Submitter Info:
Name: Sofija Velkovska
Email: sofijavelkovska27 [at] gmail [dot] com

Computer Science Extended Essay

Investigating the time efficiencies of Prim's and Kruskal's algorithms for minimum spanning trees

Research question: How does Kruskal's algorithm compare to Prim's algorithm in finding a minimum spanning tree in a graph in terms of running time across graphs with varying densities?

May 2024

Word count: 4000

Table of contents

1 Introduction

The minimum spanning tree is a fundamental concept in graph theory. It's directly present in transportation, computer, telecommunication, and other types of networks (Graham & Hell, 1985). Consider a set of terminals and connections between pairs of them with some cost attached to each connection (the distance between them, the cost of building a connection between them, etc.). A minimum spanning tree would be a subset of those connections such that all terminals are connected, and the total cost is minimum.

Other than these networks, minimum spanning trees find application in many places ranging from computer science – image segmentation (Felzenszwalb & Huttenlocher, 2004), natural sciences – studying gene clusters (Xu et al., 2002), to social sciences – describing financial markets (Mantegna, 1999).

Two classic algorithms, by Prim (1957) and Kruskal (1956), offer a solution to this problem. Both algorithms are similar in terms of time complexity; nonetheless, there is a common belief that Prim's algorithm is faster for dense graphs, while Kruskal's is for sparse graphs.

This paper investigates exactly that. By collecting experimental data on the running times of Prim's and Kruskal's algorithm on graphs with different densities, and analyzing and comparing this data, the research question "How does Kruskal's algorithm compare to Prim's algorithm in finding a minimum spanning tree in a graph in terms of running time across graphs with varying densities?" will be answered.

2 Theory

2.1 Graph terminology

A **graph** is a collection of **nodes** (vertices) and **edges**. A graph models pairwise relations (represented by the edges) between objects (the nodes). Henceforth, V and E will be used to represent the number of nodes and edges in a graph, respectively.

A **path** is a sequence of edges that joins a sequence of nodes in graph. The length of the path is the number of edges it includes.

A **cycle** is a path whose first and last node are the same.

In a weighted graph, each edge is assigned a numeric value, a **weight**. Weights of edges are usually seen as edge length. The length of a path in a weighted graph is the sum of the weights of the edges in it.

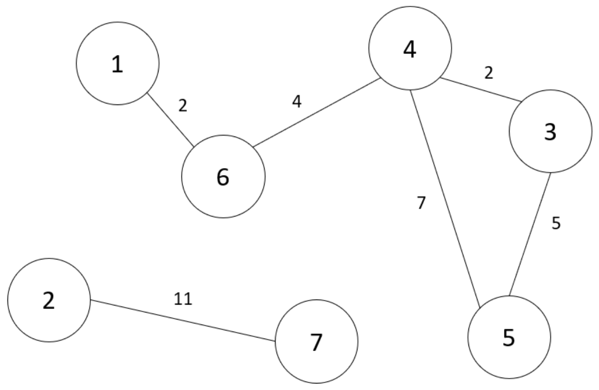


Figure 1: An example of a weighted graph. 1→6→4→3 is a path in this graph, and its weight is 8. 5→4→3 is a cycle.

A **simple graph** is a graph that does not have more than one edge between any two vertices and contains no loops (an edge that connects a node to itself).

Graphs can be **directed** or **undirected**. In a directed graph, edges have a direction, so, an edge "leaves" the first node and "enters" the second node. In an undirected graph, all edges are bidirectional.

In this paper, the term "graph" will indicate a simple, undirected, weighted graph.

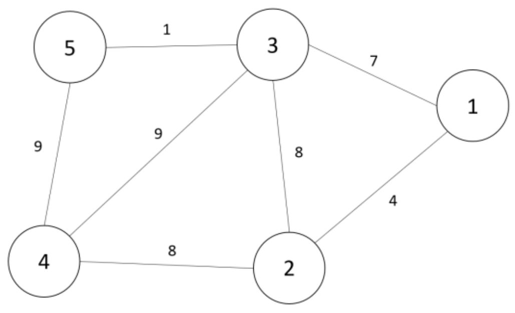
We say that a graph is **connected** if, for every pair of nodes, a path exists between them.

A **tree** is a graph in which every two nodes are connected by exactly one path. Equivalently, a tree is connected graph that contains no cycles. If a graph with V nodes is a tree, it will have $V - 1$ edges.

A **spanning tree** of a connected graph is a tree that contains all of the nodes of the graph and some of its edges (consequently, there is a path between any two nodes).

A **minimum spanning tree** (MST) is a spanning tree that has the minimum possible sum of edge weights.

While a graph in which all edge weights are distinct only has one minimum spanning tree (Borůvka, 1926), a graph that includes pairs of edges with equal weight could have an enormous number of MSTs (Gabow & Myers, 1978). Usually, the objective of programs is to find one of these MSTs, as finding them all wouldn't be very useful, or is downright impossible. Accordingly, Prim's and Kruskal's algorithms focus on finding one, not specifically chosen, MST.



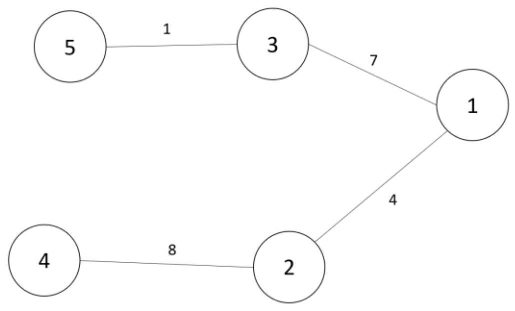


Figure 2: A graph and its minimum spanning tree

2.2 Prim's algorithm

Prim's algorithm begins with the initialization of a tree consisting of a single, arbitrarily chosen, node. The algorithm then builds the tree one node at a time. In each step, the edge with the minimum weight that connects the tree to a node that hasn't been already added, is added to the tree. When all nodes have been added, the constructed tree will be an MST of the graph.

Consider the following graph. The node 9 is the starting node and it's added to the tree. Any other node could have been chosen as a starting node.

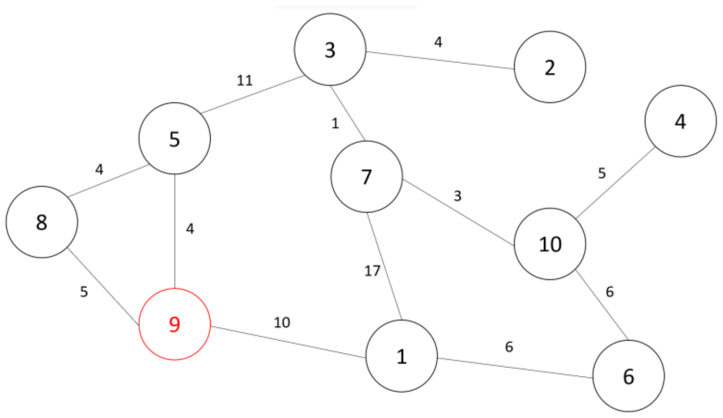


Figure 3: Finding an MST of a graph with Prim's algorithm 1

The edges 9-1, 9-5 and 9-8 are edges that can be considered in the next step, since they would add a node to the current tree. Out of them, 9-5 has the minimum weight, so it's the next edge to be added.

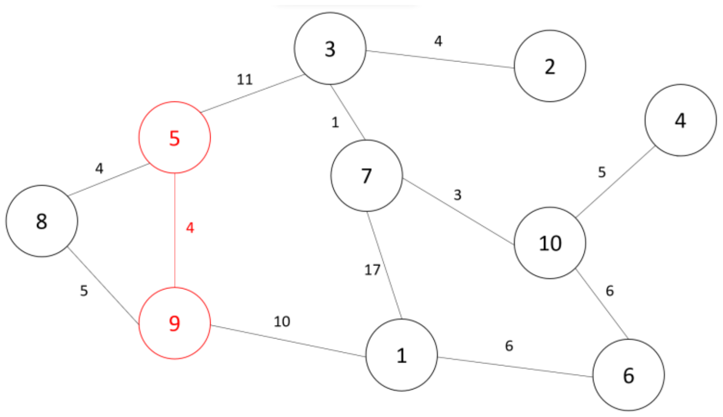


Figure 4: Finding an MST of a graph with Prim's algorithm 2

Now, 5-3, 5-8, 9-1 and 9-8 are possible next edges. 5-8 is the one with the minimum weight so it's added to the tree.

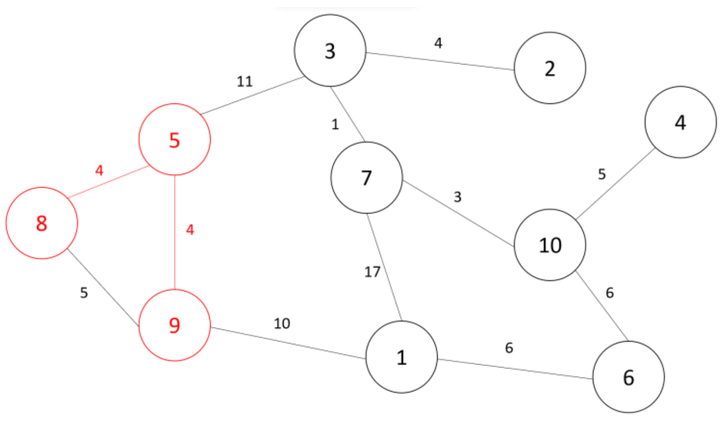


Figure 5: Finding an MST of a graph with Prim's algorithm 3

Similarly, 9-1 is added.

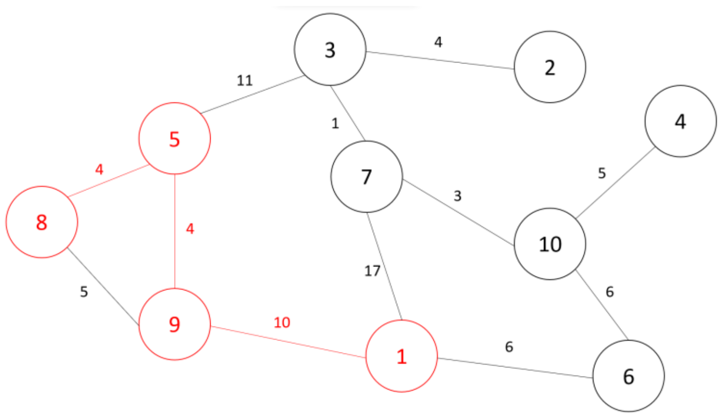


Figure 6: Finding an MST of a graph with Prim's algorithm 4

This step is repeated until all the nodes are part of the tree. The result is an MST of the initial graph.

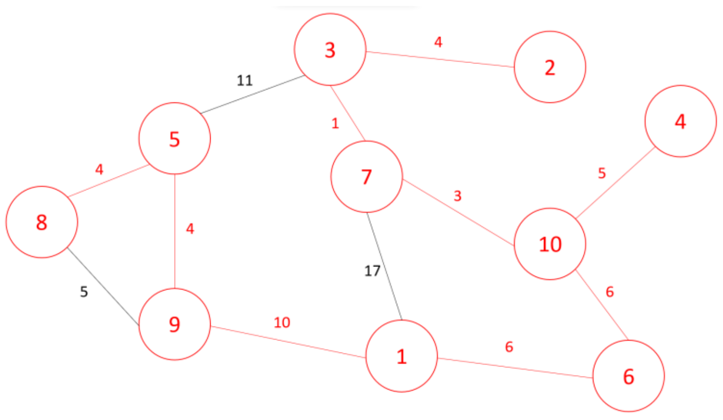


Figure 7: Finding an MST of a graph with Prim's algorithm 5

In each step, Prim's algorithm should be able to identify the minimum weight edge that would add a new node to the tree. Checking all of the appropriate edges to find the one with the smallest weight takes linear time, and since this needs to be done for every addition of a new node to the tree it would result in a quadratic time complexity. A priority queue is a data structure that maintains a set of elements in which each element has a value assigned to it (a key), and enables access to the element with the highest priority based on the keys of the elements (usually, the key with the highest/lowest value) (Kleinberg & Tardos, 2005). Due to its $O(1)$ retrieval of the smallest element and $O(\log n)$ insertion and removal of elements (Kleinberg & Tardos, 2005), using a priority queue would significantly improve the time complexity of Prim's algorithm. The implementation of Prim's algorithm in C++ can be found in Appendix I.

During the execution of the algorithm, every edge will be processed once, and the new node it connects the tree to needs to be added to the priority queue, or if its already in the priority queue, the value of its key in the priority queue needs to be updated, if the weight of this edge is smaller than the current key – both of these operations have a $O(\log V)$ time complexity. They will be executed E times, leading to a total of $O(E \log V)$. Additionally, every node will be removed from the priority queue exactly once, which will take $O(V \log V)$ time. Therefore, the total time complexity is $O(E \log V + V \log V)$. $E \log V$ is the leading term here, since E is usually significantly bigger than V (V can be at most $E + 1$), making time complexity of Prim's algorithm $O(E \log V)$.

2.3 Kruskal's algorithm

In Kruskal's algorithm, the first step is to sort the edges by weight. The sorted edges are looped through, starting with the smallest weight, and an edge is added to the MST if it joins two different components, i.e. if it doesn't create a loop in the graph with the already chosen edges. Initially, every node is in its own component. After all the edges are processed, the constructed graph is in fact an MST of the initial graph.

Consider the following graph. At first, all the nodes are in separate components.

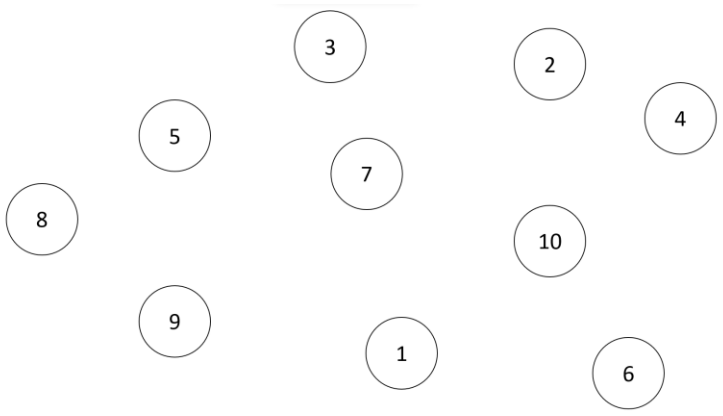


Figure 8: Finding an MST of a graph with Kruskal's algorithm 1

The edge 3-7 is the first one to be considered, as it's, with length 1, the shortest among all edges. 3 and 7 are in different components, so the edge is added. Now, 3 and 7 are in the same component.

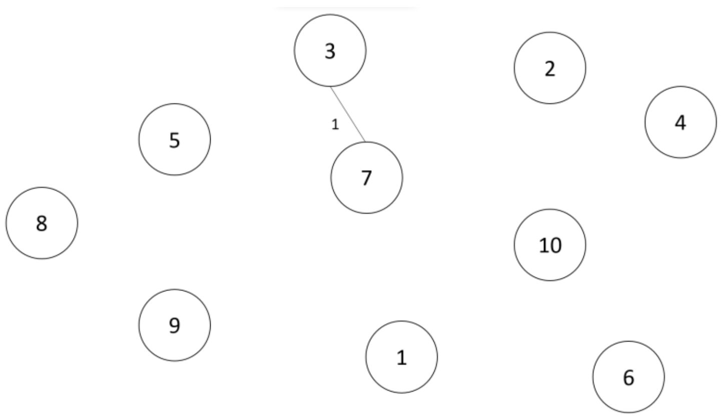


Figure 9: Finding an MST of a graph with Kruskal's algorithm 2

The edges 7-10, 2-3, 5-8 and 5-9 are considered next, in that order. They are all added since they connect two different components. It should be noted that when there are multiple edges of the same weight, it doesn't matter in what order they will be processed. After these operations, the graph consists of two components.

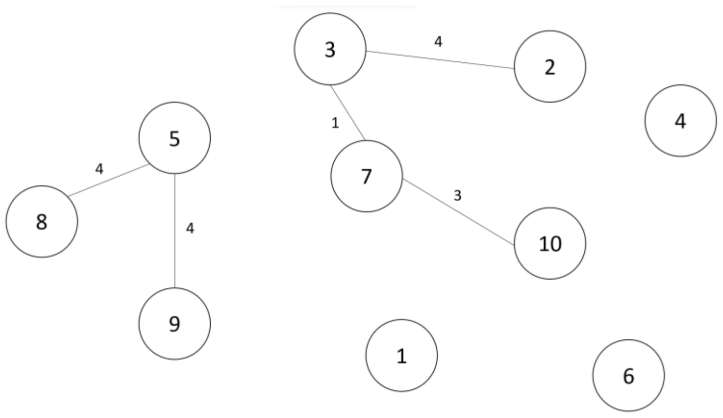


Figure 10: Finding an MST of a graph with Kruskal's algorithm 3

Next, the edge 8-9 won't be added since the nodes 8 and 9 are already in the same component. In other words, adding this edge will create a loop (5-8-9).

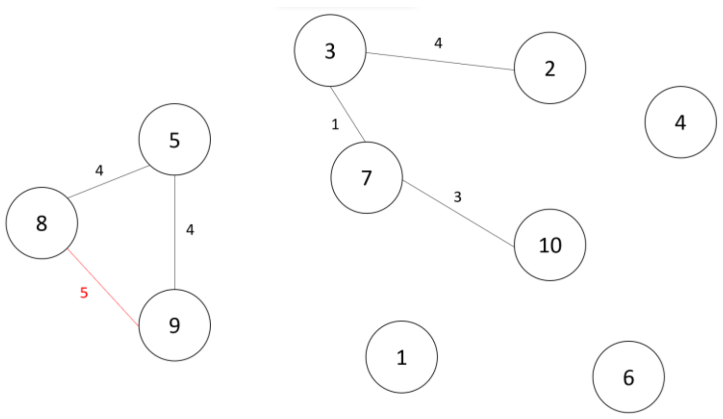


Figure 11: Finding an MST of a graph with Kruskal's algorithm 4

The next four edges by weight, 5-10, 1-6, 6-10 and 1-9, are all added to the MST.

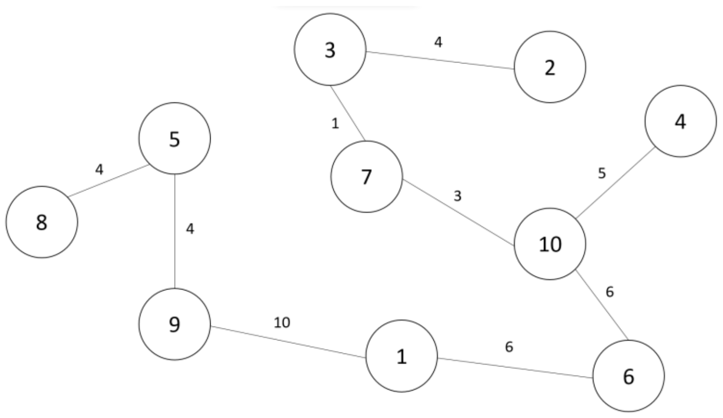


Figure 12: Finding an MST of a graph with Kruskal's algorithm 5

Then, the edge 3-5 won't be added since the nodes 3 and 5 are already in the same component.

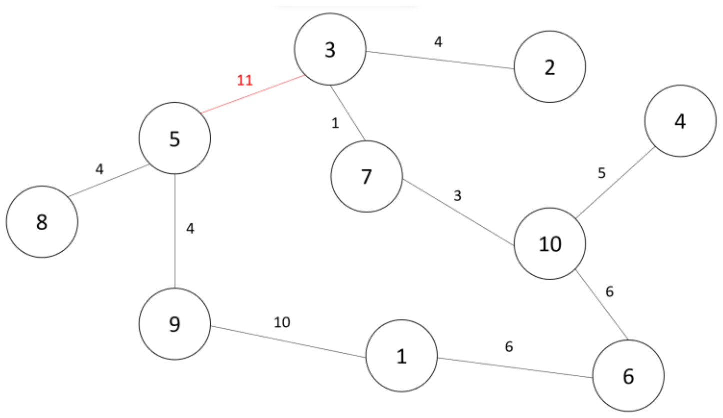


Figure 13: Finding an MST of a graph with Kruskal's algorithm 6

Similarly, the edge between 1 and 7 also won't be added. 1-7 was the last egde to be processed because it has the biggest weight. The algorithm comes to an end, with an MST formed by all the edges that were added.

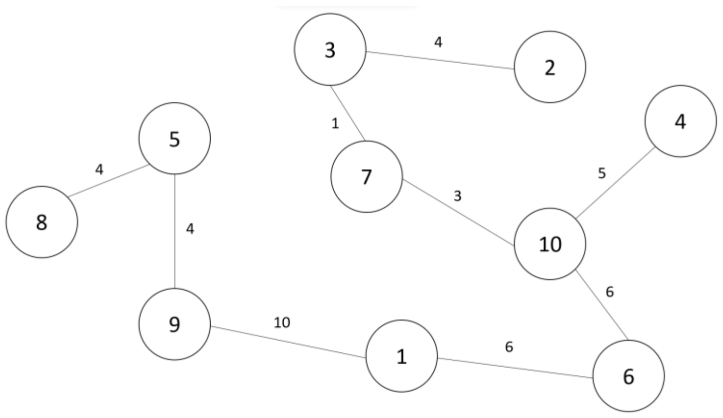


Figure 14: Finding an MST of a graph with Kruskal's algorithm 7

In one execution of the algorithm, a loop will go through all of the edges, and for each edge it needs to be found out if the nodes on its end belong to the same component. One way to check this is to run a graph traversal and check if a path exists between the two nodes. Graph traversal algorithms, like BFS and DFS, have a linear time complexity, and a graph traversal should be made for every edge, resulting in a quadratic time complexity. Thus, a more efficient way for checking if two nodes belong to the same component is needed for big input data. This can be done with the help of a union-find data structure, making it possible to check whether two nodes belong to the same component in $O(\log n)$ time complexity and uniting two components in $O(1)$ (Cormen et al., 2001). The implementation of Kruskal's algorithm with a union-find structure in can be found in Appendix II.

The sorting of the edges by weight that needs to be done before the algorithm starts building the MST can be done in $O(E \log E)$ time complexity. With the use of a union-find structure, the construction of the minimum spanning has a time complexity of $O(E \log V)$. Thus, the total time complexity is $O(E \log E +$

$E\log V$). However, the maximum value that E could obtain is $\frac{V(V-1)}{2} \approx V^2$, in a graph where every pair of nodes is directly connected. So, the time complexity can be described as $O(E\log V^2 + E\log V)$. By the properties of logarithms, $E\log V^2$ is equal to $2E\log V$. We have $2E\log V + E\log V = 3E\log V$, and since 3 is a constant, it shouldn't be taken into account. Finally, the time complexity of Kruskal's algorithm is $O(E\log V)$.

2.4 Time complexity analysis

The time complexity of Prim's and Kruskal's algorithms was discussed above – both algorithms have the same time complexity of $O(E\log V)$. Thus, it's expected that the algorithms will have similar running times on the same input. Nevertheless, the Big O notation doesn't give an exact running time for a particular algorithm, it's just an estimation of the number of elementary operations performed. Due to this, the running time of Prim's algorithm will differ from the running time of Kruskal's algorithm, with one of them being faster than the other in some cases, while the other is faster in other cases.

One trait of graphs that can be linked to the time performance of MST algorithms is the graph's density. The density of a graph can be defined as the number of its edges over the total number of possible edges in that graph. Let d denote the density of a graph; we have $d = \frac{E}{\frac{V(V-1)}{2}} = \frac{2E}{V(V-1)}$ (Coleman & Moré, 1983). Even though the terms dense graph and sparse graph are not strictly defined, it can be assumed that a graph is dense if the number of its edges is about quadratic in its number of nodes, and a sparse graph is a graph whose number of edges is about linear in its number of nodes (Diestel, 2018). The number of edges in dense graphs is close to them maximal number of edges ($d \approx 1$), while in sparse graphs it's close to the minimal number of edges ($d \approx \frac{1}{V}$).

The density of a graph is dependent both on the number of nodes and the number of edges it consists of. The time complexity of Prim's and Kruskal's algorithms can be represented as $\frac{V(V-1)}{2} d\log V$ using the number of nodes and the density of the graph. For a fixed number of nodes, $\frac{V(V-1)}{2}\log V$ is constant over all values of d. So, the expression is solely dependent on d. Specifically, since the time complexity can be represented as $k \cdot d$, where k is a constant factor, the running times of the algorithms will linearly increase as d increases.

Concerning the effect of density on the running time when the number of edges is fixed, it is expected that the formula $E\log\sqrt{\frac{2E}{d}}$ will describe the time complexity of the algorithms (the derivation of this formula can be found in Appendix III). Since d acts as a denominator in this expression, the $E\log\sqrt{\frac{2E}{d}}$ term, and therefore the running time, will logarithmically decrease as d increases.

As previously mentioned, the total number of operations done by Prim's algorithm can be more accurately estimated by $E\log V + V\log V$, in contrast to Kruskal's $E\log E + E\log V$. Consider the following two cases:

1. A dense graph. By definition, $E \approx V^2$. We have,

Prim's algorithm:
$E \log V + V \log V$
$\approx V^2 \log V + V \log V$
$= V \log V \, (V + 1)$

Kruskal's algorithm:
$E \log E + E \log V$
$\approx V^2 \log V^2 + V^2 \log V$
$= 2V^2 \log V + V^2 \log V$
$= 3V^2 \log V$
$= V \log V \, (3V)$

Thus, Prim's algorithm will perform less elementary operations than Kruskal's algorithm when finding the MST of a dense graph, and therefore, have a faster running time. Since $\frac{V \log V (V+1)}{V \log V (3V)} = \frac{V+1}{3V} \approx \frac{V}{3V} = \frac{1}{3}$, Prim's running time is expected to be three times faster than Kruskal's.

2. A sparse graph. By definition, $E \approx V$. We have,

Prim's algorithm:
$E \log V + V \log V$
$\approx V \log V + V \log V$
$= 2V \log V$

Kruskal's algorithm:
$E \log E + E \log V$
$\approx V \log V + V \log V$
$= 2V \log V$

So, Prim's and Kruskal's algorithm will perform approximately the same number of operations when finding the MST of a sparse graph.

3 Hypothesis

Using the presented theoretical background, a hypothesis can be formed to answer the research question. For graphs with a fixed number of nodes and increasing density, the running time of both Prim's and Kruskal's algorithms will linearly increase. For graphs with a fixed number of edges and increasing density, the running time will decrease with a logarithmic rate. Furthermore, Prim's algorithm will have a better running time for dense graphs, while both algorithms will have similar running times for sparse graphs.

4 Methodology

4.1 Independent variable

The independent variable in this investigation will be the density of the graph that will act as an input for the MST algorithms. The same data will be subjected to both Prim's and Kruskal's algorithm.

In Experiment 1, the number of nodes (1000 nodes) will be constant while the number of edges will be changed to get a certain density. In particular, the densities from 0.05 to 1.00 in 0.05 increments will be examined. These values were chosen because they cover the whole range of possible densities and there are enough levels of the independent variable for the relationship between the independent and dependent variable to be established.

In Experiment 2, the number of edges will be constant (500000 edges), while the number of nodes will be changed. This experiment will specifically focus on graphs with extremely small densities, as the hypothesis suggest they might be especially interesting for the comparison between Prim's and Kruskal's time complexity. Graphs with densities ranging from 0.00005 to 0.001 with increments of 0.00005 will be used.

For each case in both experiments, five different graphs with the given density will be tested. The graphs used are randomly generated with a program written in C++, which can be found in Appendix IV.

4.2 Dependent variable

The dependent variable of the experiment is the time taken for the execution of the algorithm, whether Prim's or Kruskal's. The runtime will be measured using high_resolution_clock from C++ STL's chrono header since it provides high precision (cppreference.com, n.d.). Three trials are conducted per graph, with the arithmetic mean of their times being taken (the procedure and the test program are available in Appendix V and VI, respectively). The measured times will be represented in seconds with up to 6 decimal places. Even though the difference between running times is expected to be as small as microseconds in some cases, the second will be used as it's more comprehensible.

4.3 Controlled variables

Variable	**Description**	**Importance**
Number of nodes/edges	The number of nodes in Experiment 1 and the number of edges in Experiment 2 are kept constant (at 1000 and 500000, respectively).	This eliminates the possibility of the number of nodes/edges int the graph (not just its density) being the cause of change in the running time of the algorithm.
Program used	All trials are done using the same program.	Using even a slightly different code will obviously have an effect on the running time since different number of lines of code and different statements have different running times.
Edge weights	The edges in all of the generated graphs had a natural number between 1 and 1000, inclusive, assigned to them as their weight.	Limiting the maximum weight to a relatively small number prevents integer overflow and, hence, uncontrolled behavior of the program. Also, only integers are used, instead of decimal numbers, to avoid the effects of floating-point arithmetic's lack of exactness.
How the graphs are spread out	Since the graphs were randomly generated, they were spread out in various ways, i.e. the graphs had different	The algorithms might perform better on certain types of graphs, which might lead to incorrect assumptions about

	distributions of node degrees (the number of other nodes a node is connected to (Diestel, 2018)) and different diameters (the greatest shortest path between any two nodes (Diestel, 2018)).	the quality of their general performance on any type of graph.
Computer and operating system	Computer: Lenovo IdeaPad Flex 5 14ARE05 Processor: AMD Ryzen 5 4500U with Radeon Graphics 2.38 GHz Memory: 8,00 GB RAM Operating system: Windows 11 Home	All trials were run on the same computer and operating system, as hardware and the software have a major effect on the running time of a program.

Table 1: List of controlled variables

5 Experimental data

5.1 Experiment 1

In Experiment 1, the number of nodes in the test graphs was fixed to 1000. The number of edges was changed to obtain the desired density. Raw data is available in Appendix VII. The mean of the running times for the five different graphs per level of the independent variable for which the algorithms were run was calculated. The results can be observed in Table 2.

Density	Number of edges	Prim's running time	Kruskal's running time
0.05	24975	0.010039	0.010578
0.10	49950	0.015454	0.02219
0.15	74925	0.019895	0.037365
0.20	99900	0.026381	0.045986
0.25	124875	0.030275	0.053145
0.30	149850	0.029871	0.062808
0.35	174825	0.031759	0.074418
0.40	199800	0.035208	0.089974
0.45	224775	0.036766	0.093857
0.50	249750	0.039869	0.114746
0.55	274725	0.051094	0.122075
0.60	299700	0.054528	0.133555
0.65	324675	0.056439	0.149329
0.70	349650	0.055221	0.149402
0.75	374625	0.058067	0.161185
0.80	399600	0.060901	0.174256
0.85	424575	0.065921	0.188293

0.90	449550	0.067257	0.198209
0.95	474525	0.065314	0.203301
1.00	499500	0.067726	0.221975

Table 2: Prim's and Kruskal's average running times in Experiment 1

5.2 Experiment 2

In Experiment 2, the number of edges in the test graphs was fixed to 1000. The number of nodes was changed to obtain the desired density. Raw data is available in Appendix IX. The mean of the running times for the five different graphs per level of the independent variable for which the algorithms were run was calculated. The results can be observed in Table 3.

Density	Number of nodes	Prim's running time	Kruskal's running time
0.00005	141422	0.833602	0.264096
0.00010	100001	0.715338	0.263843
0.00015	81651	0.652133	0.248409
0.00020	70712	0.582783	0.282104
0.00025	63247	0.521185	0.268772
0.00030	57736	0.476734	0.260165
0.00035	53453	0.467139	0.278673
0.00040	50001	0.402542	0.232291
0.00045	47141	0.448252	0.239502
0.00050	44722	0.390707	0.232663
0.00055	42641	0.395231	0.244519
0.00060	40826	0.385441	0.260613
0.00065	39224	0.453120	0.264859
0.00070	37797	0.372805	0.266481
0.00075	36516	0.339085	0.259841
0.00080	35356	0.341841	0.246979
0.00085	34301	0.361435	0.236130
0.00090	33334	0.328125	0.234965
0.00095	32445	0.358975	0.240204
0.00100	31624	0.318995	0.256451

Table 3: Prim's and Kruskal's average running times in Experiment 2

6 Data analysis and discussion

6.1 Experiment 1

It can be inferred that Prim's algorithm's running time is generally smaller than Kruskal's algorithm's. To test whether this hypothesis that Prim's algorithm had lower running time than Kruskal's algorithm is statistically significant, a paired samples *t*-test was conducted on the results from each test graph for each level of independent variable (Appendix VIII). An extremely statistically significant result with $p<0.001$ was obtained on all levels except the first one where the density of the graphs is 0.05. So, Prim's algorithm was faster for graphs with 1000 nodes and a density between 0.10 and 1.00 than Kruskal's algorithm. For graphs with density of 0.05, both algorithms performed similarly. It's possible that no difference was observed for this density because of the relatively small input size (1000 nodes and 24975 edges), so the timer wasn't able to catch the small difference in time. Furthermore, constant factors might have played a deciding role here – time complexity analysis focuses on asymptotic behavior, so it doesn't predict the behavior of small inputs whose running time can be strongly influenced by constants.

Additionally, the ratio of the average of Prim's over Kruskal's running times was calculated (Table 4).

Density	Ratio of average running times
0.05	1.054
0.10	1.436
0.15	1.878
0.20	1.743
0.25	1.755
0.30	2.103
0.35	2.343
0.40	2.556
0.45	2.553
0.50	2.878
0.55	2.389
0.60	2.449
0.65	2.646
0.70	2.706
0.75	2.776
0.80	2.861
0.85	2.856
0.90	2.947
0.95	3.113
1.00	3.278

Table 4: The ratio of the average of Prim's over Kruskal's running times in Experiment 1

It can be noticed how the values of the ratio get closer to 3 as the density of the graphs gets closer to 1.00, i.e. the graphs get denser. This is in line with the analytically derived prediction that Prim's algorithm will be three times faster than Kruskal's algorithm for dense graphs.

The average running times of Prim's and Kruskal's algorithm for the different densities plotted on the same set of axes can be observed in Figure 15.

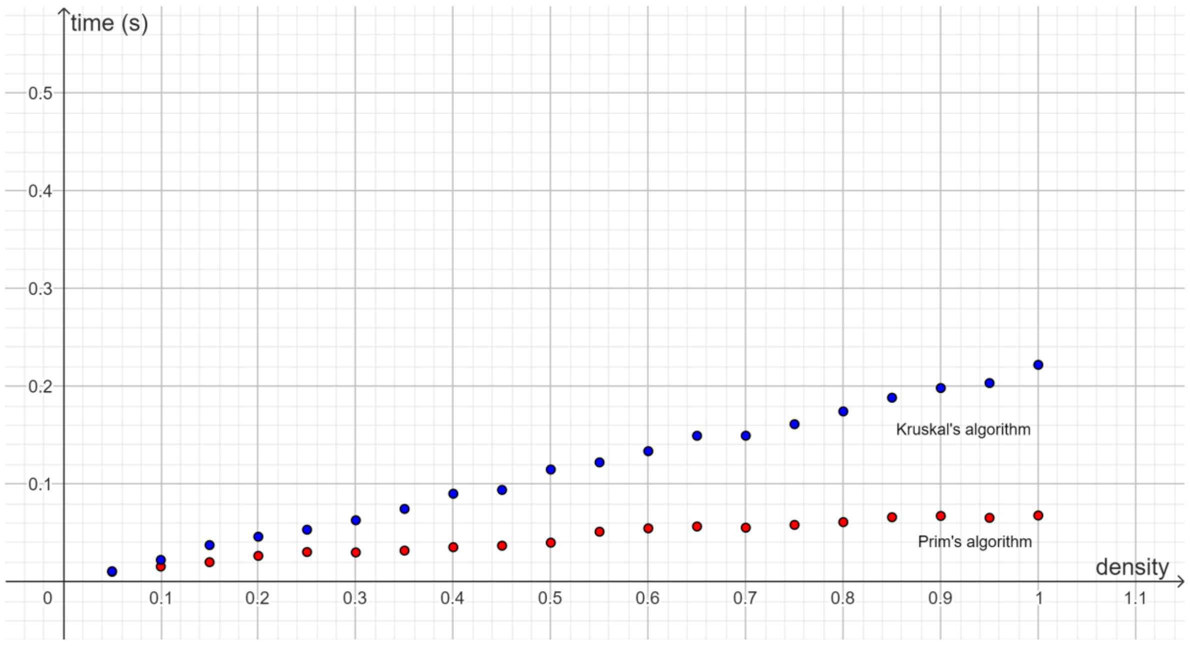


Figure 15: A graph showing the running times of Prim's and Kruskal's algorithms for graphs with 1000 nodes and various densities

Functions to describe the relationships between the density of the graphs and the running times of both algorithms were developed. The functions were of the form $f(d) = \frac{V(V-1)}{2} d \log_2 V$. In this case, $V = 1000$, therefore, we get $f(d) = 499500d \log_2 1000$. To account for constants that are not included in time complexity analysis, $f(d)$ will have the form $a \cdot 499500d \log_2 1000 + b$, where a and b are constants. Finally, notice that this give us the total number of elementary operations that will be performed. To get the running time of the algorithms the whole expression will be divided by 10^8. This number was chosen as it's approximately number of elementary operations that can be executed in a second on the computer that the experiment was conducted. The final function is $f(d) = \frac{a \cdot 499500d \log_2 1000 + b}{10^8}$.

For Prim's algorithm, the $f_1(d) = \frac{1.24 \cdot 499500d \log_2 1000 + 1158068.63}{10^8}$, was developed as a fit for the data points, with the help of a mathematics software. An R squared value of $R^2 = 0.97$ was calculated, suggesting that this function is a strong model for the data.

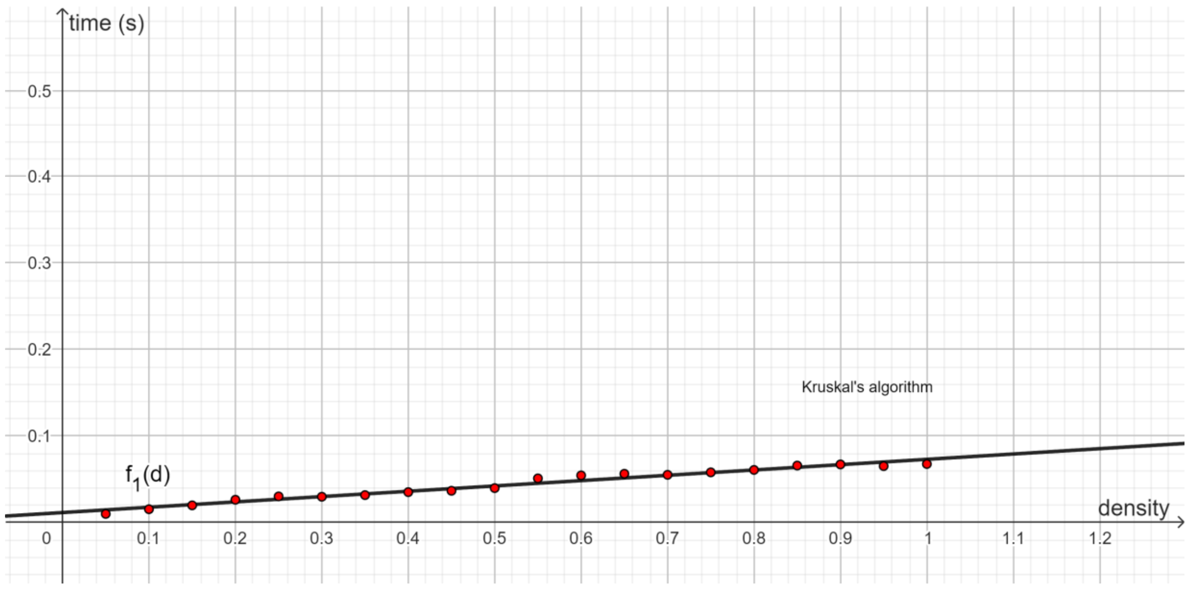


Figure 16: A graph showing Prim's running time for graphs with 1000 nodes and various densities, and f1

Similarly, the function $f_2(d) = \frac{4.39 \cdot 499500d \log_2 1000 + 67087.5.}{10^8}$ was developed for the data from Kruskal's algorithm. With $R^2 = 1.00$, it's a very good fit for the data.

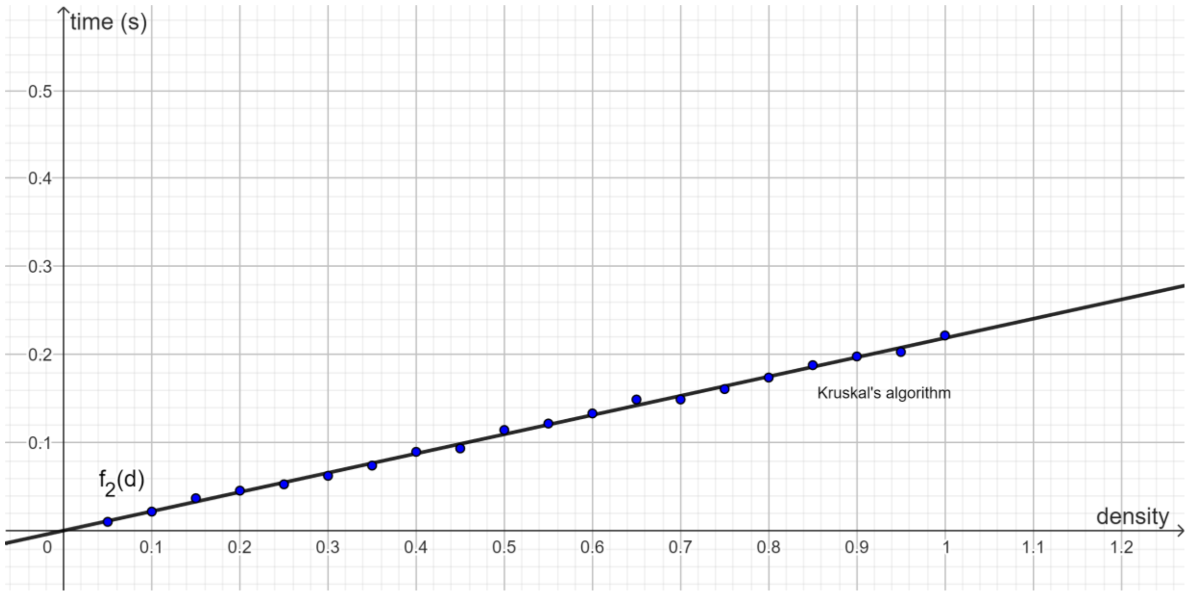


Figure 17: A graph showing Kruskal's running time for graphs with 1000 nodes and various densities, and f_2

In this experiment, Prim's algorithm had a better running time than Kruskal's for graphs with density from 0.10 to 1.00 in 0.05 increments. For graphs with density of 0.05, no algorithm performed significantly better. Also, a strong linear relationship was established between the running time and the density of the graph. However, only graphs with 1000 nodes were considered in this experiment. These findings can't be accepted as universal rules for all graphs, as they might not be true for graphs with different number of nodes.

6.2 Experiment 2

From the data in Table 3, it can be assumed that Kruskal's algorithm had lower running times than Prim's for these graphs. A *t*-test was conducted, similarly to Experiment 1 (Appendix X). With *p* value less than 0.05 on all levels, the hypothesis that Kruskal's algorithm has significantly lower running times was accepted.

In this experiment, the densities of the graphs were extremely small. In the theoretical part of this paper, it was hypothesized that Prim's and Kruskal's algorithms will have similar running times for sparse graphs, but the experimental results showed otherwise. This could have happened due to various unaccounted for reasons, like the difference between the time efficiency of the different data structures used.

The average running times of Prim's and Kruskal's algorithm for the different densities plotted on the same set of axes can be observed in Figure 18. It was hypothesized that the relationship between the density and the running time of the algorithms can be described by the formula $E \log \sqrt{\frac{2E}{d}}$. However, this wasn't the case, as it can also concluded from the visual representation of the data.

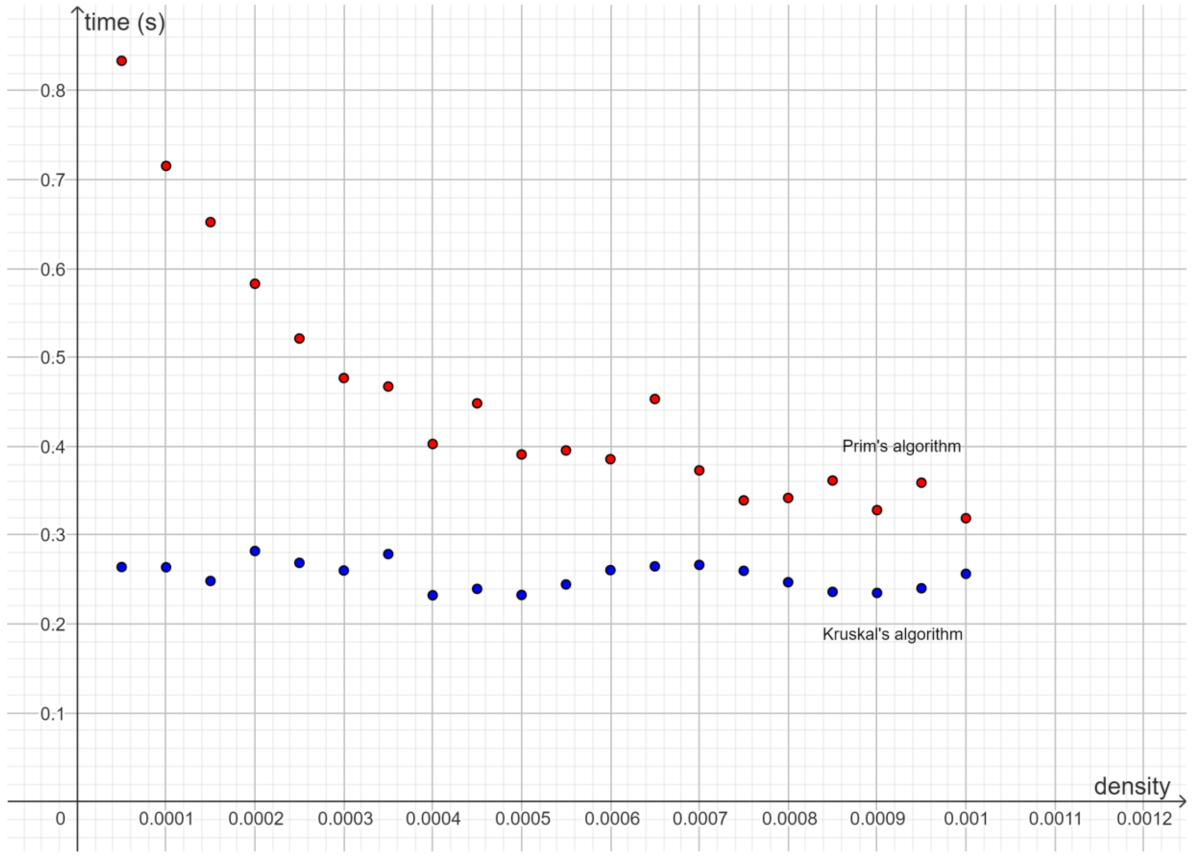


Figure 18: A graph showing the running times of Prim's and Kruskal's algorithms for graphs with 500000 edges and various densities

The running time of Prim's algorithm decreases at a rapid rate as the density of the graph increases. Moreover, for very small densities, Prim's running time is much bigger than Kruskal's even though that difference gets smaller as the density gets larger. With decreasing density, the number of nodes in the graph increases. Recall that a $V \log V$ term is present in the expression for the number of operations in Prim's algorithm. So, it's possible that the number of nodes has a big effect on the running time of Prim's algorithm.

The way the data points for Kruskal's algorithm are spread out on the graph vaguely resembles a horizontal line. This suggests a constant function for the relationship between graph density and the running time for Kruskal's algorithm. This indicates that density doesn't have an effect on the running time, i.e. the running time is only dependent on the number of edges, for graphs with very small density and 500000 edges. $E \log E$ was another way to write down the time complexity of Kruskal's algorithm – it's possible that for graphs with many edges this term totally takes over the time complexity and, hence, E determines the running time by itself, unaffected by V and d.

6.3 Limitations

It was attempted to make the research as rigorous and controlled as possible; however, there were still some flaws in the design and execution of the experiment. For example, the graphs used in the experiment, that were generated by a random graph generating program, weren't completely random. Even though they program aims for randomness and succeeds in this to a sufficient extent, the way the algorithm works makes the nodes that were first added more likely to have a large degree (number of neighboring nodes). In a truly random graph, each node has the same probability of having a certain degree. While less biased algorithms for generating graphs exist, they weren't considered as they are very complicated.

Moreover, C++'s rand() function that was used for generating the graphs doesn't provide actual random number but pseudo-random numbers (cppreference.com, n.d.). This could have had an uncontrolled effect on the results.

Another limitation is the handling of the edge sorting part of Kruskal's algorithm. This is a crucial part of the algorithm and has a big role on the running time, so, choosing the appropriate sorting method is important. An $O(n \log n)$ sorting algorithm was used, even though other sorting algorithms might have been more suitable. For example, counting sort could have been a better choice with its $O(n)$ time complexity, since all weights were integers from 1 to 1000.

7 Conclusion

This paper aimed to answer the research question "How does Kruskal's algorithm compare to Prim's algorithm in finding a minimum spanning tree in a graph in terms of running time across graphs with varying densities?". Even though both algorithms have the same time complexity of $O(E \log V)$, two experiments were conducted to investigate their running times, the first focusing on the effect of density on the running times for graphs with fixed number of nodes, and the second, for fixed number of edges.

The results from Experiment 1 indicate a positive linear relationship between graph density and running time of both algorithms. For graphs with densities from 0.10 to 1.00 and 1000 nodes, Prim's algorithm was faster. Furthermore, the results suggest that Prim's algorithm is three times faster than Kruskal's for very dense graphs. Taking into consideration Experiment 2, it appears that Kruskal's algorithm has significantly lower running time for very sparse graphs than Prim's algorithm. Moreover, the running time of Prim's algorithm rapidly decreases as graph density increase, while Kruskal's appears to stay constant.

Bibliography

Borůvka, Otakar (1926). O jistém problému minimálním [About a certain minimal problem]. *Práce Moravské přírodovědecké společnost*, 3, 37–58

Coleman, T. F., & Moré, J. J. (1983). Estimation of sparse Jacobian matrices and graph coloring blems. *SIAM Journal on Numerical Analysis*, *20*(1), 187–209. https://doi.org/10.1137/0720013

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). *Introduction to algorithms*. MIT Press.

cppreference.com (n.d.). *std::chrono::high_resolution_clock*. https://en.cppreference.com/w/cpp/chrono/high_resolution_clock

cppreference.com. (n.d.). *rand*. https://en.cppreference.com/w/c/numeric/random/rand

Diestel, R. (2017). *Graph theory* (5th ed.). Springer.

Felzenszwalb, P. F., & Huttenlocher, D. P. (2004). Efficient Graph-Based image segmentation. *International Journal of Computer Vision*, *59*(2), 167–181. https://doi.org/10.1023/b:visi.0000022288.19776.77

Gabow, H. N., & Myers, E. W. (1978). Finding all spanning trees of directed and undirected graphs. *SIAM Journal on Computing*, *7*(3), 280–287. https://doi.org/10.1137/0207024

Graham, R. L., & Hell, P. (1985). On the History of the Minimum Spanning Tree Problem. *IEEE Annals of the History of Computing*, *7*(1), 43–57. https://doi.org/10.1109/mahc.1985.10011

Kleinberg, J., & Tardos, E. (2005). *Algorithm Design*. Pearson.

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. https://doi.org/10.1090/s0002-9939-1956-0078686-7

Mantegna, R. N. (1999). Information and hierarchical structure in financial markets. *Computer Physics Communications*, *121–122*, 153–156. https://doi.org/10.1016/s0010-4655(99)00302-1

Prim, R. C. (1957). Shortest connection networks and some generalizations. *Bell System Technical Journal*, *36*(6), 1389–1401. https://doi.org/10.1002/j.1538-7305.1957.tb01515.x

Xu, Y., Olman, V., & Xu, D. (2002). Clustering gene expression data using a graph-theoretic approach: an application of minimum spanning trees. *Bioinformatics*, *18*(4), 536–545. https://doi.org/10.1093/bioinformatics/18.4.536

Appendices

Appendix I: Code for Prim's algorithm

```
const int INF=1e9;

vector<pair<int, int> > prim(int n, int m, vector<pair<pair<int, int>, int> > &graph)
{
    vector<vector<pair<int, int> > > adj(n);
    for (auto edge : graph)
    {
        int from=edge.first.first;
        int to=edge.first.second;
        int weight=edge.second;
        adj[from].push_back({to, weight});
        adj[to].push_back({from, weight});
    }

    vector<pair<int, int> > minSpanningTree;
    set<pair<int, int> > q;
    vector<bool> visited(n, false);
    vector<pair<int, int> > minEdge(n, {INF, -1});

    minEdge[0]={0, -1};
    q.insert({0, 0});

    for (int i=0; i<n; i++)
    {
        int currentNode=(*q.begin()).second;
        q.erase(q.begin());
        visited[currentNode]=true;
        if (minEdge[currentNode].second!=-1)
            minSpanningTree.push_back({currentNode, minEdge[currentNode].second});
        for (auto edge : adj[currentNode])
        {
            int nextNode=edge.first;
            int weight=edge.second;
            if (!visited[nextNode] && weight<minEdge[nextNode].first)
            {
                q.erase({minEdge[nextNode].first, nextNode});
                minEdge[nextNode]={weight, currentNode};
                q.insert({weight, nextNode});
            }
        }
    }

    return minSpanningTree;
}
```

Appendix II: Code for Kruskal's algorithm

```
vector<int> parent, componentSize;
```

```
int findComponent(int v)
{
    if (v==parent[v])
        return v;

    return parent[v]=findComponent(parent[v]);
}

void uniteComponents(int a, int b)
{
    a=findComponent(a);
    b=findComponent(b);
    if (a==b)
        return;
    if (componentSize[a]<componentSize[b])
        swap(a, b);
    componentSize[a]=componentSize[a]+componentSize[b];
    parent[b]=a;

    return;
}

vector<pair<int, int> > kruskal(int n, int m, vector<pair<pair<int, int>, int> >
&graph)
{
    vector<pair<int, pair<int, int> > > sortedEdges(graph.size());

    int i=0;
    for (auto edge : graph)
    {
        int from=edge.first.first;
        int to=edge.first.second;
        int weight=edge.second;
        sortedEdges[i]={weight, {from, to}};
        i=i+1;
    }

    sort(sortedEdges.begin(), sortedEdges.end());

    parent.resize(n);
    componentSize.resize(n);
    for (int i=0; i<n; i++)
    {
        parent[i]=i;
        componentSize[i]=1;
    }

    vector<pair<int, int> > minSpanningTree;

    for (auto edge : sortedEdges)
    {
        int from=edge.second.first;
        int to=edge.second.second;
        int weight=edge.first;
        if (findComponent(from)!=findComponent(to))
        {
            minSpanningTree.push_back({from, to});
            uniteComponents(from, to);
        }
        if (minSpanningTree.size()==n-1)
            break;
    }

    return minSpanningTree;
}
```

Appendix III: Derivation of formula for time complexity

$d = \frac{2E}{V(V-1)}$ (by definition)

$\therefore V(V-1) = \frac{2E}{d}$

$\therefore V^2 - V = \frac{2E}{d}$

$\therefore V = \sqrt{\frac{2E}{d}}$ (for simplicity, $V^2 - V$ is approximated to V^2)

$\therefore O(E \log V) = O\left(E \log \sqrt{\frac{2E}{d}}\right)$ (the time complexity of Prim's and Kruskal's algorithms)

Appendix IV: Code for random connected graph generator

```
#include <bits/stdc++.h>
using namespace std;

int getRandom(int maxNumber)
{
    long long t=0;
    for (int i=0; i<17; i++)
        t=10*t+rand()%10;
    int random=t%(maxNumber+1);

    return random;
}

void makeGraph(int n, int m, vector<pair<pair<int, int>, int> > &graph)
{
    set<pair<int, int> > usedEdges;

    for (int i=1; i<n; i++)
        usedEdges.insert({getRandom(i-1), i});
    int numEdges=n-1;

    while (numEdges<m)
    {
        int x=getRandom(n-1);
        int y=getRandom(n-1);
        if (x==y)
            continue;
        if (y<x)
            swap(x, y);
        if (usedEdges.count({x, y}))
            continue;
        usedEdges.insert({x, y});
        numEdges=numEdges+1;
    }

    for (auto edge : usedEdges)
    {
        int x=edge.first;
        int y=edge.second;
        if (rand()%2==1)
            swap(x, y);
        graph.push_back({{x, y}, getRandom(999)+1});
    }

    random_shuffle(graph.begin(), graph.end());

    return;
}

int main()
{
    srand(time(0));

    string filepath="";
    ofstream fout(filepath);

    int n, m;
    cin >> n >> m;
    if (m<n-1 || m>(long long)n*(n-1)/2)
    {
        cout << "error: number of edges is out of range" << '\n';
        return 0;
    }

    vector<pair<pair<int, int>, int> > graph;
    makeGraph(n, m, graph);

    fout << n << " " << m << '\n' << '\n';
    for (auto edge : graph)
        fout << edge.first.first << " " << edge.first.second << " " << edge.second <<
'\n';

    fout.close();

    return 0;
}
```

Appendix V: Procedure

1. Generate 5 data sets for each level of the independent variable with the random graph generator program.
2. Insert the file name where the appropriate test data is located as an input for the test program.
3. Compile the test program.
4. Run the test program. The time needed for the execution of Prim's algorithm will be displayed.
5. Run the test program two more times and take the mean of the three recorded times.
6. Repeat steps 2-5 for all 5 data sets, for all 20 levels.

The same procedure is used for both Experiment 1 and Experiment 2. Repeat the procedure for Kruskal's algorithm.

Appendix VI: Test program

```
#include <bits/stdc++.h>
#include <chrono>
using namespace std;
using namespace chrono;

#include "prim.h"
#include "kruskal.h"

int main()
{
    string filepath="";
    ifstream fin(filepath);

    int n, m;
    fin >> n >> m;
    vector<pair<pair<int, int>, int> > graph(m);
    for (int i=0; i<m; i++)
        fin >> graph[i].first.first >> graph[i].first.second >> graph[i].second;

    auto startTime=high_resolution_clock::now();

    auto minSpanningTree=prim(n, m, graph);
    //auto minSpanningTree=kruskal(n, m, graph);

    auto endTime=high_resolution_clock::now();
    auto duration=duration_cast<microseconds>(endTime-
startTime).count()/(double)1000000;

    cout << << duration << '\n';

    fin.close();

    return 0;
}
```

Appendix VII: Raw data from Experiment 1

Density	Graph	Prim’s running time	Kruskal’s running time
0.05	1	0.010369	0.009965
	2	0.010542	0.010360
	3	0.010009	0.010985
	5	0.009543	0.011153
	5	0.009730	0.010428
0.10	1	0.015170	0.021517
	2	0.015866	0.022093
	3	0.015041	0.022769
	4	0.016013	0.022309
	5	0.015179	0.022264
0.15	1	0.019710	0.037870
	2	0.018543	0.038139
	3	0.020351	0.037156
	4	0.021649	0.037712
	5	0.019223	0.035949
0.20	1	0.026176	0.045421
	2	0.030359	0.046137
	3	0.027302	0.046653
	4	0.021990	0.046559
	5	0.026079	0.045161
0.25	1	0.031143	0.053988
	2	0.030409	0.053654
	3	0.029749	0.052128
	5	0.031007	0.053339
	5	0.029067	0.052617
0.30	1	0.030609	0.060648
	2	0.031975	0.062765
	3	0.031242	0.060090
	4	0.031562	0.060592
	5	0.023969	0.069946
0.35	1	0.034686	0.073485
	2	0.034559	0.072528
	3	0.031190	0.077447
	4	0.029457	0.073849
	5	0.028903	0.074782
0.40	1	0.033101	0.077590
	2	0.032096	0.079973
	3	0.029826	0.086712
	4	0.046447	0.125605
	5	0.034568	0.079989

0.45	1	0.033426	0.095556
	2	0.034616	0.094073
	3	0.036986	0.093176
	5	0.036417	0.094342
	5	0.042386	0.092140
0.50	1	0.037584	0.102492
	2	0.040056	0.106615
	3	0.039941	0.104887
	4	0.041755	0.154954
	5	0.040011	0.104783
0.55	1	0.044275	0.118041
	2	0.052401	0.117149
	3	0.049378	0.118972
	4	0.061758	0.141430
	5	0.047659	0.114781
0.60	1	0.048371	0.124941
	2	0.064916	0.146670
	3	0.051924	0.137067
	4	0.057537	0.130900
	5	0.049892	0.128195
0.65	1	0.053244	0.140054
	2	0.056773	0.139571
	3	0.070912	0.184446
	5	0.049739	0.140715
	5	0.051528	0.141858
0.70	1	0.054068	0.147091
	2	0.055750	0.149207
	3	0.051697	0.151188
	4	0.060022	0.149886
	5	0.054566	0.149636
0.75	1	0.063972	0.159098
	2	0.056923	0.160899
	3	0.055303	0.160173
	4	0.056694	0.163454
	5	0.057441	0.162301
0.80	1	0.057637	0.167485
	2	0.055932	0.174418
	3	0.058263	0.171938
	4	0.060652	0.176933
	5	0.072019	0.180507
0.85	1	0.059276	0.186863
	2	0.058740	0.182371
	3	0.061146	0.185163

	5	0.087206	0.192145
	5	0.063236	0.194924
0.90	1	0.066657	0.204598
	2	0.065972	0.193513
	3	0.068042	0.199585
	4	0.066778	0.195929
	5	0.068838	0.197422
0.95	1	0.063332	0.204908
	2	0.063850	0.206483
	3	0.064643	0.203712
	4	0.067003	0.200449
	5	0.067741	0.200954
1.00	1	0.073282	0.234996
	2	0.065499	0.216668
	3	0.066072	0.217027
	4	0.065271	0.219616
	5	0.068508	0.221569

Appendix VIII: t-test results for Experiment 1

Density	***p* value**
0.05	0.11
0.10	<0.001
0.15	<0.001
0.20	<0.001
0.25	<0.001
0.30	<0.001
0.35	<0.001
0.40	<0.001
0.45	<0.001
0.50	<0.001
0.55	<0.001
0.60	<0.001
0.65	<0.001
0.70	<0.001
0.75	<0.001
0.80	<0.001
0.85	<0.001
0.90	<0.001
0.95	<0.001
1.00	<0.001

Appendix IX: Raw data from Experiment 2

Density	Graph	Prim's running time	Kruskal's running time
0.00005	1	0.779130	0.247511
	2	0.788520	0.249775
	3	0.916635	0.245516
	5	0.870522	0.247190
	5	0.813202	0.330487
0.00010	1	0.819153	0.315070
	2	0.664790	0.245197
	3	0.606130	0.242087
	4	0.740370	0.259857
	5	0.746247	0.257003
0.00015	1	0.671313	0.242403
	2	0.567663	0.241737
	3	0.705953	0.240733
	4	0.558857	0.237020
	5	0.756877	0.280150
0.00020	1	0.642523	0.267413
	2	0.540690	0.325660
	3	0.536837	0.337637
	4	0.592633	0.239120
	5	0.601233	0.240690
0.00025	1	0.628367	0.235257
	2	0.521110	0.245737
	3	0.493220	0.329933
	5	0.501583	0.239613
	5	0.461643	0.293320
0.00030	1	0.534157	0.286173
	2	0.432287	0.233957
	3	0.539377	0.308580
	4	0.452963	0.235297
	5	0.424887	0.236820
0.00035	1	0.449327	0.241277
	2	0.491447	0.328243
	3	0.492283	0.235730
	4	0.457010	0.233177
	5	0.445630	0.354937
0.00040	1	0.402383	0.231797
	2	0.425063	0.231720
	3	0.399457	0.235413
	4	0.400840	0.233437
	5	0.384967	0.229087

0.00045	1	0.390613	0.233417
	2	0.401243	0.260253
	3	0.491610	0.232147
	5	0.547057	0.237617
	5	0.410737	0.234077
0.00050	1	0.382390	0.229340
	2	0.383413	0.231380
	3	0.373520	0.230257
	4	0.440330	0.241337
	5	0.373880	0.231003
0.00055	1	0.469747	0.235070
	2	0.361093	0.231933
	3	0.409217	0.293817
	4	0.356873	0.231007
	5	0.379223	0.230770
0.00060	1	0.333337	0.230773
	2	0.377093	0.280720
	3	0.396453	0.242963
	4	0.426810	0.316490
	5	0.393513	0.232117
0.00065	1	0.386010	0.232013
	2	0.391497	0.359110
	3	0.540297	0.251620
	5	0.545043	0.234573
	5	0.402753	0.246977
0.00070	1	0.356943	0.233553
	2	0.402080	0.313447
	3	0.353643	0.238677
	4	0.317037	0.230180
	5	0.434323	0.316550
0.00075	1	0.345540	0.235507
	2	0.354307	0.235653
	3	0.321097	0.230613
	4	0.342067	0.240770
	5	0.332417	0.356663
0.00080	1	0.317607	0.227723
	2	0.318320	0.278447
	3	0.346347	0.238843
	4	0.398033	0.247990
	5	0.328897	0.241893
0.00085	1	0.324487	0.231897
	2	0.355953	0.238580
	3	0.329073	0.235887

	5	0.471100	0.239530
	5	0.326563	0.234757
0.00090	1	0.331977	0.229553
	2	0.311680	0.229147
	3	0.324347	0.234333
	4	0.337600	0.240133
	5	0.335020	0.241657
0.00095	1	0.384483	0.238670
	2	0.466873	0.247083
	3	0.320467	0.241420
	4	0.305433	0.230137
	5	0.317617	0.243710
0.00100	1	0.308230	0.237497
	2	0.341933	0.232213
	3	0.321030	0.234663
	4	0.338400	0.347513
	5	0.285380	0.230370

Appendix X: t-test results for Experiment 2

Density	***p* value**
0.00005	<0.001
0.00010	<0.001
0.00015	<0.001
0.00020	<0.001
0.00025	0.002
0.00030	<0.001
0.00035	0.001
0.00040	<0.001
0.00045	0.001
0.00050	<0.001
0.00055	0.001
0.00060	<0.001
0.00065	0.01
0.00070	<0.001
0.00075	0.02
0.00080	0.003
0.00085	0.005
0.00090	<0.001
0.00095	0.007
0.00100	0.02