European Journal of Operational Research 53 (1991) 257–278
North-Holland

Invited Review

The maximum flow problem: A max-preflow approach

Giuseppe Mazzoni, Stefano Pallottino and Maria Grazia Scutellà

Dipartimento di Informatica, Università di Pisa, Corso Italia 40, I-56125 Pisa, Italy

Received February 1991

Abstract: In the first part of the paper a general maximum flow procedure, which finds a maximum preflow and converts it into a maximum flow, is defined using a non-standard presentation of the maximum flow problem, which is viewed as a particular case of the maximum preflow one. This procedure enables several significant max-flow algorithms to be derived by instantiation, including the ones based on Goldberg's approach and the so-called 'distance directed' algorithms. Moreover, with this procedure a new max-flow algorithm can be defined, parametric with respect to the bound k on the length of the flow push paths. This procedure represents a generalization both of Goldberg's approach and of the 'distance directed' algorithm DD1.

In the second part, some interesting results from a wide computational experimentation on max-flow algorithms are presented.

Keywords: Maximum flow, maximum preflow, distance function, algorithm, experimentation

1. Introduction

In this work we consider the maximum flow problem, one of the basic combinatorial optimization problems.

Several algorithms have been proposed in literature for its solution. In fact, although the problem is simple in its formulation, it is very important in practice, and quite often its solution is needed in order to solve other more complex problems, like the minimum cost circulation problem and the parametric maximum flow problem.

Although most of the algorithms presented in literature look rather different from one another, quite often they share a common structure. For example, a large class of algorithms is characterized by the fact that the value of the current flow is increased at each iteration by adding additional flow along augmenting paths, until no augmenting path is found, so obtaining a maximum flow. Another important class contains algorithms which find flow relaxations at each step, the so-called preflows, and gradually convert these preflows into a maximum flow, using particular labels associated with the vertices of the graph in order to orient the pushing of the flow.

As a consequence, some of the maximum flow algorithms can be 'unified' in a few procedures which are parametric compared to the data structures used, and possibly compared to some of their operations, as suggested in several surveys on the subject (Tarjan, 1983; Ahuja, Magnanti and Orlin, 1989; Scutellà, 1990a).

In this paper, we try to provide a non-standard presentation of the maximum flow problem by viewing it as a particular case of the maximum

preflow one, and using this presentation we suggest a general maximum flow procedure, from which several significant max-flow algorithms can be derived.

The paper consists of three parts. In the first part (Section 2), we present the maximum preflow problem, by reporting the classical Ford and Fulkerson's theory of network flows (1962) rewritten in terms of preflows, and we show how to derive from it the main results on maximum flows. In the second part (Sections 3 and 4), we propose a general maximum flow procedure which first finds a maximum preflow, and then converts it into a maximum flow, and we show that many significant max-flow algorithms can be derived from it by instantiation, together with some new ones. Finally, in the third part (Section 5) we present some results from a wide computational experimentation on the algorithms described in Mazzoni (1990).

2. The max-preflow problem

Let $G=(V, E)$ be a digraph with vertex set V of size n and edge set E of size m (we assume for simplicity that no pair of vertices of V is connected by more than one edge of E). Let s and t be two distinguished vertices of G, the source and the sink respectively, and c be a capacity function that assigns a real positive number $c(v, w)$ to each edge (v, w).

A *preflow* on G is a real-valued function f on vertex pairs with the following three properties (for convenience, let us define $c(v, w)=0$ if $(v, w) \notin E$):

(i) $f(v, w) = -f(w, v)\ \forall v, w \in V$ (skew symmetry);

(ii) $f(v, w) \leq c(v, w)\ \forall v, w \in V$ (capacity constraint);

(iii) $\Sigma_w f(w, v) \geq 0\ \forall v \in V, v \neq s$ (preflow conservation).

Let $\Delta_f(v)=\Sigma_w f(w, v)$ be the *excess* of v, i.e. the net flow into v. Note that $\Delta_f(v)$ may be positive for a vertex v; in this case v is said to be *unbalanced*, and *balanced* otherwise (i.e. $\Delta_f(v)=0$). In the following, the source s will be considered unbalanced.

The *value* $|f|$ of a preflow f is the net flow into the sink, i.e. $|f|=\Delta_f(t)$. The *maximum preflow problem* is to determine a preflow of maximum value.

A *flow* on G is a particular preflow, in which each vertex other than s and t is balanced; namely, a flow satisfies the following restriction of the preflow conservation constraint:

(iii′) $\Sigma_w f(w, v)=0\ \forall v \in V, v \neq s, t$ (flow conservation).

The *maximum flow problem* is to determine a flow of maximum value. Clearly, this is a particular case of the maximum preflow problem.

As described below, the classical theory of network flows developed by Ford and Fulkerson (1962) can be rewritten in terms of preflows with only minor changes.

A key concept is *cut*. A *cut* on a graph G is a partition (X, X') of the vertex set V such that X contains s and X' contains t. The *capacity* of (X, X') is $c(X, X')=\Sigma_{v \in X, w \in X'} c(v, w)$. A cut of minimum capacity is called a *minimum cut*.

Give a preflow f and a cut (X, X'), the *flow across the cut* is

$$f(X, X')=\sum_{v \in X,\ w \in X'} f(v, w).$$

More generally, by $f(X, Y)$ we shall denote the sum of the flows on the edges with tail belonging to X and head belonging to Y, where X and Y are two disjoint non-empty subsets of V:

$$f(X, Y)=\sum_{v \in X,\ w \in Y} f(v, w).$$

Lemma 1. *Let (X, X') and (Y, Y') be two cuts on a graph G such that $X \subseteq Y$. Then $f(X, X') \geq f(Y, Y')$ for any preflow f.*

Proof. Let $Z=Y/X$. Obviously, if $Z=\emptyset$, then

$$f(X, X')=f(Y, Y').$$

Otherwise, since

$$f(X, X')=f(X, Y')+f(X, Z)$$

and

$$f(Y, Y')=f(X, Y')+f(Z, Y'),$$

then

$$f(X, X')-f(Y, Y')=f(X, Z)-f(Z, Y').$$

By the skew symmetry property, $f(Z, Y') = -f(Y', Z)$. So

$$f(X, X') - f(Y, Y') = f(X, Z) + f(Y', Z) = \Delta_f(Z),$$

where $\Delta_f(Z) = \sum_{v \in Z} \Delta_f(v)$ denotes the sum of the net flows into the vertices of Z.

Since, by definition of a preflow, $\Delta_f(v) \geq 0$ for each $v \in Z$, it follows that $f(X, X') \geq f(Y, Y')$. □

Corollary 1. *If (X, X') and (Y, Y') are two cuts such that $X \subseteq Y$, and f is a preflow such that $\Delta_f(v) > 0$ for some vertex $v \in Z = Y/X$, then $f(X, X') > f(Y, Y')$.*

Corollary 2. *For any preflow f and any cut (X, X'), the flow across the cut is bounded from below by the net flow into the sink and from above by the net flow out of the source, that is*

$$|f| = \Delta_f(t) \leq f(X, X') \leq -\Delta_f(s).$$

Corollary 3.

$$f(X, X') = |f| + \sum_{v \in X' \setminus \{t\}} \Delta_f(v)$$

for any preflow f and any cut (X, X').

The result follows by imposing $(Y, Y') = (V \setminus \{t\}, \{t\})$ in the proof of Lemma 1.

Lemma 2. *Let f be a preflow, and (X, X') and (Y, Y') be two cuts such that $X \subset Y$. Then $f(X, X') = f(Y, Y')$ if and only if*

$$\Delta_f(v) = 0 \quad \forall v \in Z = Y/X.$$

Proof. By Lemma 1,

$$f(X, X') - f(Y, Y') = \Delta_f(Z) = \sum_{v \in Z} \Delta_f(v).$$

So, since f is a preflow, $f(X, X') = f(Y, Y')$ implies

$$\Delta_f(v) = 0 \quad \forall v \in Z,$$

and vice versa. □

Property 1. *Given a preflow f, the flow across any cut (X, X') can not exceed the capacity of the cut (i.e. $f(X, X') \leq c(X, X')$).*

If the equality in Property 1 holds, then (X, X') is said to be a *saturated cut* for f.

By Property 1, the value of a maximum preflow is no greater than the capacity of a minimum cut. The max-preflow min-cut theorem states that these two values are equal. In order to present the theorem, we need to introduce some new notations.

Let f be a preflow on a graph G. The *residual capacity* of G is the function on vertex pairs given by $r_f(v, w) = c(v, w) - f(v, w)$.

The *residual graph*, R_f, is the graph with vertex set V, source s, sink t, and an edge (v, w) of capacity $r_f(v, w)$ for every pair v, w such that $r_f(v, w) > 0$ ((v, w) is called a *residual edge*).

Given a pair of vertices i and j, an *augmenting path* from i to j is a path P from i to j in R_f. The *residual capacity* of P, denoted by $r_f(P)$, is the minimum value of $r_f(v, w)$ for (v, w) an edge of P.

Theorem 1 (*max-preflow min-cut*). *Let G be a graph and f be a preflow on G. f is a maximum preflow if and only if, for each unbalanced vertex v, no augmenting path exists from v to t.*

Proof. ($\Leftarrow$) Let us consider the cut (X, X') such that X is the set of the vertices which are reachable from each unbalanced vertex v in the residual graph R_f, and $X' = V/X$ (clearly, $t \in X'$).

If $v \in X$ and $w \in X'$, the edge (v, w) does not belong to R_f (i.e. $f(v, w) = c(v, w)$). So:

$$f(X, X') = \sum_{v \in X, w \in X'} f(v, w) = \sum_{v \in X, w \in X'} c(v, w) = c(X, X').$$

Since each vertex in X' is balanced, by Corollary 3

$$f(X, X') = c(X, X') = |f|.$$

So, by Corollary 2, f is a maximum preflow.

($\Rightarrow$) Let us suppose that, for an unbalanced vertex v, there is an augmenting path P from v to t. Then we can increase the current preflow value by pushing flow along P, i.e. we can increase the value of f by any positive amount δ_P up to $r_f(P)$. That is not possible, since f is a maximum preflow. □

Corollary 4. *f is a maximum preflow if and only if there is a (saturated) cut (X, X') such that $c(X, X') = |f|$.*

Corollary 5. *f is a maximum preflow if and only if, for a saturated cut (X, X'),*

$$\Delta_f(v) = 0 \quad \forall v \in X', \quad v \neq t.$$

If the preflow f is a flow, i.e. $\Delta_f(v) = 0\ \forall v \in V$, $v \neq s$, t, then the results just described state that

- $|f| = \Delta_f(t) = f(X, X') = f(Y, Y') = -\Delta_f(s)$, for any pair of cuts (X, X') and (Y, Y') (Lemmas 1 and 2, and related Corollaries);
- $f(X, X') \leq c(X, X')$, for any cut (X, X') (Property 1);
- f is maximum iff there is no augmenting path from s to t (Theorem 1, i.e. the max-flow min-cut theorem);
- f is maximum iff there is a cut (X, X') such that $c(X, X') = |f|$ (Corollary 4);
- f is maximum iff there is saturated cut (Corollary 5).

That is, we obtain some of the well-known results on the maximum flows. Moreover, by Corollary 5 one can easily derive the following relationships between maximum flows and maximum preflows (Goldberg, 1985):

(i) the value of a maximum flow is equal to the value of any maximum preflow;

(ii) given a maximum preflow, a maximum flow can simply be obtained by 'sending back' to the source the excess of every unbalanced vertex v other than t; note that $v \in X$, where (X, X') is the minimum cut associated with the maximum preflow.

In the following section, we will describe a general max-flow procedure which first finds a maximum preflow, and then (if necessary) converts it into a maximum flow along the lines traced in (ii).

3. A maximum flow procedure

Given a graph G, the following procedure, MAX_FLOW, first finds a maximum preflow f on G by means of the procedure MAX_PREFLOW, and then converts f into a maximum flow by calling SEND_BACK.

Procedure MAX_FLOW(G, f):
begin
  MAX_PREFLOW(G, f);
  if f is not a flow
  then SEND_BACK(G, f)
end.

Let us first consider how to find a maximum preflow.

The max-preflow min-cut theorem suggests that a maximum preflow can be found by iterative improvements, namely by selecting an unbalanced vertex v, and pushing some preflow amount along an augmenting path Q from v to t, until no augmenting path of this kind exists.

Instead of pushing preflow to the sink t, a more general approach is to push a preflow amount from v to a possibly internal vertex w of Q, i.e. along a subpath

$$P = \{(i_0 = v, i_1), (i_1, i_2), \ldots, (i_{k-1}, i_k = w)\}$$

of Q. Our procedure MAX_PREFLOW follows such an approach.

In order to push preflow along P, one of the following two alternative strategies can be used.

The first strategy, called PATH_PUSH, consists in pushing along P a preflow amount $\delta_f(P) = \min\{\Delta_f(v), r_f(P)\}$. Note that PATH_PUSH decreases the excess of v of $\delta_f(P)$, increases the excess of w of the same amount, and does not change the excess of any internal vertex of P.

The second strategy, called EDGE_BY_EDGE, pushes along each edge (i_h, i_{h+1}) of P, for $h = 0, 1, \ldots, k-1$, a preflow amount $\delta_f(i_h, i_{h+1})$ which is equal to the minimum between the excess of i_h and the residual capacity of (i_h, i_{h+1}),

$$\delta_f(v, i_1) = \min\{\Delta_f(v), r_f(v, i_1)\},$$

$$\delta_f(i_h, i_{h+1}) = \min\{\Delta_f(i_h) + \delta_f(i_{h-1}, i_h), r_f(i_h, i_{h+1})\},$$

$$h = 1, \ldots, k-1.$$

Note that, in the latter case, the excess of any vertex of P may change. Clearly,

$$\delta_f(i_h, i_{h+1}) \geq \delta_f(P) \quad \text{for } h = 0, \ldots, k-1.$$

In both cases, a choice with guarantees a polynomial number of iterations is to select always the shortest (in the number of edges) augmenting

paths, as suggested by Edmonds and Karp (1972). In order to find such paths, either a subgraph of the residual graph containing the shortest augmenting paths can be constructed, or some measures can be used to estimate the length of the paths.

MAX_PREFLOW follows the second approach, that is, at each step, it looks for one of the shortest augmenting paths by using an approximate length measure, the *valid distance function* proposed by Goldberg (1985), in order to estimate the length of the augmenting paths.

Given a preflow f on a graph G, a *distance function* d is a function from the set of the vertices of G to the non-negative integers; a distance function d is *valid* if it satisfies the following two conditions:

(1) $d(t) = 0$;

(2) $d(i) \leq d(j) + 1$ for every edge (i, j) of G with a positive residual capacity (i.e. $r_f(i, j) > 0$).

It is easy to prove by induction on the distance from t in the current residual graph that $d(v)$ is a lower bound on the length of the shortest augmenting paths from v to t for every vertex v. Clearly, $d(v) \geq n$ means that no augmenting path exists from v to t.

If $d(v)$ is equal to the length of the shortest augmenting paths from v to t for every vertex v, then the distance function d is said to be *exact*; otherwise it is called an *approximate* distance function.

Using a valid distance function d, at each iteration MAX_PREFLOW selects an unbalanced vertex v such that $d(v) < n$ (since otherwise no augmenting path from v to t exists), and finds a residual path P of *candidate edges* from v to a vertex w, where a residual edge (i, j) is called candidate for d if $d(i) = d(j) + 1$ (in the following P will be referred to as a *candidate path*).

By definition of valid distance function we can note that, if d is exact, then P is a subpath of a shortest augmenting path from v to t. In this case MAX_PREFLOW first sends a preflow amount along P, and then finds the new exact distance function in $O(m)$ time, by means of breadth-first search on G starting from t, or by applying a reoptimization technique starting from the current d.

Otherwise, when d is an approximate distance function, the candidate path P is from v to a vertex w which is only 'estimated' closer to the sink t, i.e. P does not belong necessarily to a shortest augmenting path. In this case, if MAX_PREFLOW realizes that no preflow can be pushed from v to t through w, it may not send any preflow along P. Nevertheless, as proved in Goldberg (1987), if d is updated by means of the following rule, a maximum preflow is still obtained in a polynomial number of iterations: let $i \neq t$ be a vertex such that $d(i) \leq d(j)$ for every residual edge (i, j); then increase $d(i)$ to the value $d_i = \min\{d(j) + 1 \mid (i, j) \text{ is a residual edge}\}$.

As shown in Goldberg (1985), this type of updating technique maintains d as a valid distance function.

Now we will present MAX_PREFLOW in more detail.

Procedure MAX_PREFLOW(G, f).
begin
  INITIALIZE(f, d, Q, k);
  repeat
    SELECT(Q, v)
    UPDATE_PREFLOW(v, k, w, f, d, Q);
    UPDATE_DISTANCE(d, w, Q)
  until $Q = \emptyset$
end.

INITIALIZE finds an initial preflow f on G and a valid distance function d, and then sets

$$Q := \{v : v \neq t,\ \Delta_f(v) > 0 \text{ and } d(v) < n\},$$

i.e. it inserts into Q every unbalanced vertex v for which an augmenting path from v to t might exist; in addition, the operation initializes an integer $k \geq 1$ which is an upper bound on the length of the candidate paths constructed in order to find a maximum preflow.

SELECT(Q, v) selects a vertex v from Q; the selection rule depends on the implementation of Q.

UPDATE_PREFLOW(v, k, w, f, d, Q) looks for a candidate path P of maximum length $\leq k$ from v to a vertex w; different search techniques can be used.

If $|P| = k$ or $w = t$, then a preflow amount is pushed along the path, using either the PATH_PUSH strategy, or the EDGE_BY_EDGE approach, previously described. In the latter case, f can be updated during the construction of P. After the pushing of the preflow, if v is balanced, it is removed from Q; in addition, every new unbal-

anced vertex i of P is inserted into Q, and every vertex i of P in Q which is balanced after the pushing is removed from Q, by means of operations INSERT(Q, i) and DELETE(Q, i) respectively.

If $|P| < k$ ($|P|$ may be 0, i.e. $w = v$) and $w \neq t$, then no candidate edge outgoing from w exists (i.e. $d(w) < d_w = \min\{d(i) + 1 | (w, i)$ is a residual edge$\}$). In this case, w can be considered temporarily 'blocked', since no candidate path outgoing from it exists until $d(w)$ increases to the value d_w. This is why UPDATE_PREFLOW may choose to avoid any preflow pushing along P. In any case, if $w \in Q$, UDATE_PREFLOW deletes w from Q: Q is thus the set of all the unbalanced and 'unblocked' vertices of the graph from which residual paths toward the sink may exist.

Note. Whenever UPDATE_PREFLOW changes $f(i, j)$ for a vertex pair (i, j), it must change $f(j, i)$ by a corresponding amount in order to maintain the skew symmetry property. We shall not refer to this explicitly below.

Finally, let us consider the procedure UPDATE_DISTANCE(d, w, Q). The operation updates the current distance function d if $w \neq t$ is 'blocked' (i.e. $w \notin Q$). The updating must be performed in such a way that the constraints of the valid distance functions be satisfied.

In general, the following two strategies are used:

(a) if d is an approximate distance function, it is updated immediately after blocking w, by increasing $d(w)$ to the value d_w previously defined (SINGLE_RELABEL strategy). Clearly, this updating makes w 'unblocked'.

(b) if d is an exact distance function, it is updated only when there is no candidate vertex for pushing preflow towards the sink, i.e. $Q = \emptyset$. Note that d is only partially exact until the next relabel updating (PARTIAL_EXACT_LABEL strategy). The new exact distance labels are computed either by means of a breadth-first search on G starting from t, or by applying some reoptimization techniques starting from the current d.

However, if d is approximate, intermediate strategies between (a) and (b) can be used. An example might be periodically to update d, making it exact.

In all cases, Q is suitably updated by inserting into it every unbalanced and 'unblocked' vertex i such that $d(i) < n$.

In order to complete the description of MAX_FLOW, let us briefly consider the procedure SEND_BACK. By the behaviour previously described, MAX_PREFLOW terminates with a maximum preflow f (thus, every vertex $v \neq t$ with $d(v) < n$ is balanced) and a minimum cut (X, X'), where $X = \{v : d(v) \geq n\}$. Then, if $\Delta_f(v) = 0$ for each $v \in X \setminus \{s\}$, f is a maximum flow. Otherwise, SEND_BACK considers the subgraph $G(X)$ of G induced by X, and converts f into a maximum flow by selecting each unbalanced vertex $v \in X$ and sending its excess back to the source along the shortest augmenting paths of $G(X)$. This operation can be performed in $O(mn)$ time. For more details, see Goldberg (1987), Derigs and Meier (1989).

4. Instances of MAX_FLOW

In this section we will show that some of the most significant max-flow algorithms proposed in literature are particular cases of the MAX_FLOW procedure described in Section 3. In addition, a new max-flow algorithm will be derived.

Since some of the instance maintain a flow at each step, and in all cases a different kind of valid distance function can be used (either approximate or exact), for more clarity the instances of MAX_FLOW will be classified into:

(1) max-flow algorithms maintaining preflows and approximate distance functions;

(2) max-flow algorithms maintaining preflows and exact distance functions;

(3) max-flow algorithms maintaining flows and approximate distance functions;

(4) max-flow algorithms maintaining flows and exact distance functions.

We will restrict our attention to the instances of MAX_PREFLOW.

It is immediately noticeable that, if we want to maintain a flow at each step, the PATH_PUSH strategy needs to be used in UPDATE_PREFLOW, and that, if an exact distance function is required, a strategy which computes the exact distance labels has to be chosen in UPDATE_DISTANCE. In the other cases, in general the EDGE_BY_EDGE strategy and the SINGLE_RELABEL approach are used for pushing the preflow and for the distance function updating, respectively. However, different choices

can be made in MAX_PREFLOW, as will be shown below.

Clearly, different instantiations of set Q (and of the related operations SELECT, INSERT and DELETE), of parameter k and of the operation INITIALIZE can determine the development of different max-flow algorithms within every class.

4.1. Preflows and approximate distance functions

Let k be an integer value ≥ 1. Using k, a new algorithm, called K_PREFLOW, can be derived from MAX_PREFLOW. The algorithm starts with a preflow f and a valid distance function d. Then it finds a maximum preflow by pushing a preflow along candidate paths of maximum length $\leq k$ by the EDGE_BY_EDGE strategy. When necessary, d is updated by means of the SINGLE_RELABEL rule.

In general, the initial preflow used by K_PREFLOW is either the zero flow, or the *source preflow*, i.e. the preflow that saturates every edge outgoing from s and is zero on all the other edges. Clearly, in the latter case the initial valid distance function d is such that $d(s) = n$, since no augmenting path from s to t exists initially.

In more detail, if the *source preflow* is used in the initialization phase, the K_PREFLOW instantiations of MAX_PREFLOW are

```
Procedure INITIALIZE (f, d, Q, k).
begin
  f(s, i) := c(s, i), f(i, s) := −c(s, i)
    ∀(s, i) ∈ E;
    f(i, j) := 0 otherwise;
  Q := {i:(s, i) ∈ E};
  INIT_DISTANCE(d)
end.
```

In INITIALIZE, INIT_DISTANCE(d) can be performed in different ways: if an exact distance function is preferred, a breadth-first search starting from t is needed: otherwise, one possibility is to set $d(s) := n$, $d(t) := 0$ and $d(i) := 1$ for each $i \neq t, s$. In all cases, INITIALIZE runs in $O(m)$ time.

```
Procedure UPDATE_PREFLOW(v, k, w, f, d, Q).
begin
  P := ∅; w := v; endpath := false;
  repeat
    SEARCH_CANDIDATE_EDGE((w, j));
    if (w, j) ≠ nil
    then
      begin
        EDGE_BY_EDGE((w, j), f, Q);
        P := P ∪ {(w, j)}; w := j;
        if w = t
        then endpath := true
      end
    else
      begin
        endpath := true;
        if w ∈ Q
        then DELETE (Q, w)
      end
  until | P | = k or endpath
end.
```

In UPDATE_PREFLOW, EDGE_BY_EDGE((w, j), f, Q) first pushes the preflow along the candidate edge (w, j) using the EDGE_BY_EDGE strategy, and then updates Q using a suitable sequence of INSERT and DELETE.

```
Procedure UPDATE_DISTANCE(d, w, Q).
begin
  if w ≠ t and w ∉ Q
  then
    begin
      d(w) := d_w; *single_relabel strategy*
      if d(w) < n
      then INSERT(Q, w)
    end
end.
```

Note that the K_PREFLOW algorithm depends on the implementation of the set Q and on the upper bound k.

By choosing specific realizations of set Q (and, consequently, of SELECT, INSERT and DELETE), different implementations of K_PREFLOW can be obtained, which differ from each other in the strategy used for selecting unbalanced and 'unblocked' vertices of the graph.

First of all, the following two opposite strategies can be used in order to implement Q: either a vertex v is repeatedly selected until it becomes balanced or it needs to be relabelled, or the vertex selection is independent of such conditions.

In both cases, typical implementations of Q are

- Q is a stack, i.e. LIFO strategy (K_PREFLOW_STACK): by this strategy, each insertion operation (both of a newly unbalanced vertex and of a newly 'unblocked' vertex) and each selection

operation is performed at the same end of Q, i.e. its *head*;

– Q is a queue, i.e. FIFO strategy (K_PREFLOW_QUEUE): in this case, each insertion operation is performed at the *tail* of Q, whereas each selection is made at the head; as a consequence, since each unbalanced and 'blocked' vertex made 'unblocked' by UPDATE_DISTANCE is inserted at the tail of Q, it will be processed again by UPDATE_PREFLOW after all the other vertices currently in Q;

– Q is a deque, i.e. *mixed* LIFO–FIFO selection rule (K_PREFLOW_DEQUE): such a rule uses the following composite policy for inserting the vertices in Q: each vertex which is made unbalanced by UPDATE_PREFLOW is inserted in Q at the tail, whereas each unbalanced and 'blocked' vertex which is made "unblocked' by UPDATE_DISTANCE is inserted in Q at the head;

– Q is a priority queue, i.e. highest label selection rule (K_PREFLOW_PRQUE): this rule selects an unbalanced vertex with the highest distance label; the insertion policy depends on the type of priority queue chosen in order to implement Q (unordered, ordered or partially ordered).

If the PATH_PUSH strategy is used in UPDATE_PREFLOW instead of the EDGE_BY_EDGE one, a variant of K_PREFLOW, called K_PREFLOW_PATHPUSH, can be obtained. In such a variant UPDATE_PREFLOW looks for a candidate path P of maximal length $\leq k$ outgoing from the selected vertex v, and pushes flow along it by means of the PATH_PUSH strategy only if $|P| = k$ or the end vertex of P is the sink t.

```
Procedure UPDATE_PREFLOW(v, k, w, f, d, Q).
begin
  P := ∅; w := v; endpath := false;
  repeat
    SEARCH_CANDIDATE_EDGE((w, j));
    if (w, j) ≠ nil
    then
      begin P := P ∪ {(w, j)}; w := j
      end
    else
      begin endpath := true;
        if w ∈ Q
        then DELETE(Q, w)
      end
  until |P| = k or endpath;
  if w = t or |P| = k
  then PATH_PUSH(P, f)
end.
```

If a vertex v is selected until it becomes balanced or it is relabelled, then, when the path P found in UPDATE_PREFLOW has $|P| < k$ and it ends in a vertex $w \neq t$, the next path-search operation can start from the subpath P' connecting v to the predecessor of w in P, instead of starting from v. That can be efficiently implemented by maintaining the current path P outgoing from v, and updating it by removing the last edge from it (backtrack step). Similarly, after the PATH_PUSH operation, the next pathsearch operation can start from the unsaturated path fragment of P (such a path fragment can be found during the flow pushing).

By disregarding the implementation of set Q, if $k = 1$ and the initial *source preflow* are chosen in the K_PREFLOW approach, then the well known *Goldberg's* algorithm is obtained (Goldberg, 1985). To be more precise, 1_PREFLOW corresponds to the first phase of Goldberg's algorithm, which finds a maximum preflow f by pushing preflow along one candidate edge at a time, and SEND_BACK corresponds to its second phase, which converts f into a maximum flow. Note that 1_PREFLOW can equivalently start from the *zero flow*; in fact, after a finite number of iterations on the unbalanced vertex s, the *source preflow* is obtained.

As proved in Goldberg (1985), 1_PREFLOW returns a maximum preflow in $O(n^2m)$ time, independently of the implementation of the set Q. Goldberg's time complexity analysis can be applied to the K_PREFLOW algorithm as well. In fact, the upper bound on the number of relabelling operations is the same in both algorithms (i.e. $O(n^2)$). Moreover, since every preflow pushing operation in K_PREFLOW performs $O(k)$ consecutive preflow pushing operations of 1_PREFLOW, the overall number of pushing operations along single edges of the graph performed by K_PREFLOW is the same as in 1_PREFLOW (i.e. $O(n^2m)$). So, this algorithm, too, finds a maximum preflow in $O(n^2m)$ time.

Note that 1_PREFLOW is also an instance of K_PREFLOW_PATHPUSH, since on paths of length

1 the PATH_PUSH strategy and the EDGE_BY_EDGE one coincide.

Several algorithms, described in literature as specializations of Goldberg's, can be viewed as instantiations of 1-PREFLOW by suitably implementing set Q. These algorithms, which improve Goldberg's time complexity, are

- 1_PREFLOW_STACK (Derigs and Meier, 1989), which selects a vertex v until it is balanced or its label reaches the value n, and returns a maximum preflow in $O(n^3)$ time;
- 1_PREFLOW_QUEUE (Goldberg and Tarjan, 1988), which selects a vertex v until it is balanced or relabelled, runs in $O(n^3)$ time;
- 1_PREFLOW_DEQUE (Goldberg and Tarjan, 1988), also running in $O(n^3)$ time; note that when an unbalanced vertex v is selected from Q, a sequence of UPDATE_PREFLOW and UPDATE_DISTANCE is applied to it until it eventually becomes either balanced or its label reaches the value n;
- 1_PREFLOW_PRQUE (Cheriyan and Maheshwari, 1989), whose time complexity is $O(m^{1/2} n^2)$.

Other efficient implementations of Goldberg's algorithm have been proposed in literature. Some of these use the RELABEL_GLOBAL strategy suggested in (Derigs and Meier, 1989) and, independently, in (Mazzoni, 1990).

Given a preflow f and a valid distance function d, a number $z \in \{1, 2, \ldots, n-2\}$ is called *gap* if the following properties hold:

(1) $d(v) \neq z \ \forall v \in V$.

(2) $z < d(v) < n$ for some $v \in V$.

When a gap z is found, Derigs and Meier's implementation updates the current distance function d by setting $d(v) := n \ \forall v \in V$ such that $d(v) > z$. In fact it is easy to show that, if d is a valid distance function, then no augmenting path exists from any vertex v such that $d(v) > z$ to the sink t. Note that this implementation, which we call K_PREFLOW_GLOBAL, is not an instance of 1_PREFLOW, but can be derived from it by suitable specifying UPDATE_DISTANCE in MAX_PREFLOW.

Mazzoni (1990) proposed using a mixed relabelling strategy: his implementation, K_PREFLOW_MIXED, computes the exact distance function each time a gap is discovered (and, in any case, every n distance updating). This is because Goldberg and Tarjan (1988) proved that finding the exact distance function every n distance updating does not increase the time complexity of the algorithm. A variant of Mazzoni's implementation, with the same time complexity, finds the exact distance function each time a gap is discovered and, in any case, every $\tilde{n}$ distance updatings, where $\tilde{n}$ is the current number of vertices having a distance label $< n$. The latter implementation will be referred to as K_PREFLOW_TILDE. Obviously, also these kinds of distance updating can be performed by UPDATE-DISTANCE in our approach.

Finally, other interesting implementations of Goldberg's algorithm can be derived from 1_PREFLOW. They are:

- the *single phase* algorithm (Goldberg, 1985), which merges the two phases of Goldberg's; in *single phase*, the set Q contains all the unbalanced and 'unblocked' vertices of the graph. As a consequence, a vertex v selected from Q may have $d(v) > n$; the flow excess of v is pushed toward t if $d(v) < n$ or toward s if $d(v) \geq n$ (the single phase implementation can be obtained from 1_PREFLOW by suitably implementing UPDATE_DISTANCE);
- Goldberg and Tarjan's algorithm (1988), which improves Goldberg's time complexity to $O(nm \log(n^2/m))$ by expediting the pushing of the preflow using a special data structure, the Linking and Cutting Trees structure (Sleator and Tarjan, 1983);
- Ahuja and Orlin's algorithm (1989), which uses a 'scaling' technique in Goldberg's approach and runs in $O(nm + n^2 \log U)$ time in the case of integer edge capacities, where U is an upper bound on the capacities of the graph;
- Ahuja, Orlin and Tarjan's algorithm (1989), which uses the Linking and Cutting Trees data structure in Ahuja and Orlin's 'scaling' approach; the time complexity of the algorithm is $O(nm \log[2 + (n \log U)/(m \log\log U)])$.

4.2. *Preflows and exact distance functions*

As a straightforward exemplification of this kind of max-flow algorithm we will consider EXACT_K_PREFLOW, which uses an exact distance function instead of an approximate one in the K_PREFLOW approach.

EXACT_K_PREFLOW is derived from K_PREFLOW by using the PARTIAL_EXACT_LABEL strategy instead of the SINGLE_RELABEL one. As a

consequence, UPDATE_DISTANCE updates the label $d(w)$ of any 'blocked' vertex w only when there is no candidate vertex for pushing preflow towards the sink, i.e. $Q = \emptyset$. When this happens, a breadth-first search is applied starting from t in order to compute the new exact distance function, by setting to n the label of any vertex unreachable from t. Clearly, more efficient reoptimization techniques can be used. These updating operations are performed by EXACT_DISTANCE(d, Q), together with the updating of Q.

Procedure UPDATE_DISTANCE(d, w, Q).
begin
  if $Q = \emptyset$
  then EXACT_DISTANCE(d, Q)
end.

It is easy to prove that the time complexity of EXACT_K_PREFLOW is the same as the K_PREFLOW one, that is $O(n^2m)$ time.

Clearly, all the implementation choices of k, of Q and the initial preflow made for K_PREFLOW can be applied to EXACT_K_PREFLOW as well. In particular, by choosing $k = 1$ and the *source preflow* we obtain EXACT_1_PREFLOW, which can be viewed as a variant of Goldberg's algorithm using an exact distance function instead of an approximate one. On the other hand, if $k = n$ is chosen, then EXACT_N_PREFLOW is obtained, which pushes preflow along candidate paths of length $\leq n$ by the EDGE_BY_EDGE strategy.

Note that if the PARTIAL_EXACT_LABEL strategy is used in K_PREFLOW_PATHPUSH instead of in K_PREFLOW, then a variant of EXACT_K_PREFLOW, called EXACT_K_PREFLOW_PATHPUSH, is obtained. By imposing $k = n$ and choosing the *zero flow* as the initial preflow, a particular algorithm, EXACT_N_PREFLOW_PATHPUSH, can be derived, which pushes preflow from the source to the sink using the PATH_PUSH technique, thus finding a flow at each step. This kind of algorithm will be analysed in more detail in Section 4.4, where the procedures based on flows and exact distance functions will be treated in more depth.

4.3. Flows and approximate distance functions

der to obtain a flow at each step, several
ts have to be imposed in MAX_PREFLOW.
ch candidate path used by the procedure to push preflow must have the source s as its origin and the sink t as its destination, and the PATH_PUSH strategy needs to be adopted. It follows that only the source s can be unbalanced (i.e. $Q = \{s\}$ at each iteration), and no limitation on the path length can be imposed (e.g. $k = n$ can be chosen in the initialization phase).

MAX_PREFLOW operations can consequently be simplified. Firstly, each explicit handling of the set Q and each check on the upper bound k can be avoided. And then, when the path P found in UPDATE_PREFLOW ends in a vertex $w \neq t$, i.e. no candidate edge (w, j) exists, the next path-search operation can start from the subpath P' connecting s to the predecessor of w in P, instead of starting from s, as described for K_PREFLOW_PATHPUSH.

The resulting procedure, N_FLOW, starts with a feasible flow (e.g. the *zero flow*), and at each step looks for a candidate path P from the source s to the sink t starting from P'. If t is reached, N_FLOW saturates P using of the PATH_PUSH strategy; otherwise the new starting candidate path is obtained by applying the *backtrack step*, i.e. by removing the last edge from P.

Due to the properties of the approximate distance functions, each candidate path from s to t found by the procedure is one of the shortest augmenting paths. So N_FLOW returns a maximum flow by saturating shortest augmenting paths.

If the *zero flow* and the exact distance function are chosen in the initialization step, then a particular case of N_FLOW is obtained:

Procedure N_FLOW(G, f).
begin
  INITIALIZE(f, d);
  repeat
    UPDATE_PREFLOW(s, w, f, d);
    UPDATE_DISTANCE(d, w)
  until $d(s) \geq n$
end.

Procedure INITIALIZE(f, d).
begin
  $f(i, j) := 0 \quad \forall i, j \in V$;
  $P := \emptyset$;
  end_P := s;
  INIT_EXACT_DISTANCE(d)
end.

In INITIALIZE, INIT_EXACT_DISTANCE(d) finds the current exact distance function, in $O(m)$ time, and end_P represents the end vertex of P.

```
Procedure UPDATE_PREFLOW(s, w, f, d):
begin
  w := end_P;
  endpath := false;
repeat
    SEARCH_CANDIDATE_EDGE((w, j));
    if (w, j) ≠ nil
    then
      begin
        P := P ∪ {(w, j)};
        w := j
      end
    else endpath := true
  until endpath;
  if w = t
  then PATH_PUSH (P, f)
  else
    begin
      P := p \ {(i, w)}
        * i is the predecessor of w in P *;
      end_P := i
    end
end.
```

In UPDATE_PREFLOW, if $w \neq t$, i.e. w is 'blocked', no pushing operation is performed; otherwise, PATH_PUSH(P, f) pushes flow along P using the PATH_PUSH strategy, and 'empties' P (i.e. $P := \emptyset$, end_P := s).

In the latter case, a more efficient strategy would be to find the unsaturated path fragment of P outgoing from s (this can be performed during the pushing of the flow, by suitably updating P and end_P), and to use it as the new starting candidate path instead of the empty path. We will call this second kind of strategy sfap (*s*ave the *f*ragment *a*fter the *p*ushing), and the first one epap (*e*mpty the *p*ath *a*fter the *p*ushing).

If $w \neq t$, UPDATE_DISTANCE updates the label $d(w)$ by means of the SINGLE_RELABEL strategy.

```
Procedure UPDATE_DISTANCE(d, w).
begin
  if w ≠ t
  then d(w) := d_w
    *single_relabel strategy*
end.
```

N_FLOW runs in $O(n^2m)$ time independently of the feasible flow and the approximate distance function chosen in the initialization phase, and of the strategy used in UPDATE_PREFLOW in order to update the starting candidate path.

The specialization of N_FLOW starting from the *zero flow* and the exact distance function, and using the epap strategy in UPDATE_PREFLOW, is the well-known max-flow algorithm DD1 (Orlin and Ahuja, 1987). In our notation, this specialization will be called N_FLOW_EPAP.

A more efficient version of DD1, N_FLOW_SFAP, can immediately be obtained by using the sfap strategy in UPDATE_PREFLOW instead of the epap one.

Another rather intelligent technique, called PARTIAL_FLOW_PUSH, consists in using the EDGE_BY_EDGE strategy in UPDATE_PREFLOW instead of the PATH_PUSH one, i.e. pushing a maximum flow along each single edge of the path P during its construction. Clearly, as a consequence of this kind of pushing, some unbalanced vertices may be created in P. So, in order to obtain a maximum flow, the following operations on the current path P are performed.

If P reaches the sink t, the path search is restarted from the unbalanced vertex of P which is the 'closest' to t. Otherwise, i.e. the end-vertex w of P is 'blocked' and a *backtrack* step needs to be performed, the excess of w is 'moved back' along the edge (i, w) of P (by UPDATE_PREFLOW in our approach), and the current end vertex of P is updated to i. This flow adjustment has to be viewed as a correction of an attempt of pushing. In fact, when the adjustment is performed, (w, i) is not a candidate edge and therefore no legal flow pushing along it should be possible in our approach.

Note that the procedure may create unbalanced vertices only along the current candidate path P. Consequently, whenever P is empty, the current preflow is in fact a flow. In particular, when the procedure terminates, a maximum flow is obtained. This is why the variant of N_FLOW using the PARTIAL_FLOW_PUSH has been described in this section, even though it could be seen as a particular procedure based on preflows and approximate distance functions.

In conclusion, the particular instance of N_FLOW which represents DD1 can be viewed as the specialization of K_PREFLOW_PATHPUSH start-

ing with $k = n$ and with the *zero flow*. Since the 'opposite' specialization of K_PREFLOW_PATHPUSH is 1_PREFLOW, K_PREFLOW_PATHPUSH represents a 'link' between Goldberg's approach and Ahuja and Orlin's one.

4.4. Flows and exact distance functions

This class of max-flow algorithms can be described as a specialization of the N_FLOW procedure, which uses an exact rather than an approximate distance function. The resulting procedure, EXACT_N_FLOW, starts with a feasible flow, for example the *zero flow*, and at each step looks for a candidate path from the source s to the sink t by using an exact distance function d, and saturates this path using the PATH_PUSH strategy. When necessary, the new exact distance function is found by means of the PARTIAL_EXACT_LABEL strategy.

The exact distance function d used by the procedure allows us to define implicitly the set of all the shortest augmenting paths from s to t (of length $k = d(s)$). So, EXACT_N_FLOW returns a maximum flow by saturating one of the shortest augmenting paths at each step. Note that, by definition of PARTIAL_EXACT_LABEL, the updating of d is performed only when the source s is 'blocked', i.e. no candidate path of length k exists from s to t. Consequently this updating corresponds to finding the new set of shortest augmenting paths when the current set is saturated.

In EXACT_N_FLOW, the instances of INITIALIZE and of UPDATE_PREFLOW are the same as described for N_FLOW. The instance of UPDATE_DISTANCE is the one described in Section 4.2, without the explicit handling of set Q.

Procedure UPDATE_DISTANCE(d, w).
begin
  if $w = s$
  then EXACT_DISTANCE(d)
end.

Like N_FLOW, EXACT_N_FLOW runs in $O(n^2m)$ time independently of the feasible flow chosen in the initialization phase, and of the strategy used in UPDATE_PREFLOW in order to update the starting candidate path (either sfap or epap).

The specialization of EXACT_N_FLOW starting with the *zero flow* coincides with EXACT_N_PREFLOW_PATHPUSH, introduced in Section 4.2. If, in addition, the epap strategy is used in UPDATE_PREFLOW, then the classical Dinic's max-flow algorithm is obtained (Dinic, 1970). In fact, the current level graph used in Dinic's algorithm is implicitly defined by the exact distance function d, and finding the new current level graph corresponds to the computation of the new exact distance function, performed by UPDATE_DISTANCE. In our terminology, Dinic's instance of MAX_PREFLOW is EXACT_N_FLOW_EPAP.

Using particular data structures in order to implement the input graph, other maxflow algorithms, described in literature as implementations of Dinic's, can be viewed as instantiations of EXACT_N_FLOW_EPAP. This set of algorithms includes Galil and Naamad's algorithm (1980), which uses the 2–3 *tree* data structure (Aho and Hopcroft and Ullman, 1974) and runs in $O(nm \log^2 n)$ time, and Sleator and Tarjan's (1983), which makes use of the data structure Linking and Cutting Trees and finds a maximum flow in $O(nm \log n)$ time. Similar implementations can be found in Shiloach (1978) and in Sleator (1980).

To conclude our survey on the algorithms using flows and exact distance functions, let us consider the variant of EXACT_N_FLOW which starts with the zero flow and maintains an exact distance function at each step. From the properties of the exact distance functions, a particular tree, called the *candidate path tree*, can be associated with each exact distance function d, which is a tree of candidate edges rooted at the sink t whose leaves are all the vertices with distance label $< n$. The unique path from every vertex of a candidate path tree to the sink is one of the shortest augmenting paths; so the unique tree path outgoing from s is one of the shortest augmenting paths from s to t. The variant of EXACT_N_FLOW maintains a candidate path tree T at each step, and uses the shortest augmenting path defined by T in order to perform the pushing flow operations.

The first candidate path tree T is constructed in INITIALIZE using a breadth-first search on the input graph G starting from t.

Procedure INITIALIZE(f, d, T).
begin
  $f(i, j) := 0, \quad \forall i, j \in V$;
  INIT_EXACT_DISTANCE(d, T)
end.

UPDATE_PREFLOW selects the unique path P from s to t in T, and pushes flow along it by means of the PATH_PUSH strategy.

Procedure UPDATE_PREFLOW(s, f, d, T).
begin
SELECT(P, T, s);
PATH_PUSH(P, f)
end.

If the candidate path tree is computed from scratch at each step, by performing a breadth-first search in UPDATE_DISTANCE starting from t, then Edmonds and Karp's algorithm (1972) is obtained. This particular instance can thus be called EXACT_N_FLOW_BFS.

Procedure UPDATE_DISTANCE(d, T).
begin
EXACT_DISTANCE(d, T)
end.

Since the time complexity of UPDATE_DISTANCE is $O(m)$, and the number of iterations is bounded by mn, Edmonds and Karp's algorithm runs in $O(nm^2)$ time.

If a reoptimization technique is used in UPDATE_DISTANCE in order to update the candidate path tree T, then a more efficient algorithm, EXACT_N_FLOW_REOPT, can be obtained.

Let us consider the set of the edges of T which have been saturated by the PATH_PUSH operation. These edges are no longer candidates, and so they have to be removed from T in order to update the candidate path tree; a forest F of subtrees of T is thus obtained. By successively inserting new candidate edges, and deleting edges which are no longer candidates, the forest F can be converted into a new candidate path three T in the following way.

Let V_F be the set of the roots of F other than t. For each $v \in V_F$, first v is removed from V_F, and then a new candidate edge (v, w) is determined, by increasing the label $d(v)$ to the value d_v if necessary. If $d(v) < n$, v is connected to w by inserting (v, w) into the forest; otherwise v is deleted from F, since it cannot belong to any candidate path tree. In any case, if $d(v)$ is increased, each edge (u, v) of the forest is deleted from F (since it is not candidate anymore), and the new root u is inserted into V_F.

The updating operations are performed until V_F is empty. When this happens, the forest has been transformed into a single tree, the new candidate path tree. The updating of T can be described more formally as follows:

Procedure UPDATE_DISTANCE(d, T).
begin
E' = edge-set of T;
V' = vertex-set of T;
$V_F := \emptyset$;
for each saturated edge (u, v) **do**
begin $E' := E' \setminus \{(u, v)\}$;
$V_F := V_F \cup \{u\}$
end;

Table 1

Approach	MAX_PREFLOW features					Algorithm name	Authors
	k	Relabel	Preflow push	epap/sfap	Q		
Goldberg	1	single	any		stack	1_PR_STACK	Derigs and Meier
	1	single	any		queue	1_PR_QUEUE	Goldberg and Tarjan
	1	single	any		deque	1_PR_DEQUE	Goldberg and Tarjan
	1	single	any		prque	1_PR_PRQUE	Cheri. and Mahesw.
	1	global	any		any	1_PR_GL	Derigs and Meier
Distance	n	single	PATH_PUSH	epap		N_FL_EPAP	Orlin and Ahuja DD1
directed	n	single	PATH_PUSH	sfap		N_FL_SFAP	–
	n	EXACT_REOPT.	PATH_PUSH			E_N_FL_REOPT	Orlin and Ahuja DD2
shortest augm.	n	PART._EXACT	PATH_PUSH	epap		E_N_FL_EPAP	Dinic
paths	n	EXACT_BFS	PATH_PUSH			E_N_FL_BFS	Edmonds and Karp
K_PREFLOW	any	any	any	any	any	K_PR	

legenda for algorithm name; PR = preflow; FL = flow; GL = global; E = Exact

```
while V_F ≠ ∅ do
begin
   SELECT(V_F, v); DELETE(V_F, v);
   w := argmin{d(j) | (v, j) is a residual edge};
      * d_v = d(w) + 1 *
   if d(w) ≥ d(v) then
      begin
         d(v) := d(w) + 1;
         for each (u, v) ∈ E' do
         begin E' := E' \ {(u, v)};
               V_F := V_F ∪ {u}
         end
      end;
   if d(v) ≥ n
   then V' := V' \ {v}
   else E' := E' ∪ {(v, w)}
end;
T := (E', V')
end.
```

The above updating operation is an intelligent reoptimization technique for computing the current exact distance function d, suggested in Orlin and Ahuja (1987). In fact, EXACT_N_FLOW_REOPT, just described, is Orlin and Ahuja's algorithm DD2. Thanks to the reoptimization technique, the time complexity of DD2 is $O(n^2m)$.

The features of the main algorithmic approaches described in this section, and the corresponding author names, are summarized in Table 1.

5. Experimental results

This section reports some interesting results from a wide-ranging experimentation on max-flow algorithms presented in Mazzoni (1990). The experimentation first compares the main algorithms described in Section 4, and then studies the efficiency of the K_PREFLOW approach, which was introduced in Section 4.1. Our results should be viewed as a first step toward the characterization of the practical efficiency of several max-flow algorithms: a detailed analysis of the subject does not fall within the scope of this paper.

All the algorithms, implemented in Pascal language, were tested using the two graphs generators described in Section 5.1. Experiments were carried out on a VAX8560, under Ultrix Operating System and Berkeley Pascal Compiler.

In Sections 5.2 and 5.3 the more promising algorithms from the first part of the experimentation are selected for comparison, and then the more efficient instantiations of K_PREFLOW are described in Section 5.4.

5.1. Graph generators

In the max-flow experimentation, a key problem is to produce test graphs in which the minimum cut position can be 'guided' using appropriate parameters, in order to avoid unbalanced cuts (those with too many vertices in one subset of the partition). This problem is solved by using some ad-hoc graph generators, like RMFGEN and MPGEN. On the other hand, all the classical graph generators for general network flow problems failed to achieve that result, producing graphs with unbalanced cuts with high probability.

Both generators RMFGEN (Goldfarb and Grigoriadis, 1988) and MPGEN (Mazzoni, 1990) were used in our experimentation. Below, their main features will be described.

5.1.1. RMFGEN

Each graph produced by RMFGEN in our experimentation consists of b *frames*, where a frame is a squared grid graph of side a. Within each frame, a pair of neighbor vertices is connected by a pair of opposite edges. Between two consecutive frames, there are $2a^2$ edges: a^2 edges connect each vertex in the first frame to a vertex, randomly chosen, in the second one; the remaining a^2 edges link the second frame to the first one using the same rule. The random selection rule must guarantee that, for each vertex of the two frames, there is exactly one ingoing and one outgoing edge.

The resulting graph has $n = a^2b$ vertices and $m = 6a^2b - 4ab - 2a^2$ edges. Vertex 1 is chosen as the source, and vertex n as the sink.

The capacities of the edges between the frames are chosen randomly in the range [1,1000]; the edges within the frames have a capacity of $1000a^2$. This choice guarantees that the minimum cut is located between two consecutive frames.

The values of the parameters a and b in the experimentation here described are

$a = 2, 3, 4, 6, 8, 12, 16;$

$b = 2, 4, 8, 16, 32, 64.$

5.1.2. MPGEN

Each graph produced by MPGEN in our experimentation has n vertices and m edges. As for RMFGEN, vertex 1 is the source, and n is the sink.

The outdegree and the indegree of each vertex other than 1 and n is chosen randomly in the range $[1,2m/n-1]$. The indegree and the outdegree of the source and of the sink are produced differently: the indegree of 1 and the outdegree of n are 0; the outdegree of 1 and the indegree of n are chosen randomly in the range $[2,4m/n-2]$. This latter greater range is used to avoid unbalanced minimum cuts such as separating the source or the sink from the other vertices of the graph.

To guarantee that n is connected to 1, MPGEN generates the edge $(i, i+1)$ for $i = 1, 2, \ldots, n-1$. The remaining $m-n+1$ edges are randomly assigned to pairs of vertices according to their indegrees and outdegrees.

Let (X_h, X_h') be the cut with $X_h = \{1, 2, \ldots, h\}$. The capacities of the edges are chosen randomly in the range [1,2000] with two different distribution functions in such a way as to have a high probability that the capacity of any edge (i, j) with $i \leq h$ and $j > h$, i.e. the directed edges of the cut, takes a small value, and the capacity of each remaining edge takes a large value. MPGEN sets $h = \lfloor pn \rfloor$, where p is an input parameter $(0 < p < 1)$.

Our tests prove that the minimum cuts (X, X') obtained by solving the max-flow problem on the graphs generated for a given parameter p have $|X| \approx h$. That is, it is possible to prefix the cardinality of the minimum cut with high precision.

The values of the parameters n, m and p in the tests of our experimentation here described are

$n = 100, 200, 250, 400, 500;$

$m = 500, 1000, 2000, 2500, 4000, 5000, 8000;$

$p = 0.10, 0.25, 0.50, 0.75, 0.90.$

5.2. Algorithm implementation

Each algorithm in our experimentation was implemented uniformly to ensure correct comparison. That is, every 'abstract' data type used by the algorithms was implemented with the same style using the same basic data types (arrays, pointer-arrays or pointer-lists). As far as the graph implementation is concerned, lists of candidate edges were implemented to facilitate the candidate path search. In fact, following and improving Goldberg's approach, a dynamic structure of the 'possible candidate' edges outgoing from each vertex of the graph was used. This structure, which is built each time a vertex v is relabelled, links the edges incident to v which 'can be candidate', and it is scanned using a pointer to the 'current possible candidate edge'.

Since, as shown in Section 3, there are several choices for many max-flow algorithms, the initial part of our experimentation investigated which particular choices provide efficient versions. Our results are that it is better to use the sfap strategy rather that the epap. Moreover, the RELABEL_GLOBAL strategy and the techniques which find the exact distance function each time a gap is discovered (see Section 4.1.) seem very promising for classes based on approximate distance functions. In fact, the more efficient implementations from our experimentation are some instances of N_FLOW_SFAP and of EXACT_N_FLOW_SFAP, some specializations of 1_PREFLOW_GLOBAL, and, in certain cases, the variants 1_PREFLOW_MIXED and 1_PREFLOW_TILDE.

Let us briefly describe these implementations, which will be compared in the next section (the acronyms in parenthesis will appear in the running time tables).

EXACT_N_FLOW_SFAP (E_N_FL_SFAP) is a particular version of Dinic's algorithm using the sfap strategy within EXACT_N_FLOW; the implementation improves the DNSUB code of Goldfarb and Grigoriadis (1988), based on the epap approach;

N_FLOW_SFAP (N_FL_SFAP): is one of the particular versions of Orlin and Ahuja's algorithm DD1 which have been described in Section 4.3, using the sfap strategy within N_FLOW;

1_PREFLOW_STACK_GLOBAL (1_PR_STACK_GL) is a particular instantiation of 1_PREFLOW, using the LIFO selection rule and the RELABEL_GLOBAL technique; the implementation corresponds to the first phase of the GOLDNET code, which is the most efficient implementation of Derigs and Meier's experimentation (1989) on test graphs generated by NETGEN (Klingman et al., 1974);

1_PREFLOW_PRIQUE_GLOBAL (1_PR_PRQUE_GL) is another instantiation of 1_PREFLOW, using the RELABEL_GLOBAL updating and the *highest label* selection rule, where the priority queue is implemented by means of buckets; the algorithm

corresponds to the GOLDRMF code, which is the most efficient implementation of Derigs and Meier's experimentation on test graphs generated by RMFGEN;

1_PREFLOW_DEQUE_MIXED (1_PR_DEQUE_MX) is the instantiation of 1_PREFLOW which uses the LIFO–FIFO selection rule, and which computes the exact distance function whenever a gap is discovered (and, in any case, every n distance updatings);

1_PREFLOW_DEQUE_TILDE (1_PR_DEQUE_TD) is a specialization of 1_PREFLOW_DEQUE_MIXED, which finds the exact distance function whenever a gap is discovered and, in any case, every $\tilde{n}$ distance updatings, where $\tilde{n}$ is the current number of vertices with a distance label $< n$.

5.3. Computational results

All the implementations previously described were tested on the same graphs, using both RMFGEN and MPGEN. Below the results of our comparison on graphs generated by RMFGEN, and the results obtained by using MPGEN are reported.

5.3.1. Computational results on RMFGEN

In our experimentation, the graphs generated by RMFGEN were obtained by using the same pseudo-random number generator and the same seeds proposed by Goldfarb and Grigoriadis (1988). The results on some of the values of a and b used in Goldfarb and Grigoriadis' experimentation are reported in Table 2, entries denote the average CPU time in seconds on twenty-five runs.

As shown in Table 2, when b increases, the CPU times of E_N_FL_SFAP, N_FL_SFAP and 1_PR_STACK_GL grow significantly; on the other hand, the efficiency of the remaining algorithms depends on b to a minor measure.

When a increases, E_N_FL_SFAP has the worst behavior, and 1_PR_PRQUE_GL (i.e. Goldberg's approach with the *highest label* selection rule and the RELABEL_GLOBAL strategy) is the most efficient on average, in agreement with Derigs and Meyer's results (1989). For high values of a and very small values of b, 1_PR_STACK_GL shows a good behavior as well.

Note that the relative efficiency of 1_PR_PRQUE_GL compared to E_N_FL_SFAP and

Table 2

a	b	n	m	E_N_FL_SFAP	N_FL_SFAP	1_PR			
						STACK_GL	PRQUE_GL	DEQUE_MX	DEQUE_TD
2	32	128	504	0.15	0.11	0.33	0.08	0.11	0.12
2	64	256	1016	0.46	0.38	1.27	0.15	0.25	0.28
3	16	144	654	0.21	0.16	0.32	0.12	0.13	0.14
3	32	288	1326	0.62	0.45	1.00	0.24	0.26	0.27
3	64	576	2670	2.18	1.70	4.08	0.44	0.64	0.68
4	8	128	608	0.26	0.17	0.22	0.15	0.15	0.15
4	16	256	1248	0.79	0.55	0.76	0.27	0.29	0.33
4	32	512	2528	2.00	1.90	2.68	0.58	0.71	0.74
4	64	1024	5088	7.96	6.52	10.32	0.88	1.79	1.85
6	4	144	696	0.38	0.28	0.33	0.22	0.28	0.27
6	8	288	1464	1.11	0.84	0.96	0.52	0.69	0.54
6	16	576	3000	2.90	2.44	3.24	1.12	1.27	1.10
6	32	1152	6072	8.58	7.10	8.94	1.81	2.36	2.34
6	64	2304	12216	26.80	22.00	30.20	3.11	4.74	5.32
8	4	256	1280	1.06	0.74	0.73	0.55	0.69	0.55
8	8	512	2688	2.60	1.95	2.68	1.43	1.96	1.25
8	16	1024	5504	6.86	5.68	6.70	2.82	3.68	2.52
8	32	2048	11136	21.80	16.86	21.60	6.05	5.92	5.18
12	2	288	1344	1.24	0.85	0.42	0.91	0.83	0.81
12	4	576	2976	3.96	2.90	2.70	1.82	2.95	2.22
12	8	1152	6240	10.90	7.94	8.20	3.95	7.95	4.40
12	16	2304	12768	27.80	19.88	21.00	10.40	17.52	8.38
16	2	512	2432	3.30	2.20	1.05	2.40	2.08	1.89
16	4	1024	5376	9.30	6.50	5.92	4.24	7.03	5.18
16	8	2048	11264	25.60	18.40	19.30	11.20	19.14	10.72

Table 3

	$n = 500$; $m = 5000$			$n = 500$; $m = 25000$		
	$\lvert X\rvert < 5$	$\lvert X\rvert \approx 250$	$\lvert X\rvert > 495$	$\lvert X\rvert < 5$	$\lvert X\rvert \approx 250$	$\lvert X\rvert > 495$
E_N_FL_SFAP	1.35	1.63	1.72	2.78	3.72	2.85
N_FL_SFAP	0.41	0.96	1.60	1.69	3.80	3.64
1_PR_STACK_GL	0.54	1.16	1.73	1.78	3.78	4.43

1_PR_STACK_GL in Table 2 is worse than reported in Derigs and Meyer's experimentation (i.e. GOLDRMF compared to DINIC and GOLDNET). This is probably due to the fact that, in order to guarantee a uniform implementation of the algorithms, the priority queue in 1_PR_PRQUE_GL was implemented using pointer-lists in Pascal language.

Note also that our version of 1_PR_PRQUE_GL is on average worse than 1_PR_DEQUE_TD for high values of a ($a \geq 8$). That is, a periodical computation of the exact distance function may be crucial for algorithm efficiency.

5.3.2. Computational results on MPGEN

First of all, let us show that the minimum cut position influences the performance of the algorithms, thus proving that an accurate experimentation on the max-flow algorithm behavior should take such a position into account. That is shown by considering the three algorithms E_N_FL-SFAP, N_FL_SFAP and 1_PR_DEQUE_MX, instantiations of EXACT_N_FLOW, N_FLOW and 1_PREFLOW, respectively. The results (Table 3) are on two different kinds of graphs, one sparse and one more dense, and on three different minimum cut positions, i.e. toward the source ($\lvert X\rvert \approx 1$), in the middle ($\lvert X\rvert \approx \frac{1}{2}n$), and toward the sink ($\lvert X\rvert \approx n - 1$). Every entry is the average CPU time in seconds on twenty runs.

Our tests show that, on the sparse graph, the three algorithms have a comparable performance when the minimum cut is toward the sink; whereas, when the minimum cut is toward the source or at the middle, E_N_FL_SFAP is worse than the other two. A different behavior is obtained on the more dense graph; in fact the three algorithms have a comparable performance when the minimum cut is at the middle, whereas E_N_FL_SFAP is the best one when the minimum cut is toward the sink, and

Table 4

	p	$n = 100$			$n = 200$			$n = 400$		
		$m = 500$	$m = 1000$	$m = 2000$	$m = 1000$	$m = 2000$	$m = 4000$	$m = 2000$	$m = 4000$	$m = 8000$
E_N_FL_SFAP	0.25	0.08	0.12	0.25	0.20	0.25	0.51	0.58	0.63	1.27
	0.50	0.08	0.12	0.22	0.25	0.28	0.46	0.76	0.63	1.24
	0.75	0.07	0.10	0.23	0.19	0.24	0.49	0.52	0.59	1.21
N_FL_SFAP	0.25	0.06	0.09	0.17	0.12	0.20	0.21	0.23	0.43	0.82
	0.50	0.08	0.13	0.24	0.20	0.27	0.47	0.48	0.64	1.20
	0.75	0.09	0.14	0.29	0.21	0.35	0.63	0.54	0.76	1.55
1_PR_STACK_GL	0.25	0.06	0.09	0.17	0.12	0.20	0.38	0.27	0.46	0.80
	0.50	0.08	0.15	0.25	0.22	0.36	0.53	0.61	0.73	1.21
	0.75	0.11	0.19	0.32	0.33	0.47	0.70	1.02	1.04	1.66
1_PR_DEQUE_MX	0.25	0.10	0.13	0.23	0.20	0.43	0.48	0.38	1.07	1.00
	0.50	0.11	0.16	0.28	0.25	0.35	0.55	0.61	0.82	1.28
	0.75	0.14	0.19	0.36	0.32	0.50	0.74	0.87	0.95	1.80
1_PR_DEQUE_TD	0.25	0.10	0.14	0.25	0.20	0.43	0.57	0.39	1.15	1.14
	0.50	0.12	0.19	0.32	0.28	0.41	0.69	0.63	0.93	1.48
	0.75	0.14	0.22	0.40	0.32	0.54	0.89	0.72	0.97	2.10

the worst one in the opposite case (i.e. minimum cut toward the source).

Having shown the influence of the minimum cut position on the algorithm performances, we decided to study the max-flow algorithms described in Section 5.2 on three different cut positions ($p = 0.25$, $p = 0.50$, $p = 0.75$). The results are shown in Table 4, the entries denote the average CPU time in seconds on twenty-five runs. The results which relate to 1_PR_PRQUE_GL are not reported in Table 4, since the algorithm showed quite an inefficient behavior in our experimentation, in agreement with Derigs and Meier's results (1989).

In agreement with our previous results, it follows that:

- when $p = 0.25$ (i.e. minimum cut positioned near to the source), N_FL_SFAP and 1_PR_STACK_GL have the best performance on all the tested graphs;
- when $p = 0.50$ (i.e. minimum cut at the middle), usually N_FL_SFAP shows the best behavior; moreover, 1_PR_STACK_GL and E_N_FL_SFAP have a good behavior as well;
- when $p = 0.75$ (i.e. minimum cut positioned near to the sink), E_N_FL_SFAP is the most efficient algorithm; in many cases, N_FL_SFAP is very fast as well.

When the minimum cut is positioned near to the sink, E_N_FL_SFAP has a good behavior because the depth-first search used in order to visit the current level graph is particularly efficient. On the other hand, 1_PR_STACK_GL (i.e. Goldberg's approach) is not so efficient, even if the RELABEL_GLOBAL strategy is used, because of the high number of preflow pushings, forward and backward, along certain edges of the graph; for this reason 1_PR_STACK_GL is more efficient when the minimum cut position is located toward the source.

N_FL_SFAP (i.e. Ahuja and Orlin's approach using an approximate distance function) subsumes all the advantages of E_N_FL_SFAP and of 1_PR_STACK_GL, and so it generally has the best behavior on all the tested graphs, independently of the minimum cut position. In fact, it explores the graph in the same manner of E_N_FL_SFAP, uses an approximate distance function like 1_PR_STACK_GL, and avoids pointless reroutings of the preflow thanks to the PATH_PUSH strategy used along candidate paths from s to t.

In conclusion, N_FL_SFAP seems to be the most efficient algorithm on any random graph, regardless of the minimum cut position.

5.4. The case of K_PREFLOW

Since 1_PREFLOW and the instance of N_FLOW representing DD1 are very efficient on graphs of type RMFGEN and MPGEN, respectively, we decided to test the variant of K_PREFLOW introduced in Section 4.1, K_PREFLOW_PATHPUSH, which has 1_PREFLOW and N_FLOW as its particular instantiations. To be more precise, the following implementations of K_PREFLOW_PATHPUSH were analysed:

- K_PREFLOW_PATHPUSH_DEQUE_MIXED (K_PR_PP_DEQUE_MX) is the variant of K_PREFLOW_PATHPUSH which uses the LIFO–FIFO selection rule and computes the exact distance function whenever a gap is discovered; clearly, if $k = 1$, 1_PR_DEQUE_MX is obtained, and if $k = n$, N_PR_PP_DEQUE_MX is a particular version of the algorithm N_FL_SFAP;
- K_PREFLOW_PATHPUSH_DEQUE_TILDE (K_PR_PP_DEQUE_TD) is a specialization of K_PR_PP_DEQUE_MX, which finds the exact distance function whenever a gap is discovered and, in any case, every $\tilde{n}$ distance updatings; note that, if $k = 1$, 1_PR_PP_DEQUE_TD is the same as 1_PR_DEQUE_TD.

5.4.1. Computational results on RMFGEN

In order to evaluate the efficiency of K_PR_PP_DEQUE_MX and K_PR_PP_DEQUE_TD, different values of the parameter k were considered, both dependent and independent of the number of the vertices and/or the number of the edges of the graph.

As described in Mazzoni (1990), for low values of a, and for high values of a and low values of b, the values of k which provide the most efficient instantiations belong to the range [10,20]; in the other cases, the best range is [2,5]. To be more precise, $k = 5$ always showed a good behavior for the K_PR_PP_DEQUE_MX instantiation, whereas the values $k = 10$ and $k = 2$ showed the best behavior in the case of K_PR_PP_DEQUE_TD, by considering the first and the second range, respectively.

In order to study the relative efficiency of the two algorithms, we thus compared 5_PR_PP_DE-

Table 5

a	b	n	m	1_PR_DEQUE_MX	K_PR_PP_DEQUE_MX			N_FL_SFAP
					$k=1$	$k=5$	$k=n$	
2	32	128	504	0.11	0.16	0.14	0.16	0.11
2	64	256	1016	0.25	0.35	0.29	0.56	0.38
4	8	128	608	0.15	0.21	0.19	0.22	0.17
4	64	1024	5088	1.79	2.42	2.22	7.08	6.52
6	4	144	696	0.28	0.44	0.30	0.31	0.28
6	64	2304	12216	4.74	7.20	6.44	23.60	22.00
8	4	256	1280	0.69	0.96	0.69	0.82	0.74
8	32	2048	11136	5.92	8.26	6.32	17.28	16.86
12	2	288	1344	0.83	1.19	0.89	1.01	0.85
12	8	1152	6240	7.95	11.56	6.78	8.82	7.94

QUE_MX with the particular instantiations 1_PR_DEQUE_MX and N_FL_SFAP, and 2_PR_PP_DEQUE_TD and 10_PR_PP_DEQUE_TD with 1_PR_DEQUE_TD and N_FL_SFAP. The results of this comparison, on twenty runs, are reported in Tables 5 and 6, respectively.

To compare 1_PR_DEQUE_MX (1_PR_DEQUE_TD) and its corresponding version 1_PR_PP_DEQUE_MX (1_PR_PP_DEQUE_TD), and N_FL_SFAP and its corresponding version N_PR_PP_DEQUE_MX (N_PR_PP_DEQUE_TD), the average CPU times for $k=1$ and for $k=n$ are also reported in Table 5 (Table 6). An evaluation of the overhead due to the handling of the parameter k in both algorithms is thus possible.

Let us consider Table 5. In agreement with our previous results, 5_PR_PP_DEQUE_MX, i.e. the instance of K_PR_PP_DEQUE_MX obtained by imposing $k=5$, is always more efficient than 1_PR_PP_DEQUE_MX and N_PR_PP_DEQUE_MX. Moreover, 1_PR_PP_DEQUE_MX and N_PR_PP_DEQUE_MX are always less efficient than their corresponding versions, i.e. 1_PR_DEQUE_MX and N_FL_SFAP, respectively. This is due to the handling of the parameter k in the K_PREFLOW approach during the candidate path searching.

Note that, in most cases, 5_PR_PP_DEQUE_MX is less efficient than 1_PR_DEQUE_MX; in fact, 5_PR_PP_DEQUE_MX shows the best behavior only in the case $a=12$ and $b=8$. As far as the comparison between 5_PR_PP_DEQUE_MX and N_FL_SFAP is concerned, for high values of b, N_FL_SFAP is very inefficient and worse than 5_PR_PP_DEQUE_MX.

Table 6 shows that either 2_PR_PP_DEQUE_TD or 10_PR_PP_DEQUE_TD is always more efficient than 1_PR_PP_DEQUE_TD and N_PR_PP_DEQUE_TD, according to our previous results. Moreover, as observed for K_PR_PP_DEQUE_MX, 1_PR_PP_DEQUE_TD and N_PR_PP_DEQUE_TD are always less efficient than their corresponding versions 1_PR_DEQUE_TD and N_FL_SFAP.

Table 6 reveals also that 1_PR_DEQUE_TD is always better than any instance of K_PR_PP_DE-

Table 6

a	b	n	m	1_PR_DEQUE_TD	K_PR_PP_DEQUE_TD				N_FL_SFAP
					$k=1$	$k=2$	$k=10$	$k=n$	
2	32	128	504	0.12	0.17	0.17	0.15	0.18	0.11
2	64	256	1016	0.28	0.38	0.34	0.31	0.61	0.38
4	8	128	608	0.15	0.22	0.20	0.22	0.23	0.17
4	64	1024	5088	1.85	2.44	2.28	2.76	7.26	6.52
6	4	144	696	0.27	0.46	0.35	0.33	0.33	0.28
6	64	2304	12216	5.32	7.42	6.58	8.46	23.40	22.00
8	4	256	1280	0.55	0.98	0.78	0.73	0.85	0.74
8	32	2048	11136	5.18	8.20	6.26	8.16	17.48	16.86
12	2	288	1344	0.81	1.24	0.98	0.90	1.04	0.85
12	8	1152	6240	4.40	12.14	8.00	6.82	9.04	7.94

QUE_TD, and that, when N_FL_SFAP shows a good behavior (i.e. for small values of b), 10_PR_PP_DEQUE_TD is on average better than 2_PR_PP_DEQUE_TD, whereas it is dominated by 2_PR_PP_DEQUE_TD when N_FL_SFAP has a bad behavior (i.e. for high values of b).

5.4.2. Computational results on MPGEN

The results of our experimentation on MPGEN presented in Subsection 5.3.2 show that, regardless of the minimum cut position, N_FL_SFAP is on average the most efficient algorithm, and that 1_PR_DEQUE_MX is more efficient than 1_PR_DEQUE_TD. In agreement with such results, our experimentation on the K_PREFLOW approach on graphs of type MPGEN showed that K_PR_PP_DEQUE_TD is always worse than K_PR_PP_DEQUE_MX, and that the more efficient versions of K_PR_PP_DEQUE_MX are the ones based on high values of k, possibly depending on the vertex number n.

For these reasons, in Table 7 we report the results of some tests on the instances of K_PR_PP_DEQUE_MX obtained for $k = n/10$, $n/4$, $n/2$ and n (on twenty runs), together with the average CPU times of N_FL_SFAP, 1_PR_DEQUE_MX and of the instance 1_PR_PP_DEQUE_MX (in order to compare 1_PR_DEQUE_MX and its corresponding version 1_PR_PP_DEQUE_MX).

Table 7 shows that all the instances with k depending on n have a similar behavior, and that such instances are better than 1_PR_PP_DEQUE_MX and than 1_PR_DEQUE_MX in most cases. Moreover, the overhead due to the handling of the parameter k in the K_PREFLOW approach is considerable, as suggested by the comparison between N_PR_PP_DEQUE_MX and N_FL_SFAP CPU times in Table 7.

In conclusion, on graphs of type MPGEN, i.e. on very random graphs, Ahuja and Orlin's approach generally has the best behavior, independently of the minimum cut position.

On graphs of type RMFGEN, i.e. on frame composed graphs, Goldberg's approach with either the *highest* or the LIFO selection rule and with the RELABEL_GLOBAL strategy is quite efficient on average; the K_PREFLOW approach using the LIFO–FIFO selection rule and computing the exact distance function whenever a gap is discovered (and, possibly, every $\tilde{n}$ distance updatings) may be efficient as well for low values of k.

Our investigation on the K_PREFLOW approach thus suggests that a max-flow procedure which is parametric with respect to the length of the paths used for finding a maximum preflow may be interesting not only from a theoretical point of view, but also in practice. This subject should be matter of further investigation.

6. Conclusions

In this paper we have described a non-standard, and we hope interesting, presentation of the maximum flow problem, based on the concept of preflow.

Our presentation allows one to classify a large variety of max-flow algorithms by instantiation from a general max-flow procedure, called MAX_FLOW. In fact, as shown in Section 4, several significant max-flow algorithms can be derived from MAX_FLOW by specifying a few parametric

Table 7

	k	$n = 250$; $m = 2500$			$n = 500$; $m = 2500$			$n = 500$; $m = 5000$		
		$p = 0.25$	$p = 0.50$	$p = 0.75$	$p = 0.25$	$p = 0.50$	$p = 0.75$	$p = 0.25$	$p = 0.50$	$p = 0.75$
1_PR_DEQUE_MX		0.41	0.46	0.72	0.46	0.76	0.89	0.82	1.03	1.32
K_PR_PP_DEQUE_MX	1	0.56	0.60	0.85	0.72	1.11	1.30	0.99	1.36	1.76
	$n/10$	0.39	0.50	0.66	0.46	0.66	0.75	0.65	0.98	1.17
	$n/4$	0.38	0.50	0.65	0.46	0.66	0.76	0.66	0.99	1.17
	$n/2$	0.39	0.50	0.65	0.46	0.66	0.75	0.66	0.99	1.17
	n	0.39	0.51	0.66	0.45	0.65	0.76	0.66	0.98	1.18
N_FL_SFAP		0.27	0.38	0.56	0.32	0.51	0.65	0.40	0.75	1.03

components, like how to represent the set Q of the vertices from which to push preflow, how to perform the candidate path search from the vertices in Q, what kind of technique to use in order to push preflow along such paths, and what kind of valid distance updating to choose.

Note that none of the numerous network simplex algorithms oriented to max-flow computations (Fulkerson and Dantzig, 1955; Goldfarb and Grigoriadis, 1979; Grigoriadis and Hsu, 1979; Goldfarb and Hao, 1988; Goldfarb and Grigoriadis, 1988; Goldberg and Grigoriadis and Tarjan, 1988; Bertsekas and Eckstein, 1988) have been considered in this paper. In fact, only ad hoc max-flow algorithms have been analysed. Moreover, some important ad hoc algorithms, like Karzanov's (Karzanov, 1974; Tarjan, 1984), Cherkasky's (Cherkasky, 1977; Scutellà, 1990b), Galil's (1980), Shiloach and Vishkin's (1982), Gabow's (1985) and the so-called 3_INDIANS algorithm (Malhotra, Kumar and Maheswari, 1978), have not been described either, since none of them can be derived from the general procedure MAX_FLOW. In effect, with only a few minor changes in the definition of MAX_FLOW, also some of those algorithms could be considered in our framework. The version of MAX_FLOW described here was chosen because it seems to have the most significant properties, and to be suitable to a more immediate comprehension.

In addition to providing a unifying framework for many important max-flow algorithms, our general procedure suggests the definition of new max-flow procedures, like the K_PREFLOW approach introduced in Section 4. As described in Section 5, where some results from a wide experimental computation on max-flow algorithms are reported, the K_PREFLOW approach also seems to be interesting from a computational point of view, at least for some values of the parameter k. As a matter of investigation, also the study of the K_PREFLOW efficiency when k changes dynamically might be interesting in practice.

In conclusion, we would like to emphasize that not all the possible max-flow algorithms derivable from MAX_FLOW have been analysed in our work. In fact, only the approach which seemed the most promising to us, i.e. the K_PREFLOW, was considered. However, different approaches might be interesting, from a theoretical and/or practical point of view. So, we hope that the maximum flow framework introduced here may stimulate further research in the max-flow field.

References

Aho, A.V., Hopcroft, J.E., and Ullman, J.D. (1974), *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA.

Ahuja, R.K., Magnanti, T.L., and Orlin, J.B. (1989), "Network flows", in: G.C. Nemhauser, A.H.G. Rinnooy Kan and M.J. Todd (eds.), *Optimization*, Handbooks in Operations Research and Management Science, Vol. 1, North-Holland, Amsterdam, 211–369.

Ahuja, R.K. and Orlin, J.B. (1989), "A fast and simple algorithm for the maximum flow problem", *Operations Research* 37, 748–759.

Ahuja, R.K., Orlin, J.B., and Tarjan, R.E. (1989), "Improved time bounds for the maximum flow problem", *SIAM Journal on Computing* 18, 939–954.

Bertsekas, D.P., and Eckstein, J., (1988), "Dual coordinate step methods for linear network flow problems", *Mathematical Programming* 42, 203–243.

Cheriyan, J., and Maheswari, S.N. (1989), "Analysis of preflow push algorithms for maximum network flow", *SIAM Journal on Computing* 18, 1057–1086.

Cherkasky, B.V. (1977) "Algorithm of construction of maximal flow in networks with complexity of $O(V^2E^{1/2})$ operations", *Mathematical Methods of Solution of Economical Problems* 7, 117–125 (in Russian).

Derigs, U., and Meier, W. (1989), "Implementing Goldberg's max-flow-algorithm – A computational investigation", *Zeitschrif für Operations Research* 33, 383–403.

Dinic, E.A. (1970), "Algorithm for solution of a problem of maximum flow in networks with power estimation", *Soviet Mathematics Doklady* 11, 1277–1280.

Edmonds, J., and Karp, R.M. (1972), "Theoretical improvements in algorithmic efficiency for networks flow problems", *Journal of the ACM* 19, 248–264.

Ford, L.R. jr., and Fulkerson, D.R. (1962), *Flows in Networks*, Princeton University Press, Princeton, NJ.

Fulkerson, D.R., and Dantzig, G.B. (1955), "Computations of maximal flows in networks", *Naval Research Logistics Quarterly* 2, 277–283.

Gabow, H.N. (1985), "Scaling algorithms for network problems", *Journal of Computer and System Science* 31, 148–168.

Galil, Z. (1980), "An $O(V^{5/3}E^{2/3})$ algorithm for the maximal flow problem", *Acta Informatica* 14, 221–242.

Galil, Z., and Naamad, A. (1980), "An $O(EV\log^2 V)$ algorithm for the maximal flow problem", *Journal of Computer and System Science* 21, 203–217.

Goldberg, A.V. (1985), "A new max-flow algorithm", Techn. Rep. MIT/LCS/TM291, Laboratory for Computer Science, M.I.T., Cambridge, MA.

Goldberg, A.V. (1987), "Efficient graph algorithms for sequential and parallel computers", Ph.D. Thesis, M.I.T., Cambridge, MA.

Goldberg, A.V., Grigoriadis, M.D., and Tarjan, R.E. (1988), "Efficiency of the network simplex algorithm for the maximum flow problem", Techn. Rep. LCSR-TR-117, Labora-

tory of Computer Science Research, Department of Computer Science, Rutgers University, New Brunswick, NJ.

Goldberg, A.V., and Tarjan, R.E. (1988), "A new approach to the maximum flow problem", *Journal of the ACM* 35, 921–940.

Goldfarb, D., and Grigoriadis, M.D. (1979), "An efficient steepest-edge algorithm for maximum flow problems", *Tenth International Symposium on Mathematical Programming,* Montreal, Canada.

Goldfarb, D., and Grigoriadis, M.D. (1988), "A computational comparison of the Dinic and network simplex methods for maximum flow", in: B. Simeone et al. (eds.), *Fortran Codes for Network Optimization; Annals of Operation Research* 13, 83–123.

Goldfarb, D., and Hao, J. (1988), "A primal simplex algorithm that solves the maximum flow problem in at most nm pivots and $O(n^2m)$ time", Manuscript, Dept. of Industrial Engineering and Operations Research, Columbia University, New York, NY.

Grigoriadis, M.D., and Hsu, T. (1979), "The Rutgers minimum cost network flow subroutines", *SIGMAP Bulletin of the ACM* 26, 17–18.

Karzanov, A.V. (1974), "Determining the maximal flow in a network by the method of preflows", *Soviet Mathematics Doklady* 15, 434–437.

Klingman, D., Napier, A., and Stutz, J. (1974), "Netgen: a program for generating large scale capacitated assignment, transportation and minimum cost flow network problems", *Management Science* 20, 814–821.

Malhotra, V.M., Kumar, M.P., and Maheswari, S.N. (1978), "An $O(|V|^3)$ algorithm for finding maximum flows in networks", *Information Processing Letters* 7, 277–278.

Mazzoni, G. (1990), "Analisi, sviluppo e valutazione sperimentale di algoritmi per il problema di flusso massimo", Tesi di Laurea in Scienze dell'Informazione, Università di Pisa, Pisa.

Orlin, J.B., and Ahuja, R.K. (1987), "New distance-directed algorithms for maximum flow and parametric maximum flow problems", Working Paper 1908-87, Sloan School of Management, M.I.T., Cambridge, MA.

Scutellà, M.G. (1990a), "A unified algorithmic framework for max-flow computations (Toward the design of a combinatorial optimization programming environment)", Ph.D. Thesis, Dipartimento di Informatica, Università di Pisa, Pisa.

Scutellà, M.G. (1990b), "A note on Cherkasky's algorithm for the maximum flow problem", *Ricerca Operativa* 53, 65–75.

Shiloach, Y. (1978), "An $O(nI \log^2 I)$ maximum-flow algorithm", Techn. Rep. STAN-CS-78-702, Computer Science Dept., Stanford University, Stanford, CA.

Shiloach, Y., and Vishkin, U. (1982), "An $O(n^2 \log n)$ parallel max-flow algorithm", *Journal of Algorithms* 3, 128–146.

Sleator, D.D. (1980), "An $O(nm \log n)$ algorithm for the maximum network flow" Tech. Rep. STAN-CS-80-831, Computer Science Dept., Stanford University, Stanford, CA.

Sleator, D.D., and Tarjan, R.E. (1983), "A data structure for dynamic trees", *Journal of Computer and System Science* 26, 362–391.

Tarjan, R.E. (1983), *Data Structures and Network Algorithms*, SIAM, Philadelphia, PA.

Tarjan, R.E. (1984), "A simple version of Karzanov's blocking flow algorithm", *Operations Research Letters* 2, 265–268.