Survey

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

November 25, 2013

Contents

1 Introduction

The max flow problem is a directed 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. An example can be seen in Figure 1. The numbers of the edges signify the capacity constraint. It is not allowed to send more flow over the edge than this number.

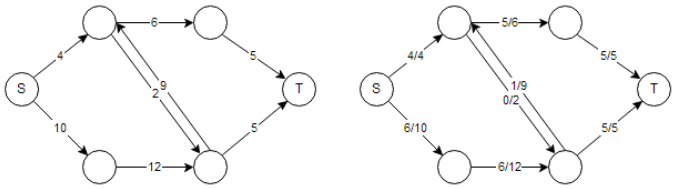

Figure 1: An example of a max flow problem and a solution to it

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 thesis, we will compare the practical running time on a selected subset of the max flow algorithms. We have decided only to consider max flow problems with integer capacities. The reason for this is that it makes it simpler to implement the algorithms if we don't have to take floating point errors into account.

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 5 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 symbolise 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 a directed edge where $u \in V$, $v \in V$ and $u \neq v$. We use n and m to symbolise the number of nodes and the number of edges in the graph, respectively. 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 a list of nodes $(v_1, \cdots, v_k)$ where $(v_i, v_{i+1}) \in E$ for $i = 1 \cdots k - 1$ and the list contains no duplicates.

Every edge (u, v) has a *capacity* associated with it denoted by $cap(u, v)$. The capacity of an edge must be a non negative integer. 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 $f(u, v)$. As with capacity, the flow on an edge must be a non negative integer. *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 $r(u, v) = cap(u, v) - f(u, v) + f(v, u)$. For edges $(u, v) \notin E$, we define $cap(u, v) = f(u, v) = r(u, v) = 0$. An edge (u, v) is said to be *saturated* if $r(u, v) = 0$, and the act of saturating an edge is changing the flow to make the edge become saturated.

A path $P = (v_1, \cdots, v_k)$ is said to be residual if $\forall i < k : r(v_i, v_{i+1}) > 0$.

An *augmenting path* $P = (v_1, \cdots, v_k)$ is a residual path in G where $v_1 = s, v_k = t$. In other words, an augmenting path is a path from s to t in the residual network, where it is possible to send more flow. The *bounding edges* of an augmenting path is the edges that have the minimum residual capacity of all edges in the path. As a consiquense, if flow was pushed on the path, theese edges would become saturated.

The *distance* between two nodes u and v is denoted $distance(u, v)$, and is the number of edges connecting nodes in the shortest residual path connecting the two nodes. If no such path exists, $distance(u, v) = n$.

The *excess* of a node v, $e(v)$ is how much flow currently resides in the node $e(v) = \sum_{u \in V} (f(u, v) - f(v, u))$. This may generally only be negative for the node s and positive for the node t and 0 for all other nodes, to a flow to be valid.

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. $\forall (u, v) \in E, f(u, v) \leq cap(u, v)$

The first is refered to as the flow conservation constraint, and the second is the capacity constraint. Some algorithms works by manipulating $preflow$,

which is a flow in the graph where the excess of nodes are allowed to be positive, thus violating the flow conservation constraint.

Tables of all definitions can be found in Appendix A.

3 Paradigms

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

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. 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.

The number of augmenting paths an algorithm finds depends on the order in which it finds them. For instance if you consider the graph in Figure 2, an augmenting path can be made with minimum residual capacity 1. Following that path, another can be found with residual capacity 1 if you follow the middle edge in the opposite direction. So on this graph, anywhere from 2 to 1000 augmenting paths can be found. If we didn't have integer capacities, the

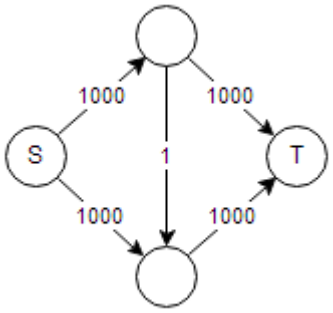

Figure 2

middle edge could have an infinitely small capacity, resulting in an infinite number of augmenting paths.

The idea that no more flow can be sent if no augmenting path can be found 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.

For an example of an augmenting paths algorithm, see section 6.

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 increase the distance from s. So an edge (u, v) only exists in the layer graph if $distance(s, u) < distance(s, v)$, and $r(u, v) > 0$. 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 updating the residual network of the original graph with the blocking flow, and calculating a new layer graph, which will have a bigger k. This process repeats until all augmenting paths have been found.

For an example of a blocking flow algorithm, see section 7.

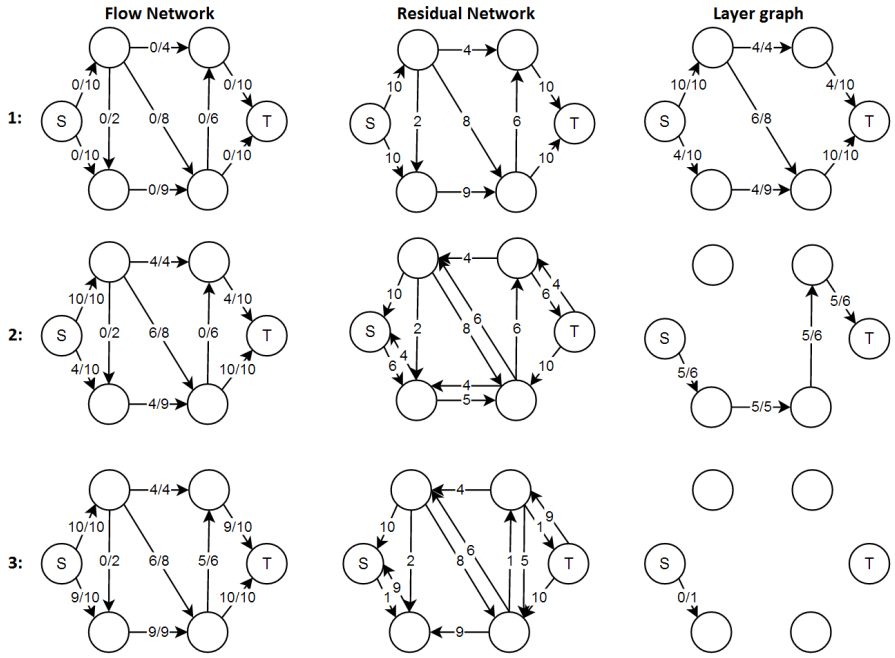

Figure 3: An example of running a blocking flow algorithm.

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 works by

manipulating a preflow in the algorithm, by violating the flow convservation constraint throughout the algorithm, and pushing excess between individual nodes by adding flow on the edges in the graph.

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.

A relabel should only be performed if a node has some excess, but no place to send it. If a node recieves excees, it can always send it back to the node it recieves it from, so if it has no place to send its excess, there must be some neighbor with a higher label, where the excess can be sent.

What is going to happen when running a push relabel algorithm is a sequence of pushes and relabels that move excess around the graph from node to node.

At some point, the nodes will start to be relabed above n. When this happens, t is no longer reachable. A result of having labels above n is that excess will begin to be pushed back towards s. Eventually, all the exces 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 a valid flow, which is in fact the max flow.

For an example of a push relabel algorithm, see section 8.

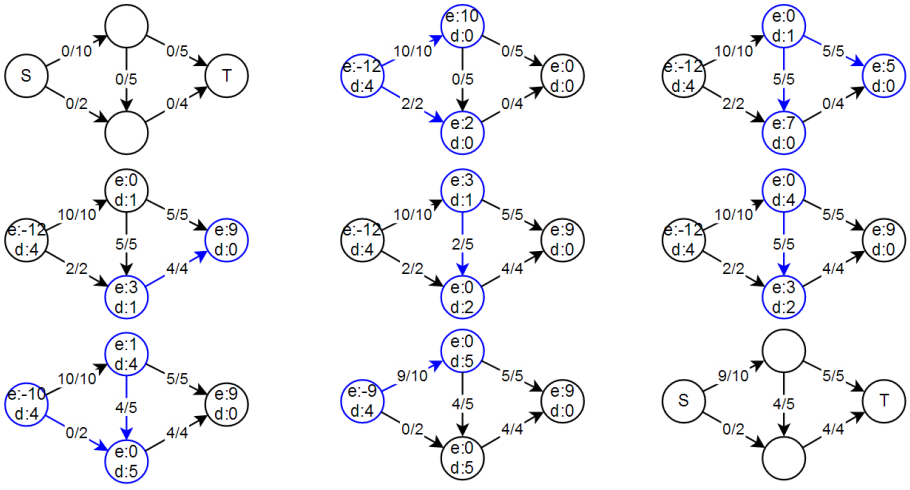

Figure 4: An example of running a push relabel algorithm.

4 Dynamic Trees

Dynamic trees, also called Link Cut Trees, is a datastructure that was presented by D. D. Sleator and R. E. Tarjan in 1983 [ST83]. It is a tree structure where each node can have a number of values assigned to them, such as cost. Updates done to the tree are performed in amotized $O(\log k)$ time, where k is the size of the tree the update is performed on. The operations supported by dynamic trees are

Link(a, b)
: Make a a child of b.

Cut(a)
: Remove a from its parent, making a a root node in its own tree.

SetCost(a, v)
: Set the cost of a to c.

GetCost(a)
: Returns the cost of a.

AddCost(a, c)
: Modify the cost of a and all of its ancestors by adding c.

GetPathLength(a)
: Returns the number of nodes on the path from a to the root of the tree a is in.

GetRoot(a)
: Returns the root of the tree a is in.

GetChildren(a)
: Returns the children of a.

GetMinCostNode(a)
: Returns the first node on the path from a to the root of the tree that has the minimum cost of all the nodes on the path.

GetBoundingNode(a, c)
: Returns the first node on the path from a to the root of the tree that has a cost less than or equal to c.

In max flow algorithms, a dynamic forest is typically used to represent paths of nodes in a way such that each node in the graph has a corrosponding node in the dynamic forest. The nodes in the graph will have an edge which is the prefered edge to move flow along. Linking a to b represents that the edge (a, b) is the prefered edge of a. The cost of the nodes in the dynamic

forest will represent the residual capacity on the prefered edge. This way, a *Tree Push* can be performed, where moving flow along a path can be done in $O(\log n)$ time instead of $O(n)$ time. The specifics of how this is done depends on the max flow algorithm, but one way is to perform the operations

$$
\begin{aligned}
&v_j \leftarrow \text{GetMinCostNode}(v_i)\\
&c \leftarrow \text{GetCost}(v_j)\\
&\text{SetCost}(v_j, 0)\\
&\text{Cut}(v_j)\\
&\text{AddCost}(v_i, -c)
\end{aligned}
$$

After this a new prefered edge will have to be found for v_j, which means v_j should be linked, and its cost should be set to the residual capacity of the new prefered edge.

5 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	$O(nmU)$	[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	$E\left(\min\begin{pmatrix} nm\log n \\ nm+n^2\log^2 n\end{pmatrix}\right)$	[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	$O(\min\{n^{\frac{2}{3}},\sqrt{m}\}m\log(\frac{n^2}{m})\log U)$	[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 $O(nmU)$ for integer capacity constraints, and is not guaranteed to terminate on real valued constraints. In 1972, J. Edmonds and R. M. Karp [EK72] observed that if the augmenting path found in the algorithm by Ford and Fulkerson always is a shortests augmenting path, the maximum number of augmenting paths is $O(nm)$. This algorithm runs in $O(nm^2)$ time. We explain this algorithm in more detail in Section 6.

About the same time, in 1970, E. A. Dinic [Din70] published another improvement over the algorithm by Ford and Fulkerson. The paper by Dinic also includes the algorithm by Edmonds and Karp, but Dinic includes additional techniques to reduce the running time to $O(n^2m)$. 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 augmented with the blocking flow. With this new residual network, he 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 7.

The first optimization to Dinic's algorithm 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 $m = \Theta(n^2)$. There have been serveral publications that use 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 time $O(n^2\sqrt{m})$. He groups some consecutive layers together, and runs a combination of Dinic's and Karzanov's algorithms. 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. Galil 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 $O(n^3)$ time, and then they combined it with dynamic trees and got an algorithm that runs in time $O(nm \log \frac{n^2}{m})$. Details on the algorithms presented in this paper can be found in Section 8. 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 and 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 and Tarjan. The general idea was to find the lowest integer number, called the excess dominator, that is a power of two 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 time $O(nm + n^2 \log U)$.

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 [Tar84]. 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], and 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 nonconstant 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 $O(nm + n^2 \log^3 n)$, 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 $O(\min\{nm \log n, nm + n^2 \log^2 n\})$. 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 technique from [Alo90].

The paper by V. King and S. Rao [KR92] builds on top of [AO89]. It modifies a special subroutine that selects 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 9.

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 $O\left(\frac{n^2}{\log n}\right)$ if $m = O(n)$.

6 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 limits the number of augmenting paths to $O(nm)$, and brings the worst case running time from $O(nmU)$ to $O(nm^2)$.

6.1 The Algorithm

The algorithm works by repeatedly 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 $\min_{i=1,\cdots,k-1} r(v_i, v_{i+1})$, 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 residual 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.

6.2 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 if the distances never decrease, an edge can only be saturated n times.

The only way we modify the distances is by pushing flow along the augmenting path. Saturated edges are effectively removed, and back edges are added back in if their residual capacity was zero. Removing an edge can not reduce the distance to a node. Adding an edge could, but the edges (v_i, v_{i-1}) we might add point the opposite way on the augmenting path which was found in a breath first search. Adding (v_i, v_{i-1}) back in can not reduce the distance to v_{i-1}, because the distance to v_i was already greater than the distance to v_{i-1}.

To summerize, there are m edges that can be saturated n times, each time requiring a breath first search which takes time $O(m)$. This results in the running time of $O(nm^2)$.

7 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 the algorithm by Edmonds and Karp.

7.1 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 property of 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 the algorithm by Edmonds and 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 in Section 6. 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.

7.2 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 reason is the same as in Section 6.2. The distance to a node never decrease because the nodes in an augmenting path have increasing distance from s. Instead of this being due to using a breath first search, it is because the level graph only contains edges to nodes that has a higher distance from s. Since the distance to t never decrease, and we push along all augmenting paths of length k, all subsequent augmenting paths must have a length greater 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 $O\left(n(m+nm)\right) = O(n^2m)$.

Dynamic trees can be utilized to find the blocking flow in $O(m \log n)$ time, reducing the running time to $O(nm \log n)$, but dynamic trees was not introduced until 1983 by D. Sleator and R. E. Tarjan [ST83].

8 Goldberg Tarjan 1988

8.1 Introduction

The Push Relabel algorithm of Goldberg and Tarjan [GT88] works by manipulating the preflow in a graph. First step is saturating all the edges exiting the source. Next step is moving the excess into nodes estimated closer to the sink. If at some point the excess of a node can not reach the sink, the excess is moved back into the source. In the end the preflow of the algorithm satifies the flow conservation constraint and thus the preflow is an actual flow, infact it is the maximum flow.

Section 8.2 describes a version of the algorithm which is quite simple and runs in $O(n^3)$ time. Section 8.3 describes and analyzes a new algorithm which uses the dynamic trees datastructure, described in 4, and modifies the $O(n^3)$ algorithm slightly. Any actual modifications done to the implementations of the algorithms are described in 8.4. Finally the last section 8.5 describes in which direction the project can be taken given more time.

8.2 The Push-Relabel algorithm with a $O(n^3)$ running time

In this section a simple $O(n^3)$ version of the Push-Relabel algorithm will be described and analyzed.

8.2.1 Notation

The algorithm estimates the distance from nodes to the source/sink by setting a label $d(v)$ to each node $v \in V$. The label $d(s)$ of the source is set to n and the label $d(t)$ of the sink is set to 0, and neither is changed throughout the algorithm. The label of a node v has a constraint based on its edges and neighbours' labels: $\forall w \in V : r(v,w) > 0 \implies d(v) \leq d(w)+1$. A labbelling fulfilling this constraint is called *valid*. The idea of the algorithm is that it always pushes flow to nodes with a lower label.

In the theory of the algorithm a notation f' will be used for allowing the flow to be negative. $f'(u,w) = f(u,w) - f(w.u)$, meaning if f' is negative there is flow on the edge going the opposite direction. This simplifies excess to: $e(v) = \sum_{u \in V} f'(u,v)$ ~~instead of the earlier notion of $e(v) = \sum_{u \in V} f(u,v) - f(v,u)$.~~ The constraint $\sum_{w \in V} f'(v,w) = 0, \forall v \in V \setminus \{s,t\}$ is refered to as the anti-symmetry constraint.

An edge (v,w) is stated as *eligeble* if it has $r(v,w) > 0$

A node v is active if $e(v) > 0$.

8.2.2 The algorithm

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: Transfer $\delta = \min(e(v), r(v,w))$ 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: $d(v) \leftarrow \min\{d(w)+1 | (v,w) \in E, r(v,w) > 0\}$

6: **end procedure**

The two key methods of the algorithm can be seen in Algorithm 1. They are only applicable to *active* nodes.

The job of the Push procedure is to move flow from one node v to another, w. The amount of flow~~ed~~ able to be moved are constrained by the capacity of the edge, (v,w)~~, linking the 2 nodes~~ and on the excess of v. The excess of a node never becomes negative, and the capacity constraint is never violated.

Pushes can only happen on edges where there is a positive residual capacity and the label $d(v)$ is one higher than $d(w)$.

Since the Push only applies in under certain conditions the Relabel procedure's job is to make sure that these conditions can occur. Otherwise no flow can be moved in the graph. All nodes in the graph, besides the source, have their initial labels set to 0. The label of the source is n. This means that in the beginning no flow can be pushed around. The minimum function in the Relavek procedure finds the neighbouring nodes with minimum labels c. This means than when a node v is relabeled to $c+1$, v can now push to all these nodes, thus enabling the Push operation.

To start the algorithm all edges going out from the source are saturated, meaning all these nodes becomes active and there is some flow to be pushed around. In case these edges are a minimum cut all the flow will be moved to the sink. If not, some of the flow must be moved back into the source. All edges not outgoing from the source has their flows initialized to 0.

The algorithm uses the edge list for each node to determine what kind of operation to do. Each node keeps a pointer, called the *current-edge*, to an edge in its edge-list. When the algorithm works on a node it looks at the current-edge and if the edge fulfills the requirements for an Push operation it makes one. If the Push operation does not apply it sets the current-edge to be the next edge in the node's edge-list. If the current-edge was the last edge in the edge-list this operation can not be done and instead it relabels the node and sets the current-edge to be the first edge in the list of edges. All this logic is encapsulated in the PushRelabel method, the pseudocode can be seen in Algorithm 2.

All the requirements are fullfilled when applying a Relabel operation in this manner. The requirements for applying a Relabel operation on node v were that v is active and $\forall w \in V, r(v,w) > 0 \implies d(v) \leq d(w)$. v must be active as the PushRelabel procedure is used. Either $d(v) \leq d(w)$ or $r(v,w) = 0$. $r(v,w) = 0$ because the labelling $d(v)$ has not changed since (v,w) was the current-edge, so all the residual capacity must have been pushed on if it could. If it pushed without using all the residual capacity it must be a non-saturating push and v would not be active. if it could not push at all then $d(w) \geq d(v)$, and the labelling $d(w)$ could only have increased since then. Lastly $r(v,w)$ could only have increased since (v,w) was the current edge by pushing from w to v but that would imply $d(w) > d(v)$. So in any case doing the Relabel as the algorithm does it is a valid operation.

The PushRelabel works on a single active node. The only thing left to describe is how the algorithm chooses which node to apply the method on.

The way the algorithm keeps track of which are nodes are active are by keeping a first-in-first-out queue over all active nodes. When a node is taken from the front of the queue the algorithm keeps applying the PushRelabel to it, until it either gets relabeled or it becomes inactive. If it gets relabeled it

Algorithm 2 The $O(n^3)$ PushRelabel procedure

Require: v is active

```
1:  procedure PushRelabel(v)
2:      Edge e ← current edge of v
3:      if Push(e) is applicable then
4:          Push(e)
5:      else
6:          if e is not the last edge of the edgelist of v then
7:              set the current edge of v to be the next edge
8:          else
9:              Set the current edge of v to be the first edge in the edgelist
10:             Relabel(v)
11:         end if
12:     end if
13: end procedure
```

is added back into the queue at the rear. This means that when working on node v with a current edge (v, w), v is deleted from the queue, and v and/or w may be added to the queue. This way of choosing which nodes to work on can be seen in the Discharge function in Algorithm 3. The initialization and main loop can be seen in the same pseudocode.

8.2.3 Correctness

To prove the correctness it is going to be shown that:

1. The labels of the nodes stay valid throughout the execution of the algorithm.
2. It is always possible to apply either a Relabel or a Push to an active node, meaning excess can not be stuck at such a node.
3. If a node is actice a path exist to the source, so flow can always be moved back.
4. The number of relabels of a node is bounded.
5. After the algorithm no residual path exists from the source to the sink

The first item is to make sure that the algorithm doesn't miss a chance to push on a residual edge. The second to fourth items guarantee that excess are moved around and that ~~in the end~~ the excess not moved to the sink will be moved to the source, turning the preflow into an actual flow. The last item is combined with a classic theorem from Ford and Fulkerson, [FF56], to prove that the flow is actually a maximum-flow. The following paragraphs

Algorithm 3 The Goldberg Tarjan Initialization and Main-Loop parts

```
1:  function MaxFlow(V, E, s, t)
2:      d(s) ← n
3:      for all v ∈ (V \ {s}) do
4:          d(v) ← 0
5:          e(v) ← 0
6:      end for
7:      for all (v, w) ∈ E do
8:          f(v, w) ← 0
9:      end for
10:     for all (s, v) ∈ E do
11:         Send max capacity flow through (s, v) update (v, s) accordingly
12:         Update excess of v
13:         add v to the back of Q
14:     end for
15:     while Q ≠ ∅ do                                   ▷ Main-Loop
16:         Discharge
17:     end while
18:     return e(t)
19: end function

20: procedure Discharge
21:     Node v ← first element of Q, removed from the queue.
22:     repeat
23:         PushRelabel(v)
24:         if w becomes active then
25:             Add w to the back of Q
26:         end if
27:     until e(v) = 0 or d(v) increases
28:     if e(v) > 0 then
29:         add v to the back of Q
30:     end if
31: end procedure
```

shows all the items

The labels in the graph are valid when the algorithms has initialized, since $\forall v \in V \setminus \{s\} : d(v) = 0$ and $\forall v \in V : r(s, v) = 0$. A Relabel operation to a node v keeps the label valid. This is the case as it assigns a value to $d(v)$ that is 1 higher than the minimum label of all the neighbours w, where $r(v, w) > 0$. Doing a push operation on the edge (v, w) may make (w, v) eligeble and may remove (v, w). This keeps the labels valid because pushing

on (v, w) means $d(v) = d(w) + 1$, so adding edge (w, v) is fine. Removing an edge means removing the constraint, so that also keeps the labelling valid. This means that the labelling stays valid throughout the execution of the algorithm.

If a node v is active and the algorithm is in a consistent state, meaning the current flow is a preflow and the labels are valid, it is always possible to apply either a Push or a Relabel operation to it. A relabel is only applicable when no pushes can be done since it is part of the requirements of the Relabel method. When a relabel operation assigsn $d(v)$ to a node v, it opens up the possibility of at least pushing to a single node, one of the nodes with label $d(v) - 1$. Hence one of the actions are always applicable.

The application of a relabel operation to a node v increases its label. This is true since when applying a relabel operation to node v, the labels of all the neighbours where v has an eligeble edge to have labels higher or equal to v's, it is part of the requirements. This implies that the operation $\min\{d(w) + 1 | (v, w) \in E, r(v, w) > 0\}$, has a value $> d(v)$. As the Relabel operation is the only one changing the labels, this implies that the labels are always increasing.

Lemma 8.1 *Given a preflow f if v is active, $e(v) > 0$ then the source s is reachable from v in the residual graph*

Proof Proof by contradition. Denote the set of reachable vertices from v in the residual graph S, and assume that $s \notin S$, Let $\bar{S} = V \setminus S$. Since there can be no residual edge from a vertics in S to a vertex in $\bar{S}$ this means that for every pair $u \in \bar{S}$ and $w \in S$, $f'(u, w) \leq 0$

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

$\sum_{u,w \in S} f'(u, w)$ is equal to 0 because of anti-symmetry. Since all nodes have excess ≥ 0 this means that all nodes $w \in S$ has $e(w) = 0$, in particular v, leading to a contradiction. ∎

~~Since~~ according to lemma 8.1 an active node v_k have a path to s, denote it $v_k, v_{k-1}, \ldots, v_o, s$, this gives an upper bound on the label. ~~Because~~ when relabling all the nodes in the entire path $v_k, v_{k-1}, \ldots, v_o, s$ the label difference of each edge is $d(v_i) - d(v_{i-1}) \leq 1$, the maximum possible labelling of v_k becomes

$$\begin{aligned} d(v_k) &\leq d(s) + (d(v_0) - d(s)) + \cdots + (d(v_k) - d(v_{k-1})) \\ &\leq d(s) + k + 1 \\ &\leq n + (n-1) \\ &\leq 2n \end{aligned}$$

Since the labels are bounded and each Relabel operation increases the label of a node, this means that the number of Relabel operations are bounded.

To further argue about the correctness of the algorithm the classic theorem from the article by Ford and Fulkerson [FF56] is used:

Theorem 8.2 *A flow f is maximum if and only if the exist no augmenting path. That means t is not reachable from s in the residual network*

Lemma 8.3 *Given a preflow f a valid labelling d then the sink is not reachable from the source in the residual graph*

Proof Proof by contradition. Assume there exist a residual path $s = v_0, v_1, \ldots, v_l = t$, since the labelling is valid $d(v_i) \leq d(v_{i+1}) + 1$ for all the edges in that path, since $l < n$ that means $d(s) \leq d(t) + l = 0 + l < n$, but that's a contradiction since $d(s) = n$ ∎

Theorem 8.4 *If the algorithm terminates the preflow is a maxflow, meaning the algorithm is correct*

Proof When the algorithm terminates the excess of all nodes $v \in V \setminus s, t$ must have $e(v) = 0$, this means the preflow is a valid flow. Theorem 8.2 and Lemma 8.3 together means it must be a maximum flow. ∎

The next section analyzes the running time of the algorithm. It will be done by bounding the number of relabel- and push-operations being made.

8.2.4 Running time

Less than $2n^2$ relablings are being done in the algortihm, since each node can be relabeled at most 2n times, and only $n-2$, $V \setminus \{s, t\}$, nodes are being relabeled.

To analyze the number of pushes being done, they will be split into 2 different

types of pushes, *saturating* and *non-saturating* pushes. A saturating push is when the pushing node has enough excess to use the full residual capacity of the edge, said in other words: In the minimum in the push operation, $\min(e(v), r(v,w))$, either the 2 values are equal or the second is smaller. Non-saturating pushes are then the other case, where the excess in the node is not enough to fully utilize the residual edge.

At most $2nm$ saturating pushes are being done in the course of the algorithm. After a saturating push has happened on edge (u,v) a push has to be done on (v,u) before another push on (u,v) can happen. Since the pushing node has to have a label 1 higher than the node being pushed to, 2 pushes along the same edges has to have had relabels happen in between leading to at least a label of 2 higher when the next saturating push happens on the same edge. As the max label is $2n$ this means that at most n saturating pushes can be done on any edge, leading to a maximum of $2n$ saturating pushes for an edge and its linked edge. The number of sets of edge + linked-edge is at most m meaning the maximum number of saturating pushes happening in total are $2nm$.

The number of non-saturating pushes depends heavily upon which order the Push and Relabel operations are applied in. In the next sections it will be shown that the way the Discharge methos does it bounds it to $O(n^3)$.

To analyze the Discharge operation the concept of *passes* over the queue Q is used. The first pass, pass one, consists of applying Discharge to all the nodes added in the initialization of the algorithm. pass $i+1$ consists of treating all the ones added in pass i.

Lemma 8.5 *The maximum number of passes over the queue Q is $4n^2 = O(n^2)$*

Proof A potential function is used. $\phi = \max\{d(v)|\text{v is active}\}$. If over a pass no relabels are done, all the excess are moved to nodes with lower distances decreasing ϕ. If a relabel is happening and it is increasing ϕ this means that the change in $\phi \leq$ Change-in-label. The maximum changes in labels that can happen was shown in section 8.2.4 to be $2n^2$. Since the maximum of passes where it decreases are then also $2n^2$, the total amount of passes are $4n^2 = O(n^2)$ ∎.

Since all the nodes $v \in V \setminus \{s,t\}$ can at most have one non-saturating push per pass as they become inactive afterwards, the non-saturating pushes are bounded by $(n-2)4n^2 \leq 4n^3$.

Theorem 8.6 *The PushRelabel implementation of the algorithm leads to a running time of $O(nm) + O(1)$ per non-saturating push.*

Proof Let v be a vertex in $V \setminus \{s,t\}$ and δ_v the number of edges in v's edge list. Each node only runs through its edge-list a certain number of times. At most $2n$ relablings are happening to each node and each contribute 2 run-throughs, since before each relabling the entire list has been run through and the list is run through once in the Relabel operation itself. This leads to a total of $\leq 4n = O(n)$ runs through the edge list, for a total of $O(n\delta_v)$ work being done pr. node. Meaning a total of $\sum_{v \in V \setminus \{s,t\}} n\delta_v = O(nm)$

The rest of the work done by the algorithm comes from the pushes. Each Push operation is constant, $O(1)$, work. The number of saturating pushes was bound to $O(nm)$. Adding the work, for the saturating pushes, to the work done by the run-throughs/relabels gives the theorem. ∎

Combining lemma 8.5 with theorem 8.6 gives a running time of $O(n^3)$, since $(n-1) \leq m \leq n^2$.

8.3 The Push-Relabel algorithm with dynamic trees and a $O(nm \log \frac{n^2}{m})$ running time

The Push-Relabel algorithm described and analyzed in the following sections are basically a slight modification of the $O(n^3)$ one described in the previous parts. The idea is to use the dynamic trees data structure, described in section 4, to bring the cost of doing non-saturating pushes down. To do this the PushRelabel method of the previous section has been replaced by a new version called *TreePushRelabel*. A new function called *Send* is also added. The new pseudocode can be seen in Algorithm 4.

8.3.1 The algorithm

The dynamic tree datastructure uses amortized $\log k$ time per operation, where k is the path length in a dynamic trees. The trees are introduced to bring the time spent on each non-saturing push down to sub-constant. The issue is that if a node v has done a non-saturing push on edge e to node w then the next time v gets any excess it could likely use the same edge again each time costing a constant amount of time. The idea of using dynamic tree are then to save this edge in the tree by setting the parent of v to be w in the tree. This way an entire path can be built. The algorithm can push flow along such a path of length k in $\log k$ time.

The new algorithm has to maintain these paths so that only nodes where flow can be pushed between are linked in the tree. This means that if a node v is relabeled the algorithm cuts all the children of v since the new label no longer allows for pushes from them to v.

Each node v in the dynamic tree has a cost associated to it. This cost is used to denote how much residual capacity the edge between v and its parent has. This means that the dynamic tree has a value on the residual

capacity and based on the flow/capacity values on the edge a similarly residual capacity can be calculated. These 2 values are not synchronized since that would mean updating all the edges in a tree-path leading to a $O(k)$ time operation instead of $\log k$. To solve this issue an invariant is introduced saying that every active node v, which was defined as $e(v) > 0$, is a root in the dynamic tree. To keep this invariant and also maintain the intended use of the dynamic tree all the excess added onto a node w in a path has to be pushed to the root. It may happen that a node v on the path has a too low residual capacity to allow all this excess through. In this case the algorithm pushes as much flow as v can handle through and cuts the edge between v and its parent. afterwards it repeats this operation until all the excess is pushed to roots of w or w itself becomes a root. The Send operation does all this.

In case a node is a root the stated residual-capacity/cost is set to infinity. In the initialization all nodes in the graph each have a node representing them in the dynamic tree datastructure, each which is a root and has infinity as cost.

To bound the cost of each dynamic tree operation a node is only linked to its parent if the new combined path size stays below a constant k. This means that if a Push from v to w apply, either the two nodes are linked and a Send operation is applied to v or a Push happens from v to w followed by a Send from w. These different if-branches are part of the new TreePushRelabel method which replaces the old PushRelabel procedure.

8.3.2 Correctness

Parts of proof of correctness follows from the correctness from the previous algorithm. This is the case as the algorithm still does a relabel at the same time as before, and because the Send method is basically a bunch of Push operation done together.

The Send operation only sends flow allowed by the residual capacity, and it only links v to w if $d(v) = d(w) + 1$. The link is cut if v or w is relabeled. All this combined means that all the 'pushes' the Send method does are legal and hence does not break the correctness.

The only issue left is termination. If a cycle exist in the dynamic tree the algorithm will never terminate as it would keep trying to push the flow closer to the root, but there are no root. As described the algorithm only links v to w if $d(v) = d(w) + 1$, so a cycle can not happen.

This means the algorithm terminates and outputs the correct result.

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, also for each child u Update the edge (u, v) and its linkededge with the values from the dynamic trees
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 δ value of flow in the tree by calling ADDCOST($-\delta$)
26: **while** FINDMINVALUE(v) $= 0$ **do**
27: $u \leftarrow$ FINDMIN(v)
28: Update the edge $(u, parent(u))$ and its linkededge with the values from the dynamic trees
29: CUT(u)
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**

8.3.3 Running time

Again the concept of passes over the queue needs to be used to make the following analysis. Nothing has changed from lemma 8.5, so the number of passes are still $4n^2$.

Lemma 8.7 *The maximum number of additions of nodes to Q is $O(nm + n^3/k)$*

Proof A node is added to Q when it is relabeled or when its excess is increased from 0. The total number of relabels were bounded to $2n^2$. The excess only increases when a Push and/or Send operation has been done. This can happen in 2 different cases the first, labelled a, is when the 2 trees are small enough to be linked and thus they are linked and a Send operation is done. In the second case, labelled b, the trees are too big to be linked, so a Push operation is made followed by a Send. Algorithm 4 showed the pseudocode for it. The number of additions to Q are in both these cases equal to the number of cuts being done in the Send operation + perhaps 1 additional per call of the Send operation. When a cut happens in the Send operation it corresponds to a saturating push, when it happens just before the Relabel method it corresponds to the run-through of the node's edge-list. These were previously bounded to both be $O(nm)$, giving a bound on the number of cuts. The number of links is at most the number of cuts $+(n-1)$.

To bound the number of Send operations the occurences of a and b is bound.

The number of times a can happen is at most $O(nm)$, the maximum amount of link operations.

To bound b the concept of non-saturating occurences is used. A non-saturating occurence is when no cut happens in the Send operation, since the number of saturating occurences has already been bound to $O(nm)$, this should suffice.

Some notation: The dynamic tree containing node v is called T_v. If b happens that means that $|T_v|+|T_w| > k$, that means that either T_v or T_w has a size $> k/2$ in which case the tree is noted as *large* otherwise it is *small*.

First look at the case where T_v is *large*. Since this is a non-saturating occurrence all the excess is moved from node v to the root. Meaning this can only happen once per pass. If the tree T_v has changed(linked/cut) the cost of the non-saturating operation is paid for by this operation. This happens at most $O(nm)$ times over the course of the algoritm. If the tree has not been changed since the beginning of the pass the cost is paid for by the tree T_v. Since at most $n/(k/2) = 2n/k$ *large* trees exist at a given pass, the total cost is $4n^2 * 2n/k = O(n^3/k)$ over all passes.

In the case that T_w is *large* a similarly argument can be made also leading to a cost of $O(nm + n^3/k)$

Adding all the costs together gives a bound of $O(nm+n^3/k)$ additions ∎

The next theorem is quite like 8.6.

Theorem 8.8 *The Push-Relabel algoritm using dynamic trees has a running time of* $O(nm \log k) + \log k$ *for each addition of a node to the queue* Q.

Proof Since the algorithm bounds each dynamic tree to a maximum size of k that means each tree operation costs $O(\log k)$. Each tree-PushRelabel operation takes $O(1)$ time + $O(1)$ tree-operations + $O(1)$ tree-operations for each cut either happening in the Send method or just before a Relabel. Just as in theorem 8.6 the number of tree-PushRelabels are $O(nm)$ plus some extra. In 8.6 the extra was bounded by the number of pushes, in this theorem it's bounded by the number of nodes added to Q, each addition leading to $O(1)$ tree-operations. Putting all these facts together gives the theorem. ∎

Theorem 8.8 and lemma 8.7 combined gives a running time of $O(nm \log k + (nm + n^3/k) \log k)$ setting $k = n^2/m$ gives a final running time of the dynamic version of the Push-Relabel algorithm of $O(nm \log \frac{n^2}{m})$

8.4 Implementation modifications

This section describes some of the modification done to actually implement the algorithm.

Two different version of the Push-Relabel algorithm have been implemented. One using dynamic trees and one without them.

The algorithm without dynamic tree swaps the edges in each edge-list for a node. Is has the invariant that each edge with 0 residual capacity are at the end of the edge-list. This has a small overhead cost, $O(1)$ per push operation, to maintain but allows the algorithm to end it run-through of the edge-list as soon as it sees the first 0 residual capacity edge.

When the algorithm with dynamic trees terminates some of the nodes might still be linked in a dynamic tree, meaning the calculated residual capacity on the edges might be wrong. To fix this the algorithm runs an extra procedure after it has terminated. The procedure runs through the dynamic trees and looks for any linked nodes, if it finds some, it updates the graph with the values from the tree.

8.5 Future work

In this section some one the ways the project could be taken, if given more time, is described.

Since the labelling of the nodes are a lower bound on the distances to the target if $< n$ and to the source if $> n$ it is a viable option to try to match the exact distances by running a couble of breadth-first searches and set the distance labels based on these. It could be interesting to do some testing of heuristics stating when and how often to run these searches.

9 King Rao 1992

V. King and S. Rao published in [KR92] an algorithm 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 called the game, that selects current edges. The current edge problem is about determining which edge to push on when pushing excess from a node.

The game 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. The algorithm by Goldberg and Tarjan [GT88] chooses the edges in any order. By selecting a specific order that aims to more effectively spread the flow in the graph, the worst case time can be reduced.

In Section 9.1, 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 9.2. Section 9.3 will contain the main algorithm, and its relation to the game, and in Section 9.5 , we will show how the algorithm achieves the runtime of $O(nm + n^{2+\varepsilon})$. Finally, the algorithm turned out to have some issues in practice, especially related to memory consumption. In Section 9.6 we will describe the modifications we have done to make the algorithm more usable in practice, including reducing the memory requirement while staying within the same theoretical bound.

9.1 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 9.3. 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 amount 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.

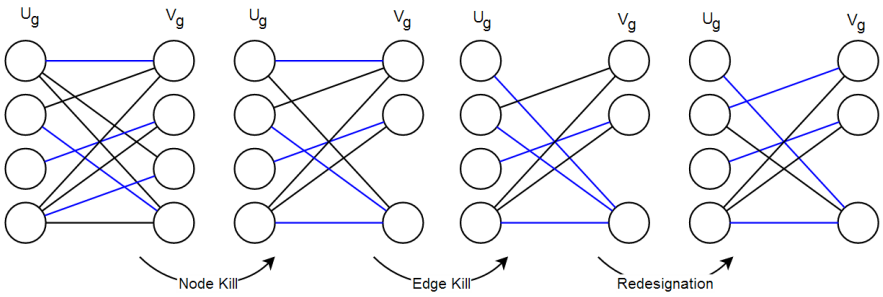

Figure 5: An example of moves in the game. The adversary gains two points from these moves.

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.

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 $U'_g = \{u \in U_g \mid \text{degree}(u) > l\}$ as the subset of U_g where we use the advanced strategy.

Algorithm 5 The game of [KR92]

```
1: procedure AdversaryNodeKill(v)
2:     Perform AdversaryEdgeKill on all edges incident to v ∈ V_g
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 ∈ 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) ≤ l then
15:        Designate any edge incident to u ∈ U_g
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), rl(v) + 1] then
23:        erl(v) ← rl(v)
24:        if erl(v) = t then
25:            Reset()
26:        end if
27:    end if
28: end procedure
29: procedure Reset
30:    k ← t
31:    while |U_{k−2}| ≥ (r_{k−1}l)|U_k|/4 do
32:        k ← k − 2
33:    end while
34:    Set erl(v) ← rl(v) for all v ∈ V_{k−2}
35:    Undesignate the designated edge for all u ∈ U_k
36:    Set erl(v) = rl(v) = 0 for all v ∈ V_k
37:    Designate an edge for all u ∈ U_k
38: end procedure
```

We define the ratio $r(v)$ of a node $v \in V_g$ as $r(v) = \frac{\text{degree}_{\text{designated}}(v)}{\text{degree}_{\text{initial}}(v)}$, where $\text{degree}_{\text{designated}}(v)$ is the number of designated edges to v from nodes in U', and $\text{degree}_{\text{initial}}(v)$ 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

$$r_i = 2^i r_0 \forall i \in \{1, ..., t\}$$

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

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 goal of the strategy is to make sure that $erl(v)$ is low when v is killed. We make an invariant that says that $erl(v) < t$ for all $v \in V_g$ at the end of the player's turn. This means that no node v can be killed while having $erl(v) \geq t$.

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.

When doing a reset, we place 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.

$$V_i = \{v \in V_g \mid rl(v) \geq i\}$$
$$U_i = \left\{u \in U'_g \mid \text{designatedEdge}_{\text{target}}(u) \in V_i\right\}$$

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.

9.2 Analysis of The Game

In this section we will argue for the points gained by the adversary $P(N, M)$, and the cost of implementing the player's strategy $C(N, M)$.

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 $\text{degree}_{\text{designated}}(v)$ 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

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

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 the 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 9.1 *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 $U_k(v) = \{u \in U_k \mid designatedEdge_{target}(u) = v\}$. The size of $U_k(v)$ is $\text{degree}_{\text{designated}}(v)$, 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}.

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

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 9.2 *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 9.1 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 9.2 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

$$\begin{aligned} P(N,M) &\leq N \cdot N^{\varepsilon}\sqrt{M/N} + 2^{\frac{1}{\varepsilon}-1}\frac{N^{\varepsilon}}{\sqrt{M/N}}M + \frac{8\#edgeKills}{N^{2\varepsilon}} \\ &= N^{0.5+\varepsilon}M^{0.5} + 2^{\frac{1}{\varepsilon}-1}N^{0.5+\varepsilon}M^{0.5} + \frac{8\#edgeKills}{N^{2\varepsilon}} \\ &= O\left(N^{0.5+\varepsilon}M^{0.5} + \frac{\#edgeKills}{N^{\varepsilon}}\right) \end{aligned}$$

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 corresponding 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 data structure 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 data structure 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 $r_0 \operatorname{degree}_{\text{initial}}(v)$ edge kills or designations must have occured before the *erl* of a node changes. The cost of updating the data structure is $degree(v)$, so the cost for all updates to v is at most

$$\begin{aligned} cost(v) &\leq degree(v)\frac{\#edgeKills(v) + \#edgeDesignations(v)}{r_0 \operatorname{degree}_{\text{initial}}(v)} \\ &\leq \frac{\#edgeKills(v) + \#edgeDesignations(v)}{r_0} \\ \sum_{v \in V_g} cost(v) &\leq \sum_{v \in V_g} \frac{\#edgeKills(v) + \#edgeDesignations(v)}{r_0} \\ &\leq \frac{\#edgeKills + P(N,M) + N}{r_0} \end{aligned}$$

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

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

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

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

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

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

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

The final total cost for maintaining the game becomes

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

9.3 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 tree push 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]

```
1: function MaxFlow(V, E, s, t)
2:     Initialize()
3:     edges ← {(u, v) ∈ E | u ≠ s ∧ v ≠ s ∧ u < v}
4:     for all (u, v) ∈ edges ordered by ucap(u, v) decreasing do
5:         Add (u, v) and (v, u) to F
6:         if d(u) > d(v) then
7:             Saturate(u, v)
8:         else if d(u) < d(v) then
9:             Saturate(v, u)
10:        end if
11:        while ∃v ∈ V \ {s, t} : e*(v) > 0 do
12:            if CurrentEdge(v) ≠ nil then
13:                TreePush(v)
14:            else
15:                Relabel(v)
16:            end if
17:        end while
18:    end for
19:    return e(t)
20: end function
21: procedure Initialize
22:    Create dynamic forest F
23:    d(s) ← n
24:    for all (s, v) ∈ E do
25:        Add (s, v) and (v, s) to F
26:        Saturate(s, v)
27:    end for
28: end procedure
29: procedure TreePush(u)
30:    (u, v) ← CurrentEdge(u)
31:    link(u, v) if not linked
32:    if ∃ edge (x, y) on path to root from u in F : u(x, y) ≤ e*(u) then
33:        Saturate(x, y)
34:        cut(x, y)
35:    end if
36:    send e*(u) units of flow along path from u to its root in F
37: end procedure
38: procedure Relabel(v)
39:    for all u ∈ V : CurrentEdge(u) = (u, v) do
40:        cut(u, v)
41:    end for
42:    d(v) ← d(v) + 1
43: 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.

9.4 Correctness

Lemma 9.3 *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 9.4 *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 9.3, 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 9.5 *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 9.4 we can get that $d(v_1) \leq d(v_2) + 1 \leq d(v_3) + 2 \leq \cdots \leq d(v_k) + k - 1$. 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.

9.5 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 $O(m \log m) = O(m \log n)$ time, since $m \leq n^2$.

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 9.6 *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 9.7 *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 9.2 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 $\log n = O(n^\varepsilon)$ 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

$$C(N,M) = O\left(M + \frac{\#edgeKills + P(N,M) + N}{r_0}\right)$$
$$C(n^2,nm) = O\left(nm + \frac{\#\text{saturating pushes} + P(n^2,nm)}{\frac{n^\varepsilon}{\sqrt{m/n}}} + \frac{n^2}{\frac{n^\varepsilon}{\sqrt{m/n}}}\right)$$
$$C(n^2,nm) = O\left(nm + \frac{n^{1.5+\varepsilon}m^{0.5}}{\frac{n^{0.5+\varepsilon}}{m^{0.5}}} + \frac{n^2}{\frac{n^{0.5+\varepsilon}}{m^{0.5}}}\right)$$
$$C(n^2,nm) = O\left(nm + nm + n^{1.5-\varepsilon}m^{0.5}\right)$$
$$C(n^2,nm) = O\left(nm + n^{1.5-\varepsilon}m^{0.5}\right)$$

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 $\Omega(M + Nt) = \Omega\left(nm + \frac{n^2}{\varepsilon}\right)$ space, which makes it difficult to run it on medium to large graphs where m is close to n^2.

9.6 Contributions

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 $O(\text{degree}_{\text{initial}}(v))$, 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 designated edge kills when doing a node kill. With the change to the game, it is now possible to kill an edge multiple times. When doing a node kill on a node v, the adversary can get at most $r_{t-1} degree(v)$ points, but he can relable a node $O(n)$ times, which yields $r_{t-1} degree(v) n$ points. If we sum this over all v, we get $r_{t-1} nm$, which is the same bound as above.

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

$$\begin{aligned} P(n, m) &\leq n^2 \cdot n^\varepsilon \sqrt{m/n} + 2^{\frac{1}{\varepsilon}-1} \frac{n^\varepsilon}{\sqrt{m/n}} nm + \frac{8\#edgeKills}{n^\varepsilon} \\ &= n^{1.5+\varepsilon} m^{0.5} + 2^{\frac{1}{\varepsilon}-1} n^{1.5+\varepsilon} m^{0.5} + \frac{8\#edgeKills}{n^\varepsilon} \\ &= O\left(n^{1.5+\varepsilon} m^{0.5} + \frac{\#edgeKills}{n^\varepsilon} \right) \end{aligned}$$

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 C 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 to make a version of the algorithm that does not use dynamic trees. 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 $O(\log n)$. This n propagates through the runtime analysis, to yield a running time of $O(nm+n^{2.5+\varepsilon}m^{0.5})$.

This means we have three versions of the algorithm. One according to the specifications, one with optmized memory, and one with both optmized memory and without dynamic trees.

10 Goldberg Rao 1998

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 7 Blocking flow Push, Pull and Block procedures

```
1: procedure PUSH( Edge (v, w) )
2:     Send δ = 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

4: procedure PULL(Edge(w, v))
5:     Subtract δ = min(e(v), f(w, v)) units of flow on (w, v) by updating
   the edges, (w, v) and (v, w), and the excess, e(v) and e(w).
6: end procedure

7: procedure BLOCK(v)
8:     block ← true
9:     for all (v, w) ∈ E do
10:        if r(v, w) > 0 and d(v) = d(w) + EDGELENGTH((v, w)) and
   w.blocked = false then                          ▷ Is able to push?
11:            block ← false
12:        end if
13:    end for
14:    if block then
15:        v.blocked ← true
16:    end if
17: end procedure
```

Algorithm 8 Blocking flow Initialization and Main Loop

```
1: function BLOCKINGFLOW(V, E, s, t)
2:     for all (v, w) ∈ E do
3:         f(v, w) ← 0
4:     end for
5:     for all (s, v) ∈ E do
6:         Send max capacity flow through (s, v) update (v, s) accordingly
7:         Update excess of v
8:     end for
9:     while Queue Q ≠ ∅ do                            ▷ Main-loop
10:        DISCHARGE( )
11:    end while
12:    return e(t)
13: end function
```

Algorithm 9 Blocking Flow Tree-PushPullRelabel and Discharge procedures

```
1:  procedure DISCHARGE
2:      Node v ← first element of Q, removed from the queue.
3:      repeat
4:          PUSHRELABEL(v)
5:          if w becomes active then
6:              Add w to the back of Q
7:          end if
8:      until e(v) = 0 or d(v) increases
9:      if v is still active then
10:         add it to the back of Q
11:     end if
12: end procedure

Require: v is active
13: procedure TREE-PUSHPULLRELABEL(v)
14:     Edge (v, w) ← current edge of v
15:     if r(v, w) > 0 and d(v) = d(w) + EDGELENGTH((v, w)) and
    w.blocked = false and v.blocked = false then
16:         PUSH((v, w))
17:         SEND(w)
18:         if r(v, w) > 0 and GETSIZE(v) + GETSIZE(w) ≤ k then
19:             LINK(v, w)
20:             SETCOST(v, r(v, w))
21:         end if
22:     else if v.blocked = true and f(w, v) > 0 then
23:         PULL((w, v))
24:         SEND(w)
25:         if f(w, v) > 0 and GETSIZE(v) + GETSIZE(w) ≤ k then
26:             LINK(v, w)
27:             SETCOST(v, f(w, v))
28:         end if
29:     else
30:         if e is not the last edge of the edgelist of v then
31:             set the current edge of v to be the next edge
32:         else
33:             Set the current edge of v to be the first edge in the edgelist
34:             Cut all the childs of v the has been linked in line 19
35:             BLOCK(v)
36:         end if
37:     end if
38: end procedure
```

Algorithm 10 Routing Flow - algorithm

Require: A list L of Strongly Connected Components $\in G(V, E)$ and a way SameSCC(v, w) of knowing if 2 nodes are in the same component.

```
1:  procedure RouteFlow
2:      for all v ∈ V do
3:          Calculate supply and demand for v
4:      end for
5:      for all Strongly-Connected-Components ∈ L do
6:
7:          choose a node v to be the root
8:          BuildInTree(v)
9:          BuildOutTree(v)
10:         RouteFlow(v)
11:     end for
12: end procedure

13: procedure BuildInTree(v)
14:     Q ← Queue
15:     Add v to the rear of Q
16:     while Q ≠ ∅ do
17:         Node w ← first element of Q, removed from the queue.
18:         for all {(w, u)|SameSCC(w,u) and IsZeroEdge((w, u))} do
19:             if u.InTreeParent = nil then
20:                 u.InTreeParent ← w
21:                 Add u as a child of w in the InTree
22:             end if
23:         end for
24:     end while
25: end procedure
```

Algorithm 11 Routing Flow - algorithm(cont.)

```
1: procedure RouteFlow(v)
2:     CalculateDescendantDemandsRecursively(v)
3:     MoveSupplyForwardRecursively(v)
4:     MoveDemandBackwardRecursively(v)
5: end procedure

6: function CalculateDescendantDemandsRecursively(v)
7:     v.dd ← v.demand
8:     for all w ∈ OutTreeChildren(v) do
9:         v.dd ← v.dd+CalculateDescendantDemandsRecursively(w)
10:    end for
11: end function

12: procedure MoveSupplyForwardRecursively(v)
13:    for all w ∈ InTreeChildren(v) do
14:        MoveSupplyForwardRecursively(w)
15:    end for
16:    supplyToMove ← min (v.supply, δ − v.dd)
17:    v.supply ← v.supply − supplyToMove
18:    (v, w) ← v.InTreeParentEdge
19:    f(v, w) ← f(v, w) + supplyToMove
20:    w.supply ← w.supply + supplyToMove
21: end procedure

22: procedure MoveDemandBackwardRecursively(v)
23:    for all w ∈ OutTreeChildren(v) do
24:        MoveDemandBackwardRecursively(w)
25:    end for
26:    demandToMove ← v.demand − v.supply
27:    (w, v) ← v.OutTreeParentEdge
28:    f(w, v) ← f(w, v) + demandToMove
29:    w.demand ← w.demand + demandToMove
30: end procedure
```

11 Global Relabeling Heuristic

All of our implementations of the Goldberg Tarjan and King Rao algorithms had a problem. After the minimum cut of the graph has been saturated, these algorithms have to push the remaining excess back to the source. This requires that all the nodes behind the minimum cut are relabeled above n. We found that there often was very far between the label of the nodes and n, at the time the minimum cut was saturated. This made the algorithms spend a lot of time taking small steps relabeling towards n, while pushing flow back and forth.

To alliviate this problem, we implemented a heuristic for all versions of these algorithms. This heuristic updates the label of all the nodes in the graph, based on their distance to the target and source nodes. This is done by first doing a breath first search from the target, visiting nodes that can push more flow towards the target. The label of these nodes are updated to their distance to the target. After this, we run a similar breath first search from the source, but only look at nodes that was not visited in the previous breath first search, and that can send flow towards source. These nodes v get the label $n + \mathit{distance}(v, s)$.

This heuristic is run during initialization, and every time a node has to be relabeled more than one label up. The reason we run it at the start is that a node v will have to be relabled to $\mathit{distance}(v, t)$ anyway before its excess can be pushed to the target node. The other situation where we run the global relabel is to avoid the situation where the algorithm pushes flow back and forth while relabeling. To be more specific, what we want to avoid is flow being pushed around in a cycle $(v_1, v_2, \cdots, v_k, v_1)$. To push flow from v_i to v_{i+1}, $d(v_i) > d(v_{i+1})$. This means that if no label has been relabled twice by the time v_k gets the excess, then $d(v_1) > d(v_k)$, so v_k will have to be relabeled at least twice to send the flow to v_1.

This functionallity runs in $O(n^2m)$ time. It takes $O(m)$ time to do the breadth first searches, and we have n nodes that can be relabeled twice in a row n times.

For all algorithms except the King Rao algorithm without optimized memory, relabeling multiple labels up could be done in the same time as relabeling one label up. The problem with the King Rao algorithm that uses $O(nm)$ memory is that when relabeling from k to l, the game has to be updated for all labels between k and l. It has to perform edge kills on all edges incident to the nodes that corrospond to labels between k and l. This is not an issue for the version of the algorithm that uses $O(m)$ memory, because it only has two nodes in the game for each node in the graph. Here edges for those two nodes just have to be added or removed according to the new label.

12 Tests

In this section we will describe what tests we have run on the algorithms, and what we expect to see from them. Section 12.1 contains a brief explanation of the verification we do to ensure that the max flow value returned by our algorithm implementations is the correct one. Section 12.2 contain a list of the different types of graphs we will be running on.

12.1 Algorithm Correctness

To verify that our algorithms work, we use the method of certifying algorithms [MMNS10]. We implement our algorithms to return both the value of the max flow, and the residual network after all flow has been sent from s to t. We implemented a verifier that runs after the algorithms. It uses the residual network in combination with the original graph to calculate the flow on each edge in the graph. It then verifies that the value of the max flow returned by the algorithm is the same as both the flow going out of s and the flow going into t. Additionally, it verifies 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 what is 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 have proof that the max flow returned by the algorithms is the correct ones.

12.2 Graph generators

To generate graphs to test on, we use a range of graph generation algorithms from DIMACS [JM93]. For examples of the output of each algorithm, please refer to Appendix B.

12.3 AC

The AC graph generator produces an acyclic graph with nodes $v_0, \cdots, v_{n-1}$. A node v_i has edges to all nodes $v_{i+1}, \cdots, v_{n-1}$ with capacities randomly generated in the range $[1, 10000]$.

The purpose of this generator is to get examples of graphs where $m = \Theta(n^2)$.

When we ran the push-relable algorithms on this type of graph we got big jumps in the time spent. The reason is the random capacities. If everything sent out of source can be sent to the target, the algorithms won't have to relable all the nodes very far. On the other hand, if not everything can be sent to the target, it is likely that all nodes will have to be relabled to $n+1$, since the graph is fully connected.

To get a better picture, we split the AC graphs up into two groups. An easy group where everything can be sent to the target, and a hard group where some of the flow will have to be pushed back. Further more, we connected all nodes except s and t, so the graph is cyclic. Capacities on edges from s and to t are random, but all other edges have capacity $10000n$.

We will abbriviate theese types of graphs with CRE and CRH for Connected graphs, Randomized capacities, Easy/Hard for push relabel.

12.4 Connected Deterministic

We observed very good results for the implementation of the Dinic algorithm in the AC graphs. This is because there are a small number of augmenting paths, and all of those paths have length 2 or 3.

To alliviate this, we made a new type of fully connected graph. This graph is designed to contain as many augmenting paths as possible. This should make the graph very difficult for the algorithm by Edmonds and Karp, and the algorithm by Dinic. The layer graphs in Dinic will be very big.

The graph generator is deterministic for a specific size, and is constructed so that it considers layer graphs of increasing length. An example of how the graph is constructed can be seen in Figure 6.

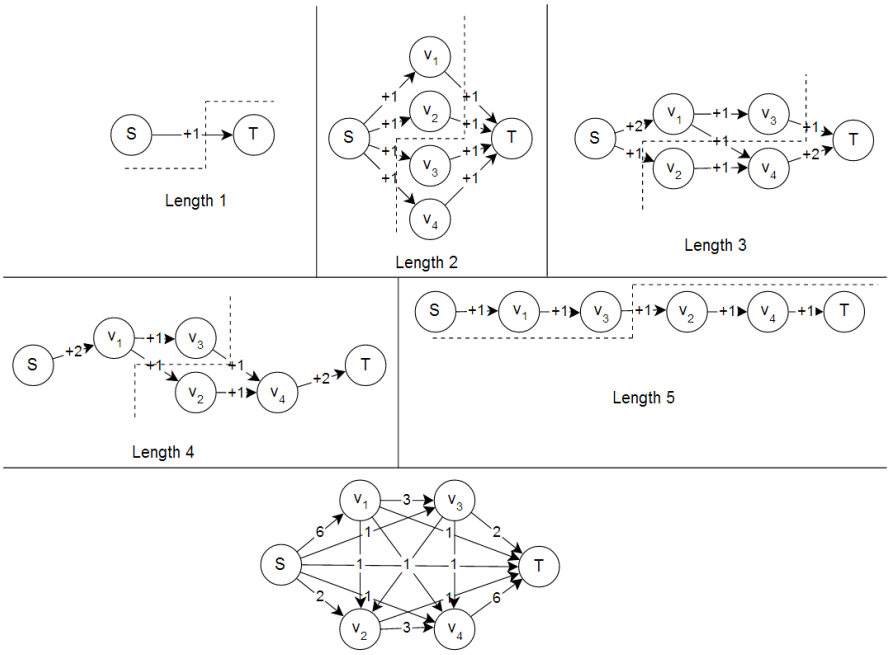

Figure 6: How the custom connected graph is constructed for $n = 6$.

The nodes in each layer graph is partitioned into a top half and a bottom half set. The goal is to have the min cut of each layer graph between the top and bottom half sets, so that the bottom half can be offset to the next

column in the following layer graph. The edges going internally in the top or bottom sets just has enough capacity to route the flow that is sent over the cut.

This allows for nodes to be in different sets in different layer graphs. For example, picture the construction of $n = 18$. In that construction, v_{11} would go from being in the bottom set in the layer of length 2 to the top set in layer 3 and 4, to the bottom set in layers 5 to 8, and then in the top set in layers 9 to 17.

Note that for $n \geq 18$, the situation arises where flow will have to be sent from v_i to v_j, and in a later layer graph, from v_j to v_i. In that situation, we do not increase the capacity of (v_j, v_i), since it will already have the required residual capacity. For instance, for $n = 18$, in layer 4, (v_{11}, v_{14}) is part of the cut. In layer 8, (v_{14}, v_{11}) is in a cut, and in layer 12, (v_{11}, v_{14}) is back in the cut.

The way the bottom half set is offset ensures that once an edge (v_i, v_j) has been part of a cut, v_i and v_j can not be next to each other in one set, because v_j will be moved to a column at least two columns in front of v_i. So, the only way (v_i, v_j) can be used again is if (v_i, v_j) is part of a future cut. In order for that to happen, v_i first has to be moved in front of v_j though, and in order for that to happen, v_j must be in the top set, and v_i in the bottom set. At that point, (v_j, v_i) would be part of of the cut, meaning that the residual capacity of (v_i, v_j) would return to one, and we can use it again in a cut without increasing its capacity. This means that once an edge has been part of a cut, its capacity will never increase, so capacity increases won't interfere with the capacity of a cut in a previous layer graphs.

Once two nodes have been placed in the same row, they will be in the same row for the rest of the layers. This means that the edge between them will never be part of the min cut in the future. For this reason, when flow is routed inside the top or bottom set, it is just sent to the next node in the same row. This ensures that routing flow inside the sets does not interfere with future cuts.

There is one problem in the construction of this graph that we have not been able to elliminate. It is possible for the max flow algorithm to route flow inside a set across rows, because such an edge could be part of a future cut. Doing this will create an augmenting path in the next layer graph where the edge can be taken the opposite way, so it does not change the number of augmening paths. It does mean that the execution was not always as expected though.

This construction results in a graph with $m = \frac{n(n-1)}{2}$ non-zero edges. If you also count zero capacity edges (v_j, v_i) that are added as a result of (v_i, v_j) being added, this type of graph contains all edges possible.

The abbriviation for this type of graphs will be CD for Connected graphs, fully Deterministic construction.

12.5 AK

The AK graph generator takes a paramter k, and produces deterministic graphs where $n = 4k + 6$ and $m = 6k + 7$. These graphs are designed to be very hard instances of the max flow problem.

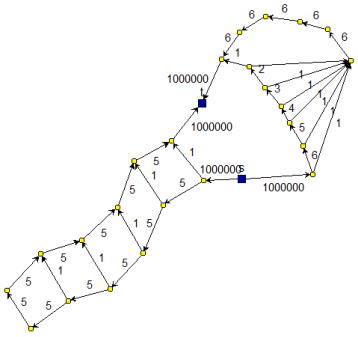

Figure 7: An example of the output of the AK generator, where $n = 26$. See Appendix B for a bigger version.

As can be seen in Figure 9, this type of graphs contain two hard patterns. The left pattern require that the flow is pushed far out in the graph, and then back. For Push Relabel algorithms, this mens that flow will be pushed back and forth in the graph while relabeling. With the global relabel heuristic, a lot of global relabels will occur due to the left pattern.

The right pattern contains very long paths of increasing length. This is peticularly hard for the Dinic algorithm, since will get layer graphs with only one or two augmenting paths.

12.6 GenRmf

The GenRmf generator produces a special kind of graphs developed by Goldfarb and Grigoriads [GG87]. It takes parameters a, b, c_{min} and c_{max}. The graph produced will consist of b layers of nodes, with $a \times a$ nodes laid out in a grid in each layer. Each node in a layer has an edge connecting it to the two to four nodes adjacent 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 between layers are randomly generated in the range $[c_{min}, c_{max}]$. Capacities inside layers are big enough so that all flow can be pushed around inside the layer.

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

As a consiquence of the construction, the min-cut will always be between two layers.

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

12.7 Washington

The Washington library 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 three random 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.

We will use this to create two versions of the Wash graphs. One with a constant 64 rows and a variable number of columns, and another with a constant 64 columns and a variable number of rows.

12.8 Test Environment

All tests were run on a Windows 7 64 bit PC with the following hardware.

Processor	Intel Core I7 950
Speed	3.07 GHz
Cores	4 (8 with hyperthreading)
L1 Cache (Instruction)	32 KB 4-way associative
L1 Cache (Data)	32 KB 4-way associative
L2 Cache	256 KB 8-way associative
L3 Cache	8 MB 16-way associative (shared)
Cache Line	64 bytes
Main Memmory	Corsair CMZ12GX3M3A1600C9
Capacity	12 GB (3x4GB)
Speed	DDR3 1600 (PC3 12800)
Latency	CAS9

13 Results

In total, we have tested 13 algorithm implementations on 8 different types of graphs, yielding 104 different test runs. Each test was given an hour to solve max flow problems of exponentially increasing size.

We will be using the following abbreviations for our implementations of the algorithms:

Abbreviation	Presented	Special Features
EK	Section 6	
Dinic	Section 7	
GT	Section 8	
GT D	Section 8	Dynamic Trees
GT GR	Section 8	Global Renamig
GT D GR	Section 8	Dynamic Trees, Global Renamig
KR D	Section 9	Dynamic Trees
KR LM	Section 9	Low Memory
KR LM D	Section 9	Low Memory, Dynamic Trees
KR D GR	Section 9	Dynamic Trees, Global Renamig
KR LM GR	Section 9	Low Memory, Global Renamig
KR LM D GR	Section 9	Low Memory, Dynamic Trees, Global Renamig
GR	Section 10	

In Appendix C, you will find charts of the results for each type of graphs.

In the following Sections we will discuss the performance of each algorithm.

13.1 AC

Figure 12 and Figure 14 shows the results from the tests on fully connected graphs. Figure 12 is the data for the graphs where all excess can be pushed to the target, and Figure 14 is results from the graphs where some of the excess will have to be pushed back to source.

As we expected, the run time difference for push-relabel algorithms is huge, they are hundreds of times faster when they don't need to push flow back to source. The only exception is the unmodified King Rao algorithm, but we will exclude that one due to its massive overhead managing memory. The non push-relable algorithms such as Edmonds Karp and Dinic show no differences between the two types of input. The only thing that matters to these algorithms is the total number of augmenting paths, which does not change significantly. The Goldberg Rao algorithm also does not change much. The majority of its time is spent compressing the graph, which dominates the time spent running its push-relable algorithm. Dinic is doing very well on this type of graph, because it only needs to construct two level graphs, where it is able to cut away the majority of the edges.

13.2 Edmonds Karp

The Edmonds Karp algorithm gets no first places and no last places. Dinic's algorithm is faster than Edmonds Karp, in all examples except for the AK graphs. This makes sence with respect to the worstcase time of $O(nm^2)$ for Edmonds Karp, and $O(n^2m)$ for Dinic. The running time of the Edmonds Karp algorithm depends on the number of augmenting paths, and the number of edges in the graph. The more augmenting paths there are, the more breadth first searches will have to be done, and the time for each breath first search depends on the number of edges in the graph. This is also why its worst case time is $O(nm^2)$. We have however not found that the algorithm runs in $O(nm^2)$ time in practice, because nm is an over aproximation on the number of augmenting paths. Figure 21 in Appendix C shows the time for all measurements of Edmonds Karp, where the running time is divided by the product of the number of edges and augmenting paths in the graph. This chart is very flat until $n = 2^{15}$, which means that for $n < 2^{15}$, $m \cdot$ #AugmentingPaths is a good aproximation of the running time. At $n = 2^{15}$, the graph itself takes up 4-5 MB. The algorithms adds additional variables to the nodes and edges in the graph, and it allocates memory for a queue of size n. The reason the algorithm starts to take longer at $n \geq 2^{15}$ is therefore that it starts to require more memory than the 8 MB that can be stored in the L3 cache of the CPU. This means it has to go to main memory, and that is substantially slower than the cache.

13.3 Dinic

Dinic's algorithm generally performs very well. Before the Global Renamig heuristic, it is the best algorithm on GenRmf graphs, Wash and AC graphs where flow has to be pushed back. The only places that it is beaten is the AK graphs, the AC graphs where flow does not have to be pused back and the CC graphs. It is beaten on the AC graph because it is the best case scenario for the push relable algorithms, and it is beaten on the AK graphs because the AK graph contains a pattern which is the worst case scenario of Dinic. The effect of this pattern is that Dinic only finds one augmenting path in each layer graph, which means that it ends up spending too long time constructing the layer graphs. It was expected to have bad performance on the CC graphs, since these graphs are designed to be hard for Dinic.

For most of our graphs types, our implementation seems to be running closer to nm time than the expected $O(n^2m)$ time, as can be seen from Figure 22 in Appendix C. The only exception is the results from the CC graphs, which are not displayed on the figure, but has a higher time bound. For graphs other than CC, we do have $O(n)$ layer graphs, but the time it takes to do the depth first search in the layer graph is closer to $O(m)$ than $O(nm)$. This is because each individual path in the layer graphs tend to be

used a low number of times.

This is the reason we decided to add the CC graph type. We wanted to get some tests where Dinic performs closer to the worst case bound. On the CC graphs, the algorithm performs worse than nm time, but it is still $O(n^2m)$ time. We have not fond an expression that makes sense for the exact observed running time.

13.4 Goldberg Tarjan

The Goldberg Tarjan algorithms also perform quite well. They are the fastest algorithms in the AC graphs where all flow can be pushed to the target, the custom connected graphs with many augmenting paths, and the AK graphs.

The AC graphs where all flow can be pushed to the target is basically the best case graph for Goldberg Tarjan. The maximum label of nodes in this graph is 2, so the algorithm can terminate very early. The reason the maximum label is two is that a node is only relabeled from 1 to 2 when no more flow can be sent to the target node. The graph is fully connected with very high capacities, so once a node has been relabeled to 2, it will be able to send all its excess to a node with label 1.

The AC graphs where some flow has to be pushed back to the source on the other hand is the worst case for this algorithm. All nodes will have to be relabled to n before the algorithm can terminate. In between the relabels, the algorithm will of course push the excess back and forth too.

The Goldberg Tarjan algorithm has an advantage in the CC graphs, because no flow will ever have to be pushed back to source. The way the CC graphs are constructed, no edge has more capacity than what is needed to get the flow to the target node. The maximum label depends on the order the algorithm processes the nodes. For each CC graph, there is one order which results in no nodes getting a label higher than 2. There is also an order that results in nodes having labels n, $n-1$, $n-2$, etc.

The algorithm will have to push flow back to the source in the AK graph, but this is only for the two nodes right next to the source. All other nodes is able to push all their excess to the target node, so the algorithm won't have to spend time relabeling a lot of nodes to get flow back to the source. There are long paths in the graph though, so labels get high even though the flow is pushed to the target.

It seems aparent that the running time of the N3 algorithm is directly proportional to the number of relabels required. When we try to compare it to $O(n^3)$ or $O(nm)$, we get a lot of turbulence in the results. As can be seen from Figure 23, the turbulence comes from the GenRmf and Washington graphs. Figure 24 shows the same data without the GenRmf and Washington algorithms. The randomized capacities in the GenRmf and Washington algorithms cause the distance from s to the minimum cut to vary a lot. In these graphs, all nodes on the same side of the minimum cut as s will have

to be relabeled above n so that the excess can be sent back to s. We believe that this is the source of the turbulence, but we have not had time to investigate the size of the min cut in the graphs. Based on the data in Figure 24, it seems that the algorithm is a little faster than nm for the AK graphs. CC and CWC graphs seems to be leveling out towards the end of their chart, but we do not have enough points to be sure. It would seem that nm is a better estimate for the running time of N3 than the worst case bound of $O(n^3)$.

The dynamic version has the same turbulence as the N3 verion. If we exclude the GenRmf and Wash graphs, the running time is faster than nm, but we have not been able to find a good estimate for the actual running time. Figure 25 and Figure 26 shows the running time of the dynamic version with and without GenRmf and Washington.

For most graphs we tested pm, it is not worth it to run the algorithm with dynamic trees. The only type of graphs where the dynamic tree version was faster than the N3 version was sufficiently large instances of the AK graphs, as can be seen in Figure 15. In all other graphs, there was either very little difference, or a big hit when using dynamic trees.

Where the algorithm should benefit from dynamic trees is situations where a long path is being used often. Such a pattern is found in the AK graphs, and this is why the dynamic tree version is faster.

13.5 King Rao

The unmodified version of the King Rao algorithm is the worst algorithm in every test we have done. The problem is that the game requires $2n^2$ nodes and nm edges. This results in the issue that every time a node is relabeled, a new node in the game will have to be loaded from main memory, or possobly even the hard disk. We do not have any examples where the algorithm went to the disk though, because it failed with a memory allocation error for large inputs. The performance is only worse when going to the disk, so running for larger inputs will not yield any new information. For this reason, we did not want to investigate this error further.

The modified versions are very similar to the Goldberg Tarjan algorithm, except that they are about ten times slower. The only difference between the Goldberg Tarjan algorithms and the modified King Rao algorithms is the choice of current edges. In our tests, the overhead that results from the more complicated way of choosing current edges is far greater than any time saved.

Figure 27 show the results of the modified King Rao algorithm without dymaic trees in relation to nm. Just like with the Goldberg Tarjan algorithm, there is a lot of turbulence with the GenRmf and Washington algorithms, for the same reasons as before. The algorithm seems to follow a runtime of nm for the AK and CC graphs. It is significantly faster on the CBC graphs, but these are also the best case scenario for a push relabel algorithm.

13.6 Goldberg Rao

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 J. B. Orlin. A fast and simple algorithm for the maximum flow problem. *Operations Research*, vol 37(5):pages 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] J. Cheriyan, T. Hagerup, and K. Mehlhorn. Can a maximum flow be computed on $o(nm)$ time? In Mike Paterson, editor, *ICALP*, volume 443 of *Lecture Notes in Computer Science*, pages 235–248. Springer, 1990.

[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] J. Edmonds and R. 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] H. 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.

[GG87] D. Goldfarb and M. D. Grigoriadis. *A Computational Comparison of the Dinic and Network Simplex Methods for Maximum Flow.* LCSR-TR-94. Department of Computer Science, Rutgers University, 1987.

[GN79] Z. Galil and A. 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] A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *J. ACM*, 45(5):783–797, September 1998.

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

[Hoc98] D. 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.

[JM93] D. S. Johnson and C. C. McGeoch, editors. *Network Flows and Matching: First DIMACS Implementation Challenge*. American Mathematical Society, Boston, MA, USA, 1993.

[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. E. 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.

[MMNS10] R. M. McConnell, K. Mehlhorn, S. Naher, and P. Schweitzer. Certifying algorithms. 2010.

[Orl13] J. B. Orlin. Max flows in $O(nm)$ time, or better. In *Proceedings of the Fortyfifth 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] R. E. Tarjan. A simple version of karzanov's blocking flow algorithm. *Operations Research Letters*, 2(6):265–268, 1984.

A Terminology Tables

Appendix A.A Section 2: General terminology

Name	Symbol	Short Description
Graph	G	
Node set	V	
Edge set	E	
Path	$(v_1, \cdots, v_k)$	List of nodes connected by edges
Residual path		A path where all edges have $r(u,v) > 0$
Augmenting path		A residual path from s to t
Bounding edge		The edges in a residual path that have minimum residual capacity
Capacity	$cap(u,v)$	Maximum flow that can be sent on an edge
Maximum capacity	U	$\max_{(u,v)\in E} cap(u,v)$
Flow	$f(u,v)$	The flow sent on an edge
Residual capacity	$r(u,v)$	The amount of flow that can be sent without exceeding cap
Residual Edge		An edge where the residual capacity replaces the capacity
Edge set	E_f	The set of residual edges of E based on flow f
Residual Graph	G_f	$G_f = (V, E_f)$
An Eligeble edge		Has positive residual capacity
Saturated edge		An edge where $r(u,v) = 0$
Distance	$distance(u,v)$	Number of edges in the shortest residual path connecting u to v
Excess	$e(v)$	The sum of flow entering a node v, minus the sum of flow exiting it.

Appendix A.B Section 3: Paradigms

Name	Symbol	Short Description
Blocking flow		
Layer Graph		
Push, Relabel		
Label	$d(v)$	The label of a node v. Used by Push-Relable algorithms.

Appendix A.C Section 9.1: King Rao 1992 - The Game

Symbol	Short Description
$G_g = (U_g, V_g, E_g)$	A bipartite graph used in the game
$N = \|U_g\| = \|V_g\|$	The number of nodes in the game graph.
$M = \|E_g\|$	The number of edges in the game graph.
$P(N, M)$	A function that specifies a bound on how many points the adversary can obtain.
$C(N, M)$	A function that specifies a bound on the cost of implementing the player's strategy.
r_0	Determines when a node changes ratio level.
$r_i = 2^i r_0$	
t	The highest ratio level allowed is r_t.
l	Nodes with fewer edges than the parameter l can use any edge as the designated edge.
$U'_g = \{u \in U_g \mid \text{degree}(u) > l\}$	The nodes in U_g that has degree greater than l.
$r(v) = \frac{\text{degree}_{\text{designated}}(v)}{\text{degree}_{\text{initial}}(v)}$	The ratio of the node $v \in V_g$.
$rl(v) = \begin{cases} 0 \text{ if } r(v) < r_0 \\ i \text{ if } r_i \leq r(v) < r_{i+1} \\ t \text{ if } r_t \leq r(v) \end{cases}$	The ratio level of the node $v \in V_g$.
$erl(v) \in [rl(v), rl(v) + 1]$	The estimated ratio level of the node $v \in V_g$
$V_i = \{v \in V_g \mid rl(v) \geq i\}$	
U_i	Nodes in U'_g whose designated edge go to a node in V_i.

Appendix A.D Section 9.2: King Rao 1992 - Analysis of The Game

Symbol	Short Description
$U_i(v)$	The nodes in U_i whose designated edge go to v.

Appendix A.E Section 9.3: King Rao 1992 - The Algorithm

Symbol	Short Description
$E^* \subseteq E$	The edges that are added to the graph
$h(v) = \sum_{(v,u) \in E \setminus E^*} cap(v, u)$	The hidden capacity of a node.
$e^*(v) = max\,(0, e(v) - h(v))$	The visible excess of a node. A node is not allowed to push more flow away from it than its visible excess.

B Graph Examples

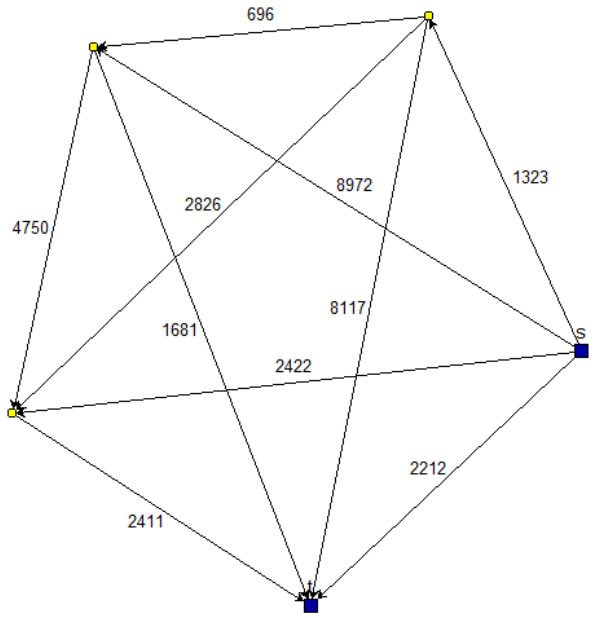

Figure 8: An example of the output of the AC generator, where $n = 5$

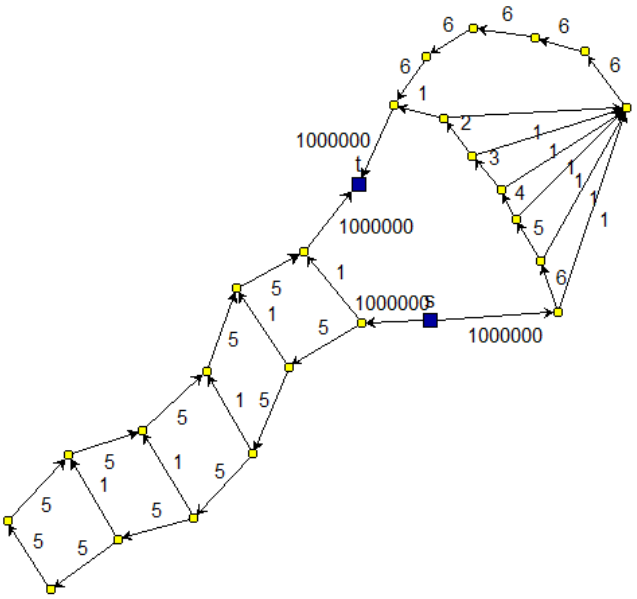

Figure 9: An example of the output of the AK generator, where $n = 26$

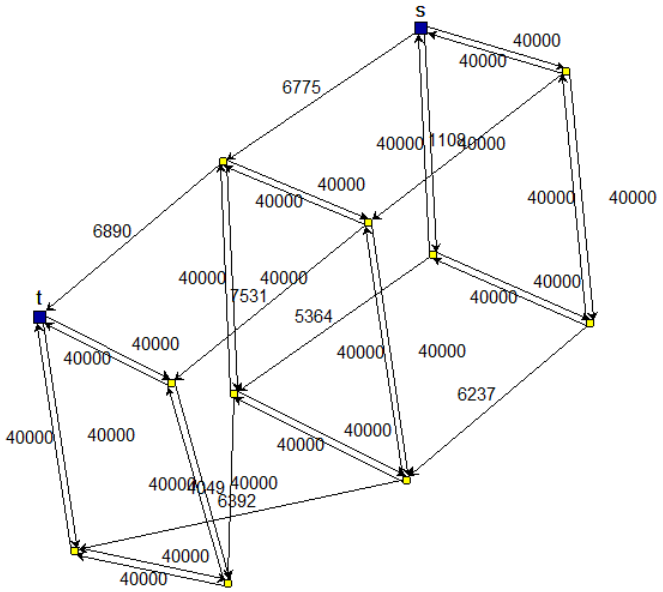

Figure 10: An example of the output of the GENRMF generator, where $a = 2$ and $b = 3$

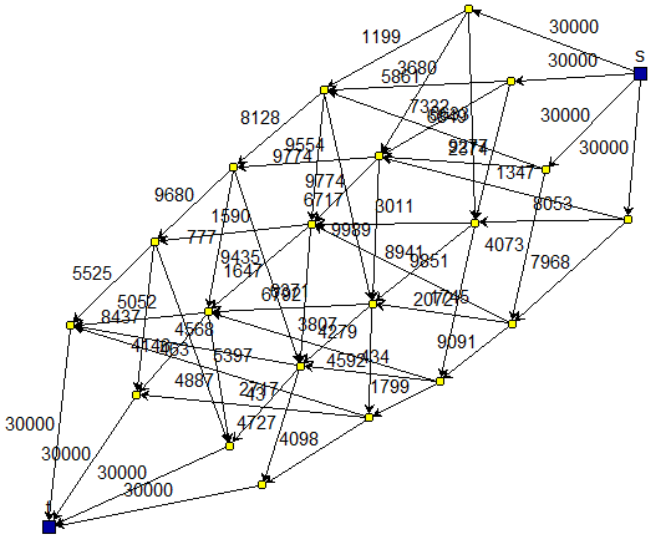

Figure 11: An example of the output of the washington level graph generator, with 5 rows and 4 columns

C Charts

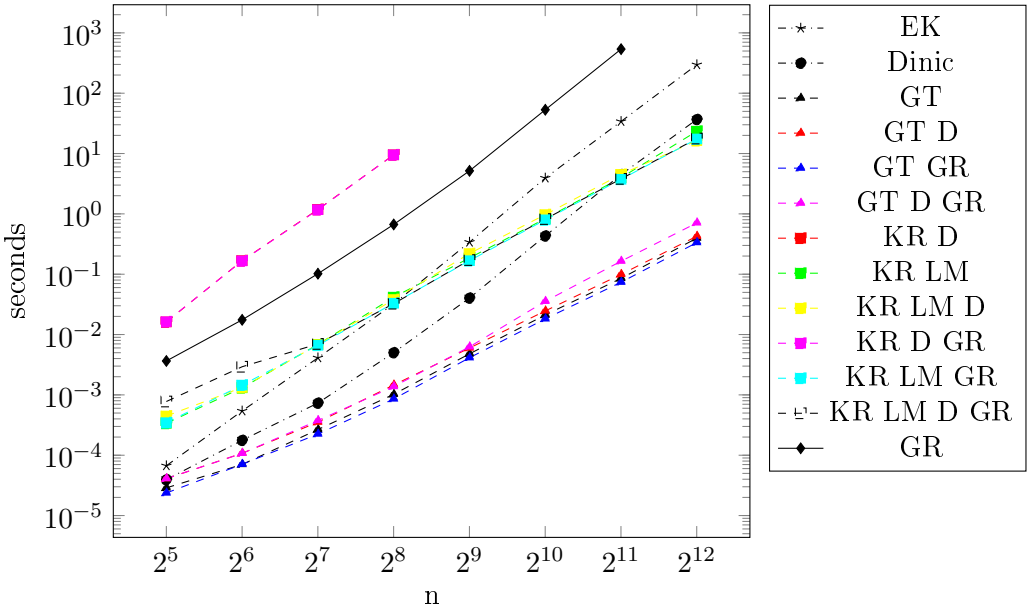

Figure 12: Results from the CRE graphs

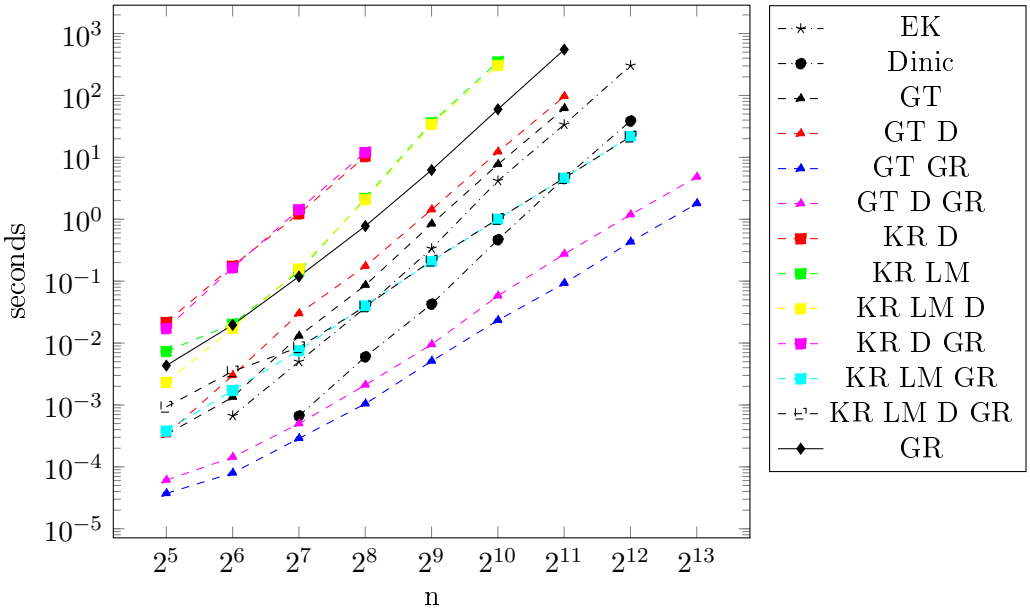

Figure 13: Results from the CRH graphs

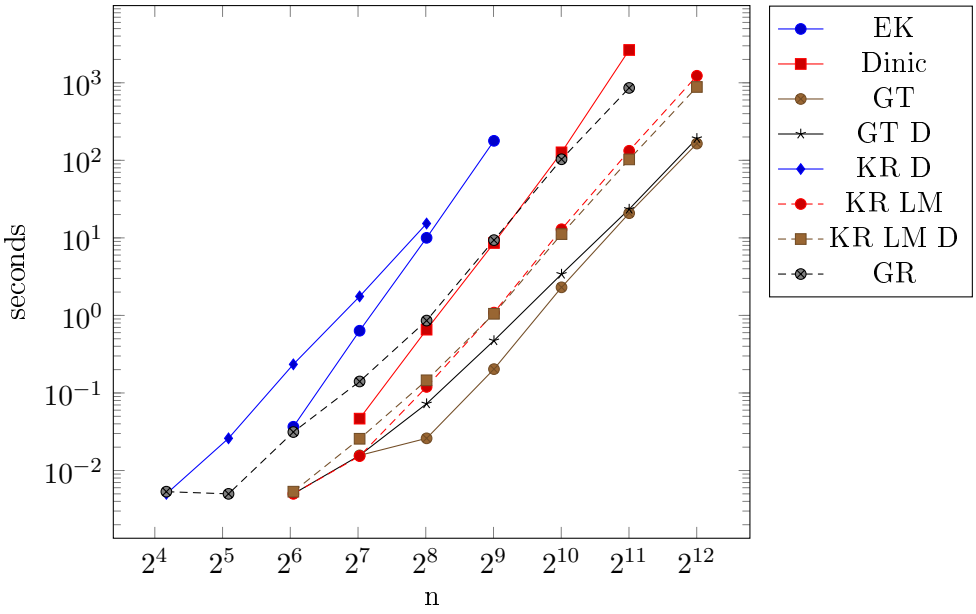

Figure 14: Results from the CD graphs

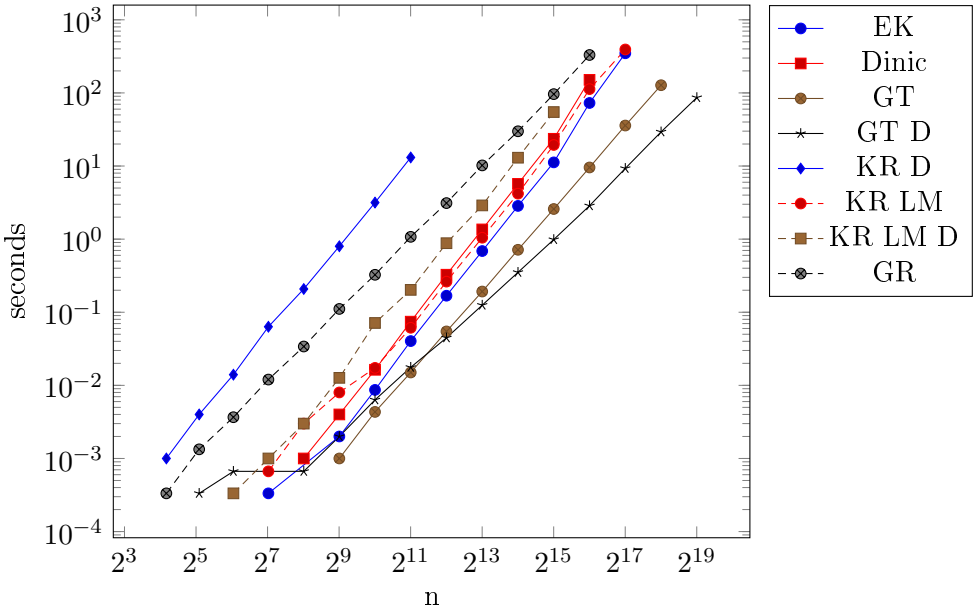

Figure 15: Results from AK graphs

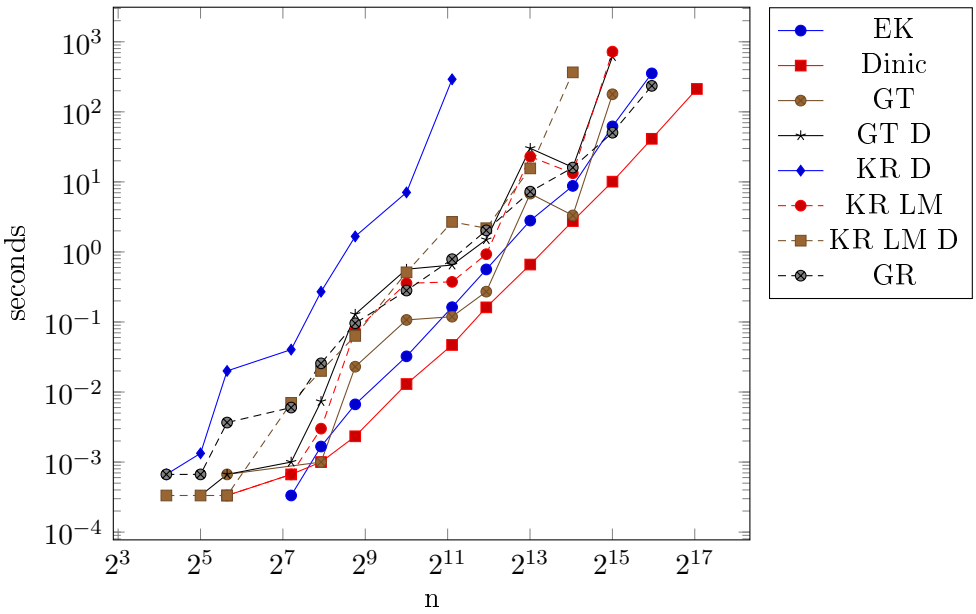

Figure 16: Results from GenRmf graphs with $a = b^2$

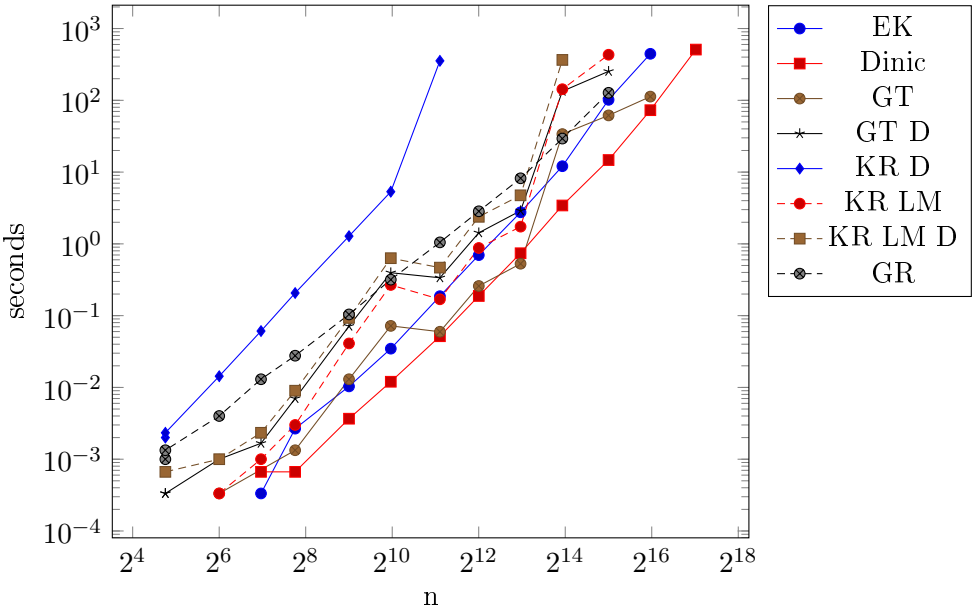

Figure 17: Results from GenRmf graphs with $a = b$

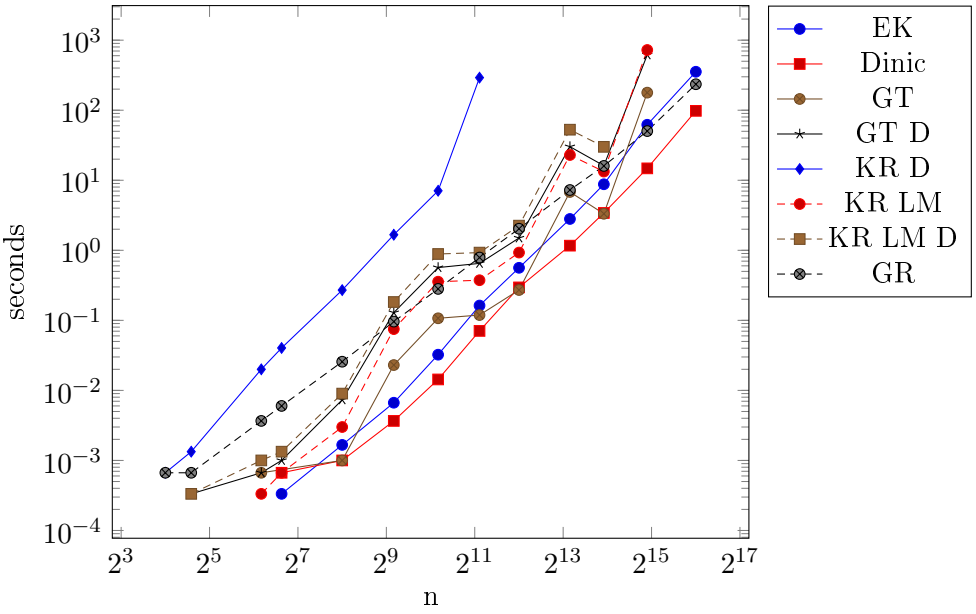

Figure 18: Results from GenRmf graphs with $b = a^2$

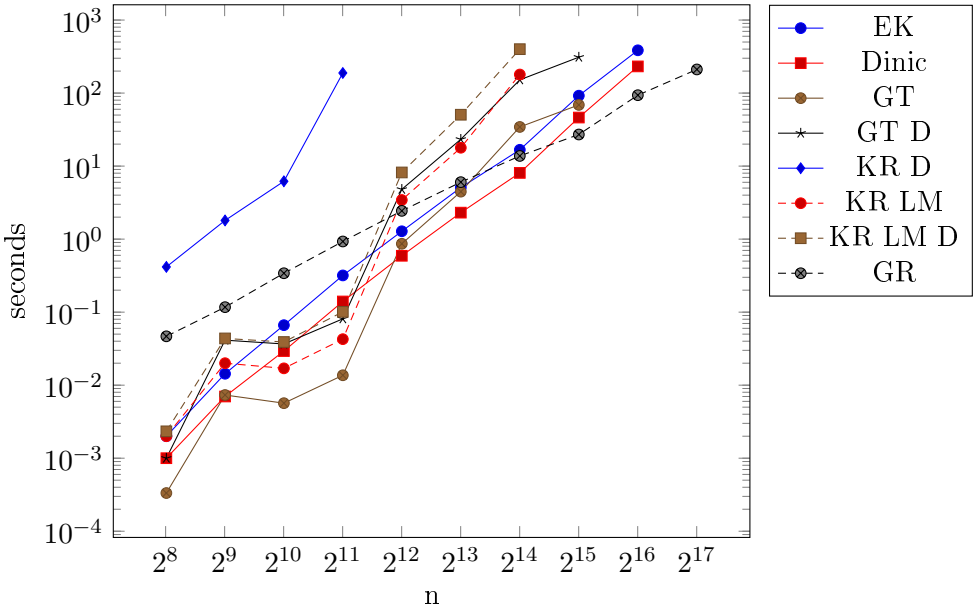

Figure 19: Results from Wash with 64 rows

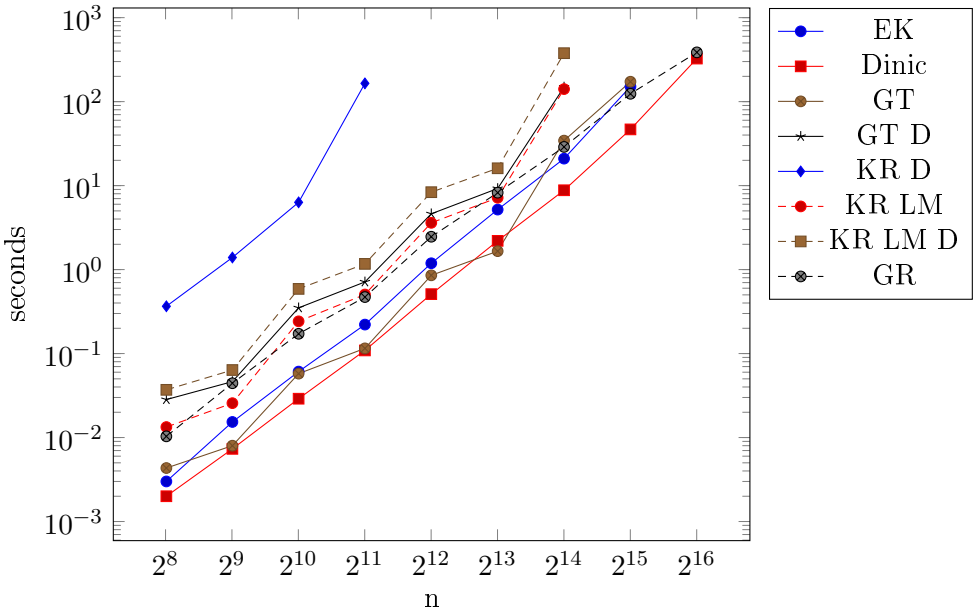

Figure 20: Results from Wash with 64 columns

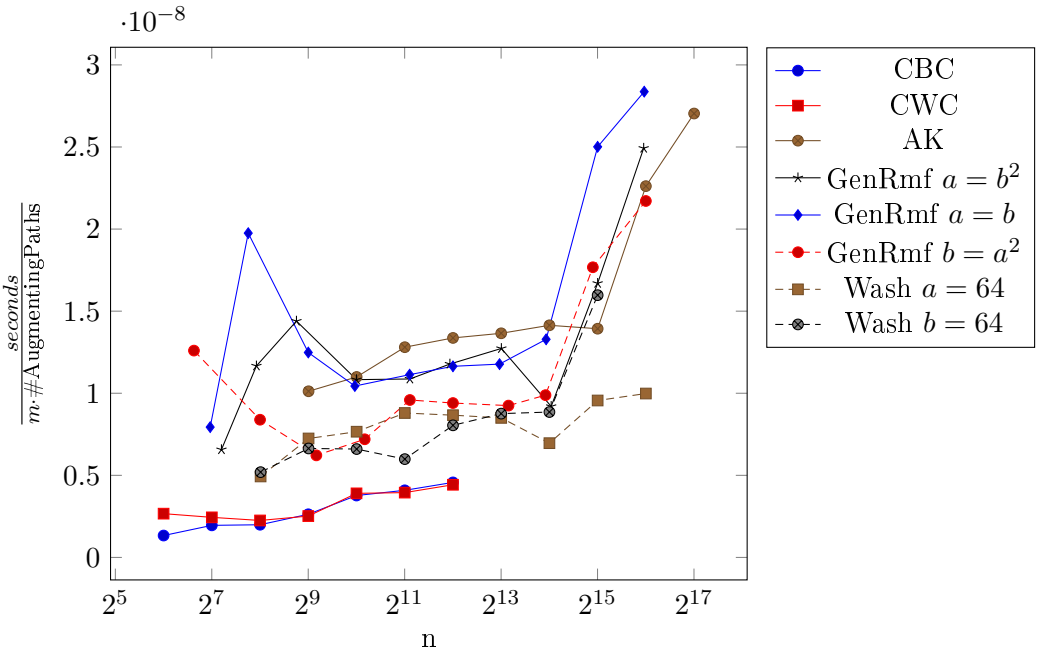

Figure 21: Runtime of the Edmonds Karp algorithm per edge and augmenting path.

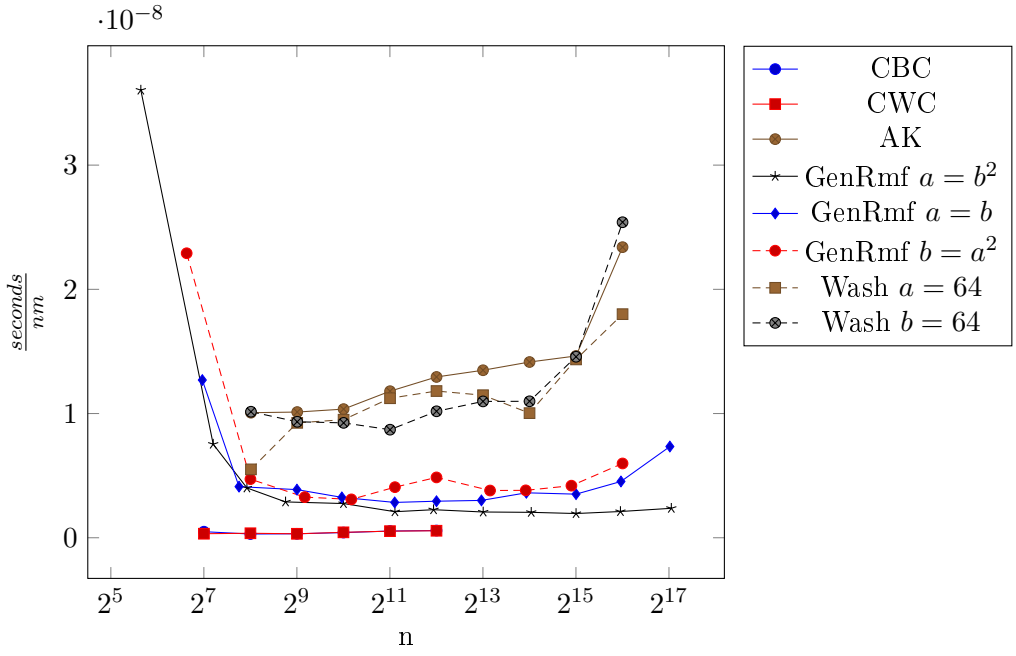

Figure 22: Runtime of the Dinic algorithm per nm.

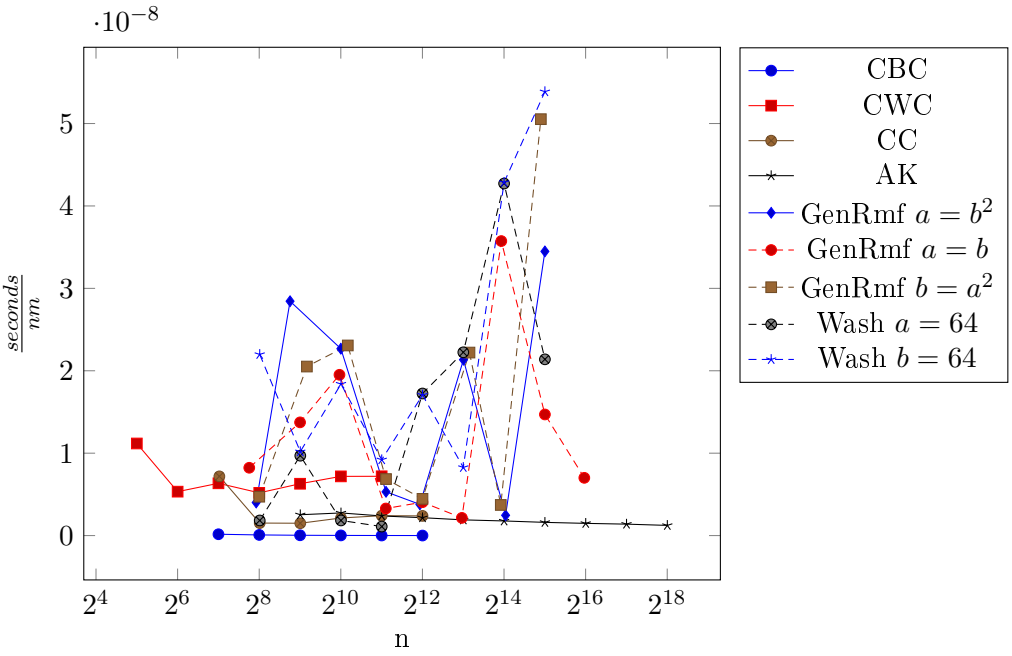

Figure 23: Runtime of the GT N3 algorithm per nm.

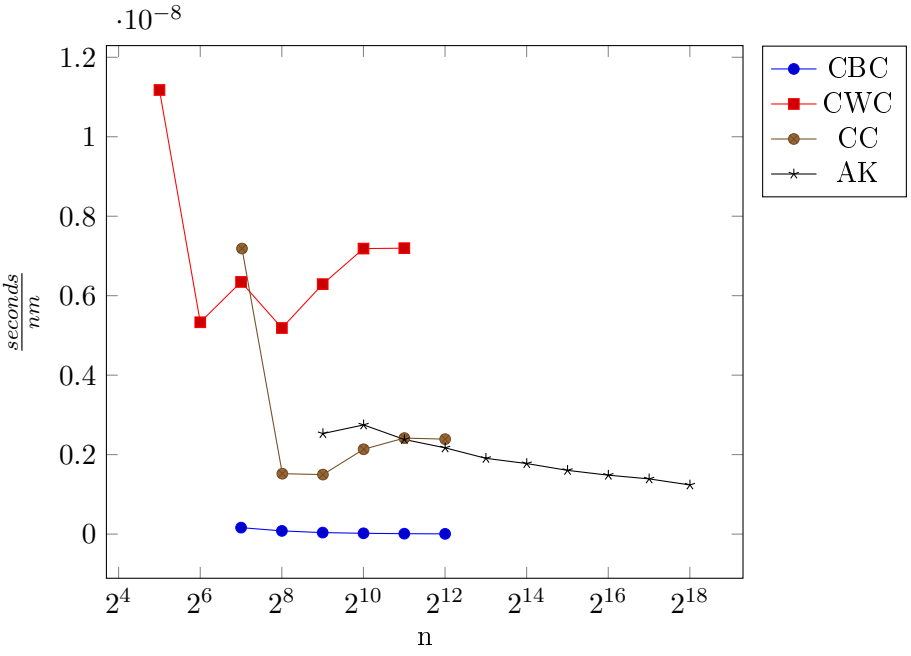

Figure 24: Runtime of the GT N3 algorithm per nm without GenRmf and Washington.

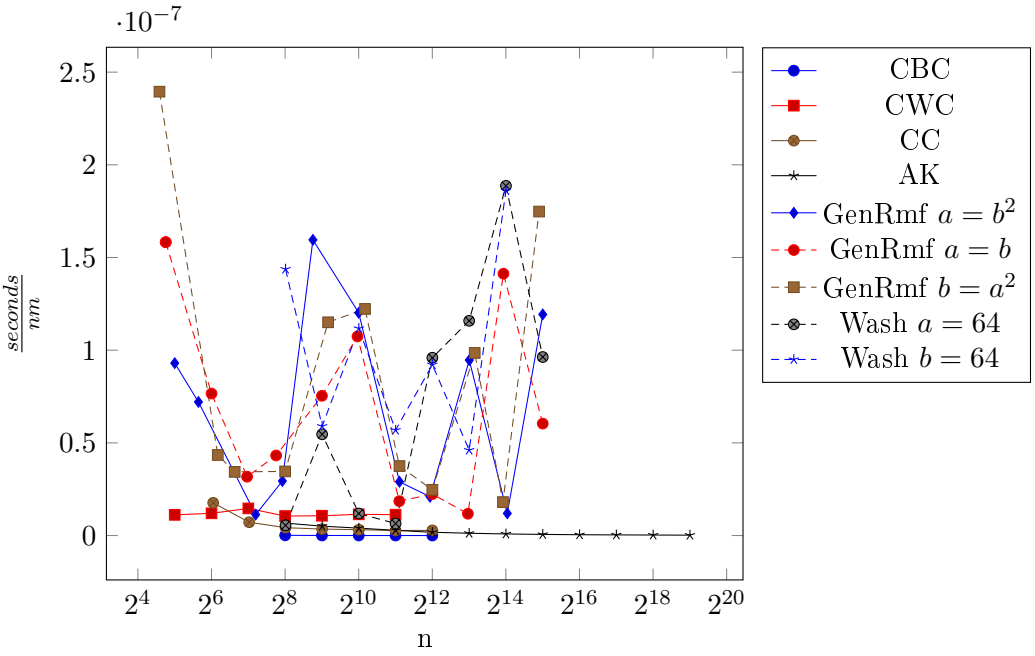

Figure 25: Runtime of the GT Dynamic algorithm per nm.

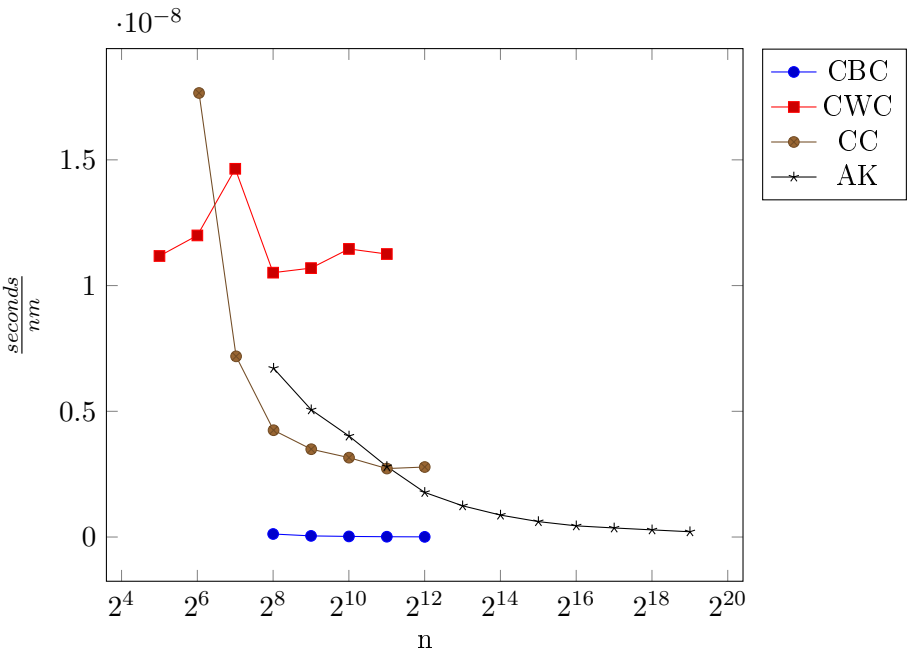

Figure 26: Runtime of the GT Dynamic algorithm per nm without GenRmf and Washington.

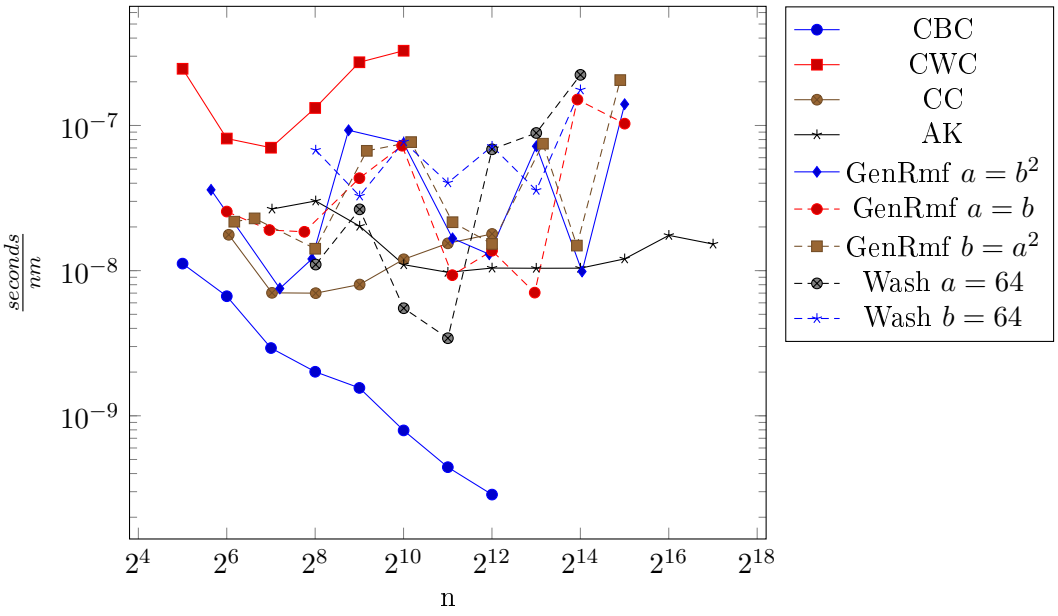

Figure 27: Runtime of the low memory KR algorithm per nm. Note that the y-axis is logaritmic.