UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA

CARLOS WESTERMANN

K-ary Heaps and Dijkstra's algorithm

Porto Alegre
2025

1 Introduction

The following work presents an implementation of a k-ary heap and Dijkstra's algorithm in order to evaluate optimal k-values, compare theoretical worst-case complexity and real complexity, and measure how well the algorithm scales.

2 Testing

2.1 Testing Environment

All test cases were run on a machine with the following specifications:

- Chip: Apple M3 Pro
- Total Number of Cores: 12 (6 performance and 6 efficiency)
- Memory: 18 GB

2.2 Testing Methodology

The evaluation of the algorithm and data structure evaluation was carried out in three phases, each evaluating different aspects of performance and scalability.

1. **Operation Counting:** This phase focused on determining the complexity of heap operations (*insert*, *deletemin*, *update*) and the Dijkstra algorithm by counting the number of elementary operations performed. Randomly generated graphs were used with varying numbers of vertices (n) and edges (m), and multiple trials were performed for each graph. The ratios $i = \frac{\#insert}{n}$, $d = \frac{\#deletemin}{n}$, and $u = \frac{\#update}{m}$ were calculated to calculate the efficiency of the algorithm. Similarly, the ratios of heapify operations to the theoretical maximum ($log_k n$) were computed to analyze heap performance.
2. **Time Measurement:** This phase measured the execution time of Dijkstra's algorithm to compare empirical performance with theoretical bounds. Two sub-experiments were performed:
    - **Varying Edges (Fixed Vertices):** The number of vertices was fixed at $n = 2^{20}$, while the number of edges (m) was varied across a range of values $m =$

$\sqrt{2}^i$, where $i \in \{41, 42, \ldots, 50\}$. The execution time was recorded for each configuration, and the ratio $\frac{T}{(n+m)\log(n)}$ was plotted against m to observe the behavior as the graph changed.

- **Varying Vertices (Fixed Edges):** The number of edges was fixed at $m = 2^{20}$, while the number of vertices (n) was varied $n = 2^i$, where $i \in \{11, 12, \ldots, 20\}$. The same analysis as above was performed, plotting $\frac{T}{(n+m)\log(n)}$ against n.

Finally, a linear regression model was applied to the combined timing data to estimate the relationship between execution time T, number of vertices n, and number of edges m, expressed as $T(n, m) \sim an^b m^c$.

3. **Scalability on real Graphs:** This phase evaluated the performance of the implementation on real-world graph datasets ("NY.gr"and "USA.gr") for various values of k. For each graph and k value, 30 tests were performed and the average execution time and memory consumption were recorded to assess how the algorithm scales in practical scenarios.

3 Relevant implementation aspects

The implementation was done based on the pseudo-code implementation provided in the text "notas de aula".

Two similar implementations were made, one that follows the specifications for the assignment and one that generates the data that was used in testing. They both implement the data structure and algorithm in the exact same way, but one captures the operation count and outputs a verbose summary.

The implementation utilizes the following key data structures:

- **K-ary Heap (Priority Queue):**
  - `heap_arr`: Integer array to store vertex indices in the heap.
  - `positions`: Integer array to map vertex indices to their positions within `heap_arr`.
  - `dist`: Integer vector to store the current shortest distance for each vertex.
- **Graph Representation:**
  - `Adjacency list`: vector of vector of pairs where each element of the outer vector represents a vertex, and the inner vector stores the pair representing

adjacent vertices and the corresponding edge weights.

- **Dijkstra's Algorithm:** `k-heap`: An instance of the k-ary heap serves as a priority queue, storing vertices to be processed based on their current shortest distance estimate.

4 Experimentation results and analysis

4.1 Operation Counting (Section 1)

4.1.1 Heap-Level Operations (Section 1.1)

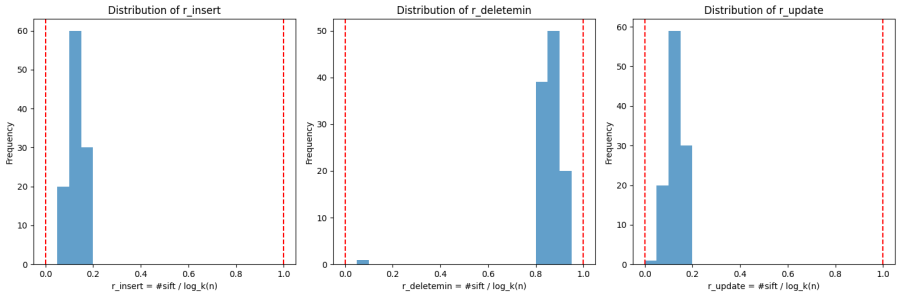

Figura 1 – Distribution of r-values for Heap Operations

As shown in Figure 1, the distributions of r-values ($\frac{\#sift}{\log_k n}$) for *insert*, *deletemin*, and *update* operations are bounded between 0 and 1, as expected. This indicates that the number of heapify operations performed in practice aligns with the theoretical $O(\log_k n)$ bound. The distributions for *insert* and *update* operations are concentrated closer to 0, suggesting efficient heap operations relative to the theoretical upper bound. The distribution for *deletemin* operations is more towards 1, indicating a performance closer to the theoretical worst-case scenario. This may suggest that the `deletemin` operation is the bottleneck in the heap implementation.

4.1.2 Algorithm-Level Operations (Section 1.2)

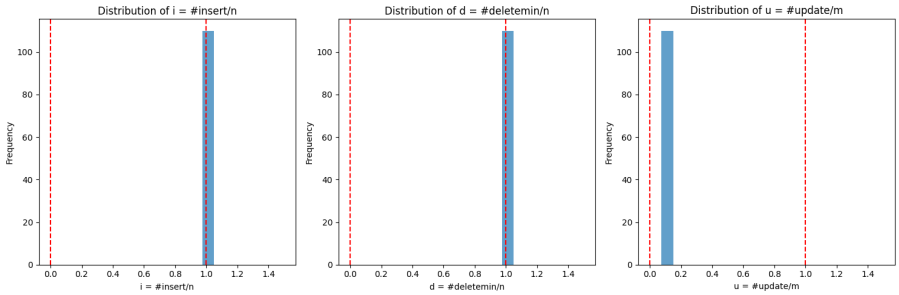

Figura 2 – Distribution of i, d, and u Ratios for Algorithm Operations

Figure 2 displays the distributions of the ratios i, d, and u, representing the number of *insert*, *deletemin*, and *update* operations, normalized by the number of vertices (n) or edges (m). The distributions of i and d are clustered around 1. This confirms that each vertex is inserted into and removed from the heap exactly once. This happens because we set the destination vertex outside the graph, forcing the algorithm to explore the entire graph.

4.2 Time Complexity Measurement (Section 2)

4.2.1 Varying Edges (Fixed Vertices)

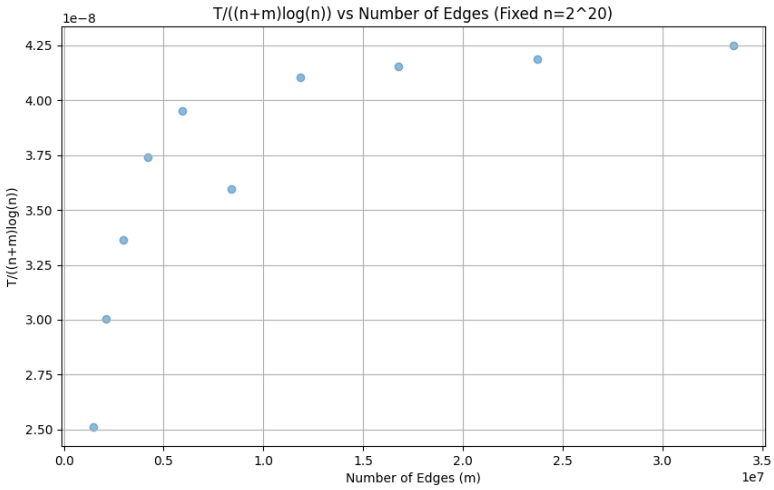

Figura 3 – Normalized Time vs. Number of Edges (Fixed Vertices)

Figure 3 illustrates the relationship between the normalized execution time and the number of edges when the number of vertices is fixed. The execution time increases

as the number of edges increases. This suggests that the algorithm's performance is more dependent on the number of edges.

4.2.2 Varying Vertices (Fixed Edges)

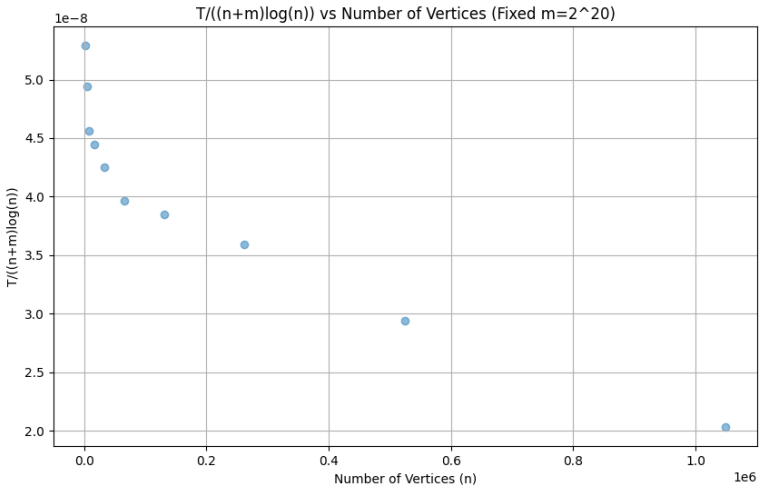

Figura 4 – Execution Time vs. Number of Vertices (Fixed Edges)

Figure 4 presents the relationship between the execution time and the number of vertices when the number of edges is fixed. The execution time decreases as the number of vertices increases. This is the opposite of what happened for edges, where even though it increased, the numbers of vertex had a low impact. With this scenario, in which time decreased as vertices increased, this might indicate edge density as a more relevant factor to the execution time.

4.2.3 Linear Regression

The linear regression analysis yielded the following model:

$$T \approx e^{-15.0642} \cdot n^{0.0708} \cdot m^{1.0087} \approx 0.0000 \cdot n^{0.0708} \cdot m^{1.0087}$$

The R-squared value of 0.9987 indicates a strong fit, suggesting that the model accurately represents the relationship between execution time, the number of vertices, and the number of edges. The exponent of m is very close to 1, indicating a nearly linear relationship between execution time and the number of edges. The exponent of n is close to 0, suggesting that execution time is relatively insensitive to the number of vertices within the tested range of n and m. This behavior could be due to the fact that, with a

fixed number of edges, increasing the number of vertices primarily affects the density of the graph rather than the workload of the algorithm.

4.3 Scalability on Real Graphs (Section 3)

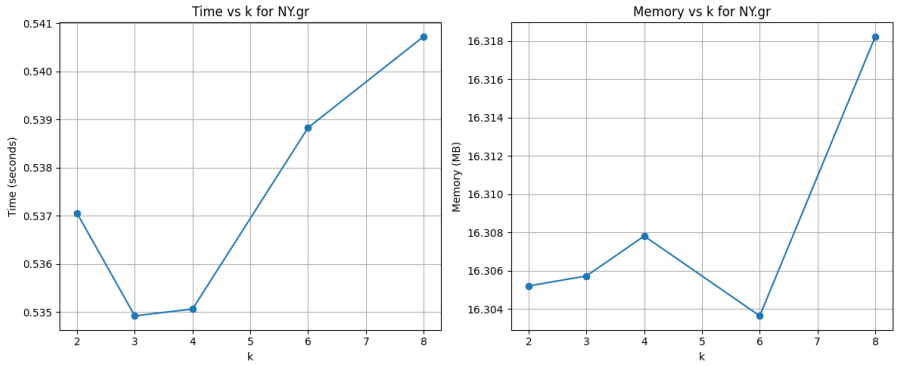

Figura 5 – Performance on NY.gr

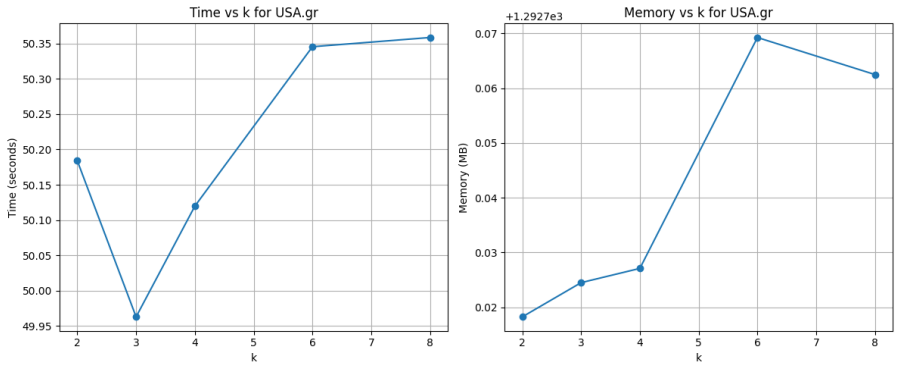

Figura 6 – Performance on USA.gr

Figures 5 and 6 illustrate the performance of Dijkstra's algorithm with varying values of k on the "NY.gr"and "USA.gr"graphs. While smaller values of k tend to exhibit slightly better performance, the differences are marginal. Specifically, for the larger "USA.gr"graph, the execution time difference is less than 0.5 seconds, representing an improvement of less than 0.6%. This suggests that the choice of k has a limited impact on the overall performance of Dijkstra's algorithm on these real-world graphs. Other factors, such as the graph structure and memory access patterns, might play a more significant role.

5 Conclusion

The experimentation and analysis presented in this work provide an evaluation of a k-ary heap implementation within Dijkstra's algorithm. Operation counting reveals that heap operations align with theoretical complexity bounds. Scalability tests on real-world graphs indicate that the heap arity (k) has a relatively minor impact on overall performance, suggesting real-world graph performance isn't directly related to theoretical perfomance of heap arity. These insights collectively contribute to a deeper understanding of the practical performance characteristics of k-ary heaps in the context of dijkstras algorithm.