Balanced Network Flows. III. Strongly Polynomial Augmentation Algorithms

Christian Fremuth-Paeger, Dieter Jungnickel

Lehrstuhl für Diskrete Mathematik, Optimierung und Operations Research,
University of Augsburg, D-86135 Augsburg, Germany

Received 8 December 1997; accepted 29 April 1998

Abstract: We discuss efficient augmentation algorithms for the maximum balanced flow problem which run in $O(nm^2)$ time. More explicitly, we discuss a balanced network search procedure which finds valid augmenting paths of minimum length in linear time. The algorithms are based on the famous cardinality matching algorithm given by Micali and Vazirani. A comprehensive description of the double depth first search is included. © 1999 John Wiley & Sons, Inc. Networks 33: 43–56, 1999

Keywords: capacitated matching problems; network flows; augmenting a matching; balanced flow networks; skew-symmetric graphs; antisymmetrical digraphs; augmenting paths of minimum length; double depth-first search

PRELIMINARIES

Balanced flow networks were studied extensively in [4]. Solving the maximum balanced flow problem means solving the wide range of nonweighted matching problems. In [5], we gave a large amount of pseudocode for solving this problem. In particular, we proposed an augmentation procedure and a disjoint set union mechanism which will be reused here.

Balanced networks are defined on **skew-symmetric** graphs. The inherent symmetry partitions the node set into pairs of **complementary** nodes and the arc set into pairs of complementary arcs. The paths used for augmentation are **valid,** that is, they avoid complementary arc pairs with residual capacity one. Furthermore, augmentation is always done pairwise on complementary paths.

The most important part of the augmentation algorithm is the **balanced network search** procedure *BNS* which checks whether a valid augmenting path exists or not. The implementations in [5] are highly efficient if the maximum flow value is small compared with the size of the original graph G, in particular, if G is simple. However, this was not a polynomial time procedure in the general setting.

Here, we show how to derive augmenting paths of minimum length. Our BNS procedure heavily depends on the **double depth first search** method, which was discussed by Vazirani [10] before. The idea applies to other BNS procedures as well. For this reason, we describe the general setting.

An arc a of a balanced network N which can be accessed from the source by a valid path is called **strictly accessable.** If the complementary arc a' is also strictly accessable, then a is called **bicursal** and, otherwise, **uni-**

Correspondence to: D. Jungnickel; e-mail: jungnickel@math.uni-augsburg.de

The results of this paper form part of the first author's doctoral thesis which has been written under the supervision of the second author.

AMS subject classification: 05C70, 90B10, 90C35

cursal. A **blossom** of N is a connected component of $N[Bic]$, where Bic consists of all bicursal arcs in N. This is not an algorithmic definition, but coincides with the definition of Edmonds [2]. The most important result about blossoms is the following:

Theorem 18.1 (base identity). *Let B is a blossom not containing the source s. Then, there is an arc $a = ub$ with $rescap(a) = 1$ which is traversed by every valid path p connecting s to a node v in B.*

The node b of this theorem is called the **base,** and the arc a is called the **prop** of the blossom B. The blossom containing a node v is denoted by $B(v)$.

In [4], we showed that a node v is in a blossom iff v and the complementary node v' are strictly reachable. If v is strictly reachable, but v' is not, we call the set $\{v, v'\}$ a **bud** with **base** v. The **props** of $B(v)$ are the arcs by which v is reached on a valid sv-path of minimum length. Buds are treated as blossoms. Where it is necessary to exclude buds, we speak of **proper** blossoms.

The **layered auxiliary network** $Aux(N)$ corresponds to shrinking the blossoms of the original network N: The nodes of $Aux(N)$ are the blossom bases, and the arcs correspond to the props. More explicitly, two bases u, v are joined by an arc uv for every prop wv with $u = base(B(w))$. In that case, we assign $auxcap(uv) := rescap(wv)$.

Using explicit distance labels and the base identity, it turns out that $Aux(N)$ is indeed acyclic. Note that directed paths in $Aux(N)$ correspond to minimum length valid paths in N.

Of course, the blossom structure with respect to the layered auxiliary network is computed recursively by the BNS algorithm from certain subnetworks $N[A]$. Here, $A \subseteq A(N)$ shall denote the set of arcs which have been investigated so far. In what follows, let A be chosen such that

($\alpha 1$) A is self-complementary,

($\alpha 2$) A does not contain (A)-acursal arcs,

($\alpha 3$) Every (A)-unicursal arc in A is an (A)-prop.

Under these assumptions, $Aux(N[A])$ is obtained from $N[A]$ by shrinking the blossoms and deleting all nonaccessable arcs.

Suppose that $a = uv' \notin A$ is investigated next and u and v are strictly (A)-reachable, but in different (A)-blossoms. This is the only situation where (A)-blossoms have to be merged and to which we refer in the next section. The arc a is called an α-**bridge.**

Concerning the iterated network $N[\tilde{A}]$, with $\tilde{A} := A \cup \{a, a'\}$, we showed in [4] that

- Both u and v are in a common proper $(\tilde{A})$-blossom, denoted by $B_{\tilde{A}}(u, v)$.
- Except for $B_{\tilde{A}}(u, v)$, every $(\tilde{A})$-blossom is an (A)-blossom.
- An (A)-blossom B is contained in $B_{\tilde{A}}(u, v)$ iff $base_A(B)$ is on some directed $(b, base_A(u))$-path or $(b, base_A(v))$-path in $Aux(N[A])$ (Here we have put $b := base(B_{\tilde{A}}(u, v)))$.

Let a be an arc of the layered auxiliary network $Aux(N)$ with $auxcap(a) = 1$, and let v be some blossom base of N. If a is on every directed sv-path in $Aux(N)$, we call a a **bottleneck** of v. Note that the bottlenecks of v are ordered by the distance labels of their end nodes.

Actually, BNS algorithms do not compute the blossom base b but rather the arc $bot_A(u, v)$ of the layered auxiliary network $Aux(N[A])$ which denotes the (A)-bottleneck of $base_A(u)$ and $base_A(v)$ with a maximum distance label. If x is the unique (A)-predecessor of b, then $bot_A(u, v)$ connects $base_A(x)$ and b in the layered auxiliary network.

19. DOUBLE DEPTH FIRST SEARCH

The algorithms presented in [5] are very efficient implementations of the balanced network search and just as simple as possible, since the occurring layered auxiliary networks are trees. Unfortunately, no BNS algorithm is known which finds minimum-length valid augmenting paths in general and that grows the network as a tree. In this section, we will describe a procedure called **double depth first search** (DDFS) which is shown in Procedure 1 and searches the layered auxiliary network $Aux(N[A])$ in order to determine the bottleneck $bot_A(u, v)$.

Procedure 1. The Double Depth First Search Algorithm

```
function DDFS(bridge);
var a, b, barrier, left, next, right, root;

  procedure LeftBacktrack;
  begin
  if $left \neq root$ and $left \neq s$
  then $left := LeftProp[left]^+$
  else
    if $left = s$ then $b := s$ else
      $RightProp[right] := NO\_ARC$;
      $RightSupport.DELETE(right)$;
      $b := barrier$
    fi;
    $FirstProp[bridge] := LeftProp[b]$;
    $FirstProp[bridge'] := RightProp[b]$;
  exit[bridge] := b;
  exit[bridge'] := NONE
 fi
 end;


 procedure RightBacktrack;
 begin
 if right ≠ barrier
 then right := RightProp[right]^+
 else
   right := left;
   barrier := LeftProp[left]^+;
   RightProp[right] := LeftProp[right];
   RightSupport.INSERT(right);
   LeftProp[left] := NO_ARC;
   LeftSupport.DELETE(left);
   left := barrier
 fi
 end;


begin
left, root := BASE(bridge^-);
right, barrier := BASE(bridge^+);
LeftSupport.INSERT(left);
RightSupport.INSERT(right);
Props[left].ACCESS_L;
Props[right].ACCESS_R;
  b := NONE;
  while b = NONE do
    if d[left] ≥ d[right]
    then
      if Props[left].EOS_L
      then LeftBacktrack
      else
        Props[left].READ_L(a);
        next := BASE(a^-);
        if rescap(a) > 1 or RightProp[next] ≠ a
        then
          if LeftProp[next] = NO_ARC then
            LeftProp[next] := a;
            LeftSupport.INSERT(next);
            Props[next].ACCESS_L;
            left := next
          fi
        fi
      fi
    else
      if Props[right].EOS_R
      then RightBacktrack
      else
        Props[right].READ_R(a);
        next := BASE(a^-);
        if rescap(a) > 1 or RightProp[next] ≠ a
        then
          if RightProp[next] = NO_ARC then
            RightProp[next] := a;
            RightSupport.INSERT(next);
            Props[next].ACCESS_R;
            right := next
          fi
        fi
      fi
    fi
  od;
  return b
  end.
```

We assume that $DDFS(a)$ is started with the following environment: The arcs of the layered auxiliary network $Aux(N[A])$ with end node v are encoded into the queue $Props[v]$ which consists of the (A)-props. The label $d[v]$ is the distance $d_A(v)$ between s and v on a valid path in $N[A]$. In our later BNS algorithm, $d[v]$ is the correct distance for every node v reached so far.

Initially, *LeftSupport* and *RightSupport* are empty *STACK* objects, and we have $LeftProp[u]$, $RightProp[u] = NO\_ARC$ for every (A)-base u. The base of the (A)-blossom containing a node v can be computed by $BASE(v)$, using the DSU pseudocode given in [5].

As the name suggests, $DDFS(a)$ grows two arborescences in a depth-first manner, called the **left** and the **right** DFS tree. These trees have roots $base_A(u)$ and $base_A(v)$, respectively, are represented in vectors *LeftProp* and *RightProp*, respectively, and grow toward the source s. The nodes *left* and *right* which occur are called the **active** nodes of their respective DFS trees. Under some circumstances, the queue $Props[v]$ is read simultaneously by the left and the right DFS. To distinguish the respective operations, we added L/R indices to the pseudocode.

If we have $d[left] \geq d[right]$ at some stage, an operation of the left DFS follows. In that situation, we call the left DFS **active** and the right DFS **inactive.** Otherwise, if we have $d[left] < d[right]$, an operation of the right DFS follows, and we call the right DFS **active** and the left DFS **inactive.**

Both trees may have some arcs in common, but share no (A)-arc $\tilde{a}$ which has $auxcap(\tilde{a}) = 1$. If both of the DFS are required to search such an arc $\tilde{a}$, then a is searched by the left DFS first, and the right DFS tries to circumvent $\tilde{a}$. If this fails, $\tilde{a}$ is included into the right tree and the left DFS tries to circumvent $\tilde{a}$. If this also fails, the arc $\tilde{a}$ is the desired bottleneck and $DDFS(a)$ halts.

The return value of $DDFS(a)$ is the base of the updated blossom. However, a call of *DDFS* has various side effects: At the end, the sets *LeftSupport* and *RightSupport* exactly consist of the blossom bases which are shrunk into $B_{\tilde{A}}(u, v)$.

The information of *LeftProp* and *RightProp* will be reused by our later augmentation algorithms for the expansion of $(\tilde{A})$-valid paths through $B_{\tilde{A}}(u, v)$. There are, however, some restrictions:

When the (A)-blossoms are merged into $B_{\tilde{A}}(u, v)$ by *UNION* operations, the information of $base_A$ is lost. At some later point of time, also the information of $base_{\tilde{A}}$ is lost. Since we need $base_{\tilde{A}}(u, v)$ for the path expansion, we assign the $(\tilde{A})$-base $exit[bridge] := b$ to the investigated α-bridge. Note that $exit[bridge']$ remains undefined, and we can decide later whether the left DFS tree was rooted at $base_A(bridge^-)$ or $base_A(bridge^+)$.

The labels $LeftProp[b]$ and $RightProp[b]$ may be overwritten by a later call of $DDFS$. To this end, additional labels $FirstProp[bridge] := LeftProp[b]$ and $FirstProp[bridge'] := RightProp[b]$ must be assigned.

Theorem 19.1. *The call of* $DDFS(a)$ *returns the base of* $B_{\tilde{A}}(u, v)$. *Furthermore, LeftSupport and RightSupport consist of all* (A)*-blossom bases which have to be merged into* $B_{\tilde{A}}(u, v)$.

Proof. The call of $DDFS(a)$ inspects nodes and arcs of the layered network $Aux(N[A])$. Since layered auxiliary networks are acyclic, *LeftProp* and *RightProp* define forests. By induction on the iteration steps of the **while**-loop in $DDFS(u, v)$, we can show that

- *LeftProp* and *RightProp* define trees with node sets *LeftSupport* respectively *RightSupport,*
- No arc $\tilde{a}$ with $auxcap(\tilde{a}) = 1$ can be in both trees,
- No arc can leave the right search tree,
- An arc can only leave the left tree if it is included into the right tree immediately,
- At least one of the active nodes *left* and *right* is a leaf of the corresponding tree,
- $min(d[left], d[right])$ cannot increase during one iteration step.

It is obvious that these conditions are satisfied initially. The induction step is straightforward if the current iteration step is an ordinary DFS operation, namely, an extension of one of the trees or an ordinary backtracking operation, that is, a backtracking with $right \neq barrier$ respectively $left \neq root$.

We first consider those iterations during which *RightBacktrack* is called with $right = barrier,$ especially the beginning of such an iteration, and some leaf x of the right tree with $d[x] = min\{d[z] : z \in RightSupport\}$. There is some node $y \in LeftSupport$ with $d[y] < d[x]$ (Otherwise, the right DFS would have been inactive since x was reached.) Note that the left DFS was inactive since y was reached. Hence, y is the unique node in *LeftSupport* with $d[y] < d[x]$ and a leaf of the left DFS tree.

Since some arc a in the layered network with end node x exists, but the right DFS did not include this arc, a has been included into the left DFS tree before. By the above argument, we have $a = (y, x)$ and $y = left$. But y is a leaf and can be included into the right tree by $RightProp[y] := LeftProp[y]$ and deleted from the left tree by $LeftProp[y] := NO\_ARC$. The induction step is now obvious.

These calls of *RightBacktrack* are also interesting for another reason: Actually, a backtracking of the left DFS occurs, which proves, in turn, that any arc of the left search tree is traversed once as a forward edge and once as a backward edge. The assignment $barrier := x$ prevents the right DFS from backtracking on any arc more than once. Thus, the call of $DDFS(a)$ terminates, since every arc of the layered network is considered at most four times.

Finally, we consider the point before the last call of *LeftBacktrack:* In the most simple case, we have $left = right = s$ and $DDFS(a) = s$, but also $base_{\tilde{A}}(u, v) = s$, since, otherwise, $bot_A(u, v)$ would occur in both search trees, a contradiction.

Next, assume that $s \neq base_{\tilde{A}}(u, v)$. Then, we have $barrier = RightProp[right]^+$ and $barrier = left$ immediately after the last call of *RightBacktrack*. Thus, the node *barrier* is contained in both search trees. Let p be any directed $(s, barrier)$-path in $Aux(N[A])$, q the directed $(barrier, BASE(u))$-path induced by *LeftProp,* and r the directed $(barrier, BASE(v))$-path induced by *RightProp*. Each of $p \circ q$ and $p \circ r$ traverses the bottleneck $bot_A(u, v)$ which even is on p, since, otherwise, the search trees would share an arc a with $auxcap(a) = 1$.

Since *RightBacktrack* was executed the last time, the right DFS was inactive and therefore *right* and *barrier* did not change. It is now obvious that the left DFS has backtracked from any node $\mathbf{x} \in LeftSupport$ and the right DFS has backtracked from any node $\mathbf{x} \in RightSupport \setminus \{right\}$. Therefore, all arcs of the layered auxiliary network with end node $\mathbf{x}$ have been investigated. To see that $(right, barrier)$ is a bottleneck, we have to show that it is the unique arc of the layered auxiliary network with end node in and start node not in $Support := LeftSupport \cup (RightSupport \setminus \{right\})$.

Let (y, x) be any arc of the layered auxiliary network not equal (but possibly parallel) to $(right, barrier)$ so that y is not in *Support,* but x is in *Support*. If y and *right* would be different, then (y, x) would have been included into one of the search trees during the investigation of x, contradicting the choice of $y \notin Support$. Thus, $y = right$ holds.

By the previous discussion, we observe the inequality $d[right] < min\{d[z] : z \in Support\}$ and also that *right* has been included into each of the trees. But, then, also a node z with $d[z] < d[right]$ would have been explored, contradicting the above inequality. As a consequence, $(right, barrier)$ is a bottleneck of u and v. Since the search trees do not contain bottleneck arcs by Corollary 10.9, we have $base_{\tilde{A}}(u, v) = barrier = DDFS(a)$. ■

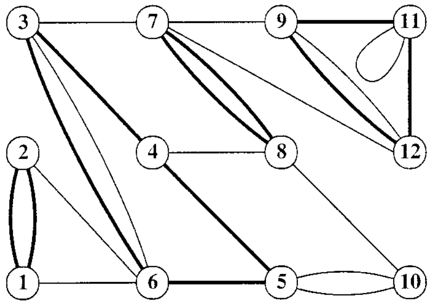

Fig. 1. A 2-factor search.

Up to this point, the discussion was rather general. We will now demonstrate the behavior of the DDFS by the sample graph given in Figure 1 and the search strategy which has been introduced in [4] for finding minimum valid paths. In fact, this coincides with the search strategy of the Micali/Vazirani [8] cardinality matching algorithm.

This graph has been taken in [4, 5] as a running example, and we will search for a 2-factor of this graph again. Let N denote the balanced flow network associated with this matching problem, and f, the balanced flow corresponding to the shown subgraph (see [4] for a formal description of the problem reduction mechanism).

Assume that A to be the set of (4)-arcs of the residual network $N(f)$ that are the props $(s, 10)$, $(10, 5')$, $(10, 8')$, $(5', 4)$, $(5', 6)$, $(8', 7)$, $(6, 1')$, $(6, 2')$, $(6, 3')$, $(7, 3')$, $(7, 9')$, $(7, 12')$ and their complementary arcs. Note that no proper (4)-blossoms and no (4)-bridges exist. There is a figure giving the networks $N_4(f)$, $N_5(f)$, and $N_6(f)$ in [4].

There are four different (5)-bridges of $N(f)$, namely, $(1', 2)$, $(9', 12)$, and their complementary arcs. We show how to compute the (5)-blossoms and especially describe the call $DDFS(1', 2)$ with the mentioned environment:

In the beginning, we have $root = left = BASE(u) = base_4(u) = 1'$, and $barrier = right = BASE(v) = base_4(v) = 2'$. Because of $d[left] = d[right] = 4$, a left DFS operation is performed, namely, a prop of $1'$ is chosen and included into the left tree by $LeftProp[6] := (6, 1')$. Then, *left* is updated to node 6.

In the next iteration, we have $3 = d[left] < d[right] = 4$ so that a right DFS operation takes place. The only prop $(6, 2')$ becomes an arc of the right DFS tree by $RightProp[6] := (6, 2')$ and *right* is updated to node 6.

Now, we have $d[left] = d[right]$ again so that a left DFS operation follows. Here, the arc $(5', 6)$ is read from $Props[6]$ which becomes an arc of the left DFS tree. The following is a right DFS operation again.

Since there are no further members to read from $Props[6]$, *RightBacktrack* is called which results in $right := 2'$. But we still have $d[left] < d[right]$. So, *RightBacktrack* is called once more. Since we have reached $right = barrier$, we move the arc $(5', 6)$ which the right DFS could not circumvent from the left to the right tree. The effects are $right := 5'$, $barrier = 6$ (so that we do not search the right tree twice), $RightProp[5'] := (5', 6)$, $LeftProp[5'] := NO\_ARC$, and $left := 6$.

Subsequently, we have $d[left] > d[right]$, and *LeftBacktrack* is called twice. The first call results in $left = 1'$ so that the second call finds $left = root$, and the search ends. The arc $(5', 6)$ is the required bottleneck and deleted from the right tree by the operation $RightProp[5'] := NO\_ARC$. The node $5'$ is deleted from *RightSupport* again by the operations *RightSupport. DELETE(right)*.

Before halting, the procedure puts $exit[(1', 2)] := 6$, $exit[(2', 1)] := NONE$, $FirstProp[(1', 2)] := (6, 1')$ and $FirstProp[(2', 1)] := (6, 2')$. Finally, we have $LeftSupport = \{1', 6\}$, $RightSupport = \{2', 6\}$, and the call of $DDFS(1', 2)$ returns the node 6 which is the base of a (5)-blossom.

In Table I, we list two further examples for the course of the DDFS in case of our running example. We only mention what is different compared to the example discussed above: The investigation of the second (5)-bridge $(9', 12)$ is applied to the updated labels d, with $A := A_4 \cup \{(1', 2), (2', 1)\}$. Actually, the calls of $DDFS(1', 2)$ and $DDFS(9', 12)$ are independent, and each of them traverses the nodes of a (5)-blossom.

By the call of $DDFS(9', 12)$, $LeftProp[7] := (7, 9')$, $RightProp[7] := (7, 12')$, and $left = right = 7$ are set first. In the next iteration, $LeftProp[8'] := (8', 7)$ and $left := 8'$. Since we have $rescap(8', 7) = 2$, this arc is available to the right DFS also. Indeed, $RightProp[8'] := (8', 7)$ and $right := 8'$ is set in the following step. The remaining operations can be seen from Table I.

The call $DDFS(9', 12')$ returns the node $8'$ which is the base of the (5)-blossom containing 7, 8, 9, 12, and their complements. The searched nodes are collected into $LeftSupport = \{9', 7, 8'\}$ and $RightSupport = \{12', 7, 8'\}$. As additional effects, we have $exit[(9', 12)] := 8'$, $exit[(12', 9)] := NONE$ and $FirstProp[(9', 12)]$, $FirstProp[(12', 9)] := (8', 7)$.

As a final example, we consider the (6)-bridge $(3', 6)$ and call $DDFS(3', 6)$ with $A = A_5$, $BASE \equiv base_5$, and $d \equiv d_5$. This is interesting for two new operations which occur. First, the arc $(5', 6)$ is searched which triggers off back-tracking operations of both DFS but is not the desired bottleneck. Circumventing this arc, the DDFS reaches $left = right = s$, and no further backtracking operations occur. At the end, we have $LeftSupport = \{3', 6, 8', 10, s\}$ and $RightSupport = \{6, 5', 10, s\}$.

Note that $DDFS(3', 6)$ finds an augmenting path encoded into the auxiliary network $N_5(f)$. The expansion of this path is presented later.

TABLE I. Examples for the double depth first search

	Left	*Right*	*Barrier*
DDFS(1′, 2)	1′	2′	2′
	6	—	—
	—	6	—
	5′	—	—
	—	2′	—
	6	5′	6
	1′	—	—
DDFS(9′, 12)	9′	12′	12′
	7	—	—
	—	7	—
	8′	—	—
	—	8′	—
	10	—	—
	—	7	—
	—	12′	—
	8′	10	8′
	7	—	—
	9′	—	—
DDFS(3′, 6)	3′	6	6
	6	—	—
	5′	—	—
	6	5′	6
	3′	—	—
	8′	—	—
	10	—	—
	—	10	—
	s	—	—
	—	*s*	—
	—	—	*s*

20. FINDING MINIMUM VALID PATHS

In this section, we will study a complete BNS algorithm. Like the simple algorithm discussed in [5], it will compute the blossoms of a given network. Even more, the algorithm introduced now is able to determine the correct distance label $d(v)$ and a $d(v)$-path for every strictly reachable node v. These paths are encoded into some data structures, and we will also give a corresponding path expansion rule. The BNS algorithm is shown in Procedure 2.

Procedure 2. Determination of the Distance Labels

```
class BALANCED_NW;
private
  array Props of object: QUEUE;
  array petal, exit, FirstProp, LeftProp, RightProp;
  object LeftSupport, RightSupport: STACK;

  function DDFS;

  function BNS(s, t);
  array Anomalies, Bridges, Q of object: QUEUE;
  var i, v;

    procedure Min(i);
    var a, v, w;
    begin
    while not Q[i].EMPTY do
      Q[i].DELETE(v);
      while not EOS(v) do
        READ(v, a);
        w := $a^+$;
        if $d[w'] = \infty$ and $d[w] > i$ then
          if $d[w] = \infty$ then
(1)         Props[w].INIT;
            Anomalies[w'].MAKE(m);
            $d[w] := i + 1$;
            F.BUD(w);
            base[F.FIND(w)] := w;
            Q[i + 1].INSERT[w]
          fi;
          Props[w].INSERT(a)
        else
(2)        if $d[v'] \neq d[w'] + 1$ or $d[v'] > d[v]$ then
            if $d[w'] < \infty$
            then Bridges[$d[v] + d[w'] + 1$].
              INSERT(a)
            else Anomalies[w'].INSERT(a)
            fi
          fi
        fi
      od
    od
    end;

    procedure ShrinkBlossom(tenacity, CurrentPetal);
    var a, w;
    begin
    b := DDFS(CurrentPetal);
    while not LeftSupport.EMPTY do
      LeftSupport.DELETE(w);
      F.UNION(b, w);
      if $d[w'] = \infty$ then
        $d[w'] := tenacity - d[w]$;
        Q[d[w']].INSERT(w');
        while not Anomalies[w'].EMPTY do
          Anomalies[w'].DELETE(a);
          Bridges[$d[a^-] + d[w'] + 1$].INSERT(a)
        od;
        petal[w'] := CurrentPetal'
        fi
    od;
    while not RightSupport.EMPTY do
        RightSupport.DELETE(w);
        F.UNION(b, w);
        if $d[w'] = \infty$ then
            $d[w']$ := tenacity $-$ $d[w]$;
            $Q[d[w']]$.INSERT(w');
            while not Anomalies[w'].EMPTY do
                Anomalies[w'.DELETE(a);
                Bridges[$d[a^-] + d[w'] + 1$].INSERT(a)
            od;
            petal[w'] := CurrentPetal
        fi
    od;
    base[F.FIND(b)] := b;
    LeftProp[b], RightProp[b] := NO_ARC
    end;

    procedure Max(i);
    var a, b;
    begin
    while not Bridges[$2i + 1$].EMPTY do
        Bridges[$2i + 1$].DELETE(a);
        if BASE($a^-$) $\neq$ BASE($a^+$) or $d[a^-] = \infty$
        then ShrinkBlossom($2i + 1$, a)
        fi
    od
    end;

begin
allocate Anomalies[0, 1, . . . , n], Bridges[1, 3, . . . , 2n
    + 1], Q[0, 1, . . . , n];
F.INIT;
for v := 0 to n do
    $d[v] := \infty$;
    LeftProp[v], RightProp[v] := NO_ARC
od;
for i := 0 to n do
    Q[i].MAKE(n);
    Bridges[$2i + 1$].MAKE(m)
od;
d[s] := 0;
F.BUD(s);
base[F.FIND(s)] := s;
Q[0].INSERT(s);
INVESTIGATE;
for i := 0 to $n - 1$ do
    Min(i);
    Q[i].FREE;
    Max(i);
    Bridges[$2i + 1$].FREE
od;
CLOSE;
for v := 0 to n do Anomalies[v].FREE od;
    disallocate Anomalies, Bridges, Q;
    if $d[t] < \infty$ then return TRUE else return FALSE fi
    end
end.
```

As suggested by the theory of Section 12 in [4], we will compute the iterated labels d_{i+1} from d_i, each of which corresponds to a particular subgraph of the network. Note that the BNS algorithm consists of two procedures *Min* and *Max* and that *Min*(i) reaches the minlevel nodes v with $d(v) = i + 1$ while *Max*(i) reaches the maxlevel nodes w with $t(w) = 2i + 1$. To prove the correctness of the algorithm, we use as an induction hypothesis that right before the call of *Min*(i) the following is true:

- We have $d[v] = d_i(v)$ and $BASE(v) = base_i(v)$ for every node v.
- $Q[i]$ consists of all nodes v with distance label $d(v) = i$.

It is obvious that both statements are true when *Min*(0) is called. So, assume that the hypothesis is true when *Min*(i) is called for some i.

Then, the call of *Min*(i) does the following: All nodes v with $d[v] = d(v) = i$ are expanded so that all arcs a with start node v are considered. Let $w := a^+$ as in the algorithm. We will show that a is investigated by case (1) if it is a prop and that a is inspected (but not yet investigated!) by case (2) if it is a bridge. We consider case (1) first.

If we have $d[w'], d[w] = \infty$, then w actually is a minlevel node with $d(w) = i + 1$. The algorithm constructs the set *Props*$[w]$ containing a, generates the bud $\{w, w'\}$, and assigns the correct distance label. If we have $d[w'] = \infty$ and $i < d[w] < \infty$, then w has been explored by *Min*(i) before, so that we have $d[w] = i + 1$, and a must be appended to *Props*$[w]$.

Note that every prop xy is investigated by such an operation if we have $d(y) = i + 1$. [We have $d(x) = i$ by Lemma 8.2, and $d_i(x) = d(x)$ holds by Corollary 12.3 and the induction hypothesis.]

In all other cases, a is not a prop, and the case (2) of procedure *Min* is reached. Note that a is either a bridge or a co-prop, but co-props are excluded from inspection. If $d[w']$ is finite, then we have $d[w'] = d(w')$ by the induction hypothesis and Theorem 12.1, and a is appended to *Bridges*$[t(a)]$. Otherwise, a is appended to the set *Anomalies*$[w']$. Resolving anomalies is the critical part of the algorithm, and it is necessary to say something about that task.

Anomalies have been introduced in Section 11 where only tree growing BNS algorithms were studied. In the general case, an **anomaly** is an α-bridge uv which is searched by some BNS algorithm, but the tenacity label $t(u, v)$ is infinite at this point of time. In the concrete algorithm, bridges and α-bridges are the same.

An anomaly must be resolved if v becomes strictly reachable later. For this goal, all anomalies with end node w are collected into the set *Anomalies*$[w']$. Note that w' always is a maxlevel node and that the correct tenacity labels of these arcs are available once the bud $\{w, w'\}$ is shrunk into a proper blossom. Then, the queue *Anomalies*$[w']$ is flushed, and every member a is placed on the queue *Bridges*$[t(a)]$.

To perform $Max(i)$, we must know all bridges $a = vw$ with $t(a) = 2i + 1$. Note that we have $d(v) \le i$ or $d(w') \le i$ or both. Without loss of generality, we assume that $d(v) \le d(w')$. Thus, a has been searched by a previous call of *Min*, namely, $Min(d(v))$. But if a has been recognized as an anomaly, it must be resolved in time. By Lemma 12.8, w' is also strictly (i)-reachable. Hence, the anomaly a has been resolved by the call of $Max(j)$ where we have $t(w') = 2j - 1$ and $j < i$.

At the start of $Max(i)$, the queue *Bridges*$[2i + 1]$ does not contain every bridge a with $t(a) = 2i + 1$, but at least one of a complementary pair. The members of *Bridges*$[2i + 1]$ are considered successively, and exactly one arc a of a complementary pair is searched by the call of $b := DDFS(a)$. By Theorem 19.1, *LeftSupport* and *RightSupport* consist of those (i)-bases, which must be merged together by the investigation of a and a'. It is obvious that the blossoms are merged and that the d-labels are assigned correctly. Since *LeftProp*$[b]$, *RightProp*$[b]$:= *NO_ARC* is set at the end of the investigation step, a further DDFS may run.

The induction step is obvious now. Hence, we get the following correctness result:

Theorem 20.1. *After the call of BNS, we have* $d[v] = d(v)$ *and* $base[v] = base(v)$ *for every node* v. ■

Theorem 20.2. *A call of BNS needs* $O(m\alpha(m, n))$ *time.*

Proof. Any node v can be in only one of the $Q[i]$'s, and any arc can be inspected by *Min* only once, namely, during the expansion of its start node. In particular, the calls of *Min* require $O(m)$ time altogether.

So let us consider the performance of the procedure *Max*. A bridge a can be contained in the set *Bridges*$[2i + 1]$ only if $t(a) = 2i + 1$ holds and can be an anomaly of at most one node. After the call of $b := DDFS(a)$, *LeftSupport* and *RightSupport* together consist of all nodes searched by the DDFS operation but the predecessor of b. The only node searched by $DDFS(a)$ which can be searched by a later DDFS operation again is the base b. These two nodes do not affect the time complexity since only $O(m)$ calls of *DDFS* are needed.

A similar statement holds for the arcs searched by $DDFS(a)$. To see this, note that the auxiliary arcs can be treated as props of the original network and that no arc searched by $DDFS(a)$ can be searched again, except for the unique prop of b. But any arc can be traversed at most four times by a DDFS operation. If we exclude the DSU process from consideration, we see that the BNS runs in $O(m)$ time.

To get the time bound $O(m\alpha(m, n))$ for the DSU process, one must show that the total number of *BASE* calls is $O(m)$. This is easy and left to the reader. ■

TABLE II. The investigation order of the procedure *Min*

i	$Q[i]$
0	s
1	10
2	$5', 8'$
3	4, 6, 7
4	$1', 2', 3', 9', 12'$
5	11, 1, 2, 9, 12
6	$6', 7', 11'$
7	8, 3
8	$4'$
9	5
10	$10'$
11	t

We describe the effects of this BNS algorithm when searching the residual network of our running example. Some parts have been discussed in the last section so that we must only fill the gaps. Note that the distance labels computed by the procedure can be seen from Table II in which the node sets $Q[0], Q[1], \ldots, Q[n]$ are shown.

In this example, the procedure $Max(i)$ is inactive for $i < 4$ since no bridges with tenacity < 9 exist. The procedure grows the auxiliary layered network treelike, and every searched arc is recognized as a prop until the node 4 is searched. In that order, the nodes 10, $5'$, $8'$, 4, 6, and 7 are reached and placed on $Q[1]$, $Q[2]$, and $Q[3]$, respectively, depending on their distance label d.

Then, the call of $Min(3)$ does the following: First, node 4 and arc $(4, 8')$ are searched. Because of $d(8') = 2 < d(4) + 1$, we have found a bridge, but still $d(8) = \infty$ holds. So the arc $(4, 8')$ is added to the set *Anomalies*$[8]$. Subsequently, the nodes 6 and 7 are searched which have the common successor $3'$. Since 6 is searched before 7, the bud $3'$ is generated during the investigation of $(6, 3')$, and $(6, 3')$ is placed on *Props*$[3']$ before $(7, 3')$. We also reach $1'$, $2'$, $9'$, and $12'$ during that stage.

Then, $Min(4)$ is called, and $Q[4]$ is scanned. While $1'$ and $2'$ are expanded, the bridges $(1', 2)$ and $(2', 1)$

TABLE III. The investigation order of the procedure *Max*

t	$Bridges[t]$
9	(1′, 2), (2′, 1), (9′, 12), (12′, 9)
11	(3′, 6), (4, 8′), (11, 11′), (9, 12′), (12, 9′)
13	(3′, 4)

are found, each of which is placed on *Bridges*[9]. By the expansion of 3′, we find two bridges (3′, 4) and (3′, 6) with unknown tenacity which are appended to *Anomalies*[4′] and *Anomalies*[6′], respectively. Next, the nodes 9′ and 12′ are expanded. During these operations, (9′, 12) and (12′, 9) are placed on *Bridges*[9] and the arcs (9′, 11) and (12′, 11) are appended to *Props*[11].

Next, *Max*(4) is called which inspects the arcs on *Bridges*[9]. The first arc inspected is (1′, 2), and since (1′, 2) is not contained in a proper blossom yet, *DDFS*(1′, 2) is called. This call has been discussed as an example for the DDFS in the last section. As a result, the (5)-blossom {6, 6′, 1, 1′, 2, 2′} is constructed. To the nodes 6′, 1, and 2 which have become strictly reachable by the investigation step, we assign *petal*[6′] := (2′, 1), *petal*[1] := (2′, 1), and *petal*[2] := (1′, 2). Furthermore, we get *d*[6′] = 6; thus, *Anomalies* [6′] is flushed, and (3′, 6) is placed on *Bridges*[11].

Then, *Max*(4) inspects the bridge (9′, 12) by the call of *DDFS*(9′, 12) which has been already discussed. This operation determines the other (5)-blossom {8, 8′, 7, 7′, 9, 9′, 12, 12′} and assigns *petal*[8] := (12′, 9), *petal*[7′] := (12′, 9), *petal*[9] := (12′, 9), and *petal*[12] := (9′, 12). We get *d*[9] = *d*[12] = 5, *d*[7′] = 6, and *d*[8] = 7. Hence, the queue *Anomalies*[8] is flushed, and (4, 8′) is appended to *Bridges*[11]. Note that *Max* suppresses the investigation of the bridges (2′, 1) and (12′, 9), since these arcs are already contained in a proper blossom.

The following call of *Min*(5) finds the bridges (11, 11′), (9, 12′), and (12′, 9) which are appended to *Bridges*[11] in that order. Then, the arcs of *Bridges*[11] are inspected by the call of *Max*(5). The first arc drawn is (3′, 6) so that *DDFS*(3′, 6) is triggered off. Again, this was one of our examples for the DDFS in the last section. After that operation, we assign *d*[*t*] = 11, *d*[10′] = 10, *d*[3] = 7, *d*[5] = 9, as well as *petal*[*t*], *petal*[10′], *petal*[3] := (6′, 3), and *petal*[5] := (3′, 6) and merge into the blossom

$$\{s, t, 10, 10', 5, 5', 6, 6', 1, 1', 2, 2', 3, 3', 7, 7', 9, 9', 12, 12', 8, 8'\},$$

which is not yet a (6)-blossom. There are two further

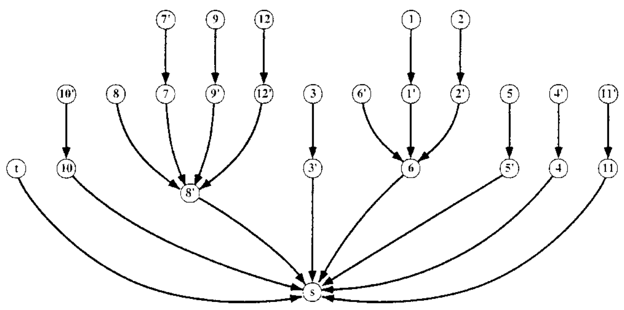

Fig. 2. A DSU tree of the running example.

calls of the DDFS nested into *Max*(5). These are almost trivial and may be left to the reader.

By the investigation of (4, 8′), the bud {4, 4′} is merged into the former blossom and *d*[4′] = 8 and *petal*[4′] := (8, 4′) are assigned. Moreover, *Anomalies*[4′] is flushed, and (3′, 4) is placed on *Bridges*[13]. By the investigation of (11, 11′), the bud {11, 11′} is merged into the former blossom so that all nodes of the network are in a common blossom now. In particular, there are no DDFS operations after the call of *DDFS*(11, 11′). The whole DSU tree for this BNS example is shown in Figure 2.

21. PATH EXPANSION

As mentioned before, there is a valid *st*-path encoded into the information of *petal, LeftProp, RightProp, FirstProp, Props,* and *Exit*. A path expansion rule which is compatible with the BNS algorithm of Procedure 2 is shown in Procedure 3. Unfortunately, the pseudocode presented is rather lengthy since we have to distinguish several cases.

Procedure 3. Minimum Valid Path Expansion

```
class BALANCED_NW;
private
  procedure EXPAND(x, y);
  var a, b, w;
  begin
  if $x \neq y$ then
    if $d[y] < d[y']$ then
      $a := Props[y].FIRST$;
      $EXPAND(x, a^-)$;
      $p[y] := a$
    else
      if $exit[petal[y]] \neq NONE$ then
        $b := exit[petal[y]]$;
        $EXPAND(x, b)$;
        $w := b$;
        $a := FirstProp[petal[y]]$;
      while $a \neq NO\_ARC$ do
        $EXPAND(w, a^-)$;
        $p[a^+] := a$;
        $w := a^+$;
        $a := LeftProp[w]$
      od;
      $EXPAND(w, petal[y]^-)$;
      $p[petal[y]^+] := petal[y]$;
      $w := y$;
      if $y = b'$ then $a := FirstProp[petal[y]']'$ else
        $a := RightProp[y']'$ fi;
      while $a \neq NO\_ARC$ do
        $CO\_EXPAND(a^+, w)$;
        $p[a^+] := a$;
        $w := a^-$;
        $a := RightProp[w']'$
      od
      $CO\_EXPAND(petal[y]^+, w)$;
    else
      $b := exit[petal[y]']$;
      $EXPAND(x, b)$;
      $w := b$;
      $a := FirstProp[petal[y]]$;
      while $a \neq NO\_ARC$ do
        $EXPAND(w, a^-)$;
            $p[a^+] := a$;
            $w := a^+$;
            $a := RightProp[w]$
          od;
          $EXPAND(w, petal[y]^-)$;
          $p[petal[y]^+] := petal[y]$;
          $w := y$;
          if $y = b'$ then $a := FirstProp[petal[y]']'$
            else $a := LeftProp[y']'$ fi;
          while $a \neq NO\_ARC$ do
            $CO\_EXPAND(a^+, w)$;
            $p[a^+] := a$;
            $w := a^-$;
            $a := LeftProp[w']'$
          od;
          $CO\_EXPAND(petal[y]^+, w)$
        fi
    fi
fi
end;

procedure $CO\_EXPAND(x, y)$;
var $a, b, w$;
begin
if $x \neq y$ then
  if $d[x'] < d[x]$ then
    $a := Props[x'].FIRST'$;
    $p[a^+] := a$;
    $CO\_EXPAND(a^+, y)$
  else
      if $exit[petal[x']] \neq NONE$ then
        $b := exit[petal[x']]$;
        $w := x$;
        if $x = b$ then $a := FirstProp[petal[x']']$ else
          $a := RightProp[x]$ fi;
        while $a \neq NO\_ARC$ do
            $EXPAND(w, a^-)$;
            $p[a^+] := a$;
            $w := a^+$;
            $a := RightProp[w]$
        od;
        $EXPAND(w, (petal[x']')^-)$;
      $p[(petal[x']')^+] := petal[x']'$;
      $w := b'$;
      $a := FirstProp[petal[x']]'$;
      while $a \neq NO\_ARC$ do
        $CO\_EXPAND(a^+, w)$;
        $p[a^+] := a$;
        $w := a^-$;
        $a := LeftProp[w']'$
      od;
      $CO\_EXPAND((petal[x']')^+, w)$
      $CO\_EXPAND(b', y)$
    else
      $b := exit[petal[x']']$;
      $w := x$;
      if $x = b$ then $a := FirstProp[petal[x']']$ else
        $a := LeftProp[x]$ fi;
      while $a \neq NO\_ARC$ do
        $EXPAND(w, a^-)$;
        $p[a^+] := a$;
        $w := a^+$;
        $a := LeftProp[w]$
      od;
      $EXPAND(w, (petal[x']')^-)$;
      $p[(petal[x']')^+] := petal[x']'$;
      $w := b'$;
      $a := FirstProp[petal[x']]'$;
      while $a \neq NO\_ARC$ do
        $CO\_EXPAND(a^+, w)$;
        $p[a^+] := a$;
        $w := a^-$;
        $a := RightProp[w']'$
      od;
      $CO\_EXPAND((petal[x']')^+, w)$
      $CO\_EXPAND(b', y)$
    fi
  fi
fi
end
end.
```

There is a rule *EXPAND* which computes certain minimum valid paths p. As in the simple path expansion procedure in [5], we give an explicit rule *CO_EXPAND* for computing the complementary path p'. The symmetry of these procedures is obvious.

In addition to that, we must distinguish whether a max-level node has been explored by a left DFS operation or by a right DFS operation. Both cases are treated in a similar way. Hence, checking the correctness of the given path expansion rule is considerably simpler than its lengthiness suggests.

Theorem 21.1. *Let y be a strictly (i)-reachable node, x be an (i)-base which is traversed by every $d(y)$-path q, and $p \equiv NO\_ARC$. Assume that we just have called BNS. Then, $EXPAND(x, y)$ determines a valid xy-path of length $|q[x, y]|$ which is encoded into the p-labels, and $CO\_EXPAND(y', x')$ computes the complementary $y'x'$-path.*

Proof. The assertion follows by induction on i and is obvious for $i = 0$ since we have $s = x = y$ in that case. Let the statement be true for every $i < j$ and assume that $i = j, x \neq y$.

First, assume that y is a minlevel node. By the correctness analysis of the procedure *BNS,* we see that $a := Props[y].FIRST$ is a prop of y and that $z := a^-$ is strictly $(i - 1)$-reachable. Hence, x is on every $d(z)$-path, and $q[z]$ is a $d(z)$-path by Lemma 8.2. By the induction hypothesis, $EXPAND(x, z)$ is a valid xz-path of length $|q[x, z]|$. The induction step is obvious.

From now on, let y be a maxlevel node and $a := petal[y]$. Then, y has been explored by the DDFS as a consequence of Theorem 20.1. We consider the case that y was explored by a left DFS operation, that is, $b := DDFS(a)$ has been called during the BNS. Let A be the set of arcs which have been investigated by the BNS before a and a', and let $\tilde{A} := A \cup \{a, a'\}$.

Note that b is strictly $(i - 1)$-reachable and that $B_{\tilde{A}}(y)$ is, in general, not an (i)-blossom. By the second shrinking property 10.6, there is a $d(y)$-path which traverses b and a. Let us assume that q is such a path. Then, $q[b]$ is a $d(b)$-path, and x is on every $d(b)$-path by the base identity 9.2 and Theorem 12.7.

Hence, $EXPAND(x, b)$ determines a valid xb-path of length $q[x, b]$ by the induction hypothesis. Note that a proper (i)-blossom which is nested into $B_{\tilde{A}}(y)$ must be a leaf of at least one of the DFS trees. As a consequence of the second shrinking property and the induction hypothesis, $EXPAND(b, y)$ is a $d(b, y)$-path. The induction step is obvious now.

If y was explored by a right DFS operation, $DDFS(a')$ was called and $exit[a] = NONE$ was set. This case, as well as the induction step for the procedure CO_EXPAND are analogous to the cases which have been discussed before. ■

Corollary 21.2. *If v is strictly reachable, then $EXPAND(s, v)$ determines a $d(v)$-path in $O(n)$ time.* ■

Consider the call of $EXPAND(s, t)$ in case of the running example. In order of occurrence, the assignments to the p-labels and the recursive calls of *EXPAND* and *CO_EXPAND* are the following:

$EXPAND(s, s)$	
$EXPAND(s, s)$	$p[10] := (s, 10)$
$EXPAND(10, 10)$	$p[10] := (10, 5')$
$EXPAND(5', 5')$	$p[10] := (5', 6)$
$EXPAND(6, 6)$	$p[3] := (6', 3)$
$CO\_EXPAND(t, t)$	$p[t] := (10', t)$
$CO\_EXPAND(10', 10')$	$p[10'] := (8, 10')$
$CO\_EXPAND(7', 8)$	$p[t] := (3, 7')$
$CO\_EXPAND(3, 3)$	

There are merely two nontrivial recursive operations, namely $EXPAND(6, 6')$ and $CO\_EXPAND(7', 8)$. The former call will result in

$EXPAND(6, 6)$	
$EXPAND(6, 6)$	$p[2'] := (6, 2')$
$EXPAND(2', 2')$	$p[1] := (2', 1)$
$CO\_EXPAND(6', 6')$	$p[6'] := (1, 6')$
$CO\_EXPAND(1, 1)$	

The reader is asked to supply the details for the call of $CO\_EXPAND(7', 8)$. The resulting $d(t)$-path is

$$(s, 10, 5', 6, 2', 1, 6', 3, 7', 8, 10', t).$$

22. PHASE-ORDERED ALGORITHMS

Using the given BNS algorithm of the last section together with the simple augmentation procedure *MAX_BAL_FLOW* given in [5], one can determine maximum balanced flows in time $O(nm^2\alpha(m, n))$ by Theorem 6.4. (The general idea is that the length of the augmenting paths is monotonically increasing.)

However, this algorithm can still be improved since it suffices to run *BNS* once during a **phase.** Here, a **phase** is a period of the augmentation algorithm during which all augmenting paths have equal length.

The necessary modifications are discussed throughout this section. We will not present a complete pseudocode which can be found in the first authors doctoral thesis [3]. The improved algorithm specializes to the state-of-the-art cardinality matching algorithm given by Micali and Vazirani in [8].

We can use the original BNS up to that moment when $DDFS(a) = s$ is reached first and finish the (i)-phase by completing $Max(i)$. Of course, we need some modifications of the former BNS procedure, especially of the procedure *Max*. In comparison with the former BNS algorithm, we have two new procedures *Augment* and *TopologicalErasure*.

Procedure 4. An Improved Augmentation Rule

```
procedure Augment;
var $a$, $\epsilon$, $v$;
begin
EXPAND($s$, $t$);
$\epsilon$ := BAL_PATH_CAP;
BAL_AUGMENT($s$,$t$,$\epsilon$);
$a$ := LeftProp[$s$];
while $a \neq$ NO_ARC do
  if rescap($a$) = 0 and InDegree[Exit[$a$]] > 0
  then TopologicalErasure($a$)
  fi;
  $a$ := LeftProp[$a^+$]
od;
$a$ := RightProp[$s$];
while $a \neq$ NO_ARC do
  if rescap($a$) = 0 and InDegree[Exit[$a$]] > 0
  and LeftProp[Exit[$a$]] $\neq a$
  then TopologicalErasure($a$)
  fi;
  $a$ := RightProp[$a^+$]
od;
while not LeftSupport.EMPTY do
  LeftSupport.DELETE($v$);
  LeftProp[$v$] := NO_ARC
od;
while not RightSupport..EMPTY do
  RightSupport.DELETE($v$);
  RightProp[$v$] := NO_ARC
od;
end;
```

Assume that *Max*(i) just has investigated the bridge a, that A is the set of arcs which have been investigated before a, and that f is the flow reached so far. If we reach $b = DDFS(a) = s$ the first time, we can compute an augmenting path by the call of *EXPAND*(s, t) and augment f then. The obvious problem is to update the layered auxiliary network so that further DDFS operations apply. This might be done by another call of the balanced network search, but can be accomplished more efficiently by the process of **topological erasure.**

We call an (A)-blossom B **blocked** iff none of the $d(base(B))$-paths of the begin of the (i)-phase is valid at some later point of the (i)-phase. If $\tilde{a}$ is a prop and $B(\tilde{a}^-)$ is blocked, then we say that $\tilde{a}$ is **blocked.** We also call $\tilde{a}$ blocked if we have $rescap(\tilde{a}) = 0$. By the topological erasure, every blocked blossom base x is labelled $InDegree[x] := 0$.

Let p and q be the paths which connect s with $BASE(a^-)$ and $BASE(a^+)$, respectively, in the layered auxiliary network and which have been expanded to the augmenting path. These paths are encoded into *LeftProp* and *RightProp,* respectively. The arcs of p and q, which have no longer residual capacity, are deleted from the layered auxiliary network one by one.

The deletion is done implicitly by using some additional data structures. If x is an (A)-base, $a \in Props[x]$, and $rescap(a) = 0$ is reached by some augmentation step, then a is not really deleted from $Props[x]$. Instead of this, we update the label $InDegree[x]$.

When $InDegree[x] = 0$ is reached, then x is blocked and no further DDFS operation should search x. We avoid this by performing the topological erasure of x. This requires a reverse adjacency list $Successors[x]$ which consists of all (A)-blossom bases with predecessor x in the layered auxiliary network before the last augmentation step. The queue $Successors[x]$ is flushed, and $InDegree[y]$ is decreased by one for every y on $Successors[x]$. If we have $InDegree[y] = 0$, then y is blocked and treated in the same way as x.

Procedure 5. The Topological Erasure

```
procedure TopologicalErasure($a$);
var $x$, $y$;
object Blocked: QUEUE;
begin
  $x := a^+$;
  InDegree[$x$] := InDegree[$x$] − 1;
  if InDegree[$x$] = 0 then
    Blocked.INSERT($x$);
    while not Blocked.EMPTY do
      Blocked.DELETE($x$);
      while not Successors[$x$].EMPTY do
        Successors[$x$].DELETE($y$);
        InDegree[$y$] := InDegree[$y$] − 1;
        if InDegree[$y$] = 0 then Blocked.INSERT($y$)
          fi
      od
    od
fi
end;
```

Note that *Max*(i) should initiate a DDFS only if the blossom bases involved are not blocked yet. Furthermore, *Min* should compute the correct *InDegree* labels. The *DDFS* procedure must be modified so that no blocked blossom bases or props are searched.

Lemma 22.1. *The topological erasure process requires $O(m)$ time altogether during a phase.*

Proof. A blossom base b is a minlevel node. Note that *InDegree* increases only when a prop is appended to $Props[b]$ during the call of $Min(d(b) - 1)$. Note also that *InDegree* decreases only during the last call of *Max*. Thus, b can be placed on *Blocked* at most once, and *Successors*$[b]$ is flushed then. ■

We are now in the situation where we can exclude most parts of the algorithm from the remainder of the complexity analysis. The blossom shrinking process and also the topologic erasure process need (almost) $O(m)$ time per phase. If we neglect the arc investigation steps which initiate a flow augmentation, the BNS part runs also in $O(m)$ time. These bounds are optimal even in the case of 0–1 balanced flow networks.

Next, consider the augmentation part of a phase. Obviously, the expansion of an augmenting path and an augmentation step need $O(n)$ time. Hence, the total time spent for the procedure *Augment* is $O(nm)$ in the general case. In the case of 0–1 networks, we know by Lemma 6.5 that the augmenting paths of a single phase are disjoint; hence, *Augment* needs $O(m)$ time during a whole phase.

There are, however, some operations which we have not counted yet. These are the calls of the DDFS which yield an augmenting path. Since a single call of *DDFS* runs in $O(m)$, we have the obvious time bound $O(m^2)$ for a whole phase of the algorithm. But then the new algorithm would be only a minor improvement.

Despite all the parallels to the Dinic algorithm discussed so far, this bound might be tight. Note that the Dinic algorithm for the maximum flow problem merely traverses one single directed path of the layered auxiliary network during a certain augmentation step. A call of the DDFS, however, searches a considerable part of the layered auxiliary network.

Theorem 22.2. *Maximum balanced flows can be computed in $O(nm^2)$ time.* ■

Especially in the case of 0–1 balanced flow networks, where a version of the Micali/Vazirani algorithm results, the bound of $O(m^2)$ is considerably worse than is the bound of $O(m)$ claimed by these authors. Note that this bound has not been discussed in [8–10] explicitly. The gap is closed as follows:

Lemma 22.3. *Consider the* 0–1 *balanced flow network N_G associated with the graph G and a DDFS operation which yields $b = s$. Then, every prop searched by the DDFS is blocked after that augmentation step.*

Proof. Let a be a prop searched by the DDFS and $v := Exit[a]$. The assertion follows by induction on $d(v)$. In the case of $d(v) = 0$, we have $a = LeftProp[s]$ or $a = RightProp[s]$ which have no residual capacity after the augmentation step and, hence, are blocked. Thus, we can assume $d(v) = i + 1$ and that the assertion is proven for every prop $\tilde{a}$ with $d(Exit[\tilde{a}]) \leq i$.

We first assume that a has been included into one of the DFS trees, say the left one. If the left DFS has backtracked on a, all props of v have been searched before and are erased by the induction hypothesis. But then a is also erased. If the left DFS does not backtrack on a, then a is on the left active path at the end of the DDFS and also on the augmenting path generated by *EXPAND*. Since we have $rescap(a) = 0$ after the augmentation step, a is blocked.

Otherwise, a has not been included into one of the DFS trees, but searched by one of the DFS trees, say the left one. Then, v has been visited by the left DFS before and has been included into the left DFS tree. If v would be an inner node, then $\{v, v'\}$ would be a bud by Corollary 9.15 and a^+ would be the unique successor of $v = a^-$. But, then, v could have been visited only by searching a, a contradiction. Hence, v is an outer node and the unique prop of v has been searched by one of the DFS and is blocked by the induction hypothesis. Then, a is blocked likewise. ■

Theorem 22.4. *Let N_G be the* 0–1 *balanced flow network associated with the graph G. Then, a maximum balanced flow on N_G can be found in $O(\sqrt{n}m\alpha(m, \mathrm{n}))$ time.*

Proof. By Theorem 6.6, we have $O(\sqrt{n})$ phases in the case of 0–1 networks. The procedure *Min* requires $O(m)$ time during a whole phase. The time required by *DDFS* operations during a phase is $O(m\alpha(m, n))$ by Lemma 22.3 and the results of [10] for the disjoint set union process. Furthermore, the augmenting paths of a particular phase are pairwise disjoint by Lemma 6.5. Hence, the time required for *Augment* operations is $O(m)$. ■

We mention that there are other polynomial time algorithms for the maximum balanced flow problem: Apparently, an $O(m^2 \log U)$ algorithm is available which uses the idea of capacity scaling (where U denotes the maximum arc capacity). Such an algorithm could include one of the augmentation procedures given in [5]. However, the algorithm can be formulated in such a way that only the last scaling phase needs a BNS procedure. We expect that the bound can be improved to $O(nm \log U)$.

Anstee [1] proposed an algorithm which computes an ordinary maximum flow for the balanced network first. By a rather simple procedure, the maximum flow is transformed into a balanced flow which is almost maximum balanced. One can achieve a maximum balanced flow then in $O(nm)$ time by using any linear balanced augmentation procedure.

It turns out that the max flow computation is the dominating part in the complexity analysis and can be implemented in $O(n^3)$ time, $O(nm \log n)$ time, or $O(n^2\sqrt{m})$ time. The Anstee algorithm is state of the art.

A paper which contains an analysis of each of these algorithms is in preparation [6]. The concepts seem to be similar but more simple than the algorithm presented here.

On the other hand, augmenting along a shortest path means changing a precomputed matching as little as possible. Hence, there are explicit applications of the BNS algorithm discussed here.

REFERENCES

[1] R. P. Anstee, An algorithmic proof of Tutte's f-factor theorem, J Algor 6 (1985), 112–131.

[2] J. Edmonds, Paths, trees and flowers, Can J Math 17 (1965), 449–467.

[3] C. Fremuth-Paeger, *Degree Constrained Subgraph Problems and Network Flow Optimization, Wissner Verlag, Augsburg, 1997*.

[4] C. Fremuth-Paeger and D. Jungnickel, Balanced network flows (I): A unifying framework for design and analysis of matching algorithms, Networks, to appear.

[5] C. Fremuth-Paeger and D. Jungnickel, Balanced network flows (II): Simple augmentation algorithms, Networks, to appear.

[6] C. Fremuth-Paeger and D. Jungnickel, Balanced network flows (IV): Duality and structure theory, Networks, submitted.

[7] S. Micali and V. V. Vazirani, An $O(\sqrt{V}E)$ algorithm for finding maximum matching in general graphs, Proceedings of the 21st Annual IEEE Symposium in Foundation of Computer Science, 1980, pp. 17–27.

[8] P. A. Peterson and M. C. Loui, The general maximum matching algorithm of Micali and Vazirani, Algorithmica 3 (1988), 511–533.

[9] R. E. Tarjan, Data Structure and Network Algorithms, SIAM, Philadelphia, PA 1983.

[10] V. V. Vazirani, A theory of alternating paths and blossoms for proving correctness of the $O(\sqrt{V}E)$ general graph maximum matching algorithm, Combinatorica 14 (1994), 71–109.