Balanced Network Flows. II. Simple 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: In previous papers, we discussed the fundamental theory of matching problems and algorithms in terms of a network flow model. In this paper, we present explicit augmentation procedures which apply to the wide range of capacitated matching problems and which are highly efficient for k-factor problems and the f-factor problem. © 1999 John Wiley & Sons, Inc. Networks 33: 29–41, 1999

Keywords: capacitated matching problems; network flows; balanced flow networks; skew-symmetric graphs; antisymmetrical digraphs; augmenting a matching; matching heuristics

13. INTRODUCTION OF THE PSEUDOCODE FORMALISM

Balanced networks can be considered as a network flow description of matching problems. Such networks are defined on skew-symmetric graphs where arc labelings satisfy a certain symmetry constraint. A comprehensive introduction into the terminology and theory of balanced network flows was given in [5], where the reader can find problem reduction mechanisms and explicit examples.

Instead of repeating this general setup, we present some object-oriented pseudocode for the methods associated with balanced networks, especially for solving the maximum balanced flow problem. Our pseudocode is object-oriented for the following reasons:

We want to point out which techniques are common to the known matching algorithms and which are different. At least, the search strategy and the disjoint set union mechanism used belong to the latter class. Hence, these components should be encapsulated for better replacements. Many applications of network flow algorithms are reductions of other optimization problems to network flows. A certain part of an algorithm may be best performing in the general problem setting, but should be replaced in special cases. Hence, an inheritance mechanism is useful.

As an example for the formalism, consider the following class declarations: The class *OBJECT* represents the universe of all available data objects. The class *SET* represents set objects. The expression *SET*(*OBJECT*) indicates the **subclass relationship** between *SET* and *OBJECT*. In a way, the declarations of Procedure 1 are dummies since there are no implementations given.

A set S can be allocated in computer storage and initialized by the **constructor** *S.MAKE* and can be disallocated by the **destructor** *S.FREE*. Note that these procedures are not associated with class *SET* but with class *OBJECT* in the declaration above, since any dynamic data object

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 was written under the supervision of the second author.

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

has a lifetime. One says that *SET* **inherits** the features of the class *OBJECT*.

There is an additional constructor, *S.INIT,* which checks whether S already is allocated or not. In the former case, the memory is reused but all information is deleted. In either case, an empty set results.

The prefix S. denotes that a method is applied to object S. If the mathematical context is clear, or in method implementations of the same class, the object prefix is omitted. If a class property is declared **public,** it can be accessed from outside the class declaration.

The call of *EMPTY* decides whether S is an empty set or not. To access the members of S, one can prepare a stream of set members by the call of *ACCESS* which references to some ''first'' member then. In this stream, every member has to occur exactly once. The ''next'' member w can be read by *READ*(w). If all members have been read, *EOS* should return *TRUE* and *FALSE* otherwise. While a stream is active, no element can be added to or removed from the set. Thus, *CLOSE* is needed to terminate the set member stream.

To a set S, an element w may be added by the call of *INSERT*(w) and an element v may be removed by the call of *DELETE*(v). The elements removed from a set are chosen accordingly to a predefined strategy, and there shall be no other way to remove an element. This strategy is **last-in first-out** in the case of stack objects and **first-in first-out** in the case of queue objects. If S is a stack, the **uppermost element,** the element which entered S last, can be accessed as *TOP*:

Procedure 1. Set Objects

```
class OBJECT;
public
  constructor MAKE;
  constructor INIT;
  destructor FREE
end;

class SET(OBJECT);
public
  procedure ACCESS;
  procedure READ(var w);
  function EOS;
  procedure CLOSE;

  function EMPTY;
  procedure INSERT(w);
  procedure DELETE(var w)
end;

class STACK(SET);
public
  function TOP
end;

class QUEUE(SET);
public
  function FIRST;
  function LAST
end.
```

Before we start the description of the algorithms, we briefly describe our implementation of network flow objects: The nodes and arcs of a *NETWORK* object N are the integers $0, 1, \ldots, n-1$ and $0, 1, \ldots, 2m-1$, respectively. With any arc a, the reverse arc is available by $\bar{a}$. Furthermore, *NONE* and *NO_ARC* denote an undefined node and arc, respectively.

Both nodes incident with an arc a can be accessed as a^- for the start node and a^+ for the end node, respectively. In contrast, the arcs incident with a certain node cannot be accessed directly, but by a network search process: A network search is initialized by a call of *INVESTIGATE* which assigns a stream of incident arcs to every node v. This stream contains all arcs with $a^- = v$ and $rescap(a) > 0$ which are accessed by the iterated call of *READ*(v, a). If all arcs have been read, *EOS*(v) becomes *TRUE*. The network search is finished by *CLOSE*.

Procedure 2. Network Objects

```
class NETWORK(OBJECT);
private
  constant NONE, NO_ARC;
  constant $m$, $n$;
  function $a \to a^-$;
  function $a \to a^+$;
  function $a \to \bar{a} : \bar{a} := 2m - 1 - a$;
  function $rescap(a)$;

  procedure INVESTIGATE;
  procedure READ($v$, var $a$);
  function EOS($v$);
  procedure CLOSE;
end.

class FLOW_NETWORK(NETWORK);
private
  function $cap(a)$;
  function $f(a)$;

  function $rescap(a)$
  begin
  if $a < n$
  then return $cap(a) - f(a)$
  else return $f(\bar{a})$
  fi
  end;

  procedure PUSH($a$, $\gamma$)
end.
```

A whole network search should be implemented so that it runs in $O(m)$ time. To make our later examples deterministic, we will assume that the streams of incidences are implemented in such a way that the numbers of the end nodes are ascending.

A *FLOW_NETWORK* object encapsulates capacity information, but also information about a certain flow which can be accessed by $cap(a)$ and $f(a)$. Here, $cap(a) > 0$ and $0 \leq f(a) \leq cap(a)$ are required. These values are associated with the forward arcs only which are labeled $0, 1, \ldots, m - 1$. Since we are concerned with matching problems, we assume that neither parallel nor antiparallel forward arcs exist. Under this assumption, the residual capacities can be implemented as in Procedure 2.

A single-flow value can be modified by $PUSH(a, \epsilon)$ decreasing $rescap(a)$ by an amount ϵ, where $rescap(a) > 0$ and $0 \leq \epsilon \leq rescap(a)$ holds. This operation will also affect $f(a)$. While a network search is active, no *PUSH* operation is allowed.

Concerning the time complexity, we assume that all but the network search operations are elementary. Note that memory allocation is treated as an elementary operation although it has considerable effect on the performance of algorithms in practice.

14. BALANCED NETWORK OBJECTS

At this point, we can give a rough sketch of the behavior of balanced network objects. Of course, we will add some implementations to Procedure 3 as our discussion progresses.

The crux is the symmetry of nodes defined by $v' := n - 1 - v$, which implies that $(v')' = v$. By this symmetry, the node set splits into pairs of **complementary** nodes. If a is an arc with start node u and end node v, then a' is required to have start node v' and end node u'. Similarly, a and a' are called **complementary** arcs. We also require that capacities and flow values are **balanced,** that is, $cap(a) = cap(a')$ and $f(a) = f(a')$.

The method *MAX_BAL_FLOW* of Procedure 4 is a generic augmentation algorithm for the problem of finding a maximum balanced flow for a given balanced flow network. It is comparable to the algorithm of Ford and Fulkerson [4] for the ordinary max flow problem. The remainder of the Procedures 3 and 4 are components of this algorithm:

The declaration of balanced network objects heavily depends on the **balanced network search** (BNS) algorithms which will be discussed in the bulk of this paper. The call of $BNS(s, t)$ searches for valid paths connecting the source s to the other nodes of the balanced network. If the node t is reached, $BNS(s, t)$ returns *TRUE* and *FALSE* otherwise. Efficient implementations will follow.

The call of $EXPAND(s, v)$ uses the information collected by *BNS* before and extracts a path $p = (v_0 = s, v_1, \ldots, v_{k-1}, v_k = v)$ of length $k = d[v]$. This path is encoded into the labels $p[v_1], \ldots, p[v_k]$, and $p[v_i]$ denotes the arc on p with end node v_i.

Procedure 3. Balanced Network Objects

```
class $BALANCED\_NW(NETWORK)$;
private
  array $d, p$;

  function $v \mapsto v' : v' := n - 1 - v$;
  function $a \mapsto a' : a' := \left\{\begin{matrix} a+1, & \text{if } a \text{ is even} \\ a-1, & \text{if } a \text{ is odd}\end{matrix}\right\}$;

  function $BNS(s, t)$;
  function $EXPAND(u, v)$;

  function $BAL\_PATH\_CAP(u, v)$;
  var $a, \epsilon, w$;
  begin
  $\epsilon := \infty$;
  $w := v$;
  repeat
    $a := p[w]$;
    $w := a^-$;
    if $p[w'] = a'$
    then $\epsilon := \min\{\epsilon, \lfloor rescap(a)/2 \rfloor\}$
    else $\epsilon := min\{\epsilon, rescap(a)\}$
    fi
  until $w = u$
  return $\epsilon$
  end
end.
```

Recall that the path is called **valid** iff $\epsilon := BAL\_PATH\_CAP(s, v) > 0$ holds. Then, the call of $BAL\_AUGMENT(u, v, \epsilon)$ augments the flow along the encoded path, but also along its complementary path. We will give linear time implementations of *EXPAND, BAL_PATH_CAP,* and *BAL_AUGMENT.*

The "correctness" of the method *MAX_BAL_FLOW* is an immediate consequence of the augmenting path Theorem 4.2 given in [5] and a rather intuitive result. The analysis of the BNS procedures will frequently use the terminology and the results of [5]. For understanding the algorithms, it is probably necessary to study this preceding paper first. If we refer to some mathematical statement, it can be found there.

In [5], we considered networks N_G, $N_{\mathcal{M}}$, and $N^*_{\mathcal{M}}$ resulting from the network flow reduction of certain matching problems. If we want to compute maximum balanced flows, it is inefficient to generate these networks explicitly. We would rather declare a subclass of *BAL_FLOW_NW* which has access to the data of the original graph. By using inheritance, we can reimplement parts of the algorithms which get simpler in the special situation (0–1 capacities for example).

Procedure 4. Balanced Flow Network Objects

```
class $BAL\_FLOW\_NW(BALANCED\_NW, FLOW\_NETWORK)$;
private
  procedure $BAL\_AUGMENT(u, v, \epsilon)$;
  var $a, w$;
  begin
  $w := v$;
  repeat
    $a := p[w]$;
    $PUSH(a, \epsilon)$;
    $PUSH(a', \epsilon)$;
    $p[w] := NO\_ARC$;
    $w := a^-$
  until $w = u$
  end;

  procedure $MAX\_BAL\_FLOW(s, t)$;
  var $\epsilon$;
  begin
  while $BNS(s, t)$ do
    $EXPAND(s, t)$;
    $\epsilon := BAL\_PATH\_CAP(s, t)$;
    $BAL\_AUGMENT(s, t, \epsilon)$
  od
  end
end.
```

15. PATH EXPANSION

If we want to analyze a given balanced network search procedure, we have to describe the blossom structure for certain subnetworks $N[A]$. Here, N is some balanced network, and $A \subseteq A(N)$ is the set of arcs which have been investigated by the BNS algorithm so far.

Procedure 5. A Path Expansion Procedure

```
class $BALANCED\_NW$;
private
  array $prop, petal$;

  procedure $EXPAND(x, y)$;
  var $u, v$;
  begin
  if $x \neq y$ then
    if $prop[y] \neq NO\_ARC$ then
      $p[y] := prop[y]$;
      $EXPAND(x, prop[y]^-)$
    else
      $u := petal[y]^-$;
      $v := petal[y]^+$;
      $p[v] := petal[y]$;
      $EXPAND(x, u)$;
      $CO\_EXPAND(v, y)$
    fi
  fi
  end;

  procedure $CO\_EXPAND(v, y)$;
  var $u, w$;
  begin
  if $y \neq v$ then
    if $prop[v'] \neq NO\_ARC$ then
      $p[(prop[v']')^+] := prop[v']'$;
      $CO\_EXPAND((prop[v']')^+, y)$
    else
      $u := (petal[v']')^-$;
      $w := (petal[v']')^+$;
      $p[w] := petal[v']'$;
      $EXPAND(v, u)$;
      $CO\_EXPAND(w, y)$
    fi
  fi
  end
end.
```

In [5], we presented a series of statements which relate the blossoms of $N[A]$ to the blossoms of the iterated network $N[\tilde{A}]$, where $\tilde{A}$ results from A by adding a suitable pair of complementary arcs. We have used **layered auxiliary networks** $Aux(N)$, an approach comparable to shrinking the blossoms of the original network N.

The BNS algorithms that we discuss in this paper are **tree growing,** that is, the occurring layered auxiliary networks are trees. For the time being, let BNS be arbitrary, but tree growing. The following path expansion rule applies in that general setting and, in particular, to the BNS procedures which follow.

In [5], we assigned to every strictly reachable node v either a **prop** or a **petal.** By that, we denote the arc which was investigated when v has become strictly reachable. We showed that an augmenting path can be found by a call of $path(s, t)$ which is defined by $path(x, x) := (x)$:

$$path(x, y) := path(x, z') \circ prop[y]$$

if $prop[y] = z'y$ is assigned, and

$$path(x, y) := path(x, u) \circ petal[y] \circ [path(y', v)]'$$

if $petal[y] = uv'$ is assigned.

This was convenient for the discussion of examples and also in the mathematical context. A concrete matching algorithm would rather call the method *EXPAND* which is shown in Procedure 5. It returns the valid path in terms of the labels $p[v_1], \ldots, p[v_k]$ and is compatible with the methods *BAL_AUGMENT* and *FIND_BAL_CAP* of the Procedures 3 and 4.

We did not devise a method which computes the complementary path of a given valid path. Instead, we use the method *CO_EXPAND*(v, y) in recursion which computes $(path(y', v'))'$ more directly. Both procedures *path* and *EXPAND* are equivalent—hence, the correctness of the new procedure follows by Theorem 11.4. The details are left to the reader.

16. DISJOINT SET UNION STRUCTURES

Every balanced network search algorithm based on the results of [5] will split into three parts: Growing the subnetwork $N[A]$, computing all (A)-blossoms and the base of the new blossom, if the investigated arc has the shrinking properties, and, finally, merging the (A)-blossoms which have been computed.

We required that scanning the network N needs $O(m)$ operations. In the following sections, we will show how to implement the second part also in $O(m)$ time. It is somewhat surprising that the merging of blossoms is the critical part in the complexity analysis. Although we do not want to spend too much effort on the description of data structures, we are forced to discuss the crux of set union algorithms briefly.

Let *Blossoms* $:= (B_1, B_2, \ldots, B_k)$ a family of nonempty pairwise disjoint subsets of the node set $V(N)$. We allow three kinds of manipulation of this set family:

- *BUD:* Appending to *Blossoms* a nonempty set, called *bud*, $B \subseteq V(N)$ disjoint to every current set in *Blossoms*.
- *UNION:* Merging one set B_i into another set B_j of the family *Blossoms,* while deleting the set B_i from *Blossoms*.
- *FIND:* Verifying whether a given element $v \in V$ is a member of some set $B_i \in$ *Blossoms* and, if so, returning the name B_i.

The process of successive application of these operations is called **disjoint set union** (DSU). We apply *UNION* whenever (A)-blossoms are shrunk and *BUD* whenever a new bud is generated. We may assume that there are $O(m)$ *FIND* operations totally. An efficient implementation is given by Procedure 6 and works as follows:

To every blossom, a **canonical element** v is assigned. This node is determined by the identity $B[v] = v$ and may differ from the blossom base. All blossom nodes form a tree which is determined by the B-labels, which is rooted at v, and to which we refer as the **DSU tree.** An example is given in Figure 1.

A blossom and its canonical element are identified then. A call of *FIND*(w) computes the canonical element v of the blossom containing w and then puts $B[w] := v$. The latter operation is called **path compression.**

If we ignore the effects of path compression, then $rank[v]$ denotes the nesting depth of the DSU tree rooted at v. Since every node v with $rank[v] \geq 2$ has at least two children, the rank is $O(\log n)$. Together with the effects of path compression, the total effort for *FIND* operations is $O(m + n \log n)$.

This time bound is not tight. By a more careful analysis, Tarjan [12] established the bound $O(m\alpha(m, n))$, where $\alpha(m, n)$ is some kind of inversion of the **Ackerman function.** This analysis and the formal definition of $\alpha(m, n)$ can also be found in [3] and [13].

In our later BNS algorithms, we use the *DISJOINT_SET_FAMILY* class methods, additional labels $base[v]$, and some method *BASE* to obtain the actual base of blossoms. In Procedure 6, we included a method *NUCLEI* which can be used to compute the nuclei of a balanced network. Since the nuclei are computed from the blossoms, one must run a BNS first.

Procedure 6. Data Structures for the Management of Shrinking Families

```
class DISJOINT_SET_FAMILY;
private
  array B;
  array rank;
  constant n;
  constant NONE;

public
  procedure INIT;
  var v;
  begin
  for v := 0 to n − 1 do B[v] := NONE od
  end;

  procedure MAKE(r);
  begin
  n := r;
  allocate B[0, 1, . . . , n − 1];
  allocate rank[0, 1, . . . , n − 1];
  INIT
  end;

  procedure BUD(v);
  begin
  B[v], B[v'] := v;
  rank[v] := 1
  end;
  procedure UNION(u, v);
  begin
  u := FIND(u);
  v := FIND(v);
  if rank[u] < rank[v]
  then
    B[u] := v
  else
    B[v] := u;
    if rank[v] = rank[u] and $u \neq v$
    then rank[u] := rank[u] + 1
    fi
  fi
  end;

  function FIND(v);
  begin
  if B[v] $\neq$ v and B[v] $\neq$ NONE
  then B[v] := FIND(B[v])
  fi;
  return B[v]
  end
end;

class BALANCED_NW;
private
  object F: DISJOINT_SET_FAMILY;
  array base;

public
  function BASE(v);
  begin
  if F.FIND(v) = NONE
  then return NONE
  else return base[F.FIND(v)]
  fi
  end;

  procedure NUCLEI;
  var a, u, v;
  begin
  for a := 0 to m - 1 do
    u := BASE($a^-$);
    v := BASE($a^+$);
    if $u \neq v$ and $d[u] < d[v] < d[v'] < \infty$ and $d[u']$
      $< \infty$ then
      F.UNION(u, v);
      base[F.FIND(u)] := u
    fi
  od
  end
end.
```

In some sense, the base of a blossom is a canonical element. However, if we merge a series of (A)-blossoms into an $(\tilde{A})$-blossom $\tilde{B}$ successively by their rank, we cannot guarantee the canonical element of $\tilde{B}$ to be its base b. In our example, the blossom bases and the canonical elements coincide. The reader is asked for some counterexample. A simple DSU mechanism which treats blossom bases as canonical elements would run in $O(n^2)$ time (see Fig. 1).

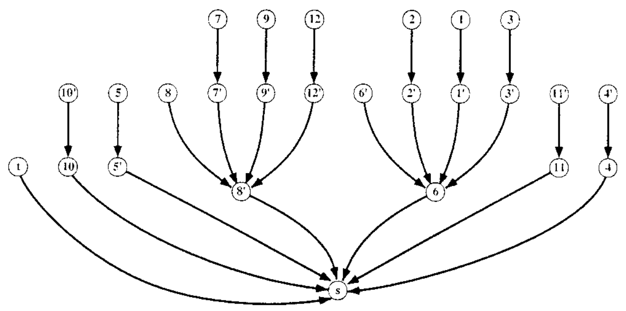

Fig. 1. A DSU tree.

A linear time bound for the *FIND* operations can be achieved by using the **incremental tree set union** data structure which was described by Gabow and Tarjan [7]. Despite the slight theoretical improvement, we expect no better performance in practice.

We mention that the presented type of DSU data structure does not apply to weighted matching algorithms, where blossoms must be expanded during the BNS process. This can be accomplished by an additional *SPLIT* operation and merging by rank again. Path compression is then not available so that a *FIND* operation takes $O(\log n)$ time.

17. A BREADTH FIRST BNS ALGORITHM

In this paper, we present two special implementations of the balanced network search. The first of these algorithms is based on the paper by Kocay and Stone [9]. We have managed to speed up the procedure to almost linear time complexity. However, our procedure has become more involved than the original one by Kocay and Stone.

Procedure 7. A Refined Version of the Kocay/Stone Algorithm

```
class BALANCED_NW;
private
  function BNS(s, t);
    object Q: QUEUE;
    object Support: SET;

    procedure ShrinkBlossom;
    var Tenacity, x, y, z;
    begin
    x := BASE(u);
    y := BASE(v');
```

```
        Tenacity := d[u] + d[v'] + 1;
(1)     while $x \neq y$ do
            if $d[x] > d[y]$ then
                if $d[x'] = \infty$ then
                    petal[x'] := a';
                    d[x'] := Tenacity − d[x];
                    Q.INSERT(x')
                fi;
                Support.INSERT(x);
                x := BASE(prop[x]$^-$)
            else
                if $d[y'] = \infty$ then
                    petal[y'] := a;
                    d[y'] := Tenacity − d[y];
                    Q.INSERT(y')
                fi;
                Support.INSERT(y);
                y := BASE(prop[y]$^-$)
            fi
        od;
(2)     while $x \neq s$ and rescap(prop[x]) > 1 do
                if $d[x'] = \infty$ then
                    petal[x'] := a;
                    d[x'] := Tenacity − d[x];
                    Q.INSERT(x')
                fi;
                Support.INSERT(x);
                x := BASE(prop[x]$^-$))
        od
        if $d[x'] = \infty$ then
            petal[x'] := a;
            d[x'] := Tenacity − d[x];
            Q.INSERT(x')
        fi;
        repeat
            Support.DELETE(y);
            F.UNION(x, y)
        until Support.EMPTY;
        base[F.FIND(x)] := x;
        end;

    begin
    F.INIT;
    for v := 0 to n do
        prop[v] := NO_ARC;
        d[v] := $\infty$
    od;
    d[s] := 0;
    F.BUD(s);
    base[F.FIND(s)] := s;
    INVESTIGATE;
    Support.MAKE(n);
    Q.MAKE(n);
    Q.INSERT(s);
    while not Q.EMPTY do
        Q.DELETE(u);
        while not EOS(u) do
            READ(u, a);
            v := $a^+$;
            if $d[v'] = \infty$ then
                if $d[v] = \infty$ then
(3)                 d[v] := d[u] + 1;
                    prop[v] := a;
                    F.BUD(v);
                    base[F.FIND(v)] := v;
                    Q.INSERT(v)
                fi
            else
                if $prop[u] \neq \bar{a}$ and $prop[u'] \neq a'$ then
                    ShrinkBlossom fi
            fi
        od
    od;
    Q.FREE;
    CLOSE;
    if $d[t] < \infty$ then return TRUE else return FALSE fi
    end
end.
```

As the title suggests, the search order is breadth first like, that is, a queue Q manages the selection of the current node u in first in-first out manner, and each investigated node is *expanded,* that is, all successors of u are placed on Q immediately. Note that the FIFO selection is not essential for the further analysis and could be exchanged by some other node selection rule.

In the original paper of Kocay and Stone, the canonical element of a blossom is always its base and blossoms are not merged by their cardinality or rank, respectively. Thus, the running time was $O(n^2)$ according to the results discussed in the preceding section. In our version, additional labels $B[v]$ and $rank[v]$, encapsulated into a DSU process, are used to get the bound $O(m\alpha(m, n))$.

Our second improvement concerns the determination of the new blossom base during the shrinking operations. For this goal, we added distance labels d to the procedure which prevent *ShrinkBlossom* from investigating the nodes outside of the new blossom.

Theorem 17.1. *Let N be a balanced network. The call of BNS has the following effects:*

(*a*) *If the node v is strictly reachable, then path(s, v) determines a valid sv-path with length d[v].*
(*b*) *If one of v or v' is strictly reachable, then BASE(v) is the base of the blossom containing v.*

Proof. First, observe that the outer **while**-loop organizes the node selection. The inner **while**-loop expands the selected node u. In each iteration, an arc a with residual capacity > 0 and start node u is searched. The arc a is *investigated,* and the end node $v = a^+$ is *explored.*

We grow the set A of investigated arcs simultaneously with the algorithm by setting $A := A \cup \{a, a'\}$ after each iteration of the inner **while**-loop. We will use as the induction hypothesis that in the beginning of an arc investigation the following is true:

- $d[v]$ is finite, and $path(s, v)$ is a valid sv-path of length $d[v]$ iff $d_A(v)$ is finite.
- $F.FIND(v) = NONE$ iff $d_A(v) = d_A(v') = \infty$.
- $F.FIND(u) = F.FIND(v)$ iff u and v are in a common (A)-blossom.
- $BASE(u)$ is the base of the (A)-blossom containing u.

In the very beginning, A is empty so that all statements are true. Let A be the set of arcs investigated before a and assume that the hypothesis is true up to this point. In particular, the algorithm is tree growing until a is searched. If one of the cases

(a) $d[v] < d[v'] = \infty$
(b) $prop[u] = \bar{a}$, $d[v'] < \infty$
(c) $prop[u'] = a'$, $d[v'] < \infty$

applies, the algorithm just ignores the arc $a = uv$. Hence, we have to prove that investigating a has no effect on the reachability of nodes and on the accessibility of residual arcs in these cases:

The case (c) is trivial, since a and a' were investigated when a' was searched. Consider case (b): If $B_A(u)$ and $B_A(v)$ are different, then u is an (A)-blossom base, and every (A)-valid su-path p ends with $\bar{a}$, since the algorithm is tree growing. Hence, p cannot be extended by a validly. If u and v' are in the same (A)-blossom, then this blossom is proper since the prop a' cannot be a loop. Hence, the investigation of a is redundant by Corollary 10.4.

If case (a) applies, the arc a is an anomaly of v. If v' becomes strictly reachable later, the algorithm will expand v' and search a'. At this point, case (a) cannot apply, so the algorithm will investigate a and a' (i.e., the anomaly is resolved). Theorem 11.2 shows that the unresolved anomalies cannot prevent the algorithm from finding all strictly reachable nodes and all accessible arcs.

Suppose that a is searched by the algorithm. Then, the investigation of a is either a bud generation [starting at label (3)] or a blossom shrinking operation (managed by the procedure *ShrinkBlossom*). In the former case, the induction step is obvious since we can apply Theorem 10.2. In the latter case, we can apply Theorems 10.3–10.9 which refer to the shrinking condition.

The procedure *ShrinkBlossom* visits the paths $p := aux(path(s, u))$ and $q := aux(path(s, v'))$ of the layered auxiliary network because of the tree-growing strategy. Hence, $bot_A(u, v')$ is the last common arc of p and q which has auxiliary capacity one. Note that the loop starting at (1) traverses the disjoint parts of p and q and the loop starting at (2) traverses the common part of p and q until $bot_A(u, v')$ is reached.

By Corollary 10.9, x is $base_{\bar{A}}(u, v')$ at the end of loop (2). By Theorem 10.8, *Support* consists of all (A)-blossom bases other than x that have to be merged. Hence, $B_{\bar{A}}(u, v')$ is correctly merged together by the algorithm, and the new base is assigned to the canonical element. By the induction hypothesis, we have

$$Tenacity = d[u] + d[v'] + 1 = |p| + |q| + 1.$$

Any node w which becomes strictly reachable by the investigation of (u, v) and (v', u') satisfies $w' \in Support$ with $d[w] = d_A(w) = \infty$. For such a node, the call of $path(s, w)$ will expand parts of p and q so that $d[w] = Tenacity - d[w']$ and $|path(s, w)|$ are indeed equal. This is the induction step. ■

We give a detailed example of what is going on. Consider the balanced network N which is associated with the search for a 2-factor of the graph G in Figure 2. In Table I, we list all assignments to d, *prop,* and *petal* in order of their occurrence during the call of $N.BNS(s, t)$. Furthermore, column v shows the order in which the nodes are placed on Q.

In the beginning, the procedure grows an ordinary BFS tree including the nodes s, 10, $5'$, $8'$, 4, 6, 7, $1'$, $2'$, $3'$, $9'$, and $12'$. Then, node $1'$ is expanded, and $a = (1', 2)$ is the only arc investigated. Since $2'$ is already reachable, the procedure *ShrinkBlossom* is called. At this point, we have $path(s, 1') = (s, 10, 5', 6, 1')$ and $path(s, 2') = (s, 10, 5', 6, 2')$ which are paths of the auxiliary network since no blossoms have been shrunk yet. The procedure puts

$$Tenacity := d[u] + d[v'] + 1 = 4 + 4 + 1 = 9,$$

and $x := 1'$, $y := 2'$. Because of $d[x] = d[y]$, to the node $y' = 2$, the petal $a = (1', 2)$ and the distance $d[2] = Tenacity - d[2'] = 5$ are assigned. After that, 2 is placed on Q and $y := 6$ is set.

In the next iteration, we have $d[x] > d[y]$, and $x' = 1$ is treated in a similar way. But now $x = y = 6$ is reached, and the first **while**-loop ends. Because of $rescap(5', 6) = 1$, the second **while**-loop is not executed. Then, $x' = 6'$ is labeled just as 1 and 2 before. After this shrinking operation, 6 is the base and the canonical element of the new blossom $\{1, 1', 2, 2', 6, 6'\}$.

We will not describe all of the details since nearly

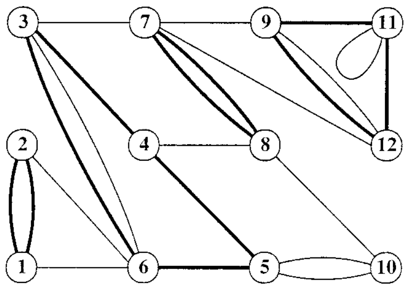

Fig. 2. A 2-factor problem.

all of the subsequent investigation steps lead to similar shrinking operations. The reader is asked to close the gaps with the aid of Table I and to perform the DSU process for himself. One can use Figure 1 which shows the DSU tree coming from this example.

Note that $path(s, 3') = (s, 10, 5', 6, 3')$ and $path(s, 6') = (s, 10, 5', 6, 1', 2, 6')$ correspond to the auxiliary paths $(s, 10, 5', 6, 3')$ and $(s, 10, 5', 6)$, respectively. Hence, the following investigation of $(3', 6)$ affects node 3 only which is labeled and placed on the queue. Furthermore, $\{3', 3\}$ is merged into the blossom with base 6.

Then, node $9'$ is expanded. During that, the arc $(9', 11)$ is investigated so that $d[11] := d[9'] + 1 = 5$ is assigned, the bud $\{11, 11'\}$ is generated, and 11 is placed on Q.

The next shrinking operation occurs during the investigation of the arc $(9', 12)$ where $path(s, 9') = (s, 10, 8', 7, 9')$ and $path(s, 12') = (s, 10, 8', 7, 12')$ are considered. Note that $(10, 8')$ instead of $(8', 7)$ is the desired bottleneck, since we have $rescap(8', 7) = 2$. In this situation, the second **while**-loop of *ShrinkBlossom* is needed. By this shrinking operation, $8'$ becomes the base and the canonical element of the new blossom.

Now, the arc $(3, 7')$ is investigated so that the paths $(s, 10, 5', 6)$ and $(s, 10, 8')$ of the auxiliary network are considered. Since $(s, 10)$ is not a bottleneck, all nodes on these paths are merged into one blossom. In particular, t becomes strictly reachable, and a valid augmenting path can be obtained by

$$\begin{aligned}path(s,t) &= path(s,3)\circ(3,7')\circ[path(s,7)]'\\ &= path(s,6')\circ(6',3)\circ[path(3',3')]'\\ &\quad \circ(3,7')\circ(s,10,8',7)'\\ &= (s,10,5',6,1')\circ(1',2)\circ(6,2')'\\ &\quad \circ(6',3)\circ(3,7',8,10',t)\\ &= (s,10,5',6,1',2,6',3,7',8,10',t).\end{aligned}$$

If we would complete the computation, we would obtain the correct distance labels of the residual network. So, the preceding example and the FIFO management suggest that the BNS algorithm might find minimum valid paths in general. As a counterexample, consider the balanced network N in Figure 3. The reader may check for himself or herself that the call of BNS will result in

$$\begin{aligned}&path(s,t)\\ &\quad = (s,25,1',2,3',4,5',6,7',8,9',10,11',12,26',t)\end{aligned}$$

instead of the shorter path

$$(s, 25, 24', 23, 22', 21, 16', 15, 14', 13, 26', t).$$

18. A DEPTH FIRST BNS ALGORITHM

In this section, we present our version of the little-known cardinality matching algorithm of Kameda and Munro [8] which is adapted to balanced flow networks. This algorithm was intriguing for its simplicity which results from the data structures used and the depth first search strategy. Unfortunately, the original paper contains a seri-

TABLE I. The call of *BNS* for our running example

v	$d[v]$	$prop[v]$	$petal[v]$
s	0	*NO_ARC*	*NO_ARC*
10	1	$(s, 10)$	*NO_ARC*
$5'$	2	$(10, 5')$	*NO_ARC*
$8'$	2	$(10, 8')$	*NO_ARC*
4	3	$(5', 4)$	*NO_ARC*
6	3	$(5', 6)$	*NO_ARC*
7	3	$(8', 7)$	*NO_ARC*
$1'$	4	$(6, 1')$	*NO_ARC*
$2'$	4	$(6, 2')$	*NO_ARC*
$3'$	4	$(6, 3')$	*NO_ARC*
$9'$	4	$(7, 9')$	*NO_ARC*
$12'$	4	$(7, 12')$	*NO_ARC*
2	5	*NO_ARC*	$(1', 2)$
1	5	*NO_ARC*	$(2', 1)$
$6'$	6	*NO_ARC*	$(1', 2)$
3	7	*NO_ARC*	$(6', 3)$
11	5	$(9', 11)$	*NO_ARC*
12	5	*NO_ARC*	$(9', 12)$
9	5	*NO_ARC*	$(12', 9)$
$7'$	6	*NO_ARC*	$(9', 12)$
8	7	*NO_ARC*	$(9', 12)$
5	9	*NO_ARC*	$(7, 3')$
$10'$	10	*NO_ARC*	$(3, 7')$
t	11	*NO_ARC*	$(3, 7')$
$11'$	6	*NO_ARC*	$(11, 11')$
$4'$	8	*NO_ARC*	$(8, 4')$

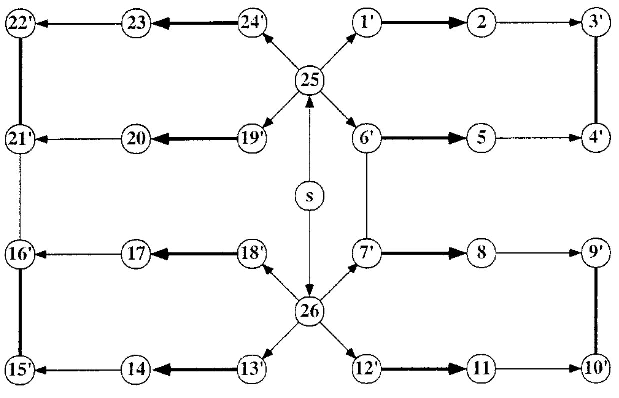

Fig. 3. Missing the minimum valid path.

ous logical mistake which has not been noticed in the subsequent literature but was observed by one of our students [2].

The algorithm determines props and petals, but maintains the layered auxiliary networks $Aux_A(N)$ very implicitly. In contrast to the breadth first search procedure of the last section, the shrinking of blossoms is managed by an additional stack instead of a DSU process.

Procedure 8. A Refined Version of the Kameda/Munro Algorithm

```
class BALANCED_NW;
private
  function BNS(s, t);
    object S1, S2: STACK;
    var a, UnresolvedAnomaly, tenacity, u, v;

    procedure Backtrack;
    begin
    if d[u'] = ∞ then S1.DELETE(u)
    else
      if S2.TOP = u then S2.DELETE(u) fi;
      if d[S1.TOP] ≤ d[S2.TOP] and not S2.EMPTY
      then u := S2.TOP
      else
        S1.DELETE(u);
        if S1.EMPTY then u := NONE else S1.
          DELETE(u)
      fi
    fi
    end;

    procedure ShrinkBlossom;
    begin
    S1.INSERT(u);
    tenacity := d[u] + d[v'] + 1;
    repeat
      S1.DELETE(u);
      if d[u'] = ∞ then
        petal[u'] := a';
        d[u'] := tenacity − d[u];
        S2.INSERT(u);
        S2.INSERT(u')
      else
        S1.DELETE(u)
      fi
    until S1.EMPTY or (d[u] ≤ d[v'] and rescap
      (prop[u]) = 1));
    S1.INSERT(u);
    u := S2.TOP
    end;

  begin
  S1.MAKE(n);
  S2.MAKE(n);
  UnresolvedAnomaly := FALSE;
  for v := 0 to n do
    prop[v] := NO_ARC;
    petal[v] := NO_ARC;
    d[v] := ∞
  od;
  d[s] := 0;
  u := s;
  INVESTIGATE;
  while u ≠ NONE do
      if EOS(u) then Backtrack
      else
        READ(u, a);
        v := a⁺;
        if d[v'] = ∞ then
          if d[v] = ∞ then
            d[v] := d[u] + 1;
            prop[v] := a;
            S1.INSERT(u);
            u := v
          fi
        else
          if prop[u] ≠ ā and prop[u'] ≠ a'
          then
            if EOS(v')
            then UnresolvedAnomaly := TRUE
            else ShrinkBlossom
            fi
          fi
        fi
      fi
  od;
  CLOSE;
  if d[t] < ∞ then return TRUE else return FALSE fi
  end
end.
```

The active path p which connects s and the active node u in terms of $Aux_A(N)$ is held on a stack $S1$ with the blossom base or the predecessor of u atop. If a proper (A)-blossom is traversed by p, then $S1$ contains the blossom base followed by the last node of B on p. If blossoms are shrunk during the investigation of some arc uv, the DFS procedure backtracks up to the base of the new $(\tilde{A})$-blossom which is an element of $S1$. All nodes on $S1$ on top of the base belong to the new blossom and are moved to $S2$ for further investigation.

This simple shrinking procedure works if the arc uv under consideration is not a **resolved anomaly.** By that we denote the following situation: The arc $v'u'$ has been considered before, but u' was reachable yet and no shrinking operation available at that point of time. In between, uv and $v'u'$ have become available for blossom shrinking.

It may happen that $base_A(v)$ is not a member of the active path or the stack $S1$. Since $v'u'$ has been considered before, we have $EOS(v') = TRUE$ from the moment of backtracking from v'. Hence, $EOS(v')$ is checked by the algorithm. The behavior of the procedure should be obvious now, although no formal correctness proof is available.

We will study the effects of Procedure 8 for our running example in Figure 2 and the associated residual balanced network N. Initially, the algorithm grows an active path $(s, 10, 5', 4, 8', 7, 3', 6, 1', 2)$ where all of the nodes but $u = 2$ form the content of $S1$. The stack $S2$ is still empty.

Then, the arc $(2, 6')$ is investigated which triggers off a blossom shrinking operation. All nodes on $S1$ are popped until $u = 6$ is reached. The stack $S2$ looks like $S2$: 2, 2′, 1′, 1, 6, 6′ now, and $u = 6' = S2.TOP$ holds. The nodes 1′ and 2 are popped from $S1$, whereas the base 6 is left for reconstructing auxiliary paths later.

The next arc investigated is $(6', 5)$, and *ShrinkBlossom* is called again. Because of $rescap(s, 10) = rescap(10, 5') = 2$, the stack $S1$ is entirely flushed, and all nodes (together with their complements) are moved to $S2$ which consists of 2, 2′, 1′, 1, 6, 6′, 3′, 3, 7, 7′, 8′, 8, 4, 4′, 5′, 5, 10, 10′, s and t now.

Then, by operation *Backtrack,* the nodes on $S2$ are searched and popped without any further effect until $u = 7$ is reached. By the investigation of $(7, 9')$ and $(9', 11)$, the nodes 7 and 9′ are pushed onto $S1$. Then, the arc $(11, 11')$ is investigated, resulting in $S1$: s, 7, 9′, 11, $S2$: 2, 2′, 1′, 1, 6, 6′, 3′, 3, 7, 11, 11′ and $u = 11'$.

Finally, $(11', 12)$ and $(12, 7')$ are investigated such that *ShrinkBlossom* is called and $S1$ is flushed again. All nodes have been found to be in one blossom. The algorithm searches the remaining nodes on $S2$ (without any effect) and then halts. We get the following augmenting path:

$$\begin{aligned} path(s,t) &= path(s,5') \circ (5',6) \circ [path(s,6')]' \\ &= path(s,10) \circ (10,5',6) \circ [path(s,6) \\ &\quad \circ (6,2') \circ [path(6,2)]']' \\ &= (s,10,5',6) \circ [path(s,3') \circ (3',6,2') \\ &\quad \circ [path(6,1') \circ (1',2)]']' \\ &= (s,10,5',6) \circ [(s,10,5',4,8',7,3',6,2') \\ &\quad \circ (6,1',2)']' \\ &= (s,10,5',6,1',2,6',3,7',8,4',5,10',t). \end{aligned}$$

Although this code is considerably simpler than the breadth first BNS procedure of the last section, there are some serious disadvantages:

- The algorithm is incorrect in the general case, since anomalies are not resolved. Incorrect results seem to be relatively rare but no one has researched this up to now.
- Depth first procedures cannot be adapted to parallelized computing techniques.
- If a matching is computed from scratch, there is a preferred direction of search given by the node numbering. Hence, arcs are flipped several times. To avoid this, the search order should be disturbed anyway.

Figure 4 shows an example where the algorithm does not find an augmenting path although such a path exists. In this figure, the lines with arrowheads are the α-props instead of the props of Section 8. The arcs $(8, 9')$ and $(17, 18')$ are α-petals, and the arcs $(1, 7')$ and $(10, 16')$ are α-anomalies which are not resolved by the algorithm. Hence, the bridge $(6, 15')$ is not investigated at all, and, thus, the augmenting path is missed.

If one wants to extend Procedure 8 to a correct BNS algorithm, one would have to resolve anomalies and to perform a DSU process again. This would be incompatible with the main idea of the algorithm and result in a rather ugly code. Therefore, we have not devised such a procedure.

Except for the disadvantages mentioned above, Procedure 8 is an economic augmentation rule which contains a simple anomaly check. Indeed, there is no augmenting path if $d[t] = \infty$ and $UnresolvedAnomaly = FALSE$ hold at the end of the search. On the other hand, if we have $d[t] < \infty$, we can expand an augmenting path and can neglect possible mistakes.

Only in the case of $d[t] = \infty$ and $UnresolvedAnomaly = TRUE,$ the result is uncertain. One might use another, probably slower BNS algorithm to verify the result in that case. Hence, it seems likely that the depth first BNS algorithm will yield very powerful heuristical matching algorithms.

SUMMARY

The BNS procedures presented in this paper lead to efficient algorithms for a broad class of matching problems. If the graphs are simple, the maximum flow value for the corresponding balanced flow networks are bounded by the number of arcs in the original graph. Hence, an f-factor can be found in $O(m^2\alpha(m, n))$ time and a k-factor (where k is fixed) in $O(nm\alpha(m, n))$ time. Even (f, g)-factors can be derived in $O(m^2\alpha(m, n))$ time.

If the graphs are not necessarily simple, the best-known algorithm for the f-factor problem is due to Anstee [1]. The algorithm determines a maximum flow for the balanced flow network. This flow is transformed into a balanced flow and finally augmented $O(n)$ times to obtain a maximum balanced flow. Here, our BNS algorithms apply. We will discuss this approach in a forthcoming paper [6].

Although Procedure 7 is almost linear, it can be improved in practice where we are looking for a valid augmenting path only: Shrinking blossoms is the expensive part of the algorithm and such operations should be avoided.

To achieve this goal, a node u is not entirely expanded, but all outgoing arcs which have the shrinking properties are collected in an additional set. These arcs are investigated only if no other operation is possible. But, of course, every augmenting path traverses a bridge.

If u is adjacent to the sink t, then either su' has been investigated before or the investigation of ut leads to an augmentation immediately. The latter situation can be detected without any effort. Hence, it is reasonable to delay the investigation of the arcs starting at the source s at long as possible.

A significant improvement would be a BNS procedure which finds valid augmenting paths of minimum length. Such an algorithm is available by the theory of Vazirani [13], but very complex. In a forthcoming paper, we present an extension of the algorithm of Micali and Vazirani [10] to balanced network flows which also runs in (almost) linear time.

TABLE II. **The call of *BNS* for our running example**

v	$d[v]$	$prop[v]$	$petal[v]$
s	0	*NO_ARC*	*NO_ARC*
10	1	$(s, 10)$	*NO_ARC*
$5'$	2	$(10, 5')$	*NO_ARC*
4	3	$(5', 4)$	*NO_ARC*
$8'$	4	$(4, 8')$	*NO_ARC*
7	5	$(8', 7)$	*NO_ARC*
$3'$	6	$(7, 3')$	*NO_ARC*
6	7	$(3', 6)$	*NO_ARC*
$1'$	8	$(6, 1')$	*NO_ARC*
2	9	$(1', 2)$	*NO_ARC*
$2'$	8	*NO_ARC*	$(6, 2')$
1	9	*NO_ARC*	$(6, 2')$
$6'$	10	*NO_ARC*	$(6, 2')$
3	7	*NO_ARC*	$(5', 6)$
$7'$	8	*NO_ARC*	$(5', 6)$
8	9	*NO_ARC*	$(5', 6)$
$4'$	10	*NO_ARC*	$(5', 6)$
5	11	*NO_ARC*	$(5', 6)$
$10'$	12	*NO_ARC*	$(5', 6)$
t	13	*NO_ARC*	$(5', 6)$
$9'$	6	$(7, 9')$	*NO_ARC*
11	7	$(9', 11)$	*NO_ARC*
$11'$	8	*NO_ARC*	$(11, 11')$
12	9	$(11', 12)$	*NO_ARC*
$12'$	6	*NO_ARC*	$(7, 12')$
9	9	*NO_ARC*	$(7, 12')$

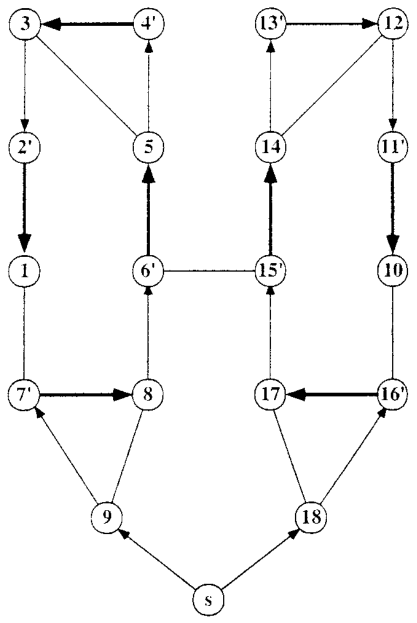

Fig. 4. Missing an augmenting path.

REFERENCES

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

[2] M. Bergdolt, Tiefensuche in balancierten Netzwerken: Ein neuer Ansatz zur Lösung von Matching-Problemen, Master's Thesis, Universität Augsburg, 1996.

[3] T. H. Corman, C. E. Leiserson, and R. L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, 1990.

[4] L. R. Ford and D. R. Fulkerson, Flows in Networks, Princeton University Press, Princeton, NJ, 1962.

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

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

[7] H. N. Gabow and R. E. Tarjan, A linear-time algorithm for a special case of disjoint set union, J Comput Syst Sci 30 (1985), 209–221.

[8] T. Kameda and I. Munro, An $O(|V|*|E|)$ algorithm for maximum matching of graphs, Computing 12 (1974), 91–98.

[9] W. Kocay and D. Stone, An algorithm for balanced flows, JCMCC 19 (1995), 3–31.

[10] 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.

[11] R. E. Tarjan, Efficiency of a good but not linear set union algorithm, J ACM 22 (1975), 215–225.

[12] R. E. Tarjan, Data Structures and Network Algorithms, SIAM, Philadelphia, PA, 1983.

[13] 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.