July, 1992 Revised: Apr., 1993
This paper appeared in
J.of Combinatorial Math and Combinatorial Computing 19 (1995) 3-31.

An Algorithm for Balanced Flows

William Kocay* and Doug Stone
Computer Science Department, University of Manitoba
Winnipeg, Manitoba, CANADA R3T 2N2
e-mail: bkocay@cs.umanitoba.ca

Abstract : The Balanced Network Search (BNS) is an algorithm which finds a maximum balanced flow in a balanced network N. This algorithm is a way of using network flows to solve a number of standard problems, including maximum matchings, the factor problem, maximum capacitated b-matchings, etc., in general graphs. The value of a maximum balanced flow equals the capacity of a minimum balanced edge-cut. Flow-carrying balanced networks contain structures called generalized blossoms. They are not based on odd cycles. Rather they are the connected components of a residual sub-network of N. An algorithm is given for finding a maximum balanced flow, by constructing complementary pairs of valid augmenting paths.

1. Balanced Networks.

Balanced networks were introduced in [7]. They are special bipartite directed graphs. Let $X=\{x_1, x_2, \ldots x_n\}$ and $Y=\{y_1, y_2, \ldots y_n\}$ be two sets of vertices. Then a balanced network N with vertices X Y has two additional vertices, the *source* s and the *target* t. A directed edge is a pair (u,v). Its *capacity* is a non-negative integer, denoted cap(uv). There is an edge (s,x_i) for each x_i X and an edge (y_i,t) for each y_i Y. All other edges (u_i,v_j) have one end in X and one end in Y, so either u_i X, v_j Y, or u_i Y, v_j X. Fig. 1 shows an example of a balanced network (although the capacities are not shown in the diagram). The vertices of N can be divided into complementary pairs. Vertices s and t are complementary ; so are x_i and y_i. Write $s'=t$, $t'=s$, $x_i = y_i$, and $y_i = x_i$ to indicate complementarity. The edges can also be divided into complementary pairs. Edges (s,x_i) and (y_i,t) are complementary edges. So are (x_i,y_j) and (x_j,y_i). In general, $(u,v)'=(v',u')$. Complementary edges must always have *equal capacities*. Thus the network N is said to be *balanced*.

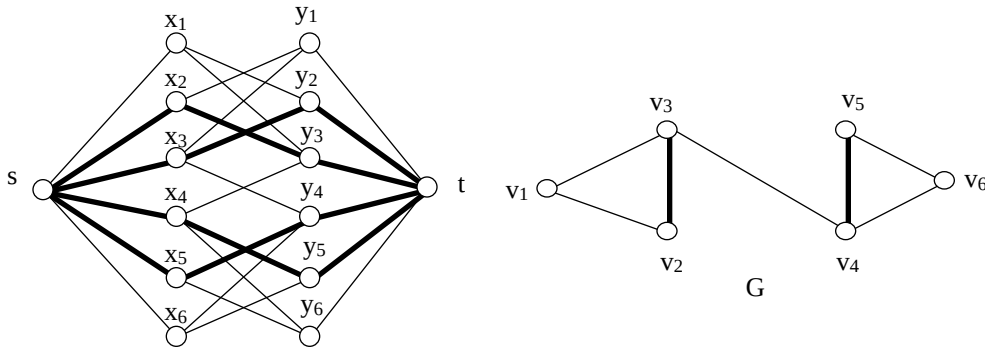

Fig. 1, A balanced network N and a graph G.
The edges of N are directed from left to right.

Let G be an undirected simple graph with n vertices $\{v_1, v_2, \ldots, v_n\}$. Edges of G are unordered pairs of vertices. We denote an edge $\{u,v\}$ of G by the pair uv, where the order is not important. Sometimes we will use the notation uv to indicate one of the edges (u,v) or (v,u) of a directed graph, when the

* This work was supported by an operating grant from the Natural Sciences and Engineering Research Council of Canada.

direction is not explicitly given. This will be clear from the context. The opposite direction will then be indicated by vu. We construct a balanced network N from G as follows. Create two sets of vertices $X=\{x_1, x_2, \ldots x_n\}$ and $Y=\{y_1, y_2, \ldots y_n\}$, and a source s and target t. Add edges (s,x_i) and (y_i,t) to N. For each edge v_iv_j of G, create edges (x_i,y_j) and (x_j,y_i) in N. The edges directed from X to Y are denoted by [X,Y]. Assign all edges capacity one. This is illustrated in Fig. 1.

Balanced networks are interesting because the methods of network flow theory [2,9] can be used in N to solve subgraph problems in G. The most natural method of solving the max-matching problem in bipartite graphs, for example, is to use network flows. The size of a max-matching equals the capacity of a min-edge-cut. Previously, network flow techniques have not been applicable to non-bipartite graphs. Balanced networks allow flow theory to be used in non-bipartite graphs, by constructing *balanced* flows. These are flows with an additional balance condition, which is defined below. The value of a max-flow will equal the capacity of a min-edge-cut. Different assignments of the capacities will be suitable for solving different problems. The edges [X,Y] of N which carry flow define a subgraph of G. In Fig. 1, the flow-carrying edges are drawn with thicker lines. For example, a maximum matching in G will correspond to a maximum flow with all capacities equal to one. Since N is bipartite even when G is not, we can expect an efficient algorithm for the general matching problem. If we choose capacities $cap(sx_i)=cap(y_it)=b(i)$, where b(i) is a non-negative integer-valued function, then a maximum flow in N will correspond to a subgraph of G in which vertex v_i has degree b(i), whenever this is possible. Thus we obtain a solution to the f-factor problem [8,11,12,13]. The same network flow algorithm that finds a maximum matching will also find the f-factor, if it exists. When the f-factor does not exist, the algorithm will, like all flow algorithms, find an edge-cut proving that the flow can not be augmented. This corresponds to an f-barrier in the theory of f-factors [12,13]. The capacity of the edge-cut will be the value of a maximum balanced flow. The same algorithm also solves the capacitated b-matching problem [10]. Balanced networks provide a simplification of factor theory for graphs. Tutte's factor theorem [12,13] is equivalent to a max-flow-min-cut theorem for balanced networks. This is described in [7].

In this paper an algorithm is presented which finds a maximum balanced flow in a balanced network. A standard network flow algorithm will also produce a maximum flow, but it will not in general be balanced. We describe a method of finding a max-flow, while maintaining the balance condition. The algorithm reveals the structure of flow-carrying balanced networks, highlighting the sub-structures called generalized blossoms. They are the connected components of a residual sub-network. The involutionary symmetry of balanced networks makes for an interesting structure for the blossoms. They arise because of the use of complementary pairs of augmenting paths. An algorithmic technique for constructing and manipulating them is developed.

The provable complexity of the current algorithm is at most $O(Kn^2)$, where K is the value of the max-flow. However, the algorithm may run faster than this in practice. If the algorithm is used to find an f-factor in a graph G with v vertices and e edges, then the network N will have n=2v+2 and m=2e+2v edges. The value of the max flow is then at most e, so the complexity of finding an f-factor is at most $O(mn^2)=O(ev^2)$. For finding a k-factor, where k is fixed, the complexity is at most $O(ev+v^3)$. This can be substantially better than existing algorithms. For example, the method of reducing a k-factor problem in G to a max matching problem in a larger graph H tends to have a higher complexity, which can be as great as $O(ve^2)$. It may be possible to improve the complexity of the balanced flow algorithm as further understanding of balanced networks is developed. One technique that comes to mind is the construction of an auxiliary network consisting of all shortest valid augmenting paths.

Let N be a balanced network with sets X,Y, source s, and target t. If P is a path from u to v, that is, a *uv-path*, then its complement P′ is a v′u′-path. This is illustrated in Fig. 2. Note that the complement of a vv′-path is another vv′-path. Thus if P is an st-path, its complement is also an st-path, since s′=t. This holds for augmenting paths, too.

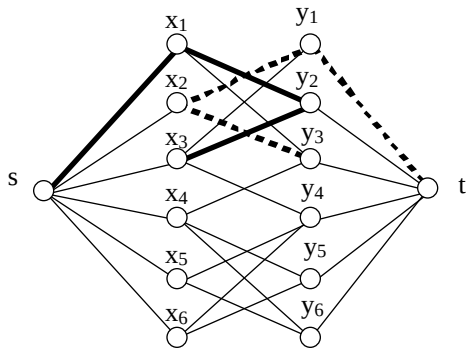

Fig. 2, An sx_3-path P and its complement P′, a y_3t-path.

Suppose that f is a flow defined on N, that is, a non-negative integer-valued function on the edges. A flow satisfies the

i) capacity constraint, $0 \le f(uv) \le cap(uv)$, for every edge uv;
ii) conservation condition, $f^+(u)=f^-(u)$ for all u X Y, where $f^+(u)$ is the total flow out of u, and $f^-(u)$ is the total flow into u.

In addition, a *balanced* flow satisfies the
iii) balance condition, $f(uv) = f((uv)')$.

The *value* of a flow is $val(f) = f^+(s) - f^-(s)$, the net flow out of the source. Basic flow terminology is from [3,9]. Let P be an sw-path for some vertex w. The direction of P is from s to w, although P need not be a directed path. Each edge uv on the path will either have the same direction or opposite direction as P. We call these *forward* and *backward* edges, respectively, of P. The *residual capacity* of an edge is

$$rescap(uv) = \begin{cases} cap(uv) - f(uv), & \text{if uv is a forward edge,} \\ f(uv), & \text{if uv is a backward edge} \end{cases}$$

The residual capacity of an edge depends on the path P. We can indicate this explicitly by writing it as rescap(uv,P). It is easy to see that rescap(uv,P) = rescap((uv)′,P′). The residual capacity of P is

$$\delta(P) = \min_{uv \in P} rescap(uv).$$

Clearly $\delta(P)=\delta(P')$. If P is an st-path of positive residual capacity, then we can *augment* on P, by changing the flow values of the edges on P according to the rule

$$f(uv) := \begin{cases} f(uv) + \delta(P), & \text{if uv is a forward edge,} \\ f(uv) - \delta(P), & \text{if uv is a backward edge} \end{cases}$$

This results in a new flow function whose value has increased by $\delta(P)$. P is called an *augmenting* path. If we augment in a balanced network, then the new flow will no longer be balanced. However, it is easy to see that if P is an augmenting path, then so is P′. If we can augment on both P and P′, then the new flow will still be balanced.

1.1 Lemma . Let P be an augmenting path in a balanced network N. Then we can augment on both P and P′ if and only if P does not contain a pair of complementary edges uv and v′u′ with rescap(uv,P)=1.

Proof. If P does not contain a pair of complementary edges, then we can augment on both P and P′ by the amount $\delta(P)$. If P does contain one or more pairs of complementary edges uv and v′u′ all with rescap(uv)>1, then we can augment on both P and P′ by at least 1 or $\lceil \delta(P)/2 \rceil$, whichever is larger. The result will still be balanced. If P contains a pair of complementary edges uv and v′u′ with

rescap(uv)=1, then after augmenting on P, we can no longer augment on P′, since rescap(v′u′, P′) now equals 0.

This lemma is illustrated in Fig. 3.

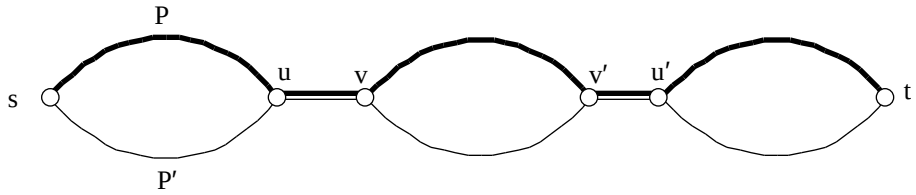

Fig. 3, Augmenting paths P and P′ with a pair of complementary edges.

1.2 Definition. We say that a uv-path P is a *valid* path if it has positive residual capacity, and it does not contain a pair of complementary edges with a residual capacity of one. A vertex v in N is said to be *s–reachable* if N contains a valid sv-path.

The maximum flow algorithm for balanced networks must distinguish between valid and invalid paths, accepting only valid augmenting paths. Write S for the set of all s-reachable vertices. Since s S, the set is non-empty. If t S, then N contains a valid augmenting path P, so that the flow can be augmented on both P and P′. If t S, then let K=[S,$\bar{S}$], the set of all edges directed from S to $\bar{S}$. K is an edge-cut that has a special structure.

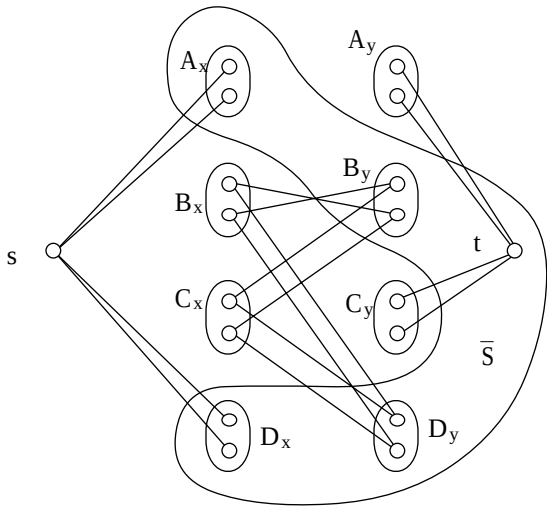

Fig. 4, Some edges of a balanced edge-cut K=[S,$\bar{S}$].
Edges from Y to X are not shown.

Let

A={ x_i, y_i | x_i S, y_i S }
B={ x_i, y_i | x_i S, y_i S }
C={ x_i, y_i | x_i S, y_i S }
D={ x_i, y_i | x_i S, y_i S }

The sets A,B,C, and D together contain all of X and Y. Write $A_X = A \cap X$, $A_Y = A \cap Y$, and so forth for B, C, and D. The capacity of K is cap(K)= $\sum_{uv \in K}$ cap(uv). The subgraph of N induced by C is N[C]. In general, if C≠Ø, it will consist of a number of connected components C_1, C_2, …, C_k, where k≥1. Write K_i for those edges of K with one endpoint in C_i. The following theorem was proved in [7].

1.3 Theorem. Let K=[S,$\bar{S}$], where S is the set of s-reachable vertices, and t $\bar{S}$. Then

i) Each C_i is self-complementary, that is, $C_i' = C_i$, for i=1, 2, … k.
ii) There are no edges between C and D, that is, [C,D]=[D,C]=Ø
iii) Each K_i has odd capacity.

1.4 Definition. Any edge-cut $K=[S,\bar{S}]$ which satisfies the 3 conditions of this theorem is called a *balanced edge-cut*. Its balanced capacity is balcap(K)=cap(K) – odd(K), where odd(K) is the number of connected components of N[C]. We call this number odd(K), since each $cap(K_i)$ is odd.

The following three important results are from [7], where their proofs can be found.

1.5 Lemma. Let f be a balanced flow in N, and let K be any balanced edge-cut. Then val(f)≤balcap(K).

1.6 Max-Balanced-Flow-Min-Balanced-Cut Theorem. The value of a maximum balanced flow equals the capacity of a minimum balanced edge-cut, that is, val(f)=balcap(K) when f is maximum and K is minimum.

1.7 Corollary. A balanced flow f in N is maximum if and only if N does not contain any valid augmenting path.

These theorems mean that flow-theoretic techniques can be used to solve subgraph problems in graphs. The algorithm used to find a maximum balanced flow must use only valid augmenting paths. When the flow value can no longer be increased, the set of s-reachable vertices will provide a proof that f is maximum. The algorithm presented in this paper is based on a breadth-first search of N, which constructs the set S of all s-reachable vertices, storing a valid sv-path for each v S.

2. The Balanced Network Search Algorithm.

Let N be a balanced network, and let f be a flow in N. Initially f will be the zero-flow, that is, f(uv)=0, for all edges uv. The *Balanced Network Search* (BNS) is an algorithm that searches for a valid augmenting path. As soon as it finds a single valid augmenting path P, the flow f must be augmented on P and on its complement P′. It finds a valid augmenting path by building a *mirror* network M, described below. The algorithm is based on a breadth-first search. Starting at the source s, a tree T is built. Vertices are stored on a queue, called the ScanQ, which initially contains only s. The ScanQ is used to accumulate the s-reachable vertices. The tree T contains a valid sv-path for each v T. Simultaneously with the building of T, the complementary tree T′ is also built. This is indicated by the pseudo-code following, and illustrated in Fig. 5. The trees T and T′ must have no edges or vertices in common. Initially T will be built just as in a breadth-first search, and T′ will be the complementary tree. T will contain a valid sv-path for each v T, and T′ will contain the complementary valid v′t-path.

```
Procedure BNS; { first version }
{ N is a network with source s, and target t. Build trees T and T′ }
ScanQ: queue of vertices { stored as an array }
QSize: integer { size of ScanQ }
T, T′: trees
begin
  ScanQ[1] := s; QSize := 1 { put s on ScanQ }
  initially T contains s, and T′ contains t
  k := 1
  repeat
    u := ScanQ[k] { select u from head of ScanQ, u is an s-reachable vertex }
    for all v adjacent to u do
      if v ScanQ and rescap(uv)>0 then
        if v′ ScanQ then begin
          add v to T, add v′ to T′
          add uv to T, add (uv)′ to T′
          QSize := QSize + 1; ScanQ[QSize]=v { add v to ScanQ }
```

```
        end
        else begin { v′  ScanQ,    v′  T,  v  T′ }
          { T contains a valid su-path and sv′-path, T′ contains a valid  u′t-path and vt-path }
          does the addition of edges uv and v′u′ to T and T′ create a pair of valid st-paths ?
          this will be handled by constructing blossoms, as described in the following pages.
        end
    k := k + 1
  until  k> QSize
end
```

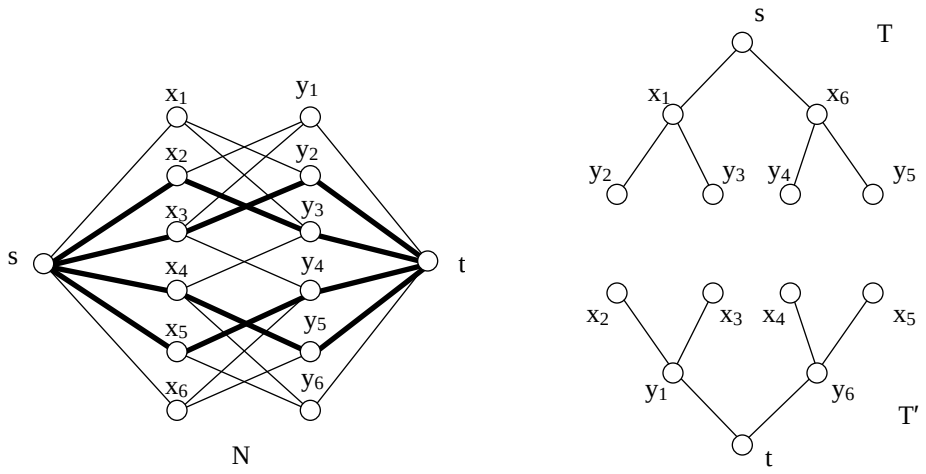

Fig. 5, Trees T and T′ built by a BNS from network N.

In each iteration of the algorithm, a vertex u ScanQ is selected. In this pseudo-code, only vertices of T are placed on the ScanQ. So u T. All v adjacent to u are considered, in order to extend T. If v ScanQ and v′ ScanQ, then the edge uv is added to T, and v′u′ is added to T′, thereby simultaneously extending both T and T′. Each time an edge uv is added to T, the complementary edge (uv)′ is added to T′. This continues until the situation arises in which the else-clause above becomes effective. At that point u is adjacent to a vertex v ScanQ such that v′ ScanQ. This means that v′ T and therefore v T′. An edge uv connecting T and T′ has been discovered and the calculation of the valid paths becomes more complicated. The pseudo-code must be extended in order to specify what happens in this situation. This requires the introduction of *blossoms*.

The tree T that is built is a breadth-first tree, rooted at s, of edges of positive residual capacity such that T has no edges or vertices in common with its complement T′. T is the largest such tree that can be built. T contains a valid sv-path for each v T, and T′ contains the complementary v′t-path. The BNS considers all edges of N of positive residual capacity. They form a subnetwork M of N. So M contains all of T and T′. It will also contain a number of other edges, as described below, which are always added to M in complementary pairs. The subnetwork M constructed by the BNS is called the *mirror* network. The reason for this appellation is the involution of complementarity evident in Fig. 5.

Blossoms.

The ScanQ contains the set of all s-reachable vertices in the mirror network M. Call this set S. When T and T′ are being built, S will consist of vertices of T only. Write K=[S,$\bar{S}$]. As in Fig. 4, X and Y can be decomposed into sets A, B, C, and D. Initially all vertices are contained in D. As the trees T and T′ are built, their vertices will be placed into the sets A and B, but C will still be empty. As edges uv connecting T to T′ are discovered and added to M, the set C will start to grow. The connected components $C_1, C_2, \ldots, C_k$ of M[C] are important.

2.1 Definition . The connected components of M[C] are called *blossoms*. Let u C be any vertex. The blossom containing u is denoted C(u).

The network M is changing dynamically, and so, therefore, are the blossoms. M is a balanced

subnetwork of N consisting of all valid paths discovered so far, and their complements. The connected components $C_1, C_2, \ldots, C_k$ of M[C] have a number of important properties, whose proofs can be found in [7].

2.2 Property. Each C_i is connected.
2.3 Property. Each C_i is self-complementary, $C_i' = C_i$.
2.4 Property. Each C_i contains a unique vertex b_i, called its *base*, such that every valid sv-path contains b_i, for all v C_i,

The connected components of M[C] are a kind of generalization of Edmonds' blossoms [4,9]. In the case when the capacities are all one, when a balanced flow in N corresponds to a matching in G, the connected components are equivalent to Edmond's blossoms. One interesting point is that Edmond's blossoms are based on odd cycles, but since M is bipartite, it has no odd cycles. The blossoms are the *connected components* of M[C]. The key properties of blossoms are that they are connected and self-complementary, and that each blossom has a unique base. Note that blossoms belong to the mirror network M, not to N. Note also that M may have other connected components than the blossoms.

Data Structures.

The BNS builds the mirror network M. The ScanQ contains the set S of all s-reachable vertices in M. For each v S, a valid sv-path is also constructed, and stored in the data structures. Initially there are no blossoms, so that C=Ø, and the BNS constructs a breadth-first tree T, rooted at s. The easiest way to store T is to keep a value PrevPt[v], for each v T, being the parent of v in the rooted tree T. Thus if we begin at v and successively follow the previous pointers PrevPt[v], we eventually reach s. This gives a means of constructing a valid sv-path, provided v T. The complementary tree T′ is rooted at t. It is not stored explicitly, since it is just the complement of T, and can be accessed via T.

At the point in the algorithm illustrated in Fig. 5, the ScanQ will contain the vertices of T. On the next iteration, a vertex $u=y_2$ ScanQ will be selected and all adjacent vertices v will be considered. When $v=x_3$, the situation arises in which u T, v T′, u′ T′, v′ T, as shown in Fig. 6. When uv and (uv)′ are added to M, C will no longer be empty, for u and u′, and v and v′ will all be s-reachable. A blossom will appear.

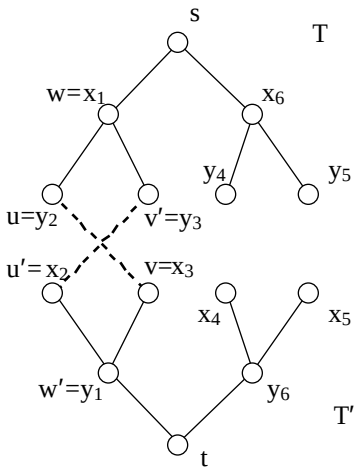

Fig. 6, The creation of a blossom.

The blossoms are the connected components of M[C]. M is changing dynamically. Before any edges connecting T to T′ are discovered, M will consist of T T′, so C=Ø. When the edge uv in Fig. 6 is discovered and added to M, together with v′u′, the vertices C={u,v,w,u′,v′,w′} form a self-complementary set of s-reachable vertices. They induce a connected subgraph of M, so they form a blossom. The trees T and T′ have not changed.

In general, suppose that at some stage of the algorithm M[C] has components $C_1, C_2, \ldots, C_k$. When an edge uv connecting u C_i to v C_j is discovered, C_i and C_j become connected to each other, and must be merged into a new, larger component. Since blossoms are self-complementary, we know that u′ C_i and v′ C_j. This is an ideal situation for the merge-find (also called set-union) data structure [1,5]. Each blossom C_i is represented by its base b_i (see Property 2.4 above). So blossoms can be distinguished by their bases. We store an array

$$\text{BasePtr}[u] = \begin{cases} 0, & \text{if u is not in a blossom,} \\ -1, & \text{if u is the base of its blossom,} \\ v, & \text{a vertex closer to the base, otherwise.} \end{cases}$$

The BasePtr[u] points toward the base of the blossom containing u. We use a function FindBase(u) to find the base of the blossom containing u.

```
FindBase (u: vertex): vertex
var b: vertex
begin
  if BasePtr[u]=0 then FindBase := 0
  else if BasePtr[u]<0 then FindBase := u
  else begin
    b := FindBase(BasePtr[u])
    BasePtr[u] := b  { path compression }
    FindBase := b
  end
end  { FindBase }
```

In order to merge two blossoms C_i and C_j, it is only necessary to assign BasePtr[b_j] := b_i. This can be made more efficient by always merging the smaller to the larger (see [1]). Initially there are no blossoms. When a vertex v is placed into T, and v′ is placed into T′, it is convenient to create a trivial blossom C(v)={v,v′}, in order to facilitate the merging and construction of blossoms.

2.5 Definition. A *trivial blossom* is a pair {v, v′}, for any vertex v T.

Strictly speaking these are not blossoms at all, since they are not connected. However they have the desirable property of being self-complementary. When they are merged to form blossoms they will create connected components, and will retain their self-complementarity. Initially the algorithm creates a single trivial blossom {s,t} by assigning BasePtr[t] := s and BasePtr[s] := –1. All other vertices have BasePtr[v]=0. When v is added to the tree T, a trivial blossom {v,v′} is created.

In the example of Fig. 6 before the edge uv is added to M, M contains a trivial blossom for each vertex of T, namely {u,u′}, {v,v′}, {w,w′}, etc. Once uv and v′u′ are added to M, these 3 trivial blossoms become merged into a single blossom, which is now a connected component M[C], where C={u,v,w,u′,v′,w′}. M[C] will therefore satisfy Properties 2.2, 2.3, and 2.4. As we shall see, the algorithm will recognize that all vertices of C now become s-reachable, and thereupon place u′, v, and w′ on the ScanQ. There will still be 4 trivial blossoms in M at this point.

Suppose now that u T is adjacent to v T′, as in Fig. 6. For any vertex z S, P_z denotes the valid sz-path in M stored by the algorithm. P_u and $P_{v'}$ are paths in the tree T. They both begin at s. Let w be the last vertex common to P_u and $P_{v'}$, travelling from s. Then P_w is the common portion of P_u and $P_{v'}$. Let P_{wu} denote the portion of P_u from w to u, and let $P_{wv'}$ denote the portion of $P_{v'}$ from w to v′. Paths can be concatenated, so that $P_u=P_wP_{wu}$ and $P_{v'}=P_wP_{wv'}$. Then w will be the base of the new blossom C(u) containing u and v. For let Q := $P_{wu}uvP'_{w\diamond}$. Q is a valid ww′-path which contains u and v. Similarly, Q′ is also a valid ww′-path. It contains w, u′, v′ and w′. Since P_wQ and P_wQ' are both valid paths, all vertices of Q Q′ are therefore in one connected component of M[C], that is, in one blossom. The BNS constructs this new blossom by following the paths P_{wu} and $P_{wv'}$, merging existing blossoms as it goes. Since the vertices in these trivial blossoms all become s-reachable, they are placed on the ScanQ as they are merged. So the ScanQ will contain vertices of both T and T′. The ScanQ will intially contain only vertices of T. When a blossom C_i is created, all its vertices will be placed on the ScanQ. So all vertices of the ScanQ are either members of trivial blossoms or of blossoms.

Switch Edges.

Before the algorithm can be presented in pseudo-code, the idea of a switch-edge must be introduced. If u T, then P_u, a valid su-path, can be constructed by successively executing u:=PrevPt[u], until u=s. But for vertices in T′, the situation is different. Consider the situation of Fig. 6 above, where uv and v′u′ are added to M thereby creating a blossom. Here u,v′ T, and u′,v T′. Vertices v and u′ now become s-reachable; furthermore, the valid paths to v and u′ must use the edges uv and v′u′, respectively. For each vertex z S, we define a switch-edge, being an edge that allows a valid sz-path to be constructed. SwitchEdge[v]= uv, and SwitchEdge[u′]=v′u′. (We don't know the direction of the edge uv; it may be (u,v) or (v,u). In either case, the complementary edge is indicated by v′u′.) Vertex w′ also becomes s-reachable when uv is added to M, and a valid sw′-path must use one of uv and v′u′. Therefore we choose one of them, say uv, and define SwitchEdge[w′]= uv. This is summarised as follows.

2.6 Definition. For each vertex z S, we define a *switch-edge*. When the addition of an edge uv to M causes a vertex z to become s-reachable (where z was previously non-reachable), z is placed on the ScanQ, that is, into S. The edge uv is said to be a switch-edge for z. We choose the order of the vertices uv to be such that the valid sz-path consists of a valid su-path, followed by edge uv, followed by a valid vz-path.

For vertices in z T, we can take SwitchEdge[z]:=yz, where y=PrevPt[z], since it is the edge yz that allows z to be s-reachable. As the above example shows, switch-edges are more complicated for vertices of T′. It is clear that every vertex z S has a switch-edge, since z is s-reachable. If z S, we take SwitchEdge[z]=Ø. An important part of the BNS algorithm is to be sure that switch-edges are properly computed, since they are used for constructing the valid sz-path. These edges are called switch-edges for the following reason. Suppose that SwitchEdge[z]=uv. When uv was added to M, it caused two existing blossoms (possibly trivial blossoms) to be merged into a larger blossom. When the valid sz-path is being traversed, the edge uv *switches* the path from the su-path to the vt-path.

The network M need not be stored explicitly, since it is given implicitly by the tree T, and the set of valid paths, which in turn are implicitly given by the switch-edges and the methods (procedures) which construct the valid paths.

```
BNS(N: balanced network)  { refined version }
{ N has source s, target t , the mirror network M is constructed }
ScanQ: queue of vertices { stored as an array, it contains the set S of s-reachable vertices }
QSize: integer  { current size of ScanQ }
PrevPt[v]: vertex  { the parent of v, for vertices v  T }
BasePtr[v]: vertex  { pointer toward the base of C(v) }
SwitchEdge[v]: edge { a pair of vertices }
begin
  set all SwitchEdge[·] := Ø,  all PrevPt[·] := 0,  all BasePtr[·] := 0
```

```
    ScanQ[1] := s;  QSize := 1   { put s on ScanQ }
    BasePtr[t] := s;  BasePtr[s] := –1  { create a trivial blossom C(s) with base s }
    k := 1
(A):repeat
      u := ScanQ[k]  { select u from head of ScanQ }
      { since u is on the queue, it has a blossom C(u) with base b_u  }
      b_u := FindBase(u)
      for all v adjacent to u do
      if PrevPt[u]≠v { avoid edges of T } and rescap(uv)>0 then begin
        b_v := FindBase(v)
        if b_v=0 then begin
          { v is not yet in T or T′ — add it to T and M }
          PrevPt[v] := u  { this effectively adds v to T and v′ to T′ }
          QSize := QSize + 1;  ScanQ[QSize] := v
          SwitchEdge[v] := uv  { this effectively adds uv and v′u′ to M }
          BasePtr[v′] := v;  BasePtr[v] := −1  { create a trivial blossom C(v) with base v }
        end
        else if v′ is s-reachable  { if v′ is s-reachable, an st-path is given by P_u uv P′_v′ }
        and PrevPt[u′]≠v′  { avoid edges of T′ }
        and b_u≠b_v then begin
          { there is now a valid sv-path via u avoiding b_v  (unless v=b_v,),
                u, v, u′, and v′ now all become part of the same connected component of M[C] }
          w := MakeBlossom(u, v, b_u, b_v)  { this constructs the new blossom and returns its base }
          b_u := w   { the new base of C(u) }
          if w′=t then begin
            { t is now s-reachable, a valid augmenting path P exists in M }
            δ := FindPathCap(s, t, 10000)    { compute the residual capacity of P   P′ }
            PullFlow(s, t, δ)  { augment on a pair of valid st-paths }
            return
          end
        end
      end
      k := k + 1  { advance ScanQ }
    until k>QSize
    { if this point is reached, no valid augmenting path exists, ScanQ contains
      the set S of all s-reachable vertices and K=[S,S̄] is a minimum balanced edge-cut }
  end  { BNS }
```

The main part of the blossom construction is contained in the procedure MakeBlossom(u, v, b_u, b_v) which must still be described. We begin by stating a number of simple properties of M and T.

2.7 Property. The vertices of M consist of T T′. T∩T′=Ø.
Proof. As the BNS progresses, each vertex encountered is added to one of T or T′, but not both.

2.8 Property. The edges of M consist of the switch-edges and their complements.
Proof. The edges of T are all switch-edges. Edges are added to M in complementary pairs, one of which is always a switch-edge.

2.9 Lemma . T is really a forest of rooted trees.
Proof. By induction on the number of vertices in T. Refer to Fig. 7. Initially T={s}, which is a forest consisting of one tree, rooted at s. Consider the way in which vertices are added to T by the BNS. When u ScanQ is being processed, all adjacent vertices v are taken in turn. If b_v=0 then v is not in any blossom yet, so v is added to T (and v′ to T′). We set PrevPt[v] := u. If u T, then u is contained in some component T(u) of the forest T. T(u) is a rooted tree. So adding v just extends T(u) to a larger tree, and T remains a forest with the same number of components. But if u T′, then it does not belong to any component of T. Setting PrevPt[v] := u effectively begins a new component T(v) of T, rooted at v, since u T. So adding v to T creates a new component. Thus T is a forest of rooted trees.

By complementarity, so is T′.

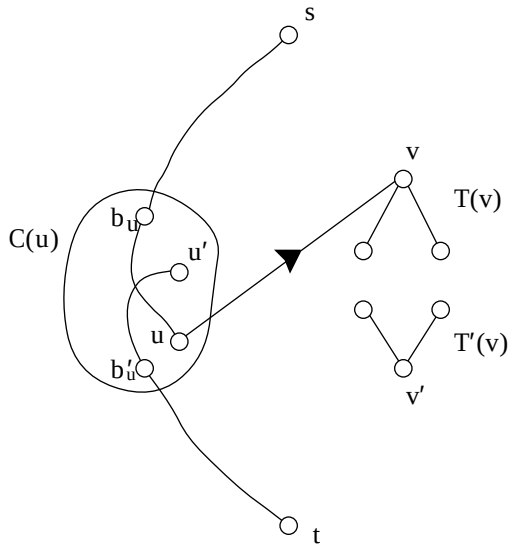

Fig. 7, T is a forest of rooted trees.

We use T(v) to denote the component of T containing v, and T′(v) for its complement. Since T is really a forest, constructing the valid path P_z from z T(v) to s will turn out to be more complicated than following PrevPt[·] when T(v)≠T(s). Suppose that v is the root of T(v), with PrevPt[v]=u. Then $P_z=P_u uv P_{vz}$, where P_{vz} is the valid vz-path contained within T(v). In order to construct P_z, we need to be able to construct P_u, where u T′(s). A recursive method will be given below.

2.10 Property . The base of every blossom is in T.
Proof. Let C_i be a blossom with base b_i. Since $C'_i=C_i$, C_i contains vertices of T and of T′. By Property 2.4, every valid sv-path to v C_i contains b_i, so b_i is the first s-reachable vertex of C_i. All vertices of T(s) are s-reachable on valid paths contained within T. So if b_i T(s) T′(s), then b_i T(s). If T(v) is any other component of T, the root of T(v) is the first s-reachable vertex of T(v). T(v) is constructed in the same way as T(s), so T(v) contains a valid path from v to every vertex in T(v). So if b_i T(v) T′(v), then b_i T(v). Therefore the base of every blossom is in T.

Suppose now that the mirror network M has been constructed up to some point, and that M contains a valid sz-path to all vertices z ScanQ. Let P_z denote this path. Every vertex of P_z is s-reachable, so its vertices are all in blossoms or trivial blossoms. We say that a vertex w P_z is a *blossom base* if it is the base of its blossom or trivial blossom.

2.11 Lemma. Let z be an s-reachable vertex and let P_z be a valid sz-path in M. Then every valid sz-path in M contains exactly the same sequence of blossom bases as P_z.
Proof. Suppose that P were another valid sz-path. Let b_z be the base of C(z). By Property 2.4, b_z is the last blossom base on both P_z and P. Suppose that the previous vertex on P_z is x and the previous vertex on P is y. By Properties 2.2 and 2.3 we know that C(z) contains a valid $b_z b_{z'}$-path Q. Then x′ and y′ are not s-reachable, for otherwise x, x′, y, and y′ would all be part of C(z). But if x≠y, then $xb_zQb_{z'}y'$ would be a valid path making y′ s-reachable. Hence x=y, and the edge xb_z is common to P_z and P. Now x is either in a blossom or trivial blossom, since it is s-reachable, and C(x)≠C(z). By Property 2.4, we find that b_x, the base of C(x) is common to P_z and P. Continuing in this way up to s we find that P_z and P contain exactly the same sequence of blossom bases.

2.12 Theorem. Consider an iteration of the repeat loop (A) in which a vertex u ScanQ is selected, and all vertices v adjacent to u are taken in turn. Suppose that u C(u) and v C(v), where C(u)≠C(v). Let w be the last blossom base common to P_u and $P_{v'}$, travelling from s, such that if w≠s, then rescap(zw)=1, where z=PrevPt[w]. Then:
(1) the base of the new blossom containing u and v is w ;
(2) the new blossom consists of all blossoms whose base appears on Q Q′, where $Q = P_{wu}uvP'_{w\diamond}$.

Proof. Refer to Figs. 8 and 9. Let b_u be the base of C(u) and b_v the base of C(v). The new blossom is the connected component of M[C] containing u, once edges uv and v′u′ have been added to M. It is constructed by merging together a number of existing blossoms and trivial blossoms of M. The only vertices that can be affected by the addition of uv and v′u′ are those which become s-reachable on paths containing uv or v′u′. Suppose that x is a vertex which becomes s-reachable on a path P containing uv. Then since u P, by Lemma 2.11 P_u and P contain the same sequence of blossom bases up to b_u. Without loss of generality, we can assume that x C(v). By Property 2.4, every path of M exiting C(v) leaves by b'_v, so b'_v P. Therefore b_v P′. But b_v $P_{v'}$, too. As in the proof of Lemma 2.11, the subsequent blossom bases of P′ and $P_{v'}$ are the same. Therefore x is in a blossom or trivial blossom C(x) whose base b_x $P_{v'}$. Similarly, if x is a vertex that becomes s-reachable on a path containing v′u′, we find that b_x P_u. So every vertex x that becomes s-reachable using one of uv or v′u′ has its blossom base b_x on one of P_u or $P_{v'}$. If w is the vertex described above, and $Q=P_{wu}uvP'_{w\phi}$, then P_wQ is a valid sw′-path. Note that $P'_{w\phi} = P_{vw'}$. So all blossoms whose bases appear on Q Q′ will form part of the new blossom. No other blossoms will be involved since they must have their bases on P_u or $P_{v'}$, but rescap(wz)=1, and all sw-paths contain the edge zw, by Lemma 2.11. This completes the proof.

Initially the BNS works like a breadth-first search, building T and T′. There is a unique valid sz-path to each z T. The only blossoms are trivial blossoms representing the pairing of vertices in T with their complements in T′. This happens until the statement "if v′ is s-reachable and PrevPt[u′]≠v′ and $b_u \neq b_v$ then begin" takes effect for the first time. At this point, C(u) and C(v) are trivial blossoms, u T, v T′, $b_u=u$, and $b_v=v'$. $Q = P_{wu}uvP'_{w\phi}$ and Q′ are both valid ww′-paths. Therefore Q Q′ is a self-complementary connected set of vertices all s-reachable. It becomes a blossom (see Figs. 6 and 8).

Notice that Q and Q′ are always two distinct valid ww′-paths. They may intersect, but if they do, the common edges have residual capacity of least 2. For example, in Fig. 6, the blossom base is $w=x_1$ since the edge sx_1 has residual capacity 1. If rescap(sx_1) were ≥2, then the base of the blossom constructed would be s rather than x_1. This would then imply the existence of a pair of valid augmenting paths.

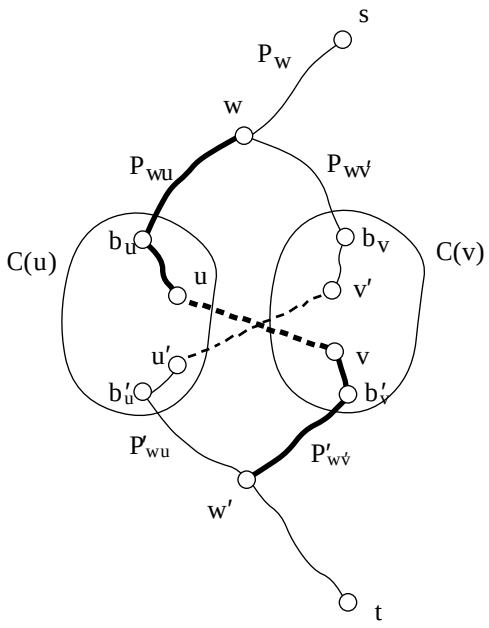

Fig. 8, Construction of a blossom. Q (in bold) = $P_{wu}uvP'_{w\phi}$.

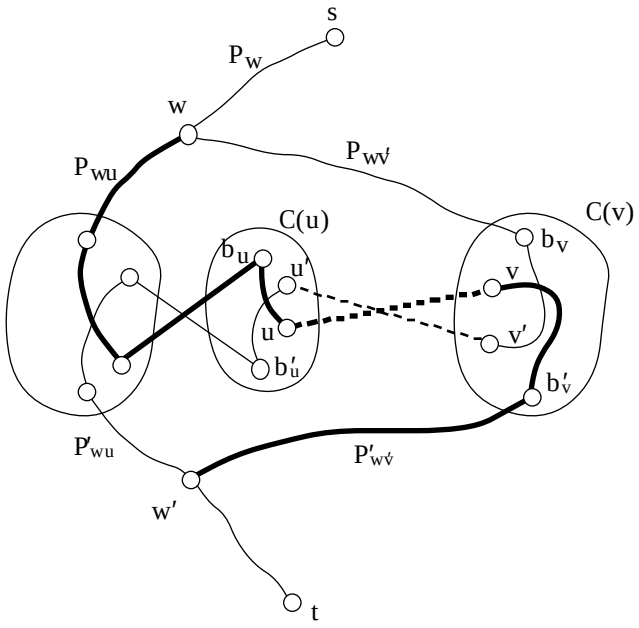

Fig. 9, Construction of a blossom. Q (in bold) = $P_{wu}uvP'_{wv'}$.

So the key to constructing a new blossom is to find the path Q and the new base w. It is easy to find w. We construct P_u and $P_{v'}$ by following PrevPt[·] back to s, storing the sequence of vertices on an array, and then finding the last common point. Q is then given by the vertices of P_u and $P_{v'}$. T is a forest of rooted trees, and the paths can meander through several blossoms (see Fig. 9). Since we only need the bases of these blossoms, a call is made to FindBase(·) at each step. Vertices z T will be in trivial blossoms, so that FindBase(z) will just return z in such cases.

```
Function FindPath (x): array
{ x is the base of a blossom, construct a valid path P of blossom bases to s }
P: array
begin
  i := 1;
  P[1] := x
  while x≠s do begin
    x := FindBase(PrevPt[x])
    i := i + 1
    P[i] := x
  end
  FindPath := P
end { FindPath }
```

In order to find the base of the new blossom, the paths P_u and $P_{v'}$ are constructed and compared in order to find the last blossom base they have in common which is reachable on a valid path.

```
Function MakeBlossom (u,v, bu, bv: vertex): vertex
{ edge uv connects two blossoms, their bases are bu and bv }
begin
  Pu := FindPath(bu)
  Pv := FindPath(bv)
  i := length(Pu); j := length(Pv) { the lengths of the paths will be available }
  { initially Pu[i] and Pv[j] both equal s, but Pu[1]≠Pv[1] }
  { find the last blossom base common to Pu and Pv }
  while Pu[i]=Pv[j] and i>0 and j>0 do begin
    i := i – 1; j := j – 1
  end
```

```
    i := i + 1;   w := Pu[i]  { w is the last common vertex }
    z := PrevPt[w]
    { now extend the blossom if rescap(zw)≥2 }
    while w≠s and rescap(zw)≥2 do begin
      i := i + 1;  w := Pu[i];  z := PrevPt[w]
    end
    { w is the base of the new blossom }
    { first follow the path Pu from w to b_u }
    for i:=i–1 downto 1 do begin
      z := Pu[i]  { z is the base of a blossom }
      BasePtr[z] := w;  BasePtr[z′] := w  { w is the new base of the blossom }
      { z and z′ may already be part of a blossom that is being swallowed into a larger blossom.
        We don't want to change the switch edge in that case }
      if z′  ScanQ then begin
        SwitchEdge[z′] := v′u′   { set the switch edge of z′ }
        QSize := QSize + 1;  ScanQ[QSize] := z′  { add z′ to ScanQ }
        mark z′ s-reachable
      end
    end
    { now follow the path Pv }
    for j:=j downto 1 do begin
      z := Pv[j]  { z is the base of a blossom }
      BasePtr[z] := w;  BasePtr[z′] := w  { w is the new base of the blossom }
      { z and z′ may already be part of a blossom that is being swallowed into a larger blossom.
        We don't want to change the switch edge in that case }
      if z′  ScanQ then begin
        SwitchEdge[z′] := uv   { set the switch edge of z′ }
        QSize := QSize + 1;  ScanQ[QSize] := z′  { add z′ to ScanQ }
        mark z′ s-reachable
      end
    end
    if w′  ScanQ then begin  { add w′ into the blossom }
      SwitchEdge[w′] := uv   { set the switch edge of w′ }
      QSize := QSize + 1;  ScanQ[QSize] := w′  { add w′ to ScanQ }
      mark w′ s-reachable
    end
    MakeBlossom := w
  end  { MakeBlossom }
```

When t is found to be s-reachable, a valid st-path P is known to exist. Its complementary path P′ is also valid. Once the residual capacity δ(P) is known, the flow is augmented by calling PullFlow(s, t, δ). It constructs the path P by using the switch-edges. Let uv=SwitchEdge[t]. Then P is given by a valid su-path, followed by the edge uv, followed by a valid vt-path. More generally, let x and y be two vertices of P, such that x comes before y on P. PullFlow(x, y, δ) is a recursive procedure that constructs the xy-portion of the path, and its complement, recursively. Let wz=SwitchEdge[y]. PullFlow(x, y, δ) uses the xw-portion and the zy-portion of P, as illustrated in Fig. 10. Since it must also augment on P′ simultaneously, the zy-portion is replaced by the y′z′-portion.

```
  PullFlow (x, y: vertex, δ: integer)
  { augment the flow by δ on all edges and their complements on a path P between x and y }
  w, z: integer
  begin
    wz := SwitchEdge[y]
    { P consists of a path from x to w, then wz, then a path from z to y }
    augment on wz and z′w′ by δ
    { w may equal x, in which case there is no need to call PullFlow(x, w) }
```

```
  if w≠x then PullFlow(x, w, δ)  { augment between x and w }
  { z may equal y, in which case z is just PrevPt[y] }
  if z≠y then PullFlow(y′, z′, δ)  { augment between z and y }
end  { PullFlow }
```

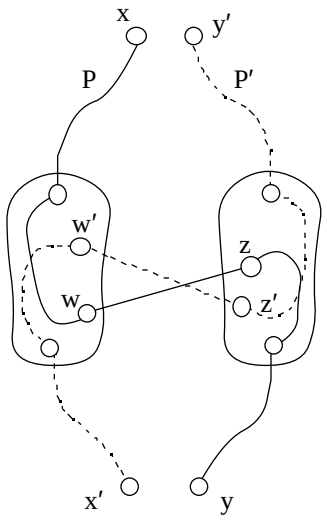

Fig. 10, Using a switch-edge to construct P and P′

Before augmenting on the two paths, it is necessary to find δ(P). This can be done by following the paths and computing the minimum residual capacity of all edges on P. An edge on both P and P′ counts for only half of its actual residual capacity, since augmenting on P by δ will simultaneously reduce its capacity on P′ by δ. The path P can only be followed by using the switch-edges, as in PullFlow(x, y, δ). FindPathCap(x, y, δ) is a recursive procedure that finds the residual capacity on the portion of P between x and y. δ is the minimum capacity found so far along the path.

```
Function FindPathCap  (x, y: vertex, δ: integer): integer
{ find the minimum residual capacity of all edges between x and y in a valid st-path P }
{ δ is the minimum found so far }
{ the vertices occur in the order s,…,x,…,y,…,t along P }
{ the vertices occur in the order s,…,y′,…,x′,…,t along P′ }
w, z, cap: integer
begin
  wz := SwitchEdge[y]  { wz is on path P }
  cap := rescap(wz)
  if z′w′ is also on P, then cap := cap/2  { wz, z′w′  P∩P′ }
  if cap<δ then δ := cap
  { P consists of a path from x to w, then wz, then a path from z to y }
  if w≠x then δ := min(δ, FindPathCap(x, w, δ)) { the portion between x and w }
  if z≠y then δ := min(δ, FindPathCap(y′, z′, δ)) { the portion between z and y }
  FindPathCap := δ
end  { FindPathCap }
```

3. Correctness and Complexity.

The mirror network M is a subnetwork of N, built by the BNS. M contains the trees T and T′, as well as all the switch-edges and blossoms discovered at any point in the algorithm.

3.1 Lemma. Let k≥1. At the beginning of iteration k of the repeat loop (A):
— the ScanQ consists of all s-reachable vertices of M;
— M contains a valid su-path P_u for all u ScanQ;
— if the switch-edge of u is xy, then P_u consists of an sx-path P_x followed by xy, followed by a yu-path P_{yu}, that is, $P_u = P_x x y P_{yu}$;
— PullFlow(s, u, δ) will construct P_u P'_t.

Proof. Initially the BNS behaves like a breadth-first search, constructing a breadth-first tree T, and its complement T′. If u T, then SwitchEdge[u] is the edge xu, where x=PrevPt[u], the parent of u in T. All vertices of T are s-reachable; the valid su-path P_u is given by P_x followed by xu. PullFlow(s, u, δ) will use the switch-edges xy := SwitchEdge[u] to follow PrevPt[·], if u T, augmenting on xy and y′x′. So it will construct P_u P'_u.

Consider the beginning of the k^{th} iteration of repeat loop (A) in which the k^{th} vertex u ScanQ is selected, and all adjacent vertices v are taken in turn. Use induction on k. Suppose that at the beginning of this iteration, the ScanQ contains all s-reachable vertices of the mirror network M constructed so far, that M contains a valid su-path P_u for all u ScanQ, that P_u is given by $P_x xy P_{yu}$, where xy is the switch-edge of u, and that PullFlow(s, u, δ) will construct P_u P'_u. This is certainly true initially, when k=1, and the above comments indicate that it will continue to be true up to the point when a non-trivial blossom is detected. At that point we have an edge uv, where u C(u), v C(v), C(u)≠C(v), and v′ is s-reachable. When uv and v′u′ are added to M, blossoms C(u) and C(v) must be merged into a new blossom. By Theorem 2.12, the base w of the new blossom lies on $P_u \cap P_{v'}$. It is constructed by merging together the blossoms and trivial blossoms whose bases lie on the ww′–path $Q=P_{wu}uvP'_{w'}$. This is exactly what MakeBlossom(u,v,b_u,b_v) does. The vertices in these blossoms are already known to be s-reachable, and a switch-edge and valid path is available for each. The vertices of the trivial blossoms are on the paths $P'_{v'}$ and P'_u (see Lemma 2.11). They are marked s-reachable and placed on the ScanQ, and a switch-edge is defined for each. The switch-edge is defined as uv for vertices z on $P'_{v'}$. A valid sz-path is then given by $P_u uv P_{vz}$. The switch-edge is defined as v′u′ for vertices z on P'_u. A valid sz-path is then given by $P_{v'}v'u'P_{u'z}$. So at the beginning of iteration k+1 of the repeat loop, the ScanQ will contain all s-reachable vertices of the new mirror network M. M will contain a valid sz-path P_z to every s-reachable vertex z, and the switch-edge will be correctly defined for each vertex when it is placed on the ScanQ. We still must show that PullFlow(s,z,δ) will construct P_z P'_z. Suppose that SwitchEdge[z]=uv. Then PullFlow(s,z,δ) calls PullFlow(s,u,δ) and PullFlow(z′,v′,δ) recursively. It also augments on both uv and v′u′. The first call effectively reduces the situation to the previous iteration k, before the switch-edges uv and v′u′ were added to M, so it constructs P_u P'_u. As noted above, z $P'_{v'}$, since SwitchEdge[z]=uv. $P'_{v'}$ is the complement of the valid sv′-path $P_{v'}$, and z′ $P_{v'}$. Since PullFlow(s,v′,δ) constructs $P_{v'}$ $P'_{v'}$, we conclude that PullFlow(z′,v′,δ) will construct that portion of $P_{v'}$ $P'_{v'}$ belonging to P_z P'_z. So if SwitchEdge[z]=uv, PullFlow(s,z,δ) will construct P_z P'_z. If SwitchEdge[z]=v′u′ the situation is similar. Thus the result holds on iteration k+1. By induction, these properties hold for all iterations.

3.2 Theorem. The BNS finds a valid augmenting path in N, if one exists.

Proof. Initially M is built as a breadth-first tree T and its complement T′, beginning with T={s}. The BNS considers in turn *all* vertices v of N adjacent to *all* s-reachable vertices u of M. If rescap(uv)>0 and v ScanQ, or v ScanQ with v′ s-reachable and C(u)≠C(v), then uv and v′u′ are added to M. Either T and T′ are extended, or else a blossom is created. The s-reachable vertices are the vertices of T and the blossoms taken together. The blossoms are the self-complementary connected components of s-reachable vertices of M. Theorem 2.12 describes how to construct a blossom when the edge uv is added, and this is what MakeBlossom(u,v,b_u,b_v) does. All vertices of T and of the blossoms are placed on the ScanQ so that a valid path can be built via any vertex u on the ScanQ. The only valid paths that may possibly be missed are those using the edge uv, where rescap(uv)>0, v ScanQ and v′ is not s-reachable, so that the edge uv will not be added to M. Can this edge be required in a valid path? In such a case M already contains a valid sv-path P_v. Since v′ is not s-reachable, v is the base of a trivial blossom. When v comes to the head of the ScanQ, valid paths through v will be built, so that the edge uv will not be needed, except in one case. Namely, when the existing path P_v has rescap(P_v)=1, when the use of some edge of P'_v would make an invalid path, and the invalid path can be avoided if the edge uv were to be used instead in an sv-path. In order to use edges of P'_v, the vertex v′ must have first become s-reachable, so that v and v′ will have become part of a blossom. But then v′ will come to the head of the ScanQ in the normal course of events, and the edge v′u′ will be discovered, with rescap(v′u′)>0, with u′ ScanQ, where (u′)′=u is s-reachable, and C(u′)≠C(v′). At this point MakeBlossom(v′,u′,b_v,b_u) will be called, and the edges uv and v′u′ will be added to M. Thus the algorithm will eventually find uv and v′u′, so that no valid paths will be missed. Therefore every s–reachable vertex of N will eventually be found. If N contains a valid augmenting path, t will be found s-reachable, and PullFlow(s,t,δ) will augment the flow on that path.

It follows from the Max-Balanced-Flow-Min-Balanced-Cut Theorem (1.6), that the BNS can be used to construct a maximum balanced flow in N. When the flow is maximum, the ScanQ will contain the set S of all s-reachable vertices of N. K=[S,$\bar{S}$] will then be a minimum balanced edge-cut. In the language of [12,13], this is an *f-barrier* in N, since it prevents the addition of more flow to the network.

A Note on Implementing the Algorithm.

A few questions arise on how to implement the algorithm, what data structures are convenient, how to store the switch-edges, how to tell if a vertex v is s-reachable, how to find v′ from v, etc. One effective method of doing these is indicated here. The balanced network N is stored as an array of adjacency lists. Each list contains the out-edges and in-edges of a vertex, as well as the capacity of the edge and the flow on it. An edge uv will appear in the adjacency list of u, and of v, because it must be accessible from both endpoints. The flow on uv equals the flow on v′u′, so the same flow must appear in the adjacency lists of u, v, u′, and v′. The most convenient way of doing this is to store a *pointer* to the flow in each record. All four records point to the *same* flow f(uv).

```
AdjRecord = record
  AdjPt: integer  { adjacent vertex }
  Cap:  integer   { capacity of the edge }
  Flow: ^integer { pointer to the flow }
  NextNode: ^AdjRecord
end
```

The direction of each edge must also be indicated. This can be done by storing another field. Another way is to use Cap>0 to indicate an out-edge and Cap<0 to indicate an in-edge. It may also be convenient to store a pointer to the AdjRecord of the other endpoint of the edge, to make it easy to find. This is useful in MakeBlossom(·) when SwitchEdge[z′] is assigned as one of uv or v′u′.

When v T, the switch-edge of v is just wv, where w=PrevPt[v]. Therefore there is no real need to store both PrevPt[·] and SwitchEdge[·], so we store only the switch-edge. When the switch-edge is being followed, it is helpful to have more than just the pair of vertices xy=SwitchEdge[v]. It is convenient to have an actual pointer to vertex y in the adjacency list of vertex x. The reason is that the network, being stored as adjacency lists, will have the capacity and flow of each edge stored in the adjacency list, and these need to be easily accessible, in order to compute the residual capacity. Therefore we store two arrays

```
SwitchPt[·] : array of integer  { an array of vertices }
SwitchNode[·] : array of ^AdjRecord  { pointers into the adjacency lists }
```

Given a vertex v, we need to find v′ quickly. One way is to number the vertices $x_1, x_2, \ldots, x_n$ as 1, 2, …, n, to number s=n+1 and t=n+2, and to number the vertices $y_n, y_{n-1}, \ldots, y_1$ as n+3, n+4, …, 2n+2, respectively. Then v′ is given by

```
v′ := TwoNplus3 – v,
```

where TwoNplus3=2n+3.

When computing the residual capacity of an augmenting path P, FindPathCap(x,y,δ) first finds wz, the switch-edge of y. We need to know if z′w′ is also on P. One way to do this is for FindPathCap(·) just to mark the flow of edge wz negative. If it later encounters an edge with negative flow, it knows that either wz or z′w′ has appeared previously on one of P or P′, and so it divides the residual capacity in half. This leaves some of the edges with negative flow, but PullFlow(s,t,δ) will traverse the same edges again, and reset the flows to positive values.

We also need a way of determining whether a vertex is s-reachable, and whether it is in T or T′, or not yet in M. A convenient way of doing this is to store an array and 4 constants

Tree[v]: array { which tree vertex v is in }
NotinM = 0; Tree2 = 1; sReachable = 2; Tree1 = 3

Then Tree[v]=Tree1 means that v is in T, so v is s-reachable. Tree[v]=Tree2 means that v is in T′, and is not s-reachable. Tree[v]=sReachable means that v is in T′ and in a blossom, so v is s-reachable. Thus Tree[v]≥sReachable is a simple test indicating that v is s-reachable. The same array tells us which of T and T′ v is in, and whether v is on the ScanQ (namely, Tree[v]≠NotinM).

The BNS as written above is a procedure which finds one valid augmenting path, augments, and then returns. It begins by initiallizing several arrays. In practice it is considerably better to nest this inside another loop, to initiallize the arrays *once*, and at the end of each iteration just re-initiallize the arrays for those vertices on the ScanQ. Many of the iterations will only look at a small number of vertices of the network before finding an augmenting path. Because of this, the actual complexity of the algorithm will be better than the formula proved in the next paragraph.

Complexity.

With the data structures as descibed above, the BNS is fairly efficient to program. Let $n = |X| = |Y|$, so that N has 2n+2 vertices in total. Let m be the number of edges of N. A breadth-first search takes O(m) steps, since every v adjacent to every u is taken in turn. If there were no blossoms, the complexity of the search would be O(m). When MakeBlossom(u,v,b_u,b_v) is called, the paths Pu and Pv are constructed. The maximum length of these paths is the depth of T, and the maximum depth of T is n, since it is always matched by T′. Each time MakeBlossom(·) is called, at least two blossoms are merged, so that the next time it is called the paths will be shorter. So the maximum amount of work done by MakeBlossom(·) is at most $O(n^2)$ before an augmenting path is found. There is also the work done by FindBase(u) in computing the base of the blossoms. Using the algorithm of Gabow and Tarjan [5], this can be done in O(n) steps per call to MakeBlossom(·). In summary, the BNS takes at most $O(m+n^2)$ steps to find an augmenting path, and at most O(n) steps to augment. In actual fact it will likely find and augment on several paths in this time. The number of augmenting paths found will depend on the maximum flow in N. In the worst case, the flow will be augmented by only 1 for each path found. If the value of the max-flow is K, the complexity will then be at most $O(Kn^2)$. In the case when the BNS is used to find an f-factor in a graph with n vertices, the number of edges in the f-factor will be ≤m, giving a total complexity of $O(m^2+mn^2)=O(n^4)$. If the BNS is used to find a k-factor, where k is constant, the complexity is at most $O(knm+kn^3)=O(n^3)$. It seems likely that the general complexity could be improved to $O(n^3)$ or $O(n^{2.5})$, independent of the value of the max-flow, by using an auxiliary network of all valid shortest augmenting paths, as this is a technique that works for standard flow theory (see [2,9]).

The complexity of $O(knm+kn^3)$ is comparable to existing b-matching algorithms for k-factors, except in the case of capacitated b-matchings. Sometimes it is better. If Tutte's method [6, 10, 13] is used to transform a b-matching problem in G with v vertices and e edges, to a max-matching problem in a related graph H, then the number of vertices of H is $V = 4e-\Sigma_u b(u)$. The number of edges is $E = \Sigma_u deg(u)[deg(u)-b(u)] + e$. We can assume that deg(u)–b(u)≥1 for each u. Then E ≥ 3e, although E could be as large as O(ve). Suppose that we want to find a 2-factor, for example. Then each b(u)=2, so that V=4e–2v. The augmenting path algorithm in H will take O(VE) steps. If we take E≥3e this reduces to $O(3e(4e-2v))=O(e^2)=O(v^4)$. In a graph for which E=O(ve), the complexity works out to $O(ve(4e-2v))=O(ve^2)=O(v^5)$. The BNS can solve this same problem by first constructing N with n=2v+2 vertices and m=2e+2v edges. The BNS has complexity at most $O(2nm+2n^3)$ for finding a 2-factor, which reduces to $O((2v+2)(2e+2v)) = O(ev) = O(v^3)$, which is substantially faster than transforming G into H and finding a perfect matching.

For capacitated b-matchings, a method is needed of making the complexity of the BNS independent of the value of the max-flow. In his survey paper [10], Schrijver sets up the capacitated b-matching problem as a linear program, and uses the ellipsoid method for polynomial solvability. It seems reasonable to expect that the BNS can be modified to compute a max-flow, independent of its value. Aside from the issue of complexity, balanced networks have a theoretical interest of their own.

The BNS finds a max-flow in a network N having an involutionary symmetry, such that the flow also

respects the symmetry. If N has more symmetry as well, would it also be possible to find a flow respecting this further symmetry? These are interesting questions for further research.

We finish this section by summarizing several suggestions for futher research.

1. The complexity estimate for the time to find an augmenting path is $O(n^2)$. Can this be improved?

2. Can the algorithm be improved by augmenting simultaneously on a maximal set of shortest valid augmenting paths? Is it possible to construct a set of shortest valid augmenting paths efficiently?

3. Find a method of making the number of iterations independent of the value of the maximum balanced flow.

Summary.

Balanced networks can be used to solve a number of graph problems using flow theory, including maximum matchings in general graphs, the factor problem, b-matchings, and capacitated b–matchings in general graphs, etc. The Balanced Network Search will solve these problems, using the techniques of flow theory. When a saturating flow does not exist, a minimum edge-cut will be found, corresponding to an f-barrier in the theory of f-factors. A flow-carrying balanced network contains structures called blossoms. These are defined as the connected components of a residual sub-network. They are a natural generalization of Edmonds' blossoms used in matching theory, but are not based on odd cycles. When used to find a k-factor in a graph with v vertices, the complexity is $O(v^3)$.

References

1. A. Aho, J. Hopcroft and J. Ullman, *The Design and Analysis of Computer Algorithms,*Addison Wesley, Reading, Massachusetts, 1974.
2. R. Ahuja, T. Magnanti, and J. Orlin, "Network Flows", Sloan W.P. #2059–88, Alfred P. Sloane School of Management, Massachusetts Institue of Technology, 1989.
3. J.A. Bondy and U.S.R. Murty, *Graph Theory with Applications,*American Elsevier Publishing Co., New York, 1976.
4. Jack Edmonds, Paths, trees, and flowers, Canadian J. Maths. 17, 1965, pp. 449-467.
5. Harold N. Gabow and Robert Endre Tarjan, A linear-time algorithm for a special case of disjoint set union, J.C.S.S. 30, (1985), pp. 209-221.
6. M. Gondran and M. Minoux, *Graphs and Algorithms,* John Wiley & Sons, New York, 1979.
7. William Kocay and Doug Stone, Balanced network flows, Bulletin of the Institute of Combinatorial Mathematics and its Applications 7 (1993), 17-32.
8. L. Lovasz and M.D. Plummer, *Matching Theory*, Annals of Discrete Mathematics 29, North-Holland, Amsterdam, 1986.
9. C. Papadimitriou and K. Stieglitz, *Combinatorial Optimization,* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1982.
10. A. Schrijver, Min-max results in combinatorial optimization, in *Mathematical Programming — the State of the Art,*Ed. Grötschel, Korte, Springer Verlag, Berlin, 1983.
11. W.T. Tutte, The factorization of linear graphs, J. London Math. Soc. 22, 1947, 107-111, reprinted in *Selected Papers of W.T. Tutte,*Ed. D. McCarthy and R.G. Stanton, Charles Babbage Research Centre, Canada, 1979.
12. W.T. Tutte, The factors of graphs, Can. J. Math., 4, 1952, pp. 314-328, reprinted in *Selected Papers of W.T. Tutte,*Ed. D. McCarthy and R.G. Stanton, Charles Babbage Research Centre, Canada, 1979.
13. W.T. Tutte, A short proof of the factor theorem for finite graphs, Can. J. Math., 6, 1954, pp. 347-352, reprinted in *Selected Papers of W.T. Tutte,*Ed. D. McCarthy and R.G. Stanton, Charles Babbage Research Centre, Canada, 1979.