Navigation and Flow Algorithms

Building a Navigation Framework

M.Sc. Thesis
INF/SCR-08-20

Wouter Penard, wrjpenard@gmail.com
Center for Algorithmic Systems - Utrecht University

December, 2008

Utrecht University

Daily Supervisor:
dr. Hans Bodlaender
hansb@cs.uu.nl

Second Supervisor:
dr. Jurriaan Hage
jur@cs.uu.nl

Logica

Daily Supervisor:
ir. Mark Olthof
mark.olthof@logica.com

Second supervisor:
drs. Herbert Leenstra
herbert.leenstra@logica.com

Abstract

Incited by the observation that in recent years positioning and navigation have become increasingly pervasive, we designed and constructed a navigation framework. This framework aims to provide advanced navigation functionality by solving various flow problem. In order to solve these problems, it supports implementations of Dijkstra's algorithm, the Push-Relabel and Successive-approximation by Cost Scaling algorithms. These algorithms solve the shortest path, maximum flow and minimum-cost flow problems respectively. The framework also contains functionality to create a time-expanded graph. Solutions to dynamic problems, such as the quickest flow and earliest arrival problems can be found by executing the Successive-approximation by Cost Scaling algorithm on this time-expanded graph. Furthermore, the performance of the Push-Relabel and Successive-approximation by Cost Scaling algorithms was studied empirically.

Based on the framework and these algorithms an application was constructed that simulates an evacuation from a building. Such an evacuation can be modeled as quickest flow problem. The application uses the framework to construct a model of the building and to calculate a solution to this quickest flow problem. This simulation is created based on the solution of this problem. This evacuation application shows that the framework itself and the algorithms that it implements can be used in a practical application.

Contents

List of Tables

List of Figures

Part I

Project

Chapter 1

Introduction

This chapter provides the context of the problem, which is followed by a discussion of the problem addressed by this thesis. Based on the problem description the research questions, objectives and scope of the project are inferred. The chapter is concluded by an overview of the structure of the document.

1.1 Context

1.1.1 Location Based Services

In the last decade mobile networks have become increasingly pervasive and with the introduction of new broadband technologies for mobile network, new types of services become feasible. One such type are Location Based Services (LBS). LBS are services that incorporate location information with contextual data to provide a value-added experience to users on the web and wireless devices[45]. A wide range of applications based on LBS is conceivable. A common classification is shown in Table 1.1[55]. As shown in Figure 1.1 navigation services form a significant part of the estimated LBS revenue. This thesis focuses on this particular application segment.

Table 1.1: LBS: application segments

Segment	Description
Navigation:	Map display, turn-by-turn navigation
Entertainment:	Instant messaging, dating, games
Tracking:	People, asset tracking, emergency services
Information:	Local search, weather, traffic
Location Based Billing:	Voice and data transfer at location dependent tariffs

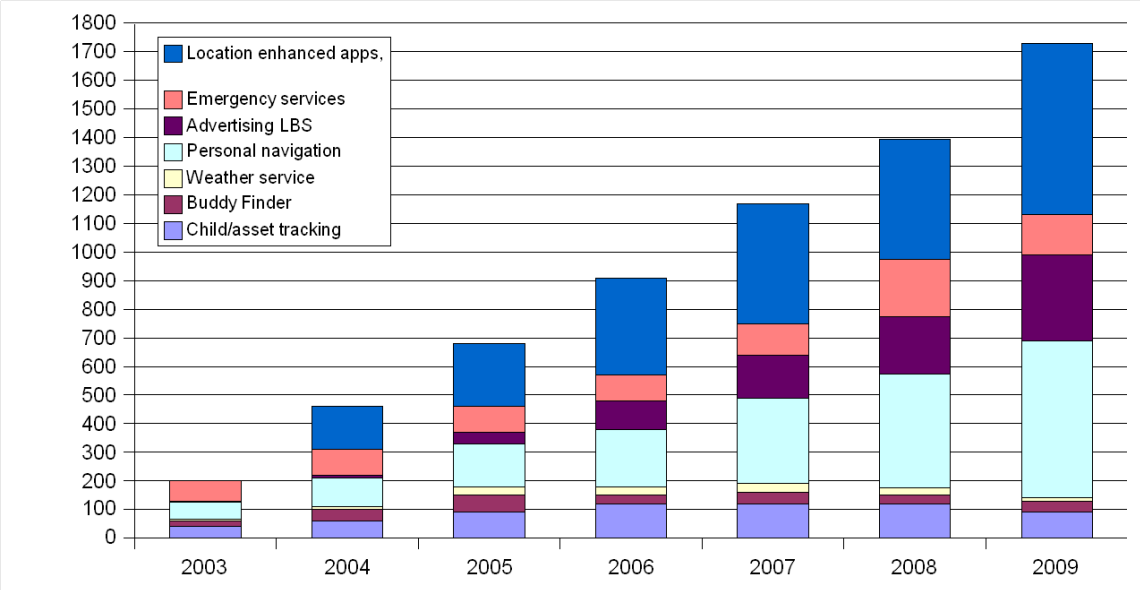

Figure 1.1: LBS : estimated market revenues [56]

1.1.2 Navigation

Recent years have seen a rapid increase in the use of Global Positioning System (GPS) based navigation for outdoor environments. First introduced within navigation system for cars, we now see cellphones with GPS modules being introduced. However, the use of GPS for positioning and navigation has shortcomings. Most importantly, its signal degrades easily [11] and is not usable indoors. For this reason among others alternate methods for positioning are an active area of research. Possibilities include the use of terrestrial Radio Frequency (RF) signals (also called "pseudolites") to augment GPS indoors; or the use of various wireless communication technologies, such as cellular networks [32], WLAN [54, 44] or Bluetooth [34]. Rizos [57] gives an overview of positioning systems and makes a key distinction between user-centric and network-centric positioning systems. In an user-centric scenario the mobile device actively detects its own position, and as a result most of the computational effort is located at the mobile device. On the other hand, in a network-centric positioning system the network, or a system within the network, is responsible for determining the device's position. An example of the former is GPS, while the latter is typical for cellular networks, WLAN and Bluetooth.

1.2 Problem Description

The observations that LBS and navigation services have great potential makes this an exciting area of research. During an early brainstorming session it became clear there exist numerous applications that contain a navigation element. Some examples include:

- Position someone with bluetooth and present a route to some destination within a building. The destination could be a room or a moving person.

- Set within a themepark. Propose a route to an attraction based on the visitors preferences, traveling time to attractions and the waiting queues in front of attractions.
- Track which booths have been visited by a visitor on a fair and propose routes to booths which are of interest and have not been visited yet.
- Give people, that are present in a complex, directions, such that the time required to evacuate the building is minimized.

1.2.1 Navigation Framework

We previously noted there exists a wide range of technologies based on which a users position can be determined, while at the same time a number of applications can benefit from navigation support. In order to provide this functionality we intend to construct a framework which is independent from the specific method used to determine the user's position. Hence, the framework will be applicable for a wide range of applications. Apart from offering plain navigational functionality we wish to make use of the additional possibilities enabled by network-centric positioning.

By exploiting the positioning information of groups of users the navigational feedback can be enhanced. We envision using this position data to model traffic flows and anticipate future flows. This anticipated state can then be used to update the navigational feedback. In addition to the positioning method, we identify two more application specific aspects. The framework will abstract away from these aspects. Specifically, the general map context, which is used to plan routes, and the components which further processes the output of the framework. We motivate each shortly.

In order to plan routes the framework has to be supplied with map information, however there exist a multitude of different data formats (for example shape files[1] and GML[2]) and geographic coordinate systems (such as WGS84[3], which is used worldwide, and the RD-system[4] used in the Netherlands). Applications using the framework have to provide a component which translates data from the specific data sources to the internal model used within the framework. In this way the framework provides maximum flexibility.

How a particular application uses the output of the framework is of course application specific. Traditionally one can think of an application displaying a map of the environment and drawing a route on this map. However, the information can be relayed by other media, such as SMS, MMS or a web based service.

Finally, conventional navigation software execute on a static model of the environment, whereas our framework will provide functionality to dynamically modify the model and properties of the model. One can think of temporarily broken connections, for example due to road maintenance, which trigger such a modification.

Fayad *et al.*[22] define a framework as a "reusable, semi-complete application that can be specialized to produce custom applications." Using our framework it is possible to create a custom application by, minimally, providing a component which creates a to the framework understandable model from a data source and a component which further processes the output as returned by the framework.

[1] ESRI Shapefile [21]
[2] Geography Markup Language [51]
[3] World Geodetic System 84 [48]
[4] Rijksdriehoekscordinaten [16]

1.2.2 Graph Algorithms

In this section we give an informal introduction to graphs and relevant graph problems and how our practical problem at hand relates to these. A formal description is given in Chapter 2 and Chapter 4.

Informal Introduction

In common navigation systems the environment is modeled as a graph, and graph algorithms are used to calculate the route from a source node to a destination. A graph is a collection of nodes which are connected to other nodes by arcs. Properties can be associated with these nodes and arcs. Dijkstra's algorithm [62] is used to solve the problem of finding a shortest path between two nodes in such a graph. A number of effective heuristics which speed-up Dijkstra's algorithm are known. These often employ pruning to reduce the search space by adding precomputed data in the form of annotations to the graph [62].

To support the novel functionality proposed in Section 1.2.1 we require a more advanced problem model, such as maximum flow or minimum-cost flow problem. While the shortest path (SP) problem has weights associated with each arc. A capacity is associated with each arc in the maximum flow problem. A flow can be send across such an arc. This flow can never exceed the capacity of the concerned arc. The problem is to find the maximum amount of flow that can be send between two distinct nodes in the graph. Specifically, from a flow source node, which produces flow, to a sink node, which consumes flow. The minimum-cost flow problem associates a cost (or weight), in addition to the capacity, with each arc. Here the source produces a fixed amount of flow, where as this amount of flow was variable in the maximum flow problem. This fixed amount of flow is also called demand, where a negative demand indicates a surplus. The minimum-cost flow problem is to find a flow on the given graph of minimum cost.

Relation to Navigation Problem

Finding the shortest route between two places in the real world translates easily to a graph problem. Locations are mapped onto nodes and arcs indicate a connection between two locations. The travel time or distance between two locations can be modeled by the weight associated with each arc. The relation with the maximum flow and minimum-cost flow problems is also intuitive. The capacity, or maximum throughput of a connection can be modeled as the capacity of the corresponding arc. We can represent the number of entities at a certain location as a negative demand (or surplus) and their destination with a positive demand. The solution of the minimum-cost flow problem corresponds to a routing of entities in the real world which minimizes (for example) travel time.

Changes in the real world situation can be reflected by modifying properties of arcs or even the structure of the graph. One can think for example of changes in expected travel time or the temporary closure of a road, leading to a modification of the cost associated with an arc or the deletion of an arc respectively.

1.3 Research Questions and Objectives

The problem description in the previous section gives rise to the fundamental questions we aim to address with this work. In this section we first discuss these research questions. Based on these

questions we formulate the objectives we intend to accomplish with the project and finally we define the scope of the project.

1.3.1 Research Questions

The main question of this work is listed below and is derived from the earlier problem description (Section 1.2). In the subsequent paragraph we further divide this question in more detailed subquestions.

> *Construct a navigational framework incorporating dynamic aspects as previously explained. How can existing theoretical graph algorithms be applied to support the required functionality provided by the framework?*

Central is the application of theoretic algorithms, this aspect is further emphasized by the following subquestions.

1. What are the state-of-the-art algorithms and heuristics for the single-source shortest path, maximum flow and minimum-cost flow problems?
2. Which of these algorithms are best suited to be implemented?
3. Can these algorithms be implemented such that they support the framework?
4. How can we introduce the notion of time in these algorithms?
5. In what way can we incorporate dynamic changes of structure and properties?
6. What is the measured performance of the maximum and minimum-cost flow algorithms that are implemented?
7. What are the requirements for such a framework?
8. What design satisfies the requirements?
9. Can we construct such a framework?

Questions 1-6 are related to the algorithmic solution to the problem, while those after 6 concern the design and construction of the framework.

1.3.2 Objectives

This section lists the main objectives of the project.

1. Determine and describe the state-of-the-art algorithms for the maximum flow and minimum cost flow problems. Based on this description, the requirements and constraints of the framework it will be decided which algorithms are best suited to be implemented in the framework.

2. Derive and describe requirements for a framework as described in the problem description and produce a design based on these requirements.

3. Report on the algorithmic implementation details and how they address research questions stated in the previous section.

4. Perform a first iteration in the development of the framework.

5. Report on the performance of the implemented algorithms by empirical study. The purpose of this is two fold: first it provides insight into the implementation which can be used to improve it. Secondly, it provides additional insight into the performance of theoretical algorithms in a practical setting.

6. Construct a prototype application. Do this, with the purpose of gaining insight into the framework and use this additional insight to identify and propose improvements to the framework.

1.4 Project Environment and Scope

The project is part of the Master of Science programme Applied Computing Science at Utrecht University and will be conducted at Logica, division Energy, Utility and Telecom (EUT), within the Working Tomorrow program. Logica is a leading IT and business services company and provides business consulting, system integration and IT and business process outsourcing services. The Working Tomorrow program is intended for graduate students in Dutch higher professional and scientific education. The program allows students to perform innovative projects in an open, professional environment. In general these projects are not performed in collaboration with customers, which allows for considerable freedom. The duration of the graduation project is 9 months, starting at February 1st, 2008.

1.4.1 Scope

Our primary focus is the algorithmic realization of the project and in particular the utilization of maximum and minimum-cost flow algorithms in a practical environment. We will extensively discuss the state-of-the-art in the area of SP algorithms and heuristics used to improve their performance in the domain of navigation. However, after implementing a basic SP algorithm, our attention will shift to the implementation of the flow algorithms and we will leave the implementation of heuristics for the SP problem to future iterations.

Am important limitation is the duration of the project. Fayad *et al.*[22] argue that the inception of a framework is a complicated task which requires a thorough domain understanding and multiple iterations. They advise to first use the framework to construct one or more relatively small pilot projects and use the experience gained during these projects to further improve and refine the framework. We view our project in this light. It is impossible to construct a mature framework in the given time period. However in order to validate and gain insight into the practical applicability of the framework we will construct a small prototype application.

In Section 1.2 we indicated several spots within the framework which require an application specific implementation. It is not our intention to provide a default implementation for these,

and consequently a simulation or a minimal ad-hoc implementation will suffice in the prototype application.

Finally, the project is a purely technical study, it is not the intention to perform a market analysis or derive any business cases.

1.5 Evacuation Simulation

One of the objectives of this project is to construct a prototype application based on the framework. In this section we further describe this application. It is our aim to construct an application which simulates the evacuation of an office building. Because the application is intended to be a prototype, no complete design is presented. However, some functional requirements are listed.

Description

In event of an emergency a building needs to be evacuated as quickly and efficiently as possible. A mass stampeding through hallways and stairwells can easily cause catastrophic congestions. With the purpose of preventing such a disaster, evacuation plans are devised and based on these directions are offered to people present. Such a plan can be constructed based on evacuation simulations. An evacuation scenario can be modeled as a graph problem, specifically as a quickest flow problem[12]. In a traditional case these simulations are run upfront (referred to as offline) for example during construction of the building. It will result in a static evacuation plan and based on this plan directions are fixed to walls of the building.

As an innovative approach we consider the use of novel technology to improve these directions. Given the ability to determine the location of people present in the building we can use this information to simulate an evacuation during the disaster (online). Based on the results of this simulation we can present people with improved directions. Additionally, it is quite conceivable, that certain routes are inaccessible due to the disaster. By, in a sense, constructing an evacuation plan during the evacuation we can take these blocked routes into account and deliver more accurate directions to the evacuees and update these directions as the situation changes. The directions can be presented to people by means of auditive signals (which might also be given by rescue workers), dynamic signs based on for example LCD screens, or individually to people by means of their cellphone.

Requirements

- The prototype should be a demonstrable graphical application.
- Load and visualize a graph from a file.
- Able to save the current graph to a file.
- Connect the graph with a map of the building and show this map.
- Modification to the graph structure and properties can be made using the GUI.
- It is possible to assign 'people' to various rooms within the building.
- Multiple exits can be created.

- The application can show the simulation of the evacuation from the building.

These requirements impose no further requirements on the framework itself, other then that algorithms have to be implemented that are able to solve such an evacuation problem.

1.6 Document Structure

It is possible to identify two aspects in this thesis project. First the tasks related to the algorithmic solutions and in the second place the effort related to the design and development of the framework. These two aspects are both separated into three phases: first analysis, then implementation and finally evaluation. This structure is reflected in the thesis and is visualized in Figure 1.2.

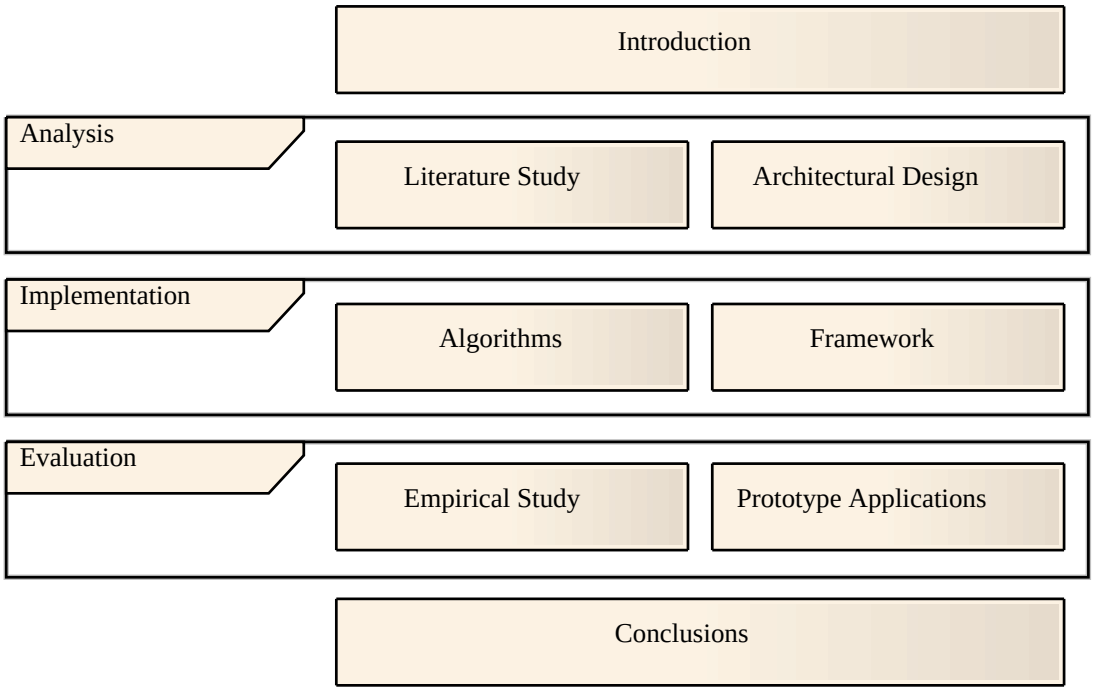

Figure 1.2: Document Structure

We see there are two common chapters: the introduction (Chapter 1) and conclusion (Chapter 8) of the work.

The thesis is separated into three parts: Analysis, Implementation and Evaluation. Each part consists of one chapter which discusses the algorithmic aspect and the second chapter discusses aspects related to the navigation framework itself. The algorithmic aspect of the project is thus discussed in:

- Chapter 2 : Literature Study. This chapter establishes graph theoretic notation and definitions. It gives a fairly broad, but brief, overview of algorithms that solve relevant graph

problems. Algorithms are selected to be implemented, based on the discussion in the chapter.

- Chapter 4 : Implementation of Algorithms. A detailed description of the algorithms that are implemented is given in this chapter and it reports relevant implementation details for them.
- Chapter 6 : Empirical Study. This chapter reports on the performance of the algorithms on generated problems with different properties. It compares several configurations of these algorithms with each other.

In the following chapters we report on aspects related to the navigation framework itself.

- Chapter 3 : Architectural Design. This chapter reports requirements for the framework and based on these requirements a design is constructed.
- Chapter 5 : Implementation of the Framework. We discuss several crucial implementation details of the framework in this chapter.
- Chapter 7 : Prototype Applications. On top of the framework several applications are developed during the project. This chapter gives a brief description of these applications.

These aspects are reasonably self contained. The reader with a particular interest in either of these can limit him/herself to the introduction, depending on interest the even or odd chapters and the conclusion.

Part II

Analysis

Chapter 2

Literature Study

In this chapter we discuss the current research in relevant graph algorithmic topics. The purpose is to establish common notation, definitions and assumptions used throughout the document. This chapter provides the reader with a brief introduction to various algorithms and conveys basic ideas behind them. It does not explain each algorithm in detail, but does provide references to more detailed sources of information. Furthermore, an answer to Research Question 1 is given.

> *What are the state-of-the-art algorithms and heuristics for the single-source shortest path, maximum flow and minimum-cost flow problems?*

As part of the answer to this research question we study the practical performance of these algorithms. Based on this study we answer Research Question 2:

> *Which of these algorithms are best suited to be implemented?*

This chapter is structured as follows: in Section 2.1 common definitions and notation are established. In Section 2.2 the shortest path problem and commonly used heuristics are treated. Of special interest are the maximum flow and minimum-cost flow problems. Algorithms for these problems are discussed in Section 2.3 and Section 2.4 respectively. Both these sections provide insight into the practical performance of algorithms concerned. In Section 2.5 we discuss several problems which include a time component. This section will answer Research Question 4:

> *How can we introduce the notion of time in these algorithms?*

2.1 Graph Algorithms

In this, and the following sections dealing with graph algorithms we base ourselves on books by Cormen *et al.*[14] and Ahuja *et al.*[3]. These works provide an excellent and in dept overview of the concerned topics.

2.1.1 Preliminaries and Definitions

A *graph* is defined as $G = (V, E)$ with finite vertex set V and finite edge set E, and each edge $e \in E$ is a pair (v, w) with $v, w \in V$. If the edges have a direction then G is called a *directed graph* or

digraph, otherwise the graph is an *undirected graph*. In the directed case we denote v the tail and w the head of the pair. We denote the number of vertices $|V|$ with n and the number edges $|E|$ by m. Unless otherwise stated we assume graphs considered contain no self loops or multiple arcs. For a vertex $v \in V$ the *degree* of v is given by $|(v, w) \in E|$. A *path* P is a sequence of vertices $(v_1, \ldots, v_k)$ such that $\forall i, 1 \leq i < k : (v_i, v_{i+1}) \in E$ and all $v_i \in P$ are distinct, i.e. there is no repetition of vertices on the path. The sequence of vertices is called a *walk* if there is repetition. If $v_1 = v_k$ the path is called a *cycle*, and a graph without a cycle is called *acyclic*. For graph without parallel edges or self loops n^2 is an upper bound on the number of edges, a graph is called *sparse* if it has $O(n)$ edges.

Throughout the document we will use the words 'vertex' and 'node' interchangeably to denote any $v \in V$. The word 'arc' always denotes a directed edge. Finally, the word 'network' is regarded as a synonym of graph. Next a list of assumptions used throughout the document.

Assumption 2.1.1. *The graph is a directed graph.*

In general it is possible to convert an undirected graph $G = (V, E)$ into a directed one $G' = (V, E')$ by replacing each edge $(v, w) \in E$ with two directed arcs (v, w) and (w, v) in E'. Both these two edges have the same weight as the original edge. However, if G contains an edge with negative arc length this will always create a negative cycle.

Assumption 2.1.2. *The graph does not contain parallel arcs or self-loops.*

This assumption is for notational convenience, if parallel arcs are allowed an edge is not uniquely identified by its head and tail (v, w).

Assumption 2.1.3. *All arc and node properties, such as weight, cost or capacity are integers.*

As we will see in the next chapters it is possible to associate various properties with edges or nodes. We assume these properties are always integer. In general this is required for algorithms that use scaling techniques to solve the problem. However, this assumption is not restricting in practice, because data is represented as a rational number in a computer. Such a number can be converted to an integer by multiplying it with a suitably large integer.

2.2 Shortest Path Problem

2.2.1 Definitions

Consider a directed graph $G = (V, E)$. We can associate a weight $w_{i,j}$ with each edge (i, j). The weight of a path $P = (v_1, \ldots, v_k)$, or cycle when $v_1 = v_k$, is the sum of the weight of its edges: $\sum_{(v_i, v_i+1) \in P} w_{i,i+1}$. If the weight of a cycle is negative we call this a *negative cycle*. The *single source shortest path problem* is defined as finding the paths with lowest weight between a given node s and all other nodes. A related problem is finding the shortest paths between all pairs of nodes called the *all-pairs shortest path problem*. When using the terms shortest path (SP) we always mean the single source shortest path problem. When considering the shortest path between two points we denote the source and destination vertices with s and t respectively.

A lot of graph algorithms make certain assumptions about the structure and properties of the graph they are applied to. In this paragraph we will list several assumptions which we consider to hold in general throughout the document unless otherwise stated.

Assumption 2.2.1. *The graph does not contain a negative cycle.*

Finding a shortest path in the presence of negative cycles is a substantial more difficult problem. In fact the problem is NP-complete[3].

2.2.2 Dijkstra's Shortest Path Algorithm

One of the most well known algorithms for solving the single source shortest path problem is Dijkstra's algorithm[17], which is used in practice to solve routing problems. In this paragraph we discuss Dijkstra's algorithm, which is given in Algorithm 1. Dijkstra's algorithm requires an additional assumption (2.2.2)

Assumption 2.2.2. *The graph does not contain any negative weight arcs. I.e.* $\forall (v, w) \in E : \ w(v, w) \geq 0$.

Within the domain of navigation this assumption is not restrictive. Alternatively, the Bellman-Ford [8, 39] algorithm is able to find a shortest path in the presence of negative arc lengths and detects negative cycles. However, its running time is inferior to Dijkstra's algorithm ($O(nm)$, Dijkstra using Fibonacci heaps $O(m + n \log n)$).

Dijkstra's algorithm associates two labels with every node. A current upper bound of the shortest path distance from the source to the concerned node (d_i). This value is initiated with ∞, except for the source node. Instead the source its label is set to 0. The second label is a pointer to the previous node on the current shortest path from the source to this node($prev$). The algorithm maintains two lists of vertices. First a list to which a final shortest path distance has been assigned (S) and a list of vertices which still have to be considered (Q). In each iteration of the algorithm it selects and removes the vertex (i) with the current minimum shortest path estimate from Q and adds it to S. Next it updates all neighbors connected to i, that is to say all vertices j for which arc $(i, j) \in E$. If the current label $d_j < d_i + w_{ij}$, then the label of d_j is set to $d_i + w_{ij}$ and the value $prev_j$ is set to i. When the algorithm terminates the shortest path from s to any node t can be found by following the $prev$ pointers starting with $prev_t$. The while loop is executed exactly n times, because every vertex is removed from S and added Q exactly once. The final asymptotic running time of the algorithm depends on the implementation used for list S. Specifically the operation to extract the vertex with the current minimum distance label. The best theoretic strongly polynomial worst-case running time is achieved by using Fibonacci heaps [23] and results in a running time of $O(m + n \log n)$. Faster weakly polynomial algorithms achieve an $O(m + n\sqrt{\log D})$ [2] and $O(m \log \log D)$ [38, 40] bounds. For a complete analysis and correctness proof we refer to [14].

2.2.3 Heuristics for Dijkstra

Wagner *et al.*[62] give an overview of speed-up techniques for the shortest path problem. These still guarantee correctness and do not change the worst case behavior of the algorithm. Typically, they do reduce the number of vertices considered, and thus the running time substantially. In this section we give a short overview of these heuristics. The interested reader is advised to consult [62] and references therein for a more detailed description of these heuristics.

Algorithm: Dijkstra

Input: Graph $G = (V, E)$
Input: Vertex s

$Q \leftarrow V$, $S \leftarrow \emptyset$;
foreach $i \in V$ **do** $d_i \leftarrow \infty$;
$d_s \leftarrow 0$, $\text{prev}_s \leftarrow 0$;

while $Q \neq \emptyset$ **do**
    Select $i \in Q$ for which $d_i = \min\{d_j : j \in Q\}$;
    $S \leftarrow S \cup \{i\}$;
    $Q \leftarrow Q - \{i\}$;
    foreach $(i, j) \in E$ **do**
        if $d_j > d_i + w_{ij}$ **then**
            $d_j \leftarrow d_i + w_{ij}$;
            $\text{prev}_j \leftarrow i$;

Algorithm 1: Dijkstra's Algorithm for the Shortest Path Problem

Bidirectional Search

Bidirectional search[46] performs a search, using Dijkstra's algorithm, in two directions at once. One emanating from the source and the other originating from the destination node. The search starting from the destination is applied to a reverse graph. I.e. consider graph $G = (V, E)$ and *reverse graph* $G^r = (V, E^r)$, with $(w, v) \in E^r \ \forall (v, w) \in E$. The algorithm can terminate once their search fronts meet; i.e. when there is a node, that is assigned a permanent label by both the forward and reverse search waves. Let d_{if} denote the distance label assigned to vertex i by the forward search and d_{ir} the label assigned by the reverse search. The shortest path is defined as $\min_{i \in V}\{d_{if} + d_{ir}\}$.

Goal-Directed search or A*

Dijkstra's algorithm always considers nodes in FIFO order. A* on the other hand attempts to guide the search in the direction of the destination by changing the order of nodes considered. During the search A* assigns a lower bound estimate of the distance h_i from node i to the destination to each node i. This in addition to the assigned distance label (d_i). This lower bound estimate is defined as a function $h : V \rightarrow \mathbb{R}$. Let $h^*(i)$ denote the actual minimum distance from node i to the destination, a heuristic function $h(i)$ is called feasible if $\forall v \in V : 0 \leq h(v) \leq h^*(v)$. Now the ordering of nodes in the priority queue is determined by $d_i + h_i$. A feasible heuristic function can be obtained by using the Euclidean distance between a node and the destination. Hart *et al.*[35] give a proof of termination and optimality of A*.

Hierarchical Methods

Heuristics in this category work by adding additional short-cut edges to the original graph during a preprocessing step. Two main approaches for constructing such a hierarchical graph are the multi-level approach and highway hierarchies. The multi-level approach is discussed next.

It works by identifying a hierarchy of subsets of V such that $S_n \subset \ldots \subset (S_0 = V)$. Call a vertex $i \in S_j$ a selected vertex at level j. The graph $G = (V, E)$ is enriched by additional edges: upward edges pointing from a non-selected vertex to a selected vertex on a higher level, downward edge going from a selected edge at some level to a non-selected vertex on a lower level and level edges traversing from one selected vertex to another selected vertex on the same level. Vertices could be selected based on the degree of a node or one could exploit domain specific knowledge.

Edge Labels

With this approach a label is associated with each edge indicating all nodes to which the shortest path starts with the concerned edge. More precisely: $\forall(v, w) \in E$ find set $S(v, w)$ such that $\forall t \in V$ if the shortest path v, t starts with (v, w) then $t \in S(v, w)$. Its trivial to see that if (u, v) is part of the shortest s, t path then its sub path u, t is also a shortest path. Dijkstra's algorithm can be modified such that only edges containing the target node in their label are considered at each step. Unfortunately this approach requires $O(n^2)$ space.

2.3 Maximum Flow Problem

2.3.1 Definition

A *flow problem* consists of a directed graph and two distinguished vertices $s, t \in E$. These vertices are called the *source* and *sink* respectively. Furthermore a non negative capacity $c(v, w)$ is associated with each arc $(v, w) \in E$. Let C denote the largest arc capacity. A *flow* is defined as a function $f : E \to \mathbb{R}$ such that the capacity (2.1), the antisymmetry (2.2) and the flow conservation (2.3) constraints hold.

$$\forall(v, w) \in E : \; f(v, w) \leq c(v, w) \tag{2.1}$$

$$\forall(v, w) \in E : \; f(v, w) = -f(w, v) \tag{2.2}$$

$$\forall v \in V - \{s, t\} : \sum_{(v,w)\in E} f(v, w) - \sum_{(w,v)\in E} f(w, v) = 0 \tag{2.3}$$

To solve the *maximum flow problem* on a graph G we wish to find a flow f with maximum value. The value of a flow $|f|$ is the net flow into the sink and is defined by $|f| = \sum_{v\in E} (v, t)$. Another important concept is that of residual capacity, which we define next. The *residual capacity* $r_f(v, w)$ of an edge $(v, w) \in E$ is defined as $r_f(v, w) = c(v, w) - f(v, w)$ and if $r_f(v, w) > 0$ we call (v, w) a *residual edge*. The *residual graph* G_f consists the vertex set V and the set of all residual edges E_f ($G_f = (V, E_f)$).

We can partition V into two subsets P and Q with $Q = V - P$ and $s \in P$ and $t \in Q$. This is called a *cut* and we denote such a cut with $[P, Q]$. The capacity of a cut is defined as $c(P, Q)$ and the cut with the minimum capacity of all cuts in the graph is referred to as *minimum-cut* ($[P, Q]^*$). Let f be a flow with maximum value in G, then $\exists[P, Q]$ such that $|f| = c(P, Q)$ and we have $[P, Q] = [P, Q]^*$. This is the maximum flow minimum-cut theorem. A proof is given in [14]

Note that we can support multiple sources $S = (s_1, \ldots, s_k)$ and sinks $T = (t_1, \ldots, t_l)$ by converting our original graph $G = (V, E)$ to $G' = (V', E')$. To obtain G' two vertices $s*$ and $t*$ are added to original V. These are denoted *supersource* and *supersink* receptively, to the V. Furthermore:

$\forall s_i \in S : (s*, s_i)$ is added to V' and $\forall t_i \in T : (t_i, t*)$ with infinite capacity is added to E, now called E'. The solutions for the maximum flow problem on G and G' will be equivalent.

In addition the the assumptions made in the previous sections we make the following assumption for the maximum-flow problem.

Assumption 2.3.1. *If arc* $(v, w) \in E$ *then* $(w, v) \in E$.

If $(w, v) \notin E$, we add (w, v) with $c(w, v) = 0$.

2.3.2 Overview

Specialized algorithms for the maximum flow problem are based on four different methods. Fulkerson and Dantzig [24] applied the Simplex method to the problem. Ford and Fulkerson [39] gave a method based on augmenting paths. Edmonds and Karp [19] showed the algorithm had a polynomial time bound when augmenting along shortest paths, which are found using Breadth First Search (BFS). Dinitz [18] constructed a method based on calculating blocking flows. Finally, Goldberg and Tarjan [29] published the push-relabel method. The best theoretical bound is achieved by Goldberg and Rao [28] based on Dinitz' blocking flow method. They achieve a $O(\min\{m^{3/2}, n^{2/3}m\} \log(n^2/m) \log C)$ (with C the largest capacity) worst case bound.

2.3.3 Algorithms

Ford-Fulkerson

The algorithm of Ford and Fulkerson [39] is based on finding paths from the source to the sink in the residual network G_f, called *augmenting paths* $P_f = \{v_s, \ldots, v_t\}$ with $\forall v_i \in P_f - \{v_t\} : (v_i, v_{i+1}) \in E_f$. Such an augmenting path can be found with, for example, BFS. The flow along such a path can be augmented with the minimum capacity of an arc on this path. A drawback of this method is that each augmentation might only carry a small amount of flow.

Capacity Scaling

The capacity scaling algorithm by Edmonds and Karp [19] solves the maximum flow problem by ensuring that in each step a sufficiently large value is augmented. They define a Δ-residual Graph ($G_f(\Delta)$). Such a graph only contains those residual edges that have a capacity larger then Δ. In each iteration the algorithm looks for augmenting paths in $G_f(\Delta)$ and flow is augmented along this path. Once no more augmenting paths exist the algorithms reduces Δ with $\frac{\Delta}{2}$ and the next iteration commences. The algorithm terminates when no more paths exist and $\Delta = 1$. Because $G_f(1) = G_f$ the algorithm terminates with a maximum flow. The algorithm has a worst case complexity of $O(m^2 \log C)$.

Shortest Augmenting Path

As it's name suggests the Shortest Augmenting Path algorithm only augments along the current shortest path from source to sink in the residual graph. It spends an average time of $O(n)$ per augmentation by determining this shortest path in a smart way. It has a time complexity of $O(n^2 m)$.

Push-Relabel

The push-relabel algorithm takes a different approach. Instead of augmenting flow along a path from source to sink it makes local adjustments by relaxing the flow conservation constraint (2.3) and using 2.4 instead. It thus allows vertices to have excess flow. Such a flow, which allows vertices to have excess flow, is called a *pseudo-flow*. During execution a labeling of the vertices is maintained. The algorithm then relies a push and a relabel operation to make local modification in order to transform the pseudo-flow into a flow. The push operation causes flow to be pushed to a vertex with a lower label while the relabel operation increases the label of a vertex if there is no neighbor with a lower label. Conceptually this labeling first causes flow to flow towards the sink. In later stages of the algorithm excess flow will be pushed back towards the source. The algorithm terminates when no vertex has excess and the pseudo-flow is in fact a maximum flow.

$$\forall v \in V - \{s,t\} : \sum_{(v,w)\in E} f(v,w) - \sum_{(w,v)\in E} f(w,v) \geq 0 \tag{2.4}$$

2.3.4 Empirical Results

In [6] Anderson and Setubal compare an implementation of the push-relabel algorithm with implementations of Ford-Fulkerson's, Edmonds-Karp's and Dinitz' algorithms. In addition they evaluate different strategies and heuristics for the push-relabel algorithm. The performance of the push-relabel algorithm is clearly superior to the other implementations except for a class of acyclic dense graphs where Dinitz' algorithm performs better. However, the use of the global relabeling heuristic is essential. The performance of their implementation based on a queue is shown to be most robust, and it never loses with a wide margin. The highest label strategy often performs best, but performs far worse on some problem classes. Nguyen and Venkateswaran [49] confirm the conclusions made in [6]. Their tests show the importance of the global relabeling heuristic and conclude the implementation based on a queue performs best in general. A global relabeling frequency of once every n relabellings works well in practice.

2.4 Minimum-Cost Flow Problem

In this section the minimum-cost flow problem is discussed. We first shortly describe a number of classical approaches to solve the problem and end our discussion with a more elaborate review of the Successive Approximation by Cost Scaling algorithm. This algorithm is closely related to the push-relabel method treated in the previous section. Empirical results indicate implementations of this algorithm achieve good practical performance.

2.4.1 Definition

We obtain a *minimum-cost flow problem* by extending the definition of the previously given Maximum flow problem. An additional cost $u(v,w)$ is associated with each arc $(v,w) \in E$, and a demand $d(v)$ is specified such that equation 2.5 holds. Here our objective is to minimize the cost: $\sum_{(v,w)\in E} f(v,w) \times u(v,w)$. We denote the largest value of $u(v,w)$ with U.

We extend the definition for the residual network to apply to the minimum-cost flow problem. The residual capacity $r_f(v,w)$ of an edge $(v,w) \in E$ is defined as $r_f(v,w) = c(v,w) - f(v,w)$ and if

$r_f(v,w) > 0$ we call (v,w) a residual edge. The cost $u(v,w)$ is associated with the residual edge. In addition we add arc (w,v) with $r_f(w,v) = f(v,w)$ and cost $-u(v,w)$ if $r_f(w,v) > 0$. The residual graph G_f consists the vertex set V and the set of all residual edges, E_f, with positive residual capacity ($G_f = (V, E_f)$).

$$\sum_{(s,v)\in E} f(s,v) = d \tag{2.5}$$

Assumption 2.4.1. *If arc $(v,w) \in E$ then $(w,v) \in E$.*

If $(w,v) \notin E$, we add (w,v) with $c(w,v) = 0$.

Assumption 2.4.2. *No arc costs are negative.*

If there is such an arc (v,w) we can reverse its direction and add (w,v) with cost $c(w,v) = -c(v,w)$.

2.4.2 Overview

Already as early as 1951 Dantzig [15] applied the simplex method to the minimum-cost flow problem. Edmonds and Karp [19] published the successive shortest path algorithm based on reduced costs. The original cycle canceling algorithm is due to Klein [42] and was improved upon by Goldberg and Tarjan [31] and Barahona and Tardos [7]. The relaxation method (not discussed here) is due to Bertekas and Tseng [9]. Currently the best bounds are achieved by using scaling approaches. The first to apply scaling approaches to the problem were Edmonds and Karp [19]. In 1987 a method using cost scaling was published by Goldberg en Tarjan [26]. Based on this method they established a $O(n^3 \log(nU))$ worse case bound. Ahuja *et al.*[1] applied scaling to both cost and capacity. Their best algorithm runs in $O(nm \log\log(C) \log(nU))$. The best strongly polynomial algorithm is due to Orlin [52, 53] with an $O((m \log n)(m + n \log n))$ worse case behavior.

2.4.3 Algorithms

Successive Shortest Path

The successive shortest path algorithm finds a shortest path based on reduced costs in the residual network. In each iteration it starts at a node with excess flow (v) and finds a shortest path to a node with deficit flow (w). The reduced cost of an arc $u_p(v,w)$ is defined as $u_p(v,w) = u(v,w) + p(v) - p(w)$, for a given price function, $p : V \to \mathbb{R}$. Let P denote such a shortest path, with (i,j) an arc on this path and r_{ij} the residual capacity of such an arc. Furthermore, let be $e(v)$ the excess of start node v and $e(w)$ the demand (negative excess) of the destination node. Once such a path is fond we augment δ units of flow along this path. Where δ is defined as: $\delta = \min\{e(v), -e(w), \min_{(i,j)\in P}\{r_{ij}\}\}$.

An important drawback of the algorithm is that there is no guarantee how much flow is augmented in each step, in fact each step might carry only a small amount of flow. The capacity scaling approach improves this method by ensuring each augmentation augments at least a minimum value of flow.

Cycle Canceling

In contrast with the previous successive shortest path algorithm the cycle canceling algorithm[42] starts with a feasible flow, which is obtained by a single execution of any maximum flow algorithm. The algorithm then continues to update this feasible flow by repeatedly finding negative cycles in the residual network and augmenting flow along these cycles. This approach is based on the observation that, if there is a negative cycle in the residual network, then it is possible to send flow along this cycle. A new feasible solution with a reduced cost is obtained in this way. Alternative approaches based on this method augment flow along cycles which result in maximum improvement[31], and [7] augments along the minimum mean negative cost cycle.

Capacity Scaling

This approach improves the successive shortest path algorithm by making sure that in each iteration the algorithm augments a sufficiently large amount of flow. In each iteration a shortest paths, with a minimum capacity of at least δ, are found, between nodes with supply and demand. Flow is sent along such a path. Once there are no more paths with a capacity of at least δ, the value of δ is halved and the next iteration commences.

Successive Approximation by Cost Scaling

It is possible to obtain an efficient algorithm by scaling on cost, instead of capacity. Goldberg and Tarjan [26] described such an algorithm. The *reduced cost* of an arc $u_p(v, w)$ is defined as $u_p(v, w) = u(v, w) + p(v) - p(w)$. For a given constant $\epsilon > 0$ and a price function, $p : V \to \mathbb{R}$, a flow is ϵ-*optimal* with respect to p if $\forall (v, w) \in E_f \; u_p(v, w) \geq -\epsilon$. The algorithm iteratively transforms an ϵ-optimal flow into an $\frac{\epsilon}{\alpha}$ (with a constant $\alpha > 1$) pseudo flow, which is then updated to become an $\frac{\epsilon}{\alpha}$-optimal flow. In this way the algorithms successively obtains a better approximate of the minimum cost flow until its approximate is in fact a minimum cost flow.

2.4.4 Empirical Results

Empirical results for the minimum-cost flow problem are less decisive as those reported for the maximum flow problem. Goldberg and Kharitanov [30] published a study of their implementation of the Successive Approximation algorithm, and comparing with an implementation of the Relaxation Algorithm (RELAXT)[9]. They conclude that in general the Successive Approximation performs well, usually better then RELAXT. However, on some problem instances it performs worse. Goldberg later published a paper [27], that extends those previous results. He compares a new improved implementation (CS2[1]) of the successive approximation algorithm with RELAXT, and two implementations of the simplex method (NETFLO [41] and RNET [33]) applied to the Minimum Cost flow problem. The conclusion in this work supports the earlier results by Goldberg and Kharitanov. The CS2 implementation usually outperforms the other algorithms, and never loses with a large difference. Earlier empirical studies showed the network simplex and relaxation methods are superior to other non-scaling methods, such as successive shortest path and cycle canceling methods [3].

[1] Source available at `http://avglab.com/andrew/soft.html`.

2.5 Dynamic Flows

Until now all problems discussed were static flows and the problems did not incorporate a time component. However, in many problems a time component is essential. In this section we give the definition of a dynamic graph and specify three problems that contain a time element. Specifically, these three problems are the maximum dynamic flow, earliest arrival and quickest flow problem. Finally, we give a method to solve the quickest flow problem by converting a dynamic graph into a static minimum cost flow graph, which is called a time expanded graph. The solution to the quickest flow problem can be obtained by finding a minimum cost flow in the time expanded graph. An evacuation scenario can be modelled as a quickest flow problem.

2.5.1 Definition

The definition of a *dynamic graph* ($G_\tau = (V, E_\tau)$ is similar to that of a minimum cost graph. Here instead of costs we associate a transit time $\tau(v, w)$ with $\tau(v, w) \geq 0$ with each $(v, w) \in E$. Let Θ denote the finite ordered set of time steps. A *dynamic flow* is defined as function $f : E, \Theta \to \mathbb{R}$. We use the notation that $f(v, w, \theta + \tau(v, w)) = -f(w, v, \theta)$.

We modify the various constraints to apply to dynamic flows.

$$\forall (v, w) \in E, \forall \theta \in \Theta : \ f(v, w, \theta) \leq c(v, w) \tag{2.6}$$

$$\forall (v, w) \in E, \forall \theta \in \Theta : \ f(v, w, \theta) = -f(w, v, \theta) \tag{2.7}$$

$$\forall v \in V - \{s, t\}, \forall \theta \in \Theta : \sum_{(v,w)\in E} f(v, w, \theta) + \sum_{(w,v)\in E} -f(w, v, \theta) = 0 \tag{2.8}$$

The flow value of a dynamic graph is defined as $|f| = \sum_{v \in V} \sum_{\theta \in \Theta} f(v, t, \theta)$. Here t denotes the sink. Several problems are formulated based on dynamic graphs. The *maximum dynamic flow problem* aims to find a flow such that $|f|$ is maximized within a given time bound. The solution to the *earliest arrival problem* maximizes the amount of flow that reaches the sink in each time step. The *quickest flow problem* attempts to minimize $\theta \in \Theta$ such that all demand has reached the sink.

2.5.2 Time Expanded Graphs

We can obtain a *time expanded graph* $G_e = (V_e, E_e)$ from a dynamic graph $G_\tau = (V, E_\tau)$ and a time horizon Θ as follows. Graph G_e is a static graph and each edge $(v, w) \in E_e$ has capacity $c(v, w)$ and cost $u(v, w)$ properties according to the definition of the minimum cost flow problem.

$\forall \theta \in \Theta, \forall v \in V$ we create a copy v_e. Here v_i represents vertex v at time i. We denote the set of these time copies with V_e. $\forall (v, w) \in E_\tau$ we add edge (v_i, w_j) to E_e if $j - i = \tau(v, w)$ and assign it a capacity $c(v_i, w_j) = c(v, w)$ and cost $u(v_i, w_j) = i + \tau(v, w)$. Thus an edge is assigned a weight equal to the timestep at which flow leaves the concerned edge. In this way the edge weights increase over time. This approach is called the *turnstile costing approach* by Chalmet *et al.*[12]. Jarvis and Ratliff [37] proof that it is possible to solve both the quickest flow problem and the earliest arrival problem at the same time, when using this approach. To deal with multiple sources and sinks we can add a supersource and supersink like we did with static flows.

We can now find a maximum dynamic flow in G_τ by solving the maximum flow problem in G_e and find a solution to the quickest flow and earliest arrival problems by solving the minimum cost flow problem in G_e [3].

2.6 Conclusion

In this chapter we provided a brief description of the main ideas behind algorithms relevant for our work. Section 2.2 defines the shortest path problem and gives Dijkstra's algorithm for solving it. Within the navigational domain this algorithm is commonly used to solve the problem. In order to gain a substantial speed-up in practical performance a number of heuristics can be used. In Section 2.3.4 and 2.4.4 we reported on empirical results comparing a wide range of algorithms for the maximum flow and minimum-cost flow problems respectively. Based on the reviewed reports we can conclude that the push-relabel and successive approximation by cost scaling algorithm constitute to the state-of-the-art algorithms for these two problems. The push-relabel often outperforms competitive algorithms by a significant margin [27], provided that the global relabeling heuristic is used. For the minimum-cost problem the reported results are less decisive, however in general the successive approximation algorithm outperforms competing algorithms and never loses by a wide margin. As an additional advantage these algorithms allow intuitive parallelization. This answers Research Questions 1 and 2. Respectively

> *What are the state-of-the-art algorithms and heuristics for the single-source shortest path, maximum flow and minimum-cost flow problems?*

and

> *Which of these algorithms are best suited to be implemented?*

Finally, we defined a dynamic graph and discussed a method to obtain solutions for the maximum dynamic, earliest arrival and quickest flow problems. This answers Research Question 4:

> *How can we introduce the notion of time in these algorithms?*

Chapter 3

Architectural Design

In this chapter the architectural design of the framework is described. This description is structured according to the Institute of Electrical and Electronics Engineers (IEEE) 1471[50] recommended practice. This document defines common concepts, terminology and a methodology for creating an architectural description for software systems. An architectural document has to identify stakeholders interested in the system and the concerns they have regarding the system. Then requirements applicable to the system must be elicitated and documented. Based on the concerns of the stakeholders and these requirements an architectural design is constructed. Such a design consists of a collection views. Each conforms to a specific viewpoint and highlights certain aspects of the system. A viewpoint specifies how a view is constructed, it provides a rationale for the view, and it lists the diagrams or models that are used to illustrate the view. Furthermore it states which stakeholders and concerns are addressed by the view.

Viewpoints are selected based on Kruchten's 4+1 Model of Software Architecture [43]. This model defines five viewpoints[1] and specifies how views conforming to these viewpoints can be constructed. The five defined viewpoints are the logical, process, development, physical and scenario viewpoint. This last scenario viewpoint is used to relate the four previous ones with one another. For this reason the scenario view represents the '1' in '4+1'. All viewpoints are further described in Section 3.3. In the discussion of the physical viewpoint we argue that the physical view is not relevant for a framework. We omit the physical view, because of the arguments that are given in this discussion.

The next section identifies various stakeholders and lists concerns for each of them. Section 3.2 reports on the requirements for the framework. It aims to answer Research Question 7:

> *What are the requirements for such a framework?*

The section following the requirements specifies the selected viewpoints. The consecutive sections give the different views corresponding to the specified viewpoints. In order: the logical view in Section 3.4, the process view in Section 3.5, the development view in Section 3.6 and finally the scenario view which connects the three previous views. This collection of views gives an answer to Research Question 8:

> *What design satisfies the requirements?*

[1] Kruchten actually uses the term view, however this term maps to viewpoint in the IEEE 1471 specification.

Finally, it can observed from the requirements and design how the framework addresses Research Question 5:

In what way can we incorporate dynamic changes of structure and properties?

3.1 Stakeholders and their Concerns

In this section we identify all stakeholders that are concerned with the framework at some point during its life cycle. Specific concerns are elicitated for each of these stakeholders. Due to the open nature of the project, certain roles have no concrete representation. However, their concerns are nevertheless important, therefore it is decided to still list those. To validate the completeness of the listed stakeholders and concerns a software engineer at Logica was interviewed.

To indicate a priority among concerns a '+' is used to specify concerns of greater importance. We distinguish between two groups of stakeholders, those involved with developing the framework and those who instantiate the framework to build a specific application. These two parties could be part of different organizations or there might be some overlap between these functions.

- Developer: management, designers, developers and maintainers.
- Acquirer: management and application developers (the users of the framework).

3.1.1 Developer

With the developer we mean the organization responsible for initiating the development of the framework based on this architecture document. Once the initial development has been completed they will be responsible for maintaining it, which might involve additional development cycles to provide new functionality.

Management

+ Feasibility. It should be feasible to develop the framework within the given constraints, including skill of developers, time allocated for the project.
+ Applicability. One of the aims of a framework is to enable code reuse. To that end it should be applicable within its intended domain.
- Documentation. A framework's utility depends for a large part on the ease of use for its users. Properly documenting a software artifact, however, is an extensive effort. This should be taken into account by the management.
- Interoperability. by definition a framework is not a complete application but will have to work together with other pieces of software. The integration with other software will be easier if the framework is designed with interoperability in mind. The use of standards supports this concern.
- Portability will allow the framework to be deployed on different platforms potentially increasing its applicability.

Designers

The designers are responsible for producing a clear design based on the requirements and concerns of stakeholders.

- Clear and unambiguous requirements.
- Complete and consistent requirements. It must be possible to produce a feasible design from the requirements.

Developers

These are the actual developers of the framework. They are responsible for implementing the framework such that resulting artifact meets the requirements.

+ Clear and understandable design. The developers have the task of creating the framework based on the knowledge contained in the architecture and technical designs. In order to fulfill this task efficiently, it is important that the required knowledge is readily and easily available.
- Documentation. Writing documentation is often regarded as a time consuming and tedious job. Enough time should be allocated to properly document the framework.
- A modular structure. This allows effective work division among team members.
- Technologies used for implementing the framework. Preference is given to mature, well supported technologies for which expertize is available within the team.

Maintainer

The maintainer is responsible for maintaining the framework, i.e. fixing bugs and adapting the framework to newly discovered requirements during the lifetime of the framework. The original developer team might be assigned to this task, but it is also possible that another team assumes this role. If a new team takes on this task, then it is important they can quickly familiarize themselves with the overall architecture and structure of the framework.

+ Documentation: proper and clear documentation allows them to understand the structure of the framework and as a result it both increases their work efficiency and quality. They will also be responsible for keeping the documentation up-to-date.
+ Modularity: A modular structure will allow them to, in general, make quick fixes without affecting the framework as a whole.

3.1.2 Acquirer

The Acquirer denotes the party which uses the framework to implement a specific application. Their management needs to be able to assess the suitability of the framework for their application [22]. Once it is decided, that the framework will be applied in development of the application, the developers within this party will use the framework to develop the specific application at hand. In order to fulfill their task, they largely depend on the documentation of the framework. A number of concerns follow directly from ease-of-use. In this section we identify concerns which hold in general for any acquiring party, this in contrast with application specific requirements. Of course, the weight of certain concerns will depend on specific application details.

Management

With this category we mean the group of people responsible for evaluating the framework and deciding on whether it is suitable for the problem at hand. A number of concerns are specific to the intended use of the framework. However, we can identify some general concerns. These are listed next.

+ Ease of use for developer team.

- Cost. The cost involved with using the framework as opposed to just developing this specific application. This depends on the fit between the framework and the intended application domain, the effort required to learn to work with the framework and whether it is expected more similar applications will be build. If more similar applications are expected to be build, then the additional cost of learning to use or developing a framework is justified by the expected gain in application development productivity.

- Extensibility. The effort required to tailor the framework to the specific needs of the application, ideally the frameworks fits the application domain perfectly and hardly any effort is required to instantiate it.

- Maturity and stability. The external interfaces of a framework must remain stable. This will ease migration to a new version of the framework, and, if during the evolution of the framework its exported interfaces change, then this might lead to a cascade of changes in the application code. A mature framework, and in general thoroughly used code, is expected to have fewer bugs and to be more reliable and robust [22]. These properties can be acquired by sound design and the investment of effort over time.

Users of the framework

The users of a framework are actually the developers building a specific application using the framework. As such they are most concerned with the ease-of-use of the framework, which, among other things, depends on clarity of the documentation.

+ Ease of use: it should be relatively easy to build a new application with the framework. This concern depends on several things, for example clear documentation and stable interfaces.

+ Documentation: in order to instantiate the framework and construct application with it the developers need to understand how the framework is designed and how they can easily integrate the framework into their application. Good documentation, explaining the design of the framework and how to use it, facilitates this.

- Stability. The interfaces exported by the framework should remain stable. It becomes difficult to migrate to a new version of the framework, if the interfaces it exports change during the evolution of the framework. Such a change in external interfaces might cause a cascade of changes in the application code.

- Interoperability. Frameworks have to work together with other components. The use of standards supports this and makes it easier to connect frameworks with other artifacts. This indirectly supports the ease of use of the framework.

- Portability. If a framework does not support a target platform initially, it may very well be nearly impossible to port it to that particular platform [60].
- Scalability. The performance of an application based on the framework will depend on the implementation of the framework. To avoid the situation, in which the framework becomes the bottleneck of the system, it should be designed with scalability in mind.

3.2 Requirements

Requirements elicitation and specification are an essential part of a software project, because all subsequent phases of design and development depend on it. Incomplete or flawed requirements are very costly to correct in later stages of the project and are a major risk to project success[20]. In this section we list functional and quality requirements, followed by some constraints imposed on the solution. Functional requirements specify the concrete behavior of the system. In contrast quality requirements describe often subjective general properties, such as maintainability, of the system. Sometimes the term non-functional requirements is used instead of quality requirements. This section thus provides an answer to research question 7:

What are the requirements for such a framework?

The requirements listed here are derived from the project description given in Chapter 1, the theoretical background discussed in Chapter 2, the authors own ideas and a discussion of these ideas with a software engineer at Logica. Based on these requirements a design for the framework will be constructed, which is given in subsequent sections.

3.2.1 Functional Requirements

Functional requirements define the behavior of the system. They are listed in Figure 3.1, by category. We identify five categories, with 'operations' being a subcategory of 'algorithms':

- General: This category consists of requirements applicable to the system as a whole and regarding the exposure of functionality to external applications.
- Graph management: Contains requirements imposing constraints on the Graph data structure used in the system and those related to the administrations of Graphs.
- Algorithm management: Contains requirements regarding the management and execution of algorithms.
- Entity management: Contains requirements regarding the position of entities and updates of these positions.
- Algorithms: This category and the Operations subcategory list all requirements with respect to the actual algorithms and operations which should be implemented.

The next sections provide further clarification.

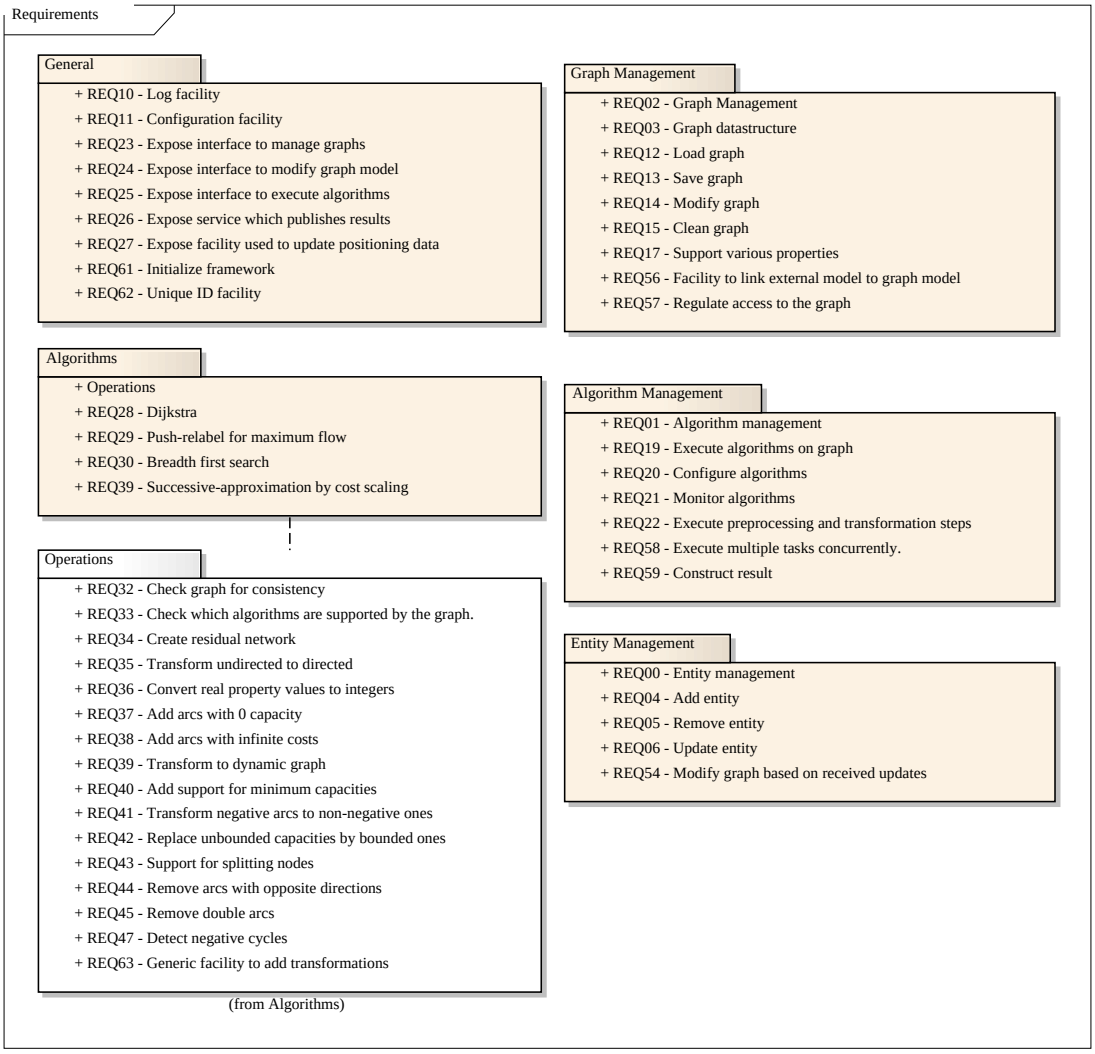

Figure 3.1: Functional Requirements

General

This category contains requirements relating to functionality to manage and configure the framework as a whole. Requirements specifying how the framework interacts with external components are listed in this category as well. Often, requirements in this group are related to specific requirements in one of other categories. For example, the requirement for a 'Unique Id facility' (REQ62), stems from the requirement in the Graph management category specifying it should be possible to relate the external environment to the Graph data structure (REQ18).

Graph Management

Requirements in this group are related to the construction and management of the graph data structure. An internal Graph data structure has to be defined and functionality to load and create a Graph from a source must be provided. As noted previously there is a wide range of sources are conceivable and, therefore, it should be easy to add support for new sources. It is only required to offer facilities which allow addition of these modules with minimal effort. It is *not* required to provide an implementation. Apart from the functionality to load a graph, it has to be possible to save the internal graph model, but it is not required to support translation between various data formats. The Graph data structure should be flexible in order to support various properties required by different algorithms. Furthermore, there should be some facility which allows an application to relate the elements of the Graph data structure to the external environment.

Algorithm Management

Here we deal with requirements that relate to the management and execution of algorithms. Requirements concerning the actual implementation of algorithms and support transformations on graphs are mentioned in the next section. Most requirements in this group are self explanatory; with the ability to configure algorithms we mean support for setting various parameters and selecting specific heuristics.

Algorithms

In this sections we describe requirements concerning the concrete algorithms that will be implemented. Often graph algorithms require certain preprocessing steps to be executed before the actual algorithm is run. Additionally, some algorithms allow for the implementation of heuristics which can sometimes drastically improve performance. The algorithms and heuristics mentioned here are discussed in Chapter 2.

Algorithms:

- Breadth first search. A basic search algorithm which is used as preprocessing step or as subroutine within an algorithm.
- Dijkstra. In Chapter 2 a number of heuristics are discussed. These heuristics are considered as 'nice-to-have' features, but not essential to the initial version of the framework.
- Push-Relabel for the maximum flow problem, based on a FIFO-queue implementation and the global relabeling heuristic.

- Successive-approximation by Cost Scaling for the minimum-cost flow problem, which is related to the previously mentioned Push-Relabel .

Operations and Transformations:

The different graph algorithms make different assumptions about the structure and properties associated with the graph. This category lists a number of requirements dealing with transformations of the graph and ensuring that it is and remains consistent. Most operations are derived from the various assumptions algorithms make about the structure of the input Graph. (Consult Chapter 2 for an explanation). Most important is a facility which allows easy addition (REQ63) and execution of new graph operations. A few operations are strong requirements, namely; REQ32, REQ33, REQ34, REQ37. The other graph operations can be implemented 'on-demand'.

Entity Management

Requirements in this group are related to entity management. The framework keeps track of the position information for groups of entities, in order to support novel functionality like anticipated traffic congestion. When the position of an entity is updated by an external application the Graph has to be updated. To avoid continuous changes to the graph structure, modifications can be applied in batches.

3.2.2 Quality Requirements

Sommervile [59] advises to specify quality requirements in terms of objective and measurable values. However, he also mentions that in practice the effort and cost required to objectively measure these requirements may not be justified. Nevertheless, a description of quality requirements is valuable even when not objectively measured. If for each requirement a priority is indicated, then developers can use these as a goal or guideline when developing the system.

Zeist *et al.*[61] give the Quint2[2] framework for specifying software quality. This framework is an extension of the International Organization for Standardization (ISO) 9126[36] model for software quality. It is shown graphically in Figure 3.2. This model defines a number of software quality properties and several sub characteristics for each property. For a characteristic a number of concrete indicators are given, which can be used to quantify 'quality'. For example it defines a property 'reliability' with sub characteristics maturity, fault tolerance, recoverability, availability and degradability. For availability it gives indicators such as 'availability ratio' and it specifies a protocol, that indicates how such an indicator can be measured. The added benefit over the ISO standard is that it gives such a protocol for all indicators. However, some characteristics are difficult to measure and as a result the validity of these specifications varies. In fact, the authors themselves indicate a validity rating for each of the protocols.

From this collection of characteristics we selected a small subset, which we think are key to the framework. We intend to use these requirements as guideline for design and development of the framework and will not make objective measurements. The definitions are adapted from [61].

- Changeability. The effort required to modify or add new functionality to the framework. By nature a framework has to be modified and extended in order to instantiate it to form a complete application. Changeability is therefore a characteristic of prime importance for the

[2] `http://www.serc.nl/quint-book/`

success of a framework. A structure with highly decoupled modules facilitates changeability. Such a structure enables replacement of a certain module with a different implementation while limiting the impact to other modules.

- Reliability and fault tolerance. The ability of the framework to remain functional, even if exceptions or errors occur. The framework will support the execution of various algorithms and operations. Therefore it is important that the framework is able to continue operating normally even after a fault occurs during the execution of an algorithm. Moreover a badly implemented algorithm may not terminate and the framework must be able to monitor execution and abort if necessary.
- Understandability and clarity. The effort that is required by users of the framework to recognize functionality of the framework and to understand the logical concepts behind it. To use the framework it must be clear to developers what functionality is offered by the framework and how this functionality can be used and extended. This requirement can be met by clear coding standards and proper documentation.

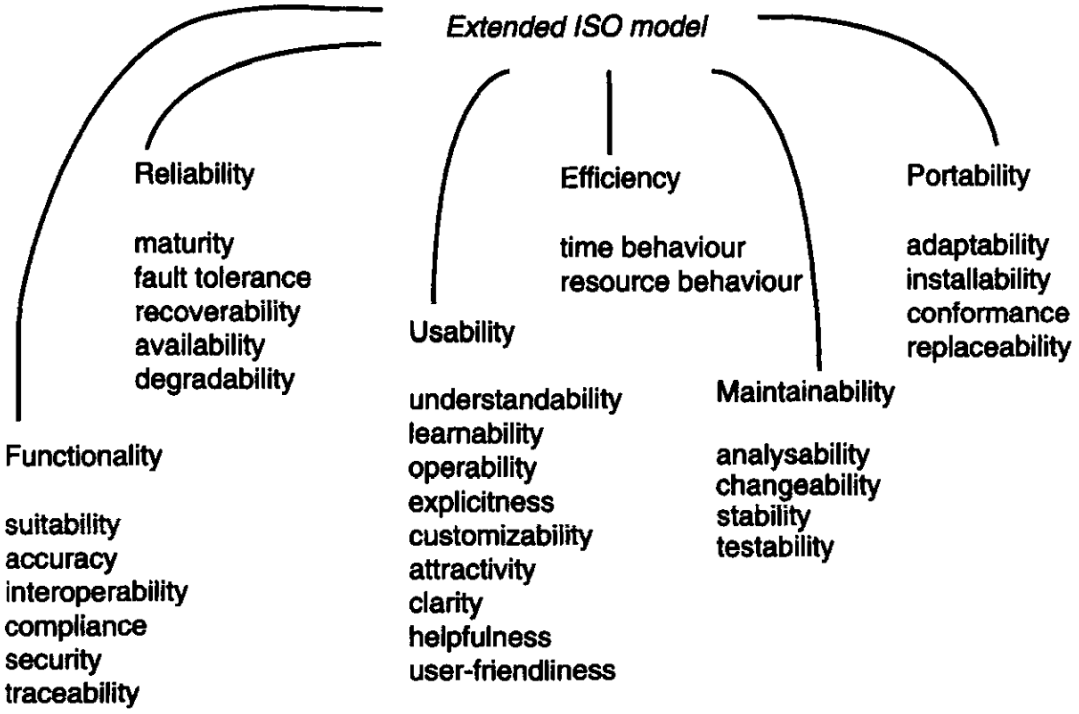

Figure 3.2: Quint2 model for software quality.

3.2.3 Constraints

In this sections we list and describe a number of constraints which limit the solutions space.

- Time allocated to the project. The deadline for completion of the encapsulating project is set to October 2008. The project consists of the creation of additional artifacts apart from the implementation of the framework, hence not all available time can be assigned to the inception of the framework.
- Language used to implement the framework: Java. This constraint stems from the expertise available within the development team. It will require time and effort to learn a different programming language and associated libraries. Using a different programming language is therefore expected to increase the risk involved with the project to a level which is not acceptable.

In Chapter 3.6 these constraints will be evaluated with the purpose of identifying to what extent they restrict the solution.

3.3 Viewpoints

This sections describes the different viewpoint, lists which stakeholders have interest in particular views and which requirements are addressed by each view.

3.3.1 Logical Viewpoint

The logical view shows how functional requirements are mapped onto static modules and relationships between them. These modules translate into collaborations of classes in object-oriented programming languages. This translation is made by developers based on this view. Furthermore it eases understanding, and thus usability, of the framework by decomposing it in smaller modules. Apart from the clear interest developers have in the logical view, it is also of interest to the maintainers, users of the framework and management parties. Maintainers use the logical view in conjunction with the process view to gain understanding of the structure of the framework and to determine where the framework needs to be adapted in order to add support for new requirements. Users utilizing the framework to construct an application can determine from the logical view at which points the framework interacts with external applications. Managers can be assured the framework provides the required functionality.

The logical view uses a component diagram to give a high level overview of the framework and its relation with external components. An internal structure diagram is used to provide insight in the internal decomposition of the framework into modules and their interrelationships.

To summarize: Stakeholders with interest in this view:

- Developer: management, developers and maintainers.
- Acquirer: management, application developers.

Requirements addressed:

- Functional requirements.
- Maintainability: changeability.
- Usability.

3.3.2 Process Viewpoint

The process view gives a view of the system at run-time. As such it deals with processes, threads and issues of concurrency and distribution. In this role it primarily addresses issues of scalability and fault tolerance. It shows how abstractions of the logical view are mapped onto processes and threads. In general the process view complements the logical view and provides an understanding of the behavior of the system at runtime. As such it is of particular interest to developers of the framework, application developers and maintainers.

In the process view, sequence diagrams and process diagrams are used to further illustrate this. The process diagram shows how the functionality of framework is decomposed into processes and threads, whereas the sequence diagrams illustrate how common requests are handled by the system. They show how different elements of the framework work together at runtime to provide results for these requests.

Stakeholders with interest in this view:

- Developer: Developers and maintainers.
- Acquirer: Application developers.

Requirements addressed:

- Efficiency: scalability.
- Reliability: fault tolerance.

3.3.3 Development Viewpoint

The purpose of the development view is to provide a decomposition of the framework into smaller development tasks. It presents a general strategy to accomplish these tasks and gives a road map based on this strategy. This road map can be used to keep track of the progress of the development effort. The development view can therefore confirm the feasibility of the project and is of particular interest to the management of the developing organization. Clearly the developers are also concerned with this planning. In addition, the development view specifies and motivates different technologies used to implement the project. The choice of technologies will influence quality requirements of portability, applicability and conformance. In this view it is argued that the constraints specified in Section 3.2.3 are not overly constraining and that the project is feasible within these constraints.

Stakeholders with interest in this view:

- Developer: Management and developers.

Requirements and constraints addressed:

- Feasibility.
- Constraints: Java and Java Remote Method Invocation (RMI).

3.3.4 Physical Viewpoint

The physical viewpoint illustrates the mapping of processes onto computational nodes or servers and their physical interconnection. However, such a mapping will mostly depend on the application that uses the framework. For example a web service based on the navigation framework might deploy the framework on multiple servers. The requests from users can then be balanced across these servers. Therefore the physical view is omitted from the remainder our description; this view must be addressed by the application instantiating the framework.

3.3.5 Scenario Viewpoint

The purpose of the scenario view is to connect the other views, it points out how each view shows the system from a different angle and how they complement each other. It does so by specifying a limited number of relevant use cases. It shows how the system realizes the use cases and does so by redirecting readers with specific concerns to the view addressing this concern.

3.4 Logical View

The view described in this section conforms to the logical viewpoint. As such it explicitly models how the functional requirements of the system map to abstract modules. It also shows how these modules relate to each other and the external world. A component diagram is used to visualize interaction between the framework and the external components. To convey the internal structure of the framework, a composite structure diagram is used. Source code is written by developers based on the presented static structure. The logical view is therefore of particular interest to developers of the framework. Moreover it is also of interest to application developers, because it provides insight into the relation of the framework with external components. Furthermore management stakeholder can verify that the specified functional requirements are met.

To complement the logical view the process view in Section 3.5 shows the system in terms of processes and threads, and defines how communication is handled. The next section presents the general high level structure of the framework. The following section explains the internal structure of the framework in terms of modules and their interrelationships. In addition it assigns responsibilities to each module. Section 3.4.2 gives an overview of how external applications interact with the framework.

3.4.1 General Structure

The main architectural style in the design is the Model-View-Controller pattern. The framework implements the model and the controller, but leaves the implementation of views to applications will be build upon the framework. The usage of this pattern allows decoupling between the model and the views. This decoupling is especially important for our framework since the framework itself does not implement a view. The model consists of the Graph data structure and the algorithms which operate upon them, and the controller consists of various management modules as depicted in Figure 3.4. Application specific views communicate with the framework by means of the various ports described later in this section.

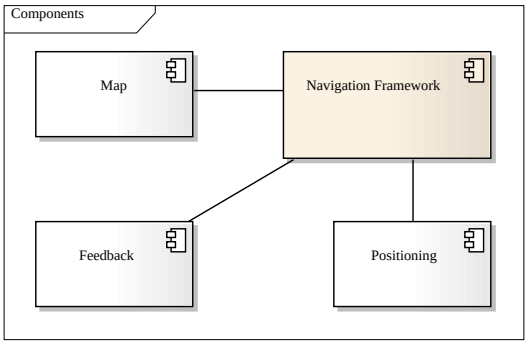

Figure 3.3: External Components

High Level Overview

Figure 3.3 gives a high level overview of the framework and visualizes connections between the framework and three external components (Map, Feedback and Positioning in the figure). The map component represents the environment from which a Graph data structure is constructed, this Graph is input to the framework and navigation algorithms are executed on this Graph. The map component is also responsible for communicating changes in the environment (think of blocked roads for example) which trigger a modification to the Graph structure. The feedback component is responsible for processing the output returned by the framework, possibly relaying and presenting them to users. Finally the positioning component is responsible for sending property (e.g. position) updates of entities.

3.4.2 Internal Structure

The navigation framework consists of five internal modules which work together to provide navigation functionality. In this section we discuss all five of them, reason what their responsibilities are, how they interact with other modules and how they relate to the functional requirements as specified in Section 3.2.1. The internal structure is visualized in Figure 3.4. This diagram type shows how components are composed of their internal parts. It shows provides and requires interfaces for parts and the interfaces exposed to external components by means of ports (labeled P# in the figure). Internal parts are assembled by connecting matching provides and requires interfaces.

Interaction with External Components

As shown in Figure 3.4, the framework communicates with the outside world by means of five defined ports (labeled P#) in the figure. Access to all ports except for P1 is done through a 'handler' object which forwards requests and messages to the appropriate component and returns results to the application. GraphLoaders, objects which construct a Graph from input data, have to be provided at start-up time. The Process View (Section 3.5) gives a thorough explanation of communication methods utilized and shows how requests are handled by different components within the framework.

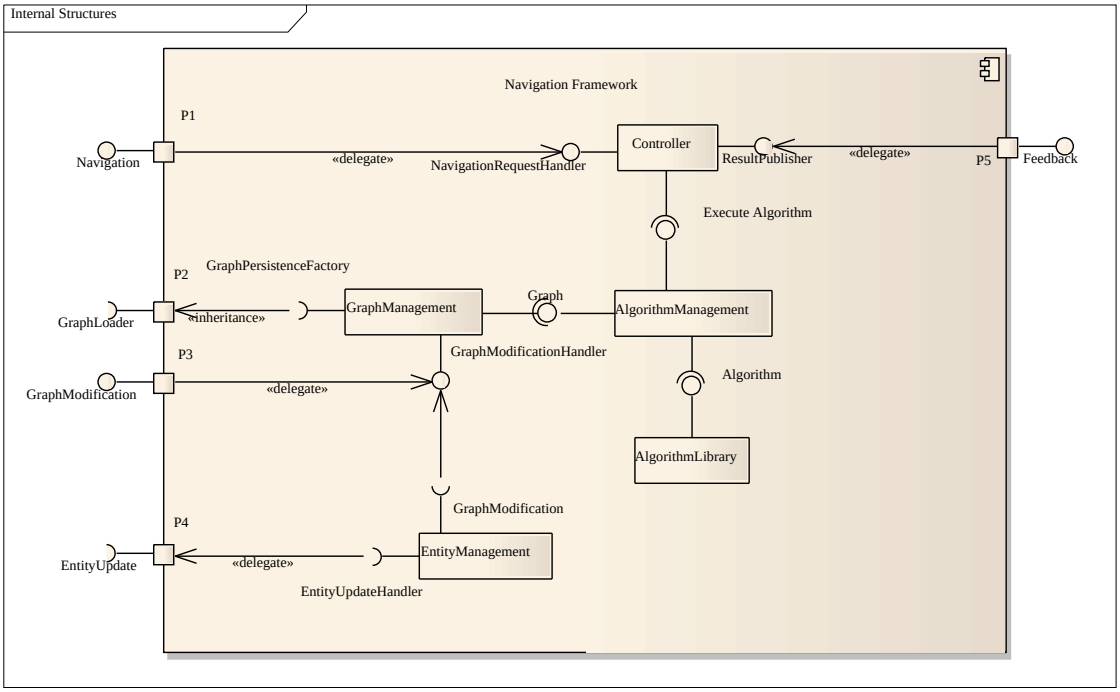

Figure 3.4: Framework Internal Structure - Ports are labeled P#

- Port 1: NavigationRequestHandler. The NavigationRequestHandler is the interface used by applications to make use of the navigational services provided by the framework.
- Port 2: GraphPersistenceFactory. The application developer supplies the framework with a class implementing the GraphLoader interface, which is able to construct a Graph from a desired data source. This class is registered with the GraphPersistenceFactory, which is called by the framework when a Graph is required.
- Port 3: GraphModificationHandler. This port provides an interface to modify properties or the structure of the graph. Requests to modify the graph are received here and are forwarded to the GraphManagement module. The modifications are applied here.
- Port 4: EntityUpdateHandler. This port requires an EntityUpdate event source, these events are forwarded to the EntityUpdateHandler.
- Port 5: ResultPublisher. Port 4 provides a service which publishes results returned by algorithms to subscribed applications.

Graph Management

The framework uses an internal Graph data structure, which has to be flexible in order to support a number of different algorithms. Furthermore in order to relate an external model, for example a geographic map, to the Graph model, it must support an identification property. A Graph can be constructed from a variety of data sources. One can think of common geographic datafile formats such as shape files[3] or GML[4]. For this reason the GraphPersistenceFactory has to be provided with a specific implementation of the GraphLoader interface by an instantiating application. The GraphPersistenceFactory uses this specific GraphLoader to construct a Graph on request. The framework itself will only provide a facility to directly serialize and deserialize the internal Graph data structure. In addition it should be possible to make modifications to the Graph at runtime, reflecting changes in the underlying model. One can think of changes to properties, adding or removing arcs. These modifications are supplied to the framework in the form of GraphModificationRequest, which are processed by the GraphModificationHandler. Some operations or modifications will require a 'lock' on the graph structure to ensure no algorithm is executed while the graph is in an inconsistent temporary state. Finally the GraphManager has the purpose of hiding the strategy used to retrieve and store Graphs from other modules.

The responsibilities of this module are listed below. Here each responsibility is linked to one or more functional requirements by a given Id (REQX). These ids correspond to those shown in Figure 3.1.

- Defines a Graph data structure on which the various supported graph algorithms can operate.(REQ16, REQ17).
- Possibility to link an external model with the Graph; this can be done by means of Identifiers (Ids) (REQ56).
- Facility to provide the framework with GraphLoaders. A GraphLoader is responsible for the construction of a Graph from some kind of data source. (REQ12). The framework itself will not provide an implementation; this is the responsibility of the application instantiating the framework.
- Facility to serialize a Graph (REQ13).
- A method to make modifications to a Graph. (REQ14).
- Regulate access to the graph (REQ57).

It has the following relations with other modules:

- External port P2 (described earlier).
- Provides the GraphModificationHandler which processes GraphModificationRequests issued through external port P3 and the EntityManagement module.
- Provides the AlgorithmManagement module with a Graph.

[3]ESRI Shapefile [21]
[4]Geography Markup Language (GML) [51]

Entity Management

The framework maintains a register of entities. An entity can, for example, represent a car or a person. Each entity has an identification, position and direction associated with it. Updates of these properties are handled by the EntityUpdateHandler and might result in updates to the Graph structure. For example consider a change of position in the scenario where each entity is modeled as a unit of flow. Such a change in position might result in the removal of a unit of flow at the node corresponding to the entity's original position and the addition of a unit of flow at the entity's new position. The component sends these GraphModificationRequests in batches so that multiple changes to the Graph can be applied in a single pass over the Graph. This module is actually optional: if the application does not require these advanced features it can be omitted.

The EntityManagement module has the following responsibilities:

- Addition and removal of entities (REQ4, REQ5).
- Modification of an Entity's properties (REQ6).
- When appropriate sends a Message to the GraphModificationHandler to trigger a modification of the Graph (REQ54).

It has the following relations with other modules:

- External port P3 (described earlier).
- Makes use of the GraphModificationHandler provided by the GraphManagment module.

Algorithm Management

The AlgorithmManagement contains generic logic for configuration and execution of algorithms and returning their results; as such it is the central unit in the framework. Furthermore it contains logic to support concurrent execution of algorithms, in order to support a scenario in which the framework receives online queries for route information from the application.

The AlgorithmManagement module has the following responsibilities:

- Execute preprocessing steps (REQ22).
- Execute graph transformations (REQ22).
- Execute graph algorithms (REQ19).
- Configure algorithms (REQ20).
- Monitor algorithms (REQ21).
- Execute algorithms concurrently (REQ58).
- Construct a result (REQ59).

It has the following relations with other modules:

- It requires the AlgorithmLibrary to supply algorithms and operations.
- Graph data structures are provided by the GraphManager component.
- It receives commands from the Controller.

Algorithm Library

The framework will provide navigational services by executing graph algorithms on a supplied Graph. This module contains a collection of algorithms and operations which can be applied to the Graph data structure. Each algorithm can be composed of multiple suboperations, possibly multiple implementations of a specific operation and certain functionality can be added by means of decorators[5]. For example, in this way extensive monitoring functionality can be added. Configuration and composition of an algorithm is the responsibility of the AlgorithmManager module. It is essential that the AlgorithmLibrary is constructed in such a way that it is easy to add new algorithms and operations. Additionally, this module contains operations to validate and possibly transform a Graph. E.g. checks to ensure certain required properties are set, and operations to transform an undirected graph to a directed one. Chapter 2 provides a description of the different algorithms and concepts.

Responsibilities are subdivided in two categories, namely, responsibilities in the form of concrete graph algorithms and responsibilities dealing with transformations and consistency checking of the Graph structure.

Algorithms:

- Breadth First Search (REQ30).
- Dijkstra's algorithm for shortest path (REQ28).
- Push-Relabel algorithm for maximum flow (REQ29).
- Successive-approximation by Cost Scaling for minimum-cost flow (REQ39).

Operations:

- Check graph for consistency (REQ32).
- Check which algorithms are supported by a graph (REQ33).
- Create a residual network (REQ34).
- Transform an undirected graph to a directed graph (REQ35).
- Convert double / float properties to integers (REQ36).

Additional operations, such as REQ37-47 will be implemented when necessary.
This module has a single relation: it provides algorithms to the AlgorithmManager module.

[5]Decorator Pattern [25]

Controller

The Controller module starts and initializes configures the framework. It will provide utilities such as logging and handing out unique Ids to the framework. IDs are extensively used to relate the outside environment to the data structures used by the framework. For example arcs have an Id which points to a connection or road in the environment. In this way the results returned by the algorithm can be related to the environment. Furthermore it provides external components with an entry point to the framework.

The Controller has the following responsibilities:

- Start and initialize framework (REQ61).
- Provide utilities to enable logging and configuration of the framework (REQ10, REQ11).
- Provide utility to create framework wide unique IDs (REQ62).
- Start a server which exports services provided by the framework (REQ23,REQ24,REQ25,REQ26,REQ27).

It has the following relations with other modules:

- External port P1.
- External port P5.
- Uses the AlgorithmManager to execute algorithms.
- Forwards results returned by AlgorithmManager to ResultPublisher, which makes the result available to client applications.

3.5 Process View

The process view gives a view of the system at runtime in terms of processes and threads. It relates abstractions introduced in the logical view to processes and threads, i.e. which thread controls certain computations, and specifies what protocols are used for communication between them. Therefore the process view mainly deals with quality requirements such as scalability, reliability, fault tolerance and to a certain extent analysability. This section uses sequence diagrams, and a custom diagram to clarify the process and thread design of the framework.

The logical view suggests that the main work of the framework is done by the AlgorithmManagement and to a lesser extent by the GraphManagement module. Here the actual algorithms are executed and modifications are made to the graph. Furthermore, we expect that in virtually any application based on the framework the actual execution of graph algorithms will be the computationally most intensive part. We emphasize the fact that there are several points (ports) where external components interact with the framework during runtime. Specifically they interact by means of the NavigationRequestHandler, GraphModificationHandler, EntityUpdateHandler and ResultPublisher.

3.5.1 Framework Concurrency Model

The requirement (REQ58) directly suggests a multi-threaded approach for the execution of algorithms and graph modifications and transformations. We aim to create a multi-threaded single process initially. However it is a clear design goal to enable straightforward migration to a multi-process application. A multi-threaded application allows multiple tasks to be executed concurrently and can improve efficiency and responsiveness. Furthermore, computationally intensive tasks can be isolated within separate threads. This improves reliability and fault tolerance, because if a fault occurs in such a thread, then this particular thread can be terminated without effecting other threads and the application as a whole. A multiprocess application offers additional benefits of scalability. Here computationally intensive parts can be isolated within a separate processes, which can run on dedicated servers. If properly designed and constructed this allows an increase throughput by deploying additional servers.

Thread Decomposition

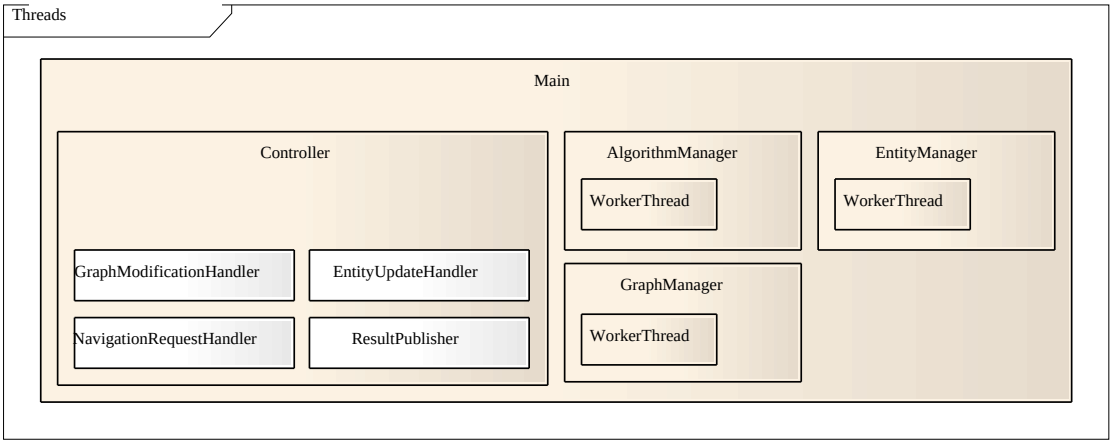

Figure 3.5: Thread Composition

Figure 3.5 shows how the framework is decomposed into a thread hierarchy. In this figure each non-white box denotes a separate thread. The white boxes represent certain functionality and are included for clarity. However, they do not represent distinct threads. We can identify the 'Main' thread which starts up the application and four sub threads: the Controller, AlgorithmManager, GraphManager and the EntityManager. The Controller thread contains logic to receive requests and forward these to the appropriate manager, as well as logic to relay results back to clients of the framework. Each of the Algorithm Manager, Graph Manager and Entity Manager contains a thread pool with worker threads that execute the computationally intensive work. Communication between the threads is done by in process calls.

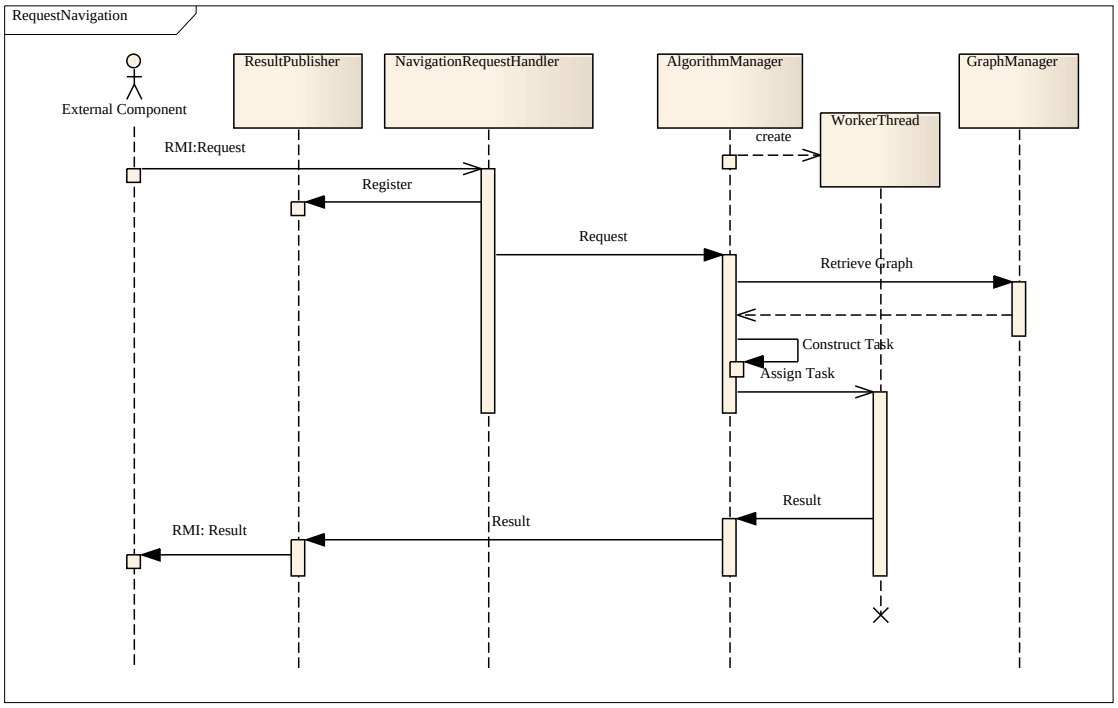

Figure 3.6: Request Navigation Sequence

Control Flow

Figure 3.6 contains a sequence diagram. It illustrates which objects are involved in handling navigation and graph modification requests, respectively. The figure shows how an external component makes a navigation request to the NavigationRequestHandler. The NavigationRequestHandler registers the request with the ResultPublisher, such that the ResultPublisher knows which parties are interested in the result of the computation. Next the NavigationRequestHandler forwards the request to the AlgorithmManager, which constructs and schedules a task to be executed. Construction of a task is done by selecting an algorithm and retrieving the Graph data structure. Onces a task is scheduled it will be assigned an idle WorkerThread in the thread pool. This WorkerThread will execute the algorithm. Once finished the WorkerThread will construct a result and return it to the AlgorithmManager, which forwards it to the ResultPublisher which in turn sends it to interested parties.

It is important to note that most communication between different parts of the framework are asynchronous. This improves decoupling and isolates the different components from each other.

3.5.2 Future Improvements

The process and thread design as given in the previous section enables migration to a multiprocess architecture. Here we shortly describe two change scenarios.

Because external applications communicate through an isolated set of 'handlers' (specifically the NavigationRequestHandler, GraphModificationHandler, EntityUpdateHandler and ResultPublisher) it is fairly easy to create a layer in front of these handlers. For example, the main thread could start a Java RMI[6] server. This server would receive requests from external processes, which can be located on a different physical machine. The RMI server would then forward these request to the appropriate handlers.

A further improvement can be made by making the framework itself distributed. This can be achieved relatively easily because the different components communicate with one another asynchronous. It is possible to create a proxy between the different components which is responsible for sending and receiving messages across process or machine boundary.

It should be noted, however, that communication between different processes, and especially if this communication has to travel across a network, is less efficient then in process communication. Furthermore not all applications that build upon the framework will require this distribution. A solution can be to enable or disable this distribution by configuration.

3.6 Development View

The development view provides a view of the framework from a development perspective. It subdivides the complete development effort in smaller tasks and considers the dependency between those tasks. Based on this dependency a feasible planning is made. This sections defines a couple phases and specifies for each what functionality, in terms of requirements, must be implemented. In addition to the planning this chapter discusses technologies used in the project. Therefore this view is of prime importance to developers, but it is also interesting to managers as it provides a means to track the progress of the project and assures the project is feasible.

3.6.1 Development of the Framework in Perspective

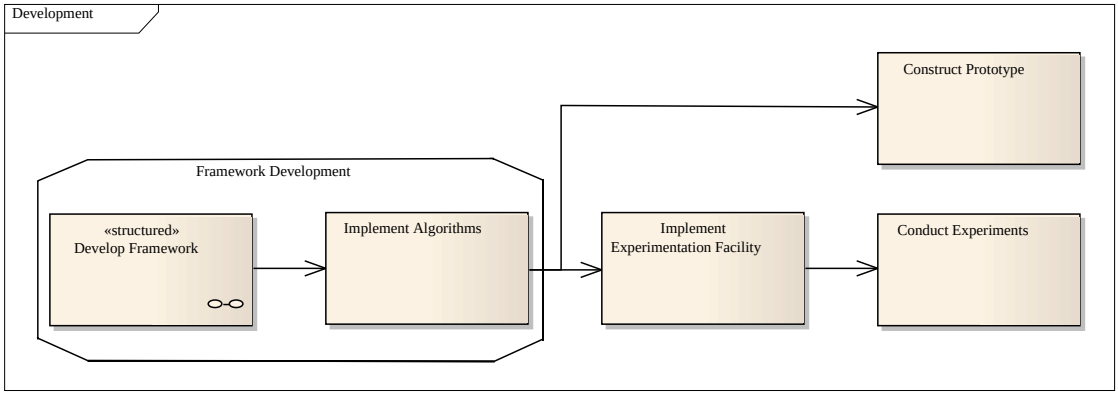

Figure 3.7: Development Phases

[6]Java Remote Method Invocation: whitepaper: [47]

Untill now we limited the discussion to the framework itself. However, the objectives (Section 1.3.2) of the project go beyond constructing a navigation framework. In this section the development of the framework is put in perspective of the larger encapsulating project. Figure 3.7 visualizes the different development items within the project. We can see the development of the framework is subdivided in two items, namely the development of the framework itself and subsequently the implementation of various algorithms. In addition three more items are identified. These items concern the construction of a prototype application based on the framework and the measurement of the performance of the implemented algorithms.

3.6.2 Development of the Framework

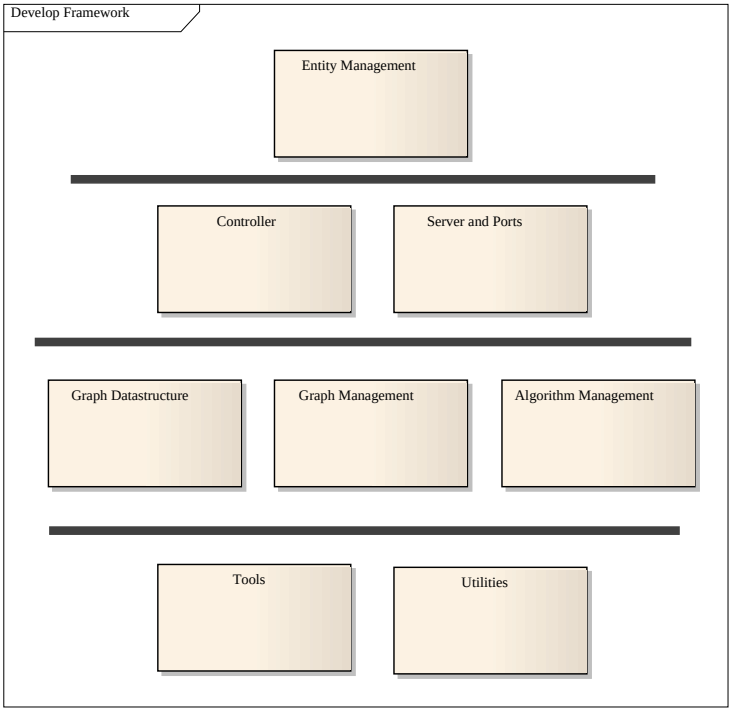

Figure 3.8: This figure shows hierarchy of dependencies between tasks. Tasks higher in the hierarchy depend on tasks below it.

Figure 3.8 is a more detailed view of the 'Develop Framework' item seen in Figure 3.7. It shows the dependencies between tasks identified within the development of the framework itself. Elements shown in the figure largely correspond to the various modules described in the logical view, but the 'Utilities' item indicate a general dependency of all development effort on tools, for example the IDE, and utilities, such as logging, configuration and the unit test framework. The figure shows a hierarchy of tasks. This hierarchy should be interpreted as follows: development elements appearing on some level only depend on the development of other elements at the same level and those on

lower levels. Hence we can see that the development of the Ports to the framework are dependent on the Algorithm and Graph Management and Graph Data Structure modules.

Development Approach

The structure of the general development process itself is to some extent inspired by agile methods. That is to say a minimal functional version of the framework will be constructed first, and this first version will be improved incrementally by adding additional functionality. This approach will reduce risk, because there will be a working framework at any point in time and it is an especially suitable method given the relative short nature of the project and small number of people involved with the development. Furthermore the development of the framework is fairly easily decomposed into small tasks, which can be seen next.

Road Map

The development of the framework is subdivided into three phases, which are listed in tables 3.1, 3.2 and 3.3. For each of these phases a number of tasks is given, each linked to the requirements (given in Figure 3.1) they intend to satisfy. Nine weeks are assigned to implement the framework. In phase one and two the framework itself is implemented. These two phases combined correspond to the 'Develop Framework' element in Figure 3.7. Phase one results in a minimal 'working' application and this 'working' property is maintained during phase two, when additional functionality is iteratively added. For phase one, two weeks are reserved and phase two is realized in three weeks. In phase three, during the last four weeks, the various algorithms are added. This phase corresponds to the 'Implement Algorithms' element in Figure 3.7.

Documentation

The framework is delivered with documentation.

3.6.3 Technologies Used

In this section the technologies used for developing the framework will be discussed.

- **Programming language**: Java and Java RMI The framework will be developed in Java. Java is a mature, full fledged object-oriented programming which is developed with the specific purpose of ensuring portability between different operating systems. There are a huge number of libraries available for Java including libraries to handle XML and various Geographic Information Systems (GIS) related file formats and standards. Furthermore it has support for creating distributed applications based on Java RMI. As such the constraints to use Java and Java RMI as specified in Section 3.2.3 are not considered restricting.
- **Logging**: The Apache commons logging[7] Application Programming Interface (API) will be used in combination with the Log4j[8] library which implements this API.
- **Testing**: JUnit[9].

[7] Commons Logging: `http://commons.apache.org/logging/commons-logging-1.1.1/index.html`
[8] Log4j: `http://logging.apache.org/log4j/1.2/index.html`
[9] JUnit: `http://www.junit.org/`

Table 3.1: Phase 1: Minimal Initial Implementation

Item	Description	Requirements
Controller	• Utilities: Logging, Configuration, Id facility, Unit Test facility • A minimal implementation required to initialize and start the framework	REQ10, REQ11, REQ61, REQ62
GraphManager	• Graph data structure • GraphLoader facility	REQ12 REQ16, REQ17, REQ18, REQ56
AlgorithmManager	• Single threaded execution of algorithms and operations	REQ19, REQ22
AlgorithmLibrary	• Basic functionality • BFS	REQ30

Table 3.2: Phase 2: Improvements

Item	Description	Requirements
GraphManager	• Threaded Implementation • Save graphs • Modify graphs • Regulate access to graph	REQ13, REQ14, REQ57
AlgorithmManager	• Threaded implementation • Facility to allow algorithm configuration • Facility to monitor algorithms • Construct result	REQ20, REQ21,REQ58, REQ59
EntityManager	• Adding, removing and updating entities	REQ4, REQ5, REQ6
Controller	• NavigationRequestHandler • GraphModificationHandler • ResultPublisher • EntityUpdateHandler	REQ23, REQ24, REQ25, REQ26, REQ27

Table 3.3: Phase 3: Implementation Algorithms

Item	Description	Requirements
Shortest path	• Dijkstra	REQ28
Maximum flow	• Push-relabel	REQ29
Minimum-cost flow	• Scaling push-relabel	REQ39
Operations	• Check graph for consistency • Check which algorithms are supported by the raph • Create residual graph • Transform undirected graph to directed	REQ32, REQ33, REQ34, REQ35
EntityManager	• Modify graph based on received entity updates	REQ57

3.7 Scenario View

As mentioned in the introduction the purpose of the scenario view is to connect the other views. It does so by explaining several use cases and connecting those use cases to the previous views. In this section we first describe three subfunctions by means of the use cases given in Table 3.4, Table 3.5 and Table 3.6. The final use case (given in Table 3.7) shows how the framework uses these three subfunctions to satisfy a request made by an external application.

Consider this final use case, which is given in Table 3.7. This use case illustrates how an external application requests the shortest route between two locations. The logical view shows how different modules collaborate in order to handle requests and it lists the responsibilities of each of these modules. This view illustrates how a request enters the system via one of the ports and how it is delegated to the appropriate handler (the NavigationRequestHandler in this particular case). Furthermore it shows how the AlgorithmManagement module requires a graph, which is provided by the GraphManagement module. As such this view provides an understanding of the structure of the framework as a whole and how different parts collaborate to implement specific requirements. The process view is provided to complement the logical view. For this particular case it shows how the AlgorithmManager constructs a Task, which is executed by a WorkerThread. In general the process view can be observed to learn how different parts of the framework communicate and it shows which process or thread executes certain tasks. It deals with issues of concurrency and reliability. Finally the development view addresses the planning with respect to the development of the framework, it indicates when certain functionality is expected to be implemented and can be used to track the progress of the project.

Table 3.4: Use Case: Request Graph. This use case shows how a component within the framework requests a graph from the GraphManagement component. Such a request is made by the components that are responsible for executing algorithms and performing modifications to the graph

Use case:	Request Graph
Level:	Subfunction
Primary Actor:	NavigationFramework

A graph is required by some internal component in order to satisfy a request.

1. The component requests the graph from the GraphManager.
2. The GraphManager:
   (a) Checks if the requested graph is registered.
   (b) Checks if the requested graph is available. If so:
   (c) Locks the graph and returns the graph.
3. The component that requested the graph, receives the graph.

Table 3.5: Use Case: Instantiate Algorithm. This use case shows how the framework instantiates an algorithm.

Use case:	Instantiate Algorithm
Level:	Subfunction
Primary Actor:	NavigationFramework: AlgorithmManagement component

AlgorithmManagement requires an algorithm to satisfy a request.

1. AlgorithmManagement asks the AlgorithmLibrary to instantiate the algorithm and supplies a classname and configuration properties.
2. The AlgorithmLibrary:
   (a) Checks if an algorithm with the given classname is registered. If so:
   (b) Instantiates the algorithm.
   (c) Configures the algorithm.
   (d) Instantiates components part of the algorithm (for example optional heuristics), if required.
   (e) Adds those components to the algorithm.
   (f) Returns the instantiated and configured algorithm.
3. AlgorithmManagement receives the algorithm.

3.8 Conclusion

This chapter presented our design for the framework. To structure the design we followed the IEEE 1471[50] recommended practice. It therefore provided an answer to research questions 7 and 8.

In the first section of this chapter we identified stakeholders with concerns in the project. Section 3.2 listed functional and quality requirements. These requirements are based on the project description in Chapter 1, the aforementioned concerns of stakeholders in the project and the authors' own ideas. The requirements have been discussed with an experienced software engineer at Logica to ensure their completeness and clarity. This answered Research Question 7:

What are the requirements for such a framework?

Based on these requirements a design of the framework was created. In accordance with the IEEE 1471 recommend practice we provided multiple views. These views were selected based on Kruchten's 4+1 Model of Software Architecture [43]. The logical view provided insight into the static structure of the framework, how the framework is decomposed into five modules and how functional requirements are mapped to these five modules. Furthermore, in this view we can observe that Graph Management component is responsible for processing modifications to the graph

Table 3.6: Use Case: Execute Algorithm. It shows how an algorithm is executed by a separate thread.

Use case:	Execute Algorithm
Level:	Subfunction
Primary Actor:	NavigationFramework: AlgorithmManagement component

AlgorithmManagement component requires an algorithm to be executed on a graph in order to satisfy a request.

1. AlgorithmManagement:
   (a) Constructs a task that will execute the algorithm on the given graph.
   (b) Schedules the task for execution.
2. The task is assigned to a WorkerThread and the WorkerThread executes the task.
3. The task:
   (a) Adds the given graph to the algorithm.
   (b) Initializes the algorithm.
   (c) Executes the algorithm.
   (d) Constructs a result.
   (e) Returns the result to the AlgorithmManager.
4. The AlgorithmManager receives the result.

structure and properties. This provides an answer to Research Question 5:

In what way can we incorporate dynamic changes of structure and properties?

This view is complemented by the process view, which shows the framework from the perspective of runtime threads and processes. Our process and thread design focused on decoupling the various components in our design. In this way we aim to create a framework, which is easily extensible and is decomposable in multiple processes.

In the development view the development effort was broken down in several smaller tasks and the dependencies between these tasks were identified. Based on this decomposition and the dependencies a detailed road map was given. Finally in the scenario view we used a use case to explicitly connect the three previous views.

This collections of views gives an answer to Research Question 8:

What design satisfies the requirements?

Table 3.7: Use Case: Request Algorithm Execution. This use case explains how an external application makes a concrete request to the framework. It shows how the framework step-by-step satisfies the request by executing previously described use cases.

Use case:	Request Algorithm Execution
Level:	User Goal
Primary Actor:	External Application

External Application sends a request to calculate the shortest route from 'MyHome' to 'MyShoppingMall' in graph 'MyCity' using the 'Dijkstra' algorithm.

1. The Framework receives request, and:
   (a) Registers the external application with ResultPublisher.
   (b) Requests the 'MyCity' graph from the GraphManagement component by executing the *Request Graph* use case.

   Once the required graph is acquired:
2. The request and the graph are forwarded to the AlgorithmManagement component. The AlgorithmManagement:
   - Requests 'Dijkstra' by executing the *Instantiate Algorithm* use case.
   - Executes the algorithm by executing the *Execute Algorithm* use case.
   - Receives the result and forwards it to ResultPublisher.
3. The ResultPublisher publishes result to the external application.

Part III

Implementation

Chapter 4

Implementation of Algorithms

Chapter 2 aimed provide a fairly broad overview of relevant topics. It established definition and assumptions, which we will continue to use in this chapter. Furthermore it concluded the Push-Relabel and Successive-approximation by Cost Scaling are state-of-the-art algorithms for the maximum flow and minimum-cost flow problems, and that they are suitable to be implemented by the framework. In this chapter we describe these algorithms and their implementation in more detail. Therefore it gives an answer to research question 3:

Can these algorithms be implemented such that they support the framework?

4.1 Graph Data Structure

The graph data structure we use is provided by the JGraphT[1] library, which is a flexible high performance library for Java. It maintains a HashMap of all vertices and two lists to store all incoming and outgoing edges, which in turn have a pointer to both vertices. Chapter 5 further discusses this library and the design decisions made.

4.2 Push-Relabel

In this section we will discuss the push-relabel method by Goldberg and Tarjan [29] and practical results obtained using this method. They relax the flow conservation constraint (2.3), instead they use the following condition:

$$\sum_{w \in V - \{s\}} f(v, w) \geq 0 \tag{4.1}$$

A flow satisfying 2.1, 2.2 and 4.1 is called a pseudo flow. Using pseudo flows it is possible for the amount of flow into a node to exceed the amount of flow leaving that node. This excess flow is defined as: $e(v) = \sum_{v \in V} f(v, w)$. Additionally the algorithm maintains a labeling d of all vertices such that $d(t) = 0$, $d(s) = n$ and $\forall (v, w) \in E_f : \ d(v) \leq d(w) + 1$. Let $d_G(v, w)$ be defined as the

[1] `http://jgrapht.sourceforge.net/`

minimum number of edges on the path from v to w in G. Finally, a vertex $v \in V - \{s, t\}$ is called active if $e(v) > 0$. The algorithm is initialized by filling all arcs leaving the source to maximum capacity and setting all labels d, except $d(s)$, to 0. $d(s)$ is set to n. The complete algorithm is shown as Algorithm 2 and uses two operations, push and relabel.

The push operation is applicable if a node v is active and there is an arc out of v with positive residual capacity. It pushes the maximum amount of flow possible down arc (v, w) to node w with a smaller label. This maximum corresponds the minimum of the nodes excess $e(v)$ and the residual capacity $r_f(v, w)$. If the push completely fills the capacity of the arc its called a saturating push, otherwise a non-saturating push. In the latter case no excess flow is left in v.

The relabel operation is applicable if node v is active and for all residual arcs (v, w) out of v the label $d(v) \leq d(w)$. The operation sets the label of v to the minimum of all labels $d(w) + 1$.

The main algorithm then continues to apply push and relabel operations in any order while any of these are applicable. Intuitively the algorithm either pushes more excess flow 'downstream' to nodes with a lower label or it increases the 'height' of a node by applying the relabel operation. Each relabel operation will create at least one new possibility to push more flow downstream. Initially, the algorithm will push flow in the direction of the sink. Eventually, however, as the 'height' of nodes increases and their label becomes larger then the source's label, excess flow will be returned to the source. The algorithm terminates when there are no active vertices anymore. Therefore none of the nodes has an excess flow and the pre-flow is actually a maximum flow.

Algorithm: Push-relabel

`// Initialize Pre-flow`
foreach $(v, w) \in E$ **do** $f(v, w) \leftarrow 0$, $f(w, v) \leftarrow 0$;
foreach $v \in V$ **do** $f(v, s) \leftarrow c(v, s)$, $f(s, v) \leftarrow -c(v, s)$;
foreach $v \in V - \{s\}$ **do** $d(v) \leftarrow 0$, $e(v) \leftarrow f(s, v)$;
$d(s) \leftarrow n$;

`// Main loop`
while *Push* ***or*** *Relabel is applicable* **do**
  Select and apply applicable operation;

Algorithm 2: Push-Relabel for the Maximum Flow Problem

Function:push
Input: (v, w)
if v *is active* ***and*** $r_f(v, w) > 0$ ***and*** $d(v) = d(w) + 1$ **then**
  Let $\delta = \min(e(v), r_f(v, w))$ and;
  $f(v, w) \leftarrow f(v, w) + \delta$;
  $f(w, v) \leftarrow f(v, w) - \delta$;
  $e(v) \leftarrow e(v) - \delta$;
  $e(w) \leftarrow e(w) + \delta$;

Function `push`

Function:relabel

Input: v
if v *is active* ***and*** *for all* w, $(v,w) \in E_f$ *we have* $d(v) \leq d(w)$ **then**
  $d(v) \leftarrow \min_{(v,w)\in E_f}\{d(w)+1\}$;

Function `relabel`

4.2.1 Strategies and heuristics

This generic algorithm allows considerable freedom in the sense that there is no specification on the order in which the basic operations are applied. A simple implementation makes use of a FIFO queue to store active vertices. In each step it takes the first vertex v from the queue and continues to apply push operations on v as long as these are applicable. If new vertices become active as a result of a push operation these are added to the rear of the queue. When no push operation is applicable anymore it relabels v and adds v to the rear of the queue if it is still active. However alternative strategies are possible. Goldberg and Tarjan [29] show the algorithm based on a FIFO queue has a complexity of $O(n^3)$. By using a dynamic tree data structure [58] an $O(nm\log(n^2/m))$ algorithm is obtained. Cheriyan and Maheshwari [13] use a strategy which applies pushes to the vertex with maximal excess. This approach results in an improved bound of $O(n^2\sqrt{m})$. Alternatively the vertex with the highest label can be selected for the push operation. Ahuja and Orlin [4] use excess scaling to obtain an algorithm with a running time of $O(nm + n^2\log(U))$, where U is an upper bound on the arc capacities.

The global relabeling heuristic proposed in [29] works very well in practice. Periodically all distance labels are updated and the label of all nodes $v \in V$ are set to the minimum of $d_{Gf}(v,s)+n$ and $d_{Gf}(v,t)$ by performing a BFS in the residual graph starting in the source and sink nodes. At last we remark that the algorithm allows intuitive parallelization. This was reported in the original paper and by Anderson and Setubal [5].

4.2.2 Implementation

The behavior of our implementation of the Push-Relabel algorithm can be adapted in three important ways. It is possible to select one of three implementations of the active vertex queue. The three implementations provided are one based on a First In First Out (FIFO) queue, a Last In First Out (LIFO) stack and a priority queue. The priority queue implementation sorts the vertices based on their label in descending order and returns the vertex with the highest label. The global relabel heuristic is also implemented and can added to the algorithm. Finally, it is possible to have the algorithm calculate a feasible flow, instead of a maximum flow. This is possible by supplying a demand for all of the sources and sinks and setting the 'bounded' property to true. This mode of operation is used as a subcomputation in our Successive Approximation by Cost Scaling (SA) algorithm.

4.3 Successive Approximation by Cost Scaling

In this section we will discuss a minimum-cost flow algorithm based on cost scaling, which is closely related to the push-relabel algorithm for the maximum flow problem described earlier.

For a given price function, $p : V \to \mathbb{R}$, the reduced cost of an arc $u_p(v, w)$ is defined as $u_p(v, w) = u(v, w) + p(v) - p(w)$. A residual arc (v, w) is admissible if $u_p(v, w) < 0$. E_A is the set of admissible arcs and $G_A(V, E_A)$ is the graph induced by this set. Again we define the excess of a node $e(v)$ with $e(v) = \sum_{v \in V} f(v, w)$. Given a constant $\epsilon > 0$ a flow f is said to be ϵ-optimal with respect to p if $\forall (v, w) \in E_f \ u_p(v, w) \geq -\epsilon$. As is shown by [3]: if $\epsilon \geq U$ then any feasible flow is ϵ-optimal and if $\epsilon \leq 1/n$ then any ϵ-optimal flow is an optimal flow given the assumption that all edge costs are integer (Assumption 2.1.3).

The SA algorithm by Goldberg and Tarjan [27] is given in Algorithm 5. It maintains an ϵ-optimal flow throughout its execution. After initializing the value of ϵ with U it continues to refine its solution till the ϵ-optimal flow is an optimal flow. Refinement is done by iteratively applying the refine function. This function reduces ϵ by a factor α (typical values are between 4 and 8). It then converts the current ϵ-optimal flow in a $\frac{\epsilon}{\alpha}$-optimal pseudoflow[2]. By repeatedly applying push and relabel the $\frac{\epsilon}{\alpha}$-optimal pseudoflow is transformed into a $\frac{\epsilon}{\alpha}$-optimal flow. The push function is applicable to (v, w) if v is active and there is an outgoing arc (v, w) with positive residual capacity (r_f) and negative reduced cost (a_p). The relabel function is applicable to (v) if v has no residual arcs $(v, w) \in E_f$ with negative reduced cost. Finally, when $\epsilon \leq 1/n$ the current flow is optimal and is returned. A correctness proof is given in [27].

Algorithm: Successive Approximation

```
// Initialize
ε ← C;
foreach v ∈ V do p(v) ← 0;
f ←(execute some max-flow algorithm);
while ε ≥ 1/n do
    (ε, f, p) ← refine(ε, f, p);
return f;
```

Algorithm 5: Successive Approximation by Cost Scaling for the Minimum-Cost Flow Problem

Function:refine

Input: ϵ, f, p

```
ε ← ε/2;
foreach (v, w) ∈ E do
    if u_p(v, w) < 0 then
        f(v, w) ← c(v, w);
while Push or Relabel is applicable do
    Select and apply applicable operation;
return ε, f, p;
```

Function `refine`

[2]Recall a pseudoflow is a flow satisfying 4.1 instead of 2.3: which allows vertices to have excess flow.

Function:push

Input: v, w
if v *is active* ***and*** $r_f(v,w) > 0$ ***and*** $u_p(v,w) < 0$ **then**
  Let $\delta = \min(e(v), r_f(v,w)$ and;
  $f(v,w) \leftarrow f(v,w) + \delta$;
  $f(w,v) \leftarrow f(v,w) - \delta$;
  $e(v) \leftarrow e(v) - \delta$;
  $e(w) \leftarrow e(w) + \delta$;

Function `push`

Function:relabel

Input: v
if v *is active* ***and*** $\forall w$, $(v,w) \in E_f$ *we have* $u_p(v,w) \geq 0$ **then**
  $p(v) \leftarrow \max_{(v,w)\in E_f}\{p(w) - u(v,w) - \epsilon\}$;

Function `relabel`

4.3.1 Strategies and Heuristics

Goldberg [27], Goldberg and Kharitanov [30] and Bland *et al.*[10] propose several strategies and heuristics that may improve the practical running time of the algorithm. In this section we discuss several of them.

Selection strategy

Like the Push-Relabel algorithm the order in which the push and relabel operations are applied is not defined. The discharge procedure selects an active vertex v and continues to apply push and relabel operation on this vertex until it becomes inactive. In addition we need some data structure to store active nodes and which defines the order in which active vertices are selected. Possibilities include a FIFO queue, a LIFO stack or a list which is topologically sorted with respect to the admissible graph. Based on the latter choice two different strategies can be formulated, namely the first-active and wave methods[30]. However, these two methods require more computation and both Goldberg [30] and Bland *et al.*[10] conclude that the method based on a queue is superior.

Simple Relabel

Instead of the 'complex' relabel operation a simple alternative operation can be used.

Function:simple-relabel

Input: v
if v *is active* ***and*** $\forall w$, $(v,w) \in E_f$ *we have* $u_p(v,w) \geq 0$ **then**
  $p(v) \leftarrow p(v) - \epsilon$;

Function `simple-relabel`

Lookahead Heuristic

Goldberg [27] reported, and we observed in preliminary executions, that in practice the following scenario occurs frequently. Say vertex v is active and is selected for discharging. The algorithm then pushes excess flow from v to a neighbor w. Then w either is, or becomes, active. Now the first time thereafter, that w is selected for discharging, the algorithm immediately pushes the excess flow from w back to v. This scenario only occurs when at w does not have any other outgoing admissible arcs. A heuristic which attempts to prevent this scenario is given as Function push-lookahead. This operation only pushes flow from v to w if either w has negative excess, or w has outgoing admissible arcs. Otherwise w is relabeled. Goldberg [27] shows, this does not violate the ϵ-optimality constraint.

Function:push-lookahead
Input: v, w
if v *is active* ***and*** $(e(w) < 0$ ***or*** $\exists(w, x) \in E_A)$ **then**
  push(v,w);
else
  relabel(w);

Function `push-lookahead`

4.3.2 Implementation

Our implementation of the SA algorithm can be configured with several options. It is possible to either choose the complex or the simple variant of the relabel operation. It is also possible to set the desired epsilon scale factor (α) and a different implementation of the active vertex queue can be supplied. Once again an implementation based on a queue, stack or highest label priority queue can be selected. Furthermore, we implemented the push-lookahead heuristic. The algorithm uses the Push-Relabel algorithm to find an initial feasible flow. If no such flow is found, then execution is aborted indicating that no feasible flow exists. Once the algorithm finds an optimal flow, it uses BFS to calculate the minimum cost value. Finally, we remark that our implementation of the SA algorithm shares a considerable amount of code with the Push-Relabel algorithm.

4.4 Miscellaneous Algorithms

Besides the Push-Relabel and the Successive Approximation by Cost Scaling algorithm, the framework also contains implementations of the Breadth First Search, Dijkstra's shortest path and Bellman-Ford's shortest path algorithms. Dijkstra's algorithm is previously discussed in Section 2.2.2 and both other algorithms are well known. This section only reports some implementation specific details of these algorithms and the reader is advised to consult the book by Cormen *et al.*[14] for a complete description. This sections also explains our method of creating a time expanded graph.

Breadth First Search

Our implementation of the BFS is designed to be easily extendable to perform various operations and transformations. The algorithm can be configured with multiple parameters. By default the algorithm will select a random start vertex and perform BFS from that vertex. Alternatively a set of start vertices can be provided. In this case the algorithm will start BFS from each of these vertices successively, unless this new start vertex has already been visited. Furthermore, a parameter 'crossComponentSearch' can be supplied. This causes the algorithms to expand its search to include disconnected components. It does so by repeatedly starting the BFS first from the set of start vertices and then from a vertex randomly selected from the remaining set of not-yet-visited vertices. Finally, a parameter 'reversedSearch' can be set. This causes the algorithm to traverse directed edges in opposite direction e.g. from head to tail.

Dijkstra's Shortest Path Algorithm

We use a Fibonacci heap to maintain the vertex queue. Furthermore it is possible to supply the algorithm with an alternative distance function. By default the algorithm uses the weight of an edge as distance. Our implementation of Dijkstra can be configured to either calculate the shortest path between two given vertices, or to calculate the distance between a start vertex and all other vertices. In the former case both a start vertex and end vertex must be supplied. In the latter case the 'singeDestinationProperty' should be set to false.

Bellman-Ford

The implementation of the Bellman-Ford algorithm is configurable with the same parameters as Dijkstra's algorithm. It uses a HashMap to maintain it's 'visited vertices' list.

Time Expand Graph Operation

This operation is implemented using BFS. Its behavior is configured by two additional properties. The 'interval' (I) property indicates the the time window of interest and the 'resolution' (R) property influences the accuracy of the model. The following holds $|\Theta| = \frac{I}{R}$, where Θ is the set of timesteps.

Whenever the algorithm first encounters a vertex v in the original graph (pre-visit) it will create time copies v_i of v, where $i \in \Theta$ indicates the i-th timestep. Vertices v_i and w_j are linked when the edge connecting them (and thus $(v, w) \in E$) is visited and $i + \tau(v, w) = j$, where $\tau(v, w)$ is the time required to travel from v to w. This edge visit always occurs after both vertices have already been visited. If due to a high resolution j does not exist, then i is connected to the smallest j such that $i + \tau(v, w) < j$.

Finally, it is possible to prune away edges and vertices that are not reachable from the sources or sinks in the time expanded graph by a 'PruneGraphOperation'. This operation is also implemented using BFS.

4.5 Summary

In this chapter we discussed the various algorithms implemented by the framework in more detail. We gave a complete description of the Push-Relabel and the Successive-approximation by Cost

Scaling algorithms and mentioned details specific to our implementation of these algorithms. This chapter thus provides an answer to research question 3:

Can these algorithms be implemented such that they support the framework?

We ended this chapter with a brief discussion of the implementation details of several well known algorithms. Chapter 6 will establish and discuss the performance of our implementation of the Push-Relabel and Successive-approximation by Cost Scaling algorithms.

Chapter 5

Implementation of the Framework

Research Question 9 read:

> *Can we construct such a framework?*

This question is in the first place answered by the framework itself, but not entirely. It implies the following questions: does the framework that was developed conform the design and does it meet the requirements? Functional requirements tend to be fairly concrete, at least when specified properly. It can be verified from the implementation that the framework meets certain functional requirements. Therefore, we will only state the few functional requirements that are not satisfied by our implementation (in Section 5.4).

In contrast to the functional requirements, quality requirements are usually subjective and can often, unless automated tools exist, only be specified and verified objectively with considerable effort[63]. For this reason, we discuss some implementation details of the framework with respect to the three selected quality requirements: changeability, reliability and understandability. The definitions of quality requirements given here, are reiterated from Section 3.2.2.

5.1 Changeability

Changeability is the effort required to modify or add new functionality to the framework.

Our framework makes use of the JGraphT[1] library version 0.7.3. This is an open source Java library that provides various graph theoretic data structures. Although the current framework makes use of JGraphT, it does not depend on JGraphT. An alternative data structure can be used by providing an alternative implementation of a graph factory[2].

Furthermore, it is possible to register new algorithms, operations and graph loaders with the framework by modifying a configuration file. Apart from the effort required to implement the concerned class itself no changes have to be made to the framework.

In order to ease interaction between the application and the framework a facade[3] is created. This creates a single point of entry and has the additional advantage that it is fairly easy to create a layer (for example a Java RMI or web server) in front of the framework.

[1] `http://jgrapht.sourceforge.net/`
[2] Abstract Factory Pattern [25]
[3] Facade Pattern [25]

According to the specification in the design the communication between various components occurs asynchronous. In essence every components has a queue from which it consumes events or tasks and another queue onto which it pushes the results it produced. This makes it in principle possible to create a proxy between components. Such a proxy would enable different parts of the framework to run on dedicated servers.

5.2 Reliability

Reliability is the ability of the framework to remain functional, even if exceptions or errors occur.

All tasks within the framework are executed within separate threads taken from a thread pool. If an exception occurs during execution then this thread will simply terminate and the exception will be reported to the external application.

The algorithm management component can be configured to use a policy that monitors running threads and decides when a task should be aborted. A simple implementation that aborts tasks based on a timeout is supplied with the framework. However, it is trivial to add more advanced policies.

Finally, unit tests have been written.

5.3 Understandability

Understandability is the effort required by users of the framework to recognize functionality of the framework and to understand the logical concepts behind it.

Understandability of the framework, its design and functionality is addressed in several ways. First and foremost this thesis provides an extensive design (Chapter 3), it discusses the implemented graph theoretic algorithms and gives few relevant implementation details. In addition the framework is delivered with JavaDoc[4] code comments and a manual. Finally, the prototype shows how to interface with the framework and gives code examples.

5.4 Limitations

The most profound missing functionality is the EntityManagement component as specified in the Logical View. This component is responsible for maintaining a registry of entities with their position and their direction. It aggregates updates of these properties in modifications to the graph structure and state. In order to make these updates, the component uses the same interface to the GraphManagement component, that external applications use. This component is therefore not essential to the functioning of the framework as a whole.

We previously mentioned, that the framework contains a basic implementation of a policy to monitor execution threads. Similarly, the framework contains only a basic functionality to regulate access to graphs. These basic implementations can be easily replaces with more sophisticated ones.

[4] JavaDoc: `http://java.sun.com/j2se/javadoc/`

5.5 Conclusion

In this chapter we discussed some relevant implementation details of the framework. This discussion was structured according to the thee selected quality requirements, because the design decisions behind these details are often based on these requirements. This chapter therefore provided concrete arguments how the framework satisfies the quality requirements. These arguments provide a degree of confidence that the framework is indeed changeable, reliable and understandable.

Part IV

Evaluation

Chapter 6

Empirical Study

During our project we made extensive use of our Successive-approximation by Cost Scaling algorithm, because it is employed in our prototype to calculate a quickest flow. On the other hand the Push-Relabel algorithm is primarily used as a subroutine of the SA algorithm. Specifically it is used to calculate the first feasible flow. For this reason we spend more effort in developing and testing the SA algorithm. In this chapter we report on the performance of both algorithms. Our discussion of the Push-Relabel algorithm, however, is more brief and serves primarily as a baseline, against which future improvements can be compared. The chapter provides an answer to Research Question 6:

> *What is the measured performance of the maximum and minimum-cost flow algorithms that are implemented?*

6.1 General setup

The experiments are executed on a Intel Pentium 4 machine running at 3Ghz and equipped with 2 GB RAM. The system runs on Ubuntu Hardy Heron (8.04.1) Server Edition. Our code is compiled using Sun Java SE 1.6.0_04. In order to facilitate the execution of the experiments an 'experimentation' application has been constructed on top of the framework.

6.2 Push-Relabel

This sections reports and discusses the performance of our implementation of the Push-Relabel algorithm based on an empirical study.

6.2.1 Experimental Setup

Our implementation of the Push-relabel algorithm is tested on three problem families. For each of these families problem instances are generated using two generators available from the dimacs ftp-site[1]. The different families are described below.

[1] `ftp://dimacs.rutgers.edu/pub/netflow`

Generators

The Washington generator generates moderately dense networks and has four input parameters. The first specifies the graph type, which is set to 6 to generate 'Basic Line' graphs. The next three parameters specify the number of rows, columns and degree. In order to generate graphs with a problem size of 2^x these parameters are set to values 2^{x-2}, 4 and $\frac{\sqrt{n}}{4}$ respectively. With a probability of 0.5 the generated graphs have their minimum-cut near the sink and with a probability of 0.5 it is located near the source. Experiments are executed for problem sizes of 2^6, 2^8, 2^{10} and 2^{12} vertices. The family of graphs generated with this generator are labeled with 'wlm'.

The GenRMF generator is used to generate two families. The first family has a wide structure and is denoted with 'rmfw'. Graphs in the other family have a long structure, these are marked with 'rmfl'. The generator accepts five parameters: the output file, then the frame size (a) and number of frames (b), and the minimum and maximum capacity. Each frame has $a * a$ vertices and each vertex is connected to its neighbors. From each frame all vertices are connected to a randomly selected counterpart in the next frame.

For both GenRMF families the capacity parameters are set to 1 and 10^4 respectively. Given problem size of 2^x, then for the rmfl-family let $y = x/4$ and $a = 2^y$ and $b = 2^{2y}$. For the rmfw-family let $y = x/5$ and $a = 2^{2y}$ and $b = 2^y$.

Table 6.1 shows the problem sizes for the various families. For the wlm family approximate values are given.

Table 6.1: Problem Sizes for Maximum Flow Problems

(b) Problem sizes for rmfw and rmfl families

Family	Size	Vertices	Edges
rmfw	6	50	185
	8	243	1026
	10	1024	4608
	12	3645	16956
	14	13824	65664
rmfl	6	32	92
	8	256	1008
	10	800	3335
	12	4096	18368
	14	15488	71687

(b) Problem sizes for the wlm family

Family	Size	Vertices	Edges
wlm	6	66	128
	8	258	1000
	10	1026	8070
	12	4098	65020
	14	16386	33776

6.2.2 Results

Table 6.2 gives execution time and operation counts for both rmf families. The results for the wlm family are splitted between instances with their minimum-cut near the source (labeled wlm.s) and those with their minimum-cut near the sink (label wlm.t). They are shown in Table 6.3. The first row in this table indicates the number of instances in that category. Figure 6.1 visualizes the data in a chart.

Discussion

The data clearly shows that the Push-Relabel algorithm is sensitive to the location of the minimum-cut. The algorithm is significantly faster on graphs with their cut near the source. On the other hand there is no significant difference in execution time between graphs with a wide and long structure. Note that the size of problem instances for the rmfl family as shown in Table 6.2b is slightly smaller then those for the rmfw family.

Table 6.2: Execution times for rmf families

(a) Results for the rmfw family

Vertices	Edges	Execution Time (ms)	Relabels	Saturating Pushes	Non-Saturating Pushes
50	185	15.3	1052.7	1139.0	316.4
243	1026	367.4	25244.5	29070.6	4030.5
1024	4608	7539.4	458576.5	528267.1	47145.5
3645	16956	113348.1	5596195.2	6425340.4	470933.8
13824	65664	2145414.8	83401183.8	95257331.8	6122841.1

(b) Results for the rmfl family

Vertices	Edges	Execution Time (ms)	Relabels	Saturating Pushes	Non-Saturating Pushes
32	92	7.3	409.3	426.0	111.3
256	1008	449.5	31632.5	35329.1	4363.4
800	3335	4153.0	261379.7	292793.1	27971.4
4096	18368	167440.8	7672229.2	8545208.5	620069.8
15488	71687	2516919.0	96543242.2	107060280.7	7232016.9

6.3 Successive Approximation

This section discusses the empirical study of our implementation of Successive Approximation by Cost Scaling algorithm.

6.3.1 Experimental Setup

The implementation of the successive approximation by cost-scaling algorithm is evaluated using problems generated by two generators. The grid-on-torus, or goto, generator is used to generate three problem families, while another three families are generated with the gridgraph generator. The source code for both generators is available through the dimacs ftp-site[2]. The generators and their input parameters are discussed in more detail below. Forty instances are generated for each family. Data is recorded on the execution time, number of saturating pushes, non-saturating pushes, relabels, discharges and refines. Correctness is ensured by comparing the solution calculated with our implementation with the one calculated by the CS2 algorithm by Goldberg[3].

[2] `ftp://dimacs.rutgers.edu/pub/netflow`

[3] freely available for academic purposes at `http://www.avglab.com/andrew/soft.html`

Generators

Three problem families are generated using the grid-on-torus generator. For all three families the maximum capacity and maximum cost are kept constant at 16384 and 4096 respectively. The variable property is the edge density and given a problem size of of 2^x vertices the density is set to $8 \times x$, $16 \times x$ and $\lceil x^{3/2} \rceil$. The resulting families are called 'goto8', 'goto16' and 'gotoi' respectively. Experiments are done for problems sizes 2^6, 2^8, 2^{10} and 2^{12}.

The gridgraph generator is used to create three families namely Grid-Wide(gridw), Grid-Square (gridsq) and Grid-Long (gridl). As expected the graphs in these families have a wide, square and long structure. The generator takes five parameters. The grid height, width, the maximum capacity and maximum cost and finally a seed for the random number generator. For all three families the maximum capacity is set to 10^4 and maximum cost to 10^4. Given problem size 2^x. The wide family is generated by setting the width to 2^{x-4} and height to 2^4. The square family is created with both width and height set to 2^{x-1}. Height is set to 2^{x-4} and width to 2^4 for the long family. For each family graphs are generated with 2^6, 2^8, 2^{10}, 2^{12} and 2^{14} vertices.

If a particular generator was not able to generate a graph with exactly the said amount of vertices, then parameters were chosen to approximate that value. An overview of graph sizes is given in Table 6.4.

Parameters

In our experiments the following parameters can be influenced. Consult Section 4.3 for a description.

- E-Scale factor: the factor by which ϵ is reduced in each refine. Goldberg [30] suggest values between 4 and 8.
- Implementation of the relabel operation {simple , complex}.
- Lookahead heuristic {enabled, disabled}.

Experiments are executed with the following parameter values:

Measured attributes

During execution of the algorithm the following attributes are recorded.

- Execution Time.
- Number of Refine operations.
- Number of Relabel operations.
- Number of Saturating Push operations.
- Number of Non-Saturating Push operations.

The 'execution time' does not include the time required to initialize the algorithm (calculation of a initial feasible flow is part of the initialization), nor does it include time to construct the result. Time is measured by calls to the System.currentTimeMillis() method.

Table 6.3: Execution times for wlm family

(a) Results for the wlm instances with cut close to source

#	Vertices	Edges	Exec. Time (ms)	Relabels	Sat. Pushes	Non-Sat. Pushes
24	66	128	7.5	606.0	633.3	165.8
15	258	1000	38.8	2232.7	2798.4	545.1
20	1026	8070	333.1	12771.4	14645.6	2113.8
19	4098	65020	2589.2	61266.8	66166.2	7162.8

(b) Results for the wlm instances with cut close to the sink

#	Vertices	Edges	Exec. Time (ms)	Relabels	Sat. Pushes	Non-Sat. Pushes
16	66	128	24.3	1878.8	2071.6	402.3
25	258	1000	771.5	54701.5	57374.8	10450.0
20	1026	8070	23744.8	981934.9	989373.0	183554.8
21	4098	65020	699415.4	16303348.8	16309025.5	2671669.2

Table 6.4: Problem Sizes for Minimum Cost Flow Problems

(a) Problem sizes for grid families

Family	**Size**	**Vertices**	**Edges**
gridl	6	66	140
	8	258	512
	10	1026	2000
	12	4098	7952
	14	16389	31760
gridsq	6	66	128
	8	258	512
	10	1026	2048
	12	4098	8192
gridw	6	66	116
	8	258	512
	10	1026	2096
	12	4098	8432
	14	16386	33776

(b) Problem size for goto families

Family	**Size**	**Vertices**	**Edges**
goto8	6	64	512
	8	256	2048
	10	1024	8192
	12	4096	32768
goto16	6	64	1024
	8	256	4096
	10	1024	16384
	12	4096	65536
gotoi	6	64	512
	8	256	4096
	10	1024	32768
	12	4096	262144

Table 6.5: Experiment parameter values for minimum-cost flow

Id	**E-scale factor**	**Relabel Operation**	**Lookahead Heuristic**
4.S	4	Simple	Disabled
4.C	4	Complex	Disabled
4.S.L	4	Simple	Enabled
8.S	8	Simple	Disabled
8.S.L	8	Simple	Enabled

6.3.2 Results

Table 6.6 gives the mean execution times for experiments executed on graphs in the goto families. This data is visualized in figures 6.2, 6.3 and 6.4. Table 6.7 gives the results for graphs that are part of the grid families. These tables are visualized in figures 6.5, 6.6 and 6.7. In each table row the lowest execution time is highlighted in bold, even though the margin is sometimes insignificant. Unless otherwise indicated, all data presented is the mean over 40 datapoints.

We can observe that on small and medium problem sizes of the grid families the lookahead heuristic offers a significant improvement in execution time. However, on larger grid problems its lead is less significant. Furthermore, the algorithm with heuristic enabled performs far worse on larger instances of the goto16 and gotoi families. On families with a higher number of edges, which is the case for the goto16 and the larger gotoi families, the algorithm with an ϵ-scale factor of 4, the simple relabel operation and with the lookahead heuristic disabled generally performs best.

6.3.3 Discussion

Analysis Lookahead Heuristic

Table 6.8 provides more detailed data for the gridl and gotoi experiment families.

This allows us to further analyze the performance of this heuristic. From this table we can clearly see that the heuristic significantly reduces the number of push operations. However, at the same time it increases the number of relabel operations and for larger problems it greatly increases the number of relabels. We can observe an increase of roughly 64% and 122% when 4.S is compared with 4.S.L on the gridl family with 4098 and 16386 vertices respectively. This increase in number of relabel operations is explained by the heuristic itself. When the algorithm considers to push flow from vertex v to w, it will check if w has any admissible outgoing arcs. If not, then it will relabel w (as shown in Section 4.3.1, which explains the heuristic).

A further explanation of the detrimental influence of the heuristic on the performance on graphs with a larger number of edges comes from our specific implementation of the heuristic. Because our implementation does not explicitly maintain the admissible graph, it potentially has to check all outgoing arcs of w in the residual graph to determine that w does indeed not have any outgoing admissible arc.

This in fact changes the asymptotic running time of the algorithms. Goldberg [27] proves the number of push operations is bound by $O(n^2m)$. When the algorithm selects vertex v to push flow to vertex w it potentially examines all outgoing arcs of w and this number is bound by $O(n)$ in a fully connected graph. As a result the algorithm has a bound of $O(n^3m \log nC)$.

Nevertheless the heuristic shows a significant reduction in push operations, with a decrease of 86% when comparing 4.S with 4.S.L on gotoi instances with 4096 vertices and 262144 edges. This results in a lower execution time on smaller graphs. There are multiple possibilities to improve the heuristic. First, explicitly maintain the admissible graph. Second, adapt the relabel strategy of the heuristic. Goldberg and Kharitanov [30] describe a generalization of the heuristic, which aims to reduce the number of relabels. They actually report a reduction in relabel operations for the lookahead heuristic.

Analysis Relabel Operation

Here we further study difference between the complex and simple relabel operations as described in Section 4.3.1. Table 6.9 gives more detailed information for the gridw and gotoi experiments.

As expected the difference in number of push and relabel operations is insignificant. However, the simple relabel operation performs better in general, especially on smaller problems. On the other hand it seems to lose its advantage on larger problems. Still, it is not possible to make firm conclusions with respect to the larger problem sizes; additional study into the performance of the complex relabel operation on these larger problems is required.

Analysis E-scale factor

Table 6.10 contains data useful to further analyze the change in behavior as a result of different ϵ-scale factors. This table shows the number of refines and gives the number of push and relabel operation per refine. Obviously, a higher scale factor results in fewer refines. Yet the amount of work (i.e. push and relabel operations) done per refine increases significantly. This indicates a trade-off between the work required to convert an ϵ-optimal flow into an $\frac{\epsilon}{\alpha}$-optimal pseudoflow and the work done to convert the $\frac{\epsilon}{\alpha}$-optimal pseudoflow[4] into an $\frac{\epsilon}{\alpha}$-optimal flow. As shown in our discussion of the SA algorithm in Section 4.3, the first step is done within the refine itself, while the later is achieved by repeatedly applying push and relabel operations. Based on the results of our experiments we can conclude that in general it is better to execute more refines and do fewer work when converting the pseudoflow to a flow. Only on larger dense problems, such as gotoi instances with 1024 and 4096 vertices, the difference becomes less clear. However, further study on larger problems is required to determine whether the balance actually tips in favor of a larger ϵ-scale.

6.4 Conclusion

In this chapter we reported on the performance of the Push-Relabel and Successive-approximation by Cost Scaling algorithms. For the Push-Relabel algorithm we provided a baseline performance against which future improvements can be compared. Based on the results for the wlm family of graphs we conclude that that the running time of the algorithm depends heavily on the location of the minimum-cut. This is clearly shown in Figure 6.1. There is no significant difference in performance on graphs with a long versus wide structure.

For the Successive-approximation by Cost Scaling algorithm we implemented two relabels methods and the lookahead heuristic. Results indicate that this heuristic greatly decreases running time on small to medium size graphs with a low edge density. This in contrast to its performance on graphs with higher edge density and larger graphs in general. On these graphs the algorithm performs far worse when this heuristic is enabled. We can conclude that this performance degradation is due to the relabel strategy of the heuristic and the fact that the admissible graph is not explicitly maintained in our implementation. We can therefore suggest improvements: maintain the admissible graph and adapt the relabel strategy of the heuristic.

For small and medium sized problems the algorithm with the complex relabel operation is outperformed by the one with the simple relabel operation. However, on larger problems the difference is less clear. This needs additional investigation.

Analysis of the effect of different ϵ-scale factors shows a trade-off between the work required to convert an ϵ-optimal flow into an $\frac{\epsilon}{\alpha}$-optimal pseudoflow and the work done to convert the $\frac{\epsilon}{\alpha}$-optimal pseudoflow into a $\frac{\epsilon}{\alpha}$-optimal flow. We conclude that for the problems we studied it is better to do more refines and do less work during each refine. However, on large graphs with a high edge

[4]Recall: in a pseudoflow vertices are allowed to have excess flow.

density the results are less clear and further study is required to assess if this conclusion is valid for problems larger then the ones we studied.

Table 6.6: Execution times for goto families

(a) Results for the goto8 family

Size		Execution Time (ms)				
Vertices	Edges	4.S	4.C	4.S.L	8.S	8.S.L
64	512	221.8	434.3	**161.2**	296.0	165.8
256	2048	2264.4	4735.7	**1792.9**	3131.1	1934.4
1024	8192	48812.8	85294.8	38277.1	51328.4	**31938.3**
4096	32768	492649.8	862093.4	397817.7	553008.9	**397236.0**

(b) Results for the goto16 family

Size		Execution Time (ms)				
Vertices	Edges	4.S	4.C	4.S.L	8.S	8.S.L
64	1024	**323.9**	633.4	353.7	393.3	382.2
256	4096	**3332.7**	6861.7	3549.1	4102.4	3666.8
1024	16384	58692.5	102190.4	59201.9	64477.8	**56211.2**
4096	65536	**862968.8**	938897.5	912028.0	960345.9	864057.7

(c) Results for the gotoi family

Size		Execution Time (ms)				
Vertices	Edges	4.S	4.C	4.S.L	8.S	8.S.L
64	512	219.6	410.3	**150.8**	268.1	164.1
256	4096	**3508.6**	7014.8	3612.5	4289.4	3828.3
1024	32768	**95935.0**	173121.9	166062.9	100471.6	157604.5
4096	262144	**3434949.1**	3741921.7	10405170.0[5]	3543468.4	10002877.2

[5]Means based on 15 problem instances.

Table 6.7: Execution times for grid families

(a) Results for the gridl family

Size		Execution Time (ms)				
Vertices	Edges	4.S	4.C	4.S.L	8.S	8.S.L
66	140	61.5	113.4	**33.2**	77.6	34.5
258	512	664.3	1342.8	**367.4**	793.4	409.6
1026	2000	8636.1	14450.1	**4417.5**	11257.8	5479.2
4098	7752	81590.1	156630.5	**59928.5**	112005.9	69547.8
16389	31760	896164.8	949192.5	**839003.3**	1321073.6	1066389.4

(b) Results for the gridsq family

Size		Execution Time (ms)				
Vertices	Edges	4.S	4.C	4.S.L	8.S	8.S.L
66	128	80.6	147.4	**42.4**	97.4	45.8
258	512	678.5	1355.5	**371.9**	786.0	391.1
1026	2048	7487.8	12335.2	**3715.6**	9309.8	4457.4
4098	8192	61007.6	114705.6	**36629.3**	76786.9	42900.8

(c) Results for the gridw family

Size		Execution Time (ms)				
Vertices	Edges	4.S	4.C	4.S.L	8.S	8.S.L
66	116	85.3	160.2	**45.0**	111.8	51.4
258	512	670.6	1396.8	**371.6**	806.7	410.3
1026	2096	6643.3	10687.6	**3335.4**	7989.5	4079.4
4098	8432	50216.8	95027.1	**37297.1**	64021.7	44377.3
16389	33776	649502.4	**605408.2**	675441.5	738416.8	802116.3

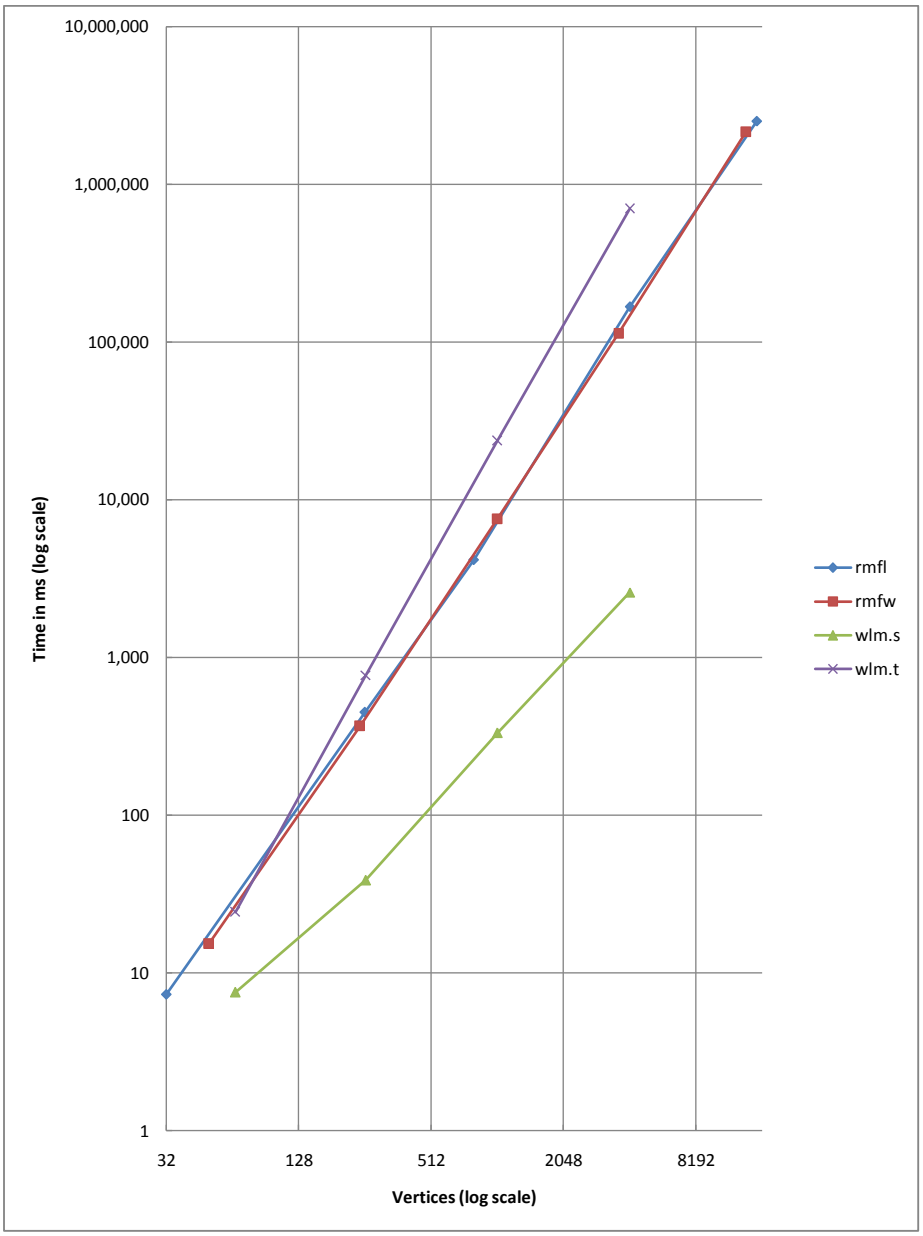

Figure 6.1: Mean execution times for the maximum flow families

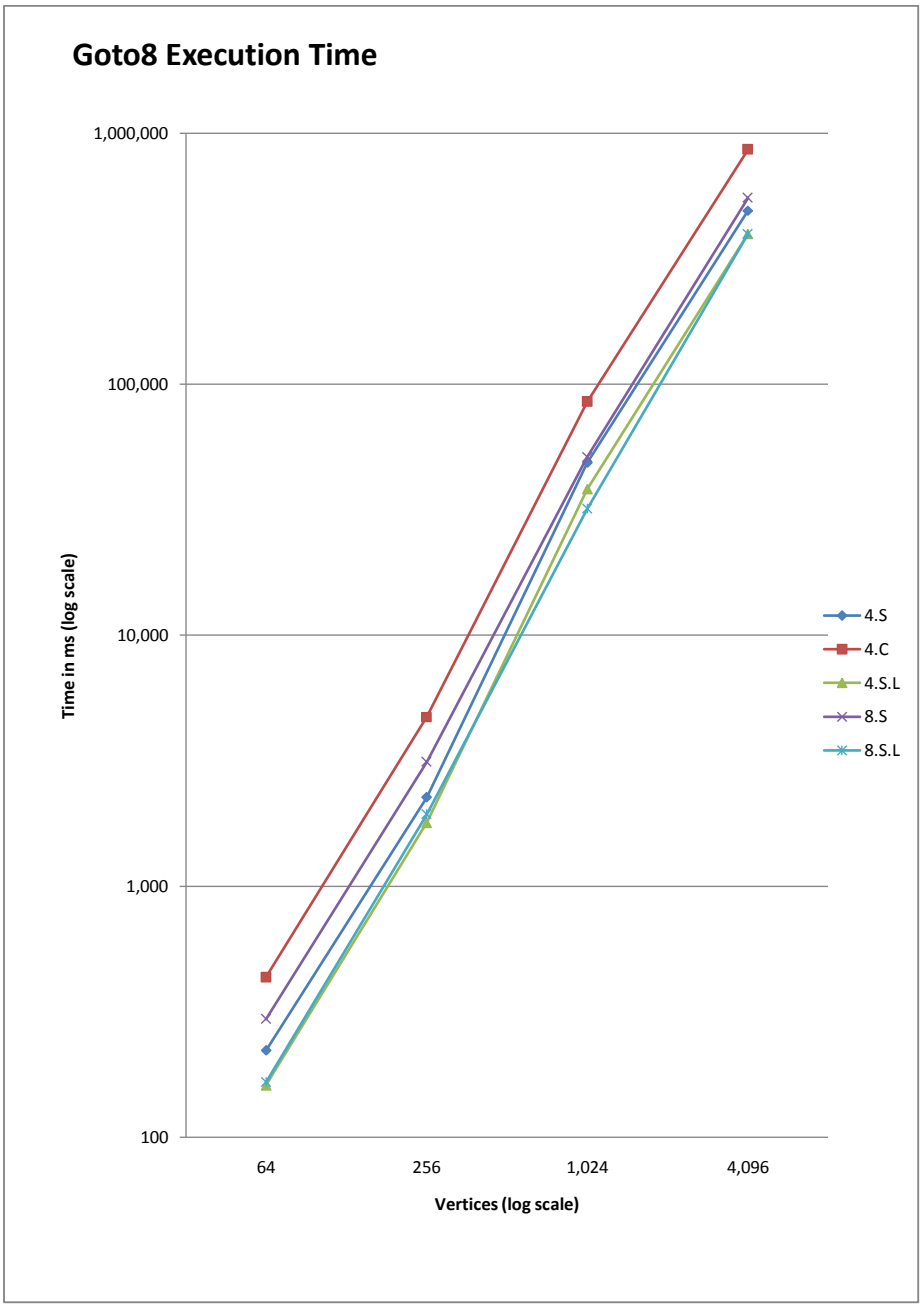

Figure 6.2: Mean execution times for the goto8 family

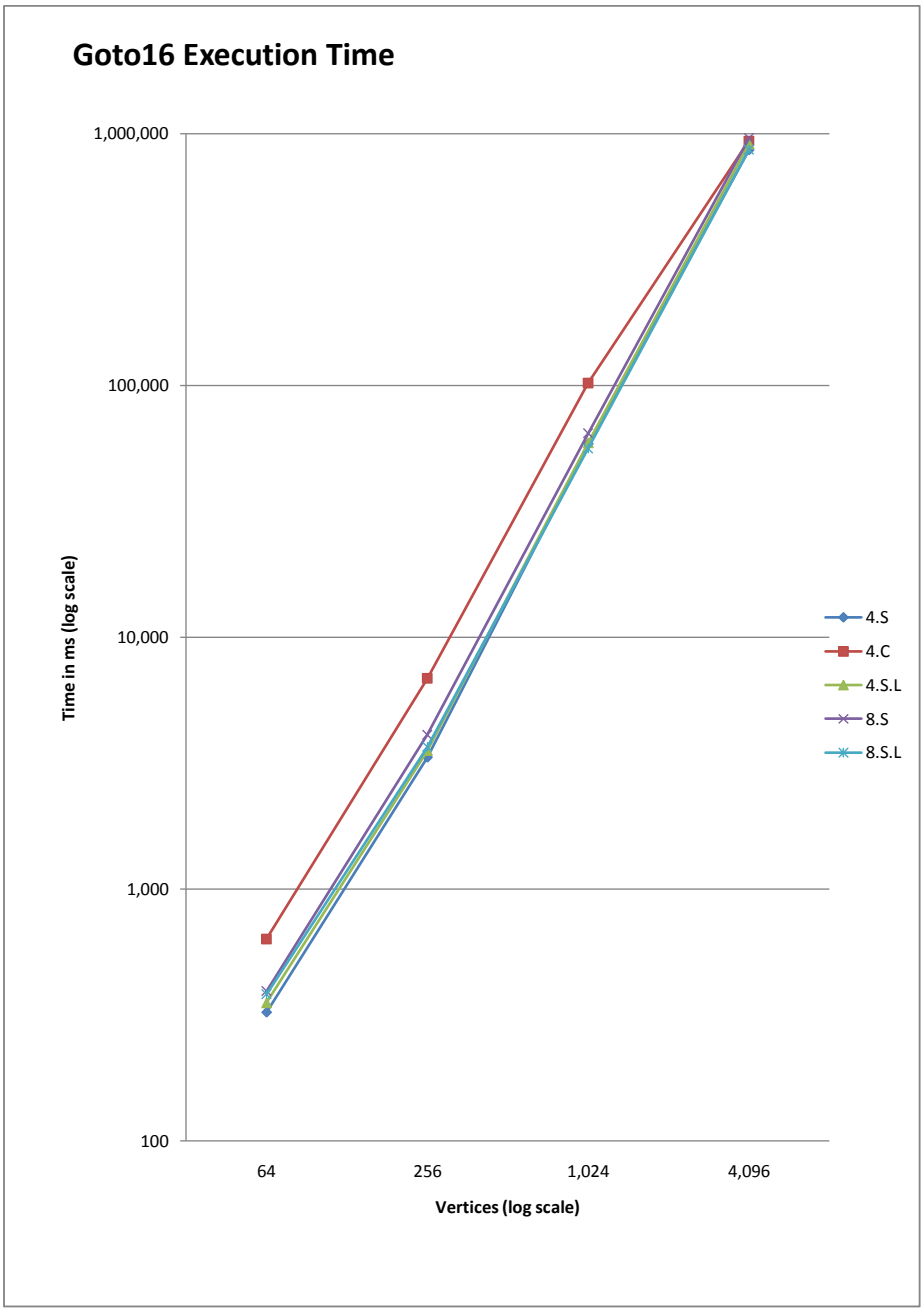

Figure 6.3: Mean execution times for the goto16 family

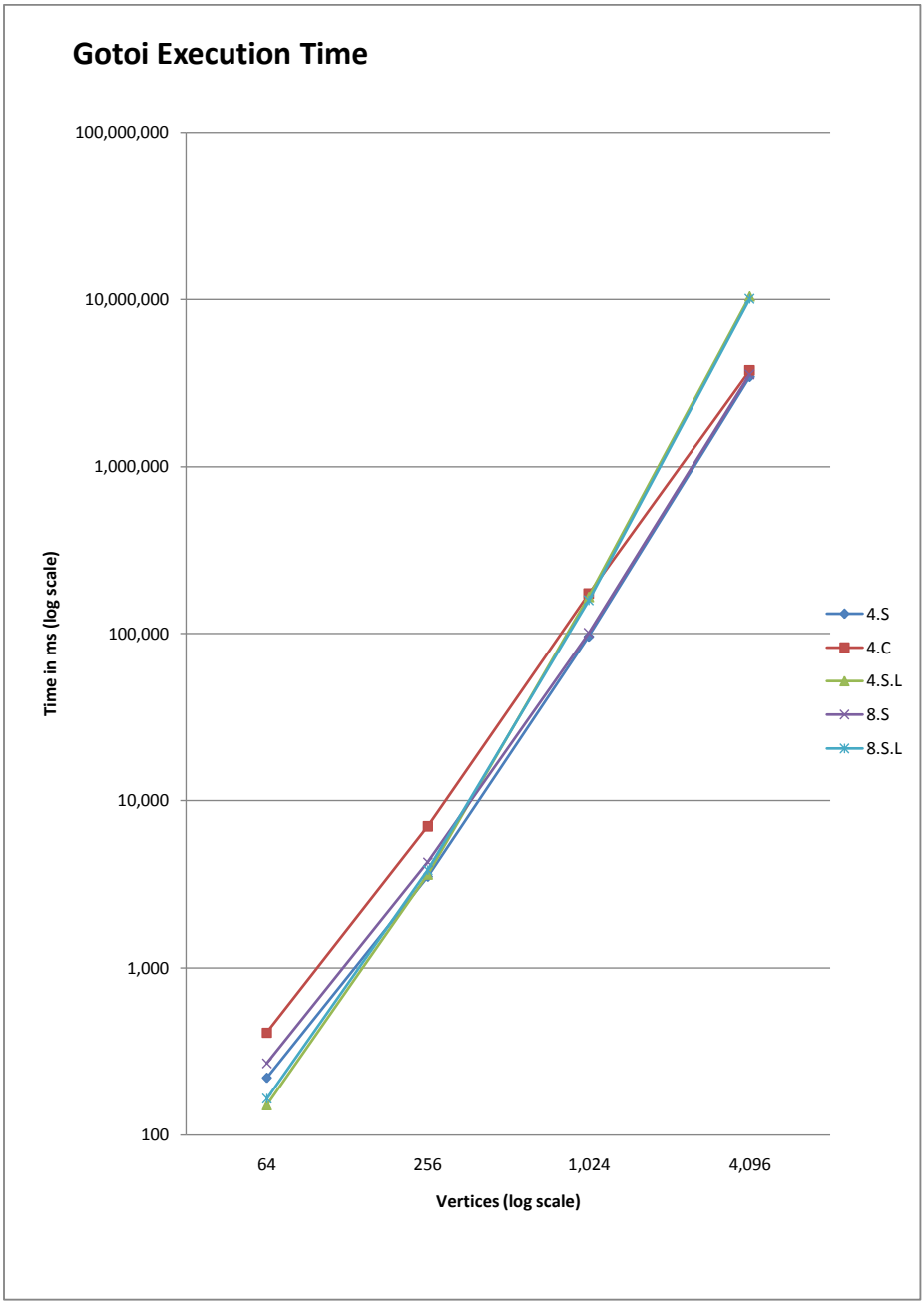

Figure 6.4: Mean execution times for the gotoi family

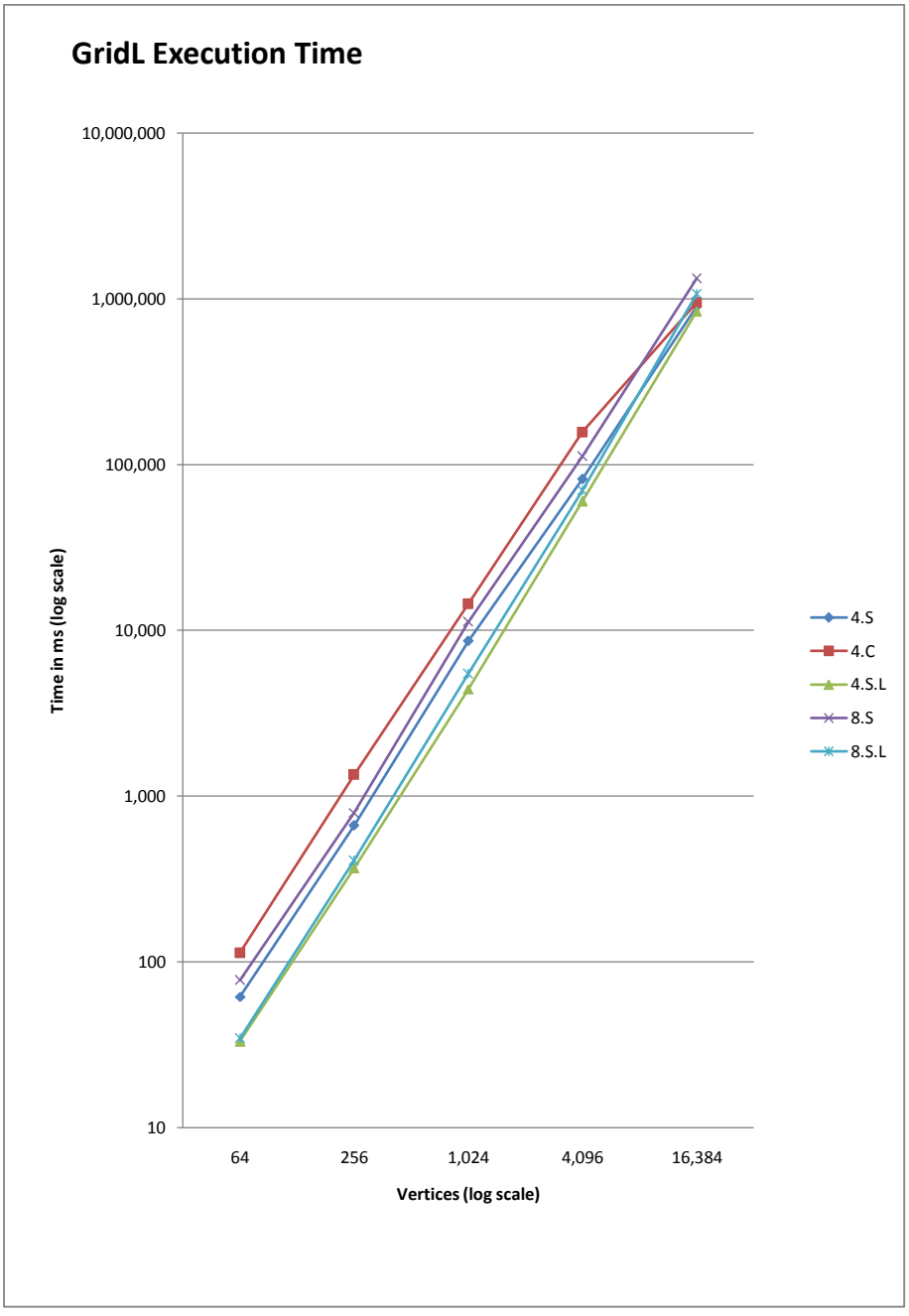

Figure 6.5: Mean execution times for the gridl family

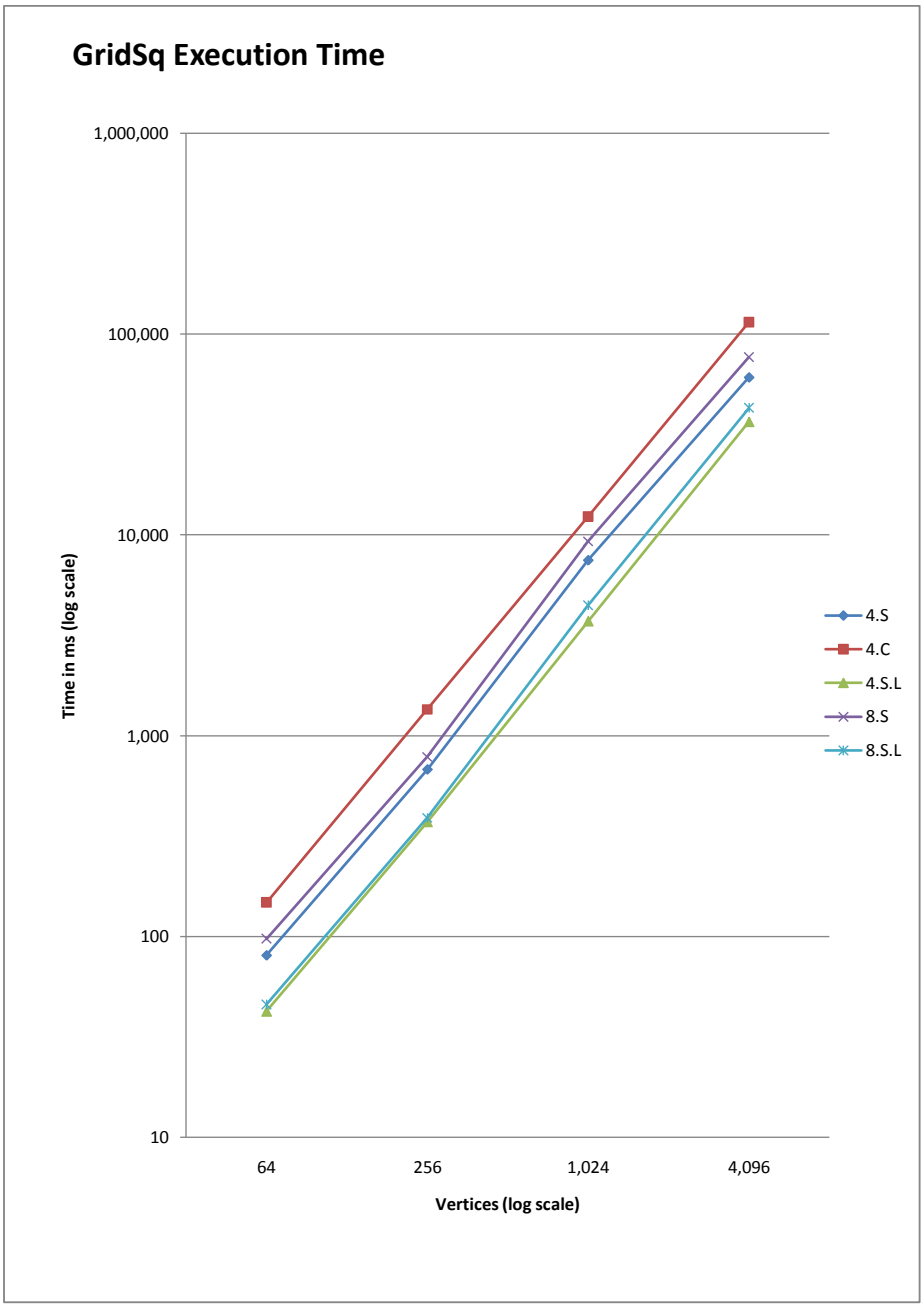

Figure 6.6: Mean execution times for the gridsq family

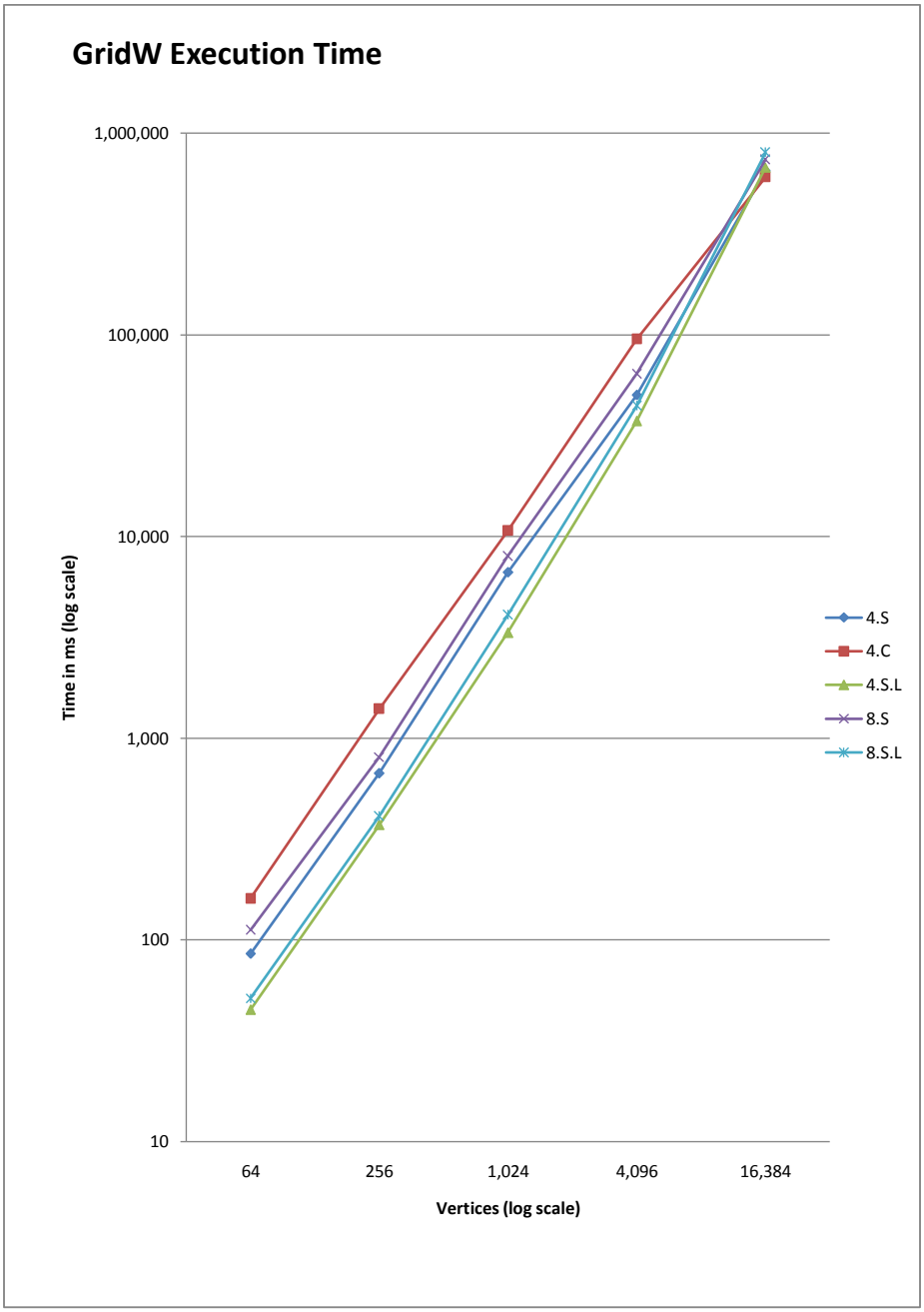

Figure 6.7: Mean execution times for the gridw family

Table 6.8: Analysis of the lookahead heuristic

(a) Data for the gridl family

		Execution Time(ms)		Relabels		Pushes	
Vertices	Edges	4.S	4.S.L	4.S	4.S.L	4.S	4.S.L
66	116	62	33	5293	5819	7545	2473
258	512	664	367	54800	65795	90953	33942
1026	2096	8636	4417	491407	659370	872751	373255
4098	8432	81590	59928	4447991	7307973	8212003	4325095
16386	33776	896165	839003	42727382	90551038	80787449	54693229

(b) Data for the gotoi family

		Execution Time		Relabels		Pushes	
Vertices	Edges	4.S	4.S.L	4.S	4.S.L	4.S	4.S.L
64	512	220	151	11932	14747	20089	6097
256	4096	3509	3612	111651	157940	210082	55536
1024	32768	95935	166063	1353868	2077323	2702428	528970
4096	262144	3434949	10405170^5	18366418	31993440^5	36771987	5014330^5

Table 6.9: Analysis of the relabel operation

(a) Data for the gridw family

		Execution Time (ms)		Relabels		Pushes	
Vertices	Edges	4.S	4.C	4.S	4.C	4.S	4.C
66	140	85	160	7651	7805	11762	12002
258	512	671	1397	55482	57194	92315	94989
1026	2000	6643	10688	371416	357437	652426	626007
4098	7952	50217	95027	2091707	2111111	3769226	3805762
16389	31760	649502	605408	11093651	11013480	20324615	20215037

(b) Data for the gotoi family

		Execution Time (ms)		Relabels		Pushes	
Vertices	Edges	4.S	4.C	4.S	4.C	4.S	4.C
64	512	220	410	11932	11732	20089	19677
256	4096	3509	7015	111651	109436	210082	205383
1024	32768	95935	173122	1353868	1352313	2702428	2704180
4096	262144	3434949	3741922	18366418	18582860	36771987	37125519

Table 6.10: Analysis of ϵ-scale factor

(a) Data for the gridl family

		Execution Time (ms)		Refines		Relabels / Refine		Pushes / Refine	
Vertices	Edges	4.S	8.S	4.S	8.S	4.S	8.S	4.S	8.S
66	140	62	78	10.0	7.0	529	970	754	1367
258	512	664	793	11.0	7.0	4982	9919	8268	15938
1026	2000	8636	11258	12.0	8.0	40951	78273	72729	139319
4098	7952	81590	112006	13.0	9.0	342153	666709	631693	1233078
16389	31760	896165	1321074	14.0	9.0	3051956	6952895	5770532	13226182

(b) Data for the gotoi family

		Execution Time (ms)		Refines		Relabels / Refine		Pushes / Refine	
Vertices	Edges	4.S	8.S	4.S	8.S	4.S	8.S	4.S	8.S
64	512	220	268	9.0	6.0	1326	2455	2232	3919
256	4096	3509	4289	10.0	7.0	11165	19964	21008	36559
1024	32768	95935	100472	11.0	8.0	123079	184007	245675	353978
4096	262144	3434949	3543468	12.0	8.0	1530535	2519552	3064332	4726935

Chapter 7

Prototype Applications

Fayad *et al.*[22] argues that it requires multiple iterations to develop a framework. With each iteration more experience into the framework and the problem domain is gained. Based on this experience the framework can be improved. For this reason it was one of our objectives to develop an application. The application that we created, simulates an evacuation from a building. However, apart from this evacuation application, two more applications have been developed during the course of the project. In early stages a simple console based application was constructed, and later an application to manage and schedule experiments was required for our empirical study. These applications can serve as an example of how the framework can be used within an application. This chapter describes these three applications and serves as a confirmation of Research Question 9:

> *Can we construct such a framework?*

7.1 Console Application

During the early phases of the development of the framework a small text based interface to the framework was constructed. As development of the framework progressed, new functionality was added to this 'frontend' application. This resulted in a simple application which gives a fairly comprehensive overview of the functionality offered by the framework. It supports functionality to load graphs, execute algorithms on them and modify the structure of the graph. Moreover, the application was instrumental in identifying problems and suggesting improvements to the framework. A screenshot is shown in Figure 7.1. The screenshot shows how three commands are executed in succession. First a graph is loaded from a file using the DimacsGraphLoader and it is assigned and id ('MCF'). Next the successive approximation algorithm is executed on this graph. Vertices 0 and 9 are assigned source and sink with a supply of 12 and -12 respectively. Once the framework completes computation it returns the solution, which is printed by the application. Finally, the resulting graph is shown. On each line a vertex and it's set of outgoing edges are shown. An edge is represented by the following example string $0 : 2[0- > (3/6)5- > 2]$. Here:

- $0 : 2[0 \rightarrow (3/6)5 \rightarrow 2]$ Shows the id assigned to the edge. In this case this id is composed of the head and tail vertices, but that is not required.
- $0 : 2[0 \rightarrow (3/6)5 \rightarrow 2]$ Are the tail and head vertices of the edge.

- $0 : 2[0 \rightarrow \textit{(3/6)}5 \rightarrow 2]$ Represents the flow value, capacity and the cost assigned to an edge.

```
FrontEnd [Java Application] C:\Program Files\Java\jre1.6.0_04\bin\javaw.exe (23 okt 2008 14:44:17)
Framework front end running.
execute loadgraph graphloader=framework.graph.persistence.DimacsGraphLoader filename=custom02.txt graphId=MCF
Graph loaded with id: MCF
execute algorithm graphId=MCF algorithm=framework.algorithm.SuccessiveApproximation sourceVertexIds=0:12 sinkVertexIds=9:-12
Minimum cost: 233
show graph graphId=MCF verbose=true
Directed Graph: MCF V:12 E:15
0:      0:2[0->(3/6)5->2] 0:4[0->(4/11)3->4] 0:7[0->(5/11)8->7]
1:      1:2[1->(0/9)7->2] 1:5[1->(0/3)3->5]
2:      2:7[2->(3/3)2->7]
3:      3:6[3->(4/8)6->6]
4:      4:7[4->(4/4)2->7]
5:      5:3[5->(0/9)1->3]
6:      6:9[6->(4/14)5->9]
7:      7:3[7->(4/13)9->3] 7:8[7->(8/12)5->8]
8:      8:9[8->(8/8)4->9] 8:10[8->(0/7)1->10]
9:
10:
11:     11:1[11->(0/5)3->1]
```

Figure 7.1: Console FrontEnd screenshot.

7.2 Experimentation Application

The experimentation application allows us to schedule the execution of experiments. These experiments are generated from two input files. The first specifies properties of the graphs, such as filename and the GraphLoader class that should be used to load the graph. The second file specifies configuration properties of the algorithms and the graph on which the algorithm should be executed. These experiments are then executed consecutively. Figure 7.2 shows a screenshot of the console output of the application while it is running on a dedicated machine. The application writes the results of each experiments to a file. Figure 7.2 shows the log output of the application. Observe the application repeatedly loads a graph, executes an experiment on the graph, and then unloads the graph. The figure shows that each time a single experiment is executed on the graph, but the application also contains functionality to execute multiple experiments on the same graph.

```
11:50:43,616 [Framework] INFO  experiments - Unloading graph 12.rmf1.6
11:50:43,616 [Framework] INFO  experiments - loading graph 12.rmf1.9 : ./experiments/maxflow/graphs/12.rmf1.9
11:50:45,717 [Framework] INFO  experiments - executing experiment 12.rmf1.9
11:52:05,528 [Framework] INFO  experiments - Unloading graph 12.rmf1.9
11:52:05,528 [Framework] INFO  experiments - loading graph 12.rmf1.8 : ./experiments/maxflow/graphs/12.rmf1.8
11:52:07,652 [Framework] INFO  experiments - executing experiment 12.rmf1.8
11:54:19,449 [Framework] INFO  experiments - Unloading graph 12.rmf1.8
11:54:19,449 [Framework] INFO  experiments - loading graph 12.wlm.28 : ./experiments/maxflow/graphs/12.wlm.28
11:54:22,553 [Framework] INFO  experiments - executing experiment 12.wlm.28
12:06:13,689 [Framework] INFO  experiments - Unloading graph 12.wlm.28
12:06:13,690 [Framework] INFO  experiments - loading graph 12.wlm.27 : ./experiments/maxflow/graphs/12.wlm.27
12:06:17,315 [Framework] INFO  experiments - executing experiment 12.wlm.27
```

Figure 7.2: Experimentation application screenshot

7.3 Evacuation Application

One of the objectives of this thesis was to develop a graphical prototype application on top of the framework. The choice was made to simulate an evacuation from a building. This evacuation application was further described in Section 1.5. That section also listed some requirements for the application. In this section we describe the actual application and discuss how it interacts with the framework. In Section 7.3.3 two screenshots of the application are shown and described.

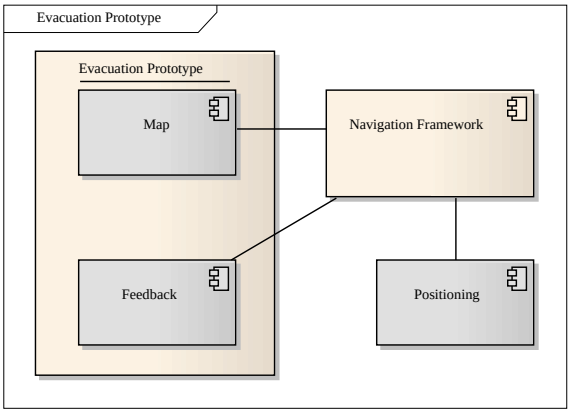

Figure 7.3: Relation of the evacuation prototype with the framework. The box 'Evacuation Prototype' shows the components that are implemented by the application.

7.3.1 Relation to Framework

The framework requires domain specific implementations of several components. Figure 7.3 shows how the evacuation application relates to the framework schematically. The evacuation prototype provides a custom GraphLoader that allows location information to be stored in a file along with the graph. The GUI presents feedback from the framework to the user. People within the building are simulated and no component explicitly implementing the positioning component is provided.

7.3.2 Functionality

Graphs can be loaded within the framework from files using the GraphLoaders registered with the framework. The graph is visualized on screen and it is possible to place context information in the background of the graph. It is possible to add or remove vertices and edges from the graph and edge properties can be edited. The modifications are made to the graph within the framework and the changes are displayed on screen. Vertices can be assigned as source or sink and their supply value can be edited. Based on these values a minimum-cost flow can be calculated by the framework. The resulting minimum-cost flow is shown on screen.

In the context of an evacuation problem the supply value of a source corresponds to the number of people present at that location. A sink indicates an exit of the building. The most efficient way to evacuate a building is obtained by calculating a quickest flow. In order to calculate this quickest

flow based on the graph that is shown, two additional parameters need to be set: namely interval and resolution. The interval denotes the time horizon for which a quickest flow is calculated and resolution determines the length of a timestep. The capacity property of edges is now interpreted as capacity per timestep. Cost denotes the number of timesteps required to travel across an edge from the tail vertex to the head vertex. Calculating a quickest flow based on this graph involves a few steps. First a time expanded graph is created, then this time expanded graph is pruned and finally a minimum-cost flow algorithm is executed on the time expanded graph. As explained in Section 2.5 the minimum-cost flow in the time-expanded graph is actually a quickest flow. Based on the solution to quickest flow problem the evacuation simulation is created. The application now shows a timeline on screen. At each step of the timeline the graph shown on screen is updated to reflect the status of the evacuation at that timestep. By automatically advancing along the timeline an animation of the evacuation is shown.

7.3.3 Evacuation Screenshots

Figures 7.4 and 7.5 shows screenshots of the evacuation application. The application in the screenshots show a map of the Buys Ballot Laboratorium (BBL) building of Utrecht University situated on the Uithof in the background. On top of this map a graph is drawn. This graph is thus a model of the building. However, we did not perform any research to ensure the accuracy of the model. The graph is visualized using jGraph[1], which is an opensource library for graph visualization.

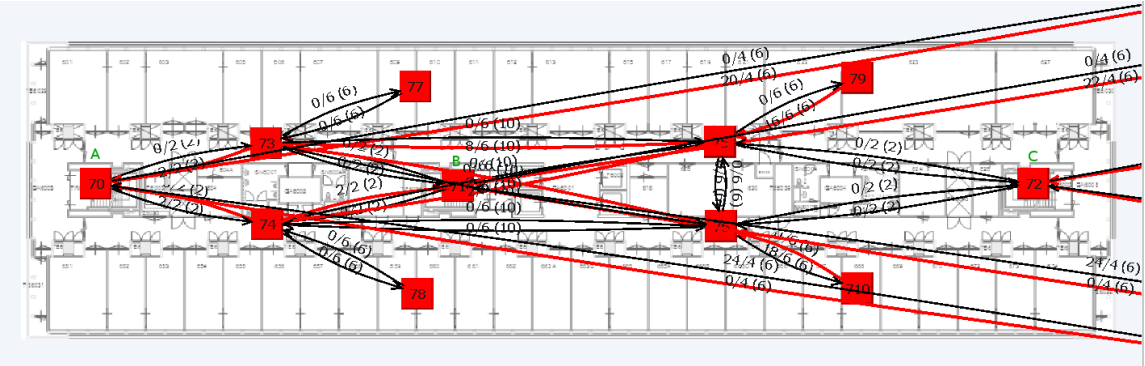

Figure 7.4: Evacuation application: detail

Figure 7.4 depicts the seventh floor of this building. We can observe 10 nodes in this closeup picture and several edges connecting them. To reduce model complexity, each node represents a certain area within the building. For example node '77' actually represents several nearby rooms. Three nodes are marked 'A', 'B' and 'C'. These nodes are located within the staircases and are connected with upper and lower floors. We can see these edges actually leaving the figure on the right hand side.

Each edges has a label, which displays properties of the edge. Each label consists of three values 'flow / capacity (cost)'. In this context, flow denotes the number of people currently traveling across the edge. The capacity is the maximum number of people that can travel across the edge in a time period. At last, cost gives the time it takes to travel across the edge from tail to head. In

[1] `http://www.jgraph.com/`

the figure several edges are colored red. This indicates that people are traveling across the edge in the current timestep.

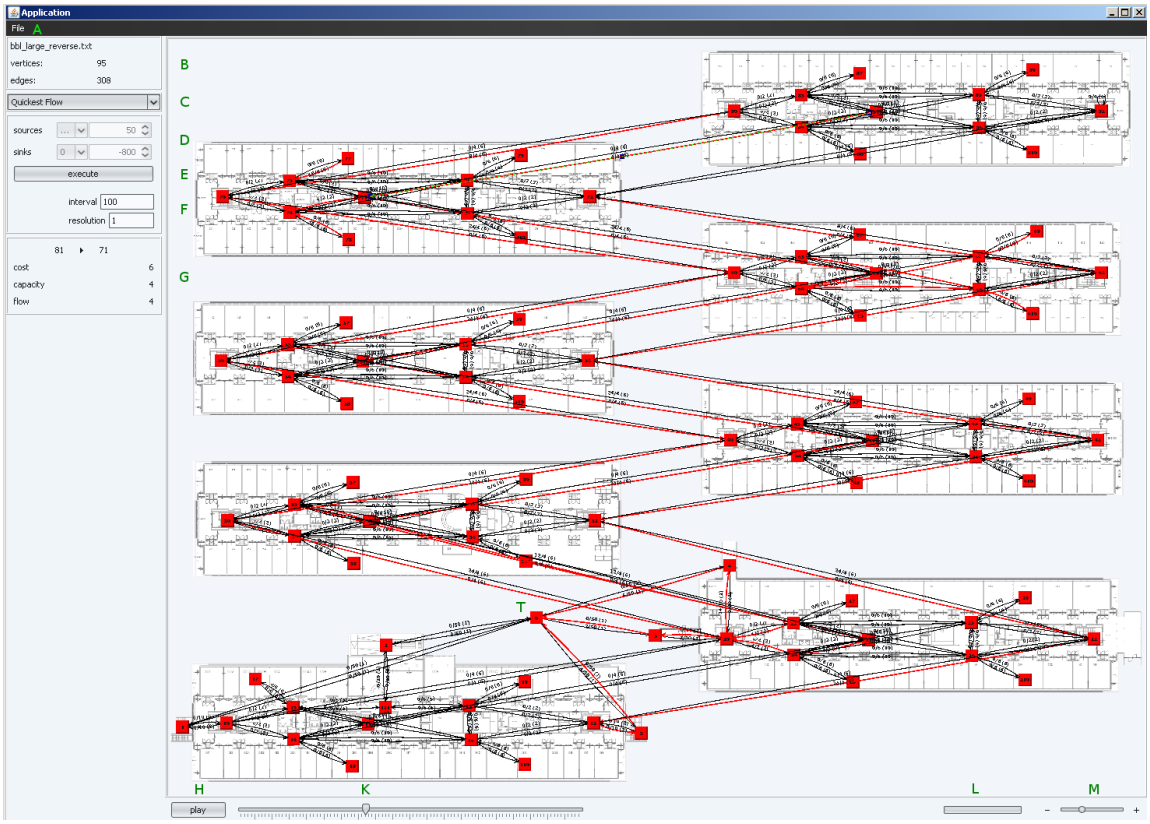

Figure 7.5: Evacuation application: overview

The next figure, Figure 7.5 gives a screenshot of the entire application. Here we can see 8 floors in the BBL building. The simulation that is shown on the screenshot has 100 timesteps (indicated by 'interval' Graphical User Interface (GUI) element - marked by a green F). Each timestep can, for example, be interpreted as a three second period. In this example we placed 200 people in rooms on floors five to eight. Observe how people are traveling across the staircases from the top floor towards the exits. The building has five exits, two on the first floor and three at ground level. On the first floor the BBL is connected to two adjacent buildings by bridge. For technical reasons each of these exits is connected to the 'supersink' (marked by a green T).

The screenshot also shows the GUI. Several elements are marked with green letters. A short description for each of these is given next.

- A: The file menu allows you to load and save graphs.
- B: Shows general information about the current graph. Its name, number of vertices and edges.

- C: Allows you to switch between algorithms.
- D: Box that lists sources and sinks within the graph.
- E: Executes the current algorithm.
- F: Allows you to modify the interval and resolution parameters.
- G: Shows information about the currently selected graph element. In this case the edge between node 86 and 84.
- H: Play or Pause the evacuation simulation.
- K: Timeline for the evacuation simulation. Allows you to rapidly scroll forwards and backwards in time.
- L: Indicates the framework is currently processing input.
- M: Allows you to zoom in and out.

Some functions are not shown. For example, it is possible to scroll the map view by using the arrow keys and when the right mouse button is pressed a context menu is shown. This menu allows you to edit the graph. We previously described how this simulation is calculated. Here we give little anecdotal information about this process. In the screenshot it can be observed that the shown graph consists of 95 vertices, which are connected by 308 edges. The time expanded graph that is constructed from this graph has about 9500 vertices and during the pruning step roughly 3800 vertices are removed. The resulting graph still has 5700 vertices and about 18500 edges. It takes our algorithm 12 seconds to find a quickest flow in this graph.

7.4 Discussion

The development effort of all three applications was dominated by the effort to collect input data, and, especially for the evacuation prototype, the effort to present the data that is received from the framework. Interaction with the framework proved to be straightforward. Development of these applications was invaluable in verifying that (parts of) the framework and algorithms worked as intended and suggesting fixes if it did not. The three applications are remarkably different from each other and each serves a different purpose.

The evacuation application is a visual application and can be used as demonstrator. From the description of this application in the previous section we can observe that it meets the requirements specified in Section 1.5. The application therefore satisfies the objective to construct a prototype application based on the framework. Additionally, the application shows that the abstract graph algorithms that are supported and implemented by the framework can be applied to a practical problem.

The console application intends to provide a comprehensive overview of the functionality offered by the framework. Because of this, it can be used, in conjunction with the evacuation application to verify if the framework satisfies functional requirements. Additionally, because this code does not contain GUI and event handling code, it may be better suited as a code example for developers that want to build their own application on top of the framework.

The experimentation application is quite different from the previous two and can show the flexibility of the framework. It aims to consecutively load a large number of graphs and execute algorithms on them. With this application we established baseline performance for the Push-Relabel and Successive-approximation by Cost Scaling algorithms. It thus allowed us to answer Research Question 6. Furthermore, if optimizations are made to the framework or the algorithms, then this application can be used to measure the performance gain. In this way, one can be confident that these optimization did result in a better performance.

We introduced this chapter with the observation of Fayad *et al.*[22], that it requires multiple iterations to construct a mature framework. They argue that the best way to learn if a framework is sufficiently flexible and general for its intended purpose is to develop applications on top of the framework. On the other hand they mention that it is dangerous to develop production applications based on a framework, while it is still evolving.

Because of this apparent contradiction: to construct a mature framework you need to build applications upon it, but to develop applications based on the framework you require a mature framework, we developed three prototype applications. The experience that was gained in the process provides us with degree of confidence that the framework can be applied in a complete application. In Section 5.4 we identified limitations of the framework; clearly these need to be addressed.

7.5 Conclusion

This chapter described three applications that were built on top of the framework. Each of these applications has a different purpose. The evacuation application can serve as visual demonstrator and shows that the graph algorithms supported by the framework can be applied in a practical application. Developers can use the console application as a code example. This application can also be used to verify the framework satisfies functional requirements. The third application enables the analysis of the algorithms implemented by the framework. These three applications provide arguments that the framework is flexible and that it can be applied in a complete application.

Chapter 8

Conclusion

8.1 Research Questions and Objectives

In this section we present the conclusion of this thesis. This conclusion is structured according to the structure of the thesis. Observe how all research subquestions are answered by the chapters.

8.1.1 Analysis

Literature Study

In Chapter 2 an overview of relevant graph algorithmic topics was given. Dijkstra's algorithm [17] for the SP problem was discussed and several heuristics to speed up Dijkstra were mentioned.

In the following sections several algorithms for both the maximum flow and minimum-cost flow problem were discussed. This discussion focused on the empirical results, that have been published for these algorithms. Based on this discussion we reached the conclusion that the Push-Relabel [29] and Successive-approximation by Cost Scaling algorithms [26] are state-of-the-art algorithms for respectively, the maximum flow and minimum-cost flow problem. This conclusion answers Research Question 1: *What are the state-of-the-art algorithms and heuristics for the single-source shortest path, maximum flow and minimum-cost flow problems?* and Research Question 2: *Which of these algorithms are best suited to be implemented?*

After discussing the algorithms for the maximum flow and minimum-cost flow problems, a method for introducing a time component was treated. A dynamic graph contains such a time element. The chapter gave definitions for the maximum dynamic, earliest arrival and quickest flow problems. It was shown that these problems can be solved by executing an ordinary minimum-cost flow algorithm on a so called time expanded graph. This time expanded graph is a static graph that can be created from a dynamic graph. This answered Research Question 4: *How can we introduce the notion of time in these algorithms?*

Architectural Design

Chapter 3 described the design of the framework. This design was structured according to the IEEE 1471 [50] recommend practice. Section 3.2 listed functional and quality requirements for our

framework. Changeability, reliability and clarity were identified as key properties of the framework. The remainder of the chapter described a collection of views. The Logical View showed a decomposition into multiple modules. Functional requirements are mapped onto these modules. The Process View shows the framework from a runtime perspective. Both the Functional and Process View aim to ensure decoupling between different parts of the framework. This chapter thus answered both Research Question 7: *What are the requirements for such a framework?* and Research Question 8: *What design satisfies the requirements?*

8.1.2 Implementation

Implementation of Algorithms

The thesis continued to discuss the algorithms, that are implemented and supported by the framework in Chapter 4. This chapter gives a complete definition of the Push-Relabel [29] and Successive Approximation by Cost Scaling (SA) [26] algorithms. Moreover, it describes implementation specific aspects for these algorithms. Additionally, it mentions few heuristics for these algorithms and provides references to others. These heuristics can often decrease the practical runningtime of these algorithms. In addition to the Push-Relabel and SA algorithms several other algorithms are implemented. Relevant implementations details were mentioned for each of these. This chapter therefore provides an answer to Research Question 3: *Can these algorithms be implemented such that they support the framework?*.

Implementation of the Framework

In Chapter 5 attention shifted to the framework. Research Question 9: *Can we construct such a framework?* is in the first place answered by the framework itself. However, this chapter argues how the framework satisfies the requirements. It shows how certain design decisions were influenced by the quality requirements. Furthermore, it identifies areas where the framework can be changed and improved. It, for example, argues that the framework can be made distributed, because different components communicate through asynchronous queues.

8.1.3 Evaluation

Empirical Results

In Chapter 4 we reported on our implementation of the Push-Relabel and SA algorithms. Chapter 6 reported on the practical performance of these algorithms by an empirical study. It thus provides an answer to Research Question 6: *What is the measured performance of the maximum and minimum-cost flow algorithms that are implemented?*

First, the performance of the Push-Relabel algorithm was studied on problems generated with the Washington and GenRMF generators that are available from the dimacs ftp-site[1]. A baseline performance was established and we can conclude that the performance of the algorithm is sensitive to the location of the minimum-cut. The algorithm performed significantly better on graphs with their cut close to the source.

[1] `ftp://dimacs.rutgers.edu/pub/netflow`

Next, the performance of our implementation of the SA algorithm was measured. We executed experiments on graphs generated by two generators, namely: the gridgraph and goto generator. These are also available through the dimacs ftp-site. The behavior of the algorithm was evaluated with a different ϵ-scale factor, two implementations of the relabel operation and the lookahead heuristic.

The results showed that in general an ϵ-scale factor of 4 is preferable over one of 8, the 'simple relabel' operation usually outperforms the 'complex relabel' operations. However, for both these parameters additional study into their behavior on larger problems is required. Analysis of the lookahead heuristic showed a significant performance gain on small and medium sized problems. This in contrast with far worse behavior on large and dense graphs. Several approaches were proposed to improve the heuristic's behavior.

Prototype Applications

One of the objectives of the thesis project was to construct a visual prototype application. Chapter 7 describes this, and two other applications that have been build during the project. The first application visually simulates an evacuation from a building. It shows that the algorithms implemented by the framework and the framework itself can be used in a practical application. The console application provides a console interface to the framework. It aims to provide a comprehensive overview of the functionality offered by the framework. Therefore, it can be used to verify if the framework satisfies its functional requirements. The third application enabled the empirical study of the performance of the Push-Relabel and SA algorithm. This experiment application schedules the sequential execution of algorithms on graphs. These three applications provide arguments that the framework can be applied in a complete application.

8.1.4 Objectives

At the start of the project we defined several objectives. Table 8.1 shows each of them has been satisfied.

8.2 Future Work

The framework itself and the algorithms that are implemented can be improved upon in several ways. Often these improvements are already mentioned in relevant sections. In the following sections we propose and describe few of these improvements. Again, these improvements are categorized according to aspect: algorithmic and framework.

8.2.1 Algorithmic Improvements

The framework currently supports the previously mentioned algorithms, namely: Dijkstra, Push-Relabel and Successive Approximation by Cost Scaling. Effort can be directed into improving the performance of each of these. Suggestions to this end are made later in this section. Furthermore, it would be interesting to investigate to what extent previous solutions can be used as input for later computations, after for example, the graph structure has been changed.

- Dijkstra's Algorithm. A lot of research has been done on improving the performance of Dijkstra's algorithm. In Chapter 2 we shortly described some heuristics for Dijkstra. These

Table 8.1: Objectives

Objective	Satisfied by
Determine and describe the state-of-the-art algorithms for the maximum flow and minimum cost flow problems.	Chapter 2
Derive and describe requirements for a framework as described in the problem description and produce a design based on these requirements.	Chapter 3
Report on the algorithmic implementation details and how they address research questions stated in the previous section.	Chapter 4
Perform a first iteration in the development of the framework.	The framework and Chapter 5
Report on the performance of the implemented algorithms by empirical study.	Chapter 6
Construct a prototype application.	The evacuation prototype and Chapter 7

heuristics can lead to a substantial gain in performance. Chapter 2 and the references therein provide a good starting point. Our implementation of Dijkstra uses Fibonacci heaps. Alternatively, a faster, but weakly polynomial implementation can be used. For example, the one described by Ahuja *et al.*[2].

- Push-Relabel. Research[6, 49] has shown that the global relabel heuristic can improve the performance of the Push-Relabel algorithm significantly. Alternatively, effort could be made to develop a parallel implementation of the algorithm. Articles by Goldberg and Tarjan [29] and Anderson and Setubal [5] reported on this.
- Successive Approximation by Cost Scaling. In our discussion of the Successive-approximation by Cost Scaling algorithm we cited several articles ([27, 30, 10]), that describe heuristics for this algorithm. Implementation of these heuristics can lead to additional performance gains. We implemented the lookahead heuristic. In Chapter 6 we analyzed the behavior of the heuristic and suggested several improvements. Again, this algorithm can be parallelized [26].

8.2.2 Framework

In Chapter 5 we mentioned several areas, where the framework can be improved upon.

- More sophisticated functionality for monitoring and terminating execution threads. The current implementation of the framework provides only a simple policy, that terminates threads after a fixed delay.
- More sophisticated functionality for regulating access to graphs.

- The framework currently lacks an implementation of the Entity Management module. This module was described in Section 3.4.2 of the design.
- Chapter 5 provides arguments where and how the framework can be distributed.

8.3 Conclusion

Section 8.1 highlighted the answers to each of the subquestions and Section 8.2 proposed future work. During this thesis project a navigation framework was designed and constructed. This framework aims to provide advanced navigation functionality by solving various flow problem. In order to solve these problems, it supports implementations of Dijkstra's algorithm, the Push-Relabel and Successive-approximation by Cost Scaling algorithms. These algorithms solve the shortest path, maximum flow and minimum-cost flow problems respectively. The framework also contains functionality to create a time-expanded graph. Solutions to dynamic problems, such as the quickest flow and earliest arrival problems can be found by executing the Successive-approximation by Cost Scaling algorithm on this time-expanded graph. Furthermore, the performance of the Push-Relabel and Successive-approximation by Cost Scaling algorithms was studied empirically.

Based on the framework and these algorithms an application was constructed that simulates an evacuation from a building. Such an evacuation can be modeled as quickest flow problem. The application uses the framework to construct a model of the building and to calculate a solution to this quickest flow problem. The simulation is created based on the solution of this problem.

Based on this, and the conclusions we described in the previous sections, we can conclude that we have answered the main Research Question:

> *Construct a navigational framework incorporating dynamic aspects as previously explained. How can existing theoretical graph algorithms be applied to support the required functionality provided by the framework?*

.

Part V

Appendices

Appendix A

Acronyms

ACS Applied Computing Science

API Application Programming Interface

AOP Aspect Oriented Programming

BBL Buys Ballot Laboratorium

BFS Breadth First Search

DFS Depth First Search

EUT Energy, Utility and Telecom

FIFO First In First Out

GIS Geographic Information Systems

GML Geography Markup Language

GPS Global Positioning System

GUI Graphical User Interface

Ids Identifiers

IDE Integrated Development Environment

IEEE Institute of Electrical and Electronics Engineers

ISO International Organization for Standardization

LBS Location Based Services

LCD Liquid Crystal Display

LIFO Last In First Out

MMS Multimedia Messaging Service

M.Sc. Master of Science

RD Rijksdriehoekscordinaten

req Requirement

RF Radio Frequency

RMI Remote Method Invocation

SA Successive Approximation by Cost Scaling

SMS Short Message Service

SP shortest path

UU Utrecht University

WLAN Wireless Local Area Network

WGS84 World Geodetic System 84

Bibliography

[1] R.K. Ahuja, A.V. Goldberg, J.B. Orlin, and R.E. Tarjan. Finding minimum-cost flows by double scaling. *Mathematical Programming*, 53:243–266, 1992.

[2] R.K. Ahuja, K.Mehlhorn, J.B.Orlin, and R.E. Tarjan. Faster algorithms for the shortest path problem. *J. ACM*, 37(2):213–223, 1990.

[3] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[4] R.K. Ahuja and J.B. Orlin. A fast and simple algorithm for the maximum flow problem. *Operations Research*, 37:748–759, 1989.

[5] R.J. Anderson and J.C. Setubal. On the parallel implementation of Goldberg's maximum flow algorithm. In *SPAA '92: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 168–177, New York, NY, USA, 1992. ACM.

[6] R.J. Anderson and J.C. Setubal. Goldberg's algorithm for maximum flow in perspective: a computational study. In *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 1–18, Boston, MA, USA, 1993. American Mathematical Society.

[7] R. Barahona and E. Tardos. Note on Weintraub's minimum-cost circulation algorithm. *SIAM J. Comput.*, 18(3):579–583, 1989.

[8] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1956.

[9] D.P. Bertsekas and P. Tseng. Relaxation methods for minimum cost ordinary and generalized network flow problems. *Oper. Res.*, 36(1):93–114, 1988.

[10] R.G. Bland, J. Cheriyan, D.L. Jensen, and L. Ladanyi. An empirical study of min cost flow algorithms. In *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 119–156, Boston, MA, USA, 1993. American Mathematical Society.

[11] J.V. Carroll, K. Van Dyke, J. Kraemer, and C. Rodgers. Vulnerability assessment of the U.S. transportation infrastructure relying on the global positioning system. *ION National Technical Meeting*, 2001.

[12] L.G. Chalmet, R.L. Francis, and P.B. Saunders. Network models for building evacuation. *Management Science*, 28(1):86–105, 1982.

[13] J. Cheriyan and S. N. Maheshwari. Analysis of preflow push algorithms for maximum network flow. *SIAM J. Comput.*, 18(6):1057–1086, 1989.

[14] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[15] G.B. Dantzig. *Application of the Simplex Method for Solving Assignment Problems - Motivation and Computational Experience*. Wiley, NY, USA, 1951.

[16] A. de Bruijne, J. van Buren, A. Kösters, and H. van der Marel. Geodetic reference frames in the Netherlands. Technical report, Netherlands Geodetic Commission, 2005.

[17] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[18] Y. Dinitz. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Mathematics-Doklady*, 11:1277–1280, 1970.

[19] J. Edmonds and R.M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.

[20] A. Endres and D. Rombach. *Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories*. Pearson Education Limited, England, 2003.

[21] ESRI. ESRI shapefile technical description. Technical report, ESRI, 1998.

[22] M. E. Fayad, D. C. Schmidt, and R. E. Johnson. *Building application frameworks: object-oriented foundations of framework design*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[23] M. L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[24] D.R. Fulkerson and G.B. Dantzig. *Naval Research Logistics Quarterly*, pages 277–283, 1955.

[25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[26] A. Goldberg and R. Tarjan. Solving minimum-cost flow problems by successive approximation. In *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 7–18, New York, NY, USA, 1987. ACM.

[27] A. V. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *J. Algorithms*, 22(1):1–29, 1997.

[28] A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *J. ACM*, 45(5):783–797, 1998.

[29] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 136–146, New York, NY, USA, 1986. ACM.

[30] A.V. Goldberg and Kharitonov M. On implementing scaling push-relabel for the minimum-cost flow problem. In *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 157–198, Boston, MA, USA, 1993. American Mathematical Society.

[31] A.V. Goldberg and R.E. Tarjan. Finding minimum-cost circulations by canceling negative cycles. *J. ACM*, 36(4):873–886, 1989.

[32] D. Grejner-Brzezinska. *Positioning and Tracking Approaches and Technologies*, chapter 12, pages 70–110. CRC Press, Boca Raton, FL, USA, 2004.

[33] M.D. Grigoriadis. An efficient implementation of the network simplex method. *Mathematical Programming Study*, pages 83–111, 1986.

[34] J. Hallberg, M. Nilsson, and K. Synnes. Positioning with bluetooth. In *10th International Conference on Telecommunications*, volume 2, pages 954–958, 2003.

[35] P.E. Hart, N.J Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics*, 4:100–107, 1968.

[36] ISO. ISO 9126:2000 software engineering - product quality - part 1: Quality model. Technical report, International Organization for Standardization, Geneva, Switzerland, 2001.

[37] J.J. Jarvis and H.D Ratliff. Some equivalent objectives for dynamic network flow problems. *Management Science*, 28(1):106–109, 1982.

[38] D.B. Johnson. A priority queue in which initialization and queue operations take O(loglog d) time. *Mathematical Systems Theory*, 15:295–309, 1982.

[39] L.R. Ford Jr. and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, USA, 1962.

[40] R.G. Karlsson and P.V. Poblete. An O(m loglog d) algorithm for shortest paths. *Discrete Applied Mathematics*, 6:91–93, 1983.

[41] J.L. Kennington and R.V.Ramakrishnan. *Algorithms for Network Programming*. Wiley, 1980.

[42] M. Klein. A primal method for minimal cost flows with applications to the assignment and transportation problems. *Management Science*, 14(3):205–220, nov 1967.

[43] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[44] B. Li, A.G. Dempster, C. Rizos, and J. Barnes. Hybrid method for localization using WLAN. In *Spatial Sciences Conference*, pages 341–350, 2005.

[45] X.R. Lopez. *Location-Based Services*, chapter 6, pages 171–188. CRC Press, Boca Raton, FL, USA, 2004.

[46] M. Luby and P. Ragde. A bidirectional shortest-path algorithm with good average-case behavior. *Algorithmica*, 4:551–567, 1987.

[47] Sun Microsystems. Java Remote Method Invocation - Distributed Computing for Java. Technical report.

[48] U.S. National Imagery and Mapping Agency. Department of Defense World Geodetic System 1984, its definition and relationships with local geodetic systems. Technical report, National Imagery and Mapping Agency, 1984.

[49] Q.C. Nguyen and V. Venkateswaran. Implementations of Goldberg-Tarjan maximum flow algorithm. In *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 19–42, Boston, MA, USA, 1993. American Mathematical Society.

[50] Institute of Electrical and Electronics Engineers. IEEE recommended practice for architecture description of software-intensive systems. Technical report, New York, NY , USA, 2000.

[51] Inc. Open Geospatial Consortium. OpenGIS Geography Markup language (GML) Encoding Standard. Technical report, 2007.

[52] J.B. Orlin. A faster strongly polynomial minimum cost flow algorithm. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 377–387, New York, NY, USA, 1988. ACM.

[53] J.B. Orlin. A faster strongly polynomial minimum cost flow algorithm. *Operations Research*, 41(2):338–350, mar 1993.

[54] S. Pandey, F. Anjum, B. Kim, and P. Agrawal. A low-cost robust localization scheme for wlan. In *WICON '06: Proceedings of the 2nd annual international workshop on Wireless internet*, page 17, New York, NY, USA, 2006. ACM.

[55] I. Pruijn. Location based messaging services: A generic architecture. Master's thesis, Enschede, The Netherlands, 2008.

[56] ABI research. Location-based services - operator strategies, revenue opportunities, subscribers, devices, and applica- tions for mobile lbs. 2006.

[57] C. Rizos. Trends in geopositioning for lbs, navigation and mapping. In *International Symposium and Exhibition on Geoinformation 2005*, 2005.

[58] D.D. Sleator and R.E. Tarjan. A data structure for dynamic trees. In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 114–122, New York, NY, USA, 1981. ACM.

[59] I. Sommerville. *Software Engineering: (Update) (8th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[60] S. Sparks, K. Benner, and C. Faris. Managing object oriented framework reuse. *IEEE Computer*, 29(9):52–61, 1996.

[61] B. van Zeist, P. Hendriks, R. Paulussen, and J. Trienekens. *Kwaliteit van softwareprodukten - Praktijkervaringen met een kwaliteitsmodel.* Kluwer Bedrijfswetenschappen, 1996.

[62] D. Wagner and T. Willhalm. Speed-up techniques for shortest-path computations. In *STACS 2007*, pages 23–36, 2007.

[63] R. H. J. Zeist and P. R. H. Hendriks. Specifying software quality with the extended iso model. *Software Quality Journal*, 5:273–284, 1996-12-01.