Aarhus University

Master Thesis, Computer Science

An experimental comparison of max flow algorithms

Authors:
Jakob Mark Friis
20080816, jfriis@cs.au.dk

Steffen Beier Olesen
20080991, beier@cs.au.dk

Supervisor:
Gerth Stølting Brodal

February 23, 2014

Contents

Abstract

Max flow algorithms have improved a lot since the paper by L. R. Ford and D. R. Fulkerson in 1956 [FF56]. The most recent contribution by Orlin [Orl13] shows that all instances can be solved in $O(nm)$ time. In this thesis, we will outline the history of max flow problems, as well as implement and compare some of the most interesting ones. In an effort to improve performance, we managed to reduce the memory used by an algorithm by V. King and S. Rao [KR92] from $O(nm)$ to $O(m)$, without compromising the theoretical running time.

1 Introduction

The max flow problem is a directed graph problem. A graph consists of a set of nodes and a set of edges connecting the nodes. The edges are directed, meaning that if an edge goes from the node v_i to the node v_j, you will be able to follow that edge from v_i to v_j, but not the other way. The graph contains two special nodes, a *source* node s and a *target* node t. The problem is to determine how much flow can be sent through the graph from s to t. Each edge has a limit to the amount of flow that you can send over that edge. This limit is called the capacity of the edge, and that is what limits how much flow that you can send from s to t. In the end of any max flow algorithm, there is not allowed to be any flow imbalance in nodes other than s and t. That means that for any node other than s and t, the total amount of incoming flow has to be equal to the total amount of outgoing flow. An example can be seen in Figure 1. The numbers on the edges in the first graph signify the capacity of the edges. The numbers in the solution graph signify how much flow is sent on the edge out of the capacity on the edge.

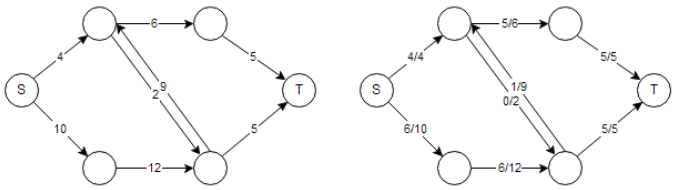

Figure 1: An example of a max flow problem and a solution to it

A number of problems can be reduced to the max flow problem, such as maximum cardinality bipartite matching, maximum independent path and maximum edge-disjoint path. Maximum cardinality bipartite matching problem is, given a bipartite undirected graph with node sets U and V, select the maximum number of edges connecting $u \in U$ and $v \in V$ such that no nodes are connected to two selected edges. An example of using max flow to solve maximum cardinality bipartite matching can be seen in Figure 2.

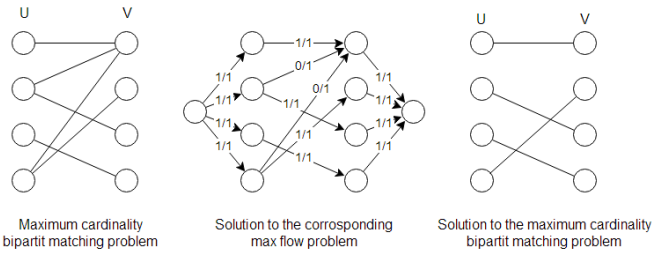

Figure 2: A max flow solution to maximum cardinality bipartite matching

An area where real world max flow problems arise is in computer vision. The problem of identifying objects in an image can be reduced to a min-cut problem, which is the dual of the max flow problem. Min cut being the dual of the max flow means that solutions to one of the problems can easily be transformed into a solution to the other. Furthermore, making a cut in the graph gives an upper bound to the solution while a valid flow in the graph is a lower bound on the solution.

Max flow algorithms have been around since 1956 [FF56], and since then many interesting algorithms have been published in the field. A recent publication by Orlin [Orl13] proved that max flow problems can be solved in $O(nm)$ time for sparse graphs where $m = O(n^{16/15-\varepsilon})$. Combined with a result by V. King and S. Rao [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. With integer capacities, all correct implementations are guaranteed to return exactly the same output, which makes it easier to verify.

We start by going over the terminology that we will use throughout the thesis in Section 2. Some central ideas are repeated throughout several papers. We give a general overview of these ideas in Section 3. Section 5 will contain a survey where we give a brief overview of the main improvements made since the first paper in the field [FF56]. Sections 6 to 10 contain more detailed descriptions of the algorithms we selected to study. We explain how the algorithms work, how they achieve their bounds, and what, if any, modifications we have done to implement them. Sections 13 and 14 contain a description of the tests we have run, and the results we have obtained.

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, 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.

We assume without loss of generality that all nodes can be reached from the source, and that all nodes can reach the target. Graphs that do not satisfy this assumption can be trimmed to fit the constraint in $O(m)$ time

by performing breadth first searches from the source and from the target.

A *path* in a graph is defined as a list of nodes $(v_1, \ldots, v_k)$ where $(v_i, v_{i+1}) \in E$ for $i = 1, \ldots, k-1$ 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 $P = (v_1, \ldots, v_k)$ is said to be *residual* if $r(v_i, v_{i+1}) > 0$ for $i = 1, \ldots, k-1$. An *augmenting path* $P = (v_1, \ldots, v_k)$ is a residual path in G where $v_1 = s$ and $v_k = t$. 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 consequence, if additional flow equal to the minimum residual capacity was pushed on the path, these 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, $distance(u, v) = n$. This could be infinity instead of n, but since the longest path possible contain all nodes, there can at most be $n - 1$ edges in such a path. Infinity can not be represented in a traditional integer, so n is easier to work with.

The *excess* of a node v, $e(v)$ is how much flow has been sent to the node, but not sent on to other nodes. It is defined as $e(v) = \sum_{u \in V} (f(u, v) - f(v, u))$. This may generally only be negative for the node s and positive for the node t and 0 for all other nodes.

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 referred 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 used throughout the thesis can be found in Appendix A.

3 Paradigms

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

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 repeatedly find and saturate 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 augmenting paths are found. For instance if you consider the graph in Figure 3, 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

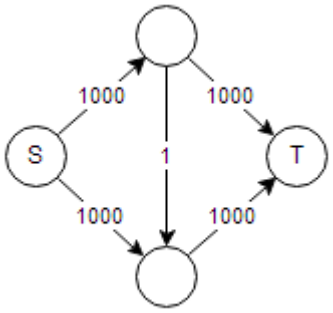

Figure 3

integer capacities, graphs can be created for which infinity many augmenting paths could be found if they are found in a special order.

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.

The algorithm explained in Section 6 is based on finding augmenting paths.

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. The nodes in the layer graph are the same as the nodes in the graph G, and an edge (u, v) only exists in the layer graph

if $distance(s,u) < distance(s,v)$, and $r(u,v) > 0$. In most blocking flow algorithms, it is sufficient that the layer graph is a directed acyclic graph (DAG). The general idea is that all paths from s to t have the same length, so you can find the augmenting paths shortest to longest, but algorithms like Goldberg Rao [GR98] actually might add some edges (u,v) to the graph where $distance(s,u) = distance(s,v)$, as long as the graph remains acyclic.

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.

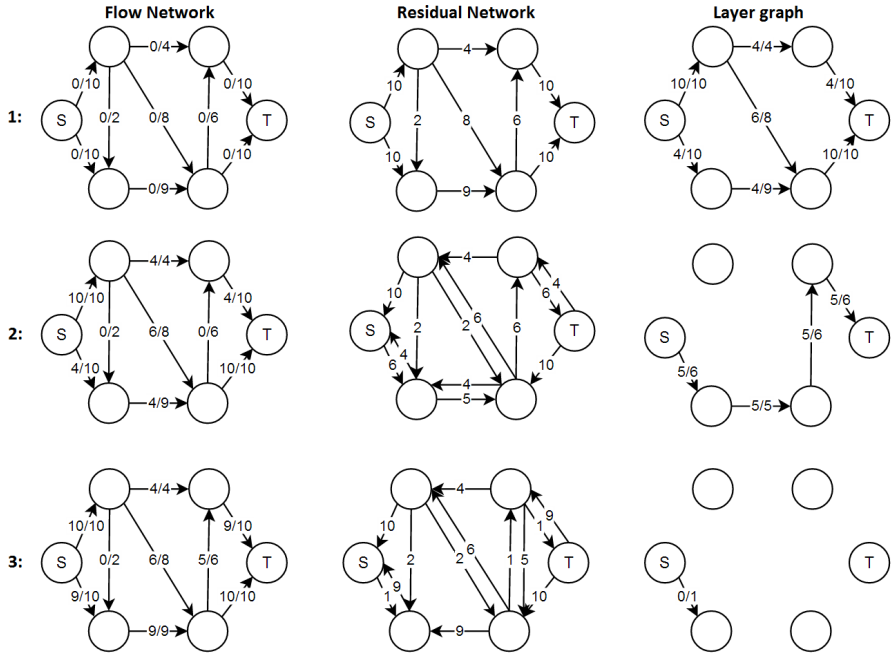

Figure 4: An example of running a blocking flow algorithm.

The algorithm explained in Section 7 is an example of a blocking flow algorithm. For an example of running a blocking flow algorithm, see Figure 4.

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 conservation 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 non-negative integer *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 neighbours 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 relabelled, 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 receives excess, it can always send it back to the node it received it from, so if it has no place to send its excess, there must be some neighbour 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 relabelled 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 excess will have been pushed to either s or t, which means that the flow conversation constraint is fulfilled. At this point, a push relabel algorithm will have found a valid flow, which is in fact the max flow.

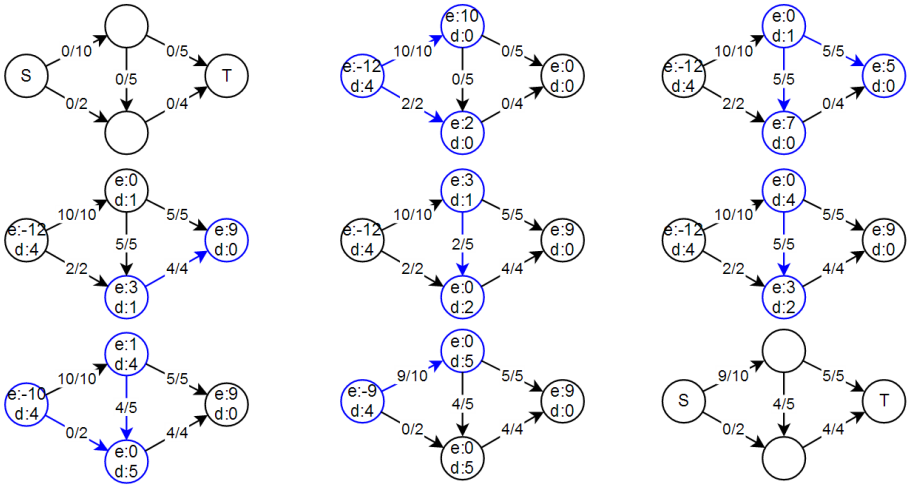

Figure 5: An example of running a push relabel algorithm.

An example of running a push relabel algorithm can be seen in Figure 5. See Section 8 for details on a concrete push relabel algorithm.

4 Dynamic Trees

Dynamic trees, also called link cut Trees, is a data structure that was presented by D. D. Sleator and R. E. Tarjan in 1983 [ST83]. It is a forest where each node in a tree can maintain set of values assigned to them, such as a cost. Updates done to the tree are performed in amortized $O(\log k)$ 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. a has to be a root for this to be possible.

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

SetCost(a, c)
: 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 last 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 corresponding node in the dynamic forest. The nodes in the graph will have an edge which is the preferred edge to move flow along. Linking a to b represents that the

edge (a, b) is the preferred edge of a. The cost of the nodes in the dynamic forest will represent the residual capacity on the preferred 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, the way it is done for Goldberg Tarjan in Section 8 is

$v_j \leftarrow \text{GetMinCostNode}(v_i)$
$c \leftarrow \text{GetCost}(v_j)$
$\text{AddCost}(v_i, -c)$
Cut all GetMinCostNodes with 0 cost

Since the dynamic tree can update the residual capacity of k edges in $\log k$ time, it is not possible to keep a value updated in the graph. For this reason, algorithms will have an incorrect residual capacities in the graph while an edge or its corresponding back edge is in the dynamic tree. The residual capacities are instead updated when a node is cut from the tree.

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. This bound is achieved by observing that nU is an upper bound on the flow that can be sent, since there can be n edges out of s, and each of them has at most U capacity. Each augmenting path sends at least 1 unit of flow, so this results in $O(nU)$ augmenting paths. Finding an augmenting path can be done in $O(m)$ time, which leads to the bound of $O(nmU)$. It 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 shortest 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 algorithm 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 A. V. 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 several publications that use the same basic ideas as A. V. 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 by R. E. Tarjan in 1984 [Tar84].

B. V. Cherkasky in 1977 gave an algorithm 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 published 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 greater 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]. They 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 improvement was to use a non constant scaling factor, so the excess dominator did not have to be a power of two. 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 use of dynamic trees. They changed Goldberg and Tarjans algorithm to use scaling, just as Ahuja and Orlin [AO89] did, though with a non constant scaling factor. To achieve a better time bound than [GT88] they randomly permutated the adjancency list of each vertex at the start, and for a single vertex

when relabelling 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 arithmetic 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 adjacency matrix and partitions 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 de-randomize 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 feasible flow, but instead creates pockets of nodes. Excess pockets are pockets with more incoming that outgoing flow, and deficit 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 sim-

ply 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 cycles of 0 labelled distance edges and calculates the max-flow in the resulting graph using a blocking flow algorithm developed by A. V. Goldberg and R. E. Tarjan [GT90]. This idea leads to an algorithm with a running time of $O(\min\{n^{\frac{2}{3}}, \sqrt{m}\}m \log \frac{n^2}{m} \log U)$. More details on this algorithm can be found in Section 10.

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 $O(nm + m^{31/16} \log^2 n)$. Orlin show that in the case of m being $O(n^{16/15-\varepsilon})$, the algorithm runs in $O(nm)$ time. Combined with the result from [KR92], this means that max-flow can always be calculated in a running time of $O(nm)$. [Orl13] also developed an algorithm running in $O\left(n^2/\log n\right)$ 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 according 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 decreases, so 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 summarize, 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 according 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. 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 that goes from s to t, 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. When the depth first search reaches t, 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. This means that the entire part of the graph that is in front of the bounding edge can be visited again if there is another path going to it. To avoid having the search process a part of the graph that can not reach t multiple times, we mark nodes that can not reach t as dead when they are discovered. When we have visited all edges on a node, we know that that node can no longer reach t. Edges going to dead nodes are removed once discovered. When the depth first search is done, and we have found all augmenting paths in the layer graph, we compute the residual network of the original graph. A new layer graph can then be computed based on the residual network. This process repeats until we arrive at a layer graph that is not connected to t, at which point all augmenting paths have been found.

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 strictly 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 have 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 of length n, so we need to calculate level graphs and blocking flows in at most n iterations.

Every time the depth first search visits a node, it runs through all of its edges. When it is done with one of the edges, and that part of the graph is dead, it removes the edge. This means that edges to nodes that can not reach t can only be used once. So discovering and trimming the dead parts of the graph takes $O(m)$ time in total.

If the part of the graph it visits is not dead, the path will end up by reaching t. In this case, an augmenting path is saturated, and the bounding edge is removed. This path will have length k, which is at most n, so finding and saturating the path will take $O(n)$ time. Since we remove an edge every time we saturate a path, we can find at most m such paths. This means that finding the paths take $O(nm)$ time.

Computing the level graph was done with a breath first search that stops when it reaches t. A breath first search takes $O(m)$ time, so 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 And Tarjan 1988

The push relabel algorithm of A. V. Goldberg and R. E. 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 that are estimated to be closer to the target. If at some point the excess of a node can not reach the target, the excess is moved back into the source. In the end, the preflow of the algorithm satisfies the flow conservation constraint. The preflow is therefore an actual flow and in fact it is the maximum flow.

The extra notation used is in Section 8.1. 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 analyses a new algorithm which uses the dynamic trees data structure, described in Section 4, and modifies the $O(n^3)$ algorithm slightly to obtain a running time of $O(nm \log \frac{n^2}{m})$. Any actual modifications done to the implementations of the algorithms are described in Section 8.4.

8.1 Notation

The algorithms estimate the distance from nodes to the source/target by giving a label $d(v)$ to each node $v \in V$. The label of the source, $d(s)$, is set to n and the label of the target, $d(t)$, is set to 0. 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 labelling 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 the notation f' will be used for allowing the flow to be negative. It is defined as $f'(u,w) = f(u,w) - f(w,u)$, meaning if f' is negative there is flow on the edge going the opposite direction. This simplifies excess from Section 2 to: $e(v) = \sum_{u \in V} f'(u,v)$. The constraint

$$\sum_{w \in V} f'(v,w) = 0, \forall v \in V \setminus \{s,t\}$$

is referred to as the anti-symmetry constraint. An edge (v,w) is stated as *eligible* if it has $r(v,w) > 0$ A node v is *active* if $e(v) > 0$.

8.2 The Push Relabel Algorithm Without Dynamic Trees

In this section, a simple $O(n^3)$ version of the push relabel algorithm will be described and analysed.

8.2.1 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: $d(v) \leftarrow \min\{d(w) + 1 \mid (v,w) \in E, r(v,w) > 0\}$
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 node w. The amount of flow that can be moved are limited by the residual capacity of the edge, (v,w), and by 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 (v,w) where there is a positive residual capacity and the label $d(v)$ is one higher than $d(w)$.

Since the push only applies 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 relabel procedure finds the neighbouring nodes with minimum labels c. This means than when a node v is relabelled to $c+1$, v can push to all these nodes, thus enabling the push operation.

To start the algorithm all edges going out from the source are saturated, which results in all the endpoint nodes of these edges become active and that there is some flow to be pushed around. In case these edges form the minimum cut all the flow will be moved to the target. If not, some of the flow must be moved back into the source. All edges not outgoing from the source have 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 fulfils the requirements for a Push operation it performs 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.

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 eligible edge to have labels higher or equal to v's, it is part of the requirements. This implies that the value of $\min\{d(w)+1 \mid (v,w) \in E, r(v,w) > 0\}$, is greater than $d(v)$. As the relabel operation is the only one changing the labels, this implies that the labels are always increasing. Note that if a node is being relabelled, it is because it has excess, and eligible edge to push on. It will always be able to push it back to the node it received it from, so if it is not allowed to push back to the node it received it from, it is because the other node has a higher label. This means that the set $\{(v,w) \in E, r(v,w) > 0\}$ is always non empty when performing a relabel operation.

In this paragraph it will be shown that all the requirements of the Relabel operation are fulfilled when it is being applied. The requirements for applying a relabel operation on node v were that v is active and $\forall w \in V, r(v,w) > 0 \implies d(v) \leq d(w)$. If a relabel is done on node v then v must be active. It is then enough to show that either $d(v) \leq d(w)$ or $r(v,w) = 0$ for all outgoing edges of v. We assume the labels are valid. If $r(v,w) \geq 0$ and $d(v) > d(w)$ then a push can be done on the edge (v,w). But this push can not be true because the labelling $d(v)$ has not changed since (v,w) was the current-edge, so all the residual capacity must have been used at that time, this means that if $r(v,w) \geq 0$ then $d(v) \leq d(w)$. If at some point after the last relabel $r(v,w)$ was equal to 0, then either that is still true or a push has been performed on (w,v). If this push was done then that means that $d(w) > d(v)$, which is still the case as only $d(w)$ could have increased since then.

The PushRelabel method 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 nodes are active, is 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 operation to it, until it either gets relabelled or it becomes inactive. If it gets relabelled, it is reinserted into the back of the queue. 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

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

Require: v is active
1: **procedure** PUSHRELABEL(v)
2: Edge $e \leftarrow$ 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**

on can be seen in the Discharge function in Algorithm 3. The initialization and main loop can be seen in the same pseudocode.

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:        f(s, v) ← cap(s, v)
12:        e(v) ← cap(s, 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 a push was performed on edge (v, w) and 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
```

8.2.2 Correctness

To argue for correctness of the algorithm, it will 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 operation to an active node, meaning excess can not be stuck at any node.
3. If a node is active, then 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 target

The first item is to make sure that the algorithm does not miss a chance to push on a residual edge. The second to fourth items guarantee that excess is moved around and that the excess that can not be moved to the target will be moved back 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 show all the items

The labels in the graph are valid when the algorithm has been 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) eligible 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, 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 assigns $d(v)$ to a node v, it opens up the possibility of pushing to at least one node, one of the nodes with label $d(v) - 1$. Hence one of the actions are always applicable.

Lemma 8.1 *Given a preflow f if v is active then the source s is reachable from v in the residual graph.*

Proof Proof by contradiction. Denote the set of reachable nodes 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 node in S to a node in $\bar{S}$ this means that for every pair $u \in \bar{S}$ and $w \in S, f'(u,w) \leq 0$

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

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

According to Lemma 8.1 an active node v_k has a path to s, denote it $(v_k, v_{k-1}, \ldots, v_o, s)$, which gives an upper bound on the label of v_k. 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 $d(v_i) - d(v_{i-1}) \leq 1$, so the maximum possible label 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 there exists no augmenting path. That means t is not reachable from s in the residual network*

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

Proof Proof by contradiction. 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 *When the algorithm terminates the preflow is a max flow, 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 analyses the running time of the algorithm. It will be done by bounding the number of relabel- and push-operations being made.

8.2.3 Running time

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

To analyse 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 (v, w), said in other words: $r(v, w) \leq e(v)$. 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 one higher than the node being pushed to, two 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 method does it bounds the number of non-saturating pushes to $O(n^3)$.

To analyse the discharge operation the concept of *passes* over the queue Q is used. The first pass, consists of applying the discharge operation 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 at most $4n^2$.*

Proof A potential function is used. $\phi = \max\{d(v) \mid v \text{ is active}\}$. If over a pass no relabels are done, all the excess is moved to nodes with lower

distances thus decreasing ϕ. If a relabel is happening and it is increasing ϕ this means that the change in ϕ is less than or equal to the change in the distance label. The total amount of change in labels that can happen was shown in Section 8.2.3 to be less than or equal to $2n^2$. This means the potential only increases by $2n^2$. Every pass where it decreases, decreases it by at least 1, thus the number of passes where it decreases are also $2n^2$. Adding the passes together gives a maximum of $4n^2$ in total. □

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 $O(nm)$ plus $O(1)$ 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 $2n$ relabel operations are happening to each node and each contribute 2 run-throughs, since before each relabelling 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 at most $4n$ runs through the edge list, for a total of $O(n\delta_v)$ work being done per 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 work. The number of saturating pushes was bound to $O(nm)$. Adding the work done in the 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 $n \leq m \leq n^2$.

8.3 The Push Relabel Algorithm With Dynamic Trees

The push relabel algorithm described and analysed in the following sections is basically a slight modification of the $O(n^3)$ algorithm described in the previous sections. This new algorithm has a running time of

$$O(nm \log \frac{n^2}{m})$$

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 found in Algorithm 4.

8.3.1 The Algorithm

The dynamic trees data structure uses amortized $O(\log k)$ time per operation, where k is the path length in a dynamic tree. The trees are introduced to bring the time spent on each non-saturating push down to sub-constant. The issue is that if a node v has done a non-saturating 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 $O(\log k)$ 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 relabelled 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 two values are not synchronized since that would mean updating all the edges in a tree-path leading to an $O(k)$ time operation instead of $O(\log k)$. To solve this issue an invariant is introduced stating that every active node v, which was defined as all nodes v where $e(v) > 0$, 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 have a node representing them in the dynamic tree data structure, which is a root with 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 is applicable, 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.

Algorithm 4 The 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 edge-list 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 edge-list
17: Cut all children of v in the tree, also for each child u Update the edge (u, v) and its linked edge 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 linked edge 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**

8.3.2 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 relabelled. 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.

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 at most $4n^2$.

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 relabelled or when its excess is increased above 0. The total number of relabels were bounded to $2n^2$. The excess only increases when a push or send operation has been done. This can happen in two different cases the first, labelled a, is when the two 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 plus perhaps 1 extra addition 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 $(n - 1)$ plus the number of cuts.

To bound the number of send operations the number of occurrences 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 occurrences is used. A non-saturating occurrence is when no cut happens in the send operation. This corresponds to a non-saturating push.

We will refer to the dynamic tree containing node v as T_v. Any tree with a path length greater than $k/2$ is noted as *large* and otherwise it is *small*. If b happens it means that $|T_v| + |T_w| > k$, this means that either T_v or T_w is large.

We first look at the case where T_v is *large*. Since this is a non-saturating occurrence all the excess is moved from root node v in T_v into T_w making v inactive. Meaning this can only happen once per pass.

If the tree T_v has changed(linked/cut) since the beginning of the pass, we charge the cost of the non-saturating operation to this operation. This happens at most $O(nm)$ times over the course of the algorithm. 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 $4n^2 \cdot 2n/k = O(n^3/k)$ over all passes. Added together this is $O(nm + n^3/k)$.

In the case that T_w is large, the push and send operation done adds the root of T_w to Q, this can only happen once per pass. From here a similarly argument to the previous paragraph 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□

Theorem 8.8 *The push relabel algorithm using dynamic trees has a running time of* $O(nm \log k)$ *plus* $\log k$ *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 Optimizations

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

Two different versions of the push relabel algorithm have been implemented. One using dynamic trees and one without them.

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.

9 King Rao 1992

V. King and S. Rao published in [KR92] an algorithm which runs in time $O(nm + n^{2+\varepsilon})$ for any $\varepsilon > 0$. 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. The purpose of the game is to select current edges, which determines 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 bipartite 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.

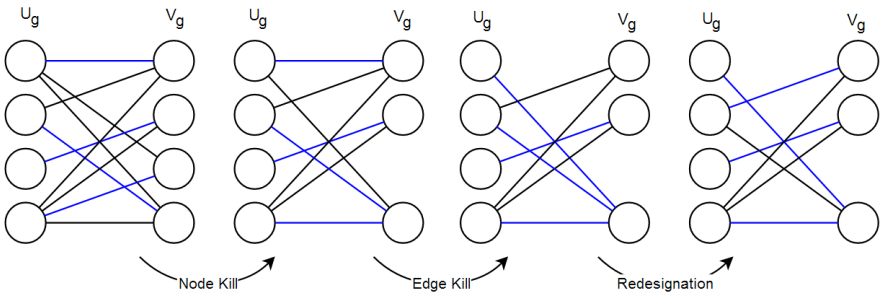

Figure 6: An example of moves in the game. The adversary gains two points from these moves.

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, 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 $u \in U'_g$ whose 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 number 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 happened 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 occurred.*

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 neighbours 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 neighbours 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 neighbours 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 designations 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 designations 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 $\#\text{edgeKills}_k$ represent the number of edge kills since last reset on level k, and $\#\text{redesignations}_k$ to represent the number of redesignations during the specific reset operation. This gives us the equation

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

So, the total points scored by the adversary is

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

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

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

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 neighbour 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 require $O(tP(N, M) + tN)$ 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 $r_0 \,\text{degree}_{\text{initial}}(v)$ 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

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

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

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

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

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

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

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

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

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

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 relabelled 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 relabelled. 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 possible label $d \in \{0, ..., 2n\}$. For every $(u, v) \in E$ 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 corresponding edge $((u, d(u)), (v, d(u) - 1))$ is killed by the adversary in the game. When a node u is relabeled to $d(u) + 1$,

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
```

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 relabelled, and when a current edge is redesignated in the game.

9.4 Correctness

Lemma 9.3 *When a node is relabelled, it has no eligible edges.*

Proof A node is v relabelled 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 corresponding 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 relabelled do $d(v)$. Both cases result in the corresponding edge being ineligible. □

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 relabelled from $d(v_i)-1$ to $d(v_i)$. According to Lemma 9.3, for a node to be relabelled, it can not have any eligible 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 relabelled 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 augmenting 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 length 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 relabelled 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 relabelled 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 eligible edges for a node. It only delays push and relabel operations the nodes until 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 according to *ucap* can be done in $O(m \log m) = O(m \log n)$ time, since $m \leq n^2$.

The relabelling 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 relabelled $2n$ times, which 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 analyse dynamic tree operations separately.

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 excess 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 directly 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 categories. 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 relabelled at least twice to reach a label higher than the source node. If the edge is not yet saturated, the adversary will receive a point when doing the relabelling, unless the player redesignated the edge before the relabelling. 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] and [CHM90, 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 $P(N,M) = O\left(N^{0.5+\varepsilon}M^{0.5} + \frac{\#\text{edgeKills}}{N^\varepsilon}\right)$. 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

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

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

$$C(n^2, nm) = O\left(nm + \frac{n^{1.5+\varepsilon}m^{0.5}}{n^{0.5+\varepsilon}/m^{0.5}} + \frac{n^2}{n^{0.5+\varepsilon}/m^{0.5}}\right)$$

$$C(n^2, nm) = O\left(nm + nm + n^{1.5-\varepsilon}m^{0.5}\right)$$

$$C(n^2, nm) = O\left(nm + n^{1.5-\varepsilon}m^{0.5}\right)$$

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 $m \geq n^{1+\varepsilon}$.

Unfortunately, the game has M edges, and N nodes, and each of those nodes have t linked lists. This means that the algorithm uses $\Omega(M + Nt) = \Omega\left(nm + n^2/\varepsilon\right)$ space, which makes it difficult to run it on large graphs. This is particularly bad when where m is close to n^2, since we would require $O(n^3)$ memory.

9.6 Contributions

The [KR92] algorithm has some major problems that makes it very slow in practice compared to other max flow algorithms. The biggest problem is that the game takes up too much space. For many graphs, we found that the algorithm crashed because it tried to allocate too much memory. We decided to make an algorithm that uses the same basic strategy for calculating the max flow, but with $O(nt+m)$ space. This also improves the running time in practise because it allows the algorithm to keep the memory inside the CPU cache for longer.

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 corresponding node kill, we not only have to remove edges that are now ineligible due to labels, but we also have to add edges that have become eligible from the node that we relabel. To do this efficiently, we keep a linked list of edges that are ineligible due to level in each node $u \in U_g$. When v is relabelled from $d(v)$ to $d(v)+1$, we first run through all incident active edges (u, v), and move them into the ineligible 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 ineligible linked list of the node $u \in U_g$ that corresponds to v. If any edge now goes 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 per node becomes $O(nm)$ time for relabels in total, without counting the time for designating edges.

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

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

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 receives more edges when it is relabelled, and a node can only be relabelled $2n$ 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 relabel 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 corresponds to saturating pushes, and the number of saturating pushes remain unchanged.

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

To make the numbers fit, we set

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

We get

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

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 $O\left(\frac{\#\text{edgeKills}+P(N,M)+N}{r_0}\right)$ 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 relabelled. 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 relabel, and we already bounded the total time for relabelling. So, since each edge can be killed $O(n)$ times, we get a cost of $O(nm)$ for maintaining eligible 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 $O\left(\frac{\#\text{edgeKills}+P(n,m)+n^2}{r_0}\right)$.

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

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 optimized memory, and one with both optimized memory and without dynamic trees.

10 Goldberg And Rao 1998

The algorithm by A. V. Goldberg and S. Rao, [GR98], is a maximum flow algorithm built on the blocking flow paradigm. The main idea is to use a binary length function on the edges. This means that the edges can have either have 0 or 1 in length as opposed to other algorithms, where they always have length 1. Edges with large capacity is assigned this new 0 length. The idea is to saturate these large edges earlier by including them in an earlier layer graph.

Assigning zero length edges might lead to a cycle of zero-length edges. This is an issue, as our blocking flow algorithm only works on DAGs. To fix this we contract all the zero-length cycles into super-nodes and run the algorithm on the graph of these instead. When the blocking flow algorithm is done we route the flow internally in each super-node. This might lead to a problem if the flow found is too big to be routed. We therefore decrease

the found flow to something we are guaranteed to be able to route before the actual routing is done.

The algorithm keeps a value F as an upper-bound on how much flow that can still be sent from the source to the target. This means that when F is zero the algorithm is done and have found a maximum flow. Every iteration of the algorithm we calculate a new value for F, but we only update F if the value is less than or equal to $F/2$.

The maximum flow algorithm has a running time of

$$O\left(\left(\min n^{\frac{2}{3}}, m^{\frac{1}{2}}\right) m \log \frac{n^2}{m} \log U\right)$$

The Goldberg Rao Maximum Flow algorithm is described in broads terms with pseudocode in Algorithm 7 In the Sections 10.1 to 10.7 we describe how

Algorithm 7 Goldberg Rao Maximum Flow Algorithm

```
1:  procedure MaxFlow(V, E, s, t)
2:      while F > 0 do
3:          Update all the distance labels based on the residual edge capaci-
    ties
4:          Find all the strongly connected components, formed by zero
    length edges.
5:          Contract all these components into super-nodes
6:          Find the blocking flow
7:          if The blocking flow is too great to be routed within the SCCs
    then
8:              Adjust the flow.
9:          end if
10:         Route the flow in the SCCs, then add the flow to the graph.
11:         Update F if necessary.
12:     end while
13:     return e(t)
14: end procedure
```

the different steps of the algorithm are achieved. Some of these steps requires algorithms described in other papers, each of these algorithms have their own Section at the end of Section 10. Before that we analyse the correctness and running time in Section 10.8 and 10.9, respectively. Section 10.10 describes the modifications we have done in the implementation of the maximum flow algorithm.

In the following sections we will use the following short-hand notation

$$\Lambda = \min\{n^{\frac{2}{3}}, m^{\frac{1}{2}}\}$$

We will also use SCC to denote a strongly connected component.

10.1 Updating The Distance Labels

Every edge is assigned a binary length to it. Based on these lengths each node v is giving a *distance labelling* $d(v)$. The distance labelling must have the following properties: $d(t) = 0$ and $d(v) \leq d(w) + l(v, w)$ for all edges with positive residual capacity. $l(v, w)$ is a length function $l : E \to \{0, 1\}$. If the length of an edge (v, w) satisfy the distance labelling equation with equality we call the edge *admissible*. We call the graph where the only edges are the admissible edges the *admissible graph*. the admissible graph is a layer graph.

We base the length function on an value called Δ and use the following binary length function:

$$l(v, w) = \begin{cases} 0 & \text{if } r(v, w) \geq 2\Delta \\ 1 & \text{otherwise} \end{cases}$$

So that edges with large capacities have zero-length. We want the distance from the source to the target to increase when we augment with a blocking flow. Therefore special consideration has to be given to the place where a zero-length edge (v, w) exist in one direction, and its linked-edge (w, v), has a capacity, close to 2Δ. When augmenting flow along (v, w) this could then lead to (w, v) becoming a zero-length edge. Which could case the labels of v and w to stay the same and the distance between s and t to remain the same. To fix this we call edges (v, w) that satisfies the following three constraints *special*:

1. $\Delta \leq r(v, w) \leq 2\Delta$
2. $d(v) = d(w)$
3. $r(w, v) \geq 2\Delta$, which would mean (w, v) is a zero edge in the previous definition

Based on the special edge we use the following length function instead:

$$\bar{l}(v, w) = \begin{cases} 0 & \text{if } r(v, w) \geq 2\Delta \text{ or } (v, w) \text{ is special} \\ 1 & \text{otherwise} \end{cases}$$

The distance labels are actually the same regardless whether you use l or $\bar{l}$. The change is that the special edges now also can be used to send flow over, and that any nodes v and w with a special edge linking them has to be in a zero-length cycle together, so they are contracted. The reason why we choose that the residual capacity of a special edge should be in the range $[\Delta; 2\Delta]$ is that we will later bound the value of the augmenting flow to be at most Δ, see Section 10.9.

When we update the distance labels we do it by using a modified BFS. We have a value i for the current distance label we are assigning to nodes. We initialize i to 0. We keep two queues. One with nodes we should assign i to called 'the current-queue' and one with nodes we should assign $i+1$ to called 'the next-queue'. Whenever we process a node v, we look at all its incoming edges. If the edge (w, v) has positive residual capacity we calculate the length of the edge using $\bar{\ell}$. If the length of (w, v) is zero we add node w to the current-queue, otherwise we add it to the next-queue. Whenever a node has been fully processed we get the next node to process from the current-queue. When the current-queue is empty we increase i by one and use the next-queue as the the current-queue and vice versa. We start this method by processing the target node, and are done when both queues are empty.

To prevent us from processing the same node several times we reset the distance label of each node in the graph to ∞ before we process them. Then we can recognize whether a node has been processed by looking at its label.

The time to reset the labels are $O(n)$ As each edge is only considered once in this modified BFS the running time is $O(m)$. The total time to update the labels are therefore $O(m+n) = O(m)$.

10.2 Finding All The Strongly-Connected-Components

To find all SCCs composed of only zero-length edges we use a slightly modified version of an algorithm made by R. E. Tarjan, [Tar72]. The running time of the algorithm is $O(m)$. For further details on this algorithm see Section 10.11. The returned value of the algorithm is a list of sub-lists. Each sub-list contains a number of nodes which form a SCC.

10.3 Contracting The Strongly Connected Components

When we run the blocking flow algorithm each super-node consist of a bunch of sub-nodes. The edge-list of the super-node is made by having an iterator run through the edge-lists of the sub-nodes. To prevent any overhead to occur we give each super-node a list of all the sub-nodes it consists of. We also runs through all the sub-nodes and gives them a link to the super-node representing them. In this manner we can work with the super-nodes with only $O(1)$ amount of overhead.

There exists at most n super-nodes. The initialization of each super-nodes takes $O(1)$ time, when not counting the time to make the lists of sub-nodes. The total time to make these sub-lists sums to $O(n)$. The last thing we do is to set up the pointer from the sub-nodes, this also takes $O(n)$ time. Combine these running time and the cost of contraction, which is the cost to set up the super-nodes, is $O(n)$.

10.4 Finding The Blocking Flow

To find the blocking flow in the graph consisting of super-nodes we use an algorithm by A. V. Goldberg and R. E. Tarjan, [GT90]. Further details on the algorithm can be seen in Section 10.12. The blocking flow algorithm runs in

$$O(m \log \frac{n^2}{m})$$

We have modified the blocking flow algorithm slightly. The analysis of how this affect the running time of the maximum flow algorithm can be found in Section 10.10

The result of the algorithm is a flow, that could be used to augment the current flow to get a blocking flow in the admissible graph.

10.5 Adjusting The Flow

If the flow f found in the previous Section has a value greater than Δ then we have no guarantee that we can route it internally in the super-nodes. We therefore decrease f in the following manner.

Set $U = f - \Delta$. Place U units of excess at the target node. Process each node in a reverse-topological order. When processing a node, decrease the flow on incoming edges until the excess is zero.

Since each edge is only considered once this has a running time of $O(m)$.

10.6 Routing the flow

To route the flow internally in each SCC we use an algorithm by B. Haeupler and R. E. Tarjan, [HT07]. For more details see Section 10.13. The motivation for the algorithm was that the authors observed you could get a small constant factor decrease in the theoretical running time of the A. V. Goldberg And S. Rao maximum flow algorithm by doing it. Using the new algorithm saves us a Δ on the length-functions. The original article had $l(v, w)$ being zero only if the residual capacity was higher than 3Δ. The effect of the change is included in the running time Section 10.9

The running time of this algorithm is $O(m)$.

When the routing algorithms has been run, a flow f of most Δ has been found and routed. We therefore augment the flow in graph with f. This means running through all edges, adding the new flow and updating the residual capacity of each edge and its linked edge. This takes $O(m)$ time.

10.7 Updating F

F represents an upper bound on the flow that can still be send from the source to the target. This value must be lower or equal to the value of every S, T-cut. This is because it is the dual problem. This means that if we could

saturate a S,T-cut then it would be the min-cut, if the cut could not be saturated then it is because another cut with a smaller capacity have been saturated. This smaller cut has to be the min-cut. Therefore the S,T cut had a capacity greater than the min-cut.

To bound F from above we therefore find a series of cuts. A *canonical cut* is a cut S_k, T_k where $S_k = \{v \in V \mid d(v) \geq k\}, T_k = V \setminus S_k$.

To calculate the canonical cuts we initialize an array with $d(s)$ entries. We iterate over all edges. If the nodes of an edge (v, w) have distances $d(v) > d(w)$ then we add the residual capacity of the edge to entry $d(v)$ in the array. When done we find the minimum among the cuts. If this value is less than $F/2$ we update F to this value.

We use $O(1)$ time per edge. We initialize an array of at most n entries, and scan it once. This leads to a running time of $O(m + n) = O(m)$.

We have now described all the different pieces of the algorithm. The next sections will show the correctness of the algorithm and analyse the running time.

10.8 Correctness

We keep augmenting the flow in the graph by flow found by the blocking flow algorithm. Assuming the blocking flow is valid and the routing of the flow is also valid the flow stays valid throughout the run of the maximum flow algorithm.

We do sometimes change the blocking a little before we route it. This happens if the amount sent in the blocking flow algorithm is too large. When we decrease it we keep making sure that the excess of all nodes, besides the source and target node, stay zero. Therefore the flow must still be a valid flow after the decrease.

F represent the maximum amount of flow we can still send from the source to the target, and we keep augmenting until F is zero. This must mean that the flow at this point in time is a maximum flow, as no augmenting path exists.

10.9 Running Time

We need to have an initial value for F. We look at an initial S,T-cut, where the set S only consist of the source node and T then the rest. Only the edges from the source crosses the cut. That is at most n edges. Each edge has at most U capacity. Therefore we initialize F to nU. We define a phase to be a number of iterations until F is updated. This means F is at least halved every new time a phase starts. Which means we have at most $\log nU$ phases.

We will now bound how much work is done in each iteration. We have already described all the operations done each iteration, and they all, except the blocking flow algorithm, have a running time of $O(m)$. The running

time of the blocking flow is $O(m \log \frac{n^2}{m})$. Summing up all the contributions therefore gives a running time for each iteration of $O(m \log \frac{n^2}{m})$.

We will look at how many iterations can be done each phase. We bound the number of iterations done where the augmenting flow is not a blocking flow. We used Δ to bound the value of each augmenting non-blocking flow. If we want to get the wanted total running time of $O\left(\Lambda m \log \frac{n^2}{m} \log nU\right)$, then the number of non-blocking iterations per phase can be no more than $O(\Lambda)$. We therefore set Δ equal to $\frac{F}{\Lambda}$. This means that after $O(\Lambda)$ non-blocking iterations F has to be updated and a new phase is begun.

To bound the number of blocking iterations in each phase we want an upper bound on the capacity of the minimum canonical cut.

Lemma 10.1 *Using a binary length-function the capacity of the minimum canonical cut satisfies:*

$$\textit{cap(min-canonical-cut)} \leq \frac{mM}{d(s)}$$

Where M is the maximum residual capacity of all the edges with length 1.

Proof The sum over all edges of their residual capacities has to be less than or equal to mM. There has to exist $d(s)$ cuts. Therefore the minimum has to satisfy the equation. □

If the graph is dense, meaning that m is high, a better bound of the minimum canonical cut can be found in the following lemma. We use the same terminology as in the previous one.

Lemma 10.2

$$\textit{cap(min-canonical-cut)} \leq \left(\frac{2n}{d(s)}\right)^2 M$$

Proof We define $V_k = \{v \in V \mid d(v) = k\}$. We want to show that there exist a k such that both V_k and V_{k+1} contains at most $\frac{2n}{d(s)}$ nodes. If this is the case then the total amount of edges is less than or equal to $\left(\frac{2n}{d(s)}\right)^2$, this gives a maximum cut value of the stated value in the lemma.

We know that $\sum_{k=0}^{d(s)} |V_k| = n$. Using the pigeon hole principle that means that at least $\lceil d(s)/2 \rceil + 1$ of the sets V_k has at most $\frac{2n}{d(s)}$ nodes. Therefore there has to exist two consecutive sets with at most this number of nodes, giving us the lemma. □

Only edges with length 1 is counted in the canonical cuts. Therefore M has to be lower or equal to 2Δ.

Lemma 10.3 *If we assume that the distance between the source and the target node increases after a blocking flow. Then there is at most $O(\Lambda)$ blocking iterations per phase.*

Proof Suppose Λ is $m^{1/2}$. Every blocking flow iteration increase the distance to the source by at least one. Then after 4Λ iterations $d(s)$ has to be greater than or equal to $4m^{1/2}$. Combined with Lemma 10.1 this gives the following equations:

$$\text{min-canonical-cut} \leq \frac{mM}{d(s)} \leq \frac{m2}{d(s)}\Delta \leq \frac{m2}{4m^{1/2}}\frac{F}{m^{1/2}} = \frac{F}{2}$$

This means that after at most 4Λ iterations the phase ends, regardless of how many non-blocking iterations have been made. If we used the original length-function the number of iteration would have been 6Λ instead of 4Λ.

A similar calculation can me made if Λ is $n^{2/3}$. Here 4Λ iterations are needed. Combined with Lemma 10.2 this is:

$$\text{min-canonical-cut} \leq \left(\frac{2n}{d(s)}\right)^2 M \leq \frac{8n^2}{d(s)}\Delta \leq \frac{8n^2}{4^2 n^{4/3}}\frac{F}{n^{2/3}} = \frac{F}{2}$$

□

So we have $O(\Lambda)$ iterations between phases. Each iteration takes $O(m \log \frac{n^2}{m})$ time. Combined with the fact that the number of phases is $O(\log nU)$ This gives a total running time of $O(\Lambda m \log \frac{n^2}{m} \log nU)$.

The next Theorem reduces the running time to the one we want.

Theorem 10.4 *The running time of the maximum flow algorithm is*

$$O(\Lambda m \log \frac{n^2}{m} \log U)$$

. Assuming a blocking flow increases the distance from s to t

Proof We will look at the work done when Δ is in three different value ranges.

$\Delta = 1$. If Δ is equal to 1 the algorithm finishes after Λ iterations. If Δ is 1 then that means that F is less than or equal to Λ. Each iteration augments the flow with 1. Combining these facts mean that F is an upper bound on the number of iterations.

$\Delta \in]1 : U]$. In this interval there has to be $\log U$ phases by the way we have defined F and Δ. The number of phases are equal to the number of times we halves F. When we halve F we also halve Δ, as Δ is equal to $\frac{F}{\Lambda}$, Λ is constant throughout the run of the algorithm. The number of times we can halve $\Delta = U$ before it reaches 0 is $\log U$.

$\Delta > U$. In this case the number of iterations are also equal to Λ. As Δ is greater than U every edge has to have length 1. It also means that every iteration has to be a blocking flow. Therefore after 2Λ iterations the distance of the source, $d(s)$, has to be greater or equal to 2Λ. Then that means that after at most 2Λ iterations $F \leq \Lambda U$, which implies that Δ is less than or equal to U. We can prove the previous statement by using Lemma 10.1 and 10.1: If $\Lambda = m^{1/2}$ then:

$$F \leq \frac{m}{d(s)}M \leq \frac{m}{2m^{1/2}}U < m^{1/2}U = \Lambda U$$

If $\Lambda = n^{2/3}$ then:

$$F \leq \frac{4n^2}{d(s)^2}M \leq \frac{4n^2}{4n^{4/3}}U = n^{2/3}U = \Lambda U.$$

The amount of phases done by the algorithm can therefore be reduced to $\log U$. The rest of the work is hidden by the O-notation removing the constant in front of Λ. □

We now to show that the distance from the source to the target is increasing when a blocking flow is used to augment the current flow in the graph. We denote distance labelling giving based on a length function l d_l. We use l for the unmodified version of the distance function and $\bar{l}$ for the modified one.

Theorem 10.5 *Let $\bar{f}$ be the a flow found in the graph of admissible edges. $f' = f + \bar{f}$ is the flow after the current flow has been augmented with $\bar{f}$. Let l' be the length function corresponding to f'. Then we want to show the three following statements*

1. *The distance labelling d_l used to calculate $\bar{f}$ is also a distance labelling with respect to l'.*
2. *$d_{l'}(s) \geq d_l(s)$.*
3. *If $\bar{f}$ is a blocking flow then $d_{l'}(s) > d_l(s)$.*

Proof For the proof, please see [GR98, Lemma 4.1], [GR98, Corollary 4.2] and [GR98, Theorem 4.3]. □

10.10 Implementation Modifications

Since we used a modified version of the blocking flow algorithm with a running time of

$$O\left((n^2 + nm)\log\frac{n^2}{m}\right)$$

Instead of only $O(m \log \frac{n^2}{m})$ we get a factor n slowdown throughout each iteration, which also means a factor n slowdown in the total running time, giving a running time of

$$O\left(\left(\min n^{\frac{2}{3}}, m^{\frac{1}{2}}\right) nm \log \frac{n^2}{m} \log U\right)$$

for the maximum flow algorithm.

When we decrease the flow found in the blocking flow algorithm, in case it is too large, we do not do it in a reverse topological order. We do not do any topological sorting, since our modified version of the blocking flow algorithm does not require it. We instead keep a queue over the nodes to process. When a node is processed it decreases any incoming flow until its excess is zero. We start the process by putting the target in the queue. We define pass $i + 1$ over the queue to mean to process all the nodes added in pass i. The first pass, pass zero, is when we add the target to the queue. The flow does not contain any cycles, this means that when we decrease the flow in each pass, we move it one edge closer to the source. As the maximum length of a path is n this gives at most n passes. Each node does a constant amount of work on each edge it decreases on. The amount on edges sums to m meaning that the total running time of this decreasing flow algorithm is $O(nm)$. The original algorithm only took $O(m)$ time. The time does not affect the total running time, as we have already increased it by n per iteration by using the modified blocking flow algorithm.

It may happen that the source and the target is in the same super-node. In this case the algorithm may fail, so the algorithm does these iteration in a special fashion. Since we know that we can send Δ flow from the source to the target internally in the strongly connected component they are both a part of, we skip the blocking flow calculation. Instead we just run the routing flow algorithm. we make sure that the source it the root of the trees being constructed and then we place a demand with value Δ in the target and route it as usually. This does not affect the running time in any negative way.

10.11 Tarjan's SCCs Algorithm

The algorithm was designed by R. E. Tarjan and published in 1972, [Tar72].

The algorithm returns a list of sub-lists. Each sub-list consists of nodes which are in the same SCC.

The general idea is that you iterate over unprocessed nodes. When you process a node v you push it on a stack. Then you recursively visit and process all its friends if they are unprocessed. The friends are all the nodes where v has an outgoing edge with positive residual capacity to. The recursion is done in a DFS manner.

A node is marked if it is on the stack.

Every node is given an unique index, which is given the first time they are pushed onto the stack. The indexes given start at zero and is increased by one each time a new one is given, in this way you can distinguish in which order the nodes were pushed on the stack. Each node also has a value called the lowlink, which represent the lowest node on the stack which is an friend. It starts out being its own unique index.

When you process a node v and it has chosen a new friend to visit if the friend is on the stack v updates its own lowlink to the minimum of its current lowlink and the friend's lowlink.

sWhen returning from the recursion on a friend a node similarly updates it lowlink to be the minimum of its current lowlink and the friend's lowlink.

If a node v has run through its entire list of friends it is done. If it has a lowlink value lower than its own index then it is allowed to stay on the stack, otherwise it and everything above it on the stack is popped off.

If v is popped and its lowlink is still equal to its own index, then v has to be part of a strongly connected component with all the elements on top of it. This has to be the case as all of them were put on the stack later based on a DFS starting from v. Since they are still there they have to have a lowlink lower than them selves, but v would have gotten this same value meaning v has to be the one they all have a lowlink to. Leading to the fact that v can reach them and they can all reach v, so they are part of the same SCC.

The algorithm creates a new sub-list whenever it encounters a node w that has its lowlink being equal to its own unique index. The sub-list is filled with w and all the nodes above it.

Each node is only processed once, where its entire edge-list is read through once. This sums up to $O(n + m)$ over all nodes.

10.11.1 Implementation modifications

The maximum flow algorithm only contracts strongly connected components that are linked by zero-length edges. Therefore the algorithm to find SCCs has been changed to only visit nodes connected with edges of zero-length.

As we ran the algorithm on large graphs we ran into an issue with the recursion. The stack-frames for each recursion simply takes up to much space. If we have a massive SCC component it is all on the stack and a lot of recursions have been done, leading to all the memory being spent and the algorithm crashing. Therefore we transformed the algorithm to avoid recursion. This allowed us to run on larger instances.

10.12 Goldberg Tarjan Blocking Flow Algorithm

To find the blocking flow in a layered graph the maximum flow algorithm uses the blocking flow algorithm made by A. V. Goldberg and R. E. Tarjan

[GT90]. The blocking flow algorithm has a running time of

$$O(m \log \frac{n^2}{m})$$

The general concept of the blocking flow algorithm is in the paper, though the proof of correctness and the running time is left to the reader. The proofs are left out in the original paper because there is another algorithm in the paper that has an analysis similar to the blocking flow algorithm. We found an error in the theory, so we have made an addition to fix the error. It can be seen in Section 10.12.3

The algorithm has a lot in common with a maximum flow push relabel algorithm. It also works by manipulating a pre-flow. The blocking flow algorithm only works on a DAG.

Every node in the graph is either in a *blocked* state or in a *unblocked* state. If a node is blocked it means that the node can not be pushed to. At the beginning of the algorithm all nodes are unblocked and they can only go from being unblocked to being blocked.

The algorithm works by first saturating edges leaving the source. Afterwards it pushes the excess around. If at some point a node can not push the flow further it blocks itself and starts pulling the flow backwards instead. When a node pulls it moves the excess back towards where it came from by decreasing the flow on incoming edges with flow on. The algorithm terminates when the excess of all nodes, besides the source and target, is zero.

Algorithm 8 Blocking flow Push, Pull and Block procedures

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**

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

Require: $\forall (v, w) \in E : r(v, w) = 0$ or $w.blocked$
7: **procedure** BLOCK(v)
8: $v.blocked \leftarrow true$
9: **end procedure**

The three main operations can be seen in Algorithm 8.

All three operations only apply to *active* nodes. A node is active if it has positive excess.

The push method moves excess from node v to w by adding flow on the edge (v, w). The amount of excess moved is the minimum of the excess of v and the residual capacity of edge (v, w), in this way the excess never goes negative, and the capacity constraint is not violated. When doing a push the receiving node w can not be blocked.

The block operation blocks a node if the algorithm can not do a push operation on any of its outgoing edges. A push can not be done on an edge if either the residual capacity is 0 or the receiving node is blocked.

The pull operation works on an edge (v, w) and only applies if node w is blocked. It pulls flow sent to w from v back by subtracting the minimum of the excess in w and the flow on (v, w) from the flow on (v, w). The pull operation only makes sense to apply to an edge with flow on it.

To know which edge to do a pull or push operation on each node has an edge pointer called the *current-edge*. The current-edge is initially set to the first edge in a node's edge list. If neither a pull nor push operation applies to the current-edge of a node, the next edge in the node's edge list is set as the current-edge. If the current-edge was the last edge of the node, then the current-edge resets to the first edge and a block operation is done.

The algorithm uses the dynamic-trees data structure, see Section 4 for a description. Each node in the graph also has a node in the dynamic tree representing it, all nodes in the dynamic tree start out not linked. The idea is to link two nodes in the dynamic tree if the edge between them can either be pulled or pushed upon later. This way entire paths in the graph can be saved in the dynamic tree. The value on each node in the dynamic tree shows how much flow can either be pulled or pushed on the edge to the parent. In the case of it being linked as part of a push operation we call it a *push edge* and the value saved is the residual capacity of the edge. Otherwise it is a *pull edge* and the value is instead the flow on the edge. This means that flow can be pulled and pushed along an entire path simple by doing an AddCost operation on the dynamic tree. A path of length k takes amortized $O(\log k)$ to do an operation on, but in this case doing the equal amount of push and pull operations would take $O(k)$ time, so adding the data structure saves time.

The addition of the dynamic trees gives some changes when pushing and pulling. Now in addition to doing a push/pull operation the algorithm also does a send operation. If after a send operation an edge (v, w) has enough flow/residual-capacity to do a pull/push the node v is linked to w in the dynamic tree, and the edge (v, w) is saved there as well. This is where we can distinguish whether the edge is a push or pull edge, if it is saved as part of a Link operation done after a push operation it is a push edge, otherwise it is a pull-edge. The linking is only done if the combined size of the dynamic trees containing v and w are lower than a constant k, v is a root and v and w are not already in the same tree. All this logic can be seen in pseudocode in Algorithm 10, in the Tree-PushPullBlock method. The constant k should

always be equal or larger than two to make sure that the dynamic trees are used.

Something to note is that the values in the tree are not synchronized with the values on the edges. The values meaning the residual capacity or the flow. The issue is that keeping the values completely synchronized would mean that when doing an AddCost operation all the edges should be updated. This means the cost becomes $O(k)$ instead of amortized $O(\log k)$ for a path of length k, destroying the entire reason to use dynamic trees. To make sure this mismatch of values does not become a problem an invariant is introduced; saying that all nodes that are active has to be a root in the dynamic tree. A root does not have a linked edge to push/pull on in the dynamic tree, so there are no value that has to be synchronized, this also means that the value of a root does not have any meaning and is generally ignored, only the excess of the root matters.

To keep the invariant any new excess added to a path has to be transferred to the root. It might be that a node v on the path has a too low value to do this. In this case enough excess is transferred to use all the value on the edge e between v and its parent. Afterwards the edge e is cut and the algorithm reiterates with the rest of the excess. This keeps going until either all the excess has been moved to roots or the active node has itself become a root, fulfilling the invariant. When cutting a node in this way the actual value are updated with the value from the tree. If it is a pull-edge this means setting the flow to 0, if it is a push-edge this instead means setting the residual capacity to 0, which in actuality is setting the flow of the edge equal to the capacity of the edge. All this combined is called the send method. Algorithm 9 shows the pseudocode for it.

Whenever a node v is being blocked, flow can not be pushed to it. To prevent the dynamic trees from doing an AddCost operation, corresponding to a push, each child of v, which has a push edge to v, is cut loose. The values are then updated for the edges in the graph based on the edges cut in the tree.

The algorithm uses a data structure to keep track of all the active nodes. It is a linked list L of sub-lists. The first element of each sub-list is called the *head*. The invariant on the data structure is that every head of a sub-list, besides the head of the first sub-list, has to be an active node and each active node has to be a head in a sub-list. To maintain this invariant the sub-lists need to be split or concatenated when nodes become active or inactive.

The initial state of L is a single sub-list containing all nodes in the algorithm sorted in topological order.

When a node becomes active the sub-list containing the node is split right before the node and the new sub-list is added to the L as a succeeding sub-list of the one it was previously a part of.

When a node becomes inactive the sub-list containing the node is concatenated to the preceding sub-list, if such a list exist.

Algorithm 9 Blocking Flow Send procedure

```
Require: v is active
 1: procedure SEND(v)
 2:     while GETROOT(v) ≠ v and e(v) > 0 do
 3:         δ ← min(e(v), FINDMINVALUE(v))
 4:         send δ value of flow in the tree by calling ADDCOST(v, −δ)
 5:         while FINDMINVALUE(v) = 0 do
 6:             u ← FINDMIN(v)
 7:             e ← (u, parent(u))
 8:             if e is an push-edge then
 9:                 f(e) = c(e)
10:             else
11:                 f(e) = 0
12:             end if
13:             CUT(u)
14:         end while
15:     end while
16: end procedure

17: function FINDMINVALUE(v)
18:     minNode ← GETMINCOSTNODE(v)
19:     return GETCOST(minNode)
20: end function
```

If a node v is blocked it is moved to the front of the entire structure. This is done by deleting v in the list it is a head of and then adding it as the first element of the first sub-list of L. The deletion is done by splitting out v and then removing the sub-list containing only it. The addition is done by concatenating v with the rest of the first sub-list.

When doing a split the outcome is two sub-lists s_1 and s_2. When doing concatenations we concatenate two sub-lists s_1 and s_2. If we scanned through the sub-lists to concatenate and split we would use linear time in the size of the smaller sub-list to do so. To do this faster every sub-list is represented as a Finger tree data structure instead. This data structure can do splits and concatenations in logarithmic time instead of linear.

The discharge method in the blocking flow algorithm takes the first active head from L and keeps calling the Tree-PushPullBlock on it until the excess is 0, meaning it no longer is active. This may activate new nodes, which each split the sub-lists. If no more active nodes exist after a discharge operation the blocking flow algorithm terminates. The pseudocode for the discharge method is in Algorithm 10

Algorithm 10 Blocking Flow Tree-PushPullRelabel and Discharge procedures

```
1: procedure DISCHARGE
2:     Node v ← first active head of L
3:     repeat
4:         TREE-PUSHPULLBLOCK(v)
5:         if w is activated do to a push or pull from v then
6:             Split the sub-list containing w and add the sub-list to L
7:         end if
8:     until e(v) = 0
9:     Concatenate the sub-list containing v to its predecessor sub-list in L.
10: end procedure

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

In the initialization step of the blocking algorithm the following things are done

- The flow on all edges is set to 0
- The nodes are topologically sorted and added to L as a single sub-list
- All edges going out from the source are saturated and the excess of the endpoint nodes are updated. The endpoint nodes are then activated in L

When all nodes, besides the source and target, have excess 0 the algorithm is done. The last thing the algorithm needs to do is to synchronize the flow with the values in the dynamic trees. This is done by running through the dynamic trees and looking for any linking edges, if some are found the corresponding edges in the graph are updated with the values from the trees. The pseudocode for the initialization, main-loop and final clean-up can be seen in Algorithm 11.

Algorithm 11 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:      Sort-Topological
6:      Initialize L
7:      for all (s, v) ∈ E do
8:          Transfer maximum capacity flow through (s, v) update (v, s) accordingly
9:          Update excess of v
10:     end for
11:     while A node is active do                          ▷ Main-loop
12:         Discharge
13:     end while
14:     Update edges with the remaining values in the dynamic trees
15: end function
```

10.12.1 Correctness

Lemma 10.6 *Whenever the Tree-PushPullBlock method does a Block operation it is allowed*

Proof This has to be the case as the block operation are only allowed to block a node if it can not push on any edge. When the Tree-PushPullBlock method blocks node v the algorithm has already been through the entire edge-list of v and done all the pushes that could be done. This has to be the case as an edge that could not previously be pushed on will never be able to be pushed on at a later time. If an edge (v, w) could not be pushed on it was because either the residual capacity was zero or that the receiving node was blocked. If the reason was that w was blocked, then that reason remains, as a node is never unblocked. If a push could not happen because the residual capacity of the edge was zero, then the only way that could change is by doing a pull operation on the edge, but that would mean that w node was blocked, preventing further pushes. □

Lemma 10.7 *A push,pull or block operation always apply to an active node*

Proof If an active node can not do a push operation and is unblocked a block operation can be done. If an active node is blocked a pull operation has to apply since the excess has to come from flow on at least one edge, on which a pull can be done. □

Lemma 10.8 *When the algorithm terminates there does not exist a path from the source to the target where flow can be sent, meaning at least one edge has to be saturated.*

Proof Proof by contradiction. Assume there exist a path $(s, v_0, v_1, v_2, \ldots, v_l)$, where node v_l is the target, when the algorithm has terminated. The edge (s, v_0) can only have a positive residual capacity if the node v_0 has been blocked and a pull has been done on (s, v_0). A block only happens when a node can not do any pushes. A push is only unavailable if the endpoint node of the edge has been blocked or there is no residual capacity left on the edge. This means that if the a node $v_i, i = 0, \ldots, l-1$ is blocked it has to be because either the edge (v_i, v_{i+1}) has no residual capacity or that v_{i+1} is blocked. Node v_l, the target, is never blocked therefore one of the edges has to have zero residual capacity and hence be saturated. □

This means that every path from source to target have a saturated edge, meaning the flow is a blocking flow.

A fear could be that the dynamic tree causes the algorithm to never terminate. This could happen if a dynamic tree contains a cyclic path of nodes. A cycle could consist of either pure push-edges, pure pull-edges or a

mixture of the two. Both of the pure cases never happens as the push-/pull-edges corresponds to actual edges in the graph, thus a cycle would mean an actual cycle of nodes in the graph, but the graph is a DAG.

The only difference in case of a pure cycle is what node is a child and which node is a parent in the dynamic tree. On a push edge (v, w) v would be the child and w the parent, on a pull edge it is the other way around. In the mixed case a node v can not be a child of a pull edge and at the same time be a parent of a push edge. If it is the child of a pull edge, that would mean that a pull has been done from v, but that implies that v is blocked. When v was blocked, the algorithm cut all push edge children and since v is blocked no new push edges could have been linked since then as no push operation to v could have been performed. This breaks a mixed cycle.

10.12.2 Running time

The number of saturating pushes are at most m. Each edge (v, w) can only have one saturating push happen to it, afterwards the only way to get more residual capacity is to do a pull operation, but that means that node w has been blocked and no further pushes can be done. The number of saturating pulls also has a maximum of m. When the flow on an edge has become zero due to a pull, it never changes, and hence only one saturating pull happen on each edge.

The work done when running through the edge-lists of the nodes is at most $2m$. All the edge-lists, summing to m, can at most be run through once before blocking and once afterwards. After an edge-list has been run through the second time for a node, all the incoming excess must have been pulled back, and no further excess can be pushed to the node, since it must be blocked.

The Tree-PushPullBlock operation only calls the block on each node, besides the source and the target, once. Leading to at most $n - 2$ block operations being done.

The number of cut operations on the dynamic trees is at most $3m$. The cuts in the send operation happens after a transfer corresponding to a saturating push or pull. Leading to at most $2m$ cut operations in the send method. The rest of the cuts happen when the children of a node are cut just before calling the block method. Each cut node has an outgoing edge to the node being blocked. Summed over all nodes the number of outgoing edges is m, leading to at most $2m + m = O(m)$ cut operations.

When linking two nodes in the dynamic trees data structure one of the nodes is a root. Each node start out as a root, and each cut creates a new root. The number of linking operations done it therefore closely linked to the number of cuts. In the end of the algorithm all the nodes might have been linked without a corresponding cut. Thus the total upper limit of link operations becomes: $\#links \leq \#starting\_roots + \#cuts + \#nodes =$

$3m + 2n = O(m)$

In the following parts a node becoming *activated* means either a root node becoming active, or an active node becoming a root. A node can only be activated in the initialization steps or in the push or pull branches of the Tree-PushPullBlock method. In the push case a node can be activated when doing the push operation or when transferring excess doing the send method, similarly for the pull case. Lastly a send operation from node v can activate v if the last transfer cuts the edge between v and its parent, making v a root.

A node being *discharged* means that the node is the first active node and is therefore selected in the discharge method.

Lemma 10.9 *The number of dynamic tree operations are at most $O(m+k)$. Where k is the number of node activations.*

Proof The number of cut and link operation were bound by $O(m)$. The remaining dynamic tree operations happen in the send operation, where there might be an addCost iteration which does not have a cut following it. This happens if the excess of the node being processed v is less than all the edges along the dynamic tree path. In this case v becomes inactive afterwards. Thus the number of addCost and getCost operations is bounded by the number of cuts plus the number of node activations. □

To bound the node activations the original authors used a fact that each node can only be activated once between nodes being blocked. This fact does not hold with the original implementation. In Section 10.12.3 we solve this issue, but for now we accept the fact.

Lemma 10.10 *The number of node activations are $O(m + n^2/k)$*

Proof All the node activations happen when a push or pull operation are done, followed by the send operation. Some of the push/pull operations lead to activations with an added link operation. In each call to the send method all but at most one node activation also does an cut. All these activations with a cut/link operation was already bounded to $O(m)$. The remaining activations comes from the last addCost operation done in a send when it does not lead to any Link or cut being performed. Denote these occurrences as *critical*. The rest of the lemma is about bounding the number of critical occurrences. The idea is to charge each critical occurrence to a cut, link or block operation.

Denote the dynamic tree in which node v appears T_v As no link occurs in a critical occurrence, this means that when doing the pull or push operation on edge (v, w) right before the send operation the combined tree size $|T_v| + |T_w| > k$. This means at least one of the trees has to have a size greater or equal to $\frac{k}{2}$. Since only n nodes exist this leads to a maximum of $\frac{2n}{k}$ trees of

this size. A tree of this size is denoted as being *large*. The analysis will look into the case of either T_v or T_w being large.

In case of T_v being large. The critical occurrence in the send operation removes all the excess from root node v, making it inactive. This can only happens once between block operations since each node only can become active once in such an interval. If the tree T_v has been changed by a link or cut operation since the last call of the block method, then charge the critical occurrence to this link/cut. In case it has not been changed, charge it to the last performed block operation. Since only $\frac{2n}{k}$ trees of this size exist, the cost of each block is at most $\frac{2n}{k}$.

If T_w is large the critical occurrence activates the root in T_w. Again this activation can only happen once in between nodes being blocked, so again the critical occurrences are charged to the cut, link and block operations, just as in the previous paragraph, with the same cost.

Thus the number of activations become: $\#Activations = O(\#Cuts + \#Links + \#Blocks\frac{2n}{k})$ inserting the bounds for cuts, links and blocks gives $\#Activations = O(m + m + \frac{n^2}{k})$ giving the lemma. □

Combining Lemma 10.9 and Lemma 10.10 gives that at most $O(m+n^2/k)$ dynamic tree operations are performed. As each tree has a size of at most k the total amount of work spent on these operations is $O\left((m + n^2/k)\log k\right)$

The running time of the rest of the algorithm will be analysed in the next sections.

Every time the Tree-PushPullBlock method is called the current edge of the node is changed or a push, pull or block operation is done. The cost of changing the current edge to the next in the edge-list is $O(1)$, each node only runs through its edge-list twice leading to a total cost of $O(m)$.

Each push/pull operation takes $O(1)$ time and each operation are followed by at least $O(1)$ dynamic tree operations. The total number of tree operations were bound to $O(m + n^2/k)$, leading to a maximum cost of $O(m + n^2/k)$ for all pushes and pulls done.

$O(n)$ block operations are done, each taking constant time.

Thus the combined cost, besides the dynamic tree operations and the time spent maintaining L is $O(m + n^2/k)$.

The theory gives a bound of maintaining L and selecting active nodes of $O((m + \frac{n^2}{m})\log k)$. In Section 10.12.3 we show that our contribution has a similar time bound.

The time was spent on 3 parts in the algorithm.

- Doing dynamic tree operations.
- Using and maintaining the lists L and L_R.
- Doing the Tree-PushPullBlock, Push, Pull and Block operations.

The time spent on first two was each bounded to $O((m + \frac{n^2}{k}) \log k)$, and the last one was $O(m + \frac{n^2}{k})$. This leads to a total running time of $O((m + \frac{n^2}{k}) \log k)$. Setting k equal to $\frac{n^2}{m}$ gives a total running time of $O(m \log \frac{n^2}{m})$. Since there is the added constraint that k should be equal or greater than 2 then the real running time is $O(\max m, m \log \frac{n^2}{m})$. This is the case as $\log \frac{n^2}{m}$ converges to zero when m converges towards n^2.

10.12.3 Fixing the theory

In this Section we describe the problem with the theory in the original article. We show how it can be fixed, so that each node is only activated once between block operations.

We have given an simple example in Figure 7 of a node being activated twice in-between block operations.

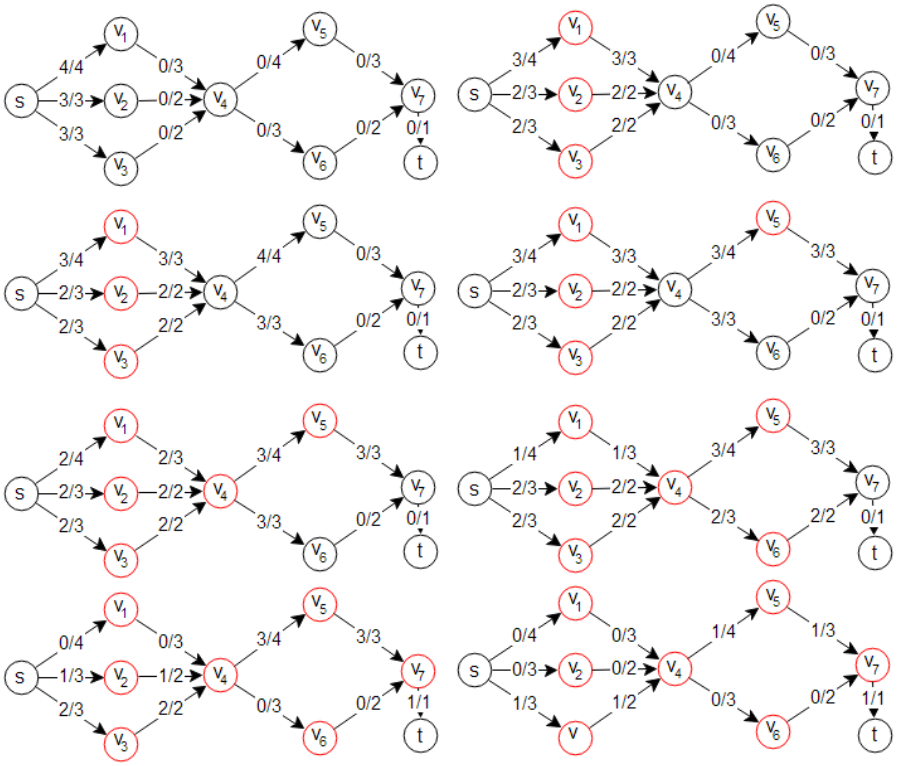

Figure 7: Example of multiple activations of a single node

The following table shows the order of the nodes in L and the work done throughout the execution of the blocking flow algorithm on the example. Blocked nodes are underlined. The images are named from a in the topmost left corner to h in the bottommost right corner. The third column show what operations have been done to reach the next image. The first image

shows how all edges outgoing from the source has been saturated.

In the table: To pull flow means to perform a pull operation in the graph. Pushing flow is the same as performing a push operation. Sending flow is doing a send operation. Linking is doing a link operation.

image	order of L	Operations done
a	$v_1v_2v_3v_4v_5v_6v_7$	Node v_1 pushes 3 flow to v_4, blocks itself and pulls 1 flow back to the source. Node v_2 and v_3 both pushes 2 flow to v_4, blocks them self and pulls 1 flow back to the source
b	$\underline{v_1v_2v_3}v_4v_5v_6v_7$	Node v_4 pushes 4 saturates its outgoing edges and becomes inactive
c	$\underline{v_1v_2v_3}v_4v_5v_6v_7$	v_5 saturates its edge, blocks itself and pulls 1 flow back to v_4
d	$\underline{v_5v_1v_2v_3}v_4v_6v_7$	v_4 is blocked and pulls one flow to v_1, v_1 sends it to the source
e	$\underline{v_4v_5v_1v_2v_3}v_6v_7$	v_6 saturates its edge, is then blocked and pulls one flow to v_4 which in turn sends it to the source
f	$\underline{v_6v_4v_5v_1v_2v_3}v_7$	v_7 saturates its outgoing edge, blocks itself and pulls two flow to v_6. v_6 sends this flow to the source, but this means that v_4 get active to cut it dynamic tree edge to v_1 as there is not capacity on it do pull two units of flow. v_4 links to v_2 and sends the remaining one flow to the source.
g	$\underline{v_7v_6v_4v_5v_1v_2v_3}$	v_7 continues by pulling two units of flow to v_5. This is again sent to the source. v_4 becomes active again as it saturates its pull edge in the dynamic tree. v_4 end up linking itself to v_3 to pull the last one unit of flow to the source.

In between f and g node v_7 is blocked. No further nodes are blocked in the rest of the run of the algorithm, yet node v_4 is activated twice afterwards. Twice v_4 has to cut its edge in the dynamic trees and set up a new one. This breaks the theory of a node only being activated once in-between nodes being blocked. This example does not appear that bad, as we only have a factor increase in activations. It is possible to construct a even worse example with $O(n^3)$ activation using $O(n^2)$ edges. This means that the running time bound is broken even if we spend only constant amount of time on each activation, which we do not.

What is the actual issue that breaks the theory? It appears as if the

action of moving blocked nodes to the front of the list L should make sure that flow having reached a dead end is moved back into nodes which can potentially use it before the algorithm continues. The intuition is that if a flow has been moved through a series of nodes and get stuck, then simply reverse the direction of the flow and push it back until it can move in a new direction. The way to do this would be to put the blocked nodes in a reverse order in L, compared to the order they had in an unblocked state. Since the graph is a layered graph, the order in each layer does not matter, only the order in which the different layers appear should be reversed. The example graph shows that the blocked nodes foremost in L is not always in a reversed order layer-wise. Notice how the layer containing node v_6 and v_5 is split by the layer containing v_4. If the order had been corrected both v_6 and v_5 would have been processed before v_4 when doing the pulling and v_4 would only have been activated once in between node being blocked. We therefore propose an addition to L to make it work.

Our idea is to add another linked-list L_R of sub-lists. This list initially contains all the nodes in a reverse topological sorted order, meaning the reverse order of L. If a node is blocked we remove it from L and instead manipulates the corresponding node in L_R in the future. Before a node gets blocked its corresponding node in L_R is not touched. When getting the first active node, the algorithm should then look into L_R, and only if no active nodes exist here should it look into L. Adding this new list means that the nodes in L and L_R never has to change order. All in all this gives a new selection behaviour that is quite like the old one, now just with a perfect reverse ordering in front of L in the form of L_R.

Each graph node is giving a link to its corresponding node in L and L_R. The nodes in L and L_R has a similar link in the opposite direction to the graph node they represent. In this way we can with $O(1)$ overhead manipulate the correct nodes by determining whether a node should be manipulated in L or L_R. This is done by following the links and checking if the graph node is blocked.

The Finger tree data structure has the following amortized time operations:

Initialize(V)
: Initializes a Finger tree with $|V|$ nodes. It takes $O(n \log n)$ time.

Head()
: Access first element, the Head, of a Finger Tree. Takes $O(1)$ time.

Split(v)
: Given a node v in a Finger tree split the Finger tree in two trees l_1 and l_2. l_1 contains all the nodes before v and l_2 the rest. Takes $O(1 + \log(\min l_1 l_2))$ time.

Concatenate(l_1, l_2)

Concatenate the two Finger trees l_1 and l_2. Takes $O(1 + \log(\min l_1 l_2))$ time.

The later proofs of the running time requires the split operation to take constant time. To facilitate this we would like to prove that the concatenation operations can pay for the extra work done by the splits.

Lemma 10.11 *The way the blocking flow algorithm uses the Finger trees, allows the amortized time of the split operation to be* $O(1)$

Proof The argument uses a potential method $\Phi = \sum_{i=0}^{k} -1 - \log |l_i|$ where $l_0, \ldots, l_k$ are all the sub-list, represented as Finger trees, in L and L_R. The potential starts out as zero and ends at a negative of at most $-2 - 2\log n$ in the end of the algorithm, where L_R only consist of a single sub-list containing all elements, and L similarly is only a single sub-list containing all elements, except the ones having been removed due to block operations.

This means we end up owing up to $-2-2\log n$ at the end of the algorithm. Paying this $O(\log n)$ cost does not break the cost of algorithm.

The initialization leads to an increase of $-2\log n$, which does not change the cost of the operation.

The access operation does not change the potential at all.

A split leads to a change in potential of:

$$\begin{aligned}\Phi_{after} - \Phi_{before} &= (-1 - \log l_1 - 1 - \log l_2) - (-1 - \log l_1 + l_2) \\ &\leq (-1 - \log l_1 - \log l_2 + 1 + \log \max l_1, l_2) \\ &= -\log \min l_1, l_2\end{aligned}$$

A fact used here is that $\log a + b \leq \log 2 \max(a, b) = 1 + \log \max a, b$. This leads to a new running time of the split operation:

$$\begin{aligned}\text{Amortized time} &= \text{Actual-Time} + \text{Change in potential} \\ &\leq (1 + \log(\min l_1, l_2)) - \log(\min l_1, l_2) \\ &= O(1)\end{aligned}$$

Similarly the new cost of the concatenations becomes

$$\begin{aligned}\Phi_{after} - \Phi_{before} &= (-1 - \log l_1 + l_2) - (-1 - \log l_1 - 1 - \log l_2) \\ &\leq (1 - \log \max l_1, l_2 + \log l_1 + \log l_2) \\ &= (1 + \log \min l_1, l_2)\end{aligned}$$

Adding this additional increase of potential to the running time, gives a new amortized running time of

$$\begin{aligned}\text{Amortized time} &= \text{Actual-Time} + \text{Change in potential} \\ &\leq (1 + \log(\min l_1, l_2)) + (1 + \log \min l_1, l_2) \\ &= O(1 + \log(\min l_1, l_2))\end{aligned}$$

□

The following proof gives a bound on the time spent maintaining L and L_r, and also selecting which nodes to discharge.

Lemma 10.12 *Using Finger trees to represent the sub-lists in L and L_r leads to a maintaining time of $O((m + \frac{n^2}{k}) \log k)$. The time spent selecting the nodes to discharge is also included in this bound.*

Proof To initialize the lists the nodes are fist sorted in topological order. This takes $O(m+n)$ time. Then L is initialized with the nodes in this order, taking $O(n \log n)$ time. Finally L_R is similarly initialized with all the nodes, though in the reverse order of L, also taking $O(n \log n)$ time.

According to lemma 10.10 the number of node activations is $O(m + \frac{n^2}{k})$. Each active node has to be discharged once. to discharge a node the algorithm only has to look at the head of the two first sub-list in both L and L_R. This takes constant time. Leading to a bound on selecting nodes in the discharge operation of $O(m + \frac{n^2}{k})$.

Each node activation leads to a split, each split takes amortized constant time according to the previous lemma. The time spent splitting sub-lists whenever a node gets activated is thus $O(m + \frac{n^2}{k})$.

When a nodes gets blocked it is removed in L and activated in L_R. To remove a node v in L we first split the sub-list starting right after v. Since v has to be active to be blocked it has to be the head of its sub-list, therefore one of the sub-lists after the split only contains v. This sub-list containing only v is deleted after the split. Finally the remainder of the old sub-list is concatenated to the predecessor sub-list in L. The node corresponding to v is then activated in L_R leading to a split. A block operation therefore leads to two splits and a single concatenation being performed. The splits takes constant time, the concatenation at most $O(\log n)$. Only $O(n)$ block operations happen leading to a combined time of $O(n \log n)$.

The last thing to bound is the number of concatenation happening due to a discharge operation making a node inactive. Let us look at an interval between two block operation and bound how much work is done concatenating the sub-lists. When doing a discharge operation on node v this leads to the sub-list containing v, denoted $S(v)$, to be concatenated with its preceding list, if such a list exist. Since the discharge treats all the nodes in a left to right manner, each successive concatenated sub-list $S(v), S(w), \ldots$ is vertex-disjoint.

We denote a sub-list *small* if it contains at most k elements and *large* otherwise.

Each small sub-list takes $O(\log k)$ time to concatenate. Since each active node has to be discharged, the number of small sub-lists could be equal to the number of node activations leading to the time spent on the small lists to also be $O((m + \frac{n^2}{k}) \log k)$.

The number of large sub-lists is $l \leq \frac{2n}{k}$. Denote the large sub-lists $S_1, S_2, \ldots, S_l$. The time spent to concatenate them in a single interval is:

$$\begin{aligned}\text{Time to concatenate} &= \sum_{i=1}^{l} \log\left(1 + |S_i|\right) \\ &\leq l \log\left(1 + \frac{2n}{l}\right) \\ &\leq \frac{2n}{k} \log\left(1 + k\right)\end{aligned}$$

The first inequality is because the worst case situation occurs when all the sub-list have an equal size. Summing over all $O(n)$ intervals in-between nodes being blocked gives a bound of $O(\frac{n^2}{k} \log k)$ time spent on the large sub-lists.

Adding all the different contributions together gives the bound in the lemma of $O((m + \frac{n^2}{k}) \log k)$. We removed the $n \log n$ term since it is dominated by the rest of the expression as k has to be equal or larger than two. □

This means our contribution has the exact same bound as the original theory.

10.12.4 Implementation modifications

The above sections described the blocking flow algorithm as it was described in the paper [GT90]. We have had to make some modifications to make it work in our scenario, where we work on a graph of super-nodes.

The blocking flow algorithm only work on DAGs, but since we do not remove any edges, the input graph does not have to be a DAG. What we really want is to only use the edges that are admissible. The definition from admissible comes from the Goldberg And Rao maximum flow algorithm. We therefore adds the same length computation to the blocking flow, so that we can distinguish which edges to use and which not to use. On last thing we need to check is that we do not make any push or pull operations between sub-nodes of the same super-node. This is easily done by following the super-node pointers of the two operands and check whether they are equal.

In many of the maximum flow algorithms we store the flow by manipulating the residual graph to save space. In the blocking flow algorithm we actually store the flow values explicitly on the edges, as the maximum flow algorithm needs to manipulate these values before they are used to augment the flow in the graph.

The biggest change compared to the theory is that we do not user Finger-trees to represent the sub-lists in L. Instead we have chosen to replace L by a first-in-first-out queue similarly to the Goldberg and Tarjan 1988 push relabel algorithm. This was done to make the implementation simpler. The

cost however is a big change in the running time. The change comes from an increase in node-activations, the following lemma gives a bound:

Lemma 10.13 *The number of node-activations using a FIFO queue is* $O(n^2 + \frac{n^3}{k})$

Proof Connect all the nodes with the edges the can transfer flow on called this graph G_f. In case of unblocked nodes this will be the edges that they can push on, for blocked nodes it is the edges they can pull on. A node is only able to activate those it is connected to. No cycles can appear in G_f. This is the case as that would mean a cycle in the labels, which can not happen as the admissible edges create a DAG. Another case creating cycles in G_f would be if an edge can both be pushed and pulled on at the same time, but that can not be the case.

The scenario giving the most node activations is if G_f is one long line of nodes $v_0, v_1, \ldots, v_n$. Where v_0 is the source or target. Node v_i can transfer excess to v_{i-1}, meaning they are connected and that v_i can activate v_{i-1}. Assume the worst, that all nodes, besides the source and target, are active. Then the worst order to pick the nodes in would be to follow this simple procedure: Process the node with the lowest index i by transferring all it excess flow into node v_{i-1}, reiterate until no nodes are active. This would lead to a worst case behaviour where v_1 would be made inactive by transferring its flow to v_0, then v_2 would transfer all its flow to v_1 activating it again, v_1 would then be made inactive by it transfering its flow to v_0, v_3 would then be processed activating v_2, which in turn activates v_1 etc. This would lead to the number of activations being $1 + 2 + 3 + \cdots + n = O(n^2)$.

The dynamic tree data structure saves this a little. If a node is processed it would link itself in the dynamic tree to its predecessor in G_f making the transfer of flow cheaper next time it would be activated. If the entire path is in dynamic trees then the number of nodes being activated for a node with index i would be $\frac{i}{k}$ as each dynamic tree has a maximum size of k, so each time k nodes has been passed, a new dynamic tree is used, costing a node-activation. This would mean the new bound is $(1 + \frac{1}{k}) + (1 + \frac{2}{k}) + (1 + \frac{3}{k}) + \cdots + (1 + \frac{n}{k}) = O(n + \frac{1}{k}(1 + 2 + 3 \cdots + n)) = O(n + \frac{n^2}{k})$. The constant term each paying for first time a node is processed, where it does its link operation.

Every time we block a node we could change G_f. This means that all these activations could happen again in each interval between a block. Since there is $O(n)$ intervals in-between nodes being blocked the number of node activations are $O(n + \frac{n^2}{k}) \cdot O(n) = O(n^2 + \frac{n^3}{k})$ $\square$

This is an increase compared to the previous bound of $O(m + \frac{n^2}{k})$. This increase propagates through the rest of the theory making the final running

time of our implementation $O((n^2 + \frac{n^3}{k}) \log k)$. Still setting k equal to $\frac{n^2}{m}$, leads to a bound of $O\left((n^2 + nm) \log \frac{n^2}{m}\right)$.

Since we no longer need a topological sorting to make L initially, we have removed it.

10.13 Haeupler And Tarjan Routing Algorithm

We use a routing flow algorithm made by B. Haeupler and R. E. Tarjan, it was published in 2007, [HT07]. Using this algorithm instead of the original one leads to a constant decrease in running time of the maximum flow algorithm. The two routing flow algorithms have the same theoretical running time.

When the blocking flow algorithm has been run some sub-nodes in each super-node can have flow going into them or out of them. We can calculate these values. We denote all flow going into a node as *supply* and everything going out as *demand*. Every super-node, besides the ones containing the source and the target, has to have equal supply and demand when summed over all the sub-nodes in super-node. The super-node containing the source has a higher demand than supply, and the super-node containing the target is the opposite way.

Since each super-nodes is a strongly connected component, it is possible for each SCC to pick a node v and build an in-tree and an out-tree with v as its root. The in-tree is a tree where every node in the tree can reach the root. The out-tree is a tree where the root can reach every node in the tree. The idea is then to route all the supply via the in-tree to the root, and all the demand via the out-tree to the root as well. The pseducode for the main idea of the algorithm can be seen in Algorithm 12

The pseudocode in Algorithm 12 also shows how to build an in-tree. A queue is used to keep track of all the nodes that should be processed. When a node is processed it looks at all its incoming edges. If an edge (u, w) goes between nodes in the same SCC and is a zero-length edge it could be that the edge should be part of the in-tree. It should only be part of the tree if u has not been processed before. We can check this by seeing if it has a parent assigned in the tree. If that is not the case then we add w as u's parent and add u to the queue.

To start the construction any node v is selected from the SCC and appointed as the root.

An out-tree is constructed in a similar way, the edges considered are the outgoing edges instead of the incoming ones.

When constructing the in- and out-tree for the SCC containing the source and the target, the source and target nodes are always selected to be the root of the trees.

The rest of the algorithm consists of three DFS's of the trees constructed.

Algorithm 12 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 {(u, w) | SameSCC(u,w) and IsZeroEdge((u, w))} do
19:             if u.InTreeParent = nil then
20:                 u.InTreeParent ← w
21:                 Add u as a child of w in the InTree
22:                 Add u to the rear of Q
23:             end if
24:         end for
25:     end while
26: end procedure
```

First we do a DFS run of the out-tree where we calculate a value called the *descendant demand*, it is the sum of all the demands of the children plus the node itself.

We then do a DFS run in the in-tree where we process the nodes bottom-up. The idea is to route the supply to the root. When we process a node v with parent w we calculate the amount of supply to move. The supply moved, sm, is set to the minimum of v's supply and Δ minus the descendant demand of v. The flow on edge (v, w) is increased by sm, the supply of v is decreased by sm and the supply of w is increased by the same amount. The idea is that we only move extra flow and excess closer to the root if we know that another part of the sub-tree of v does not have a demand for it.

The last thing done is a DFS run in the out-tree where the demand is moved towards the root. When a node v with parent w is processed we move an amount corresponding to the difference in demand and supply closer to the root in a similar fashion as in the previous DFS run.

The pseudocode for the three DFS runs can be seen in Algorithm 13.

Algorithm 13 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
```

The original algorithm just routed the entire supply and demand to the root. This could lead to the flow of an edge being 2Δ, Δ flow being used

on the supply and Δ flow on the demand. Since they also needed Δ slack for the special edges the original length function had to have non-special zero-length edges have a capacity greater or equal to 3Δ. Since this new algorithm accounts for demand being used in the sub-trees it never routes both supply and demand to the root, thereby saving up to Δ amount of flow on each edge, leading to the distance function we have used.

Building the in- and out-tree takes $O(m)$ time. Each node is only processed once. The work done when processing a node is having its edge-list run through once plus some constant time work. This sums up to $O(m)$ over all nodes.

The DFS's all takes $O(n)$ time. The total running time of the routing algorithm is therefore $O(m + n) = O(m)$

10.13.1 Implementation modifications

This algorithm uses recursion. This gives us some trouble as the recursion stacks ends up taking to much memory, crashing the algorithm in instances with large strongly connected components. We therefore rewrote the code to use just an array as our own version of a stack. When we visit a child in the DFS we put a pointer to the parent node on the first empty entry in the array, then we can easily go back to the parent when done processing the child. Since each root to node path can be at most n we only use an array of this size. The extra book-keeping time is constant, so the running time is the same. Since we use the same array all the time, we pre-allocate it and use it in all iterations.

11 Global Relabelling 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 relabelled 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 relabelling towards n, while pushing flow back and forth.

To alleviate this problem, we implemented a heuristic for all versions of these algorithms. The heuristic is taken from a paper by Cherkassky and Goldberg [CG97], and is called a global relabelling heuristic. It 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 $n + \mathit{distance}(v, s)$.

We run this heuristic during initialization, and once in a while during the execution of the algorithms. The reason we run it at the start is that a node v will have to be relabelled to $\mathit{distance}(v, t)$ anyway before its excess can be pushed to the target node. The other situation where we want to run the global relabel is right after the minimum cut has been saturated. When the minimum cut has been saturated, many nodes before the min cut have to be relabelled above n.

For all algorithms except the King Rao algorithm without optimized memory, relabelling multiple labels up could be done in the same time as relabelling one label up. The problem with the King Rao algorithm that uses $O(nm)$ memory is that when relabelling 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 correspond 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.

There is no easy way to detect when the minimum cut has been saturated. If there were, we could stop the algorithm there, and report the max flow. The best way to check if the minimum cut has been saturated is to just do a global relabelling. So we need to make some trigger that decides when to run the global relabel algorithm. We have several different implementations of such triggers. One is based on detecting when flow is pushed around in a cycle, and the others are based on monitoring the excess of the target node.

Doing a single global relabelling is just a breath which takes $O(m)$ time. If the trigger only run the heuristic $O(n)$ times, this can be done within the $O(nm)$ time bound of most algorithms.

For algorithms that use dynamic trees, it takes $O(\log n)$ time to get the capacity on the edges instead of just $O(1)$. The algorithm need to read the capacity because it is only allowed to follow edges with positive residual capacity when doing the bfs. This causes the global relabelling to take $O(m \log n)$ time instead.

11.1 Cycle Trigger

When the minimum cut has been saturated, flow is being pushed back and forth between nodes behind the cut while they are being relabelled up above n. This means that flow is being pushed around in cycles $(v_1, v_2, \cdots, v_k, v_1)$. When that happens, at least one of the nodes in the cycle has to be relabelled more than one label up. To push flow from v_i to v_{i+1}, $d(v_i)$ must be greater than $d(v_{i+1})$. This means that if no label has been relabelled twice by the time v_k gets the excess, then $d(v_1) > d(v_k)$, so v_k will have to be relabelled at least twice to send the flow to v_1.

Based on this trigger, the global relabelling heuristic runs in $O(n^2 m)$ time. It takes $O(m)$ time to do the breadth first searches, and we have n nodes that can be relabelled twice in a row n times.

11.2 Pass Trigger

In some of our tests, we found that nodes were very often relabelled more than one label up, even though the minimum cut had not been reached. In particular, in the tests of graphs described in Section 13.5, it very often relabels two labels up without pushing flow around in a cycle. For this reason, we decided to make another way of triggering the global relabel.

Our second idea is to use the FIFO queue of nodes that all of our push-relabel algorithms use to decide the order that nodes are being processed. Like in the description of the Goldberg Tarjan algorithm, we use the notation of passes. After a global relabelling check, we note which node is the last node in the queue. After that node has been processed, we consider a pass to be finished, and we perform another check to see if we should do a global relabelling. To decide whether to do a global relabelling, we check if the target node has received any excess during the pass, and since last relabel. We only perform another global relabelling, if the target has received flow since the last global relabelling, but not during the pass that just finished.

The idea behind this is that once the minimum cut has been saturated, the excess behind the cut can not reach the target. This means that once the excess in front of the cut has reached the target, the target will no longer get any excess. After a global relabelling, if the minimum cut was found,

then nodes before the cut was relabelled above n. In that case, there is no reason to do any more global renaming, and the target won't receive any more excess. If the minimum cut was not found, there must be some excess that can be pushed to the target. The minimum cut can not be saturated before this excess has reached the target. In either case, there is no reason to do a global relabel if the target has not received excess since the last global relabel.

There are $O(n^2)$ passes, so this also results in a heuristic that runs in runs in $O(n^2m)$ time.

11.3 Node Count Trigger

While testing the Pass Trigger, we found that basing the check on the passes sometimes caused the global relabelling to run very often. If there are few nodes in the passes, we might run a second global relabel after as little as two nodes has been processed. This is a problem, since the global relabelling algorithm is a bit expensive, especially for algorithms that use dynamic trees.

To resolve the issue with the small passes, we made a third trigger that is triggered after processing $f(G)$ nodes, instead of after a pass. Here $f(G)$ is some function of the graph. More detail on how we choose $f(G)$ can be seen in Section 14.1.

The logic checking whether the target node has received excess in the Pass Trigger is also used here.

The cycle trigger, and the idea of running the heuristic after $O(n)$ nodes have been processed come from the article by Cherkassky and Goldberg [CG97]. We have not seen the idea with checking the excess on the target node anywhere though.

11.4 Gap relabelling

Cherkassky and Goldberg [CG97] also presented a heuristic called gap relabelling. The idea is that if no nodes have label k, then all nodes with a label higher than k can never reach t, so they can be relabelled to n. We decided to focus on the global relabelling heuristic, and skipped the gap heuristic. According to Cherkassky and Goldberg [CG97], FIFO implementations, as we have, do not benefit significantly from the gap relabelling when global relabelling is already used.

12 Implementation

Our first decision when we began implementing the algorithms was how to represent the graph. We wanted constant time lookup of a the edges of a node. Additionally, if we have an edge (u, v), we wanted in constant time

to be able to find the nodes v, u and the edge (v, u) that we call the linked edge. We represent this such that each node has a pointer to an array of edges, each edge has a pointer to its target node and to its linked edge. To get the node an edge goes from, we have to go to the target node of the linked edge.

Many algorithms need to be able to add their own information to the nodes and edges. We allowed them to do this by adding a void pointer to each node and edge that the algorithms can set to point to a custom object that contains the information needed. Most of our algorithms start out by allocating an array of n such objects, and assigning them to the nodes. A better option could have been using templates so that the nodes could contain the special information inside them rather than in a totally different place in memory. That could save some cache misses when reading the information on a node, but it would require cloning the input graph to match the new template argument.

We choose to implement our own dynamic trees. We did find a library [Eis13], but we found it too difficult to customize and to understand what exactly was going on. Our implementation is based on splay trees instead of the original version of dynamic trees. Implementing it our selves allowed us to more easily profile what takes time during the algorithms.

Every time we run a test, we do the following steps

1. Load the graph from a text file in digraph format
2. Clone the graph
3. Start the timer
4. Allocate the algorithm
5. Find the max flow
6. Deallocate the algorithm
7. Stop the timer
8. Verify that the output is correct
9. Deallocate the clone

Step 2 to 9 is repeated three times per test. The verification is explained in detail in Section 13.1. The order in which we start the timers is done to ensure that the time we measure contains all work required except from loading the graph.

It is a little different for the library algorithms, since they have to load the graph them selves.

1. Allocate the algorithm

2. Get the algorithm to load the graph
3. Start the timer
4. Find the max flow
5. Deallocate the algorithm
6. Stop the timer
7. Verify that the output is correct

In order to verify the library algorithms, we compare the max flow value to a value returned by one of our own algorithms. A potential concern here is that we deallocate the library algorithm before we stop the timer, which probably also deallocates the graph. We wanted to make sure that any additional memory allocated during the execution of the library algorithms is being cleaned up, and not cached for future runs. The time for cleaning is measured in our own algorithms, so we wanted that to be true for the library algorithms as well. We have run some performance tests on the library algorithms, and found that the vast majority of the time spent is spent calculating the actual flow, and not on deallocating the algorithm.

Other data structures we implemented include a circular queue and a linked list. The linked list is implemented to reuse elements so that it does not have to allocate and deallocate linked list nodes every time something is added or removed from a linked list. To do this, we implemented a singleton linked list node cache that is shared between all linked lists. All algorithms clear the linked list cache as part of their clean up procedure so they do not affect subsequent algorithms.

Other things we did to improve performance is to always allocate large chunks of memory at the time. As an example, the linked list cache has a method to bulk allocate a number of linked list nodes. Algorithms that use the linked list call this method as part of the initialization with an estimate for the highest number of linked list nodes they will need at any point in the algorithm.

13 Tests

In this section we will describe what tests we have run on the algorithms, and what we expect to see from them. Section 13.1 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 13.2 contain a list of the different types of graphs we will be running on.

13.1 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.

13.2 Graph Generators

We have various different algorithms to generate different types of graphs to test on. The graph generation algorithms described in the next Sections are a mix of custom algorithms, and algorithms taken from from DIMACS [JM93].

13.3 Connected Randomized

The AC graph generator from DIMACS 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 fully connected graphs where m is close to n^2.

When we ran the push-relabel algorithms on this type of graph we got big jumps in the time spent due to the random capacities. If everything sent out of source can be sent to the target, the algorithms won't have to relabel 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 relabelled 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. In these graphs, we connected all nodes except s and t, so the graph is cyclic. Capacities on edges from s and edges to t are random in the range $[1, 10000]$. All other edges have capacity 10000 to allow all flow to be sent from one node to the other.

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

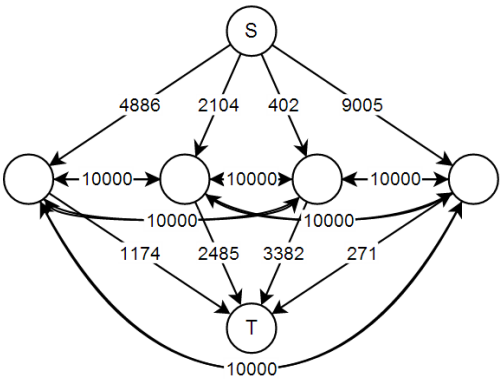

Figure 8: An example of a CRH graph.

13.4 Connected Deterministic

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

To alleviate this, we made a new type of fully connected graph. This graph is designed to contain as many augmenting paths as we could get in there. We had hoped to get $\Omega(nm)$ augmenting paths to reach the worst case running time for Dinic, but the analysis in the end of this section shows that there are $O(m \log n)$ augmenting paths. This should make the graph very difficult for augmenting path algorithms like the algorithm by Edmonds and Karp and the algorithm by Dinic.

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 9.

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.

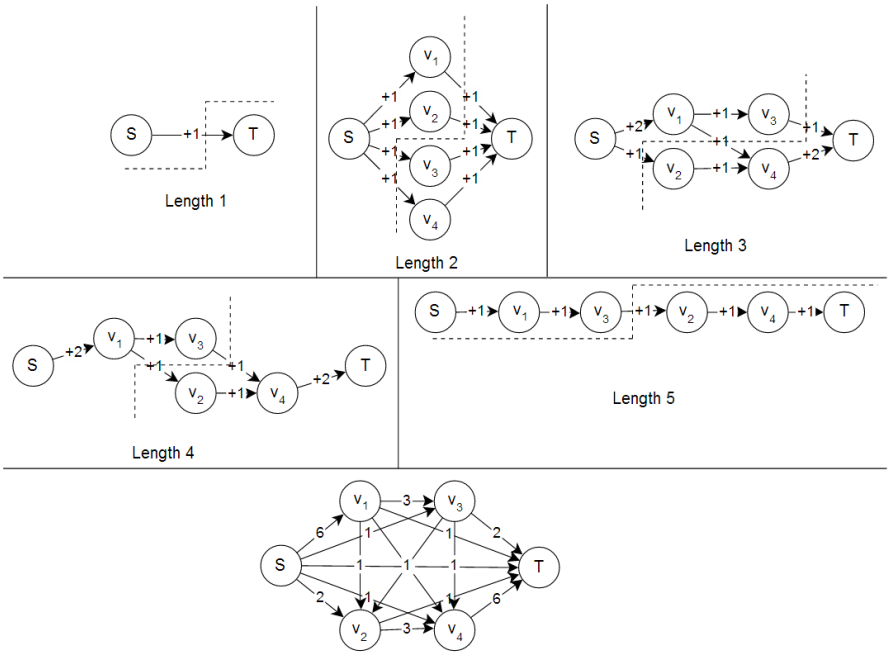

Figure 9: How the custom connected graph is constructed for $n = 6$.

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 neighbours 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 eliminate. 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 augmenting paths. However, it does mean that the execution of Dinic does not always find the expected layer graphs.

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

To calculate the number of augmenting paths in a graph like this, we consider a different type of graph that has more paths, but is harder to construct in practise. If we consider a layer graph of length k, we can partition the nodes into sets such that $V_i = \{v \in V \mid \mathit{distance}(s, v) = i\}$. Every node in $v \in V_i$ will then have an edge from at least one node in V_{i-1}, but it is not possible for them to have any edges from nodes in V_j where $j < i - 1$, since that would place v in V_{j+1}. If all of edges to a node $v \in V_i$ from nodes in V_{i-1} are saturated as a part of finding the blocking flow, v will be promoted to a set V_j where $i < j$ in the next layer graph. The maximum number of unique paths from s to t in a layer graph of length k is $\prod_{i=1}^{k-1} |V_i|$, which is the case when all nodes in V_i has an edge to every node in V_{i+1}. This product is maximized when $|V_i| = \frac{n}{k}$, meaning that all sets have the same size. Our CD graph is not perfectly rectangular in every layer graph, which is why it has fewer unique paths.

A layer graph could be constructed such that you saturate every edge from V_i to V_{i+1} when finding the blocking flow. This could lead to $O(n^2)$ edges being saturated, but it would also mean that no more paths can be found from s to t since all nodes in V_{i+1} would have been promoted to V_{i+2}, and no edges can exist from V_i or less to V_{i+2}. The worst case scenario is if as many edges are saturated as possible, such that the size of all the sets go from $\frac{n}{k}$ to $\frac{n}{k+1}$ in the next layer graph. To do this, $\frac{i}{k+1}\frac{n}{k}$ nodes have to be promoted from V_i to V_{i+1}. We let $p(i, k)$ and $s(i, k)$ denote the number of nodes that has to be promoted from set i in the layer graph of length k, and the number of nodes that stay respectively.

$$
\begin{aligned}
p(i,k) &= \frac{i}{k+1}\frac{n}{k} \\
s(i,k) &= \left(1 - \frac{i}{k+1}\right)\frac{n}{k} \\
&= \frac{k+1-i}{k+1}\frac{n}{k}
\end{aligned}
$$

With this, we can verify that all V_i contains $\frac{n}{k+1}$ nodes after going from the layer graph of size k to $k+1$.

$$\begin{aligned}
|V_i| &= s(i,k) + p(i-1,k) \\
&= \frac{k+1-i}{k+1}\frac{n}{k} + \frac{i-1}{k+1}\frac{n}{k} \\
&= \frac{k}{k+1}\frac{n}{k} \\
&= \frac{n}{k+1}
\end{aligned}$$

When going from a layer graph of length k to $k+1$, we are allowed to saturate edges from each node that stays in V_i to nodes in V_{i+1} that are promoted. That means we can have at most $s(i,k)p(i+1,k)$ edges that are saturated between two sets, and by extension at most $p(1,k) + s(k,k) + \sum_{i=1}^{k-1} s(i,k)p(i+1,k)$ augmenting paths in the layer graph of length k. In that expression, $p(1,k)$ corresponds to edges from s to nodes in V_1 that are promoted, and $s(k,k)$ corresponds to edges to t from nodes in V_k that are not promoted. To get the number of augmenting paths in all layer graphs, we sum k from 1 to n.

$$\begin{aligned}
Paths &= \sum_{k=1}^{n} p(1,k) + s(k,k) + \sum_{i=1}^{k-1} s(i,k)p(i+1,k) \\
&= \sum_{k=1}^{n} \frac{1}{k+1}\frac{n}{k} + \frac{k+1-k}{k+1}\frac{n}{k} + \sum_{i=1}^{k-1} \frac{k+1-i}{k+1}\frac{n}{k}\frac{i+1}{k+1}\frac{n}{k} \\
&= n\sum_{k=1}^{n} \frac{2}{k(k+1)} + n\sum_{i=1}^{k-1} \frac{(k+1-i)(i+1)}{k^2(k+1)^2} \\
&\leq n\sum_{k=1}^{n} \frac{2}{k(k+1)} + n(k-1)\frac{(k+1-k/2)(k/2+1)}{k^2(k+1)^2} \\
&= n\sum_{k=1}^{n} \frac{2}{k(k+1)} + n(k-1)\frac{(k+2)^2}{4k^2(k+1)^2} \\
&= O\left(n\sum_{k=1}^{n} \frac{2}{k(k+1)} + \frac{n}{k}\right) \\
&= O\left(n^2\sum_{k=1}^{n} \frac{1}{k}\right) \\
&= O\left(n^2 \log n\right)
\end{aligned}$$

Note that $(k+1-i)(i+1) = -i^2+ki+(k+1)$ quadratic function ai^2+bi+c with optimum in $i = -\frac{b}{2a} = \frac{k}{2}$. That is how the inner sum is removed.

Since our graph is fully connected, this is equal to $O(m \log n)$. What this means is that CD will not match the worst case bound of Dinic. The worst

case bound of $O(n^2m)$ came from a bound of $O(m)$ paths per layer graph, $O(n)$ layer graphs, and k time to send flow on a path of length k where $k = O(n)$. So, on this graph, we would expect Dinic to run in $O(nm \log n)$. We were not able to come up with a graph with $O(nm)$ augmenting paths, and it is possible that no such graph exists. If no such graphs exist for any m, it could be used as an improved analysis of the Edmonds and Karp, and Dinic algorithms.

13.5 AK

The AK graph generator takes a parameter 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.

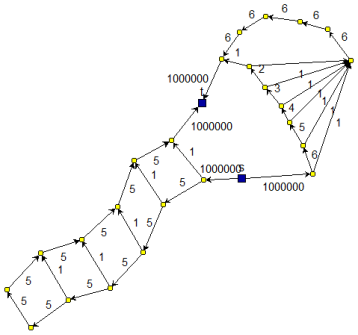

Figure 10: An example of the output of the AK generator, where $n = 26$.

As can be seen in Figure 10, 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 means that flow will be pushed back and forth in the graph while relabelling.

With the first global relabel heuristic, a lot of global relabels will occur due to the left pattern, even though flow is not sent around in a cycle. This is the reason we decided to implement a second global relabelling heuristic.

The right pattern contains very long paths of increasing length. This is particularly hard for the Dinic algorithm, since will get layer graphs with only one or two augmenting paths. It is also a place where the algorithms should benefit from dynamic trees, because you can send flow over the long path in $O(\log k)$ time instead of $O(k)$ time.

13.6 GenRmf

The previous graphs are designed to be very hard or very easy for some of the algorithms. The last two types of graphs will be more randomized, to better illustrate typical performance.

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 square 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.

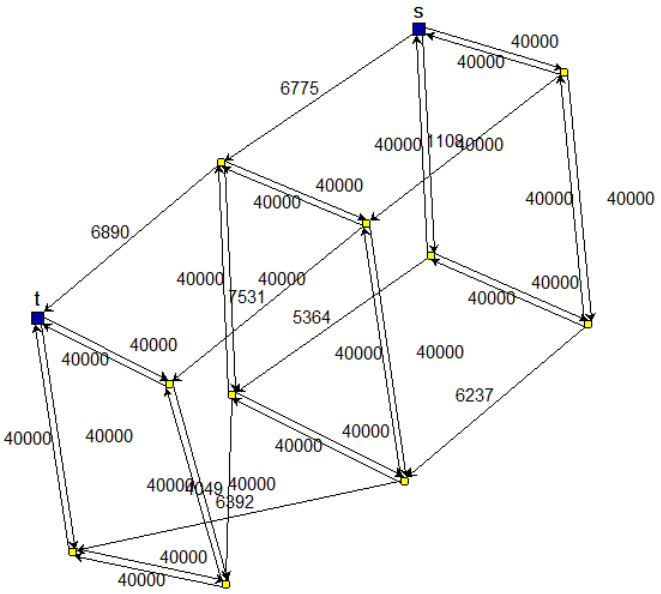

Figure 11: An example of the output of the GENRMF generator, where $a = 2$ and $b = 3$.

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 consequence of the construction, the min-cut will always be between two layers.

We will use the generator 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 $a = b$.

13.7 Washington

The Washington library is a collection of graph generators from DIMACS. 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 nodes in the last row has edges to the target.

We will use this to create two versions of the Wash graphs. A wide set of graphs that all have a constant 64 rows and a variable number of columns, and a long set of graphs with a constant 64 columns and a variable number of rows.

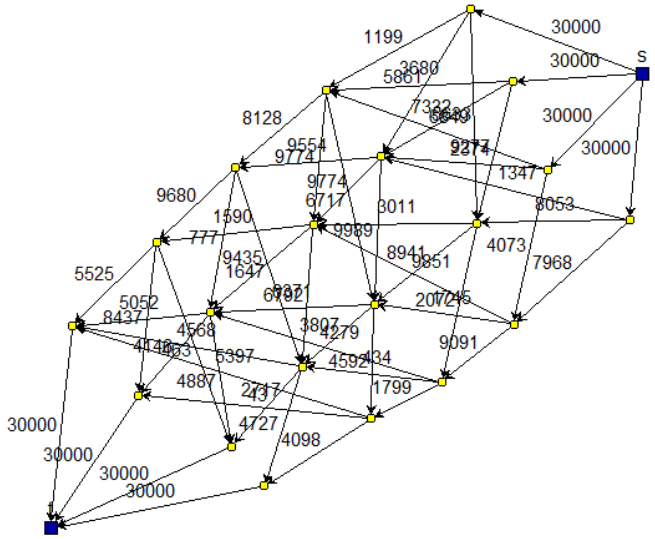

Figure 12: An example of the output of the washington level graph generator, with 5 rows and 4 columns

13.8 Test Environment

All tests were run on a Windows 7 64 bit PC with the hardware specified in Table 1.

Processor	Intel Core I7 950
Speed	3.07 GHz
Cores	4 physical, 8 logical
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 Memory	Corsair CMZ12GX3M3A1600C9
Capacity	12 GB (3x4GB)
Speed	DDR3 1600 (PC3 12800)
Latency	CAS9

Table 1: Hardware specifications

14 Results

In total, we have tested 25 algorithm implementations on 10 different types of graphs, yielding 250 different test runs. Each test was given 15 minutes to solve max flow problems of exponentially increasing size. In Appendix B, you will find charts of the results for each type of graphs.

We will be using the following abbreviations for our implementations of the algorithms. The algorithm by Edmonds and Karp, presented in Section 6, is referred to as EK. The Goldberg and Tarjan implementations presented in Section 8 are called GT. KR is short for the algorithms by King Rao that are presented in Section 9, and GR is used to describe the algorithm by Goldberg and Rao, presented in Section 10. To describe our implementation of the algorithm developed by Dinic that are presented in Section 7, we will just use Dinic. The implementations that use dynamic trees will be marked with a D. The versions of the King Rao algorithms that use the optimized memory modification will be marked as LM for Low Memory. The algorithms that use the global relabelling heuristics will be marked as GRC, GRP and GRN for Global Relabelling Cycle trigger, Global Relabelling Pass trigger and Global Relabelling Node count trigger, respectively. Finally, some algorithms are marked with Lib. These are algorithms that we have not implemented our selves, but taken from a c++ library to compare against our own algorithms. A table of all algorithms can be found in Appendix A.H.

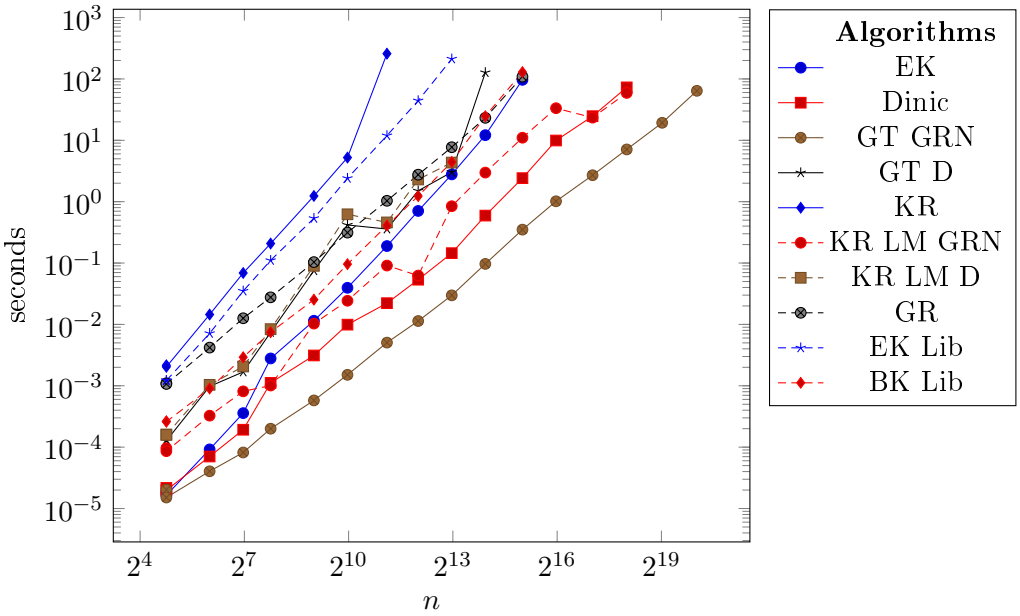

Figure 13: Best and worst results from the GenRmf square graphs

In the following sections we will discuss the performance of each algo-

rithm, but the general performance can be seen from Figure 13.

14.1 Choosing $f(G)$ for the GRN Heuristic

We implemented the GRN heuristic for the algorithms GT GRN, GT D GRN, KR LM GRN and KR LM D GRN. The reason we did not implement a KR D GRN algorithm is that previous tests have shown the KR D algorithm to have another bottleneck. This bottleneck is discussed in Section 14.5.

To find the best function, we tested each algorithm on graphs of the GenRmf and Wash types. On each size of graph in each family, we did an exponential search followed by a binary search for the best constant $f(G) = c$ for that particular graph and algorithm. We did not test on CRE, CD or AK because no flow will have to be pushed back to source in those graphs, so the optimal solution would be to only do one global relabelling at the start of the algorithm. We also did not test CRH because the construction of this graph is very special, and we believe that the GenRmf and Wash graphs are closer to a typical max flow problem. Additionally, the GRC heuristic is already optimal for the CRH graphs, because it only does two global relabellings; one at the start, and one right after the minimum cut has been saturated.

We ran each test three times, which yielded about 180 tests per algorithm. We plotted these points, and used a regression tool to find the function on the form $f(G) = an^b$ that best describes the data. This yielded the following equations

Algorithm	Function	R^2
GT GRN	$f(G) = 1.8956n^{0.6548}$	0.9442
GT D GRN	$f(G) = 0.54n^{0.7144}$	0.9128
KR LM GRN	$f(G) = 1.5834n^{0.7578}$	0.7824
KR LM D GRN	$f(G) = 1.6078n^{0.6509}$	0.6219

Charts of the data points and the regression lines can be seen in Appendix A.I.

14.2 Edmonds Karp

The Edmonds Karp algorithm generally does not perform very well. Dinic's algorithm is faster than Edmonds Karp, in all examples except for the AK graphs. This makes sense with respect to the worstcase time of $O(nm^2)$ for Edmonds Karp, and $O(n^2m)$ for Dinic. We found that the running time of the Edmonds Karp algorithm is best described as a function of 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 $O(nm^2)$.

Figure 14 shows the time for all measurements of Edmonds Karp, where the running time is divided by the product of the number of edges and the

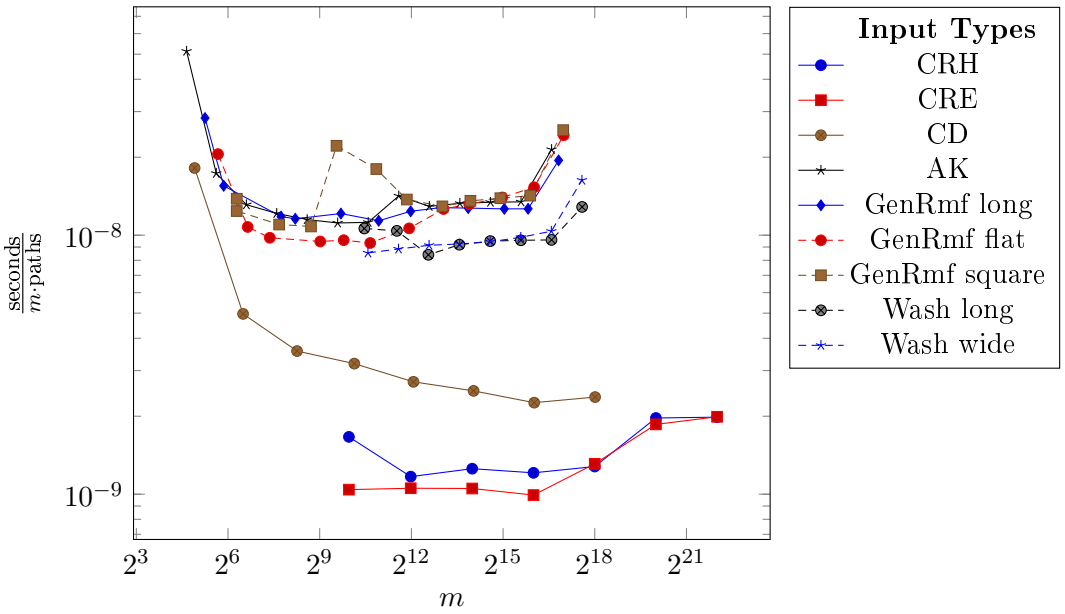

Figure 14: Edmonds and Karp performance per m and the number of augmenting paths

number of augmenting paths in the graph. Since this chart is mostly flat, $m \cdot \#\text{AugmentingPaths}$ is a good approximation of the running time. At $m = 2^{16}$, the algorithm starts to exit L3 cache, which is the reason for the increase in running time for larger graphs.

14.3 Dinic

Dinic's algorithm generally performs well. It generally beats the push relabel algorithms without heuristics, and many of the heuristics as well. It is not the fastest algorithm in any of our tests though. It especially has problems on the AK and CD graphs. The AK graphs has a low number of augmenting paths in each layer graph. That means that doing a BFS to compute the layer graph, and then a DFS to find the flow in it is slower than just doing a few BFS's to find each augmenting path in the layer graph. For this reason, we see EK perform better than Dinic on the AK graphs.

Our CRE and CRH graphs are easy for Dinic because there are never more than two layer graphs. The CD graphs are designed to be hard for Dinic as it has $\Omega(n)$ layer graphs, and as many paths as we could fit in each layer graph. We made the CD graph, because we were missing a fully connected graph that Dinic struggles with. The results here are better than expected. Dinic does not perform as well as the GT algorithms, but it is not far behind. The fact that Dinic was so efficient on CD graphs led us to conduct the previous analysis of the number of augmenting paths in the graph. The

observed number of augmenting paths found by Dinic fits well with the $m \log n$ bound achieved by the analysis. When plotting Paths/$m \log n$ all points lie on a horizontal line.

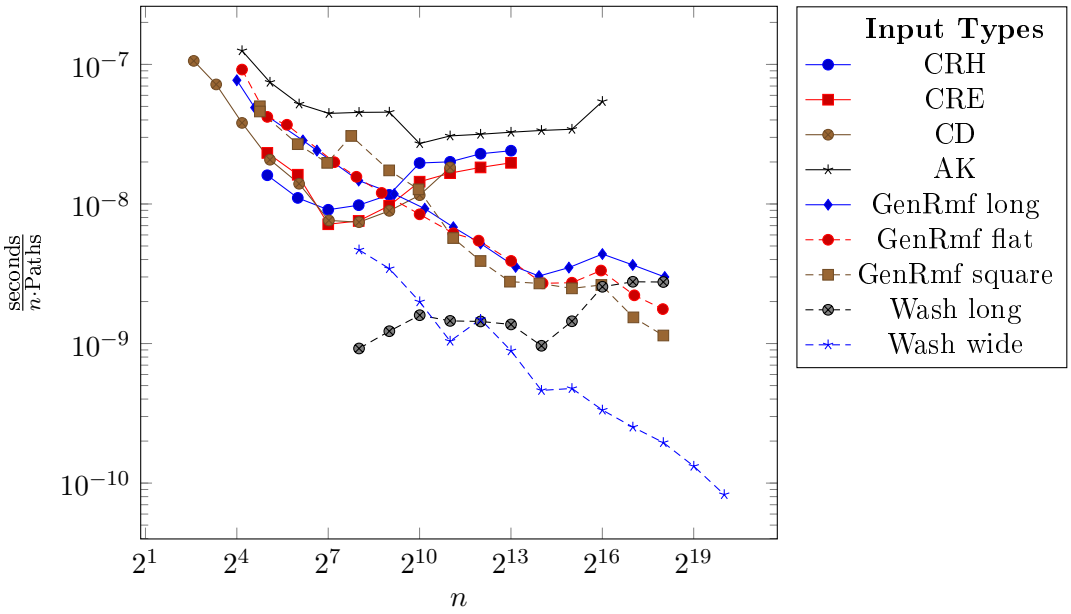

Figure 15: Dinic performance per nP

With regard to the running time, we found that the running time of Dinic was mostly proportional to n times the total number of paths the algorithm found. This can be seen on Figure 15. Here we divided the running time in each test by $n \cdot \text{Paths}$. Many of the graphs, such as the GenRmf and Wash graphs, seems to be faster than $n \cdot \text{Paths}$. We believe the problem is that not all paths are of length n. For many of the layer graphs, k is a lot smaller than n. In fact, as k approach n, the layer graph will not be able to contain as many paths as when k is significantly smaller than n. For example, if $k = n$, there can only be one augmenting path that contains all nodes. This means that n is an over estimation on the average length of the augmenting paths.

14.4 Goldberg Tarjan

The Goldberg Tarjan algorithms perform very well, especially with the heuristics. Before the heuristics, GT was the fastest on the CRE, CD and AK graphs. With heuristics, in all our tests, it is always some version of GT that is the fastest.

It makes good sense that GT is the fastest on the CRE, CD and AK graphs. CRE is basically the best case graph for GT, because it can ignore the majority of all the edges due to the fact that nodes never go above a label

of 2. The reason the maximum label is two is that a node is only relabelled 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 relabelled to 2, it will be able to send all its excess to a node with label 1.

The GRH graphs on the other hand is the worst case for this algorithm. All nodes will have to be relabelled to n before the algorithm can terminate. In-between the relabels, the algorithm will also spend time pushing the excess back and forth between nodes. That is why, without heuristics, the GT algorithm falls far behind Dinic on the CRH graphs.

The Goldberg Tarjan algorithm has an advantage in the CD graphs, because no flow will ever have to be pushed back to source. The way the CD 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 CD 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. Regardless of the label, the same number of pushes is required though.

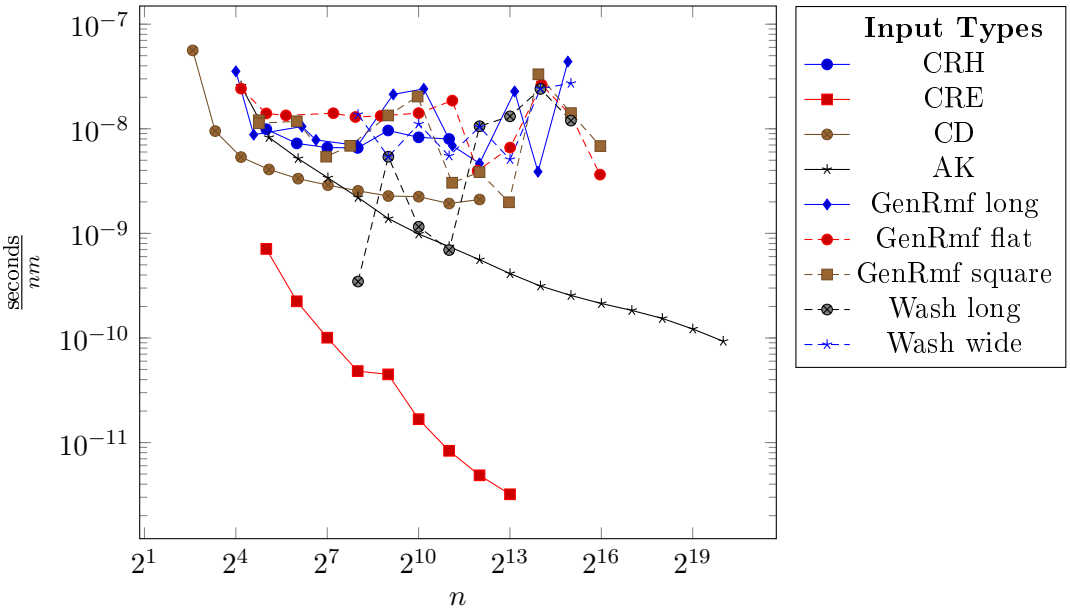

Figure 16: GT performance per nm

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 relabelling 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 all the excess is pushed to the target.

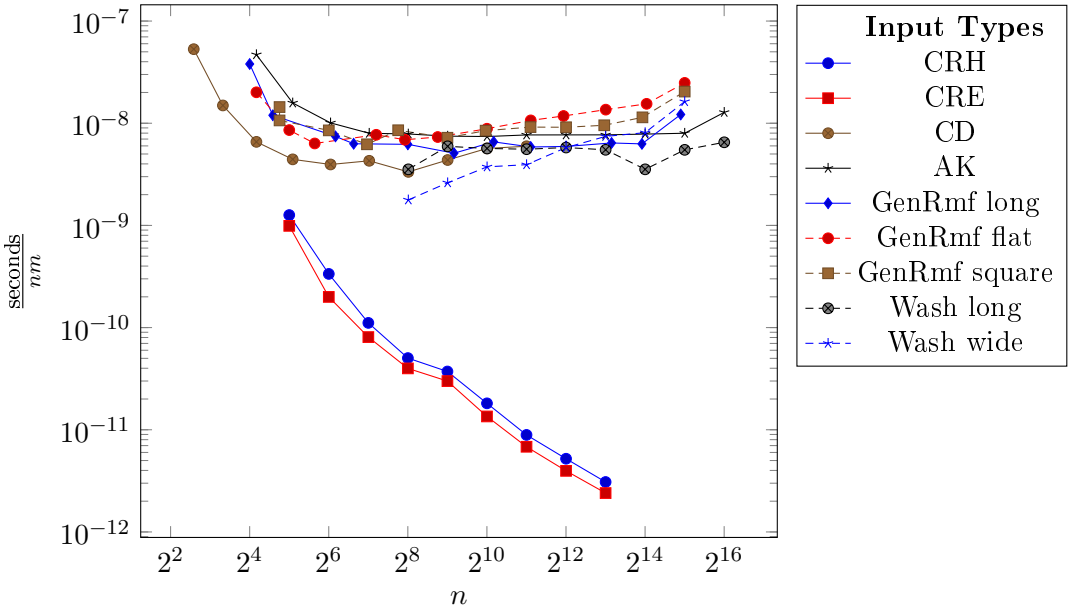

Figure 17: GT GRC performance per nm

It seems apparent that the running time of the GT 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 big jumps up and down in the results. This tells us that there is something other than n and m that has a big impact on the running time. As can be seen from Figure 16, the jumps comes from the GenRmf and Washington graphs. We found that the jumps are caused by the randomized capacities in the GenRmf and Washington. If we consider the minimum cut in such a graph, which is an (S, T) cut, then the problem is that the size of the set S can get very big. All nodes in S would have to be relabelled above n in order for the excess to be sent back to s. This means that if the size of S is big in one graph and small in another, even though the second graph is bigger than the first, the second graph could be solved faster.

This is the reason we decided to implement the global relabelling heuristics. As can be seen from Figure 17, the jumps disappeared when we implemented the GRC heuristic. This is because the algorithm no longer has to relabel the nodes in S all the way to n one step at the time.

Based on the data in Figure 16, it seems that the GT algorithm is a faster than nm for the AK and CRE graphs. CD and CRH graphs seems to be levelling 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 GT than the worst case bound of $O(n^3)$ in our test data.

The GT D algorithm has the same jumps as the GT algorithm. The effect

of dynamic trees are described in Section 14.7. Additionally, the benefits and drawbacks of the different heuristics are described in Section 14.8.

14.5 King Rao

The KR D, KR D GRC, KR D GRP algorithms are the worst algorithms 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 relabelled, a new node in the game will have to be loaded from main memory, or possibly even the hard disk. We do however not have any examples where the algorithm went to the disk, 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. We did not see any major improvements when using the GRC or GRP heuristics on this algorithm. As mentioned in Section 11, the problem is that we can not efficiently relabel a node multiple steps at a time since the data structure has to be updated for all the labels in-between. These three algorithms perform about the same in all of our tests. An example can be seen in Figure 19. Since there is no benefit from the other heuristics, we decided not to spend time on implementing the GRN heuristic for the KR D algorithm.

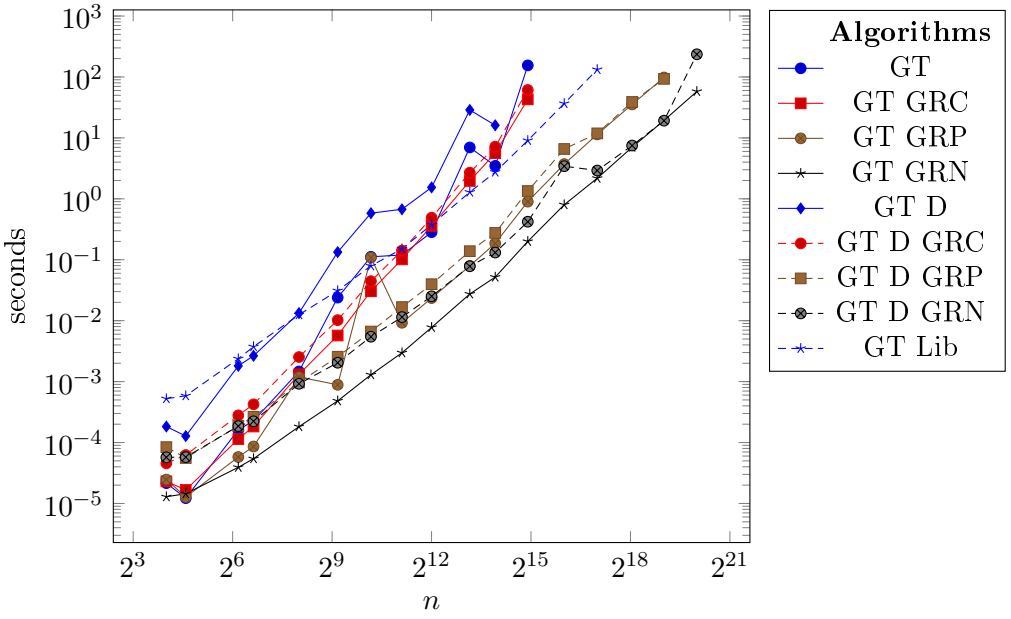

Figure 18: Goldberg and Tarjan results from the GenRmf long graphs

The LM versions behave similar to the corresponding GT algorithms, except that they are slower. A good example of this can be seen from com-

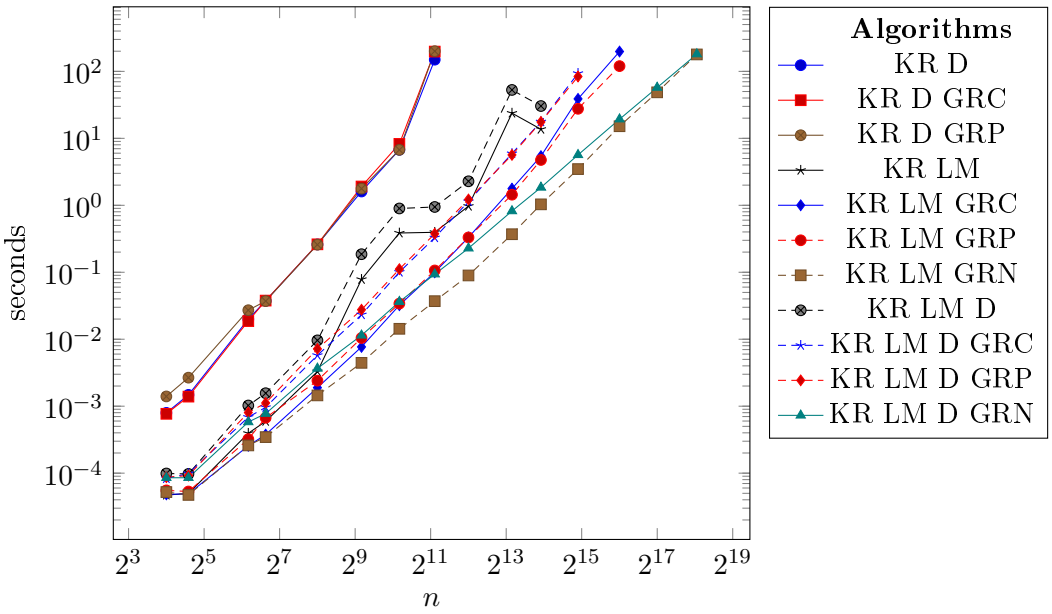

Figure 19: King and Rao results from the GenRmf long graphs

paring Figure 18 with Figure 19. If you look at KR LM and KR LM D in Figure 19 and compare the curves to GT and GT D in Figure 18, you see that the curves have the same ups and downs, but the KR algorihms are slower. The main difference between the GT algorithms and the KR LM 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. We have no example where the fastest KR algorithm beats the fastest GT algorithm, but KR LM D does beat GT D on GenRmf and Wash graphs.

When we first ran our different implementations of the King Rao algorithm, we observed big differences in running time between runs on the same graph, especially on the AK graphs. As an example, when we tested KR LM D GRP on an AK graph with $n = 8194$, we got results in the range from 0.21 seconds to 6.61 seconds. This was unexpected, since the algorithm is not randomized, and it is the exact same input. We managed to track the issue down to the initial sorting of the edges according to unsigned capacity. We used the implementation of quick sort from the c++ library, which is not a stable sort. If two edges have the same unsigned capacity, the position of them relative to each other is random. We managed to get consistent results by making sure that the comparator we use never returns 0. With this, we got the example above down to $0,06$ seconds. This shows that the order in which the edges are added has a very big impact on the running time. It affects in the order in which nodes a processed, because the act of

adding an edge modifies the amount of visible edges on the nodes next to the edge. In the AK graphs, many edges have the same capacity, so the the issue can have a very big effect on graphs of this type. The results displayed for the KR algorithms in the charts are results from after we made the sorting deterministic.

14.6 Goldberg Rao

The GR algorithm is slower than most of our other algorithms. The only algorithms that are consistently worse than GR are the KR algorithms without memory optimizations. On some graphs, GR does beat the KR LM and GT algorithms without heuristics on large inputs. We think that the reasons that GR is so slow compared to the other algorithm is mainly that it is doing a lot of different things, which means it has allocated a lot more memory than the others. All the memory is being used at some point in the main loop of the algorithm, so it will have to swap memory between caches more often than the other algorithms. Also, we have not spent much time on locating and removing bottlenecks in this algorithm or on tweaking the constants.

Tweaking the constants is something we think could make a big difference. We see big differences in the running time of GR on long and wide GenRmf graphs. On GenRmf graphs, the edges within each layer has capacities $10000a^2$, which is needed to route all flow that could be sent between layers. The first cut of the graph used to bound F in the algorithm is going to be made between s and the rest of the graph. This cut contains two edges with $10000a^2$ capacity and one edge with a random capacity up to 10000. So after the first iteration, $F = 20000a^2$, which influences Δ. On long GenRmf graphs, $b = a^2$, which means that a is and by extension the capacities on the edges inside the layers are small compared to capacities on edges between layers. This causes F and by extension Δ to be very small after the first iteration. The differences in capacity on edges inside layers and edges between layers is so small that after saturating an edge between each layer, all nodes become a single connected component. That means that in the majority of the iterations, the algorithm does not run the blocking flow algorithm, but just routes Δ flow inside the big connected component.

On the wide GenRmf graphs however, edges inside the layers have a very big capacity compared to the edges between the layers. This causes F and by extension Δ to be so big that only edges inside layers are going to be zero length arcs. The first layer graph where connected components are formed will be single list of super nodes.

In both cases, we can send about $5000a^2$ on average between the cuts. We are only allowed to send $\Delta = F/\Lambda = 20000a^2/\Lambda$ units of flow from s to t. Since GenRmf graphs are very sparse, we get $\Lambda = \sqrt{m} = O(\sqrt{n})$. So when $n > 16$, Δ becomes so small that we are not allowed to send $5000a^2$

units of flow, which means we wont be able to send the max flow in the first iterations.

If we experimented with other values of Δ, we could get the wide GenRmf graphs to solve the max flow problem in very few steps. In the wide GenRmf graphs, the layer graph will consist of a single path of super nodes, and the minimum cut will be between two of them. If Δ allowed this cut to be saturated, the algorithm would be finished.

We see a similar story in the wash graphs. In the long version, the cuts contain few edges which results in a low Δ compared to the capacity on the edges as opposed to the wide version where there is a lot of edges in the cuts. Also here, the long version with the relatively low Δ is faster. After flow has been sent on a path $(s, v_1, \ldots, v_k, t)$ in the long version, a connected component can form from the smallest i and largest j such that there is an alternate path from v_i to v_j. Any other alternate paths from v_k to v_l where $i \leq k < l \leq j$ would also be part of that connected component. As soon as two connected components share a node, they become one connected component. This results in very quickly getting the entire graph into one connected component.

What takes time depends heavily on how Δ is in relation to the capacities on the edges, which affects how strongly connected components are formed. In graphs that almost only consist of strongly connected components, we naturally see almost no time spent on the blocking flow algorithm. Vice versa, if the graph has few strongly connected components, a lot of the time is spent on the blocking flow algorithm. If we compare GenRmf long, wide and square graphs of the same size, then GR is fastest on the long graphs and slowest on the wide graphs with the square ones in between. We would like to have some input graphs where we can vary the number and size of connected components without changing the result of the algorithm or the resulting flow graph. That would give us a more clear image of the trade off between spending time on connected components and spending time on the blocking flow algorithm. Based on the results we have now, it would seem that bigger is better in terms of strongly connected components.

14.7 Dynamic Trees

Dynamic trees tend to make the running time op the algorithms slightly worse. It does not change the running time as much as heuristics can, but it does make it slower. A good example is Figure 20, where we see every dynamic version of GT being slightly slower than its non dynamic counterpart.

An exception to the rule can be seen in Figure 21 which depicts GT results for AK graphs. Here, the dynamic tree version of the algorithms are slower, but coupled with the GRN heuristic, it is the fastest algorithm. The big jump in the running time of the GT D GRN algorithm is due to an overflow error in the code. We expect the GT D GRN algorithm to behave

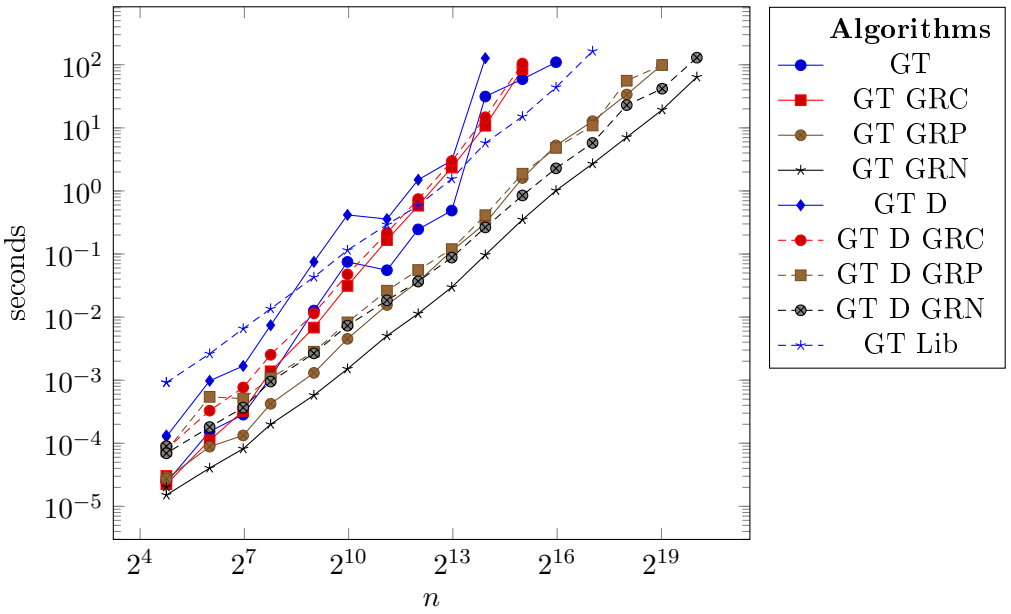

Figure 20: Goldberg and Tarjan results from the GenRmf square graphs

like the KR LM D GRN algorithm in Figure 22 if the error is corrected. We find that this speed-up is especially interesting, since adding dynamic trees or heuristics on their own makes the algorithm slower, but doing both gives a big speed-up.

It is worth noting that we see a similar result for the KR algorithms on the AK graphs in Figure 22. Here KR LM D GRN is significantly faster than all others. The reason the KR LM D GRP algorithm is slow is explained in Section 14.8.2.

We did expect dynamic trees to perform well on the AK graphs, since they feature a very long path that has to be pushed on often. With dynamic trees, this can be done in logarithmic time instead of linear time. In the other part of the graph however, without global relabelling, it will often push on very short paths which would be more efficient to just do directly. With global relabelling, we avoid pushing excess back and forth on these small paths. In fact, it creates a longer and longer path that is repeatability pushed on as the algorithm progresses. That is why the algorithm is so fast with heuristics and dynamic trees, but slow with just one of them. The reason why GT D GRC is performing bad is due to the heuristic trigger, and will be explained in Section 14.8.

Our conclusion with regard to dynamic tress is that algorithms only benefit from dynamic trees if the tree pushes mainly push on the same paths repeatability, and the paths are very long. This generally does not occur, and we only found it in the AK graphs, which are very artificially con-

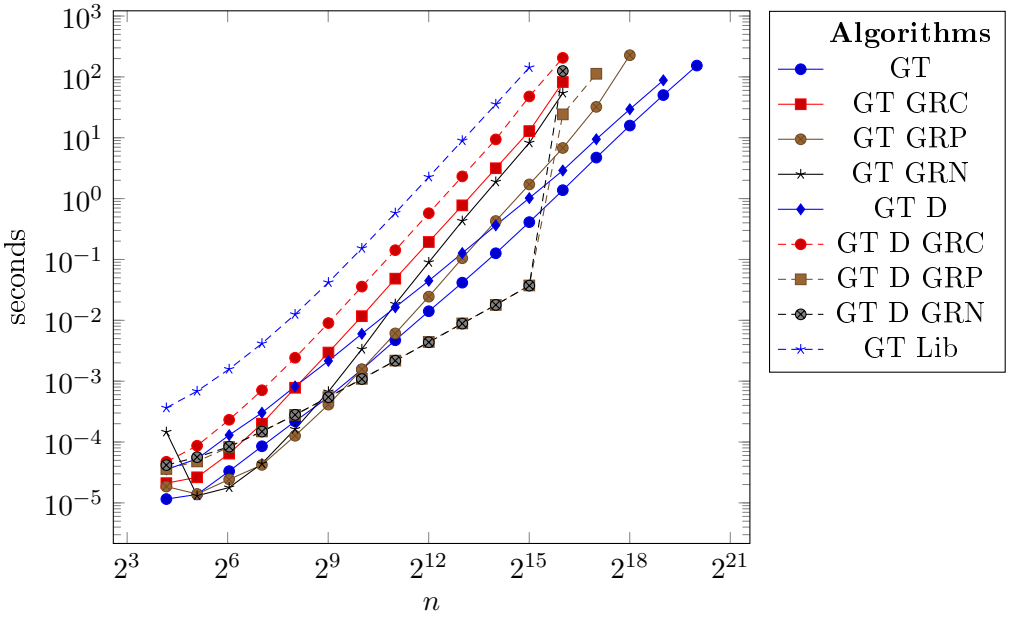

Figure 21: Goldberg and Tarjan results from the AK graphs

structed. On our more randomized graphs like GenRmf and Wash, we saw no improvements by using dynamic trees.

14.8 Global Relabelling

All of our heuristics perform the same basic global relabelling, but the triggers that determine when to run the global relabelling are different. Regardless of the trigger, we always run a global relabelling at the start of the algorithm.

All of our heuristics perform worse than no heuristics on the CD graphs. This is because it is never needed to send flow back to the source in the CD graph. None of the global relabels runs relabel any nodes significantly up. The AK graphs also do not require any excess to be routed back, but here GT D GRP and GT D GRN are faster than all other GT algorithms. Likewise, KR LM D GRN is faster than other KR algorithms. This can be seen from Figure 21 and Figure 22. The left side of the AK graphs as depicted in Figure 10 from Section 13.5 still cause flow to be pushed back and forth if no global relabelling is done however.

14.8.1 GRC

The first heuristic we implemented was the GRC heuristic. It does a global relabelling whenever a node is relabelled more than one label up. It performs

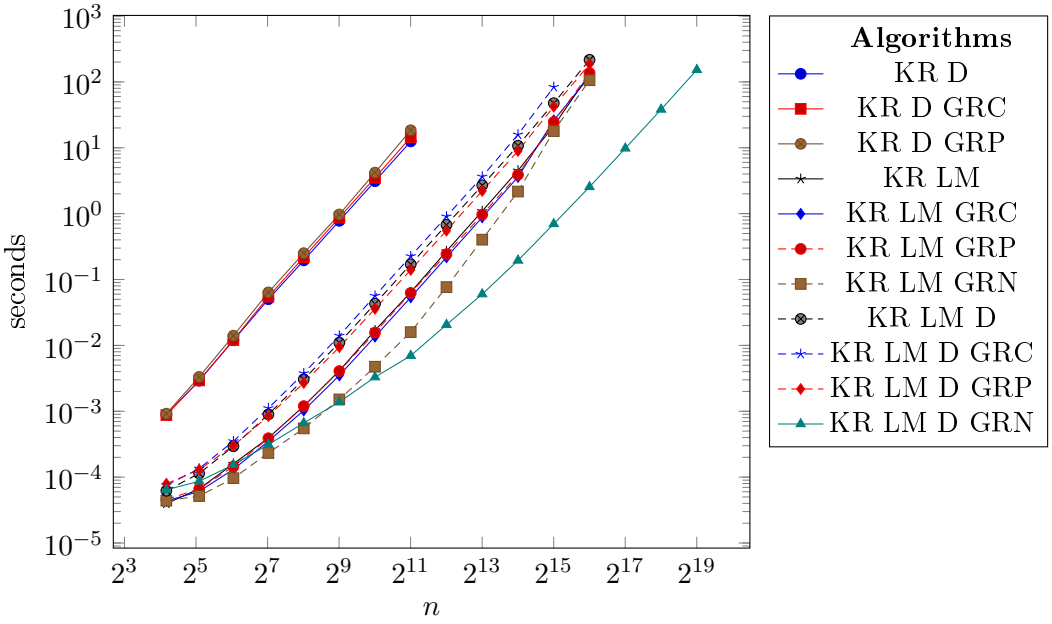

Figure 22: King and Rao results from the AK graphs

very well on the CRH graphs, but not very well on AK graphs. On GenRmf and Wash graphs, they are more stable than not using heuristics, and generally slightly faster. The reason we implemented the heuristic was that the running time of the algorithms was very unstable in the GenRmf and Wash graphs, so we did expect the heuristic to be more stable and faster on those graphs. Examples of the instability in the algorithms without global relabelling can be seen in Figure 18 and Figure 19.

CRH graphs are the best case graphs for this particular trigger. As soon as the last edge to t is saturated, that node will have to be relabelled twice since all other nodes have a higher label than it does. That triggers the global relabel, and the excess is then sent to s. If you consider the AK graphs however, the square pattern on the left side as depicted in Figure 10 from Section 13.5 causes the heuristic to trigger very often. After the first global relabel, all nodes is labelled according to their distance to t. Excess is then pushed from the first node in the lower part of the left side to the top part of the left side and on to t. For the algorithm to continue here, excess must be pushed to the next node in the lower part of the left side. This requires that the first node in the lower part is relabelled twice, even though nothing has been pushed in a cycle. The global relabel is not going to change any labels on the left side, so it is just wasted effort. This happens again and again each time flow is pushed further along the lower part of the left side. The performance on CRH and AK graphs can be seen in Appendix B.

This triggering of the heuristic when there are no cycles is why we decided

to implement the GRP trigger.

14.8.2 GRP

The GRP trigger triggers when flow has not been sent to the target since the last pass, if flow has been sent to the target since the last global relabelling. This gave a major speed-up for the GT algorithms on GenRmf and Wash graphs. This can for instance be seen in Figure 20. As discussed in Section 14.7, GT D GRP gained major speed-ups on the AK graphs as well.

As can be seen from Figure 19, we did not see any major speed-ups for the KR algorithms when using the GRP heuristic instead of the GRC heuristic. Due to the way edges are added in the KR algorithm, there are not many active nodes at a time in the KR algorithm. When an edge (u, v) is added, it might cause the visible excess of u become positive and activate the node. This causes the excess to be pushed around the graph, which might activate other nodes. The number of nodes that are active at the same time remain low though, which means the global relabelling check is called very often. A result of this is that the KR algorithms with the GRP heuristic perform a global relabel every time the excess of t changes.

To get a more consistent speed-up, we decided to base the trigger off the number of nodes processed instead of passes, which lead to the GRN heuristic.

14.8.3 GRN

Instead of checking the excess of t after each pass, the GRN heuristic does it after $f(G)$ nodes have been processed. This improved upon the running times for the GT and KR algorithms on the AK, GenRmf and Wash graphs. For examples of this, see Figures 18, 19, 20, 21, 22, or Appendix B.

14.9 Library Implementations

We tested a set of max flow algorithms from the C++ Boost Library [DAR13]. It has implementations for the Edmonds and Karp algorithm, the Goldberg and Tarjan algorithm, and an algorithm by Boykov and Kolmogorov. The Boykov and Kolmogorov algorithm is an algorithm that is designed to be efficient on a special type of max flow graphs that arise from computer vision.

We tested the algorithms on all of our graphs, and found that the library algorithms are relatively slow. The library implementation of the Edmonds and Karp algorithm is almost 100 times slower than our implementation, even though we have not done any significant optimizations to it. The implementation of the Goldberg and Tarjan algorithm is also slow compared to our implementations. If you look at Figure 20, you can see that GT Lib is about the same time as our implementations without heuristics or with the GRC heuristic. It is slower for small graphs, and faster for larger graphs.

The slope is about the same as our algorithms with the GRN heuristic, but it is again about 100 times slower. The slope and lack of jumps tell us that they do use some form of heuristics to detect when the min cut is saturated. We have profiled it to ensure that it is not something in our code that is causing the slowdown, but all the work is done in functions inside the boost library.

As for the BK Lib algorithm, if we compare it to GT Lib, then BK Lib is faster on small graphs, and GT tends to win as the graphs become bigger. It is still far behind our best algorithms on all graphs.

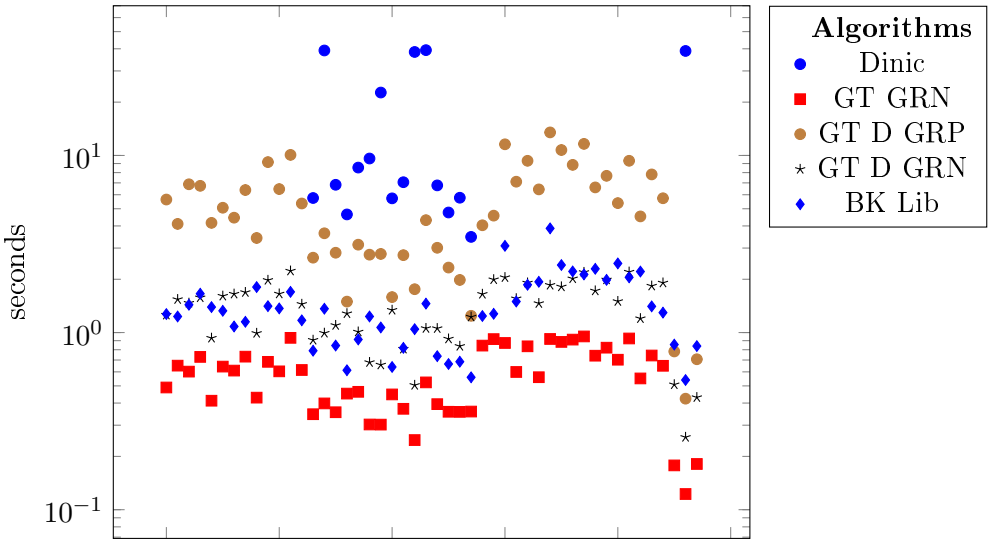

Figure 23: Results from the Computer Vision graphs

To get some graphs that BK Lib should work well on, we found some max flow graphs that are generated from computer vision problems [atUoWO12]. The graphs are rather big, so we have not been able to run all of our algorithms on them, but GT GRN, GT D GRP, GT D GRN and BK Lib all perform well enough for us to get data on them within reasonable time. The data for the CV graphs can be seen in Figure 23. When we ran the tests, we ran each algorithms on the graphs for 15 minutes. The order the graphs were tested was sorted by file size on disk, but in in Figure 23 the points are alphabetically sorted. From left to right, the chart contains data for BVZ-sawtooth number 4, 6-16 and 18, BVZ-tsukuba number 0-15, BVZ-venus number 1-17, and the graphs KZ2-sawtooth17, KZ2-tsukuba13 and KZ2-venus19.

As expected, the BK Lib algorithm performs quite well on these graphs. It is not fast enough to beat GT GRN, but it is the second fastest algorithm we have tested.

15 Future Works

If we had more time, we would have liked to spend more time optimizing our algorithms. We believe that there is still room for improvements in experimenting with some of the parameters in the algorithms.

Also, better algorithms for routing the blocking flow in Dinic might speed up this algorithm. A problem with Dinic right now is that it might process the same part of the layer graph multiple times. It might be possible to speed it up by saving information about the paths in the layer graph that has already been visited once. One way of doing this could be using dynamic trees, but we have not seen a big improvement for the other algorithms when using dynamic trees. That having been said, layer graphs does tend to be long paths with reusable edges, so we could see a good performance boost by adding dynamic trees. Alternatively, we could use the push pull block algorithm that is used in Goldberg Rao to compute the blocking flow, which would also reduce the theoretical running time to $O\left(nm \log \frac{n^2}{m}\right)$.

As mentioned in Section 12, we could probably gain some performance by using templates for custom data in nodes instead of a pointer to an object containing the custom data. A graph node takes up 24 bytes, and an edge takes 32 bytes. With a cache line of 64 bytes, this means we load about two nodes with every cache miss. The probability that we need the other node is low however as proximity in memory does not imply proximity in the graph. Even if the first node has an edge to the other node, it probably has edges to other nodes as well. In the table below we listed the sizes for some of the objects in our algorithms.

Object	Size	Padding
Node	24 bytes	4 bytes
Edge	32 bytes	4 bytes
EK Node Tag	16 bytes	0 bytes
Dinic Node Tag	24 bytes	0 bytes
GT Node Tag	24 bytes	4 bytes
GT D Node Tag	32 bytes	4 bytes
KR LM Node Tag	128 bytes	4 bytes
Dynamic Tree Node	80 bytes	0 bytes

Variables are aligned to start on an address that is a multiple of their size. For instance, a 4 byte variable can start on address $4, 8, 12$ etc. Variables are aligned by padding the previous variable. For instance, if we define a 32 bit variable followed by a 64 bit variable, the 32 bit variable will be padded to 64 bit so the 64 bit variable starts at a multiple of 8. To allow putting structs next to each other in an array, the struct is also padded so that its size is a multiple of the size of the biggest primitive in it. For instance, we use 64 bit pointers, which means that any struct that contains a pointer is padded

so that its size in bytes is a multiple of 8. Padding to align variables can generally be avoided by defining them in order of biggest to smallest. The size column specifies the total size of the object, and the padding column specifies how much of that is padding. This means that if we were to merge two objects each of size 24 bytes with 4 bytes of padding, we would be able to merge those to an object of size 40 bytes with no padding. Additionally, merging a tag with the node removes the need for the pointer in the node to the tag.

What is interesting about this table is that node tags from all algorithms except from KR can be merged with the graph node into an object that fits into one cache line. In fact, merging GT results in an object that takes up 24 bytes from the node, 24 bytes from the GT tag, minus 8 bytes from padding, and minus 8 bytes for one less pointer in the node. In total, 32 bytes. That means that two nodes would fit perfectly into one cache line. We went up to 64 bit pointers due to memory issues in KR, but with 32 bit pointers, a merged graph node and GT D node tag would take up 28 bytes. This would also allow for GT D to place two merged nodes in one cache line. Since 28 bytes is not a perfect half cache line though, 1/4 of the nodes will be placed to span two cache lines. It might be worth it to pad these nodes with an extra 4 bytes so that no nodes span two cache lines.

The KR tags take up a lot of memory because they need to contain the game in them, which means a lot of linked lists. This means that we would not be able to read an entire node in one cache line, but we would still expect to see a performance increase. The reason is that the CPU could load the information it needs directly instead of having to load a pointer and then load the information it needs.

As mentioned before, the downside of this optimization is that the graph needs to be cloned. For graphs such as CRE and CRH, that is probably not worth it due to the large number of edges that do not need to be used, but we would expect to see a speed up on our other graphs.

We have not had much time to optimize the Goldberg Rao algorithm, so more work on that could bring its running time down.

Finally, it would be interesting to implement Orlin and compare it to the other algorithms. We don't expect it will be faster though, since it adds a lot of complexity and memory use.

16 Conclusion

Although we did not get to implementing the Orlin algorithm [Orl13] as we had hoped, we did get some interesting results. Our contribution to the algorithm by King and Rao is not only a theoretical result, but it also makes a big difference in practice. The addition to the Goldberg Rao theory fixes an issue it has with one of its assumptions, and make the theory hold.

On the performance side of things, we can conclude that a naive implementation of the algorithm by Dinic is often better than a naive implementation of the other algorithms. However, we found more room for optimizations and heuristics on the Push-Relabel algorithms such as the algorithms by Goldberg and Tarjan, and by King and Rao. We did not find the added complexity of the King Rao algorithms in relation to the Goldberg Tarjan algorithms to be worth it however. Likewise, the algorithms do for the most part not benefit from dynamic trees. It appears as though the Goldberg Rao algorithm has too many different things going on for it to be really efficient, but there is room for optimizations in our implementation.

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.

[atUoWO12] Computer Vision Research Group at the University of Western Ontario. Max-flow problem instances in vision, http://vision.csd.uwo.ca/maxflow-data, December 2012.

[CG97] B. V. Cherkassky and A. V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.

[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.

[DAR13] Beman Dawes, David Abrahams, and Rene Rivera. Boost c++ libraries v1.55.0, http://www.boost.org/, November 2013.

[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.

[Eis13] David Eisenstat. dtree: dynamic trees a la carte (c++), http://www.davideisenstat.com/dtree/, October 2013.

[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.

[GT90] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by successive approximation. *Mathematics of Operations Research*, 15(3):pp. 430–466, 1990.

[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.

[HT07] Bernhard Haeupler and Robert Endre Tarjan. Finding a feasible flow in a strongly connected network. *CoRR*, abs/0711.2710, 2007.

[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.

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[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 8: A. V. Goldberg R. E. Tarjan 1988

Symbol	Short Description
Eligible	An edge (v, w) is eligible if it has $r(v, w) > 0$
Active	A node v is active if $e(v) > 0$
Current-edge	The edge the algorithm would try to push on next. Also the edge linked in the Dynamic Tree data structure
Saturating push	A push operation on an edge that used all the residual capacity
Pass	Pass $i + 1$ consist of working on all the nodes that was put in the queue at pass i
Large	A dynamic tree T is large if $\|T\| \geq k/2$

Appendix A.D 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.E 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.F 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.

Appendix A.G Section 10: A. V. Goldberg S. Rao 1998

Symbol	Short Description
$\Lambda = \min\{n^{\frac{2}{3}}, m^{\frac{1}{2}}\}$	
$\Delta = \frac{F}{\Lambda}$	Bounds which edges are admissible
$d(v)$	The distance label of a node v
$l(v,w)$	Length function. Zero iff $r(v,w) \geq 2\Delta$
Special edge (v,w)	$\Delta \leq r(v,w) \leq 2\Delta, d(v) = d(w), r(w,v) \geq 2\Delta$
$\bar{l}(v,w)$	Length function. Zero if special edge or $r(v,w) \geq 2\Delta$
Admissible	An edge (v,w) is admissible if $d(v) = \bar{l}(v,w) + d(w)$ and $r(v,w) > 0$
F	Upper bound on the remaining flow that can be sent. It is updated to capacities of cuts in the graph.
Iteration	One iteration is one sequence of constructing a layer graph, running the blocking flow algorithm and routing flow in SSCs.
Phase	One phase is a series of iterations resulting in an update to F

Appendix A.H Algorithm Abbriviations

Abbriviation	Dynamic Trees	Low Memory	Global Relabeling Trigger	Library
Edmonds and Karp				
EK				
EK Lib				×
Dinic				
Dinic				
Goldberg and Tarjan				
GT				
GT GRC			Cycle	
GT GRP			Pass	
GT GRN			Node Count	
GT D	×			
GT D GRC	×		Cycle	
GT D GRP	×		Pass	
GT D GRN	×		Node Count	
GT Lib				×
King and Rao				
KR D	×			
KR D GRC	×		Cycle	
KR D GRP	×		Pass	
KR LM		×		
KR LM GRC		×	Cycle	
KR LM GRP		×	Pass	
KR LM GRN		×	Node Count	
KR LM D	×	×		
KR LM D GRC	×	×	Cycle	
KR LM D GRP	×	×	Pass	
KR LM D GRN	×	×	Node Count	
Goldberg and Rao				
GR	×			
Boykov and Kolmogorov				
BK Lib				×

Appendix A.I Global Relabeling Node count trigger functions

Algorithm	Function	R^2
GT GRN	$f(G) = 2.018n^{0.6488}$	0.946
GT D GRN	$f(G) = 0.5281n^{0.7145}$	0.9325
KR LM GRN	$f(G) = 2.1674n^{0.7244}$	0.7766
KR LM D GRN	$f(G) = 11.649n^{0.4249}$	0.1684

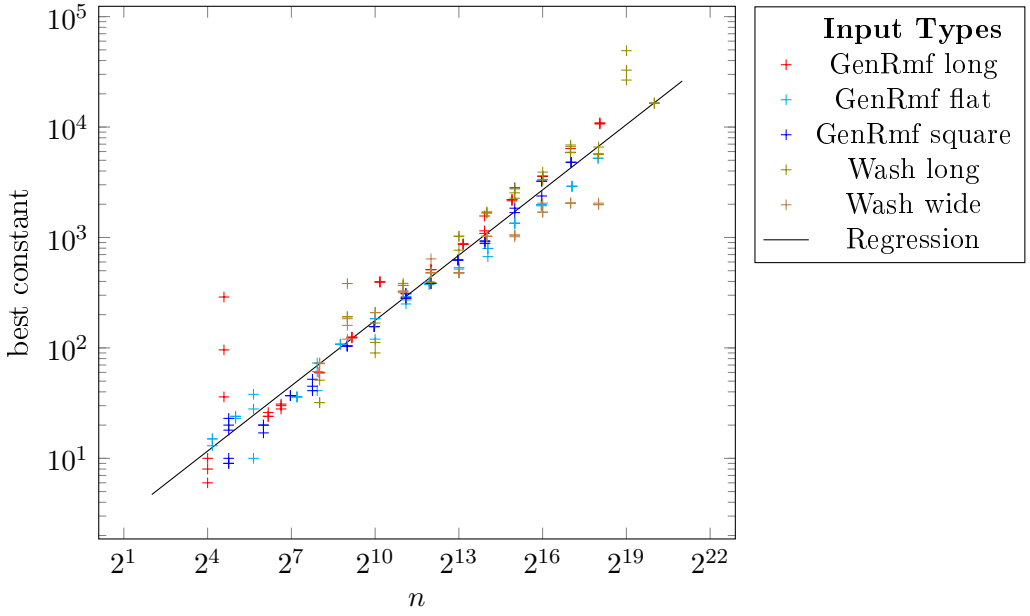

Figure 24: Constant estimation for Global Relabel with node count trigger for Goldberg Tarjan (GT GRN)

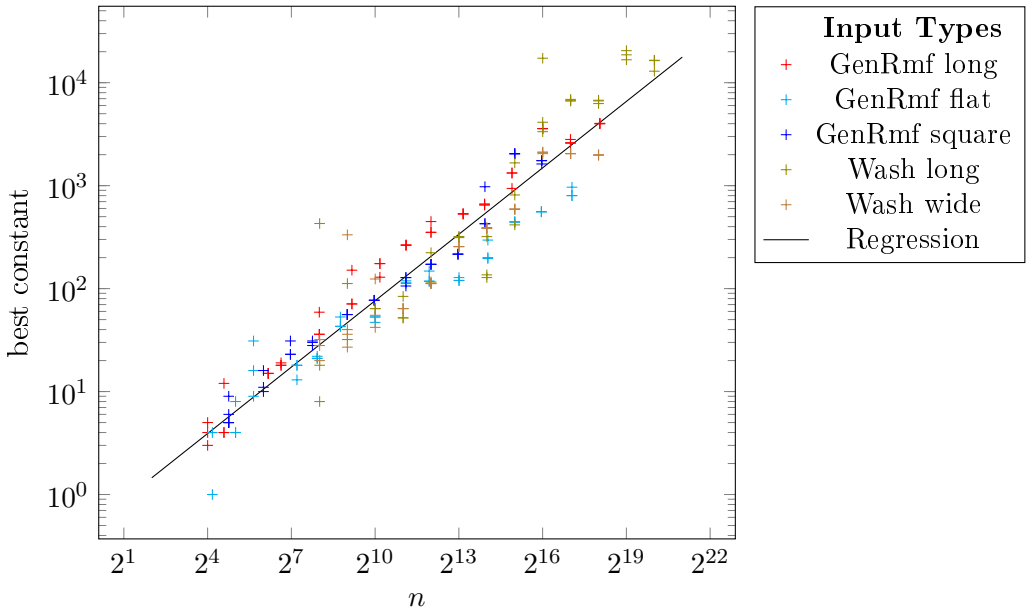

Figure 25: Constant estimation for Global Relabel with node count trigger for Goldberg Tarjan Dynamic (GT D GRN)

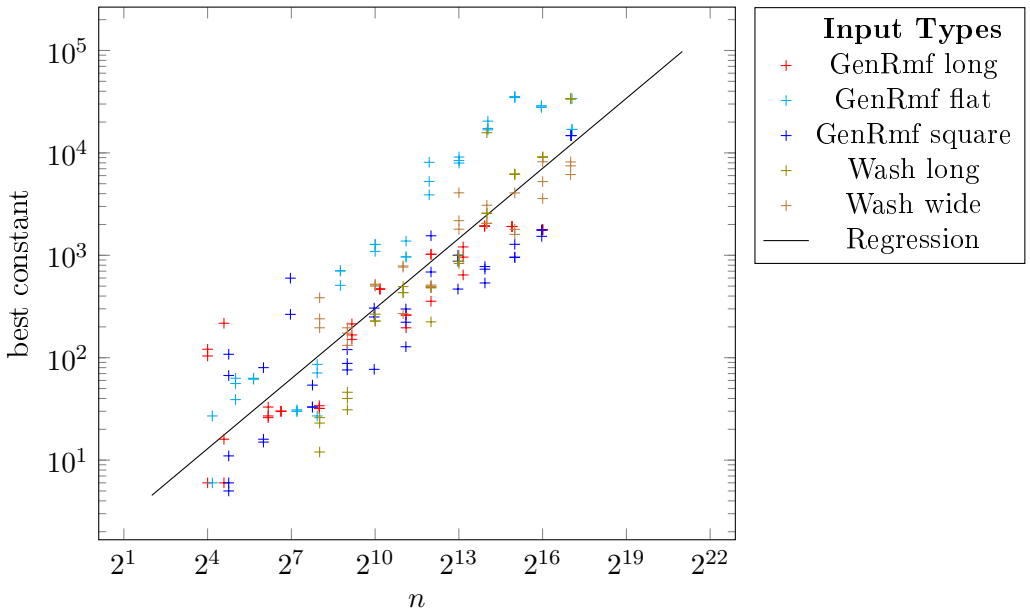

Figure 26: Constant estimation for Global Relabel with node count trigger for King Rao Low Memory (GT LM GRN)

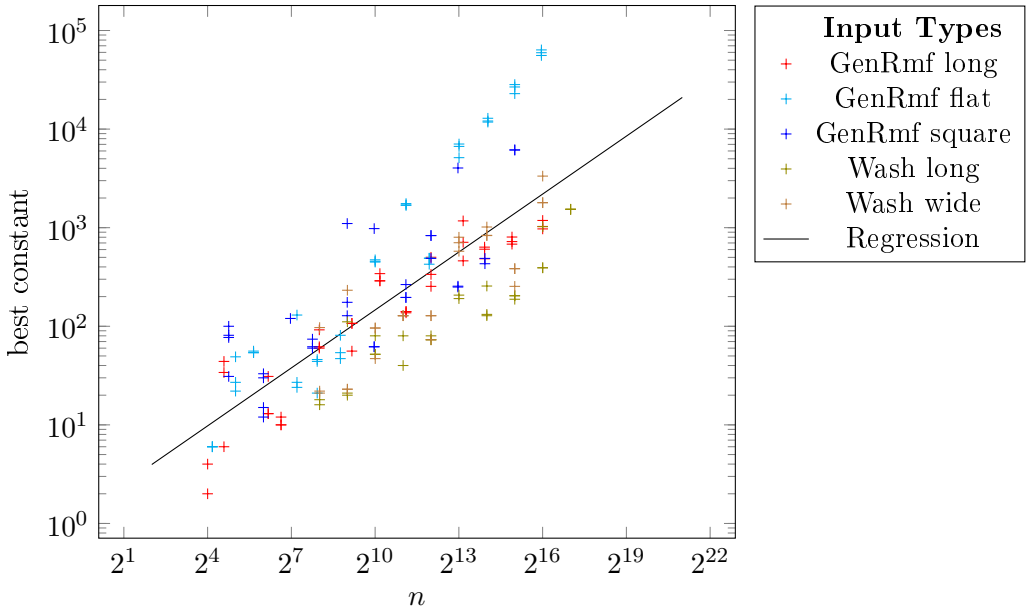

Figure 27: Constant estimation for Global Relabel with node count trigger for King Rao Low Memory Dynamic (KR LM D GRN)

B Charts

Appendix B.A Results from CRH graphs

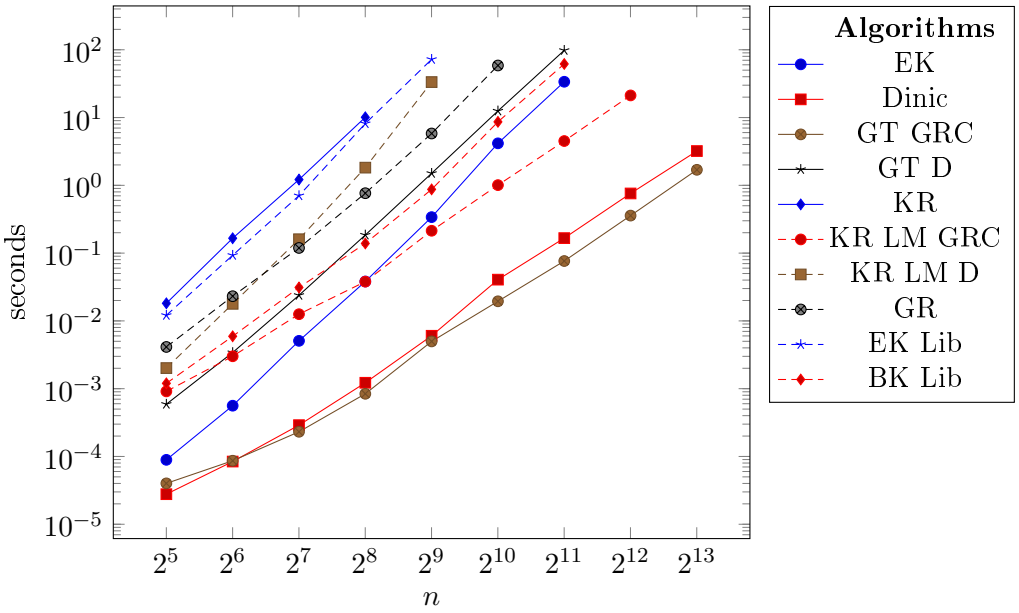

Figure 28: Best and worst results from the CRH graphs

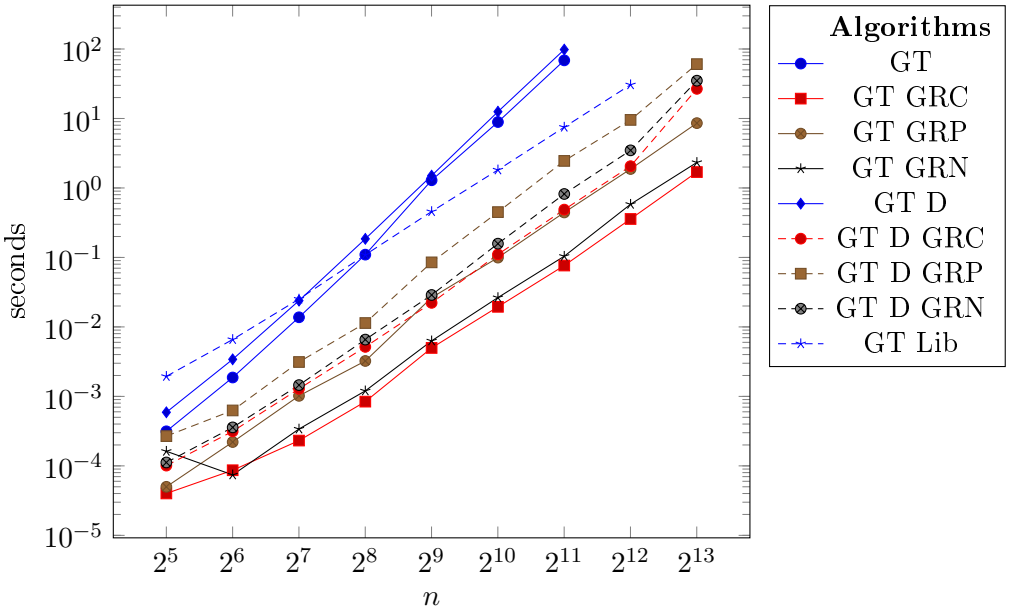

Figure 29: Goldberg and Tarjan results from the CRH graphs

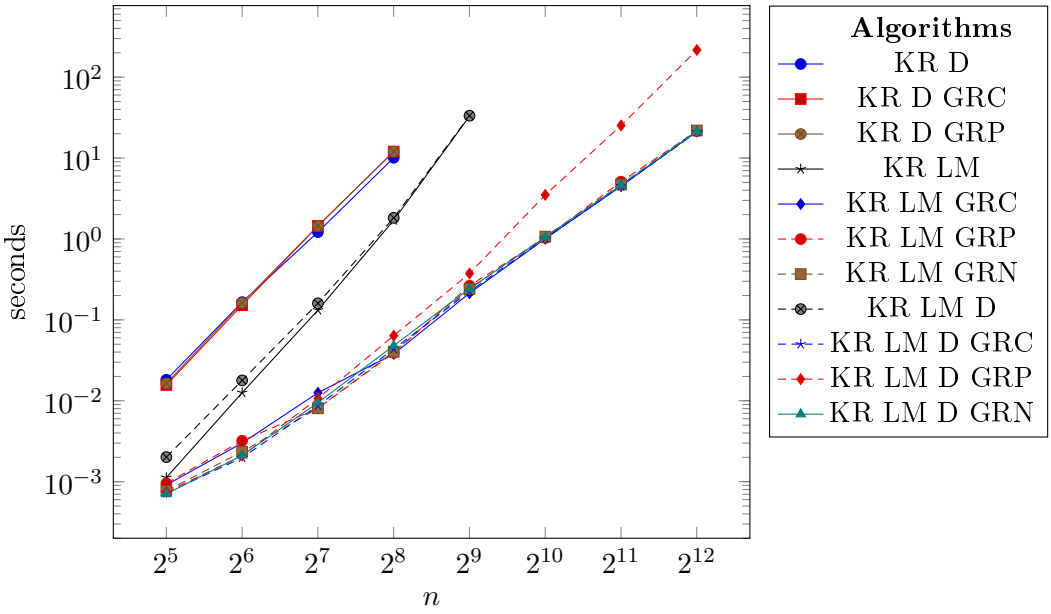

Figure 30: King and Rao results from the CRH graphs

Appendix B.B Results from CRE graphs

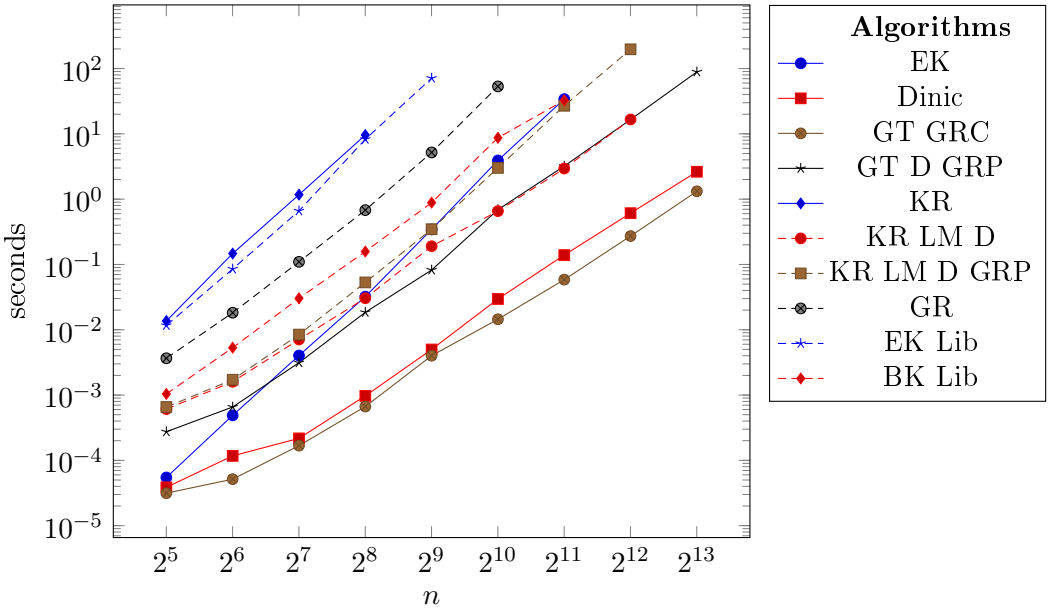

Figure 31: Best and worst results from the CRE graphs

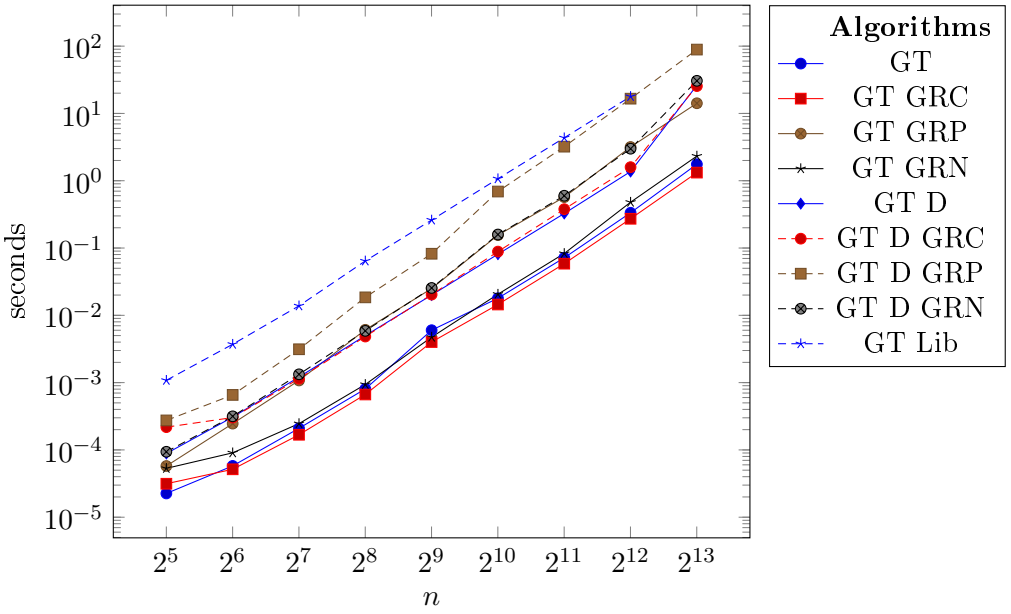

Figure 32: Goldberg and Tarjan results from the CRE graphs

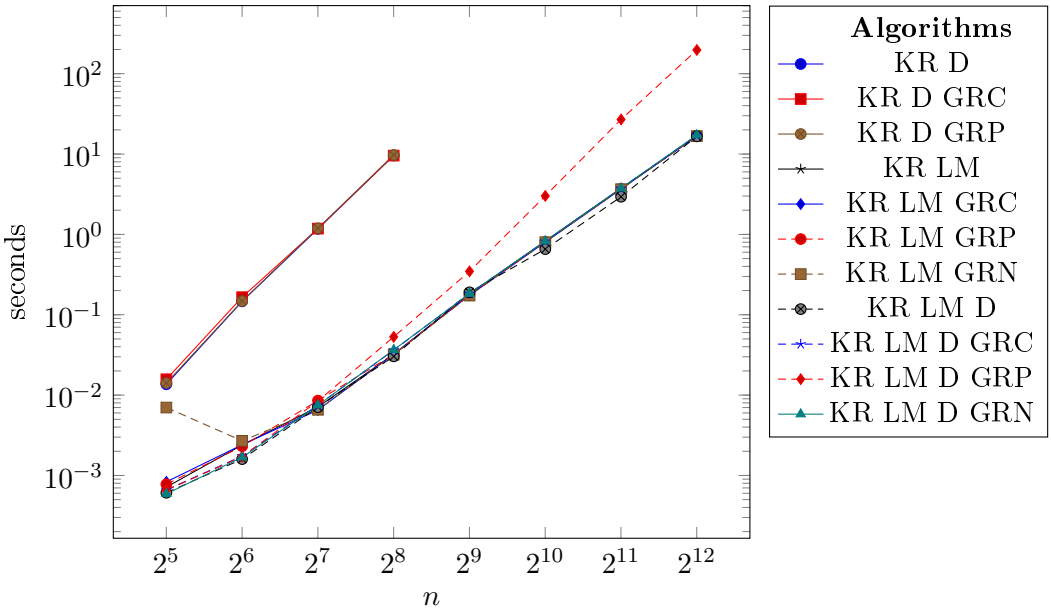

Figure 33: King and Rao results from the CRE graphs

Appendix B.C Results from CD graphs

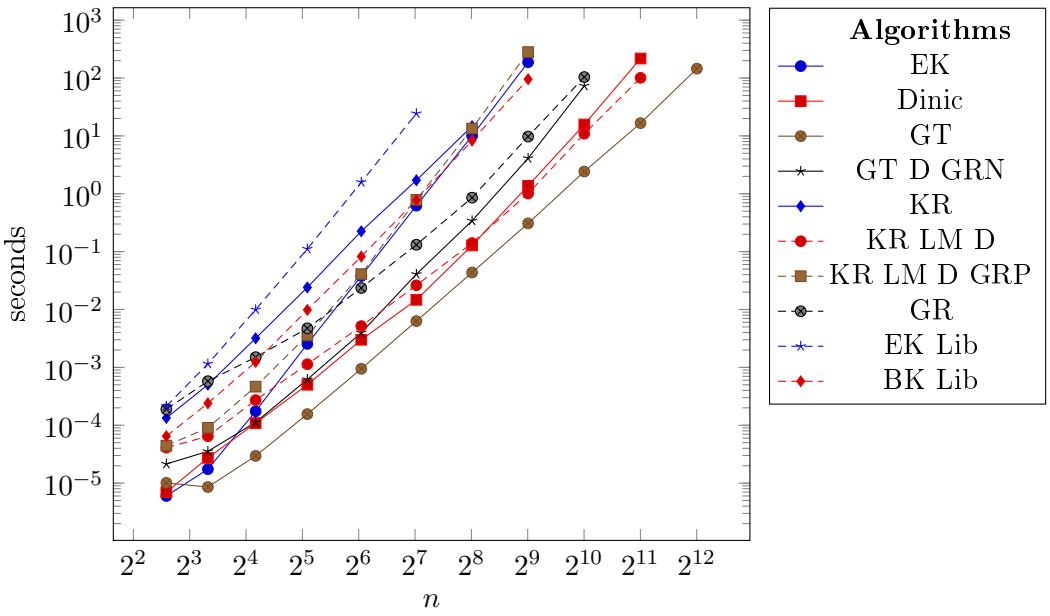

Figure 34: Best and worst results from the CD graphs

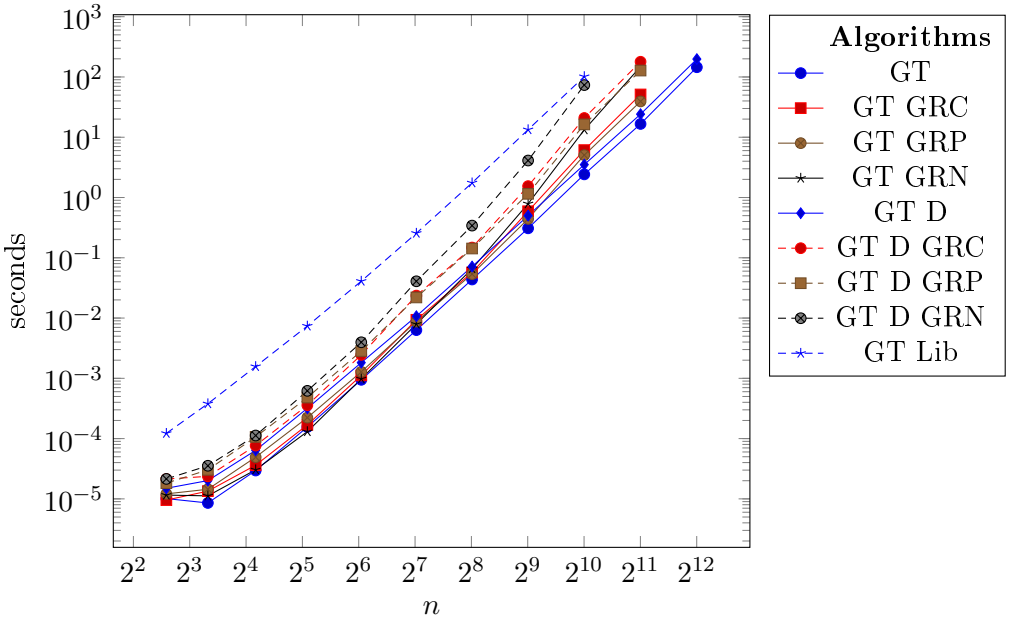

Figure 35: Goldberg and Tarjan results from the CD graphs

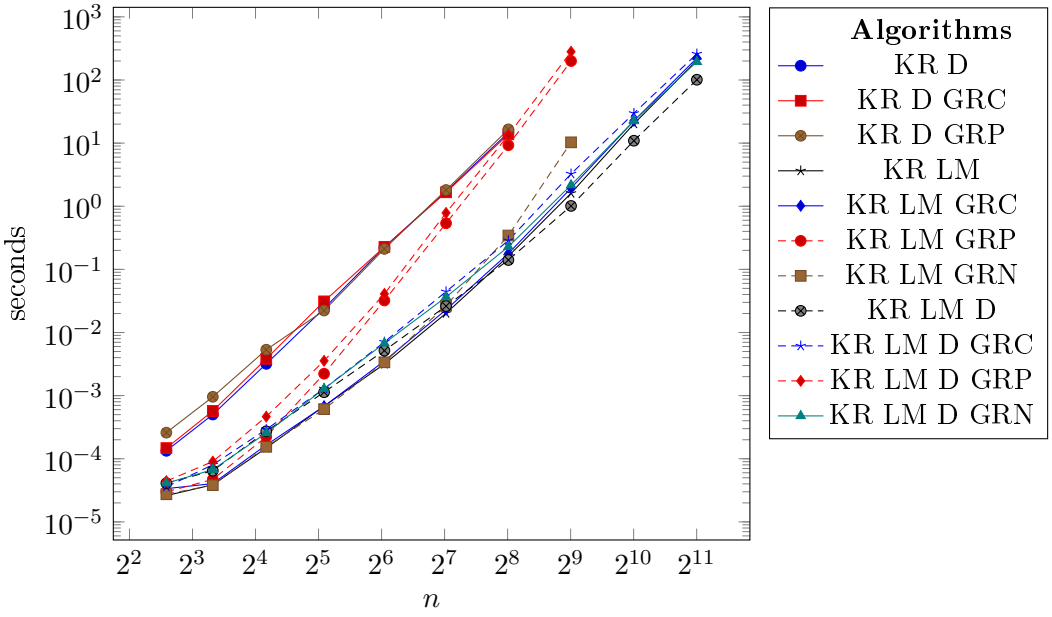

Figure 36: King and Rao results from the CD graphs

Appendix B.D Results from AK graphs

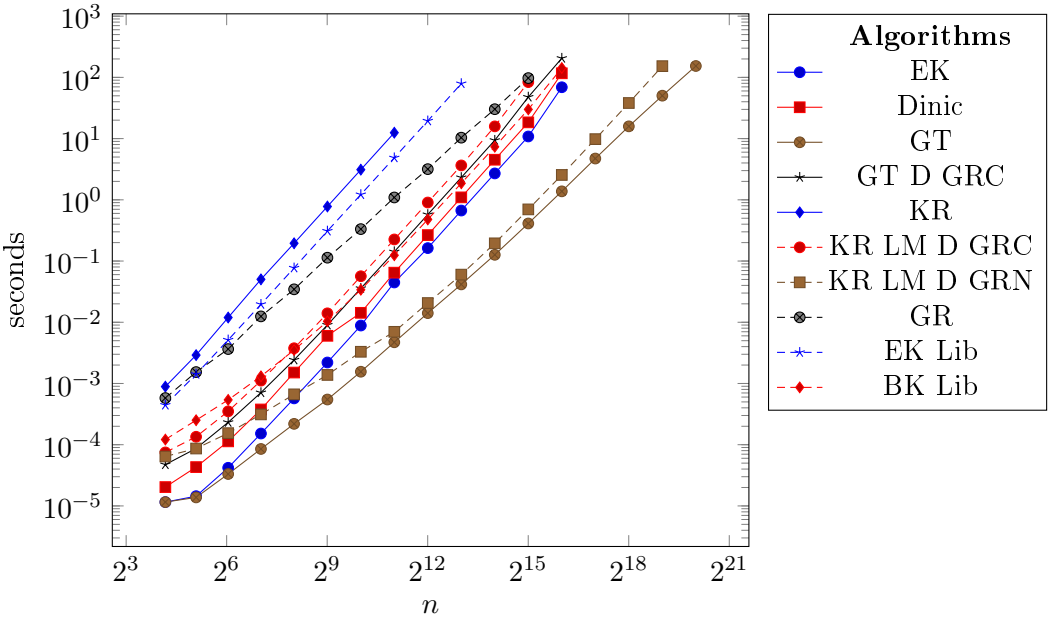

Figure 37: Best and worst results from the AK graphs

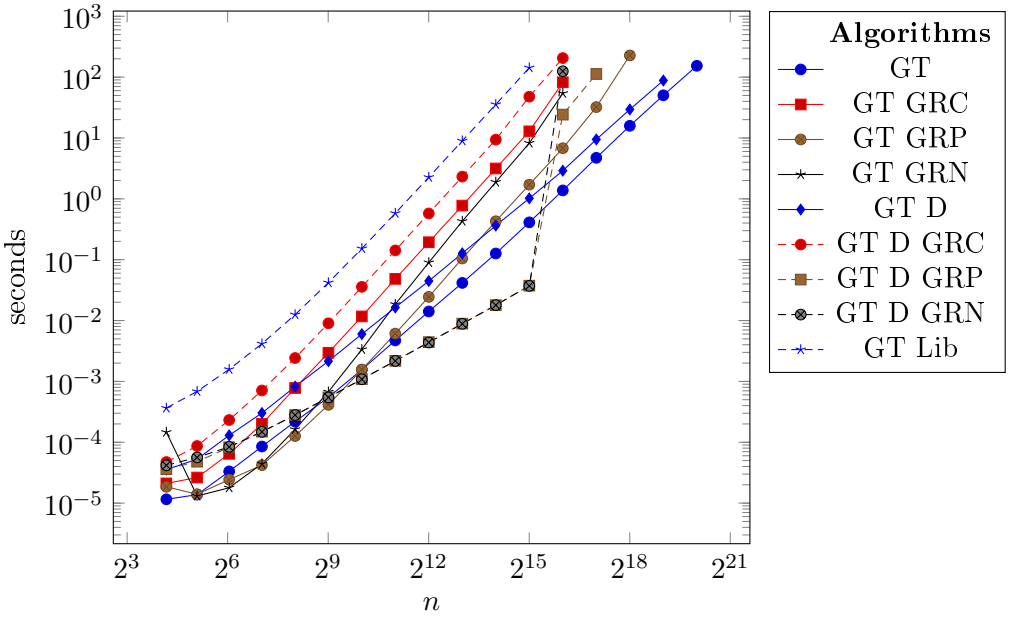

Figure 38: Goldberg and Tarjan results from the AK graphs

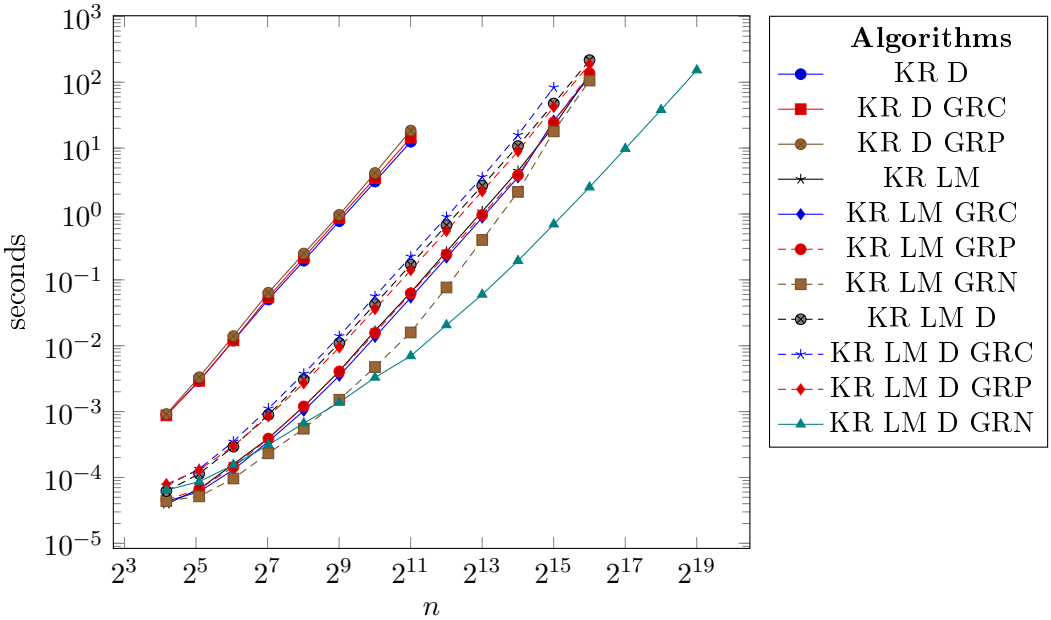

Figure 39: King and Rao results from the AK graphs

Appendix B.E Results from GenRmf long graphs

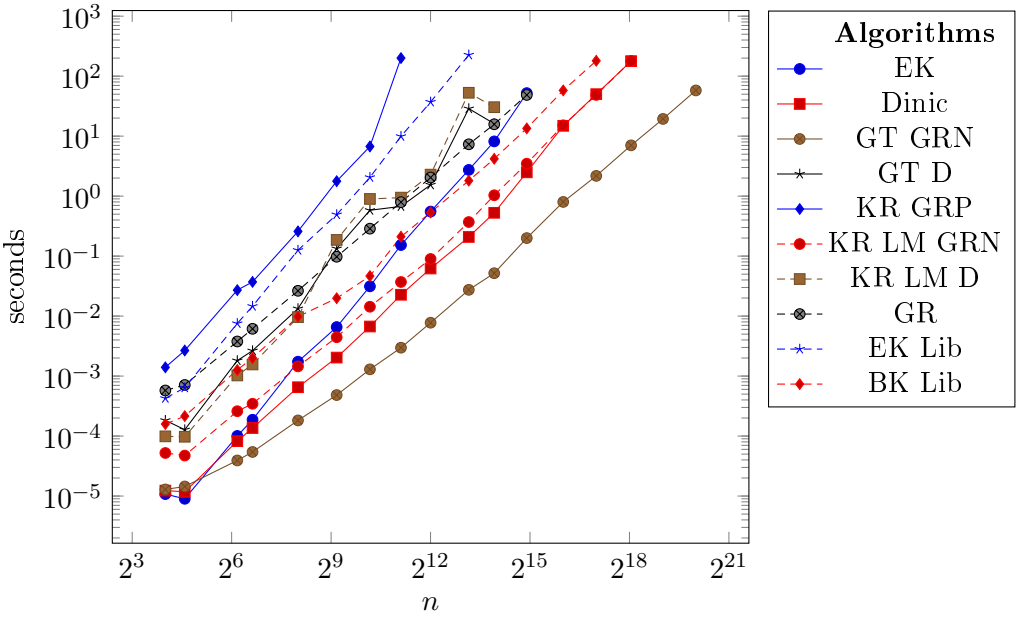

Figure 40: Best and worst results from the GenRmf long graphs

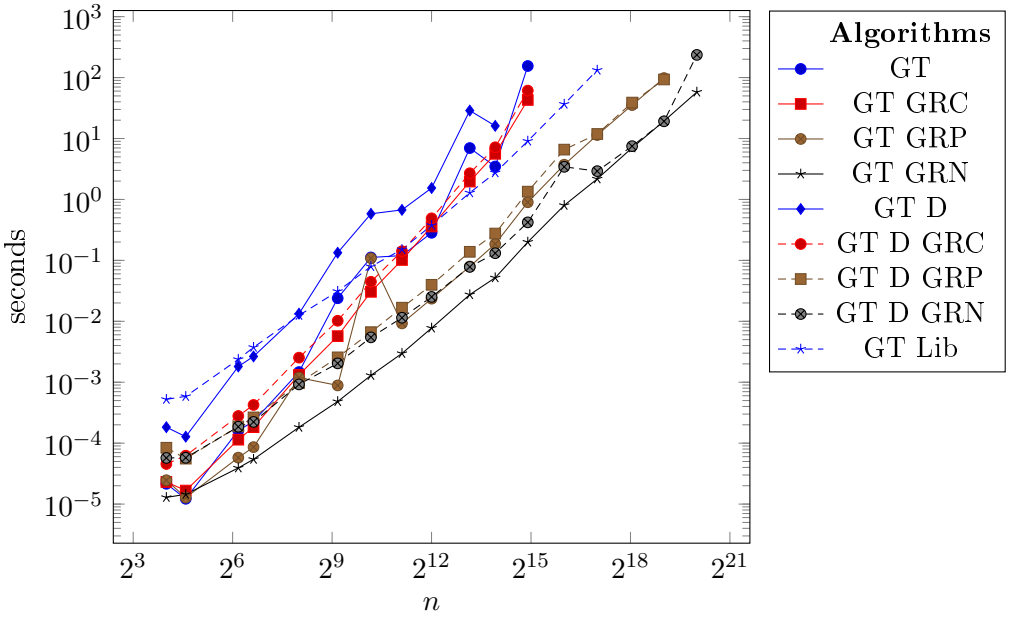

Figure 41: Goldberg and Tarjan results from the GenRmf long graphs

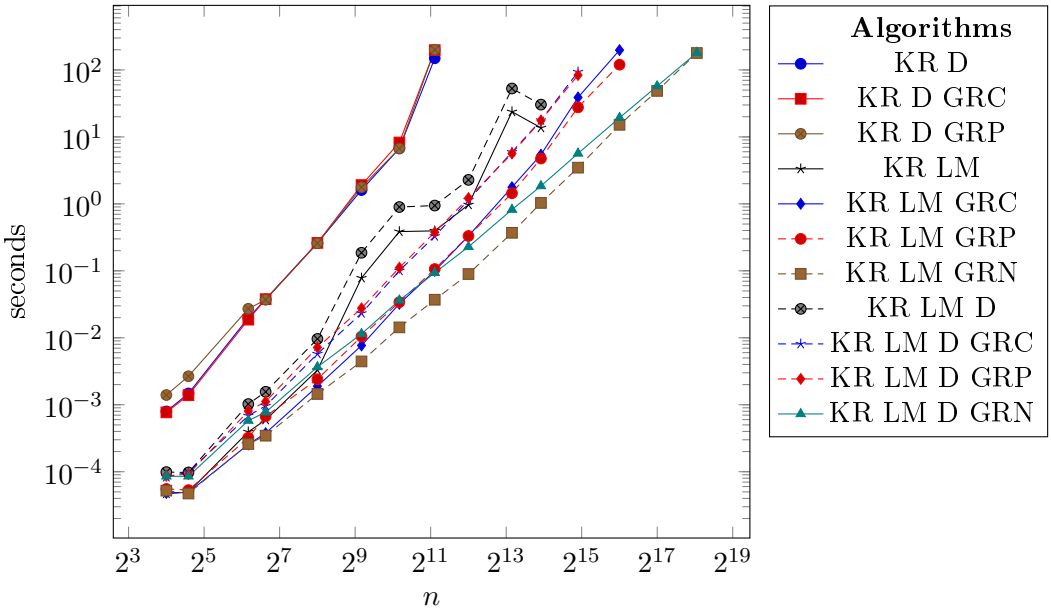

Figure 42: King and Rao results from the GenRmf long graphs

Appendix B.F Results from GenRmf flat graphs

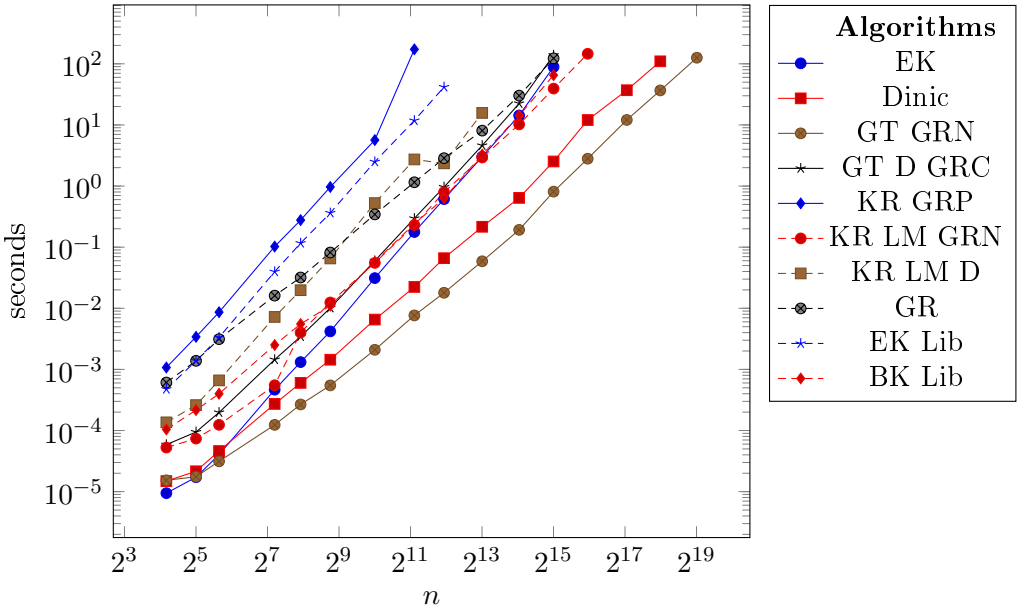

Figure 43: Best and worst results from the GenRmf flat graphs

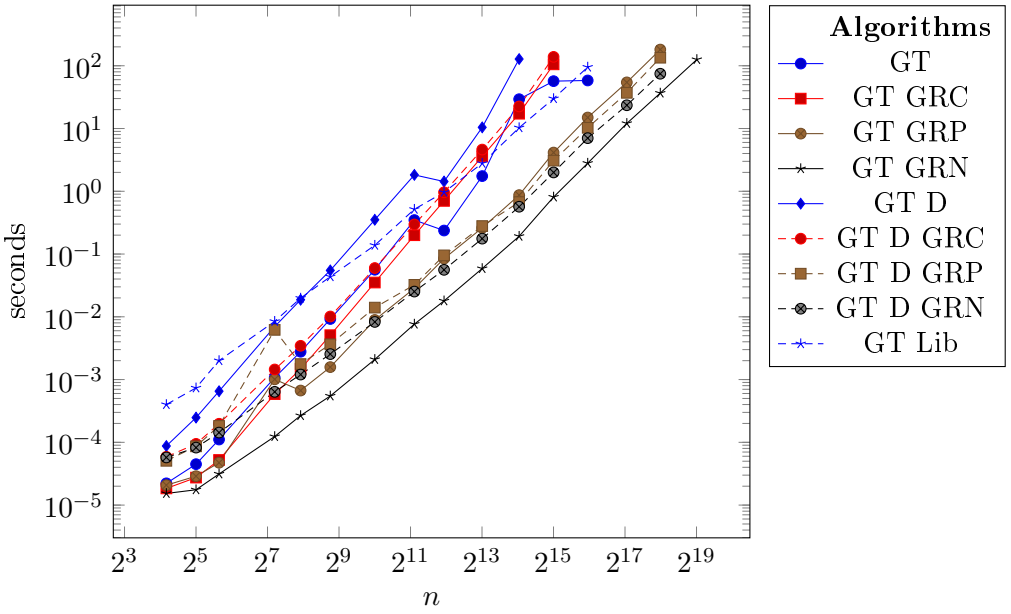

Figure 44: Goldberg and Tarjan results from the GenRmf flat graphs

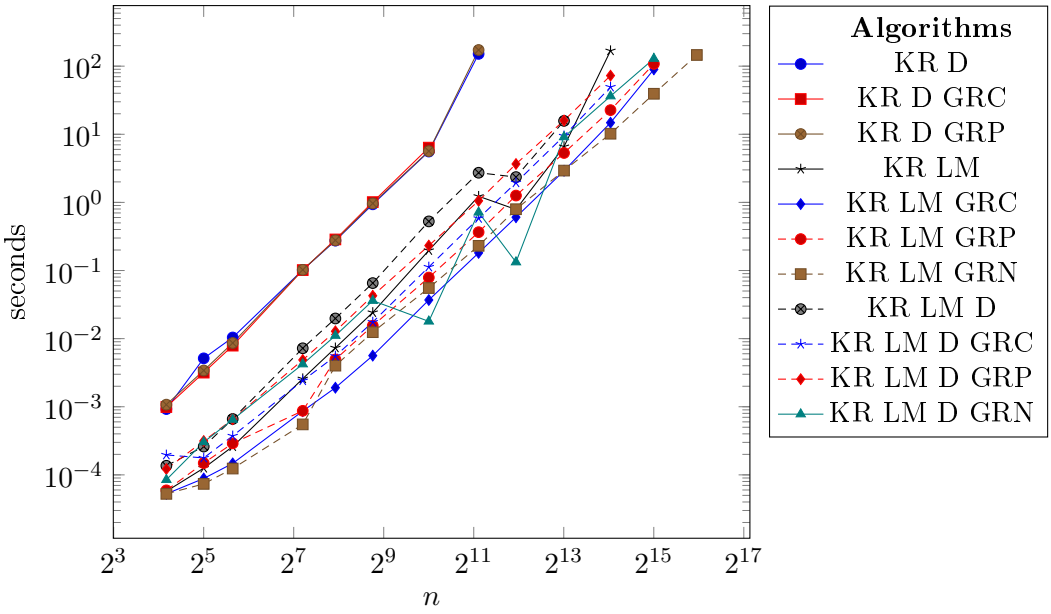

Figure 45: King and Rao results from the GenRmf flat graphs

Appendix B.G Results from GenRmf square graphs

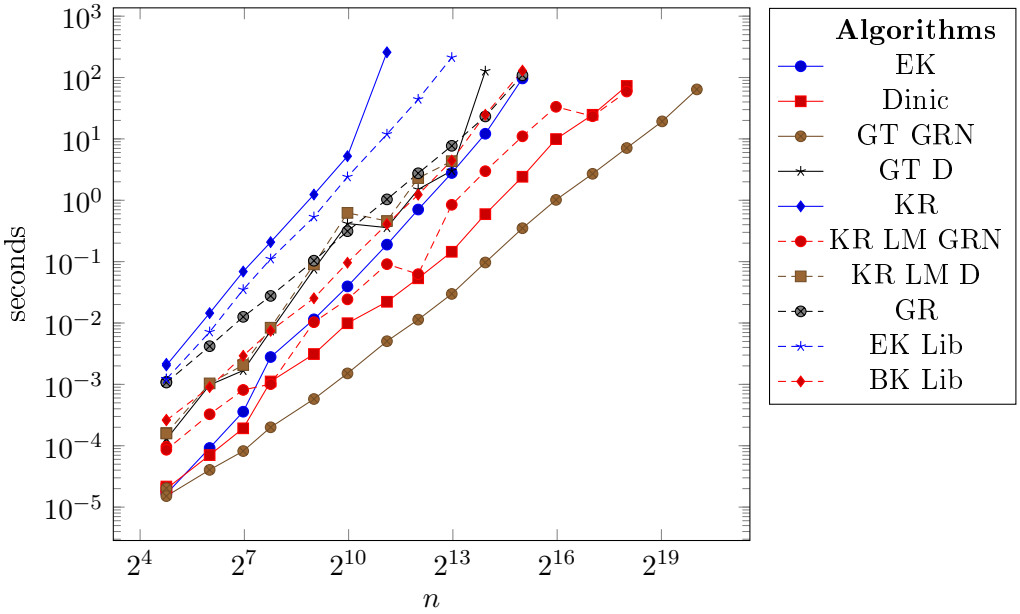

Figure 46: Best and worst results from the GenRmf square graphs

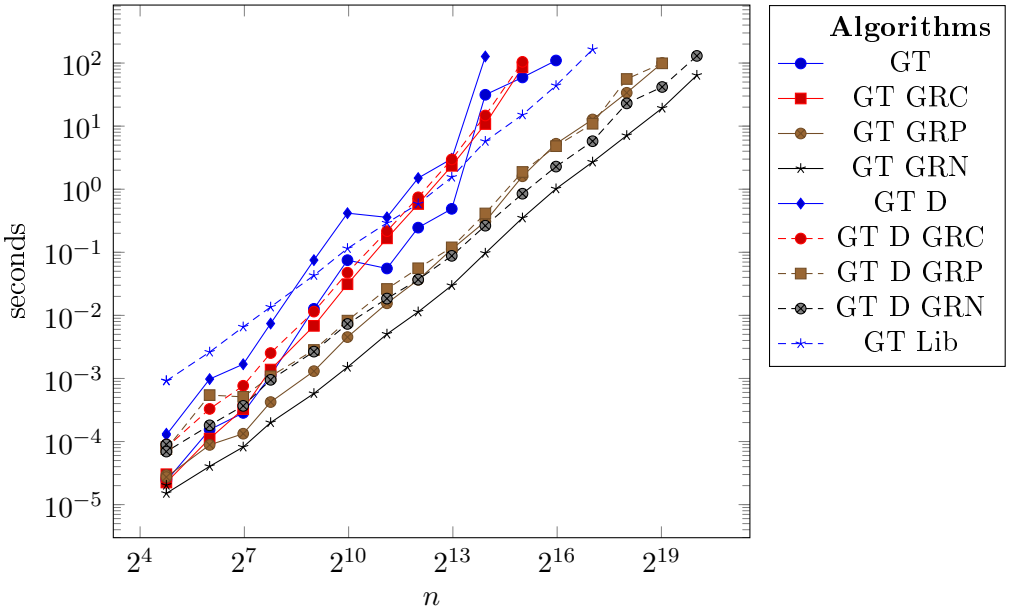

Figure 47: Goldberg and Tarjan results from the GenRmf square graphs

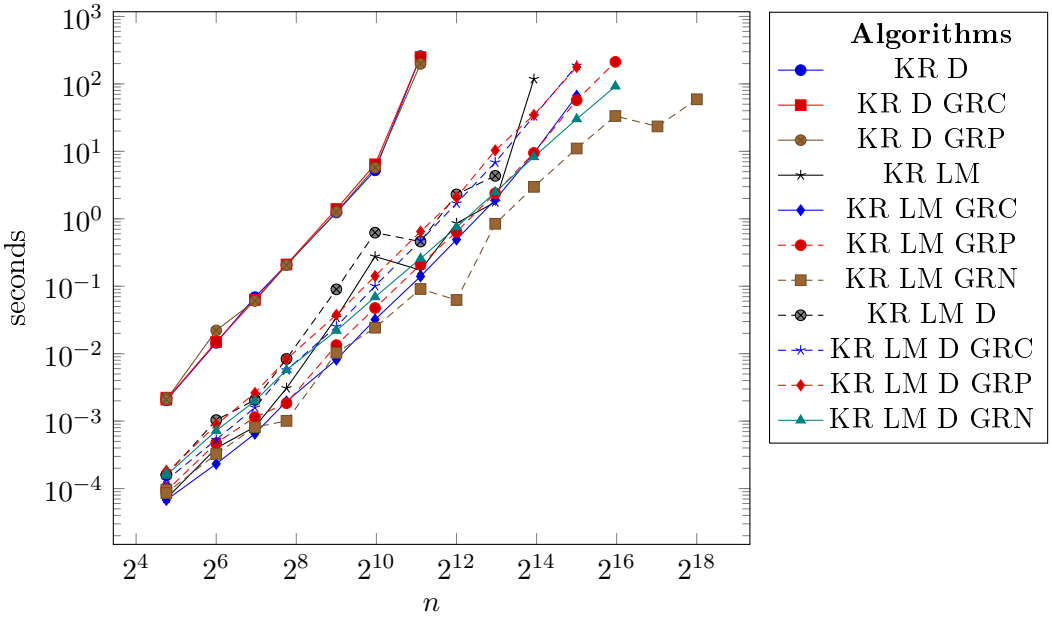

Figure 48: King and Rao results from the GenRmf square graphs

Appendix B.H Results from Wash long graphs

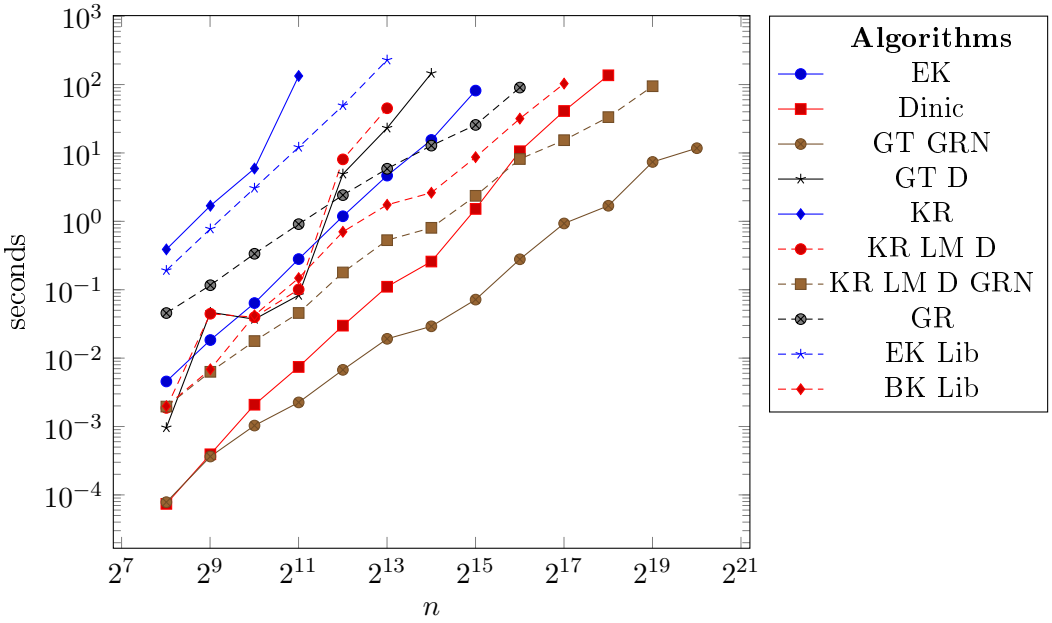

Figure 49: Best and worst results from the Wash long graphs

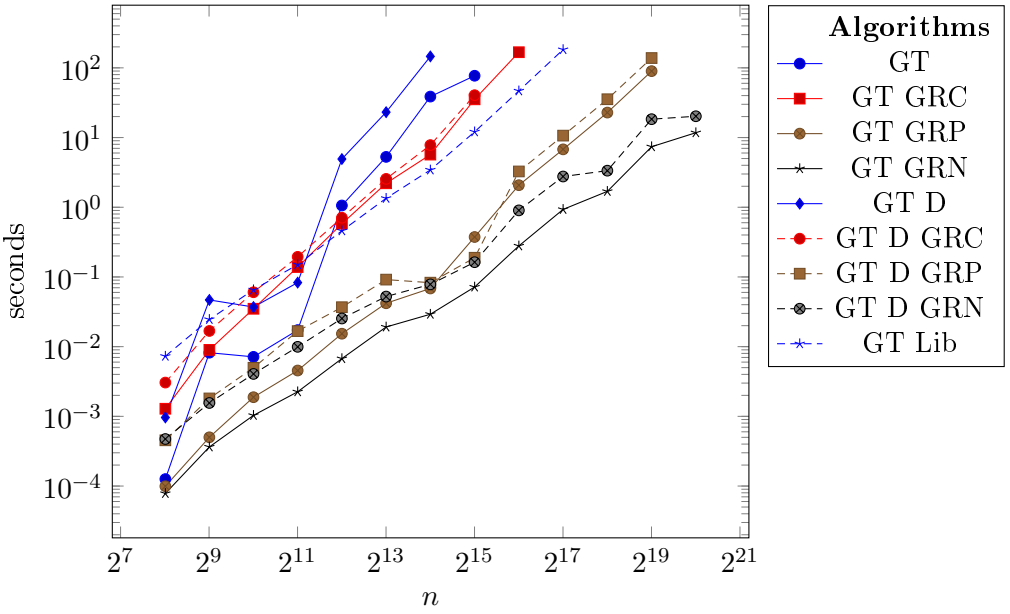

Figure 50: Goldberg and Tarjan results from the Wash long graphs

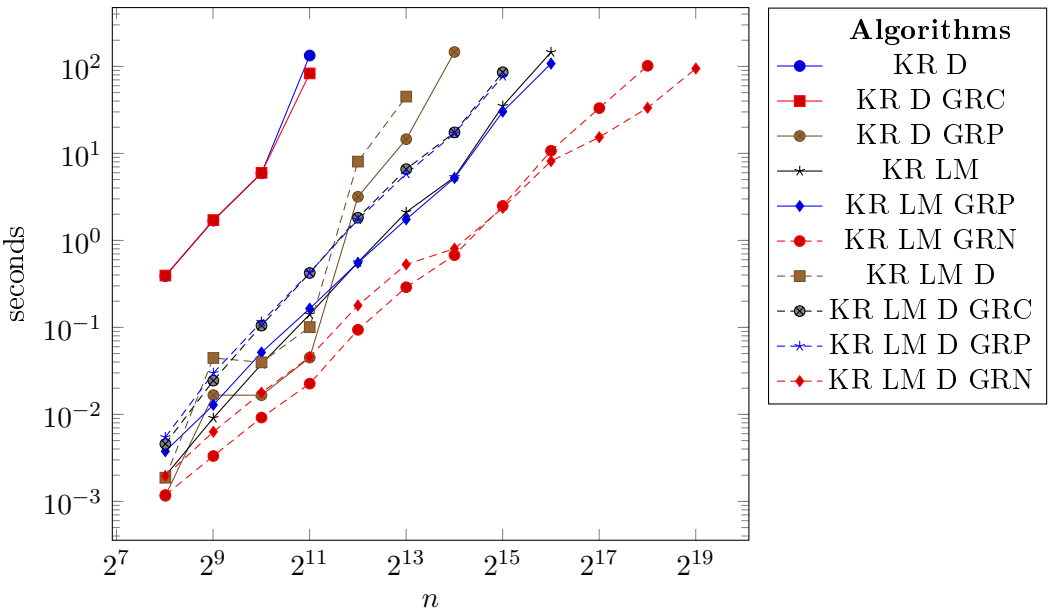

Figure 51: King and Rao results from the Wash long graphs

Appendix B.I Results from Wash wide graphs

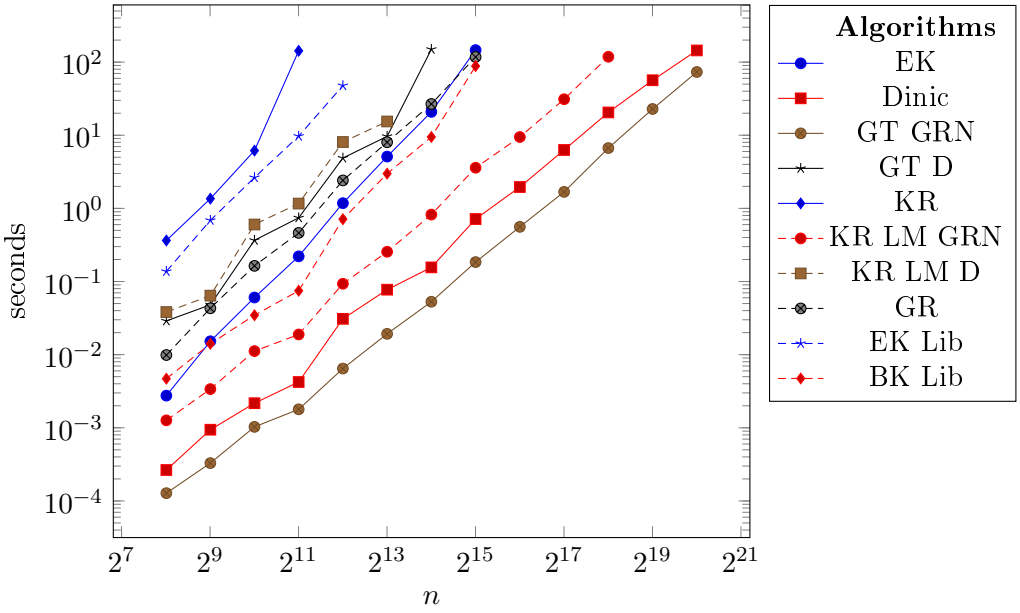

Figure 52: Best and worst results from the Wash wide graphs

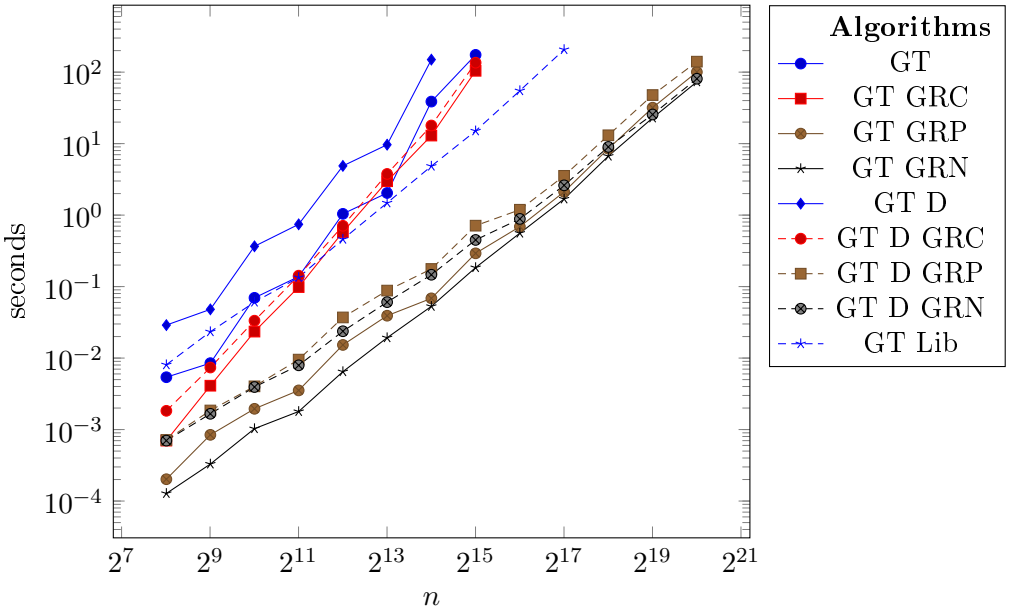

Figure 53: Goldberg and Tarjan results from the Wash wide graphs

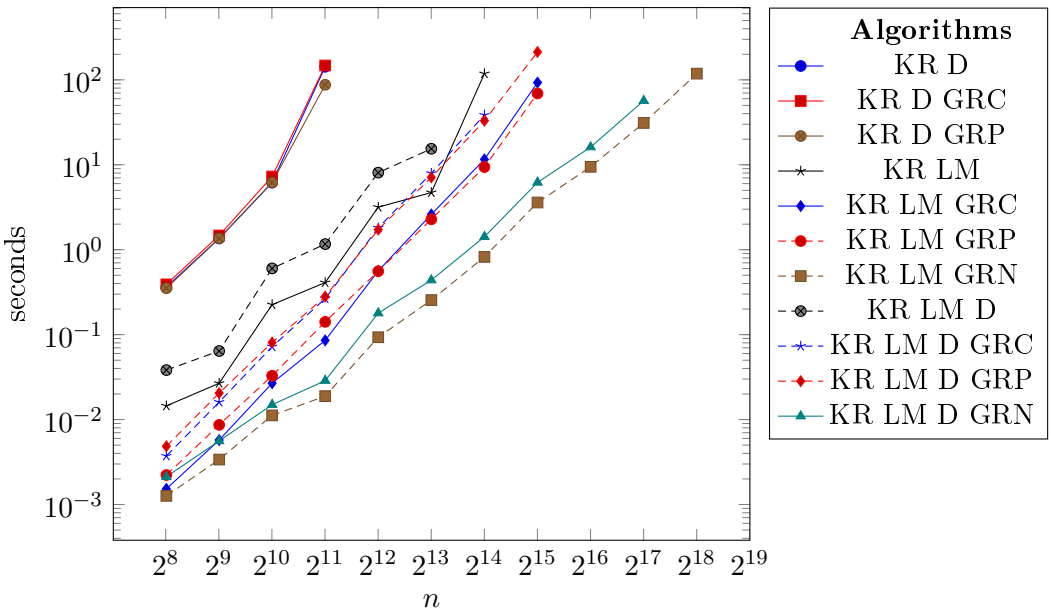

Figure 54: King and Rao results from the Wash wide graphs

Appendix B.J Results from Computer Vision graphs

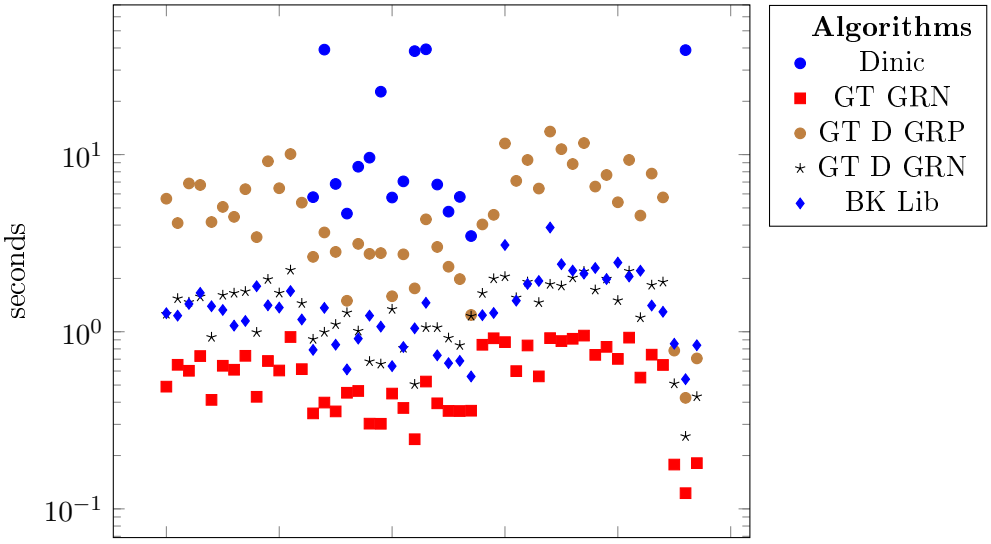

Figure 55: Results from the Computer Vision graphs

Appendix B.K Algorithm Running Times

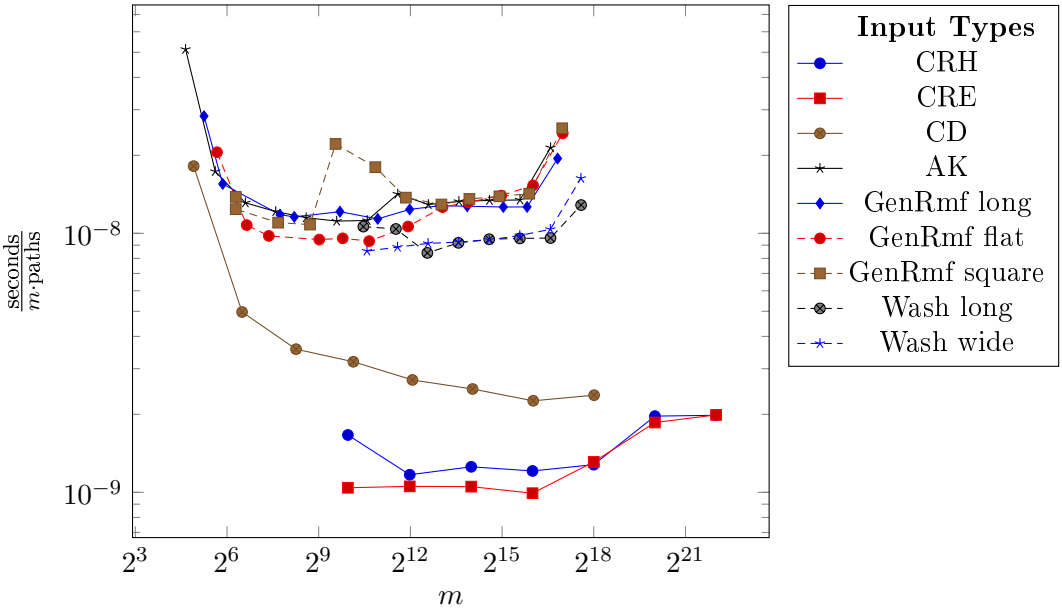

Figure 56: Edmonds and Karp performance per m and the number of augmenting paths

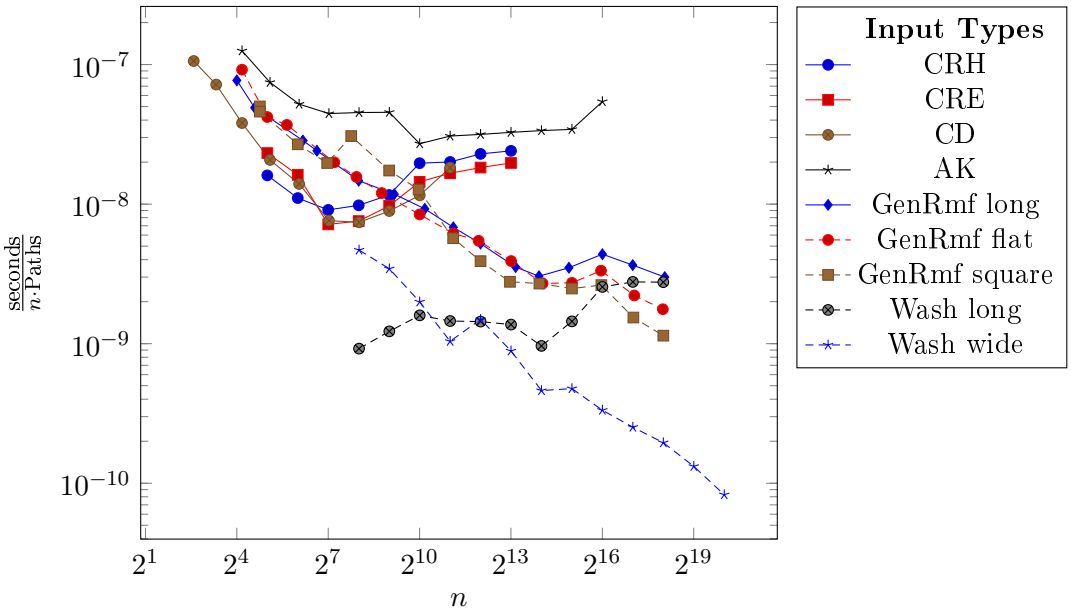

Figure 57: Dinic performance per nP

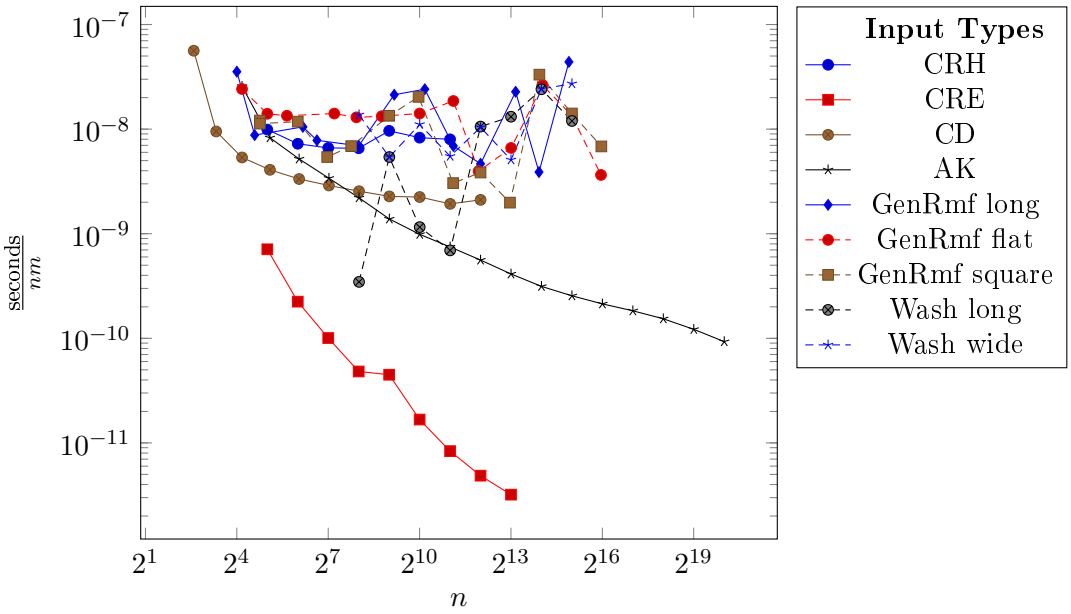

Figure 58: GT performance per nm

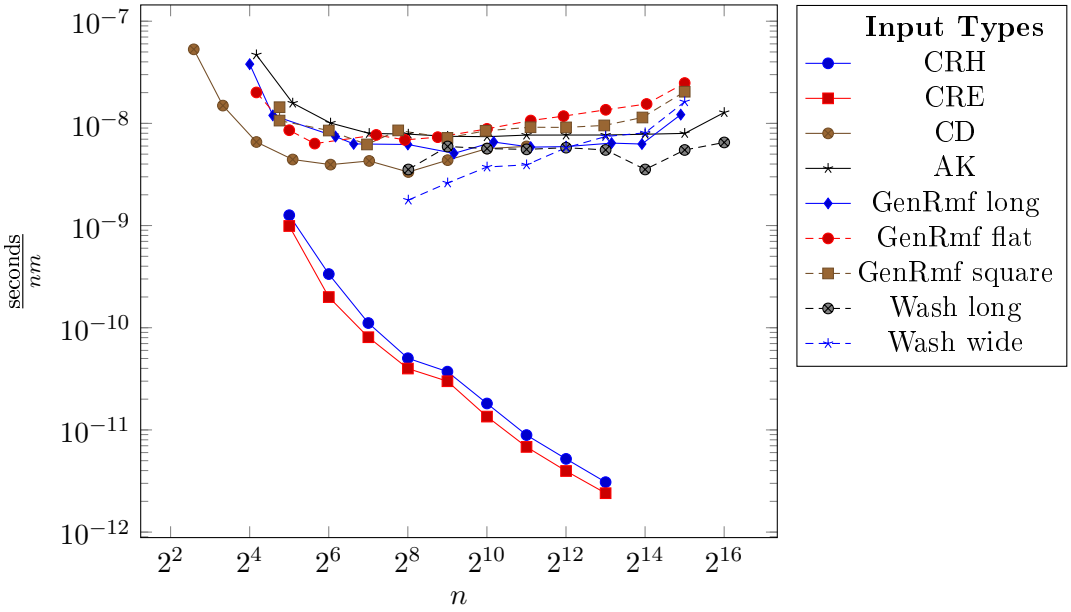

Figure 59: GT GRC performance per nm

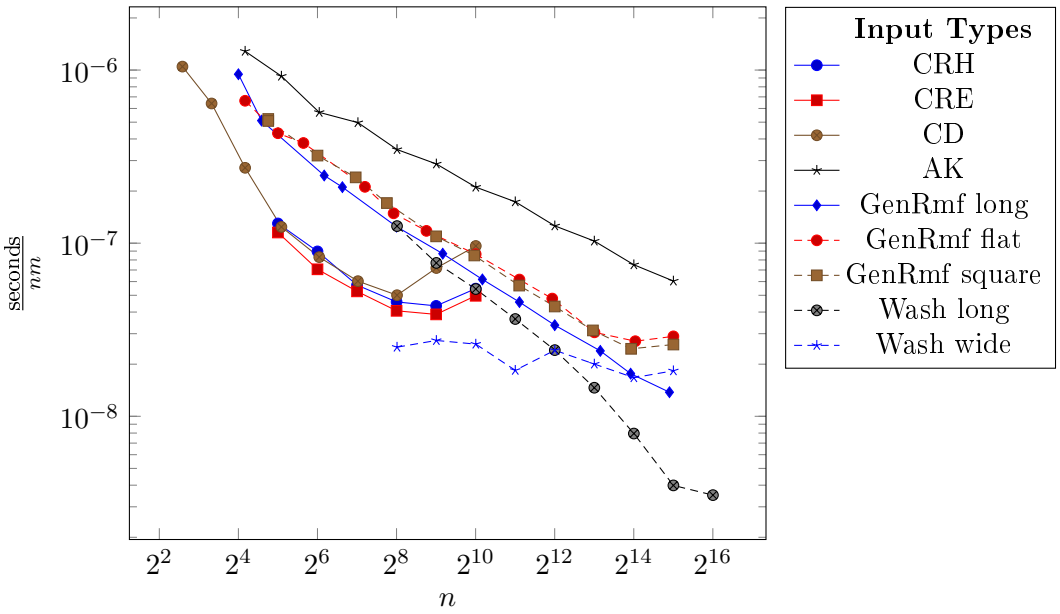

Figure 60: GR performance per nm