Available online at www.sciencedirect.com
ScienceDirect
Computers & Operations Research 35 (2008) 2694–2708
computers & operations research
www.elsevier.com/locate/cor

Efficient preflow push algorithms

R. Cerulli[a,*], M. Gentili[a], A. Iossa[b]

[a] *Dipartimento di Matematica ed Informatica, Università di Salerno, Via Ponte Don Melillo, 84084 Fisciano (SA), Italy*
[b] *Dipartimento di Statistica, Probabilità e Statistiche Applicate, Università degli Studi di Roma "La Sapienza", Piazzale A. Moro 5, 00185 Roma, Italy*

Available online 22 December 2006

Abstract

Algorithms for the maximum flow problem can be grouped into two categories: *augmenting path* algorithms [Ford LR, Fulkerson DR. Flows in networks. Princeton University Press: Princeton, NJ: 1962], and *preflow push* algorithms [Goldberg AV, Tarjan RE. A new approach to the maximum flow problem. In: Proceedings of the 18th annual ACM symposium on theory of computing, 1986; p. 136–46]. Preflow push algorithms are characterized by a drawback known as *ping pong effect*. In this paper we present a technique that allows to avoid such an effect and can be considered as an approach combining the augmenting path and preflow push methods. An extended experimentation shows the effectiveness of the proposed approach.

Keywords: Maximum flow; Ping pong effect; Preflow push

1. Introduction

The maximum flow problem is one of the most studied problems in the literature and there exist many contributions proposing different efficient algorithms to solve the problem. Algorithms for the maximum flow problem can be grouped into two main categories: *augmenting path* algorithms [1], and *preflow push* algorithms [2]. There are also contributions in the literature presenting a generic resolution approach that combines both methods [3]. These methods capture the computational efficiency of both these classical approaches, however, they could not overcome the main drawback of the preflow push algorithms known as the *ping pong effect*. Many efforts have been devoted to overcome such an effect (the gap property introduced by [4] is an example), however, they are not always effective, as it will be shown in the sequel of the paper.

We present a new generic technique, namely the *budget algorithm*, that can be considered as an approach combining the two classical augmenting path and preflow push methods. Therefore, on the one hand, we maintain the computational efficiency of the classical approaches, and, on the other hand, we overcome the ping pong effect greatly improving the computation times.

The rest of the paper is organized into three parts as follows. In the first part (Section 2), we report some well known results on network flows, we briefly describe the preflow push algorithms, and we show a bad example for the preflow

* Corresponding author. Tel.: +39 89963326; fax: +39 89963303.
E-mail addresses: raffaele@unisa.it (R. Cerulli), mgentili@unisa.it (M. Gentili), aiossa@unisa.it (A. Iossa).

0305-0548/$ - see front matter © 2007 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cor.2006.12.023

push algorithm that generates the ping pong effect. The second part (Section 3), describes our budget algorithm and shows how it overcomes the ping pong effect. In the third part of the paper (Section 4), we present the results coming from a wide experimentation to evaluate our algorithm's performance.

2. The maximum flow problem and existing approaches

In this section we recall the main definitions and properties about admissible flows and admissible preflows, for additional details we refer the reader to [5].

A network is a directed graph $G=(N, A, s, t, u)$ where N is a set of n nodes and A is a set of m arcs; s is the source node and t is the sink node; u is a capacity function that assigns to an arc $(i, j) \in A$ a nonnegative real value u_{ij}.

An *admissible flow* is a real function $X : A \longrightarrow R$ satisfying the flow conservation (1) and capacity (2) constraints:

$$\sum_{(j,i)\in\delta_G^-(i)} x_{ji} - \sum_{(i,j)\in\delta_G^+(i)} x_{ij} = 0 \quad \forall i \in N\backslash\{s,t\}, \tag{1}$$

$$0 \leqslant x_{ij} \leqslant u_{ij} \quad \forall (i,j) \in A, \tag{2}$$

where x_{ij} are the classical flow variables denoting the flow on each arc $(i, j) \in A$, $\delta_G^+(i)$ is the set of arcs outgoing from i in G, that is $\delta_G^+(i)=\{(i, j) \in A | j \in N\}$ and $\delta_G^-(i)$ is the set of arcs coming into i in G, that is $\delta_G^-(i)=\{(j, i) \in A | j \in N\}$. The value $|f|$ of a flow X is the net flow entering t, that is $|f| = \sum_{(i,t)\in\delta_G^-(t)} x_{it}$. In the *maximum flow problem*, we want to determine an admissible flow of maximum value.

A *preflow* is a real function $X : A \longrightarrow R$ that satisfies the capacity constraints (2) and the following relaxation (3) of the flow conservation constraints (1):

$$\sum_{(j,i)\in\delta_G^-(i)} x_{ji} - \sum_{(i,j)\in\delta_G^+(i)} x_{ij} \geqslant 0 \quad \forall i \in N\backslash\{s,t\}. \tag{3}$$

In the *maximum preflow problem* we want to determine a preflow of maximum value, that is, a preflow where the net flow entering the sink t is maximum.

All the algorithms solving the maximum flow problem work on the residual network $G(X)$, obtained from G once an initial admissible flow and the residual capacity of each arc is computed. Given a flow (preflow) X, the *residual capacity* r_{ij} of an arc $(i, j) \in A$ is the maximum additional flow that can be sent from node i to node j through the arcs (i, j) and (j, i). The residual capacity has two components:

- u_{ij}—x_{ij}, that represents the not used capacity of arc (i, j);
- the current flow x_{ji} on the arc (j, i) that can be reduced in order to increase the flow from node i to node j.

Therefore, we can compute the residual capacity as $r_{ij}=(u_{ij}-x_{ij})+x_{ji}$. Given a network flow $G=(N, A, s, t, u)$ and a flow (preflow) X, the *residual network* induced by X is the graph $G(X)=(N, A')$ where $A'=\{(i, j) \in N \times N | r_{ij} > 0\}$. Given a pair of vertices i, j of G, an augmenting path from i to j in G is a directed path P from i to j in $G(X)$. The residual capacity of an augmenting path P is the value $r(P) = \min\{r_{ij} | (i, j) \in P\}$. An augmenting path from i to j can be used to move at most $r(P)$ units of flow from i to j.

Any algorithm based on the augmenting path approach, (i) finds an initial admissible flow X, (ii) looks for an augmenting path P on the residual graph $G(X)$ if it exists, and (iii) determines a new admissible flow X' pushing $r(P)$ units on the arcs of the path. When it is not possible to find any augmenting path then the actual flow is optimum and the algorithm stops.

On the other hand, any algorithm based on the preflow push approach, uses the relation between the Max Flow and the Max Preflow problems. Indeed, we can see the maximum flow problem as a particular case of the maximum preflow problem, as explained next.

Given a preflow X, we define the excess of a node i, namely $e(i)$, as the value $\sum_{(j,i)\in\delta_{G(X)}^-(i)} x_{ji} - \sum_{(i,j)\in\delta_{G(X)}^+(i)} x_{ij}$. That is, the net flow of node i in the residual graph $G(X)$. A node is *active* if it has a strictly positive excess. Thus, we

can see an admissible flow as a particular preflow where no node is active, that is, the excess in each node is zero. Also, see [6] for details, the following relations between maximum flows and maximum preflows hold:

- the value of a maximum flow is equal to the value of any maximum preflow;
- given a maximum preflow X, we can obtain a maximum flow X' from X *sending back* to the source the flow of any active node (other than t).

A general procedure to find a maximum preflow, is suggested by the well known *max-flow min-cut* theorem, formulated in terms of preflow in [3]:

Theorem 1. *Let G be a network and X a preflow on G. X is a maximum preflow if and only if, for each active node v, no augmenting path exists from v to t in G.*

A generic approach combining both methods has been proposed in [3]. It finds at first a maximum preflow (the so-called first phase), and then converts the preflow in a maximum flow (the so-called second phase). In the first phase the generic algorithm, as suggested by the above theorem, selects an active node v, detects an augmenting path from v to t, and moves some amount of flow from v to t along such a path. Then, it repeats this process until no augmenting path exists from any active node to the sink. Once a maximum preflow is found, in the second phase, the generic algorithm sends to the source the excess flow of each active node along augmenting paths from active nodes to the source node. If the algorithm always selects the shortest augmenting path from the selected node v to the sink t (or to the source s) then, as suggested in [7], the running time is polynomial.

These combined methods capture the computational efficiency of both the classical approaches, however, they could not overcome the main drawback of the preflow push algorithms known as the *ping pong effect*. In the next section a more detailed description of the preflow push approach is given and an example of a class of graphs illustrating the ping pong effect is also shown.

2.1. Preflow push algorithms

The preflow push algorithms use an approximate length measure to estimate the length of the augmenting paths, proposed by Goldberg in [2]: the so-called *distance* function.

A *distance* function $d : N \to Z^+ \cup \{0\}$ is a function from the node set N to the set of not negative integers. Given a preflow X, we say a distance function is *valid* if the following conditions are satisfied:

$$d(t) = 0 \tag{4}$$

$$d(i) \leqslant d(j) + 1 \quad \forall (i, j) \in G(X). \tag{5}$$

We define $d(i)$ as the *distance* of node i, and, conditions (4) and (5) as validity conditions. It is possible to show, see [5], that a valid distance label $d(i)$ is a lower bound to the length (i.e., number of arcs) of any directed path from node i to node t in the residual network. Since the maximum length of any path from a generic node to the sink is $n - 1$, then, if $d(s) \geqslant n$ the residual network does not contain any directed path from the source to the sink. If for each $i \in N$, $d(i)$ is equal to the length of a shortest path from i to t, we say that the distance labels are *exact* (it is possible to determine the exact distance labels by a backward *breadth first search* starting from the sink node). We say that an arc (i, j) in the residual network is *admissible* if the relation $d(i) = d(j) + 1$ holds. A directed path from a node v to the node t is *admissible* if it is composed only of admissible arcs. It is not difficult to see that an admissible path from a node v to the sink node t is a shortest augmenting path from v to t.

The preflow push algorithms slightly differ from the generic algorithm described in the previous section: they send flow along arcs, rather than along paths. Moreover, we speak about the preflow push *algorithms* rather than the preflow push *algorithm*, because by specifying the node selection rule, we obtain several algorithm variants.

The preflow push algorithms maintain a preflow (3) into intermediate phases. For each preflow, we have $e(i) \geqslant 0, \ \forall i \in N \backslash \{s, t\}$. However, since there is not outgoing flow from node t, we also have $e(t) \geqslant 0$. The basic operation of the preflow push algorithms consists in selecting an active node and in trying to send the excess flow toward adjacent nodes in the residual network. Since the goal of the algorithms is to send flow toward the sink, the excess flow is sent to an adjacent node that is closer to the sink than the selected one. The algorithms measure the distance from the sink node with the

help of the distance labels; therefore, sending flow toward a node near to the sink is equivalent to sending flow along admissible arcs. If the active node, that the algorithms extract, does not have admissible arcs, then its distance label is increased in order to create at least one admissible arc. This increment comes from a rule proposed by Goldberg in [2] to maintain valid the distance labels. The algorithms terminate when the network does not have any active node. We now describe formally a generic preflow push algorithm.

Algorithm 1. Generic preflow push algorithm

Input: A network $G = (N, A, s, t, u)$
Output: A maximum flow X of G
1. $initialize(G)$
2. **while** there is an active node **do**
3. select an active node i
4. *push-relabel*(i)
5. **end while**

Procedure 2. *initialize*(G)

1. **for all** $(i, j) \in A$ **do**
2. $x_{ij} \leftarrow 0$
3. **end for**
4. Calculate the exact distance labels $d(i)$
5. **for all** $(s, j) \in \delta^+_{G(X)}(s)$ **do**
6. $x_{sj} \leftarrow u_{sj}$
7. **end for**
8. $d(s) \leftarrow n$

The *initialize* procedure performs several operations. First of all, it sets to zero the flow along each arc of the network and computes the exact distance labels; then it sends flow from the source to the adjacent nodes, in order to activate them. Afterwards, since the previous operation saturates all the arcs outgoing from s, it sets $d(s) = n$ such that the residual network does not contain any directed path from s to t.

Procedure 3. *push-relabel*(i)

1. **if** there is an admissible arc (i, j) in $\delta^+_{G(X)}(i)$ **then**
2. $\delta \leftarrow \min\{r_{ij}, e(i)\}$ {push}
3. $x_{ij} \leftarrow x_{ij} + \delta$
4. $e(i) \leftarrow e(i) - \delta$
5. $e(j) \leftarrow e(j) + \delta$
6. $r_{ij} \leftarrow r_{ij} - \delta$
7. $r_{ji} \leftarrow r_{ji} + \delta$
8. **else**
9. $d(i) \leftarrow \min\{d(j) + 1 | (i, j) \in \delta^+_{G(X)}(i)\}$ {relabel}
10. **end if**

The *push* operation sends δ units of flow from node i to node j in the residual network, decreases of δ units both $e(i)$ and r_{ij} and increases of δ units both $e(j)$ and r_{ji}. The *relabel* operation increases the distance label of a node in order to create at least one admissible arc outgoing from i.

The algorithm running time depends mainly on the number of push operations and this in turn depends on the strategy used to select the active nodes. The FIFO preflow push algorithm maintains the active nodes in a list, selects the head node, and inserts a new active node in the tail of the list. The worst-case running time of this algorithm is $O(n^3)$ [2]. The *highest label* selection rule guarantees that the number of pushes is $O(n^2\sqrt{m})$. This bound was first proved by Cheriyan and Maheshwari [8] for a particular implementation of the algorithm that uses the so-called *current edges*. Tunçel [9]

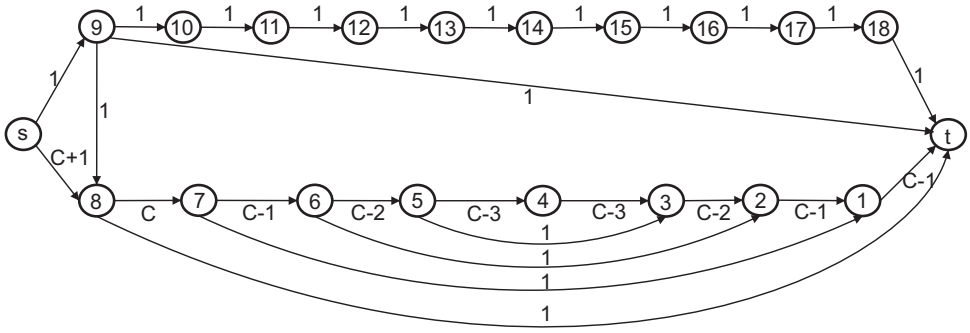

Fig. 1. The initial graph; c is an integer greater than 4.

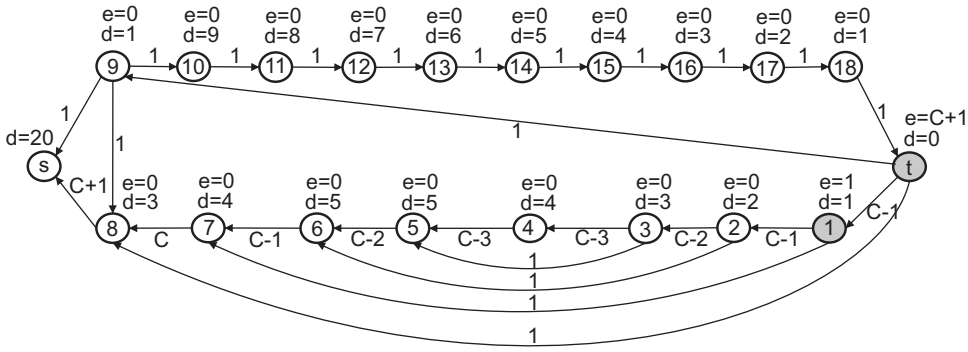

Fig. 2. The residual network after some iterations. The only active node is node 1.

gave a more general proof that applies to the standard algorithm, i.e., the algorithm does not need to follow the current edges implementation. A detailed implementation analysis and a different proof is given in [10]. A strategy improving the practical running time is the *gap* strategy, proposed in [4]. Suppose that at a certain time during the algorithm execution there exists a value $0 < k_1 < n$ such that there are no nodes with label equal to k_1, and there exist nodes with label strictly greater than k_1. Then the sink node is not reachable from any of the nodes i such that $d(i) > k_1$. Thus the label of these nodes can be increased to n. Another strategy is the *global relabeling* that consists in a periodic update of exact distance label through a backward BFS from the sink node. Goldberg and Tarjan [2] proved that, computing the exact distance labels every n relabeling operations does not change the worst-case performance of the preflow push algorithm.

2.2. *Preflow push algorithm and ping pong effect*

Let us execute the preflow push algorithm (with the highest label selection rule), on the graph in Fig. 1. After the initialization phase and several iterations, we obtain the residual network shown in Fig. 2. For each node i, there are two labels: the distance label $d(i)$ and the excess $e(i)$. The label on each arc is its residual capacity. In the residual network shown in Fig. 2, the active node with the greatest distance label is node 1, which is extracted and relabeled to 3 by the algorithm. Then the algorithm pushes 1 unit of flow from node 1 to node 2 along the arc $(1, 2)$, making active node 2, and inactive node 1. Successively, the algorithm selects node 2, relabels it to 4 and push 1 unit of flow from node 2 to 3 along arc $(2, 3)$, making node 3 active. After various push and relabel operations, the residual network shown in Fig. 3 is obtained. On the graph shown in Fig. 3, the algorithm selects node 8, relabels it to 5 and then pushes the unit of excess flow from 8 to 7. Then, it selects node 7, relabels it to 6 and pushes the only unit of flow from 7 to 6. The algorithm repeats the operations of relabel and push for each nodes along the path $\{6, 5, 4, 3, 2, 1\}$, obtaining

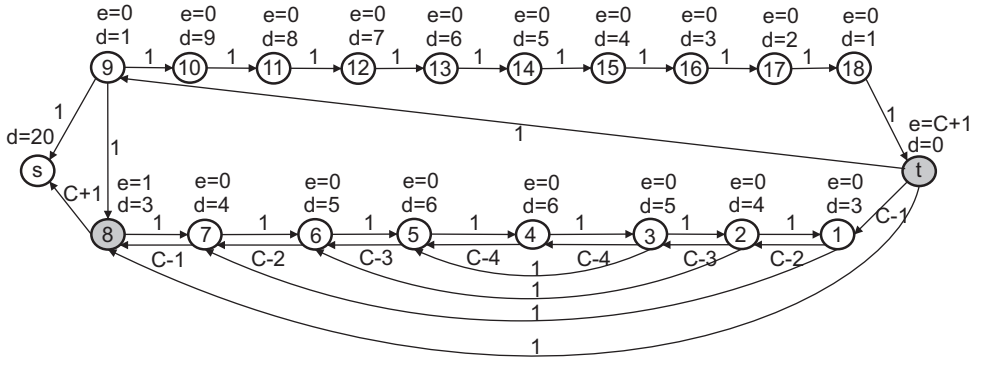

Fig. 3. The only active node is node 8.

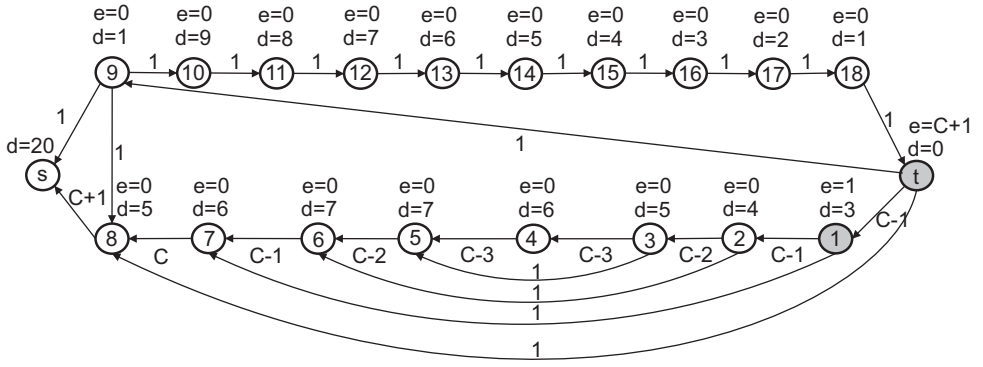

Fig. 4. Node 1 is active again.

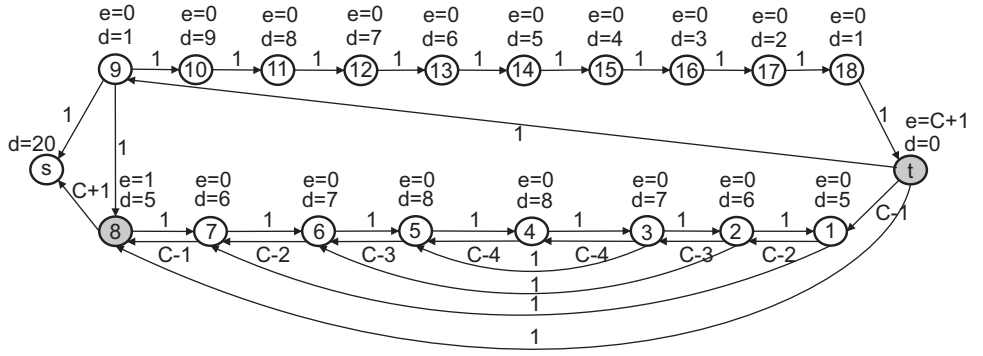

Fig. 5. Node 8 is active again.

the network in Fig. 4. Starting from the residual network shown in Fig. 4, the algorithm selects node 1, relabels it to 5 and pushes flow along arc (1, 2). Then, the algorithm selects node 2, relabels it to the value 6 and pushes one unit of flow from node 2 to node 3, making the latter active. Therefore, the flow in excess in the node 3 is moved to node 8 by several alternating operations of relabel and push on the nodes in the path {3, 4, 5, 6, 7, 8}, obtaining the network shown in Fig. 5. As we can see, the unit of excess flow of node 1 (Fig. 2), is moved to node 8 (Fig. 3) by the algorithm. Then, from node 8 the exceeding flow returns to node 1 (Fig. 4), and again the unit of flow is moved to node 8 (Fig. 5). This continuous flow exchange between nodes 1 and 8, known as *ping pong effect*, is the main drawback of the preflow

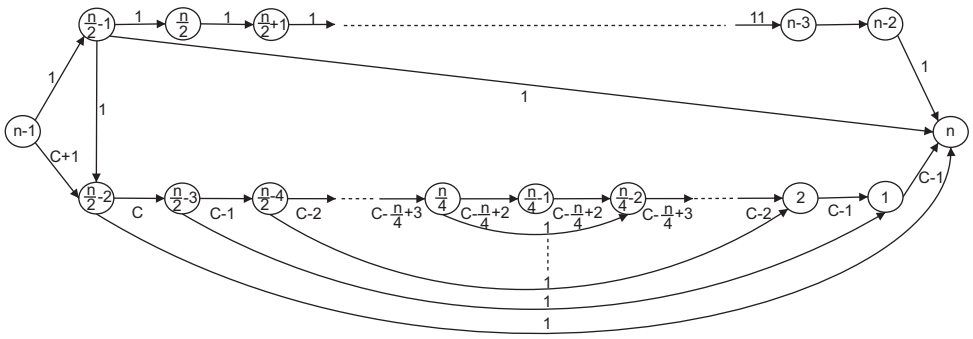

Fig. 6. Graph family where the preflow push algorithm suffers from the ping pong effect.

push algorithms. When does this flow exchange between nodes 1 and 8 terminate? First of all, observe that if the gap property is satisfied such an effect could be reduced. However, in our example the property is not satisfied (see the labels of nodes from 9 to 18). To see when the ping pong effect terminates, let us compare the residual network shown in Fig. 4 with the one shown in Fig. 2. As we can see, the two residual networks are the same, except for the distance label values for nodes from 1 to 8, that are greater. Analogously, the two residual networks shown in Figs. 3 and 5 differ only in the distance labels of the nodes from 1 to 8, that again are greater. Then, the algorithm generates a sequence of alternating residual networks of two types, that differ in the distance label values. Consider a residual network of the type shown either in Fig. 3 or 5. Given this type of residual network, the algorithm terminates when, relabeling node 8, it identifies s as the smallest label node among the ones adjacent to 8, that is when $d(8)$ become greater than $d(s) = n$. Note that, from the network in Fig. 3 to the network in Fig. 5 each distance label of the nodes from 1 to 8, is increased by 2 (indeed between two consecutive saturations of an arc (i, j) both $d(i)$ and $d(j)$ increase at least by 2 units, see [5]). Then, before the distance label of node 8 becomes greater than n, the algorithm performs $\Omega(n)$ flow exchanges between nodes 1 and 8. But how many push operations are performed by the algorithm for these flow exchanges? To answer this question, consider the graph shown in Fig. 6, which generalizes the previous example and where there are $n = 4d$ nodes, where d is an odd positive integer greater than 3. Node $n - 1$ is the source node, node n is the sink. When the algorithm is on this graph, analogously to the previous example, $\Omega(n)$ flow exchanges between nodes 1 and $n/2 - 2$ are performed. For each of these flow exchanges, $n/2 - 3$ push operations are performed, once along the path $(n/2 - 2, n/2 - 3, \ldots, 2, 1)$, and once along the path $(1, 2, \ldots, n/2 - 3, n/2 - 2)$. Then, the push operations are $\Omega(n^2)$.

3. Budget algorithm

In the previous section, we showed how the ping pong effect can be onerous. The main problem of preflow push algorithms is the *inherent characteristic of working locally*. That is, given an active node with distance label $d(i)$, the preflow push algorithm will send flow to a node j such that $d(i) = d(j) + 1$, without taking into account whether node j can "digest" or not the flow that it will receive. It would be suitable, and this is the intuition characterizing the algorithm we propose, that the algorithm, before sending flow from an active node i with distance label $d(i)$, makes sure that such a flow can reach a node l of label $d(l)$ such that $d(i) - d(l) \geqslant k$, for a given constant k. This condition can be verified by searching, from an extracted active node i, a path P_i in the residual network of length at least k; that is a path composed only by admissible arcs from node i to a node l such that $d(i) - d(l) \geqslant k$, and sending flow along this path. We can obtain such a result "merging" opportunely the preflow push and augmenting path algorithms.

In particular, we associate to such a path P_i a positive quantity defined $budget(P_i)$ that, initially, is equal to $kd(i)$ (where k is a given integer constant). Once the active node i is extracted and $budget(P_i)$ is initialized, the algorithm tries to build a directed path starting from i toward the sink, without necessarily reaching it. To do this, any time it is possible to extend the partial path on a node j, $budget(P_i)$ is decreased of the value $d(j)$. As soon as, extending on a node j, $budget(P_i)$ becomes negative, the algorithm sends flow along the path from i to j (the path has been found and the flow is pushed along it). Since P_i is composed by admissible arcs, $budget(P_i)$ becomes negative when the length of P_i is at least k.

It can happen, however, that starting from a node i such a path P_i cannot be found. That is, there is a node j with positive budget but without admissible arcs. In such a case, the algorithm contracts the path: removes the terminal node, and tries to extend the path from a different node.

A first tentative of overcoming the myopic characteristic of the preflow push algorithm is present in [2,3]. Indeed, the authors in [2] present an implementation of the algorithms that runs in $O(n^3)$ and propose a different implementation that uses a more sophisticated data structure, namely, the *dynamic tree data structure* that requires $O(nm\ \log(n^2/m))$-time. Such an implementation, somehow, reduces the time spent in performing the nonsaturating push operations by keeping track of partial augmenting paths with positive reduced capacity. The used data structure, however, has a substantial overhead that reduces the advantage of its practical utilization.

On the other hand, the combined algorithm proposed in [3] differs from our budget algorithm since it fixes the length of the path built from the selected active nodes. However, fixing a priori the length of the path is not effective from a computational point of view as we observed from our experiments. In our algorithm the length of the path is a function of the distance of the selected node: further from the sink the selected node, longer the path that the algorithm will build. Our budget algorithm is described next.

Algorithm 4. Budget algorithm

Input: A network $G = (N, A, s, t, u)$
Output: A maximum flow X of G

```
1.  initialize(G)
2.  while there is an active node do
3.    i ← argmax{d(i)|i ∈ N, e(i) > 0}
4.    budget ← kd(i)
5.    i_t ← i
6.    while budget ⩾ 0 and i_t ≠ s and i_t ≠ t do
7.      if there is an admissible arc (i_t, j) in δ+_G(i_t) then
8.        extend(i_t, j, budget)
9.      else
10.       contract(i, i_t, budget)
11.     end if
12.   end while
13.   augment(i)
14. end while
```

Procedure 5. *extend*$(i_t, j, budget)$

```
1. succ(j) ←NIL
2. pred(j) ← i_t
3. succ(i_t) ← j
4. i_t ← j
5. budget ← budget − d(j)
```

Procedure 6. *contract*$(i, i_t, budget)$

```
1. d' ← min{d(j) + 1|(i, j) ∈ δ+_G(i_t)}
2. if i_t ≠ i then
3.   budget ← budget + d(i_t)
4.   i_t ← pred(i_t)
5. else
6.   budget ← kd'
7. end if
8. d(i_t) ← d'
9. succ(i_t) ←NIL
```

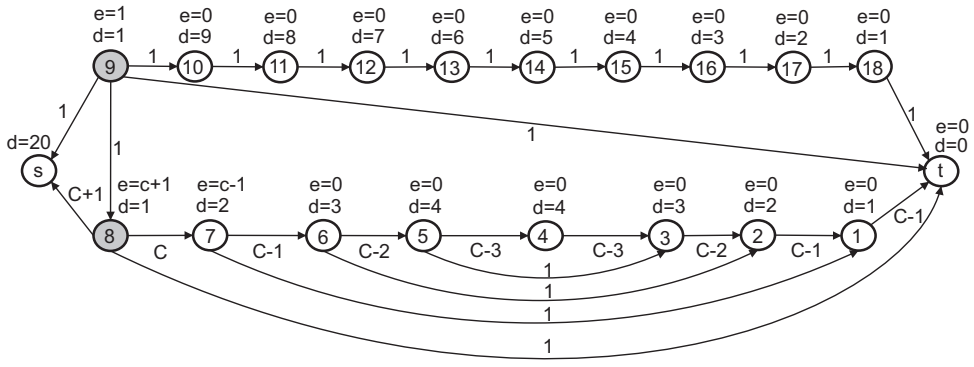

Fig. 7. The graph after the initialize procedure.

Procedure 7. *augment*(i)

1. **while** $succ(i) \neq$NIL **do**
2. $\quad j \leftarrow succ(i)$
3. $\quad \delta \leftarrow \min\{e(i), r_{ij}\}$
4. $\quad$ Send δ units of flow from i to j
5. $\quad i \leftarrow succ(i)$
6. **end while**

The algorithm, analogously to the preflow push method, begins calculating the exact distance labels (indeed we use the same *initialize* procedure) and saturating the arcs outgoing from the source. In this way, it defines a first set of active nodes. Successively, it iteratively selects an active node with the greatest distance label and from this node it tries to build a path. To do this, it maintains a partial path P_i, initially composed only of the selected active node i (the same path search method is used in [5] for the labeling algorithm). The path P_i is represented by the links *succ* and *pred* that point, respectively, to the next and to the previous node in the path; i_t is the actual terminal node of P_i. Within an inner while iteration, either an admissible arc is found and a node is added to the partial path, or a contract operation is performed. The path research phase terminates either when $budget(P_i) < 0$, or when the sink node is reached, or when the source node is reached. Only when the path search phase ends, the *augment* procedure is carried out. Such a procedure, $\forall(i, j) \in P_i$, sends a flow equal to the minimum between $e(i)$ and r_{ij} from i to j. This augmentation, defined "greedy" in [11], is advantageous for paths whose initial arcs have high capacity, and whose final arcs have small capacity.

We now describe the algorithm execution when applied on the graph of the previous example.

In Fig. 7 the graph after the *initialize* procedure is shown. We fix the value k for the budget algorithm equal to 3. The active nodes with the greatest distance label are nodes 8 and 9; suppose the latter is extracted by the algorithm. With the initial path $P_9 = \{9\}$ is associated the value $budget(P_9) = 3$. The algorithm performs an *extend* operation on the sink node and, since the sink is reached, an *augment* along the path $P_9 = \{9, t\}$ is performed. Then, node 8 is extracted; the initial path is $P_8 = \{8\}$ and $budget(P_8) = 3$. The algorithm performs an *extend* operation on the sink node and an *augment* along the path $P_8 = \{8, t\}$. Node 8 is again extracted, the initial path is $P_8 = \{8\}$, the budget value is 3. Since there do not exist any admissible arc outgoing from node 8, a *contract* operation is performed. The *contract* operation relabels the distance label of node 8 to the value 3, and $budget(P_8)$ is now 9. The algorithm performs an *extend* operation on node 7, decreasing $budget(P_8)$ to the value 7. The algorithm executes two additional extend operations, the first on node 1 and the second on the sink node . Then, the algorithm performs an *augment* along the path $P_8 = \{8, 7, 1, t\}$. In Fig. 8, it is possible to see the graph after this *augment* operation. Subsequently, the algorithm extracts nodes 7, 6 and 5 and performs *augment* operations along the paths $P_7 = \{7, 6, 2, t\}$, $P_6 = \{6, 5, 3, 2, 1, t\}$ and $P_5 = \{5, 4, 3, 2, 1, t\}$, obtaining the graph shown in Fig. 9, that is the same as that in Fig. 2. The algorithm extracts node 1. Since there is no admissible arc, a *contract* is performed, and node 1 is relabeled to the value 3: the budget associated to the path $P_1 = \{1\}$ is then 9. The algorithm performs an *extend* operation on node 2 building the path $P_1 = \{1, 2\}$ and decreasing $budget(P_1)$ to

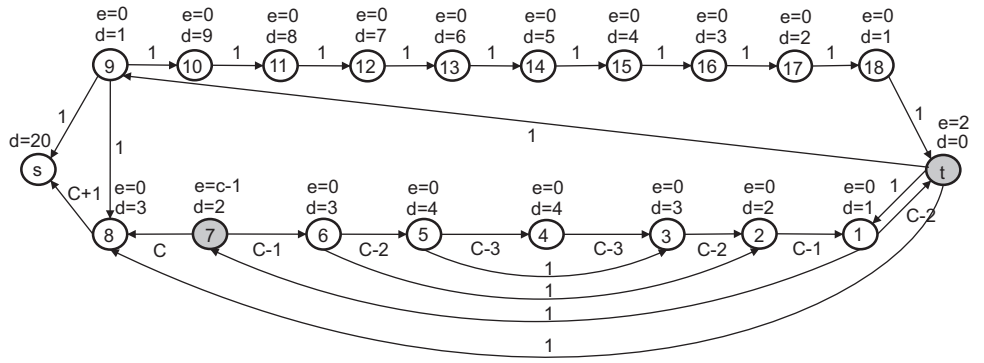

Fig. 8. The graph after the augment operation along the path $P = \{8, 7, 1, t\}$.

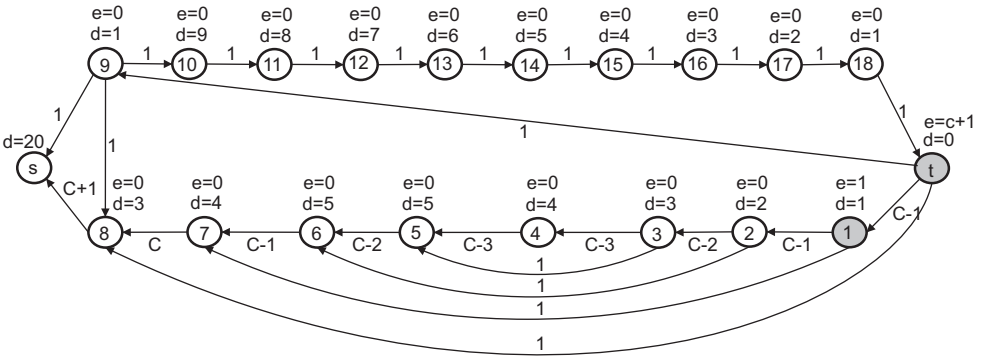

Fig. 9. The graph after the selection of nodes 7, 6 and 5.

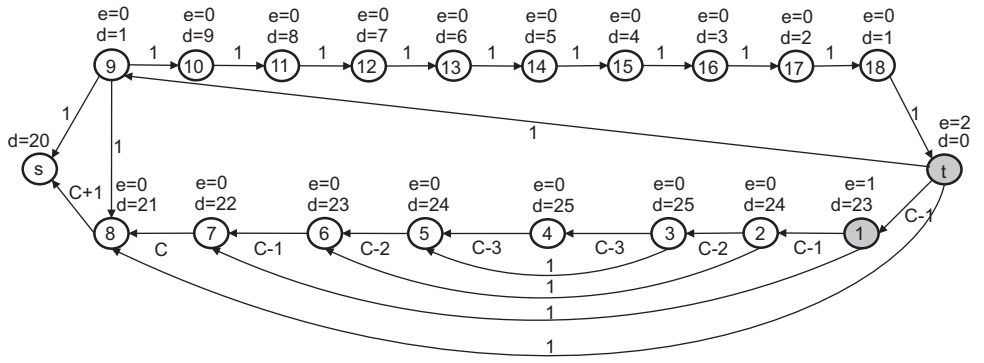

Fig. 10. The graph before the augment operation along the path $P = \{1, 6, 7, 8, s\}$.

the value 7. Since node 2 does not have any admissible arcs, a *contract* operation is performed, relabeling node 2 to the value 4, increasing *budget*(P_1) to the value 9, and removing node 2 in the path that is now $P_1 = \{1\}$. Since node 1 does not have any admissible arc, another *contract* operation is performed, relabeling node 1 to the value 5. The budget associated with P_1 is now 15. Two *extend* operations are performed, on nodes 2 and 3. The path is $P_1 = \{1, 2, 3\}$ and the budget is 8. Observe that, up to now, *not any augment operation has been performed*. After some of these operations, without any augment operation, the graph shown in Fig. 10 is obtained. Starting from this graph, the algorithm selects node 1 and builds the path $P_1 = \{1\}$; *budget*(P_1) = 69. The algorithm performs an extend operation, obtaining the path

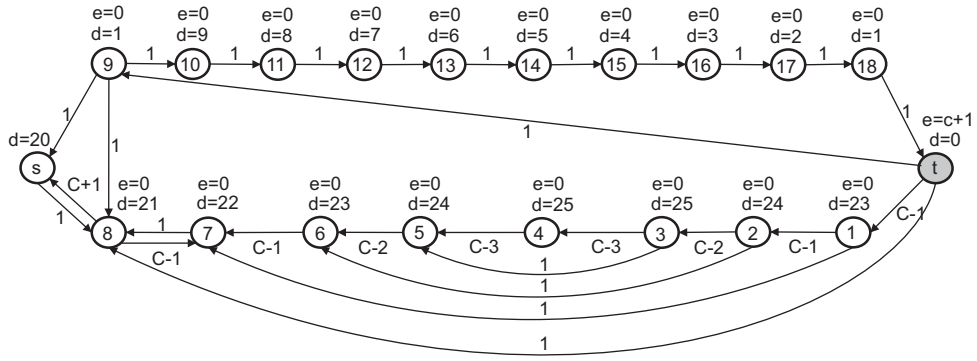

Fig. 11. The final graph.

$P_1 = \{1, 2\}$ and decreasing $budget(P_1)$ to the value 45. Then the algorithm performs two extend operations, building the path $P_1 = \{1, 7, 8, s\}$. Since the source node has been reached, the algorithm performs an augment operation, sending back to the source the excess flow in the node 1, obtaining the graph shown in Fig. 11. The algorithm terminates because there are no active nodes.

As we can see, an augment operation is performed either when it is sure that the flow can reach a node enough far away from the selected node, or when either the sink or the source nodes are reached. This allows the algorithm to avoid unnecessary flow exchanges between nodes and therefore the ping pong effect.

3.1. Correctness of budget algorithm

In this section we show that the budget algorithm correctly computes a maximum flow.

Theorem 1. *The budget algorithm maintains valid distance labels.*

Proof. We show the result by induction on the number of operations that modify the distance labels. The operations on the distance labels are augment and contract. Actually, the former does not modify the distance label, but it can create additional arcs in the residual network. Initially, the algorithm calculates exact distance labels, that is $d(i)$ is the length of a shortest path from i to t in $G(X)$, therefore the distance labels are valid. Then, in the initialization procedure, it saturates the arcs $(s, j) \in \delta_G^+(s)$ and sets $d(s)$ to n. Since for each node j adjacent to the source node we have $d(j) < n$, the relation $d(j) \leqslant d(s) + 1\ \forall (j, s) \in \delta_G^-(s)$ holds. Thus, initially, the distance labels are valid. We verify that the distance labels remain valid after a contract operation. Let $d(i)$ and $d'(i)$ be the distance labels of node i before and after a contract operation, respectively. A contract is performed when node i does not have any admissible arc, that is when $d(i) < d(j) + 1\ \forall (i, j) \in \delta_G^+(i)$. The algorithm sets $d'(i)$ to $\min\{d(j) + 1 | (i, j) \in \delta_G^+(i)\} > d(i)$. Therefore $d'(i) > d(i)$ and $d'(i) \leqslant d(j) + 1\ \forall (i, j) \in \delta_G^+(i)$. We consider the arcs coming into i. If before a contract we have $d(k) \leqslant d(i) + 1\ \forall (k, i) \in \delta_G^-(i)$, because $d'(i) > d(i)$, then after a contract we have $d(k) < d'(i) + 1\ \forall (k, i) \in \delta_G^-(i)$.

Consider now the augment operation. Such an operation is performed either on a directed path P_i from the selected active node i to a node j such that $d(i) - d(j) \geqslant k$, or on a path from i to the source or the sink. We show that distance labels remain valid $\forall (k, l) \in P_i$. An augment on an arc (k, l) can remove such an arc from the residual network. If this is the case, it is not necessary that the condition $d(k) \leqslant d(l) + 1$ is satisfied, otherwise the arc is not removed, since any arc of P_i is admissible, the relation $d(k) = d(l) + 1$ holds. An augment can also create the arc (l, k), for which $d(l) \leqslant d(k) + 1$, that comes from the admissibility of (k, l). □

Theorem 2. *The budget algorithm correctly computes the maximum flow.*

Proof. The algorithm terminates when there are no active nodes. That is, when the preflow is a flow. Since the residual network does not contain any directed path from the source to the sink (i.e., $d(s) = n$), at the end of the algorithm the flow is a maximum flow. □

The algorithm's running time depends on the parameter k. For $k \geqslant n$, the running time is the same of the algorithm of Edmonds and Karp [7]. Indeed, when $k \geqslant n$ the *budget* algorithm builds shortest augmenting paths from the nodes adjacent to the source node, to the sink node. Therefore the running time is $O(n^2 m)$. For $k = 0$ the *budget* algorithm becomes the highest label preflow push algorithm, so the running time is $O(n^2\sqrt{m})$.

4. Experimental results

In this section we present some interesting results from a wide computational experimentation of our *budget* algorithm.

4.1. Standard dimacs and problem families

In our implementations, we adopted the conventions defined for the DIMACS standard for the maximum flow problem. We used different problem families in our experimentations, some of which are used in the "First DIMACS Implementation Challenge" [12]. Such families are produced by different public domain generators: GENRMF of Goldfarb and Grigoriadis [13], AC of Setubal [14], and AK of Goldberg [15]. These generators, except AK that is deterministic, are random generators, in the sense that they produce different problem instances for the same input parameters. All problem instances are available upon request to the authors.

We now specify the problem families obtained from the generators.

- *GENRMF LONG*. This family contains the networks composed of 2^x nodes generated by specifying as input parameters $a = 2^{x/4}$ and $b = 2^{x/2}$.
- *GENRMF WIDE*. This is the family composed of networks that have 2^x nodes obtained by specifying as parameter the values $a = 2^{2x/5}$ and $b = 2^{x/5}$.
- *AK*. A network in this family has $4k+6$ nodes and $6k+7$ arcs and it is generated by specifying the input parameter k.
- *AC*. A network that belongs to this family has 2^x nodes and $2^{x-1}(2^x - 1)$ arcs and is generated by specifying as input the parameter x and the seed generator.

4.2. Implementations

We implemented the *budget* algorithm with the global relabel strategy and the gap property. In particular, we fixed the global relabel frequency to n. That is, we have a global relabeling every n contract operations. We evaluated also the algorithm for different values of parameter k.

Table 1
Execution times on different instances of GENRMF LONG family

Budget	k	GENRMF LONG				
		$n = 32\,768$ $m = 155\,392$	$n = 65\,536$ $m = 311\,040$	$n = 131\,072$ $m = 49\,159$	$n = 262\,144$ $m = 1\,276\,928$	$n = 524\,288$ $m = 2\,554\,880$
	1	0.26	0.41	0.77	9.49	17.20
	2	0.22	0.36	0.70	5.24	8.77
	3	0.20	0.37	0.63	4.50	7.71
	4	0.21	0.35	0.57	3.93	7.49
	5	0.22	0.36	0.60	3.97	6.78
	6	0.19	0.37	0.72	3.99	6.94
	7	0.19	0.41	0.71	3.87	6.95
	8	0.24	0.42	0.72	3.88	6.80
	9	0.25	0.42	0.75	3.53	7.69
	10	0.26	0.42	0.76	3.65	6.89
	15	0.27	0.55	0.90	4.18	7.47
	20	0.35	0.59	1.03	4.42	8.69
Pr. push(h.l.)		0.26	0.51	0.94	9.90	23.74

Table 2
Execution times on different instances of GENRMF WIDE family

Budget	k	GENRMF WIDE				
		$n = 32\,768$ $m = 157\,969$	$n = 65\,536$ $m = 319\,488$	$n = 131\,072$ $m = 634\,880$	$n = 262\,144$ $m = 1\,286\,144$	$n = 1\,048\,576$ $m = 5\,193\,728$
	1	1.89	3.47	13.38	24.26	104.28
	2	1.61	2.65	11.63	18.28	83.25
	3	1.48	2.42	10.78	18.36	79.62
	4	1.48	2.44	10.68	18.28	74.09
	5	1.47	2.46	10.62	17.58	73.58
	6	1.47	2.51	11.55	17.91	70.84
	7	1.70	2.54	11.94	18.94	75.48
	8	1.70	2.61	11.62	21.40	73.16
	9	1.72	2.53	12.22	20.24	74.02
	10	1.73	2.89	12.55	20.74	74.64
	15	2.02	3.07	14.88	22.68	75.42
	20	2.13	3.46	16.83	23.96	74.76
Pr. push(h.l.)		1.52	3.07	13.77	26.70	121.48

Table 3
Execution times on different instances of AK family

Budget	k	AK				
		$n = 4102$ $m = 6151$	$n = 16\,360$ $m = 24\,583$	$n = 32\,774$ $m = 49\,159$	$n = 65\,542$ $m = 98\,311$	$n = 131\,078$ $m = 196\,615$
	1	0.08	2.80	16.14	69.69	281.09
	2	0.07	2.46	15.56	72.11	273.59
	3	0.05	1.84	12.97	58.51	240.69
	4	0.05	1.59	12.23	55.61	229.58
	5	0.05	1.37	11.76	54.14	223.86
	6	0.05	1.40	11.46	52.96	219.20
	7	0.04	1.30	11.31	52.22	216.26
	8	0.04	1.18	11.06	51.31	212.87
	9	0.05	1.24	11.03	51.27	210.45
	10	0.04	1.18	10.84	50.53	209.73
	15	0.04	1.18	10.63	49.72	206.93
	20	0.04	1.23	10.43	49.25	205.40
Pr. push(h.l.)		0.07	2.25	13.37	58.86	237.32

Codes used for comparison: We compared our *budget* algorithm with codes written in C language, developed as described in [15]. We compared our algorithm with the highest label preflow push algorithm, the fastest among the preflow push algorithms. Also for the highest label preflow push implementation, the gap property and the global relabeling (every n relabel operations) were used.

Computing environment: We performed our experiments on a personal computer with processor AMD Athlon 1.1 GHz, 640 MB of RAM, with suse linux (kernel 2.4.0) as operating system. We implemented the *Budget* algorithm in C language, and we used the gcc compiler (version 2.95.2) using the optimization option -O.

4.3. Test results

In this section we report the experimental results, specifying the execution time in seconds of a given algorithm on a given problem. Each table reports the execution times of the budget algorithm for different values of parameter k

Table 4
Execution times on different instances of AC family

Budget	k	AC				
		$n = 500$ $m = 124\,750$	$n = 1000$ $m = 499\,500$	$n = 1500$ $m = 1\,124\,250$	$n = 2000$ $m = 1\,999\,000$	$n = 3000$ $m = 4\,498\,500$
	1	0.09	1.14	2.19	3.58	13.98
	2	0.07	1.01	1.86	3.81	10.84
	3	0.07	1.04	1.53	3.87	8.81
	4	0.07	0.83	1.88	3.74	8.94
	5	0.07	0.85	1.54	3.03	8.91
	6	0.07	0.94	1.50	3.24	9.02
	7	0.07	0.94	1.51	3.25	9.09
	8	0.06	0.84	1.49	3.10	8.94
	9	0.07	0.73	1.50	3.30	8.69
	10	0.07	0.95	1.49	3.00	9.04
	15	0.07	0.90	1.40	3.30	8.91
	20	0.07	0.94	1.58	3.15	9.09
Pr. push(h.l.)		0.08	0.96	1.51	3.31	9.87

(it varies from 1 to 20), for different values of n (number of nodes) and m (number of arcs). All the values are average values over 10 different instances. The last row of each table reports the execution time of the highest label preflow push algorithm.

Tables 1–4 show the execution times of the tested algorithms on the considered families. From these results, interesting properties of the *budget* algorithm come up. First of all, for all families, there is a range of values for k in correspondence of which the *budget* algorithm is faster than the highest label preflow push algorithm. By analyzing the experimental results, for the considered families, a good choice seems to set k in the interval [3,7]. In addition, the performances of the *budget* algorithm worsen when k increases. Indeed, when k stretch to n, *budget* algorithm becomes the shortest augmenting path, except for the initialization phase.

5. Conclusions

In this paper we presented an approach combining the augmenting path and preflow push methods that avoids the drawback known as *ping pong effect*. Our computational results show the effectiveness of the proposed approach. Further research will investigate potential relations between k and some other network parameter such as the graph diameter.

Acknowledgments

We dedicate this work to the memory of Prof. Stefano Pallottino, for his careful reading and his useful comments and suggestions.

The authors are also grateful to the anonymous referee for his valuable comments.

References

[1] Ford LR, Fulkerson DR. Flows in networks. Princeton, NJ: Princeton University Press; 1962.
[2] Goldberg AV, Tarjan RE. A new approach to the maximum flow problem. In: Proceedings of the 18th annual ACM symposium on theory of computing. 1986. p. 136–46.
[3] Mazzoni G, Pallottino S, Scutellà MG. The maximum flow problem: a max-preflow approach. European Journal of Operational Research 1991;53:257–78.
[4] Derigs U, Meier W. Implementing Goldberg's max flow algorithm, A computational investigation. Zeitschrif für Operations Research 1989;33:383–403.
[5] Ahuja RK, Magnanti TL, Orlin JB. Network flows: theory, algorithms, and applications. Englewood Cliffs, NJ: Prentice-Hall; 1993.
[6] Goldberg AV, Tarjan R. A new approach to the maximum flow problem. Journal of the Association for Computing Machinery 1988;35:921–40.

[7] Edmonds J, Karp RM. Theoretical improvements in algorithmic efficiency for networks flow problem. Journal of the ACM 1972;19:248–64.
[8] Cheriyan J, Maheshwari SN. Analysis of preflow push algorithms for maximum network flow. Lecture notes in computer science, vol. 338. Berlin: Springer; 1988. p. 30–48.
[9] Tunçel L. On the complexity of preflow-push algorithm for maximum flow problem. Algorithmica 1994;11:353–9.
[10] Cheriyan J, Mehlhorn K. An analysis of the highest-level selection rule in the preflow-push max-flow algorithm. Information Processing Letters 1999;69:239–42.
[11] Bertsekas D. An auction algorithm for the max-flow problem. Journal of Optimization Theory and Applications 1995;87(1):69–101.
[12] Johnson DS, McGeoch CC, editors. Network flows and matching: first DIMACS implementation challenge. Providence, R1: AMS, 1993.
[13] Goldfarb D, Grigoriadis MD. A computational comparison of the dinic and network simplex methods for maximum flow. Annals of Operation Research 1988;13:83–123.
[14] Anderson RJ, Setubal JC. Goldberg's algorithm for the maximum flow in perspective: a computational study. In: Johnson DS, McGeoch CC, editors. Network flows and matching: first DIMACS implementation challenge. Providence, RI: AMS; 1993. p. 1–18.
[15] Cherkassky BV, Goldberg AV. On implementing the push-relabel method for the maximum flow problem. Algorithmica, 1997; 390–410.