Survey

Jakob Mark Friis(20080816, jfriis@cs.au.dk)
Steffen Beier Olesen(20080991, 20080991@cs.au.dk)

October 17, 2013

Contents

1. Introduction **3**

2. Terminology **3**

3. Paradigms **4**
- 3.1. Augmenting Paths 4
- 3.2. Blocking Flow 4
- 3.3. Push Relabel 5

4. Survey **5**

5. Edmonds Karp 1972 **10**
- 5.1. The Algorithm 10
- 5.2. Analysis 10

6. Dinic 1970 **11**
- 6.1. The Algorithm 11
- 6.2. Analysis 11

7. Goldberg Tarjan 1988 **12**

8. King Rao 1992 **13**
- 8.1. The Game 15
- 8.2. Analysis of The Game 18
- 8.3. The algorithm 23
- 8.4. Analysis of the algorithm 26
- 8.5. Implementation modifications 28

9. Tests **30**
- 9.1. Algorithm Correctness 31
- 9.2. Graph generators 31

A. Graph Examples **35**

1 Introduction

The max flow problem is a graph problem. The problem is to determine how much flow can be sent through the graph from a node s to another node t, while not exceding capacity constraints on the edges. A number of other problems can be reduced to the max flow problem, such as maximum cardinality bipartite matching, maximum independant path and maximum edge-disjoint path.

Max flow algorithms have been around since 1956 [FF56], and since then many interesting algorithms have been published in the field. A recent publication [Orl13] proved that max flow problems can be solved in $O(nm)$ time for sparse graphs. Combined with [KR92], this means that we have an $O(nm)$ time algorithm for all max flow problems.

Many of the max flow algorithms focus on providing theoretical improvements, and have little to no focus on practical running time. In this paper, we will compare the practical running time on a selected subset of the max flow algorithms.

We will start by going over the terminology that we will use throughout the paper in section 2.
Some central ideas are repeated throghout several papers. We will give a general overview of these ideas in section 3.
Section 4 will contain a survery where we give a brief overview of the main improvements made since the first paper in the field [FF56].
The following sections will contain more detailed descriptions of the algorithms we selected to work with. We will explain how the algorithms work, how they achieve their bounds, and what, if any, modifications we have done to implement them.
The last part of the paper will contain the results of our comparisons.

2 Terminology

We use $G = (V, E)$ to signify the graph that we are running the max flow algorithms on. Here V is a set of nodes, and E is a set of edges, and $(u, v) \in E$ is an edge where $u \in V \wedge v \in V \wedge u \neq v$. We use n to signify the number of nodes in the graph, and m to signify the number of edges in the graph. If (u, v) exists in E, we assume that (v, u) also exists in E. With the max flow problem, two nodes, source and target are given. We denote them by s and t respectively.

A path in a graph is defined as an ordered set of nodes $\{v_0, ..., v_k\}$ where $\forall i \in [0, ..., k] : v_i \in V$, and the set contains no duplicates.

Every edge has a capacity associated with it denoted by $cap(u, v)$. This is an upper bound on the amount of flow we are allowed to send on the edge. We use U to represent the maximum capacity over all edges in the graph. The actual flow sent on an edge is denoted by $f(u,v)$. Residual capacity on an edge is the amount of flow that can still be sent on the edge without violating the capacity constraint. It is defined as $u(u,v) = cap(u,v) - f(u,v) + f(v,u)$. For edges $(u,v) \notin E$, we define $cap(u,v) = f(u,v) = u(u,v) = 0$.

The excess of a node v, $e(v)$ is how much flow currently resides in the node $e(v) = \sum\limits_{u \in V} f(u,v) - f(v,u)$. This may generally never be negative, except for the node s.

In order to have a valid flow, the following conditions must be met:

1. $\forall v \in V \setminus \{s,t\}, e(v) = 0$
2. -e(s) = e(t)
3. $\forall (u,v) \in E, f(u,v) \leq cap(u,v)$

The first two are refered to as the flow conservation constraint, and the third is the capacity constraint.

3 Paradigms

There are three general ideas that repeat throughout the max flow algorithms.

3.1 Augmenting Paths

The first idea was introduced by L. R. Ford and D. R. Fulkerson in 1956 [FF56], and consists of finding augmenting paths in the graph. An augmenting path $P = \{v_0, ..., v_k\}$ is a path in G where $v_0 = s, v_k = t, \forall i < k : u(v_i, v_{i+1}) > 0$. In other words, a path from s to t in the residual network, where it is possible to send more flow.

The basic idea is that if you find all augmenting paths in the graph, no more flow can be sent from s to t, and you must have a max flow. This idea is also what is often used to prove correctness of a max flow algorithm. If the flow is valid, and there is no augmenting paths in the residual graph, you must have found the max flow.

3.2 Blocking Flow

The Blocking Flow idea was introduced by E. A. Dinic in 1970 [Din70]. The idea is to construct a layer graph that only contains the edges that point forward towards t. So if we calculate the distance from s to each node, an edge (u,v) only exists in the layer graph if $distance(s,u) < distance(s,v) \wedge u(u,v) > 0$. The nodes in the layer graph are the same as the nodes in the graph G.

The interesting thing about the layer graph is that it contains all augmenting paths of a certain length k, where k is the length of the shortest augmenting path in the residual network of G. The algorithm can now find the max flow in this smaller layer graph. This flow is denoted the blocking flow. Most blocking flow algorithms then continue by calculating the next layer graph, which must have a bigger k, until all augmenting paths have been found.

3.3 Push Relabel

The Push Relabel idea was introduced by A. V. Goldberg and R. E. Tarjan in 1988 [GT88]. This idea differs substantially from the previous two ideas, in that it does not explicitly find augmenting paths. Instead it violates the flow convservation constraint throughout the algorithm, and pushes flow between individual nodes in the graph, not only s and t.

The idea is to assign a label $d(v)$ to each node. It starts with giving s the label n, and all other nodes the label 0. It then pushes as much flow as possible from s to the neighbors of s. The main part of the algorithm is a sequence of pushes and relabels. A relabel on a node increases its label by at least one. A push sends flow from one node u to another node v, but this is only allowed if $d(u) > d(v)$. Apart from in the initialization, the nodes s and t are never relabeled, and are never the source of a push.

What is going to happen when running a push relabel algorithm is that flow will be pushed around the graph towards t.

At some point, the nodes will start to be relabed above n. The longest path that does not include s is of length $n - 1$, so when a node is relabled to $n + 1$, t is no longer reachable from this node. A result of having labels above n is that flow will begin to be pushed back towards s. Eventually, all the flow will have been pushed to either s or t, which means that the flow conversation constraint is fulfilled.
At this point, a push relable algorithm will have found the max flow.

4 Survey

The purpose of this survey is to give an overview of the most important papers about solving the max flow problem. For the algorithms presented, we give a short introduction to the main ideas and techniques used, but for the details we direct the reader to the original articles.

Year	Authors	Running Time	Ref
1956	Ford, Fulkerson	$O(n^2mU)$	[FF56]
1970	Dinic	$O(n^2m)$	[Din70]
1972	Edmonds, Karp	$O(nm^2)$	[EK72]
1974	Karzanov	$O(n^3)$	[Kar74]
1977	Cherkasky	$O(n^2\sqrt{m})$	[Che77]
1978	Malhotra, Kumar, Maheshwari	$O(n^3)$	[MKM78]
1979	Gali, Naamad	$O(nm\log^2 n)$	[GN79]
1980	Gali	$O(n^{\frac{5}{3}}m^{\frac{2}{3}})$	[Gal80]
1983	Sleator, Tarjan	$O(nm\log n)$	[ST83]
1984	Tarjan	$O(n^3)$	[Tar84]
1985	Gabow	$O(nm\log U)$	[Gab85]
1988	Goldberg, Tarjan	$O(nm\log\frac{n^2}{m})$	[GT88]
1989	Auija, Orlin	$O(nm+n^2\log U)$	[AO89]
1989	Auija, Orlin, Tarjan	$O(nm\log\left(\frac{n}{m}\sqrt{\log U}+2\right))$	[AOT89]
1989	Cheriyan, Hagerup	$E\left(\min\begin{pmatrix} nm\log n \\ nm+n^2(\log n)^2 \end{pmatrix}\right)$	[CH89]
1990	Alon	$O(\min\{nm\log n, n^{\frac{8}{3}}\log n\})$	[Alo90]
1992	King, Rao	$O(nm+n^{2+\varepsilon})$	[KR92]
1994	King, Rao, Tarjan	$O(nm\log_{\frac{m}{n}\log n} n)$	[KRT94]
1998	Goldberg, Rao	$O(\min\{n^{\frac{2}{3}}, \sqrt{m}\}m\log\left(\frac{n^2}{m}\right)\log U)$	[GR98]
2012	Orlin	$O(nm+m^{31/16}\log^2 n)$	[Orl13]

The first algorithm for solving the max flow problem was introduced in 1956 by L. R. Ford and D. R. Fulkerson [FF56]. They proposed an algorithm that iteratively finds augmenting paths. Since they posed no restrictions on the order with which paths are found, their algorithm runs in $O(n^2mU)$ for integer costs, and is not guaranteed to terminate on real valued costs. In 1972, J. Edmonds and R. M. Karp [EK72] observed that if the augmenting path found in the Ford, Fulkerson algorithm always is a shortests augmenting path, the maximum number of augmenting paths are $O(nm)$. This algorithm runs in $O(nm^2)$ time. We explain this algorithm in more detail in Section 5.

About the same time, in 1970, E. A. Dinic [Din70] published another improvement over Ford, Fulkerson, an algorithm that runs in $O(n^2m)$ time. His idea was to remove some edges in the graph, to get a layer graph which contain all paths from s to t that have length k, where k is the length of the shortest augmenting path in G. He then finds all augmenting paths in this layer graph, which is called the blocking flow. After that, he calculates the residual network of the original graph, and finds a new layer graph where $k' > k$. To find all paths in the layer graph, he used a depth first search. Many subsequent algorithms are based on this idea of using layer graphs, but have an optimized alogithm for finding the blocking flow. More details on this algorithm can be found in Section 6.

The first optimization was published by Karzanov in 1974 [Kar74]. He came up with a very complicated algorithm for finding the blocking flow that uses preflows. This algorithm reduced the running time to $O(n^3)$, which is $O(nm)$ for very dense graphs where $m = O(n^2)$. There have been serveral publications that uses the same basic ideas as Karzanov, but tries to simplify the algorithm. One example is an algorithm by V. M. Malhotra, M. P. Kumar and S. N. Maheshwari published in 1978 [MKM78]. Another example was done by R. E. Tarjan in 1984 [Tar84].

B. V. Cherkasky published an algorithm in 1977 that runs in $O(n^2\sqrt{m})$. He groups some consecutive layers together, and runs a combination of Dinic and Karzanov. Z. Gali builds on top of this idea in an algorithm published in 1980 [Gal80]. He uses the idea of grouping the layers and improves it by contracting some paths in the graph into a single edge, and achieves a running time of $O(n^{\frac{5}{3}}m^{\frac{2}{3}})$.

In 1979 Z. Gali and A. Naamad made a paper [GN79] where they give an improved variation on the Dinic algorithm. They noticed that the Dinic algorithm has the problem that when it finds an augmenting path, it jumps back to the node just before the bounding arch, and forgets the rest of the path, which might be reused in a later path. Gali and Naamad built a data structure for saving the paths already visited, reducing the overall running time to $O(nm\log^2 n)$.

D. D. Sleator and R. E. Tarjan published an algorithm in 1983 [ST83] where they introduced the data structure for dynamic trees, also called link-cut trees. They use their data structure to make a max flow algorithm based on Dinic, that has a running time of $O(nm\log n)$. The advantage of using dynamic trees is that it allows you to push flow on a path in logarithmic time instead of linear time.

In 1985 H. N. Gabow gives a rather simple scaling algorithm for finding the maximum flow [Gab85]. His Idea is to check if the graph has any capacities greather than m/n, and if so, half all capacities and run the algorithm recursively. Since the capacities are integers, this only gives a near optimum solution. He uses Dinics algorithm on the residual network to find the correct solution. At the base of the recursion it is also running Dinics algorithm. This yields a running time of $O(nm\log U)$.

After this, the max flow algorithms started moving away from the layered idea from Dinic. A. V. Goldberg and R. E. Tarjan published an algorithm [GT88] that combined the preflow idea with dynamic trees, without using layer graphs. This algorithm is called the push relabel algorithm. They gave a simple version of it that runs in $O(n^3)$, and then they combined it with dynamic trees and got an algorithm that runs in $O(nm\log\frac{n^2}{m})$. Details on the algorithms presented in this paper can be found in Section ?. Most later algorithms are based on this algorithm in some way.

One of these algorithms was published in 1989 by R. K. Ahuja and J. B. Orlin [AO89]. It modified the simple $O(n^3)$ algorithm from Goldberg, Tarjan [GT88] with scaling ideas from Gabows paper 1985 [Gab85]. They used these ideas to decrease the number of non-saturating pushes, which was a bottleneck in the algorithm by Goldberg, Tarjan. The general idea was to find the lowest integer number, called the excess dominator, that is a power of 2 and is higher than the excess in all nodes. In each scaling iteration, a flow of at least half of the excess dominator should be pushed from nodes who can do so onto nodes which can recieve it, without invalidating the excess dominator. This idea lead to an algorithm running in $O(nm + n^2 \log U)$.

R. K. Ahuja, J. B. Orlin and R. E. Tarjan published an algorithm the same year [AOT89] which improved upon this algorithm. The first improvement was to make a better strategy for choosing the order for selecting which nodes to push flow from. The second improvment was to use a non constant scaling factor, so the excess dominator did not have to be a power of 2. They also added dynamic trees to the algorithm, and incorporated some ideas from the paper by Tarjan 1984. All this lead to a running time of $O(nm \log\left(\frac{n}{m}\sqrt{\log U} + 2\right))$.

In 1989 J. Cheriyan and T. Hagerup published a paper describing a new algorithm for solving the maximum-flow problem [CH89]. The algorithm was a randomized algorithm building on top of the algorithms described in Goldberg, Tarjan [GT88] and Ahuja, Orlin [AO89], it also included the dynamic trees. The algorithm changed Goldberg and Tarjans algorithm to use scaling, just as Ahuja, Orlin [AO89] did, though with a non constant scaling factor. To achieve a better timebound than [GT88] they randomly permutated the adjancency list of each vertex at the start, and for a single vertex when relabling it. They also tried to decrease the number of dynamic tree operations by only linking an edge when sufficiently large flow can be send over it. The algorithm has an expected running time of $O(nm + n^2(\log n)^3)$, and a worst case running time of $O(nm \log n)$. According to [CHM90], personal communication between the authors of [CH89] and Tarjan lead to a better analysis of the algorithm, which resulted in an expected running time of $O(\min\{nm \log n, nm + n^2(\log n)^2\})$. Later work by Alon [Alo90] derandomized the algorithm to a deterministic algorithm having a running time of $O(\min\{nm \log n, n^{8/3} \log n\})$

J. Cheriyan, T. Hagerup and K. Mehlhorn [CHM90] combined ideas from [GT88], [AO89] and [CH89] resulting in a new max flow algorithm. The idea in the algorithm is to work on a preflow in a sub-network and gradually add the edges as the algorithm progresses. By adding the edges in order of decreasing capacities they decrease the number of arithmethic operations. The bottleneck in the algorithm then becomes finding the current-edge, which is the first edge in each node eligible to apply a push operation to. To solve this problem faster than $O(nm)$ they represent the graph as an adjencency matrix and partions the matrix into sub-matrices. The resulting algorithm has a running time of $O(\frac{n^3}{\log n})$. During the process of designing the algorithm they make a randomized version and then derandomize it using the techique from [Alo90].

The paper by V. King and S. Rao [KR92] builds on top of [AO89]. It modifies a special subroutine that select which edges to push on, and achieves a running time of $O(nm + n^{2+\varepsilon})$. This means that we after this paper can solve the max flow problem in $O(nm)$ time for graphs where $m > n^{1+\varepsilon}$, which is everything but sparse graphs. More details on this algorithm can be found in Section 8.

V. King, S. Rao and R. E. Tarjan improved upon their algorithm in [KRT94], resulting in a new running time of $O(nm \log_{\frac{m}{n} \log n} n)$.

D. S. Hochbaum tried a new approach to the maximum flow problem in [Hoc98]. The idea was to look at a tree data structure designed by Lerchs and Grossman in 1965. The data structure solves the s-excess problem that is equivalent to the min-cut problem, which itself is the dual problem of the max-flow problem. The idea in the new algorithm is to manipulate pseudoflows, which like the preflow may have nodes with a higher incoming flow than outgoing, but also allows nodes to have a higher outgoing flow than incoming. Interestingly the algorithm does not try to maintain or even progress towards a feasable flow, but instead creates pockets of nodes. Excess pockets are pockets with more incoming that outgoing flow, and defict pockets are pockets with more outgoing than incoming flow. The pockets are manipulated so that no excess pockets can send additional flows to any deficit pockets. The complexity of the algorithm is $O(nm \log n)$

A. V. Goldberg and S. Rao published an algorithm in 1998 [GR98] in which they combines the layer graph ideas from [Din70] with the push-relabel algorithm from [GT88]. When constructing the layer graph, instead of simply having each edge have a unit distance they use a distance function. The distance function used is binary, with an edge length being 0 if it has high capacity and 1 otherwise. The algorithm contracts the 0 labeled distance edges and calculates the max-flow in the resulting graph using the algorithm described in [GT88]. This idea leads to an algorithm wih a running time of $O(\min\{n^{\frac{2}{3}}, \sqrt{m}\} m \log(\frac{n^2}{m}) \log U)$.

In the paper by J. B. Orlin [Orl13] a new notion of compacting a network is introduced. It marks edges with a relatively high residual capacity as abundant. It then has various methods for contracting nodes incident to abundant arcs. The algorithm finds the max-flow in the contracted graph, and transforms it into a flow in the original graph. The flow in the compacted graph is calculated using the algorithm described in [GR98]. The article present several bounds on the running times. The overall running time is $O(nm + m^{31/16} \log^2 n)$. Which in the case of m being $O(n^{16/15-\varepsilon})$ is

$O(nm)$. Combined with the result from [KR92], this means that max-flow can always be calculated in a running time of $O(nm)$. [Orl13] also developes an algorithm running in $O(\frac{n^2}{\log n})$ if $m = O(n)$. Further details can be found in Section ?.

5 Edmonds Karp 1972

The Edmonds Karp algorithm is one of the first and simplest max flow algorithms. It was published in 1970 by Yefim Dinic [Din70] and in 1972 by Jack Edmonds and Richard Karp [EK72]. It is a small variation on the Ford Fulkerson algorithm from 1956 [FF56], that brings the worst case running time from $O(n^2mU)$ to $O(nm^2)$.

5.1 The Algorithm

The algorithm works by finding the shortest augmenting path using a breadth first search from s to t.

When such a path $P = \{v_1, v_2, ..., v_k\}$ where $k \geq 2, v_1 = s, v_k = t$ is found, it calculates the bounding capacity $\min_{i=[1,...,k-1]} cap(v_i, v_{i+1})$, and sends that much flow over the path.

It keeps doing this in the residual network until no more augmenting paths exist.

Correctness follows from the fact that the algorithm terminates when no more augmenting paths from s to t are found in the residual network, and the fact that the algorithm always keeps a valid flow.

The algorithm never violates any capacity constraints, because when it sends flow, it sends flow accoring to the minimum capacity on the path. It also never produces any excess in nodes other than s and t, because all flow is pushed along paths from s to t.

5.2 Analysis

The algorithm performs a breadth first search for each augmenting path in the graph. A single breadth first search takes $O(m)$ time. Every time the algorithm finds an augmenting path, it does a push along it. There must be at least one edge (u, v) on this path that is saturated, namely the edge with the minimum capacity. For this edge to be in the path, the distance from s to u must be less than the distance from s to v. After the edge has been saturated, it can not be used again before flow has been pushed the opposite way, which requires that the distance from s to v becomes less than the distance from s to u. The distance from s to any node can not be greater than n, and the distances never decrease, so an edge can only be saturated n times. There are m edges, so this results in the running time of $O(nm^2)$.

6 Dinic 1970

The dinic algorithm was published in 1970 by Yefim Dinic[Din70]. It is the paper that introduced the level graph and blocking flow, which is basically a way of reducing the size of the graph before looking for augmenting paths. The running time of this algorithm is $O(n^2m)$ which should make it perform better on dense graphs than Edmonds Karp.

6.1 The Algorithm

The algorithm first does a breath first search to filter some of the edges. In the search it marks nodes acording to their distance from s. Only edges (u, v) where the distance from s to u is less than the distance from s to v are used in the next step. Additionally, once t has been reached, no edges (u, v) should be added where the distance from s to v is greater than the distance from s to t. Finally, the graph should be trimmed, so it does not contain edges to nodes that can not reach t. This results in a level graph that potentially has much fewer edges than the original graph. The special thing about this graph is that all paths will go from s to t, and will have the same length k. We then run a single depth first search on the graph to find all augmenting paths of length k. For every augmenting path, we send the flow on the path like in Edmonds Karp, jump back behind the first bounding edge on the path, and continue the depth first search from there. Once that is done, we compute the residual network of the original graph, and repeat the algorithm until we find no more augmenting paths.

Correctness follows from the same argument as for Edmonds Karp. We always have a valid flow, and at the end of the algorithm, no augmenting path can be found from s to t in the residual network.

6.2 Analysis

Every time we have found a blocking flow in a level graph, we have found all augmenting paths of length k. The next level graph must have augmenting paths longer than k. The longest path possible from s to t is n, so we can calculate the level graph and blocking flow in at most n iterations.

Every time we find an augmenting path, we saturate one of the edges in the graph, so we can at most find m paths of length k. The maximum size of k is n, so the running time of the depth first search is $O(nm)$. Computing the level graph was done with a breath first search that stops when it reaches t, followed by a depth first search to trim nodes that can not reach t. A breath first search takes $O(m)$ time, and a depth first search that does not need to process any nodes twice also takes $O(m)$ time. This yields the running time $n(m + nm) = O(n^2m)$.

Dynamic trees can be utilized to find the blocking flow in $O(mlog(n))$

time, reducing the running time to $O(nmlog(n))$, but dynamic trees was not introduced until 1984 by R. E. Tarjan [Tar84].

7 Goldberg Tarjan 1988

Algorithm 1 Goldberg Tarjan Push and Relabel procedures

```
Require: $v$ is active, $r(v,w) > 0$ and $d(v) = d(w) + 1$
 1: procedure PUSH( Edge $(v,w)$ )
 2:     Send $\delta = \min(e(v), r(v,w))$ units of flow by updating the edges,
        $(v,w)$ and $(w,v)$, and the excess, $e(v)$ and $e(w)$.
 3: end procedure

Require: $v$ is active, and $\forall w \in V, r(v,w) > 0 \implies d(v) \leq d(w)$
 4: procedure RELABEL($v$)
 5:     $d(v) \leftarrow= \min\{d(w) + 1 | (v,w) \in E_f\}$
 6:     (if the set is empty, $d(v) \leftarrow \infty$)
 7: end procedure
```

Algorithm 2 Goldberg Tarjan Initialization and Main Loop

```
 1: function MAXFLOW($V, E, s, t$)
 2:     $d(s) \leftarrow n$
 3:     for all $v \in (V/\{s\})$ do
 4:         $d(v) \leftarrow 0$
 5:         $e(v) \leftarrow 0$
 6:     end for
 7:     for all $(v,w) \in (V/\{s\}) \times (V/\{s\})$ do
 8:         Set the flow to 0 on $(v,w)$ and $(w,v)$
 9:     end for
10:     for all $(s,v) \in \{s\} \times (V/\{s\})$ do
11:         Send max capacity flow through $(s,v)$ update $(v,s)$ accordingly
12:         Update excess of $v$
13:     end for
14:     while Queue $Q \neq \emptyset$ do                          ▷ Main-loop
15:         DISCHARGE( )
16:     end while
17:     return $e(t)$
18: end function
```

Correctness of the algorithm follows directly from the pseudocode. It is trivial to see that the running time of the generic alg is $O(n^3)$, and the running time of the one using dynamic trees is $O(nm \log \frac{n^2}{m})$

Algorithm 3 The Generic Goldberg Tarjan PushRelabel and Discharge procedures

```
procedure DISCHARGE
    Node $v \leftarrow$ first element of Q, removed from the queue.
    repeat
        PUSHRELABEL(v)
        if $w$ becomes active then
            Add $w$ to the back of Q
        end if
    until $e(v) = 0$ or $d(v)$ increases
    if $v$ is still active then
        add it to the back of Q
    end if
end procedure

Require: $v$ is active
procedure PUSHRELABEL($v$)
    Edge $e \leftarrow$ current edge of $v$
    if PUSH(e) is applicable then
        PUSH($e$)
    else
        if $e$ is not the last edge of the edgelist of v then
            set the current edge of $v$ to be the next edge
        else
            Set the current edge of $v$ to be the first edge in the edgelist
            RELABEL($v$)
        end if
    end if
end procedure
```

8 King Rao 1992

V. King and S. Rao published an algorithm [KR92] which runs in time $O(nm + n^{2+\varepsilon})$. The main part of the algorithm is based on a Push Relabel algorithm by J. Cheriyan, T. Hagerup and K. Mehlhorn [CHM90]. The contributions done by [KR92] are primarily modifications to a subroutine that selects current edges. This subroutine is described as a game played between the algorithm and an adversary. Cheriyan et al. [CHM90] showed that their algorithm runs in $O(nm + n^{2/3}m^{1/2} + P(n^2, nm) + C(n^2, nm))$, where the function $P : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ represents the number of points scored by the adversary in the game, and $C : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ represents the cost of implementing the algorithm's game strategy.

Algorithm 4 The Goldberg Tarjan Tree-PushRelabel and Send procedures

```
Require: $v$ is an active tree root
 1: procedure Tree-PushRelabel($v$)
 2:     Edge $(v, w) \leftarrow$ current edge of v
 3:     if $d(v) = d(w) + 1$ and $r(v, w) > 0$ then
 4:         if GetSize($v$) + GetSize($w$) $\leq k$ then
 5:             Make $w$ the parent of $v$ in the tree by calling Link($v, w$)
 6:             SetCost($v, r(v, w)$)
 7:             Send($v$)
 8:         else
 9:             Push($(v, w)$)
10:             Send($w$)
11:         end if
12:     else
13:         if $e$ is not the last edge of the edgelist of v then
14:             set the current edge of v to be the next edge
15:         else
16:             Set the current edge of v to be the first edge in the edgelist
17:             Cut all children of $v$ in the tree, set their values to $\infty$
18:             Relabel(v)
19:         end if
20:     end if
21: end procedure

Require: $v$ is active
22: procedure Send($v$)
23:     while GetRoot($v$) $\neq v$ and $e(v) > 0$ do
24:         $\delta \leftarrow \min(e(v)$, FindMinValue($v$))
25:         send $\delta$ value of flow in the tree by calling AddCost($-\delta$)
26:         while FindMinValue(v) $= 0$ do
27:             $u \leftarrow$ FindMin($v$)
28:             Cut($u$)
29:             SetCost($u, -\infty$)
30:         end while
31:     end while
32: end procedure

33: function FindMinValue($v$)
34:     $minNode \leftarrow$ FindMin($v$)
35:     return GetCost(minNode)
36: end function
```

In section 8.1, we will describe the general game, without relating it to the algorithm. We will then argue for the bounds on P and C in section 8.2. Section 8.3 will contain the main algorithm, and its relation to the game, and in section 8.4 , we will show how the algorithm achieves the runtime of $O(nm + n^{2+\varepsilon})$. Finally, in section 8.5 we will describe the modifications we have done to make the algorithm more usable in practice.

8.1 The Game

The game is played between the player and the adversary on a bibartite graph $G_g = (U_g, V_g, E_g)$. We will use N to signify the number of nodes, and M to signify the number of edges, such that $N = |U_g| = |V_g|$, $M = |E_g|$. This is not the same graph as the graph G we run max flow on, but we will describe how to construct G_g from G in section 8.3. For every node $u \in U_g$, the player must at all times have chosen a single edge incident to u to be the designated edge, unless no edges are incident to u. Certain moves done by the player or the adversary on these designated edges might award points to the adversary.

The goal for the player is to minimize the ammount of points gained by the adversary. We use $P(N, M)$ to represent the points scored by the adversary, and $C(N, M)$ to represent the cost of implementing the player's strategy.
The moves the adversary can do are:

Edge kill
: The adversary can kill any edge (u, v), permanently removing it from the game. He scores no points for this move.

Node kill
: The adversary can kill any node $v \in V_g$, permanently removing it and all incident edges from the game. He scores a point for every edge removed that was a designated edge.

The player can respond with any sequence of the following moves:

Edge designation
: The player must designate an edge for each node $u \in U_g$ that does not currently have a designated edge, unless no edges are incident to u.

Edge redesignation
: The player can change the designated edge of a node $u \in U_g$ that already have a designated edge, but he awards a point to the adversary for this move.

The game starts with the player designating edges. Then it progresses by repeatedly having the adversary do a move, followed by zero or more moves by the player.

Algorithm 5 The game of [KR92]

```
1: procedure AdversaryNodeKill($v$)
2:     Perform an Adversary Edge Kill on all edges incident to $v$
3: end procedure
4: procedure AdversaryEdgeKill($u$, $v$)
5:     Remove $(u, v)$ from the game
6:     if $(u, v)$ was the designated edge of $u$ then
7:         if $u \in U'_g$ then
8:             UpdateRatioLevel($v$)
9:         end if
10:        DesignateEdge($u$)
11:    end if
12: end procedure
13: procedure DesignateEdge($u$)
14:    if $degree(u) \leq l$ then
15:        Designate any edge incident to $u$
16:    else
17:        Designate edge $(u, v)$ such that $erl(v)$ is minimal over all edges incident to $u$
18:        UpdateRatioLevel($v$)
19:    end if
20: end procedure
21: procedure UpdateRatioLevel($v$)
22:    if $erl(v) < rl(v) \vee rl(v) \leq erl(v) - 2$ then
23:        $erl(v) \leftarrow rl(v)$
24:        if $erl(v) = t$ then
25:            Reset()
26:        end if
27:    end if
28: end procedure
29: procedure Reset
30:    $k \leftarrow t$
31:    while $|U_{k-2}| \geq (r_{k-1}l)|U_k|/4$ do
32:        $k \leftarrow k - 2$
33:    end while
34:    Set $erl(v) \leftarrow rl(v)$ for all $v \in V_{k-2}$
35:    Undesignate the designated edge for all $u \in U_k$
36:    Set $erl(v) = rl(v) = 0$ for all $v \in V_k$
37:    Designate an edge for all $u \in U_k$
38: end procedure
```

The strategy we will use for the player takes three parameters; l, t and r_0. For nodes u with fewer than l edges, we will simply designate any edge. We thus define $U'_g = \{u \in U_g | degree(u) > l\}$ as the subset of U_g where we use the advanced strategy.

We define the ratio $r(v)$ of a node $v \in V_g$ as $r(v) = \frac{degree_{designated}(v)}{degree_{initial}(v)}$, where $degree_{designated}(v)$ is the number of designated edges to v from nodes in U', and $degree_{initial}(v)$ is the degree of v before any edges was removed. The idea for the player's strategy is that when designating an edge $u \in U_g$, to look at all $v \in V_g$ incident to u, and designate the edge to the node v with the lowest $r(v)$. This way, the adversary won't score that many points when he performs a node kill. It will be too expensive to maintain a sorted list of ratios for every u though, so we partition them into ratio levels $rl(v)$.

We use t to represent the highest ratio level allowed, and r_0 as a seed for when a node changes ratio level. We define

$$r_i = 2r_{i-1} \forall i \in \{1, ..., t\}$$

$$rl(v) = \begin{cases} 0 \text{ if } r(v) < r_0 \\ i \text{ if } r_i \leq r(v) < r_{i+1} \\ t \text{ if } r_t \leq r(v) \end{cases}$$

Instead of keeping track of the ratio level of all nodes in V_g, we keep track of the estimated ratio level $erl(v)$, which might not represent the exact ratio level. When the ratio level of a node increases, we update the estimated ratio level to reflect the change, but when it decreases, we don't update the estimated ratio level until the ratio level has decreased twice. The reason behind this is that we want to avoid doing a lot of work if a ratio level oscillates between two levels.

The strategy for the player is as follows. When the game starts, the player must designate an edge for each node in U_g. When designating an edge for a node u, we designate any edge if $degree(u) < l$, and otherwise an edge (u, v) such that $erl(v)$ is minimal over all incident edges. If this causes the ratio level of v to increase, we must update its estimated ratio level.

When the adversary kills a designated edge (u, v), either through a node kill or an edge kill, the player designates a new edge for u. If as a result of an edge designation, the estimated ratio level of a node v becomes equal to t, the player performs a reset operation. The reset operation performs a number of edge redesignations to reduce the estimated ratio levels of all nodes above a certain level. The invariant is that all nodes v have $erl(v) < t$ at the end of the player's turn, which means that no nodes can be killed while having $erl(v) \geq t$.

When doing a reset, we partition the nodes into sets based on their ratio level. We define V_i to be all nodes $v \in V_g$ with $rl(v) \geq i$, and U_i to be all nodes $u \in U'_g$ whoose designated edge goes to a node in V_i.

$$V_i = \{v \in V_g | rl(v) \geq i\}$$
$$U_i = \left\{u \in U'_g | designatedEdge_{target}(u) \in V_i\right\}$$

Let $k > 3$ be the last level that satisfies $|U_{k-2}| < (r_{k-1}l)|U_k|/4$. The reset operation first updates the erl of all v with $rl(v) = k-2$, so $erl(v)$ gets the current value of $rl(v)$. It then undesignates all designated edges from nodes in U_k, and updates rl and erl for all nodes in V_k to 0. Finally, it redesignates edges for nodes in U_k.

8.2 Analysis of The Game

In this section we will argue for the values of $P(N,M)$ and $C(N,M)$.

First, we will bound the value of $P(N,M)$. The adversary gains a point when he kills a designated edge while killing a node, and when the player redesignates an edge.

When the degree of a node u falls below l, we will award l points to the adversary for the remaining edges. This is the maximum amount of points he could possibly score for u in the remainder of the game, since we will not redesignate any edges for nodes $u \notin U'_g$. Every node in U_g only drops below l once, so the total numer of points that can be gained this way is lN.

When the adversary performs a node kill on v, he gains $degree_{designated}(v)$ points. But due to the reset procedure, a node can not be in ratio level t or above when it is removed. We then get the inequalities

$$\begin{aligned} r(v) &\leq r_{t-1} \\ \frac{degree_{designated}(v)}{degree_{initial}(v)} &\leq r_{t-1} \\ degree_{designated}(v) &\leq r_{t-1} degree_{initial}(v) \\ \sum_{v \in V_g} degree_{designated}(v) &\leq r_{t-1} \sum_{v \in V_g} degree_{initial}(v) \\ points &\leq r_{t-1} M \end{aligned}$$

The number of points the adversary can gain this way is thus at most $r_{t-1}M$, because a node only can be killed once.

The last source of points are the redesignations done by the player. All of these are done in the reset procedure of the strategy. We will show that when a reset occurs on level k, there have been many edge kills since last reset on level k. We can then assign the cost of the redesignations to these edge kills. When we say that a reset occurs on level k, we mean that it redesignateded edges for all $u \in U_k$. In the following sections we will argue about what happend previous to a reset on level k. Specifically, what has happened since the previous reset on level k, or since the start of the algorithm.

Lemma 8.1 *When a reset occurs on level k, there has been at least $r_{k-1}l|U_k|/2$ designated edges to nodes in V_{k-1} since the previous reset at or below level k, or the start of the algorithm if no such reset has occured.*

Proof Let $v \in V_k$, and let $U_k(v) = \{u \in U_k | designatedEdge_{target}(u) = v\}$. The size of $U_k(v)$ is $degree_{designated}(v)$, due to the definition of U_k. At the time that (u, v) was designated, $erl(v)$ must have been the smallest erl amongst all neighbors of u. Since $v \in V_k$, $erl(v)$ must have been at least $k-1$, and since it was the smallest erl, the erl of all neighbors of u must also have been at least $k-1$. At some point since the last reset at or below level k or since the start of the algorithm, $rl(v)$ must have been less than k. Further more, the erl of all neighbors to u must have been below k, because all nodes start out with $rl(v) = 0$, and the reset procedure resets the ratio levels of nodes with $rl(v) \geq k$ to 0.

For $rl(v)$ to increase from 0 to k, at least $|U_k(v)|/2$ edge designitions must have been done to v. Since all nodes in $u \in U'_g$ have $degree(u) > l$, there must have been $l|U_k(v)|/2$ edges incident to nodes in V_{k-1} since the previous reset at level k or lower. These edges are the edges incident to u that are not designated. If we sum this up over all $v \in V_k$, we get $l|U_k|/2$ edges incident to nodes in V_{k-1}, because all $U_k(v)$ are disjoint.

We can then calculate how many designated edges there must be for a node $w \in V_{k-1}$, and for all nodes in V_{k-1}.

$$\begin{aligned}
r_{k-1} &\leq r(w) \\
r_{k-1} &\leq \frac{degree_{designated}(w)}{degree_{initial}(w)} \\
degree_{designated}(w) &\geq r_{k-1} degree_{initial}(w) \\
\sum_{w \in V_{k-1}} degree_{designated}(w) &\geq r_{k-1} \sum_{w \in V_{k-1}} degree_{initial}(w) \\
\sum_{w \in V_{k-1}} degree_{designated}(w) &\geq r_{k-1} l |U_k|/2
\end{aligned}$$

For each individual w, this only holds at the time the edge (u, v) was designated, so the sum says that there has been $r_{k-1}l|U_k|/2$ designated edges to nodes in V_{k-1} since the previous reset at or below level k, or the start of the algorithm.

Lemma 8.2 *At any point in the algorithm, at every level $k \geq 3$, at least one of the following two statements hold:*

1. $|U_{k-2}| \geq r_{k-1}l|U_k|/4$
2. *There was at least $r_{k-1}l|U_k|/8$ edge kills at level $k-2$ or higher, since the previous time a reset occured at or below level k.*

Proof To prove this lemma, we will assume that condition 1 does not hold, and show that condition 2 must hold. Lemma 8.1 gives us that there has been $r_{k-1}l|U_k|/2$ edge designitions to nodes in V_{k-1} since last reset at level k or below, but if $|U_{k-2}| < r_{k-1}l|U_k|/4$, at least $r_{k-1}l|U_k|/4$ designated edges were removed by the adversary from nodes that are or had been in V_{k-1}. A node that drops from level $k-1$ to below $k-2$ must lose at least half its designated edges at level $k-2$, which implies that at least $r_{k-1}l|U_k|/8$ designated edges was removed when they were incident to nodes v with $rl(v) \geq k-2$. That means that if condition 1 does not hold, then condition two must hold.

When a reset occurs at level k, we ensure that condition 1 from lemma 8.2 does not hold. The reset performs $|U_k|$ redesignations, and there have been at least $r_{k-1}l|U_k|/8$ edge kills at level $k-2$ or above since the previous reset at level k or below. We will let $\#edgeKills_k$ represent the number of edge kills since last reset on level k, and $\#redesignations_k$ to represent the number of redesignations during the specific reset operation. This gives us the equation

$$\begin{aligned}
\#edgeKills_k &\geq \frac{r_{k-1}l|U_k|}{8} \\
\#edgeKills_k &> \frac{r_0 l \#redesignations_k}{8} \\
\#redesignations_k &< \frac{8\#edgeKills_k}{r_0 l} \\
\sum_{\text{All Resets}} \#redesignations_k &< \sum_{\text{All Resets}} \frac{8\#edgeKills_k}{r_0 l} \\
\#redesignations &< \frac{8\#edgeKills}{r_0 l}
\end{aligned}$$

So, the total points scored by the adversary is

$$P(N, M) \leq N \cdot l + r_{t-1}M + \frac{8\#edgeKills}{r_0 l}$$

To make this more interesting, we can assign some values to the parameters r_0, l and t.

$$\begin{aligned}
r_0 &= \frac{N^{\varepsilon}}{\sqrt{M/N}} \\
l &= N^{\varepsilon}\sqrt{M/N} \\
t &= O(1/\varepsilon)
\end{aligned}$$

When we insert this into our bound on $P(N, M)$ we get

$$\begin{aligned}
P(N,M) &\le N \cdot N^{\varepsilon}\sqrt{M/N} + 2^{\frac{1}{\varepsilon}-1}\frac{N^{\varepsilon}}{\sqrt{M/N}}M + \frac{8\#edgeKills}{N^{2\varepsilon}} \\
&= N^{0,5+\varepsilon}M^{0,5} + 2^{\frac{1}{\varepsilon}-1}N^{0,5+\varepsilon}M^{0,5} + \frac{8\#edgeKills}{N^{2\varepsilon}} \\
&= O\left(N^{0,5+\varepsilon}M^{0,5} + \frac{\#edgeKills}{N^{\varepsilon}}\right)
\end{aligned}$$

Next, we will bound the value of $C(N, M)$, which was the cost of implementing the player's strategy. The algorithm will have to do the following things:

It will need to be able to find the neighbor with minimum erl when designating an edge. To do this easily, we keep an array of size t of linked lists for each node $u \in U'_g$. An edge will be placed in the $i'th$ linked list, if the corrosponding node v has $erl(v) = i$. This means that we can designate an edge in $O(t)$ time, by enumerating the linked lists from 0 to t, and pick any edge from the first non-empty linked list. For nodes $u \in U_g \setminus U'_g$, we just keep a single linked list of edges. There are $P(N, M) + N$ designations in total, so the designations requre $O(tP(N, M) + tN)$ time.

An edge can be removed from this datastructure in constant time by keeping a pointer to the linked list element in the edge object. The edges are never added back, so this takes $O(M)$ time total.

The datastructure will have to be updated when the erl for a node v changes. This means enumerating over all the edges incident to v, and moving each edge it into another linked list. The erl of a node is updated when rl increases by one or decreases by two, and during the reset operation. If we only consider the first two cases, at least $r_0\ degree_{initial}(v)$ edge kills or designations must have occured before the erl of a node changes. The cost of updating the datastructure is $degree(v)$, so the cost for all updates to v is at most

$$\begin{aligned}
cost(v) &\le degree(v)\frac{\#edgeKills(v) + \#edgeDesignations(v)}{r_0 degree_{initial}(v)} \\
&\le 1\frac{\#edgeKills(v) + \#edgeDesignations(v)}{r_0} \\
\sum_{v\in V_g} cost(v) &\le \sum_{v\in V_g} \frac{\#edgeKills(v) + \#edgeDesignations(v)}{r_0} \\
&\le \frac{\#edgeKills + P(N,M) + N}{r_0}
\end{aligned}$$

Finally, we have the updates to *erl* during the reset operation. The *erl* is only updated for nodes in or above level $k-2$. For nodes in V_{k-2}, we have

$$\begin{aligned}
r_{k-2} &\le r(v) \\
r_{k-2} &\le \frac{degree_{designated}(v)}{degree_{initial}(v)} \\
degree_{initial}(v) &\le \frac{degree_{designated}(v)}{r_{k-2}} \\
\sum_{v \in V_{k-2}} degree_{initial}(v) &\le \sum_{v \in V_{k-2}} \frac{degree_{designated}(v)}{r_{k-2}} \\
\sum_{v \in V_{k-2}} degree_{initial}(v) &\le \frac{|U_{k-2}|}{r_{k-2}}
\end{aligned}$$

So there are at most $|U_{k-2}|/r_{k-2}$ edges incident to nodes in V_{k-2}. As part of the reset, we ensure that $|U_{k-2}| < r_{k-1}l|U_k|/4$, so we can bound the number of edges further by

$$\sum_{v \in V_{k-2}} degree_{initial}(v) \le \frac{|U_{k-2}|}{r_{k-2}} < \frac{r_{k-1}l|U_k|}{4r_{k-2}} = \frac{2r_{k-2}l|U_k|}{4r_{k-2}} = \frac{l|U_k|}{2}$$

Each reset occurs after at least $r_{k-1}l|U_k|/8$ edge kills, so the cost of updating all the edges are

$$\begin{aligned}
cost &< \frac{l|U_k|}{2} \\
&< \frac{l}{2}\frac{8\#edgeKills}{r_{k-1}l} \\
&< \frac{4edgeKills}{r_0} \\
&= O\left(\frac{\#edgeKills}{r_0}\right)
\end{aligned}$$

This brings the total cost for $C(N, M)$ to

$$C(N, M) = O\left(tP(N, M) + tN + M + \frac{\#edgeKills + P(N, M) + N}{r_0}\right)$$

We can show that $t < \frac{1}{r_0}$ by

$$\begin{aligned}
r_t &\le 1 \\
2^t r_0 &\le 1 \\
2^t &\le \frac{1}{r_0}
\end{aligned}$$

By using the mathematical fact that $t \geq 0 \Rightarrow t < 2^t$, we get $t < \frac{1}{r_0}$.

The final total cost for maintaining the game then becomes

$$C(N, M) = O\left(M + \frac{\#edgeKills + P(N, M) + N}{r_0}\right)$$

8.3 The algorithm

The algorithm is a version of the push-relabel algorithm, with an additional operation; addEdge. It starts out with no edges in the graph, and then adds them one by one as the algorithm progresses. We define $E^* \subseteq E$ to be the edges that are added to the graph at any point in the algorithm. The hidden capacity of a node v is defined as $h(v) = \sum_{(v,u) \in E \setminus E^*} cap(v, u)$, the sum of capacities on edges going out of v that have not yet been added. We can then define the *visible excess* to be $e^*(v) = max\,(0, e(v) - h(v))$. We will use this instead of $e(v)$, to determine when to push or relabel a node. A push or relabel is only performed if the visible excess of the node is greater than zero, and it is never allowed to push more than the visible excess away from a node.

The initialization is the same as in the algorithm by Goldberg and Tarjan [GT88], in that we start with $d(s) = n$ and $\forall v \in V \setminus \{s\} : d(v) = 0$. We then saturate all edges (s, v) to get some excess into the graph. Like in the algorithm by Goldberg and Tarjan [GT88], a dynamic tree is used to keep track of paths of current edges.

We define the *undirected capacity* of an edge (u, v) to be $ucap(u, v) = cap(u, v) + cap(v, u)$. The main part of the algorithm adds the edges in order of decreasing $ucap(u, v)$. When (u, v) is added, (v, u) is added as well.

When an edge (u, v) is added, the algorithm checks if $d(u) > d(v)$, and if so, saturates the edge. The reason it can do this is that $d(u) > d(v) \Rightarrow d(u) > 0$, so u was relabeled at some point. When u was relabeled, $e^*(u) > 0 \Rightarrow h(u) < e(u)$. After that, $h(u)$ can never become greater than $e(u)$, since $h(u)$ only decreases, and $e(u)$ only decreases to the point where $e^*(u) = 0$. When an edge is added, $\forall v \in V : e^*(v) = 0$, so when (u, v) is added, and $h(u) \leftarrow h(u) - cap(u, v)$, then $e^*(u) \leftarrow cap(u, v)$, which means we now have enough visible excess to saturate the edge.

When a node gets $e^*(v) > 0$, a treepush is performed on it if it has a current edge, and otherwise it is relabeled. When doing a tree push on v, the algorithm uses the dynamic tree to find the first edge with capacity less than $e^*(v)$, It saturates this edge, and cuts from the dynamic tree. It then pushes $e^*(v)$ along the part of the path leading up to the bounding edge, by doing an add value operation on the dynamic tree.

To choose which edges to use when pushing, an instance of the game is used where $N = O(n^2)$ and $M = O(nm)$. More precisely, U_g and V_g contain a node for every node in V, and every possibel label $d \in \{0, ..., 2n\}$. For every

Algorithm 6 [KR92]

```
function MaxFlow($V, E, s, t$)
    Initialize()
    $edges \leftarrow \{(u,v) \in E | u \neq s \wedge v \neq s \wedge u < v\}$
    for all $(u,v) \in edges$ ordered by $ucap(u,v)$ decreasing do
        Add $(u,v)$ and $(v,u)$ to $F$
        if d(u) $>$ d(v) then
            Saturate(u, v)
        else if d(u) $<$ d(v) then
            Saturate(v, u)
        end if
        while $\exists v \in V \setminus \{s,t\} : e^*(v) > 0$ do
            if $CurrentEdge(v) \neq nil$ then
                TreePush(v)
            else
                Relabel(v)
            end if
        end while
    end for
    return $e(t)$
end function
procedure Initialize
    Create dynamic forest $F$
    $d(s) \leftarrow n$
    for all $(s,v) \in E$ do
        Add $(s,v)$ and $(v,s)$ to $F$
        Saturate(s, v)
    end for
end procedure
procedure TreePush($u$)
    $(u,v) \leftarrow CurrentEdge(u)$
    $link(u,v)$ if not linked
    if $\exists$ edge $(x,y)$ on path to root from $u$ in $F : u(x,y) \leq e^*(u)$ then
        Saturate(x, y)
        $cut(x,y)$
    end if
    send $e^*(u)$ units of flow along path from $u$ to its root in $F$
end procedure
procedure Relabel($v$)
    for all $u \in V : CurrentEdge(u) = (u,v)$ do
        $cut(u,v)$
    end for
    $d(v) \leftarrow d(v) + 1$
end procedure
```

$(u, v) \in E$ and every $d \in \{1, ..., 2n\}$, there is an edge connecting $(u, d) \in U_g$ to $(v, d-1) \in V_g$ in the game. The current edge of a node $v \in V$ is the designated edge of the node $(v, d(v)) \in U_g$. When an edge (u, v) is saturated in the max flow algorithm, the corrosponding edge $((u, d(u)), (v, d(u)-1))$ is killed by the adversary in the game. When a node u is relabeled to $d(u)+1$, it is treated as an adversary node kill on $(u, d(u))$.

Note that the add edge operation does not affect how the game chooses the current edges. It only affects the amount of visible excess in each node.

The dynamic tree is updated to match the current edges obtained from the game. That means that we will be updating it when a current edge is saturated, when a node is relabeled, and when a current edge is redesignated in the game.

We will now argue for correctness.

Lemma 8.3 *When a node is relabled, it has no egible edges.*

Proof A node is v relabled from $d(v)$ to $d(v)+1$ when it has visible excess and its current edge is *null*. If the current edge is *null*, that means that all edges incident to the corrosponding node $(v, d(v)) \in U_g$ in the game has been killed, either as a result of a saturating push, or because the target node was relabled do $d(v)$. Both cases result in the corrosponding edge being inegible.

Lemma 8.4 *If at the end of the algorithm, an augmenting path $s, v_1, ..., v_k, t$ exist in the residual network, then $d(v_i) \leq d(v_{i+1}) + 1$.*

Proof If $d(v_i) \leq 1$, this is trivially true, since $\forall v \in V : d(v) \geq 0$. Otherwise, consider the time that v_i was relabeled from $d(v_i)-1$ to $d(v_i)$. According to Lemma 8.3, for a node to be relabeled, it can not have any egible outgoing edges, so either $d(v_i) - 1 \leq d(v_{i+1})$ or $u(v_i, v_{i+1}) = 0$. We know that at the end of the algorithm, $u(v_i, v_{i+1}) > 0$, since we have a residual path, so if $u(v_i, v_{i+1}) = 0$ when v_i was relabeled to $d(v_i)$, flow must have been pushed from v_{i+1} to v_i at some later point, and that means that $d(v_i) < d(v_{i+1})$.

Theorem 8.5 *No argmenting path $s, v_1, ..., v_k, t$ can exist at the end of the algorithm.*

Proof Since we saturate (s, v_1) during initialization, flow must have been pushed back to make (s, v_1) residual, so $d(v_1) > d(s) = n$. Further more, since the maximum lenght of a path is n, $k \leq n-2$. From Lemma 8.4 we can get that $d(v_1) \leq d(v_2) + 1 \leq d(v_3) + 2 \leq ... \leq d(v_k) + k - 1$. So we have $n < d(v_1) \leq d(v_k) + k - 1 \leq d(v_k) + n - 3 \Rightarrow d(v_k) > 3$. At the time v_k was relabeled from 1 to 2, it must have held that $u(v_k, t) = 0$, since $d(t) = 0$ throughout the algorithm. However, no flow is ever pushed away from t, so if (v_k, t) was not residual when v_k was relabeled to 2, it can not be residual at the end of the algorithm, and we could not have had an augmenting path.

This proof does not take the add edge operation into account. The reason for this is that the add edge operation does not change the set of egible edges for a node. It only delays push and relabel operations the nodes untill they have positive visible excess, instead of just positive excess.

8.4 Analysis of the algorithm

The algorithm uses $C(n^2, nm)$ time to manage the game. The sorting of the edges acording to *ucap* can be done in $m \log m$ time, but since $m \leq n^2$, $\log m \leq \log n^2 = 2 \log n$, we get $m \log m = O(m \log n)$.

The relabeling is constant time, if we omit the time it takes to update the game and the dynamic tree. There are n nodes, and each node can at most be relabeled $2n$ times, wich means that the total time for relabel is $O(n^2)$. We can ignore the time it takes to update the game, because this is included in $C(n^2, nm)$, and we will analyze dynamic tree operations seperately.

The treepush operation does a find bounding edge operation, and an add value operation on the dynamic tree. This takes $O(\log n)$ time per tree push. Each link and cut in the dynamic tree takes $\log n$ time.

This leads us to the running time of

$$O\left(C\left(n^2, nm\right) + m \log n + n^2 + (\#\text{treepushes} + \#\text{links} + \#\text{cuts}) \log n\right)$$

Each tree push results in either a cut, or it reduces the visible exces in a non root node to zero. A non root node only gets positive visible excess as a result of a saturating push to it, or as a result of an edge being added. This means that $\#\text{treepushes} \leq \#\text{cuts} + \#\text{saturating pushes} + m$.

We perform a link in the tree when the current edge changes. This is either at the start of the algorithm, or direcly after a cut, so $\#\text{links} \leq n + \#\text{cuts}$.

We only cut things from the dynamic tree when we saturate an edge, or when a point is scored by the adversary, so $\#\text{cuts} \leq P(n^2, nm) + \#\text{saturating pushes}$

This means that we can update the running time to

$$O\left(C\left(n^2, nm\right) + m \log n + n^2 + \left(P\left(n^2, nm\right) + \#\text{saturating pushes}\right) \log n\right)$$

To bound the number of saturating pushes, we split them up into two catagories. An edge is saturated by a regular push bundle if at some point after having zero residual capacity in one direction, all subsequent pushes are done in the other direction until the edge is saturated in that direction.

Lemma 8.6 *The number of non regular push bundles is bounded by $P(n^2, nm)$.*

Proof In order for the direction to change, the target node must be relabeled at least twice to reach a label higher than the source node. If the edge is not yet saturated, the adversary will recieve a point when doing the relabeling, unless the player redesignated the edge before the relabling. Such a redesignation would also award a point to the adversary.

Lemma 8.7 *The number of regular push bundles is bounded by* $O(n^{1,5}m^{0,5}\log n)$.

Proof The proof for this can be found in [CHM90], Lemma 8.2 combined with Lemma 8.4.

This brings us to the bound

$$\#\text{saturating pushes} \leq P(n^2, nm) + n^{1,5}m^{0,5}\log n$$

We know from section 8.2 that $P(N,M) = O\left(N^{0,5+\varepsilon}M^{0,5} + \frac{\#edgeKills}{N^\varepsilon}\right)$. Since #edgeKills = #saturating pushes, we get

$$P(n^2, nm) \leq n^{1,5+\varepsilon}m^{0,5} + \frac{\#\text{saturating pushes}}{n^\varepsilon}$$

If we insert this with the bound on saturating pushes, we get

$$\#\text{saturating pushes} \leq n^{1,5+\varepsilon}m^{0,5} + \frac{\#\text{saturating pushes}}{n^\varepsilon} + n^{1,5}m^{0,5}\log n$$
$$\#\text{saturating pushes}(1 - \frac{1}{n^\varepsilon}) \leq n^{1,5+\varepsilon}m^{0,5} + n^{1,5}m^{0,5}\log n$$

$\frac{1}{n^\varepsilon} \to 0$ for sufficiently large n, and $logn = O(n^\varepsilon)$ for any positive ε, so

$$\#\text{saturating pushes} = O\left(n^{1,5+\varepsilon}m^{0,5}\right)$$

We can now solve for $P(n^2, nm)$, and get

$$P(n^2, nm) = O\left(n^{1,5+\varepsilon}m^{0,5} + \frac{\#\text{saturating pushes}}{n^\varepsilon}\right)$$
$$P(n^2, nm) = O\left(n^{1,5+\varepsilon}m^{0,5} + \frac{n^{1,5+\varepsilon}m^{0,5}}{n^\varepsilon}\right)$$
$$P(n^2, nm) = O\left(n^{1,5+\varepsilon}m^{0,5}\right)$$

If we insert this into the running time of the algorithm, we get

$$O\left(C\left(n^2, nm\right) + m\log n + n^2 + n^{1,5+\varepsilon}m^{0,5}\right)$$

If we evaluate $C(n^2, nm)$, based on the bound on $C(N,M)$ we obtained in the previous section, we get

$$
\begin{aligned}
C(N,M) &= O\left(M + \frac{\#edgeKills + P(N,M) + N}{r_0}\right)\\
C(n^2,nm) &= O\left(nm + \frac{\#\text{saturating pushes} + P(n^2,nm)}{\frac{n^\varepsilon}{\sqrt{m/n}}} + \frac{n^2}{\frac{n^\varepsilon}{\sqrt{m/n}}}\right)\\
C(n^2,nm) &= O\left(nm + \frac{n^{1{,}5+\varepsilon}m^{0{,}5}}{\frac{n^{0{,}5+\varepsilon}}{m^{0{,}5}}} + \frac{n^2}{\frac{n^{0{,}5+\varepsilon}}{m^{0{,}5}}}\right)\\
C(n^2,nm) &= O\left(nm + nm + n^{1{,}5-\varepsilon}m^{0{,}5}\right)\\
C(n^2,nm) &= O\left(nm + n^{1{,}5-\varepsilon}m^{0{,}5}\right)
\end{aligned}
$$

This leads us to the running time of $O(nm + n^{1{,}5+\varepsilon}m^{0{,}5})$ for the algorithm.
If $m = n^2$, then nm dominates $n^{1{,}5+\varepsilon}m^{0{,}5}$.
If $m = n$, then $n^{1{,}5+\varepsilon}m^{0{,}5} = n^{2+\varepsilon}$ dominates nm.
The cross point is when $nm = n^{1{,}5+\varepsilon}m^{0{,}5} \Rightarrow m = n^{1+\varepsilon}$.
So, the algorithm runs in time $O(nm + n^{2+\varepsilon})$, and that is $O(nm)$ when $m \geq n^{1+\varepsilon}$.

Unfortunatly, the game has M edges, and N nodes, and each of those nodes have t linked lists. This means that the algorithm uses $\Omega\left(M + Nt\right) = \Omega\left(nm + \frac{n^2}{\varepsilon}\right)$ space, which makes it difficult to run it on medium to large graphs where m is close to n^2.

8.5 Implementation modifications

The [KR92] algorithm has some major problems that makes it unusable in practice. The biggest problem is that the game takes up too much space. We decided to make an algorithm that uses the same basic strategy for calculating the max flow, but with $O(nt + m)$ space.

The first thing we did was to only use one layer in the game, and keep track of which edges are active for each node by adding and removing them from the game. This change, means that when we relabel a node, and need to do the corrosponding node kill, we not only have to remove edges that are now inegible due to lables, but we also have to add edges that have become egible from the node that we relabel. To do this efficiently, we keep a linked list of edges that are inegible due to level in each node $u \in U_g$. When v is relabeled from $d(v)$ to $d(v)+1$, we first run through all incident active edges (u,v), and move them into the inegible linked list of u if $d(u) \leq d(v)+1$. This is the same amount of work as we had to do in the previous version of the algorithm.

Next we run through the inegible linked list of the node $u \in U_g$ that corrosponds to v. If any edge now go to a node v' with $d(v') < d(v)+1$, we add it to the active lists.

The total run time of the relabel becomes $O(degree_{initial}(v))$, which summed up over n nodes and a maximum of $2n$ relabels pr node becomes $O(nm)$ time for relabels in total, without counting the time for designating edges.

Recall that the numer of points that could be scored by the adversary was

$$P(n^2, nm) \leq n^2 l + r_{t-1} nm + \frac{8\#edgeKills}{r_0 l}$$

The first term was because the adversary was awarded l points every time the degree of a node goes below l, and this could only happen once per node. With this change, it can happen multiple times per node, however a node only recieves more edges when it is relabled, and a node can only be relabled $2n$ times. This means that although the degree a node can drop below l $O(n)$ times, since the number of nodes in the game is now n instead of n^2, we still get a cost of $n^2 l$ here.

The second term is points gained from edge kills. With the change to the game, it is now possible to kill an edge multiple times. In order to kill an edge more than once, it has to have been added back into the game through a relable, so it can only be killed $O(n)$ times. This means that the cost of this also remains at $r_{t-1}nm$, because there now only are m edges.

The last term represents the points gained from redesignations. There is no change in the analysis here. We can still attribute the cost to the edge kills, which corrosponds to saturating pushes, and the number of saturating pushes remain unchanged.

$$P(n, m) \leq n^2 l + r_{t-1} nm + \frac{8\#edgeKills}{r_0 l}$$

To make the numers fit, we set

$$\begin{aligned} r_0 &= \frac{n^\varepsilon}{\sqrt{m/n}} \\ l &= n^\varepsilon \sqrt{m/n} \\ t &= O(1/\varepsilon) \end{aligned}$$

And we get

$$\begin{aligned} P(n, m) &\leq n^2 \cdot n^\varepsilon \sqrt{m/n} + 2^{\frac{1}{\varepsilon}-1} \frac{n^\varepsilon}{\sqrt{m/n}} nm + \frac{8\#edgeKills}{n^\varepsilon} \\ &= n^{1{,}5+\varepsilon} m^{0{,}5} + 2^{\frac{1}{\varepsilon}-1} n^{1{,}5+\varepsilon} m^{0{,}5} + \frac{8\#edgeKills}{n^{2\varepsilon}} \\ &= O\left(n^{1{,}5+\varepsilon} m^{0{,}5} + \frac{\#edgeKills}{n^\varepsilon} \right) \end{aligned}$$

The other thing that changes is C. According to the old analysis, this was $O(tP(N,M)+tN)$ for all edge designations, $O(M)$ for keeping track of removed edges, and $O\left(\frac{\#edgeKills+P(N,M)+N}{r_0}\right)$ for moving edges between linked lists when the *erl* of a node changes.

It still takes $O(t)$ time to designate an edge, and we still have to designate an edge whenever the adversary gains a point, and at the start, which yields $O(tP(n,m)+tn)$. One extra place where we need to do edge designations are after a node has been relabled. This can happen $O(n)$ times per node, so that yields $O(tn^2)$.

When an edge is killed it takes constant time to remove it from the linked list in the node it came from. Adding it back in is only done in the relable, and we already bounded the total time for relabeling. So, since each edge can be killed $O(n)$ times, we get a cost of $O(nm)$ for maintaing egible edges.

Finally, we have the cost of moving edges when the *erl* of a node changes. This analysis does not really change, except for that the number of edge designations are $P(n,m)+n^2$ instead of $P(N,M)+N$, which yields $O\left(\frac{\#edgeKills+P(n,m)+n^2}{r_0}\right)$.

$$
\begin{aligned}
C(n,m) &= O\left(tP(n,m)+tn+tn^2+nm+\frac{\#edgeKills+P(n,m)+n^2}{r_0}\right)\\
&= O\left(nm+\frac{\#edgeKills+P(n,m)+n^2}{r_0}\right)
\end{aligned}
$$

The new bounds on P and M are the same as inserting $N=n^2$ and $M=nm$ in the original bound, so the rest of the analysis remains the same.

The second modification we did was that we decided not to use dynamic trees, because studies show that dynamic trees generally do not speed up max flow algorithms in practice. When doing a tree push on v, instead of using the dynamic tree, we follow a path of current edges until we reach an edge with capacity less than $e^*(v)$. This gives tree push a worst case time of $O(n)$ instead of $O(logn)$. This n propagates through the runtime analysis, to yield a running time of $O(nm+n^{2,5+\varepsilon}m^{0,5})$.

9 Tests

In this section we will describe what tests we have run on the algorithms, and what we expect to see from them. Section 9.1 contains a brief explanation of the verification we do to ensure that the max flow value returned by the algorithms is the correct one. Section 9.2 contain a list of the different types of graphs we will be running on.

9.1 Algorithm Correctness

To verify that our algorithms work, we implement them to return both the value of the max flow, and the residual network after all flow has been sent from s to t. After running the algorithm, we use the residual network in combination with the original graph, to calculate the flow on each edge in the graph. We then verify that the flow going out of s is the same value as the flow going into t and the value of the max flow returned by the algorithm. Additionally, we verify that the excess in all nodes apart from s and t are 0, and that there are no edges where more flow is sent than allowed by the capacity on the edge in the original graph. By this, we have ensured that the capacity and flow constraints are fulfilled. The last check we do is to make sure that no more flow can be sent from s to t. This is done by looking for an augmenting path.

With these checks, we can be absolutely sure that the values returned by the algorithm are the correct ones.

9.2 Graph generators

To generate graphs to test on, we use a range of popular graph generation algorithms. For examples of the output of each algorithm, please refer to Appendix A.

9.2.1 AC

The AC graph generator produces an asyclic graph where a node $v \in \{0, ..., n-1\}$ have edges to all nodes in $\{v+1, ..., n-1\}$, with capacities randomly generated in the range $[1, 10000]$. As a consiquence, the graph will be fully connected, but edges going back in the graph start out with capacity zero.

9.2.2 AK

The AK graph generator produces deterministic graphs for each value of n. These graphs are designed to be very hard for the Push-Relable algorithms to solve.

9.2.3 GENRMF

The GENRMF generator produces a special kind of graphs developed by Goldfarb and Grigoriads. It takes parameters a, b, c_{min} and c_{max}. The graph produced will consist of b layers of nodes, with a x a nodes in each layer. Each node in a layer have an edge connecting it to the two to four nodes ajacent to it, as well as a single edge to a random node in the next layer. The source node is placed in the first layer, and the target node is placed in the last layer. The capacities are randomly generated in the range $[c_{min}, c_{max}]$.

This means we will have $n = a^2b$ nodes, and $m = 4*a*(a-1)*b+a*(b-1)$ edges, which results in a relatively sparse graph.

We will use the generation in two modes, one which is long, where $a^2 = b$, and one which is flat, where $a = b^2$.

9.2.4 Washington

The washington generator is a collection of graph generators. We will use it to produce random level graphs. The random level graph is a graph where the nodes are laid out in rows and columns. Each node in a specific row has edges to all nodes in the following row. The source has edges to all nodes in the first row, and all edges in the last row has edges to the target.

References

[Alo90] N. Alon. Generating pseudo-random permutations and maximum flow algorithms. *Inf. Process. Lett.*, 35(4):201–204, August 1990.

[AO89] R. K. Ahuja and James B. Orlin. A fast and simple algorithm for the maximum flow problem. *Operations Research*, 37(5):pp. 748–759, 1989.

[AOT89] R. K. Ahuja, J. B. Orlin, and R. E. Tarjan. Improved time bounds for the maximum flow problem. *SIAM J. Comput.*, 18(5):939–954, October 1989.

[CH89] J. Cheriyan and T. Hagerup. A randomized maximum-flow algorithm. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, SFCS '89, pages 118–123, Washington, DC, USA, 1989. IEEE Computer Society.

[Che77] R. V Cherkasky. An algorithm for constructing maximal flows in networks with complexity of $O(V^2\sqrt{E})$ operations. *Math. Methods Solution Econ. Probl. 7*, pages 112–125, 1977.

[CHM90] Joseph Cheriyan, Torben Hagerup, and Kurt Mehlhorn. Can a maximum flow be computed in $O(nm)$ time? In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, pages 235–248, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[Din70] E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, April 1972.

[FF56] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Can. J. Math. 8*, pages 399–404, 1956.

[Gab85] Harold N. Gabow. Scaling algorithms for network problems. *J. Comput. Syst. Sci.*, 31(2):148–168, September 1985.

[Gal80] Z. Galil. An $O(V^{5/3}E^{2/3})$ algorithm for the maximal flow problem. *Acta Inf 24*, pages 221–242, 1980.

[GN79] Zvi Galil and Amnon Naamad. Network flow and generalized path compression. In *Proceedings of the eleventh annual ACM symposium on Theory of computing*, STOC '79, pages 13–26, New York, NY, USA, 1979. ACM.

[GR98] Andrew V. Goldberg and Satish Rao. Beyond the flow decomposition barrier. *J. ACM*, 45(5):783–797, September 1998.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, October 1988.

[Hoc98] Dorit S. Hochbaum. The pseudoflow algorithm and the pseudoflow-based simplex for the maximum flow problem. In *Proceedings of the 6th International IPCO Conference on Integer Programming and Combinatorial Optimization*, pages 325–337, London, UK, UK, 1998. Springer-Verlag.

[Kar74] A. V. Karzanov. Determining a maximal flow in a network by the method of pre-flows. *Soviet Math. Dokl.*, 15(2), 1974.

[KR92] V. King and S. Rao. A faster deterministic maximum flow algorithm. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, SODA '92, pages 157–164, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

[KRT94] V. King, S. Rao, and R. Tarjan. A faster deterministic maximum flow algorithm. In *selected papers from the third annual ACM-SIAM symposium on Discrete algorithms*, SODA, pages 447–474, Orlando, FL, USA, 1994. Academic Press, Inc.

[MKM78] V. M. Malhotra, M. P. Kumar, and S. N. Maheshwari. An $O(|V|^3)$ algorithm for finding maximum flows in networks. *Znf Process. Lett. 7*, pages 277–278, 1978.

[Orl13] James B. Orlin. Max flows in $O(nm)$ time, or better. In *Proceedings of the 45th annual ACM symposium on Symposium on theory of computing*, STOC '13, pages 765–774, New York, NY, USA, 2013. ACM.

[ST83] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Computer and System Sciences*, 24:362–391, 1983.

[Tar84] Robert Endre Tarjan. A simple version of karzanov's blocking flow algorithm. *Operations Research Letters*, 2(6):265 – 268, 1984.

A Graph Examples

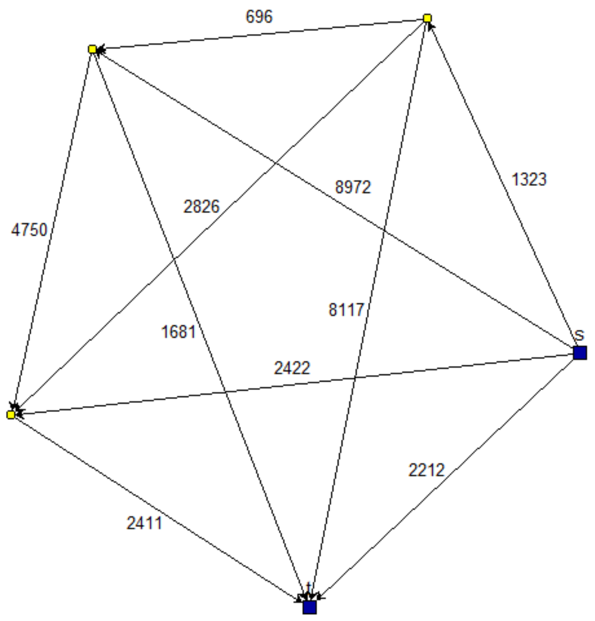

Figure 1: An example of the output of the AC generator, where $n = 5$

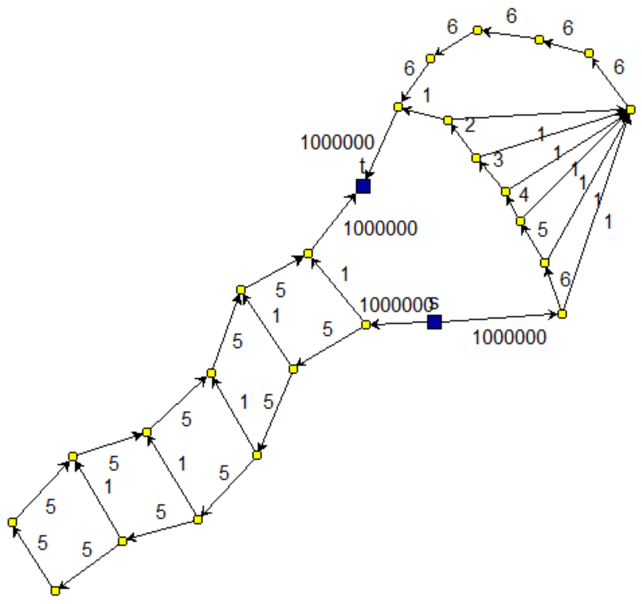

Figure 2: An example of the output of the AK generator, where $n = 26$

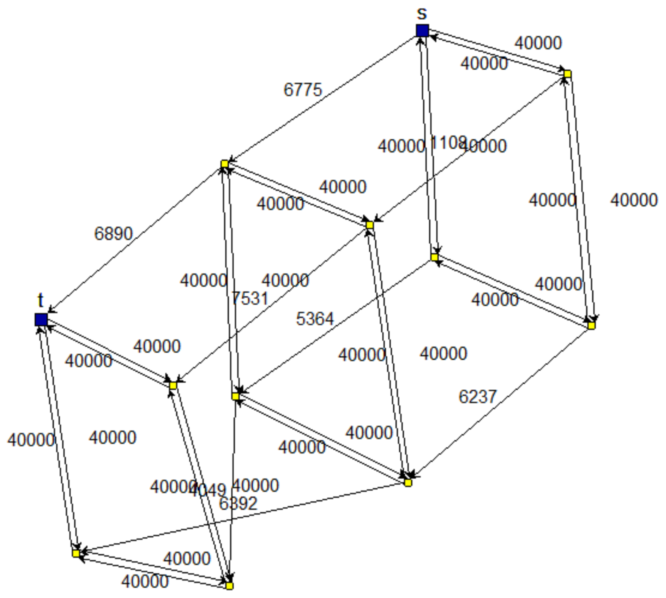

Figure 3: An example of the output of the GENRMF generator, where $a = 2$ and $b = 3$

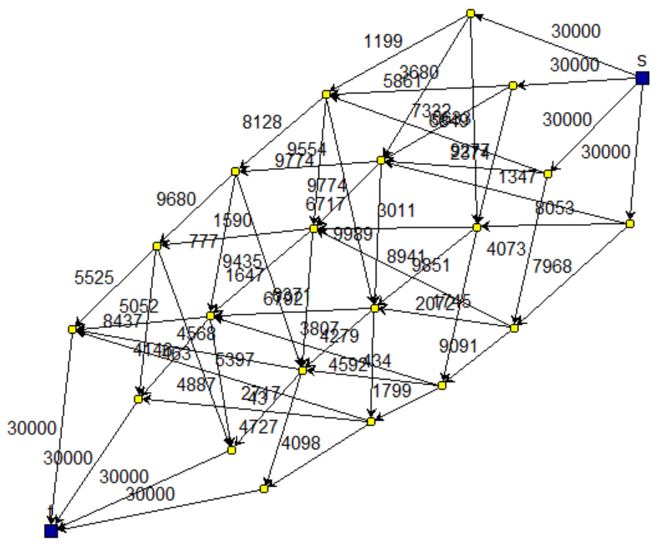

Figure 4: An example of the output of the washington level graph generator, with 5 rows and 4 columns