Implementation of NWB and Comparison with Dijkstra and Bellman-Ford

Faculty of Informatics and Mathematics
at the Hochschule München

Masterarbeit

to obtain the academic degree
Master of Science

submitted by

Marcel Hofbauer
born on 07.08.1998 in Munich
degree program: Master Informatic
student ID number: 33381418

in September 2024

First examiner: Prof. Dr. Güdemann Matthias
Second examiner: Prof. Dr. Christoph Böhm

statutory declaration

Hereby I confirm that I have written this thesis independently and only using the sources and aids indicated by me. Both content and literal citations taken from other sources have been marked as such. This thesis has not been submitted to any other examination board in this or a similar form.

date: München, 24.09.2024 signature: M. Hofbauer

abstract

This master thesis provides an in-depth explanation of the implementation of a newly developed algorithm aimed at solving the classic negative-weight single-source shortest path (SSSP) problem in near linear time, alongside an evaluation of the implementation. It introduces the first open-source implementation of the NWB algorithm, named after its creators Nanongkai, Wulff-Nilsen, and Bernstein. The thesis explores the performance of the implementation and identifies areas for future enhancement. The implementation strictly follows the methodology outlined in the original research paper and has been tested using a variety of example graphs. While it produces correct results, the current implementation lacks the efficiency required for large-scale networks. With further optimizations, this algorithm holds the potential to solve the SSSP problem involving negative-weight edges in near linear time, offering a simple yet effective approach.

Contents

1 Introduction

The shortest path problem is a fundamental concept in the field of graph theory and computer science. There are different type of shortest path problems, to distinguish from. The SSSP problem, in which we have to find the shortest paths from a source vertex u to all other vertices of the graph. The single-destination shortest path problem, in which we have to find the shortest paths from all vertices in the directed graph to a single destination vertex v. This can be reduced to the SSSP problem by reversing the arcs in the directed graph. The all-pairs shortest path problem, in which we have to find the shortest paths between every pair of vertices u, v in the graph.[1] The k-shortest paths problem, in which all k-th best shortest paths are computed.[2] That means the best, second best, third best until the k-th best shortest path will be computed. This is just mentioned for completion purposes. In this master thesis, we focus our attention on the single-source shortest-path problem.

One of the earliest documented instances of the shortest path problem is related to the work of Leonard Euler in 1735. Euler's studies on the Seven Bridges of Königsberg laid the groundwork for graph theory.[3] The modern formulation of the shortest path problem, however, can be attributed to the mid-20th century, with contributions from scientists such as Edsger Dijkstra, who introduced the well-known Dijkstra algorithm in 1956.

Dijkstra's algorithm is a cornerstone in the study of shortest paths. It utilizes a greedy approach to efficiently find the shortest path from a single source node to all other nodes in a weighted graph with non-negative edge weights. The Bellman-Ford algorithm is also an important algorithm to evade the non-negative edge weight restriction. The algorithm considers in every iteration all edges to find the shortest path. If the shortest path does not change in an iteration, then the shortest path has been found. These two remain the most popular algorithms for teaching and practical application to this day.

2 Related work

This chapter provides an overview of the significant works and advancements that have been made in the area of the shortest path problem. Over the years, many algorithms and methodologies have been developed to solve various forms of the shortest path problem, each with its own strengths and applications. This chapter reviews significant contributions and advances in this field. The foundational algorithms for the shortest path problem were established in the mid-20th century, beginning with the Dijkstra and Bellmann-Ford algorithm, as mentioned in the introduction.(1)

Floyd-Warshall algorithm, another classical approach invented by Robert Floyd and Stephen Warshall in 1962, computes the shortest paths between all pairs of vertices. This algorithm, with a time complexity of $O(V^3)$, is optimal for dense graphs and is useful in scenarios requiring complete distance matrices.[4] With the rise of large-scale and real-time applications, exact algorithms often become computationally impossible. This has led to the development of heuristic and approximation algorithms.

The A* search algorithm, introduced by Peter Hart, Nils Nilsson, and Bertram Raphael in 1968, combines the principles of Dijkstra's algorithm with heuristic methods to improve efficiency. By incorporating a heuristic function that estimates the cost of reaching the goal. The algorithm significantly reduces search space, making it ideal for applications in AI and robotics.[5]

The Johnson´s algorithm is another algorithm, developed by Donald B. Johnson in 1977, that efficiently solves the all-pairs shortest path problem for sparse graphs. By reweighting the graph's edges to eliminate negative weights, we then apply Dijkstra's algorithm.[6] The k-shortest path problem, shortly mentioned in 1, requires specialized algorithms to solve. Algorithms such as Yen's algorithm from 1971 and Eppstein's algorithm from 1997 address this need with varying efficiency and complexity.[7, 8]

Another notable contribution is the work on dynamic shortest path algorithms, where algorithms like the Dynamic Dijkstra proposed by Ramalingam and Reps in 1996 adapt to changes in the graph, such as edge weight updates or topology modifications.[9] These dynamic algorithms are particularly relevant for applications in dynamic and evolving networks, such as traffic networks.

The shortest path problem has seen significant developments from classical algorithms to modern approaches that address efficiency and dynamism in large-scale and dynamic graphs. The applications of these algorithms continue to grow, while classical algorithms provide a strong foundation. Future research is likely to focus on further enhancing the efficiency and applicability of these algorithms in ever-more complex and dynamic environments.

3 Motivation of Master Thesis

All state-of-the-art algorithm for solving the negative-weight SSSP are largely based on min-cost flow algorithms and therefore depend on complex optimization methods, dynamic algebraic techniques, and advanced graph algorithms.[10, 11] This master thesis aims to present an implementation and evaluation of a tailored negative-weight SSSP algorithm, designed to address this problem specifically. The focus of this work is to implement an algorithm that operates in near-linear time while maintaining a straightforward and accessible implementation.

The exploration of this problem has both academic and practical relevance. Academically, it contributes to a deeper understanding of theoretical foundations such as computational complexity and algorithm analysis. It also allows for the exploration of the algorithm's limits and capabilities, including the handling of special cases like negative cycles. The straightforward implementation offers an algorithm that is easy to teach, making it an excellent tool for illustrating key concepts like low-diameter decomposition.

The practical applications of shortest path algorithms span various domains such as transportation, networking, and robotics. In transportation, shortest path algorithms optimize route planning and navigation systems, as illustrated in works like „Route Planning in Transportation Networks“ written by Bast, Delling, Goldberg just to name a few.[12] In computer networking, these algorithms aid in efficient routing protocols to minimize latency and maximize throughput.[13] In robotics, algorithms such as Dijkstra's and A* are integral to path planning for autonomous systems.[14]

This master thesis focuses on the implementation of the new NWB algorithm, specifically in the context of graphs without negative cycles. The following chapters will discuss the basic terminology needed to understand the concepts employed in the algorithm, followed by a detailed description of its implementation. Subsequently, an evaluation of the implementation will be conducted, including comparisons with the Dijkstra and Bellman-Ford algorithms. Finally, the thesis will conclude with a discussion of the evaluation results and potential possibilities for further development.

4 Foundations

The following chapter provides an overview of the relevant definitions and concepts necessary for understanding the subsequent chapters.
Let us start with the definition of a graph and how it is structured. A graph consists of a set of vertices V and a set of edges E, for which we write G = (V, E). V(G) describes the set of vertices of graph G and E(G) is the same with the edges. Let n be the size of V(G) and m the size of E(G). We have an edge $e \in E$ and $u, v \in V$. If e connects u and v, we define $e = (u, v)$. In this case, vertex u and v are said to be adjacent and edge e is said to be incident with vertices u and v, respectively. Each edge e can be associated with a real number w(e), which is called its weight. This is called a weighted graph.
In this master thesis, if an edge $e = (u, v)$ exists, we define u as source and v as target. In graph theory, there exist two types of graph: directed and undirected. In an undirected graph, if an edge $e = (u, v)$ exists, then the edge $e = (v, u)$ also exists. In the SSSP the most general setting is with directed graphs and that will be our focus. In a directed graph, edges are in order. $e = (u, v)$ is different from $e = (v, u)$.[15] A vertex v in V(G) is reachable from u, if there is a directed (u,v)-path in G. A directed path in a directed graph is a sequence of edges which joins a sequence of distinct vertices. These edges are all distinct and are pointing in the same direction.[16] We define $dist_G(u, v)$ to be the shortest distance from u to v.[10] The next definitions describe more advanced concepts, which are used in the new SSSP algorithm. A directed acyclic graph (DAG) is a directed graph without any directed cycle.[10] A cycle consists of a sequence of adjacent and distinct nodes in a graph. The only requirement is that the first and last nodes of the cycle sequence must be the same node.[17] In a weighted graph a cycle C can be negative, if the sum of the weights of the edges in the graph are negative.[10] A directed graph is strongly connected, if there exists a directed path between every pair of distinct vertices in both directions. That means, if a path from u to v exist, then exist a path from v to u. Strongly connected components (SSCs) of a directed graph G are subgraphs of G, where every subgraph is strongly connected. The neighbors of a vertex v are all vertices that are adjacent to v. Because they are directed, there is a distinction of direction. This can be formulated precisely using mathematical notations.

$$N_{in}(v) = \{u \in V(G) | v \neq u, \exists e \in E(G) : e = (u, v)\} \tag{4.1}$$
$$N_{out}(v) = \{u \in V(G) | v \neq u, \exists e \in E(G) : e = (v, u)\} \tag{4.2}$$

The set of neighbors N(v) is the union of N_{in} and N_{out}. The number of edges entering a vertex $v \in V$ is called the in-degree $\delta_{in}(v)$ of v and the edges exiting v is called out-degree $\delta_{out}(v)$. Constant out-degree means that all vertices in a graph have the same number of

outgoing edges. A graph H is a subgraph of G if V(H) is a subset of V(G) and E(H) is a subset of E(G) such that for all edges $e \in E(H)$ with $e = (u, v)$, we have u,v in V(H).
Given a subset V^* of V(G). The subgraph induced by V^* is $G_{induced}(V^*,E^*)$, where E^* contains all edges, which connects two vertices in V^*. A price function can be any function that projects $V \rightarrow Z$, where Z is a set of integers, with a weight function $w_\phi(u, v) = w(u, v) + \phi(u) - \phi(v)$. Given any graph G = (V, E, w). w^B is a weight function defined as $w^B(e) = w(e) + B$ for all negative edges in G. A graph G^B_ϕ is a graph with the weight function w^B and a price function ϕ was applied.[10]

5 Implementation of NWB

The algorithm is implemented in C++ and utilizes the Boost Graph Library for the graph representation. When selecting a library for this purpose, I considered three prominent C++ libraries: Boost Graph Library, LEMON and CGAL. The Boost Graph Library is designed to provide a comprehensive set of tools for graph data structures and algorithms. The library is header-only and does not need to be built to be used.[18] LEMON stands for Library for Efficient Modeling and Optimization in Networks. It is a C++ template library providing efficient implementations of common data structures and algorithms with focus on combinatorial optimization tasks.[19] CGAL is an open source software project that provides easy access to efficient and reliable geometric algorithms in the form of a C++ library.[20] In the end, I choose the Boost Graph Library for its ease use and extensive documentation, which facilitated the implementation of the algorithm.
The „main.cpp“ file contains a function named get_graph, which accepts a number and a file path as arguments. The file path, which can be passed as a parameter to the main function, should point to a .gr-file. This file must follow a specific structure: each line should specify an edge with a source, target, and weight, prefixed by the letter „a“ to denote an edge. The function returns the graph as specified in the .gr-file, provided the file path and structure are correct. If the file path is not provided, the function will select one of the example graphs that are hard-coded into the program. Currently, there are 15 such example graphs available. The function throws an error, if the number does not select any of the graphs.

5.1 Timer class

The Timer class is designed for performance benchmarking. Upon instantiation of a Timer object, the current time is recorded. When the object is subsequently destroyed, the recorded start time is subtracted from the end time, yielding the duration for which the object existed. The method of benchmarking with the Timer class will be explained in the evaluation chapter.(6)

5.2 Dijkstra class

The Dijkstra class is a straightforward implementation that utilizes the Boost Graph Library to compute the Dijkstra algorithm. The class inputs a graph into a function provided by the library. To display the shortest path from a specified source to all vertices, the function iterates over each vertex, storing the predecessors of each node until the source is reached. The vertices are then iterated in reverse order to reconstruct the shortest path.[21] The

algorithm maintains a set of visited vertices and a set of unvisited vertices. It starts at the source vertex and iteratively selects the unvisited vertex with the smallest distance from the source. Then it visits the neighbors of this vertex and updates their distances if a shorter path is found. This process continues until the destination vertex is reached, or all reachable vertices have been visited. The Dijkstra algorithm does not work with negative edge weights. If an array is used to store distances and vertices, the time complexity is $O(V^2)$. This can be reduced to $O(E * log_2(V))$ by using priority_queue to access the vertex with the current smallest distance.[22] According to the Boost Graph Library website, the time complexity of their implementation is $O(V * log_2(V))$.[23]

5.3 Bellman-Ford class

The Bellman-Ford class is, similar to the Dijkstra class, a simple implementation with Boost Graph Library. The graph is passed into a function provided by the library, which returns a boolean value. If it returns true, the shortest paths from one vertex to all others can easily be displayed. If it returns false, it indicates the presence of a negative cycle, which is then reported.[24] The Bellman-Ford algorithm computes the shortest paths from a single source to all other vertices in a graph, even when some edges have negative weights. It initializes the distance of the source as 0 and to all other vertices as infinity. Then, it iterates over each edge repeatedly over n-1 iterations, updating the shortest known distances. After these iterations, the algorithm checks for any negative weight cycles by attempting one more update of the distances. If a shorter path is found, there is a negative cycle. The final result is either the shortest paths or the detection of a negative weight cycle. As an optimization the algorithm ends earlier, if in an iteration was nothing changed. The time complexity of Bellman-Ford is O(n*m) on average and in the best case O(m). O(m) is achieved if there are no distances to update.[25] In the Boost Graph Library, Bellman-Ford has the mentioned time complexity. [26]

5.4 NWB algorithm

The concept behind the new SSSP algorithm involves iteratively transforming negative edges within the graph into positive edges. This transformation is carried out without altering the shortest path in the graph, thereby rendering the graph suitable for the Dijkstra algorithm. This is achieved by calculating the appropriate price functions. To demonstrate that the shortest path $P_{x,y}$ remains unchanged after applying the price function to a graph, consider the following example:

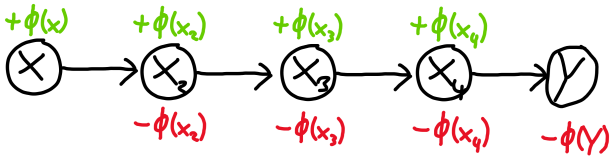

Figure 5.1: Proof price function[27]

Applying the definition of the price function from the last chapter 4 to each edge reveals that most alterations to a path cancel each other out, yielding the result: $w_\phi(P_{x,y}) = w(P_{x,y}) + \phi(x) - \phi(y)$ for every path $x, y \in V$.

To calculate the correct price functions, the algorithm employs two main functions: SPmain and ScaleDown. We are using the following graph to better illustrate the different steps of the functions:

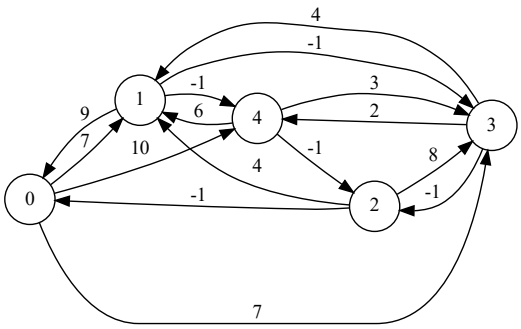

Figure 5.2: Input Graph

The SPmain function accepts the original input graph and a source vertex as its primary input. Additionally, it receives the names and the number of vertices, although these parameters are included only for convenience. The graph must fulfill three conditions:

(a) $w(e) \geq -1$ for all $e \in E$

(b) each vertex must have constant out-degree (4) and

(c) the graph must be free of negative weight cycles.

Initially, the algorithm constructs a graph, denoted as $\overline{G}$ by doubling the edge weights and rounding up 2n to the nearest power of 2, which is stored in a variable named B, to ensure that all values remain integral. The graph would look like this:

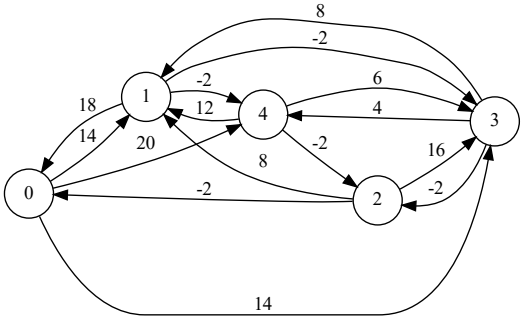

Figure 5.3: doubled weight graph

Iteratively, ScaleDown will be called until a price function ϕ_t is obtained, such that $w_{\phi_t} \geq -1$ holds for all edges. In the final step, the algorithm generates a modified graph G^* with the weight function defined as $w^*(e) = w_{\phi_t}(e) + 1$. This is achieved by first transforming the input graph using w_{ϕ_t}, followed by a transformation with a price function w_{ϕ_1}, where w_{ϕ_1} consists only of ones. The Dijkstra algorithm is then executed on G^* to calculate the shortest paths. A convenient shortcut has been implemented. If the graph lacks negative edges, the Dijkstra algorithm can be applied directly.[10]

The ScaleDown function has three inputs: the graph, which does not contain a negative cycle, Δ and B. The input requirements are as follows:

(a) B is a positive integer, the weights are integral and $w_{\phi_t} \geq -2B$ for all edges

(b) For every $v \in V$ there is a shortest sv-path in G_s^B with at most Δ negative edges

(c) All vertices in G have constant out-degree

G_s^B is a graph with weight function $w_B = w_G + B$, then added a dummy source s. Further explanation comes up in chapter 5.4. Upon satisfying the specified requirements, the algorithm produces an integral price function ϕ such that $w_{\phi_t} \geq -B$ for all edges holds.[10] The ScaleDown procedure can be described into approximately four phases. Prior to initiating these phases, a preliminary check is performed to verify whether $\Delta \geq 2$. If this condition is satisfied, we set $\phi_2 = 0$ and proceed directly to phase 3. Otherwise, if the condition is not met, we proceed by defining $d = \Delta/2$ and constructing the graph $G_{\geq 0}^B$. In this graph, the edge weights are redefined as $w_{\geq 0}^B(e) = \max\{0, w^B(e)\}$ for all edges $e \in E$. The graph $G_{\geq 0}^B$ is needed for the Low-Diameter Decomposition (LDD) process.

Phase 0: Creating strongly connected components (SCCs) with weak diameter

To create SCCs with weak diameter, we employ LDD. The decomposition of a graph is a list of subgraphs such that each edge appears in exactly one subgraph in the list.[28] This method guarantees to return a set of edges E^{rem}, such that each SCC of $G \setminus E^{rem}$ has weak diameter at most D. That means that if u, v are in the same SCC, then the distance $dist_G(u, v)$ between u and v is at most D.
LDD(G, D) operates exclusively on graphs with non-negative edge weights, denoted as $G_{\geq 0}^B$, and aims to achieve a diameter D to equal to dB.

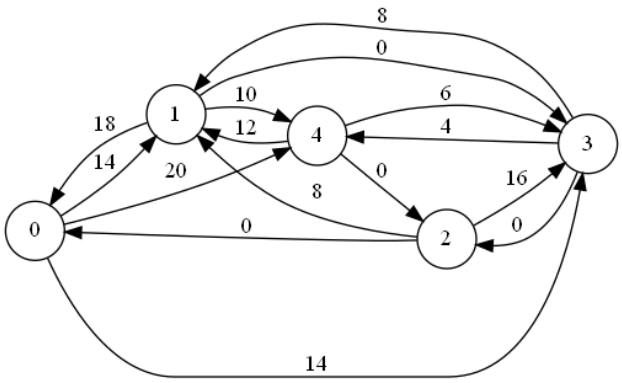

Figure 5.4: $G^B_{\geq 0}$

It is easy to see, by comparing 5.3 with 5.4, that all negative edges were set to 0 while the other edges remained the same.

Before the algorithm starts, we make a copy of the G into G_0. The LDD runs on G. LDD consists of three important phases. In Phase 1, all vertices are categorized as in-light, out-light, or heavy. For every $v \in V$, we compute $Ball^{in}_G(v, R)$ and $Ball^{out}_G(v, R)$ with radius $R = D/4$. We define $Ball^{out}_G(v, R) = \{u \in V | dist(v, u) \leq R\}$, $Ball^{in}_G(v, R) = \{u \in V | dist(u, v) \leq R\}$, alongside a vector S containing k randomly selected vertices from the graph. Here, k is defined as $k = cln(n)$, where c is a large constant.

At this stage, I implemented two different methods to mark the vertices. First, we compute the intersection of S with $Ball^{in}_G(v, D/4)$ and $Ball^{out}_G(v, D/4)$ and verify for each vertex if the cardinal number of the intersection is less than 0.6k. This marking approach ensures, with high probability, that if a vertex v is marked as in-light, then the cardinality of the ball satisfies $|Ball^{in}_G(v, D/4)| \leq 0.7n$. A similar condition applies to out-light marked vertices. The remaining vertices are marked as heavy, ensuring $|Ball^{in}_G(v, D/4)| > 0.5n$ and $|Ball^{out}_G(v, D/4)| > 0.5n$ with high probability. The second method involves directly computing the cardinal number of $Ball^{in}_G(v, D/4)$ and $Ball^{out}_G(v, D/4)$ by constructing the balls and taking their sizes. This approach simplifies the process as it merely requires iterating over the graph to create the balls for each vertex, without the need to compute the intersection if a vertex was not included in S. I choose the second method to use for the evaluation, because it does not need the computation of the intersection with S which makes it the faster approach.

I focused on optimizing this function due to the performance analysis discussed in the next chapter. The initial approach involved recursively iterating over the graph while reducing the radius by the weight of the edges with each recursive call. Let's take the creation of $Ball^{out}_G$ as an example. We start by adding the initial vertex to the ball. Then, we iterate over all outgoing edges from this vertex, extracting the target and its corresponding weight. If the weight is less than or equal to the radius, we recursively call the function with the updated $radius = radius - weight$ on the target vertex. When the edge's weight exceeds the remaining radius, we reach the end of the ball and return the result. To avoid duplicates, we check if the vertices in the returned ball are already included in the current ball. The list, which recorders the visited vertices, is a global variable. After finishing the computation the

list gets emptied. The creation of $Ball_G^{in}$ follows the same logic, except that the recursive calls are made using the source vertices and iterating over incoming edges. Unfortunately, after testing this approach, it fails to detect a shorter path if the path includes outgoing edges for $Ball_G^{in}$or ingoing edges for $Ball_G^{out}$. This issue arises because the current method does not account for the direction of the edges when calculating the balls, which leads to incomplete or incorrect paths being identified.
The second approach involved using Dijkstra's algorithm to find the shortest paths from a single vertex to the others. We add every vertex whose distance is less than or equal to the radius. This approach works for $Ball_G^{out}$. For $Ball_G^{in}$, we reverse the graph, a function provided by the Boost Graph Library, and apply the same procedure. I incorporated multithreading to accelerate the algorithm and allowing multiple balls to be calculated simultaneously. The second approach is used for the evaluation, because it is slightly faster. The multithreading is not considered, because it does not improve the computation speed.
The second phase carves out balls, which have as center a not heavy marked vertex. We start by sampling a radius R_v from the geometric distribution. The geometric distribution Geo(p) is the probability distribution of the number X of independent events executed, until success was reached.[10] The probability is $p = min\{1, 80log_2(n)/D\}$. According to an open paper on Low-Diameter Decomposition, published from one of the main author of the original paper, Nanongkai Danupon, R_v has an upper limit of D/4.[29] This is checked after generating R_v. Unfortunately, it is quite difficult to choose randomly a vertex from a given graph with the Boost Graph Library. The vertex v is drawn from a uniform distribution in range from 0 to $|V(G)|$. Vertex v is checked if it is marked and is still in the graph. If both are true, we compute $Ball^*(v, R_v)$. Now we can compute the boundary edges of $Ball^*(v, R_v)$, which are part of E^{rem}. We define boundary($Ball_G^{out}(v, R)$) $= \{(x, y) \in E \mid x \in Ball_G^{out}(v, R) \wedge y \notin Ball_G^{out}(v, R)\}$ and boundary($Ball_G^{in}(v, R)$) $= \{(x, y) \in E \mid x \notin Ball_G^{in}(v, R) \wedge y \in Ball_G^{in}(v, R)\}$.[10] The function iterates over the graph and adjusts the marking to the correct definition accordingly. The algorithm may terminate at this point if $Ball^*(v, R_v) > 0.7n$ is true, returning all edges in the graph as E^{rem}. Otherwise, we recursively call the LDD on a subgraph containing only the vertices and edges from $Ball^*(v, R_v)$ with diameter D.

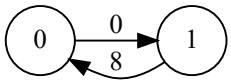

Figure 5.5: Graph created by a ball

The figure 5.5 gives an example of a graph created by a ball with $V = \{0, 1\}$, 0 as center and a diameter $D = 8$. Through this process, we obtain $E^{recurse}$, which becomes part of E^{rem}. The creation of the subgraph in 5.5 presents some problems. For example, when adding the edge (3, 4), the Boost Graph Library adds the edge along with every vertex up to 4. In most cases, this behavior is not required. Instead, it is necessary to first add all vertices from the

ball, followed by adding edges where both the source and target vertices are within the ball. A drawback of this approach is that the vertex indices become incorrect. The original index must be stored in the graph property „vertex_color“ . Every edge added to E^{rem} must be converted to use the „vertex_color“ as the new index.

The last step of phase 2 involves removing $Ball^*(v, R_v)$ from the graph. The removal of a vertex automatically decreases the index by 1. This change is tracked by a variable and accounted for if the ball contains more than one vertex.
In the final phase, called „Clean Up“, we check if the remaining vertices have weak diameter in G_0. We select an arbitrary vertex v in G and check $Ball^{in}_{G_0}(v, D/2) \not\supseteq V(G)$ or $Ball^{out}_{G_0}(v, D/2) \not\supseteq V(G)$. If either condition holds, the edges E(G) are added to E^{rem}. Otherwise, the E^{rem} obtained this far is returned as the result. After obtaining E^{rem} from our LDD, we can create a graph without E^{rem}, denoted by $G \setminus E^{rem}$.

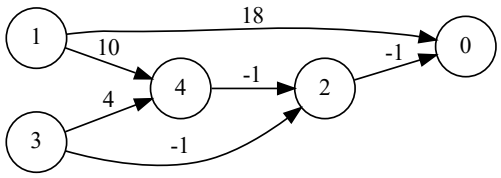

Figure 5.6: $G \setminus E^{rem}$

The randomization of LDD can produce different sets of E^{rem} and consequently alter $G \setminus E^{rem}$. Finally, we compute the SCCs from $G \setminus E^{rem}$ using the Boost Graph Library function „strongly_connected_component“. These SCCs are labeled as V_1, V_2, and so on.

Phase 1: Make edges inside SCCs non-negative

We construct a graph H that contains only the edges within the SCCs, using $G \setminus E^{rem}$ and the SCCs from phase 0. To compute H, we start by copying $G \setminus E^{rem}$ and then remove every edge that does not connect two vertices within the same SCC. After removal of these edges, we iterate over H and eliminate any vertex whose in-degree plus out-degree is zero. This step is necessary to ensure that the graph does not contain vertices that are disconnected. A vertex is disconnected, if there exist no path to any other vertex in the graph.
If we look at figure 5.6, it is evident that there are no SCCs in the graph. According to the definition, the graph H is empty. If it were not empty, it would be necessary to reindex the graph, as mentioned in 5.4. To obtain the first price function ϕ_1, we recursively call ScaleDown with H, $\Delta/2$ and B. The price function ϕ_1 makes all edges inside SCCs non-negative.

Phase 2: Make the edges between SCCs non-negative

We compute the procedure FixDAGEdges, which produces a price function that ensures non-negative edges between SCCs. The algorithm takes as input a graph G and a partition P, which contains the SCCs, such that

1. for every i, the induced subgraph $G[V_i]$ contains no negative weight edges and
2. when we contract every V_i into a vertex, the resulting graph is a DAG.[10]

First, we construct the required graph. We take G^B, apply ϕ_1 to create $G^B_{\phi_1}$ (4) and removing E^{rem} from it, which concludes into the graph $G^B_{\phi_1} \setminus E^{rem}$. This graph fulfills the first requirement, because of the transformation with ϕ_1.(5.4)
For the second requirement, we contract every SCC into one node. This automatically results into a DAG.
The procedure for contracting every SCC into a single vertex is straightforward. As mentioned in Chapter 5.4, the Boost Graph Library provides a method for identifying SCCs. If the graph only contains one SCC, then we add a vertex and return the contracted graph. If the graph contains multiple SCCs, we iterate over the edges of the graph, checking whether the source and target vertices belong to the same SCC. We add an edge only if the source and target are in different SCC, retaining the original edge weight.
The partition P is a vector that stores V_1, V_2, etc. From the contraction process, we determine which vertex belongs to which component and store this information in a vector. Finally, we can call FixDAGEdges with the contracted graph and the partition $P = \{V_1, V_2, ...\}$.

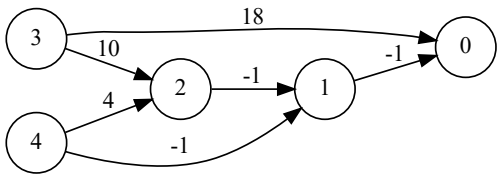

Figure 5.7: transformed and contracted graph

The algorithm is relatively simple. We iterate in a simple loop over the vertices in topological order and set $\phi(v_i)$ for each $v_i \in V_i$, if the incoming edge to V_i has a negative weight. To iterate over the vertices in topological order, we must relabel the SCCs in P, such that for any edge $(u, v) \in E$ with $u \in V_i$ and $v \in V_j$, it holds $i \leq j$.[10] This task is simplified by the Boost Graph Library´s „topological_sort“ function, which returns the vertices in reverse topological order. To obtain the correct order, we iterate from the end, adding the vertices in the proper sequence. The graph is initially oriented backwards, so we correct this by iterating over the edges e = (u, v) and adding them correctly to a new graph. If $u < v$, then add the edge (u, v); otherwise, we add the edge (v, u).

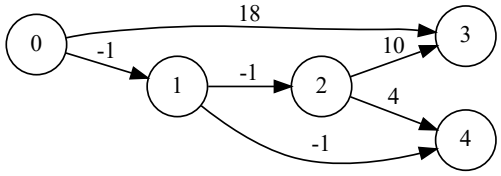

Figure 5.8: relabeled graph

We then iterate over each partition, starting with the second one, identifying the most negative edge weight, denoted u_j, entering the partition and accumulating these weights into a variable M_j. It is sufficient to begin with the second partition, due to the topological sort and the DAG, there are no edges entering the first partition. Consequently, M_1 and u_1 are set to 0. Once u_j is identified and added to M_j, we assign $\phi(v) = M_j$ for every v in the current partition.
The output is a price function ψ, that makes all edges in between SCCs non-negative. To acquire a price function ϕ_2, which satisfies the requirements of Phase 1 and Phase 2, we add ϕ_1 and ψ element-wise.

Phase 3: Make the edges in G^B non-negative

We add a dummy source s to G^B, forming the graph G_s^B. A dummy source s is a vertex that has an edge with weight zero to every other vertex in the graph and no incoming edges.[10] After adding s, we define the price function $\phi_2(s) = 0$. At this point, we apply ϕ_2 to G_s^B. It is crucial to follow this order of operations: first, add s, then transform the graph using ϕ_2. If we first apply ϕ_2, then adding s, the edges of s would always be zero and the price function would contain only zeros.

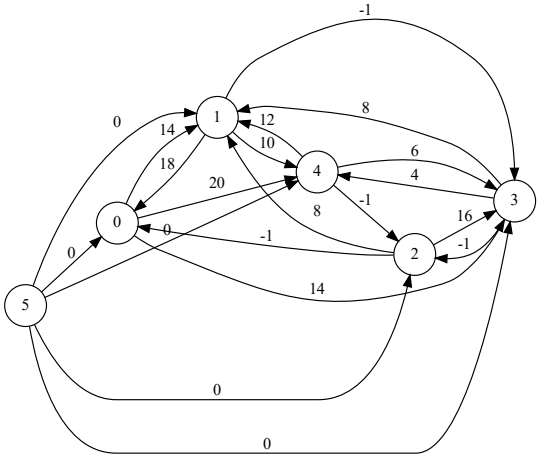

Figure 5.9: added dummy source

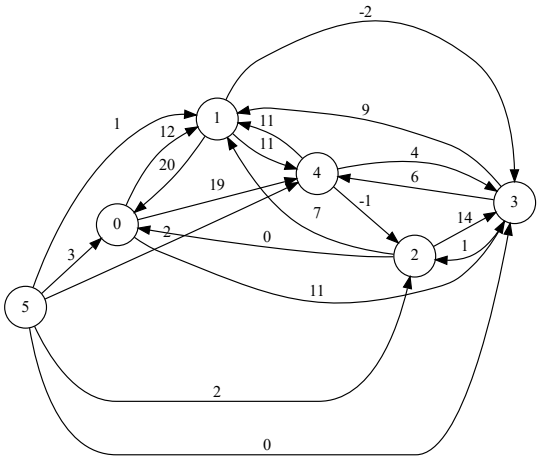

Figure 5.10: transformed with ϕ_2

With the preparations completed, we can call the ElimNeg algorithm on the transformed G_s^B with the dummy source s. All necessary conditions are satisfied: all vertices, except for s, have a constant out-degree and s can reach all vertices in the graph. As before, the presence of a negative weight cycle is not possible. If such a cycle existed, the algorithm would not terminate.
Initially, we set the distance of s, denoted $d(s)$, to zero, and for all other vertex, we set $d(v) = \infty$. We then initialize a queue Q as a vector and add s to it. Additionally, we need a vector to track the marked vertices.
Within a while loop, we alternate between a Dijkstra and Bellman-Ford phases until Q is empty. In the Dijkstra phase, we look for the vertex v with the minimum distance $d(v)$ and mark it. For each outgoing edge $(v, x) \in E \leq E^{neg}(G)$, we check if there is a shorter distance to x using $d(v) + w(v, x) < d(x)$. If this condition holds and $x \notin Q$, we add x to Q and update $d(x)$ with the shorter distance. Finally, we extract v from Q.
During the Bellman-Ford phase, we iterate over the outgoing edges $(v, x) \in E(G)$ from the marked vertices. While it suffices to consider only $(v, x) \in E^{neg}(G)$, as pointed out in 5.4, it is challenging to create subgraphs, so we iterate over all outgoing edges from vertex v. We perform the same final check as in the Dijkstra phase: if a shorter distance exists and x is not in Q, we add x to Q and update the distance. Once all outgoing edges from v are checked, we unmark v.
When Q is empty, we have determined the correct distances d and can return them as a price function.
I implemented the same procedure using a priority queue as the data structure. A priority queue is a type of queue in which each element is associated with a priority and ordered accordingly.[30] The priority queue is modeled as a min-heap, where the smallest element is at the top. The priority queue stores pairs of vertices and their current distances $d(v)$, ordering the pairs by minimum distance. This makes extracting the minimum distance constant in time complexity, although we still need a set to check if vertex x is inside the queue,

as the priority queue structure does not facilitate this check easily. For the results I used a vector to store the distances. The priority_queue did not run faster, because of the check if a vertex is inside the queue.
After adding ϕ_2 and the calculated distances, we created a price function which makes the edges in G^B non-negative.

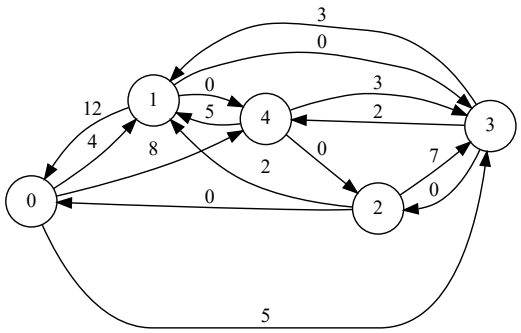

Figure 5.11: transformed with ϕ_t

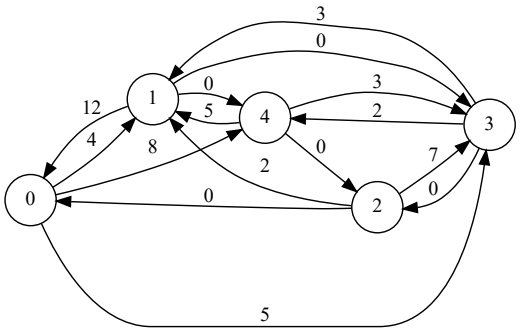

Figure 5.12: transformed with $\phi_t + 1$

As depicted in figure 5.11, both graphs no longer contain any negative edges and are now suitable for use as input for Dijkstra's algorithm. The paper argues that since we are working with a scaled graph resulting from the doubling of the input graph, adding one is insignificant.[10] The pictures support this claim, as they are identical.

6 Evaluation and Comparison with Dijkstra and Bellmann-Ford

This chapter presents an explanation on the correctness and robustness of the implemented algorithm. Furthermore, an analysis of the performance of the three algorithm and a comparison will be discussed.

The correctness of the new NWB algorithm was verified through two approaches. First, small graphs with manually computable shortest paths were used, and the algorithm's output was compared against these known results. Second, the algorithm was tested against established implementations like Dijkstra's algorithm for graphs with positive edges and Bellman-Ford for graphs with negative edges. In both cases, the outputs were consistent, confirming that the shortest paths were correctly identified.

To test the robustness of the implementation, a variety of graphs were used, including empty graphs, disconnected graphs, graphs with cycles, single-vertex graphs, and graphs with negative edge weights. These tests ensured that the algorithm could handle different edge cases, which can often lead to incorrect outputs in other algorithms. Notably, the Dijkstra algorithm from the Boost Graph Library, which the NWB implementation relies on, cannot process empty graphs. As a result, NWB also cannot handle empty graphs. However, for all other cases, the algorithm ran successfully, producing correct results. The algorithms are computed on a laptop with an Intel Core i7-8550U CPU at 1.8 GHz. As mentioned in chapter 5.1, upon instantiation of a Timer object, the current time is recorded and the duration calculated after destruction of it. This is used to compare the runtime of the Dijkstra, Bellman-Ford and NWB algorithm combined with a time complexity analysis. Furthermore, a benchmarking of the use of CPU time for each function will be done.

To get a better result, we compute a specified amount of runs and take the average of the durations. Aside from the first entry of the table, which is a graph I used to develop and test the new algorithm, the benchmarking graphs are from the 9th DIMACS Implementation Challenge for shortest paths.[31] For the analysis we used, the graphs depicted in this table:

Graph	vertices	edges
Graph from 5.2	5	15
New York graph	264,346	733,846
COL graph	435,666	1,057,066
FLA graph	1,070,376	2,712,798
CAL graph	1,890,815	4,657,742
W graph	6,262,104	15,248,146

Table 6.1: Properties of used graphs for analysis

Graph pairs	vertices	edges
New York/COL	1.65	1.44
COL/FAL	2.46	2.57
FAL/CAL	1.77	1.72
CAL/W	3.31	3.27

Table 6.2: Ratio of vertices and edges of some graphs

The example graph from the previous chapter is used to demonstrate the runtime of the NWB algorithm on a graph with negative edges. Additionally, other graphs are utilized to conduct a time comparison and observe how the runtime increases with larger graph sizes. These graphs were selected due to their ratios of vertices or edges, which mostly fall between 1.5 and 2.5 times that of the smaller graphs. The exception is the W graph, which is more than three times larger than the CAL graph. The New York graph represents the network in New York, the COL graph represents Colorado, FLA represents Florida, CAL represents California and Nevada and W represents Western USA. These ratios of vertices and edges provide useful data for a time complexity analysis of the NWB algorithm's performance as graph size increases.

The results in the following table were achieved by running the algorithm 100 times, except for the W graph from Bellman-Ford. There were only 5 runs, due to the long runtime. The time is in milliseconds.

Graph	Dijkstra	Bellman-Ford	NWB
Graph from 5.2	-	0.01	0.68
New York graph (positive edges)	58.39	6,418.0	303.6
COL graph	96.07	14,243.7	558.03
FLA graph	449.9	59,085.4	1471.78
CAL graph	1001.97	293,760	2,942.47
W graph	2,315.51	760,532	18,329.8
New York graph (negative edges)	-	6,445.45	-

Table 6.3: Runtime from algorithm on different graphs

Graph pairs	Dijkstra	Bellman-Ford	NWB
New York/COL	1.64	2.22	1.84
COL/FLA	4.68	4.15	2.64
FLA/CAL	2.23	4.97	2
CAL/W	2.31	2.59	6.23

Table 6.4: Time ratio of graph pairs

First, it is quite easy to see the bigger graphs are the longer the runtime. As expected, the Dijkstra algorithm is the fastest one. It runs nearly 110 times faster than Bellman-Ford on the New York graph and on the W graph even 328 times faster. As long as the graph only contains positive edges, the NWB algorithm runs much faster than the Bellman-Ford algorithm. This is because before all the subroutines starting, which are explained in chapter 5, the graph is checked if it contains at least one negative edge. If not then we just call Dijkstra. This is sufficient, because all subroutines making parts of the graph non-negative. If everything is positive, the subroutines only compute price functions that do not change the graph. Rather unexpected are the long runtimes of the NWB algorithm on the bigger graphs. If we take the W graph as an example. It takes roughly 2400 ms to execute the Dijkstra algorithm. That means, that around 16000 ms is needed to iterate through the graph and check for a negative edge. On small graphs, such as the first in the table, the NWB finishes and produces the correct output, further proving the correctness even with negative edges. Unfortunately, the implementation is not efficient enough to finish on the smallest graph of the DIMACS Implementation Challenge. This is shown by the first row of the table 6.3. It takes 68 times the time of the Bellman-Ford algorithm to complete, if all subroutines are used. To evaluate the subroutines on a bigger graph, we randomly change one edge weight in the graph to -1. In the last row this was applied to the smallest

graph from the challenge and after 3 hours the algorithm did not stop. It was expected that Bellman-Ford runs nearly with the same time, regardless of the existence of negative edges, because the time complexity is not tied to the signs of the edges either on the vertices and edges. This behavior is evident when comparing the runtime of the algorithm on the New York graph with both positive and negative edges.
This means that for the analysis of the time complexity, we focus exclusively on the case with positive edge weights. Another factor to consider is the space complexity, which involves the amount of memory required to store the data structures, distances, and other necessary elements for the algorithm to run. However, this factor can be omitted from the comparison, as all algorithms have similar space complexity, making it irrelevant to the analysis. Therefore, the focus remains on time complexity when differentiating their performance.[22, 25] This analysis becomes somewhat complex because the actual runtime depends on several factors, such as the implementation, optimizations, and the inherent complexity of the algorithm. Additionally, while the big-O notation provides an upper bound on the time complexity, it typically omits constant factors tied to the size of the graph, such as n and m. In practice, these constants can be significant, meaning that even algorithms with the same theoretical time complexity can have different actual runtimes depending on these hidden factors.
Let's first examine the Bellman-Ford algorithm. The first two graph pairs exhibit behavior within the expected time complexity. If we analyze the ratio of vertices and edges in table 6.2, for example, in the graph pair New York/COL, there are 1.65 times more vertices and 1.44 times more edges, while the time ratio, as shown in 6.4, is 2.22. Ideally, this ratio would be 2.38, but this discrepancy is within an acceptable range, as the algorithm might have terminated earlier, as explained in 5.3. The graph pair CAL/W is exceptional fast. This can be attributed to the W graph requiring considerably fewer iterations, as the CLA graph, despite being much larger. Conversely, the FLA/CAL graph pair, despite the CAL graph having only 1.77 times more vertices and 1.72 times more edges than the FLA graph, exhibits prolonged execution time. This likely stems from the same reason as with the CAL/W graph pair. The FLA graph requires substantially more time to complete, whereas the CAL graph likely terminates with an average number of iterations.
When analyzing the time ratios of the Dijkstra algorithm, the first three graph pairs exhibit significantly slower performance than expected based on time complexity. One possible explanation is hardware limitations, which may prevent the computations from being processed efficiently. Another reason could be that the calculation of the shortest path approaches a near worst-case scenario. Dijkstra's algorithm is a greedy algorithm, which always selects the locally smallest distance to the next vertex. However, this local optimization can result in unnecessary exploration of suboptimal paths. Conversely, the CAL/W graph pair performs much faster than anticipated, with a time ratio of 2.31 compared to the expected 5.72. This discrepancy could be attributed to the fact that fewer unnecessary path explorations occurred in this case, allowing the algorithm to perform more efficiently.
In the original research paper [10], the time complexity is calculated as $O(m * log_2^8(n) * log_2(W))$, where W is the smallest number which satisfies $w(e) \geq -W$. For this comparison, I decided to use the time complexity $O(n * log_2(n) + m)$. This choice is justified by the fact that the algorithm, in this context, does not employ the subroutines described in the paper.

Given that all edges are positive, the algorithm iterates over the edges with a complexity of $O(m)$ and then applies Dijkstra's algorithm with a complexity of $O(n * log_2(n))$.
Using this time complexity, we observe that all graph pairs perform significantly faster than anticipated. Since the algorithm first iterates over the graph and then runs Dijkstra's algorithm, it would be expected that the overall runtime of the NWB algorithm would be always slower. This is indeed reflected in the actual runtimes, as shown in table 6.3. However, based on the time complexity, we would have predicted much longer runtimes for NWB, which the graph pairs COL/FLA and FLA/CAL are disproven. The most plausible explanation for this behavior is that the C++ compiler is applying optimizations during the computation. The compiler might optimize loops to execute iterations simultaneously, if the operations are independent. For example, when iterating over the graph to check for negative edges, parts of the Dijkstra algorithm could start running simultaneously due to loop unrolling or instruction pipelining.
For analyzing the CPU time usage of the functions, we use Intel Vtune Profiler 2024.2. This tool allows analyzing application, system performance, and more performance metrics on Windows. The profiler is multilingual, supporting C, C++, C#, Python and other programming languages.[32]
The following performance analysis ran for 1 and a half hours.

Function Stack	CPU Time: Total ▼	CPU Time: Self	Module	Function (Full)
Total	100.0%	0s		
RtlUserThreadStart	100.0%	0s	ntdll.dll	RtlUserThrea..
BaseThreadInitThunk	100.0%	0s	KERN...	BaseThreadIn.
func@0x140001125	100.0%	0s	SSSP_...	func@0x1400.
func@0x140001154	100.0%	0s	SSSP_...	func@0x1400.
main	100.0%	0s	SSSP_...	main
Shortest_path::SPmain	100.0%	0s	SSSP_...	Shortest_path.
Shortest_path::ScaleDown	98.5%	0.063s	SSSP_...	Shortest_path.
Shortest_path::LDD	96.8%	0s	SSSP_...	Shortest_path.
create_ball_out_recursive	96.6%	0s	SSSP_...	create_ball_o..
create_ball_in_recursive	0.2%	0s	SSSP_...	create_ball_in.
boost::adjacency_list<boost::vecS, boost::vecS, boost::bidirectionalS, b	0.0%	0s	SSSP_...	boost::adjace..
boost::adjacency_list<boost::vecS, boost::vecS, boost::bidirectionalS, boo	0.4%	0s	SSSP_...	boost::adjace..
add_change_weight_neg_edges	0.4%	0.189s	SSSP_...	add_change_..

Figure 6.1: Performance analysis NWB

In the benchmark analysis shown in figure 6.1, it is clear that the ball creation process consumes the most time. This is expected, especially for large graphs, as the radius of the balls increases with the graph's size. As discussed in chapter 5.4, two different approaches were implemented to create these balls. As mentioned in the implementation, the first method does not work. Aside from that, it is computationally expensive, both in terms of memory and time, because the balls for larger graphs are substantial. Each recursive call requires storing the previously calculated balls, creating new variables to store target and weight, leading to significant memory usage. After approximately 3.5 hours, this approach encountered a "bad allocation" error, which occurs when a function fails to allocate the necessary storage.[33]
The second approach employed the Dijkstra algorithm to compute the distances from the ball's center, adding each vertex within the radius. This reduced the computation time from 0.78 ms to 0.68 ms, but the process still did not complete after approximately 3 hours or through errors. To estimate the time required for this approach, we can use the New York

graph as an example, where running the Dijkstra algorithm takes an average of 58 ms. Given this and considering the time required to build the ball by comparing distances, which are estimated at 0.08 s per ball creation, it would take around 352 minutes to compute all balls, based on 264,346 vertices.
To improve performance, I attempted to introduce multithreading, enabling the computation of multiple vertices simultaneously. However, this did not lead to a significant improvement. The hardware provides 8 logical processors, which theoretically should reduce the computation time to approximately 44 minutes for all balls. This estimation is theoretical. The program still does not finish. Another potential optimization could involve using a priority queue to find the smallest element more efficiently. However, this was not pursued further because the algorithm's bottleneck remained in the ball creation process. Further investigation and testing are needed to explore the optimization of functions after the ball creation process.

7 Conclusion

This master thesis introduces the significance of the shortest path problem, presents the implementation of a new algorithm designed to solve it, and evaluates the performance of this implementation in comparison to two well-established algorithms. Theoretical analysis suggests that Dijkstra's algorithm should be the fastest, followed by the NWB algorithm, and finally the Bellman-Ford algorithm. This holds true for graphs with only positive edges, but the scenario changes when the graph contains at least one negative edge. In cases involving negative edges, the algorithm needs to first transform the graph to ensure all weights are non-negative, allowing Dijkstra's algorithm to be applied. The most time-consuming component of the new algorithm is the creation of the balls for the LDD. Therefore, enhancing the performance of this process is crucial to improving the overall speed of the algorithm.

In my limited timeframe, I was unable to provide an implementation that conclusively demonstrates the theoretically faster computation time of NWB due to the years of optimization embedded within the Boost Graph Library.

Future work should focus on further accelerating the ball creation process. The evaluation chapter discusses several potential approaches, some of which have been partially implemented, such as recursively creating the balls, utilizing multithreading to parallelize the procedure, or applying Dijkstra's algorithm to determine distances from a vertex. Additionally, improvements to functions like ElimNeg could be achieved by incorporating a priority queue. Once performance bottlenecks are addressed, the NWB algorithm could also be expanded to handle negative cycles. This potential enhancement, mentioned in the paper, could be implemented after performance improvements are made, making the algorithm even more robust and versatile for a broader range of applications.

This master thesis delivers the first open-source implementation of the NWB algorithm, which not only computes the correct output, but also offers a detailed explanation of the implementation's workings and the expected output of each component. The implementation posed several practical challenges, such as handling the relabeling of graphs or contracting strongly connected components while tracking which vertex corresponds to which SCC. Additionally, it was important to maintain the constraints on the edge weights, ensuring that the most negative edge weight is -1. If this condition is not met, it could compromise the reliability of the algorithm.

In conclusion, the primary objective of this thesis, to provide a simple yet effective algorithm specifically tailored for the SSSP problem, has been achieved. This algorithm, with its potential for near-linear time complexity, holds practical utility and, due to its simplicity, offers significant academic value for teaching purposes.

8 List of Abbreviations

SSSP single-source shortest path

DAG directed acyclic graph

SCC strongly connected component

LDD Low-Diameter Decomposition

Bibliography

[1] Schrijver, Alexander: On the history of the shortest path problem. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=6eb38c11c7ce6d0711483078a969bf82b83eff9f

[2] Pan, Tong ; Pun-Cheng, Shuk C.: A Discussion on the Evolution of the Pathfinding Algorithms. (2020). http://dx.doi.org/10.20944/preprints202008.0627.v1. – DOI 10.20944/preprints202008.0627.v1

[3] Euler, Leonhard ; Velminski, Wladimir: *Leonhard Euler, die Geburt der Graphentheorie: Ausgewählte Schriften von der Topologie zum Sodoku*. Berlin : Kulturverl. Kadmos, 2009. – ISBN 3865990568

[4] Floyd, Robert W.: Algorithm 97: Shortest path. In: *Communications of the ACM* 5 (1962), Nr. 6, S. 345. http://dx.doi.org/10.1145/367766.368168. – DOI 10.1145/367766.368168. – ISSN 0001–0782

[5] A. Goldberg ; Chris Harrelson: Computing the shortest path: A search meets graph theory. In: *ACM-SIAM Symposium on Discrete Algorithms* (2005). https://www.semanticscholar.org/paper/Computing-the-shortest-path%3A-A-search-meets-graph-Goldberg-Harrelson/e39ba4c989874285e9473b4532dc275c12ed78c0

[6] Johnson, Donald B.: Efficient Algorithms for Shortest Paths in Sparse Networks. https://dl.acm.org/doi/pdf/10.1145/321992.321993

[7] Yen, Jin Y.: An algorithm for finding shortest routes from all source nodes to a given destination in general networks. In: *Quarterly of Applied Mathematics* 27 (1970), Nr. 4, S. 526–530. http://dx.doi.org/10.1090/qam/253822. – DOI 10.1090/qam/253822. – ISSN 0033–569X

[8] Eppstein, David: Finding the k Shortest Paths. In: *SIAM Journal on Computing* 28 (1998), Nr. 2, S. 652–673. http://dx.doi.org/10.1137/S0097539795290477. – DOI 10.1137/S0097539795290477. – ISSN 0097–5397

[9] Ramalingam, G. ; Reps, Thomas: An Incremental Algorithm for a Generalization of the Shortest-Path Problem. In: *Journal of Algorithms* 21 (1996), Nr. 2, 267–305. http://dx.doi.org/10.1006/jagm.1996.0046. – DOI 10.1006/jagm.1996.0046. – ISSN 0196–6774

[10] Bernstein, Aaron ; Nanongkai, Danupon ; Wulff-Nilsen, Christian: *Negative-Weight Single-Source Shortest Paths in Near-linear Time*. http://arxiv.org/pdf/2203.03456

[11] Cohen, Michael B. ; Madry, Aleksander ; Sankowski, Piotr ; Vladu, Adrian: *Negative-Weight Shortest Paths and Unit Capacity Minimum Cost Flow in $\tilde{O}(m^{10/7} \log W)$ Time*. http://arxiv.org/pdf/1605.01717

[12] Bast, Hannah ; Delling, Daniel ; Goldberg, Andrew ; Müller-Hannemann, Matthias ; Pajor, Thomas ; Sanders, Peter ; Wagner, Dorothea ; Werneck, Renato F.: *Route Planning in Transportation Networks*. http://arxiv.org/pdf/1504.05140

[13] Leighton, F.: *Introduction to Parallel Algorithms and Architectures*. 1st edition. [Erscheinungsort nicht ermittelbar] and Boston, MA : Morgan Kaufmann and Safari, 2014 https://learning.oreilly.com/library/view/-/9781483221151/?ar. – ISBN 9781483221151

[14] Lavalle, Steven M.: *Planning algorithms*. New York, N.Y : Cambridge University Press, 2006. http://dx.doi.org/10.1017/CBO9780511546877. http://dx.doi.org/10.1017/CBO9780511546877. – ISBN 9780511546877

[15] What Is the Difference Between a Directed and an Undirected Graph. In: *Baeldung on Computer Science* (27.06.2020). https://www.baeldung.com/cs/graphs-directed-vs-undirected-graph#undirected-graphs

[16] van Steen, Maarten: *Graph theory and complex networks: An introduction*. S.l. : Maarten van Steen, 2010. – ISBN 978–90–815406–1–2

[17] Fulber-garcia, Vinicius: Graph Theory: Path vs. Cycle vs. Circuit. In: *Baeldung on Computer Science* (31.01.2022). https://www.baeldung.com/cs/path-vs-cycle-vs-circuit

[18] *The Boost Graph Library - 1.53.0*. https://www.boost.org/doc/libs/1_53_0/libs/graph/doc/index.html. Version: 14.08.2024

[19] Kovacs, Peter: *LEMON*. https://lemon.cs.elte.hu/trac/lemon. Version: 03.10.2017

[20] Board, CGAL E.: *Documentation*. https://www.cgal.org/. Version: 23.07.2024

[21] Khallaghi, Siavash: *lignum-vitae/examples/djikstra/main.cpp GitHub*. https://github.com/siavashk/lignum-vitae/blob/master/examples/djikstra/main.cpp. Version: 23.07.2024

[22] Time and Space Complexity of Dijkstra's Algorithm. In: *GeeksforGeeks* (09.02.2024). https://www.geeksforgeeks.org/time-and-space-complexity-of-dijkstras-algorithm/

[23] *Boost Graph Library: Dijkstra's Shortest Paths - 1.41.0*. https://www.boost.org/doc/libs/1_41_0/libs/graph/doc/dijkstra_shortest_paths.html. Version: 14.08.2024

[24] Jeremy G. Siek, Andrew Lumsdaine, Lie-Quan Lee: *libs/graph/example/bellman-example.cpp*. https://www.boost.org/doc/libs/1_75_0/libs/graph/example/bellman-example.cpp. Version: 15.04.2024

[25] Time and Space Complexity of Bellman–Ford Algorithm. In: *GeeksforGeeks* (09.02.2024). https://www.geeksforgeeks.org/time-and-space-complexity-of-bellman-ford-algorithm/

[26] *Bellman Ford Shortest Paths - 1.85.0*. https://www.boost.org/doc/libs/1_85_0/libs/graph/doc/bellman_ford_shortest.html. Version: 14.08.2024

[27] CSAChannel IISc Danupon Nanongkai: *Negative-Weight Single-Source Shortest Paths in Near-linear TimeDanupon Nanongkai*. https://www.youtube.com/watch?v=awvBpvlbG1M&t=4s. Version: 19.03.2024

[28] West, Douglas B.: *Introduction to graph theory*. 2. Aufl. Upper Saddle River, NJ : Prentice Hall, 2000. – ISBN 0–13–014400–2

[29] Nanongkai, Danupon: *[Public Notes] Low-Diameter Decomposition*. https://hackmd.io/@UOnm1XUhREKPYLt1eUmE6g/Sycpovkiq. Version: 03.05.2024

[30] Online, Shiksha: What is a Priority Queue? - Shiksha Online. In: *Shiksha Online* (15.12.2022). https://www.shiksha.com/online-courses/articles/priority-queue-all-that-you-need-to-know/

[31] *9th DIMACS Implementation Challenge: Shortest Paths*. https://www.diag.uniroma1.it/challenge9/download.shtml. Version: 14.06.2010

[32] Intel: *Fix Performance Bottlenecks with Intel® VTune™ Profiler*. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html. Version: 13.08.2024

[33] *std::bad_alloc - cppreference.com*. https://en.cppreference.com/w/cpp/memory/new/bad_alloc. Version: 15.08.2024