UNIVERSIDADE D
COIMBRA

Marco Henrik Frieden

MINIMUM SPANNING TREE ALGORITHMS, ONE-DIMENSIONAL RANKING AND COMPARISON

Dissertação no âmbito do Mestrado em Matemática, Ramo Tecnomatemática orientada pelo Professor Doutor José Luis Esteves dos Santos e apresentada ao Departamento de Matemática da Faculdade de Ciências e Tecnologia.

Setembro 2021

Minimum Spanning Tree Algorithms, one-dimensional ranking and comparison.

Marco Frieden

UNIVERSIDADE D
COIMBRA

Master in Mathematics

Mestrado em Matemática

MSc Dissertation | Dissertação de Mestrado

September 2021

Acknowledgements

This work is the culmination of every tear, drop of sweat, effort and hard work I have done trough 6 years of university. These were years of self-discovery, growth, lots of fun and even more learning, not only mathematics and programming but on how to successfully multi-task. They were everything I hoped for when I moved back to Coimbra. It might never have happened and been successfully finished if it were not for the people I honourably mention here. To my parents, the always supporting people, who never doubted for a minute on my ability to conclude such a hard degree as a master of mathematics. Through all these years, they saw me change my mind about what to study and what to become, more than once. Even so, always supporting, teaching me how to become a respectful person, they always had my back, in whatever situation I saw myself in. I am so very grateful to be your son. To my grandparents, the so very patient couple that even not knowing regularly what I was doing and how my bachelor and master were going, always believed I would conclude this successfully. The stability you provided during all these years made my life so much easier and with less worries. Thank you for spoiling me, you are a true blessing. To Inês, the always available person, there to hear me out and making sure I never felt alone. Your friendship is something I truly cherish and will never let go of. To Costa, the creative mind that always had a solution when I was desperately looking for answers. Thank you for your friendship and for being the art in my work. To João, the person that gave me back my ability to believe in myself and knowing my worth. You truly are the force of so many, including me. Thank you for always seeing the best in all. To Ana Rita, the always available person to listen to my desperations during the 10 months of writing this work. I hope this life continues to put us in similar paths, so we can always support each other. To Picado, the "pro-bono professor" of half of the mathematics department students. Thanks you for always having an idea, believing in my mathematical abilities and, specially, being the fantastic friend you are. To António, the one that forced me out of my office on a regular basis to take a dive in the ocean. Thank you for always being available to hear me and making me a confident person. To Quen Gui, the person that knew this would be my path before anyone else. It has been more than 10 years since we cannot speak anymore but I would have loved to call you and tell you I became a master of mathematics. It is not the MIT, but UC is something to be proud as well. Thank you for teaching me proper math. To my guiding professor José Luís, thank you so much for all the orientation and guidance through these last 10 months. I know I was not an easy student but you knew how to help and guide me. To Refego, the everlasting group of friends. Thank you to each and everyone of you, for believing in me and distract me every time I was too involved in my work. To Rui, the mentor of my professional career start. I knew it would be complicated to write this thesis and have a full time job at the same time, but I never imagined how lucky I could be for having a superior like you. Thank you for all the understanding, giving me chances to focus myself on this work and believing in my abilities. To all of you: **F-R-A!**

Abstract

In this work we focus on the Minimum Spanning Tree (MST) problem and efficient implementations for it, as well as the study of a variant of the classical problem - ranking of MST's. In particular, we cover implementations of the attainment of such trees, with simple lists and more complex data structures such as binary and Fibonacci heaps. In combination with the most known Kruskal and Prim algorithms, we compare all the implementations to verify that the heaps do not only show themselves as theoretically more efficient, but also in practical cases, in particular for complete graphs. We also look and implement an algorithm for the attainment of the k best spanning trees, using the implementations studied before. We leave open some possible continuation of this work, namely applying it to multi-objective graphs and turning the knowledge into a corporative or industrial application.

Resumo

Neste trabalho focamo-nos no problema de árvores de cobertura mínima (MST) e em implementações eficientes de o resolver, tal como no estudo da variante do problema clássico - ranking de MST's. Em particular, estudamos diferentes implementações de como obter estas árvores, usando estruturas de dados simples, como listas, e mais complexas, como *heaps* binárias e *heaps* de Fibonacci. Usando os largamente conhecidos algoritmos de Kruskal e Prim, comparamos todas as diferentes implementações, provando a grande eficiência destas estruras *heap*, não apenas no sentido teórico mas comprovando com testes práticos, em particular para grafos completos. Também estudamos e desenvolvemos um algoritmo para a obtenção das k melhores árvores de cobertura. Deixamos em aberto a continuação deste trabalho, com a sujestão de aplicar estas implementações em grafos multi-objetivo e usar o investigado para criar uma aplicação prática para o mundo corporativo ou industrial.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

In this work we intend to study different algorithms to solve the Minimum Spanning Tree problem, applying them later to the retrieval of all possible spanning trees, and use all this knowledge in a real life application.
Firstly, let us start with some basic definitions, needed to understand the task at hand.

1.1 Basic Definitions

Definition 1 (Graph) *We may define a graph as $G = (V, E)$, a set of objects in which pairs are, in some way, related. V represents the objects, henceforth called vertices, and E the pairs that are related to each other, the edges.*

Throughout this work we will define the number of vertices of a graph as $|V| = n$, and the number of edges as $|E| = m$.

We may define a graph as *directed* or *undirected*, which tells if all the relations are oriented, or not oriented. In an undirected graph we know that the relations are then symmetric. We will be working with these. We may also refer that the edges of a graph may have a weight associated to them, then the graph is called a *weighted graph*.

Definition 2 (Path) *A path is a sequence of edges which joins a sequence of vertices of a graph, i.e. two consecutive edges have a vertex in common, both of the edges connect themselves to the same vertex. We call a sole vertex an empty path.*

Definition 3 (Cycle) *A path that ends and starts at the same vertex is called a cycle.*

Definition 4 (Connected Graph) *If all vertices of a graph are connected to others, i.e. we may define a path between any two vertices, we call the graph connected.*

Definition 5 (Complete Graph) *If each vertex of a graph is connected to all vertices, we call this graph complete.*

Proposition 1 (Complete Graph) *A complete undirected graph has exactly $\frac{n\cdot(n-1)}{2}$ edges.*

This is easy to follow as we have to connect all n vertices to the other $n-1$ and eliminate the repetitions as we count the edges. As we connect vertex i to all others, we also connect it to j. Then, while connecting j to all others, we repeat the connection to i. As this happens for every 2 vertices of the graph, we divide $n \cdot (n-1)$ by 2.

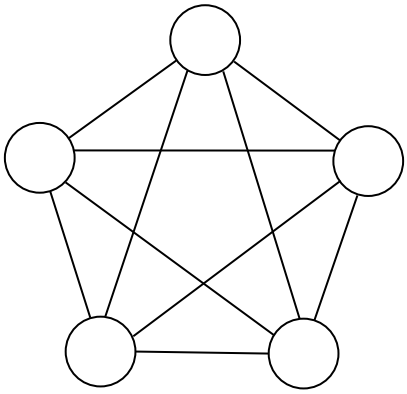


Fig. 1.1 A complete undirected graph with 5 vertices and 10 edges.

Definition 6 (Tree) *A tree is a connected undirected graph in which any two given vertices are connected by only one path. This means, there is only one possibility to choose a subset of edges s.t. we may create a path between two vertices.*

Definition 7 (Spanning Tree) *Given a graph $G = (V,E)$, we may define a Spanning Tree T as a subset of G that is a tree and includes all vertices of V.*

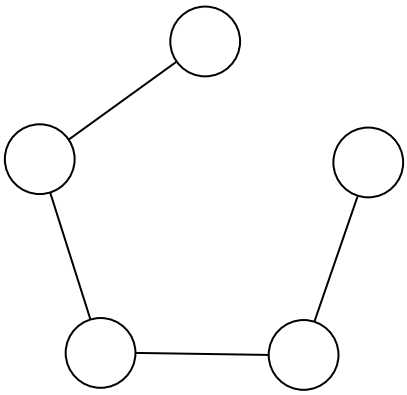


Fig. 1.2 A spanning tree of the complete graph from Figure 1.1.

After introducing the basic concepts about graphs, we will define the main problem addressed in the work which is relevant topic nowadays.

Problem 1 *Let $G = (V,E)$ be a connected weighted undirected graph with V vertices and E edges. For each edge (a,b), connecting vertices a and b, there is a weight $w(a,b)$ associated. We wish to find a Spanning Tree $T = (V,E'), E' \in E$, of G, such that $\sum_{(a,b)\in E'} w(a,b)$ is minimal.*

1.2 Historical Background

Although mentioned repeatedly in the beginning of the last century it was firstly solved by Boruvka in 1926 [3]. In this article a very intuitive and simplified formulation of what we are trying to achieve is given [25]:

There are n points given in the plane (in the space) whose mutual distances are all different. We wish to join them by a net such that

1. *Any two points are joined either directly or by means of some points,*
2. *The total length of the net would be the shortest possible.*

The proposed algorithm already had a time complexity of $O(m\log(n))$. Similiar algorithms were written and rediscovered by Choquet in 1938 [10] and Sollin in 1951 [28]. (In Chapter 2 we will explain what this O function represents.)
One of the most iconic algorithms is the very much known Jarnik-Prim-Dijkstra (JPD) algorithm (commonly referred to as Prim's algorithm), named by the three scientists that developed it in very distanced years . Beginning in 1930, Jarnik invented this algorithm [19], with Prim [27] and Dijkstra [12] rediscovering and republishing it in 1957 and 1959, respectively. It works by choosing a vertex at random, following the lightest edge connecting that one to another vertex, adding this edge and end-vertex to the tree we are constructing, and repeating this process with the vertices we already have in the new tree reached, until all vertices are in the tree.
Another very known algorithm is Kruskal's, presented in 1955 [20]. Here the process differs significantly from the before mentioned Prim algorithm. The idea with this algorithm is to firstly sort the edges by their weights. Each single vertex is, from starters, considered a tree. Iteratively the algorithm joins the trees, by using the smallest edges that connect them until they are combined into one tree, excluding the edges that may create a cycle.
Both these algorithms have a time complexity of $O(m\log(n))$.
After the development of Fibonacci Heaps in priority queuing in 1984 by Fredman and Tarjan [14], the time complexity was reduced to $O(m\beta(m,n))$ with $\beta(m,n) = min\{i : \log^i(n) \leq \frac{m}{n}\}$. A year later Gabow et al. were able to reduce the time complexity even further to $O(m\log(\beta(m,n)))$, using the idea of *Packets.* [15, 16].
Only with the turn of the millennium in 2000, Chazelle was able to discover a new heap structure, the soft heap, implemented it in the priority queuing and the time complexity was reduced further to $O(m\alpha(m,n))$ [8], with α the inverse of the Ackerman function.
Opposed to the before mentioned sequential algorithms, in the last 40 years there have also been presented some distributed and parallel MST algorithms. The difference between these two types of algorithms lies in how the computers communicate between each other: in distributed algorithms many computers, each with processors and memory, may communicate between each other, while in parallel algorithms we only have one memory and many processors, that communicate trough this shared memory. The second implies the use of only one computer, no matter the amount of processors, while the first enables the use of many computers, in different locations.
As foreseeable, the computational time of the these types of algorithms are indeed better than the sequential. Gallager et al. [17] presented the first distributed MST algorithm in 1983, which had the

computational time of $O(n\log(n))$, but it is supposed that there is a processor at each vertex. Since the 1980's, many algorithms have been presented and built on each together, leading to very good results of the latest algorithm by Elkin, in 2020 [13]. With the parallel algorithms we do not have similar results. The computational time is still quite low compared to the sequential ones but there have not been such good results as from the distributed ones. Even so, Chong et al. [9] were able to give us a time complexity of $O(\log(n))$ in 2001 using a linear number of processors.
Opposed to the before mentioned exact algorithms, there have also been proposed a lot of approximate algorithms. These tend to improve the time complexity as the first step is to reduce the size of the graph, by creating a sparse graph from the complete one, as in [4], or partitioning the graph into clusters using Hilbert curve [21] or K-means [36]. The second step of these algorithms is usually applying an exact MST algorithm to the smaller dataset.
In this work we will mainly focus on the implementations of the very known Prim and Kruskal algorithms [20, 27], with different types of data structures. Here we will rely on the works of Fredman and Tarjan, with Fibonacci Heaps [14] and Binary Heaps [32]. We will compare the different running times of these algorithms and these implementations.

Chapter 2

Heaps

As the relevance of this thesis is the applications of the referenced MST algorithms, and these applications imply big amounts of data, and therefore a great size of the graphs, it is important to dedicate an entire chapter to Heaps, the data structures that truly make a difference when it comes to comparing the different algorithms.
First let us add some more definitions that will be useful to the understanding of this chapter.

Definition 8 (Ordered Tree) *An ordered tree is an oriented tree in which the vertices are in a specific order. It is a rooted tree in which every vertex after the so called root is a descendent, called children. A child may have itself children, and the relation between a parent its child must be the same as the relation between the root and its children.*

Definition 9 (Binary Tree) *A Binary Tree is an ordered tree in which each node has at most two children, the right and left child. In this type of tree we need the idea of level, which simply means the distance from its root, the most distant parent.*

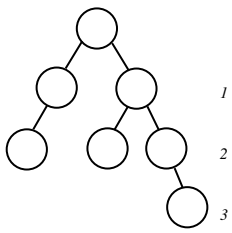


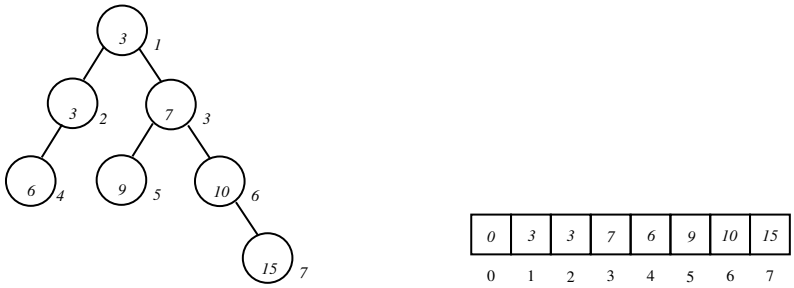
Fig. 2.1 Binary tree with 3 levels.

Theorem 1 (Cayley's Formula [5]) *The maximum amount of distinct spanning trees in a graph is n^{n-2}. This is the exact number of distinct spanning trees if the graph is complete.*

The proof of Theorem 1 is omitted as it is too extensive and not relevant for the subject of this work.

A *Heap* is a data-structure that is constructed as a tree and was firstly introduced by JWJ Williams in 1964 [33]. These structures respect the following property:

Definition 10 (Heap Property) *A heap is a Maxheap (Minheap), which means that a child of a node, is always less or equal (bigger or equal, respectively) than it's parent. Just as in an ordered tree, the most distant parent of all nodes, which does not have a parent, is called the root node.*



(a) Min Heap in graph representation (b) Min Heap in vector representation

Without loss of generalisation, we will continue the rest of this work only using and referencing to minheaps, to simplify understanding. In the figures above we are able to notice that the first entry in the vector representation is empty, followed by the root element and then, ordered bigger or equal elements. On the right hand side of each vertex in the graph representation, we can see the index it corresponds to in the vector representation.
Every kind of heap has some operations that are the same. In the following points are they described and later particularised for each kind we will introduce:

- Make-heap: Creates the structure, without adding any element;
- Insert: Adds an element to the heap;
- Find-min: Returns the minimum element of the heap;
- Delete-min: Returns the minimum element of the heap and removes it from the heap;
- Decrease-key: Finds a specific element in the heap, changes it's value and reorganises it into its rightful position;
- Union: Merges two heaps into one;
- Delete: Eliminates all elements of a heap and the structure itself.

2.1 Binary Heaps

The *Binary Heap* is essentially defined as a Binary tree with two additional conditions:

- Heap Property - as mentioned before;
- Shape Property.

Definition 11 (Shape Property) *The tree needs to be a complete binary tree, i.e. all levels of the tree are fully filled, except for the bottom level, the children with biggest distance to the root.*

These structures have some operations associated to them as they are needed to use them with data.

- Insert: Add the element to the bottom level of the heap, compare the element with its parent, and if needed, swap them and compare the item again with its new parent, until it is smaller than the parent.
- Delete-min: Eliminate the root from the heap is done by changing the smallest element with the biggest and then compare the new root element with its children, while they are not in order, the parent should be swapped with its child.
- Decrease-key: To change the value of a certain key, we firstly need to locate it in the heap. After we have found the index, we change it's value and, as the new value is now smaller than before, we check its value with its parent and swap them, if necessary. This swap-operation must continue until there is no longer a need to swap.
- Union: To merge two different heaps we need to extract each element of one of the heaps and insert them, one after another, into the other heap.

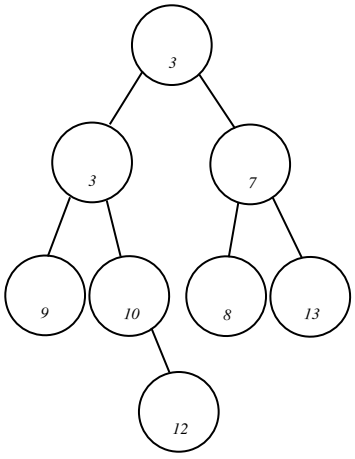


Fig. 2.3 Binary heap example.

2.2 Binomial Heaps

Definition 12 (Degree) *The amount of levels a certain tree has is called degree. If the tree is a single node, it has degree* 0*. Any more levels that are below the root element, are incremented in the degree.* [1]

Binomial heaps are a collection of binomial trees, so let us first understand what a binomial tree is.

A binomial tree of degree 0, B_0, is a single node. Binomial heaps are formed recursively: to create binomial tree of degree k we need to *link* two binomial heaps of degree $k-1$.

[1] Example: The tree in Figure 2.3 has degree 3.

Definition 13 (Linking) *In this type of structure we have a fundamental operation called linking, which combines two item-disjoint trees into one, by comparing the values of their roots and turning the bigger one into a child of the smaller. Linking works in a similar way as the mentioned Union, but for two trees instead of two heaps.*

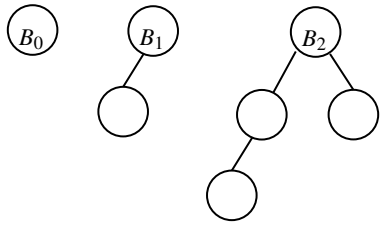


Fig. 2.4 Binomial trees of degree 0, 1 and 2.

The Binomial tree with degree k B_k, has 2^k elements, as it is the product of two B_{k-1} trees: $2^{k-1} + 2^{k-1} = 2^k$.
Binomial heaps are a collection of binomial trees where each tree follows the minheap property, and there is at most one binomial tree of any degree. The following figure represents an example of a binomial heap.

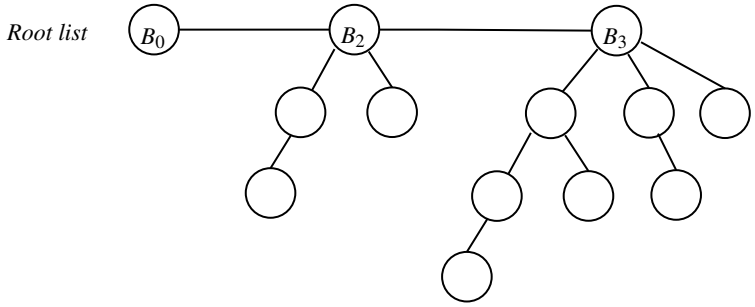


Fig. 2.5 Binomial heap with binomial trees of degrees 0, 2 and 3.

We will not be going into detail with this type of structure as its implementation does not lead to significant improvements of the algorithms, as we were able to conclude from the presented table 2.1. This is to be explained by the process of *linking*, which is done after every operation. This very rigid structure does complicate the computation as the creation of each tree is done by the *linking* of two lower degree trees, which slows down the process. What is important although is the concept, in particular the presence of many trees in one structure and the presence of only one of any degree. This concept is also used in the next type of structure in a much less rigid way, which improves it's running times.

2.3 Fibonacci Heaps

Firstly mentioned and created by Fredman and Tarjan [14], the Fibonacci heaps are a collection of trees that work with the minheap property, so the root object is always the smallest. In Fibonacci heaps, *linking* is once again used. This operation is vital in a Fibonacci heap as it helps to guarantee

that no two trees in a heap have the same *degree*, a condition that is also imposed on this type of structure. Even if very similar to binomial heaps, Fibonacci heaps have a much less rigid structure as the consolidation (to keep the heap structure) is done somewhat lazy, i.e. *linking* is only executed after a Delete-min operation.

Definition 14 (Rank) *The number of children that a certain node x in a Fibonacci heap has, is called the rank of x, $r(x)$.*

In terms of implementation, Tarjan and Fredman explained it very clearly: *"Each node contains a pointer to its parent (or to a special node null if it has no parent) and a pointer to one of its children. The children of each node are doubly linked in a circular list. Each node also contains its rank and a bit indicating whether it is marked. The roots of all the trees in the heap are doubly linked in a circular list. We access the heap by a pointer to a root containing an item of minimum key; we call this root the minimum node of the heap."* [14]
As referenced, the rank of a node is essential to organise the heap, and to keep its structure. To have a well-formed Fibonacci heap we need that there exists at most one tree of any rank. Using *linking* we guarantee such condition, joining any two with the same rank. The by Tarjan and Fredman called *marking*, is used to backtrace how many children a node has lost: if node x has been cut from its child, then it is marked; if it loses another child, due to other operations, then x becomes unmarked and is itself cut from its parent, becoming a root node. This marking possibilitates better computational times for the *Decrease-key* operation as it forms more trees in the heap, sending nodes to the root. Let's now look at an example of this implementation with for the Graph in Figure 1.1, now numbered and weighted.

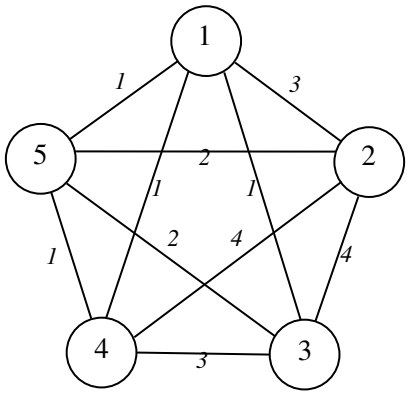


Fig. 2.6 A complete undirected weighted graph with 5 vertices and 10 edges.

We now want to show how the edges of this graph would be registered into a Fibonacci heap. We will not be showing every step of the creation as that would take too much space, but will insert an edge followed by a consolidation. We will admit that the edge to add is the one that connects vertex 3 to 5, without loss of generality.
Inserting the last edge into the heap, we will add it to the root list, side by side the 4, that is a tree itself, as seen in Figure 2.7. To consolidate, we would have to also *link* that node 2 with the previous node in the root list of weight 4. As $2 < 4$, 4 will become child of the added 2, leading to Figure 2.8.

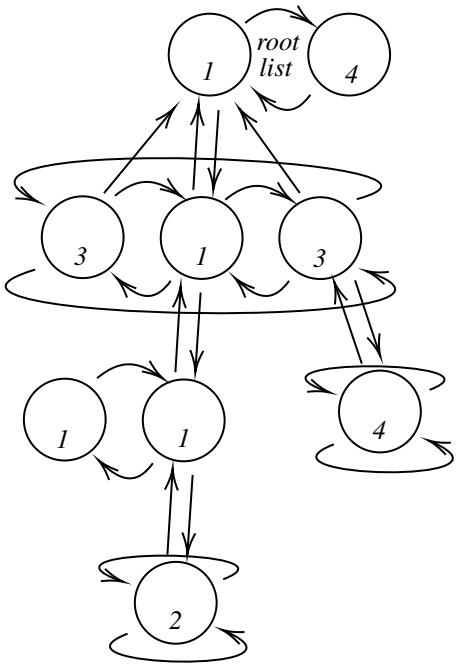


Fig. 2.7 Fibonacci heap containing 9 edges from the graph presented in Figure 2.6.

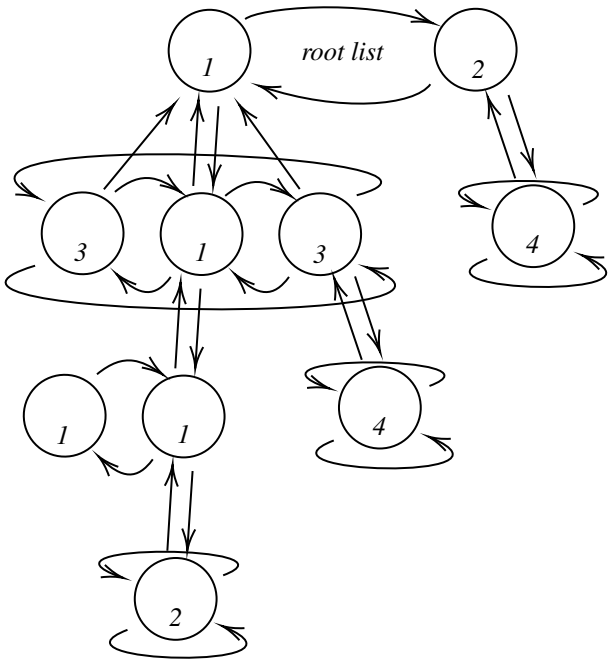


Fig. 2.8 Fibonacci heap of 2.6 with all 10 edges inserted.

Below we present the operations that Fibonacci heaps have to process it's nodes.

- Make-heap: here we only need to return a pointer to null;

- Union: we combine the root lists of two different heaps into a single list and return as the minimum node of the new heap is the smallest of the two (This operation is represented with an example in Figure 2.12);
- Insert: we create a new heap consisting of the one node we wish to insert and join the original heap and the one-element
- Find-min: we return the item in the root of the heap;
- Decrease-key: This operation can have three different cases: if the heap-property is not violated, then we only update the min-pointer, if necessary; if the heap property is violated and the parent of this node is unmarked, we mark the parent and add the node to the root list; if the heap property is violated and the parent is already marked, add the node to the root list, add the parent to the root list as well and mark the parent of the parent. If this one is also already marked repeat the process with it and its parents as well.

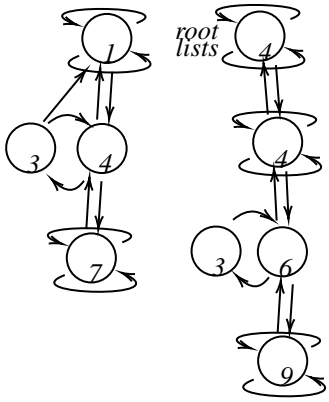


Fig. 2.9 (a)

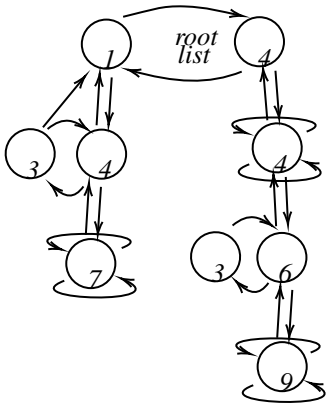


Fig. 2.10 (b)

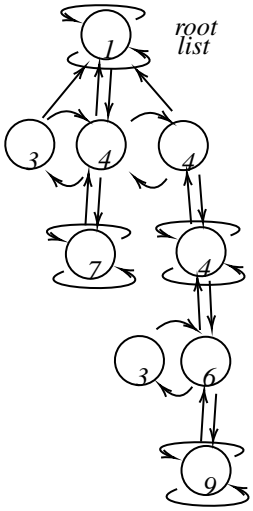


Fig. 2.11 (c)

Fig. 2.12 Example of the *Union* of two heaps from (a) to (c).

In the following we will describe how Delete-min works.

- We begin by removing the minimum node of the heap, which we may call x
- Secondly we need to concatenate the list of children of x with the list of roots of the heap that are not x
- *Linking*: this step is the most crucial one in this type of structure. Find two trees with the same rank and link them as before mentioned, by inserting the smaller root element as a child of the other.
- Repeat *linking* until no root elements have the same rank.
- We now form a list of the remaining roots, finding the one with minimum key, to use as the minimum node of the newly formed heap.

With all these operations we can work a heap as we like or need and save information about edges into it, and access fairly fast the minimum of these. In this work we chose to save into each node the weight of vertex of a certain graph. At the same time, we did not lose information about each edge as we always maintained the information about which vertices a certain edge connects.
These two data structures, binary and Fibonacci heaps, were essential in the implementations we developed in this work. Using them together with the MST algorithms that follow, we were able to distinguish their real advantages and disadvantages.

2.4 Comparing the different heaps

Now, before we are able to compare the different types of heaps we introduced, we need the definitions of functions that give us the ability to compare them.

Definition 15 (Amortised time) *Amortised time is the the average time a certain operation takes, even with occasionally having a worse or better result.*

Definition 16 (Big-Θ) *Big-Θ is used to study and compare different computational times. With this concept, we take into account the amortised time, best and worst case scenarios. As such, we may look at it as follows:*
For a function $f(n)$ that describes a certain time complexity, the actual time it takes may be between $k_1 * f(n)$ and $k_2 * f(n)$, depending on the used programming language, the computers speed or other non-algorithm related factors.

This means that Big-Θ is not a specific function but more of an area, i.e. if we say that a particular operation takes $\Theta(f(n))$ time, we mean that for n large enought, the running time is at least $k_1 * f(n)$ and at most $k_2 * f(n)$. Intuitively, with Big-Θ we say that we have an asympticaly tight bound, it applies for big enough n and it is narrowed between two constants.

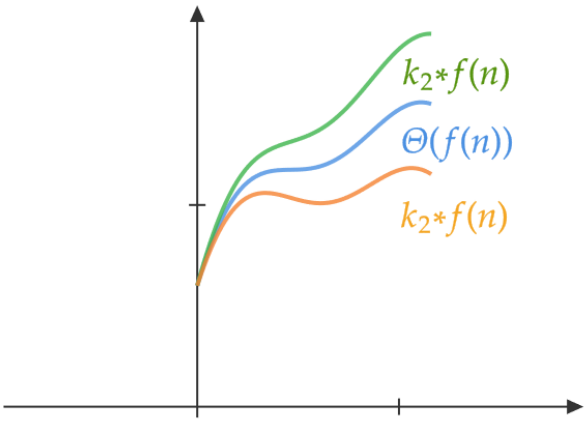


Fig. 2.13 Example of a Big-Θ function with upper and lower bounds.

Definition 17 (Big-O) *In a similar form as Big-Θ notation, we use Big-O to asymptotically bound the running time of an algorithm, but only upperly.*

If we say that running time is $\Theta(f(n))$, then it is also $O(f(n))$. The opposite is not necessarily valid.

The following table shows the different computational time costs for the different types of heaps we introduced before.[11]

Function	Binary Heap (worst case)	Binomial Heap (worst case)	Fibonacci heap (amortised)[14]
MAKE-HEAP [2]	$\Theta(1)$	$\Theta(1)$	$O(1)$
INSERT	$\Theta(log(n))$	$O(log(n))$	$O(1)$
FIND-MIN	$\Theta(1)$	$O(log(n))$	$O(1)$
DELETE-MIN	$\Theta(log(n))$	$\Theta(log(n))$	$O(log(n))$
DECREASE-KEY	$\Theta(log(n))$	$\Theta(log(n))$	$O(1)$
UNION	$\Theta(n)$	$O(log(n))$	$O(1)$
DELETE	$\Theta(log(n))$	$\Theta(log(n))$	$O(log(n))$

Table 2.1 Table with the computation times of the heaps' main functions. The number of values in the heap(s) is here denoted by n. In the case of UNION, n is the sum of elements of both heaps.

[2] This is the running time to construct an empty heap.

Chapter 3

Algorithms for the MST problem

From the different algorithms to obtain a Minimum Spanning Tree, we chose two particular approaches that we mentioned in the beginning of this work and we now choose to describe in detail, the Kruskal and the Prim algorithms.
In what follows we will only study undirectioned weighted graphs.

3.1 Kruskal

3.1.1 Kruskal Algorithm

For this algorithm we need that all the graph's edges are ordered by their weight, from smallest to biggest.

1. Create an empty tree;
2. Pick smallest edge of the graph. Verify that if this edge forms a cycle in the tree we are creating. If it does, exclude that edge. If it does not, add it to the tree.
3. Repeat Step 2 until there are $(n-1)$ edges in the tree.

With this implementation it is easy to understand that we will not be creating only one tree. If the cheapest edge is between two vertices that are not yet included in the MST, we will be creating a second (or n-th tree), that later will be added to the MST. In this sense, and to also prevent the formation of cycles, we will use an auxiliary array where we will register what vertex *represents* each vertex.

3.1.2 Representatives

For each of the edges, we will be able to sort them into different trees. These will be the ones that are added into the final minimum spanning tree. The idea follows:

- In an auxiliary array, we register what vertex represents each vertex - at the beginning each vertex represents itself;

- When adding an edge to the MST, we register which vertices it connects and verify in the auxiliary array which vertices represent the ones being connected in the MST. Let's consider vertices a and b and $a < b$.

- If in the auxiliary array a and b are represented by themselves (i.e. their number is the same as their index value, so the value of a is a, and value of b is b), we define that b is now represented by a.

- If in the auxiliary array one of the vertices, a, is represented by some vertex than not itself (so, value of $a \neq a$), we iterate from vertex to vertex until we find a vertex that is represented by itself, say x. We then define the original vertex a is represented by x. As b is represented by itself, we define now that it is represented by x as well.

- If in the auxiliary array both of the vertices are represented by some other than themselves, we do a similar approach of the point presented before but the one represented by bigger vertex index x must now be represented by the vertex of the other tree (representative of the other vertex, of the edge that is being added). If by chance, the value of b is x, i.e. a and b are represented by the same vertex, we cannot add this edge to the MST, as it would form a cycle. Therefore this edge is rejected from consideration.

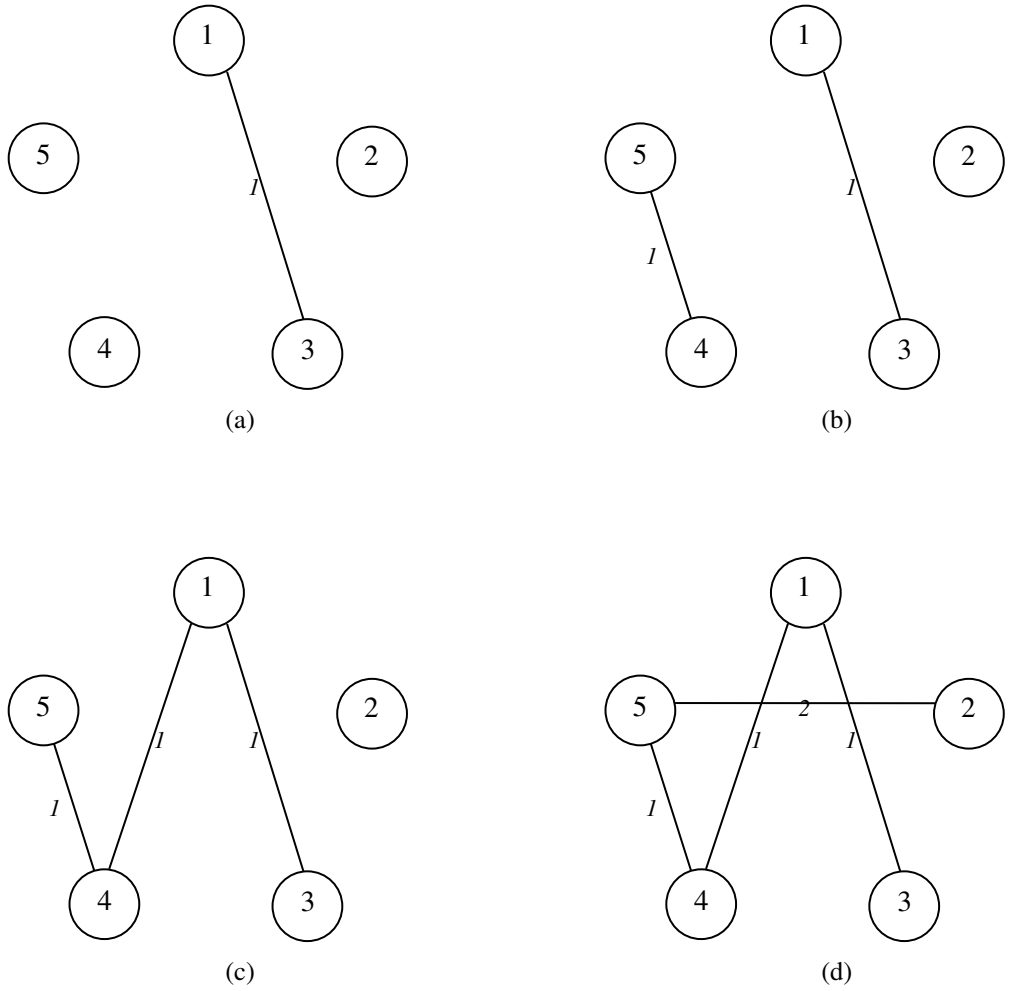


Fig. 3.1 The Kruskal algorithm for the graph in Figure 2.6.[1]

3.2 Prim

In a similar way as the before presented Kruskal Algorithm we will also need to organize the edges by their weights and do this for each of the vertices i.e., we need to know for each vertex which are the cheapest edges connecting that vertices to others.
We will also have to keep track which vertices are already included in the MST.

Definition 18 (Visited vertices) *The register of which vertices have been added in the MST will be done using an auxiliary array where it will be registered if an array has been visited or not.*

3.2.1 Prim Algorithm

1. Pick a vertex at random (in this work we will always start at the one with lowest index) and add the cheapest edge that connects that vertex to another.

2. From the whole set of edges that connect the vertices that already are in the MST to others, select the cheapest one , making sure the new vertex that will be added has not yet been *visited*.In case of ties, the selection of the cheapest edge to be added, is done at random, from all edges with minimum value.

3. Repeat Step 2 until there are $n-1$ edges in the MST.

In the example that follows, the algorithm becomes clearer. Starting from vertex 1, we add one of the edges with weight 1, let's say the one that connects 1 and 3, obtaining (a). From all edges that connect the vertices 1 and 3 to the remaining 2,4 and 5, select the cheapest again, leading to (b). Repeating this idea gets us (c) and then (d).

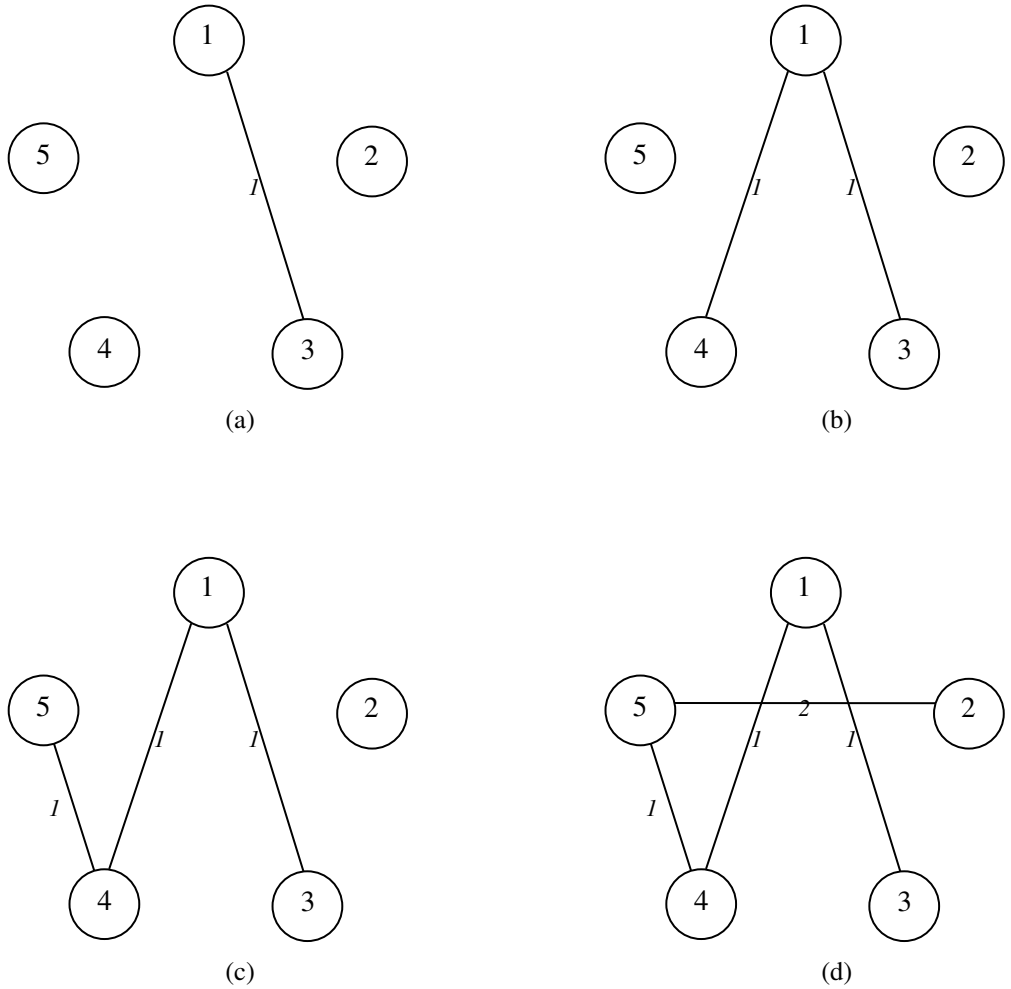


Fig. 3.2 The Prim algorithm for the graph in Figure 2.6.[2]

[2]Represents, organised by index $(1,2,3,4,5)$: (a) - $(1,2,1,4,5)$, (b) - $(1,2,1,1,4)$, (c) - $(1,2,1,1,1)$, (d) - $(1,1,1,1,1)$

3.3 Implementations

Having now all the bases, it is possible to look at the implementation that were done for the elaboration of this study. These implementations were written in *Python*.
As a starter, we implemented a very simple adjacency matrix graph representation and developed the two mentioned Prim and Kruskal Algorithms. As literature already suggested, this was not an efficient data structure for fast MST calculation and therefore rapidly excluded from further implementations and results we wanted to present here.

3.3.1 Adjacency List

With the objective of comparing the various implementations, we introduced the Adjacency List representation.

Definition 19 (Adjancency List) *An adjacency list is an array of size k in which we may find in each entry i a linked list, corresponding to the edges of i/th vertex. In these linked list, each entry j is the weight associated to the j-th edge that is connected to the i-th vertex. In another adjancency list we may then find, in the same entries, to which other vertex that edge connects itself to.*

Using this previous definition, we were able to not only register the entire graph and save it, but also to calculate the MST directly, using the Prim and Kruskal Algorithms.
Although not famously known for it's good computational time cost, it's simplicity gave the possibility to implement in *Python* and using *Numpy* very fast algorithms for smaller sized graphs. As most the operations used for working with this data type were already functions or libraries in this programming language, and these are already highly optimised because they can work multi-core, it seemed to be a very good option. As theory suggests, this was indeed a fast approach for these smaller graphs.

Registration of the graph

To create graphs of the size we wish, complete or with random edges, we used a graph generator, provided by Prof. Doutor José Luís Esteves dos Santos. In Chapter 4, we will later specify which we chose to create. These graphs were saved in a *.dat* file and with *Python* imported into the program we wrote. Reading one of these files, our program registered all vertices and edges that connect them into lists and arrays using adjacency lists in forward star representation. Later on, these arrays with the entire data were called into the algorithms to create the MST's.

Definition 20 (Forward Star Representation) *Although similar to an adjacency list, the forward star representation holds one big difference: all information is stored in one single array instead of a linked list.*
We need to store all edges into an array in order from where they start, i.e., first all that connect node 1 *to others, then all that connect node* 2*, and so on. In the same order we store in other arrays, the information of to which that edge connects vertex i to j, in "head", and the weight of that same edge, in "cost". At the same time we need to have an auxiliary array point where we store in entry j the smallest numbered edge (index in the order we saved them by) that leaves vertex j.*

Table 3.1 shall help understand this concept. In this example we based ourselves on the graph in Figure 2.6, an undirected graph. Knowing this, it is important to notice that each edge will be registered twice, as for a certain edge, leaving a vertex is the same as arriving at it. In the table below we may see on the left-hand side the *point* array, detailing where the first edge for every vertex is. On the right-hand side are the other arrays. As an example, let's see the entry 10: in *tail*, we may see that this vertex goes from the vertex 3, to the vertex 2, saved in *head*. The corresponding cost of this edge is readable in the most right column.

point
1
5
9
13
17

	tail	head	cost
1	1	2	3
2	1	3	1
3	1	4	1
4	1	5	1
5	2	1	3
6	2	3	4
7	2	4	4
8	2	5	2
9	3	1	1
10	3	2	4
11	3	4	3
12	3	5	2
13	4	1	1
14	4	2	2
15	4	3	2
16	4	5	1
17	5	1	1
18	5	2	2
19	5	3	2
20	5	4	1

Table 3.1 Example of a forward star representation of the graph in Figure 2.6.

3.3.2 Heap representation

Similarly to the adjacency list, we had to develop the heap structures in code, which was also written in *Python*. Here we used open-source code to base ourselves on the approaches of other programmers. We were able to find code by Dan Stromberg, who *translated* into *Python* what Keith Schwarz originally wrote in *Java* [Stromberg], Daniel Borowski's implementation and Quang Tran's reposit [Tran], concerning Fibonacci Heaps. All three of these were very helpful but had to be somewhat adapted to our needs, as we had the data saved in a Forward Star Representation and we wanted to save the pointer to the node and the weight of it. As it may be intuitive, the Binary Heap structure was easier to implement, and we were able to use almost directly the works of Bradley Miller and David Ranum [24].

Having the structures functional we then needed to use the operations presented in Chapter 2 in the Prim and Kruskal Algorithms. All functions were developed by the authors, and during the months of work, improved and optimized. The code is commented throughout to be understandable and used by anyone who wishes to do so.

Chapter 4

Computational performance of the implementations

In this chapter we will present results on the different implementations we explained in the previous chapter. As mentioned before, we will focus on 6 different implementations:

- Kruskal Algorithm using Binary Heaps;
- Prim Algorithm using Binary Heaps;
- Kruskal Algorithm using Fibonacci Heaps;
- Prim Algorithm using Fibonacci Heaps;
- Kruskal Algorithm using Adjacency Lists;
- Prim Algorithm using Adjacency Lists.

To study the effectiveness of these implementations, we divided the study into two parts: complete graphs, because of their high density and most amount of possible edges; and sparse random graphs, with a fixed density of 3 or 6, i.e. in average, each vertex of a graph is connected to 3 or 6 others.
For each size, we ran 50 simulations, i.e. 50 different graphs, with a fixed number n of vertices and edge weights between 1 and 4000. All times can be found in the Appendix A of this work, explicitly written into tables.
We will begin by showing the results for smaller size graphs and will then show the simulations for bigger graphs.

4.1 Small Graphs

In the figures that follow Graph Size is to be understood as n, the number of vertices in each.

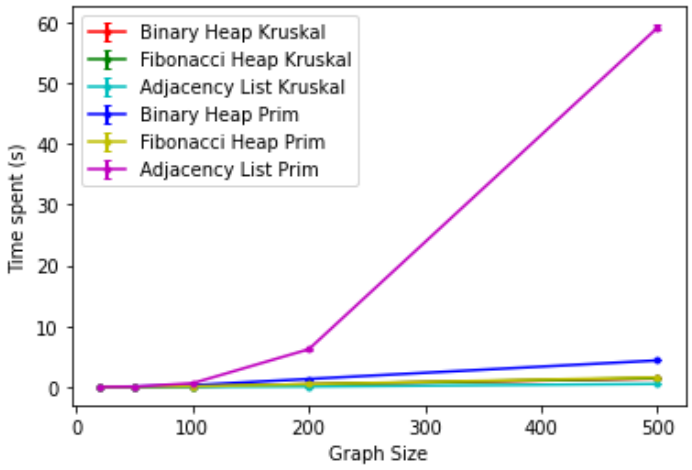


Fig. 4.1 All implementations for small complete graphs.

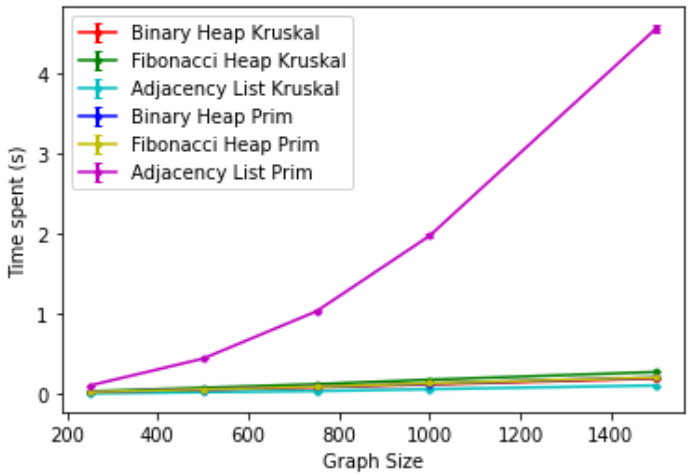


Fig. 4.2 All implementations for small random graphs with density 3.

From both the simulations for small graphs we were able to conclude that the Prim algorithm using adjacency lists was not the best option as the time it took was much bigger than all others. Therefore we excluded them from the next results and from tests for bigger graphs.

4.2 Large Graphs

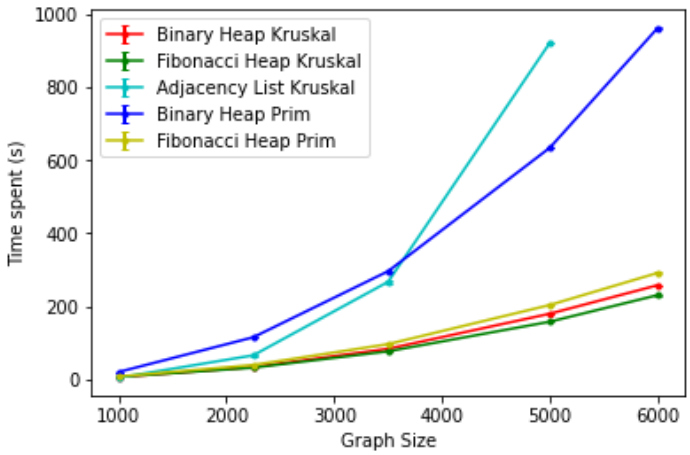


Fig. 4.3 Implementations for large complete graphs.

Even though the adjacency list implementation held on very good and was very quick for smaller graphs, it was not efficient enough for bigger data as each operation took too much computational time. We may also see in this plot that what we described in the table 2.1 shows it's differences very clearly here. As in the Prim algorithm we need to join the *to be compared trees*, we use the operation *UNION* at each addition of an edge into the MST. Having a $\Theta(n)$ *vs* $\Theta(1)$ may be an advantage for smaller graphs as the constant time for one repetition may be bigger than desired. But, as n grows, the time it takes a binary heap to merge with another also increases, but for a fibonacci heap it is fixed and therefore better for bigger graphs. Overall, we are able to conclude that Kruskal's algorithm is indeed faster than Prim's, for any implementation.

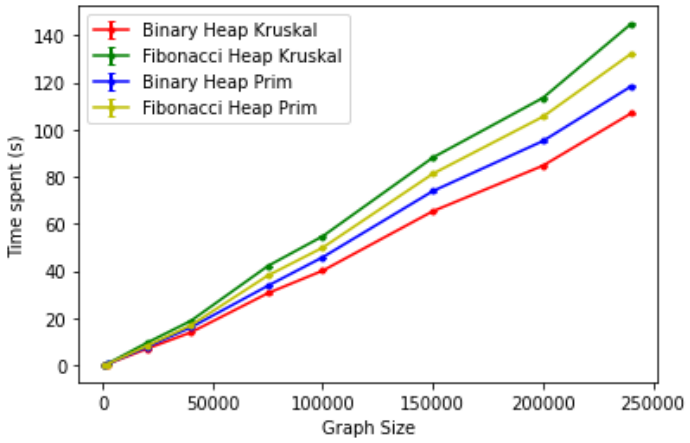


Fig. 4.4 Implementations for large random graphs with density 6.

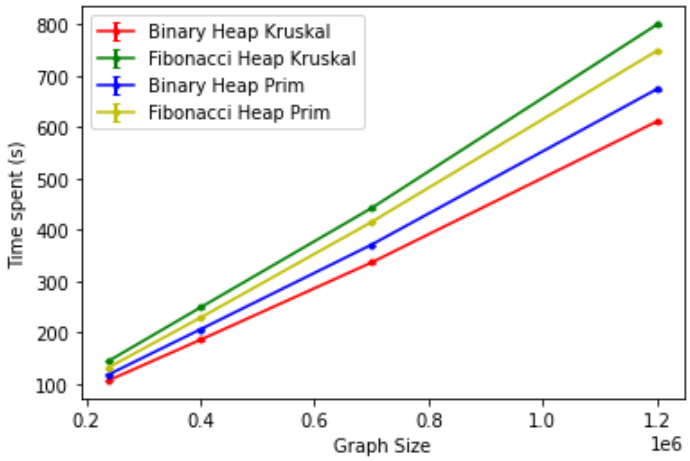


Fig. 4.5 Implementations for even larger random graphs with density 6.

From Figure 4.4 it is not obvious, as before, which implementation proves to be the most efficient. In this sense, we wanted to study the behaviour of the algorithms with graphs with density 6 and the same amount of edges as the complete graphs shown before. For this, we would have to create graphs with 6 million vertices. This proved to be impossible with the computational power that was at hand. We were able to reach graphs with 1.2 million vertices, where each graph took 10 hours to be created. As the time for the graphs to be created did not grow linearly, the computation of graphs with 5 million vertices proved to be impossible. Although we tried, the computer was not able to create one with this size, even after 132 non-interrupted hours of computation. Nevertheless, we can compare the efficiency of the algorithms for high and low dense graphs. For a complete graph with 2600 vertices we are able to see that the Binary Heap with Prim algorithm implementation proved itself much less efficient. The same was not observable for a graph with density 6 and similar amount of edges, there the time it took to solve grew constantly. This leads us to conclude that these algorithms are quite similar in terms of efficiency for sparse graphs but, especially the Fibonacci heap implementations, are much more efficient for very dense graphs, as the computational times for the complete graphs, with the same amount of edges are much lower than for sparse graphs. This also means that their efficiency is necessarily related to the amount of edges that a graph has, as it would be expected. Since the Big-O times depend on the number of elements in a data structure it would be expected that, for the same amount of elements, here the edges, similar times are computed. We conclude that this must be related to the amount of possible vertices there are to be checked on, to verify if a MST has been formed and has connected all vertices of the original graph. Generally, it is clear that the heaps implementations are far better in terms of computational times than the adjacency lists.

Chapter 5

Ranking of spanning trees

In this section we will be exploring further the creation of Minimum Spanning Trees - concretely, the k best spanning trees of a graph G. In a real application, sometimes it is not only relevant to know the cheapest way of connecting all points: it can be also very relevant to know all the possible ways to connect these points and to order them by their cost.

In this sense, we will explain what has been explored in this area in terms of different algorithms and approaches to obtain this ranked list of possible Minimum Spanning Trees.

5.1 Historical Background

Various efficient algorithms have been proposed since the beginning of exploration of the subject. As early as 1954 where Trent was able to give formal ways to retrieve the number of possible different Spanning Trees of a Graph and specifying them [22], and continuing on the 1960's where many authors were able to present different algorithms on how to retrieve efficiently these spanning trees [2, 7, 18, 23]. The interest in these algorithms, and creating ways to retrieve these trees, continued in the 1980's and thenceforth, as computational power grew and researchers were able to improve the before created implementations and reduce computational costs. More complex and advanced implementations were developed in the 2010's, so that in 2018, Chakraborty et al. were able to compare many of the until then developed algorithms and rank them for their computational space and time costs [6].

Doing some literature review we were able to identify three major approaches when it came to the creation of possible Spanning Trees. For each of these we will refer one algorithm as example.

5.2 Trees by test and select

This method upholds itself on one major underlying notion: every spanning tree of G has exactly $n-1$ edges and does not form a cycle, for graph G with n vertices and m edges. Knowing this it is easy to follow that there are ${}^{m}C_{n-1}$ distinct edge combinations, from which we need to understand which are spanning trees. Each algorithm of this type has two phases:

1. Using some logic behind to reduce the total amount of tries, the algorithm will create possible and plausible edge combinations;

2. From the combinations created in the first phase it will identify and select the spanning trees that cover G.

From the found algorithms that based themselves on this approach one standed out as it created the least amount of possible sequences in the first step, therefore saving up in space and time complexity as it did not need to register and test as many sequences as other algorithms:

5.2.1 Onete's algorithm [26]

This recent algorithm was created by Onete et al. in 2010. This algorithm uses a structure that is called a Reduced Incidence Matrix. To understand this one, we need to understand the definition of a Incidence Matrix before:

Definition 21 (Incidence Matrix) *Given a graph with n vertices and m edges, an Incidence Matrix is a $n \times m$ matrix IM where each entry IM_{ij} corresponds to the weight of the edge j that connects vertex i to another vertex in a graph.*

Definition 22 (Reduced Incidence Matrix) *A RIM_j is obtained by removing the row with reference to vertex j of a Incidence Matrix.*

The following matrices are based on the graph in Figure 2.6 and shall help understand the before presented definitions. The shown RIM_2 is considering vertex 2 as reference.

$$IM = \begin{pmatrix} 3 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 3 & 4 & 0 & 0 & 0 & 0 & 0 & 2 & 4 & 0 \\ 0 & 4 & 3 & 0 & 0 & 1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 1 & 0 & 0 & 0 & 0 & 4 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 2 & 2 & 0 & 0 \end{pmatrix}$$

$$RIM_2 = \begin{pmatrix} 3 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 4 & 3 & 0 & 0 & 1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 1 & 0 & 0 & 0 & 0 & 4 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 2 & 2 & 0 & 0 \end{pmatrix}$$

Having understood these definitions, we may go on to the actual algorithm.
Using an auxiliary diagonal $m \times m$ matrix D, where the diagonal elements are all m edges, we may calculate the matrix U, which will help creating all possible MST's.
U is calculated in the following way:

$$U = D \cdot (RIM)^T$$

From this $(m \times (n-1))$-matrix U one is able to retrieve non-singular submatrices (This means submatrices U_i of size $(n-1) \times (n-1)$, with $Det(U_i) \neq 0$). If U_i contains at least a row with only one entry and the columns can be permuted s.t. all diagonal elements are nonzero, then these diagonal elements correspond to the edges in a possible spanning tree of G.

5.2.2 Discussion on this approach

This method should not be used for dense graphs as the possible combinations increases exponentially for more edges there are in G. One big advantage of this approach, in particular of the presented Onete's algorithm, is the possibility of parallelising the construction of the possible combinations, speeding up the whole algorithm.

5.3 Trees by elementary tree transformation

Oppositely to the before presented approach, this method starts with the direct creation of a Spanning Tree, generally using a Breadth-First or Depth-First Traversal.

Definition 23 (Traversal) *The path that is done to percolate all vertices of a graph, creating a spanning tree.*

Definition 24 (Breadth-First Traversal) *As the name of this traversal intuits, we start at a random node i of G and start by adding, on by one, all the edges that connect that node to others in G, starting with j, e.g.. If it has done so and the graph is still not connected, it repeats the process for node j, only adding edges that do not connect that node to a before visited one. It stops at $n-1$ edges, where the tree will be connected.*

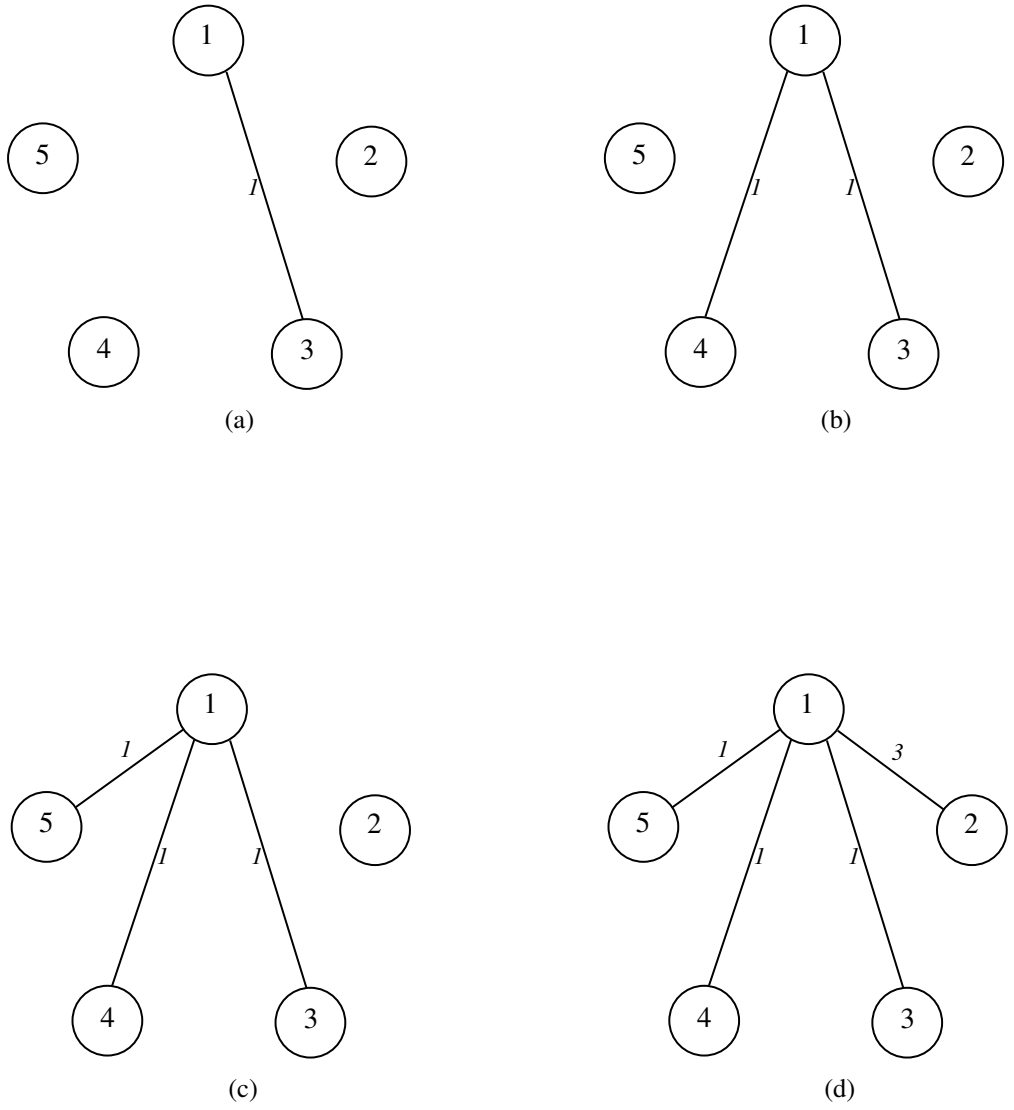


Fig. 5.1 Breadth-First Traversal of the graph in Figure 2.6.

Definition 25 (Depth-First Traversal) *Opposed to Breadth-First, this traversal goes trough an graph continuing its path from the last visited vertex, i.e.: Starting at node i, the edge that connects i to j is added, followed by the edge that connects j to k, for $i \neq j \neq k$, maintaining the tree structure. If some vertex in this process is not connected to any remaining not-visited vertices, connect the last vertex to another not yet visited vertex. If needed, repeat this last step recursively until a spanning tree is created.*

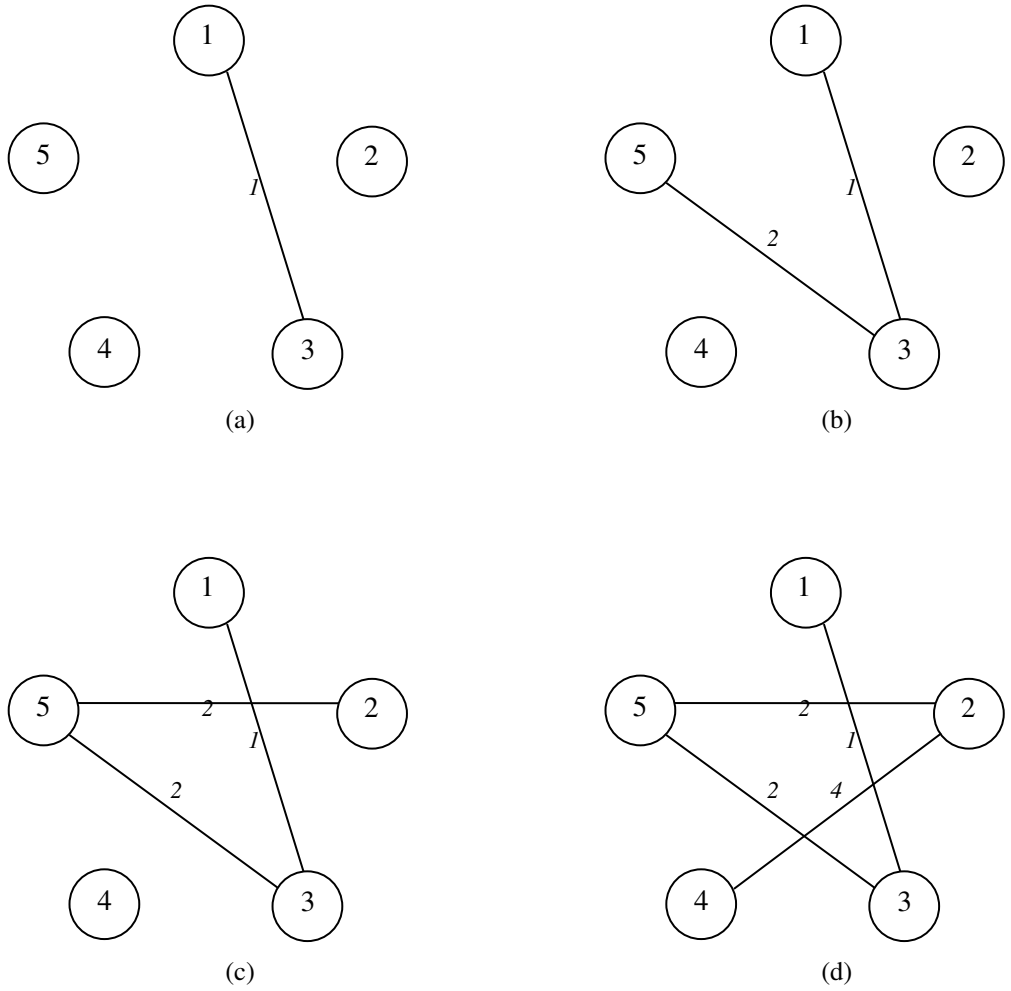


Fig. 5.2 Depth-First Traversal of the graph in Figure 2.6.

As these examples show, it is very clear that neither of these traversals guarantee a minimum spanning tree, they only create a spanning tree, with no concern to the graphs' weights.
After retrieving a spanning tree of G, one is able to, quite intuitively, divide G's edges into two subsets: included and excluded edges, created by the set of edges that were included in the spanning tree, and those that were not. The idea behind this approach is, at each step an edge of the included set is replaced by one of the excluded set s.t. no cycle is formed and all vertices remain connected, i.e. it remains a spanning tree. Many algorithms using this approach have been presented and improved. Here we will present a specific one, presented by Shioura and Tamura in 1993 [1].

5.3.1 Shioura and Tamura's algorithm [1]

As mentioned in the chapter before, these approaches start with the creation of spanning tree. With this algorithm, we find it using Depth-First Traversal. Let us define S_0 as the Depth-First spanning tree of G. As also mentioned before, this gives us the mutually exclusive subsets of G's included edges

INC and excluded edges *EXC*, with $INC = S_0$ and $EXC = E \setminus S_0$. Let's admit $INC = \{e_1, e_2, \ldots, e_i\}$ and $EXC = \{e_{i+1}, e_{i+2}, \ldots, e_m\}$. To create all possible spanning trees, this algorithms picks one of the edges in the included set at a time, in order, and replaces it with one edge of the excluded set, in order as well, adding it to the list if it remains acyclic and covers all vertices. After having done so one time, it repeats itself recursively with a not-yet used edge of the excluded edge. The following example will help understand this algorithm in a more detailed way.
Let's consider the following graph:

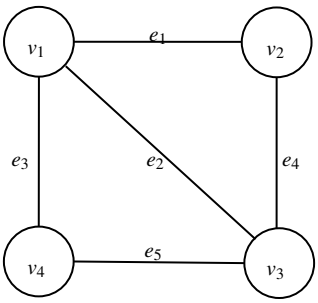


Fig. 5.3 A connected graph.

Starting at vertex v_1, we create the first spanning tree

$$T_0 = \{e_1, e_4, e_5\}$$

using the depth first traversal. Having T_0 we can create others switching the edges as described before:

$$T_{01} = \{e_2, e_4, e_5\}, T_{02} = \{e_3, e_4, e_5\}, T_{03} = \{e_1, e_2, e_5\}, T_{04} = \{e_1, e_3, e_5\}, T_{05} = \{e_1, e_4, e_3\},$$

Exchanging e_5 by e_2, in similar way as all others, we would not get a spanning tree, because a cycle would be formed: $T_{21} = \{e_1, e_4, e_2\}$.
Doing this recursively we will still be able to create the following:

$$T_{012} = \{e_2, e_4, e_3\}, T_{052} = \{e_1, e_2, e_3\}$$

5.3.2 Discussion on this approach

The main challenge of this approach is to guarantee the noncreation of duplicate trees. This is guaranteed in the before presented Shioura and Tamura algorithm. It also guarantees that only trees are generated, saving up in computational time.

5.4 Trees by edge reduction

The big concept of this approach is to divide a graph into smaller sub-graphs which are easier to compute and therefore faster to find a spanning tree. This reduction is done recursively until a subgraph is only an edge. With this approach, we can guarantee that only distinct trees will be generated, saving up the time that was before spent on tree testing.

The algorithm that follows will help understanding this method in general. We will explain an algorithm developed by Winter in 1986 [34].

5.4.1 Winter's algorithm [34]

Before we start describing this algorithm it is important that we explain what a *contraction* is.

Definition 26 (Contraction) *Without loss of generality, let's assume that $e = i, j$ is an edge of G with $i < j$. When we say that a vertex i is contracted into another vertex j, we mean that the edge that unites these two ceases to exist and all edges connected to j are now connected to i. When a contraction happens, the contracted vertex i is the highest labeled vertex in the graph G and j the highest labeled vertex adjacent to i.*

The algorithm begins by selecting a certain edge e of G and constructing all spanning trees which include e. Then, it constructs all spanning trees which include another edge f but do not include e, and repeats the process for all edges.
Knowing this, the algorithm works as follows: the graph G is registered into a root node in a computation tree. To add the children of this tree, the node suffers from a contraction or an edge elimination, therefore creating two possible children. The elimination of edge e (which connects vertices i and j) happens only if the vertex i has other adjacent vertices than j. The contraction happens at each node until the graph is reduced to a single vertex (labeled 1, as it is the highest labeled). The contracted edges are stored in sequences, which correspond to the paths between the root node and the leaves of the created binary tree. From these sequences we may retrieve the possible spanning trees of G.

5.4.2 Discussion of this approach

One big advantage of this approach and specially the presented algorithm is that it may be parallelised, as it can examine more than one sequence at the same time. This obviously permits the computational time to be much lower. The dispensable tree searching, as usual for other approaches, is also a very big advantage as it also saves in time and space complexity.

5.5 The chosen algorithm

Even with these very good algorithms, we chose to pick a different approach and algorithm than the ones mentioned before, as the objective we wanted to fulfil was to find the k-best spanning trees, without needing to create and register all and then rank them. For this we chose to follow [29], which gives us the possibility to stop the algorithm after we have found the first cheapest k spanning trees.
It will be interesting and clear to the reader that throughout this implementation we use some characteristics of all three approaches presented before.

5.5.1 Implementation

As we wanted to build onto the work that was done before, the different implementations to find a Minimum Spanning Tree of a certain graph, we chose to use the Kruskal algorithm with graph

registration in adjacency lists as it was the most adaptable to our needs. The first step of this algorithm is to find the MST of the graph G, let's call it S_0. This MST may be written as $S_0 = \{e_1, e_2, \dots e_{n-1}\}$, for edges e_1 to e_{n-1} ordered from cheapest to most expensive, cost-wise. As this is the cheapest spanning tree of the graph G, it may be added to the list of spanning trees, being the cheapest spanning tree. In a similar way as in the Section 5.3 method, we will be partitioning this set S_0 into different subsets with elements that shall be included or excluded. To simplify, we will use the notation e_i for included and $\overline{e_i}$ for excluded edges e_i. Having this said, we partition S_0 into the following $n-1$ partitions:

$$
\begin{aligned}
&P_1 = \{\overline{e_1}\} \\
&P_2 = \{e_1, \overline{e_2}\} \\
&\dots \\
&P_i = \{e_1, e_2, \dots, e_{i-1}, \overline{e_i}\} \\
&\dots \\
&P_{n-1} = \{e_1, \dots, e_{n-2}, \overline{e_{n-1}}\}
\end{aligned}
$$

For a spanning tree S we shall partition it into $n-1$ sets where each set P_i shall have the i-th cheapest edge e_i, as an excluded edge and all the edges in S that are cheaper than e_i, as included. These partitions will be added to a list L, that the algorithm will now, and later on, use.

At every repetition, after partitioning a spanning tree, we need to pick every set in the L and with that, using the included and excluded edges create the possible spanning trees that shall include the included edges and not consider the excluded edges while computing the spanning tree. If a spanning tree is not constructable with a certain set, this set may be eliminated from L. The cheapest of the spanning trees created from the different sets of L, S_1 shall be included into the list of spanning trees, being now the second cheapest spanning tree. Then we have to partition S_1, as done previously with S_0, taking now into account the excluded edges of the set that created S_1 and repeat the explained process until we have added S_{k-1}, the k-th cheapest spanning tree. At each step all possible partitions are added to L, ensuring that to no possible tree is excluded.
The following pseudo-code helps to implement this algorithm with a computer.

First we will describe how the Partition function works and then the whole algorithm.

Data: Partition L_l, with included and excluded edges discriminated
Result: Partitions formed by L_l
if *L_l forms a valid spanning tree S* **then**
  for *All edges in S_l, e_i* **do**
    Add e_i to the excluded set
    for *All $j < i$* **do**
      Add e_j to the included set
    end
    if *Set of included and excluded form a valid Spanning tree* **then**
      Add partition of included and excluded to L
    else
      Nothing
    end
  end
else
  Nothing
end

Algorithm 1: Partition Function

Data: Graph G
Result: The k cheapest spanning trees
$S_0 = MST(G)$
$L = \{\}$
Partition(S_0)
Add S_0 to the Result List
while *not k spanning trees in Result List: ($j < k$)* **do**
  $j = 1$
  for *all elements in L, L_i* **do**
    $MST(L_i)$
    if *If $MST(L_i)$ is cheaper than $MST(L_{i-1})$* **then**
      $S_j = MST(L_i)$
      Increment j
    else
      Nothing
    end
  end
  Add S_j to the Result List
  Remove L_i from L
  Partition(L_i)
end

Algorithm 2: Create the k cheapest spanning trees

The implementation of this algorithm in *Python* may be found in the Appendix B.

5.5.2 Algorithm evaluation

The computational time of this algorithm, for the creation of all spanning trees, was reported by Sorenssen and Janssens to be $O(p \cdot n + n + m)$ [29] (with p being the number of possible different partitions that are creatable), which is actually worse than the computational times of the before mentioned algorithms. Still, as said before, this algorithm presents the opportunity to be stopped after k spanning trees have been found, and if the k chosen is not close n^{n-2}, this is a major advantage and allows shorter total times.

Chapter 6

Possible applications and Final Thoughts

In this chapter we will present a possible applications of the presented MST algorithms and some considerations about this work.

6.1 Clustering

This application we would have liked to develop in this work as well is very relevant in classification and grouping of any kind of data. As the amount of data grows, the importance of partitioning it does as well. With this capacity, we are able to detect anomalies in a certain dataset, image or even a social network, for example. In the last 20 years, this has become very important as large companies have to keep track of their online computerised data, for it to be organised and more accessible. Also in image processing clustering has proved itself relevant, as it provides the ability to segment it into different parts, and therefore facilitates the identification of anomalies, in medical images for example.

More than 5 decades ago, Zahn was the first to propose an MST-based clustering algorithm [35]. The basic idea he proposed was to identify and remove inconsistent edges of a MST until a certain number of clusters has been reached.

Definition 27 (Inconsistent Edge) *Inconsistent edges are edges whose weight is larger than the average of the nearby edges' weights.*

The following example shall help understand the definition. In our simple one-dimensional example, let's consider the refered weights, the distances between the vertices. If we observe these distances, it becomes clear that the edge that connects vertex a to b is heavier than the others around or in proximity of any of the 2 vertices. This is an inconsistent edge.

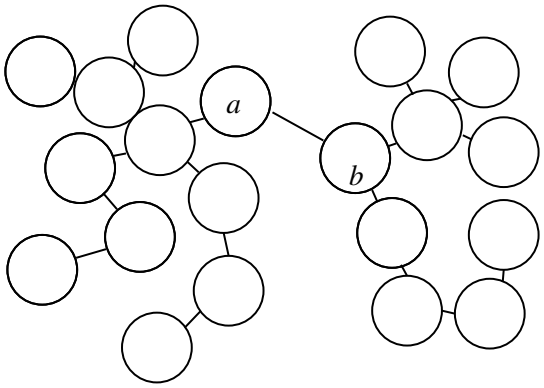


Fig. 6.1 An example of an inconsistent edge in a MST of a graph.

This approach depends hardly on the calculation of this average. For this reason it is commonly used and applied only on graphs with up to 2 parameters, which may be calculated with the usual one-dimensional average, for one parameter and more complex average functions using double integrals, different metrics or specific statistical functions. As clustering becomes more and more interesting for multi-objective graphs, the calculation of this average does as well. In this work we focused ourselves on the one-dimensional graphs, i.e. one weight parameter. This means that to use this application it would be fairly easy to implement, as the calculation of these inconsistent edges would not be a problem. If we would generalise it for higher dimensions, then this calculation would be very important and many algorithms have been presented for clustering, distinguishing themselves on different measures and heuristics to do these calculations. One major advantage of Zahn's simple approach is that the points of the cluster are not grouped around a certain centre, and therefore do not have a specific geometrical form. This possibilitates the detection of irregular shaped clusters. The development of a clustering algorithm onto the work that has been done in this thesis is something that we would have liked to explore. For this we would have also have had to implement the developed algorithms for multi-objective graphs, to have an real-life application.
During the time of writing, the author was enrolled in an internship at an energy company, in the procurement unit. It became clear that there, the devolved algorithms would be useful. Although not implemented, the idea will be left here as an example of practical usefulness of these algorithms. For any company with a wide grid of suppliers, it is important to consult as many suppliers as possible for the acquisition of a certain product. It is clear that not all suppliers can be consulted for every need. In this sense, it seems to be crucial that suppliers are allocated into different categories/groups, in our case clusters. As such, we would be able to organize all suppliers depending of what they have supplied before, defining weights/distances between them depending on the common products they have provided or are able to provide. It would be strictly necessary to define a weight/distance function, that could be redefined depending on the needs and tests that the business unit would have.

6.2 Listing

The information presented in Chapter 5 of this work has a quite intuitive and useful possible application. When we think about transportation or delivery systems/services, these need to calculate the best

paths between all delivery addresses, minimising distances or costs i.e., using the roads(edges) that exist in the map(graph), covering all addresses (vertices). This calculation is based not only on one parameter such as distance, but also on toll costs or others. In this sense, it could be very useful to use this ranking of the best minimum spanning trees to present a user of such system not only the best choice, taking into consideration total time or total covered distance, but to present to it the next best choices, so that the user may choose routes that go specifically to her or his needs and desires. We leave this application open, as it is necessary to adapt the presented algorithm to higher dimensions. This is a possible application that could easily be implemented in the future, constructing on what has been developed in this work.

References

[1] A., S. and A., T. (1993). Efficiently scanning all spanning trees of an undirected graph. *J. Operation Research Society Japan*, 38:331–344.

[2] Berger, I. (1967). The enumeration of trees without duplication. *IEEE Transactions on Circuit Theory*, 14(4):417–418.

[3] Boruvka, O. (1926). O jistém problému minimálním. *Práce Mor. Prırodved. Spol. v Brne (Acta Societ. Scienc. Natur. Moravicae)*, 3(3):37–58.

[4] Callahan, P. B. and Kosaraju, S. R. (1993). Faster algorithms for some geometric graph problems in higher dimensions. 93:291–300.

[5] Cayley, A. (1889). A theorem on trees. *Quart. J. Math.*, 23:376–378.

[6] Chakraborty, M., C. S. C. J. e. a. (2019). Algorithms for generating all possible spanning trees of a simple undirected connected graph: an extensive review. *Complex Intell. Syst.*, 5(1):265–28.

[7] Char, J. (1968). Generation of trees, two-trees, and storage of master forests. *IEEE Transactions on Circuit Theory*, 15(3):228–238.

[8] Chazelle, B. (2000). A minimum spanning tree algorithm with inverse-ackermann type complexity. *Journal of the ACM (JACM)*, 47(6):1028–1047.

[9] Chong, K. W., Han, Y., and Lam, T. W. (2001). Concurrent threads and optimal parallel minimum spanning trees algorithm. *Journal of the ACM (JACM)*, 48(2):297–323.

[10] Choquet, G. (1938). Étude de certains réseaux de routes. *Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences*, 206:310–313.

[11] Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. (2001). *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition.

[12] Dijkstra, E. W. et al. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271.

[13] Elkin, M. (2020). A simple deterministic distributed mst algorithm with near-optimal time and message complexities. *Journal of the ACM (JACM)*, 67(2):1–15.

[14] Fredman, M. L. and Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615.

[15] Gabow, H. N., Galil, Z., Spencer, T., and Tarjan, R. E. (1986). Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122.

[16] Gabow, H. N., Galil, Z., and Spencer, T. H. (1989). Efficient implementation of graph algorithms using contraction. *Journal of the ACM (JACM)*, 36(3):540–572.

[17] Gallager, R. G., Humblet, P. A., and Spira, P. M. (1983). A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and systems (TOPLAS)*, 5(1):66–77.

[18] Hakimi, S. (1961). On trees of a graph and their generation. *Journal of the Franklin Institute*, 272(5):347–359.

[19] Jarník, V. (1930). O jistém problému minimálním. *Práca Moravské Prírodovedecké Spolecnosti*, 6:57–63.

[20] Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50.

[21] Lai, C., Rafa, T., and Nelson, D. E. (2009). Approximate minimum spanning tree clustering in high-dimensional space. *Intelligent Data Analysis*, 13(4):575–597.

[22] M., T. H. (1954). A note on the enumeration and listing of all possible trees in a connected linear graph. *Proceedings of the National Academy of Sciences of the United States of America*, 40(10):1004 – 1007.

[23] Mayeda, W. and Seshu, S. (1965). Generation of trees without duplications. *IEEE Transactions on Circuit Theory*, 12(2):181–185.

[24] Miller, B. N. and Ranum, D. L. (2011). *Problem solving with algorithms and data structures using python, Second Edition*. Franklin, Beedle & Associates Inc.

[25] Nešetřil, J., Milková, E., and Nešetřilová, H. (2001). Otakar boruvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete mathematics*, 233(1-3):3–36.

[26] Onete, C. E. and Onete, M. C. C. (2010). Enumerating all the spanning trees in an un-oriented graph - a novel approach. pages 1–5.

[27] Prim, R. C. (1957). Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401.

[28] Sollin, M. (1965). La trace de canalisation. *Programming, Games, and Transportation Networks*.

[29] Sörensen, K. and Janssens, G. (2005). An algorithm to generate all spanning trees of a graph in order of increasing cost. *Pesquisa Operacional*, 25.

[Stromberg] Stromberg, D. R. fibonacci-heap-mod. *PyPI*. Available at https://pypi.org/project/fibonacci-heap-mod/,retrievedon24.03.2021.

[Tran] Tran, Q. fibheap. *Github*. Available at https://github.com/quangntran/fibheap, retrieved on 14.04.2021.

[32] Vuillemin, J. (1978). A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315.

[33] Williams, J. (1964). Algorithm 232 heapsort. *Commun. ACM*, 7(6):347–348.

[34] Winter, P. (1986). An algorithm for the enumeration of spanning trees. *BIT Numerical Mathematics*, 26:44–62.

[35] Zahn, C. T. (1971). Graph-theoretical methods for detecting and describing gestalt clusters. *IEEE Transactions on computers*, 100(1):68–86.

[36] Zhong, C., Malinen, M., Miao, D., and Fränti, P. (2015). A fast minimum spanning tree algorithm based on k-means. *Information Sciences*, 295:1–17.

Appendix A

Results - Tables

N	Binary Kruskal	Binary Prim	Fibonacci Kruskal	Fibonacci Prim	Adj-list Kruskal	Adj-list Prim
250	0.02355	0.02654	0.03501	0.02618	0.00935	0.10753
500	0.05387	0.05831	0.07991	0.05869	0.02421	0.44237
750	0.08499	0.09283	0.12529	0.09316	0.03849	1.03091
1000	0.19091	0.22923	0.26887	0.21989	0.10671	5.72974
1500	0.19434	0.20732	0.27626	0.20779	0.10896	4.55411
2000	0.42920	0.50023	0.59292	0.48229	0.34698	26.16116
20000	6.98804	7.66472	9.49663	8.18335		
40000	13.91837	16.05284	18.81185	16.92903		
75000	30.56366	33.69274	42.08766	38.03917	461.62576	
100000	40.15111	45.85677	54.74341	49.92596		
150000	65.34107	73.86829	88.09180	81.22113		
200000	84.64632	95.10133	113.39486	105.41032		
240000	106.80914	118.25091	144.74644	132.06553		
400000	185.36042	205.69183	248.24449	228.43677		
700000	335.96479	370.62374	441.81312	414.75725		
1200000	610.41479	673.50231	799.03672	747.98730		

Table A.1 Average time in seconds for random graphs of size N.

N	Binary Kruskal	Binary Prim	Fibonacci Kruskal	Fibonacci Prim	Adj-list Kruskal	Adj-list Prim
20	0.00355	0.00810	0.00458	0.00495	0.00062	0.00464
50	0.02240	0.06302	0.02711	0.02865	0.00214	0.05836
100	0.09553	0.29144	0.10567	0.10868	0.00990	0.60735
150	0.22328	0.69187	0.22263	0.24755	0.02876	2.34071
200	0.41499	1.30926	0.41424	0.44846	0.06242	6.19924
500	1.38400	4.35231	1.45254	1.56964	0.49734	59.06709
1000	5.99047	19.78012	6.57812	7.63802	4.22853	
1600	16.08265	54.76658	16.31921	20.09977	18.40563	
2250	33.32162	114.95513	31.76665	39.13511	65.52805	
3500	83.39167	295.74490	76.47871	96.14724	266.75719	
5000	179.49160	633.94582	157.59210	203.36884	921.34578	
6000	257.56806	961.85444	230.46920	291.25491		
7000	358.44194		319.78655	407.13498		

Table A.2 Average time in seconds for complete graphs of size N.