

An algorithm for minimum flows

OANA GEORGESCU

Transilvania University of Braşov
Department of Computer Science
Braşov, Bd. Iuliu Maniu 50, cod 500091
ROMANIA
o.georgescu@unitbv.ro

ELEONOR CIUREA

Transilvania University of Braşov
Department of Computer Science
Braşov, Bd. Iuliu Maniu 50, cod 500091
ROMANIA
e.ciurea@unitbv.ro

Abstract: In this paper we consider an advanced topic for minimum flow problem: the use of the dynamic trees data structure to efficiently implement decreasing path algorithm. In the final part of the paper we present an example for this algorithm, some practical applications for the minimum flow problem, conclusions and open problems.

Key-Words: network flow, network algorithms, minimum flow problem, dynamic tree

1 Introduction

The theory of flows is one of the most important parts of Combinatorial Optimization. The computation of a maximum flow in a graph has been an important and well studied problem, both in computer science and operations research. Many efficient algorithms have been developed to solve this problem, see, e.g., [1], [13]. Sleator and Tarjan [14] developed the dynamic tree data structure and used it to improve the worst-case complexity of Dinic's algorithm from $O(n^2m)$ to $O(nm \log n)$. Recently, we have used this data structure to improve the performance of a range of network flow algorithms, see for example [8], [9] and [10]. Using the dynamic tree data structure, Goldberg and Tarjan [11] improved the complexity of the FIFO preflow-push algorithm from $O(n^3)$ to $O(nm \log(n^2/m))$ and Ahuja, Orlin and Tarjan [2] improved the complexity of the excess scaling algorithm and several of its variants. Recently, in [5] and [6] we achieved important results developing improved pre-flow algorithms for solving the minimum flow problem.

The computation of a minimum flow has been investigated by Ciurea and Georgescu [4] for some specific classes of directed networks and by Ciurea and Ciupală [7] for using the parallel algorithms. The minimum flow problem and the maximum cut problem arise in a wide variety of situations and in several forms. For example, some direct applications might be: machine setup problem, tanker scheduling problem, airlines scheduling problem etc.

In this paper we consider an advanced topic: the use of the dynamic trees data structure to efficiently implement algorithms for the minimum flow problem.

The brief outline of the paper is as follows: in

Section 2 we discuss some basic notions and results used in the rest of the paper. Section 3 deals with the decreasing path algorithms for minimum flow with dynamic tree implementation. In Section 4 we present an example for this algorithm. Section 5 treats some applications of the minimum flows and in Section 6 we present conclusions and some open problems.

2 Terminology and Preliminaries

In this section we discuss some basic minimum flow notations and results used in the rest of the paper.

As we describe in [4] and [10], we consider a capacitated directed network $G = (N, A, l, c, s, t)$ with a nonnegative capacity $c(x, y)$ and with a nonnegative lower bound $l(x, y)$ associated with each arc $(x, y) \in A$. We distinguish two special nodes in the network G : a source node s and a sink node t .

For a given pair of not necessarily disjoint subsets X, Y of the nodes set N of a network G we use the notation:

$$(X, Y) = \{(x, y) | (x, y) \in A, x \in X, y \in Y\}$$

and for a given function f on arcs set A we use the notation:

$$f(X, Y) = \sum_{(x,y) \in (X,Y)} f(x, y)$$

A *flow* is a function $f : A \rightarrow \mathbb{R}_+$ satisfying the next conditions:

$$f(x, N) - f(N, x) = \begin{cases} v, & \text{if } x = s \\ 0, & \text{if } x \neq s, t \\ -v, & \text{if } x = t \end{cases} \quad (1.a)$$

$$l(x, y) \leq f(x, y) \leq c(x, y), \forall (x, y) \in A, \quad (1.b)$$

for some $v \geq 0$ and we refer to v as the *value of the flow* f .

The *minimum flow problem* consists in determining a flow f for which v is minimized.

A *cut* is a partition of the nodes set N into two subsets S and $T=N-S$; we represent this cut using notation $[S, T]$. We refer to an arc $(x, y) \in A$ with $x \in S$ and $y \in T$ as a *forward arc* of the cut and an arc $(x, y) \in A$ with $x \in T$ and $y \in S$ as a *backward arc* of the cut. Let (S, T) denote the set of forward arcs in the cut and let (T, S) denote the set of backward arcs. We refer to a cut as an *s-t cut* if $s \in S$ and $t \in T$.

For the minimum flow problem, we define the *capacity* $c[S, T]$ of the *s-t cut* $[S, T]$ as the sum of the forward arcs lower bounds minus the sum of the backward arcs capacities. That is:

$$c[S, T] = l(S, T) - c(T, S) \quad (2)$$

We refer to an *s-t cut* which has the maximum capacity among all *s-t cuts* as a *maximum cut*.

An important theorem is the following [1]:

Theorem 1 *If there exists a feasible flow in the network, the value of the minimum flow from a source node s to a sink node t in network G equals the capacity of the maximum s-t cut.*

For the minimum flow problem, the *residual capacity* $\hat{r}(x, y)$ of any arc $(x, y) \in A$, with respect to a given flow f , is given by $\hat{r}(x, y) = c(y, x) - f(y, x) + f(x, y) - l(x, y)$. By convention, if $(x, y) \in A$ and $(y, x) \notin A$ then we add arc (y, x) to the set of arcs A and we set $l(y, x) = 0$ and $c(y, x) = 0$. The network $\hat{G}(f) = (N, \hat{A})$ consisting only of the arcs with positive residual capacity is referred to as the *residual network* (with respect to the flow f).

In the residual network $\hat{G}(f) = (N, \hat{A})$, the *distance function* is a function $\hat{h} : N \rightarrow \mathbb{N}$. We say that a distance function is *valid* if it satisfies the following conditions:

$$\hat{h}(t) = 0 \quad (3.a)$$

and

$$\hat{h}(x) \leq \hat{h}(y) + 1, \forall (x, y) \in \hat{A} \quad (3.b)$$

We refer to $\hat{h}(x)$ as the *distance label of node x* and we refer to $(x, y) \in \hat{A}$ as an *admissible arc* if $\hat{h}(x) = \hat{h}(y) + 1$; otherwise it is *inadmissible*. We refer to a path from node s to node t consisting entirely of admissible arcs as an *admissible path*. We say that the distance labels are *exact* if for each node

x , $\hat{h}(x)$ equals the length of the shortest directed path from node x to node t in the residual network $\hat{G}(f)$. We obtain the exact distance labels $\hat{h}(x)$ in $\hat{G}(f)$ by perform a revers breadth first search of the residual network $\hat{G}(f)$ from the sink node t to source node s . We refer to a path in G from the source node s to the sink node t as a *decreasing path* if it consists only of arcs with positive residual capacity. Clearly, there is an one to one correspondence between set of decreasing paths in G and the set of directed paths from s to t in $\hat{G}(f)$.

A *partial admissible path* is a path from some node x to node t consisting solely of admissible arcs. In this case the node x is named *current node*.

The minimum flow problem in a general directed *s-t network* can be solved in two phases:

1. establish a feasible flow f , if it exists;
2. from a given feasible flow f , establish the minimum flow \hat{f} .

The problem of determining a feasible flow consists in finding a function $f : A \rightarrow \mathbb{R}_+$ that satisfies the previous conditions (1.a) and (1.b). First, we transform this problem into a circulation problem by adding an arc (t, s) of infinite capacity and zero lower bound. This arc carries the flow sent from the source node s to the sink node t back to the source node s . Clearly, the minimum flow problem admits a feasible flow if and only if the circulation problem admits a feasible flow. Because these two problems are equivalent, we focus on finding a feasible circulation if it exists in the transformed network $\tilde{G} = (N, \tilde{A}, \tilde{l}, \tilde{c}, s, t)$, where

$$\tilde{A} = A \cup \{(t, s)\},$$

$$\tilde{l}(x, y) = l(x, y), \text{ for each arc } (x, y) \in A,$$

$$\tilde{l}(t, s) = 0,$$

$$\tilde{c}(x, y) = c(x, y), \text{ for each arc } (x, y) \in A,$$

$$\tilde{c}(t, s) = \infty.$$

The feasible circulation problem is to identify a flow f satisfying the following constraints:

$$\tilde{f}(x, N) - \tilde{f}(N, x) = 0, \text{ for each node } x \in N,$$

$$\tilde{l}(x, y) \leq \tilde{f}(x, y) \leq \tilde{c}(x, y), \text{ for each arc } (x, y) \in \tilde{A}.$$

By replacing $\tilde{f}(x, y) = \tilde{f}'(x, y) + \tilde{l}(x, y)$ and $\tilde{c}(x, y) = \tilde{c}'(x, y) + \tilde{l}(x, y)$ we obtain the following supply and demand problem:

$$\tilde{f}'(x, N) - \tilde{f}'(N, x) = \tilde{b}'(x), \text{ for each node } x \in N,$$

$$0 \leq \tilde{f}'(x, y) \leq \tilde{c}'(x, y), \text{ for each arc } (x, y) \in \tilde{A},$$

where

$$\tilde{b}'(x) = \tilde{l}(N, x) - \tilde{l}(x, N), \text{ for each node } x \in N.$$

Clearly,

$$\sum_N \tilde{b}'(x) = 0.$$

We can solve this supply and demand problem by solving a maximum flow problem defined in the network $\tilde{G}' = (\tilde{N}', \tilde{A}', \tilde{c}', s', t')$, where

$$\begin{aligned} \tilde{N}' &= \tilde{N}'_1 \cup \tilde{N}'_2 \cup \tilde{N}'_3, \\ \tilde{N}'_1 &= \{s'\}, \tilde{N}'_2 = N, \tilde{N}'_3 = \{t'\}, \\ \tilde{A}' &= \tilde{A}'_1 \cup \tilde{A}'_2 \cup \tilde{A}'_3, \\ \tilde{A}'_1 &= \{(s', y) | y \in N, \tilde{b}'(y) > 0\}, \\ \tilde{A}'_2 &= A, \\ \tilde{A}'_3 &= \{(x, t') | x \in N, \tilde{b}'(x) < 0\}, \\ \tilde{c}'(s', y) &= \tilde{b}'(y) \text{ with } (s', y) \in \tilde{A}'_1, \\ \tilde{c}'(x, y) &= \tilde{c}(x, y) \text{ with } (x, y) \in \tilde{A}'_2, \\ \tilde{c}'(y, t') &= -\tilde{b}'(x) \text{ with } (x, t') \in \tilde{A}'_3. \end{aligned}$$

Then we solve a maximum flow problem in network $\tilde{G}' = (\tilde{N}', \tilde{A}', \tilde{c}', s', t')$. If the maximum flow saturates all (s', y) arcs ($\tilde{f}'(s', y) = \tilde{b}'(y)$) and all (x, t') arcs ($\tilde{f}'(x, t') = -\tilde{b}'(x)$) then the initial problem has a feasible solution (which is the restriction of the maximum flow that saturates all the source and sink arcs to the initial set of arcs A); otherwise, it is infeasible.

There are three approaches for solving the minimum flow problem (see [1] or [3]): (1) using the decreasing path algorithms from source node s to sink node t in residual network $\hat{G}(f)$; (2) using preflow-pull algorithms from sink node t to source node s in residual network $\hat{G}(f)$; (3) using the augmenting path algorithms from sink node t to source node s or using preflow-push algorithms starting from sink node t in residual network $\tilde{G}(f)$ (residual network for maximal flow).

In this paper we present the decreasing path algorithms from source node s to sink node t with dynamic tree implementations for solving the minimum flow problem.

All the algorithms from the next table are decreasing path algorithms, i.e. algorithms which determine decreasing path from source node s to sink node t (by different rules) in residual network $\hat{G}(f)$ and then augment flows along these paths. We have $n = |N|$, $m = |A|$, $\bar{c} = \max\{c(x, y) | (x, y) \in A\}$.

Decreasing path algorithms	Running time
Generic decreasing path algorithm	$O(nm\bar{c})$
Ford-Fulkerson labeling algorithm	$O(nm\bar{c})$
Gabow bit scaling algorithm	$O(nm \log \bar{c})$
Ahuja-Orlin maximum scaling algorithm	$O(nm \log \bar{c})$
Edmond-Karp shortest path algorithm	$O(nm^2)$
Ahuja-Orlin shortest path algorithm	$O(n^2m)$
Dinic layered networks algorithm	$O(n^2m)$
Ahuja-Orlin layered networks algorithm	$O(n^2m)$

Actually, any decreasing path algorithm terminates with optimal residual capacities. From these residual capacities we can determine a minimum flow by following expression: $\hat{f}(x, y) = l(x, y) + \max\{\hat{r}(x, y) - c(y, x) + l(y, x), 0\}$.

3 Decreasing path algorithms for minimum flow. Dynamic tree implementations

Dynamic trees represent an important data structure extensively used by researchers to improve the worst-case complexity of several network algorithms. In this section we describe the use of this data structure for the shortest decreasing path algorithm. The following observation serves as a motivation for the dynamic tree structure usefulness. The shortest decreasing path algorithm repeatedly identifies a path consisting solely of admissible arcs and decreases flows on these paths. Each decrease saturates some arcs of this path and by deleting all the saturated arcs ($f(x, y) = l(x, y)$) from this path we obtain a set of path fragments. The dynamic tree data structure cleverly stores these path fragments and uses them later for quickly identifying decreasing paths.

Dynamic trees represent a special type of data structure that permits us to implicitly send flow on paths of length n in $O(\log n)$ steps on average. By doing so we are able to reduce the computational requirement of the shortest decreasing path algorithm for minimum flows from $O(n^2m)$ to $O(nm \log n)$.

The dynamic tree data structure maintains a collection T of node-disjoint rooted trees, each arc with an associated value. Each rooted tree is a directed in-tree with a unique root. Each node x (except the root node) has a unique *predecessor*, which is the next node on the unique path in the tree from that node

to the root. We store the predecessor of node x using a predecessor index $\hat{p}(x)$. If $y = \hat{p}(x)$, we say that node y is the predecessor of node x and node x is the *successor* of node y . These predecessor indices uniquely define a rooted tree and also allow us to trace out the unique direct path from the any node back to root. Similarly, we define the *ancestors* and the *descendants* of a node. The *descendants* of a node x consist of the node itself, its successors, successors of its successors and so on. We say that a node is an *ancestor* of all of its descendants. Notice that, according to our definitions, each node is its own ancestor and descendant.

This data structure supports six operations obtain by perform the following six procedures:

ROOT(x): find the root of the tree containing node x ;

VALUE(x): find the value of the tree arc leaving node x . If node x is a root node, the operation returns the value ∞ .

ANCES(x): find the ancestor u of x with the minimum value of *VALUE*(u). In case of a tie, chose the node u closest to the tree root.

CHANGE(x, \bar{w}): add a real number \bar{w} to the value of every arc along the direct path from node x to *ROOT*(x).

LINK(x, y, w): combines the trees containing tree root x and tree containing node y (the predecessor of node x) and give arc (x, y) the value w .

DELETE(x): break the tree containing node x into two trees by deleting the arc joining node x to its predecessor; we perform this operation when x is not a tree root.

The following important result lies at the heart of the efficiency of the dynamic tree data structure [1].

Theorem 2 *If q is the maximum number of nodes in any tree, a sequence of k tree operations, starting with an initial collection of singleton trees, requires a total of $O(k \log(k + q))$ time.*

The dynamic tree implementation stores the values of tree arcs only implicitly. Storing the values implicitly allows us to update the values in only $O(\log q)$ time.

Let us use the Ahuja-Orlin shortest path algorithm as an illustration. The detailed Pseudocode (1), Pseudocode (2), Pseudocode (3) and Pseudocode (4) give a formal statement of the algorithm.

The first two procedures, *TADV* and *TRET*, are straightforward, but the *TDEC* procedure requires some explanation. If node u is an ancestor of node s with the minimum value of *VALUE*(u) then *VALUE*(u) gives residual capacity of the decreasing path. The operation *CHANGE*($s, -w$) implicitly updates the residual

```
(1)ALGORITHM TDP;
(2)BEGIN
(3) let  $f$  be a feasible flow in network  $G$ ;
(4) determine the residual network  $\hat{G}(f)$ ;
(5) compute the exact distance labels  $\hat{h}(x)$  in  $\hat{G}(f)$ ;
(6) let  $T$  be the collection of all singleton nodes;
(7)  $x := s$ ;
(8) WHILE  $\hat{h}(s) < n$  DO
(9) BEGIN
(10) IF exist an admissible arc  $(x, y)$ 
(11) THEN TADV( $x$ )
(12) ELSE TRET( $x$ );
(13) IF  $x = t$ 
(14) THEN TDEC;
(15) END;
(16)END.
```

Pseudocode 1: Dynamic tree implementation for Ahuja-Orlin shortest decreasing path algorithm.

```
(1)PROCEDURE TADV( $x$ );
(2)BEGIN
(3) LINK( $x, y, \hat{r}(x, y)$ );
(4)  $x := \text{ROOT}(y)$ ;
(5)END;
```

Pseudocode 2: *TADV* Procedure

```
(1)PROCEDURE TRET( $x$ );
(2)BEGIN
(3)  $\hat{h}(x) := \min\{\hat{h}(y) + 1 \mid (x, y) \in A, \hat{r}(x, y) > 0\}$ ;
(4) FOR  $(z, x) \in \hat{A}$  DO DELETE( $z$ );
(5)  $x := \text{ROOT}(s)$ ;
(6)END;
```

Pseudocode 3: *TRET* Procedure

```
(1) PROCEDURE TDEC;
(2) BEGIN
(3)  $u := \text{ANCES}(s)$ ;
(4)  $w := \text{VALUE}(u)$ ;
(5) CHANGE( $s, -w$ );
(6) WHILE VALUE( $u$ ) = 0 DO
(7) BEGIN
(8) DELETE( $u$ );
(9)  $u := \text{ANCES}(s)$ ;
(10) END;
(11)  $x := \text{ROOT}(s)$ ;
(12) update the residual network  $\hat{G}(f)$ ;
(13)END;
```

Pseudocode 4: *TDEC* Procedure

capacities of all the arcs in the decreasing path. This decrease might cause the capacity of more than one arc in the path to become zero. The WHILE loop identifies all such arcs, one by one, and deletes them from the collection of rooted trees.

Theorem 3 *The TDP algorithm correctly computes a minimum flow.*

Proof: The TDP algorithm is same as the Ahuja-Orlin shortest path algorithm except that it performs the procedures TADV, TRET and TDEC differently, using trees. □

Theorem 4 *The TDP algorithm solves the minimum flow problem in $O(nm \log n)$ time.*

Proof. Using simple arguments, we can show that the algorithm performs each of the six tree operations in $O(nm)$ time. It performs each tree operation on a tree of maximum size n . The use of Theorem (2) establishes the result. □

4 Example

The network G is represented in figure (1) and the corresponding residual network is in figure (2), with $s=1$ and $t=6$. Initially, the dynamic tree contains the collection of singleton six nodes.

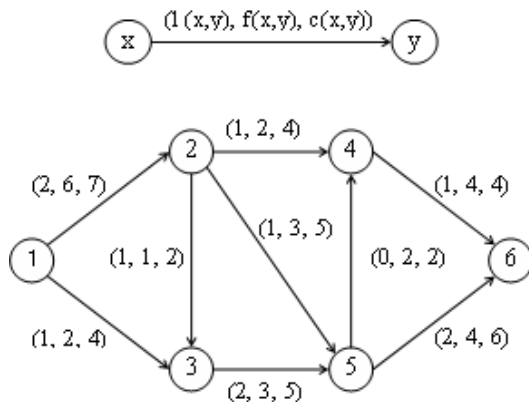


Figure 1: The network flow

We determine the exact distance labels for the nodes in the network and obtain $\hat{h} = (3, 2, 2, 1, 1, 0)$. We also set the current node as the source node s .

In the first iteration the algorithm selects the source node 1 and choose (1, 2) as the admissible arc in the residual network. We apply a TADV procedure; consequently, we add the tree arc (1, 2) with value 4 by applying LINK(1,2,4) and we set $x:=2$ following the operation $x:=ROOT(2)$. In the next iteration

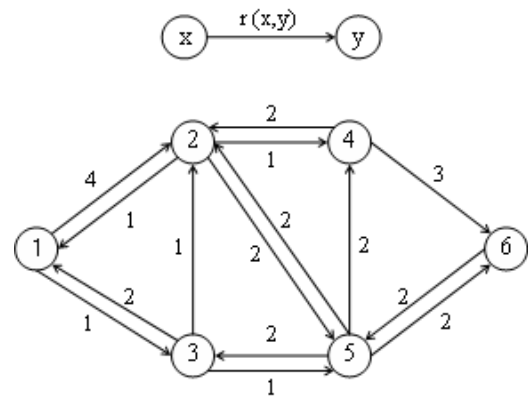


Figure 2: The initial residual network

we add the tree arc (2, 5) with value 2 and we set $x:=ROOT(5)=5$. Next, we apply LINK(5,6,2). Because the current node is the sink node, we apply a TDEC procedure on admissible path (1, 2, 5, 6). The tree is represented in figure (3).

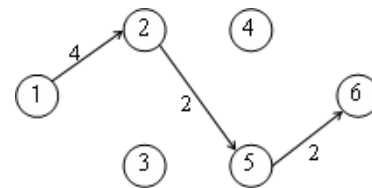


Figure 3: Dynamic tree

We determine value of u in the following manner: $u:=ANCES(1)=5$ with $w=VALUE(5)=2$. The operation CHANGE(1,-2) decreases 2 units from every tree arc. In this moment, the arcs (5, 6) and (2, 5) have value 0, so we delete these arcs from the tree (see the DELETE(5) and DELETE(2) operations). The dynamic tree is now represented in figure (4).

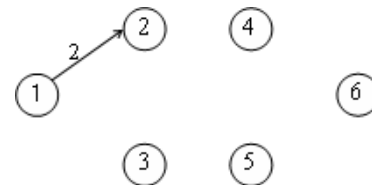


Figure 4: Dynamic tree

Next we set $x:=ROOT(1)=2$; the updated residual network is represented in figure (5).

We apply the LINK(2,4,1) and LINK(4,6,3) operations and decrease 1 unit of flow from every arc on

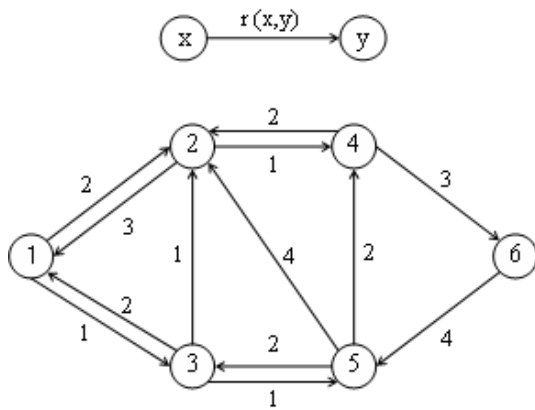


Figure 5: The intermediate residual network

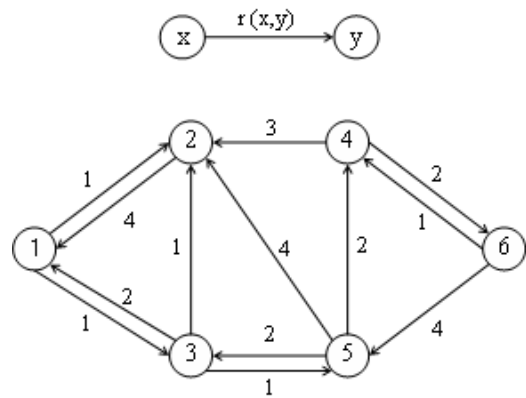


Figure 7: The intermediate residual network

path (1, 2, 4, 6). At the end of this iteration the current node is $x=6$, so by start with TDEC procedure, we determine the dynamic tree of figure (6) and the residual network of figure (7).

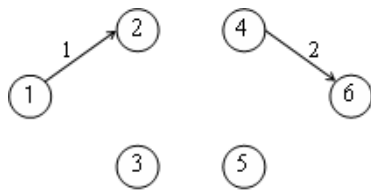


Figure 6: Dynamic tree

The current node is now $x:=ROOT(1)=2$. In the next iteration we don't have an admissible arc from node 2, so we apply the TRET(2) procedure and determine $\hat{h}(2)=\min\{\hat{h}(1) + 1\}=4$. Now, we delete the arc (1, 2) from the tree and set $x=1$. Next, we apply the LINK(1,3,1) and LINK(3,5,1) operations. Because we don't have an admissible arc from the node 5, we apply the TRET(5), TRET(3) and TRET(1) procedures, thus $\hat{h}=(4, 4, 3, 1, 2, 0)$. We obtain the directed path (1, 3, 5, 4, 6) with $u=3, w=1$ and apply the CHANGE(1,-1) procedure. Now, the dynamic tree and the residual network are represented in figures (8) and (9), respectively.

The current node is $x=1$. There is not an admissible arc starting from node 1, so the TRET(1) procedure set $\hat{h}=(5, 4, 3, 1, 2, 0)$. In the next iteration, (1, 2) is an admissible arc, but there are not an admissible arc from node 2, so we apply the TRET(2) procedure and determine $\hat{h}=(5, 6, 3, 1, 2, 0)$. The TRET(1) procedure determines $\hat{h}=(7, 6, 3, 1, 2, 0)$ and the algorithm ends with the value of minimum flow $\hat{v}=4$. The minimum network flow is represented in figure (10).

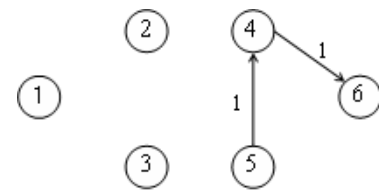


Figure 8: Dynamic tree

5 Applications of the minimum flow

In this section we present the scheduling jobs on identical machines. This problem has many practical applications, where *machines* might be workers, tankers, airplanes, trucks, processors etc. Three of such examples are treated here as subsections. Another interesting applications are presented in main works like [3] or recent publications like [12].

Let X be a set of jobs which are to be processed by a set of identical machines Y . Each job $x_i \in X$ is processed by one machine $y_j \in Y$. There is a fix schedule for the jobs, specifying that the job $x_i \in X$ must start at time $\tau(x_i)$ and finish at time $\tau'(x_i)$. Furthermore, there is a transition time $\tau''(x_i, y_j)$ required to set up a machine which has just performed the job x_i and will perform the job y_j . The goal is to find a feasible schedule for the jobs which requires as few machines as possible.

We can formulate this problem as a minimum flow problem in a network G . This network contains a node for each job $x_i \in X, i = 1, \dots, k$. We split each node x_i into two nodes x'_i and x''_i and add the arc (x'_i, x''_i) . We also add a source node s and a sink node t . We connect the source node to the nodes $x'_i \in X, i = 1, \dots, k$ and each node $x''_i \in X, i = 1, \dots, k$ to the sink node. If $\tau'(x'_i) + \tau''(x''_i, x'_j) \leq \tau(x'_j)$ then we

