacency list takes time $O(n/t)$ plus the number of admissible edges found. The total time for scanning adjacency lists is therefore $O(n^3/t + n^2 \log U)$. Finally, the tables required can be precomputed in time $O(2^{2t^2 b})$.

With the choice $t = z = \lfloor (\log n / \log\log n)^{1/2} \rfloor$, we obtain a running time of

$$O(n^3 (\log\log n / \log n)^{1/2} + n^2 \log U)$$

for each call of improve_approximation. We summarize in :

Theorem 7 *The uncapacitated transportation problem (V, W, E, b, c) with $|V| = |W| = n$, supplies and demands bounded by U and costs bounded by C can be solved in time*

$$O((n^3 (\log\log n / \log n)^{1/2} + n^2 \log U) \log nC).$$

3.3 The Assignment Problem

Let (U, W, E) be a bipartite graph (which is assumed to have a perfect matching) and let $c : E \rightarrow [0..C]$ be a cost function on the edges. A solution to the assignment problem is a perfect matching M which maximizes $c(M) = \sum_{uw \in M} c(uw)$; throughout this section we use uw to denote the unordered edge $\{u, w\}$. We show how to implement the $O(n^{2.5} \log nC)$ algorithm of Orlin and Ahuja [?] so that it runs in time $O(n^{2.5} \log nC \cdot (\log\log n / \log n)^{1/4})$. The same speed–up can also be obtained for the Gabow and Tarjan [?] assignment algorithm.

As most assignment algorithms do, the Ahuja and Orlin algorithm computes not only a matching but also a dual function defined on the vertices. A pair $(val, price)$ of functions $val : U \rightarrow Z$ and $price : W \rightarrow Z$ is called *dominating* with respect to cost function c if $val(u) + price(w) \geq c(uw)$ for all $uw \in E$. It is called *1–tight* with respect to a cost function c and a partial matching M if $val(u) + price(w) = c(uw) + 1$ for all $uw \in M$.

A high–level description of the assignment algorithm is given in Program 8. The algorithm scales the cost function. In each scaling iteration it uses the auction algorithm of Bertsekas [?] in order to compute a perfect matching $\overline{M}$ and a dual pair $(\overline{val}, \overline{price})$ which is dominating and 1–tight with respect to $\overline{M}$ and the reduced cost function $\bar{c}$. The auction algorithm will be discussed below.

proc assignment (U, W, E, c) returns $(M, val, price)$
precondition : (U, W, E) has a perfect matching and $c_{uw} \geq 0$ for all $uw \in E$.
postcondition: M is a perfect matching and $(val, price)$ is a pair of dual functions which is dominating and 1–tight with respect to c and M.

begin if $c_{uw} = 0$ for all $uw \in E$
then return $(M, 0_U, 1_W)$ where M is any perfect matching, 0_u is the constant zero function on U, and 1_W is the constant one function on W

else let $\widetilde{c}_{uw} = \lfloor c_{uw}/2 \rfloor$ for all $uw \in E$;
$(\widetilde{M}, \widetilde{val}, \widetilde{price}) \leftarrow$ assignment $(U, W, E, \widetilde{c})$;
let $\bar{c}_{uw} = c_{uw} - 2\widetilde{val}_u - 2\widetilde{price}_w - 1$;

(* $(\widetilde{val}, \widetilde{price})$ is dominating and 1–tight for $\widetilde{c}$ and $\widetilde{M}$, $\bar{c}_{uw} \leq 0$ for all $uw \in E$, and $\bar{c}(M^*) \geq -4n$ where M^* is an optimal assignment wrt. $\bar{c}$. *)

$(\overline{M}, \overline{val}, \overline{price}) \leftarrow$ auction $(U, W, E, \bar{c})$;

(* $(\overline{val}, \overline{price})$ is dominating and 1–tight for $\bar{c}$ and $\overline{M}$. *)

return $(\overline{M}, 2\widetilde{val} + \overline{val}, 2\widetilde{price} + 1 + \overline{price})$

fi

end

Program 8: The assignment algorithm

Lemma 10 (Orlin and Ahuja) *Let $0 \leq c_{uw} \leq C$ for all $uw \in E$. A call assignment $(U, W, E, (n+1) \cdot c)$ returns an optimal matching and runs in time $O((T_{auction} + n) \log nC + n^{2.5}/\log n)$ where $T_{auction}$ is the cost of a call of auction.*

Proof: Let $\widehat{c}_{uw} = (n+1) \cdot c_{uw}$, let M^* be an optimal matching, and let M be the matching computed by the call of *assignment.* Then

$$\begin{aligned} \widehat{c}(M) \leq \widehat{c}(M^*) \leq & \sum_{uw \in M^*} (val(u) + price(w)) \\ = & \sum_{u \in U} val(u) + \sum_{w \in W} price(w) \\ = & \sum_{uw \in M} (\widehat{c}_{uw} + 1) = \widehat{c}(M) + n \end{aligned}$$

Also, $\widehat{c}(M^*) - \widehat{c}(M)$ is divisible by $n+1$ and hence $\widehat{c}(M^*) = \widehat{c}(M)$.
The depth of the recursion is $\log nC$ and each level has cost $O(T_{auction} + n)$. Finally, the then–case takes time $O(n^{2.5}/\log n)$ (according to section 2.5) . ∎

Lemma 11 (Orlin and Ahuja)

(a) $\bar{c}_{uw} \leq 0$ *for all* $uw \in E$

(b) $\bar{c}(\widetilde{M}) \geq -3n$

(c) *The pair* $(2\widetilde{val} + \overline{val}, 2\widetilde{price} + 1 + \overline{price})$ *is dominating and* 1*–tight for* $\overline{M}$ *and* c.

Proof: cf. [?] ∎

The auction algorithm is defined by Program 9; $\gamma \geq 1$ is a constant to be fixed later. In this program $U_{free} = \{u \in U; \neg\exists uw \in \overline{M}\}$ is the set of free vertices in U wrt. matching $\overline{M}$. W_{free} is defined analogously.

Lemma 12

(a) $(\overline{val}, \overline{price})$ *is always dominating and* 1*–tight with respect to* $\overline{M}$ *and* $\bar{c}$.

(b) *There are at most* $2\gamma n^{3/2}$ *executions of the while–loop in Phase* 1.

(c) $|U_{free}| = |W_{free}| \leq 8n^{1/2}/\gamma$ *at the end of Phase* 1.

(d) *The total cost of Phase* 2 *is* $O(n^{2.5}/\gamma)$.

Proof:

(a) By induction on the running time; cf. [?] for details.

(b) Clearly $1 - \lfloor\gamma\sqrt{n}\rfloor \leq val(u) \leq 0$ for every $u \in U$ and $price(w) \geq 0$ for all $w \in W$ throughout Phase 1. Also, after every increase of $price(w)$, $w \in W$, there is a vertex $u \in U$ such that $c_{uw} + 1 = val(u) + price(w)$. Thus $price(w) \leq c_{uw} + 1 - val(u) \leq 1 + \lfloor\gamma n^{1/2}\rfloor - 1 \leq \gamma n^{1/2}$. Since each iteration of the while–loop either increments $price(w)$ for some $w \in W$ or decrements $val(u)$ for some $u \in U$ the bound follows.

(c) Let $\overline{M}$ be the matching at the end of Phase 1 and let M^* be an optimal assignment wrt. $\bar{c}$. Then

$$\begin{aligned}
-3n &\leq \bar{c}(M^*) \\
&\leq \sum_{u\in U} \overline{val}(u) + \sum_{w\in W} \overline{price}(w) \\
&= \sum_{uw\in\overline{M}} (\overline{val}(u) + \overline{price}(w)) + \sum_{u\in U_{free}} (\overline{val}(u)) + \sum_{w\in W_{free}} \overline{price}(w) \\
&= \sum_{uw\in\overline{M}} (c_{uw} + 1) + |U_{free}|(1 - \lfloor\gamma\sqrt{n}\rfloor) \\
&\leq M - (\gamma\sqrt{n} - 1)|U_{free}|
\end{aligned}$$

where the first inequality follows from Lemma 11, the second by domination, the third by rearranging terms, the fourth from 1–tightness and the fact that $val(u) = 1 - \lfloor\gamma\sqrt{n}\rfloor$ for $u \in U_{free}$ and $price(w) = 0$ for $w \in W_{free}$, and the last from the fact that $c_{uw} \leq 0$ for all $uw \in E$. Thus $|W_{free}| = |U_{free}| \leq 4n/(\gamma\sqrt{n} - 1) \leq 8n^{1/2}/\gamma$ for $n \geq 4$.

proc auction $(U, W, E, \bar{c})$ returns $(\overline{M}, \overline{val}, \overline{price})$
precondition: $\bar{c} \leq 0$ for all $uw \in E$. There is a matching M^* such that $\bar{c}(M^*) \geq -3n$.
postcondition: $(\overline{val}, \overline{price})$ is dominating and 1–tight for $\bar{c}$ and $\overline{M}$.

begin
$\overline{val}_u \leftarrow 0$ for all $u \in U$, $\overline{price}_w \leftarrow 0$ for all $w \in W$; $\overline{M} \leftarrow \emptyset$.

(∗ Phase 1: bidding ∗)
while $\exists u \in U_{free} : val(u) \geq -\lfloor \gamma\sqrt{n} \rfloor + 1$
do let u be such a vertex;
if $\exists w \in W : \bar{c}_{uw} = \overline{val}_u + \overline{price}_w$

then let $w \in W$ be such a vertex;
let $u' \in U$ be such that $u'w \in \overline{M}$ (∗ u' may not exist ∗);
$\overline{M} \leftarrow \overline{M} - \{u'w\} \cup \{uw\}$;
$\overline{price}(w) \leftarrow \overline{price}(w) + 1$;

else $\overline{val}(u) \leftarrow \overline{val}(u) - 1$
fi
od

(∗ Phase 2: Hungarian search ∗)

while $\exists w \in W_{free}$
do select $w_0 \in W_{free}$, direct all unmatched edges from B to A, all matched edges from A to B and solve the shortest path problem with source w and distance function $\hat{c}_{uw} = \overline{val}(u) + \overline{price}(w) - \bar{c}_{uw}$. Let $\widehat{dist}_x$ be the shortest distance from w_0 to x, let $d = \min\{\widehat{dist}_u; u \in U_{free}\}$, let $u_0 \in U_{free}$ be such that $d = \widehat{dist}_{u_0}$, and let $dist_x = \min(d, \widehat{dist}_x)$. Augment the matching $\overline{M}$ along the shortest path from w_0 to u_0, i.e., reverse the direction of all edges on this path and set:

$$\overline{val}(u) \leftarrow \overline{val}(u) - dist(u) \text{ for all } u \in U$$
$$\overline{price}(w) \leftarrow price(w) + dist(w) - 1 \text{ for all } w \in W$$

(∗ $(\overline{val}, \overline{price})$ is dominating and 1–tight for $\overline{M}$ and $\bar{c}$ ∗)
od
end

Program 9: The auction algorithm

(d) Each iteration of the while–loop in Phase 2 takes time $O(m) = O(n^2)$, since the shortest path calculation uses a nonnegative distance function and since $d \leq n$ always; cf. [?] for details.

▮

We now discuss the implementation of Phase 1. Let us call an edge uw *tight*, if $c_{uw} = \overline{val}(u) + \overline{price}(w)$. It is clear that a non–tight edge uw can only become tight by a decrement of $\overline{val}(u)$. The search for tight edges in Phase 1 can therefore be done as follows: For each vertex $u \in U_{free}$ we maintain a pointer into u's adjacency list such that all edges to the left of the pointer are nontight. If u is selected in Phase 1 the pointer is advanced until a tight edge, say uw, is found. This edge is added to $\overline{M}$ and becomes non–tight by the increment of $\overline{price}(w)$. If no tight edge is encountered $\overline{val}(u)$ is decreased and u's pointer is reset to the first edge on u's adjacency list. In this way, each adjacency list is scanned at most $\gamma\sqrt{n}$ times for a total cost of $\gamma n^{2.5}$. Note, however, that only $\gamma n^{3/2}$ tight edges are found in Phase 1 according to Lemma 12. We may therefore hope to speed up the search for tight edges by the compression method. The details are as follows:

Let $d_{uw} = \overline{c}_{uw} - \overline{val}(u) - \overline{price}(w)$. We maintain a matrix $D[u,w], (u,w) \in U \times W$, which approximates d_{uw}. More precisely, $D[u,w] \in [-z..0] \cup \{-\infty\}$, where $z = O(\log n)$ is a constant to be fixed later, and $D[u,w] = 0$ if and only if $uw \in E$ and $d_{uw} = 0$. Let $b \geq \lceil \log(z+2) \rceil$, i.e., b bits suffice to encode an entry of $D[u,w]$, and let $t \in \mathbb{N}$ be such that $t^2 \cdot b \leq \log n$. The actual value of t will be fixed later. We partition the matrix D into $t \times t$–submatrices and store each submatrix in a single RAM–word.

We now show how to maintain the invariant that $D[u,w] = 0$ if and only if $uw \in E$ and $d_{uw} = 0$, and how to search for tight edges. In order to maintain the invariant, we recompute for each $x \in U \cup W$ the row (if $x \in U$) or column (if $x \in W$) of D corresponding to x after every $z/2$ changes of $\overline{val}(x)$ or $\overline{price}(x)$ according to the following rules:

$$\text{If } x \in U \text{ then } D[x,w] = \begin{cases} d(x,w) & \text{if } xw \in E \text{ and } d(x,w) \geq -z/2 \\ -\infty & \text{if } xw \notin E \text{ or } d(x,w) < -z/2 \end{cases}$$

$$\text{If } x \in W \text{ then } D[u,x] = \begin{cases} d(u,x) & \text{if } ux \in E \text{ and } d(u,x) \geq -z/2 \\ -\infty & \text{if } ux \notin E \text{ or } d(u,x) < -z/2 \end{cases}$$

This takes time $O(n)$ for each recomputation and hence $O(\gamma n^{2.5}/z)$ throughout Phase 1.

Next we turn to the search for tight edges . Since the matrix D reflects the 0–values of d correctly, we only need to search the (compressed) matrix D for entries of value 0. This takes time $O(n/t + \#$ of tight edges found) for each scan of an adjacency list given appropriate tables. The total time throughout Phase 1 is therefore $O(n^{2.5}\gamma/t + \gamma n^{3/2})$.

Finally, whenever $\overline{val}(u)$ or $\overline{price}(w)$ is changed the appropriate row or column of D has to be updated. We assume that $-\infty + 1 = -\infty$, and $-z - 1 = -z$. Given appropriate tables, each scan takes time $O(n/t)$ for a total of $O(n^{2.5} \cdot \gamma/t)$ throughout Phase 1. Also, the various tables mentioned above can certainly be computed in time $O(2^{2t^2 b})$.

We summarize in:

Lemma 13 *The cost of a call of auction is* $T_{auction} = O(2^{2b^2b} + n^{2.5}\gamma/t + n^{2.5}\gamma/z + \gamma n^{5/2})$ *where the constants* b, t *and* z *must satisfy* $b \geq \lceil \log(z+2) \rceil$ *and* $2t^2b \leq \log\ n$.

The choices $t = z = \lfloor (\log n / 2 \log\log n)^{1/2} \rfloor$ and $\gamma^2 = t$ yield:

$$T_{auction} = O(n^{2.5}(\log\log n / \log n)^{1/4}).$$

We have thus proved.

Theorem 8 *A maximal assignment in a bipartite network* (U, W, E, c) *where* $|U| = |W| = u$ *and* $c_{uw} \in [0..C]$ *for all* $uw \in E$ *can be computed in time* $O(n^{2.5}(\log\log n/\log n)^{1/4}\log(nC))$ *by a conservative algorithm.*