University of Bergen

Department of Informatics

Algorithms

Parallel algorithms for computing k-connectivity.

Student:
Joachim Haakonsen

Supervisor:
Professor Fredrik Manne

Master Thesis

March 16, 2017

Acknowledgement

I would like to thank my supervisor for the patience he have shown me. He have given me a calm and reliable presence in times of pressing matters and without his guidance this would not have been possible. I would also like to thank my fellow graduate students for the wonderful time I have had during my stay at the Department of Informatics. At last I would like to address a special thanks to my brother who have helped me see this through, and Ida Ryland who have encouraged and helped me all the way.

Contents

Chapter 1

Introduction

Connectivity of graphs is a widely studied subject in Computer Science, that applies to many real world problems. Especially in the field of Communication, where reliability is a big concern, it is important to know that the network is still operational if something were to happen to some part of the network.

With a communication network in mind, we represent the network by a graph, as seen in Figure 1.1. The graph consisting of vertices (circles) and edges (lines) that represent routes of transmission. The vertices can communicate by edges and relay information by passing the information to a new vertex via an edge until it reaches the intended destination.

When transmitting a message from a vertex s to a vertex t, it is crucial that there is a path from s to t in the network. This is why communication networks need to be fail safe. One example is emergency communication networks which are built in a redundant fashion. There are several redundant standards based on ring network topology. The main idea is that there are always two disjoint paths between every pair of vertices in the network, one path in each direction around the ring. Thus if s is communicating with t, and a tree fell and destroyed a network line or station, it would still be possible for s and

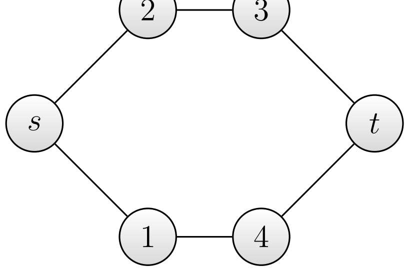


Figure 1.1: Ring Network

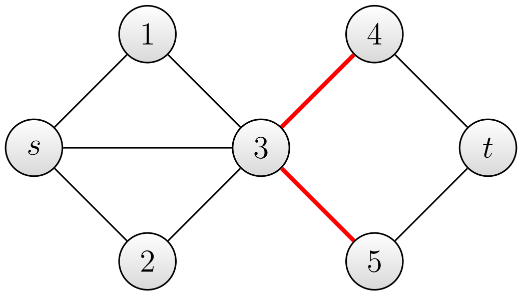


Figure 1.2: A graph with 2-edge-connectivity illustrated by red edges

t to communicate through another path in the network. This is made possible since the connectivity in the network guarantees that at least two trees would have to fall at the same time in two different locations to make s and t disconnected from each other.

The connectivity is determined by the minimum number of vertices or edges whose deletion makes the graph disconnected. We will in this thesis be focusing on the k-connectivity for both a pair s,t of vertices and the graph G. Where a pair s,t of vertices are k-edge-connected if by removing any k-1 edges the graph G is still connected, i.e., there is still a path between s and t in G. The pair of vertices can also be said to be k-vertex-connected if by removing any k-1 vertices the graph G is still connected. A graph is k-connected where k is equal to the minimum k-connected pair s,t of vertices of the graph G.

The connectivity of a graph has a strong relationship to other fundamental problems in theoretical Computer Science such as *minimum cut*, which is the minimum cut between two vertices, and *maximum flow*. Where the k-connectivity between a s,t pair is equal to the s-t minimum cut in the graph G. Based on the max-flow min-cut theorem by Ford and Fulkerson [1956] that showed the duality of minimum cut and maximum flow, we will be using both a maximum flow and a minimum cut algorithm in this thesis to find the k-connectivity of the graph G.

The approach for finding the k-connectivity will be based on the paper "A Simple Algorithm for finding All k-Edge-Connected Components" by Tianhao Wang et al. [13]. By utilizing an auxiliary graph the approach reduces the number of maximum flow calls the algorithm utilizes in contrast to a brute force approach.

Many approaches have been devised for solving k-edge connected components in a graph and finding the k-edge connectivity of a graph. Mainly these subjects have been studied for finding either the connectivity or the components. In the implementation of the algorithm for finding all k-edge connected components we subsequently end up with the k-edge connectivity for all pair of vertices in the graph.

In this thesis we will introduce the reader to the fundamental theory for what we will present. We then give a description of the algorithm proposed by Tianhao Wang et al. [13], before we introduce our implementation of the algorithm. We will first present the sequential implementation of the algorithm, then we modify the algorithm to a parallel implementation. At the end of this paper we will present the results of the sequential and parallel algorithm, before ending with a conclusion of the results.

1.1 Notation

In this thesis we define a graph G as an ordered pair $G = (V, E)$, where V is the set of vertices and E is the set of edges in G, and each edge is a pair of vertices from V. The $|V|$ is used to address the size of the set V. A is used for referring to the auxiliary graph. A set $L = \{\}$ is a set which is set to have no elements and it is called an empty set. A set C is minimum if removing more elements from C would change its property. A set C is maximum if adding more elements to C would change its property. A set $U \subseteq V(G)$ is *connected* if $G[U]$ is a connected graph. The expression $D \setminus C$ is the removal of all the elements in C from set D. $O(\Delta)$ is used for big O notation.

1.2 Graphs

A vertex, also known as a node or point, represents an object. The edges represents the relations between the objects. One simple example is a social network where a vertex represents a person, or a user, and edges expresses social bounds between vertices. The set of edges is represented by a pair of vertices $(v, u) \in E$, where $v \in V$ and $u \in V$, and u and v are the vertices incident to the edge. A vertex in the graph not present in any edge in G is called an *isolated vertex*, since it has no relation to any other of the vertices in G.

Edges of a graph can either be directed or undirected, all depending on the relation that are being expressed. In a directed graph the edges is a set of ordered pairs of vertices $(u, v) \in E$, where u has a relation to v and v has no relation to u unless $(v, u) \in E$. In other words, the bond between the vertices is directed from one vertex to another. We call u the *tail* and v the *head* of the edge as depicted in Figure 1.3b. For an undirected graph the edges is a set of unordered pairs of vertices, $<u, v> \in E$, thus the relation between u and v goes both ways in contrast from a directed graph. Thus having both $<u, v>$ and $<u, v>$ in E would be redundant since they express the same relation. We can easily see this in Figure 1.3a.

If a graph contains multi-edges and self-edges it is a *multi-graph*, as illustrated in Figure 1.3c. Multi-edges simply refers to the occurrence of edges which point to the same pair of vertices. A self-edge is an edge where a vertex is pointing to itself. The rest of this thesis we will not take in to consideration multi-graphs, multi-edges or self-edges.

The edges and vertices of a graph can be weighted or unweighted. An edge with weight w can be a symbolic representation of the strength of a relation between two vertices in a graph. The weight can represent the capacity of an edge in a flow network, limiting the flow that can flow through it. A vertex weight w can be a symbolic representation of the presence a vertex has in the graph, i.e., it may represent the number of visitors a web page have had during a time period, in a graph of web pages. This can be useful when looking for causes of web pages having more visitors then others. Do web pages with large number of visitors link to other pages with large number of visitors? This could be useful when analyzing consumer behavior and identifying patterns in different consumer segments.

An undirected weighted graph is illustrated in Figure 1.3d. Weights are assigned to each edges in the graph and each edge has its own weight associate with it, i.e., $w_e \forall e \in E$. Similarly, vertices can have a weight w associated to it, $w_v \; \forall \; v \in V$.

A graph is said to be *dense* if the number of edges is close to the maximal number of edges, or *sparse* if the number of edges are close to the number of vertices in G. Let D be the measure of the graph density, then we can measure D for an undirected graph by the expression:

$$D = \frac{2|E|}{|V|(|V|-1)}.$$

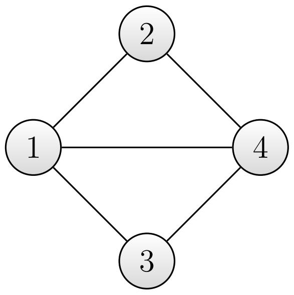
For a directed graph the graph density is measured by the expression:

$$D = \frac{|E|}{|V|(|V|-1)}.$$

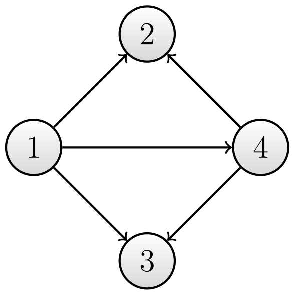
An acyclic graph is a graph without cycles. In a directed graph this is called a *Directed Acyclic Graph* (*DAG*). A DAG is a finite directed graph for which there exists a topological ordering of the vertices. This means that while traversing the vertices in order, one would never encounter an edge leading back to a vertex while preceding the vertices in order. Then for every edge, if there exists such a topological ordering, it will point from a vertex to a vertex succeeding that vertex in the ordering. A connected acyclic undirected graph is a graph where the number of edges is equal to the number of vertices subtracted by 1. It is easy to see that if the graph is connected then it must be minimally connected, since adding an edge to the graph would create a cycle.

1.2.1 Trees

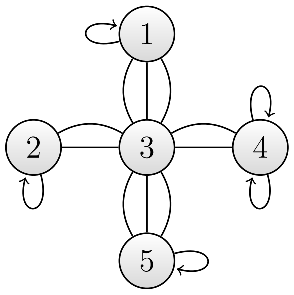
A *tree* is a mathematical structure that can be viewed as an acyclic undirected graph where every pair of vertices are connected through exactly one simple path in the graph. A collection of trees are called a *forest*. A tree is an acyclic connected graph. Thus there can not exist any cycles which, as mentioned, is exactly that of an acyclic undirected graph. The vertices of a tree can be split into two groups, the internal and *external* vertices of the tree. An internal vertex has degree at least 2, and can be seen as a fork in the graph structure allowing paths to branch outwards from the center of the tree. External vertices, usually referred to as the leaves of the tree, have degree exactly 1 and are the end vertices of the tree. In Figure 1.3e, the internal vertices are $1, 2, 3$ and the leaves are $4, 5, 6$. A tree with at most two branches at each fork and at most two leaves connected to a fork is called a *binary tree*. Figure 1.3e and 1.3f are both binary trees.



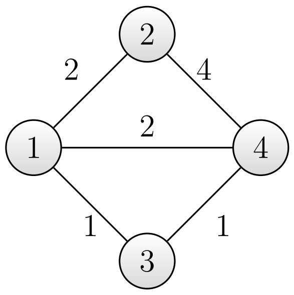
(a) Undirected Graph



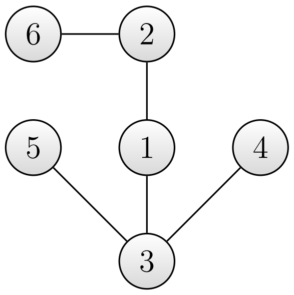
(b) Directed Graph



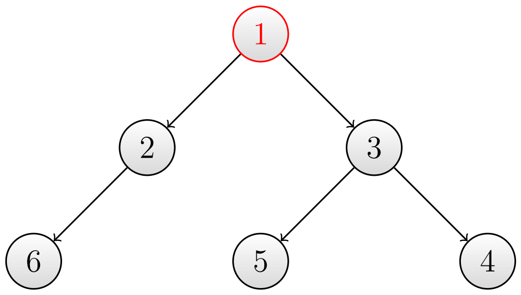
(c) Multi Graph



(d) Undirected Weighted Graph



(e) A Tree



(f) A Rooted Tree

Figure 1.3

A *rooted tree* is a tree where a vertex is picked and marked to distinguish it from the other vertices of the graph. This vertex is called a *root vertex* and is the root of the tree. The root vertex, marked red in Figure 1.3f is vertex 1. A rooted tree is usually depicted with the root at the top and edges directed away from the root called branches, referred to as a branching or an out-tree. The edges can also be directed towards the root, called an in-tree.

A vertex u is called a *parent* of a vertex v if it is adjacent to v and the distance from the root to u is smaller than the distance from the root to v. The vertex v is called a *child* of u since it is adjacent to u, and its distance to the root are longer than the distance between the root and u. In Figure 1.3e vertex 3 is the parent of the vertices $4, 5$, and $4, 5$ are the children of vertex 3. In a rooted tree every vertex have a unique parent, except for the root. For rooted trees internal vertices are the vertices that have children and the leaves are childless vertices. In Figure 1.3f the internal vertices are $1, 2, 3$ and the leaves are $6, 5, 4$.

Similarly to family trees, rooted trees have ancestors and descendants. If two vertices appear in a simple path between the root and a leaf vertex, then the vertex closer to the root in the path is called an ancestor and the vertex farther away is called a descendant. Vertices in a rooted tree can also be divided into internal vertices and leaves.

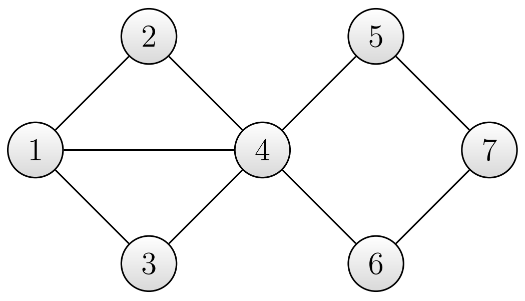
Trees have a lot of nice properties that make them ideal to use as a data structure. Its acyclic characteristics along side with its decisive vertex to vertex connection makes it ideal for use in searching and more. The structure is being exploited in areas such as statistics, data mining and machine learning. In this thesis we will take advantage of the rooted tree structure to store results of our partial calculations. We will then do look ups in the tree for fast retrieval of information. The root vertex is crucial for maintaining the information of the partial calculations without loss of information.

1.3 Connectivity

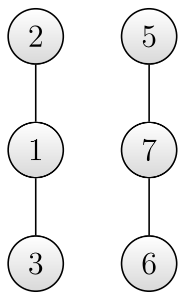
A graph is said to be connected when for every pair of vertices in the graph there exist a simple path connecting them, such that no vertices are isolated. Subsequently a graph is disconnected if there exist a pair of vertices such that there exist no simple path with the vertices as its endpoints in the graph. As an example, a single vertex is a connected graph where as a graph with more than 2 vertices and no edges is disconnected.

In an undirected graph G, two vertices u and v are connected if there exists a simple path from u to v in G. If no such simple path exist, then the two vertices are in disjoint parts of G making both the graph disconnected and the vertices u and v disconnected.

In Figure 1.4 a there exist a path between vertex 1 and 7, and we can say that they are connected. This is true for all pairs of vertices depicted in 1.4 a making the graph connected. The graph depicted in Figure 1.4 b does not have a path between 1 and 7



(a) Connected Graph



(b) Disconnected Graph

Figure 1.4

making the vertices and the graph disconnected.

Connectivity is an important issue, especially when talking about infrastructure. The connectivity or dis-connectivity can be a matter of avoiding catastrophic consequences. This is why important infrastructure such as the emergency network is designed with a high connectivity, to make sure it never fails.

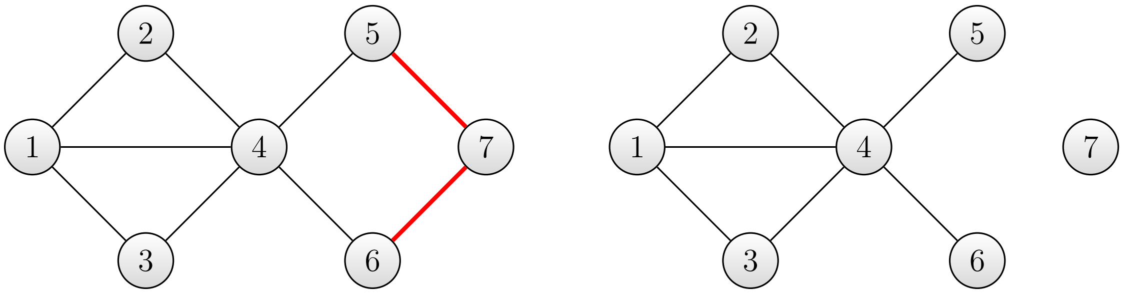
1.4 Cut

It is often difficult to say something about the connectivity of a graph. The approach used in this thesis, uses the notation of a cut to classify the connectivity of the graph. For the rest of this thesis we will assume that a graph is connected, as the cut for an unconnected graph is trivial. A cut C of a graph G is defined as the partitioning of the vertices into two disjoint subsets $S, T \subseteq V(G)$: $[C = (S, T)\ s.t\ v \in S \iff v \notin T\ and\ u \in T \iff u \notin S]$

The only path between vertices in different subsets exists through what is called a cut-set. A cut-set can either consist of edges or vertices. The vertex cut-set C_V is a subset of the vertices of the graph $C_V \subseteq V(G)$ if and only if the removal of the set C_V, $V(G) \setminus C_V = \{v \in V(G) : v \notin C_V\}$, results in G being disconnected, i.e., in Figure 1.4 *a* the vertex labeled 4 is a vertex cut-set resulting in the graph depicted in Figure 1.4 *b*.

An edge cut-set C_E is a subset of the edges of the graph $C_E \subseteq E(G)$ if and only if the removal of the set C_E, $V(G) \setminus C_E = \{e \in E(G) : e \notin C_E\}$, results in G being disconnected. In Figure 1.5 *a* the edges to the vertex labeled 7, marked red, would be an edge cut-set of the graph. $C_E = \{(7, 5), (7, 6)\}$ would be the edges in the cut, and the removal of the edges would lead to vertex 7 being isolated from the rest of the graph, as seen in Figure 1.5 *b*.

When a start vertex s and target vertex t are specified the cut, $C = (S, T)$, is referred



(a) Cut Edges in Red (b) Cut Edges Removed

Figure 1.5

to as a s-t cut, where $s \in S$ and $t \in T$.

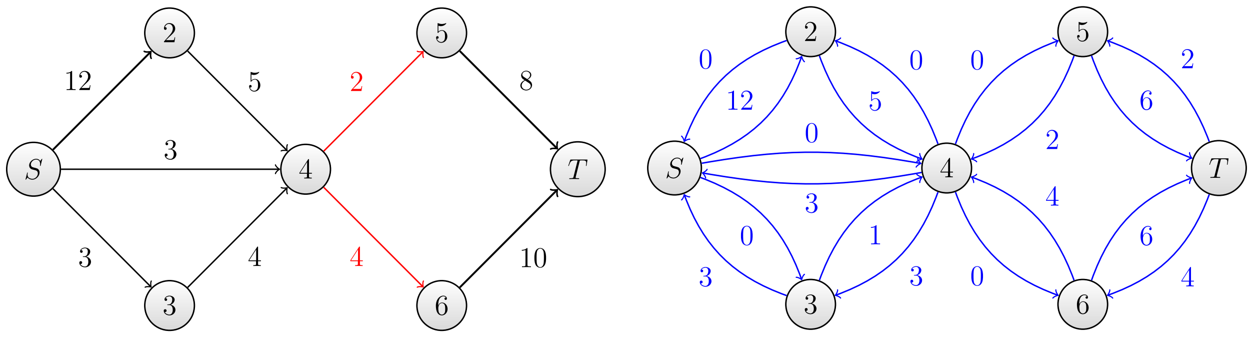
There are several ways to measure the size of the cut, depending on the instance of the graph. In an unweighted graph the size of the cut C are the number of edges (vertices) in the edge (vertex) cut-set, or more formally $size(C) = |C_E|$ ($size(C) = |C_V|$). Where as for a weighted graph the size of the cut is the sum of the edge (vertex) weights corresponding to edges (vertices) in the edge (vertex) cut-set, $size(C) = \forall\, e \sum\limits_{e\,\in\,C_E} w_e$. Consequently for a vertex cut-set, we have that the cut size in an unweighted graph is the number of vertices in vertex cut-set $size(C) = |C_V|$. The cut size in a weighted graph is the sum of all the vertex weights corresponding to vertices in the vertex cut-set, $size(C) = \sum_{\forall v \in C_V} w_v$.

Now that we know how to measure a cut, it may not come as a surprise that cuts are a classic *minimization/ maximization* problem. In minimum (maximum) cut, the task is to find an optimum answer so that the cut is minimal (maximal). While minimum cut is known to be P-complete, the max cut problem is known to be NP-complete. To prove that maximum cut is in NP, one would have to provide a certificate and check in polynomial time that the given certificate is a yes-instance. This would in the case of maximum cut be a partition of the vertices in to two sets such that the size of the cut is maximal for the graph. In "Reducibility among combinatorial problems" by Karp [6] max cut was one of the 21 problems whose NP-hardness was proven. Since then it has been a widely studied subject.

We will in this thesis focus on minimum cut for determining the connectivity of the graph.

1.4.1 Minimum Cut

Cuts are a classic minimization problem well-studied in the area of Computer Science. Finding the minimum cut of a graph, is the task of finding a cut of minimal size. We say that a cut is of minimum size if there does not exist any other cut in the graph of strictly



(a) Flow Graph, with Source S and Target T (b) Residual Graph of the Flow Graph

Figure 1.6

smaller size. There are many variations of the minimum cut problem, depending on how a cut is defined and measured. This trait makes the minimum cut problem an algorithm that is applicable for a variety of real world application such as distributed computing, network reliability and baseball elimination to mention a few.

An example of a minimum cut problem is that of finding the minimum s-t cut of the graph. Where given an undirected graph G, we want to find a set $C_E(C_V)$ of minimal size, such that $G' = G - C_E(G - C_V)$ separates s and t into two disjoint parts of G' making G' disconnected.

There exists many algorithms for solving minimum cut. In 1956 the famous Max-Flow-Min-Cut theorem by Ford and Fulkerson [3], showed the duality of the minimum s-t cut and maximum flow. The theorem states that the size of the minimum s-t cut is equal to the maximum capacity flow through a single source, single sink network. We will in this thesis use a flow algorithm to find the minimum cut.

A *flow graph* is a directed weighted graph as seen in Figure 1.6a. The objective is to maximize the flow that run through the flow graph from a *source* vertex to a *sink* vertex. With the constraints that the capacity of each edge is not exceeded, and incoming flow is equal to out going flow at each intermediate vertex. In a possible analogy, the source can be seen as the power station trying to push enough current through the power grid to a industrial plant with a high demand power consumption to be able drift its operations. It then becomes crucial that there is enough capacity in the power grid to allow for the current flow to make its way from the source to the sink.

A *residual graph* is used to keep track of the flow that passes through each edge. A residual graph is a directed weighted graph. Given a flow graph G, the residual graph G_r is constructed with a flow f associated with each edge such that:

1. $V(G) = V(G_r)$

2. $\forall\, e \in E(G)$ there $\exists\, x \in E(G_r)$ with a capacity of $c_x = c_e - f_e$
3. $\forall\, e = (u, v) \in E(G)$ there $\exists\, (v, u) \in E(G_r)$ with a capacity equal to f_e

A vertex can only be reached through an edge with capacity greater than 0, so as long as there exists a path from source to sink in G_r there is potential to increase the flow.

The maximum flow algorithm only operates on directed graphs. This means that given an undirected graph $G = (V, E)$, the graph has to be transformed to a directed graph $G' = (V', E')$. To construct G' from G, set $V' = V$ and start with E' as an empty set. Then for all edges $<u, v> \in E$, add two directed edges, (u, v) and (v, u) to E'.

We then design the maximum flow algorithm, based on Edmonds–Karp algorithm[2] for maximum flow:

Algorithm 1.1 Max Flow

```
Input: Given a flow graph $G$, source vertex $s$, sink vertex $t$
  Create a residual graph $G_r$ from $G$, and set $maxF = 0$
  while $\exists$ a path from source to sink in $G_r$ do
      set $f = \infty$
      for all $e \in path$ do
          if $c_e < f$ then
              $f = c_e$
          end if
      end for
      $maxF$ += $f$
  end while
  return $maxF$
```

The running time varies depending on the implementation. Edmonds–Karp algorithm for finding maximum flow runs in $O(|V||E|^2)$. It is possible to alter the algorithm so that the vertices on each side of the cut is also returned along with the maximum flow/minimum cut size. An approach for finding the vertices on the source side, is simply running a *depth first search* algorithm on the residual graph from the source not allowing transitions from a vertex to another vertex over edges that have capacity 0. Along the way, the algorithm marks all the visited vertices and adds them to the source side cut-set. The vertices not marked by the depth first search (DFS) belongs on the sink side cut-set, since it is not possible for the algorithm to move from the source side of the cut to the sink side. This is due to the fact that we have maximized the flow and there can therefore not exists any edges with capacity left from one side of the cut to the other side. If one such edge existed, than that would mean that the flow returned is not maximal and thereby the cut would not be a cut.

If we set all edges corresponding weight to be equal to 1, in other words we let the weight reflect the presence of an edge in the graph, then the k-edge connectivity of a graph is equal to the smallest size minimum s-t cut of the graph. Thus the k-edge connectivity tells us the minimum number of edges whose removal would make the graph disconnected. Which indicates how strong of a connectivity a graph and its components have.

1.5 K-connectivity

The k-connectivity of a graph can be measured over the edges or the vertices. We will call k-connectivity over the edges for k-edge-connectivity and k-connectivity for vertices k-vertex-connectivity. A vertex is k-edge connected to itself for all $k>0$. The k-connectivity between a pair of vertices in an undirected graph is the equivalent of the minimum s-t cut. For a pair of vertices (u, v), let $C_{u,v}$ be the minimum cut between source u and sink v and let $C_{v,u}$ be the minimum cut between source v and sink u. Then the k-connectivity between (u, v) is the smallest cut of $C_{u,v}$ and $C_{v,u}$, i.e. $k = min(C_{u,v}, C_{v,u})$.

A graph $G = (V, E)$ is said to be k-edge-connected if there does not exist a set of $k-1$ edges such that their removal from the graph would make the graph G disconnected. Equivalently, a graph G is k-vertex-connected if there does not exist a set of $k-1$ vertices such that their removal from the graph would make G disconnected. Determining the edge connectivity of the graph is the equivalent of finding the global minimum cut of the graph. In this thesis we will focus on the k-edge connectivity.

A pair of vertices are k-edge-connected if there does not exist any edge sets C_E, where $|C_E|<k$, whose removal from the graph disconnects the two vertices. This means that there are at least k edge disjoint paths between the two vertices in the graph. Thus if a pair of vertices are k-edge-connected then they must also be l-edge-connected for any integer $l<k$.

A connected component of G is a maximal set $X \subseteq V$ such that all vertices in X are connected. X is called a k-edge-connected component if X is a maximal subset of V, and every pair of vertices $u, v \in X$ are k-edge-connected, i.e. if G is k-edge connected, then it is itself a k-edge component.

Previous studies on finding the k-edge connectivity of a graph have focused on finding the k-edge connected components in undirected graphs for some small integer k.

When $k = 1$ this is equivalent to finding the connected components of the graph, which can be done with a DFS in linear time. This is the same as finding the number of connected components, since $k = 1$ would represent the presence of an edge.

For $k = 2$, Tarjan presented an algorithm [10] to find 2-edge connected components of the graph using DFS in linear time.

For $k = 3$, there exists many practical linear time algorithms using DFS to solve the problem in linear time [8, 11, 12]. These approaches utilizes what is called the *DFS tree*, which is a partitioning of the edges of the graph into *tree edges* and *back edges*. Tree edges would be an edge in a tree graph. Where a back edge would be an edge which introduces an cycle in the tree, and each back edge in the tree can be seen as increasing the connectivity between vertices. The first linear time algorithm was developed by Galil and Italiano [4]. Their approach is quite complex and involves a reduction from 3-edge connectivity problem to the 3-vertex connectivity problem, then running Hopcroft and Tarjan's algorithm for 3-vertex-connectivity to solve the problem.

For a general k, it has been showed by Matula [7] that edge connectivity for a graph can be determined in $O(VE)$ time, and that given k in advance, testing if the graph is k-edge connected can be done in time $O(k|V|^2)$. In 1993 Nagamochi and Watanabe [9] gave an algorithm, given an integer k, for finding all k-edge connected components in a directed or undirected graph in time $O(V\, min(k, V, \sqrt{E})\, E)$. Nagamochi and Tbaraki [8] also showed a reduction proving that any k-edge connected undirected graph $G = (V, E)$, has a k-edge connected spanning sub-graph $G' = (V, E') s.t. |E'| = O(k|V|)$. They reduced the time complexity for undirected graphs to $O(|E| + k^2|V|^2)$. Based on the observation that if the cardinality of a cut $C = (S, V - S)\ s.t.\ |C|<k$, then the edge connectivity of any two vertices $u \in S$ and $v \in V - S$ is also strictly less than k. Then we know that for all pair of vertices $<u, v>$ where a precalculated cut exists $s.t.\ u \in S$ and $v \in V - S$, there is no need to consider such a pair since u and v must be in different components. Thus it is enough with $O(|V|)$ time for finding the minimum s-t cuts of vertex pairs.

1.6 Parallelization

Parallelization is a field in Computer Science where the goal is to reduce the running time of an algorithm by solving parts of the problem in concurrent operations. To do tasks in parallel is nothing new, the idea has been used through out history to solve tasks of great magnitude. It is not hard to imagine that a single person building a house would take a whole lot more time than 10 persons building a house. This is what parallel programing is about, to distribute the work load that is being processed among several processors.

Parallel processing is the processing of program instructions by dividing them among multiple processors, with the objective of running a program in less time. Not every task is suitable to be parallelized. It is the developers job to look for regions of code whose instructions can be shared among the processors. Usually this is focused on distributing the work of loops to the processors. In many programs, a calculation can depend on the result of some previous calculation to be finished before itself can be processed. In these situations it is critical to identify the dependencies and make the calculations in the right order to get the correct solution.

In an ideal world, adding more processors to distribute the computation load on, would mean that a program would run faster. But this is not always true, since adding more processors leads to more overhead in the system setting up the processors for the task, i.e. more administrative work for the program to distribute the work optimally. Which is kind of a paradox as you usually are looking for an optimum.

The program in this thesis will be run on a symmetric multiprocessor (SMP) system. SMP is a computer architecture running a single operating system with two or more homogeneous processors operating with a shared common memory. A processor can be assigned any task, and more processors can be added to improve the performance.

In the early years, different vendors of SMP systems had their own notation to specify how the work of a program were dispersed out to the individual processors of the system, as well as to enforce an ordering of accesses by different threads to shared data. These notations mainly took form through instructions or directives that could be added to programs written in a sequential language. This meant that one program written for one SMP, did not necessarily execute on another SMP by a different vendor. This is why we use openMP for parallelizing the program in this thesis.

1.6.1 OpenMP

OpenMP is a specification for a set of compiler directives, library routines, and environment variables jointly defined by a group of major computer hardware and software vendors, that makes it possible to parallelize code across hardware and software. It can be used to specify high-level parallelism in Fortran and C/C++ programs, making the programs both portable and scalable.

OpenMP was defined by the openMP ARB group in the latter half of the 1990s to provide a common means for programming a broad range of SMP architectures. The first version, consisting of a set of directives that could be used with Fortran, was introduced to the public in late 1997.

Chapter 2

A Simple Algorithm for Finding All k-Edge-Connected Components

In this chapter we will first present a naive approach for finding the k-edge connectivity of a graph. Then we will present the algorithm proposed by Tianhao Wang et al. [13] that not only finds the k-edge connectivity of the graph, but subsequently finds all k-edge connected components.

2.1 Brute Force

A naive approach for solving the k-edge connectivity for a graph $G = (V, E)$, is to find the smallest minimum s-t cut between every pair of vertices in G. From this statement it is possible to design an algorithm that uses two loops to cycle through every pair of vertices to find the smallest minimum s-t cut.

Algorithm 2.1 is an example of such an implementation. It loops through every pair of vertices and calculates the minimum cut, between the two vertices, using a maximum flow algorithm. Each time it compares the cut size with the previous cut size, keeping track of the smallest cut size it have encountered. When the algorithm finishes, processing all the vertices, it returns the smallest minimum s-t cut in G.

The running time of Algorithm 2.1 is $O(|V|^2F)$, where F is the running time of the minimum cut algorithm that is used. Using Edmonds-Karp maximum flow algorithm to compute the cut, the running time is $O(|V|^2 * (|V||E|^2))$. Can we do better? First thing to notice is that in an undirected graph the s-t cut between $<v, u>$ is equal to the s-t cut between $<u, v>$. This observation leads us to an algorithm with a slightly better running time $\frac{V^2}{2} * (VE^2)$, but it still gives an asymptotic running time of $O(V^2 * (VE^2))$. To get a better time complexity we have to do something more clever.

Every pair of vertices with a minimum cut larger or equal to some integer k, is k-

Algorithm 2.1 Naive Aglorithm

```
Input: Given graph $G$
  set $k = \infty$
  for $s \in V(G)$ do
      for $t \in V(G)$ do
          size = MAX FLOW($G$,$s$,$t$)
          if $k > size$ then $k = size$
          end if
      end for
  end for
  return $k$
```

edge connected in the graph G. By storing information about the cuts during the run of the algorithm, we could tell which subset of the vertices that belongs to the same k-edge component in G. In the next section we will introduce a structure for storing information about the cuts and how to exploit this structure for an improvement of the time complexity for finding the k-edge connectivity of G.

2.2 A Simple Algorithm

The paper by Tianhao Wang et al.[13] present a simple algorithm for finding all k- edge connected components for all k>0 in a directed or undirected, simple or multi-graph $G = (V, E)$. The algorithm calls an s-t maximum flow algorithm $2|V| - 2$ times, to compute the s-t minimum cut between pair of vertices in G. For each result that is returned by the algorithm, an auxiliary graph is constructed, storing information about the minimum s-t cuts. In effect, it is tracking the edge connectivity of G and all its sub-components. The time complexity for constructing the auxiliary graph is $O(F|V|)$, where F is the time complexity of the s-t max flow algorithm, i.e., for the Edmonds-Karp algorithm $F = O(|V||E|^2)$. Subsequently, any improvement of the time complexity of F would imply a reduction on the time complexity of the algorithm.

After its construction the auxiliary graph can be traversed in time $O(|V|)$ to determine the k-edge connected components of the graph for any value of $k > 0$, and the minimum s-t cut between every pair of vertices.

2.2.1 Auxiliary Graph

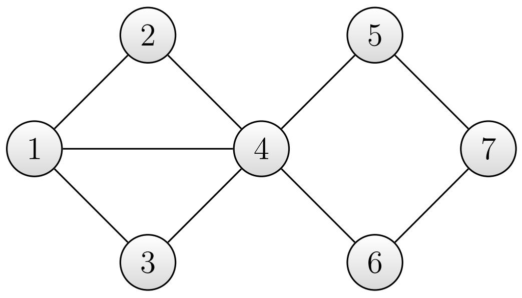
The algorithm by Tianhao Wang et al. [13] exploits the use of an auxiliary graph A to store information about the cuts between pair of vertices in a graph $G = (V, E)$. Initially A is the edgeless representation of G, with a vertex set $V_A = V$ and an empty edge set

$E_A = \{\}$. It is a forest where all the vertices of G are trees in the forest. Edges are added between the trees, ensuring that no cycles are introduced in A. Gradually transforming A from a forest into a tree.

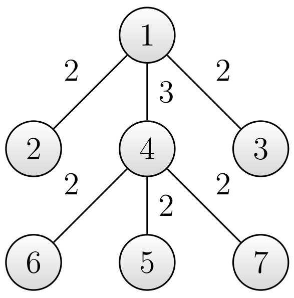
To make sure edges are only connecting trees to each other, a set $N \subseteq V$ of candidate vertices are kept for a vertex $s \in V$. N is initially a collection of all the trees in A, thus any vertex t can be chosen from $N/\{s\}$ for an edge $<s, t>$ to be added to A. If N is a *singleton* then there exists no vertices that can be connected to s, and the algorithm stops. The edge is assigned a weight equal to the size of the minimum cut $C = (S, T)$ between s and t. N is now divided into two partitions, to make sure that connected vertices are not in the same set. One partition N_s of candidate vertices for s where $N_s = S$, if $s \in S$, and the other partition N_t of candidate vertices for t where $N_t = T$. This process is repeated with each of the new partitions and the corresponding vertex as input.

The intuition for the construction of the auxiliary graph is based on the fact that the connectivity between vertices is transitive as proven in the paper by Tianhao Wang et al. [13]. Let $K(u, v)$ denote the connectivity between vertex u and v. Let $K(a, b) = x$, $K(b, c) = y$, $K(a, c) = z$, then $z \geq min(x, y)$. In Figure 2.1b the connectivity between pair of vertices from the graph depicted in Figure 2.1a have been stored in an auxiliary graph A. From the figure we can see that during the construction of A a cut was made between vertex 1 and 2 of size 2, and an edge was added between these two vertices in A. A cut was also made between vertex 1 and 4 of size 3, and an edge was added between these two vertices in A. We can then tell that the connectivity between 2 and 4 is equal to the smallest edge on the path between them, which is 2.

A set of k-edge-connected components in G, for $k>0$, is a partition of V where each k-edge connected component is the union of a collection of $k+1$-edge connected components in G, i.e. in Figure 2.1a the 2-edge connected component is $\{1, 2, 3, 4, 5, 6, 7\}$ and 3-edge connected components are $\{1, 4\}, \{2\}, \{3\}, \{5\}, \{6\}, \{7\}$. In this case the 2-edge connected component corresponds to exactly that of the union of all the 3-edge connected



(a) Connected Graph



(b) Auxiliary Graph of 2.1a

Figure 2.1

components.

Thanks to the nature of the k-edge connected components inherently hierarchical structure, it is possible to store information about the k-edge connected components in a compact form, for all $k>0$.

Given a graph G, let x be the smallest integer such that all the x-edge connected components of G are singletons. Let A_x be the edgeless spanning forest of A. Then the collection of all x-edge connected components of G is exactly that of all the connected vertex sets of A_x. Let A_{x-1} be the spanning forest of A obtained by adding edges with weights equal to $x-1$ from A to A_x. Then the collection of all $(x-1)$-edge connected components of G is exactly that of all the connected vertex sets of A_{x-1}. This leads to the general statement: for a $k<x$, let A_{k+1} be the spanning forest of A. Then the collection of all $(k+1)$-edge connected components of G is exactly that of all the connected vertex sets of A_{k+1}. Let A_k be the spanning forest of A obtained by adding edges with weights equal to k from A to A_{k+1}. Then the collection of all k-edge connected components of G is exactly that of all the connected vertex sets of A_k. This means that if we would like to know the k-edge connected components of G, we could remove all edges with weights $<k$ from A. Then all the connected vertex set in A, after the removal of edges with weight $<k$, are k-edge connected components in G.

Figure 2.1b illustrates an auxiliary graph A of the graph G depicted in Figure 2.1a. The 4-edge connected components of G is represented by the edgeless spanning forest of A, $\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\{7\}$. The 3-edge connected components of G is represented by the spanning forest of A induced by the edge-set consisting of edges of weights ≥ 3, $\{1,4\},\{2\},\{3\},\{5\},\{6\},\{7\}$. The 2-edge connected components of G is represented by the spanning forest of A induced by the edge-set consisting of edges of weights ≥ 2, $\{1,4,2,3,5,6,7\}$. Since we are only looking at connected graphs, the 1-edge connected component is equal to G itself.

The auxiliary graph is constructed by the procedure **CONSTRUCTION** shown in Algorithm 2.2. It is an essential part of the algorithm, which finds and stores information about the subsets of vertices that are k-edge-connected in G. Thus the algorithm not only finds the k-edge-connectivity for G, it determines the k-edge-connectivity for all connected components $V' \subseteq V$.

The auxiliary graph makes the algorithm applicable for a variety of cases. Not only looking for the connectivity of the entire graph, but also for gaining information regarding the connectivity between components of the graph.

In the next section we will describe the essence of the algorithm and how to construct the auxiliary graph.

2.2.2 Algorithm

Given a connected graph $G = (V, E)$, a start vertex s and a sink candidate vertex set $N = V$, the algorithm (2.2) randomly picks a vertex t as the sink from $N \setminus \{s\}$, and runs a maximum flow algorithm to find the minimum s-t cut.

If G is a directed graph it would run the maximum flow algorithm one more time, but with t as the source and s as the sink vertex. This is done since the size of the cut could differ. The two cuts, $C(S, T)$ and $C'(S', T')$, obtained are then compared. If C' is smaller then C, then $C(S, T) = C'(S', T')$. This is due to the fact that the connectivity between a pair of vertices in a directed graph is the smallest between the minimum s-t cut and minimum t-s cut.

With the obtained cut $C(S, T)$, the algorithm adds an edge $<s, t>$ to the auxiliary graph A with weight $|C|$, symbolizing the connectivity between s and t. In the final step the algorithm calls itself recursively, first with s as the start vertex and S as the vertex set, then with t as the start vertex and T as the vertex set. The algorithm terminates when S and T are reduced to singleton sets.

After the algorithm is finished constructing A, A can be traversed to obtain both the k-edge connected components and k- edge connectivity of G. To find the k-edge connected components of G we traverse all the edges in A and all edges with weight$<k$ is deleted. The connected vertex sets left in A then corresponds to the k-edge connected components in G. To find the k-edge connectivity of G, we traverse A and remember the lightest edge e of all edges in A. k is then equal to the edge with lightest weight in A, $k = w_e$. Both these searches can be done in $O(|V|)$ time, from the fact that A is a tree and has $|V| - 1$ number of edges.

Note that any s-t max flow algorithm can be used by the algorithm to determine the cut and subsequently the weights of the auxiliary graphs edges.

Algorithm 2.2 A Simple Algorithm for Finding All k-Edge-Connected Components

```
function CONSTRUCTION($G = (V, E)$,$s$,$N$)
    if $N = \{s\}$ then return
    end if
    Randomly pick a vertex $t$ from $N \setminus \{s\}$
    $(x, S, T)$ = S-T MAX FLOW($G$,$s$,$t$)
    $(x', S', T')$ = S-T MAX FLOW($G$,$t$,$s$)
    if $x'<x$ then $x = x'$, $S = S'$, $T = T'$
    end if
    Add edge $<s, t>$ with weight $x$ to $A$
    CONSTRUCTION($G$,$s$,$N \bigcap S$)
    CONSTRUCTION($G$,$t$,$N \bigcap T$)
end function
```

The algorithm calls 2 maximum flow algorithms for each time CONSTRUCTION is called. Each time CONSTRUCTION is called a vertex is removed from the set of candidate vertices for the sink, until only a singleton set is left. Thus CONSTRUCTION runs $|V| - 1$ times. The algorithm computes $2(|V| - 1)$ maximum flow in total, giving a running time of $O(F|V|)$ where F is the time complexity of the maximum flow algorithm used. If the input graph is an undirected graph it is enough running 1 maximum flow instead of 2, which would reduce the number of maximum flows calculations to $|V| - 1$.

In the next chapter we will present our implementation of the algorithm and challenges that were met during the implementation.

Chapter 3

Sequential Implementation

In this chapter we will present choices that were made when implementing the algorithm by Tianhao Wang et al.[13] for computing all k-edge connected components. We will then move on to the implementation of the algorithm, that is split into two parts. We will be talking about the implementation for finding the minimum s-t cut and the construction of the auxiliary graph. At the end of this chapter, a naive approach for solving k-edge connectivity is implemented. The algorithm checks the results stored in the auxiliary graph for correctness.

3.1 Implementing the Algorithm

The algorithm stores information about each minimum s-t cut in an auxiliary graph. To construct the auxiliary graph, a maximum flow algorithm is utilized for finding the flow between pair of vertices in the input graph. There exists a wide variety of maximum flow algorithms that can be implemented, and we will explain our implementation in this chapter. The maximum flow algorithm will be used in combination with a collection of vertex sets for tracking which vertices to run the maximum flow between. We then extract the cut set from the result of the maximum flow with a depth first search. Each time we are dividing the vertices into two cut sets, one set with vertices from the source side of the cut and one set with vertices from the sink side. We add an edge to the auxiliary graph and then update the vertex set using the cut sets returned from the depth first search. The information regarding the k-edge connectivity and k-edge connected components of the input graph can then be retrieved from the auxiliary graph.

We will first introduce the maximum flow implementation before we explain how the algorithm works and how it constructs the auxiliary graph. Finally we will describe how to retrieve information from the auxiliary graph regarding the k-edge connectivity and k-edge connected components.

3.1.1 Minimum s-t cut

Algorithm 3.1 Minimum s-t cut

```
Input: Given a flow graph $G = (V, E)$, a source vertex $s$, target vertex $t$
  function MINIMUM S-T CUT($G$, $s$, $t$)                          ▷ Start maximum flow
      set $flow = 0$, vertex set $S = \{\}$, vertex set $T = \{\}$
      while BFS reaches $t$ through edges with residual capacity $> 0$ do
          Set $x = t$
          while $x \neq s$ do
              choose a neighbour $y$ of $x$ closer to the source
              set capacity of edge $(x, y) += 1$
              set capacity of edge $(y, x) -= 1$
              set $x = y$
          end while
      end while                                                    ▷ End maximum flow
      push $s$ onto stack $L$                                      ▷ Start depth first search
      mark $s$
      while $L$ not empty do
          set $x$ to top element of $L$
          remove top element from $L$
          for all neighbours $y$ of $x$ do
              if edge $(x, y)$ capacity $> 0$ and $y$ not marked then
                  push $y$ onto stack $L$
                  mark $y$
              end if
          end for
      end while                                                    ▷ End depth first search
      for all $y \in V$ do
          if $y$ is marked and $y \in N_s$ then Add $y$ to $S$
          else if  $y \in N_s$  then Add $y$ to $T$
          end if
      end for
      return $flow$, $S$, $T$
  end function
```

To find the minimum s-t cut we implement a maximum flow algorithm and a depth first search algorithm. There are several variations for implementing maximum flow. In this thesis we have chosen to implement Edmonds–Karp algorithm [2] for computing the maximum flow. The algorithm uses a breadth first search (BFS) to determine the next augmenting path to choose, starting with the shortest path from a source vertex to a target vertex, increasing in distance until there exist no paths with capacity > 0 leading to the sink.

In this thesis we are only interested in how many edges that are in the cut, and the maximum flow algorithm reflects this in its implementation. Algorithm 3.1 can process edges with weights > 1, but it is not done in an optimal way.

Given a flow graph $G = (V, E)$, a source vertex s, target vertex t, and vertex set $N_s \subseteq V$, our implementation of the maximum flow algorithm, shown with pseudo code in Algorithm 3.1, starts with an initial $flow$ of 0. A BFS is run from s, traversing only edges with residual capacity > 0, either stopping when it reaches t or when all reachable vertices have been visited. If t was not reached then there exist no augmenting path to t from s, a path where all edges have residual capacity > 0, and the maximum flow algorithm is done. When t is reached by the BFS then every vertex v that has been visited by the BFS has an integer, $dist(v) \geq 0$, indicating the distance from s assigned to it. We then start from t, working our way back, updating capacities. Let $x = t$. While x is not equal to s, we find a neighbour vertex y of x with distance closer to s, such that $dist(y) = dist(x) - 1$. We update the edges, $(x, y)+ = 1$ and $(y, x)- = 1$, and repeat the process with $x = y$. When s is reached we run the BFS again, looking for paths to t.

What we really want to find is the minimum s-t cut between the vertices. We already know the size of the cut from the maximum flow algorithm, but not the cut sets. It is possible to find the cut sets using a depth first search algorithm. Starting the search from s, we mark this vertex as visited, and push s onto a stack L. As long as there are elements on the stack L to be processed: let x be the top element of L and remove the top element from L. Iterate through all neighbours of x. If a neighbour y exist, such that the edge (x, y) has capacity > 0 and y is not marked, then push y onto the stack and mark y as visited. Repeat the process until L is empty. When the search is finished, all vertices that have been marked belongs to the source side S of the cut. This comes from the fact that all edges crossing between the cut sets have a residual capacity of 0 after the maximum flow have finished. Thus it is not possible to cross the cut starting from the source vertex, traversing only edges with residual capacity > 0.

The final part of the maximum flow algorithm is to add the marked vertices to the vertex set S, and unmarked vertices to the vertex set T. Then the algorithm returns the *flow* along with the two cut sets S and T.

The running time of the implementation is $O(|V||E|^2)$ for the Edmonds–Karp maximum flow algorithm [2]. The depth first search algorithm for finding the cut sets runs in time $O(|V| + |E|)$. In total, the running time of the implemented algorithm for finding the s-t minimum cut is $O(|V||E|^2) + O(|V| + |E|)$, or simply $O(|V||E|^2)$.

In the next section we will introduce the implementation of the algorithm and how the auxiliary graph is constructed with the maximum flow algorithm as a subroutine.

Algorithm 3.2 The Algorithm

```
Input: Given graph $G = (V, E)$
  Let $A = (V, E')$ where $E' = \{\}$                    ▷ The auxiliary graph
  Choose a root vertex for $A$
  for all $x \in V$ do $N_x = \{\}$
  end for
  $N_{root} = V$
  add $root$ to $Q$
  while $Q$ is not empty do
      set $s$ to first element of $Q$
      if size of $N_s < 2$ then
          remove first element of $Q$
      else
          set $t$ to be a vertex from $N_s \setminus \{s\}$
          construct a flow graph $F$ of $G$
          $C(c, S, T)$ = Minimum s-t cut($F$,$s$,$t$)
          add the directed edge $(s, t)$ to $E'$ with weight $= c$
          add $t$ to $Q$
          for all $x \in N_s$ do
              if $x \in T$ then
                  add $x$ to $N_t$
                  remove $x$ from $N_s$
              end if
          end for
      end if
  end while
```

3.1.2 Constructing the Auxiliary graph

It can be argued that the auxiliary graph is the most important structure of the algorithm, as it is in a sense the essence of the algorithm. The implemented auxiliary graph is a tree structure, but unlike the algorithm by Tianhao Wang et al.[13], it is a rooted tree. This is partially due to the necessity of a vertex set that keeps track of the vertices that can be connected in A, and partially to make it more convenient to parallelize the code. The construction of A is shown in Algorithm 3.2.

Given a graph $G = (V, E)$, Algorithm 3.2 starts by initializing the auxiliary graph A to have the same vertex set as G and an empty edge set, $A = (V, E')$ $s.t$ $E' = \{\}$. A vertex is then chosen from V to be the root vertex for A. For every vertex $x \in V$, a set N_x is maintained for tracking which vertices are candidates to be the target vertex in the minimum s-t cut between the two vertices. Initially N is an empty set for every vertex in V. We then start by letting N for the root vertex to be equal to V, $N_{root} = V$. This can be done since A initially is a forest where all vertices are isolated (trees), thus adding an edge between any pair of vertices can not create a cycle.

We are now ready to start the algorithm. To keep track of which vertices that are viable as a source vertex by the algorithm, a Queue Q is used. The only vertex that can be processed to start with is the root, so we add the root to Q. As long as Q is not empty we do the following: we set s to be the the first element on Q. If $Q_s < 2$ then there exist no vertex to be chosen as sink vertex for the maximum flow algorithm with s as source, so we remove s from Q. If $Q_s > 1$, let t be a vertex in $Q_s \setminus s$. We then construct a flow graph F from G, and run the maximum flow algorithm with flow graph F, source s and sink t. The maximum flow algorithm returns the cut sets S and T along with size of the flow c. To store the information about the retrieved cut between s and t, a directed edge (s, t) is added to the edge set E' from A and t is added to Q. It now becomes crucial to correctly update the candidate vertex set for s and t respectively. To avoid corrupting the information and introducing cycles in A the cut sets returned must be processed. We are only interested in vertices in N_s, as described in the algorithm by Tianhao Wang et al.[13], therefore we only add a vertex x to N_t, the candidate vertex set for t, if $x \in T$ and $x \in N_s$. Then we remove x from N_s as it is no longer a candidate vertex for s. We do this for all vertices in T. The algorithm continues until there are no more vertices in Q.

Algorithm 3.2 runs the maximum flow algorithm a total of $O(|V|)$ times. Combining this with the run time of the maximum flow algorithm we get a total run time of $O(|V|^2|E|^2)$.

Now that the auxiliary graph is constructed, it is possible to extract information about the k-edge connectivity and k-edge connected components of G. In the next subsection we will describe how this information can be retrieved.

3.1.3 Retrieve Information from the Auxiliary Graph

After the auxiliary graph A has been constructed we can retrieve information about the k-edge connectivity and k-edge connected components of the input graph $G = (V, E)$. This information can easily be extracted by simply traversing A, Finding the k-edge connected component of G can be done by deleting edges in A with weight less than k, which can be done in $O(|V|)$ time since A is a tree. The connected components left in A, then corresponds to the k-connected components of G. Algorithm 3.3 shows how the k-edge connectivity between two vertices in G can be retrieved from A. It first finds the path between the two vertices and returns the weight of the edge with the lightest weight on that path. The weight returned indicates the k-edge connectivity between the two vertices. This algorithm will also be used by the algorithm that verifies that the information stored in A is correct.

Given auxiliary graph $A = (V, E')$, the root vertex of A, $s \in V$ and $t \in V$, Algorithm 3.3 finds k by doing a depth first search. Initially $k = \infty$. It needs two stacks to track the path from the root to each of the target vertices *sStack* for path to s and *tStack* for path to t. Since A is a directed tree every search has to start at the root to be able to search the complete tree, thus the root is added to both the *sStack* and the *tStack*.

The algorithm then starts traversing A, adding a vertex to the two stacks every time it advances away from the root and removing a vertex from the two stacks every time it moves closer to the root. When we find a vertex corresponding to either s or t, the stack corresponding to this vertex is blocked from further changes. The algorithm then continues searching for the other vertex in the same manner, adding a vertex to its stack every time it advances away from the root and removing a vertex when it moves closer to the root. Stopping when the last vertex is found. The two stacks now contains the path between the root and the corresponding vertex. At some point these two paths must intersect each other since A is a tree. If we find the intersection we have the path between the two vertices in A. In fact every edge on the path to the intersection is on the path between the two vertices.

The algorithm finds the intersection by comparing the distances of the vertices from the root, remembering the lightest weight of an edge it has encountered along the way. If the paths from the two vertices are at different distance from the root, then we remove the top vertex from the stack which is furthest away from the root. Then we compare k to the weight of the edge between the removed vertex u and the top vertex v of the stack. If k is larger then the edge (v, u) weight, set k = weight of (v, u). If the paths from the two vertices to the root is at the same distance from the root we compare the top vertex of the two stacks. If they are different we are not at the intersection, so we remove the top vertex from both stacks. We then compare k to the weight of the edges, setting k equal to the lightest weight if k is larger. When the top vertex of both stacks are the same, we have found the intersection. The k we are left with then corresponds to

Algorithm 3.3 Retrieve k-edge Connectivity

```
Input: Auxiliary graph $A = (V, E')$, root vertex of $A$, $s \in V$, $t \in V$
function GET-K($A$, root, $s$, $t$)                          ▷ Start depth first search
    $k = \infty$
    push root onto stack sStack
    push root onto stack tStack
    mark root
    while (tStack top $\neq t$ or sStack top $\neq s$) and dfsStack not empty do
        set $x$ to top element of dfsStack
        if there exists a neighbour $y$ of $x$ not marked then
            mark $y$
            if top element of sStack $\neq s$ then push $y$ onto stack tStack
            end if
            if top element of tStack $\neq t$ then push $y$ onto stack tStack
            end if
        else
            if top element of sStack $\neq s$ then remove top element from sStack
            end if
            if top element of tStack $\neq t$ then remove top element from tStack
            end if
        end if
    end while                                                ▷ End depth first search
    while sStack not empty and tStack not empty do           ▷ Find $k$-edge connectivity from $A$
        if sStack size < tStack size then
            set $x$ to top element of tStack
            remove top element from tStack
            set $y$ to top element of tStack
            if weight of edge $(y, x) \in A < k$ then $k$ = weight $(y, x)$
            end if
        else if tStack size < sStack size then
            set $x$ to top element of sStack
            remove top element from sStack
            set $y$ to top element of sStack
            if weight of edge $(y, x) \in A < k$ then $k$ = weight $(y, x)$
            end if
        else if tStack size = sStack size then
            set $x$ to top element of sStack
            set $y$ to top element of tStack
            if $x = y$ then sStack = {} and tStack = {}
            else remove top element from sStack and tStack
                set $x_2$ to top element of sStack
                set $y_2$ to top element of tStack
                if weight of edge $(x_2, x) \in A < k$ then $k$ = weight $(x_2, x)$
                end if
                if weight of edge $(y_2, y) \in A < k$ then $k$ = weight $(y_2, y)$
                end if
            end if
        end if
    end while                                  ▷ End Find $k$-edge connectivity from $A$
    return $k$
end function
```

the k-edge connectivity between the two vertices, and it is returned by the algorithm.

The running time of the algorithm is $O(|V|)$. It simply traverses the edges of a tree.

In the next section, we will describe the an algorithm that verifies the information stored in the auxiliary graph, using Algorithm 3.3 as a subroutine.

3.2 Correctness Check

To check the correctness of the auxiliary graph that were constructed above, a naive algorithm for finding the k-edge connectivity between every pair of vertices were implemented. The cut size between a pair of vertices is calculated with a maximum flow algorithm, seen in algorithm 3.4 below. A depth first search algorithm is then used to retrieve the cut size stored in the auxiliary graph, and the size of the two cuts are compared.

The correctness of the cut in the graph is checked for every pair of vertices s and t. It is enough to verify that the cut is correct in one direction from s to t, since the implemented algorithm only operates on undirected graphs.

Given a graph $G = (V, E)$, Auxiliary graph $A = (V, E')$ and the root vertex of A, Algorithm 3.4 verifies the correctness of A for every pair of vertices s and t in V. The maximum flow between s and t is calculated and the size of the flow is then compared to the k-edge connectivity, between the two vertices, stored in the auxiliary graph. Algorithm 3.3 is used as a subroutine to retrieve the k-edge connectivity. If the the two results does not match a message is printed.

We have $O(|V|^2)$ pair of vertices to compare the maximum flow and cut size between. The maximum flow algorithms running time is $O(|V||E|^2)$ and to find the cut in the auxiliary graph with a depth first search takes $O(|V| + |E|)$ time. Since the auxiliary graph is a tree, we have $|V| = |E|$, the running time of the depth first search can be written $O(|V|)$. This brings us to a total of $O(|V|^2((|V||E|^2) + (|V|)) = O(|V|^3|E|^2)$.

Running the check for correctness can take a long time. To help counter this time consuming operation, the check was parallelized with an openMP for loop in the implementation.

In the next Chapter we will introduce the parallelization of A.

Algorithm 3.4 Maximum Flow

```
Input: Given graph $G = (V, E)$, Auxiliary graph $A = (V, E')$, $root$ vertex of $A$
  for all $s \in V$ do
      for all $t \in V \setminus \{s\}$ do
          set $flow = 0$                                         ▷ Start maximum flow
          while BFS reaches $t$ through edges with residual capacity $> 0$ do
              Set $x = t$
              while $x \neq s$ do
                  choose a neighbour $y$ of $x$ closer to the source
                  set capacity of edge $(x, y) += 1$
                  set capacity of edge $(y, x) -= 1$
                  set $x = y$
              end while
          end while                                              ▷ End maximum flow
          $k =$ Get-k($A$, $root$, $s$, $t$)
          if $k \neq flow$ then $PrintError : k \neq flow$
          end if
      end for
  end for
```

Chapter 4

Parallel Algorithm

In this chapter we will describe how we identified potential areas in the algorithm for parallelization, and then give a short argumentation for the areas we choose for implementing a parallel algorithm. We will then describe how we did the implementation of the parallel algorithm and the decisions taken during the design.

4.1 Identifying Areas to Parallelize

The algorithm described in Chapter 2 can be divided into two parts. Where the first part of the algorithm is the construction of the auxiliary graph, that can be done with a fixed amount of calls to an maximum flow algorithm. This leads us to the second part of the algorithm which is the maximum flow algorithm. Both of these parts, comprising the algorithm, would be possible to parallelize.

Parallelizing a maximum flow algorithm is a complicated task to implement. This is due to the fact that the maximum flow algorithm often are implemented as an augmenting path algorithm, which pushes flow along edges from the source to the sink, making changes to the edges as it goes. This in turn would have to be handled by the algorithm when parallelized, leading to the need for locks to lock down edges as they are processed.

In this thesis we have chosen not to parallelize the maximum flow algorithm, due to the reason explained above, but instead we have set our focus on how to parallelize the construction of the Auxiliary graph. The first observation that were made is that each thread, in an parallelized algorithm, can run its own instance of the maximum flow algorithm for the computation of the connectivity between two vertices in a graph. The second observation is that adding an edge in the auxiliary graph can be done in parallel given that the data structure is fixed and not a dynamic structure.

The auxiliary graph demands a certain order to which vertices can be connected in it. First of all it is important that it is a tree structure for it to have the desired properties.

Secondly, an edge can only exist between the two vertices in the auxiliary graph after the maximum flow algorithm have processed the vertices and a cut between the two vertices have been found. This makes the process of constructing the auxiliary graph not too optimal for parallelization, as the process involves starting with one task that must be solved before two new tasks are ready for processing. Given large enough graphs this should not be a problem as we should be able to divide the graph up pretty quickly.

When distributing the tasks it is critical to not hand out the same task twice, thus the distribution of tasks becomes a process where we must use locks to ensure the safeness of the process. The same lock must also be used when adding tasks for distribution, as an insertion removal.

In the next section we will describe how we parallelized the algorithm and present our implementation.

4.2 Parallel Implementation

In the implementation of the algorithm there are parts of the sequential algorithm that can be reused without needing any form of alteration. The parts that can be reused is the minimum s-t cut algorithm 3.1, the algorithm for retrieval of the cut from the auxiliary graph 3.3 and the algorithm for correctness check of the auxiliary graph 3.4.

The minimum s-t cut algorithm will be used by each of the individual threads to calculate the cut between the pair of vertices that the thread is processing at that time, running the minimum s-t cut algorithm locally. Thus there is no interference from the other threads made to this process.

When we want to retrieve the cut from the auxiliary graph there is no alteration made to the data structure itself, we only read the information stored in the auxiliary graph. This process can also be done without concern of interference from the other threads.

The correctness check can also be used as this can be run after the computation is done in a sequential fashion, or as stated in Section 3.2 it can be parallelized with out concern using the openMP for loop.

We will now describe how the construction of the auxiliary graph were parallelized and implemented.

4.2.1 Parallel Algorithm for Constructing the Auxiliary Graph

The parallel construction of the auxiliary graph was implemented with the idea that the first task would be computed, giving two new task to compute that again would yield two new tasks for each the previous task and so on. This starting procedure leads to a limited amount of task to be available at the start, but the number of tasks grows quickly.

Given a graph $G = (V, E)$, Algorithm 4.1 starts by initializing the auxiliary graph A

to have the same vertex set as G and an empty edge set, $A = (V, E')\ s.t\ E' = \{\}$. It then chooses a vertex root from V, with highest degree, to be the root vertex for A. For every vertex $x \in V$, a set N_x is maintained for tracking which vertices are candidates to be the target vertex in the minimum s-t cut between the two vertices. Initially N is an empty set for every vertex in V. We then start by letting N for the root vertex to be equal to V, $N_{root} = V$. This can be done since A initially is a forest where all vertices are isolated (trees), thus adding an edge between any pair of vertices can not create a cycle.

We add the root to a queue Q then initialize the private variable s for every thread p to be $s_p = -1$, the reason for this will be made clear shortly. As long as A is not a tree, we do the following: If Q contains tasks to be handed out for processing. One of the tasks is assigned to a thread, if the thread does not already have a task. We continue in this manner until either there are no more tasks on Q to be divided out or there are no more unoccupied threads. If the vertex set N_{s_p} of candidate vertices for vertex s_p contains less than 2 elements this means vertex $s - p$ is processed, then we free up the thread working this vertex and set its private variable $s_p = -1$ so to not point to any vertex. If vertex set N_{s_p} is greater than 1 and the threads private variable s_p is pointing to a vertex in V then we have additional tasks to complete.

Set the private variable t_p to be vertex from $N_{s_p} \setminus \{s_p\}$, with highest degree, for a thread p and construct a private flow graph F_p from G. The thread then runs minimum s-t cut algorithm 3.1 calculating and returning the cut. With the returned cut the edge (s_p, t_p) is added to the edge E' of A with weight equal to the size of the cut. Now we must update the candidate vertex set for s_p and t_p respectively. The vertices to be divided are those present in N_{s_p}. For every vertex $x \in T$, if $x \in N_{s_p}$ add x to N_{t_p} then remove x from N_{s_p} as it is now the candidate vertex for t_p and not s_p. At last, after the vertex sets have been updated we add t_p to Q as it is now ready to be processed. When A is a tree we are done.

A critical part of the algorithm is the adding and removing of tasks from Q. It is important that this happens in an orderly fashion as to not interfere with the algorithms calculations.

One of the challenges with the implementation is to make sure that there is a task for each of the threads to process. This is what the algorithm tries to counter with picking vertices of higher degree in an attempt to produce larger cut sets when running the minimum cut algorithm, so that larger tasks can be divided among the threads. A problem with this solution is that a vertex with high degree can have mostly neighbours of degree 1 resulting in a lot of tasks involving none computation, thus countering the cascading effect of a task creating two new tasks. This implies that depending on the properties of the graph having more threads does not necessarily means a faster running algorithm.

Testing have to be done in order to assert the assumptions made in the implementation of the algorithm, that more threads can process a graph faster. In order to assert the assumptions made in the implementation of the algorithm we have to run tests on the parallel algorithm and analyse the results which we will do in the next chapter.

Algorithm 4.1 The Parallel Algorithm

```
Input: Given graph $G = (V, E)$
  Let $A = (V, E')$ where $E' = \{\}$                    ▷ The auxiliary graph
  Choose a vertex from $V$ with highest degree to be root vertex for $A$
  for all $x \in V$ do $N_x = \{\}$
  end for
  $N_{root} = V$
  add $root$ to $Q$
  Initialize the private variable $s_p$ for each thread $p$ to be $-1$
  while $|E'| < |V|$ do
      if $Q$ not empty then Only one thread $p$ can enter at any time
          set thread $p$'s private variable $s_p$ to first element of $Q$
      end if
      if size of $N_{s_p} < 2$ then
          set $s_p = -1$
      else if $s_p \in V$ then
          set thread $p$'s private variable $t_p$ to be a vertex from $N_{s_p} \setminus \{s_p\}$ with highest degree
          construct a private flow graph $F_p$ of $G$ for thread $p$
          $C_p(c_p, S_p, T_p) =$ MINIMUM S-T CUT($F_p$,$s_p$,$t_p$)
          add the directed edge $(s_p, t_p)$ to $E'$ with weight $= c_p$
          for all $x_p \in N_{s_p}$ do
              if $x_p \in T_p$ then
                  add $x_p$ to $N_{t_p}$
                  remove $x_p$ from $N_{s_p}$
              end if
          end for
          add $t_p$ to $Q$
      end if
  end while
```

Chapter 5

Parallel Algorithm Analysis

In this chapter we will delve into the results collected from running the algorithm. We will present the measured running time for both the sequential and parallel algorithm, before we compare the variation of the running time between the sequential and parallel algorithm. Next we will measure the running times of the parallel algorithm against the running time of the sequential algorithm, giving insight into the potential reduction in the running time provided more threads or in other words more machine power. Before we proceed with presenting the measures of the efficiency of the parallel algorithm with respect to the number of threads used. But first we will give a brief introduction to previous work done on the subject before we set up some test conditions for the results made.

5.1 Previous Work

Computing both the k-edge components of a graph and finding the k-edge connectivity of the graph are two well studied subjects. These subjects have mainly been studied separately in the previous work done, with the solution of one subject have lead to a partial solution to the other, and vica versa. As mentioned in Section 1.5 Most of these studies has been based on knowing some predetermined fact about the problem being solved.

A study done on something similar to the approach we have taken is the *Gomory-Hu Tree*, introduced by R. E. Gomory and T. C. Hu in 1961 [5], where the minimum s-t cut between every pair of vertices is stored in a tree structure. The approach is based on doing $|V| - 1$ maximum flow computations and storing the corresponding cuts in an auxiliary graph. Thus it is possible to find the k-edge connectivity in the Gomory-Hu tree, but not the k-edge connected components as it would require a different design for storing the cuts in a tree structure.

The uniqueness of the algorithm implemented in this thesis, is that not only does it find and store information about the k-edge connected components of the graph, but it also finds the k-edge connectivity between all pairs of vertices in the graph. All made possible by the auxiliary graph constructed.

5.2 Test Conditions

The algorithm have been evaluated on a variety of graphs from the The UF Sparse Matrix Collection by Tim Davis[1]. This is a large collection of graphs used for benchmarks testing for algorithms and during development of IT solutions. The graphs used for testing differ in size and graph density. To conduct the tests, the graphs described in Table 5.1 were selected on the criteria that they should vary in size and density. Starting with sparse and small graphs at the top, then denser and bigger graphs at the end. Except for power which is the sparsest graph in the graph collection.

To run the algorithms a super computer situated at the Department of Informatics, at the University of Bergen, were administrated as the test environment. All the runs were executed and timed on the computer to ensure that the run time environment were identical for each execution of the algorithm. Thus no biases could arise due to different hardware or software.

The sequential algorithm were run on the different graphs timing the running time for each run. Then the parallel algorithm were run with different number of threads on each graph, all while timing the running time. The number of threads used were set in a scale to the power of 2, with a range from 1 to 32, doubling the potential machine power each time the thread count was increased. When scaling the number of threads in this manner we would expect the result of the running time for an efficient algorithm to exhibit, to some extent, the same traits. Leading to the work load of each thread becoming smaller and smaller as the number of threads are increased, an efficient algorithm would be able to optimally distribute the workload thus becoming even more efficient. When timing the algorithm, we only evaluate the algorithm itself ignoring initial setup of data structures. Seeing that we are interested in the potential reduction on the running time, by utilizing more threads or machine power.

In the next section we will present the running times measured during the test runs.

5.3 Results and Analysis

Our hypothesis was that for smaller graphs the parallel algorithm would have running times close to the sequential algorithm, and that for larger graphs the expected running time of the parallel algorithm would outperform the sequential algorithm, at an increased rate when the number of threads were increased.

Graph Name	Number of Vertices	Number of Edges
bcspwr05	443	1033
bcspwr06	1454	3377
bcspwr08	1624	3837
G49	3000	6000
power	4941	6594
c-20	2921	12803
c-23	3969	17524
c-32	5975	30223

Table 5.1: Graph Information

Threads	bcspwr05	bcspwr06	bcspwr08	G49	power	c-20	c-23	c-32
sequential	0.01946	0.20619	0.27451	0.442992	2.62954	2.40613	3.76468	7.718381
1	0.01897	0.20292	0.28213	0.13330	2.87303	2.11639	3.35760	6.57627
2	0.01152	0.11860	0.19366	0.20195	1.59667	1.85601	2.46516	5.15860
4	0.00852	0.09282	0.09906	0.304398	0.89373	1.67829	2.20961	5.47688
8	0.0074	0.05185	0.06699	0.29588	1.17854	1.90471	2.25752	5.58682
16	0.011792	0.06677	0.08695	0.51916	0.49330	2.44702	3.16193	7.24083
32	0.016706	0.12787	0.1806	0.98011	0.8377	5.0904	6.75289	15.06856

Table 5.2: Running Time in Seconds

The running time is a measure of the amount of resources the algorithm requires to run. In Table 5.2 the running time of the sequential and the parallel algorithm is displayed, measured in seconds. In the left most column of the table, the number of threads used during the run time of the algorithm is shown or "sequential" for the row with reference to the sequential algorithm. The top row of the table describes which graph the running time corresponds to. To analyse the data it is not ideal with a table as it does not convey allot of information.

Figure 5.1 is a visualization of the data in the table, plotted in a line chart. In the plot we can see that the running time becomes larger as the size of the graph increases, but it is hard to extract more information than this from the chart. At the start of the plot it is hard to distinguish any significant details about the different algorithms, and we would have to enlarge that area if we wanted to analyse it.

From the results, the first thing to notice is that the running time for the parallel algorithm and the sequential algorithm did not differ to much from each other. In Figure 5.1 we see that the parallel algorithm were in some cases vastly more time consuming than the sequential algorithm, as were especially noticeable when the number of threads is set to 32, except for in the case of the power graph. When the thread count were less than this, the parallel algorithm seems to perform marginally better then sequential algorithm. There are no leading thread count based on the running time that made it stand out as the preferred number of threads. In fact almost all the thread counts had at some point the best performing running time.

Ideally the running time would go down for each thread the algorithm have available, but this behavior is none existing in our implementation. Some latency must be expected when spawning extra threads. Given a thread needs to start its own copy of the algorithm, initialize its own private data structures and variables before starting to compute. Overhead is important to take into account when parallelizing since it can lead to an increase of the running time.

A semi-log of Figure 5.1 helps to smooth out the large variation of the plots in the figure. In a semi-log plot the variation in running time becomes smaller, but the actual plot itself becomes clearer. This is due to the fact that we disregard the actual time scale of the running times in favor of the log scale of the running times. This means that the difference in running time will appear smaller then they actually are, as we have decreased the time scale. But we have attained a clearer insight into the little details of the running times.

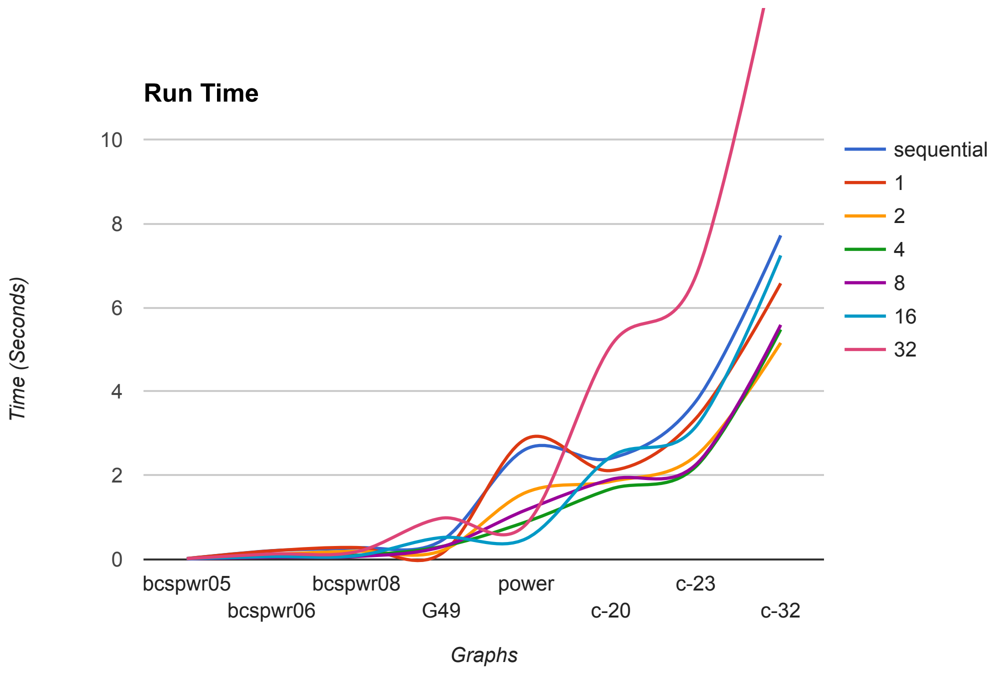


Figure 5.1: Timing of the Algorithms

Figure 5.2 is a semi-log plot of the running times in Figure 5.1. In the plot we see that the parallel algorithm for the most part have a better running time then the sequential algorithm, especially in regards to the thread count of $2, 4$ and 8. But we also see that even though the thread count were more than doubled, the running time still did not decrease. Hinting that some of the threads were ideal during some part of the execution of the algorithm, which can be seen in the two cases with thread count of 16 and 32. In addition, we observed that the two cases varies the most in its performance, that is they do not seem to steadily outperform the other running times. The running time of the parallel algorithm were not as ideal as we expected.

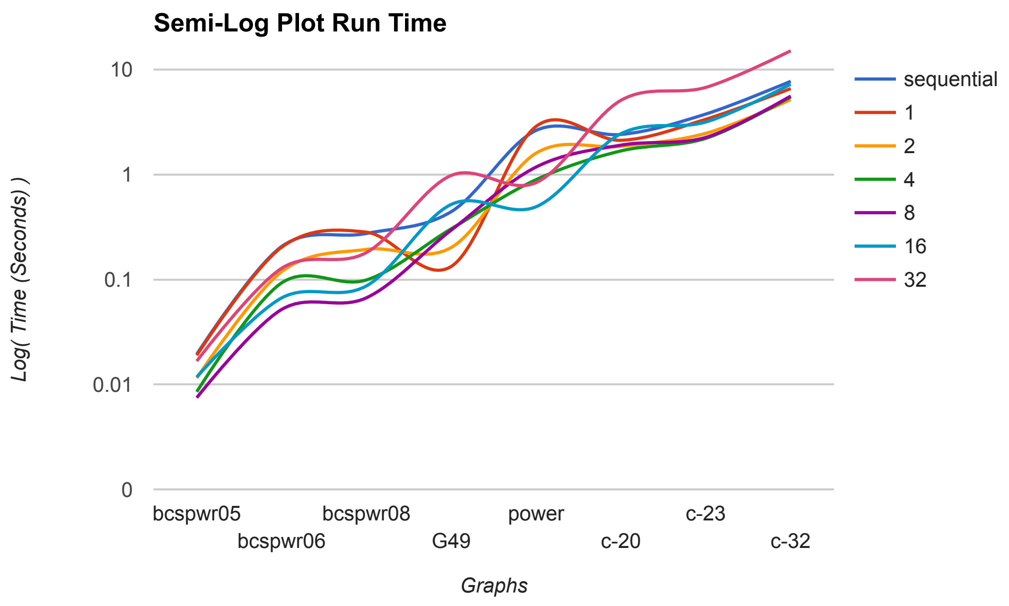


Figure 5.2: Log Plot Of the Timing

The *speed up* for a parallel algorithm is computed from the fastest known sequential algorithm T_s over the parallel algorithm T_p, where p is the number of threads. Both algorithms should be given the same work when comparing the algorithm, in this paper the same graph. The speed up S is then defined:

$$S = \frac{T_s}{T_p}.$$

It is a ratio for the sequential running time up against the parallel running time. In Figure 5.3, the speed up for the parallel algorithm have been plotted in a chart. On the x-axis we have the number of threads, and the line represents the speed up for different number of threads for solving the same graph. Ideally the desired speed up would be equal to the number of threads sharing the work, but this is infrequently achieved. The speed up reveals information of how well the parallel algorithm compares to the sequential algorithm.

To analyse the discrepancies between our hypothesis and results, we will now have a look at the speed plot in Figure 5.3. In the plot we can see that there are some outliers where the parallel algorithm actually performed pretty decent, especially for the power graph with a thread count of 16 where the speed up were more than 5 times better then the sequential. When 4 threads were used, some speed up were achieved for the smaller sparser graphs, while the bigger denser graphs had consistently marginally speed up or none. The algorithm using 8 threads had the most consecutive result. It outperformed the other thread counts for bcspwr05, bcspwr06 and bcspwr08. And it performed close to as good as the best performing thread counts on c-20, c-23 and c-32. The only deviations for the performance of the algorithm running on 8 threads where for the power and g49 graphs. An outlier to take note on is that there were a speed up of around 3 for graph G49 with one thread, which is strange since this should be close to the equivalent of running the sequential algorithm.

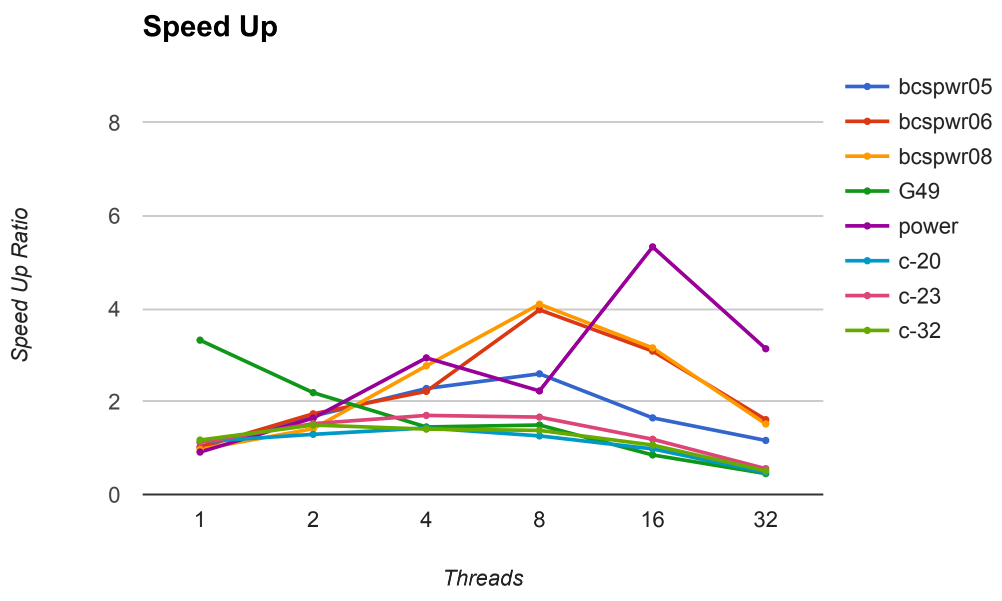


Figure 5.3: Speed Up Achieved with Parallel Algorithm

For more granular information of the parallel algorithm, knowing how efficiently a thread operates can be a good indication of the overall performance of the algorithm. The *parallel efficiency* is measured as the speed up above but instead of thinking of the algorithm as a whole process, we want to measure each of the p individual threads distribution of the tasks. We define the parallel efficiency E to be:

$$E = \frac{T_1}{T_p * p}.$$

In Figure 5.4 the efficiency of the parallel algorithm is displayed, with the number of threads on the x-axis and the efficiency score on the y-axis. The line tells us how efficiently we are using all the threads available to the algorithm. It is important to be aware that only *strong scaling* were done when measuring the efficiency due to the workings of the algorithm the possibility to perform *weak scaling* were not possible. In strong scaling the goal is to keep the problem sized fixed increasing the number of threads to find an optimally number of threads to use for solving the problem as efficiently as possible. For weak scaling the goal is to keep the problem size fixed for each thread and a larger number of threads are used to solve an increasingly bigger problem.

Prior results are pointing at some potential speed up for the overall cases when the number of threads were increased, but the results are varying depending on the graphs. Looking at the efficiency plot in Figure 5.4 we observe that the marginal efficiency is exponentially decreasing, as expected, when adding the additional threads. in addition, we observe that after the thread count of 8 the marginal efficiency are drawing close to zero.

These findings are supported by the results shown in Figure 5.5. We observe that there are still some marginal efficiency to be gained with a thread count of 16 but, when we get to a thread count of 32 the marginal efficiency are close to zero.

Figure 5.5 shows the same information as Figure 5.4 in a log log plot. From this log log plot it becomes more clearly where the point of diminishing return is located in the plot.

Looking at the log log plot in Figure 5.5 we get a clearer illustration of the degree of the efficiency. From the plot we see that the decrease of efficiency is starting to take effect right from the starter supporting our previous observations.

Some of the limitations on the efficiency and speed up of the parallel algorithm, stems from the construction procedure of the auxiliary graph limiting the work that can be processed concurrently. This could lead to threads spending significant time in an idle state, which is not optimal in regards to a high efficiency.

The parallel algorithms tries to cut down on time the threads spends in an idle state by choosing vertices with high degree to process first, with the goal of splitting the graph into larger cuts for threads to process. The effect of this procedure seems to vary depending on the graph and number of threads.

In the next chapter we will state our conclusion.

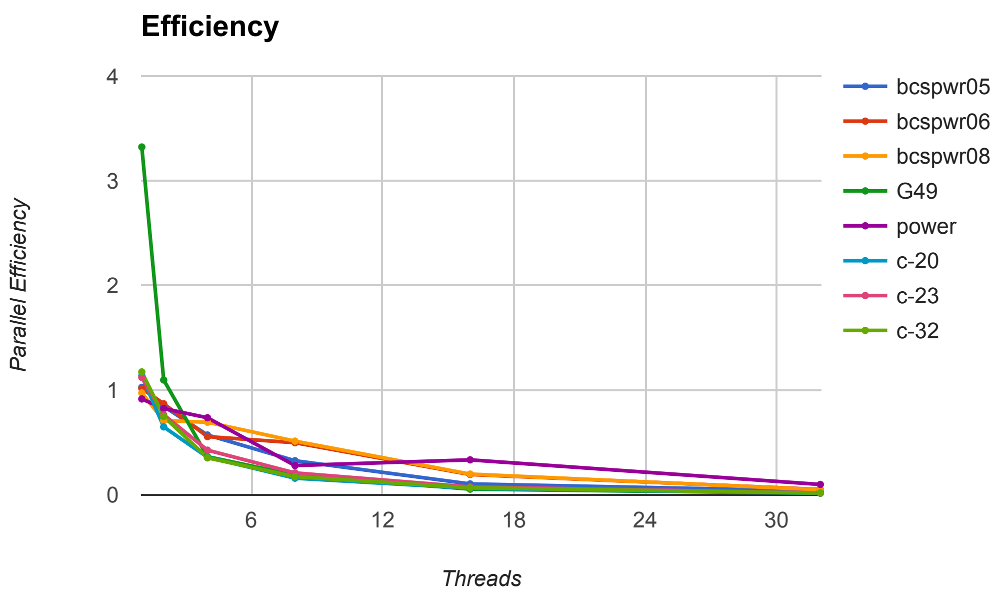


Figure 5.4: Efficiency of the Parallel Algorithm

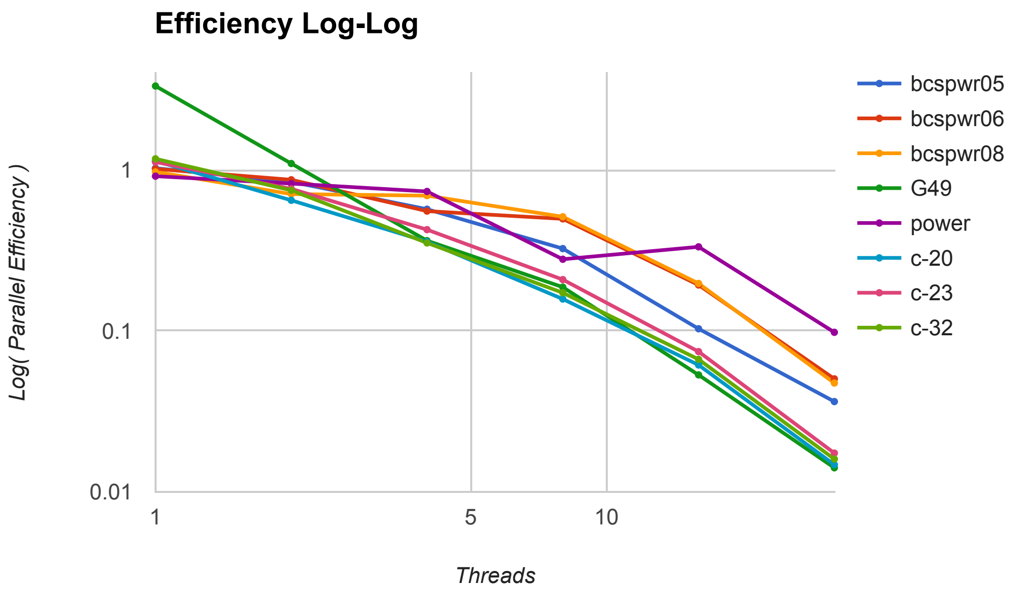


Figure 5.5: Semi-Log Efficiency of the Parallel Algorithm

Chapter 6

Conclusion

The properties of the graphs and the work load distribution due to the structure of the algorithm seem to have mixed effects on the running time. The results are indicating that the properties of the graphs have an direct implication on the marginal efficiency gained by adding additional threads. This is supported by the findings that more threads seems to be creating throttle necks. It is not always important to use more threads but rather ensuring an efficient flow of the process to avoid the creation of to many bottle necks that limits the execution.

With these results in mined, it should be looked into if knowing the property of the graph would be the best way of identifying the optimal thread count for optimizing the running time of a parallel algorithm. To improve on the algorithm some sort of heuristic for deciding which vertices are processed during the algorithm could be implemented in the future. There is also the possibility of, instead of letting the threads process individual cuts for construction of the tree, parallelizing the cut algorithm. Based on the observation that the running time of the algorithm is $O(F|V|)$ where F is the running time of the cut algorithm used. This implies that the algorithm could be implemented with a better cut algorithm to cut down on the running time, but it would not lead to an increase in the efficiency of the parallel algorithm

Bibliography

[1] T. A. Davis and Y. Hu. Acm transactions on mathematical software. *The University of Florida Sparse Matrix Collection*, 38(1):1:1–1:25, 2001.

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

[3] P. Elias, A. Feinstein, and C. Shannon. A note on the maximum flow through a network. *Information Theory, IEEE Transactions on*, 2(4):117–119, Dec 1956.

[4] Zvi Galil and Giuseppe F. Italiano. Reducing edge connectivity to vertex connectivity. *SIGACT News*, 22(1):57–61, March 1991.

[5] R. E. Gomory and T. C. Hu. Multi-terminal network flows. 9(4):551–570, December 1961.

[6] Richard M. Karp, Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

[7] David W. Matula. Determining edge connectivity in 0(nm). In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, SFCS '87, pages 249–251, Washington, DC, USA, 1987. IEEE Computer Society.

[8] Hiroshi Nagamochi and Toshihide Ibaraki. A linear time algorithm for computing 3-edge-connected components in a multigraph. *Japan Journal of Industrial and Applied Mathematics*, 9(2):163, 1992.

[9] WATANABE T NAGAMOCHI H. Computing k-edge-connected components of a multigraph. *IEICE transactions on fundamentals of electronics, communications and computer science*, E76A(4):513–517, 1993.

[10] R.Endre Tarjan. A note on finding the bridges of a graph. *Information Processing Letters*, 2(6):160 – 161, 1974.

[11] Yung H. Tsin. A simple 3-edge-connected component algorithm. *Theory of Computing Systems*, 40(2):125–142, 2007.

[12] Yung H. Tsin. Yet another optimal algorithm for 3-edge-connectivity. *Journal of Discrete Algorithms*, 7(1):130 – 146, 2009. Selected papers from the 1st International Workshop on Similarity Search and Applications (SISAP)1st International Workshop on Similarity Search and Applications (SISAP).

[13] Tianhao Wang, Yong Zhang, Francis Y. L. Chin, Hing-Fung Ting, Yung H. Tsin, and Sheung-Hung Poon. A simple algorithm for finding all k-edge-connected components. *PLOS ONE*, 10(9):1–10, 09 2015.

Appendix A

Sekvensial Algorithm

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <string>
#include <vector>
#include <queue>
#include <stack>
#include <time.h>
#include <omp.h>

#define MAX_LEN 400
#define CHUNK 50

using namespace std;

int main(int argc, char *argv[]){
 int V,E,s,t,i,j,flow,flowAux,temp,nrVerticesOnSourceSide,
    ↪ num_items_read,err=0;/*B r gj res om til Infinity. OK!*/
 double d;
 double start,end;
 int NUM_THREADS = atoi(argv[2]);
 omp_set_dynamic(0);
 /*SET THE BUFFER TO NULL SO WRITTING TO FILE DOES NOT WAIT*/
 setbuf(stdout, NULL);

 /*OPEN FILE*/
 char line[MAX_LEN];
 if(argc<1){
  printf("No␣arguments␣given\n");
  return 0;
 }
 string fileString = "./../Graphs/" + string(argv[1]);
```

```
FILE * pFile;
pFile = fopen(fileString.c_str(),"r");
if (pFile==NULL){
     printf("Could not open %s!\n", argv[1]);
     fclose (pFile);
     return 0;
  }

  /*READ BANNER*/
  do {
        if (fgets(line,200,pFile) == NULL){
             printf("Preamature EOF of %s while reading banner\n", argv
                 ↪ [1]);
             return 0;
        }
   }while (line[0] == '%');

   /*READ NR OF NODES AND EDGES*/
   if (sscanf(line, "%d %d %d", &V, &j, &E) != 3)
   do{
        num_items_read = fscanf(pFile, "%d %d %d", &V, &j, &E);
        if (num_items_read == -1) {
         printf("Preamature EOF of %s while reading nr of Nodes and
             ↪ Edges\n", argv[1]);
         return 0;
        }
   }while (num_items_read != 3);

   /*VECTOR & DATA SETUP*/
   vector<vector<int> > G(V,vector<int>());   //The Graph
   vector<vector<int> > W(V,vector<int>());   //The Weight of an edge,
       ↪ will always be one.
   vector<int> visited(V,0);       //The visited list used by DFS and
       ↪ BFS
   queue<int> Q;           //The Queue
   stack<int> S;           //The Stack

   /*READ EDGES*/
for(i = 0; i<E; i++){
 //if(fscanf(pFile, "%d %d %f", &s, &t, &d)!=3){    //IF edges given
     ↪ with FLOATE weight
 //if(fscanf(pFile, "%d %d %d", &s, &t, &j)!=3){    //IF edges given
     ↪ with weight
 if(fscanf(pFile, "%d %d", &s, &t)!=2){        //IF edges only given by
     ↪ node pair
  printf("Preamature EOF of %s while reading edges\n", argv[1]);
  return 0;
 }
 if(s!=t){
  G[(s % V)].push_back((t % V));
  G[(t % V)].push_back((s % V));
  W[(s % V)].push_back(1);
  W[(t % V)].push_back(1);
 }
}
fclose (pFile);
printf("Read file %s done, nodes=%d & edges=%d\n", argv[1],V,E);

/*CHECK THAT GRAPH IS CONNECTED*/
Q.push(0);
visited[0]++;
while(Q.size()){
 for(i = 0; i < G[Q.front()].size(); i++){
  if(visited[G[Q.front()][i]]<1){
   Q.push(G[Q.front()][i]);
   visited[G[Q.front()][i]]++;
  }
 }
 Q.pop();
}

for(i = 0; i < V; i++){
 if(visited[i]==0){printf("Graph is not connected!\n Program 
     ↪ terminated\n");return 0;}
}

/*SETUP AUXILIARY GRAPH*/
vector<int> firstChild(V);     //Pointer to first child
vector<int> nrOfChilds(V);
vector<int> childID(V);     //ID of child pointed to
vector<int> QStart(V);
vector<int> QEnd(V);
vector<int> nextChild(V);
vector<int> weight(V);
vector<int> setQ(V);
queue<int> AuxQ;
int childPos=0;
int root=0;

for(i=1;i<V;i++){
 if(G[root].size()<G[i].size())
  root = i;
}

nrOfChilds[root]=0;
QStart[root] = 1;       /* Vurderer ikke seg selv som en kandidat til
   ↪ sink node*/
QEnd[root] = V;
```

```
for(i = 0; i < V; i++){
 setQ[i]=i;
 firstChild[i]=-1;
}
setQ[0] = root;        //Swapping the root and 0 so that
setQ[root] = 0;        //Root don t consider its self as target

/*START ALGORITHM*/
start = omp_get_wtime();
AuxQ.push(root);
while(AuxQ.size()){
 s=AuxQ.front();
 if((QEnd[s] - QStart[s]) <= 0){
  AuxQ.pop();
  continue;
 }

 t = setQ[QStart[s]];

 for ( i = 1; i < (QEnd[s]-QStart[s]); i++){
  if(G[t].size()<G[setQ[QStart[s]+i]].size())
   t=setQ[QStart[s]+i];
 }

 AuxQ.push(t);
 /*MAXIMUM FLOW START*/
 flow = 0;
 for(i = 0; i < V; i++){
  for(j = 0; j < W[i].size(); j++){
   W[i][j] = 1;
  }
 }
 while(true){
  /*DISTANS BFS START*/
  visited.assign(V,-1);
  Q.push(s);
  visited[s]=0;
  while(Q.size()){
   temp = Q.front();
   Q.pop();
   for(i = 0; i < G[temp].size(); i++){
    if(visited[G[temp][i]]<0 && W[temp][i]>0){
     Q.push(G[temp][i]);
     visited[G[temp][i]] = visited[temp]+1;
     if(G[temp][i]==t){Q = queue<int>();break;}
    }
   }
  }
  if(visited[t]<0){break;}
  /*DISTANS BFS END*/

  else{ /*AUGMENTH PATH*/
   temp = t;
   POOL:while(temp != s){
     for(i = 0; i < G[temp].size(); i++){
      if(visited[G[temp][i]] == (visited[temp]-1)){
       for(j = 0; j < G[G[temp][i]].size(); j++){
        if(G[G[temp][i]][j]==temp && W[G[temp][i]][j]>0){
         W[temp][i]++;
         W[G[temp][i]][j]--;
         temp = G[temp][i];
         goto POOL;
        }
       }
      }
     }
   }
   flow += 1;
  }
}
/*MAXIMUM FLOW END*/
/*FIND THE CUT START*/
visited.assign(V,-1);
temp = (QEnd[s]-QStart[s]);

for (i = 0; i < temp; ++i)
{
  visited[setQ[QStart[s]+i]] = 0;
}
S.push(s);
visited[s]=2;
visited[t]=2;
nrVerticesOnSourceSide=0;
while(S.size()){
 temp = S.top();
 S.pop();
 for (int i = 0; i < G[temp].size(); ++i)
 {
   if(visited[G[temp][i]]==0 && W[temp][i]>0){
    visited[G[temp][i]]++;
    S.push(G[temp][i]);
    nrVerticesOnSourceSide++;
   }else if(visited[G[temp][i]]==-1 && W[temp][i]>0){
    S.push(G[temp][i]);
    visited[G[temp][i]]--;
   }
 }
}
```

```
  /*FIND THE CUT END*/
  /*PARTIOTION THE VERTICES AND UPDATE AUX GRAPH START*/
  int length = (QEnd[s]-QStart[s]);
  QEnd[t] = QEnd[s];              /*Sink tar over source sine endpoint*/
  QEnd[s] = QStart[s] + nrVerticesOnSourceSide;
  QStart[t] = QEnd[s]+1;

  j=0;
  nrVerticesOnSourceSide=0;
  for ( i = 0; i < length; ++i)
  {
   if(visited[setQ[QStart[s]+i]]==1){
    setQ[QStart[s]+(nrVerticesOnSourceSide++)]=setQ[QStart[s]+i];
   }
   else if((visited[setQ[QStart[s]+i]])==0){
    visited[setQ[QStart[s]+i]] = 2;
    temp = setQ[QStart[t]+(j)];
    setQ[QStart[t]+(j++)]=setQ[QStart[s]+i];
    setQ[QStart[s]+(i--)] = temp;
   }
  }
  setQ[QEnd[s]]=t;

  if(nrOfChilds[s]==0){
   firstChild[s] = childPos;
   nextChild[childPos] = childPos;
  }else{
   temp = firstChild[s];
   for(i = 1; i < nrOfChilds[s]; i++){
    temp = nextChild[temp];
   }
   nextChild[temp] = childPos;
  }
  weight[childPos] = flow;
  childID[childPos++]=t;
  nrOfChilds[s]++;
  /*PARTIOTION THE VERTICES AND UPDATE AUX GRAPH END*/
 }
 end = omp_get_wtime();
 double seconds = end-start;
 if(seconds<360.0)
  printf("\nDone!␣Time␣used␣was␣%f␣seconds\n␣Edges␣in␣tree␣=␣%d\n\n",
      ↪ seconds,childPos);
 else
  printf("\nDone!␣Time␣used␣was␣%f␣minutes\n␣Edges␣in␣tree␣=␣%d\n\n", (
      ↪ seconds/60.0),childPos);
 /*END ALGORITHM*/

 /*REGION TO CHECK IF RESULT IS CORRECT(Maximum flow vs Auxiliary graph
    ↪ )*/
//Check that all Vertices are in Auxiliary graph
visited.assign(V,0);
for ( i = 0; i < V; ++i)
{
 visited[childID[i]]++;
}
visited[root]++;
for ( i = 0; i < V; ++i)
{
 if(visited[i]<1)
  printf("Vertex %i is missing in Auxiliary graph",i);
 if(visited[i]==2)
  printf("Vertex %i something wrong\n",i);
}
//The Real check
printf("Testing for correctness\n");
start = omp_get_wtime();
#pragma omp parallel for schedule(dynamic, CHUNK) num_threads(
    ↪ NUM_THREADS) private(i,j,s,t,flow,flowAux,temp,visited,Q)
    ↪ firstprivate(W)
for(int k = 0; k < V; k++){
 for(int l = k+1; l < V; l++){
  /*MAXIMUM FLOW START*/
  s=k;t=l;
  flow = 0;
  for(i = 0; i < V; i++){
   for(j = 0; j < W[i].size(); j++){
    W[i][j] = 1;
   }
  }
  while(true){

   /*DISTANS BFS START*/
   visited.assign(V,-1);
   Q.push(s);
   visited[s]=0;
   while(Q.size()){
    temp = Q.front();
    Q.pop();
    for(i = 0; i < G[temp].size(); i++){
     if(visited[G[temp][i]]<0 && W[temp][i]>0){
      Q.push(G[temp][i]);
      visited[G[temp][i]] = visited[temp]+1;
      if(G[temp][i]==t){Q = queue<int>();break;}
     }
    }
   }
   if(visited[t]<0){break;}
```

```
    /*DISTANS BFS END*/

    else{
     temp = t;
     LOOP:while(temp != s){
       for(i = 0; i < G[temp].size(); i++){
         if(visited[G[temp][i]] == (visited[temp]-1)){
           for(j = 0; j < G[G[temp][i]].size(); j++){
             if(G[G[temp][i]][j]==temp && W[G[temp][i]][j]>0){
              W[temp][i]++;
              W[G[temp][i]][j]--;
              temp = G[temp][i];
              goto LOOP;
             }
           }
         }

       }
     }
     flow += 1;
    }
  }
  /*MAXIMUM FLOW END*/

  /*FIND FLOW NR FROM AUXILIARY GRAPH*/
  flowAux = E;

  int cur, next,sFound=1,tFound=1;
  visited.assign(V,0); /*bruke distans til    holde styr p   hvor
      ↪ mange barn som er traversert og bes kt.*/
  stack<int> dfsStack;
  stack<int> nextStack;
  stack<int> sStack;
  stack<int> tStack;
  dfsStack.push(root);
  sStack.push(root);
  tStack.push(root);
  if(s == root){sFound=0;}
  if(t == root){tFound=0;}
  while((sFound || tFound) && !dfsStack.empty()){

    cur = dfsStack.top();
    if(nrOfChilds[cur]>visited[cur]){
     if(visited[cur]==0){next = firstChild[cur];}
     else{
       next = nextStack.top();
       nextStack.pop();
     }
     dfsStack.push(childID[next]);
```

```
      if(sFound){
       sStack.push(next);
       if(childID[next]==s){sFound=0;}
      }
      if(tFound){
       tStack.push(next);
       if(childID[next]==t){tFound=0;}
      }
      nextStack.push(nextChild[next]);
      visited[cur]++;
     }else{
      dfsStack.pop();
      if(dfsStack.size()<nextStack.size()){nextStack.pop();}
      if(sFound){sStack.pop();}
      if(tFound){tStack.pop();}
     }
    }
    sFound=childID[sStack.top()];
    tFound=childID[tStack.top()];
    while(!sStack.empty() && !tStack.empty()){
     if(sStack.size()<tStack.size()){
      if(flowAux>weight[tStack.top()]){flowAux = weight[tStack.top()];}
      tStack.pop();
     }
     else if(tStack.size()<sStack.size()){
      if(flowAux>weight[sStack.top()]){flowAux = weight[sStack.top()];}
      sStack.pop();
     }
     else if(sStack.size()==tStack.size()){
      if(sStack.top()==tStack.top()){
       break;
      }
      else{
       if(flowAux>weight[sStack.top()]){flowAux = weight[sStack.top()];}
       sStack.pop();
       if(flowAux>weight[tStack.top()]){flowAux = weight[tStack.top()];}
       tStack.pop();
      }
     }
    }
    if(flowAux == E)
     printf("\nCould not find flow in Auxiliary graph. Should not happen
        ↪ ever, ever!\n\n");
    else if(flowAux!=flow)
     printf("%i ERROR - s=%i|t=%i flow from Auxiliary %i,%i graph=%i 
        ↪ flow from MAX-FLOW=%i\n",err++,s,t,sFound,tFound,flowAux,
        ↪ flow);
  }
```

```
 }
 end = omp_get_wtime();
 float testingSeconds = end-start;
 if(testingSeconds <360.0)
  printf("\nDone! Testing finished in %f seconds\n", testingSeconds);
 else
  printf("\nDone! Testing finished in %f minutes\n", (testingSeconds
     ↪ /60.0));
 if(err==0)
  printf("<------------------------------->\n<->  Success! <->\n
     ↪ <------------------------------->\n");
 else
  printf("\n:  (\n!-!-! Nr of errors are %i !-!-!\n", err);

 return 0;
}
```

Appendix B

Parallel Algorithm

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <string>
#include <vector>
#include <queue>
#include <stack>
#include <time.h>
#include <omp.h>

#define MAX_LEN 400
#define CHUNK 50

using namespace std;

int main(int argc, char *argv[]){
 int V,E,s=-1,t,i,j,flow,flowAux,temp,nrVerticesOnSourceSide,
    ↪ num_items_read,err=0;
 double d;
 double start,end;
 omp_set_dynamic(0);
 /*SET THE BUFFER TO NULL SO WRITTING TO FILE DOES NOT WAIT*/
 setbuf(stdout, NULL);

 /*OPEN FILE*/
 char line[MAX_LEN];
 if(argc<1){
  printf("No␣arguments␣given\n");
  return 0;
 }
 string fileString = "./../Graphs/" + string(argv[1]) + ".mtx";
 FILE * pFile;
 pFile = fopen(fileString.c_str(),"r");
```

```
if (pFile==NULL){
     printf("Could not open %s!\n", argv[1]);
     fclose (pFile);
     return 0;
  }

  /*READ BANNER*/
  do {
        if (fgets(line,200,pFile) == NULL){
             printf("Preamature EOF of %s while reading banner\n", argv
                 ↪ [1]);
             return 0;
        }
   }while (line[0] == '%');

   /*READ NR OF NODES AND EDGES*/
   if (sscanf(line, "%d %d %d", &V, &j, &E) != 3)
   do{
        num_items_read = fscanf(pFile, "%d %d %d", &V, &j, &E);
        if (num_items_read == -1) {
         printf("Preamature EOF of %s while reading nr of Nodes and
             ↪ Edges\n", argv[1]);
         return 0;
        }
   }while (num_items_read != 3);

   /*VECTOR & DATA SETUP*/
   vector<vector<int> > G(V,vector<int>());
   vector<vector<int> > W(V,vector<int>());
   vector<int> visited(V,0);
   queue<int> Q;
   stack<int> S;

   /*READ EDGES*/
for(i = 0; i<E; i++){
 if(fscanf(pFile, "%d %d %f", &s, &t, &d)!=3){    //IF edges given with
     ↪ FLOATE weight
 //if(fscanf(pFile, "%d %d %d", &s, &t, &j)!=3){    //IF edges given
     ↪ with weight
 //if(fscanf(pFile, "%d %d", &s, &t)!=2){       //IF edges only given
     ↪ by node pair
  printf("Preamature EOF of %s while reading edges\n", argv[1]);
  return 0;
 }
 if(s!=t){
  G[(s % V)].push_back((t % V));
  G[(t % V)].push_back((s % V));
  W[(s % V)].push_back(1);
  W[(t % V)].push_back(1);
 }
}
fclose (pFile);
printf("Read␣file␣%s␣done,␣nodes=%d␣&␣edges=%d\n\n", argv[1],V,E);

/*CHECK THAT GRAPH IS CONNECTED*/
Q.push(0);
visited[0]++;
while(Q.size()){
 for(i = 0; i < G[Q.front()].size(); i++){
  if(visited[G[Q.front()][i]]<1){
   Q.push(G[Q.front()][i]);
   visited[G[Q.front()][i]]++;
  }
 }
 Q.pop();
}

for(i = 0; i < V; i++){
 if(visited[i]==0){printf("Graph␣is␣not␣connected!\n␣Program␣
     ↪ terminated\n");return 0;}
}

/*SETUP AUXILIARY GRAPH*/
vector<int> firstChild(V);      //Pointer to first child
vector<int> nrOfChilds(V);
vector<int> childID(V);     //ID of child pointed to
vector<int> QStart(V);
vector<int> QEnd(V);
vector<int> nextChild(V);
vector<int> weight(V);
vector<int> setQ(V);
int childPos=0;

int root=0;

//Chose the biggest node as start node.
for(i=1;i<V;i++){
 if(G[root].size()<G[i].size())
  root = i;
}

nrOfChilds[root]=0;
QStart[root] = 1;       /* Vurderer ikke seg selv som en kandidat til
    ↪ sink node*/
QEnd[root] = V;
for(i = 0; i < V; i++){
 setQ[i]=i;
 firstChild[i]=-1;
```

```
}
setQ[0] = root;        //Swapping the root and 0 so that
setQ[root] = 0;        //Root don t consider its self as target

/*START ALGORITHM*/
printf("Number␣of␣Threads␣requested␣was␣%i\n",atoi(argv[2]));
start = omp_get_wtime();
#pragma omp parallel num_threads(atoi(argv[2])) private(i,j,t,s,flow,
    ↪ flowAux,temp,visited,Q,S,nrVerticesOnSourceSide) firstprivate(W
    ↪ )
{
 s=-1;
    int workingThreads = 1;
 int childPosPid;
 i=0;
 if(omp_get_thread_num()==0){
  s=root;
 }
 while(workingThreads){

  if(s<0){
   #pragma omp critical(AddRemoveWork)
   {
    if(!AuxQ.empty()){
     s = AuxQ.front();
     AuxQ.pop();
    }
   }
  }

  if(s<0 || (QEnd[s] - QStart[s]) <= 0){
   if(childPos==(V-1))
         workingThreads--;
   s=-1;
   continue;
  }

  t = setQ[QStart[s]];
  for(i=1;i<(QEnd[s]-QStart[s]);i++){
   if(G[t].size() < G[setQ[QStart[s]+i]].size())
    t=setQ[QStart[s]+i];
  }

  /*MAXIMUM FLOW START*/
  flow = 0;
  for(i = 0; i < V; i++){
   for(j = 0; j < W[i].size(); j++){
    W[i][j] = 1;
   }
}
while(true){
  /*DISTANS BFS START*/
  visited.assign(V,-1);
 Q.push(s);
  visited[s]=0;
  while(Q.size()){
    temp = Q.front();
   Q.pop();
    for(i = 0; i < G[temp].size(); i++){
      if(visited[G[temp][i]]<0 && W[temp][i]>0){
       Q.push(G[temp][i]);
        visited[G[temp][i]] = visited[temp]+1;
        if(G[temp][i]==t){Q = queue<int>();break;}
      }
    }
  }
  if(visited[t]<0){break;}      //there are no more paths to target
  /*DISTANS BFS END*/

  else{ /*AUGMENTH PATH*/
    temp = t;
   POOL:while(temp != s){
      for(i = 0; i < G[temp].size(); i++){
        if(visited[G[temp][i]] == (visited[temp]-1)){
          for(j = 0; j < G[G[temp][i]].size(); j++){
            if(G[G[temp][i]][j]==temp && W[G[temp][i]][j]>0){
             W[temp][i]++;
             W[G[temp][i]][j]--;
              temp = G[temp][i];
              goto POOL;
            }
          }
        }
      }
    }
    flow += 1;
  }
}
/*MAXIMUM FLOW END*/
/*FIND THE CUT START*/
visited.assign(V,-1);
temp = (QEnd[s]-QStart[s]);

for (i = 0; i < temp; ++i)
{
  visited[setQ[QStart[s]+i]] = 0; //Only want vertices that are in
      ↪ the set to be in the cut
}
```

```
S.push(s);
visited[s]=2;
visited[t]=2;
nrVerticesOnSourceSide=0;
while(S.size()){       //DFS to find cut source-side
 temp = S.top();
 S.pop();
 for (int i = 0; i < G[temp].size(); ++i)
 {
   if(visited[G[temp][i]]==0 && W[temp][i]>0){
     visited[G[temp][i]]++;
     S.push(G[temp][i]);
     nrVerticesOnSourceSide++;
   }else if(visited[G[temp][i]]==-1 && W[temp][i]>0){
     S.push(G[temp][i]);
     visited[G[temp][i]]--;
   }
 }
}
/*FIND THE CUT END*/
/*PARTIOTION THE VERTICES AND UPDATE AUX GRAPH START*/
#pragma omp critical(UpdateAux)
{
 childPosPid = childPos++;
}
int length = (QEnd[s]-QStart[s]);
QEnd[t] = QEnd[s];              /*Sink tar over source sine endpoint*/
QEnd[s] = QStart[s] + nrVerticesOnSourceSide;
QStart[t] = QEnd[s]+1;

j=0;
nrVerticesOnSourceSide=0;
for ( i = 0; i < length; ++i)  //Updating the set after the cut
{
  if(visited[setQ[QStart[s]+i]]==1){
   setQ[QStart[s]+(nrVerticesOnSourceSide++)]=setQ[QStart[s]+i];
  }
  else if((visited[setQ[QStart[s]+i]])==0){
   visited[setQ[QStart[s]+i]] = 2;
   temp = setQ[QStart[t]+(j)];
   setQ[QStart[t]+(j++)]=setQ[QStart[s]+i];
   setQ[QStart[s]+(i--)] = temp;
  }
}
setQ[QEnd[s]]=t;

if(nrOfChilds[s]==0){     //if source have no prev childs this is
    ↪ first child
 firstChild[s] = childPosPid;
```

```
    nextChild[childPosPid] = childPosPid;
   }else{
    temp = firstChild[s];
    for(i = 1; i < nrOfChilds[s]; i++){ //if source have prev child,
       ↪ must find pointer to last child
     temp = nextChild[temp];
    }
    nextChild[temp] = childPosPid;
   }
   weight[childPosPid] = flow;
   childID[childPosPid++]=t;
   nrOfChilds[s]++;

   /*PARTIOTION THE VERTICES AND UPDATE AUX GRAPH END*/

   #pragma omp critical(AddRemoveWork)
   {
    if(workingThreads>=0){
     AuxQ.push(t);
     if(!(workingThreads<0))
      workingThreads++;
    }
   }
  }
 }
end = omp_get_wtime();
double seconds = end-start;
if(seconds<360.0)
 printf("\nDone! Time used was %f seconds\n Edges in tree = %d\n\n",
     ↪ seconds,childPos);
else
 printf("\nDone! Time used was %f minutes\n Edges in tree = %d\n\n", (
     ↪ seconds/60.0),childPos);
/*END ALGORITHM*/
//
    ↪ -------------------------------------------------------------------------
    ↪
/*REGION TO CHECK IF RESULT IS CORRECT(Maximum flow vs Auxiliary graph
    ↪ )*/
//Check that all Vertices are in Auxiliary graph
visited.assign(V,0);
for ( i = 0; i < V; ++i)
{
 visited[childID[i]]++;
}
visited[root]++;
for ( i = 0; i < V; ++i)
{
 if(visited[i]<1)
```

```
    printf("Vertex %i is missing in Auxiliary graph",i);
}
//The Real check
printf("Testing for correctness\n");
start = omp_get_wtime();
#pragma omp parallel for schedule(dynamic, CHUNK) num_threads(atoi(
    ↪ argv[3])) private(i,j,s,t,flow,flowAux,temp,visited,Q)
    ↪ firstprivate(W)
for(int k = 0; k < V; k++){
 for(int l = k+1; l < V; l++){
  /*MAXIMUM FLOW START*/
  s=k;t=l;
  flow = 0;
  for(i = 0; i < V; i++){
   for(j = 0; j < W[i].size(); j++){
    W[i][j] = 1;
   }
  }
  while(true){

   /*DISTANS BFS START*/
   visited.assign(V,-1);
   Q.push(s);
   visited[s]=0;
   while(Q.size()){
    temp = Q.front();
    Q.pop();
    for(i = 0; i < G[temp].size(); i++){
     if(visited[G[temp][i]]<0 && W[temp][i]>0){
      Q.push(G[temp][i]);
      visited[G[temp][i]] = visited[temp]+1;
      if(G[temp][i]==t){Q = queue<int>();break;}
     }
    }
   }
   if(visited[t]<0){break;}
   /*DISTANS BFS END*/

   else{
    temp = t;
    LOOP:while(temp != s){
     for(i = 0; i < G[temp].size(); i++){
      if(visited[G[temp][i]] == (visited[temp]-1)){
       for(j = 0; j < G[G[temp][i]].size(); j++){
        if(G[G[temp][i]][j]==temp && W[G[temp][i]][j]>0){
         W[temp][i]++;
         W[G[temp][i]][j]--;
         temp = G[temp][i];
         goto LOOP;
        }
       }
      }

     }
    }
    flow += 1;
   }
  }
  /*MAXIMUM FLOW END*/

  /*FIND FLOW NR FROM AUXILIARY GRAPH*/
  flowAux = E;

  int cur, next,sFound=1,tFound=1;
  visited.assign(V,0);
  stack<int> dfsStack;
  stack<int> nextStack;
  stack<int> sStack;
  stack<int> tStack;
  dfsStack.push(root);
  sStack.push(root);
  tStack.push(root);
  if(s == root){sFound=0;}
  if(t == root){tFound=0;}
  while((sFound || tFound) && !dfsStack.empty()){

   cur = dfsStack.top();
   if(nrOfChilds[cur]>visited[cur]){
    if(visited[cur]==0){next = firstChild[cur];}
    else{
     next = nextStack.top();
     nextStack.pop();
    }
    dfsStack.push(childID[next]);
    if(sFound){
     sStack.push(next);
     if(childID[next]==s){sFound=0;}
    }
    if(tFound){
     tStack.push(next);
     if(childID[next]==t){tFound=0;}
    }
    nextStack.push(nextChild[next]);
    visited[cur]++;
   }else{
    dfsStack.pop();
    if(dfsStack.size()<nextStack.size()){nextStack.pop();}
    if(sFound){sStack.pop();}
```

```
      if(tFound){tStack.pop();}
    }
   }
   sFound=childID[sStack.top()];
   tFound=childID[tStack.top()];
   while(!sStack.empty() && !tStack.empty()){
    if(sStack.size()<tStack.size()){
     if(flowAux>weight[tStack.top()]){flowAux = weight[tStack.top()];}
     tStack.pop();
    }
    else if(tStack.size()<sStack.size()){
     if(flowAux>weight[sStack.top()]){flowAux = weight[sStack.top()];}
     sStack.pop();
    }
    else if(sStack.size()==tStack.size()){
     if(sStack.top()==tStack.top()){
      break;
     }
     else{
      if(flowAux>weight[sStack.top()]){flowAux = weight[sStack.top()];}
      sStack.pop();
      if(flowAux>weight[tStack.top()]){flowAux = weight[tStack.top()];}
      tStack.pop();
     }
    }
   }
   if(flowAux == E)
    printf("\nCould␣not␣find␣flow␣in␣Auxiliary␣graph.␣Should␣not␣happen
        ↪ ␣ever,␣ever!\n\n");
   else if(flowAux!=flow)
    printf("%i␣ERROR␣-␣s=%i|t=%i␣flow␣from␣Auxiliary␣%i,%i␣graph=%i␣
        ↪ flow␣from␣MAX-FLOW=%i\n",err++,s,t,sFound,tFound,flowAux,
        ↪ flow);
  }
 }
 end = omp_get_wtime();
 double testingSeconds = end-start;
 if(testingSeconds<360.0)
  printf("\nDone!␣Testing␣finished␣in␣%f␣seconds\n", testingSeconds);
 else
  printf("\nDone!␣Testing␣finished␣in␣%f␣minutes\n", (testingSeconds
      ↪ /60.0));
 if(err==0)
  printf("<------------------------------->\n<->␣Success!␣<->\n
      ↪ <------------------------------->\n");
 else
  printf("\n:  (\n!-!-!␣Nr␣of␣errors␣are␣%i␣!-!-!\n", err);

 return 0;
```

```
}
```