Sequential Optimization of Paths in Directed Graphs Relative to Different Cost Functions

Thesis by:

Malek A. Mahayni

In Partial Fulfillment of the Requirements

For the Degree of

Master of Science

King Abdullah University of Science and Technology

Thuwal, Kingdom of Saudi Arabia

July, 2011
The thesis of Malek A. Mahayni is approved by the examination committee.

Committee Member: Mohamed-Slim Alouini

Committee Member: Basem Shihada

Committee Chairperson: Mikhail Moshkov
ABSTRACT

Sequential Optimization of Paths in Directed Graphs Relative to Different Cost Functions

Malek A. Mahayni

Finding optimal paths in directed graphs is a wide area of research that has received much of attention in theoretical computer science due to its importance in many applications (e.g., computer networks and road maps). Many algorithms have been developed to solve the optimal paths problem with different kinds of graphs. An algorithm that solves the problem of paths’ optimization in directed graphs relative to different cost functions is described in [1]. It follows an approach extended from the dynamic programming approach as it solves the problem sequentially and works on directed graphs with positive weights and no loop edges.

The aim of this thesis is to implement and evaluate that algorithm to find the optimal paths in directed graphs relative to two different cost functions ($\psi, \varphi$). A possible interpretation of a directed graph is a network of roads so the weights for the $\psi$ function represent the length of roads, whereas the weights for the $\varphi$ function represent a constraint of the width or weight of a vehicle. The optimization aim for those two functions is to minimize the cost relative to the $\psi$ function and maximize the constraint value associated with the $\varphi$ function. This thesis also includes finding and proving the relation between the two different cost functions ($\psi, \varphi$). When given a value of one function, we can find the best possible value for the other function. This relation is proven theoretically and also implemented and experimented using Matlab®[2].
ACKNOWLEDGEMENTS

I would like to thank my advisor, Prof. Moshkov, and committee members, Dr. Shihada and Dr. Alouini, for their help and support throughout the course of this research. I learned a lot from Prof. Moshkov, as he was always there for me and my colleagues with his guidance and encouragement. My deepest gratitude goes to my dear parents, Firass Mahayni and Maysaa Lababidi, and to my dear siblings, Aayah; Mohammad; Abdullah and Abdurrahman, to whom I owe a lot in my life. I would like to extend my gratitude to my dear wife Duaa Al-Kanawati for her continuous support and patience throughout my research. I would like also to thank my friends and colleagues at King Abdullah University of Science and Technology for making my journey here an enjoyable experience. Special thanks go to my friends Zuhair Khayyat and Shahid Hussain for their support in this research.
# TABLE OF CONTENTS

Examination Committee ........................................................................... 2  
Copyrights .................................................................................................. 3  
Abstract .................................................................................................... 4  
Acknowledgements .................................................................................... 5  
List of Figures ............................................................................................. 9  
List of Tables .............................................................................................. 10  

I.  Introduction and Preview ........................................................................ 11  
   I.1  Background ......................................................................................... 11  
   I.2  Overview of the Research Problem and Method .............................. 12  
   I.3  Thesis Preview .................................................................................... 12  

II. Literature Review .................................................................................. 14  
   II.1  Introduction ...................................................................................... 14  
   II.2  Single Cost Function ....................................................................... 14  
      II.2.1  Dijkstra’s Algorithm ................................................................. 14  
      II.2.2  Floyd–Warshall’s Algorithm .................................................... 14  
   II.3  Multiple Cost Functions .................................................................. 15  
      II.3.1  Pareto Optimality ..................................................................... 15  
      II.3.2  ε-Approximation Algorithm .................................................... 16  
      II.3.3  Aggregate Objective Function .................................................. 16  
      II.3.4  Evolutionary Algorithms .......................................................... 17  
      II.3.5  Methods that Use Aggregate Objective Function ................. 17  

III. Definitions ............................................................................................ 18  
   III.1  Directed Graph ................................................................................ 18  
   III.2  Edges’ Weights .............................................................................. 18  
   III.3  Paths ............................................................................................... 18  
   III.4  Cost Functions ................................................................................ 19  
      III.4.1  Function $\psi$ ......................................................................... 19  
      III.4.2  Function $\varphi$ ....................................................................... 19  

IV. Method ................................................................................................... 21  
   IV.1  Introduction ..................................................................................... 21
IV.2 Input ........................................................................................................... 21
IV.3 Setting Paths .................................................................................................. 21
IV.4 Optimization Procedures .............................................................................. 23
  IV.4.1 Optimizing Relative to $\psi$ ..................................................................... 24
  IV.4.2 Optimizing Relative to $\phi$ ..................................................................... 25
IV.5 Sequential Optimization .............................................................................. 26
  IV.5.1 Relative to $\psi$ then Relative to $\phi$ ......................................................... 26
  IV.5.2 Relative to $\phi$ then Relative to $\psi$ ......................................................... 27
IV.6 Computational Complexity .......................................................................... 27
  IV.6.1 Size of $\Gamma^0$ .......................................................................................... 27
  IV.6.2 Finding Vertices Reachable from $s$ .......................................................... 28
  IV.6.3 Computations in Optimization for $\psi$ ...................................................... 28
  IV.6.4 Computations in Optimization for $\phi$ ...................................................... 29
  IV.6.5 Computations for Sequential Optimization ............................................. 29
V. Implementation ................................................................................................. 30
V.1 Introduction ....................................................................................................... 30
V.2 Algorithm Implementation .............................................................................. 30
  V.2.1 Program Input ............................................................................................ 30
  V.2.2 Setting Paths (getDAG) ............................................................................ 31
  V.2.3 Graph Pruning (pruneGraph) .................................................................... 33
  V.2.4 Optimizing Relative to $\psi$ ..................................................................... 33
  V.2.5 Optimizing Relative to $\phi$ ..................................................................... 34
  V.2.6 Graph Update (updateGraph) .................................................................... 36
  V.2.7 Create Random Directed Graphs .............................................................. 37
V.3 Sequential Optimization Implementation ...................................................... 38
  V.3.1 Relative to $\psi$ then Relative to $\phi$ ........................................................... 38
  V.3.2 Relative to $\phi$ then Relative to $\psi$ ........................................................... 38
VI. Experiments and Results .............................................................................. 40
VI.1 Introduction ..................................................................................................... 40
VI.2 Experiments on Random Graphs ................................................................. 40
  VI.2.1 Experiments .............................................................................................. 40
VI.2.2 Effect of Growing Number of Edges ................................................. 45
VI.2.3 Comparing Paths of Different Sequential Orders ................................ 45
VI.2.4 Comparing Execution Time between Graphs ..................................... 46
VI.2.5 Example .............................................................................................. 46
VI.3 Experiments on Flight Example .............................................................. 48
  VI.3.1 Experiments ....................................................................................... 48
  VI.3.2 Example .............................................................................................. 49
VII. Functions Transformation ...................................................................... 53
  VII.1 Introduction ........................................................................................ 53
  VII.2 Functions Definitions .......................................................................... 54
    VII.2.1 Function $\Phi$ ............................................................................... 54
    VII.2.2 Function $\Psi$ ............................................................................... 55
  VII.3 Functions Relation ............................................................................. 55
    VII.3.1 Transform $\Psi$ into $\Phi$ ................................................................. 56
    VII.3.2 Transform $\Phi$ into $\Psi$ ................................................................. 57
  VII.4 Application of Functions Relation ...................................................... 57
  VII.5 Implementation .................................................................................. 59
    VII.5.1 Get $\Psi$ Tuple (getPsiTuple) ......................................................... 59
    VII.5.2 Transform $\Psi$ into $\Phi$ ................................................................. 60
  VII.6 Experiments ........................................................................................ 61
    VII.6.1 Experiments on Random Graphs ................................................... 61
    VII.6.2 Flight Example .............................................................................. 64
VIII. Conclusion ............................................................................................. 66

References ..................................................................................................... 67
LIST OF FIGURES

Figure 2.1 Pareto Curve, Two Objectives, Both Minimization ........................................ 16
Figure 6.1 Execution Time, n=20 .................................................................................... 42
Figure 6.2 Execution Time, n=45 ................................................................................... 43
Figure 6.3 Execution Time, n=70 .................................................................................... 45
Figure 6.4 Execution Time for Different Values of n ....................................................... 46
Figure 6.5 Original Graph, n=4 ...................................................................................... 47
Figure 6.6 $\Gamma$ of n=4 .................................................................................................. 47
Figure 6.7 After $\psi$ then $\phi$ ....................................................................................... 47
Figure 6.8 After $\phi$ then $\psi$ ....................................................................................... 48
Figure 6.9 Flights Graph, Partial Snapshot .................................................................... 50
Figure 6.10 Select Departure City .................................................................................. 51
Figure 6.11 Select Destination City ............................................................................... 51
Figure 6.12 Flights Graph, Partial Snapshot, From-To ..................................................... 51
Figure 7.1 Tuple $\text{Tup}\psi$ (Pareto-curve) ....................................................................... 63
Figure 7.2 Tuple $\text{Tup}\phi$ ............................................................................................ 63
Figure 7.3 Fare vs. Baggage ......................................................................................... 65
Figure 7.4 Baggage vs. Fare ........................................................................................ 65
LIST OF TABLES

Table 6.1 n=20, $\psi$ first.............................................................................................. 41
Table 6.2 n=20, $\varphi$ first............................................................................................ 41
Table 6.3 n=45, $\psi$ first.............................................................................................. 42
Table 6.4 n=45, $\varphi$ first............................................................................................ 43
Table 6.5 n=70, $\psi$ first.............................................................................................. 44
Table 6.6 n=70, $\varphi$ first............................................................................................ 44
Table 6.7 Flight Example, Avg. 40 Routes................................................................. 49
Table 7.1 Time Results for Tuples.............................................................................. 62
Chapter 1

I. INTRODUCTION AND PREVIEW

I.1 Background

Directed graphs are popular for representing many applications such as road maps, computer networks, Protein-Protein Interaction networks and social networks. Using the direct representation, we can apply many well defined and studied graphs’ algorithms to solve problems related to those networks.

One famous problem is finding the optimal route [3] [4] between two points on a map or two computers in a network, hence finding optimal paths in a directed graph. Since different applications and different problems may not agree on the same definition of optimality, different evaluation criteria have been made to grade paths. For example an optimal route between two points in a map might be the shortest or the fastest, and in a computer network it can be the one that has minimum delay [5] or maximum bit rate.

To accommodate those varying criteria, different cost functions have been set to properly reflect the original problems into the algorithms on graphs. In other words, a function can evaluate the paths in a graph representing a map and return the paths’ lengths; another can evaluate paths in a graph representing a computer network and return the paths’ bit rates. Using such functions, optimization algorithms can then find the best paths, which serve the original problems. All those endless possibilities of applications and their problems serve as a motivation to design various algorithms to represent and solve them. Some examples of such algorithms are mentioned in Chapter II.
that includes Dijkstra's algorithm [6] and Floyd–Warshall’s [7] algorithm; both seek the optimal path as the shortest one.

I.2 Overview of the Research Problem and Method

For slightly difficult applications, many optimization problems work for defining the optimal paths, so the well-known algorithms might not be able to solve all these problems. This calls for more complex solutions that take into consideration all the contributing optimization problems using their corresponding functions. In this thesis we introduce an algorithm, based on dynamic programming [8], which solves the optimization problem on directed graphs for different cost functions in a sequential order. Any number of cost functions can be used in any preferred order; the algorithm optimizes the given directed graph for the first function, and then takes the resulting directed graph to be optimized for the second function and so on.

This algorithm differs from the others since it accommodates any number of different cost functions, and only requires that the functions’ optimization procedures to be defined. Two types of cost functions are stated along with the algorithm, and their optimization procedures are defined and designed to run in polynomial time. Even though the algorithm uses the approach of dynamic programming and of a polynomial complexity, it might score lower than other algorithms in some cases because it requires extra space (graph size is squared) and hence needs more execution time.

I.3 Thesis Preview

The main contribution of this thesis is in implementing the sequential optimization algorithm, defining and proving the relation between the cost functions and conducting
experiments. The thesis is organized as follows. Chapter III states the main definitions required for the problem including the definitions of the two types of cost functions. The main method of the algorithm is explained in Chapter IV including the optimization procedures for the defined functions and how the sequential optimization is applied. Detailed explanation for the program of the algorithm, along with optimization procedures and supplementary procedures, is in Chapter V. The experiments on the program takes place in Chapter VI, where different kinds of experiments and comparisons are reported. A relation between the two cost functions along with a transformation between them are defined, proved and then experimented in Chapter VII. The thesis ends with a conclusion in Chapter VIII.
Chapter 2

II. LITERATURE REVIEW

II.1 Introduction

In this chapter we discuss some well-known existing algorithms and approaches that solve similar problems to the sequential optimization algorithm.

II.2 Single Cost Function

II.2.1 Dijkstra’s Algorithm

Created by the computer scientist Edsger Dijkstra, this algorithm follows the greedy approach [9] to find the shortest paths from one source vertex to all other vertices in a graph given that there are no negative weights on edges. In terms of cost functions, it finds the optimal paths based on one cost function only, which is the sum of the edges’ weights in a path. It runs in $O(|V|^2)$ time, where $V$ is the set of vertices in the given graph.

II.2.2 Floyd–Warshall’s Algorithm

This algorithm finds the shortest paths between all pairs of vertices in a graph. Unlike Dijkstra’s, it allows negative weights on edges. It utilizes the dynamic programing approach and its running time complexity is $O(|V|^3)$. It shares the same cost function with Dijkstra. Compared to the sequential optimization algorithm, it requires much less space $\Theta(|V|^2)$ and runs it somehow similar complexity, since the sequential algorithm –
as we will see later- requires $O(|V| \times |E|)$ where $E$ is the set of edges in the graph and its size can reach $O(|V|^2)$.

II.3 Multiple Cost Functions

The problem of optimizing for more than one cost function is a well-known area of study. It is often called multi-objective optimization. Most of the time when optimizing for multi-objectives there is no one optimal solution, instead multiple solutions exist [10],[11]. This shows the tradeoff between the conflicting objectives. One of the main concepts that define the possible solutions for such optimization is called Pareto optimality [12], explained more in Sec. II.3.1. Many approaches and algorithms exist to solve such problem; some are listed in the next sections.

II.3.1 Pareto Optimality

Named after the Italian economist Vilfredo Pareto [13], this concept describes a situation or solution as Pareto optimal if this solution cannot be further improved for a certain objective without being worsened for another. The opposite case would be the dominated solution [14], which can be enhanced for an objective without getting worse for another objective, hence it is dominated by the one acquired after the enhancement. It is clear that a Pareto solution is non-dominated. Multi-objective optimization in most cases gives a set of Pareto optimal solutions (called Pareto set) [15]. Each solution would have a certain tradeoff between the objectives. Those solutions can be mapped in a curve called Pareto curve, like the example shown in Figure II.1 for a minimization of two objectives.
II.3.2 $\varepsilon$-Approximation Algorithm

Finding whether a solution lies on Pareto curve is NP-hard according to [16], which introduces an $\varepsilon$-approximation algorithm to find the Pareto set for multi-objective problems in polynomial time with respect to $1/\varepsilon$. This algorithm was used in [17] to find Pareto solutions for Internet Service Level Agreement.

II.3.3 Aggregate Objective Function

One famous approach to solve the multi-objective optimization is to gather the objectives in a single function called Aggregate objective function (AOF), and then optimize for this new function [18]. A possible method using this approach is the direct weighted sum of the objectives, but this method is subjective because the way the weights are selected directly affects the solutions. Better objective methods are made using Evolutionary algorithms [19].
II.3.4 Evolutionary Algorithms

Many artificial intelligence algorithms, such as Genetic algorithms [20] and Simulated Annealing [21], have been used as tools to solve the multi-objective optimization problem [22],[23],[24] whether to find the Pareto-optimal set or approximate it. Most multi-objective evolutionary algorithms apply ranking schemes based on Pareto optimality [25]. Though proved successful, multi-objective evolutionary algorithms that use have some downsides such as high computational complexity [26]. This called for better enhanced algorithms such as “non-dominated sorting genetic algorithm (NSGA-II)” [26],[27]; “strength Pareto evolutionary algorithm (SPEA)” [28] along with its successor (SPEA2) [29] which had its own improvement (SPEA2+) [30]; and “Pareto-frontier differential evolution approach (PDE)” [31], [32]. Some multi-objective evolutionary algorithms follow the approach of approximating the Pareto-optimal set, such algorithms are mentioned in [33],[34],[35]. The applications of evolutionary algorithms are limitless in many areas [36], such as engineering design, combinatorial logic circuits design, polymer extrusion problems and city and regional planning[36].

II.3.5 Methods that Use Aggregate Objective Function

Some multi-objective optimization methods work by constructing multiple aggregate objective functions; each AOF solution is a Pareto-point. Such methods include Normal Constraint (NC) [37],[38] and Directed Search Domain (DSD) [39]. In both methods, the AOFs are made to get equally distributed Pareto-points that best approximate the actual Pareto-set. The two methods differ in the way they filter out the local Pareto-points.
Chapter 3

III. DEFINITIONS

III.1 Directed Graph

The algorithm works on directed graphs [40] with the constraints of not allowing loops or multiple edges. A directed graph \( G = (V, E) \) is a group of vertices \( V \) and edges \( E \). Each edge is basically an ordered pair of vertices \((v_i, v_j)\), where it connects vertices \( v_i \) and \( v_j \) and is directed from the first to the second; hence, \((v_i, v_j) \neq (v_j, v_i)\). Having no loops means there are no edges that start and end at the same vertex; in other words for all edges \((v_i, v_j), \ i \neq j\). Not allowing multiple edges means there exists at most one edge between any given ordered pair of vertices; so if exists, there’s only one edge \((v_i, v_j)\) from vertex \( v_i \) to \( v_j \).

III.2 Edges’ Weights

In weighted graphs, each edge \( e \) has a weight \( w \) assigned to it, which is a nonnegative real number. The algorithm considers graphs that have more than one weight assigned to each edge. The weight assigned to the edge \((v_i, v_j)\) is denoted by \( w(v_i, v_j) \). Note that \( w(v_j, v_i) \) does not necessarily equal to \( w(v_i, v_j) \).

III.3 Paths

A path \( p \) in a directed graph is a sequence of vertices connected by edges; between each two consecutive vertices in the path there exists a directed edge that goes from the first vertex to the second. So we can represent \( p \) in a formal way, \( p = (v_1, v_2, ..., v_m) \). Each
path has one or more costs related to the weights on its edges. The length of a path is the number of edges that are used in it. Since the algorithm works on directed graphs, then from here after any mention of the name path means a directed path. Paths can be of two types: cyclic and acyclic. In a cyclic path at least one vertex appears more than once, e.g., $p = (v_1, \ldots, v_i, \ldots, v_i, \ldots, v_m)$, whereas in an acyclic path all vertices appear at most once.

### III.4 Cost Functions

Given a path, cost functions return the cost related to the path by some operations on the weights of the path’s edges. We are going to discuss two types of cost functions for paths in directed graphs.

#### III.4.1 Function $\psi$

The function $\psi$ basically calculates the cost of a path as the sum of the edges’ weights. Given a path $p = (v_1, v_2, \ldots, v_k, v_{k+1})$ the function defines the cost as $\psi(p) = \sum_{i=1}^{k} w(v_i, v_{i+1})$. If a path contains only one vertex i.e., $k = 0$ then $\psi(p) = 0$. The optimization goal for $\psi$ is to find the path or paths that have the minimum value for the cost $\psi(p)$. Many well-known algorithms, such as the ones discussed in Sec. II.2.1 and Sec. II.2.2 earlier, solve the problem of optimizing for this function. Later on, the edges’ weights corresponding to this function will be referred to as $w_\psi$.

#### III.4.2 Function $\varphi$

The function $\varphi$ deals with the weights on the edges in a different way than $\psi$. Here, a weight is like a constraint on or a capacity of the corresponding edge. A good example is Flow Networks, where each edge has a capacity $c(v_i, v_j)$ that sets the maximum flow that can pass through this edge. So the function calculates the cost of a path as the
minimum weight among the weights of the edges in the path. Given a path \( p = (v_1, v_2, ..., v_k, v_{k+1}) \), the function defines the cost as

\[ \varphi(p) = \min(w(v_1, v_2), w(v_2, v_3), ..., w(v_k, v_{k+1})) \].

If a path contains only one vertex i.e., \( k = 0 \) then \( \varphi(p) = +\infty \). The optimization goal for \( \varphi \) is to find the path or paths that have the maximum value for the cost \( \varphi(p) \). Later on, the edges’ weights corresponding to this function will be referred to as \( w_\varphi \).
Chapter 4

IV. METHOD

IV.1 Introduction

The input for the algorithm is discussed first, and then comes the stages of the actual work of the algorithm.

IV.2 Input

The algorithm takes as input a directed graph $G(V, E)$. The graph is represented using two adjacency matrices: $G_{\psi}$ and $G_{\varphi}$ that have the weights $w_{\psi}$ and $w_{\varphi}$ respectively. Each row $i$ represents the “from” vertex, and each column $j$ represents the “to” vertex. The value at the $G_{\psi}(i, j)$ cell is the weight $w_{\psi}$ on the edge $(v_i, v_j)$ that goes from vertex $i$ to $j$. The same idea applies to the $G_{\varphi}$ matrix. If an edge doesn’t exist, the corresponding cell value is zero. Since the algorithm doesn’t allow loop edges, all the diameter cells (where $i = j$) in both matrices have the value zero. The algorithm also takes the source $s$ and target $t$ vertices as input.

IV.3 Setting Paths

Let $V = \{v_1, ..., v_n\}$, $v_1 = s$, $v_n = t$, and let $P(G)$ be the set of all the paths from $v_1$ to $v_n$ in $G$ that follow these two conditions:

a) In each path, both $s$ and $t$ appear only once.

b) The maximum length for a path is $n - 1$, where $n$ is the number of vertices in $V$. 
The goal of the algorithm is to find the optimal paths among $P(G)$. To represent those paths accurately, we use a directed acyclic graph (DAG) [41] $\Gamma$ described as follows: We build $\Gamma$ in $n$ layers; the first layer contains the vertex $s$, labeled $v_1^1$, only. The $n^{th}$ layer has the vertex $t$, labeled $v_n^n$, only. Each of the other $n - 2$ layers contains all of the vertices other than $s$ and $t$, each vertex is labeled $v_i^k$, where $k$ is the layer’s number and $i$ is the vertex number, and $i, k, j = 2, 3, ..., n - 1$. The edges in $\Gamma$ correspond to the ones in $G$, and can be described as follows. The edge $(v_1^1, v_n^n)$ exists in $\Gamma$ if and only if $(v_1, v_n) \in E$. Between the first and the second layers, there is an edge $(v_1^1, v_1^2)$ in $\Gamma$ if and only if there exists an edge $(v_1, v_1) \in E$. For the middle $n - 2$ layers, there is an edge $(v_i^k, v_j^{k+1})$, $k = 2, 3, ..., n - 2$, in $\Gamma$ if and only if there exists an edge $(v_i, v_j) \in E$.

Between any middle layer and the $n^{th}$ layer, there is an edge $(v_i^k, v_n^n)$, $k = 2, ..., n - 1$, in $\Gamma$ if and only if there exists an edge $(v_i, v_n) \in E$. These described edges are the only edges in $\Gamma$. The weights of the edges in $\Gamma$ are the same as the corresponding edges’ weights in $G$. In other words, for any edge $(v_i^{k1}, v_j^{k2}) \in \Gamma$, $w(v_i^{k1}, v_j^{k2}) = w(v_i, v_j)$, of course this applies to both types of weights: $w_p$ and $w_q$. Let $P(\Gamma)$ be the set of all paths between $v_1^1$ and $v_n^n$ in $\Gamma$. We can safely say that there is a one to one mapping from the set $P(G)$ to the set $P(\Gamma)$. A path $(v_1, v_{i2}, ..., v_{it}, v_n)$ in $P(G)$ corresponds to the path $(v_1^1, v_{i2}^2, ..., v_{it}^t, v_n^n)$ in $P(\Gamma)$.

One might wonder: why bother set the paths in the DAG $\Gamma$ with layers and then apply the optimization procedures, instead of directly applying on the original graph?

There are two main reasons behind the need to represent the paths in this way:
a) The vertices in $\Gamma$ have the feature of being already sorted in a topological order [9]; this is because of the order of the layers, and the fact that the edges start from a layer and end in another one always after it and never in the same layer. This is very crucial as both of the optimization procedures, while working on a vertex, require information from its direct predecessors. So the order in which the vertices are investigated is very important, and by knowing that they are already in topological order they can be investigated safely in that order. Applying the optimization procedures on the original graph directly creates the problem of inconsistency. Contrary to $\Gamma$, the original graph can have cycles, which will cause the information on some vertices’ predecessors to change after those vertices have been already investigated. That adds the headache of having to investigate the vertices more than once. This idea will be clearer when reading the next section about the optimization procedures.

b) The structure of $\Gamma$ is in principle built around the paths from $v_1$ to $v_n$, the maximum length of a path is $n - 1$ and there are no cycles. That reserves all the acyclic paths, since they can only reach a maximum length of $n - 1$, and any cyclic path that is of length $n - 1$ or less. For example, a cyclic path $(v_1, ..., v_i, ..., v_n)$ in $G$ is represented in $\Gamma$ as $(v_1^{k_1}, ..., v_i^{k_j}, ..., v_n^{k_n})$.

**IV.4 Optimization Procedures**

Here we define the optimization procedures one at a time. Both of the next mentioned optimization procedures work independently, and the work they do is not affected by the order they are executed. Each of them works on a given $\Gamma$. Later in Sec. IV.5 we will discuss how they are used sequentially.
IV.4.1 Optimizing Relative to $\psi$

The algorithm first applies pruning on $\Gamma$; it removes all the vertices that cannot be reached from $v_1$. In other words, if there is no path from $v_1$ to a vertex then that vertex is deleted along with all its incoming and outgoing edges. The resulted subgraph is referred to by $\Gamma'$. The optimization goal here is to label each vertex $v_j^k$ in $\Gamma'$ by $\Psi(v_j^k)$ which is the minimum cost of a path in $\Gamma'$ from $v_1$ to $v_j^k$ relative to the cost function $\psi$, and refine the incoming edges to $v_j^k$. The vertices are investigated according to their order in $\Gamma'$, layer by layer. This assures that when investigating a vertex all its predecessors have already been investigated, so that each vertex is investigated only once. Accordingly, $v_1$ is labeled with $\Psi(v_1) = 0$. For $v_j^k \neq v_1$, let $v_{i_1}^{k_1}, v_{i_2}^{k_2}, ..., v_{i_r}^{k_1}$ be all the vertices in $\Gamma'$ that have outgoing edges to $v_j^k$. We can write:

$$\Psi(v_j^k) = \min_{1 \leq i \leq r} \{\Psi(v_{i_t}^{k_t}) + w_\psi(v_{i_t}^{k_t}, v_j^k)\}.$$  \hspace{1cm} (1)

After finding the label $\Psi(v_j^k)$ for $v_j^k$, we remove all the incoming edges to $v_j^k$ that give a cost greater than $\Psi(v_j^k)$. In other words, we remove all $(v_{i_t}^{k_t}, v_j^k)$ edges where:

$$\Psi(v_{i_t}^{k_t}) + w_\psi(v_{i_t}^{k_t}, v_j^k) > \Psi(v_j^k).$$

After applying the process of finding the label and removing some edges on all the vertices in $\Gamma'$ the resulting graph is called $\Gamma_\psi$. The set of paths from $v_1$ to $v_n$ in $\Gamma_\psi$ is referenced as $P(\Gamma_\psi)$. Since at every vertex the algorithm keeps at least one incoming
edge; it is safe to say that $P(\varphi)$ is not empty. The next theorem is stated in [1], where it is fully proved.

**Theorem 1.** “The set of paths $P(\varphi)$ coincides with the set of paths from $P(\Gamma)$ that have minimum cost relative to $\varphi$.” [1].

**IV.4.2 Optimizing Relative to $\varphi$**

The algorithm first applies pruning on $\Gamma$; it removes all the vertices that cannot be reached from $v_1^1$. In other words, if there is no path from $v_1^1$ to a vertex then that vertex is deleted along with all its incoming and outgoing edges. The resulted subgraph is referred to by $\Gamma'$. The optimization goal here is to label each vertex $v_j^k$ in $\Gamma'$ by a label $\Phi(v_j^k)$ which is the maximum cost associated with a path in $\Gamma'$ from $v_1^1$ to $v_j^k$ relative to the cost function $\varphi$. After labeling all the vertices, some refinement will be done on all the edges in $\Gamma'$. The vertices are investigated according to their order in $\Gamma'$, layer by layer. This assures that when investigating a vertex all its predecessors have already been investigated, so that each vertex is investigated only once. Accordingly, $v_1^1$ is labeled with $\Phi(v_1^1) = +\infty$. For $v_j^k \neq v_1^1$, let $v_{i_1}^{k_1}, v_{i_2}^{k_2}, \ldots, v_{i_r}^{k_1}$ be all the vertices in $\Gamma'$ that have outgoing edges to $v_j^k$. We can write:

$$\Phi(v_j^k) = \max_{1 \leq t \leq r} \left\{ \min \{ \Phi(v_{i_t}^{k_t}), w_{\varphi}(v_{i_t}^{k_t}, v_j^k) \} \right\}. \quad (2)$$

After finishing the work on all the vertices, we delete all the edges in $\Gamma'$ that have their $w_{\varphi}$ weight less than the label of $v_n^a$. Meaning, we remove all edges $(v_{i_1}^{k_1}, v_{i_2}^{k_2})$ that have:
After removing all such edges, the resulting graph is called $\Gamma_\varphi$. The set of paths from $v_1^1$ to $v_n^n$ in $\Gamma_\varphi$ is referenced as $P(\Gamma_\varphi)$. The value $x = \Phi(v_n^n)$ means that there exists at least one or more paths from $v_1^1$ to $v_n^n$ that have their costs equal to $x$, and none of their edges have a weight $w_\varphi$ less than $x$; therefore none of their edges will be removed. This shows that $P(\Gamma_\varphi)$ is not empty. The next theorem is stated in [1], where it is fully proved.

**Theorem 2.** “The set of paths $P(\Gamma_\varphi)$ coincides with the set of paths from $P(\Gamma)$ that have maximum cost relative to $\varphi$.” [1].

### IV.5 Sequential Optimization

After defining the optimization procedures, we now state how they are used in the overall algorithm. Sequential optimization is the core mechanism of the algorithm. It gives the ability to optimize for as many cost functions as needed, one at a time, in any order. The first procedure is given a DAG to work on, and then produce a resulting subgraph, which is then given as an input to the next procedure, etc. Let us consider sequential optimization on a graph $G$ relative to our two defined functions, $\psi$ and $\varphi$, the optimization procedure can be done in two ways:

#### IV.5.1 Relative to $\psi$ then Relative to $\varphi$

First of all, $G$ is transferred to the DAG $\Gamma$. The optimization procedure for $\psi$ is applied on $\Gamma$, the resulting subgraph is named $\Gamma_\psi$. According to Theorem 1, the set of paths $P(\Gamma_\psi)$ from $v_1^1$ to $v_n^n$ in $\Gamma_\psi$ coincides with $D$: the set of paths from $v_1^1$ to $v_n^n$ in $\Gamma$ that have the minimum cost relative to $\psi$. Next, the optimization procedure for $\varphi$ is applied on $\Gamma_\psi$, the
resulting subgraph is named $\Gamma_{\varphi\psi}$. According to Theorem 2, the set of paths $P(\Gamma_{\varphi\psi})$ from $v_1^1$ to $v_n^n$ in $\Gamma_{\varphi\psi}$ coincides with the set of paths in $D$ that have the maximum cost relative to $\varphi$.

**IV.5.2 Relative to $\varphi$ then Relative to $\psi$**

Similar to the previous order, $G$ is transferred to the DAG $\Gamma$ first. The optimization procedure for $\varphi$ is applied on $\Gamma$, the resulting subgraph is named $\Gamma'_{\varphi}$. According to Theorem 2, the set of paths $P(\Gamma'_{\varphi})$ from $v_1^1$ to $v_n^n$ in $\Gamma'_{\varphi}$ coincides with $S$: the set of paths from $v_1^1$ to $v_n^n$ in $\Gamma$ that have the maximum cost relative to $\varphi$. Next, the optimization procedure for $\psi$ is applied on $\Gamma'_{\varphi}$, the resulting subgraph is named $\Gamma'_{\varphi\psi}$. According to Theorem 1, the set of paths $P(\Gamma'_{\varphi\psi})$ from $v_1^1$ to $v_n^n$ in $\Gamma'_{\varphi\psi}$ coincides with the set of paths in $S$ that have the minimum cost relative to $\psi$.

In the same way, the optimization procedure can be then continued for more different cost functions.

**IV.6 Computational Complexity**

Let us first study the computational complexity of each algorithm part separately, and then we will state the computational complexities of some sequential optimization orders.

Let $G = (V, E)$, where $|V| = n$ and $|E| = m$.

**IV.6.1 Size of $\Gamma_0$**

Let $\Gamma_0 = (V_0, E_0)$. From its definition, $\Gamma_0$ has $n$ layers. All layers, except for the first and last one, have all the vertices of $G$ except $s$ and $t$, meaning that each of those $n - 2$ layers
has \( n - 2 \) vertices. So the total would be \((n - 2)^2 + 2\), the 2 being the vertices \( s \) and \( t \), or we can write \( |V_0| = O(n^2) \). Following the same reasoning, we can find \( |E_0| = O(mn) \).

**IV.6.2 Finding Vertices Reachable from \( s \)**

In both discussed optimization procedures \( \psi \) and \( \varphi \) we first find the vertices that can be reached from \( v_1 = s \) through paths. Let \( \Gamma = (V_1, E_1) \) be a subgraph of \( \Gamma_0 \) that has been obtained after applying some optimization procedures. Since any procedure only decreases or has no effect on the size of its input DAG, it is safe to write that \(|V_1| \leq |V_0|\), and that \(|E_1| \leq |E_0|\). To find the vertices in \( \Gamma \) that are reachable from \( v_1 \) we can use *breadth-first search* algorithm, which has a time complexity of \( O(|V_1| + |E_1|) \). Based on the size of \( \Gamma \) relative to \( \Gamma_0 \) it is safe to write this time complexity as \( O(n^2 + mn) \). After finding the reachable vertices the others are removed along with their incident edges, and the obtained subgraph of \( \Gamma \) is named \( \Gamma' \).

**IV.6.3 Computations in Optimization for \( \psi \)**

When labeling a vertex in \( \Gamma' \), the procedure first finds the minimum cost possible for the investigated vertex, and then deletes the incoming edges that produce larger costs. From the perspective of an edge \((v_{i}^{kl}, v_{j}^{kj})\) we find that three operations at most are done per edge in order to get \( \Gamma'_\psi \). It is first given a score based on the sum of its weight \( w_{\psi} \) with the label \( \Psi(v_{i}^{kl}) \). Then the minimum score along all the edges incoming to \( v_{j}^{kj} \), including \((v_{i}^{kl}, v_{j}^{kj})\), is found using one comparison for each edge. Then its score is compared to the label \( \Psi(v_{j}^{kj}) \) i.e., the minimum score, to decide whether it is going to be removed or not. Those operations per edge are one summation and two comparisons. Since we have
$O(mn)$ edges, the total number of operations equals $3 \times O(mn) = O(mn)$ operations of comparisons and additions are needed to get $I'_{\psi}$ from $I''$.

**IV.6.4 Computations in Optimization for $\varphi$**

When labeling a vertex in $I''$, the procedure first finds the maximum cost possible for each vertex, and then removes all the edges that have a weight $w_\varphi$ less than the label of $v_i^m$. From the perspective of an edge $(v_i^{k_1}, v_j^{k_2})$, we first find its score which is the minimum between its $w_\varphi$ and the label $\Phi(v_i^{k_1})$, and that one comparison. Then the maximum score along all the edges incoming to $v_j^{k_2}$, including $(v_i^{k_1}, v_j^{k_2})$, is found using one comparison for each edge. When the label $\Phi(v_i^{m})$ is found, the edge’s $w_\varphi$ is compared to that label to see whether it is going to be removed or not. Those operations per edge are three comparisons. Since we have $O(mn)$ edges, the total number of operations equals $3 \times O(mn) = O(mn)$ operations of comparisons are needed to get $I'_{\varphi}$ from $I''$.

**IV.6.5 Computations for Sequential Optimization**

For any order of sequential optimization, with fixed number of optimizations, the computation complexity will always be of $O(mn)$. That is basically due to the fact that the optimization procedures are executed one after the other and their computation complexities are just added, and it is known that the complexity of both procedures is $O(mn)$. 
Chapter 5

V. IMPLEMENTATION

V.1 Introduction

In this section, the implementation of the optimization procedures, paths construction in DAG, and the sequential optimization are explained.

V.2 Algorithm Implementation

The implementation (hereafter referenced as the program) was made in MATLAB® R2010b [2] with the use of two additional toolboxes: Bioinformatics Toolbox, and Statistics Toolbox. The implementation of each of the algorithm’s parts will be stated separately, and then the program’s execution for the full sequential optimization will be explained. Each of the algorithm’s parts is implemented in a MATLAB® function.

V.2.1 Program Input

As the algorithm’s input was defined in Sec. IV.2, the program takes two adjacency matrices: $G_\psi$ and $G_\varphi$ that represent the graph $G = (V,E)$. The program also takes the numbers of two vertices $s, t \in V$ as the source and target vertices. The user has the option whether to provide the graph or not. The user can specify the preferred order $r$ of the sequential optimization; $r = 1$ to have $\psi$ then $\varphi$, or $r = 0$ to have $\varphi$ then $\psi$, the default is $r = 1$ ($\psi$ then $\varphi$). The program checks all the inputs for errors.

V.2.1.1 Graph Provided by User

The user provides the following:
a) $G_\psi$ and $G_\varphi$.

b) $s$ and $t$.

c) $r$: Order of sequential optimization (optional).

### V.2.1.2 Random Graph

a) $G_\psi$ and $G_\varphi$ are generated randomly.

b) $s$ and $t$: The user can specify, or let the program choose them.

c) $r$: The user can specify.

### V.2.1.3 Input Checking

All the inputs are checked for errors.

a) $G_\psi$ and $G_\varphi$: Checked for negative values, and both must have the same size.

b) $s$ and $t$: Checked to be within the range of vertices, and whether they are connected with at least one path from $s$ to $t$ or not. If not connected, the program stops and gives a proper output as stated later in the output section.

c) $r$: If any value other than 1 or 0 is entered, the program executes the default: $\psi$ then $\varphi$.

### V.2.2 Setting Paths (getDAG)

#### V.2.2.1 Function Header

```latex
[ DAGpsi, DAGphi] = getDAG( Gpsi, Gphi, s, t )
```

**Inputs:**

1) Gpsi: $G_\psi$. 


2) Gphi: $G_{\phi}$.

3) s,t: the source and the target vertices, respectively.

Outputs:

1) DAGpsi: $dag_\psi$.

2) DAGphi: $dag_\phi$.

V.2.2.2 Function Details

The method of setting the paths, as mentioned in Sec. IV.3, works on the adjacency matrices $G_{\psi}$ and $G_{\phi}$ to produce two DAGs: $dag_\psi$ and $dag_\phi$, respectively. Of course the values of $s$ and $t$ are required for the method, and they are hereafter changed to $v_1^1 = s$ and $v_n^n = t$. Let $n = |V|$, found from the number of rows in $G_{\psi}$ or $G_{\phi}$. Each of the new DAGs has $(n - 2)^2 + 2$ vertices as stated in Sec. IV.6.1, and therefore is represented as an adjacency matrix with $(n - 2)^2 + 2$ rows and $(n - 2)^2 + 2$ columns. The first row in $dag_\psi$ represents the edges outgoing from $v_1^1$. The first column in $dag_\phi$ is all zeros since there are no incoming edges to $v_1^1$. The last column in $dag_\psi$ represents the edges incoming to $v_n^n$. The last row in $dag_\phi$ is all zeros since there are no outgoing edges from $v_n^n$. The rest of the rows and columns representing the edges of the other vertices are ordered by the layer’s number then the number of the vertex in $V$. For example, the row corresponding to vertex $v_4^3$ comes, not necessarily directly, before the row of vertex $v_2^4$ and, not necessarily directly, after the row of vertex $v_2^3$. All the mentioned properties of $dag_\psi$ also apply to $dag_\phi$. 
V.2.3 Graph Pruning (pruneGraph)

V.2.3.1 Function Header

\[
[ \text{G1, G2} ] = \text{pruneGraph}( \text{G1, G2, s} )
\]

Inputs:

1) G1, G2: The matrices representing the graph to be pruned.
2) s: The number of the vertex that is the source for the graph.

Outputs:

1) G1, G2: The matrices after being pruned.

V.2.3.2 Function Details

This function is needed since it is used in both of the optimization procedures for the cost functions. The function basically traverses a given graph starting from s, and using breadth-first search algorithm to find the vertices reachable from s. The vertices that are not reachable from s are removed from the graph by simply assigning zeros to all there incoming and outgoing edges. In other words, the row and column corresponding to a non-reachable vertex are set to zeros. To keep both G1 and G2 updated, they are both are given to the function and are updated in the same way.

V.2.4 Optimizing Relative to \(\psi\)

V.2.4.1 Function Header

\[
[ \text{GAMAp}si, \text{OPTpsi} ] = \text{psi}( \text{DAGpsi} )
\]

Inputs:
1) DAGpsi: \( \text{dag}_\psi \).

Outputs:

1) GAMApsi: \( \Gamma_\psi \).

2) OPTpsi: \( \text{Opt}_\psi \).

**V.2.4.2 Function Details**

At first, both \( \text{dag}_\psi \) and \( \text{dag}_\varphi \) are pruned using \([\text{dag}_\psi, \text{dag}_\varphi] = \text{pruneGraph} (\text{dag}_\psi, \text{dag}_\varphi, 1), 1 \) because \( v_1^1 \) is the first vertex in both \( \text{dag}_\psi \) and \( \text{dag}_\varphi \). Since it only needs the weights relative to \( \psi \) cost function, this function requires \( \text{dag}_\psi \) only. The goal here is to find the best label for the vertex \( v_n^n \), so a list \( vLbl \) is maintained for the vertices’ labels. According to the optimization procedure relative to \( \psi \) in Sec. IV.4.1, \( vLbl (v_1^1) = 0 \) and all other labels are initially set to \( +\infty \). The vertices are ordered in a set \( T \) based on their order in \( \text{dag}_\psi \). Following the order of \( T \) and starting from the vertex right after \( v_1^1 \), each vertex is investigated as in Sec. IV.4.1, its label is stored in \( vLbl \) and its incoming edges are refined. After reaching the end of \( T \), the value at \( vLbl (v_n^n) \) represents the minimum cost of a path in \( \text{dag}_\psi \) from \( v_1^1 \) to \( v_n^n \) relative to \( \psi \), hereafter this value is named as optimal \( \text{Opt}_\psi \). The value \( \text{Opt}_\psi \) is returned along with the resulted subgraph of \( \text{dag}_\psi \), now named \( \Gamma_\psi \).

**V.2.5 Optimizing Relative to \( \varphi \)**

**V.2.5.1 Function Header**

\[
[\text{GAMAphi}, \text{OPTphi}] = \text{phi} (\text{DAGphi})
\]
Inputs:

1) DAGpsi: $dag_{\varphi}$.

Outputs:

1) GAMAphi: $\Gamma'_{\varphi}$.

2) OPTphi: $Opt_{\varphi}$.

V.2.5.2 Function Details

At first, both $dag_{\psi}$ and $dag_{\varphi}$ are pruned using $[dag_{\psi}, dag_{\varphi}]=$pruneGraph($dag_{\psi}, dag_{\varphi}$, 1), 1 because $v_1^1$ is the first vertex in both $dag_{\psi}$ and $dag_{\varphi}$. Since it only needs the weights relative to $\varphi$ cost function, this function requires $dag_{\varphi}$ only. The goal here is to find the best label for the vertex $v_n^n$, so a list $vLbl$ is maintained for the vertices’ labels. According to the optimization procedure relative to $\varphi$ in Sec. IV.4.2, $vLbl(v_1^1) = +\infty$ and all other labels are initially set to 0. The vertices are ordered in a set $T$ based on their order in $dag_{\varphi}$. Following the order of $T$ and starting from the vertex right after $v_1^1$, each vertex is investigated as in Sec. IV.4.2 and its label is stored in $vLbl$. After finishing the work on the last vertex in $T$, all the edges in $dag_{\varphi}$ that have weights $w_{\varphi}$ less than $vLbl(v_n^n)$ are removed. The value at $vLbl(v_n^n)$ represents the maximum cost of a path in $dag_{\varphi}$ from $v_1^1$ to $v_n^n$ relative to $\varphi$, hereafter this value is named as optimal $Opt_{\varphi}$. The value $Opt_{\varphi}$ is then returned along with the resulted subgraph of $dag_{\varphi}$, now named $\Gamma'_{\varphi}$.
V.2.6 Graph Update (updateGraph)

V.2.6.1 Function Header

\[
\text{[ G2 ] = updateGraph( G1, G2 )}
\]

Inputs:

1) G1: The graph that the function uses as reference for the update.
2) G2: The graph to be updated based on G1.

Outputs:

1) G2: The graph G2 after being updated.

V.2.6.2 Function Details

After each optimization procedure the returned DAG most likely has less number of edges than the original one given to the procedure. That means that the DAG that represents the weights for the other cost function has to be updated accordingly. For example, before applying any optimization procedure we have \( dag_\psi \) and \( dag_\varphi \). After applying the optimization relative to \( \psi \) on \( dag_\psi \) we get \( \Gamma_\psi \), which most likely has less edges than \( dag_\psi \). Now, before applying the optimization procedure relative to \( \varphi \) on \( dag_\varphi \), we need to update \( dag_\varphi \) so that it has the same edges contained in \( \Gamma_\psi \), not more not less. In this example, \( G1 = \Gamma_\psi \) and \( G2 = dag_\varphi \). The updating method is very straight forward; find all the edges that are not contained in \( G1 \) by simply finding the zeros, and then set those edges to zeros in \( G2 \). The same idea applies for the opposite scenario; optimizing for \( \varphi \) then \( \psi \), where \( G2 = dag_\varphi \) is updated according to \( G1 = \Gamma_\psi \).
V.2.7 Create Random Directed Graphs

V.2.7.1 Function Header

\[
[ G_{\psi}, G_{\phi} ] = \text{createDG}(n, e, A_{\psi}, B_{\psi}, A_{\phi}, B_{\phi})
\]

Inputs:

1) \( n \): The number of nodes required in the directed graph.

2) \( e \): The number of edges required in the directed graph.

3) \( A_{\psi} \): A lower bound on the value of the weights for the graph relative to \( \psi \).

4) \( B_{\psi} \): An upper bound on the value of the weights for the graph relative to \( \psi \).

5) \( A_{\phi} \): A lower bound on the value of the weights for the graph relative to \( \phi \).

6) \( B_{\phi} \): An upper bound on the value of the weights for the graph relative to \( \phi \).

Outputs:

1) \( G_{\psi} \): The matrix representing the weights of the edges for the graph relative to \( \psi \).

2) \( G_{\phi} \): The matrix representing the weights of the edges for the graph relative to \( \phi \).

V.2.7.2 Function Details

This function creates random directed graphs with \( n \) vertices and \( e \) edges. The edges are selected randomly, and each is given two random weights relative to both \( \psi \) and \( \phi \) respectively. The weights given to the edges are within the ranges given by the boundaries \( A_{\psi} \) and \( B_{\psi} \) for \( \psi \); \( A_{\phi} \) and \( B_{\phi} \) for \( \phi \).
V.3 Sequential Optimization Implementation

After defining the functions used by the algorithm, we can now explain how the sequential optimization is implemented in both possible orders for $\psi$ and $\varphi$. The program first acquires the original graph matrices $G_\psi$ and $G_\varphi$ along with $s$, $t$ and the execution order $r$, and then does some checking on the inputs for errors. The matrices are transformed to the DAGs using $[\text{dag}_\psi, \text{dag}_\varphi] = \text{getDAG}(G_\psi, G_\varphi, s, t)$. Depending on $r$, the program continues with one of the following scenarios:

V.3.1 Relative to $\psi$ then Relative to $\varphi$

The program executes the optimization in these steps:

1) $\text{dag}_\psi$ is optimized for $\psi$: $[\Gamma_\psi, \text{Opt}_\psi] = \psi(\text{dag}_\psi)$.

2) $\text{dag}_\varphi$ is updated according to $\Gamma_\psi$: $[\text{dag}_\varphi] = \text{updateGraph}(\Gamma_\psi, \text{dag}_\varphi)$.

3) $\text{dag}_\varphi$ is optimized for $\varphi$: $[\Gamma_\varphi, \text{Opt}_\varphi] = \phi(\text{dag}_\varphi)$.

4) $\Gamma_\psi$ is updated according to $\Gamma_\varphi$: $[\Gamma_\psi] = \text{updateGraph}(\Gamma_\varphi, \Gamma_\psi)$.

5) The values $\text{Opt}_\psi$ and $\text{Opt}_\varphi$ are returned.

At the end, the remaining paths in $\Gamma_\psi$ and $\Gamma_\varphi$ from $v_1^1$ to $v_n^n$ represent the optimal paths with respect to $\psi$ then $\varphi$.

V.3.2 Relative to $\varphi$ then Relative to $\psi$

The program executes the optimization in these steps:

1) $\text{dag}_\varphi$ is optimized for $\varphi$: $[\Gamma_\varphi, \text{Opt}_\varphi] = \phi(\text{dag}_\varphi)$.

2) $\text{dag}_\psi$ is updated according to $\Gamma_\varphi$: $[\text{dag}_\psi] = \text{updateGraph}(\Gamma_\varphi, \text{dag}_\psi)$.

3) $\text{dag}_\psi$ is optimized for $\psi$: $[\Gamma_\psi, \text{Opt}_\psi] = \psi(\text{dag}_\psi)$.
4) $\Gamma_\varphi$ is updated according to $\Gamma_\psi$ : $[\Gamma_\varphi] = \text{updateGraph}(\Gamma_\psi, \Gamma_\varphi)$.

5) The values $Opt_\varphi$ and $Opt_\psi$ are returned.

At the end, the remaining paths in $\Gamma_\varphi$ and $\Gamma_\psi$ from $v_1^1$ to $v_n^n$ represent the optimal paths with respect to $\varphi$ then $\psi$. 
Chapter 6

VI. EXPERIMENTS AND RESULTS

VI.1 Introduction
Two types of experiments were conducted to properly test the program: experiments on random graphs, and experiments on a Flight cost example.

VI.2 Experiments on Random Graphs
The sequential optimization experiments were conducted on random graphs with different sizes 20, 45 and 70 vertices, number of edges and values of weights. The graphs were created using \( [G_{\psi}, G_{\varphi}] = \text{createdDG}(n, m, A_{\psi}, B_{\psi}, A_{\varphi}, B_{\varphi}) \) with \( A_{\psi} = 1, B_{\psi} = 9, A_{\varphi} = 11, B_{\varphi} = 19 \). For each graph size different numbers of edges were tested, starting from somehow sparse graph to dense graph. The number of edges \( m \) is set between 30% to 90% of complete graph [42] which has \( m = n \times (n - 1) \). The results obtained are: number of paths from the source vertex \( s \) to the target vertex \( t \) in the original graph, number of paths from \( s \) to \( t \) after the first optimization, number of optimal paths from \( s \) to \( t \), costs of the optimal paths relative to both cost functions \( \psi \) and \( \varphi \), and the execution time in seconds. The next section Sec. VI.2.1 aims to study the execution time as the number of edges in a graph grows. Sec. VI.2.4 compares the time execution time as the size of the graph grows along with a fixed percentage of edges.

VI.2.1 Experiments
The following sections are ordered based on the size.
**VI.2.1.1 Graph with 20 Vertices**

Table VI.1 shows the results for the order $\psi$ then $\varphi$.

<table>
<thead>
<tr>
<th>NO. Edges</th>
<th>NO. Original Paths</th>
<th>NO. Middle Paths</th>
<th>NO. Optimal Paths</th>
<th>Cost Relative to $\psi$</th>
<th>Cost Relative to $\varphi$</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>127</td>
<td>2.64E+13</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>13</td>
<td>1.6746</td>
</tr>
<tr>
<td>170</td>
<td>4.75E+15</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>12</td>
<td>1.9441</td>
</tr>
<tr>
<td>213</td>
<td>3.60E+17</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>13</td>
<td>2.1473</td>
</tr>
<tr>
<td>256</td>
<td>1.26E+19</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>12</td>
<td>2.44</td>
</tr>
<tr>
<td>299</td>
<td>1.95E+20</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>12</td>
<td>2.748</td>
</tr>
<tr>
<td>342</td>
<td>2.29E+21</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>11</td>
<td>2.8882</td>
</tr>
</tbody>
</table>

Table VI.1 $n=20$, $\psi$ first

Table VI.2 shows the results for the order $\varphi$ then $\psi$.

<table>
<thead>
<tr>
<th>NO. Edges</th>
<th>NO. Original Paths</th>
<th>NO. Middle Paths</th>
<th>NO. Optimal Paths</th>
<th>Cost Relative to $\psi$</th>
<th>Cost Relative to $\varphi$</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>127</td>
<td>2.64E+13</td>
<td>6.89E+05</td>
<td>2</td>
<td>24</td>
<td>16</td>
<td>2.0847</td>
</tr>
<tr>
<td>170</td>
<td>4.75E+15</td>
<td>5</td>
<td>1</td>
<td>10</td>
<td>18</td>
<td>1.9249</td>
</tr>
<tr>
<td>213</td>
<td>3.60E+17</td>
<td>2.55E+06</td>
<td>1</td>
<td>11</td>
<td>17</td>
<td>2.5435</td>
</tr>
<tr>
<td>256</td>
<td>1.26E+19</td>
<td>1.04E+09</td>
<td>1</td>
<td>9</td>
<td>17</td>
<td>2.9821</td>
</tr>
<tr>
<td>299</td>
<td>1.95E+20</td>
<td>2.61E+06</td>
<td>2</td>
<td>14</td>
<td>18</td>
<td>2.9403</td>
</tr>
<tr>
<td>342</td>
<td>2.29E+21</td>
<td>1.52E+08</td>
<td>2</td>
<td>17</td>
<td>18</td>
<td>3.2771</td>
</tr>
</tbody>
</table>

Table VI.2 $n=20$, $\varphi$ first

Figure VI.1 compares the execution time for the two orders.
VI.2.1.2 Graph with 45 Vertices

Table VI.3 shows the results for the order $\psi$ then $\varphi$. 

<table>
<thead>
<tr>
<th>NO. Edges</th>
<th>NO. Original Paths</th>
<th>NO. Middle Paths</th>
<th>NO. Optimal Paths</th>
<th>Cost Relative to $\psi$</th>
<th>Cost Relative to $\varphi$</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>660</td>
<td>4.57E+48</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>14</td>
<td>60.661</td>
</tr>
<tr>
<td>880</td>
<td>2.05E+54</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>11</td>
<td>71.623</td>
</tr>
<tr>
<td>1100</td>
<td>3.73E+58</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>12</td>
<td>84.435</td>
</tr>
<tr>
<td>1320</td>
<td>1.06E+62</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>13</td>
<td>97.093</td>
</tr>
<tr>
<td>1540</td>
<td>8.62E+64</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>15</td>
<td>109.85</td>
</tr>
<tr>
<td>1760</td>
<td>3.82E+67</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>15</td>
<td>123.6</td>
</tr>
<tr>
<td>1980</td>
<td>6.61E+69</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>17</td>
<td>137.09</td>
</tr>
</tbody>
</table>

Table VI.3 n=45, $\psi$ first
Table VI.4 shows the results for the order $\phi$ then $\psi$.

<table>
<thead>
<tr>
<th>NO. Edges</th>
<th>NO. Original Paths</th>
<th>NO. Middle Paths</th>
<th>NO. Optimal Paths</th>
<th>Cost Relative to $\psi$</th>
<th>Cost Relative to $\phi$</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>660</td>
<td>4.57E+48</td>
<td>1.95E+17</td>
<td>1</td>
<td>19</td>
<td>18</td>
<td>66.182</td>
</tr>
<tr>
<td>880</td>
<td>2.05E+54</td>
<td>6.19E+20</td>
<td>1</td>
<td>18</td>
<td>18</td>
<td>79.935</td>
</tr>
<tr>
<td>1100</td>
<td>3.73E+58</td>
<td>12693</td>
<td>1</td>
<td>35</td>
<td>19</td>
<td>81.788</td>
</tr>
<tr>
<td>1320</td>
<td>1.06E+62</td>
<td>7.87E+27</td>
<td>1</td>
<td>12</td>
<td>18</td>
<td>110.46</td>
</tr>
<tr>
<td>1540</td>
<td>8.62E+64</td>
<td>7.32E+11</td>
<td>1</td>
<td>15</td>
<td>19</td>
<td>107.77</td>
</tr>
<tr>
<td>1760</td>
<td>3.82E+67</td>
<td>1.52E+12</td>
<td>2</td>
<td>26</td>
<td>19</td>
<td>120.29</td>
</tr>
<tr>
<td>1980</td>
<td>6.61E+69</td>
<td>1.21E+20</td>
<td>1</td>
<td>9</td>
<td>19</td>
<td>136.96</td>
</tr>
</tbody>
</table>

Table VI.4 \(n=45\), $\phi$ first

Figure VI.2 compares the execution time for the two orders.

![Execution Time, n=45](image-url)
VI.2.1.3 Graph with 70 Vertices

Table VI.5 shows the results for the order $\psi$ then $\varphi$.

<table>
<thead>
<tr>
<th>NO. Edges</th>
<th>NO. Original Paths</th>
<th>NO. Middle Paths</th>
<th>NO. Optimal Paths</th>
<th>Cost Relative to $\psi$</th>
<th>Cost Relative to $\varphi$</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1610</td>
<td>2.49E+91</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>12</td>
<td>518.72</td>
</tr>
<tr>
<td>2147</td>
<td>5.83E+99</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>17</td>
<td>633.95</td>
</tr>
<tr>
<td>2684</td>
<td>4.88E+106</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>15</td>
<td>744.99</td>
</tr>
<tr>
<td>3221</td>
<td>1.23E+112</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>14</td>
<td>872.37</td>
</tr>
<tr>
<td>3758</td>
<td>4.93E+116</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>12</td>
<td>985.62</td>
</tr>
<tr>
<td>4295</td>
<td>4.18E+120</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>19</td>
<td>1121</td>
</tr>
</tbody>
</table>

Table VI.5 $n=70, \psi$ first

Table VI.6 shows the results for the order $\varphi$ then $\psi$.

<table>
<thead>
<tr>
<th>NO. Edges</th>
<th>NO. Original Paths</th>
<th>NO. Middle Paths</th>
<th>NO. Optimal Paths</th>
<th>Cost Relative to $\psi$</th>
<th>Cost Relative to $\varphi$</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1610</td>
<td>2.49E+91</td>
<td>3.79E+57</td>
<td>1</td>
<td>8</td>
<td>17</td>
<td>665.59</td>
</tr>
<tr>
<td>2147</td>
<td>5.83E+99</td>
<td>3.22E+17</td>
<td>1</td>
<td>33</td>
<td>19</td>
<td>618.84</td>
</tr>
<tr>
<td>2684</td>
<td>4.88E+106</td>
<td>6.76E+20</td>
<td>1</td>
<td>28</td>
<td>19</td>
<td>739.66</td>
</tr>
<tr>
<td>3221</td>
<td>1.23E+112</td>
<td>2.92E+32</td>
<td>1</td>
<td>38</td>
<td>19</td>
<td>874.11</td>
</tr>
<tr>
<td>3758</td>
<td>4.93E+116</td>
<td>3.50E+30</td>
<td>1</td>
<td>21</td>
<td>19</td>
<td>985.61</td>
</tr>
<tr>
<td>4295</td>
<td>4.18E+120</td>
<td>2.91E+36</td>
<td>1</td>
<td>1</td>
<td>19</td>
<td>1113.1</td>
</tr>
</tbody>
</table>

Table VI.6 $n=70, \varphi$ first

Figure VI.3 Execution Time, $n=70$ compares the execution time for the two orders.
VI.2.2 Effect of Growing Number of Edges
Looking at each of the tables above, it is clear that even with a fixed number of vertices, the execution time is increasing with the increasing number of edges.

VI.2.3 Comparing Paths of Different Sequential Orders
By comparing the number of optimal paths obtained after optimizing relative to the first cost function only (NO. Middle Paths) between the first and second table for each \( n \), it is clear that the optimization relative to \( \varphi \) returns a big number of optimal paths compared to the ones returned by the optimization relative to \( \psi \). This is logical as the number of possible costs relative to \( \varphi \) is limited by the number of different edges’ weights \( w_{\varphi} \) relative to \( \varphi \). That explains why many paths have the same cost value relative to \( \varphi \). The tables coming second in each section above show the number of paths having the optimal cost relative to \( \varphi \).

This is completely different when optimizing relative to \( \psi \), as the number of possible costs can be as many as all the combinations, with different sizes, of the edges’ weights \( w_{\psi} \) relative to \( \psi \). That explains why it is very highly unlikely that a path would share the same cost value relative to \( \psi \) with another path. The tables coming first in each section above show the number of paths having the optimal cost relative to \( \psi \).

These observations will be discussed and utilized in Chapter VII with more details.
VI.2.4 Comparing Execution Time between Graphs

Figure VI.4 shows the trend of execution time as the size $n$ of the graph increases. Each graph has an $m$ equal to 67% of the complete graph. There is almost no difference in execution time between the two optimization orders. A trend line (Polynomial Order 3) was drawn to show the approximate growth trend of the execution time as the graphs grow larger. This trend

![Time for Graphs with 67% edges](image)

**Figure VI.4 Execution Time for Different Values of $n$**

VI.2.5 Example

This is a simple example executing sequential optimization on a graph with four vertices as shown in Figure VI.5, where the source is vertex one and the target is vertex four.
Figure VI.5 Original Graph, n=4

Figure VI.6 shows the DAG obtained from the original graph. The label 3'' means that this is vertex three in the second layer.

Figure VI.6 $\Gamma$ of n=4

Figure VI.7 shows the result after sequentially optimizing for $\psi$ then $\varphi$.

Figure VI.7 After $\psi$ then $\varphi$
Figure VI.8 shows the result after sequentially optimizing for $\varphi$ then $\psi$.

VI.3 Experiments on Flight Example

A directed graph representing fourteen cities - as vertices- and the flights between them – as edges- was created based on data collected from the website [43], part of it is shown in Figure VI.9. The fourteen cities are: Jeddah, Riyadh, Dammam, Cairo, Damascus, Moscow, Paris, NYC, Beirut, Frankfurt, Amman, Rome, Munich and Berlin. The total number of flights is 116. The cost functions that we look at for the journey starting from the departure city to the destination city are:

a) $\psi$: The total amount of money needed to pay for the flights in the journey.

b) $\varphi$: The baggage allowance on the flights throughout the journey.

The weights on the edges represent the following:

a) $w_\psi$: The cost of the flight in USD $.

b) $w_\varphi$: The maximum baggage allowance in KG for the flight.

VI.3.1 Experiments

Forty experiments were made on the Flight example with randomly selected departure and destination cities to test the execution time and paths. Both sequential optimization
orders were used. The results reported are: average number of original paths between the departure and destination cities, average number of paths remaining after the first optimization, average number of paths after both optimizations (end of sequential) and average execution time. Table VI.7 shows the results for both execution orders.

<table>
<thead>
<tr>
<th>Execution Order</th>
<th>Avg. NO. Original Paths</th>
<th>Avg. NO. Middle Paths</th>
<th>Avg. NO. Optimal Paths</th>
<th>Avg. Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\psi$ then $\varphi$</td>
<td>1.95E+10</td>
<td>1.025</td>
<td>1.025</td>
<td>0.724914</td>
</tr>
<tr>
<td>$\varphi$ then $\psi$</td>
<td>1.95E+10</td>
<td>2.39E+09</td>
<td>1</td>
<td>0.861109</td>
</tr>
</tbody>
</table>

Table VI.7 Flight Example, Avg. 40 Routes

VI.3.2 Example

The user is first shown the graph with fourteen vertices; each vertex is labeled with the name of the city it represents and each edge is labeled by $w_\psi$ and $w_\varphi$ corresponding to the flight it represents.
A selection box shown in Figure VI.10 appears to let the user select the departure city, for example here Paris is selected. After that another selection box appears as in Figure VI.11 so the user can select the destination city, for example here Jeddah is selected. Notice here that the departure city Paris is replaced with the label “FROM” in the list of the second selection box.
The program shows the user the flights graph with the departure and destination cities labeled as “FROM” and “TO” respectively.
The program executes the sequential optimization in both orders then shows the results for each of them.
VII. FUNCTIONS TRANSFORMATION

VII.1 Introduction

When observing the results of executing the sequential optimization it is clear that the order of execution directly affects the values of the costs of the optimal paths. Executing the optimization procedure with respect to $\psi$ then $\varphi$ gives optimal values for the cost relative to $\psi$ and the cost relative to $\varphi$ that are always less than or equal to the ones given by the order $\varphi$ then $\psi$. On the other hand, executing the optimization procedure with respect to $\varphi$ then $\psi$ gives optimal values for the cost relative to $\psi$ and the cost relative to $\varphi$ that are always greater than or equal to the ones given by the order $\psi$ then $\varphi$. In both cases the algorithm gives one solution that seems to favor the function that is executed first and is not fair for both cost functions.

One possible solution to get unbiased results, is to find the costs for all paths then pick the one or ones that have costs fair to both $\psi$ and $\varphi$. But that solution is very computationally expensive. Here we will state another solution: to first define two functions on the set of all possible cost values for $\psi$ and $\varphi$, respectively, then come up with a relation between those two new functions so that each of them can be found in terms of the other one. Using this solution it would be easy to get Pareto-set for the two cost functions and draw the Pareto-curve. Here the functions are first defined, their
relation is then stated and proved, and then some experiments are made using the relation on random graphs and the flight example.

VII.2 Functions Definitions

In order to define the functions, let us first list some definitions:

- $P: P(\Gamma)$ the set of all paths from $s$ to $t$ in $\Gamma$.
- $\psi$ and $\varphi$: we can define them as functions from $P$ onto $C_\psi$ and $C_\varphi$, respectively.
- $C_\psi$ and $C_\varphi$: finite sets of nonnegative integers.
- $m_\psi: \min\{m: m \in C_\psi\}$.
- $n_\varphi$: a natural number such that $n \geq n_\varphi \forall n \in C_\varphi$. It can interpreted as the minimum cost possible for $\varphi$.
- $M_\psi$: a natural number such that $m \leq M_\psi \forall m \in C_\psi$. It can be interpreted as the maximum cost possible for $\psi$.
- $N_\varphi$: $\max\{n: n \in C_\varphi\}$.
- $B_\psi = \{m_\psi, m_\psi + 1, ..., M_\psi\}$.
- $B_\varphi = \{n_\varphi, n_\varphi + 1, ..., N_\varphi\}$.

We can now define the two new functions:

VII.2.1 Function $\Phi$

This function $\Phi: B_\psi \to B_\varphi$ is defined in terms of the cost functions $\psi$ and $\varphi$:

$$\Phi(m) = \max\{\varphi(a): a \in P, \psi(a) \leq m\}, \ \forall m \in B_\psi$$
To find the value $\Phi(m)$ for a given $m$, we first need to find all $a \in P$ which have

$\psi(a) \leq m$ meaning that we find $\psi(a)$ for all $a \in P$ and put the ones that have $\psi(a) \leq m$
in a set $H_m$, after that we find $\varphi(a)$ for all $a \in H_m$ and the result will be the maximum

$\varphi(a)$. Based on the cost values relative to both $\psi$ and $\varphi$ it is safe to write that $\Phi$ is a non-decreasing function.

**VII.2.2 Function $\Psi$**

This function $\Psi: B_\varphi \rightarrow B_\psi$ is defined in terms of the cost functions $\psi$ and $\varphi$:

$$\Psi(n) = \min\{ \psi(a): a \in P, \varphi(a) \geq n \}, \ \forall n \in B_\varphi$$

To find the value $\Psi(n)$ for a given $n$, we first need to find all $a \in P$ which have $\varphi(a) \geq n$
meaning that we find $\varphi(a)$ for all $a \in P$ and put the ones that have $\varphi(a) \geq n$ in a set
$H_n$, after that we find $\psi(a)$ for all $a \in H_n$ and the result will be the minimum $\psi(a)$.

Based on the cost values relative to both $\psi$ and $\varphi$ it is safe to write that $\Psi$ is a non-decreasing function.

**VII.3 Functions Relation**

From the definitions of the functions $\Psi$ and $\Phi$ we can find that they are related in a sense
that each of them can be used to find the other, meaning that by knowing $\Psi$ it is easy to
evaluate $\Phi$, and vice versa. The relation is first defined and then used to get fair values
for the cost functions $\psi$ and $\varphi$.

**Proposition.** For any $m \in B_\psi$:

$$\Phi(m) = \max\{ n \in B_\psi: \Psi(n) \leq m \}$$

(3)
And for any \( n \in B_\varphi \):

\[
\Psi(n) = \min \{ m \in B_\psi : \Phi(m) \geq n \}
\]  
(4)

**Proof.** Let for some \( m \in B_\psi \),

\[
\Phi(m) = n_0
\]  
(5)

And also let

\[
\max \{ n \in B_\varphi : \Psi(n) \leq m \} = t
\]

From (5) it follows that

(i) There exists \( b \in P \) such that \( \psi(b) \leq m \) and \( \varphi(b) = n_0 \).

(ii) For any \( a \in P \), if \( \psi(a) \leq m \) then \( \varphi(a) \leq n_0 \).

From (i) it follows that \( \Psi(n_0) \leq m \), which implies \( t \geq n_0 \). Let’s assume that \( t > n_0 \), in this case there exists \( n_1 > n_0 \) for which \( \Psi(n_1) \leq m \). Therefore, there exists \( a \in P \) such that \( \varphi(a) \geq n_1 \) and \( \psi(a) \leq m \), but this contradicts with \( \varphi(a) \leq n_0 \) from (ii). So it must be that \( t = n_0 \). Similarly, Equation (4) can be proven.

Using the proposition, the two functions can be transformed from each one to the other as the following:

**VII.3.1 Transform \( \Psi \) into \( \Phi \)**

The function \( \Psi \), given by a tuple \( (\Psi(n_\varphi), \Psi(n_\varphi + 1), ..., \Psi(N_\varphi)) \), can be transformed into the function \( \Phi \). Since the function \( \Psi \) is non-decreasing, it is clear that \( \Psi(n_\varphi) \leq \]
\[ \Psi(n_{\Phi} + 1) \leq \cdots \leq \Psi(N_{\Phi}). \]

According to Equation (3), to find \( \Phi(m) \) for a given \( m \in B_{\psi} \), we seek to find the maximum \( n \in B_{\phi} \) such that \( \Psi(n) \leq m \). This can be found by performing binary search [9] on the \( \Psi \) tuple. Binary search time complexity is \( O(\log |B_{\phi}|) \) comparisons, because in the worst case scenario the \( \Psi \) tuple has the same size as \( B_{\phi} \).

**VII.3.2 Transform \( \Phi \) into \( \Psi \)**

The function \( \Phi \), given by a tuple \( (\Phi(m_{\psi}), \Phi(m_{\psi} + 1), \ldots, \Phi(M_{\psi})) \), can be transformed into the function \( \Psi \). Since the function \( \Phi \) is non-decreasing, it is clear that \( \Phi(m_{\psi}) \leq \Phi(m_{\psi} + 1) \leq \cdots \leq \Phi(M_{\psi}) \). According to Equation (4), to find \( \Psi(n) \) for a given \( n \in B_{\psi} \), we seek to find the minimum \( m \in B_{\psi} \) such that \( \Phi(m) \geq n \). This can be found by performing binary search on the \( \Phi \) tuple. Binary search time complexity is \( O(\log |B_{\psi}|) \) comparisons, because in the worst case scenario the \( \Phi \) tuple has the same size as \( B_{\psi} \).

**VII.4 Application of Functions Relation**

Finding the cost values that are fair for both cost function requires finding the values of \( \Psi \) for all \( n \in B_{\phi} \) and the values of \( \Phi \) for all \( m \in B_{\psi} \) using the functions’ definitions, but this can be extremely expensive. However, using the relation between the two functions \( \Psi \) and \( \Phi \) enables us to get such fair cost values by only finding the values of \( \Psi \) for all \( n \in B_{\phi} \) using its definition, and then get the values of \( \Phi \) for all \( m \in B_{\psi} \) using the relation, or by only finding the values of \( \Phi \) for all \( m \in B_{\psi} \) using its definition, and then get the values of \( \Psi \) for all \( n \in B_{\phi} \) using the relation.
Based on the nature of the cost function $\varphi$, we can find that its possible cost values are discrete and are not large in number. This is true since in the worst case scenario, where each edge in the original graph $G$ has a unique weight $w_{\varphi}$, the number of all possible cost values for $\varphi$ will not exceed the number of edges in $G$. This is not true though for $\psi$, which has exponential number of possible cost values. So to have the best performance when finding the cost values that are fair to both cost functions it is better to find the tuple containing the values of $\Psi$ for all $n \in B_{\varphi}$ using its definition, and then use the relation to get the tuple of $\Phi$. This process is done following these steps:

1) A list $uniq_{\varphi}$ containing all pairwise different values of $w_{\varphi}$, in ascending order, is obtained.

2) A two columns list $Tup_{\psi}$ is created, with the first column as $uniq_{\varphi}$.

3) For every $x_{\varphi} \in uniq_{\varphi}$, in order, we do the following:
   a) Remove from the $dag$ all edges that have $w_{\varphi} < x_{\varphi}$.
   b) Get the optimal values $Opt_{\psi}$ and $Opt_{\varphi}$ by running the optimization procedure relative to $\psi$ and then $\varphi$.
   c) Store $Opt_{\psi}$ in $Tup_{\psi}$ next to the corresponding $x_{\varphi}$.
   d) Replace the $x_{\varphi}$ value in $Tup_{\psi}$ with the obtained $Opt_{\varphi}$.

4) Using the relation as in Sec. VII.3.1 on $Tup_{\psi}$ we can transform $\Psi$ into $\Phi$ for any wanted values of $m \in B_{\psi}$.

The tuple $Tup_{\psi}$ actually represents the Pareto-set, as any row in it is Pareto-optimal since it cannot be enhanced with respect to $\varphi$, by getting a larger $Opt_{\varphi}$, without getting a larger
(worse) value with respect to \( \psi \); and in the same time it cannot be enhanced with respect to \( \psi \), by getting a smaller \( \text{Opt}_\psi \), without getting less (worse) value with respect to \( \varphi \).

**VII.5 Implementation**

In addition to the MATLAB ® functions mentioned before in Chapter 5, some function were implemented for the relation between \( \Psi \) and \( \Phi \).

**VII.5.1 Get \( \Psi \) Tuple (getPsiTuple)**

```
[ tuplePsi ] = getPsiTuple ( DAGpsi, DAGphi)
```

**VII.5.1.1 Function Header**

Inputs:

1) \( \text{DAGpsi} \): \( \text{dag}_\psi \).
2) \( \text{DAGpsi} \): \( \text{dag}_\varphi \).

Outputs:

1) \( \text{tuplePsi} \): \( \text{Tup}_\psi \).

**VII.5.1.2 Function Details**

This function work is done following these steps:

1) A list \( \text{uniq}_\varphi \) containing all pairwise different values of \( w_\varphi \), in ascending order, is obtained.

2) A two columns list \( \text{Tup}_\psi \) is created, with the first column as \( \text{uniq}_\varphi \).

3) For every \( x_\varphi \in \text{uniq}_\varphi \), in ascending order, we do the following:
a) Remove from $dag_\psi$ $dag_\varphi$ and all edges that have $w_\varphi < x_\varphi$.

b) Get the optimal values $Opt_\psi$ and $Opt_\varphi$ by running the optimization procedure relative to $\psi$ and then $\varphi$.

c) Store $Opt_\psi$ in $Tup_\psi$ next to the corresponding $x_\varphi$.

d) Replace the $x_\varphi$ value in $Tup_\psi$ with the obtained $Opt_\varphi$.

4) The tuple $Tup_\psi$ is returned.

VII.5.2 Transform $\Psi$ into $\Phi$

$$[\text{tuplePhi}] = \text{psiToPhi}(\text{valuesPsi}, \text{tuplePsi})$$

VII.5.2.1 Function Header

Inputs:

1) valuesPsi: $values_\psi$.

2) tuplePsi: $Tup_\psi$.

Outputs:

1) tuplePhi: $Tup_\varphi$.

VII.5.2.2 Function Details

The function works as follows:

1) A tuple with two columns $Tup_\varphi$ is created with the first column as $values_\psi$.

2) For each value $m \in values_\psi$ the function does the following:

   a. Apply binary search on the second column of $Tup_\psi$ to find the maximum value $n \leq m$. 
b. The corresponding value to \( n \) in the first column is stored in the second column of \( T_{up_\varphi} \) next to \( m \).

3) The tuple \( T_{up_\varphi} \) is returned.

VII.6 Experiments

To show the relation between \( \Psi \) and \( \Phi \), some experiments were made on random graphs and on the flights example.

VII.6.1 Experiments on Random Graphs

Some random graphs were created with sizes varying from five vertices to 50 vertices using \([G_\varphi, G_\varphi]=\text{createdDG}(n, m, A_\psi, B_\psi, A_\varphi, B_\varphi)\) with \( A_\psi = 1, B_\psi = 9, A_\varphi = 11, B_\varphi = 19.\) Each graph had the number of edges \( m = 67\% \) of a complete graph. For each graph the program first finds the tuple \( T_{up_\psi} \), then finds the other one \( T_{up_\varphi} \). The results shown here are: the time required to find \( T_{up_\psi} \) along with its size, and the time required to get \( T_{up_\varphi} \). The values selected for \( values_\psi \) were from one to 101 with step of ten.

<table>
<thead>
<tr>
<th>NO. Vertices</th>
<th>NO. Edges</th>
<th>Size of ( T_{up_\psi} ) (no. rows)</th>
<th>Time for ( T_{up_\psi} ) (s)</th>
<th>Time for ( T_{up_\varphi} ) (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>14</td>
<td>1</td>
<td>0.00075899</td>
<td>1.76E-03</td>
</tr>
<tr>
<td>10</td>
<td>61</td>
<td>7</td>
<td>0.14202</td>
<td>1.07E-04</td>
</tr>
<tr>
<td>15</td>
<td>141</td>
<td>7</td>
<td>0.62283</td>
<td>1.05E-04</td>
</tr>
<tr>
<td>20</td>
<td>255</td>
<td>8</td>
<td>1.5398</td>
<td>9.86E-05</td>
</tr>
<tr>
<td>25</td>
<td>402</td>
<td>9</td>
<td>3.2343</td>
<td>1.18E-04</td>
</tr>
<tr>
<td>30</td>
<td>583</td>
<td>8</td>
<td>9.8776</td>
<td>1.08E-04</td>
</tr>
<tr>
<td>35</td>
<td>798</td>
<td>8</td>
<td>21.054</td>
<td>1.08E-04</td>
</tr>
<tr>
<td>40</td>
<td>1046</td>
<td>8</td>
<td>34.122</td>
<td>1.13E-04</td>
</tr>
</tbody>
</table>
The results in Table VII.1 clearly show the impact of using the functions’ relation. The time needed to find $Tup_\psi$ depends heavily on the size of the graph and the number of different values for $w_\psi$. Whereas the time needed to get $Tup_\psi$ only depends on the size of $values_\psi$, determines how many times the binary search is executed, and the size of $Tup_\psi$ since the binary search is applied on it. However, because the size of $values_\psi$ is a constant here, the change in execution time for $Tup_\psi$ reflects the changing size of $Tup_\psi$.

Figure VII.1 and Figure VII.2 show the second column vs. the first column for $Tup_\psi$ and $Tup_\varphi$, respectively, from a graph with 50 vertices and using values for $w_\varphi$ between $A_\varphi = 11$, $B_\varphi = 30$. The curve in Figure VII.1 is the Pareto-curve for the two cost functions on this graph.
Figure VII.1 Tuple $T_{up_{\psi}}$ (Pareto-curve)

Figure VII.2 Tuple $T_{up_{\psi}}$
VII.6.2 Flight Example

In this example the departure city is “Beirut” and the destination city is “Frankfurt”.

These are the results for seeking best ticket fare or best baggage allowance:

<table>
<thead>
<tr>
<th>Flying from: Beirut, to: Frankfurt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Best Ticket Fare:</td>
</tr>
<tr>
<td>• Fare: 707 $</td>
</tr>
<tr>
<td>• Baggage Allowance: 20 KG</td>
</tr>
<tr>
<td>• Route: Beirut-&gt;Jeddah-&gt;Frankfurt</td>
</tr>
<tr>
<td>2) Best Baggage Allowance:</td>
</tr>
<tr>
<td>• Fare: 721 $</td>
</tr>
<tr>
<td>• Baggage Allowance: 23 KG</td>
</tr>
<tr>
<td>• Route: Beirut-&gt;Frankfurt</td>
</tr>
</tbody>
</table>

Here Figure VII.3 shows all the possible ticket fares depending on baggage allowance, and Figure VII.4 shows all possible baggage allowances based on the ticket fare.
Figure VII.3 Fare vs. Baggage

Figure VII.4 Baggage vs. Fare
Chapter 8

VIII. CONCLUSION

The thesis described the implementation of an algorithm based on dynamic programming that solves the optimization on directed graphs sequentially. This implementation was used to conduct experiments to see how the algorithm works and the time complexity is needed. The thesis also presented a relation between the two cost functions that the algorithm optimizes for. This relation is proved and implemented and its implementation is used to experiment it to see how the relation is used to save execution time in finding fair cost values relative to the cost functions.

It is worth mentioning that the algorithm can be used for as many cost functions as needed. A graph, representing a certain problem, which cannot fit directly into the algorithm because of either one of the constraints on the input (no loop edges; no multiple edges); can still be transformed to be able to fit by simply creating different copies of the vertices violating the constraints. Another technique called “Node Splitting” [44], [45] can also be used for a problem where there is a weight on the vertices themselves, the first copy of the edge has the incoming edges and the other has the outgoing edges and they are both connected with an edge which carries the weight originally on the vertex.
REFERENCES


