UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DIEGO HOMMERDING AMORIM

Assignment 1
K-Ary Heaps and Dijkstra's Algorithm

Porto Alegre
2025

1.1 Task

Implement Dijkstra's Algorithm that allows the calculation of the minimal path value between one vertex and all the others of a given weighted graph in $O((n+m)log(n))$ time complexity, where n is the number of vertices and m is the number of edges. We want to validate that the time complexity is in fact $O((n+m)log(n))$.

1.2 Solution

Dijkstra's algorithm was implemented using a k-ary heap, with a focus on determining the optimal value for k. Experimental tests were conducted with k values ranging from 2 to 128 across 18 graphs, each with varying numbers of vertices and edges. Based on the results, it was concluded that the optimal value for k is 8. Therefore, for the remainder of this assignment, an 8-ary heap is used as the data structure. The graph is represented using an adjacency list, and the k-ary heap includes a hash array of size n (the number of vertices) to enable constant-time $O(1)$ lookup.

1.3 Test environment

The results were obtained using a *Lenovo Ideapad S145-15IWL* notebook with an **Intel(R) Core(TM) i7-8565U** processor running at **1.80 GHz** and **20 GB of RAM**. The system was running **Ubuntu 22.04 LTS** with Linux kernel 6.8.0. The code was compiled using **g++ 11.4.0** with optimization flags `-O2` enabled.

1.4 Results

The figure 1.1 shows the average performance of Dijkstra's algorithm running on 18 graphs with varied values of n (number of vertices) and m (number of edges). For each branching factor k (ranging from 2 to 128), the 18 graphs were processed 30 times to compute the average run time. As we can see, there appears to be a sweet spot for the branching factor k of the k-ary heap between $7 \leq k \leq 9$, where the runtime is minimized.

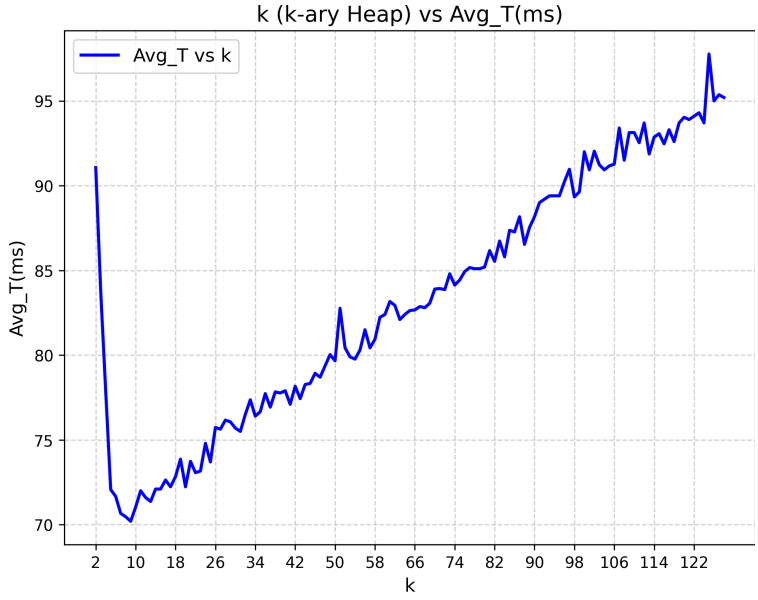

Figure 1.1 – Average runtime (ms) for Dijkstra's algorithm running on 18 varied graphs.

1.4.1 Complexity by Counting Operations

This section presents the results obtained by counting the number of operations at both the heap and algorithm levels.

1.4.1.1 Heap Level

At the heap level, the number of **sift-up** and **sift-down** operations performed during each heap operation—**insert**, **deletemin**, and **update**—was recorded. These counts were then used to compute the metric:

$$r = \frac{\#\text{sift}}{\log_k n'}$$

for each operation, where:

- $\#$sift is the total number of sift-up/sift-down operations,
- $\log_k n'$ is the logarithm of n' (current heap size) with base k (heap branching factor).

The average value $\overline{r}$ was calculated for each heap operation in 18 graphs, which varied in terms of n (number of vertices) and m (number of edges). These results are summarized in **Table 1.1**, which provides a detailed breakdown of $\overline{r}$ for each operation and graph.

Table 1.1 – Summary of $\overline{r}$ for Heap Operations

Graph	**n**	**m**	**Insert**	**Delete**	**Update**
Graph 1	32	205	0.083	0.724	0.118
Graph 2	64	601	0.161	0.779	0.048
Graph 3	128	1618	0.077	0.764	0.073
Graph 4	128	2310	0.094	0.795	0.058
Graph 5	256	6624	0.104	0.810	0.053
Graph 6	512	26267	0.076	0.874	0.048
Graph 7	512	26434	0.102	0.874	0.050
Graph 8	1024	52342	0.064	0.827	0.038
Graph 9	1024	52604	0.074	0.824	0.037
Graph 10	2048	209510	0.071	0.860	0.036
Graph 11	2048	209767	0.068	0.854	0.039
Graph 12	4096	84460	0.057	0.900	0.040
Graph 13	4096	168167	0.059	0.902	0.037
Graph 14	8192	3355588	0.058	0.804	0.037
Graph 15	8192	6713838	0.059	0.693	0.048
Graph 16	16384	269019	0.046	0.886	0.031
Graph 17	16384	13420481	0.055	0.731	0.053
Graph 18	32768	106199	0.025	0.886	0.037

1.4.1.2 Algorithm Level

At the algorithm level, the number of **insert**, **deletemin**, and **update** operations was recorded. These counts were used to compute the metrics:

$$i = \frac{\#\text{inserts}}{n}, \quad d = \frac{\#\text{deletemins}}{n}, \quad u = \frac{\#\text{updates}}{m}$$

where:

- $\#$ denotes the total count of the respective operation,
- n is the number of vertices of the graph,
- m is the number of edges of the graph.

These values were calculated for each heap operation in 18 graphs, which varied in terms of n (number of vertices) and m (number of edges). These results are summarized in **Table 1.2**, which provides a detailed breakdown of i, d and u for each graph.

Table 1.2 – Summary of i, d and u for Dijkstra Operations

Graph	**n**	**m**	**i**	**d**	**u**
Graph 1	32	205	1.000	1.000	0.083
Graph 2	64	601	1.000	1.000	0.105
Graph 3	128	1618	1.000	1.000	0.093
Graph 4	128	2310	1.000	1.000	0.085
Graph 5	256	6624	1.000	1.000	0.073
Graph 6	512	26267	1.000	1.000	0.052
Graph 7	512	26434	1.000	1.000	0.050
Graph 8	1024	52342	1.000	1.000	0.050
Graph 9	1024	52604	1.000	1.000	0.049
Graph 10	2048	209510	1.000	1.000	0.031
Graph 11	2048	209767	1.000	1.000	0.031
Graph 12	4096	84460	1.000	1.000	0.080
Graph 13	4096	168167	1.000	1.000	0.057
Graph 14	8192	3355588	1.000	1.000	0.011
Graph 15	8192	6713838	1.000	1.000	0.007
Graph 16	16384	269019	1.000	1.000	0.088
Graph 17	16384	13420481	1.000	1.000	0.007
Graph 18	32768	106199	0.955	0.955	0.064

1.4.2 Complexity by Measuring Time

This section presents the results obtained by computing the average number of inserts, deletemins, and updates, as well as the average runtime of the algorithm on two graph datasets, using 30 repetitions for each graph. The first dataset has a fixed number of vertices with a varying number of edges, while the second dataset has a fixed number of edges with a varying number of vertices. The operation plots used the definitions for i, d, and u from section *1.4.1.2*, while the time plots calculated the normalized time $\frac{T}{(n+m)\log(n)}$ in seconds, where T represents the execution time of the algorithm. To ensure consistency in the execution time measurements, the algorithm was set to run in a monothreaded configuration. It is important to note that the execution time reported does **not** include the time required to read and parse the graph; it only accounts for the execution of Dijkstra's algorithm itself.

1.4.2.1 Fixed Number of Vertices, Varying Number of Edges

In this dataset, the number of vertices is fixed at 2^{15}, while the number of edges varies according to the formula $\sqrt{2}^{i}$, where i ranges from 40 to 49. This gives us the following edge counts for each iteration:

$$\text{Edges} = \sqrt{2}^{40}, \sqrt{2}^{41}, \ldots, \sqrt{2}^{49}$$

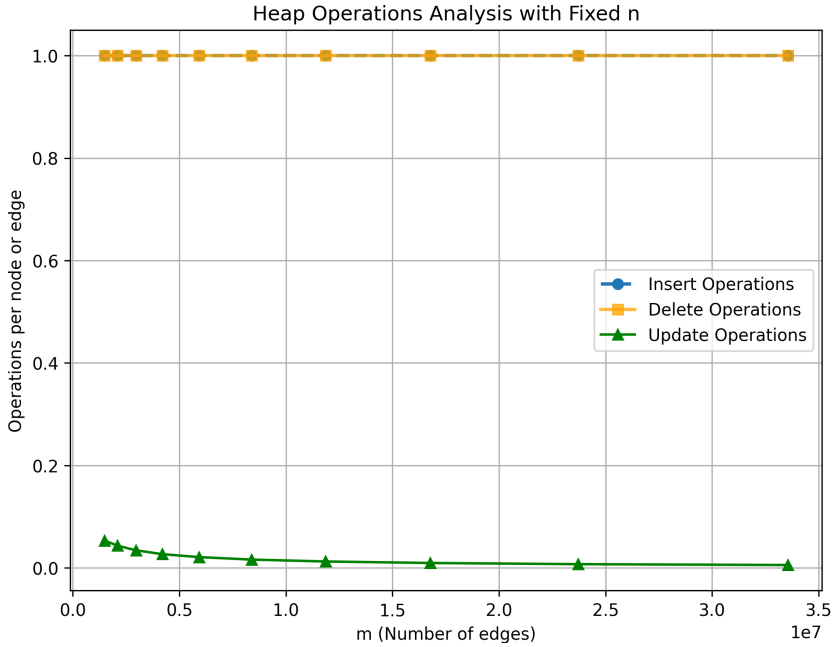

Figure 1.2 – Heap Operations Analysis with Fixed n

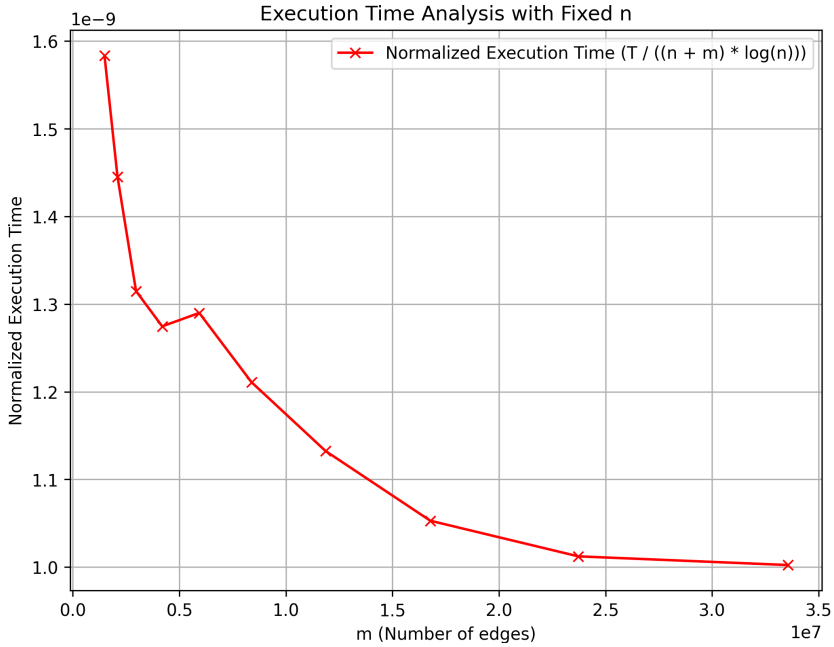

Figure 1.3 – Normalized Execution Time (s) Analysis with Fixed n

1.4.2.2 Fixed Number of Edges, Varying Number of Vertices

In this dataset, the number of edges is fixed at 2^{20}, while the number of vertices varies according to the formula 2^i, where i ranges from 10 to 19. This gives us the following vertex counts for each iteration:

$$\text{Vertices} = 2^{10}, 2^{11}, \ldots, 2^{19}$$

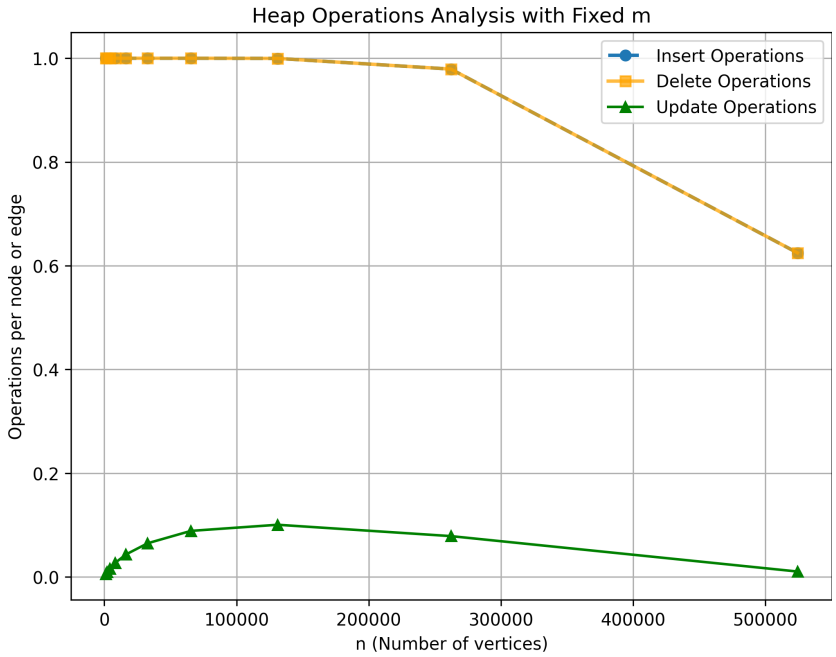

Figure 1.4 – Heap Operations Analysis with Fixed m

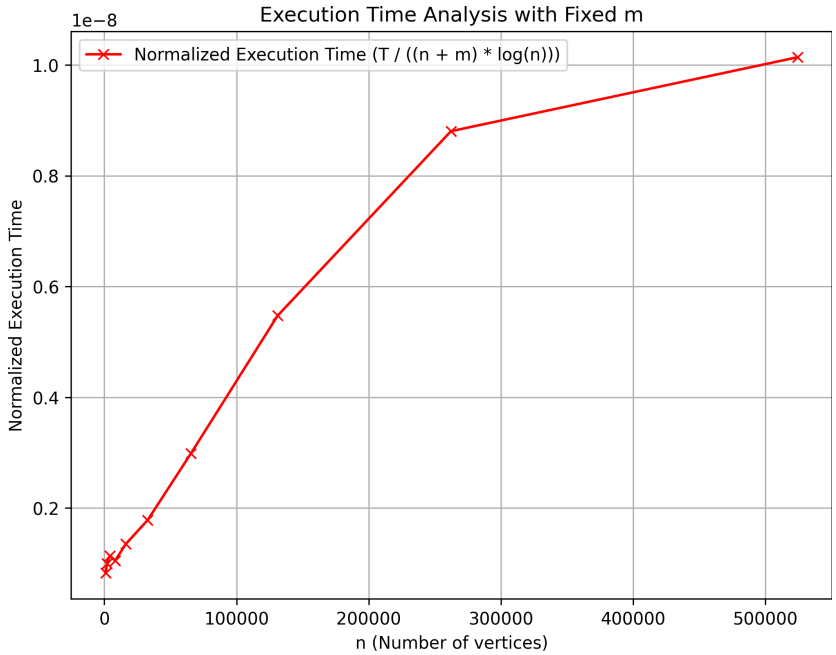

Figure 1.5 – Normalized Execution Time (s) Analysis with Fixed m

1.4.2.3 Linear Regression

Using the hypothesis $T(n, m) \sim an^b m^c$ based on the data obtained from running the algorithm on both datasets, the regression analysis provides us with the following parameters:

$$a \approx 7.87 \times 10^{-9}, \quad b \approx 0.575, \quad c \approx 0.658.$$

This implies that the execution time T of the algorithm can be approximated by the formula:

$$T(n, m) \approx 7.87 \times 10^{-9} n^{0.575} m^{0.658},$$

where $T(n, m)$ represents the time in seconds, n is the number of vertices, and m is the number of edges in the graph.

The theoretical worst-case time complexity of the algorithm is given by:

$$(n + m) \log n,$$

which suggests that, in the worst case, the algorithm's runtime grows linearly with respect to the number of vertices n and edges m, with an additional logarithmic factor based on the number of vertices n.

1.4.3 Evaluate Scalability

This section presents the results of the scalability tests, where the algorithm was executed 30 times using the NY and USA road network graphs as inputs. The summarized results are shown in **Table 1.3**, which provides a detailed breakdown of T and M for each graph. Here, T represents the average execution time (in seconds), and M denotes the average memory consumption (in MB). The graph datasets were obtained from the 9th DIMACS Implementation Challenge, available at <http://www.diag.uniroma1.it/ challenge9>.

Table 1.3 – Average execution time and memory consumption: NY and USA

Graph	**n**	**m**	**T(s)**	**M(MB)**
NY	264346	733846	0.063	0.635
USA	23947347	58333344	8.360	277.385

1.5 Conclusion

Analyzing the results, it is possible to formulate some hypotheses and conjectures:

- **Hypothesis 1:** The optimal branching factor k for the Heap data structure used in Dijkstra's algorithm is 8, which corresponds to the "sweet spot"on the performance graph in Figure 1.1 from Section 1.4. Additionally, the varying k-values seem to exhibit a parabolic behavior.

- **Hypothesis 2:** As observed in Section 1.4.1.1, the average number of sifts executed during Insert and Update operations is empirically around 5-10% of the worst-case scenario. In contrast, for Delete operations, this value is significantly higher, approximately 70-80%. This is likely due to the `heapify down` process, which involves moving the swapped root node to one of the leaf positions.

- **Hypothesis 3:** As observed in Section 1.4.1.2 and Figures 1.2 and 1.4, the number of Insert and Deletemin operations is proportional to the total number of vertices, unless the graph is not fully connected (e.g., Graph 18 in 1.2). In contrast, the number of Update operations is roughly 5-10% of the total number of edges, suggesting that empirically, the number of updates is significantly lower than the worst-case scenario.

- **Hypothesis 4**: As observed in Figures 1.3 and 1.5, the normalized execution time decreases with more edges, indicating an amortized efficiency gain in denser graphs. However, as the number of vertices increases while keeping edges fixed, execution time worsens, suggesting that sparsity negatively impacts performance (until the graph becomes too disconnected). Both graphs show logarithmic-like behavior until stagnation. Additionally, the execution time remains much lower than the worst-case bound, confirming that the algorithm performs better than the worst-case complexity, as shown by the regression formula in Section 1.4.2.3.

- **Hypothesis 5**: As observed in Section 1.4.3, the algorithm scales with larger graphs, with execution time increasing 133×, while the number of nodes increases 90× and edges 80×. However, memory usage increases 437×, reaching 277 MB, scaling faster than execution time. This indicates that memory consumption suffers more with denser graphs, which could become a limitation for even larger datasets. Despite this, the algorithm handled a massive graph in an acceptable time, demonstrating that it still scales well for large real-world networks.

Based on the findings, we conclude that the algorithm performs significantly better than its worst-case complexity and scales well for large graphs, handling even massive real world graphs within an acceptable execution time. Additionally, obtaining consistent execution time measurements remains a challenging and difficult process due to system variations and caching effects.

REFERÊNCIAS

Robert Sedgewick, *Algorithms for the Masses*, ANALCO'11, San Francisco, January 2011. Note: Modi Memorial Lecture, Drexel University, April 2012.

David S. Johnson, *A theoretician's guide to the experimental analysis of algorithms*, Proceedings of the 5th and 6th DIMACS Implementation Challenges, 2002.

Robert Sedgewick and Kevin Wayne, *Algorithms*, 4th edition, Addison-Wesley, 2011.

Stack Overflow, *How do I programmatically check memory use in a fairly portable way (C/C++)*, Available at: <http://stackoverflow.com/questions/372484/how-do-i-programmatically-check-memory-use-in-a-fairly-portable-way-c-c%29>. Accessed March 2025.

DIMACS, *9th DIMACS Implementation Challenge*. Available at: http://www.diag.uniroma1.it/ challenge9/download.shtml. Accessed March 2025.

GeeksforGeeks, *K-ary Heap*. Available at: https://www.geeksforgeeks.org/k-ary-heap/. Accessed March 2025.

W3Schools, *Python Matplotlib Plotting*, Available at: <https://www.w3schools.com/python/matplotlib_plotting.asp>. Accessed March 2025.

ResearchGate, *Example Graph for Dynamic Dijkstra Algorithm*, Available at: <https://www.researchgate.net/figure/Example-Graph-for-dynamic-Dijkstra-algorithm_fig5_323578961>. Accessed March 2025.