Computer Science Department
VU Amsterdam, 1081HV Amsterdam

Bachelor Thesis

On Dijkstra's Algorithm for Single-Source Shortest Paths: An Experimental Study

---

Author: Souriana Shakkour (2658422)

1st supervisor: Ali Mehrabi
2nd reader: Femke van Raamsdonk

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

July 5, 2024

Contents

1 Introduction

In this thesis we do an experimental study on Dijkstra's algorithm for the single-source shortest paths problem. We start with an introduction about graphs, defining the types of graphs used in the study and the terms used in the thesis. We explain the shortest path problem in general and in weighted graphs in particular, then we explain the single source shortest path problem in weighted graphs and explain why Dijkstra's algorithm is the most time efficient algorithm to solve this problem.

We move on to analyzing the algorithm's time complexity when the graph is stored in two different data structures: a matrix and an array of linked lists.

We start the experiment based on the following hypothesis: if the same graph is stored in a linked list and a matrix and given to Dijkstra's algorithm, then the running time should be lower for a linked list than a matrix if the graph is sparse, and the running time for a linked list increases as the graph gets denser, until it reaches a point where it is approximately equal to the running time of a matrix, after this point the running time for both data structures should be almost the same. Our goal is to verify the validity of this hypothesis, and if found to be valid, find the point of density at which these two data structures are almost equally efficient.
Determining the answer to this question is essential for optimizing the performance of Dijkstra's algorithm. It allows us to evaluate the practical applicability of the theory and understand the limitations of the algorithm's implementation for best performance. This experiment could lead to a better understanding of the algorithm's practical performance, enabling us to make more informed decisions about the most convenient data structure, taking into account both time and space efficiency.

In the next sections we describe our dataset and the approach we applied to generate the graphs in this dataset. We discuss the implementation and compare our expected results to the obtained results. We elaborate on the three main factors that affect the running time of the algorithm: the graph's size, its density, and the data structure used to store it. We conclude by proposing a hypothesis that explains the difference between what we expected and what we achieved.

Keywords: Single-source shortest paths problem, Dijkstra's algorithm, data structures for graphs.

2 Preliminaries

Graphs are one of the most fundamental topics in mathematics and computer science and they find applications in a wide range of domains, including network design, transportation systems, biology, and many more. The use of the term "graph" may differ across different fields, therefore, to avoid any confusion, we will go through the terms we used in this paper, providing the definitions and the notations used to refer to them [1, Chapter 26] [2, Chapter 1].

A graph G consists of a pair (V, E), where V is the non-empty finite set of **vertices** of G, and E is the finite set of **edges** connecting its vertices.
A vertex is represented by its name e.g. u.
An edge connecting two vertices $u \in V$ and $v \in V$ is represented by the pair of vertices it connects $(u, v) \in E \subseteq V \times V$.
Vertex v is called a **neighbour** of vertex u if $(u, v) \in E$. The set of neighbours of u is $N(u) = \{v \in V \mid (u, v) \in E\}$.
The **density** of a graph refers to the ratio of the number of edges $|E|$ with respect to the maximum possible edges. The higher the ratio, the **denser** the graph is, and the lower the ratio, the more **sparse** the graph is.

Weighted graphs: A graph could be either weighted or unweighted:

- **Weighted graphs** are graphs where every edge (u, v) is assigned a number called weight $w(u, v)$ using a weight function $w : E \to \mathbb{N}$ which maps every edge to a natural number. For example, if we were to take a graph where the vertices represent cities and the edges represent the roads between them, we can make the graph weighted using a weight function that assigns a weight (natural number) to every road $(u, v) \in E$. The weight could represent, for example, the distance, time or energy required to go from u to v[1].

- **Unweighted graphs** are graphs where edges have no assigned weights. For example, an unweighted graph might be used to represent whether or not there is a path from one city to another, without any further information about that path other than its existence.

2.1 Shortest path problem

The shortest path problem consists of finding the shortest path between two vertices $u \in V$ and $v \in V$. The word "shortest" here does not necessarily refer to the distance but it means "the most efficient". The most efficient path is a path that has no cycles (no repeated vertices), therefore, in a graph with n vertices, the shortest path cannot consist of more than n vertices, thus $n - 1$ edges.
We have two cases:

[1] Note that weighted graphs could contain negative-weight edges, however, in this paper we are focusing of graphs with non-negative-weight edges and will use the term weighted graphs to refer to graphs whose edges are assigned non-negative weights.

2.1.1 Case 1: Shortest path in unweighted graphs

The graph is **unweighted**, for example, a graph of social network where the vertices represent people and an edge between two vertices means that these two people know each other. In this case, the shortest way from person $u \in V$ to person $v \in V$ means the degree of connection between u and v[2]. 1st-degree connections of u is $N(u) = \{v \in V \mid (u, v) \in E\}$ or people who know u. 2nd-degree connections are people who know someone from u's 1st-degree connections and are not u or u's neighbors $N^2(u) = \left\{w \in V \mid w \in \bigcup_{v \in N(u)} N(v) \text{ and } w \notin N(u) \cup \{u\}\right\}$.. etc.

Note that if someone is in different sets of degrees of connections, only the smallest degree counts, hence finding the smallest degree of connection corresponds to solving the shortest path problem which can be done by finding out the minimum number of edges we need to pass through to go from u to v.

2.1.2 Case 2: Shortest path in weighted graphs

The graph is **weighted**, for example, a graph where the vertices are cities, the edges are one-way roads and the weight of an edge $(u, v) \in E$ is the time needed to go from $u \in V$ to $v \in V$ using this road. In this case, solving the shortest path problem from a vertex u to a vertex v is the problem of finding a path P from u to v such that the sum of the weights of edges on P is the minimum among all paths from u to v. In a weighted graph, the number of edges is irrelevant since the goal is to find a path from u to v with the minimum weight. The weight $w(P)$ of a path P consisting of k edges $v_0 \to v_1 \to ... \to v_k$ of the form $e_i = (v_{i-1}, v_i)$, equals

$$w(P) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

2.1.3 Single-source shortest path problem (SSSP) in weighted graphs

The single source shortest path problem, known as SSSP, is one of the central problems in algorithms and it has been studied extensively. Given an edge-weighted graph $G(V, E)$, the SSSP in G is the problem of finding the shortest paths from a designated source vertex s to all other vertices in G[3]. The simple way to solve this problem is as following: We go through all the vertices in the graph, for every vertex i we write down all the possible paths from the source to i, calculate the sum of their composing edges' weights (or the number of their composing edges if the graph is unweighted) and select (the) one with the minimum weight to be the shortest path[4].

This approach, as simple as it sounds, is an inefficient option. For example, in a complete directed weighted graph with $V = \{u, v, x, y, z\}$, we have 16 different paths to go from u

[2] See https://www.linkedin.com/help/linkedin/answer/a545636/your-network-and-degrees-of-connection for more details about degrees of connection.

[3] The problem is also well-defined in unweighted graphs as each edge in an unweighted graph can be assigned a weight of 1.

[4] A shortest path is not always unique, there might be multiple paths that have equal weights.

to y[5] and 64 different path from u to all of the other 4 vertices.
A graph with n vertices could have up to $n(n-1)$ edges, meaning the total number of edges has a quadratic relation to the number of vertices in the graph, making the number of paths grow quadratically as well. For that reason, this method is not efficient for large graphs due to its high time complexity.

2.1.4 Solving (SSSP) in weighted graphs

As alternatives to the previous inefficient approach, there are two well-known efficient algorithms for the single source shortest path problem, namely *the Bellman-Ford algorithm* and *Dijkstra's algorithm*. Both algorithms work for both directed and undirected graphs. The main difference between the two algorithms is that the Bellman-Ford algorithm works for *any* weighted graph, including graphs with negative weighted edges, but Dijkstra's algorithm may fail if a given graph contains negative edge weights. In fact, the Bellman-Ford algorithm can detect if a given weighted graph has a negative cycle, while Dijkstra's algorithm is unable to do so, but this comes with a downside: the running time of the Bellman-Ford algorithm on a graph with n vertices and m edges is in $O(mn)$, but the running time of Dijkstra's algorithm is in $O(m \log n)$, assuming the algorithm is implemented using appropriate data structures. See [1, Chapter 24] for details.

Next we shortly discuss the Bellman-Ford algorithm, for the sake of completeness, and defer the discussion of Dijkstra's algorithm to Section 3, as it is the main topic of this thesis.

The Bellman-Ford algorithm. The algorithm takes three input values, a weighted, directed graph $G = (V, E)$, a weight function $w : E \to \mathbb{R}$, and a source $s \in V$, and has two types of output:

- If the graph has a negative-weight cycle reachable from the source, the algorithm terminates and returns False without returning the SSSP solution since there is no solution in this case (the weight can always decrease by passing through the negative-weight cycle leading to an infinite loop).
- If no such cycle exists, then the algorithm returns True and the SSSP solution. The solution consists of the shortest paths from the source to every other vertex in the graph with their weights.

Algorithm 1 summarizes the description of the algorithm. We note that the pseudocode are taken from [1, Chapter 24]. Since the algorithm has $n-1$ rounds (Line 2) and it relaxes m edges in each round (Line 3), the number of edge relaxations (and thus the running time of the algorithm) is $O(mn)$. See [1, Section 24.1] for more details.

Our contribution. In this thesis we focus on Dijkstra's algorithm for SSSP. In particular, we are interested to learn how the format of a given input graph, that is, given by a matrix

[5] (u, y),
(u, v, y), (u, x, y), (u, z, y),
(u, v, x, y), (u, v, z, y), (u, x, v, y), (u, x, z, y), (u, z, v, y), (u, z, x, y),
(u, v, x, z, y), (u, v, z, x, y), (u, x, v, z, y), (u, x, z, v, y), (u, z, v, x, y), (u, z, x, v, y)

Algorithm Bellman-Ford(G, w, s)

```
1: Initialize-single-source (G, s)
2: for i = 1 to |V| − 1 do
3:    for each edge (u, v) ∈ E do
4:       relax (u, v)
5:    end for
6: end for
7: for each edge (u, v) ∈ E do
8:    if v.d > u.d + w(u, v) then
9:       return false
10:   end if
11: end for
12: return true
```

or given by an array of linked-lists, affects the running time of the algorithm. To this end, we first present a theoretical analysis for the running time of the algorithm when the input graph is given by a matrix and when the input graph is given by linked-lists. The analysis shows that, for sparse graphs, the algorithm is asymptotically faster if the input graph is given by linked-lists, and for dense graphs, the running time of the algorithm is asymptotically equal for both matrix and linked-lists. We then perform a preliminary evaluation of the theoretical analysis through an implementation of the algorithm and running it on different input graphs, each stored in both a matrix and linked-lists. The experimental results (somewhat) confirm the theoretical analysis (up to some level of sparseness and denseness of input graphs).

Organization. The rest of the thesis is organized as follows. In Section 3, we briefly analyze the running time of Dijkstra's algorithm for SSSP, when the input graph is stored in a matrix and when the input graph is stored in linked-lists. In Section 4, we present the details and the results of our experimental study. Finally, we close the thesis in Section 5 with some concluding remarks. We also include the source code of our implementation in Appendix 6.1.

Plagiarism declaration. I confirm that this thesis work is my own work, is not copied from any other source (person, Internet, or machine), and has not been submitted elsewhere for assessment.

3 Dijkstra's Algorithm for Single-Source Shortest Paths

Dijkstra's algorithm offers a solution to the single source shortest paths problem in weighted directed graphs whose edges have non-negative weights.

3.1 The algorithm

The algorithm takes as an input the graph $G = (V, E)$, the weight function $w : E \to \mathbb{N}$, and a source vertex $s \in V$. For each vertex v, the algorithm needs to keep track of whether or not v has been visited, this data is stored in a boolean $v.visited$. If the final shortest path is already found and will not be edited anymore then $v.visited$ is true. Another value that the algorithm needs to keep track of is the length of the current shortest path to each vertex v, which is stored in $v.d$. More precisely, given an edge-weighted (either directed or undirected) graph $G = (V, E)$ with $s \in V$ as the source vertex, Dijkstra's algorithm takes the following steps to compute the weight of the shortest paths from s to all other vertices of G (See 2 for a pseudocode of the algorithm).

1. At the start, the algorithm sets $s.d = 0$ and for every other vertex of G it sets $v.d = \infty$ as the weight of the current shortest path from s to every other vertex v (See 3).

2. All the vertices are unvisited at the start, and a vertex u is visited only when the shortest path from s to u is found (See 4).

3. Extract-min is what makes Dijkstra's algorithm a greedy algorithm. It basically selects from the unvisited vertices the one with the smallest current shortest path to be assigned to u and marked as visited. In the first iteration of the while loop, $u = s$ since $s.d = 0$ and $\forall\ v \in V \setminus \{u\}\ ,\ v.d = \infty$ (See 5).

4. At this point, the shortest path (s, u) is finalized and will not be updated, instead, it is used to find the shortest paths to u's neighbours.

5. In lines 6 and 7, we go through each unvisited neighbour of u, check if their current shortest path's weight could be decreased by going through u, and if so, update the current shortest path's weight to the improved value (the one that goes through u). This step is referred to as *relaxing* the edge (u, v) (See 6).

6. We repeat the steps $3, 4, 5$ until all vertices are marked as visited, and by then for each vertex v its current shortest path $v.d$ is the shortest path from s to v.

Algorithm 2 summarizes the description of the algorithm. We note that the pseudocode is taken from [1, Chapter 24].

Algorithm Dijkstra(G, w, s)

```
1: Initialize-single-source (G, s)
2: Initialize-visited-list (G)
3: while ∃y : y.visited is false do
4:    u = Extract-min (G, Q)
5:    u.visited = true
6:    for each vertex v ∈ N(u) do
7:       Relax (u, v, w)
8:    end for
9: end while
```

Algorithm Initialize-single-source(G, s)

```
1: for each vertex v ∈ G.v do
2:    v.d = ∞
3: end for
4: s.d = 0
```

Algorithm Initialize-visited-list(G)

```
1: for each vertex v ∈ G.v do
2:    v.visited = false
3: end for
```

Algorithm Extract-min(G, Q)

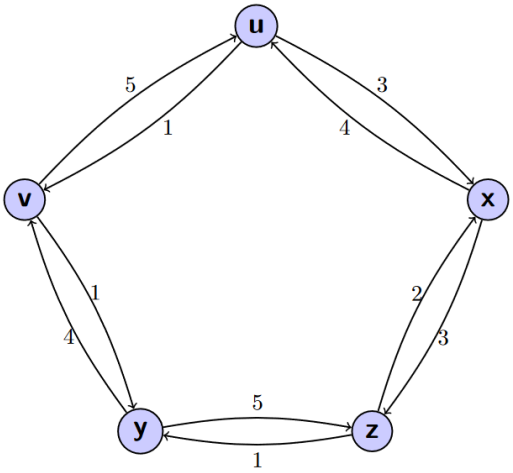
```
1: min = ∞
2: for each vertex v ∈ G.v do
3:    if  v.d < min and v.visited is false  then
4:       min = v.d
5:    end if
6: end for
7: return  min
```

Algorithm Relax(u, v, w)

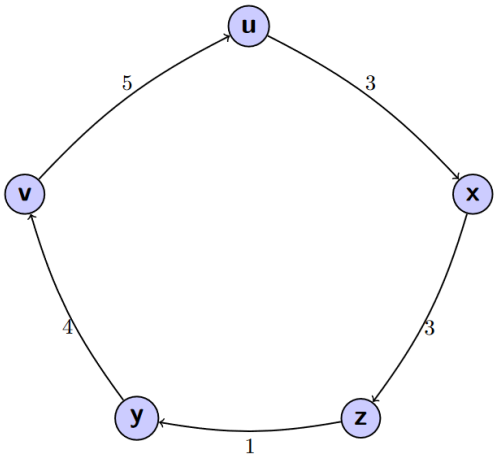
```
1: if v.d > u.d + w(u, v) then
2:    v.d = u.d + w(u, v)
3: end if
```

3.2 Running time for Dijkstra's algorithm

The running time for **Initialize-single-source** for a graph with n vertices is $\Theta(n)$ as it has to update the d value of every vertex in V.



(a) Graph $G : |V| = 5, |E| = 10$



(b) A cycle graph where each vertex has one neighbour

The running time for **Relax** is $O(1)$ since it consists of a simple *if* statement with no iteration or calling functions.
The running time for **Extract-min** is $O(n)$ as it loops over all the vertices, and only updates *min* if the conditional statement returns true.
The running time for line 6 of Dijkstra's depends on the type of the input, specifically, the data structure used to store the graph. We will discuss two different cases:

3.2.1 Running time when the input is stored in a matrix

When a graph is stored as a matrix, each cell (row i, column j) represents an edge from the vertex corresponding to row i to the vertex corresponding to column j. If the value is 0, the edge does not exist, otherwise the value is the weight of the edge. So the matrix representing graph 1a, should look like the matrix to the right[6].

$$\begin{pmatrix} & \text{u} & \text{v} & \text{x} & \text{y} & \text{z} \\ \text{u} & 0 & 1 & 3 & 0 & 0 \\ \text{v} & 5 & 0 & 0 & 1 & 0 \\ \text{x} & 4 & 0 & 0 & 0 & 3 \\ \text{y} & 0 & 4 & 0 & 0 & 5 \\ \text{z} & 0 & 0 & 2 & 1 & 0 \end{pmatrix}$$

We can see that in line 6 of Dijstra 2: "*for each vertex* $v \in N(u)$", $N(u)$ is not known in advance, instead we have a row of the matrix that represents all the edges going from u to each vertex. So for the algorithm to execute line 6, it needs to go through every cell of row 0, check whether or not it is zero, if not (if it is a neighbour of u), it calls *Relax*. We can see in this example that only 2 out of 5 cells in the first row have non zero values, meaning that more than half of the time required to loop over the elements is used to go over non neighbours of u, consuming extra unnecessary time. Therefore, the running time for line 6, in this case, is $\Theta(n)$ for each vertex, hence, $\Theta(n^2)$ for the entire algorithm.

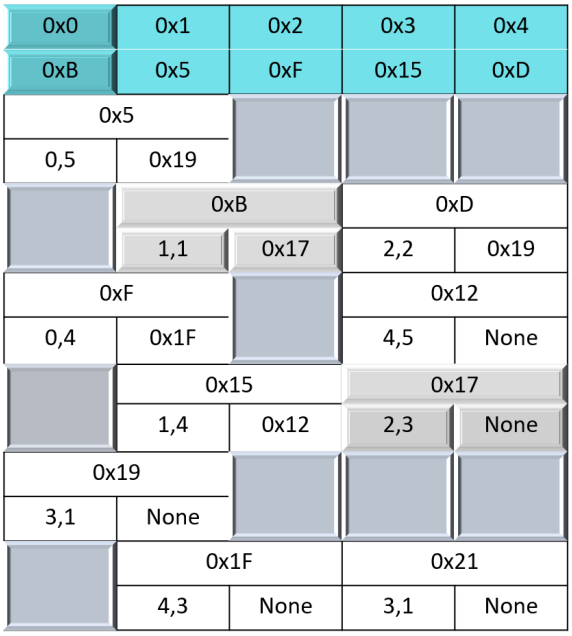
[6] In this paper we focus on positive weighted edges, so a weight of 0 does not appear in the graphs, that is why we chose 0 for the absence of an edge.

3.2.2 Running time when the input is stored in linked lists

In this case we have an array where each element e at index i is a linked list representing the neighbours of the vertex corresponding to i and the weights assigned to the edges from vertex i to each element of the linked list; in other words, element 0 is a linked list, the *data* of every element of the list is a pair, the first value of the pair is a neighbour k of vertex 0 and the second value is the weight of the edge from element 0 to k. For example in the previous graph, if vertices u, v, x, y, z correspond to indices 0, 1, 2, 3, 4 respectively, x is a neighbour of u, and $w(u, x) = 3$, then the element of the array corresponding to u (element 0) is a linked list that has an element whose $data = (2, 3)$.
As is the case for any linked list, the pair $(neighbour, weight\ of\ edge\ to\ neighbour)$ is the *data* stored in the node, while the reference to the next node in the linked list is stored in *next*. The value of *next* in the last element in the linked is $None$[7]. Figure 2b illustrates the structure of the liked list[8], and Figure 2a is a simplified representation of the memory allocation when handling linked lists. In this case, the algorithm does not need to check whether or not an element is a neighbour of u, since only neighbours are stored in the linked list. Here we notice that the time required to run line 6 is not related to the number of vertices anymore, but to the number of edges, because that determines the number of neighbours in the graph. This makes the total running time for the algorithm $= \Theta(|E|)$. For example, in the graph in figure 1b, each vertex has one neighbour, meaning line 6 of the pseudo-code 2 has a 1-iteration loop per vertex, making the total running time $= \Theta(n)$ while for a complete graph, it iterates $n - 1$ times for each vertex, making the running time go up to $\Theta(n^2)$.

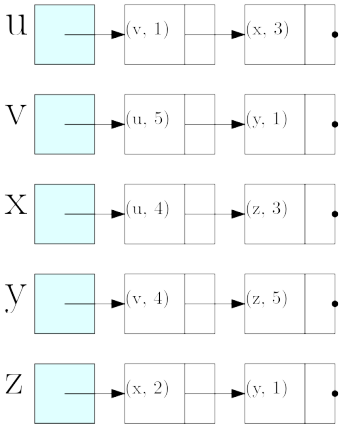
[7]We used arrows in the figure since they are the common way of illustrating linked lists, however, our code implemented the linked lists using references instead of pointers.

[8]The graphs in our dataset had thousands of vertices, therefore we could not use letters in our implementation but had to assign numbers to the vertices. So, as explained in this section, the data in a node of a linked list is a pair of numbers instead of a letter and a number, however, in figure 2b we used letters to make it easier to read and to follow the unified pattern of using letters for vertices and numbers for the weights of edges. Figure 2a shows a more accurate representation of the pairs.



(a) The memory structure for handling graphs from figure 2b. Each cell has *data* and is referenced by its *location*: The bottom part of a cell is the *data*. The upper number is the *location*, white cells represent two memory locations, the *location* here refers to the first one. For linked lists, *data* is split into a *pair* $(vertex, distance)$ and a $reference$ of the next node. Blue cells are memory locations allocated to the array, grey cells are preoccupied memory locations, and white cells are memory locations allocated to linked lists nodes[a].
You can follow the shaded cells for the first element of the array in 2b: $0x0$ stores the reference to the head of the linked list, $0xB$ stores the first node: $(v, 1)$, and $0x17$ stores the second node $(x, 3)$. If $next = None$ that indicates reaching the last element of the linked list.

[a]This is a simplified figure and does not represent the actual size of the data. The actual size of the data differs depending on multiple factors, but in this figure we assume that both references and pairs take one cell each and that one cell is 8 bytes.



(b) The graph from figure 1a stored in an array of linked lists. A pair $(v, 1)$ in the linked list stored in element u of the array represents an edge from u to v with a weight $w(u, v) = 1$.

4 Experimental Results

In this section, we perform an implementation of Dijkstra's algorithm using two different data structures, a matrix (an array of arrays) and linked lists (an array of linked lists). As we have seen previously, the running time for Dijkstra's algorithm when the graph is stored as a matrix is $\Theta(n^2)$ while if stored as linked lists it ranges from $\Theta(n)$ to $\Theta(n^2)$. In this section, we have two implementations of the algorithm, we check the running time for both and try to determine the density of the graph for which the running time for a graph stored in linked lists is almost equal to the running time for a graph stored in a matrix.

4.1 Creating the dataset

As usual in empirical studies we need a dataset that satisfies the requirements. The data set needs to consist of multiple graphs that have the same number of vertices but with different densities. To be able to control our dataset, including the number of vertices and the density of the graphs, we decided to create our own dataset. We start by creating a linear graph. In every step we read the graph from the previous step and add $\frac{n\times(n-1)}{Number\ of\ graphs}$ edges except for the final step which adds the missing edges to complete the graph. The edges are added in order; we start from the first row, we go through every cell and check its value: if it is zero that means there is no edge from the vertex corresponding to this row to the vertex corresponding to this column, we generate a random weight and assign it to the current cell, then move to the next cell. We only stop when we have added the number of edges that need to be added in this step. Every step generates a denser graph than the graph it started with. This process is repeated 3 times for graphs with different sizes: $n = 2500$, $n = 5000$, and $n = 10,000$.

4.2 Expectations

At the start of the experiment we had the following expectations:

1. In a sparse graph, the running time for the linked lists should be noticeably smaller than the running time for the matrix.

2. In a dense graph, the running time for the matrix should be approximately the same as the running time for the linked lists.

3. There is a point at which this switch from 1 to 2 happens and we expect that to be visible in the results, which would answer our question: At which density do these two data structures offer similar running time?

4.3 Discussion

In figure 3 we see the results produced by running the algorithm on the dataset described in Section 4.1. At first glance, the results show three different factors affecting the running time: The size of the graph, the data structure used to store the graph, and the density of the graph.

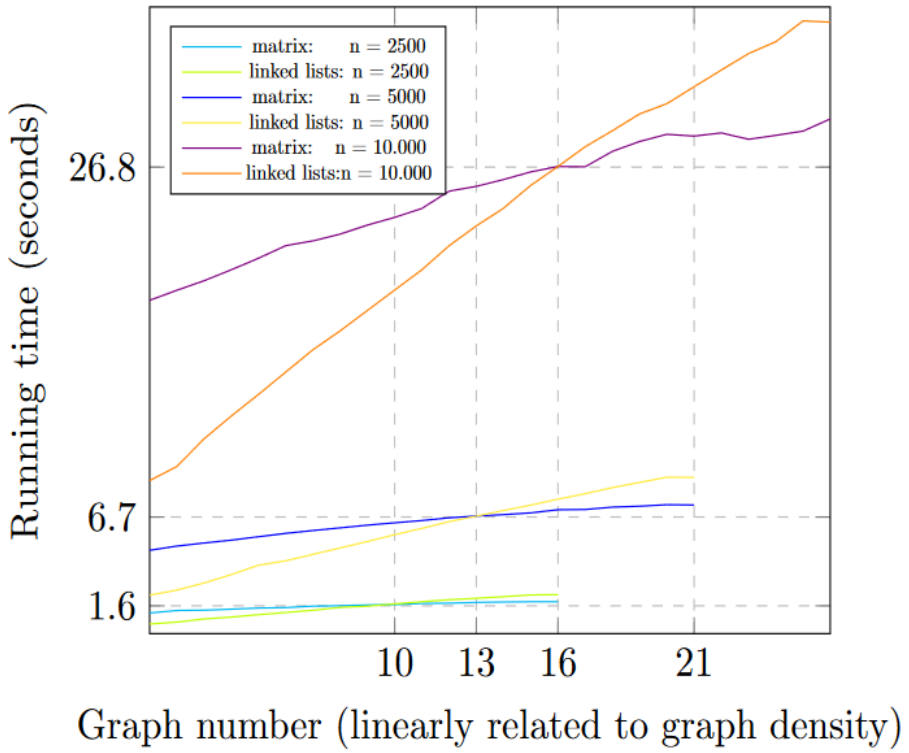


Figure 3: The running time of Dijkstra's algorithm on 16 graphs of size 2500, 21 graphs of size 5000, and 26 graphs of size 10,000 stored in matrices and linked lists.

- We can clearly see that the running time is a function of the number of vertices n, which is visible in all three cases. If we call the running time for a graph with n vertices $f(n)$ then we can say that $f(10,000) > f(5000) > f(2500)$, which gives an empirical verification to the time complexity analysis discussed previously in Dijkstra's algorithm in Section 3.2.

- We also observe an increase in the running time for both data structures as the density of the graph increases, in both cases this was explained by the theory since increasing the number of edges increases the number of neighbours for some vertices, increasing the probability of a shorter path being found in every iteration over the neighbours, thus increasing the probability of a relaxation being performed.

- For the first 60% of the graphs, it is clear that the running time for a linked list is smaller than that of a matrix, which matches our first expectation. When the graph is sparse, for example 10,000 vertices and 9999 edges, the algorithm has to visit $n \times (n-1) \approx 100,000,000$ potential edges in the matrix, while only 9999 of them actually exist, consuming significantly more time than visiting only confirmed edges which is the case for linked lists, which explains the difference in the running time for the two data structures in sparse graphs of the same size.

- The running times get closer to each other due to the extreme increase for linked lists compared to the slow increase for matrices. They reach a point where the running

times become equal for both data structures. This point is the 10^{th} graph when n = 2500 vertices for a running time of $t \approx 1.6$ seconds, the 13^{th} graph when n = 5000 vertices for a running time of $t \approx 6.7$ seconds, and the 16^{th} graph when n = 10,000 vertices for a running time of $t \approx 26.8$ seconds.

- This difference in the rate of increase continues in the last 40% of the graphs as well, where the running time for linked lists exceeds that of a matrix. Our hypothesis is that this is a result of the implementation of a linked list. Our implementation used references to link each node of the linked list to the next node, and while this implementation works in principle, it does not ensure that the memory locations used for adjacent nodes are adjacent themselves. In 2a we can see that the memory locations for array elements are allocated contiguously when the array is created, while for linked lists the memory locations are only allocated when needed leading to them not being adjacent. Even though this should not have been an issue if we were only interested in solving the SSSP problem, it turned out to affect the time performance, as we underestimated the duration of time required to fetch the values from non-adjacent memory locations, compared to the time required to do the same for adjacent memory location, as in the matrices, which explains the difference of running time for the two data structures in dense graphs of the same size.

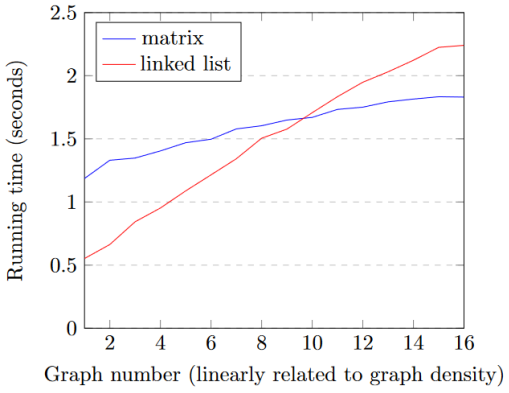


Figure 4: The running time of Dijkstra's algorithm on graphs of 2500 vertices with different densities stored in a matrix (blue) and linked lists (red)

Figure 4 illustrates the running time of graphs with 2500 vertices, when stored in a matrix and in linked lists. The first graph is a linear graph, and the higher the number of the graph the denser it gets, until it reaches a complete graph in the 16^{th} graph. We see a difference in the range of time. The running time for a matrix shows a slight increase as the graph gets denser, while the running time for linked lists shows almost twice the range of time, increasing by about 2 seconds compared to 1 second for a matrix.
The running time for linked lists shows a low start of 0.55 seconds and increases linearly to reach 2.24 towards the end, while the running time for a matrix has a higher start of 1.18 and also increases linearly yet with a lower steepness reaching only 1.8.
Giving the last part of both plots a closer look, we can see that the last 2 graphs have almost equal running times in both plots, which we will also see in larger graphs later. This insignificant change could be explained by the way the complete graph was created. Except for the last graph, every graph in this set was created by adding the same number of edges. The last graph was created by adding as many edges as needed to make the graph complete, which is in most cases only a few edges, a very small number compared to the previous steps where we add hundreds of thousands of edges at a time.

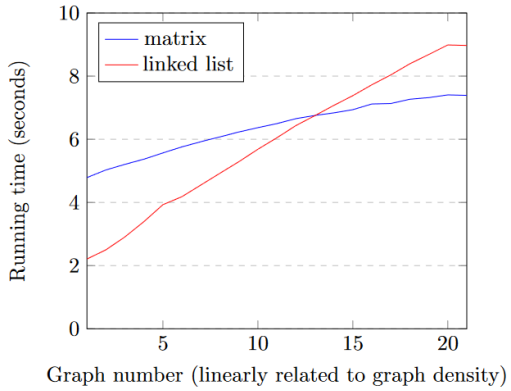


Figure 5: The running time of Dijkstra's algorithm on graphs of 5000 vertices with different densities stored in a matrix (blue) and linked lists (red)

Figure 5 illustrates the running time of graphs with 5000 vertices, when stored in a matrix and in linked lists. The first graph is a linear graph, the density increases in each step, until it reaches a complete graph in the 21st graph.
Just like in the previous plots, we see a difference in the range of time. The running time for linked lists shows more than twice the increase of time as that of a matrix, increasing by about 7 seconds compared to about 3 seconds for a matrix.
The running time for linked lists starts as low as 2.2 seconds and grows to 8.9, while the running time for a matrix has a higher start of 4.8 and an end value of 7.4.
The last 2 graphs have almost equal running times in both plots.

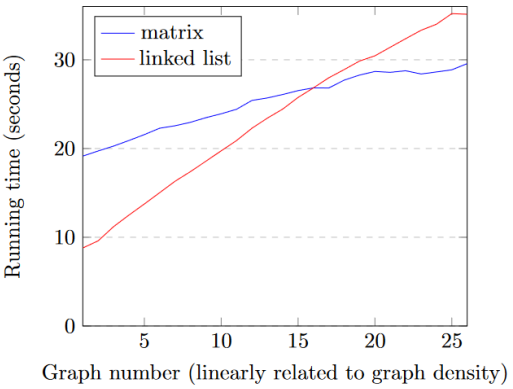


Figure 6: The running time of Dijkstra's algorithm on graphs of 10,000 vertices with different densities stored in a matrix (blue) and linked lists (red)

Figure 6 illustrates the running time of graphs with 10,000 vertices, stored in a matrix and in linked lists. The first graph is a linear graph, and the higher the number of the graph the denser it gets, until it reaches a complete graph in the 26th graph.
Here we notice an even bigger difference in the range of time. The increase of the running time for linked lists is almost 2.5 times that of a matrix, increasing by about 26 seconds compared to 10 second for a matrix.
The running time for linked lists starts at 8.8 seconds and rises to around 35 at the end, while the running time for a matrix starts around 19 and rises slowly to 29.5.

4.4 Further improvement

As we can see in the discussion 4.3, linked lists' running time starts as low as half of that of matrices for the linear graphs, however, their running time increases as the density

increases, making neither of the data structures optimal for both dense and sparse graphs. A possible improvement would be a structure that stores the graph in adjacent memory locations to avoid traversing the memory back and forth which consumed extra time in linked lists for dense graphs. At the same time, it should not go through n vertices if $(size\ of\ N(u)) < n$ to check what vertices are neighbors of u, which consumed extra time in matrices for sparse graphs.

$$\begin{pmatrix} \text{u:} & (v,1) & (x,3) & 0 & 0 & 0 \\ \text{v:} & (u,5) & (y,1) & 0 & 0 & 0 \\ \text{x:} & (u,4) & (z,3) & 0 & 0 & 0 \\ \text{y:} & (v,4) & (z,5) & 0 & 0 & 0 \\ \text{z:} & (x,2) & (y,1) & 0 & 0 & 0 \end{pmatrix}$$

The matrix to the right illustrates the suggested structure. Just like in our implementation of linked lists, an edge (u, v) is stored in the row corresponding to vertex u as a pair of (v, w) where v is a neighbor of u and w is the weight of the edge $w(u, v)$.

However, unlike our implementation for the matrices, in this matrix the columns are meaningless, i.e, column 0 may contain edges to any vertex not only u, and if u has k neighbors, then they are stored in columns 0 to $k - 1$, while columns k to n are given the value 0. Even though this implementation consumes unneeded space, it should, theoretically, be more time efficient since the algorithm can be adjusted to terminate line 6 of Dijkstra's 2 as soon as a zero is found, since zeros are only stored when no more neighbors exist. This termination can save us the time used to go through the zeros in the matrix. The structure of matrices insures that adjacent memory locations are allocated, saving the time needed to traverse the memory.

5 Conclusions

In conclusion, the results of this experiment partially matched our expectations. The running time for matrices started twice as high as that of linked lists in the linear graphs, it increased for both data structures as the density of the graphs increased until it reached a point where the two data structures had equal running times which was after running 60% of the graphs. We expected the two data structures to have almost identical values from here on out, however, this expectation neglected the time required to traverse through the memory, which in turn played a larger factor than predicted, causing the running time for linked lists to exceed that of matrices for the last 40% of the graphs. Based on these results, we can say that linked lists are more time efficient when the graph has a density lower than 60% while a matrix is more efficient if its density is higher than 60%. If the density is around 60% then the two data structures have similar running times.
These results rise the question on the optimal data structure that could be used to improve the running time, and whether a structure that combines matrices with linked lists would give a better overall running time. It also rises the question of what is the optimal implementation of Dijkstra's and what data structure is most efficient when considering other factors, such as space, for both sparse and dense graphs.

References

[1] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd ed. Springer, 2008.

[2] Robin J. Wilson. *Introduction to Graph Theory*. 4th ed. Prentice Hall, 2010.

6 Appendix

6.1 Code

In this appendix, we include the source code of our implementations. We remark that we have borrowed a few parts of the code from the Web, in order to save out times and also avoid re-writing existing codes.
All codes are in *Python*. The first code was run locally, while the other two were run on *Google Colab* for better time accuracy.

The following code calls $createGraph(numVertices, numGraphs, alaramOn)$, which generates a total number of $numGraphs$ graphs, starting with a linear graph with $n = numVertices$ vertices, saving it in a csv file, then in every step it read the previous csv file, generates a matrix of it, densifies it by adding $numNewEdgesPerGraph$ edges and restores the denser graph in a new csv file.
File names have the following form: $numVertices.graphNum.csv$. The higher $graphNum$ is the denser the graph is.

```
import numpy
import random
import winsound

def arrayToCSV (array, csv):
    numpy.savetxt(csv, array, delimiter=',', newline= '\n')

def generateLinearGraphCSV (csv, size):
    e = [[0 for i in range(size)]
         for j in range (size)]
    for i in range(size):
         for j in range (size):
             if i == j-1:
                 e[i][j] = random.randint(1, 10)
    arrayToCSV(e, csv)

def csvToMatrix (fileName):
    arr = numpy.loadtxt(fileName, delimiter=",", dtype=str)
    arrint = [[None for column in range(len(arr))]
                for row in range(len(arr))]
    rowIndex = 0
    for row in arr:
        columnIndex = 0
        for column in row:
            arrint[rowIndex][columnIndex] =
            ↪  float(arr[rowIndex][columnIndex])
            columnIndex += 1
        rowIndex += 1
    return arrint

def densifyCSV (oldCSV, newCSV ,numVertices, numNewEdges):
    if numNewEdges == 0:
```

```
        return
    e = csvToMatrix(oldCSV)
    numAddedEdges = 0
    for row in range (len(e)):
        for column in range (len(e[row])):
            if (e[row][column] == 0 ):
                e[row][column] = random.randint(1, 10)
                numAddedEdges += 1
                if numAddedEdges == numNewEdges:
                    arrayToCSV(e, newCSV)
                    return
    arrayToCSV(e, newCSV)

def createGraph (size, numGraphs, alaramOn):
    fileName = str(size) + ".1.csv"
    numEdgesToAdd = ((size * (size - 1) ) - (size - 1) )
    numNewEdgesPerGraph = numEdgesToAdd // (numGraphs - 1)
    generateLinearGraphCSV(fileName, size)
    for graphCount in range(1, numGraphs + 1):
        newFileName = str(size) + "." + str(graphCount+1) + ".csv"
        densifyCSV(fileName, newFileName, size, numNewEdgesPerGraph)
        fileName = newFileName
    if (numEdgesToAdd % numGraphs) != 0:
        densifyCSV(fileName, newFileName, size, numEdgesToAdd % numGraphs)
    if alaramOn == "Y" or alaramOn == "y":
        frequency = 2500
        duration = 1200
        winsound.Beep(frequency, duration)

createGraph(2500, 15, "y")
createGraph(5000, 20, "y")
createGraph(10000, 25, "y")

print ("Done")
```

We ran the following codes on *Google Colab*. They are essentially similar, except for using different data structures, but reading the *csv* file and converting it into a graph was time consuming, therefore, we used multi-threading for the second code to save time.

The following code reads the graphs generated previously and stored in *csv* files, converts them to **arrays of linked lists**, runs Dijkstra's algorithm on them and saves the running time in a *txt* file.
The lines $12-29$ come from https://www.geeksforgeeks.org/python-linked-list/
The lines $31-61$ come from https://www.geeksforgeeks.org/python-program-for-dijkstras-shortest-

```
!pip install numpy
!pip install time
!pip install tqdm

from google.colab import drive
```

```
drive.mount('/content/drive/')

import numpy as np
from tqdm.notebook import tqdm
import time

class Node:
        def __init__(self, data):
                self.data = data
                self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insertAtEnd(self, data):
        newNode = Node(data)
        if self.head is None:
            self.head = newNode
            return
        currentNode = self.head
        while(currentNode.next):
            currentNode = currentNode.next
        currentNode.next = newNode

class Graph():
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [LinkedList() for column in range(vertices)]

    def minDistance(self, dist, sptSet):
        print(self.V)
        min = float('inf')
        print(self.V)
        for v in range(self.V):
            if dist[v] < min and sptSet[v] == False:
                min = dist[v]
                minIndex = v
        return minIndex

    def dijkstra(self, src):
        dist = [float('inf')] * self.V
        if len(dist) != 0:
            dist[src] = 0
        sptSet = [False] * self.V
        for count in range(self.V):
            u = self.minDistance(dist, sptSet)
            sptSet[u] = True
            v = self.graph[u].head
            while v != None:
                dis = v.data[1]
```

```
                neighbour = v.data[0]
                if (sptSet[neighbour] == False and       # not visited
                                dist[neighbour] > dist[u] + int(dis)):
                                ↪  #shorter path found
                    dist[neighbour] = dist[u] + int(dis)
                v = v.next

def csvToLinkedList (fileName):
    arr =
    ↪  np.loadtxt("/content/drive/Shareddrives/dijkstra_bp/final/"+fileName,
    ↪  delimiter=",", dtype=str)
    arrint = [LinkedList() for column in range(len(arr))]
    rowIndex = 0
    for rowIn in tqdm(range(0,len(arr)), leave = True, desc="creating
    ↪  graph", ascii=True, ncols=75, dynamic_ncols=True):
        columnIndex = 0
        row = arr[rowIn]
        for column in row:
            if (float(arr[rowIndex][columnIndex]) != 0):
                arrint[rowIndex].insertAtEnd((columnIndex,
                ↪  float(arr[rowIndex][columnIndex])))
            columnIndex += 1
        rowIndex += 1
        columnIndex = 0
    return arrint

def saveRunningTime (csv):
    g = Graph(0)
    g.graph = csvToLinkedList(csv)
    g.V = len(g.graph)
    t0 = time.perf_counter()
    g.dijkstra(0)
    t1 = time.perf_counter()
    total = t1-t0
    file =
    ↪  open("/content/drive/Shareddrives/dijkstra_bp/final/time.txt","a")
    file.write(str(csv) + " time for linked list is: " + str(total) + '\n')
    file.close()

def runForGraphs (numGraphs, numVertices):
    for i in tqdm (range (1, numGraphs+ 1),
                   desc="Loading...",
                   ascii=False):
      saveRunningTime(str(numVertices) + "." + str(i) + ".csv")

runForGraphs (16, 2500)
runForGraphs (21, 5000)
runForGraphs (26, 10000)

print ("Done")
```

The following code reads the graphs generated previously and stored in *csv* files, converts them to **matrices**, runs Dijkstra's algorithm on them and saves the running time in a *txt* file.
The lines 14–38 come from https://www.geeksforgeeks.org/python-program-for-dijkstras-shortest-

```
1  !pip install numpy
2  !pip install time
3  !pip install tqdm
4
5  from google.colab import drive
6  drive.mount('/content/drive/')
7
8  import numpy as np
9  import time
10 import concurrent.futures
11 import os
12 import tqdm
13
14 class Graph():
15
16     def __init__(self, vertices):
17         self.V = vertices
18         self.graph = [[0 for column in range(vertices)]
19                       for row in range(vertices)]
20
21     def minDistance(self, dist, sptSet):
22         min = float('inf')
23         for v in range(self.V):
24             if dist[v] < min and sptSet[v] == False:
25                 min = dist[v]
26                 minIndex = v
27         return minIndex
28
29     def dijkstra(self, src):
30         dist = [float('inf')] * self.V
31         dist[src] = 0
32         sptSet = [False] * self.V
33         for cout in range(self.V):
34             u = self.minDistance(dist, sptSet)
35             sptSet[u] = True
36             for v in range(self.V):
37                 if (self.graph[u][v] > 0 and sptSet[v] == False and dist[v]
                    ↪  > dist[u] + self.graph[u][v]):
38                     dist[v] = dist[u] + self.graph[u][v]
39
40 def readChunk(filename, start, end):
41     with open("/content/drive/Shareddrives/dijkstra_bp/final/"+filename,
       ↪  'r') as f:
42         if start > 0:
43             f.seek(start)
44             f.readline()
```

```
        lines = []
        currentPos = f.tell()
        while end is None or currentPos < end:
            line = f.readline()
            if not line:
                break
            lines.append(line)
            currentPos = f.tell()
    return ''.join(lines)

def convertChunk(data):
    from io import StringIO
    return np.genfromtxt(StringIO(data), delimiter=",", dtype=float)

def loadAndConvert(fileName):
    fileSize = os.path.getsize("/content/drive/Shareddrives/dijkstra_bp/final/"+fileName)
    numThreads = 4
    chunkSize = fileSize // numThreads-1
    with concurrent.futures.ThreadPoolExecutor(max_workers=numThreads) as executor:
        readFutures = []
        for i in range(numThreads):
            start = i * chunkSize
            end = (i + 1) * chunkSize if i < numThreads - 1 else None
            readFutures.append((i, executor.submit(readChunk, fileName, start, end)))
        chunks = [None] * numThreads
        for i, future in tqdm.tqdm(readFutures, total=numThreads, desc="Reading Chunks"):
            chunks[i] = future.result()
        processFutures = [executor.submit(convertChunk, chunk) for chunk in chunks]
        arrays = [None] * numThreads
        for i, future in enumerate(processFutures):
            arrays[i] = future.result()
    combinedArray = np.vstack(arrays)
    return combinedArray.tolist()

def saveRunningTime (csv):
    g = Graph(0)
    g.graph = loadAndConvert(csv)
    g.V = len(g.graph)
    t0 = time.perf_counter()
    g.dijkstra(0)
    t1 = time.perf_counter()
    total = t1-t0
    file = open("/content/drive/Shareddrives/dijkstra_bp/final/mx_time.txt","a")
    file.write(str(csv) + " time for matrix is: " + str(total) + '\n')
    file.close()
```

```
90
91 def runForGraphs (numGraphs, numVertices):
92     for i in range (1, numGraphs + 1):
93         saveRunningTime(str(numVertices) + "." + str(i) + ".csv")
94
95 runForGraphs (16, 2500)
96 runForGraphs (21, 5000)
97 runForGraphs (26, 10000)
98
99 print ("Done")
```

6.2 Software and tools

Figures 1a and 1b were created using *tikZ* package in *Latex*, with the help of the example in the following link https://tex.stackexchange.com/questions/456751/how-to-draw-two-separate-d

Plots 3 4 5 and 6 were created using *tikZ* package in *Latex*, with the help of the 2D plot example of *Plotting from data* in the following link https://www.overleaf.com/learn/latex/Pgfplots_package.

Figure 2a was created using PowerPoint.

Figure 2b was created using The Ipe extensible drawing editor https://ipe.otfried.org/.