A Comparison of The Different Implementations of Dijkstra's Algorithm

Brandon Smith

SIT102, Introduction to Programming

Abstract

The efficiency difference between adjacency list and matrix representations shines a spotlight of the necessity of strategic data structure selection, especially in large datasets. Binary Heap and Balanced BST implementations show remarkable efficiency with adjacency lists, despite some performance variability. On the contrarily, the Basic Form List, while not as efficient, demonstrates high stability, serving as a reliable choice for predictable performance. Matrix implementations reveal the inherent inefficiencies of the Basic Form Matrix, which can be somewhat mitigated by advanced data structures like Binary Heap and Balanced BST. These findings advocate for the use of sophisticated data structures in Dijkstra's algorithm implementation, underscoring the practical importance of understanding the interplay between algorithmic choices and data structures.

1 Introduction

Dijkstra's algorithm, known for finding the shortest paths between vertices in edge- and arc-weighted graphs, finds wide-ranging applications across sectors such as transportation, telecommunications, social network analytics, and financial transactions. Its influence extends to solving real-world problems, like devising the most efficient routes in a complex road network [1].

The purpose of this is purely educational. I seek to understand the varying execution times of Dijkstra's algorithm when paired with different data structures, including self-balancing binary trees, binary heaps, and Fibonacci heaps. The objective is not only to identify the optimal method for potential real-world scenarios but also to report my learning experience, aiming to advance our understanding of the algorithm and its interactions with distinct data structures.

2 Adjacency Lists and Adjacency Matrices

There are two primary structures of graph representation in Dijkstra's algorithm: the adjacency list and the adjacency matrix. These structures can be metaphorically compared to the map of a city where buildings correspond to vertices and the roads between them represent the edges.

The adjacency list is similar to a concise travel guide, where each page is dedicated to a specific vertex (building), and a list of edges (roads) leading from that vertex to other vertices (buildings) is provided. This structure proves particularly advantageous in scenarios where the graph (city) is sparse, signifying that there are fewer edges (roads) connecting the vertices (buildings). Its compact nature is an effective solution for representing information where the edge-to-vertex ratio is low. However, the adjacency list falls short when verifying the direct connection between two specific vertices (buildings) as it requires searching through the guide, reflecting a time-consuming operation [2].

On the contrary, the adjacency matrix is more closely related to a detailed city metro map, with each cell of the matrix representing a potential edge (road) between two vertices (buildings). This matrix efficiently depicts the presence or absence of an edge (metro line) between two vertices with the use of boolean values (1 indicating presence, and 0 indicating absence). This structure excels in representing dense graphs, where numerous edges (metro lines) directly connect vertices (buildings), allowing for instant verification of direct connections. However, it requires a substantial amount of memory to store the matrix, especially in the case of a sparse graph where most cells of the matrix (details on the metro map) are utilised [3].

The choice between an adjacency list (travel guide) and an adjacency matrix (metro map) is contingent on the nature of the graph (city). While the former is suitable for sparse graphs with fewer connections, the latter is fitting for dense graphs with numerous direct connections. Dijkstra's algorithm is versatile in its application, accommodating both structures efficiently. The selection of a suitable representation can lead to an optimised implementation of Dijkstra's algorithm, ensuring a smooth exploration of the graph (city).

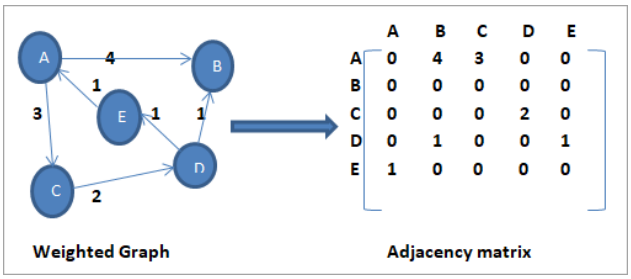

Figure 1: A representation of a weighted graph using an adjacency matrix. Unlike other graph types, the adjacency matrix for a weighted graph replaces non-zero values with the actual weights of the edges. For example, the edge AB, which carries a weight of 4, is represented in the matrix at the intersection of A and B as 4. This pattern is replicated for all edges, creating a detailed, weighted map of all vertices and their interconnections [4].

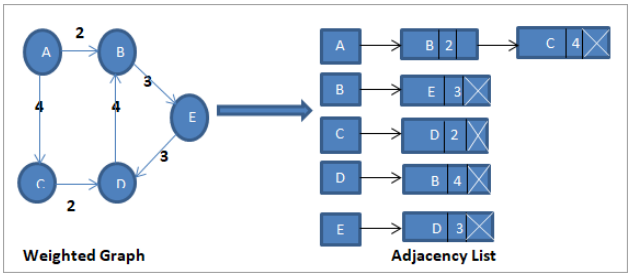

Figure 2: A representation of a weighted graph using an adjacency list. Here, an additional field is incorporated into each node of the adjacency list, representing the weight of the edge. This method allows for easy addition of vertices and provides space efficiency due to the linked list implementation. However, determining the presence of an edge between two specific vertices may require a less efficient operation, indicating the trade-offs of this representation [4].

3 Fibonacci Heaps, Binary Heaps, and Self-balancing Binary Trees

There are three main approaches: using (1) Fibonacci heaps, (2) binary heaps, or (3) self-balancing binary trees.

1. Fibonacci heaps play quite an important role in improving Dijkstra's algorithm's performance [5]. This data structure is notable for its constant amortized time $O(1)$ for operations such as find-minimum, insert, and decrease-key, all important for Dijkstra's algorithm. The delete operation, predominantly used for eliminating the minimum element, exhibits an amortized time complexity of $O(\log n)$, where n signifies the size of the heap. The efficient operation time makes Fibonacci heaps superior to binary or binomial heaps when the number of delete operations is comparatively smaller. Furthermore, Fibonacci heaps' ability to perform the merge operation in constant amortized time significantly benefits Dijkstra's algorithm, which frequently requires merging nodes [6].

2. Binary heaps serve as a common way to implement priority queues, used in Dijkstra's algorithm. These heaps are binary trees with the shape and heap properties ensuring proper functioning within Dijkstra's algorithm. The shape property guarantees that all tree levels are fully filled, except possibly the last one, which, if incomplete, is filled from left to right. The heap property maintains that each node's key is either greater than or equal to (max-heaps) or less than or equal to (min-heaps) the keys in the node's children. This ordering is important in the selection of the next node in Dijkstra's algorithm [7].

3. Self-balancing binary search trees (BSTs) are another data structure that can prove useful in certain implementations of Dijkstra's algorithm. These trees automatically maintain a small height, important for maintaining efficiency in the face of arbitrary item insertions and deletions. The height of such trees is defined to be logarithmic $O(\log n)$ in the number of items, which can offer performance benefits in Dijkstra's algorithm when dealing with large graphs. While not typically the first choice for a priority queue in Dijkstra's algorithm, self-balancing BSTs can still be utilized effectively, especially when insertions and deletions are frequent [8].

4 Implementation of Dijkstra's Algorithm

The reason this is considered the basic form of Dijkstra's algorithm is because it involves only the necessary steps to determine the shortest path in a graph from a source node to all other nodes. No other procedures, such as path reconstruction or advanced data structures like heaps, are used [10].

Algorithm 1: Dijkstra's Algorithm (Basic Form)

```
Function Dijkstra(Graph, source):
    create vertex set Q;
    for each vertex v in Graph do
        dist [v ] ← INFINITY;
        prev [v ] ← UNDEFINED;
        add v to Q;
    dist [source ] ← 0;
    while Q is not empty do
        u ← vertex in Q with min dist [u ];
        remove u from Q;
        for each neighbour v of u do
            alt ← dist [u ] + length (u, v);
            if alt < dist [v ] then
                dist [v ] ← alt;
                prev [v ] ← u;
    return dist [], prev [];
```

1. The algorithm begins by assigning to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.

2. It then repeatedly examines the node with the smallest current tentative distance from the start node that has not been marked as "visited", and visits all its neighbouring nodes.

3. At each visited node, it checks whether the path to that node from the start node via the current node is shorter than the currently known shortest path to that node. If it is, it updates the tentative distance of that node with the new, lower, value.

4. It continues this process until it has visited all nodes in the graph.

5. Once all nodes have been visited, the algorithm has found the shortest path from the start node to all other nodes.

Data Structure	Time Complexity
Basic form with adjacency matrix	$O(\|V\|^2)$
Basic form with adjacency list	$O((\|E\| + \|V\|) \log \|V\|)$
Binary Heap	$O((\|E\| + \|V\|) \log \|V\|)$
Fibonacci Heap	$O(\|E\| + \|V\| \log \|V\|)$
Self-Balancing Binary Search Tree	$O((\|E\| + \|V\|) \log \|V\|)$

Table 1: Time Complexities of Dijkstra's Algorithm with Different Data Structures

In the context of time complexity for Dijkstra's Algorithm:

- V represents the number of vertices in the graph.
- E represents the number of edges in the graph.

Dijkstra's algorithm with Binary Heap

The first step to optimise Dijkstra's algorithm is to use a Binary Heap as the priority queue instead of an array. This reduces the time complexity of finding the vertex with the minimum distance to $O(log|V|)$ from $O(|V|)$, resulting in an overall time complexity of $O((|E| + |V|)log|V|)$ [7].

Algorithm 2: Dijkstra's Algorithm (Binary Heap)

```
Function Dijkstra(Graph, source):
    create vertex priority queue Q using binary heap;
    for each vertex v in Graph do
        dist [v ] ← INFINITY;
        prev [v ] ← UNDEFINED;
        add v with dist [v ] as key to Q;
    dist [source ] ← 0;
    while Q is not empty do
        u ← vertex in Q with min dist [u ];
        // Extract-Min operation remove u from Q;
        for each neighbour v of u do
            // where v is still in Q alt ← dist [u ] + length (u, v);
            if alt < dist [v ] then
                // A shorter path to v has been found dist [v ] ← alt;
                decrease-key(v, dist [v ]) in Q;
                prev [v ] ← u;
    return dist [], prev [];
```

Dijkstra's algorithm with Fibonacci Heap

Fibonacci heap is a better choice for priority queue in Dijkstra's algorithm as the decrease-key operation can be implemented in amortized O(1) time. The Extract-Min operation takes $O(log|V|)$ time. This leads to an overall time complexity of $O(|E| + |V|log|V|)$. The only difference from algorithm 2 is instead of creating the vertex priority queue Q using binary heap, we use Fibonacci heap. The rest of the code is similar [6].

Dijkstra's algorithm with Balanced Binary Search Tree

Balanced Binary Search Trees like AVL Tree or Red-Black Tree can be used as priority queue. Both the Extract-Min and decrease-key operations can be implemented in $O(log|V|)$ time. This also leads to an overall time complexity of $O((|E| + |V|)log|V|)$. Here, the only difference again is instead of using Binary heap for creating the priority queue Q we use Balanced Binary Search Trees [8].

5 The Experiment

In this experiment, I analyse the performance of Dijkstra's algorithm in various implementations, considering the performance on both sparse and dense graphs. The experiment involves three implementations of the algorithm:

- Basic Form
- Binary Heap
- Balanced Binary Search Tree

Each of these implementations is tested using two data structures: adjacency lists and adjacency matrices. This dual-approach allows for a more comprehensive comparison and performance evaluation of these two common graph representations. While adjacency lists are often considered more efficient for sparse graphs, adjacency matrices can have an advantage in dense graphs. Therefore, both representations are evaluated to provide a thorough analysis.

To analyse the effect of graph density, edges are added to the graph randomly. A uniform random number generator is used to generate a probability for each potential edge. If the generated probability is below a certain threshold (0.1 in this case), the edge is added to the graph. The weight of the edge is also randomly generated within a range from 1 to a maximum weight of 5. This process effectively creates both sparse and dense graphs for the test scenarios.

A series of time trials are conducted to gather quantitative data on the efficiency of each implementation. Each variant of the algorithm is executed on multiple graphs with varying sizes, from 5 up to 1000 vertices, with each size tested 100 times. This methodology allows for observing how each algorithm scales with the input size and how the randomly generated edge weights impact the performance of the algorithm.

For each test, the size of the graph, edge weight, graph density, and the execution time of each algorithm version are recorded. This data is then exported to a CSV file for further analysis.

Unfortunately, due to time constraints and the complexity of other potential data structures such as the Fibonacci heap, their implementations could not be included in this experiment. A preliminary implementation is available in the provided source file, but it was not included in the time trials.

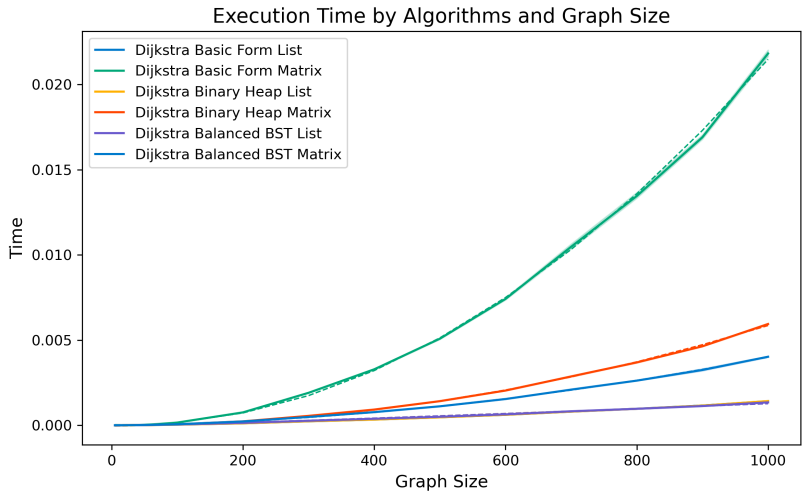

Figure 3: The data was visualised using the Seaborn library, and the theoretical time complexities were predicted with either polynomial regression of degree 2 for adjacency matrices and basic form adjacency lists (reflecting the n^2 complexity), or linear regression on a transformed feature for adjacency lists with Binary Heap or BST (reflecting the $nlog(n)$ complexity). The line styles in the plot differentiate between actual execution times (solid lines) and predicted times based on theoretical complexities (dashed lines). The x-axis represents the size of the graph, while the y-axis denotes time. Each line corresponds to a different implementation of Dijkstra's algorithm, allowing for an intuitive comparison of performance across different graph sizes. The combination of empirical data and theoretical curves provides a comprehensive view of the algorithms' behaviours, illustrating both their performance and complexity characteristics.

This line graph illustrates the CPU execution times for various implementations of Dijkstra's algorithm. I compared these empirical timings with theoretical time complexity approximations calculated via regression analysis , which allows us to observe the practical performance of each algorithm against its theoretical expectations.

It was found that Dijkstra's algorithm, implemented with a Binary Heap and an Adjacency List, exhibits a nearly identical performance profile to the same algorithm implemented with a Balanced Binary Search Tree and an Adjacency List. This observation is in line with their shared theoretical time complexity, as shown in Table 1. These results align with the findings presented in a research paper by R. Lewis [9].

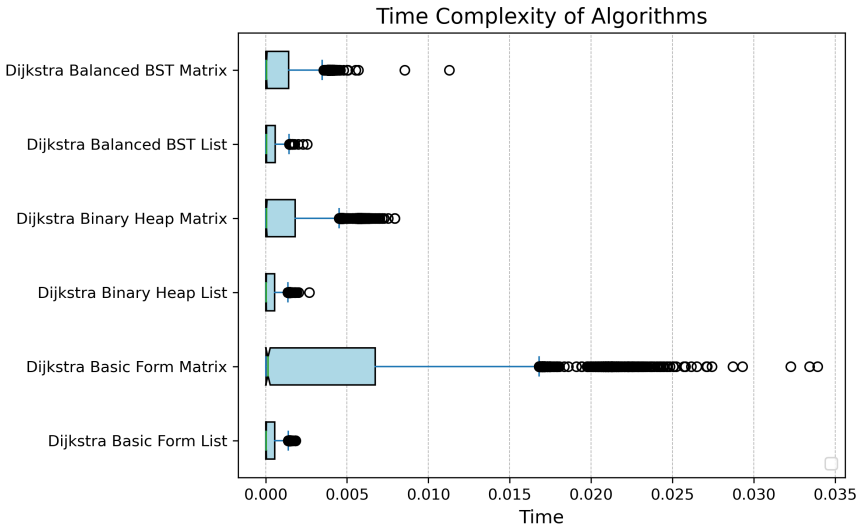

Figure 4: The box plot visualisation above exhibits a comprehensive comparison of the execution times observed for various Dijkstra's algorithm implementations. It's evident that the versions utilising matrix data structures overall demonstrate slower execution times.

The box plot visualisation provides an in-depth perspective on the distribution of execution times across the various algorithms, revealing rather interesting insights that were obscured in the smoothed line plot.

Among these revelations, the striking performance of the Binary Heap List and Balanced BST List stands out. Despite being the fastest implementations in Figure 3, the box plot shows that these two implementations have a larger spread of execution times in comparison to Dijkstra Basic Form List. This indicates the presence of more variability and outliers in their performance, which is probably due to the fluctuating efficiency of heap operations and tree balancing actions depending on the graph's properties. However, their consistently superior average performance emphasises their effectiveness, demonstrating that these implementations not only theoretically, but also practically, outshine others in efficiency.

However, the Dijkstra Basic Form List exhibits a more confined spread, suggesting greater stability in execution time. This stability stems from its straightforward operation, which is less influenced by the variability of the input graph [11]. However, its overall slower execution time affirms its position as a less optimal solution when efficiency is a priority in comparison to Binary Heap List and Balanced BST List.

The performance disparity is even more pronounced when we consider the matrix implementations. The Basic Form Matrix implementation is exceedingly inefficient, owing to its inherent $O(n^2)$ time complexity. The necessity to traverse all possible edges irrespective of their existence in a sparse graph leads to a significant waste of computational resources [11].

In contrast, using advanced data structures like Balanced BST or Binary Heap in combination with the matrix representation considerably mitigates the inefficiency [7,8,11]. These structures allow the algorithm to navigate through the edges more selectively, improving the execution time substantially. However, their performances still trail behind their list counterparts, reiterating the suitability of adjacency lists for sparse graph representation.

These observations reinforce the understanding that the choice of data structure and algorithm implementation significantly affects the performance, and such decisions should align with the nature of the problem at hand.

6 Conclusion

My research looked to conduct an in-depth performance analysis of Dijkstra's algorithm, utilising different implementations and graph representations. The results garnered from this analysis not only consolidate my understanding of the theoretical underpinnings of the algorithm's time complexity, but also shed light on the practical implications of algorithmic design choices in real-world scenarios.

For sure, the performance of Dijkstra's algorithm was found to be highly dependent on the specific implementation and data structure used - we were aware of this coming. The stark contrast in efficiency between the adjacency list and adjacency matrix representations reaffirms the notion that a careful selection of data structure can significantly optimize algorithmic performance, particularly when dealing with large datasets.

My observations showcased the remarkable efficiency of the Binary Heap and Balanced BST implementations when used with adjacency lists, both theoretically and practically. While their performance exhibited a greater degree of variability compared to the Basic Form List, their superior average execution times signified their overall effectiveness. This reiterates the fundamental principle in computer science that the design and choice of data structures can profoundly impact algorithmic efficiency.

However, the Basic Form List showed remarkable stability, despite its generally slower execution times, highlighting that while it may not be the most efficient option, it can be a reliable choice in scenarios where predictability of performance is of utmost importance.

As for the matrix implementations, the inefficiency of the Basic Form Matrix was brought to the forefront. However, the experiment demonstrated that employing advanced data structures such as Binary Heap and Balanced BST can notably improve the execution times even for these implementations, though they still lag behind their list counterparts.

References

[1] Applications of Dijkstra's shortest path algorithm. Retrieved from https://www.geeksforgeeks.org/applications-of-dijkstras-shortest-path-algorithm/

[2] Programiz. Adjacency list. Retrieved from https://www.programiz.com/dsa/graph-adjacency-list

[3] Programiz. Adjacency matrix. Retrieved from https://www.programiz.com/dsa/graph-adjacency-matrix

[4] Graph implementation in C++.
Retrieved from https://www.softwaretestinghelp.com/graph-implementation-cpp/

[5] K. Bailey, "Fibonacci Heaps and Dijkstra's Algorithm - A Visualization"

[6] Fibonacci heap. Retrieved from https://en.wikipedia.org/wiki/Fibonacci_heap

[7] Binary heap. Retrieved from https://en.wikipedia.org/wiki/Binary_heap

[8] Self-balancing binary search tree. Retrieved from https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree

[9] R. Lewis, A Comparison of Dijkstra's Algorithm Using Fibonacci Heaps, Binary Heaps, and Self-Balancing Binary Trees. Retrieved from https://arxiv.org/pdf/2303.10034.pdf

[10] Dijkstra's algorithm. Retrieved from https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

[11] Scaler. Dijkstra's algorithm. Retrieved from https://www.scaler.com/topics/data-structures/dijkstra-algorithm/