Maximum Flow: Augmenting Path Algorithms Comparison

By **Zealint**
TopCoder Member

With this article, we'll revisit the so-called "max-flow" problem, with the goal of making some practical analysis of the most famous augmenting path algorithms. We will discuss several algorithms with different complexity from $O(nm^2)$ to $O(nm\log U)$ and reveal the most efficient one in practice. As we will see, theoretical complexity is not a good indicator of the actual value of an algorithm.

The article is targeted to the readers who are familiar with the basics of network flow theory. If not, I'll refer them to check out [1], [2] or algorithm tutorial [5].

In the first section we remind some necessary definitions and statements of the maximum flow theory. Other sections discuss the augmenting path algorithms themselves. The last section shows results of a practical analysis and highlights the best in practice algorithm. Also we give a simple implementation of one of the algorithms.

Statement of the Problem

Suppose we have a directed network $G = (V, E)$ defined by a set V of nodes (or vertexes) and a set E of arcs (or edges). Each arc (i,j) in E has an associated nonnegative capacity u_{ij}. Also we distinguish two special nodes in G: a *source* node s and a *sink* node t. For each i in V we denote by $E(i)$ all the arcs emanating from node i. Let $U = \max u_{ij}$ by (i,j) in E. Let us also denote the number of vertexes by n and the number of edges by m.

We wish to find the maximum flow from the source node s to the sink node t that satisfies the arc capacities and mass balance constraints at all nodes. Representing the flow on arc (i,j) in E by x_{ij} we can obtain the optimization model for the maximum flow problem:

$$\text{Maximize } f(x) = \sum_{(i,j)\in E(s)} x_{ij}$$

$$\text{subject to } \sum_{\{j:\,(i,j)\in E\}} x_{ij} - \sum_{\{j:\,(j,i)\in E\}} x_{ji} = 0 \quad \forall i \in V \setminus \{s,t\}$$

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

Vector (x_{ij}) which satisfies all constraints is called *a feasible solution* or, *a flow* (it is not necessary maximal). Given a flow x we are able to construct the residual network with respect to this flow according to the following intuitive idea. Suppose that an edge (i,j) in E carries x_{ij} units of flow. We define the residual capacity of the edge (i,j) as $r_{ij} = u_{ij} - x_{ij}$. This means that we can send an additional r_{ij} units of flow from vertex i to vertex j. We can also cancel the existing flow x_{ij} on the arc if we send up x_{ij} units of flow from j to $i$over the arc (i,j).

So, given a feasible flow x we define the residual network with respect to the flow x as follows. Suppose we have a network $G = (V, E)$. A feasible solution x engenders a new (residual) network, which we define by $G_x = (V, E_x)$, where E_x is a set of residual edges corresponding to the feasible solution x.

What is E_x? We replace each arc (i,j) in E by two arcs (i,j), (j,i): the arc (i,j) has (residual) capacity $r_{ij} = u_{ij} - x_{ij}$, and the arc (j,i) has (residual) capacity $r_{ji}=x_{ij}$. Then we construct the set E_x from the new edges with a positive residual capacity.

Augmenting Path Algorithms as a whole

In this section we describe one method on which all augmenting path algorithms are being based. This method was developed by Ford and Fulkerson in 1956 [3]. We start with some important definitions.

Augmenting path is a directed path from a source node s to a sink node t in the residual network. The residual capacity of an augmenting path is the minimum residual capacity of any arc in the path. Obviously, we can send additional flow from the source to the sink along an augmenting path.

All augmenting path algorithms are being constructed on the following basic idea known as augmenting path theorem:

***Theorem 1 (Augmenting Path Theorem)**. A flow x* is a maximum flow if and only if the residual network Gx* contains no augmenting path.*

According to the theorem we obtain a method of finding a maximal flow. The method proceeds by identifying augmenting paths and augmenting flows on these paths until the network contains no such path. All algorithms that we wish to discuss differ only in the way of finding augmenting paths.

We consider the maximum flow problem subject to the following assumptions.

***Assumption 1.** The network is directed.*

***Assumption 2.** All capacities are nonnegative integers.*

This assumption is not necessary for some algorithms, but the algorithms whose complexity bounds involve U assume the integrality of the data.

***Assumption 3.** The problem has a bounded optimal solution.*

This assumption in particular means that there are no uncapacitated paths from the source to the sink.

***Assumption 4.** The network does not contain parallel arcs.*

This assumption imposes no loss of generality, because one can summarize capacities of all parallel arcs.

As to why these assumptions are correct we leave the proof to the reader.

It is easy to determine that the method described above works correctly. Under assumption 2, on each augmenting step we increase the flow value by at least one unit. We (usually) start with zero flow. The maximum flow value is bounded from above, according to assumption 3. This reasoning implies the finiteness of the method.

With those preparations behind us, we are ready to begin discussing the algorithms.

Shortest Augmenting Path Algorithm, $O(n^2m)$

In 1972 Edmonds and Karp -- and, in 1970, Dinic -- independently proved that if each augmenting path is shortest one, the algorithm will perform $O(nm)$ augmentation steps. The shortest path (length of each edge is equal to one) can be found with the help of breadth-first search (BFS) algorithm [2], [6]. Shortest Augmenting Path Algorithm is well known and widely discussed in many books and articles, including [5], which is why we will not describe it in great detail. Let's review the idea using a kind of pseudo-code:

```
SHORTEST—AUGMENTING—PATH
1  x ← 0
2  while in G_x exists path s ⇝ t
3        do find a shortest augmenting path P
4           δ ← min{r_ij : (i, j) ∈ P}
5           augment δ units of flow along P
6           update G_x
7  return x
```

In line 5, current flow x is being increased by some positive amount.

The algorithm was said to perform $O(nm)$ steps of finding an augmenting path. Using BFS, which requires $O(m)$ operation in the worst case, one can obtain $O(nm^2)$complexity of the algorithm itself. If $m \sim n2$ then one must use BFS procedure $O(n^3)$ times in worst case. There are some networks on which this numbers of augmentation steps is being achieved. We will show one simple example below.

Improved Shortest Augmenting Path Algorithm, $O(n^2m)$

As mentioned earlier, the natural approach for finding any shortest augmenting path would be to look for paths by performing a breadth-first search in the residual network. It requires $O(m)$ operations in the worst case and imposes $O(nm^2)$ complexity of the maximum flow algorithm. Ahuja and Orlin improved the shortest augmenting path algorithm in 1987 [1]. They exploited the fact that the minimum distance from any node i to the sink node t is

monotonically nondecreasing over all augmentations and reduced the average time per augmentation to $O(n)$. The improved version of the augmenting path algorithm, then, runs in $O(n^2m)$ time. We can now start discussing it according to [1].

Definition 1. *Distance function d: V_ Z+ with respect to the residual capacities rij is a function from the set of nodes to nonnegative integers. Let's say that distance function is valid if it is satisfies the following conditions:*

- *d(t)=0;*
- *d(i) ≤ d(j) + 1, for every (i,j) in E with rij>0.*

Informally (and it is easy to prove), valid distance label of node i, represented by $d(i)$, is a lower bound on the length of the shortest path from i to t in the residual network Gx. We call distance function *exact* if each i in V $d(i)$ equals the length of the shortest path from i to t in the residual network. It is also easy to prove that if $d(s) \geq n$ then the residual network contains no path from the source to the sink.

An arc (i,j) in E is called admissible if $d(i) = d(j) + 1$. We call other arcs *inadmissible*. If a path from s to t consists of admissible arcs then the path is admissible. Evidently, an admissible path is the shortest path from the source to the sink. As far as every arc in an admissible path satisfies condition $r_{ij}>0$, the path is augmenting.

So, the improved shortest augmenting path algorithm consists of four steps (procedures): *main cycle, advance, retreat* and *augment*. The algorithm maintains a *partial admissible path*, i.e., a path from s to some node i, consisting of admissible arcs. It performs *advance* or *retreat* steps from the last node of the partial admissible path (such node is called *current node*). If there is some admissible arc (i,j) from current node i, then the algorithm performs the *advance* step and adds the arc to the partial admissible path. Otherwise, it performs the *retreat* step, which increases distance label of i and backtracks by one arc.

If the partial admissible path reaches the sink, we perform an augmentation. The algorithm stops when $d(s) \geq n$. Let's describe these steps in pseudo-code [1]. We denoted residual (with respect to flow x) arcs emanating from node i by $E_x(i)$. More formally, $E_x(i) = \{ (i,j) \text{ in } E(i): r_{ij} > 0 \}$.

```
algorithm IMPROVED—SHORTEST—AUGMENTING—PATH
 1  x ← 0
 2  Perform a (reverse) breadth-first search of the residual network
    starting with the sink node to compute exact distance labels d(i)
 3  i ← s
 4  while d(s) < n
 5        do if G_x contains an admissible arc (i, j) ∈ E_x(i)
 6              then ADVANCE(i)
 7                   if i = t                ▷ We found an augmenting path
 8                     then AUGMENT
 9                          i ← s            ▷ Begin finding next path
10              else  RETREAT(i)      ▷ No admissible arc emanating from i
11  return x

procedure ADVANCE(i)
1  let (i, j) be an admissible arc in E_x(i)
2  π(j) ← i     ▷ We maintain the predecessor list
3  i ← j        ▷ Make node j as current node

procedure RETREAT(i)
1  d(i) ← 1 + min{d(j) : (i, j) ∈ E_x(i)}  ▷ This operation is called relabel
2  if i ≠ s
3     then i ← π(i)    ▷ Backtrack

procedure AUGMENT
1  using the predecessor indices π identity an augmenting path P
2  δ ← min{r_ij : (i, j) ∈ P}
3  augment δ units of flow along P
4  update G_x (or, E_x)
```

In line 1 of *retreat* procedure if $E_x(i)$ is empty, then suppose $d(i)$ equals n.

Ahuja and Orlin suggest the following data structure for this algorithm [1]. We maintain the arc list $E(i)$ which contains all the arcs emanating from node i. We arrange the arcs in this list in any fixed order. Each node i has a *current arc*, which is an arc in $E(i)$ and is the next candidate for admissibility testing. Initially, the current arc of node i is the first arc in $E(i)$. In line 5 the algorithm tests whether the node's current arc is admissible. If

not, it designates the next arc in the list as the current arc. The algorithm repeats this process until either it finds an admissible arc or reaches the end of the arc list. In the latter case the algorithm declares that $E(i)$ contains no admissible arc; it again designates the first arc in $E(i)$ as the current arc of node i and performs the *relabel* operation by calling the *retreat* procedure (line 10).

Now we outline a proof that the algorithm runs in $O(n^2m)$ time.

Lemma 1. *The algorithm maintains distance labels at each step. Moreover, each relabel (or, retreat) step strictly increases the distance label of a node.*

Sketch to proof. Perform induction on the number of *relabel* operation and augmentations.

Lemma 2. *Distance label of each node increases at most n times. Consecutively, relabel operation performs at most n^2 times.*

Proof. This lemma is consequence of lemma 1 and the fact that if $d(s) \geq n$ then the residual network contains no augmenting path.

Since the improved shortest augmenting path algorithm makes augmentations along the shortest paths (like unimproved one), the total number of augmentations is the same$O(nm)$. Each *retreat* step relabels a node, that is why number of *retreat* steps is $O(n^2)$ (according to lemma 2). Time to perform *retreat/relabel* steps is $O(n \sum_{i \text{ in } V} |E(i)|) = O(nm)$. Since one augmentation requires $O(n)$ time, total augmentation time is $O(n^2m)$. The total time of *advance* steps is bounded by the augmentation time plus the*retreat/relabel* time and it is again $O(n^2m)$. We obtain the following result:

Theorem 2. *The improved shortest augmenting path algorithm runs in $O(n^2m)$ time.*

Ahuja and Orlin [1] suggest one very useful practical improvement of the algorithm. Since the algorithm performs many useless relabel operations while the maximum flow has been found, it will be better to give an additional criteria of terminating. Let's introduce (n+1)-dimensional additional array, *numbs*, whose indices vary from 0 to n. The value*numbs(k)* is the number of nodes whose distance label equals *k*. The algorithm initializes this array while computing the initial distance labels using BFS. At this point, the positive entries in the array *numbs* are consecutive (i.e., *numbs(0), numbs(1), ..., numbs(l)* will be positive up to some index l and the remaining entries will all be zero).

When the algorithm increases a distance label of a node from x to y, it subtracts 1 from *numbs(x)*, adds 1 to *numbs(y)* and checks whether *numbs(x) = 0*. If it does equal 0, the algorithm terminates.

This approach is some kind of heuristic, but it is really good in practice. As to why this approach works we leave the proof to the reader (*hint*: show that the nodes i with $d(i) > x$and nodes j with $d(j) < x$ engender a cut and use maximum-flow-minimum-cut theorem).

Comparison of Improved and Unimproved versions

In this section we identify the worst case for both shortest augmenting path algorithms with the purpose of comparing their running times.

In the worst case both improved and unimproved algorithms will perform $O(n^3)$ augmentations, if $m \sim n2$. Norman Zadeh [4] developed some examples on which this running time is based. Using his ideas we compose a somewhat simpler network on which the algorithms have to perform $O(n^3)$ augmentations and which is not dependent on a choice of next path.

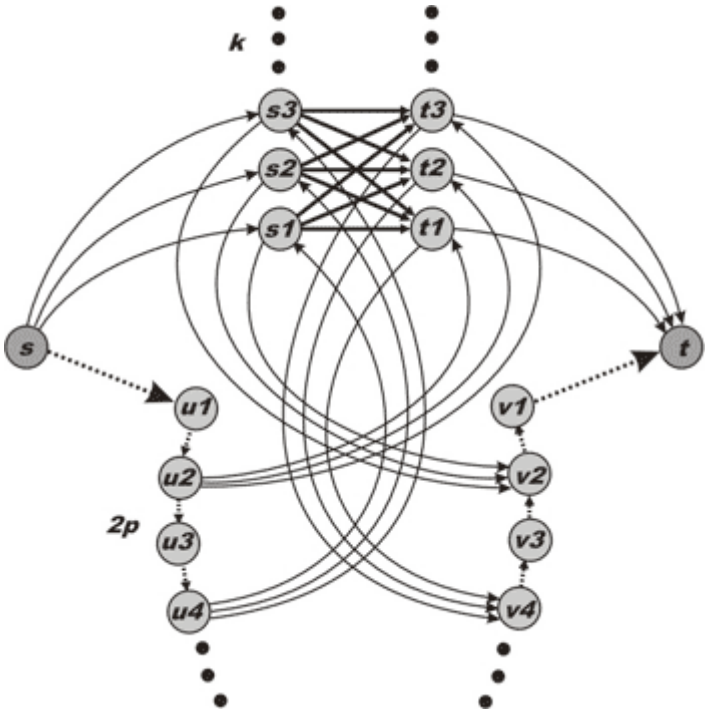

Figure 1. Worst case example for the shortest augmenting path algorithm.

All vertexes except s and t are divided into four subsets: $S=\{s1,...,sk\}$, $T=\{t1,...,tk\}$, $U=\{u1,...,u2p\}$ and $V=\{v1,...,v2p\}$. Both sets S and T contain k nodes while both sets $U$and V contain $2p$ nodes. k and p are fixed integers. Each bold arc (connecting S and T) has unit capacity. Each dotted arc has an infinite capacity. Other arcs (which are solid and not straight) have capacity k.

First, the shortest augmenting path algorithm has to augment flow k^2 time along paths (s, S, T, t) which have length equal to 3. The capacities of these paths are unit. After that the residual network will contain reversal arcs (T, S) and the algorithm will chose another k^2 augmenting paths (s, $u1$, $u2$, T, S, $v2$, $v1$, t) of length 7. Then the algorithm will have to choose paths (s, $u1$, $u2$, $u3$, $u4$, S, T, $v4$, $v3$, $v2$, $v1$, t) of length 11 and so on...

Now let's calculate the parameters of our network. The number of vertexes is $n = 2k + 4p + 2$. The number of edges is $m = k^2 + 2pk + 2k + 4p$. As it easy to see, the number of augmentations is $a = k2\ (p+1)$.

Consider that $p = k - 1$. In this case $n = 6k - 2$ and $a = k^3$. So, one can verify that $a \sim n^3 / 216$. In [4] Zadeh presents examples of networks that require $n^3 / 27$ and $n^3 / 12$augmentations, but these examples are dependent on a choice of the shortest path.

We made 5 worst-case tests with 100, 148, 202, 250 and 298 vertexes and compared the running times of the improved version of the algorithm against the unimproved one. As you can see on figure 2, the improved algorithm is much faster. On the network with 298 nodes it works 23 times faster. Practice analysis shows us that, in general, the improved algorithm works n / 14 times faster.

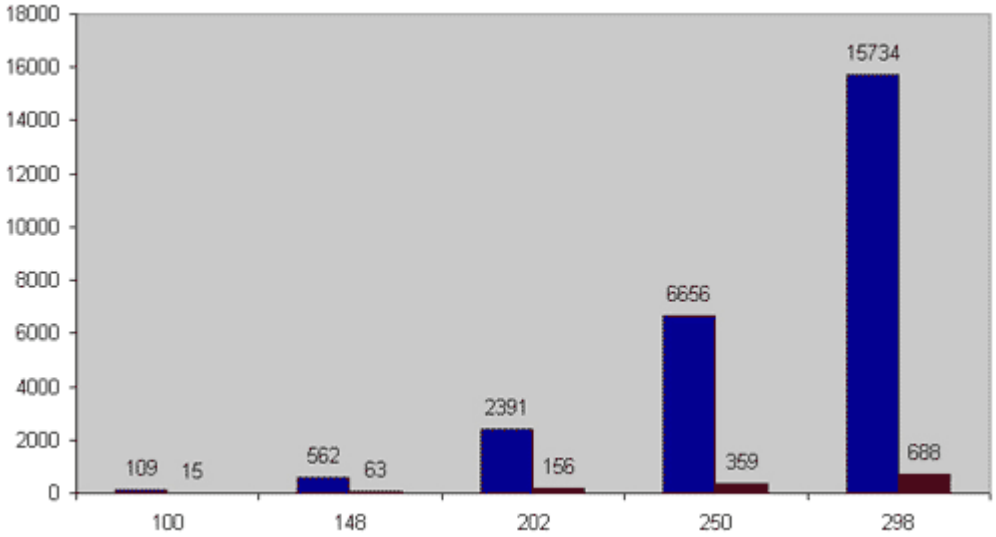

Figure 2. X-axis is the number of nodes. Y-axis is working time in milliseconds.
Blue colour indicates the shortest augmenting path algorithm and red does it improved version.

However, our comparison in not definitive, because we used only one kind of networks. We just wanted to justify that the $O(n^2m)$ algorithm works $O(n)$ times faster than the$O(nm^2)$ on a dense network. A more revealing comparison is waiting for us at the end of the article.

Maximum Capacity Path Algorithm, $O(n^2m\log nU)$ / $O(m^2 \log nU \log n)$ / $O(m^2 \log nU \log U)$

In 1972 Edmonds and Karp developed another way to find an augmenting path. At each step they tried to increase the flow with the maximum possible amount. Another name of this algorithm is "gradient modification of the Ford-Fulkerson method." Instead of using BFS to identify a shortest path, this modification uses Dijkstra's algorithm to establish a path with the maximal possible capacity. After augmentation, the algorithm finds another such path in the residual network, augments flow along it, and repeats these steps until the flow is maximal.

```
MAXIMUM—CAPACITY—PATH
1  x ← 0
2  while in G_x exists path s ⇝ t
3        do find an augmenting path P with the maximum capacity
4           δ ← min{r_ij : (i, j) ∈ P}
5           augment δ units of flow along P
6           update G_x
7  return x
```

There's no doubt that the algorithm is correct in case of integral capacity. However, there are tests with non-integral arc's capacities on which the algorithm may fail to terminate.

Let's get the algorithm's running time bound by starting with one lemma. To understand the proof one should remember that the value of any flow is less than or equal to the capacity of any cut in a network, or read this proof in [1], [2]. Let's denote capacity of a cut (S,T) by $c(S,T)$.

Lemma 3. *Let F be the maximum flow's value, then G contains augmenting path with capacity not less than F/m.*

Proof. Suppose G contains no such path. Let's construct a set $E'=\{ (i,j)\ in\ E:\ u_{ij} \geq F/m \}$. Consider network $G' = (V, E')$ which has no path from s to t. Let S be a set of nodes obtainable from s in G and $T = V \setminus S$. Evidently, (S, T) is a cut and $c(S, T) \geq F$. But cut (S, T) intersects only those edges (i,j) in E which have $u_{ij} < F/m$. So, it is clear that

$$c(S,T) < (F/m) \_ m = F,$$

and we got a contradiction with the fact that $c(S,T) \geq F$.

Theorem 3. *The maximum capacity path algorithm performs O(m log (nU)) augmentations.*

Sketch to proof. Suppose that the algorithm terminates after k augmentations. Let's denote by f_1 the capacity of the first found augmentation path, by f_2 the capacity of the second one, and so on. f_k will be the capacity of the latter k-th augmenting path.

Consider, $F_i = f_1 + f_2 + ... + f_i$. Let F^* be the maximum flow's value. Under lemma 3 one can justify that

$$f_i \geq (F^*-F_{i-1}) / m.$$

Now we can estimate the difference between the value of the maximal flow and the flow after i consecutive augmentations:

$$F^* - F_i = F^* - F_{i-1} - f_i \leq F^* - F_{i-1} - (F^* - F_{i-1}) / m = (1 - 1 / m) (F^* - F_{i-1}) \leq ... \leq (1 - 1 / m)^i \_ F^*$$

We have to find such an integer i, which gives $(1 - 1 / m)^i \_ F^* < 1$. One can check that

$$i \log_{m / (m+1)} F^* = O(m \_ \log F^*) = O(m \_ \log(nU))$$

And the latter inequality proves the theorem.

To find a path with the maximal capacity we use Dijkstra's algorithm, which incurs additional expense at every iteration. Since a simple realization of Dijkstras's algorithm [2] incurs $O(n^2)$ complexity, the total running time of the maximum capacity path algorithm is $O(n^2 m \log(nU))$.

Using a heap implementation of Dijkstra's algorithm for sparse network [7] with running time $O(m \log n)$, one can obtain an $O(m^2 \log n \log(nU))$ algorithm for finding the maximum flow. It seems to be better that the improved Edmonds-Karp algorithm. However, this estimate is very deceptive.

There is another variant to find the maximum capacity path. One can use binary search to establish such a path. Let's start by finding the maximum capacity path on piece $[0,U]$. If there is some path with capacity $U/2$, then we continue finding the path on piece $[U/2, U]$; otherwise, we try to find the path on $[0,U/2-1]$. This approach incurs additional $O(m \log U)$ expense and gives $O(m^2 \log(nU) \log U)$ time bound to the maximum flow algorithm. However, it works really poorly in practice.

Capacity Scaling Algorithm, $O(m^2 \log U)$

In 1985 Gabow described the so-called "bit-scaling" algorithm. The similar capacity scaling algorithm described in this section is due to Ahuja and Orlin [1].

Informally, the main idea of the algorithm is to augment the flow along paths with sufficient large capacities, instead of augmenting along maximal capacities. More formally, let's introduce a parameter *Delta*. First, Delta is quite a large number that, for instance, equals U. The algorithm tries to find an augmenting path with capacity not less that Delta, then augments flow along this path and repeats this procedure while any such Delta-path exists in the residual network.

The algorithm either establishes a maximum flow or reduces Delta by a factor of 2 and continues finding paths and augmenting flow with the new Delta. The phase of the algorithm that augments flow along paths with capacities at least Delta is called *Delta-scaling phase* or, *Delta-phase*. Delta is an integral value and, evidently, algorithm performs $O(\log U)$ Delta-phases. When Delta is equal to 1 there is no difference between the capacity scaling algorithm and the Edmonds-Karp algorithm, which is why the algorithm works correctly.

```
CAPACITY—SCALING
1  x ← 0
2  Δ ← 2^⌊log2 U⌋
3  while Δ > 0
4        do while in G_x exists path s ⇝ t with the capacity of at least Δ
5              do find an augmenting path P with the capacity of at least Δ
6                 δ ← min{r_ij : (i, j) ∈ P}
7                 augment δ units of flow along P
8                 update G_x
9           Δ ← Δ/2          ▷ Integer division
10 return x
```

We can obtain a path with the capacity of at least Delta fairly easily - in $O(m)$ time (by using BFS). At the first phase we can set Delta to equal either U or the largest power of 2 that doesn't exceed U.

The proof of the following lemma is left to the reader.

Lemma 4. *At every Delta-phase the algorithm performs O(m) augmentations in worst case.*

Sketch to proof. Use the induction by Delta to justify that the minimum cut at each Delta-scaling phase less that $2m\ Delta$.

Applying lemma 4 yields the following result:

Theorem 4. *Running time of the capacity scaling algorithm is $O(m^2 logU)$.*

Keep in mind that there is no difference between using breadth-first search and depth-first search when finding an augmenting path. However, in practice, there is a big difference, and we will see it.

Improved Capacity Scaling Algorithm, $O(nm\log U)$

In the previous section we described an $O(m^2 logU)$ algorithm for finding the maximum flow. We are going to improve the running time of this algorithm to $O(nm\log U)$ [1].

Now let's look at each Delta-phase independently. Recall from the preceding section that a Delta-scaling phase contains $O(m)$ augmentations. Now we use the similar technique at each Delta-phase that we used when describing the improved variant of the shortest augmenting path algorithm. At every phase we have to find the "maximum" flow by using only paths with capacities equal to at least Delta. The complexity analysis of the improved shortest augmenting path algorithm implies that if the algorithm is guaranteed to perform $O(m)$ augmentations, it would run in $O(nm)$ time because the time for augmentations reduces from $O(n^2 m)$ to $O(nm)$ and all other operations, as before, require $O(nm)$ time. These reasoning instantly yield a bound of $O(nm\log U)$ on the running time of the improved capacity scaling algorithm.

Unfortunately, this improvement hardly decreases the running time of the algorithm in practice.

Practical Analysis and Comparison

Now let's have some fun. In this section we compare all described algorithms from practical point of view. With this purpose I have made some test cases with the help of [8] and divided them into three groups by density. The first group of tests is made of networks with $m \le n1.4$ - some kind of sparse networks. The second one consists of middle density tests with $n1.6 \le m \le n1.7$. And the last group represents some kinds of almost full graphs (including full acyclic networks) with $m \ge n1.85$.

I have simple implementations of all algorithms described before. I realized these algorithms without any kind of tricks and with no preference towards any of them. All implementations use adjacency list for representing a network.

Let's start with the first group of tests. These are 564 sparse networks with number of vertexes limited by 2000 (otherwise, all algorithms work too fast). All working times are given in milliseconds.

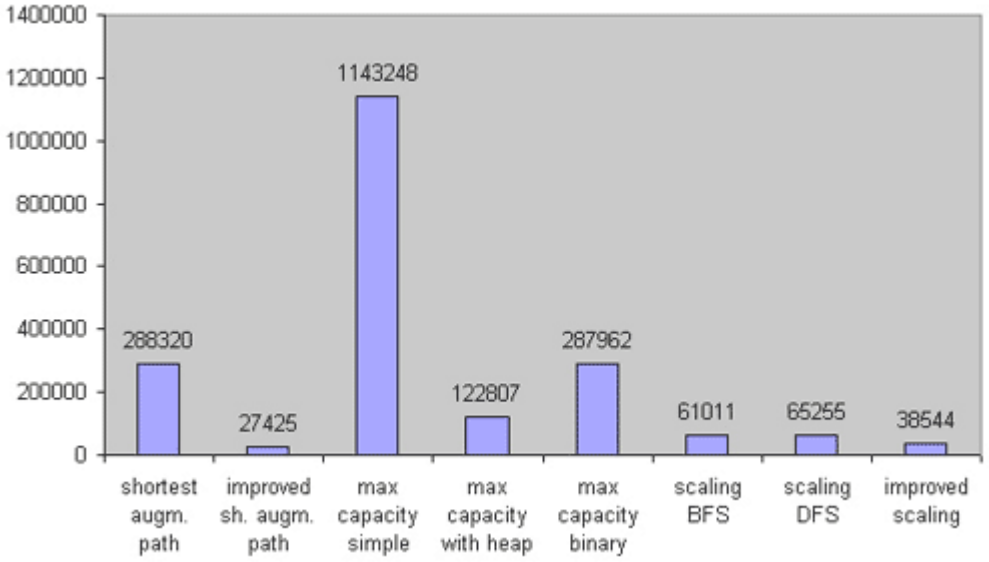

Figure 3. Comparison on sparse networks. 564 test cases. $m \le n1.4$.

As you can see, it was a big mistake to try Dijkstra's without heap implementation of the maximum capacity path algorithm on sparse networks (and it's not surprising); however, its heap implementation works rather faster than expected. Both the capacity scaling algorithms (with using DFS and BFS) work in approximately the same time, while the improved implementation is almost 2 times faster. Surprisingly, the improved shortest path algorithm turned out to be the fastest on sparse networks.

Now let's look at the second group of test cases. It is made of 184 tests with middle density. All networks are limited to 400 nodes.

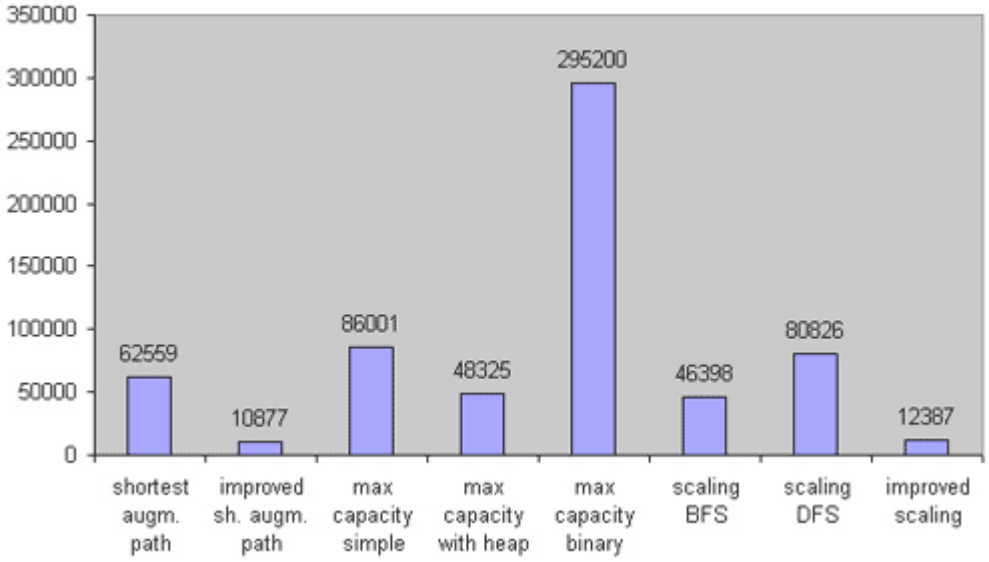

Figure 4. Comparison on middle density networks. 184 test cases. $n1.6 \le m \le n1.7$.

On the middle density networks the binary search implementation of the maximum capacity path algorithm leaves much to be desired, but the heap implementation still works faster than the simple (without heap) one. The BFS realization of the capacity scaling algorithm is faster than the DFS one. The improved scaling algorithm and the improved shortest augmenting path algorithm are both very good in this case.

It is very interesting to see how these algorithms run on dense networks. Let's take a look -- the third group is made up of 200 dense networks limited by 400 vertexes.

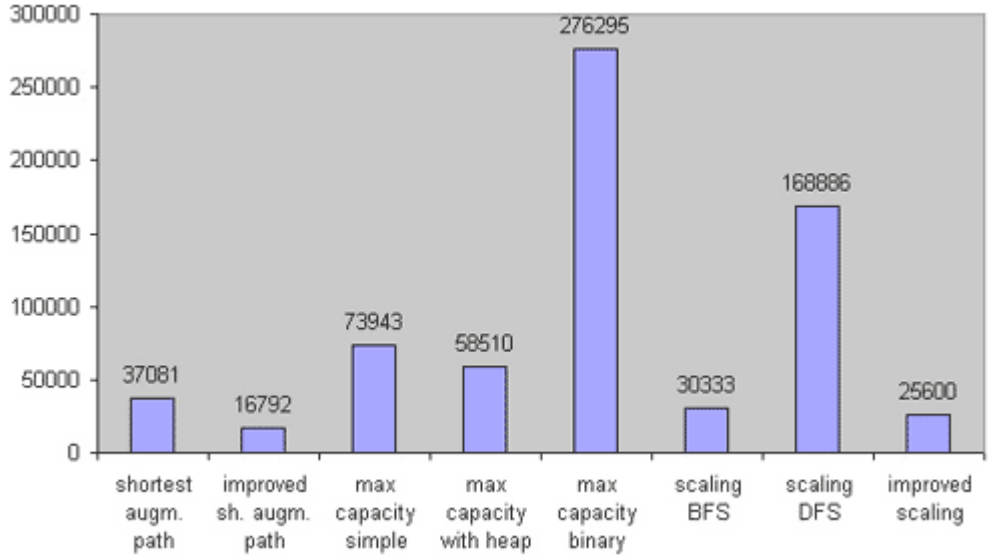

Figure 5. Comparison on dense networks. 200 test cases. $m \geq n1.85$.

Now we see the difference between the BFS and DFS versions of the capacity scaling algorithm. It is interesting that the improved realization works in approximately the same time as the unimproved one. Unexpectedly, the Dijkstra's with heap implementation of the maximum capacity path algorithm turned out to be faster than one without heap.

Without any doubt, the improved implementation of Edmonds-Karp algorithm wins the game. Second place is taken by the improved scaling capacity algorithm. And the scaling capacity with BFS got bronze.

As to maximum capacity path, it is better to use one variant with heap; on sparse networks it gives very good results. Other algorithms are really only good for theoretical interest.

As you can see, the $O(nm\log U)$ algorithm isn't so fast. It is even slower than the $O(n^2m)$ algorithm. The $O(nm^2)$ algorithm (it is the most popular) has worse time bounds, but it works much faster than most of the other algorithms with better time bounds.

My recommendation: Always use the scaling capacity path algorithm with BFS, because it is very easy to implement. The improved shortest augmenting path algorithm is rather easy, too, but you need to be very careful to write the program correctly. During the contest it is very easy to miss a bug.

I would like to finish the article with the full implementation of the improved shortest augmenting path algorithm. To maintain a network I use the adjacency matrix with purpose to providing best understanding of the algorithm. It is not the same realization what was used during our practical analysis. With the "help" of the matrix it works a little slower than one that uses adjacency list. However, it works faster on dense networks, and it is up to the reader which data structure is best for them.

```
#include <stdio.h>



#define N 2007  // Number of nodes

#define oo 1000000000 // Infinity



// Nodes, Arcs, the source node and the sink node

int n, m, source, sink;



// Matrixes for maintaining

// Graph and Flow

int G[N][N], F[N][N];
```

```
int pi[N];  // predecessor list

int CurrentNode[N]; // Current edge for each node



int queue[N];  // Queue for reverse BFS



int d[N];  // Distance function

int numbs[N];  // numbs[k] is the number of nodes i with d[i]==k



// Reverse breadth-first search

// to establish distance function d

int rev_BFS()  {

  int i, j, head(0), tail(0);



  // Initially, all d[i]=n

  for(i = 1; i <= n; i++)

    numbs[ d[i] = n ] ++;



  // Start from the sink

  numbs[n]--;

  d[sink] = 0;

  numbs[0]++;



  queue[ ++tail ] = sink;



  // While queue is not empty

  while( head != tail )  {

    i = queue[++head];  // Get the next node



    // Check all adjacent nodes

    for(j = 1; j <= n; j++)  {



      // If it was reached before or there is no edge

      // then continue

      if(d[j] < n || G[j][i] == 0)  continue;
```

```
      // j is reached first time
      // put it into queue
      queue[ ++tail ] = j;

      // Update distance function
      numbs[n]--;
      d[j] = d[i] + 1;
      numbs[d[j]]++;

    }
  }

  return 0;
}

// Augmenting the flow using predecessor list pi[]
int Augment()  {
  int i, j, tmp, width(oo);

  // Find the capacity of the path
  for(i = sink, j = pi[i]; i != source; i = j, j = pi[j])  {
    tmp = G[j][i];
    if(tmp < width)  width = tmp;
  }

  // Augmentation itself
  for(i = sink, j = pi[i]; i != source; i = j, j = pi[j])  {
    G[j][i] -= width;  F[j][i] += width;
    G[i][j] += width;  F[i][j] -= width;
  }

  return width;
}
```

```
// Relabel and backtrack
int Retreat(int &i)  {
  int tmp;
  int j, mind(n-1);

  // Check all adjacent edges
  // to find nearest
  for(j=1; j <= n; j++)
    // If there is an arc
    // and j is "nearer"
    if(G[i][j] > 0 && d[j] < mind)
      mind = d[j];

  tmp = d[i];  // Save previous distance

  // Relabel procedure itself
  numbs[d[i]]--;
  d[i] = 1 + mind;
  numbs[d[i]]++;

  // Backtrack, if possible (i is not a local variable! )
  if( i != source )  i = pi[i];

  // If numbs[ tmp ] is zero, algorithm will stop
  return numbs[ tmp ];
}

// Main procedure
int find_max_flow()  {
  int flow(0), i, j;

  rev_BFS();  // Establish exact distance function
```

```
  // For each node current arc is the first arc
  for(i=1; i<=n; i++)  CurrentNode[i] = 1;

  // Begin searching from the source
  i = source;

  // The main cycle (while the source is not "far" from the sink)
  for( ; d[source] < n ; )  {

    // Start searching an admissible arc from the current arc
    for(j = CurrentNode[i]; j <= n; j++)
      // If the arc exists in the residual network
      // and if it is an admissible
      if( G[i][j] > 0 && d[i] == d[j] + 1 )
        // Then finish searhing
        break;

    // If the admissible arc is found
    if( j <= n )  {
      CurrentNode[i] = j;  // Mark the arc as "current"
      pi[j] = i;  //  j is reachable from i
      i = j;  // Go forward

      // If we found an augmenting path
      if( i == sink )  {
        flow += Augment();  // Augment the flow
        i = source;  // Begin from the source again
      }
    }
    // If no an admissible arc found
    else  {
      CurrentNode[i] = 1;  // Current arc is the first arc again

      // If numbs[ d[i] ] == 0  then the flow is the maximal
```

```
      if( Retreat(i) == 0 )
        break;

    }

  } // End of the main cycle

  // We return flow value
  return flow;
}



// The main function
// Graph is represented in input as triples <from, to, capacity>

// No comments here
int main()  {
  int i, p, q, r;

  scanf("%d %d %d %d", &n, &m, &source, &sink);

  for(i = 0; i < m; i++)  {
    scanf("%d %d %d", &p, &q, &r);
    G[p][q] += r;
  }

  printf("%d", find_max_flow());

  return 0;
}
```

A New Approach to the Maximum Flow Problem

By **NilayVaish**
TopCoder Member

Introduction

This article presents a new approach for computing maximum flow in a graph. Previous articles had concentrated on finding maximum flow by finding augmenting paths. **Ford-Fulkerson** and **Edmonds-Karp** algorithms belong to that class. The approach presented in this article is called **push-relabel**, which is a separate class of algorithms. We'll look at an algorithm first described by **Andrew V. Goldberg** and **Robert E. Tarjan**, which is not very hard to code and, for dense graphs, is much faster than the augmenting path algorithms. If you haven't yet done so, I'd advise you to review the articles on graph theory and maximum flow using augmenting paths for a better understanding of the basic concepts on the two topics.

The Standard Maximum Flow Problem

Let **G = (V,E)** be a directed graph with vertex set **V** and edge set **E**. Size of set **V** is **n** and size of set **E** is **m**. **G** has two distinguished vertices, a source **s** and a sink **t**. Each edge**(u,v) ε E** has a capacity **c(u,v)**. For all edges **(u,v) ∉E**, we define **c(u,v) = 0**. A flow f on **G** is a real valued function satisfying the following constraints:

1. **Capacity: f(v,w) ≤ c(v,w) ∨ (v,w) ∈ V × V**
2. **Anti-symmetry: f(v,w) = – f(w,v) ∨ (v,w) ∈ V × V**
3. **Flow Conservation: ∑ u ∈ V f(u,v) = 0 ∨ v ∈ V - {s,t}**

The value of a flow f is the net flow into the sink i.e. **| f | = ∑ u ∈ V f(u,t)** . **Figure 1** below shows a flow network with the edges marked with their capacities. Using this network we will illustrate the steps in the algorithm.

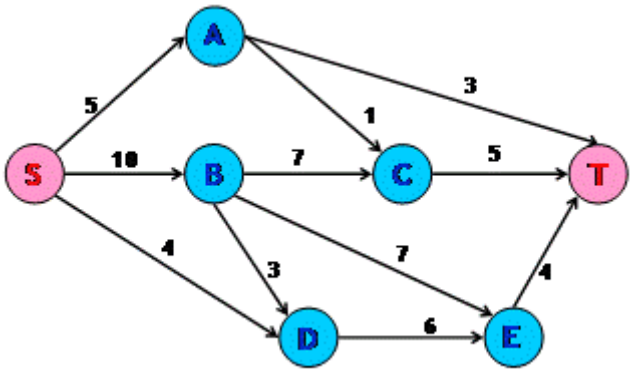

Figure 1 : Flow Network with Capacities

Intuitive Approach Towards the Algorithm

Assume that we have a network of water tanks connected with pipes in which we want to find the maximum flow of water from the source tank to the sink tank. Each of these water tanks are arbitrarily large and will be used for accumulating water. A tank is said to be overflowing if it has water in it. Tanks are at a height from the ground level. The edges can be assumed to be pipes connecting these water tanks. The natural action of water is to flow from a higher level to a lower level. The same holds for this algorithm. The height of the water tank determines the direction in which water will flow. We can push **new flow** from a tank to another tank that is downhill from the first tank, i.e. to tanks that are at a lesser height than the first tank. We need to note one thing here, however:**The flow from a lower tank to a higher tank might be positive.** Present height of a tank only determines the direction of new flow.

We fix the initial height of the source **s** at **n** and that of sink **t** at **0**. All other tanks have initial height **0**, which increases with time. Now send as much as possible flow from the source toward the sink. Each outgoing pipe from the source s is filled to capacity. We will now examine the tanks other than **s** and **t**. The flow from overflowing tanks is pushed downhill. If an overflowing tank is at the same level or below the tanks to which it can push flow, then this tank is raised just enough so that it can push more flow. If the sink t is not reachable from an overflowing tank, we then send this excess water back to the source. This is done by raising the overflowing tank the fixed height **n** of the source. Eventually all the tanks except the source **s** and the sink **t** stop overflowing. At this point the flow from the source to the sink is actually the max-flow.

Mathematical Functions

In this section we'll examine the mathematical notations required for better understanding of the algorithm.

1. **Preflow** - Intuitively preflow function gives the amount of water flowing through a pipe. It is similar to the flow function. Like flow, it is a function **f: V × V → R**. It also follows the capacity and anti-symmetry constraints. But for the preflow function, the conservation constraint is weakened.

   $$\sum u \in V \, f(u,v) \geq 0 \; \forall \; v \in V - \{s,t\}$$

   That is the total net flow at a vertex other than the source that is non-negative. During the course of the algorithm, we will manipulate this function to achieve the maximum flow.

2. **Excess Flow** - We define the excess flow **e** as **e(u) = f(V,u)**, the net flow into **u**. A vertex **u ∈ V-{s,t}** is overflowing / active if **e(u) > 0**.
3. **Residual Capacity** - We define residual capacity as function **cf: V × V → R** where

   $$cf(u,v) = c(u,v) - f(u,v)$$

   If **cf(u,v) > 0**, then **(u,v)** is called a **residual edge**. Readers would have come across this concept in augmenting path algorithms as well.

4. **Height** - This function is defined as **h: V → N**. It denotes the height of a water tank. It has the following properties -
   - **h(s) = n**
   - **h(t) = 0**
   - **h(u) ≤ h(v) + 1 for every residual edge (u,v).**

   We will prove the last property while discussing the correctness of the algorithm.

Basic Operations

Following are the three basic functions that constitute the algorithm:

1. **Initialization** – This is carried out once during the beginning of the algorithm. The height for all the vertices is set to zero except the source for which the height is kept at**n**. The preflow and the excess flow functions for all vertices are set to zero. Then all the edges coming out the source are filled to capacity. **Figure 2** shows the network after initialization.

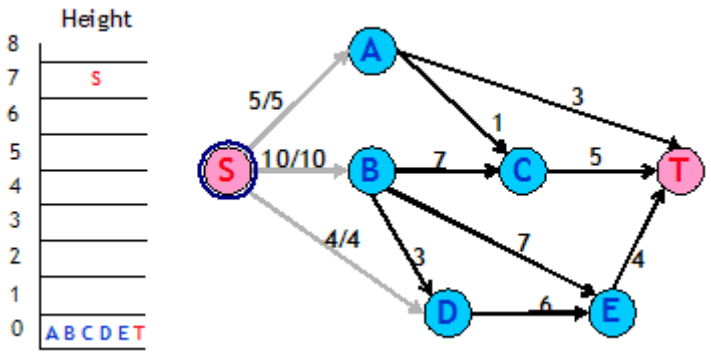

Figure 2 : Network after Initialization

```
void initialize_preflow() {
  memset(h,0,sizeof(int)*n);
  h[s] = n;
  memset(e,0,sizeof(int)*n);
  for(int i=n-1;i>=0;--i)
    memset(f[i],0,sizeof(int)*n);
  for(int i=0;i<G[s].size();++i) {
    int v = G[s][i];
    f[s][v] = c[s][v];
    f[v][s] =-c[s][v];
    e[v] = c[s][v];
    e[s] -= c[s][v];
    cf[s][v] = c[s][v] - f[s][v];
    cf[v][s] = c[v][s] - f[v][s];
  }
}
```

Figure 3 : Code for Initialization

2. **push(u,v)** - This operation pushes flow from an overflowing tank to a tank to which it has a pipe that can take in more flow and the second tank is at a lower height than the first one. In other words, if vertex **u** has a positive excess flow, **cf(u,v) > 0** and **h(u) > h(v)**, then flow can be pushed onto the residual edge **(u,v)**. The amount of the flow pushed is given by **min(e(u),cf(u,v))**.

   Figure 4 shows a vertex **B** that has an excess flow of **10**. It makes two pushes. In the first push, **7** units of flow are pushed to **C**. In the second push, **3** units of flow are pushed to **E. Figure 5** illustrates the final result.

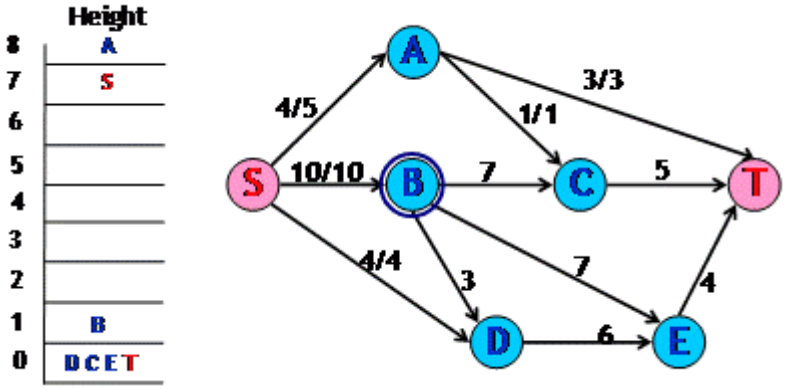

Figure 4 : Network before pushing flow from B

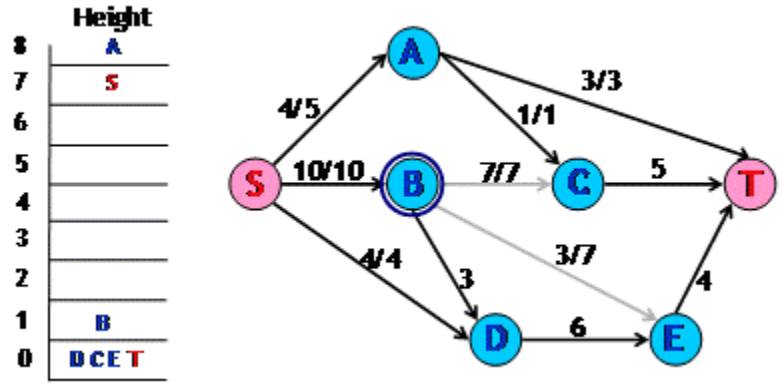

Figure 5 : Network pushing flow from B

```
void push(int u,int v) {
  temp = min(e(u),cf[u][v]);
  f[u][v] = f[u][v] + temp;
  f[v][u] = -f[u][v];
  e[u] = e[u] - temp;
  e[v] = e[v] + temp;
  cf[u][v] = c[u][v] - f[u][v];
  cf[v][u] = c[v][u] - f[v][u];
}
```

Figure 6 : Code for Push sub-routine

3. **relabel(u)** - This operation raises the height of an overflowing tank that has no other tanks downhill from it. It applies if **u** is overflowing and **h(u) ≤ h(v)** ∀ residual edges**(u,v)** i.e. on all the residual edges from **u**, flow cannot be pushed. The height of the vertex **u** is increased by **1** more than the minimum height of its neighbor to which **u**has a residual edge.

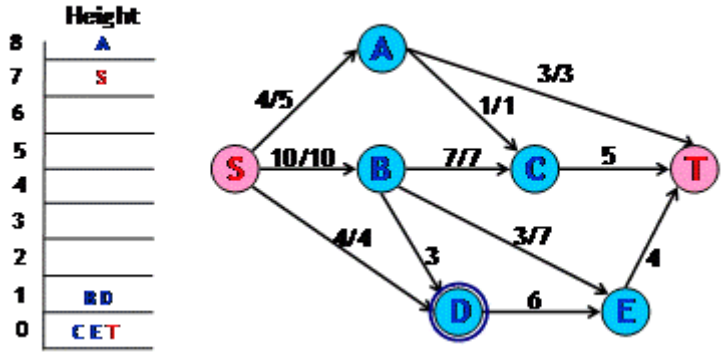

Figure 7 : Network after relabeling D

In **Figure 4**, pick up vertex **D** for applying the push operation. We find that it has no vertex that is downhill to it and the edge from **D** to that vertex has excess capacity. So we relabel **D** as shown in **Figure 5.**

```
void relabel(int u){
  int temp = -1;
  for(int i=0;i<G[u].size();++i) {
    v = G[u][i];
    if(cf[u][v] > 0) {
      if(temp == -1 || temp > h[v])
        temp = h[v];
    }
  }
  h[u] = 1 + temp;
}
```

Figure 8 : Code for Relabel sub-routine

Generic Algorithm

The generic algorithm is as follows:

```
void maxflow() {
    initialize_preflow();
    while(there is an operation that can be carried out)
      Select an operation and perform it.
}
```

The value **e(t)** will represent the maximum flow. We now try to see why this algorithm would work. This would need two observations that hold at all times during and after the execution of the algorithm.

1. **A residual edge from u to v implies h(u) ≤ h(v) + 1** -We had earlier introduced this as a property of the height function. Now we make use of induction for proving this property.
   - Initially residual edges are from vertices of height 0 to the source that is at height **n**.
   - Consider a vertex **u** getting relabeled and **v** is the vertex of minimum height to which **u** has a residual edge. After relabeling, the property holds for the residual edge**(u,v)**. For any other residual edge **(u,w)**, **h(v) ≤h(w)**. So after relabeling the property will hold for residual edge **(u,w)**. For a residual edge, **(w,u)**, since **u**'s height only goes up, therefore the property will continue to hold trivially.
   - Consider a push operation from **u** to **v**. After the operation, edge (v,u) will be present in the residual graph. Now **h(u) > h(v) ∧ h(u) ≤ h(v) + 1**

     → **h(u) = h(v) + 1**
     → **h(v) = h(u) − 1**
     → **h(v) _ h(u) + 1**

2. **There is no path for source to sink in the residual graph** - Let there be a simple path **{v_0 , v_1 , . . . , v_{k-1}, v_k }** from **v_0 = s** to **v_k = t**. Then, as per the above observation,

   h(v i) ≤ h(v i+1) + 1
   → h(s) ≤ h(t) + k
   → n ≤ k
   → n < k + 1

This violates the simple path condition as the path has **k+1** vertices. Hence there is no path from source to sink.

When the algorithm ends, no vertex is overflowing except the source and the sink. There is also no path from source to sink. This is the same situation in which an augmenting path algorithm ends. Hence, we have maximum flow at the end of the algorithm.

Analysis

The analysis requires proving certain properties relating to the algorithm.

$$\begin{aligned}\sum_{v \in S} e(v) &= \sum_{w \in V, v \in S} f(w,v) \\ &= \sum_{w \in V-S, v \in S} f(w,v) + \sum_{w \in S, v \in S} f(w,v) \\ &= \sum_{w \in V-S, v \in S} f(w,v) \\ &\leq 0\end{aligned}$$

1. **s is reachable from all the overflowing vertices in the residual graph** - Consider **u** as an overflowing and **S** as the set of all the vertices that are reachable from **u** in the residual graph. Suppose **s ∉ 2S** . Then for every vertex pair **(v,w)** such that **v ∈ S** and **w ∈ V-S** , **f(w,v) ≤ 0**. Because if **f(w,v) > 0**, then **cf(v,w) > 0** which implies **w** is reachable from **u**. Thus, since **e(v) ≤ 0**, for all **v ∈ S, e(v) = 0**. In particular, **e(u) = 0**, which is a contradiction.
2. **The height of a vertex never decreases** - The height of a vertex changes only during the **relabeling** operation. Suppose we relabel vertex **u**. Then for all vertices **v** such that **(u,v)** is a residual edge, we have **h(v) ≤ h(u)**, which clearly means **h(v)+1>h(u)**. So the height of a vertex never decreases.
3. **The height of a vertex can be at maximum 2n-1** - This holds for **s** and **t** since their heights never change. Consider a vertex **u** such that **e(u) > 0**. Then there exists a simple path from **u** to **s** in the residual graph. Let this path be **$u = v_0, v_1, \ldots, v_{k-1}, v_k = s$**. Then **k** can be at most **n-1**. Now, as per the definition of **h**, **$h(v_i) \leq h(v_{i+1}) + 1$**. This would yield **$h(u) \leq h(s) + n - 1 = 2n - 1$**.

Now we are in a position to count the number of operations that are carried out during the execution of the code.

- **Relabeling operations** - The height for each vertex other than **s** and **t** can change from **0** to **2n-1**. It can only increase, so there can be utmost **2n-1** relabelings for each vertex. In total there can be a maximum of **(2n-1)(n-2)** relabelings. Each relabeling requires at most **degree(vertex)** operations. Summing this up over all the vertices and over all the relabelings, the total time spent in relabeling operations is **O(nm)**.
- **Saturated Pushes** - A saturating push from **u** to **v**, results in **f(u,v) = c(u,v)**. In order to push flow from **u** to **v** again, flow must be pushed from **v** to **u** first. Since **h(u) = h(v) + 1**, so **v**'s height must increase by at least **2**. Similarly **h(u)** should increase by at least **2** for the next push. Combining this with the earlier result on the maximum height of avertex, the total number of saturating pushes is at most **2n-1** per edge. So that total overall the edges is at most **(2n-1)m < 2nm**.
- **Non-saturated Pushes** - Let **φ = ∑ u ∈ V, u is active h(u)**. Each non-saturating push from a vertex **u** to any other vertex **v** causes **φ** to decrease by at least **1** since **h(u) > h(v)**. Each saturating push can increase **φ** by at most **2n-1** since it could be that **v** was not active before. The total increase in **φ** due to saturating pushes is at most **(2n-1)(2nm)**. The total increase in **φ** due to relabeling operation is at most **(2n-1)(n-2)**. Therefore, the total number of non-saturating pushes is at most **$(2n-1)(2nm) + (2n-1)(n-2) \leq 4n^2m$**.

Thus the generic algorithm takes **$O(n^2m)$** operations in total. Since each push operation requires **O(1)** time, hence the running time of the algorithm will also be **$O(n^2m)$** if the condition given in the while loop can be tested in **O(1)** time. The next section provides an implementation for doing the same. In fact, by ordering the operations in a particular manner, a more rigorous analysis proves that a better time bound can be achieved.

First-in First-out Algorithm

We will make use of a first-in, first-out strategy for selecting vertices on which push/relabel operation will be performed. This is done by creating a queue of vertices that are overflowing. Initially all the overflowing vertices are put into the queue in any order. We will run the algorithm till the queue becomes empty.

In every iteration the vertex at the front of the queue is chosen for carrying out the operations. Let this vertex be **u**. Consider all the edges of **u**, both those that are incident on **u**and those that are incident on other vertices from **u**. These are edges along which **u** can potentially push more flow. Go through these edges one by one, pushing flow along edges that have excess capacity. This is done until either **u** becomes inactive or all the edges have been explored. If during the iteration any new vertex starts to overflow, then add that vertex to the end of the queue. If at the end of the iteration **u** is still overflowing, then it means **u** needs a relabeling. Relabel **u** and start the next iteration with **u** at the front of the queue. If any time during the process or at end of it **u** becomes inactive, remove **u** from the queue. This process of pushing excess flow from a vertex until it becomes inactive is called **discharging a vertex**.

```
void maxflow() {
  initialize_preflow();
  queue<int> q;
  char *l = new char[n];
  int u,v,m;
  memset(l,0,sizeof(char)*n);
  for(int i=0;i<G[s].size();++i) {
    if(G[s][i] != t) {
      q.push(G[s][i]);
      l[G[s][i]] = 1;
    }
  }
  while(q.size() != 0) {
    u = q.front();
    m = -1;
    for(int i=0;i<G[u].size() && e[u] > 0;++i) {
      v = G[u][i];
      if(cf[u][v] > 0) {
        if(h[u] > h[v]) {
          push(u,v);
          if(l[v] == 0 && v != s && v != t) {
            l[v] = 1;
            q.push(v);
          }
        }
        else if(m == -1) m = h[v];
        else m = min(m,h[v]);
      }
    }
    if(e[u] != 0) h[u] = 1 + m;
    else {
      l[u] = 0;
      q.pop();
    }
  }
}
```

Figure 9 : Code for First-In First-Out Algorithm

Analysis of First-In First-Out Algorithm

To analyze the strategy presented above, the concept of a **pass over a queue** needs to be defined. **Pass 1** consists of discharging the vertices added to the queue during initialization. **Pass i + 1** consists of discharging vertices added during the **Pass i**.

- **The number of passes over the queue is at most $4n^2$** - Let **φ = max {h(u) |u is active|}**. If no heights changes during a given pass, each vertex has its excess moved to vertices that are lower in height. Hence **φ** decreases during the pass. If **φ** does not change, then there is at least one vertex whose height increases by at least 1. If **φ**increases during a pass, then some vertex's height must have increased by as much as the increase **φ**. Using the proof about the maximum height of the vertex, the maximum number of passes in which **φ** increases or remains same is $2n^2$. The total number of passes in which **φ** decreases is also utmost $2n^2$.Thus the total number of passes is utmost $4n^2$.
- **The number of non-saturating pushes is at most $4n^3$** - There can be only one non-saturating push per vertex in a single pass. So the total number of non-saturating pushes is at most $4n^3$.

On combining all the assertions, the total time required for running the first-in first-out algorithm is $O(nm + n^3)$ which is $O(n^3)$.

Related Problems

In general any problem that has a solution using max-flow can be solved using **"push-relabel"** approach. Taxi and Quest4 are good problems for practice. More problems can be found in the article on max flow using augmenting paths. In problems where the time limits are very strict, push-relabel is more likely to succeed as compared to augmenting path approach. Moreover, the code size is not very significant. The code provided is of **61 lines** (16 + 36 + 9), and I believe it can be shortened.

In places where only the maximum flow value is required and the actual flow need not be reported, the algorithm can be changed slightly to work faster. The vertices that have heights **≥n** may not be added to the queue. This is because these vertices can never push more flow towards the sink. This may improve the running time by a factor of 2. Note that this will not change the asymptotic order of the algorithm. This change can also be applied in places where the min-cut is required. Let **(S,T)** be the min-cut. Then, **T**contains the vertices that reachable from **t** in the residual graph.