MASTER'S THESIS
Optimal Route Planning for Car Navigation Systems

by
M.G. van der Horst

Supervisors:
drs. I.C.M. Flinsenberg
dr. J.J. Lukkien
dr. J.H. Verriet

Eindhoven, October 2003
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>v</td>
</tr>
<tr>
<td><strong>Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1 Graph Partitioning for Route Planning</td>
<td>3</td>
</tr>
<tr>
<td>1.1 Basics</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Practical Aspects</td>
<td>6</td>
</tr>
<tr>
<td>1.3 Defaults</td>
<td>6</td>
</tr>
<tr>
<td>1.4 Algorithm</td>
<td>7</td>
</tr>
<tr>
<td>1.5 Terminology</td>
<td>7</td>
</tr>
<tr>
<td>2 Assignment</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Goals</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Computational Environment</td>
<td>9</td>
</tr>
<tr>
<td>2.3 The Profiling Tool</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Input Graphs</td>
<td>10</td>
</tr>
<tr>
<td>3 Partitioning Algorithm</td>
<td>13</td>
</tr>
<tr>
<td>3.1 Merging Algorithm</td>
<td>13</td>
</tr>
<tr>
<td>3.2 Analysis</td>
<td>17</td>
</tr>
<tr>
<td>3.2.1 General</td>
<td>17</td>
</tr>
<tr>
<td>3.2.2 Cellgraph</td>
<td>17</td>
</tr>
<tr>
<td>3.2.3 The Merging Procedure</td>
<td>18</td>
</tr>
<tr>
<td>3.2.4 The Selection Procedure</td>
<td>24</td>
</tr>
<tr>
<td>3.3 Implementation</td>
<td>25</td>
</tr>
<tr>
<td>3.4 Improvements</td>
<td>26</td>
</tr>
<tr>
<td>3.4.1 Random Number Generator</td>
<td>26</td>
</tr>
<tr>
<td>3.4.2 Improved Priority Function</td>
<td>27</td>
</tr>
</tbody>
</table>
D Interface
D.1 Linked Lists .................................................. 105
D.2 Double Linked Lists ......................................... 111
D.3 AVL Trees ..................................................... 113
D.4 Cellgraph ..................................................... 120
D.5 Flexfile ....................................................... 129

Bibliography ....................................................... 139
Preface

This assignment has taught me a lot, and not only about graph partitioning. By working on this assignment I gained valuable experience that I will be able to use for the rest of my career. I learned a lot of practical and useful things, like a new programming language, new data structures, etc. But, more importantly, I also learned new ways of looking at problems and solving them. My study gave me the necessary theoretical knowledge, but I find that the knowledge of how to apply that theory in practice can only be gained by practical experience. In short, I found this assignment to be a good conclusion of my study at the Technische Universiteit Eindhoven.

This research was performed in cooperation with Siemens VDO Automotive\(^1\), and I would like to thank my colleagues at Siemens VDO for the pleasant time I had working with them.

I would also like to thank my supervisors for their support:

Ingrid, you provided the basis for this research, without you this graduation assignment would never even have been possible. I wish you good luck with obtaining your Ph.D. title.

Johan, I always enjoyed our weekly discussions where we solved problems or proposed new ideas. I look forward to working with you in the future.

Jacques, your ideas and suggestions were invaluable and I admire your perseverance when checking the draft versions of this report. I wish you and Fabienne much happiness with your daughter Annelieke.

Martijn van der Horst,

---

\(^1\)This work was undertaken at Siemens VDO Automotive and may contain IPRs of Siemens VDO Automotive AG. © Siemens VDO Automotive A.G. All rights reserved.
Introduction

There is an enormous amount of literature on shortest path algorithms, as can be seen in the taxonomy of Deo and Pang [5]. There are many algorithms, but the most well-known algorithm is probably Dijkstra's shortest path algorithm [6]. Although algorithms like Dijkstra's shortest path algorithm always find an optimum route they tend to require a lot of memory and/or calculations. This makes them unsuited for use in an on-board navigation system for cars, which has neither the required memory nor the processing power for such an algorithm. This is especially true now that DVDs containing the entire road network of Europe are becoming available.

To cope with large road networks the route planners of car navigation systems often resort to approximation algorithms. These algorithms can trade finding the optimum route for smaller memory requirements and fewer calculations. This allows the system to quickly present a route to the user. The A* algorithm [13] is probably the most well-known example of an algorithm that can find an optimum route reasonably fast, but can also be used to find an approximation of the optimum route in even less time. However, even when giving up on finding the optimum route, the work required by the A* algorithm is still too much to make it practical for car navigation systems when reasonable routes are required. Besides, users do not know nor care about algorithms and the other internal workings of their navigation system. They just expect the system to quickly provide them with the best route to their destination. So, to satisfy the customer, the system should be able to calculate optimum routes quickly.

An algorithm that can quickly calculate optimum routes is presented by Ertl [7]. He first assigns a radius to each edge. His planning algorithm evaluates an edge only if the start or end point of the planning falls within the radius of that edge. This algorithm speeds up route planning, but preprocessing of the map is necessary. However, the data gained from this preprocessing step is not resistant to cost changes. So any change of the cost of a road (in case of a traffic jam for example) could cause the planning algorithm to lose the ability to plan optimal routes.

Flinsenberg [10] has developed a method to find an optimum route in reasonable time. This is also accomplished by preprocessing the road network. The graph is decomposed into several parts and some information is stored with each of those parts. The information consists of the cost of the optimum routes through that part of the graph. Navigation systems can then use that information during route planning to quickly pass through most parts of the road network. Only the parts where the car currently is and the part of the network in which the destination lies have to be taken into account.

This approach has the advantage that traffic jams and other cost changes can be taken into
account by discarding the optimum routes that may be affected by those changes. Also the preprocessing only requires the calculation of a partition and some routes. Furthermore, it improves the speed of the route planning, but does not reduce the quality of the planned routes.

The idea of using graph partitioning for route planning is not new: Jung and Pramanik [16] and Kim et al [18] also show how a decomposed graph can be used to speed up route planning. Route planning is not the only problem where graph decomposition can be used. VLSI-layout and parallel programming are only a few examples of other problems where large graphs have to be decomposed into smaller subgraphs. A lot of research on graph decomposition has been done. Berry and Goldberg [2], Falkner et al [8], Krishnan et al [20], Monien and Diekman [21] and Pothen [22] all study graph partitioning algorithms for different applications.

Although using graph decompositions for route planning is not new, Flinsenberg’s method for generating partitions is. Her approach differs in three respects from those of the researchers mentioned above. First of all she does not assume that the number of subgraphs in which the graph is to be divided is known beforehand. Secondly, the evaluation of her partitions is more complicated, because the goal is to minimize the expected number of edges that the system has to evaluate to find an optimum route. And finally, her partitioning algorithm will have to be applied to very large real-world road networks.

This report focuses on finding a fast implementation of the partitioning algorithm designed by Flinsenberg. The aim is to apply the preprocessing to a large real-world road network in reasonable time. A car navigation system needs an up-to-date map, but producing and testing a map already takes several weeks. Therefore it is preferred that partitioning does not add too much time to the production time of the map. As this report will show, we implemented the partitioning algorithm such that it can partition the road network of the Netherlands in only a few minutes, so this approach does not increase the map production time significantly.

A second objective is to improve the quality of the decompositions produced by the partitioning algorithm. It is possible to plan routes through any partition produced by the algorithm, but some partitions are better than others. What exactly constitutes a good partition and how to find it, is also discussed in this report.

The real goal is, of course, to use this approach to route planning in an actual car navigation system. Actually implementing it for such a system is beyond the scope of this research, but we are able to show that the approach is feasible. It is feasible because experiments show that planning a route through the partitioned graph of the Netherlands requires, on average, the examination of no more than 1,500 edges. This is even less than the current product route planner of Siemens VDO Automotive needs to plan non-optimum routes.

This report is structured as follows. Chapter 1 describes graph partitioning and how it can be used for route planning. The assignment and the available resources to complete it are described in Chapter 2. Chapter 3 contains an overview of the implementation of the partitioning algorithm. The two chapters after that deal with the quality of the partitions found by the algorithm. The first one, Chapter 4, deals with the improvement of the result of the partitioning algorithm, while the second one, Chapter 5, deals with the relationship between the result and the actual quality. An initial survey into some future research topics is made in Chapter 6. Chapter 7 contains the conclusion and an overview of possible future work.
Chapter 1

Graph Partitioning for Route Planning

Suppose that there is a city with only two roads leading in and out of it. If the optimum routes through that city are known, it is nonsense to recalculate them each time a route, which passes through the city, is planned. Storing those routes would eliminate the need to plan them anew each time.

The same idea can be applied to cities with multiple entrances and exits; it just requires that an optimum route between each entrance and exit pair is stored. So, when the planning algorithm enters the city, it can leave it again via a number of routes, each leading to a different exit.

This idea can be used to speed up route planning. Unfortunately, not every part of the map belongs to a city, so it cannot be applied to every part of the map. At least, not directly, but it is possible to divide the map into areas that resemble such cities; areas that can only be entered or left via a limited set of entries and exits. Such a division can be created by using graph partitioning.

Graph partitioning, or graph decomposition, is the division of a graph into subgraphs called cells. These cells do not necessarily have any relationship with the way the map is divided into cities, but they can be used in combination with the idea described above to speed up route planning. How this works in detail is described in the following sections.

1.1 Basics

This section presents the basic terminology and ideas behind graph partitioning. It also shows how graph partitioning can be used to plan optimal routes quickly.

As was mentioned before, the idea of graph partitioning is to divide a graph into subgraphs called cells. After that, the optimum routes through each cell are calculated and stored to simplify route planning later on. Cells thus have two graphs associated with them; the subgraph they represent (called the detailed map of the cell) and a graph representing the optimum routes through the cell (called the route graph of the cell), see also Figure 1.1.
As Figure 1.1 shows, there are a number of special nodes and edges in a partition. All nodes belong to a cell, but some of them are connected to an edge that leads to a different cell, such nodes are called boundary nodes. These boundary nodes mark the entry and exit points of a cell. Edges between nodes of different cells are called boundary edges and they form the connections between different cells. All other nodes and edges are called internal nodes and internal edges. The edges in the route graph are simply called route edges.

But how can this partition be used to plan routes? Standard route planning algorithms plan a route through a single graph, not through a partitioned graph. Still, it is possible to use standard route planning algorithms on a partitioned graph. This is done by hiding the fact that the graph has been partitioned and presenting a single graph to the route planning algorithm. This graph is called the search graph.

It is of course possible to construct the search graph by taking the boundary edges and detailed map of each cell, but that would mean that the search graph is the same as the original unpartitioned graph. So that would make the whole partitioning of the graph pointless. To construct a smaller search graph we introduce the concept of a boundary graph.

The graph formed by the boundary edges and the route graph of each cell is called the boundary graph (see Figure 1.2). This boundary graph can be used to construct the search graph. The search graph resembles the boundary graph, but the route graphs of some cells have been replaced by their detailed maps.

For simple route planning the search graph contains the detailed maps of only two cells: the detailed maps of the cells containing the start and endpoint of the route (see Figure 1.3).

The search graph can be used by a simple route planning algorithm to plan a route through the partitioned graph. Because the route edges contain optimum routes through each cell it is possible to plan optimum routes. Also it is quite easy to take traffic jams or other cost changes into account. The optimal routes stored in the route graph may no longer be valid after such a change, but if this happens the detailed map can be used instead of the route graph.

The main advantage, however, is that the search graph contains fewer edges than the original graph. And therefore the route planning algorithm is expected to require less time to plan an optimum route.
Figure 1.2: A boundary graph

Figure 1.3: A search graph based on the boundary graph in Figure 1.2
1.2 Practical Aspects

Although the theory behind graph partitioning is sound there is a small practical disadvantage. The drawback is that there is more information to store.

Note, however, that it is not really necessary to store the actual optimum routes through each cell (at least not for use in car navigation systems). Storing the cost of each route is enough; the route itself can be reconstructed later on, after the route through the search graph has been planned. So the planning of a route in a car navigation system comes down to planning a route through the search graph, after which it is possible to guide the driver along the route in the start cell. While the driver is guided along the first part of this route, the system has the time to reconstruct the optimum route through the cells that are on the route, but whose route graph was used instead of its detailed map.

But still, the cost of an optimum route for each criterion has to be stored (i.e. the cost of the fastest route, the shortest route, the cheapest route, etc.). So the graph partitioning approach still causes a considerable increase in the amount of information that needs to be stored.

1.3 Defaults

There are aspects of graph partitioning for speeding up route planning that seem logical, but are only default assumptions. We will see that some of them change. This section lists the assumptions we start out with.

We assume that the route graph of a cell is represented as a clique structure. It contains the nodes by which the cell can be entered or left (the boundary nodes) and edges between each pair of those boundary nodes to represent the routes. But as we will see in Section 4.1 there are other ways to construct a route graph. For now, however, we assume the route graph has a clique structure.

The goal of graph partitioning is to make the route planning algorithm faster. Some partitions speed up route planning more than others, so there has to be some measure of quality for a partition. High quality partitions speed up route planning more than low quality partitions. So ideally we would like to use the average processing time required to plan a route as a measure of quality for graph partitions. However, this time cannot be easily calculated and depends on a lot of external factors. The number of edges the planning algorithm examines to plan the route, also called the number of expansions, is a very good substitute.

The actual number of edges that are examined during the planning is hard to calculate beforehand, but an upper bound for this is the number of edges in the search graph, assuming that each edge is examined at most once. While the size of the search graph can be calculated for any two pairs of cells, it would be easier to have a single number to measure the quality of a partition.

Flinsenberg [9] has defined a formula that expresses the expected number of edges in the search graph. It assumes that every node in the graph has the same probability of being the starting point or destination of a route. The formula is discussed in more detail in Section 3.1. For now, we assume that the result of this formula corresponds to the quality of a partition and we use it as the selection function. When comparing two partitions, the one for which the
1.4. Algorithm

selection function returns the lowest result is considered the better of the two. In Chapter 5 we will see other selection functions that can be used to evaluate a partition.

1.4 Algorithm

The algorithm used to find a partition is discussed in more detail in Chapter 3. For now it is sufficient to know that it is a heuristic algorithm, since finding the optimal partition is strongly NP-hard [10].

The general partitioning algorithm is quite simple; for a given partition select a new partition from its neighborhood until a certain stop criterion is met. The priority function is the function that assigns priorities to every neighbor of a partition. The neighbor with the highest priority is chosen as the next partition. The priority function effectively steers the algorithm through the neighborhoods of the partitions. The selection function is used to evaluate each encountered partition and select the best one (according to the selection function). Hopefully the selected partition is also the partition with the highest quality in practice.

There are a few important and interesting relationships: the relationships between the selection function and the quality of the partition, between the priority function and the selection function, and between the priority function and the quality of the partition. Ideally there would be a direct connection between the partition quality and the selection function. In other words, ideally the selection function accurately represents the quality of a partition. But even if this is so, the priority function should also be such that it steers the algorithm toward partitions of good quality. If no good partitions are encountered, the algorithm will not find any good partitions, no matter how good its selection function is. This also holds the other way around: even if the priority function steers the algorithm toward the optimum partition, without a selection function that recognizes it, it will not be found by the algorithm. Therefore there is also an indirect relationship between priority function and partition quality.

1.5 Terminology

This section provides an alphabetical overview of the terms used throughout this document. Most of them have been introduced in the previous section, but some of them will be introduced later on in this report.

**Boundary Edge** An edge between nodes that belong to different cells.

**Boundary Graph** The graph formed by the route graph of each cell in the partition and the boundary edges.

**Boundary Node** A node via which a route can enter or leave a cell, i.e. a node that is connected to a node in a different cell via a boundary edge.

**Cell** A subgraph of the original graph.

**Clique** Structure for a route graph. In this structure each boundary node is connected to every other boundary node by a route edge.
Detailed Map  The internal nodes, edges and rules associated with a single cell.

Expansion  The examination of an edge during the planning of a route.

Objective Value  The expected number of edges in a search graph of a partition. The objective value is used as the default selection function.

Partition  A graph divided into cells.

Priority Function  Function that steers the partitioning algorithm.

Quality of a Partition  The quality of a partition says something about the time it takes to plan a route through it. During this research the average number of expansions needed to plan a route is used as a measurement of quality.

Route Edge  An edge in a route graph that represents (a part of) an optimal route through a cell.

Route Graph  The graph used to represent all optimum routes through a single cell.

Rule  see Turn Restriction.

Search Graph  The boundary graph, but with one or two route graphs replaced by their detailed maps. The search graph is used to plan routes.

Selection Function  Function that is used to evaluate partitions.

Star  Structure for a route graph. In this structure each boundary node is connected to a virtual center node and turn restrictions are used to store the route costs.

Turn Restriction  A restriction that adds extra costs to a combination of two specific edges. This can, for example, be used to model traffic rules like a restriction to turn right on a road by adding a restriction with infinite cost.
Chapter 2

Assignment

2.1 Goals

Flinenberg has designed an algorithm that partitions graphs, called the merging algorithm. It is discussed in detail in Section 3.1. Our goals are to analyze, optimize and improve the merging algorithm both for speed and quality of the found partition. Because the merging algorithm is a central subject in this research we will often refer to it as the algorithm, instead of the merging algorithm.

The time complexity of the initial implementation was $O(n^3)$, where $n$ is the number of nodes in the input graph. It is expected that a time complexity of $O(n^2)$ is possible. In any case, the algorithm will be optimized as much as possible, within certain limits. The law of diminishing returns holds here; at some point the effort necessary to achieve an improvement outweighs the improvement itself. At that point it will be time to look at the quality of the results of the algorithm.

We assume that the quality of a solution is indicated by the value of the partitions it finds. This value is calculated by the selection function. There are some problems here; the optimal partition is unknown. So there is no way of telling how close the found solution is to the optimal solution. The only way to say something about the quality of a partition is to compare a new partition with previously found partitions.

Furthermore the default selection function, which assigns a value to a partition, might not correspond to the actual quality of a partition. This has to be examined too.

2.2 Computational Environment

Two environments are available to complete this assignment.

The first is a Unix server: an UltraSparc with two 360MHz processors running Solaris 2.6. The second is a Linux server: a dual Pentium III 1400MHz with the Redhat 7.3 distribution of Linux. These servers have 2 GB of working memory.

The reason that there are two environments is that the Linux server is used for heavy duty calculations and the Unix server as a general server. For this reason most of the available development tools are solely available on the Solaris platform.
So while the final implementation of the algorithm is running on a Linux server, for debugging and profiling the Unix server is used.

Note that because of the difference in architecture the program may perform differently on the two servers. The Linux server has a 1400MHz processor, but that does not mean it is about 4 times faster than the Unix server. Other factors have to be taken into account like the number of instructions per clock cycle, the amount of work performed by one instruction, etc [14].

Nevertheless, we assume that the difference scales linearly. If the (Unix) profile of the program shows that function X takes 30% of the calculation time it is assumed that it also takes 30% of the calculation time on the Linux server.

2.3 The Profiling Tool

The profiling tool Quantify from Rational Software [23] is used is used to examine the speed, and the contribution of each function to the total running time.

Quantify is used when linking a program. The tool estimates the number of clock cycles needed for each block of the program. A block is a number of consecutive instructions without any (conditional) jump instructions among them. It then inserts instructions into each block to keep track of the number of cycles spent in that block.

Note that the estimation is pretty good, but this approach makes it impossible for the tool to measure memory effects like caching or paging realistically. The caching and paging is influenced by the tool, since the tool makes the program bigger, and it uses a lot of memory to keep track of the number processor cycles spent in each block.

System calls are not estimated by the tool, the time spent in these functions is actually measured and converted back to processor cycles. However, it is possible that between finishing the system call and requesting the system time another process interferes. This can cause inconsistencies in the measured time spent in system calls, causing the tool to measure a longer time than is actually necessary.

2.4 Input Graphs

Several graphs are available to experiment with, Table 2.1 contains a list of them and their properties. All of these graphs have been derived from actual maps, and as such they are directed multi-graphs with turn restrictions.

The graphs of the two cities in Germany (Siegen and Wetzlar) not only contain the cities themselves, but also an area around them.

Sophia, a small town in southern France, is primarily used for testing. The other graphs are used until an implementation is found that can partition the map of Netherlands in reasonable time. After that the map of the Netherlands is the most commonly used graph.
## Input Graphs

<table>
<thead>
<tr>
<th>Name</th>
<th>Number of Nodes</th>
<th>Number of Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sophia</td>
<td>1,705</td>
<td>2,015</td>
</tr>
<tr>
<td>Eindhoven</td>
<td>11,087</td>
<td>14,696</td>
</tr>
<tr>
<td>Siegen</td>
<td>19,681</td>
<td>23,952</td>
</tr>
<tr>
<td>Wetzlar</td>
<td>69,460</td>
<td>88,177</td>
</tr>
<tr>
<td>Antwerp</td>
<td>76,608</td>
<td>98,564</td>
</tr>
<tr>
<td>Netherlands</td>
<td>894,200</td>
<td>1,135,280</td>
</tr>
</tbody>
</table>

Table 2.1: Available input graphs
Chapter 3

Partitioning Algorithm

This research focuses on the merging algorithm. The merging algorithm is a simple algorithm that starts out with each node in one cell and then repeatedly merges two cells together. The best partition that is encountered during the execution is saved.

This chapter discusses the merging algorithm and an efficient implementation for it. Throughout the chapter new notation is introduced when necessary; a complete overview can be found in Appendix A.

3.1 Merging Algorithm

The merging algorithm is designed by Flinsenberg and it is used as the basis of our research. The idea behind the merging algorithm is to start out with each node in a separate cell, and then repeatedly selecting two cells and merging them together into one cell until there is only one cell left. During the execution of the merging algorithm, the partition with the best objective value is stored. The merging algorithm roughly works as described in Algorithm 1 and Procedure 1.

The program partitions a connected, directed multi-graph \( G = (N,E) \) into a set of node set, called a partition, \( P \) such that:

\[
\begin{align*}
    c \in P & \Rightarrow c \subseteq N \\
    \bigcup_{c \in P} c & = N \\
    (\forall c_1, c_2 : c_1, c_2 \in P : c_1 \cap c_2 = \emptyset \lor c_1 = c_2) \\
    \sum_{c \in P} |c| & = |N|
\end{align*}
\]

There are many such partitions \( P \), but the algorithm tries to find one such that \( \text{evaluate}(P) \) is minimum.

Note that there is a difference between the SelectAB function that selects the cells that are going to be merged and the selection function \( \text{evaluate}(P) \) that calculates the value of a partition. While SelectAB selects cells, the selection function selects the partition that the algorithm is going to produce.

The SelectAB function is based on two criteria. One is the priority function. The other one is that the difference in cell size may not be too large, as that would cause large variations
Algorithm 1 Merging algorithm
1: \( P := \{ \{v\} | v \in N \}; \)
2: \( P' := P; \)
3: do \(|P| > 1 \rightarrow \)
4: \( A,B := \text{SelectAB}; \)
5: \( P := (P \setminus \{A,B\}) \cup \{A \cup B\}; \{\text{MERGE CELL A AND B}\} \)
6: if evaluate\( (P) < \) evaluate\( (P') \rightarrow \)
7: \( P' := P; \)
8: fi
9: od
10: return \( P'; \)

Procedure 1 SelectAB
1: \( \text{MinNodes} = (\downarrow c \in P :: |c|); \)
2: if \((\exists C,D : C,D \in P : |C| + |D| \leq \text{Constant} \times \text{MinNodes} \rightarrow \)
3: Randomly pick A and B such that \( |A| + |B| \leq \text{Constant} \times \text{MinNodes}\) and priority\( (A,B) \)
4: else
5: Randomly pick A and B such that \( |A| = \text{MinNodes}\) and priority\( (A,B) \) is maximal;
6: fi
7: return A,B;

in the search graph size, which in turn causes large variations in the time required to plan a route. Therefore the algorithm tries to select cells that have a combined size that is smaller than a certain constant multiplied by the size of a smallest cell. In case this is impossible the smallest cell and one other cell (based on the priority function) is selected.

For every partition \( P \) we define the function \( BE \), which returns that set from the power set of \( E \) that is the set of boundary edges of a given cell:

\[
BE : \mathcal{P}(N) \rightarrow \mathcal{P}(E)
BE(c) = \{ e \in E | (\delta_1(e) \in c \land \delta_2(e) \notin c) \lor (\delta_1(e) \notin c \land \delta_2(e) \in c) \}
\]

Note that every boundary edge connects two cells and therefore it is present in the boundary edge set of two different cells. This means that \( BE(A) \cap BE(B) \) can be used to get the set of boundary edges that connect cell \( A \) and cell \( B \).

The default priority function\(^1\) is \( \text{priority}(A,B) = \frac{|BE(A) \cap BE(B)|}{|A| + |B|} \), meaning that small cells with many boundary edges between them have a high priority while large cells with few edges between them have a low priority.

Because the selection of the two cells involves a random factor the entire algorithm is actually executed several times. The best configuration from these runs is given as the final outcome.

The default selection function\(^2\) designed by Flinsenberg [10] describes the expected number of edges in the search graph. It consists of the sum of three terms called \( S, R \) and \( T \). To

\(^1\) There will be experiments with other priority functions. This is something to take into account when designing the implementation.

\(^2\) Again we have to take into account that there will be experiments with other selection functions.
3.1. Merging Algorithm

describe these we first need to introduce some more functions. The function $IE$ returns the set of internal edges of a cell and the function $BN$ returns the set of boundary nodes of a cell.

$$IE : \mathcal{P}(N) \rightarrow \mathcal{P}(E)$$
$$IE(c) = \{ e \in E | \delta_1(e) \in c \land \delta_2(e) \in c \}$$

$$BN : \mathcal{P}(N) \rightarrow \mathcal{P}(N)$$
$$BN(c) = \{ v \in c | \exists e \in BE(c) : \delta_1(e) = v \lor \delta_2(e) = v \}$$

The first term of the objective value ($S$) denotes the contribution of the internal edges of each cell to the expected size of the search graph. The formula to calculate $S$ is:

$$S(P) = \sum_{c \in P} \left( \frac{|c|}{|N|} \times (2 - \frac{|c|}{|N|}) \times |IE(c)| \right).$$

It takes into account the size of the cells. Large cells have a large probability of containing the start or destination and they add more edges to the search graph.

Term $R$ describes the edges that have to be added to the search graph to represent the routes through each cell. The formula to calculate $R$ is:

$$R(P) = \sum_{c \in P} \left( (1 - \frac{|c|}{|N|})^2 \times |BN(c)| \times (|BN(c)| - 1) \right).$$

It also takes into account that the routes through the cells containing the start or destination do not have to be included in the search graph.

Finally, $T$ describes the contribution of the boundary edges to the search graph. The formula to calculate $T$ is:

$$T(P) = \frac{1}{2} \sum_{c \in P} (|BE(c)|).$$

Note that the factor $\frac{1}{2}$ is introduced to get the correct number of boundary edges. Each boundary edge occurs once in the $BE$ set of both the cells it connects, so without the factor each boundary edge would be counted twice.

Together these three terms form the complete default selection function, also called the objective value:

$$\text{obj.val}(P) = \sum_{c \in P} \left( \frac{|c|}{|N|} \times (2 - \frac{|c|}{|N|}) \times |IE(c)| \right) +$$
$$\sum_{c \in P} \left( (1 - \frac{|c|}{|N|})^2 \times |BN(c)| \times (|BN(c)| - 1) \right) +$$
$$\frac{1}{2} \sum_{c \in P} |BE(c)|. \quad (S)$$

$$\text{evaluate}(P) = \text{obj.val}(P). \text{ In Chapter 5 we will see an other possible selection function.}$$

Calculating the value from scratch involves a rather large calculation, but there is a quicker way to calculate a new objective value. By using only the original objective value, the total number of nodes in the graph and information about the original cells and the newly merged cell, it is possible to calculate the value in an $O(1)$ operation.
Assume that $P' = (P \setminus \{A, B\}) \cup \{C\}$ where $A \in P$, $B \in P$ and $C = A \cup B$. Then for each of the terms of the selection function it is possible to calculate the objective value of $P'$ based upon the objective value of $P$, as shown below.

$$S(P') - S(P) = -\left[\frac{|A|}{|N|}\right] \times (2 - \left[\frac{|A|}{|N|}\right]) \times |IE(A)| - \left[\frac{|B|}{|N|}\right] \times (2 - \left[\frac{|B|}{|N|}\right]) \times |IE(B)|$$
$$+ \left[\frac{|C|}{|N|}\right] \times (2 - \left[\frac{|C|}{|N|}\right]) \times |IE(C)|$$

$$R(P') - R(P) = -(1 - \left[\frac{|A|}{|N|}\right])^2 \times |BN(A)| \times (|BN(A)| - 1))$$
$$- \left((1 - \left[\frac{|B|}{|N|}\right])^2 \times |BN(B)| \times (|BN(B)| - 1))$$
$$+ \left((1 - \left[\frac{|C|}{|N|}\right])^2 \times |BN(C)| \times (|BN(C)| - 1))$$

Because $BE(C) = (BE(A) \cup BE(B)) \setminus (BE(A) \cap BE(B))$ the formula for $T(P') - T(P)$ can even be simplified a bit:

$$T(P') - T(P) = -\frac{1}{2}|BE(A)| - \frac{1}{2}|BE(B)| + \frac{1}{2}|BE(C)|$$
$$\{BE(C) = (BE(A) \cup BE(B)) \setminus (BE(A) \cap BE(B))\}$$
$$= -\frac{1}{2}|BE(A)| - \frac{1}{2}|BE(B)| + \frac{1}{2}|BE(A) \cup BE(B)| \setminus (BE(A) \cap BE(B))$$
$$= -\frac{1}{2}|BE(A)| - \frac{1}{2}|BE(B)| + \frac{1}{2}(|BE(A)| + |BE(B)| - 2|BE(A) \cap BE(B)|)$$
$$= -(|BE(A) \cap BE(B)|$$

That $BE(C) = (BE(A) \cup BE(B)) \setminus (BE(A) \cap BE(B))$ holds is easy to see. The boundary edges between $A$ and cells other than $B$, and the boundary edges between $B$ and cells other than $A$ will become boundary edges of cell $C$, because the node they were connected to in cell $A$ or $B$ is now in cell $C$, while the other node were they connected to is still in another cell. But the boundary edges between $A$ and $B$ become internal edges, since both their nodes are now in the same cell, namely cell $C$.

The combination of the difference of the terms results in:

$$obj\_val(P') - obj\_val(P) = -\left[\frac{|A|}{|N|}\right] \times (2 - \left[\frac{|A|}{|N|}\right]) \times |IE(A)|$$
$$- \left[\frac{|B|}{|N|}\right] \times (2 - \left[\frac{|B|}{|N|}\right]) \times |IE(B)|$$
$$+ \left[\frac{|C|}{|N|}\right] \times (2 - \left[\frac{|C|}{|N|}\right]) \times |IE(C)|$$
$$- ((1 - \left[\frac{|A|}{|N|}\right])^2 \times |BN(A)| \times (|BN(A)| - 1))$$
$$- ((1 - \left[\frac{|B|}{|N|}\right])^2 \times |BN(B)| \times (|BN(B)| - 1))$$
$$+ ((1 - \left[\frac{|C|}{|N|}\right])^2 \times |BN(C)| \times (|BN(C)| - 1))$$
$$- (|BE(A) \cap BE(B)|$$

Since the (default) priority function and objective value can be calculated in $O(1)$ operations they should not be the limiting factor to the speed of the algorithm. But keeping track of the data required by these functions might impose such a limit.
3.2 Analysis

Before implementing the merging algorithm an analysis was performed. This section describes this analysis and how it lead to the implementation.

3.2.1 General

In general an algorithm can be made faster by storing more information. Calculating the objective value of a new partition from the previous section is a good example; by using information (the old objective value) it is possible to calculate the new one with an operation of $O(1)$, while entirely recalculating the objective value would be an operation of $O(|P|)$. Hereby we assume, of course, that all information necessary for this calculation is available.

We need to store information such that operations that are executed the most can be discarded, or implemented in a more efficient manner. This sounds easy, but keeping track of extra information costs extra time. So we have to find a balance between storing and computing information.

The three parts of the algorithm that are executed most often are the SelectAB function and the merging of cells (called the Merge procedure), followed by saving the best partition. In the next sections the information needed by these procedures and a way to store it will be discussed.

3.2.2 Cellgraph

Section 3.1 described the data required by the functions SelectAB and $P := (P \setminus \{A, B\}) \cup \{A \cup B\}$ (also called the Merge procedure). When looking at the algorithm, it seems that two types of information are needed. Information about cells (the number of (boundary) nodes/edges in it) and information about the relation between two cells (the number of edges and priority). This can be represented by a graph, which is called Cellgraph to distinguish it from the original graph and the boundary graph. The Cellgraph is the actual data structure used by the partitioning algorithm and it represents an ordinary, undirected graph.

**Definition 3.2.1** The Cellgraph consists of cellnodes containing information about cells and celledges containing information about cell pairs.

Note that there only need to be celledges between cells that have boundary edges between them. It makes no sense to merge two cells if they have no edges between them. There are two ways of looking at this; one is to see cells without a connection between them as parts of the map that are separated by natural boundaries. When dividing a map into pieces it is only natural to divide it along those natural boundaries.

But it is also possible to look at it from a more theoretical point of view; say $A$ and $B$ are cells that have no boundary edges between them (i.e. $BE(A) \cap BE(B) = \emptyset$). Then the objective value will not drop by merging them. The proof for this can be found in Appendix B.1.

In short there is no reason to merge cells without edges between them. So information on cell pairs that have no edges between them is not needed, since they will not be merged.
Therefore that information may be left out. This is fortunate, as it eliminates the need to have a celledge between every pair of cellnodes, saving a lot of memory.

So in a cellgraph there are cellnodes for every cell in the partition and there are celledges between every pair of cells A and B for which it holds that \( BE(A) \cap BE(B) \neq \emptyset \).

Below are the basic data types used by the algorithm. The notation \( cellnode(c) \) is used to describe a reference to the record that contains information about cell c. If \( cn = cellnode(c) \) then the following abstraction functions apply ('\( \rightarrow \) is used for dereferencing and accessing a field of a record):

\[
\begin{align*}
[cn \rightarrow nodes] &= c \\
[cn \rightarrow nr\_nodes] &= |c| \\
[cn \rightarrow nr\_edges] &= |IE(c)| \\
[cn \rightarrow nr\_bnodes] &= |BN(c)|
\end{align*}
\]

The nodes field represents the collection of nodes of the cell, its exact representation is discussed later in this chapter.

A celledge representing the connection between cell A and cell B is described in a record referenced by \( celledge(A, B) \). Assuming that \( ce = celledge(A, B) \) only one abstraction function currently applies:

\[
[ce \rightarrow nr\_edges] = |BE(A) \cap BE(B)|
\]

The cellgraph is undirected, it does not take into account the direction of the boundary edges between the cells, thus \( celledge(A, B) = celledge(B, A) \). Also, since the celledges represent the relationship between two cells there is never more than one celledge between any pair of cells. The idea is to group the information about a cell pair into one data structure called a celledge, allowing multiple celledges for a pair of cells would make this impossible.

Note that these types are not yet complete, they contain the minimum amount of information required to run the algorithm. In the next two sections, we analyze how this information plays a role in the Merge and SelectAB procedures, and it will become clear that more information is necessary for an efficient implementation.

### 3.2.3 The Merging Procedure

In this section, the merging procedure is examined, and in particular what has to happen to the data in the data structures to keep it up to date. To accomplish this, new data structures are introduced and existing data structures are extended in the course of this section.

**Collection of nodes**

As described before, there is a collection of nodes for each cell. The only time we need this collection is when merging two cells, then two of such collections need to be merged into one. This can easily be done by using a linked list to represent the collection and not only keeping a reference to the first element of the list, but also a reference to the last element. This way two of such lists can easily be concatenated into a new one. Checking whether an element occurs twice after concatenation is not necessary: since each node starts out in a single cell, the concatenation of these two lists will never result in a node occurring twice in the collection of nodes of a cell.
3.2. Analysis

So, we can now introduce two new types called linked list and head of list. A linked list is a record that contains an element and a reference to the rest of the linked list. If $ll$ is a reference to a linked list then:

$$ll = \text{nil} \Rightarrow [ll] = \emptyset$$
$$ll \neq \text{nil} \Rightarrow [ll] = \{ [ll \rightarrow \text{element}] \} \cup \{ ll \rightarrow \text{next} \}$$

Note that a linked list actually represents a list instead of a set. It would be possible to define an abstraction function that maps the linked list to a mathematical list and then map that mathematical list to a set, but since we have no need for the mathematical list we skip that step and map directly to a set.

The head of list is a record that contains two references to a list, the first element of the list and the last element of the list. If $hl$ is of the type head of list then:

$$[hl] = [hl.\text{first}]$$

But there is an additional constraint:

$$hl.\text{last} = \text{last}(hl.\text{first})$$

where the function $\text{last}$ is defined as:

$$\text{last}(\text{list}) = \begin{cases} 
\text{nil} & \text{if } \text{list} = \text{nil} \\
\text{list} & \text{if } \text{list.next} = \text{nil} \\
\text{last}(\text{list.next}) & \text{otherwise}
\end{cases}$$

The '.' is used to access fields of a record without dereferencing.

Simple Numbers

The number of nodes and number of edges in a cell are necessary for calculating the objective value of a new partition and for calculating the priority of its celledges. Storing them does not cause any extra work when merging two cells together, these numbers just add up. Initialization is also very easy; the number of nodes of a cell is one and the number of edges is zero.

The number of edges between two cells that are merged also has to be added to the number of edges of the new cell. The edges that used to connect the cells will be part of the new cell. Storing the number of edges between two cells in a celledge causes no real overhead: upon merging two cells the celledge between these cells is destroyed and the number of edges between the two cells is used to calculate the number of edges in the new cell and the objective value of the new partition. During initialization each cell represents a single node, so the number of edges between two cells can be set to the number of edges between the two nodes these cells represent, which should generally be one or two.

So far, it seems that everything is easy, but there are two problems: merging celledges and the number of boundary nodes of merged cells.
Merging CellEdges

First we consider the merging of celledges. Consider the example in Figure 3.1. Figure 3.1 shows only the boundary nodes, boundary edges, cells and celledges. The internal nodes and edges have been left out for clarity.

The solution presented so far assumes that one celledge represents the relation between two cells. Cells A and B both have a celledge between themselves and cell D. So if A and B are merged the new cell needs to have only one celledge between itself and cell D.

This introduces two problems: the detection that both A and B have a relation with the same cell and the actual merging of two celledges. The merging is quite easy; the information stored in the celledges (so far) is the number of edges between cells and those numbers can simply be added. The detection of such a relation is a bit harder.

A graph coloring algorithm is a simple solution that can be used to detect whether merging is necessary. Every cell connected to cell A with a celledge is colored, after that every cell connected to cell B with a celledge is examined to check its color.

Figure 3.2: The merging of celledges in a cellgraph
3.2. Analysis

Instead of a color a reference to a celledge is used. This reference by default refers to nil, but it is used in the algorithm to indicate the "color" of the cell. A cell is colored by setting the reference to the celledge that was used to reach it. This has the advantage that it is possible to quickly determine which celledges have to be merged (if any). The algorithm is displayed in Figure 3.2, which shows only cells and celledges for simplicity. It comes down to this:

Step 1 Check all celledges of cell A (the dashed celledges in the picture) and let the reference of the cellnode connected to the other end of the celledge refer to the celledge (the arrows). Note that the celledge between cell A and B may be ignored.

Step 2 Check all celledges of cell B. If the reference of the cellnode on the other end of the celledge does not refer to nil then it is referring to the celledge with which this celledge has to be merged. Again the celledge between cell A and B may be ignored.

Step 3 Set the references that were changed in Step 1 back to nil again.

If $NC(c)$ is the set of cells neighboring cell $c$:

$$ NC : P(N) \rightarrow P(P) $$

$$ NC(c) = \{cell \in P([BE(c) \cap BE(cell)] \neq \emptyset) \}

The algorithm described above is of order $O(|NC(A)| + |NC(B)|)$. Afterward it is known which celledges have to be merged.

Another option is to keep the list of celledges connected to a cellnode sorted by the id of the cell on the other side of the edge. This way the two lists of merging cellnodes can be easily checked for duplicates in $O(|NC(A)| + |NC(B)|)$ with a smaller constant than for the algorithm described above. But it would also mean that the lists of all cellnodes connected to the new cell have to be resorted, since the new cell will have a different id. So the previous solution is more efficient.

All this requires that the celledges connected to a cellnode can be easily determined. This can be achieved by storing, in the cellnode, a double linked list of references to the celledges connected to this cellnode. We choose a double linked list to make the deletion of a celledge during the merging process an efficient operation. Of course, this can only be done if the list does not have to be searched first, so a celledge will also need a reference to the element of the list that contains it.

The double linked list is the same as the linked list that was introduced earlier, the only difference is that a double linked list also has a field prev that refers to the predecessor of the element in the list.

The cedges field for cellnodes is now introduced to contain the set of celledges that are connected to the cellnode. This cedges field is of the type head of list, but this head of list contains a double linked list instead of a normal linked list. The head of list is used to make it possible to concatenate the lists of two merging cells quickly. Also, by concatenating the lists the construction of a new list for the new cell is avoided. This means that the references kept by each celledge to their position in the cedges list do not have to be updated, which saves some operations.

To keep track of its position in the list of cedges a celledge is now extended with some extra fields. The fields end1 and end2 are records of the type celledge_end that contain information
about the endpoints of the celledge. We assume for now that the functions $\delta_1$ and $\delta_2$ return a reference to the cell for which $end1$ and $end2$ respectively contain the information. This means that there are now two $\delta_1$ and $\delta_2$ functions. One applies to normal edges and one applies to celledges, but since their meaning is the same (i.e. getting the endpoint of a connection) we feel that there is little chance for confusion.

So if $ce = celledge(A, B)$ then:

$$(\delta_1(ce) = cellnode(A) \land \delta_2(ce) = cellnode(B)) \lor (\delta_1(ce) = cellnode(B) \land \delta_2(ce) = cellnode(A)),$$

meaning that the information about the endpoints of the celledge between cell $A$ and cell $B$ is either stored in $end1$ and $end2$ respectively, or in $end2$ and $end1$ respectively.

We can then define the constraint that a celledge needs to store a reference to its position in the cedges list of the cells it connects. For every $ce = celledge(A, B)$ it must hold that:

$$ce \rightarrow end1.list.pos = list.element(\delta_1(ce) \rightarrow cedges, ce)$$
$$ce \rightarrow end2.list.pos = list.element(\delta_2(ce) \rightarrow cedges, ce)$$

Where the $list.element$ function returns a reference to the linked list structure containing the specified element:

$$list.element(list, el) = \begin{cases} list & \text{if } list \rightarrow element = el \\ list.element(list \rightarrow next, el) & \text{otherwise} \end{cases}$$

**Boundary Nodes**

Now consider the number of boundary nodes after merging two cells. It is necessary to store the number of boundary nodes in a cellnode, but there is a problem when merging two cells. The number of boundary nodes of both cells cannot just be added; the point of merging is that some boundary nodes disappear. The boundary node $q$ in Figure 3.1 for example becomes an internal node if cell $A$ and $B$ are merged. So, to solve this problem, additional information is needed.

Storing the number of boundary nodes connected to a celledge does not help either. Take a look at the example in Figure 3.1 again. The boundary node $p$ in cell $A$ is a boundary node not only because it has $q$ as a neighbor in cell $B$, it also has $r$ as a neighbor in cell $D$. Thus, when merging $A$ and $B$, $p$ remains a boundary node, even though the edges between $A$ and $B$ become internal edges.

The simplest solution to this problem is to store a collection of boundary nodes in a cellnode, accompanied by a reference count. The reference count indicates the number of celledges that are connected to the boundary node. Furthermore each celledge stores a list of references to this boundary node and reference count pair. This list of the celledge only contains references to those boundary nodes that are boundary nodes because they have an edge in the celledge.

Thus we create a new record type called refcount that contains a node and a number. The abstraction function for $rc$ which is a reference to a refcount then becomes:

$$[rc] = (rc \rightarrow node, rc \rightarrow nr)$$
3.2. Analysis

We will use \( refcount(node) \) to indicate a reference to the refcount record about that particular node.

These refcount records are stored in a new field of a cellnode called \( bnodes \):

\[
[\text{cellnode}(c) \rightarrow \text{bnodes}] = \{[\text{refcount}(node)] | \text{node} \in BN([\text{cellnode}(c) \rightarrow \text{nodes}])\}
\]

We now define \( BNR(A, B) \) as the set of boundary nodes of cell \( A \) that are connected via a boundary edge to a boundary node in cell \( B \):

\[
BNR : \mathcal{P}(N) \times \mathcal{P}(N) \rightarrow \mathcal{P}(N)
BNR(c_1, c_2) = \{ n \in BN(c_1) | (\exists e \in BE(c_1) \cap BE(c_2) : \delta_1(e) = n \lor \delta_2(e) = n) \}
\]

With this definition it is possible to extend the celledge_end record with a new field. This field contains the list of refcounts that should be updated when the celledge is removed. For a celledge \( ce = \text{celledge}(A, B) \) it holds that:

\[
[\text{ce} \rightarrow \text{end1.bnodes}] = \{ \text{refcount}(node) | \text{node} \in BNR([\delta_1(ce) \rightarrow \text{nodes}], \quad )
\]
\[
[\text{ce} \rightarrow \text{end2.bnodes}] = \{ \text{refcount}(node) | \text{node} \in BNR([\delta_2(ce) \rightarrow \text{nodes}], \quad )
\]

When the celledge between \( A \) and \( B \) is deleted during the merging process the reference count of the boundary nodes can be decreased by one. When a reference count reaches zero the node is removed from the list and the number of boundary nodes of the cell is reduced by one. By storing these refcounts in a double linked list, deleting one can be done quickly and merging two cells results in the lists of the cells being concatenated. By keeping a reference to the last item in the list, as described above for the collection of nodes, this is also a quick operation.

Now merging the celledge between \( A \) and \( D \) and the celledge between \( B \) and \( D \) from the example in Figure 3.1 becomes more expensive. The \( bnodes \) field of the celledges has to be merged. While the lists of boundary nodes belonging to the \( A \) and \( B \) end of the celledges have no nodes in common, the end of the celledges in \( D \) could have some nodes in common (like node \( r \) in the example). If we keep the list of refcount pairs in a celledge sorted we can apply a merge sort algorithm to filter out duplicates. This leads to an operation of \( O(|BNR(A, D)| + |BNR(B, D)| + |BNR(D, A)| + |BNR(D, B)|) \) (Even though the lists from \( A \) and \( B \) have no duplicates, it still needs to be kept sorted for future mergings).

On the other hand, it is also possible to allow duplicates, this means the reference count becomes the total number of times the boundary node is in a celledge's boundary node list. This way merging two celledges remains \( O(1) \), since the lists can just be concatenated. But the deletion of a celledge might become more expensive, because of the possible duplicates in the list. However, allowing the list to be "polluted" with duplicates will, overall, be more time efficient. This is because the duplicates will have to be handled anyway, whether it is during the merging of two celledges or during the deletion of one. But by allowing the duplicates, the list of refcount pairs can be left unsorted, saving two merge sort operations each time two celledges have to be merged.

Note that this means that the \( bnodes \) field of a celledge_end record no longer represents a set, but a bag.
3.2.4 The Selection Procedure

The information we decided to store so far is enough to make the SelectAB procedure work, but there are ways to make it work more efficiently. For example, the SelectAB procedure selects celledges, so it is logical to let each celledge store pointers to the two cells it connects. This means that a record of type celledge.end gets an extra field called cell, such that for any \( ce = celledge(A, B) \) it holds that:

\[
\begin{align*}
    ce \rightarrow \text{end1.cell} &= \delta_1(ce) \\
    ce \rightarrow \text{end2.cell} &= \delta_2(ce)
\end{align*}
\]

Also a decision on the way to store the total collection of cellnodes and/or celledges still has to be made. Since the cellnodes contain pointers to celledges and the celledges contain pointers to cellnodes storing only the total collection of one of them is enough. But as we will see it is better to store both collections.

The cellnodes need to be stored such that it is easy to determine the cellnode with the minimum number of nodes. Furthermore two cellnodes are removed from the data structure and a new one is added during each merge operation. This can be implemented in exactly that way, or it is possible to implement it by deleting one cellnode and updating the information of the other one. So the data structure would need to support quick delete operations and either quick insert or quick update operations. Note that if we chose for the update operation the number of nodes of the cell after the update is higher than the number of nodes before the update. This information might be useful in selecting the right data structure. But so far a priority queue seems the best data structure.

The result of the priority function can be stored in a celledge. By storing the result, there is no need to calculate the priority function for every celledge and searching for the celledge with the highest result. It becomes possible to sort the celledges, and let the SelectAB procedure only search a part of the collection of celledges, namely those cells whose priority function is equal to the maximum. Furthermore we only need to recalculate the priority function for the celledges involved in the merging of two cells, i.e. the celledges connected to the two original cells. Note that this rests on the assumption that the priority function only depends on local changes, i.e. changes to one of the cells it is connected to.

\[
[celledge(A, B) \rightarrow \text{priority}] = \text{priority}(A, B)
\]

So for the celledges a data structure is needed that supports quick deletion and quick updating and during initialization fast insertion too. Deletion is used for deleting celledges during merging, and updating is used for adjusting the value of the priority function for all celledges connected to a merged cell. A priority queue is a logical choice for storing celledges. However, instead of the item with the highest priority, the item with the highest priority that satisfies a certain condition has to be found by the SelectAB function. Therefore it should be possible to iterate through the structure from high to low priority, to find the element that satisfies the condition.

Since there are several alternatives for implementing a priority queue, the exact implementation will be discussed later on in Subsection 3.4.5.
3.3 Implementation

The merging algorithm is implemented in such a way that it can be executed multiple times and select the best configuration from all runs. The default setting is to use 5 runs. This section describes the problems that arose during the implementation of the design described in the previous section.

A significant change to the algorithm was made once it turned out that not all input files contained connected graphs. Often the unconnected parts are only small. This presented a problem for the SelectAB procedure. The SelectAB procedure selects two cells whose merging does not result in a cell that is much larger than the smallest cell. Failing to find such a cell will result in the selection of the smallest cell and another cell.

The SelectAB procedure returns celledges and there are no celledges between cells that have no connection with each other. So when a small disconnected part of a graph becomes the smallest cell there is a problem; other cells keep growing through mergings and eventually the smallest cell has to be selected, but since it has no connection to other cells this is impossible. Therefore the change was made to look at the number of nodes in the smallest connected cell, instead of just the smallest cell.

Furthermore, the used graphs represent road networks. As such they have traffic rules associated with them. These rules represent, for example, turning restrictions imposed by traffic rules. On some crossroads a sign indicates that the driver may not turn right, or that he may not make a U-turn. Such restrictions are modeled by traffic rules.

It is important that there are no traffic rules between boundary edges and internal edges, since that would make planning with the partitioned graph too complicated. Therefore, before starting the actual algorithm, cells are merged to make sure that every edge with a traffic rule imposed on it is an internal edge. Also edges with traffic rules between them have to be in the same cell. To accomplish this, all edges connected to a junction were placed in the same cell, since only edges connected to the same junction can have traffic rules between them.

There were also some practical problems. Research on graph partitioning at Siemens VDO had already resulted in some tools, and more importantly, file formats. There were 7 different file formats in use, making it necessary for the program to understand all of them. There are libraries for doing this, but they load the files into their own data structures. Loading the file into the program's own data structure was more efficient. Later we developed an extensible file format to replace the different formats and allow for future extension. A description of this format can be found in Appendix C.

For the first implementation we chose to store the cells and celledges in balanced binary trees, called AVL trees [1]. All operations on AVL trees are of the order $O(log N)$ (albeit with a large constant). This is especially important in the beginning of the algorithm when the priority function assigns the same priority to most celledges. Using a tree without balancing in this case, leads to a degenerate tree. Using AVL trees probably does not lead to the most efficient algorithm, but it allows for quick testing and discovering what the most costly operations are.

The results of each implementation are displayed in a table like Table 3.1. The "Graph" column gives the name of the graph used as input. The "Time" column gives the processing time required by the algorithm. This processing time is the processing time on the Linux
<table>
<thead>
<tr>
<th>Graph</th>
<th>Time</th>
<th>Value</th>
<th>% Exp.</th>
<th>Expensive Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sophia</td>
<td>4s</td>
<td>520</td>
<td>55%</td>
<td>SelectAB</td>
</tr>
<tr>
<td>Eindhoven</td>
<td>2:40m</td>
<td>8,376</td>
<td>64%</td>
<td>SelectAB</td>
</tr>
</tbody>
</table>

Table 3.1: Results of the first implementation

<table>
<thead>
<tr>
<th>Graph</th>
<th>Time</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sophia</td>
<td>10s</td>
<td>538</td>
</tr>
<tr>
<td>Eindhoven</td>
<td>19:36m</td>
<td>8,763</td>
</tr>
<tr>
<td>Siegen</td>
<td>51:21m</td>
<td>7,524</td>
</tr>
</tbody>
</table>

Table 3.2: Results of the original implementation

server. For the smaller graphs there is little difference between the environments, but as the graphs get bigger the differences get larger. The objective value of the best partition found by the algorithm is described in the "Value" column. This value may fluctuate during the search for the fastest implementation. But in general it should not get much worse than the partition found by the original program. In any case the speed of the algorithm is the primary concern now, improving quality will be done later on, once all the speed improvements have been implemented. The last two columns describe the percentage of the processing time that is used by the most expensive operation and the name of that operation.

For reference the results of the original implementation have been displayed in Table 3.2. Since no profiling has been done for that implementation there is no information about the most expensive operation. But considering the results, the first implementation is already a big improvement over the original one.

### 3.4 Improvements

This section describes the most important improvements that were made to the merging algorithm and the implementation thereof. A lot of small improvements have been made, but only those that had a major impact on the running time of the algorithm are described here.

#### 3.4.1 Random Number Generator

It turns out that the choice for AVL-trees has little impact, it is the SelectAB function that requires the most processing time. A major part of the processing time was used to calculate a (pseudo) random number. In his book "The Stanford Graphbase" [19], Knuth describes a fast and uniform random number generator. Knuth’s generator was used to replace the random number generator in the standard library, which was slower and not as uniform. The results are shown in Table 3.3.
3.4. Improvements

<table>
<thead>
<tr>
<th>Graph</th>
<th>Time</th>
<th>Value</th>
<th>% Exp.</th>
<th>Expensive operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sophia</td>
<td>3s</td>
<td>582</td>
<td>38%</td>
<td>SelectAB</td>
</tr>
<tr>
<td>Eindhoven</td>
<td>111s</td>
<td>8,469</td>
<td>48%</td>
<td>SelectAB</td>
</tr>
</tbody>
</table>

Table 3.3: Results with the new random number generator

3.4.2 Improved Priority Function

Improving the random number generator was not enough, generating random numbers and iterating through the possible candidates still takes a lot of time. Individually these actions cost only a few processor cycles, but they are executed in a loop until a suitable candidate is found. This loop iterates many times during the course of the algorithm. So the number of iterations has to be reduced, or ideally the entire loop has to be eliminated.

The loop was needed to randomly select a celledge from all celledges with the same priority. And therein lies a possible solution. By giving the priority function the responsibility that each celledge has a different value it becomes be possible to just pick the one with the highest priority. This can be done by introducing a small random factor in the calculation of the priority function. Even if this random factor is not enough to give each celledge a unique priority, it will certainly make the selection operation random. The priority function then becomes:

\[
priority(A, B) = \frac{|BE(A) \cap BE(B)|}{|A| + |B|} \times random(1, 1.01),
\]

where \(random(a, b)\) randomly selects a number between \(a\) and \(b\). A small range was chosen to prevent the random factor from becoming the most important factor in the priority function. The random factor should only provide a small change in the value of the priority function, in order to make as much priorities differ from another as possible.

Note that the ordering of the original priority function is not preserved by this new function. It is possible that the original priority function gave celledge \(X\) a higher priority than celledge \(Y\), while the new priority function gives \(Y\) a higher priority than \(X\). But the difference in the objective value of the found partitions is not that big.

This is an efficient solution, the priority function is only recalculated when necessary, so the random function will be called a lot less. But it only works if the celledge with the highest priority satisfies the condition that merging the two connected cells does not produce a cell that is too large. If this condition is not satisfied then a search has to be done to find the celledge with the second highest priority and check if that one satisfies the condition, etc. Only if the celledge with the highest priority satisfies this condition the loop is cut off, otherwise it has to be executed as normal.

The goal of the selection operation is to select the most appropriate celledge. The value of the priority function indicates how appropriate a celledge is. The only thing the priority function does not take into account is the condition of not creating too large cells. If this condition can somehow be incorporated into the priority function then the entire loop can be eliminated.

An example is to add the factor \(\frac{1}{|A| \times |B|}\) to the priority function. Although it does not strictly enforce the condition, it does lower the priority function for large cells. Flinsenberg introduced the condition to prevent big differences in cell sizes. So if the same can be accomplished by
altered the priority function and the adaptation would provide significant improvement in
the running time, it is approved.

The improvement in running time is very high (see Table 3.4). The SelectAB function which
required the majority of the processing time is suddenly reduced to the simple matter of
selecting the element with the highest priority. Also it is no longer necessary to store the cells
in a priority queue to determine the smallest cell.

So the change in the priority function is accepted, resulting in the priority function:

\[
priority(A, B) = \frac{1}{|A| \times |B|} \times \frac{|BE(A) \cap BE(B)|}{|A| + |B|} \times \text{random}(1, 1.01).
\]

3.4.3 Decreased Saving

For the large graphs saving the best partition (the Save procedure) now requires the most
processing time. It is quite expensive and it is executed many times, because the algorithm
often encounters a partition that (slightly) improves upon the best partition so far. Since it
is hard to speed up the operation itself the number of executions has to be reduced.

The simplest way to do this, is to have a good estimate for the value of the best partition
that will be encountered. If the value of the best (encountered) partition is actually known
beforehand, only one save operation has to be done, namely when that partition is encoun-
tered.

This is (of course) not possible, but a very good estimate is the value found in the first run.
Actually for the Antwerp and Netherlands graphs the first run produced the best partition
right away, later runs did not provide any better solutions. So it is possible to perform a
run of the algorithm and only save the value of the best partition encountered, but not the
partition itself. This value can be used as an estimate in later runs.

The problem that the solution of the first run is not saved, while it actually produces the
best result, can be solved by repeating the first run exactly, if this situation occurs. So if, at
the start of the final run, the algorithm has not found a better partition, then the algorithm
seeds the random number generator with the same value as for the first run. That way the
first run is repeated exactly.

The downside of that is that one run (the last one) may be sacrificed to recalculate an earlier
partition. It is hard to tell what kind of effect this has on the quality of the found partition.
3.4. Improvements

<table>
<thead>
<tr>
<th>Graph</th>
<th>Time</th>
<th>Value</th>
<th>% Exp.</th>
<th>Expensive operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wetzlar</td>
<td>30s</td>
<td>23,709</td>
<td>32%</td>
<td>Merge</td>
</tr>
<tr>
<td>Netherlands</td>
<td>39m</td>
<td>200,751</td>
<td>32%</td>
<td>Initialization</td>
</tr>
</tbody>
</table>

Table 3.5: Results after applying decreased saving

<table>
<thead>
<tr>
<th>Graph</th>
<th>Time</th>
<th>Value</th>
<th>% Exp.</th>
<th>Expensive operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wetzlar</td>
<td>13s</td>
<td>23,709</td>
<td>36%</td>
<td>Merge</td>
</tr>
<tr>
<td>Netherlands</td>
<td>4:10m</td>
<td>200,751</td>
<td>31%</td>
<td>Merge</td>
</tr>
</tbody>
</table>

Table 3.6: Results after several optimizations

However, it turned out that the time saved by this solution is enough to allow for an extra run if that would be deemed necessary, as can be seen in Table 3.5.

3.4.4 General Optimizations

Initialization of the algorithm now becomes more and more important. In order to reduce the initialization time, the whole program is reviewed and optimized. The assumption that nodes and edges in the input file have an id ranging from 1 to the total number of nodes/edges in the file made a considerable difference. The nodes and edges can now be loaded into an array where their ids are used as the index. This resulted in a major speedup in the initialization step. It required renumbering of a few nodes and edges in the maps of Eindhoven, Antwerpen and the Netherlands, but that was no problem.

Also the memory efficiency of the implementation has been increased, both by using less memory and by using some suggestions from Henessy & Pattersons’ book [14] from the chapter on Memory Hierarchy Design to improve the locality of reference. Furthermore, the compiler optimization level is set to maximum to produce faster code. Also, the amount of information outputted by the program during a run is reduced to minimize the time the algorithm is waiting for I/O.

All these optimizations lead to the performance results in Table 3.6.

3.4.5 Priority Queues

Finally the AVL-trees were replaced by other priority queue implementations. The cells no longer need to be placed in a priority queue, because of the change in priority function (see Section 3.4.2). Only the celledges need to be placed in one, to make it possible to quickly determine the celledge with the highest priority.

We try two implementations: the binary heap [24] and the pagoda [12].

As described by Françon et al [12], pagodas are not the best implementation when looking at worst-case efficiency. But for achieving good average performance the pagoda is a good choice for large priority queues. Françon et al also compare the standard heap implementation with a pagoda. In short these are the comparisons: heaps require less memory but contiguous
storage allocation. Furthermore, on average a pagoda is faster when comparing multiple insertion operations, and pagodas can be merged.

The fact that a heap needs contiguous storage allocation poses no problem here, because the required storage size is known beforehand, because the number of celledges can never exceed the number of edges in the graph. Also the merge operation for queues is not required for our program. So, the only thing that improves by using pagodas, is the time it takes to insert the edges in the priority queue at the beginning of a run.

However, the binary heap is the fastest implementation, for several reasons. First because of its memory characteristics: the memory access pattern of the binary heap is quite regular and localized, decreasing the chance of cache misses. Secondly, the pagoda is harder to iterate through during the save operation. It has to rely on several recursive functions, while the binary heap is just an array that can be easily iterated on.

Furthermore the pagoda is hard to implement efficiently, mainly because of its memory characteristics. As can be seen in Table 3.7 the program becomes approximately 20% slower compared to using the AVL-trees (Table 3.6). With more work it should be possible to program it in such a way that it is faster than the implementation with the AVL-tree, but it is highly unlikely that it will become faster than the implementation with the binary heap (see Table 3.8). Therefore we did not attempt to improve the implementation of the pagoda.

There are many more implementations of priority queues available. However, the speed of the algorithm will probably not increase drastically by any of them. In fact, the binary heap is likely to be the best choice for this algorithm. That is why we decided to keep the binary heap.

### 3.5 Final Algorithm

The partitioning algorithm in its final implementation differs a lot from the pseudo code given in Section 3.1. This section shows the pseudo-code that corresponds to the final implementation and provides an analysis of the time complexity of this implementation.
3.5. Final Algorithm

3.5.1 Data Types & Operations

Most of the data types used in the program have been described in Section 3.2, but they are summarized here to give an overview.

Three basic types that are used are the head of list, linked list and double linked list. The linked list and double linked list are well known data structures and are not repeated here. The "head of list" is a record that contains two fields that refer to elements of a list. One field refers to the first element of the list and the other to the last element of the list. This makes the concatenation (+) of two lists represented by "head of list" records a simple operation that can be performed in $O(1)$ time.

The five important record types introduced in this report are the refcount, cellnode, celledge, celledge_end and cellgraph types, which are specified in the Tables 3.9, 3.10, 3.11, 3.12 and 3.13 respectively. If a field contains a reference to another type instead of the type itself, this is displayed by putting an asterisk (*) before the name of the type.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>N</td>
<td>Identifier of the boundary node represented by this record.</td>
</tr>
<tr>
<td>nr</td>
<td>N</td>
<td>Number of times this node occurs in the bnodes field of a celledge.</td>
</tr>
</tbody>
</table>

Table 3.9: Fields of a refcount record.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodes</td>
<td>head of list</td>
<td>Linked list of nodes that are in the cell.</td>
</tr>
<tr>
<td>nr_nodes</td>
<td>N</td>
<td>Number of nodes in the nodes list.</td>
</tr>
<tr>
<td>nr_edges</td>
<td>N</td>
<td>Number of internal edges in the cell.</td>
</tr>
<tr>
<td>nr_bnodes</td>
<td>N</td>
<td>Number of boundary nodes in the bnodes list.</td>
</tr>
<tr>
<td>bnodes</td>
<td>head of list</td>
<td>Double linked list of refcounts for the boundary nodes.</td>
</tr>
<tr>
<td>cedges</td>
<td>head of list</td>
<td>Double linked list of *celledges connected to this cell.</td>
</tr>
<tr>
<td>list_pos</td>
<td>*double linked list</td>
<td>Position of the cellnode in the cellgraph’s cellnodes list.</td>
</tr>
<tr>
<td>color</td>
<td>*celledge</td>
<td>Temporary storage of the &quot;color&quot; of the cell.</td>
</tr>
</tbody>
</table>

Table 3.10: Fields of a cellnode record.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nr_edges</td>
<td>N</td>
<td>Number of boundary edges in the celledge.</td>
</tr>
<tr>
<td>priority</td>
<td>Q</td>
<td>Result of the priority function for the celledge.</td>
</tr>
<tr>
<td>end1</td>
<td>celledge_end</td>
<td>Information about one endpoint of the celledge.</td>
</tr>
<tr>
<td>end2</td>
<td>celledge_end</td>
<td>Information about the other endpoint of the celledge.</td>
</tr>
<tr>
<td>queue_pos</td>
<td>N</td>
<td>Position of the celledge in the priority queue of the cellgraph.</td>
</tr>
</tbody>
</table>

Table 3.11: Fields of a celledge record.

Also the pseudo code used so far is extended with the statement "for all" to perform an operation on all elements of a set. The operations between the "for all" and "end for" statements are executed exactly once for each element of the set. Note that a set has no order and that therefore the result of the statements executed inside the loop should not depend on the order in which the elements are processed. For example:
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cell</td>
<td>*cellnode</td>
<td>The cell this celledge.end connects to.</td>
</tr>
<tr>
<td>bnodes</td>
<td>head of list</td>
<td>Linked list of *refcounts, referring to the boundary nodes in cell, to which the boundary edges in the celledge are connected.</td>
</tr>
</tbody>
</table>

Table 3.12: Fields of a celledge.end record.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cellnodes</td>
<td>double linked list</td>
<td>List of all cellnodes in the partition.</td>
</tr>
<tr>
<td>celledges</td>
<td>priority queue</td>
<td>Priority queue containing the celledges.</td>
</tr>
<tr>
<td>nr_cells</td>
<td>N</td>
<td>Number of cells in the partition.</td>
</tr>
<tr>
<td>nr_celledges</td>
<td>N</td>
<td>Number of elements in celledges.</td>
</tr>
<tr>
<td>nr_nodes</td>
<td>N</td>
<td>Total number of nodes in the partition.</td>
</tr>
<tr>
<td>value</td>
<td>Q</td>
<td>Value of the selection function for the partition.</td>
</tr>
</tbody>
</table>

Table 3.13: Fields of a cellgraph record.

for all \( i \in V \rightarrow \\
\{ \text{DO SOMETHING WITH } i \} \\
\{ \text{DO NOT ASSUME THAT } j \text{ WHERE } j \neq i \text{ AND } j \in V \text{ HAS OR HAS NOT BEEN PROCESSED} \}
end for

Furthermore, a new function is introduced called \( \delta \). This function takes a reference to a celledge and a cellnode and returns a reference to the cellnode on the other side of the celledge. i.e.:

\[
\delta(\text{celledge}(A, B), A) = B \\
\delta(\text{celledge}(A, B), B) = A
\]

Finally the function \( \rho \) is introduced. This function takes a reference to a celledge and a cellnode and returns a celledge.end of that celledge:

\[
\rho(\text{celledge}(A, B), A) = \text{celledge}(A, B).\text{end1} \lor \rho(\text{celledge}(A, B), A) = \text{celledge}(A, B).\text{end2} \\
\rho(\text{celledge}(A, B), B) = \text{celledge}(A, B).\text{end1} \lor \rho(\text{celledge}(A, B), B) = \text{celledge}(A, B).\text{end2}
\]

Not only that, but the celledge.end record that is returned by the function is such that \( \rho(\text{celledge}(A, B), A).\text{cell} = A \) and \( \rho(\text{celledge}(A, B), B).\text{cell} = B \). This means that the function returns the celledge.end that contains the information about the endpoint of the celledge connected to the cell.

### 3.5.2 Pseudo-Code

Algorithm 2 displays the main function of the program. The main change with the pseudo-code from Section 3.1 is that this code also shows how the algorithm makes multiple runs and makes sure that nothing is saved in the first run.

There are some procedures mentioned in the algorithm that are not described in detail because they are very simple. We describe them briefly here. One of these procedures is the SeedRandomGenerator call. That function seeds the random generator with a specific seed
Algorithm 2 Merging algorithm (cellgraph CellGraph)

1: {CELLGRAPH CONTAINS THE INITIAL PARTITION}
2: Start := CellGraph; {SAVE STARTING CONFIGURATION}
3: Best := CellGraph; {SAVE BEST CONFIGURATION SO FAR}
4: bestvalue := CellGraph.value;
5: run := 0;
6: found := FALSE; {VARIABLE INDICATING IF A PARTITION WAS FOUND IN LATER RUNS}
7:
8: {MAKE MULTIPLE RUNS}
9: do (run ≠ TOTAL_RUNS) →
10: run := run + 1;
11: 
12: {GET THE START CONFIGURATION}
13: CellGraph := Start; {LOAD STARTING CONFIGURATION}
14: if ((run = 1) ∨ ((run = TOTAL_RUNS) ∧ ¬found)) →
15: SeedRandomGenerator; {SEED RANDOM GENERATOR WITH THE SAME SEED}
16: else
17: CellGraph.Shuffle; {RECALCULATE PRIORITIES}
18: fi
19:
20: {TAKE THE CONFIGURATION THROUGH A RUN}
21: do (CellGraph.nr.celledges ≠ 0) →
22: Choice := CellGraph.SelectAB();
23: CellGraph.Merge(Choice);
24: if CellGraph.value < bestvalue →
25: bestvalue := CellGraph.value;
26: if (run ≠ 1) →
27: Best := CellGraph;
28: found := TRUE;
29: fi
30: fi
31: od
32: od
in order to make it possible to repeat the first run later on. The CellGraph.SelectAB function just takes the celledge that is on top of the priority queue and returns it. Finally the CellGraph.Shuffle function recalculates the priority function for every celledge. This is done to make the algorithm more random, by reshuffling all celledges in the priority queue. This last function is used because the data structure at the start of each run is the same, so the results of the priority function, which have been stored in the celledges, are also the same. And since the value is only updated for celledges connected to a newly merged cell, the runs would be similar. By using CellGraph.Shuffle to recalculate the priority function similar runs are prevented.

Procedure 2 also contains some procedures that are not described in detail. One of them is CellGraph.RemoveFromQueue, which removes a celledge from the priority queue. The function CellGraph.UpdatePriorities(C) recalculates the priorities of every celledge connected to cell C. CellGraph.RemoveCell and CellGraph.AddCell respectively remove and add a cell to the collection of cells of the cellgraph. The CellGraph.LowerReferenceCount(CE) in Procedure 3 lowers the reference count of boundary nodes associated with celledge CE. Should a reference count reach zero then the number of boundary nodes of the cell is updated. Finally, the Destroy(X) function frees the memory reserved for X, making it available for other purposes.

The UpdateValue function calculates the value of the new partition based on the available information (as described in Section 3.1):

\[
\text{UpdateValue}() = \text{CellGraph.value} - (A_{\text{nr_nodes}} \times (2 - \frac{A_{\text{nr_nodes}}}{B_{\text{nr_nodes}}}) \times A \rightarrow \text{nr_edges}) - (B_{\text{nr_nodes}} \times (2 - \frac{B_{\text{nr_nodes}}}{C_{\text{nr_nodes}}}) \times B \rightarrow \text{nr_edges}) + (C_{\text{nr_nodes}} \times (2 - \frac{C_{\text{nr_nodes}}}{A_{\text{nr_nodes}}}) \times C \rightarrow \text{nr_edges}) - ((1 - \frac{A_{\text{nr_nodes}}}{C_{\text{nr_nodes}}})^2 \times \text{nr_bnodes}_A \times (\text{nr_bnodes}_A - 1)) - ((1 - \frac{B_{\text{nr_nodes}}}{C_{\text{nr_nodes}}})^2 \times \text{nr_bnodes}_B \times (\text{nr_bnodes}_B - 1)) + ((1 - \frac{C_{\text{nr_nodes}}}{C_{\text{nr_nodes}}})^2 \times \text{nr_bnodes} \times (C \rightarrow \text{nr_bnodes} - 1)) - \frac{1}{2} (\text{CE} \rightarrow \text{nr_edges})
\]

Note that \(\text{nr_bnodes}_A\) and \(\text{nr_bnodes}_B\) are variables that contain the original number of boundary nodes of cell A and B respectively. However, the function CellGraph.RemoveCellEdge calls CellGraph.LowerReferenceCount, which means the \(\text{nr_bnodes}\) field of the cells could have been updated before the objective value of the new partition is calculated. Since the calculation of the objective value requires the original values of these fields, they have been stored in the temporary variables \(\text{nr_bnodes}_A\) and \(\text{nr_bnodes}_B\).

### 3.5.3 Time Complexity

The time complexity of the algorithm can give an indication of the worst-case execution time. It is examined in this section. It is assumed that the time complexity of calculating the priority function and calculating the new value of the selection function are both \(O(1)\) operations.

**Theorem 3.5.1** Under the assumptions that calculating the new value of the selection function and calculating the priority function are operations of order \(O(1)\) the worst-case time
Procedure 2 CellGraph.Merge(celledge CE)
1: {INITIALIZE}
2: A := δ₁(CE) {A IS THE CELL ON ONE SIDE OF CE}
3: B := δ₂(CE) {B IS THE CELL ON THE OTHER SIDE CE}
4: C := New(cellnode);
5: nr_bnodes₁ₐ, nr_bnodes₁₉ := A → nr_bnodes₁, B → nr_bnodes₁;
6: 
7: {CALCULATE FIELDS OF NEW CELL}
8: C → nodes := A → nodes ++ B → nodes;
9: C → nr_nodes := A → nr_nodes₁ + B → nr_nodes₁;
10: C → nr_edges := A → nr_edges₁ + B → nr_edges₁ + CE → nr_edges₁;
11: CellGraph.RemoveCellEdge(CE);
12: C → nr_bnodes := A → nr_bnodes₁ + B → nr_bnodes₁;
13: 
14: {CONNECT CELLEDGES OF A AND B TO C AND USE THE COLORING ALGORITHM}
15: for all e ∈ A → cedges →
16:   ρ(e, A).cell := C;
17:   δ₁(e, C) → color := e;
18: end for
19: for all e₂ ∈ B.cedges →
20:   if (δ₁(e₂, B) → color ≠ nil) →
21:     e₁ := δ₁(e₂, B) → color;
22:     {MERGE e₂ INTO e₁ AND DESTROY e₂}
23:     e₁ → nr_edges := e₁ → nr_edges₁ + e₂ → nr_edges₁
24:     ρ(e₁, C).bnodes := ρ(e₁, C).bnodes₁ ++ ρ(e₂, B).bnodes₁
25:     ρ(e₁, δ₁(e₁, C)).bnodes := ρ(e₁, δ₁(e₁, C)).bnodes₁ ++ ρ(e₂, δ₁(e₂, B)).bnodes₁
26:     δ₁(e₂).cedges := δ₁(e₂).cedges₁ \ {e₂}
27:     δ₁(e₂).cedges := δ₁(e₂).cedges₁ \ {e₂}
28:     CellGraph.RemoveFromQueue(e₂);
29:   else
30:     ρ(e₂, B).cell := C; {CONNECT THE CELLEDGE TO THE NEW CELL}
31:   fi
32: end for
33: 
34: for all e ∈ A.cedges →
35:   δ₁(e, C) → color := nil;
36: end for
37: C.cedges := A.cedges₁ ++ B.cedges₁;
38: 
39: {UPDATE CELLGRAPH}
40: CellGraph.UpdatePriorities(C);
41: CellGraph.value := UpdateValue();
42: CellGraph.RemoveCell(A);
43: CellGraph.RemoveCell(B);
44: CellGraph.AddCell(C);
Procedure 3 CellGraph.RemoveCellEdge(CellEdge CE)

1: $\delta_1(CE).edges := \delta_1(CE).edges \setminus \{CE\}$
2: $\delta_2(CE).edges := \delta_2(CE).edges \setminus \{CE\}$
3: CellGraph.LowerReferenceCount(CE);
4: CellGraph.RemoveFromQueue(CE);
5: Destroy(CE→end1.bnodes);
6: Destroy(CE→end2.bnodes);
7: Destroy(CE);

complexity of the partitioning algorithm is $O(|E| + |N| \times M \times (B + \log |E|))$, here $M$ is the maximum number of neighboring cells, and $B$ the maximum number of boundary nodes any cell has during a run.

To prove this, we first take a look at all simple procedures that have not been described in detail in the previous section.

SeedRandomGenerator is a simple operation of order $O(1)$. So is SelectAB because we use a binary heap as priority queue. Shuffle recalculates the priority of every edge so it is an operation of order $O(\text{CellGraph.nr.celledges})$. But these are operations that occur outside a run, the more interesting ones are the ones that occur during a run as they are executed most often.

RemoveFromQueue removes a cellgraph from the queue. Because a binary heap is used this operation is of order $O(\log \text{CellGraph.nr.celledges})$. The order of the UpdatePriorities(C) function is $O(|C.edges| \times \log \text{CellGraph.nr.celledges})$, it updates the priority function of every celledge connected to cell C and modifies the priority queue accordingly. Recall that $NC(c)$ was defined as the function that returns the set of neighboring cells of cell c:

$$NC : \mathcal{P}(N) \rightarrow \mathcal{P}(P)$$
$$NC(c) = \{cell \in P | BE(c) \cap BE(cell) \neq \emptyset\}$$

The number of celledges to which a cellnode is connected, is exactly the same as the number of neighbors it has (since every celledge connects the cell to exactly one neighbor). In mathematical notation $|C.edges| = |NC(C)|$ meaning the order of the UpdatePriorities operation can also be expressed as $O(|NC(C)| \times \log \text{CellGraph.nr.celledges})$.

There is no need to order the cells of a cellgraph, and therefore they are stored in a double linked list. This means that the RemoveCell and AddCell are $O(1)$ operations. The order of LowerReferenceCount(CE) depends on the number of pointers to reference counts in the celledge. So this is $O(|CE.end1.bnodes| + |CE.end2.bnodes|)$.

Using this information it is now possible to determine the order of the parts of the algorithm that were described in more detail. We start with Procedure 3. The removal of a celledge from the cedge sets of its cells is implemented as the deletion of an element from a double linked list (i.e. an $O(1)$ operation). The order of the LowerReferenceCount and the RemoveFromQueue has been determined. Together they make the order of the RemoveCellEdge procedure $O(|CE.bnodes1| + |CE.bnodes2| + \log \text{CellGraph.nr.celledges})$. The destruction of the $CE \rightarrow end1.bnodes$ and $CE \rightarrow end2.bnodes$ lists is of the same order as LowerReferenceCount, so it does not contribute to the worst-case time complexity. Neither does destroying
celledges, because any memory used by a celledge is in one single block and it is thus an operation of order $O(1)$.

The order of the Merge procedure (Procedure 2) is more difficult to determine. Many operations are of $O(1)$ thanks to the implementation, but a few are not. The order of RemoveCellEdge was determined above and is $O(|CE.end1.bnodes| + |CE.end2.bnodes| + \log \text{CellGraph.nr.celledges})$. The UpdatePriorities($C$) procedure also has an order that has been determined before: $O(|NC(C)| \times \log \text{CellGraph.nr.celledges})$. The interesting parts are the three loops that determine whether a pair of celledges needs to be merged, and merge the celledges if necessary (lines 16–19, 20–34 and 35–37). The first and last loop (lines 16–19 and 35–37) are simple $O(|NC(A)|)$ calculations. The middle loop (lines 20–34) requires $|NC(B)|$ iterations. What happens in those iterations depends on the if-statement on line 21. In any case the body of the loop contains mostly $O(1)$ operations except for CellGraph.RemoveFromQueue (an $O(\log \text{CellGraph.nr.celledges})$ operation). The Destroy operation is $O(1)$ because only the memory used in the celledge needs to be freed. The end1.bnodes and end2.bnodes lists need not be freed as they are concatenated to the lists of the remaining celledge.

The order of every operation in the Merge procedure is known, all that is left to do is to combine them to get the order of the entire procedure. Since we assume a worst-case scenario here we will assume the condition of the if-statement always evaluates to true. This makes the order of the entire procedure:

$$O(|CE.end1.bnodes| + |CE.end2.bnodes| + \log \text{CellGraph.nr.celledges} + |NC(C)| \times \log \text{CellGraph.nr.celledges} + 2 \times |NC(A)| + |NC(B)| \times \log \text{CellGraph.nr.celledges})$$

We can simplify it a bit and remove the constants. This results in:

$$O(|CE.end1.bnodes| + |CE.end2.bnodes| + |NC(A)| + (|NC(C)| + |NC(B)|) \times \log \text{CellGraph.nr.celledges})$$

It can be further simplified by noting the relationship between $|NC(C)|$, $|NC(A)|$, $|NC(B)|$ and the number of times the guard of the if-statement evaluates to true (say $i$). This relationship is: $|NC(C)| + i = |NC(A)| + |NC(B)| - 1$. This is caused by the fact that the number of celledges connected to $C$ is the total number of celledges connected to $A$ and $B$ minus the celledges that are destroyed in the if-statement and minus the celledge between $A$ and $B$. In the worst-case scenario $i = |NC(B)|$ meaning that $|NC(C)| = |NC(A)| - 1$. This means the order of the Merge procedure can be written as follows:

$$O(|CE.end1.bnodes| + |CE.end2.bnodes| + (|NC(A)| + |NC(B)|) \times \log \text{CellGraph.nr.celledges})$$

Further reduction is not easily possible, since it is hard to say anything about the number of reference counts of a celledge and about the number of neighboring cells of a cell. It is not unreasonable to assume that these values are quite low. After all, a partition with a lot of boundary nodes is not a good partition. Also the number of neighboring cells of a cell should not be very high. When envisioning the cells in a honeycomb structure each cell has
6 neighbors. Of course there will be cells with more, and cells with fewer neighbors, but it gives a good indication that the value will not be very high.

Now it is time to look at the time for an actual run. The saving operation is actually the most costly operation (having to iterate through each cell and celledge). In practice it is executed only a few times because we first calculate an estimate and then use it to reduce the number of times a cellgraph is saved. But in the worst-case scenario the save operation could be executed for each encountered partition. However, it is possible to make sure that the save operation is only executed once. This can be accomplished by not only saving the value of the best partition, but also the state of the random number generator at the start of the run that lead to that partition. That can be done for every run, then at the start of the last run the saved state can be used to make exactly the same run and the saved value can be used to find the same partition. This way the save operation is executed only once. The reason that this is not put into practice, is that the reduced saving method already reduced the number of saving operations enough to make their contribution to the processing time very small compared to other operations.

So that leaves the Merge operation as the most expensive operation for each iteration. We assume that \( L \) represents the maximum number of elements in a reference count list of any celledge during the run. And that \( M \) represents the maximum number of neighboring cells any cell has during the run. That way the worst-case Merge operation has an order of \( O(L + M \times \log \text{CellGraph.nr.celledges}) \). The maximum number of elements in a reference count list is a bit awkward quantity. Luckily it can be expressed in a more easily understood number. A single cell has at most \( M \) celledges. So in the worst case all of these celledges could be merged into one by mergings between cells neighboring that particular cell. Assuming that all these celledges share the same boundary nodes the final remaining celledge will have \( M \times b \) elements in its reference count list, where \( b \) is the number of boundary nodes of the cell. Assuming that \( B \) is the maximum number of boundary nodes a cell has during the run the worst-case order of the Merge operation is \( O(M \times (B + \log \text{CellGraph.nr.celledges})) \).

After each iteration of the Merge procedure the number of cells in the cellgraph has decreased by one. The number of cells the algorithm starts out with is at most \( |N| \) and the maximum number of celledges it starts out with is \( |E| \). This makes the actual run of the order \( O(|N| \times M \times (B + \log |E|)) \). What remains is the Shuffle operation and the loading of the starting configuration that happen at the start of each run. The shuffling is \( O(|E|) \) and the loading consists of loading each cell and each celledge which results in \( O(|N| \times |E|) \). The data associated with each cell and celledge can be loaded in \( O(1) \) operations for the starting configuration.

Therefore an entire run is an operation of order \( O(|E| + |N| \times M \times (B + \log |E|)) \). The last run also contains a saving operation of \( O(|N| + |E|) \), but this still leaves the order of an entire run at \( O(|E| + |N| \times M \times (B + \log |E|)) \).

Since the total number of runs is constant and the initialization takes less time than a run the complexity of a single run is also the time complexity of the entire algorithm. So, the worst-case time complexity of the partitioning algorithm is:

\[
O(|E| + |N| \times M \times (B + \log |E|))
\]

If we assume that \( B \) and \( M \) are small constants, the average time complexity of the entire algorithm is \( O(|E| + |N| \times \log |E|) \). That \( M \) does not fluctuate much is quite reasonable. A
3.5. Final Algorithm

cell is likely not to have too many neighbors when a normal map is used. For $B$ this is a bit harder, but the priority function is trying to steer the algorithm in such a way that it finds good partitions and good partitions do not have many boundary nodes.
Chapter 4

Theoretical Quality

In this chapter the theoretical quality of output of the merging algorithm is examined and ways are found to improve it. With theoretical quality we mean the value the selection function assigns to the partition found by the partitioning algorithm. In other words, with the theoretical quality we mean the objective value of the found partition.

To achieve this goal several methods are used. First of all the representation of the route graph is changed, so that it contains fewer edges. Increasing the number of runs and altering the priority function are other methods used to increase the quality. And finally, an attempt is made to find an upper bound for the quality of a partition.

4.1 Stars instead of Cliques

In her paper "Using Turn Restrictions for Faster Route Planning with Partitioned Road Networks" [11], Flinsenberg describes how it is possible to store all shortest routes through a cell by introducing an imaginary node in the center of the cell. The boundary nodes are connected to this center node instead of to each other. The length of the routes can then be stored using traffic rules to introduce costs between every pair of edges.

This greatly reduces the number of edges introduced in the boundary graph for a partition.

Figure 4.1: Example of the route graph when using the Clique and the Star structure
(See Figure 4.1). The clique structure requires a total of \( n(n - 1) \) edges to store all routes between \( n \) boundary nodes. The star structure only requires \( 2n \) edges in such a case.

Also, a clique structure requires \( n(n-1)(n-1) \) turn restrictions to prevent the concatenation of routes through a cell. These restrictions are necessary to prevent the planning of forbidden routes [11]. The star structure requires only \( n(n+1) \) turn restrictions. Note, however, that the turn restrictions in the clique structure are simpler, they represent forbidden turns, while most of the turn restrictions in the star structure only introduce an extra cost. These simple turn restrictions can be hard-coded into the planning algorithm, leaving no turn restrictions in the clique structure and \( n(n-1) \) turn restrictions in the star structure.

This new approach changes the expected number of edges in the search graph, but note that the idea of the objective value still works. The objective value represents the expected number of edges in the search graph. Because the search graph is now constructed in a different way, the formula for the objective value differs.

The \( R \) term in the new formula is:

\[
R(P) = \sum_{c \in P} \left(1 - \frac{|c|}{|N|}\right)^2 \min\{NB(c) \times (NB(c)-1), 2 \times NB(c)\}
\]

meaning that the number of edges added to the graph to represent the routes is either based on the Clique structure or the Star structure, whatever produces the smallest route graph. For cells with 3 or fewer boundary nodes the route graph for a clique is the same size or smaller than for a star, that is why these cells get a route graph based on the clique structure. From now on the default selection function will be the function that calculates the expected number of edges in the search graph based on stars.

The advantage of this approach is that with the star structure a cell can be much smaller. There do not have to be as many internal edges in a cell (compared with the clique structure), to weigh up to the number of route edges, since the number of route edges is greatly reduced.

### 4.2 Runs

The graph decomposition algorithm is randomized, so it is possible to increase the number of runs it performs. Hopefully using more runs produces a better result. To see if this works the value of the best partition after each run was plotted for 1,000 runs. Figure 4.2 shows the resulting plots. The experiment was repeated twice, to determine the effect of the seed of the random number generator.

The results show that it is impossible to tell how many runs are needed to find the best or a near optimal solution. For some maps a good solution is found very early, with only minor improvements in the subsequent runs. But for other maps a big improvement is found after more than 700 runs.

The first run of the algorithm on the Netherlands graph produces a partition that has a value of approximately 30,000 and the partition after 1,000 runs has a value of approximately 29,500. This is means that in 999 extra runs a reduction of 3.3% is achieved. Of course the change is considerable, but not compared to the processing time required to get it. If there is a way to improve the quality, there is probably a faster way than increasing the number of runs.
4.3 Priority Functions

Since more runs does not help in finding better partitions, we now turn to the priority function. As has been said before the priority function steers the algorithm. By altering it, it is possible for the algorithm to encounter better partitions, or to encounter only bad partitions.

Clearly a good priority function is very important, and in this section we will describe the priority functions we tried. Some give better results, while some produce worse results.

4.3.1 Different Functions

The original priority function was given, but there are of course many other possibilities. It is quite clear that there is no direct connection between the default priority function and the selection function. I.e. the priority function has not been derived from the selection function in some manner. Therefore it seems obvious to use a priority function that does have a relationship with the selection function. The priority function, for example, that returns the improvement of the objective value given by the merging of a celledge. This way
the algorithm always chooses the best merge and performs it, hopefully leading to the best partition, although this makes the algorithm very greedy. We also experimented with some other priority functions such as a totally random function. And there were experiments with removing the extra factor that was added to ensure that the size of the cells remains somewhat the same.

Another thing we experimented with, is guiding the merging process by giving ferries a low priority. In this experiment the priority function returned 0 when a cell edge contains an edge with the ferry property. The reason we tried this is depicted in Figure 4.3. An experiment generated Figure 4.3, which shows a solution for the Netherlands which includes a cell that contains Vlieland and Terschelling (two islands), a part of Friesland and a part of Noord-Holland. The islands are only connected to Friesland by a ferry and Friesland and Noord-Holland are only connected by a single bridge called the Afsluitdijk (see Figure 4.3). Although it might be a good solution, intuitively we would expect cells with few boundary edges between them. So we would expect the ferries and bridge to be ideal boundaries between cells.

Figure 4.3: A strange cell

Below is a list of the priority functions that are used and a name that is used to refer to them. The cell edge for which the function is calculated is denoted by $E$ and the cells it connects by $A$ and $B$, the resulting cell is denoted by $C$. The $\text{random}(a, b)$ function randomly selects a number between $a$ and $b$. 
4.3. Priority Functions

Normal = \frac{1}{|A| \times |B|} \times \frac{|BE(A) \cap BE(B)|}{|A| + |B|} \times \text{random}(1, 1.01)

No Size = \frac{|BE(A) \cap BE(B)|}{|A| + |B|} \times \text{random}(1, 1.01)

Random = \text{random}(0, 1)

Ferry = \begin{cases} 
\text{Normal} & \text{iff E does not contain a ferry} \\
0 & \text{Otherwise}
\end{cases}

\text{BNodes} = \frac{|BE(A) \cap BE(B)| \times (|BN(A)| + |BN(B)| - |BN(C)| + 1)}{|A| \times |B|} \times \text{random}(1, 1.01)

\Delta \text{Value} = \text{Objective value after merge} - \text{Objective value before merge}

Note that the \Delta \text{Value} function does not contain a random factor, therefore each run with this priority function will produce the same result. Experiments with a priority function like \Delta \text{Value} that does contain a random factor lead to no significant improvements.

After the initial experiments, a more constructive approach was used. To keep the value of a partition low three things are necessary: small cells, few boundary edges between cells and few route edges. The last factor can be accomplished by creating cells with few boundary nodes. The Normal function includes the first two, but not the last factor. Also it contains the need for small cells twice, by dividing by the product and the sum of the cell sizes. So the \text{BNodes} function was introduced. It includes each factor exactly once. The number of boundary nodes between cells that disappear from the search graph when the edge is selected for merging is multiplied by the number of boundary nodes that disappear in the merging plus one. We add one to the number of boundary nodes to ensure that if no boundary nodes disappear, the edge still can have a priority other than 0 (note that the number of edges between cells cannot be 0 for a cell edge). The $|A| \times |B|$ factor is to ensure that cells of approximately the same size are constructed.

4.3.2 Results

All maps were partitioned by the algorithm using 10 runs. The best objective value after those runs is shown in Table 4.1. The reason only 10 runs are done, is because it is not only desirable to have a function that produces a good result, but the function should also produce this result in as few runs as possible. The random function, for example, will produce the optimal result eventually, but the required number of runs is just too large.

The functions in Table 4.1 are roughly ordered from the ones that produce the best result to the ones that produce the worst result.

The value of the Netherlands graph for the Random function has not been measured, since the processing time required for the program was too large. This was because the reduced saving, by skipping saving in the first run has no effect with this function. So in the subsequent runs the save operation is executed very often. This could be solved by saving only once as described in Section 3.5.3, but we did not expect that the result would get anything near the results obtained with the other priority functions.
### Theoretical Quality

<table>
<thead>
<tr>
<th>Function</th>
<th>Sophia</th>
<th>Eindhoven</th>
<th>Siegen</th>
<th>Wetzlar</th>
<th>Antwerp</th>
<th>Netherlands</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNodes</td>
<td>412</td>
<td>2,284</td>
<td>2,335</td>
<td>5,096</td>
<td>7,040</td>
<td>28,468</td>
</tr>
<tr>
<td>Ferry</td>
<td>411</td>
<td>2,345</td>
<td>2,457</td>
<td>5,295</td>
<td>7,348</td>
<td>29,036</td>
</tr>
<tr>
<td>Normal</td>
<td>411</td>
<td>2,345</td>
<td>2,457</td>
<td>5,295</td>
<td>7,348</td>
<td>29,238</td>
</tr>
<tr>
<td>No Size</td>
<td>421</td>
<td>2,791</td>
<td>2,639</td>
<td>6,166</td>
<td>9,408</td>
<td>52,468</td>
</tr>
<tr>
<td>ΔValue</td>
<td>626</td>
<td>3,372</td>
<td>3,343</td>
<td>8,866</td>
<td>10,745</td>
<td>57,046</td>
</tr>
<tr>
<td>Random</td>
<td>1,358</td>
<td>14,615</td>
<td>22,121</td>
<td>80,899</td>
<td>91,389</td>
<td>NA</td>
</tr>
</tbody>
</table>

Table 4.1: Results with different priority functions

The ΔValue function does not perform well. By taking the best improvement each time, the algorithm gets stuck in a local optimum and from which it can never reach a good partition.

The Ferry function produces slightly better results when using a map that actually contains ferries. But the improvement is not big. Generally there will be very few ferries in a map and thus the Ferry function has very little impact on the end result. Exceptions might be the maps with many ferries like maps of Scandinavia for example. But on such maps the influence of the Ferry function could become too large, preventing the merging of cells that should be merged. Also, it would be best if the partition can be found without using extra information from the map. Once we start incorporating information from the map in the priority function, more and more information will be added (giving a low priority to bridges for example). Also once it turns out that different parts of information need different weights, it will become hard to tune the function. Therefore we would like to keep the priority function based on information that can be extracted from the graph itself (i.e. the nodes, edges and turn restrictions), instead of other attributes.

The BNodes function produces the best results. In fact, in 10 runs it produces a better result (for all maps except Sophia) than was obtained with the 1,000 runs from Section 4.2. For the map of the Netherlands the function produces a partition whose value is about 1,000 smaller than the best partition found in the 1,000 runs.

#### 4.3.3 Further Experimentation

When visually inspecting the Netherlands map that is partitioned with the BNodes function it contains some cells that are obviously not optimal. The partition contains a few very small cells, and it is better to merge them with some larger cells. In order to repair this some further experiments are conducted.

The partition resulting from the algorithm with the BNodes function is used as the starting configuration of another run, but now with ΔValue as the priority function. This method was called the BNodes2 method. So the BNodes2 method consists of a number of runs to get the best partition with the BNodes priority function, followed by a final run with the ΔValue function. This final run does not use the normal starting partition, but instead uses the best partition found in the earlier runs as the starting partition. The partitions produced with this new method do not have the small cells that are present in the partition found with the BNodes method.
4.4. Limits

The results of this new method are in Table 4.2. Since the BNodes2 method works nicely the performance of this method was also measured with 1,000 runs.

<table>
<thead>
<tr>
<th>Method</th>
<th>Sophia</th>
<th>Eindhoven</th>
<th>Siegen</th>
<th>Wetzlar</th>
<th>Antwerp</th>
<th>Netherlands</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNodes2:1,000+1 runs</td>
<td>393</td>
<td>2,215</td>
<td>2,261</td>
<td>4,947</td>
<td>6,760</td>
<td>27,853</td>
</tr>
<tr>
<td>BNodes2:10+1 runs</td>
<td>412</td>
<td>2,232</td>
<td>2,321</td>
<td>5,024</td>
<td>6,989</td>
<td>28,010</td>
</tr>
<tr>
<td>BNodes:10 runs</td>
<td>412</td>
<td>2,284</td>
<td>2,335</td>
<td>5,096</td>
<td>7,040</td>
<td>28,468</td>
</tr>
</tbody>
</table>

Table 4.2: Results of further experimentation

Obviously the BNodes2 method can find a good solution in only a few runs. The 990 extra runs only caused an improvement of less than 200 edges. So while the 1,000 runs produce a better result we do not think it is necessary to spend so much processing time on such a little improvement.

An interesting question is whether there is a better point to start using the ΔValue function. Maybe a better partition can be found if the BNodes function has not reached its optimum partition yet.

However, the current results are quite good and we do not expect that it will be easy to improve them. Intuitively there is a minimum objective value for each graph; the objective value represents the average size of the search graph. A graph that contains only 28,000 edges and can be used to plan a route through the Netherlands is quite small compared to the total number of edges in the map. So, before we keep on experimenting it is desirable to know how close we are to the optimum. When we know that, it is possible to decide whether it is reasonable to keep trying to improve the priority function.

4.4 Limits

It is certain that there is an optimum partition for each graph. But since the partitioning problem is NP-hard [10] it is hard to calculate this optimum in reasonable time. Still, it is nice to know how close the approximations are to the real optimum. One can keep trying to improve the heuristic method, but without knowing the real optimum it is hard to determine when to stop.

So a way to compare the result of the heuristic algorithm with the actual optimum of a graph is needed.

4.4.1 Quadratic Programming

Hurkens [15] advised us to look at quadratic programming. If the problem can be written in a quadratic programming form, then the optimum can be calculated.

A quadratic programming problem is a maximization problem over the vector $x$ of the form:

$$\max\{p^t x - \frac{1}{2} x^t C x | Ax \leq b; x \geq 0\}$$

where $p \in \mathbb{R}^n$, $A$ is a $m \times n$ matrix, $b \in \mathbb{R}^m$ and $C$ is a $n \times n$ matrix such that $x^t C x > 0$ for every $x \neq 0$. 
Note that the partitioning problem is a discrete problem so it is likely that there will be some variables that indicate whether a node is in a particular cell (1) or not (0). But quadratic programming works with real numbers, so the optimal partition could turn out to have a node that is (for example) for 31% in cell A and 69% in cell B. This means that the solution to the quadratic programming version of the partitioning problem will yield an upper bound for the quality, which is a lower bound for the objective value. So to find the optimum partition some way has to be found to enforce the variables to become either 0 or 1, or to find the discrete optimum by using the solution from the quadratic programming problem. Even if that is not possible, having a lower bound for the objective value is still very useful.

But there is a problem, take a look at the formula for the objective value of an entire partition:

$$\text{objective value} = \sum_{c \in P} \frac{|c|}{|N|}(2 - \frac{|c|}{|N|})|E(c)| +$$
$$\sum_{c \in P} (1 - \frac{|c|}{|N|})^2 \min\{|NB(c)|(|NB(c)| - 1), 2|NB(c)|\} +$$
$$\frac{1}{2} \sum_{c \in P} |EB(c)|.$$ 

It is clear that there is no one-on-one translation from this formula to the formula for quadratic programming, at least not if $|c|$ and $|NB(c)|$, or $|c|$ and $|E(c)|$ are variables. The function that is maximized in quadratic programming contains the sum of several multiplications. Such a multiplication contains at most two variables. Since the first and second term of the objective value formula contain a multiplication of at least three variables there is no easy translation.

So we want to try to remove either $|c|$ or $|NB(c)|$ and $|E(c)|$ as variables.

We could also assume that the size of each cell ($|c|$) is given as input for the problem and then solve the problem for every possible input. But that would mean solving a lot of quadratic programming problems, since number of possible combinations of cell sizes for a graph of reasonable size is huge. A similar combinatorial explosion occurs when we try to solve the quadratic programming problem for every possible $|NB(c)|$ and $|E(c)|$. So this approach does not work.

We could also assume that all cells have an equal size, and then solve the problem for every possible number of cells. But then the result would become a lower bound of the best possible quality (which is an upper bound for the objective value). This is because the real optimum does not need to have cells of exactly equal size. And the goal was to measure the quality of the partitioning algorithm with this result. But we already have lower bounds for the optimum quality; the partitions we found with the merging algorithm also present lower bounds. So this approach would mean a lot of work to get something we already have, which is only useful if the approach produces better lower bounds. We decided to continue our research in another direction, instead of gambling that this approach would produce good bounds.

### 4.4.2 Known Optima

Perhaps it is possible to construct graphs for which the optimum is known. In that case the value of the partition found by the algorithm can be compared with the optimum.
4.5. Conclusion

The proof that the partitioning problem is NP-hard contains graphs of which the optimum is known [10]. But these graphs are fully connected graphs with several loops on each node, they do not resemble roadmaps.

So far we have not been able to find a way to construct such graphs. The best partition depends on the entire structure of the graph. Starting with a small graph of which the best partition is known and then adding nodes and edges or entire cells seems like a good approach. But by connecting those new objects to the existing graph the optimal partition may change entirely. The problem is that local changes can have global consequences.

4.4.3 Brute Force

As a last option it is still possible to calculate the absolute minimum by trying all possible partitions. Since the problem is NP-hard this is very expensive, but for small graphs it may be feasible.

However, upon examination it turns out that the required processing time will be too high. Assuming that it is possible to examine $10^9$ partitions per second, then examining all possible partitions for a graph containing only 50 nodes will take a long time. First the best possible partition when splitting the graph into two cells has to be calculated. So there are two possible cells to place each node in, resulting in $2^{50} \approx 10^{15}$ possible combinations. Symmetric combinations can be discarded, and so can combinations leading to a partition with 1 cell, leaving $\frac{1}{2} \times 2^{50} - 1$ partitions to check. Obviously this takes quite long: about 3.5 days. After that every possible partition when using 3 cells has to be examined ($\frac{1}{6} \times 3^{50} - \frac{1}{2} \times 2^{50} - 1$ partitions). Then for 4 cells, 5 cells, etc.

So the brute force method will not work either, since examining all possible partitions for a graph with $n$ nodes is $O\left(\frac{n^n}{n!}\right)$.

For large $n$ we can use Stirling's formula to simplify this a bit:

$$\lim_{n \to \infty} \frac{n^n}{n!} = \lim_{n \to \infty} \frac{n^n}{\sqrt{2\pi n} \left(n^{1/2} \right)e^{-n}} = \lim_{n \to \infty} \frac{1}{\sqrt{2\pi n} e^{-n}}.$$

So for large $n$ the brute force method is $O(e^n)$, which is still too large to be practically feasible.

4.5 Conclusion

So far it is not possible to determine the value of the best partition of a graph. This means there is no way to say something about the quality of the approximation algorithms. At least not about how close the algorithm gets to the optimum. The only way to get an indication is to compare it to other approximation algorithms.

In our opinion the BNodes2 method described in Section 4.3 is close to what can be reached by just altering the priority function. Experiments with other priority functions were also conducted besides the ones being described there. These experiments included increasing the random factor of an existing function, or replacing $\times$ signs by $+$ signs and vice versa, or squaring some of the factors. The results from these experiments did not improve the found
partitions and they were not very interesting. Therefore they have been left out. But enough experiments have been done to determine that significantly improving upon the BNodes2 method will be no easy task.

Determining the optimum partition of a graph would be very interesting, but it may be better to save it as a separate research project. For now we can continue with the knowledge that we have an approximation algorithm that works pretty well.
Chapter 5

Practical Quality

The theoretical quality of a partition is the value given to it by the selection function. By default the selection function returns the expected number of edges in the search graph. The practical quality of a partition is the average number of expansions required to plan a route. So far, we have assumed that there is a relationship between those two. This chapter focuses on that relationship.

5.1 Theory and Practice

The objective value of a partition is used to indicate its quality. It is equal to the expected size of the search graph when planning a route. But what actually interests us, is the number of expansions the route planning algorithm has to do to plan a route.

The objective value so far is an upper bound for this number of expansions, but the actual relationship between the two has yet to be examined. This is interesting, because it can put the objective value and improvements thereof into perspective. We expect the relationship to be linear, but it would be nice if it turns out to be better (logarithmic for example). That would mean that small improvements to the quality do not really matter and it is only necessary to look for large improvements. Since the best partition of the Netherlands (so far) already has a value of approximately 28,000, we do not expect that any big improvements are possible.

So this relationship has to be examined. The easiest way to do this, is to take the partition, plan a number of random routes and then look at the number of expansions that were done. However, before this can be done, the partition has to be made ready for planning, i.e. the optimal routes through each cell have to be calculated. A utility has to be implemented to do this.

5.1.1 Implementation

One problem was encountered; the graphs can contain rules between edges that have no node in common.
Figure 5.1: Example of a rule between edges that have no node in common

An example of this is a crossroad where it is not allowed to make a U-turn (see Figure 5.1). The edges B and C will have a rule between them that indicates it is not possible to plan a route via those two edges. If the restriction would be modeled by using two rules saying it is not allowed to turn left from C to A and to turn left from A to B, then somebody approaching the crossing from the right would not be allowed to take a left turn too. So it is necessary to have rules that have no node in common.

But the input graphs only contain the edge on which the rule starts and the edge on which the rule ends. It does not contain the direction of the rule on the starting edge. Therefore it is not known in which direction the rule holds. This problem is ignored, there is no question that it can be solved, but solving it causes a lot of work (like another variant of the input file format). Furthermore the probability that such a rule happens to be in a route is fairly small. Table 5.1 shows the number of rules that have no node in common for each map.

<table>
<thead>
<tr>
<th>Map</th>
<th>&quot;Disconnected&quot; Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sophia</td>
<td>0</td>
</tr>
<tr>
<td>Eindhoven</td>
<td>6</td>
</tr>
<tr>
<td>Siegen</td>
<td>1</td>
</tr>
<tr>
<td>Wetzlar</td>
<td>13</td>
</tr>
<tr>
<td>Antwerp</td>
<td>13</td>
</tr>
<tr>
<td>Netherlands</td>
<td>246</td>
</tr>
</tbody>
</table>

Table 5.1: Number of "disconnected" rules in each map

So the problem has very little impact. Therefore it is ignored in the course of this research. A product version of the tool will has to solve it though.

### 5.1.2 Route Planning

The goal is to measure the number of expansions required for planning a route. But the number of expansions depends on the algorithm that is used for planning. To see what works best several different algorithms are used: Dijkstra’s algorithm [6], the A* algorithm [13] and a pruning algorithm.

The pruning algorithm was designed by Flinsenberg [10] and uses some extra information about the search graph: if all outgoing edges of the starting cell have been expanded, then the edges in the cell can be pruned. A pruned edge will not be examined by the algorithm
when it encounters such an edge. This can be done because at that point the shortest path from the source to every exit of the cell is known. Continuing expansions inside the cell will not yield any paths that lead out of the cell and are shorter that the paths currently leaving the cell. The same trick can also be used when all paths into the destination cell are known, then everything in the candidate list, except the edges in the destination cell, can be pruned.

Note that the pruning algorithm can be combined with other algorithms such as Dijkstra’s algorithm or the A* algorithm. It is quite likely that Dijkstra’s algorithm finds all edges leading out of a cell earlier than the A* algorithm. Other algorithms may also be combined with the pruning algorithm, but we have only combined the pruning algorithm with Dijkstra’s algorithm and the A* algorithm.

The Netherlands graph was partitioned using the BNodes2 method in 10 runs. After that 2,000 routes were planned on the partitioned graph. Each route was planned twice, once to find the fastest route and another time to find the shortest route. The results are shown in Figure 5.2. The graphs show, for each route, the size of the search graph and the corresponding number of expansions required to plan the route.

Figure 5.2: 2,000 routes planned through the Netherlands with 4 different algorithms

Note that the number of expansions sometimes exceeds the size of the search graph. This is caused by the fact that bidirectional edges are counted as a single edge, while they may have to be expanded twice (once in one direction and once in the other direction). In each graph a trend line has been drawn. When choosing for polynomial or logarithmic regression it is possible to draw a slightly curved line that suggests that the number of expansions required for bigger graphs becomes less and less. However, this curve is very small and a linear interpolation is more appropriate.

Note that the pruning algorithms do not perform much better than their non-pruning counterparts. Flinsenberg noted that the pruning algorithm works best when the cells are small. The graph used in the experiment contained fairly large cells, hence the small difference. On average the number of expansions that could be omitted by pruning was less than 100.
Therefore it might be better to use just the A* algorithm, instead of wasting time on pruning the candidate list and detecting whether this is necessary.

So the A* algorithm is used to examine the relationship between the average search graph size and the average number of expansions required to plan a route.

### 5.1.3 Examination of the Relationship between Theory and Practice

Several partitions were generated using a modified version of the partitioning program. Instead of saving the best partition, each encountered partition that fitted an empty category was saved. The categories used were 28,500, 29,000, 29,500, 30,000, ..., 60,000. The first partition whose objective value differed less than 50 from such a category was used as the representative of that category. Using each of the partitions 2,000 routes were planned with the A* algorithm using both the speed and distance criterion. The results can be found in Figure 5.3.

![Relation between objective value and quality](image)

> Figure 5.3: Average number of expansions required to plan 2,000 routes through partitions found in five runs

Each shade of gray represents a run of the algorithm. It is clear that there is a relation between partitions generated in the same run. The diamonds and squares indicate the average number of expansions when planning a route on the time criterion, and the distance criterion respectively.

The graph clearly shows that the best partition is not the partition with the lowest objective value. Also there is no linear relation between the objective value and the average number of expansions, at least not for objective values between 20,000 and 40,000.

The reason the graph is shaped like this, is that the A* algorithm has trouble leaving the start cell. About 60% of all expansions are performed in the start cell. This is because all
edges in the start cell have about the same estimated cost to the destination, while edges in the boundary graph have relatively large costs. When the partition is not optimal yet (according to the objective value), its cells are small, since there still are some mergings to do. But because they are small the A* algorithm can leave them faster. The number of expansions gained by small cells weighs up to the number of expansions lost in the larger boundary graph.

In an attempt to get more accurate results the partitioning program was modified such that a bi-directional edge is counted as two edges and routes generated by a uniform distribution were used. But the results were generally the same as those displayed in Figure 5.3 (see Figure 5.4). However, from now on, we will keep on counting bi-directional edges as two edges for the purpose of partitioning, and we will use the uniformly generated route set to plan routes, unless otherwise noted.

![Graph: Relationship between objective value and quality](image)

Figure 5.4: Average number of expansions required to plan 2,000 uniformly selected routes through partitions found in five runs

### 5.1.4 Conclusion

It is clear that the objective value is not a good measurement for the quality of a partition. This can be remedied in two ways; either the selection function is adjusted to include the size of the start cell, or an other route planning algorithm is found, one which divides its expansions equally over the start cell, boundary graph and end cell.

A possible other planning algorithm can be a bidirectional algorithm, i.e. an algorithm that starts planning at the source and at the destination. The symmetry would then cause the number of expansions inside the cells to be equally divided among the start and end cell. But the number of expansions in the boundary graph would still be unrelated to the number of expansions in the cells.
Note that if the selection function is altered, then it no longer represents the expected search graph size. It would then represent something like the estimated number of expansions.

5.2 Estimated Number of Expansions

The number of expansions required to plan a route cannot be calculated quickly. But it is possible to estimate it. We know the sizes of cells are the most important factor in the number of expansions, since 60% of the expansions take place inside a cell. This means that the size of the rest of the search graph does influence the number of expansions that much. So the weight of the rest of the search graph in the selection function should be reduced.

We assume the average route is 100 km long and that the area searched by the algorithm is an ellipse of which the foci are 100 km apart, because of the average route length. Furthermore we assume that for each point on the ellipse, the sum of the distances to the foci is $1.3 \times 100$ km (see Figure 5.5). The 100 km is an educated guess, and the factor 1.3 is based on the experience of Siemens VDO Automotive.

![Expected Search Area](image)

$AVG =$Average route length

$a + b = c \times AVG$

$c = 1.3$

Figure 5.5: The area of the map that is expected to be examined during route planning

Now, we can use the surface area of this ellipse compared to the surface area of the map to calculate the weight of the boundary graph. The surface area of the ellipse is $\frac{1}{4} \times \pi \times AVG^2 \times c \times \sqrt{c^2 - 1}$, where $AVG$ equals the average route length (100 km) and $c$ equals the factor that determines the detour (1.3). This results in a surface area of 9,159 square kilometers. The surface area of the Netherlands is 41,525 square kilometers. Therefore the weight of the boundary edges and route edges is $9159/41525 \approx 0.22$. This makes the new selection function: $S + \alpha(R + T)$ where $\alpha = 0.22$. This selection function is called the weighted objective value, or w-value for short. From now on we will use the weighted objective value as the selection
function, unless noted otherwise. If we mention the objective value we mean the original objective value the weighted objective value will be denoted by w-value.

Calculating the average route length dynamically would make α independent of a default route length. This would be a nice property, but it would also mean that each pair of cells has a different search area associated with it. This might lead to a better evaluation, but it would slow down the partitioning algorithm enormously. In each merging two cells disappear and a new one appears, each of those cells would have a relationship with every other cell in the partition and each of those relations would need to be updated. But the results with the constant route length are already very good (Figure 5.6).

![Relationship between w-value and quality](image)

Figure 5.6: Average number of expansions required to plan 2,000 routes through partitions with the w-value selection function

The formula for the trend line in the Figure 5.6 is \( y = 1.13x - 1853.6 \) and has an R-squared value of 0.9996. The R-squared value indicates how well the samples correspond to the trend line. A value of 0 indicates that there is no correlation and a value of 1 indicates that the samples are all on the trend line. So with an R-squared value of 0.9996 the w-value is clearly a better measure for the quality of a partition. Whenever the estimated number of expansions becomes lower, the average number of expansions decreases too.

Note that this selection function is based on an average route length. This means that for different route lengths, different partitions are needed if an optimal planning time is to be guaranteed. So calculating the average route length dynamically would be desired, but as noted before it is not possible to do that quickly.

There is a small difference between the minimum average number of expansions in Figure 5.4 and Figure 5.6. The best sampled partition with the objective value (Figure 5.4) had an evaluation value of 46,036 and an average of 14,371 expansions. And the best sampled partition with the w-value (Figure 5.6) had an evaluation value of 16,548 and an average of
15,664 expansions. So there is a difference of 1,293 expansions between the best samples. This means that the best partition is not found with the new selection function, but it is better than what was achieved with the objective value.

Note that the difference is quite large, but that is because samples were used. When we take a look at the best partition found with the w-value (which was not sampled), then the difference is smaller. The best partition found with the w-value has an average of 14,932 expansions. This differs only 561 expansions from the best known partition so far, which is still a little bit disappointing.

5.3 Examination of the State Space

To get a better insight into the role of each term of the objective value we decided to examine the state space of the algorithm. To do this several partitions were generated, but this time the sampling was done a bit differently. The starting partition was sampled. During the run, if $R$, $S$, $T$ (see Section 3.1) or the number of cells differed at least 1,000 from the previous sample, a new sample was taken. This resulted in samples that would give an overview of the development of each term during the run.

These sampled partitions are used to plan routes with. This gives a set of data for each partition that allowed examination and experimentation.

5.3.1 Development of the Terms

![Graph showing development of terms S, R, and T during a run](image)

Figure 5.7: Development of the terms $S$, $R$ and $T$ during a run
Figure 5.7 shows the value of $S$, $R$ and $T$ during a run. The horizontal axis contains the number of cells. Note that a run starts with many cells and ends with a few, so the graph should be read from right to left. Figure 5.8 contains a close-up view of Figure 5.7.

The $S$ term contributes very little at the beginning of the run, but near the end (when the number of cells is small) the cell size becomes very important. This is because small cells only have a small probability of being in the search graph and even if they do their impact is limited. But large cells have a large probability and have a large impact. This is reflected in the formula for $S$; the size of the cells, assuming that the number of nodes and number of edges are of the same order, has a cubic influence on $S$.

During the run the number of boundary nodes fluctuates and therefore the number of route edges does too. At the start the cells are small and have few boundary nodes, resulting in a small $R$. Mergings at the start also have a greater probability of eliminating boundary nodes. If cell $A$ and $B$ both have two boundary nodes then there is a large probability that two of them are only boundary nodes because there is an edge between $A$ and $B$. And therefore they will be eliminated in the merging, resulting in a new cell that has two boundary nodes. This causes the small decrease of $R$ at the start of the run. After that there is only a specific type of merging that can increase the number of route edges. When merging two cells that have a Star structure, even if there are no boundary nodes eliminated, the number of route edges does not increase. However when merging cells where one (or both) cells have a Clique structure, the number of route edges increases if no boundary nodes are eliminated. Remember that for small cells the Clique structure is more efficient than the Star structure and that the formula for $R$ assumes that the most efficient structure for each cell is used. So after the initial decrease $R$ remains fairly constant, as some merges eliminate boundary nodes and some do not. But after a while a point is reached where the eliminated boundary nodes
no longer reduce $R$ as much as the mergings with Clique structures in them increase it. So at that point the number of route edges starts increasing. Then, once the cells reach a certain size and most cells contain a Star structure, it is possible to find mergings again that reduce the number of boundary nodes and thus cause $R$ to decrease again.

Finally there is $T$, which represents the number of edges between the cells. This term steadily decreases as more boundary edges are turned into internal edges during each merging.

5.3.2 Development of the Selection Functions

![Graph showing development of objective value, w-value, and average number of expansions](image)

Figure 5.9: Development of the objective value, w-value and average number of expansions

Figure 5.9 shows the development of the objective value, the w-value (with $\alpha = 0.22$) and the average number of expansions needed to plan a route through the partition. The graph clearly shows that the w-value more closely resembles the number of expansions than the original objective value (see also Figure 5.3 and 5.4).

5.3.3 W-Value

With the available data it was possible to test different values for $\alpha$ in the w-value, without having to generate new partitions. It turned out that for $0.28 < \alpha < 0.40$ the w-value orders the partitions in the same way as the average number of expansions does. This means that the algorithm would find the best partition from among the samples.

Examining the set of routes that were used made clear that basing $\alpha$ on the average route length is not a bad idea. The routes had an average length of 122 km (instead of the 100 km we assumed). We assume the standard detour factor ($c$) of 1.3. The surface area of the ellipse representing the estimated search area would then be 12,623 square km. That would suggest $\alpha = \frac{12,623}{41,525} \approx 0.3$. Which upholds $0.28 < \alpha < 0.40$ and thus results in the algorithm finding the best, sampled, partition.
To test the w-value the same experiment was conducted with the map of Wetzlar. The results showed that for Wetzlar the value of $\alpha$ had to uphold $0.31 < \alpha < 0.62$ to ensure that the algorithm finds the optimum partition. The map of Wetzlar contains a small part of Germany with a surface area of only 4,500 square kilometers. The routes that were used for planning were uniformly selected and had an average length of 43 kilometers. This means that the estimated search area has a surface area of 1,568 square kilometers. Resulting in $\alpha = \frac{1568}{4300} \approx 0.35$, which means that this method of calculating $\alpha$ once again results in the algorithm finding the best, sampled, partition.

This means that it is desirable to calculate the average route length dynamically. That would remove the dependency of the value of $\alpha$ on the average route length and generate a partition that is very well suited for planning routes of different lengths.

But it is more likely that different partitions are better suited to different situations. The detour factor (c) for example depends on the type of planning that is performed. When looking for the shortest route c is generally smaller than what it would be when looking for the fastest route. So for a product version Siemens VDO will have to decide for which criterion and route length they want to optimize the partition.

5.3.4 Conclusion

The w-value seems to be the best selection function yet. But as Figure 5.9 shows the shape of the function resembles, but not quite matches, that of the number of expansions. Perhaps experimenting with separate factors for each of the terms S, R and T could lead to a selection function that does match. Finding such factors may be possible, but a manner to determine them using only the data available beforehand would be needed too. Because only then it would be possible to determine the best selection function for each particular graph.

From the similarity in the graph we can at least conclude that the three terms used to calculate the objective value and the w-value have a lot to do with the average number of expansions needed to plan a route. It is unlikely that any other data is required.

In any case, the w-value works very well and the value for $\alpha$ can be easily calculated. The surface area of the estimated search area can be calculated with $SEA = \frac{1}{4} \times \pi \times AVG^2 \times c \times \sqrt{c^2 - 1}$ where $AVG$ equals the average route length and $c$ equals the factor that determines the detour (we assumed $c = 1.3$). Divide the result with the surface area of the actual map ($SM$) and the resulting number can be used as $\alpha$ in the w-value selection function (i.e. $\alpha = \frac{SEA}{SM}$). Also the idea behind this selection function is easy to understand, and most important of all, it works in practice.

Further experiments in which finer samples are taken might be necessary to fine tune the w-value approach. We use the value 1.3 for c, but since the formula for $\alpha$ results in a value that is closer to the lower bound than to the higher bound, c may have to be adjusted.
Chapter 6

Additional Results

Some extra time was available after the research described in the previous chapters was completed. This time was used to write this report, conduct some further experiments and explore some subjects that would be placed on the list of "Future Research".

Although there was not enough time to explore these future research topics in detail, the results so far are very interesting. This chapter describes these topics and the results gained from experimenting with them.

6.1 Splitting Algorithm

Beside the merging algorithm, Flinsenberg also designed a splitting algorithm. This algorithm applies an approach opposite to that of the merging algorithm. It starts by placing all nodes in a graph in one large cell. Then, at each iteration, the largest cell is chosen and split into two cells. This process continues until each node is in a separate cell. The best partition encountered during this process is saved and used as the result of the algorithm.

The splitting of a cell is done by a splitting procedure. This procedure starts by moving one node of the old cell to a new cell. After that nodes in the old cell connected to nodes in the new cell are assigned a priority. Once these initialization steps are complete the procedure keeps repeating the same two basic operations until the original cell is empty. The first operation is moving the node with the highest priority from the old cell to the new one. And the second one is updating the priority of nodes in the old cell that are connected to the node which has just moved. The best configuration encountered during this process is selected as the final result of the splitting procedure.

The splitting algorithm needed a fast implementation too, just like the merging algorithm. Because the merging algorithm seemed to be faster and to produce better results, it was chosen as the main subject of this research. However, due to the time available the splitting algorithm could be further examined.

6.1.1 Implementation

With the experience of implementing the merging algorithm, implementing the splitting algorithm became somewhat easier. Many of the techniques used in the merging algorithm were
also used for the splitting algorithm.

The only new data structure that was used in the implementation was a union-find [4] [17]. The union-find was used to indicate the nodes that should not be separated. Recall from Section 3.3 that some nodes should stay in the same cell, to prevent a partition in which there are traffic rules between boundary edges and internal edges. In the merging algorithm this was solved by merging the cells containing nodes that needed to be in the same cell. In the splitting algorithm this is solved by moving nodes that need to remain together as a single cluster of nodes during the splitting procedure. Since these clusters are disjoint subsets of the set of nodes, a union-find is an ideal data structure to represent them. So when a node is moved all connected nodes are checked to see if they belong to the same cluster, if this is the case then they are moved too.

Aside from the union-find the only other data structures that were used were the binary heap, and basic data types like arrays, records and pointers.

The splitting procedure was designed in such a way that it is reversible. By keeping a history of moved nodes it is possible to backtrack and return to the best, encountered, split. This prevented the need to save the split each time a better one was encountered. Because this method allows us to avoid saving and loading entirely during the splitting procedure, it is often even more efficient than doing the split twice (once without saving, just to get the best value and once with saving, to get the best split).

### 6.1.2 Results

The final implementation required 4 minutes and 34 seconds to perform 5 runs. The objective value of the found partition of the map of the Netherlands was 58,687. This implementation is clearly slower (see Table 3.8) and produces results of worse (theoretical) quality than the merging algorithm (see Table 4.2).

It may be possible to make further improvements on the implementation, but there is a reasonably fast implementation to experiment with now. And, more importantly, there are more interesting research topics still open.

### 6.2 Point Structures

A way of improving the partitions is to use a different structure for storing the route graphs. The Clique and Star structures have been described in this report, but it is also possible to store the routes in a structure we call the Point structure. In a Point structure the entire route graph of a cell consists of only a single virtual node representing the entire cell. The routes are then stored with turn restrictions on the boundary edges of the cell.

The Point structure is shown in Figure 6.1, although there is not much to show. The idea is, as it is with the star structured route graph, that the routes are stored in turn restrictions. However, instead of creating route edges with turn restrictions between them, the turn restrictions are placed directly between the boundary edges of the cell. Since these boundary edges are (generally) not connected to each other the cell's boundary nodes are replaced by a single virtual node. All the boundary edges are connected to this virtual node and thus it is possible to plan routes with this new structure.
This means that no edges are necessary to store the routes, effectively eliminating the $R$ term of the selection function. There are some problems when planning routes with this approach, because it is necessary to know via which boundary node the route enters and leaves a cell. But that is something that can be solved.

On the other hand it is no longer necessary to avoid turn restrictions between boundary nodes and internal nodes, since these turn restrictions can now be incorporated in the turn restrictions between boundary edges.

Another advantage is that the selection function no longer depends on the number of boundary nodes of a cell. If a priority function can be found that does not either, it will be possible to increase the speed of the partitioning algorithm as it no longer needs to keep track of boundary nodes.

To be able to experiment with these structures a new file format was needed. The file format described in Appendix C is used for this. After the new libraries that use this new format had been implemented it was possible to experiment.

A partition was generated using the merging algorithm with the w-value selection function with a slight modification: the $R$ component of the w-value was always 0. As it turns out this partition has even smaller cells, because there are no longer any route edges in the boundary graph. In effect the number of edges in the cell no longer has to weigh up to the number of route edges of the cell. When the map of the Netherlands is partitioned this way, the resulting partition consists of 2,169 cells.

The new Point route graph structure was used to create the route graphs for this partition of the Netherlands. The result was that, on average, 7,791 expansions are required to plan a route. This is quite an improvement over the 14,547 expansions needed when planning the same, uniformly selected, set of routes on the best known partition with star structured route graphs. The reduction is achieved because the Point structure allows the planning algorithm to leave a cell and enter another one (or vice versa) in a single expansion.

Note that we used the w-value selection function to find the partition. But the value of $\alpha$ was optimized for use with star structured route graphs. It is very well possible that, when partitioning for point structured route graphs, a different value for $\alpha$ should be used to get the optimum partition. This, however, has not been investigated.
6.3 Multi-Level Partitions

After the new file format and libraries had been implemented, multi-level partitions were only a small step away.

Recall that 60% of all expansions during route planning take place in the start cell (see Section 5.1). This can be reduced by applying the method we have been using to reduce the number of expansions in the first place: graph partitioning. By recursively applying graph partitioning it is possible to reduce the number of expansions even further.

Once a graph has been partitioned, it is possible to take the detailed map of a cell and partition it again. This can be done for every cell that is large enough to be partitioned. This would create a tree of partitions with at the top the highest level partition consisting of only a few large cells. In effect a multi-level partition allows the removal of a lot more edges from the search graph than an ordinary single-level partition. However, the function for calculating the expected number of edges in the search graph becomes much more complicated. Also there are some questions to be answered, for instance: when is a cell large enough to be partitioned again? Do we start with finding the lowest level partition first, or the highest level?

There is enough to be studied on the topic of multi-level partitions. We have made an initial survey into the area by implementing the merging algorithm in such a way that it can be applied recursively and by allowing the planning library to plan on multiple levels. The route graphs of the multi-level partitions all have the Point structure, since that is the best structure we know so far. But they are partitioned as if they would have the Clique structure, since that results in large cells, which makes them better suited to partition them again.

6.3.1 Results

The results so far show that it is feasible to plan optimal routes in a car navigation system. The current route planner used by Siemens VDO requires on average 1,971 expansions to plan a route on the map of the Netherlands.

Table 6.1 contains the results for planning the same set of routes on a multi-level partition. Each row in the table describes the results for on of the approaches used to generate the multi-level partition. The "Max. Levels" and "Min. Edges" columns indicate the method used to generate the partition. The "Max. Levels" column indicates the maximum number levels that the partition has. And the "Min. Edges" column indicates the number of edges a cell must have before it is partitioned again. Note that this means that the tree of cells generated by the partitioning algorithm is not necessarily balanced.

The other columns of Table 6.1 provide information about the resulting partition. The "Cells" column indicates the total number of cells in the tree of partitions and the "Routes" column indicates the total number of routes through all these cells. The number of expansions in the "Expansions" column are the average for planning the routes on the distance and on the time criterion.

The rows in Table 6.1 are ordered from the best partitions to the worst. As can be seen the best partition requires the storage of a lot of routes. We expect that there is some balance between the number of routes that need to be stored and the number of required expansions. The partition in the first row of the table requires the storage of a lot of routes and is fast,
6.3. Multi-Level Partitions

<table>
<thead>
<tr>
<th>Max.Levels</th>
<th>Min. Edges</th>
<th>Cells</th>
<th>Routes</th>
<th>Expansions</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
<td>273,029</td>
<td>1.73 M</td>
<td>1,397</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>57,301</td>
<td>1.26 M</td>
<td>1,418</td>
</tr>
<tr>
<td>4</td>
<td>300</td>
<td>25,767</td>
<td>0.99 M</td>
<td>1,571</td>
</tr>
<tr>
<td>3</td>
<td>300</td>
<td>24,819</td>
<td>0.94 M</td>
<td>1,574</td>
</tr>
<tr>
<td>10</td>
<td>500</td>
<td>16,506</td>
<td>0.92 M</td>
<td>1,740</td>
</tr>
<tr>
<td>2</td>
<td>300</td>
<td>3,996</td>
<td>0.46 M</td>
<td>3,022</td>
</tr>
</tbody>
</table>

Table 6.1: Results when planning with multi-level partitions

but the partition in the second row requires considerably fewer routes while the number of expansions barely increases. When we look at the other rows we see that the rate at which the number of expansions increases is bigger than the rate at which the number of routes decreases.

Another thing that we noticed with the multi-level partitions was that the maximum number of expansions is about 4 times the average number. This is already very good compared to the Siemens VDO route planner, for which the maximum is about 50 times the average. This number is important, because it gives an indication of the worst-case running time. In a navigation system, that is something that needs to be minimized too.

6.3.2 Conclusion

The route planning can now be done fast enough to make it possible to use it in an actual car navigation system. However, another factor comes into play, the number of routes that have to be stored increases dramatically with multi-level partitions. Assuming that the cost of a route can be stored in 32-bits, storing the routes for the second partition in Table 6.1 costs about 5Mb. This does not seem very much, but it means 5Mb for every route planning criterion.

In short some more research into this area is desirable. Perhaps some better way of generating multi-level partitions can be found, one that keeps the number of expansions low, but also minimizes the number of routes that have to be stored. Also ways to store the partition will have to be examined, the flexfile of Appendix C works for this environment, but it was designed with extensibility and research in mind, more efficient formats may be possible for the navigation system.
Chapter 7

Conclusion

During our research many aspects of graph partitioning for route planning were discovered, but there is still more to examine. This chapter will provide a short summary of this report and gives the conclusions that have been drawn from the work that has been done. It also contains an overview of possible further research.

Graph partitioning can be used to speed up route planning, while maintaining the ability to plan optimal routes. The partitioning algorithm defined by Flinsenberg, called the merging algorithm, can be used to find a graph partition. One part of the assignment was to find a fast implementation for this algorithm. This report shows (in Chapter 3) the final implementation of the algorithm and the steps that were taken to get there.

The other part of the assignment concerned the quality of the partitions found with the algorithm. This presented a problem, because there are different types of quality measures for a partition. The quality we would like to use is the average processing time required to plan a route. But since the processing time is hard to calculate and depends on a lot of external factors, we used the average number of expansions instead. So, the average number of expansions required to plan a route was used as the measure of quality for a partition. But to get this number we needed to plan routes on the partitioned graph. This takes too much time. In order to speed up the partitioning algorithm we used a selection function to indicate the value of a partition. Ideally this selection function is easy to calculate and accurately reflects the quality of a partition.

In Chapter 4 ways were discussed to improve the value of the selection function for partitions produced by the algorithm. A method was found that produces good results, and it is hard to improve upon those results. But that chapter also explained that it is not possible to determine how close the value of the selection function is to the optimum value.

In Chapter 5 the relationship between the selection function and the quality of a partition was examined. It turned out that the original selection function, called the objective value, does not correspond well with the quality. Therefore a new selection function was designed, based on the objective value, and it was called the w-value function. The w-value function corresponds better to the quality of a partition.

Finally, in Chapter 6, we made an initial survey into further research topics. As that chapter shows it is possible to use graph partitioning in order to speed up the route planning such that it only needs to examine 1,000 edges (on average) to plan a route in a map of the Netherlands.
The results in every chapter were obtained by implementing the libraries and programs required to get them. Some programs and utilities were already available when the project started, but to get an efficient implementation we decided to build everything from the ground up. The interfaces of these libraries and programs can be found in Appendices C and D.

7.1 Speed

One part of the assignment was to improve the speed of the original partitioning algorithm. There was already an implementation available, but it took 20 minutes to partition the map of Eindhoven. The final implementation of the design presented in this report takes only a few seconds to partition that map. And it can partition the map of the Netherlands in less than 3 minutes, while it took days with the initial implementation.

We think that it will be hard to find a design or implementation that is much faster than the one presented in this report. But it must be noted that it may be possible to decrease the running time if there is no need to keep track of the boundary nodes during the run. I.e. if the selection and priority function do not depend on the number of boundary nodes of a cell, then keeping track of them is no longer necessary. The objective value and w-value functions when using the point structured route graphs are examples of such selection functions.

However, for the current selection and priority functions the program is fast enough. The reason Siemens VDO is interested in a fast partitioning algorithm, is that they plan to produce maps every month or every two months. A lot of time is spent on converting and testing the maps; if partitioning is going to be included it needs to take a minimum amount of time. The final algorithm will not increase the production time for a map in any significant way, therefore there is no need for further improvement of the algorithm’s running time.

7.2 Quality

Another part of the assignment was to improve the quality of the partitions found by the partitioning algorithm. This can be accomplished by altering the selection and priority function of the algorithm.

As shown in Chapter 5 it is now possible to calculate a selection function (the w-value) that selects the optimum partition from a few samples. Since samples were used it is not certain that the selection function will actually select the best partition encountered in a run, but it is very likely to get a partition that is close to that optimum. But there is still no guarantee that the algorithm actually encounters the optimum partition, or even a near-optimal partition. It may be possible, with a different priority function, to steer the algorithm toward even better partitions.

The trouble is that we have no way of calculating (the value of) the actual optimum partition. With the optimum value it would be possible to determine the quality of the priority and selection functions and of the algorithm itself. Now all we have to guide us are the results we get from the algorithm. Each time one of the functions is tweaked, the results can be compared with previous results. Since we have only been able to achieve small improvements we assume that we are close to the optimum, but unfortunately that is not certain.
So, it is hard to tell if there is any room for improvement. Until a way is found to calculate (an approximation of) the real optimum, or to find a better partition this remains unchanged.

In any case, the partitions currently found by the algorithm speed up planning enormously compared to planning in the unpartitioned graph. But to increase the planning speed even more it may be better to use multi-level partitions as described in Chapter 6, instead of trying to optimize the single-level partitions.

7.3 Further Research

As this report has shown there is much more research on graph partitioning for route planning possible. There are always more selection and priority functions to experiment with. Other route planning algorithms can be tried, like bi-directional planning. Or maybe a planning algorithm can be designed that makes better use of the properties of a partitioned graph. We are also very interested in finding the optimum partition for a graph, or at least the value of the optimum partition, so we have a solid way to evaluate the partitioning algorithm and its selection and priority functions.

Some of the possible topics have already been briefly examined in Chapter 6. The splitting algorithm is an example of such a topic. It presents an other way of finding a partition. Some experiments have been done with this algorithm and it seems the partitions found by it are of a lower quality than the ones found by the merging algorithm. However, more extensive experimentation may prove otherwise.

An interesting topic would be to see if the splitting and merging algorithm could be combined, resulting in an algorithm that can backtrack. With such an algorithm it would be unnecessary to use multiple runs, the algorithm is capable of examining every possible partition.

And the Point structure and multi-level partitions are, of course, topics that deserve further investigation. These methods make it possible to examine fewer edges during route planning than the current Siemens VDO route planner does, while planning optimal routes.

Before these methods can be used in practice further research is necessary. We have been using the number of expansions as a measure of speed, but the expansions in a partitioned graph may be more expensive than the ones in the current graphs. This is because the number of turn restrictions in the partitioned graph is much higher than with the original graph. On the other hand, with the graph partitioning approach, there is a possibility that the entire search graph can be cached, since it contains only a few thousand edges. CD or DVD access is something that costs a lot of time and accounts for a significant part of the route planning time in the current route planner, but it could be avoided by caching the entire search graph. With the graph partitioning approach this is possible, not only because the search graph is small, but also because its cells are known before the actual planning starts. Because of this we expect that the graph partitioning approach will eventually plan routes faster.

The new route planning method requires the storage of more data. How this data can be effectively stored is a separate research topic entirely. The cost of a lot of routes through cells may be the same, the fastest route and the fastest route without toll roads through a cell for example. For many cells these routes are the same and storing them in an effective manner reduces the needed storage space. Perhaps one can store the route cost of only a few criteria,
providing optimal route planning for these criteria and near-optimal route planning for the others.

The partitions generated in Section 6.3 were only the first attempts at generating multi-level partitions. Further research may provide better ways of generating them. The goal then becomes not to improve the planning speed, but to keep the planning speed balanced with the amount of information that has to be stored. Also, we have been focusing on the average number of expansions, but the worst-case number of expansions is also very interesting. Being able to guarantee that the driver gets any route within a small time period would be a nice feature.

Before graph partitioning can be actually used in a car navigation system a lot of further research will have to be done. In any case this report has shown that graph partitioning is a feasible approach for route planning in car navigation systems.
Appendix A

Notation

This appendix gives an overview of the important notation and functions used in this report. The report uses common graph and set notations, like $\delta_1(e)$ and $\delta_2(e)$ to denote the end nodes of an edge $e$ and $\mathcal{P}(X)$ to denote the power set of a set $X$ (the set containing all subsets of $X$).

In addition some other notations are introduced. Table A.1 contains an overview of the variables. Functions on cells are described in Table A.2 and functions on partitions are described in Table A.3.

<table>
<thead>
<tr>
<th>$G$</th>
<th>The directed graph that is being partitioned $G = (N, E)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>Set of nodes in the graph</td>
</tr>
<tr>
<td>$E$</td>
<td>Set of edges in the graph</td>
</tr>
<tr>
<td></td>
<td>$(\forall e \in E : \delta_1(e) \in N \land \delta_2(e) \in N)$</td>
</tr>
<tr>
<td>$P$</td>
<td>A partition of graph $G$</td>
</tr>
<tr>
<td></td>
<td>$P \subseteq \mathcal{P}(N)$ such that:</td>
</tr>
<tr>
<td></td>
<td>$\bigcup_{c \in P} c = N$</td>
</tr>
<tr>
<td></td>
<td>$(\forall c_1, c_2 \in P : c_1 \cap c_2 = \emptyset \lor c_1 = c_2)$</td>
</tr>
<tr>
<td></td>
<td>$\sum_{c \in P}</td>
</tr>
<tr>
<td>$PS$</td>
<td>The set of all possible partitions of graph $G$</td>
</tr>
<tr>
<td></td>
<td>(Used in Table A.3)</td>
</tr>
</tbody>
</table>

Table A.1: Variables used in this report
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$BE$</td>
<td>The set of boundary edges of a cell $\mathcal{P}(N) \rightarrow \mathcal{P}(E)$ $BE(c) = { e \in E</td>
</tr>
<tr>
<td>$IE$</td>
<td>The set of internal edges of a cell $\mathcal{P}(N) \rightarrow \mathcal{P}(E)$ $IE(c) = { e \in E</td>
</tr>
<tr>
<td>$BN$</td>
<td>The set of boundary nodes of a cell $\mathcal{P}(N) \rightarrow \mathcal{P}(N)$ $BN(c) = { v \in c</td>
</tr>
<tr>
<td>$BNR$</td>
<td>The set of boundary nodes of one cell that connect to nodes in the other cell $\mathcal{P}(N) \times \mathcal{P}(N) \rightarrow \mathcal{P}(N)$ $BNR(c_1, c_2) = { n \in BN(c_1)</td>
</tr>
<tr>
<td>$NC$</td>
<td>The set of neighboring cells of a cell $\mathcal{P}(N) \rightarrow \mathcal{P}(P)$ $NC(c) = { cell \in P</td>
</tr>
</tbody>
</table>

Table A.2: Functions on cells used in this report
| **evaluate** | The selection function that evaluates a partition  
|  | evaluate : \( PS \rightarrow Q \) |
| **obj.val** | Default selection function  
|  | \( obj\_val(P) = S(P) + R(P) + T(P) \) |
| **w.val** | A selection function based on the objective value.  
|  | It reflects the quality of the partition more accurately than \( obj\_val \).  
|  | It calculates the expected number of edges in the search graph,  
|  | but the weight of the boundary graph is \( \alpha \) instead of 1.  
|  | \( w\_val(P) = S(P) + \alpha(R(P) + T(P)) \)  
|  | \( \alpha = \frac{SEA}{SM} \)  
|  | \( SEA = \frac{1}{4} \times \pi \times AVG^2 \times c \times \sqrt{c^2 - 1} \)  
|  | where \( AVG \) equals the average route length,  
|  | \( c \) the detour factor (1.3), and  
|  | \( SM \) the surface area of the map. |

**S**  
Contribution of the cell sizes to the expected number of edges  
in the search graph.  
\( S(P) = \sum_{c \in P} \left(\frac{c^4}{|W|} \times (2 - \frac{|c|}{|W|}) \times |IE(c)|\right) \)

**R**  
Contribution of the route edges to the expected number of edges  
in the search graph.  
Note that this function has two forms:  
One is used when using Clique structures:  
\( R(P) = \sum_{c \in P} ((1 - \frac{|c|}{|W|})^2 \times |BN(c)| \times (|BN(c)| - 1)) \)  
The other is for Star structures:  
\( R(P) = \sum_{c \in P} (1 - \frac{|c|}{|W|})^2 \min\{NB(c) \times (NB(c) - 1), 2 \times NB(c)\} \)

**T**  
Contribution of the boundary edges to the expected number of edges  
in the search graph.  
\( T(P) = \frac{1}{2} \sum_{c \in P} (|BE(c)|) \)

---

Table A.3: Functions on partitions used in this report
Appendix B

Proofs

This appendix contains the proofs that were too large to include in the main text.

B.1 Merging of Disconnected Cells

The proof that merging cells with no boundary edges between them does not decrease the objective value function was left out of Section 3.2.2. The proof rests on the assumption that the cells have at least as much internal edges as route edges. Because the point of partitioning is to get a smaller search graph by creating cells with route graphs that are smaller than the detailed maps this assumption is acceptable.

Recall that:

\[ S(P) = \sum_{c \in P} \left( \frac{|c|}{|N|} \times (2 - \frac{|c|}{|N|}) \times |IE(c)| \right) \]

\[ R(P) = \sum_{c \in P} ((1 - \frac{|c|}{|N|})^2 \times |BN(c)| \times (|BN(c)| - 1)) \]

\[ T(P) = \frac{1}{2} \sum_{c \in P} (|BE(c)|) \]

\[ obj\_val(P) = S(P) + R(P) + T(P) \]

Assuming that \( P' = (P \setminus \{A, B\}) \cup \{C\} \), \( A \in P \), \( B \in P \) and \( C = A \cup B \), it is possible to prove that the objective value does not increase if cell \( A \) and \( B \) are disconnected, if \( IE(A) \geq BN(A) \times (BN(A) - 1) \) and \( IE(B) \geq BN(B) \times (BN(B) - 1) \). Because the cells are disconnected we have \( BE(A) \cap BE(B) = \emptyset \).

A few shorthands will be used in the proof: \( a = \frac{|A|}{|N|} \), \( b = \frac{|B|}{|N|} \), \( c = \frac{|C|}{|N|} \), \( d = |IE(A)| \), \( e = |IE(B)| \), \( f = |IE(C)| \), \( g = |BN(A)| \), \( h = |BN(B)| \) and \( i = |BN(C)| \).

Note that \( c = a + b \), since \( C = A \cup B \). Because cells contain at least one node and cannot contain more nodes than those in the graph it holds that \( 0 < a < 1 \), \( 0 < b < 1 \) and \( 0 < c \leq 1 \). Because there are no boundary edges between \( A \) and \( B \) that become internal edges after the merge, the number of internal edges of the new cell \( C \) is the sum of the internal edges of \( A \) and \( B \) \( (f = d + e) \). Also \( i = g + h \) because the number of boundary nodes
does not change by merging $A$ and $B$, after all, since there are no boundary edges that turn into internal edges, there are no boundary nodes that turn into internal nodes.

We start by simplifying the formulas that indicate the change in the terms of the objective value. These formulas have been introduced in Section 3.1, all that we are doing here is rewriting them:

\[
S(P') - S(P) = -\frac{|A|}{N} \times (2 - \frac{|A|}{N}) \times |IE(A)| - \frac{|B|}{N} \times (2 - \frac{|B|}{N}) \times |IE(B)|
+ \frac{|C|}{N} \times (2 - \frac{|C|}{N}) \times |IE(C)|
= -a(2-a)d - b(2-b)e + c(2-c)f
\]

\[
R(P') - R(P) = -((1 - \frac{|A|}{N})^2 \times BN(A) \times (|BN(A)| - 1))
-((1 - \frac{|B|}{N})^2 \times |BN(B)| \times (|BN(B)| - 1))
+((1 - \frac{|C|}{N})^2 \times |BN(C)| \times (|BN(C)| - 1))
= -((1-a)^2g(g - 1))
-((1-b)^2h(h - 1))
+((1-c)^2i(i - 1))
\]

\[
T(P') - T(P) = -(|BE(A) \cap BE(B)|)
= -(|\emptyset|)
= 0
\]

So, since $T(P') - T(P) = 0$, the total change in the objective value can be written as:

\[
obj\_val(P') - obj\_val(P) = S(P') - S(P) + R(P') - R(P) + T(P') - T(P)
= S(P') - S(P) + R(P') - R(P)
= -a(2-a)d - b(2-b)e + c(2-c)f
-((1-a)^2g(g - 1))
-((1-b)^2h(h - 1))
+((1-c)^2i(i - 1))
\]

We will now rewrite the formula that we want to prove (that the change is always greater than 0), assuming that each cell has more internal edges than it has route edges. Or, in more formal notation: $d \geq g(g - 1)$ and $e \geq h(h - 1)$. 
B.1. Merging of Disconnected Cells

\[-a(2-a)d - b(2-b)e + c(2-c)f\]
\[-((1-a)^2g(g-1))\]
\[-((1-b)^2h(h-1))\]
\[+(1-c)^2i(i-1) \geq 0\]
\[-a(2-a)d - b(2-b)e + c(2-c)(d+e)\]
\[-((1-a)^2d)\]
\[-((1-b)^2e)\]
\[+(1-c)^2i(i-1) \geq 0\]
\[-a(2-a)d - b(2-b)e\]
\[+(2-c)d\]
\[+(2-c)e\]
\[-((1-a)^2d)\]
\[-((1-b)^2e)\]
\[+(1-c)^2i(i-1) \geq 0\]

\[(c(2-c) - a(2-a))d\]
\[+(c(2-c) - b(2-b))e\]
\[-((1-a)^2d)\]
\[-((1-b)^2e)\]
\[+(1-c)^2i(i-1) \geq 0\]

\[(c(2-c) - a(2-a) - (1-a)^2)d\]
\[+(c(2-c) - b(2-b) - (1-b)^2)e\]
\[+(1-c)^2i(i-1) \geq 0\]

Since it is easy to see that \((1-c)^2i(i-1) \geq 0\) it only remains to be proven (because of symmetry) that \(c(2-c) - a(2-a) - (1-a)^2 \geq 0\).

\[c(2-c) - a(2-a) - (1-a)^2 \geq 0 \iff\]
\[c(2-c) - (-a^2 + 2a) - (a^2 - 2a + 1) \geq 0 \iff\]
\[c(2-c) + a^2 - 2a - a^2 + 2a + 1 \geq 0 \iff\]
\[c(2-c) + 1 \geq 0 \iff\]
\[-c^2 + 2c + 1 \geq 0 \iff\]
\[-c^2 + 2c \geq 0 \iff\]
\[-c + 2 \geq 0 \iff\]
\[c \leq 2\]

Since \(c \leq 1\), we now that the change is always positive. So the objective value does not increase when merging two cells that have no boundary edges between them and have more internal edges than route edges (which should be a common property for cells). This means that there is no use in merging disconnected cells and therefore memory can be saved by leaving celledges.
To prove the same for partitions that use Star structure for their route graphs, instead of Clique structures, the same proof can be used. In fact, the proof holds for any kind of route graph structure, as long as the number of route edges of a cell is never more than the number of internal edges of a cell.
Appendix C

Flexfile

During the assignment it became clear that there was need for a new file format that could handle new features. There were already a few file formats available, but they did not allow functionality needed later on. For example, using the Point structure instead of a Clique or Star structure presents the possibility to reduce an entire cell to a single node. But it has to be possible to find the boundary nodes belonging to the edges leading toward, or leaving, this single node. Also none of the original file formats is suited for storing multi-level partitions.

To solve this problem it was decided to design a file format with the future requirements in mind. The examples above illustrate the shortcomings of the original formats. It is likely that new data needs to be stored in the future. Therefore the new file format has to be flexible enough to allow such extensions.

C.1 Requirements

The requirements for the new file format are as follows:

1. The format must be expandable.
2. The format must be suited for large graphs.
3. The format must be suited for multi-level partitions.
4. A program must be able to load a file quickly.
5. Files should be readable by a human using a text editor.

The requirement that the file format has to be readable poses a problem. Making a file readable in a simple editor means it has to be in some kind of ASCII format. But that makes it harder to implement requirements 2 and 4.

Instead of choosing some trade-off requirement 5 is dropped. It may be convenient to be able to examine a file manually, but it is not necessary to do so very often. A binary file format is more suited for implementing requirements 2 and 4. And it is possible to make a file readable by writing a utility that converts the binary format to a text format. If editing is necessary, a utility that reverses the transformation can also be written.
C.2 The Basics

There are already several standards available on the web for storing graphs. Most of them are XML [3] based. However, XML is difficult to implement and requires a lot of disk space.

So we developed a new file format called *flexfile*. The general idea is to store a description of the file at the start of the file (called the *header*), followed by the actual data. The description contains several *types*. These types describe how a program should read the different parts of the data that come after the header. A type description consists of a *base type* and some additional information depending on the base type.

The possible base types are:

**Fixed Size** Data of this type requires a standard number of bytes. Extra information: the size in bytes.

**Variable Size** Data of this type can vary in size. Extra information: none.

**Array** Data of this type consists of a list of a variable number of elements of another type. Extra information: the type of the elements.

**Record** Data of this type consists of a collection of fields, each with a unique name, containing an element of another type. Extra information: name, type and volatility (defined later in this appendix) of each field.

Records and arrays are types that can contain data of another type. This allows arrays of arrays, arrays of records, records containing records, etc. In fact the data in a flexfile consists of only one record called the *main record*. Reading the file boils down to reading the header, followed by reading the main record.

Say we want to store an address book in a flexfile for example (see also Table C.1). Suppose the main record type is for example Address Book. That record can have the fields Creator and Addresses for example. The field Creator is of type String, a variable size type. The field Addresses is of type array of Address, which is (as the name suggest) an array of elements of type Address. The Address type is a record with one field called Name and that field is of type String. We will ignore the volatility for now.

<table>
<thead>
<tr>
<th>Name of type</th>
<th>Base Type</th>
<th>Extra information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Book</td>
<td>Record</td>
<td>Field: Creator of type String (volatile)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Field: Addresses of type array of Address (non-volatile)</td>
</tr>
<tr>
<td>String</td>
<td>Variable Size</td>
<td>None</td>
</tr>
<tr>
<td>array of Address</td>
<td>Array</td>
<td>Elements of type Address</td>
</tr>
<tr>
<td>Address</td>
<td>Record</td>
<td>Field: Name of type String (non-volatile)</td>
</tr>
</tbody>
</table>

Table C.1: Example: An address book

Records are the only types that can be expanded with additional fields. It is possible to add new fields to a record and still have an old program read the file without any problems. That is why the Address type in the example is a record, it allows us to expand our example to
C.3. Flexfile Library

contain more information than only an address. It can be expanded to contain a field E-Mail of type String for example, to contain the e-mail address of the person in the address book. Also a new program that expects some fields but does not encounter them in the file can decide to assume a default value. This default value is not in the file, but since the new program recognizes the field it is assumed it also knows the default value of the field should it be missing. The E-Mail field in the example may have the default value "Unknown" for example.

The volatility of a field indicates how a program should react if it does not recognize that field. A field is either volatile or not. If it is volatile and the program does not recognize the field, it means the program should discard the information if it alters any data in the file. Non-volatile means that the program can save the field again, even if it changes something else in the file. In the example volatility it is not a big deal, but we could envision an example where a flexfile contains a graph and there is a field that indicates whether or not there is a Hamilton cycle in the graph. When a program removes an edge from the graph the field may have to be updated. If the program does not know the field it should just discard it to prevent the field from containing incorrect information.

In our example the Creator field of the main record is volatile. So, should a program add a new address to the file and it does not recognize the Creator field then it should discard the Creator field. This prevents the Creator field from becoming invalid because two persons edit the same file. A program that does recognize the Creator field can decide what to do. It could decide to add the name of everybody that edits the file to the Creator field for example.

C.3 Flexfile Library

To read or write a file, a program can use the flexfile library, which contains a parser for flexfiles. Appendix D.5 contains the detailed description of the interface of the flexfile library. From now on, we consider flexfiles, so when referring to the format of a file we mean the format of the data in the flexfile. As a running example we use the address book example presented in Appendix C.2.

C.3.1 Loading a Flexfile

This section explains how a flexfile can be loaded by a program using the library. To explain this we discuss the loading of an address book from a flexfile. Loading a flexfile consists of three steps. First the header is loaded, returning the format of the file. The program then has the opportunity to assign callback functions to read each type and to indicate to the library which fields it can recognize. After that the file is actually loaded.

The first thing to do, is to load the header of the flexfile. This can be accomplished with format_ptr = FLEXFILE_load_header(file_ptr);. This function returns the format of the flexfile "file.ptr".

After the header has been read the layout of the data in the file is known. The program can now specify to the library how it wants to read that data. This is done by assigning callback functions like this:
For most base types two callback functions can be assigned, one to be executed before reading data of a specific type and one to be executed after the data has been read. So the above statements tell the library to execute the function example_read_string before reading data of type String. The NULL indicates that after reading a String no function has to be called. The callback function for the record does a similar thing, but it does it for the Address type instead of the String type. Note that it is possible to indicate three callback functions for a record. The first is executed before reading the record and the last is executed after reading a record. The second callback function is called by the parser to pass data to the program that is in the file, but that the program does not understand. When reading a record the parser gathers the data in all fields that the program does not recognize. Once all fields have been read by the parser it uses the second callback function to pass this data to the program, followed by a call to the last callback function to indicate the end of the record.

Calls like void FLEXFILE_known_field(format_ptr, "Address", "Name"); can be used to indicate to the library which fields the program can recognize. This specific statement indicates to the library that the Name field of an Address record is recognized by the program.

The callback functions passed to the library have different formats depending on the type they are supposed to handle and whether they are executed before or after reading. In any case the first four parameters of all callback functions have the same type and fulfill the same purpose. These parameters are: const char *, const char *, long, void *. The first parameter points to the name of the type being read. While a different function can be assigned for each type it is desirable to minimize the number of functions by letting one function be capable of reading multiple types. The second parameter is the name of the field being read, in case the data is in a record, otherwise it is NULL. The third parameter is a number indicating the index of the data being read in the array (in case of reading an array) or the id of the field being read (in case of a record). The fourth and last parameter is a pointer that can be used by the program to store some information.

A note on writing callback functions: instead of using strcmp to compare the type or field names it may be easier to compare the char * pointers with the pointers stored in the FLEXFILE_FORMAT_STRUCT. This will speed up the program.

Going back to our example, we take a look at the callback function for reading a string. Its format is:

```c
void *example_read_string(char *type_ptr, char *field_ptr,
                           long index, void *extra_ptr,
                           size_t size);
```

This callback function has an extra parameter "size". This parameter indicates the size of the String that will be read by the parser. The parser expects the function to return a pointer to the place this data must be stored.
So the function in our example should allocate some memory and return a pointer to that memory. But that pointer should somehow be saved so the program can use it later on. If we assume that extra_ptr points to a structure representing an address (the address currently being read) this is easy. We just store the pointer in the structure referenced by extra_ptr. To make sure extra_ptr points to the structure representing the address currently being read we need the help of other callback functions.

The callback function for reading a record has a similar format:

```c
void *example_start_address(char *type_ptr, char *field_ptr,
  long index, void *extra_ptr);
```

It does not have the "size" parameter. The parser does not really expect anything of this function. The program can use it to allocate some memory for the upcoming record. But the return value of this callback function will be passed as the fourth parameter of all callback functions that are called to read fields of the record.

So we can use this callback function to make sure the extra_ptr parameter points to an address structure, when the parser calls the example_read_string function. Assuming that the extra_ptr of this function points to an array of address structures we can simply use return extra_ptr[index]. Making sure that the extra_ptr of this function points to an array of address structures is then delegated to the callback function that is called before reading a record of type Address Book in a similar way.

As illustrated in the example above the extra_ptr parameter can be very useful. Its initial value is determined by the call to the function that starts loading the file:

```c
extern void FLEXFILE_load_body(FLEXFILE_FORMAT_STRUCT *,
  FLEXFILE_UNKNOWN_ACTION,
  FILE *, void *);
```

The void * parameter indicates the initial value of the fourth parameter of the callback functions. The other parameters indicate respectively the format of the file, what to do when fields are encountered that the program does not recognize and the file that has to be read.

So, after all callback functions have been assigned, the following function call is made in our example program:

```c
FLEXFILE_load_body(format_ptr, FLEXFILE_UNKNOWN_KEEP,
  file_ptr, addressbook_ptr);
```

The FLEXFILE_UNKNOWN_KEEP parameter tells the library that any data in the file that is not recognized by the program should still be read and passed to the program.

This leads us to the example_address_unknown callback function. Its format is:

```c
void *example_address_unknown(char *type_ptr, char *field_ptr,
  long index, void *extra_ptr,
  void *unknown_ptr);
```
It is called by the parser if there are some fields in the record that are not recognized by the program, but should still be read (as indicated by the FLEXFILE_UNKNOWN_ACTION of the load function). The unknown_ptr parameter of the callback function contains a pointer to the data. The program should not do anything with it, except keep it around in case it wants to save the data again later.

Consider a program that sorts an address book by name. It does not need to know any fields of the Address type, beside the Name field. But it does need to keep all data associated with each address (for example the E-Mail fields of each Address), so it can save the entire address book again once the sorting is complete. By using the callback function for unknown data it can sort any address book, no matter what kind of fields it has been extended with.

To allow the program to get rid of data it received by the callback function for unknown data there is the function FLEXFILE_destroy_unknown. It can be used to free the memory used by the unknown_ptr when the program terminates or discards the contents of the file.

Most of the callback functions have been handled in this section. The callback functions for fixed size types work exactly as those for variable size types, but they lack a size_t parameter, since the size of the data types they handle is fixed. The callback functions for arrays work the same as for record types, with the exception that they have an extra long parameter indicating the number of elements of the array.

While no data is actually read in the start and end functions for arrays and records, these functions can be used to allocate memory for the upcoming array or record and initialize it with default values.

Callback functions that are called after a type has been read have not been used in the example. For fixed and variable size types these callback functions have two extra parameters, one indicating the size of the data and one indicating the position where the data was read from or written to (depending on whether a file is being written or read). These functions may be useful for converting data after it has been read. The functions called after an array or record has been read may be useful for gathering meta data for example. In any case these callback functions do not return a pointer like their counterparts that are executed before the data is loaded.

Example Program

Here is an example program that reads a flexfile containing an address book and displays the names in the book.

```c
#include <flexfile.h>
#include <stdlib.h>

typedef struct {
    char *name_ptr;
    void *unknown_ptr;
} ADDRESS_STRUCT;

typedef struct {
    long nr_addresses;
```
ADDRESS_STRUCT *address_ptr;
void *unknown_ptr;
} ADDRESS_BOOK_STRUCT;

void *example_read_string(
    const char *type_ptr,
    const char *field_ptr,
    long index,
    void *extra_ptr,
    size_t length
)
{
    ADDRESS_STRUCT *address_ptr = (ADDRESS_STRUCT *)extra_ptr;
    void *result_ptr = NULL;

    if (strcmp(field_ptr, "Name") == 0) {
        address_ptr->name_ptr = malloc(length);
        result_ptr = address_ptr->name_ptr;
    }
    return result_ptr;
}

void example_address_unknown(
    const char *type_ptr,
    const char *field_ptr,
    long index,
    void *extra_ptr,
    void *unknown_ptr
)
{
    ADDRESS_STRUCT *address_ptr = (ADDRESS_STRUCT *)extra_ptr;
    address_ptr->unknown_ptr = unknown_ptr;
}

void *example_start_address_book(
    const char *type_ptr,
    const char *field_ptr,
    long index,
    void *extra_ptr
)
{
    ADDRESS_BOOK_STRUCT *book_ptr = (ADDRESS_BOOK_STRUCT *)extra_ptr;

    book_ptr->nr_addresses = 0;
    book_ptr->address_ptr = NULL;
    book_ptr->unknown_ptr = NULL;
return book_ptr;
}

void example_address_book_unknown(
    const char *type_ptr,
    const char *field_ptr,
    long index,
    void *extra_ptr,
    void *unknown_ptr
)
{
    ADDRESS_BOOK_STRUCT *book_ptr = (ADDRESS_BOOK_STRUCT *)extra_ptr;
    book_ptr->unknown_ptr = unknown_ptr;
}

void *example_start_addresslist(
    const char *type_ptr,
    const char *field_ptr,
    long index,
    void *extra_ptr,
    long length
)
{
    ADDRESS_BOOK_STRUCT *book_ptr = (ADDRESS_BOOK_STRUCT *)extra_ptr;
    book_ptr->nr_addresses = length;
    book_ptr->address_ptr = malloc(length * sizeof(ADDRESS_STRUCT));
    return book_ptr->address_ptr;
}

void *example_start_address(
    const char *type_ptr,
    const char *field_ptr,
    long index,
    void *extra_ptr
)
{
    ADDRESS_STRUCT *list_ptr = (ADDRESS_STRUCT *)extra_ptr;
    ADDRESS_STRUCT *address_ptr = &(list_ptr[index]);
    address_ptr->name_ptr = NULL;
    address_ptr->unknown_ptr = NULL;
    return address_ptr;
}
void example_assign_read(FLEXFILE_FORMAT_STRUCT *format_ptr)
{
    /* Assign Functions */
    FLEXFILE_assign_variable_read(format_ptr, "String",
        example_read_string, NULL);
    FLEXFILE_assign_record_read(format_ptr, "Address",
        example_start_address,
        example_address_unknown, NULL);
    FLEXFILE_assign_array_read(format_ptr, "Address List",
        example_start_addresslist, NULL);
    FLEXFILE_assign_record_read(format_ptr, "Address Book",
        example_start_address_book,
        example_address_book_unknown, NULL);

    /* Known fields */
    FLEXFILE_known_field(format_ptr, "Address Book", "Addresses");
    FLEXFILE_known_field(format_ptr, "Address", "Name");
}

int main(int argc, char *argv[])
{
    FILE *inputfile_ptr = fopen(argv[1], "rb");
    FLEXFILE_FORMAT_STRUCT *format_ptr = NULL;
    ADDRESS_BOOK_STRUCT book;
    long i;

    /* Load flexfile containing an address book */
    format_ptr = FLEXFILE_load_header(inputfile_ptr);
    example_assign_read(format_ptr);
    FLEXFILE_load_body(format_ptr, FLEXFILE_UNKNOWN_KEEP,
        inputfile_ptr, &book);

    /* Print all names in the book */
    printf("%d addresses:\n", book.nr_addresses);
    for (i = 0; i < book.nr_addresses; i++) {
        printf("%s\n", book.address_ptr[i].name_ptr);
    }

    /* Cleanup */
    fclose(inputfile_ptr);
    for (i = 0; i < book.nr_addresses; i++) {
        free(book.address_ptr[i].name_ptr);
        if (book.address_ptr[i].unknown_ptr != NULL) {
            FLEXFILE_destroy_unknown(book.address_ptr[i].unknown_ptr);
        }
    }
    free(book.address_ptr);
FLEXFILE_destroy_unknown(book.unknown_ptr);
FLEXFILE_destroy(format_ptr);

return 0;
}

C.3.2 Saving a Flexfile

Saving a flexfile is very similar to loading one. The format can be constructed using functions in the library. Or, if the data was originally loaded from a flexfile, the format of that particular file can be used.

When creating a file the creation function (FLEXFILE_create) requires the name of the main record type as parameter. The function returns the new format with only one type in it: the main record type. This type has no fields yet. The FLEXFILE_format_add_* functions can then be used to add fields to records. Make sure that when adding fields to a record the type of those fields is already defined. Also take care not to have multiple types with the same name, they could be kept apart by their ids, but the library mainly identifies types by their name. Any strings passed to the library are copied, so destroying the strings passed to the functions after wards is possible.

The address book from our example would be created like this:

format_ptr = FLEXFILE_create("Address Book");
FLEXFILE_format_add_variable_type(format_ptr, "String");
FLEXFILE_format_add_record_type(format_ptr, "Address");
FLEXFILE_format_add_array_type(format_ptr, "array of Address", "Address");
FLEXFILE_format_add_field_to_record(format_ptr,
    "Address Book",
    "Creator",
    "String",
    FLEXFILE_NON_VOLATILE);
FLEXFILE_format_add_field_to_record(format_ptr,
    "Address Book",
    "Addresses",
    "array of Address",
    FLEXFILE_NON_VOLATILE);
FLEXFILE_format_add_field_to_record(format_ptr, "Address",
    "Name", "String",
    FLEXFILE_NON_VOLATILE);

The first statement creates a format with the main record type being an Address Book. The add_variable_type, add_record_type and add_array_type function calls add types to the format. Note that adding a record type creates a record type with no fields at all, so defining the fields can be postponed. For an array type however, the type of its elements has to be specified when creating the type. This is why the Address type has to be created before the array of Address type. Once all types have been created the add_field_to_record calls are used to add fields to existing record types.
Note that because the program creates the format, the library assumes that the program knows every field it creates. So there is no need to call FLEXFILE_known_field after creating a new format.

The callback functions can be assigned in the same way as for loading a flexfile. They have the same four basic parameters as the callback functions for loading. The difference is that the extra parameters that were input parameters when loading (length of the array, size of the data) now have become output parameters. I.e. they have become pointers and the parser expects the callback function to place the correct value at the address specified by that pointer. Also the pointers returned by the callback functions for reading fixed and variable types now indicate the address where the data can be found by the parser as opposed to where the data has to be put by the parser.

So the callback function we assign for writing a string in our example would be:

```c
example_write_string(char *type_ptr, char *field_ptr,
    long index, void *extra_ptr,
    size_t *size_ptr);
```

The parser will call the function before writing a String. The function needs to do two things. First it needs to determine the size of the string in bytes and place that number at the address specified by its size_ptr parameter. After that it needs to return the address of the string.

The callback function for writing the unknown fields of a record must return the unknown_ptr parameter that was passed to the program when reading that specific record. The function is only called by the parser if there are fields in the format that the program does not recognize.

Note that the parameter of the save function that indicates how to handle unknown data should never be less strict than for the load function. So if all unknown data was discarded during the load function, all unknown data should be discarded during the save function. Keeping all unknown data during loading and discarding it (or part of it) during saving is possible.

**Example Program**

Here is an example program that creates a flexfile containing an address book that contains two names.

```c
#include <flexfile.h>
#include <stdlib.h>

#define nr_addresses 2

typedef struct {
    char *name_ptr;
} ADDRESS_STRUCT;

FLEXFILE_FORMAT_STRUCT *example_create_format(void)
{
```
/* Create a new flexfile containing an address book */
FLEXFILE_FORMAT_STRUCT *format_ptr = FLEXFILE_create("Address Book");

/* Create a string type */
FLEXFILE_format_add_variable_type(format_ptr, "String");

/* Create an address type with a field "Name" that contains a string */
FLEXFILE_format_add_record_type(format_ptr, "Address");
FLEXFILE_format_add_field_to_record(format_ptr,
   "Address",
   "Name", "String",
   FLEXFILE_NONVOLATILE);

/* Create a list of addresses */
FLEXFILE_format_add_array_type(format_ptr, "Address List", "Address");

/* Add a list of addresses to the address book, named "Addresses" */
FLEXFILE_format_add_field_to_record(format_ptr,
   "Address Book",
   "Addresses",
   "Address List",
   FLEXFILE_NONVOLATILE);

/* Add a Creator field to the address book */
FLEXFILE_format_add_field_to_record(format_ptr,
   "Address Book",
   "Creator",
   "String",
   FLEXFILE_VOLATILE);

return format_ptr;
}

void *example_write_string(
   const char *type_ptr,
   const char *field_ptr,
   long index,
   void *extra_ptr,
   size_t *size_ptr

{}

ADDRESS_STRUCT *address_ptr = (ADDRESS_STRUCT *)extra_ptr;
void *result_ptr = NULL;

if (strcmp(field_ptr, "Name") == 0) {
   /* Name field */
   (*size_ptr) = strlen(address_ptr->name_ptr) + 1;
result_ptr = address_ptr->name_ptr;
} else {
    /* Creator field */
    (*size_ptr) = strlen("Owner") + 1;
    result_ptr = "Owner";
}
return result_ptr;
}

void *example_address_book_start(
    const char *type_ptr,
    const char *field_ptr,
    long index,
    void *extra_ptr
)
{
    return extra_ptr;
}

void *example_write_address_list(
    const char *type_ptr,
    const char *field_ptr,
    long index,
    void *extra_ptr,
    long *length_ptr
)
{
    *length_ptr = nr_addresses;
    return extra_ptr;
}

void *example_address_start(
    const char *type_ptr,
    const char *field_ptr,
    long index,
    void *extra_ptr
)
{
    ADDRESS_STRUCT *list_ptr_ptr = (ADDRESS_STRUCT *)extra_ptr;

    return &(list_ptr_ptr[index]);
}

void example_assign_write(FLEXFILE_FORMAT_STRUCT *format_ptr)
{
    FLEXFILE_assign_record_write(format_ptr,
        "Address Book",

example_address_book_start,
NULL, NULL);
FLEXFILE_assign_array_write(format_ptr,
"Address List",
example_write_address_list,
NULL);
FLEXFILE_assign_record_write(format_ptr,
"Address",
example_address_start,
NULL, NULL);
FLEXFILE_assign_variable_write(format_ptr,
"String",
example_write_string,
NULL);

}

int main(int argc, char *argv[]) {

FILE *outputfile_ptr = fopen("example.out", "wb");
FLEXFILE_FORMAT_STRUCT *format_ptr = NULL;
ADDRESS_STRUCT *address_ptr =
    malloc(nr_addresses * sizeof(ADDRESS_STRUCT));
address_ptr[0].name_ptr = "Piet";
address_ptr[1].name_ptr = "Henk";

/* Create a new flexfile containing an address book */
format_ptr = example_create_format();
example_assign_write(format_ptr);

FLEXFILE_save(format_ptr, FLEXFILE_UNKNOWN_DISCARD,
    outputfile_ptr, address_ptr);

/* Cleanup */
fclose(outputfile_ptr);
free(address_ptr);
FLEXFILE_destroy(format_ptr);

    return 0;
}

C.3.3 Exceptional Cases

This section explains the behavior of the library under unusual circumstances. For example, it is possible to leave callback functions unassigned. This will not crash the library, but it may have some effects depending on which callback functions remain unassigned.

Before loading fixed or variable size types a callback function is called. Should this function
be unassigned (or its return value be NULL) then the data in the file is skipped and not passed anywhere. Should the callback function called before saving such types be unassigned or return NULL then the effect depends on the type. Variable size types will be written to the flexfile with size zero and fixed size types will be written to the file as if they were filled with zeroes.

If the callback functions for records remain unassigned then the program misses an opportunity to indicate the extra_ptr parameter for the callback functions of its fields. This means that the library passes the same parameter to them as it would pass to the callback function for the record (i.e. the extra_ptr parameter does not change). Leaving the callback function for reading the unknown fields from a record unassigned means that the data is discarded for that record. Leaving it unassigned for writing the unknown fields of a record results in the unknown fields being handled as if they were known, but their callback function were unassigned.

Say that we have a program that reads an address book as used in our example. Beside a Name, the Address record also has a field of type String called E-Mail. The program only knows the field Name. When saving the address book again without assigning the callback function for unknown fields in the Address type, the E-Mail field is handled as if the callback function for String was unassigned. So the E-Mail field is written to file with size zero.

If the callback functions for reading are left unassigned then the program misses the opportunity to allocate enough memory to store the array. The callback function called before writing an array is used to indicate the size of the array to the parser, should this function be unassigned the parser assumes that the array contains zero elements.

C.4 File Format

As described in Appendix C.2 a flexfile consists of a header containing type descriptions and a body containing the data. This section describes the actual layout of a flexfile.

C.4.1 Header

Characters are one byte large. Numbers are stored in 32-bits signed integers. Sizes are stored in 32-bit unsigned integers (size_t type). Both numbers and sizes are stored in "network format", also called Big-Endian byte order [14].

The header of a flexfile starts with the character sequence "FLEX" to indicate it is a flexfile. After that follows a number indicating the number of types in the header. Then each type is described in order of its id (i.e. first the type with id 0, then 1, then 2, etc). A type description always starts with a number indicating the length of the name of the type, followed by a character sequence representing the name. The name is followed by a number indicating the base type of the type and possibly some extra information (see Table C.2). The header is followed by the body of the file containing the actual data.

Table C.4 contains the header of a flexfile in the format of our address book example.
<table>
<thead>
<tr>
<th>Number</th>
<th>Base Type</th>
<th>Extra Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fixed Size</td>
<td>Size indicating the size of an element of the type</td>
</tr>
<tr>
<td>2</td>
<td>Variable Size</td>
<td>None</td>
</tr>
<tr>
<td>11</td>
<td>Record</td>
<td>Number of fields in the records followed by a field description of each field. Table C.3 shows the format of such a description.</td>
</tr>
<tr>
<td>12</td>
<td>Array</td>
<td>Number indicating the type of the elements of the array</td>
</tr>
</tbody>
</table>

Table C.2: Format of a type description

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>The number of characters in the name of the field</td>
</tr>
<tr>
<td>Character Sequence</td>
<td>Character sequence containing the name</td>
</tr>
<tr>
<td>Number</td>
<td>Type of the field</td>
</tr>
<tr>
<td>Volatility</td>
<td>Number indicating the volatility of the field (1 = non-volatile, 2 = volatile)</td>
</tr>
</tbody>
</table>

Table C.3: Format of a field description

C.4.2 Body

The body of a flexfile consists of a single record of with type id 0. This record is called the main record and its type is called the main type.

A record’s data is stored one field after another. The fields are in the same order as they were specified in the header of the file.

For arrays there is a number that indicates the number of elements that are in the array, followed by the data of each element.

For variable size types there is a size that identifies the size of the data that follows. For fixed size types there is no extra information, the data is just stored in the file.

Table C.5 shows how the data in a file from our running example would look if the Creator field contained "Person A" and the address book would contain two persons named "Person B" and "Person C".

C.4.3 Limitations and Restrictions

Because the format uses 32-bit signed integers to store numbers the number of different types in a file is at most $2^{31}$. Likewise the maximum number of elements contained in an array is $2^{31} - 1$. Size identifiers are 32-bit unsigned integers, so the maximum size of any data is $2^{32} - 1$ bytes.

Furthermore it is required that all types have distinct names and all fields of a single record type have distinct names. The library provides no functionality to keep types with the same name or fields of the same record with the same name apart.

It is also not allowed to have a cycle of record types. Thus if record type A contains a field of record type B then record type B may not have a field of type A. Larger, or smaller, cycles
### C.4. File Format

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Data</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–3</td>
<td>FLEX</td>
<td>Character sequence indicating that this is a flexfile</td>
</tr>
<tr>
<td>4–7</td>
<td>4</td>
<td>Number of types</td>
</tr>
<tr>
<td>8–11</td>
<td>12</td>
<td>Length of the name of type 0</td>
</tr>
<tr>
<td>12–23</td>
<td>Address Book</td>
<td>Name of type 0</td>
</tr>
<tr>
<td>24–27</td>
<td>11</td>
<td>Address Book has base type Record</td>
</tr>
<tr>
<td>28–31</td>
<td>2</td>
<td>Number of fields in Address Book</td>
</tr>
<tr>
<td>32–35</td>
<td>7</td>
<td>Length of the name of field 0 of Address Book</td>
</tr>
<tr>
<td>36–42</td>
<td>Creator</td>
<td>Name of field 0 of Address Book</td>
</tr>
<tr>
<td>43–46</td>
<td>1</td>
<td>The field Creator of Address Book is of type 1</td>
</tr>
<tr>
<td>47–50</td>
<td>2</td>
<td>The field Creator of Address Book is volatile</td>
</tr>
<tr>
<td>51–53</td>
<td>9</td>
<td>Length of the name of field 1 of Address Book</td>
</tr>
<tr>
<td>54–62</td>
<td>Addresses</td>
<td>Name of field 1 of Address Book</td>
</tr>
<tr>
<td>63–66</td>
<td>2</td>
<td>The field Addresses of Address Book is of type 2</td>
</tr>
<tr>
<td>67–70</td>
<td>1</td>
<td>The field Addresses of Address Book is non-volatile</td>
</tr>
<tr>
<td>71–74</td>
<td>6</td>
<td>Length of the name of type 1</td>
</tr>
<tr>
<td>75–80</td>
<td>String</td>
<td>Name of type 1</td>
</tr>
<tr>
<td>81–84</td>
<td>2</td>
<td>Address Book has base type Variable Size</td>
</tr>
<tr>
<td>85–88</td>
<td>16</td>
<td>Length of the name of type 2</td>
</tr>
<tr>
<td>75–80</td>
<td>array of Address</td>
<td>Name of type 2</td>
</tr>
<tr>
<td>81–84</td>
<td>12</td>
<td>Address Book has base type array</td>
</tr>
<tr>
<td>85–88</td>
<td>3</td>
<td>Elements of array of Address are of type 3</td>
</tr>
<tr>
<td>89–99</td>
<td>7</td>
<td>Length of the name of type 3</td>
</tr>
<tr>
<td>93–99</td>
<td>Address</td>
<td>Name of type 3</td>
</tr>
<tr>
<td>100–103</td>
<td>11</td>
<td>Address has base type Record</td>
</tr>
<tr>
<td>104–107</td>
<td>1</td>
<td>Number of fields in Address</td>
</tr>
<tr>
<td>108–111</td>
<td>4</td>
<td>Length of the name of field 0 of Address</td>
</tr>
<tr>
<td>112–115</td>
<td>Name</td>
<td>Name of field 0 of Address</td>
</tr>
<tr>
<td>116–119</td>
<td>1</td>
<td>The field Name of Address is of type 1</td>
</tr>
<tr>
<td>120–123</td>
<td>1</td>
<td>The field Name of Address is non-volatile</td>
</tr>
</tbody>
</table>

**Table C.4: Example header**

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Data</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>124–127</td>
<td>8</td>
<td>Size of the variable size type of the Creator field</td>
</tr>
<tr>
<td>128–135</td>
<td>Person A</td>
<td>Data for the Creator Field</td>
</tr>
<tr>
<td>136–139</td>
<td>2</td>
<td>Number of elements in the array type of the Address field</td>
</tr>
<tr>
<td>140–143</td>
<td>8</td>
<td>Size of the variable size type of the Name field</td>
</tr>
<tr>
<td>144–151</td>
<td>Person B</td>
<td>Data for the Name field</td>
</tr>
<tr>
<td>152–155</td>
<td>8</td>
<td>Size of the variable size type of the Name field</td>
</tr>
<tr>
<td>156–163</td>
<td>Person C</td>
<td>Data for the Name field</td>
</tr>
</tbody>
</table>

**Table C.5: Example body**
must also be avoided. Otherwise the parser gets stuck in that cycle.

If some sort of recursive structure needs to be implemented care must be taken to ensure that it is possible to end the recursion. So, instead of a record of type A that contains a record of type B, which again contains a record of type A. We should modify record type A to contain an array of records of type B. This way the recursion can end by setting the size of the array to 0.

C.5 Storing Graphs and Partitions

The flexfile allows for storing data in a flexible manner, but it is still necessary to have some common formatting of that data. For this research, files are needed to store graphs and partitioned graphs. These need to have at least some types and fields in common if different programs want to use the same file. This section describes those formats.

C.5.1 Graphs

To store unpartitioned graphs in a flexfile the types listed below are used. The long is a fixed size type containing a 32-bit signed integer in Big-Endian format. The unsigned char is a fixed size type containing an 8-bit unsigned number.

A graph is stored in a record of type Graph. The type is described in Table C.6. Subsequent tables describe the other record types used.

Even though each node and edge has an identifier, this identifier is not used in referencing the node or edge in the file. Instead the index of the object in the array containing it is used. This allows for easier and faster loading, as well as the faster look up of elements.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodes</td>
<td>array of Node</td>
<td>The nodes of the graph</td>
</tr>
<tr>
<td>edges</td>
<td>array of Edge</td>
<td>The edges of the graph</td>
</tr>
<tr>
<td>rules</td>
<td>array of Rule</td>
<td>The rules of the graph</td>
</tr>
<tr>
<td>criteria</td>
<td>array of Criterium-Description</td>
<td>Description of the different criteria used in the graph</td>
</tr>
</tbody>
</table>

Table C.6: Graph type

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>global.id</td>
<td>long</td>
<td>The global id of the node. This Id is unique for this node in the file. Note that the id is not used as a reference.</td>
</tr>
<tr>
<td>x</td>
<td>long</td>
<td>X coordinate of the node</td>
</tr>
<tr>
<td>y</td>
<td>long</td>
<td>Y coordinate of the node</td>
</tr>
</tbody>
</table>

Table C.7: Node type
### Edge

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>global.id</td>
<td>long</td>
<td>The global id of the edge. This Id is unique for this edge in the file. Note that the id is not used as a reference.</td>
</tr>
<tr>
<td>from</td>
<td>long</td>
<td>Start node. Id (index in the Graph’s Nodes array) of the node that this edge starts on.</td>
</tr>
<tr>
<td>to</td>
<td>long</td>
<td>End node. Id (index in the Graph’s Nodes array) of the node that this edge leads to.</td>
</tr>
<tr>
<td>rc</td>
<td>unsigned char</td>
<td>Road class of the road represented by this edge (see Table C.11)</td>
</tr>
<tr>
<td>direction</td>
<td>unsigned char</td>
<td>Direction of the edge (0 = From → To, 1 = To → From, 2 = both ways, 3 = neither)</td>
</tr>
<tr>
<td>crossing</td>
<td>unsigned char</td>
<td>Crossing or type of junction (see Table C.13)</td>
</tr>
<tr>
<td>fw</td>
<td>unsigned char</td>
<td>Form of way of the edge (see Table C.12)</td>
</tr>
<tr>
<td>urban</td>
<td>unsigned char</td>
<td>Indicates whether this is an urban or a rural road (0 = Urban, 1 = Rural)</td>
</tr>
<tr>
<td>criteria</td>
<td>array of Criterium</td>
<td>Cost of this edge according to the different criteria</td>
</tr>
</tbody>
</table>

Table C.8: Edge type

### Rule

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>long</td>
<td>Start edge. Id (index in the Graph’s Edges array) of the edge that this rule starts on.</td>
</tr>
<tr>
<td>to</td>
<td>long</td>
<td>End edge. Id (index in the Graph’s Edges array) of the edge that this rule leads to.</td>
</tr>
<tr>
<td>criteria</td>
<td>array of Criterium</td>
<td>Cost of this rule according to the different criteria</td>
</tr>
</tbody>
</table>

Table C.9: Rule type

### Criterium

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>long</td>
<td>Length in centimeters</td>
</tr>
<tr>
<td>time</td>
<td>long</td>
<td>Driving time in centiseconds</td>
</tr>
</tbody>
</table>

Table C.10: Criterium type
The number of criteria in each criteria array is the same. The index of a criterium in a criteria array of an edge or rule associates it with its description in the criteria array of the Graph. The Criterium.Description record type has only one field called name. It contains a String (variable size type, the data does not contain the final null character that is normally present at the end of C strings) with the name or description of the criterium. The record can be expanded later to contain some extra information on when to apply which criterium. This could be used for storing time profiles for example.

It is sometimes necessary to give a certain rule infinite cost, to indicate that a certain turn does not incur an extra cost, but is just not allowed. The value -1 is used to indicate an infinite length or time for a criterium. Note that a criterium that has infinite length but finite time (or vice versa) is not allowed.

The Rule type may be expanded later to include information on which direction the from edge has to be followed for the rule to apply. The currently available fields do not have this information that is why it is left out of this description. An other possible extension is to restrict the times on which a rule or edge applies.

All fields described above are set to non-volatile. It is true that a program that does not know some fields could disturb the consistency of the graph, but we assume that all programs capable of reading a graph or partition know these fields. Otherwise almost every field should be set to volatile.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Motorway</td>
</tr>
<tr>
<td>1</td>
<td>Highway</td>
</tr>
<tr>
<td>2</td>
<td>Main road</td>
</tr>
<tr>
<td>3</td>
<td>Primary connector</td>
</tr>
<tr>
<td>4</td>
<td>Local connector</td>
</tr>
<tr>
<td>5</td>
<td>Local road</td>
</tr>
<tr>
<td>6</td>
<td>Restricted access</td>
</tr>
</tbody>
</table>

Table C.11: Roadclasses
### C.5. Storing Graphs and Partitions

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Motorway</td>
</tr>
<tr>
<td>1</td>
<td>Motorway Entry</td>
</tr>
<tr>
<td>2</td>
<td>Motorway Exit</td>
</tr>
<tr>
<td>3</td>
<td>Motorway Entry and Exit</td>
</tr>
<tr>
<td>4</td>
<td>Dual Carriageway</td>
</tr>
<tr>
<td>5</td>
<td>Multi Carriageway with 3 ways</td>
</tr>
<tr>
<td>6</td>
<td>Multi Carriageway with 4 ways</td>
</tr>
<tr>
<td>7</td>
<td>Multi Carriageway with 5 ways (or with Divider)</td>
</tr>
<tr>
<td>8</td>
<td>Multi Carriageway Entry</td>
</tr>
<tr>
<td>9</td>
<td>Multi Carriageway Exit</td>
</tr>
<tr>
<td>10</td>
<td>Multi Carriageway Entry and Exit</td>
</tr>
<tr>
<td>11</td>
<td>Single Carriageway with 4 or more lanes</td>
</tr>
<tr>
<td>12</td>
<td>Single Carriageway with less than 4 lanes</td>
</tr>
<tr>
<td>13</td>
<td>Boat ferry</td>
</tr>
<tr>
<td>14</td>
<td>Railway ferry</td>
</tr>
<tr>
<td>15</td>
<td>Service road</td>
</tr>
<tr>
<td>16</td>
<td>Entrance or Exit</td>
</tr>
</tbody>
</table>

Table C.12: Form of Way

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No Junction</td>
</tr>
<tr>
<td>1</td>
<td>Squares Boundary</td>
</tr>
<tr>
<td>2</td>
<td>Special Traffic Road</td>
</tr>
<tr>
<td>3</td>
<td>Parking place Boundary</td>
</tr>
<tr>
<td>4</td>
<td>Parking place Road</td>
</tr>
<tr>
<td>5</td>
<td>Parking place Boundary and Road</td>
</tr>
<tr>
<td>6</td>
<td>Parking place Boundary and Main Road</td>
</tr>
<tr>
<td>7</td>
<td>Parking building Boundary</td>
</tr>
<tr>
<td>8</td>
<td>Parking building Road</td>
</tr>
<tr>
<td>9</td>
<td>Parking building Boundary and Road</td>
</tr>
<tr>
<td>10</td>
<td>Parking building Boundary and Main Road</td>
</tr>
<tr>
<td>11</td>
<td>Mini Roundabout</td>
</tr>
<tr>
<td>12</td>
<td>Normal Roundabout</td>
</tr>
<tr>
<td>13</td>
<td>Major Roundabout</td>
</tr>
<tr>
<td>14</td>
<td>Undefined area Boundary and Road</td>
</tr>
<tr>
<td>15</td>
<td>Undefined area Boundary and Main road</td>
</tr>
<tr>
<td>16</td>
<td>Others</td>
</tr>
</tbody>
</table>

Table C.13: Part of Junction Type
C.5.2 Partitions

For storing a partition in a flexfile the same structures as for graphs are used, as well as some extra structures. The main record of the flexfile becomes a record of type Cell. Because a cell can contain other cells this structure is suited for multi-level partitions.

In multi-level partitions the cells form a tree structure. The highest level in the tree contains only the main cell, which contains all other cells, while the cells of the lowest level contain no other cells. Note that the tree need not be balanced in any way. When discussing the higher level cells of a cell we mean the cells that are ancestors of the cell in question. Likewise, lower level cells of a cell are the cells that are children of the cell in question.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>graph</td>
<td>Graph</td>
<td>Detailed map of this cell, or the boundary graph of the partition represented by this cell</td>
</tr>
<tr>
<td>routegraph</td>
<td>Graph</td>
<td>Route graph of this cell (volatile)</td>
</tr>
<tr>
<td>bnodes</td>
<td>array of long</td>
<td>Boundary nodes of this cell</td>
</tr>
<tr>
<td>bedges</td>
<td>array of long</td>
<td>Boundary edges of this cell</td>
</tr>
<tr>
<td>cells</td>
<td>array of Cell</td>
<td>Sub-cells of this cell</td>
</tr>
</tbody>
</table>

Table C.14: Cell type

The bnodes and bedges fields of the Cell type are used to store a list of the boundary nodes and boundary edges of the cell. The bnodes field contains the index of the boundary nodes in the graph of the cell. The bedges field is a bit more complicated. For a cell's boundary edge there are two possibilities: the edge is in the boundary graph of the cell's parent, or the edge is also a boundary edge of the cell's parent. The first case is more common than the latter. It is possible to store this information in a simple long. The long contains the index of the boundary edge in the graph of the cell's parent, or it is negative. If it is negative that means the edge is also a boundary edge of the parent cell and the number is actually the two's complement form of the index in the cell's parent bedges array. This form can be translated to a normal index by adding 1 to the value and multiplying the result by -1.

The Node record type is extended with a field calledupid, which is of type long. This field is used to store the connection of a boundary node to a node in the boundary graph of the higher level cell. The field contains the index of the node in the boundary graph. This allows for easy translation when ascending a level during planning. If a boundary node is reached during planning the translation will allow the planning process to find edges in higher level cells that are connected to the boundary node.

For nodes that are not boundary nodes and nodes in the route graph the upid field contains the value -1.

The planning process also needs a translation the opposite direction. When a boundary edge is expanded the process needs to descend to a lower level containing the routegraph or detailed map of the cell. To this end the Edge type is expanded with two fields: from_down and to_down, both of type DownId (see Table C.15). These fields store the connection of a boundary edge to boundary edges in a lower level route graph. For every boundary edge of
every boundary graph these fields indicate the corresponding edges in the routegraph of a
lower level cell.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cell</td>
<td>long</td>
<td>Index of the cell containing the lower level object</td>
</tr>
<tr>
<td>id</td>
<td>long</td>
<td>Id of the lower level object in the lower level</td>
</tr>
</tbody>
</table>

Table C.15: Definition of DownId

Note that this means that the routegraph field must contain a copy of the boundary edges as well as the actual route graph structure. The reason for this is Point structures. It is not possible to map nodes from higher to lower level boundary nodes with Point structures, because the virtual node on the higher level represents multiple boundary nodes on the lower level. Therefore a mapping of boundary edges must be used to descend a level.

The from_down and to_down fields are not only valid for boundary edges in a boundary graph. The boundary edges in the route graph also have these fields. And with the information in them it is possible to descend multiple levels in the same way that it is possible to ascend multiple levels with a node’s upid field.

Note, however, that for the boundary edges in the route graph only one of the two fields (from_down or to_down can contain a mapping to a lower level. A boundary edge connects two cells, but the route graph containing a copy of boundary edge belongs to only one cell. So such a copy effectively belongs to that same cell. Therefore a copy can only contain a mapping to lower levels of the cell to which it belongs. The other field of the copy, the one that does not contain a mapping, contains the value -1 in all of its fields to indicate that it is not possible to descend any further on that side of the boundary edge.

The edges in the route graph that are not boundary edges also have these fields, but they contain the value -1 to indicate that the edges are not boundary edges. That leaves only the boundary edges in the route graph of the lowest level cells. For these boundary edges there is no lower level cell to refer to. But their fields can be used to refer to the boundary node in the detailed map to which the edge connects. This is done by setting the id field of the from_down or to_down field (depending on which side of the boundary edge connects to the cell) to contain the index of the node in the cell’s graph field. The cell field of the DownId record contains the value -1 to indicate that this is a mapping to a boundary node of the detailed map and not to a boundary edge in a lower level.

To be able to translate back up when not descending all the way to the detailed map, a mapping of boundary edges in the route graph to boundary edges in the boundary graph is needed. This is done by expanding the Edge record with a field upid of type long just like the Node record. But this field contains the index of the boundary edge in the cell’s edges array to which the edge in the route graph corresponds. Note that there is no need to store a direction, since the edge in the route graph is a copy of the edge in the boundary graph. Hence they share the same direction.

For all edges that are not in the route graph the upid field contains the value -1. Also, not all edges in the route graph refer to a boundary edge, for these edges the field also contains the
value -1.

Note that the global_id field of a node no longer has to contain a unique id. Duplicate ids are allowed for boundary nodes in different levels of the partition, as long as those boundary nodes correspond with each other. The same holds for boundary edges, they may have the same global_id as long as they correspond with each other.

It is possible to create a flexfile containing a partition without a route graph. In this case the routegraph does not exist in the file, or it contains an empty graph (0 nodes, edges and rules). This means that the program that adds the route graphs to the file must also update the from_down and to_down fields of all the boundary edges.
Appendix D

Interface

This appendix contains a description of the interface of the data structures used during this research.

D.1 Linked Lists

The linked list is a bit different from the standard implementation [17]. Since a pointer to the last element in the list has to be stored, the first element of the list can not be used as a reference to the list.

In order to make the list interface as abstract as possible, a type for the position in a linked list was constructed. The idea is that a programmer using the linked list can declare a pointer to a position and use that pointer in operations on the list. The reason that it is a pointer is that the position of an element in the list can change and thus the position would have to be updated.

In reality a position in the list is just an element in the list. So a pointer to a position is just a pointer to an element. But it adds a level of abstraction that may be useful.

Below is a description of the data structure and the macros and functions that can be used with a linked list. There is also a data type for linked list in which the elements are not pointers, but of type long. This type works exactly the same as the normal linked list, but instead of pointers longs are used. Furthermore the prefix is not LINKEDLIST but LINKEDLIST_LONG.

/* @STRUCT LINKEDLIST_ELEMENT | */
* This is an element of a linked list structure
* /
typedef struct linkedlist_element_struct
{
    struct linkedlist_element_struct *next_ptr;
    /* @FIELD Address of the next list
    * element
    */
    void *data_ptr; /* @FIELD Address of the data */

    struct linkedlist_element_struct *prev_ptr;
    /* @FIELD Address of the previous list
    */
    void *data_ptr; /* @FIELD Address of the data */
}

typedef struct linkedlist_element_struct
{
    int data; /* @FIELD Data */
};

typedef int ** LINKEDLIST_LONG;
} LINKEDLIST_ELEMENT_STRUCT;

/* @STRUCT LINKEDLIST */
* This is a linked list structure
*/
typedef struct linkedlist_struct
{
    struct linkedlist_element_struct *first_ptr;
    /* @Field Address of the first element
       * of the list
    */
    struct linkedlist_element_struct *last_ptr;
    /* @Field Address of the last element
       * of the list
    */
} LINKEDLIST_STRUCT;

/*
* A abstraction of a position in the list
* Only pointers to positions should be declared.
* (This means the pointer does not have to be updated should the
* position of an element in the list change)
*/
typedef LINKEDLIST_ELEMENT_STRUCT LINKEDLIST_POSITION;

/* @MACRO
*    LINKEDLIST_position_data
* *
* @DESCRIPTION
*    Returns a pointer to the data in the element at position
*    position_ptr in the list
* *
* @PARA
*    LINKEDLIST_POSITION *| position_ptr | IN | The position
* *
* @PRE
*    position_ptr != NULL
* *
* @RETURNS
*    A void * pointing to the data in the element
*/
#define LINKEDLIST_position_data(position_ptr) (position_ptr)->data_ptr

/* @MACRO
*    LINKEDLIST_position_first
D.1. Linked Lists

* @DESCRIPTION
* Returns a pointer to the position of the first element
*
* @PARAM
* LINKEDLIST_STRUCT *|linkedlist_ptr|IN|The linkedlist
*
* @RETURNS
* A LINKEDLIST_POSITION pointing to the first element of the list
*
#define LINKEDLIST_position_first(linkedlist_ptr) (\(\)
(linkedlist_ptr)->first_ptr)

/*@MACRO
* LINKEDLIST_position_next
*
* @DESCRIPTION
* Alters position_ptr such that it points to the position
* of the next element.
* If there is no next element position_ptr becomes NULL
*
* @PARAM
* LINKEDLIST_POSITION | position_ptr | IN_OUT | The iterator
*
* @PRE
* position_ptr != NULL
*
* @RETURNS
* Nothing
*
#define LINKEDLIST_position_next(position_ptr) position_ptr = \(\)
(position_ptr)->next_ptr

/*@NAME
* LINKEDLIST_init
*
* @DESCRIPTION
* Initializes a linked list.
*
* @RETURNS
* None
*
* @GLOBAL
*
* @PRE
*
* @POST
The list is empty, and ready for input.

void LINKEDLIST_init(
    LINKEDLIST_STRUCT *list_ptr /* @ PARA IN | The linked list
    * | that is to be
    * | initialized
    */
)

/* @ NAME
   * LINKEDLIST_term
   *
   * @ DESCRIPTION
   * Removes all items from a linked list structure.
   * In other words it empties the list.
   * The data_term parameter offers a way to
   * clean up the data in each element
   *
   * @ RETURNS
   * None
   *
   * @ GLOBAL
   *
   * @ PRE
   *
   * @ POST
   * All items in the list have been deleted and
   * data_term has been called on each data field
   *
   * @ REFERENCES
   *
   * @ REMARKS
   */
void LINKEDLIST_term(
    LINKEDLIST_STRUCT *list_ptr, /* @ PARA IN_OUT | The list
    * | to be removed
    */
    void (*data_term)(void *data) /* @ PARA IN | The function
    * | to call for each
    * | data pointer
    */
)
D.1. Linked Lists

/* @NAME
   * LINKEDLIST_insert
   *
   * @DESCRIPTION
   * Inserts a new link element at the front of the list
   *
   * @RETURNS
   * Position of the new element
   *
   * @GLOBAL
   *
   * @PRE
   *
   * @POST
   *
   * @REFERENCES
   *
   * @REMARKS
   */
LINKEDLIST_POSITION *LINKEDLIST_insert(
  LINKEDLIST_STRUCT *list_ptr, /* @PARA IN_OUT | The list
  * | to which
  * | the data
  * | will be added
  */
  void *data_ptr /* @PARA IN | The data to
  * | be added
  */
)

/* @NAME
   * LINKEDLIST_append
   *
   * @DESCRIPTION
   * Inserts a new link element at the end of the list
   *
   * @RETURNS
   * Position of the new element
   *
   * @GLOBAL
   *
   * @PRE
   *
   * @POST
   *
   * @REFERENCES
   *
* @REMARKS
* /
LINKEDLIST_POSITION *LINKEDLIST_append(
    LINKEDLIST_STRUCT *list_ptr, /* @PARA IN_OUT | The list
    *            | to which
    *            | the data
    *            | will be added
    */
    void *data_ptr    /* @PARA IN | The data to
                      *            | be added
                      */

)

/* @NAME
*   LINKEDLIST_concat
* /
* @DESCRIPTION
*   Concatenates two list into one list
* *
* @RETURNS
*   None
* *
* @GLOBAL
* *
* @PRE
* *
* @POST
*   list1 = list1 ++ list2
* *
* @REFERENCES
* *
* @REMARKS
*   list2 itself isn't copied. Modifying it will result in
*   modifications to a part of (the new) list1.
* /
void LINKEDLIST_concat(
    LINKEDLIST_STRUCT *list1_ptr, /* @PARA IN_OUT | The list
                                  *            | to which
                                  *            | the other
                                  *            | list is
                                  *            | appended.
    */
    LINKEDLIST_STRUCT *list2_ptr   /* @PARA IN | The list that
                                    *            | is appended
                                    */
)

)
D.2 Double Linked Lists

Double linked lists are a lot like the normal linked list. The exceptions are that it is possible to search for a previous element and to delete an element. All functions for linked lists can be used on double linked lists, as long as the prefix DOUBLELINKEDLIST instead of LINKEDLIST is used.

Functions and macros that are specific for double linked lists are listed here:

/* @STRUCT DOUBLELINKEDLIST_ELEMENT |
 * This is an element of a linked list structure
 */

typedef struct doublelinkedlist_element_struct
{
    struct doublelinkedlist_element_struct *next_ptr;
    /* @FIELD Address of the
     * next list element
     */
    struct doublelinkedlist_element_struct *prev_ptr;
    /* @FIELD Address of the
     * previous element
     */
}
*/
void *data_ptr; /* @FIELD Address of the data */
}
DOUBLELINKEDLIST_ELEMENT_STRUCT;

/* @STRUCT DOUBLELINKEDLIST */
/* This is a linked list structure */
typedef struct doublelinkedlist_struct {
  struct doublelinkedlist_element_struct *first_ptr;
  /* @Field Address of the first element */
  * of the list */
  struct doublelinkedlist_element_struct *last_ptr;
  /* @Field Address of the last element */
  * of the list */
} DOUBLELINKEDLIST_STRUCT;

/*
 * A abstraction of a position in the list
 * Only pointers to positions should be declared.
 * (This means the pointer does not have to be updated should the
 * position of an element in the list change)
 */
typedef DOUBLELINKEDLIST_ELEMENT_STRUCT DOUBLELINKEDLIST_POSITION;

/* @MACRO
 * DOUBLELINKEDLIST_position_prev
 * @DESCRIPTION
 * Alters the position_ptr such that it points to
 * the position of the previous element.
 * If there is no previous element position_ptr becomes NULL
 * @PARA
 * DOUBLELINKEDLIST_POSITION *|position_ptr|IN|The position
 * @PRE
 * position_ptr != NULL
 * @RETURNS
 */
#define DOUBLELINKEDLIST_position_prev(position_ptr) position_ptr = \
  position_ptr->prev_ptr
D.3. AVL Trees

The AVL trees [1] were used in the earlier programs (the ones described before Section 3.4.5, as a means to keep the cells and celledges collections sorted.

There were two variants of the data type, one that contained elements with a key of type long and one that contained a key of type double. The first has a prefix of AVLTREE and the second of AVLTREE.DOUBLE.

An empty tree is represented by the NULL pointer. A tree with a single node can be created by the AVLTREE.init function, or by calling AVLTREE.insert with *root.ptr == NULL to insert the node.
Elements of a tree structure can easily be turned into an ordered list. The nodes are added to the list in such a way that a lower-child is always in the list before its parent and a higher-child is always in the list after its parent. Though this list is never actually generated we will call it sorted_list to describe the workings of several functions.

/* @STRUCT AVL_TREE_NODE
 * This is a (node of) an avl tree
 */
typedef struct avl_tree_node
{
    struct avl_tree_node *parent_ptr; /* @FIELD Address of the parent
        * node
        */
    struct avl_tree_node *lower_ptr;  /* @FIELD Address of the subtree
        * with equal or lower keys
        */
    struct avl_tree_node *higher_ptr; /* @FIELD Address of the subtree
        * with equal or higher keys
        */
    long key;  /* @FIELD Key of this node */
    int balance; /* @FIELD Balance of this node
        * (Height(lower_ptr) -
        *   Height(higher_ptr))
        * This value should be:
        *   -1, 0 or 1
        *   any other value is only
        *   possible while updating
        *   the tree.
        */
    void *data_ptr; /* @FIELD Address of data
        *   associated with this node
        */
} AVL_TREE_NODE;

/* @NAME
   *   AVL_TREE_init
   *
   * @DESCRIPTION
   *   Initializes the root node of an avl tree
   *
   * @RETURNS
   *   A newly allocated root node
   *
   * @GLOBAL
   *
   * @PRE
*/
D.3. AVL Trees

*  
* @POST
*  The node is now the root node of an avl tree
*  
* @REFERENCES
*  
* @REMARKS
*/

AVLTREE_STRUCT *AVLTREE_init(
    long key,                /* @PARA IN | The key of the inserted data */
    void *data_ptr           /* @PARA IN | The data to be inserted */
)


/* @NAME
 *  
 * AVLTREE_insert
 *  
 * @DESCRIPTION
 *  Inserts data and a key into the tree
 *  and keeps it balanced
 *  
 * @RETURNS
 *  The new node
 *  
 * @GLOBAL
 *  
 * @PRE
 *  The tree is balanced
 *  
 * @POST
 *  A new node has been created and inserted into the tree.
 *  The tree is still balanced.
 *  
 * @REFERENCES
 *  
 * @REMARKS
*/

AVLTREE_STRUCT *AVLTREE_insert(
    AVLTREE_STRUCT **root_ptr,/* @PARA IN_OUT | The tree
    long key,                /* @PARA IN | The key of the inserted data */
    void *data_ptr           /* @PARA IN | The data to be inserted */
)
AVLTREE_STRUCT *AVLTREE_delete(
   AVLTREE_STRUCT *node_ptr
   /* @PARAM IN_OUT | The node to be deleted */
)

AVLTREE_STRUCT *AVLTREE_find
   AVLTREE_STRUCT *node_ptr
   /* @PARAM IN | The key to search for */
AVLTREE_STRUCT *AVLTREE_find(
    AVLTREE_STRUCT *root_ptr,
    /* PARA IN | The tree in which to find
     * a node
     */
    long key    /* PARA IN | The key */
)

AVLTREE_STRUCT *AVLTREE_findlowest(
    AVLTREE_STRUCT *root_ptr
    /* PARA IN | The tree in which to find
     * a node with the lowest key
     */
)

AVLTREE_STRUCT *AVLTREE_findhighest

Duplicate keys are possible, but this function only
returns one node.
The last node in the sorted_list

GLOBAL

PRE

POST

REFERENCES

REMARKS

AVL_TREE_STRUCT *AVL_TREE_findhighest(
    AVL_TREE_STRUCT *root_ptr
    /* @PARA IN */ The tree in which to find
    * a node with the highest key
    */
)

AVL_TREE_STRUCT *AVL_TREE_findnextdescending(
    AVL_TREE_STRUCT *node_ptr
    /* @PARA IN */ The node from which to
    * start the search
    */
)
AVLTREE_STRUCT *AVLTREE_findnextascending(
    AVLTREE_STRUCT *node_ptr,
    /* @para in | The node from which to */
    /*    | start the search */
)

AVLTREE_size

Returns the number of nodes in the tree

The number of nodes in the tree

@global

@pre

@post

@references

@remarks
int AVL_TREE_size(
    AVL_TREE_STRUCT *root_ptr
    /* @PARA IN | The tree whose size
    *     | has to be measured
    */
)

D.4 Cellgraph

The cellgraph functions have been split into two units, one unit containing the general functions for the algorithm, and one unit containing the I/O routines.

The global variable CELLGRAPH_priority_function_ptr contains a pointer to the priority function that is used. By altering this variable it is possible to use a different function. The variable is initialized with the default priority function.

The general functions are listed below. Note that there are more public functions in the cellgraph.h file, but these are to be used by the cellgraph_io unit, or for debugging purposes.

/*@ @STRUCT CELLGRAPH_NODE_STRUCT |
*     This is a cellnode, representing a cell
*/

typedef struct cellgraph_node_struct
{
    LINKEDLIST_LONG_STRUCT nodes;         /* @FIELD List of nodes belonging to
                                            *     this cell
                                            */
    DOUBLELINKEDLIST_STRUCT bnodes;       /* @FIELD List of boundary nodes */
    long nr_edges;                        /* @FIELD Number of internal edges */
    long nr_nodes;                        /* @FIELD Number of nodes (total) */
    long nr_bnodes;                       /* @FIELD Number of boundary nodes */
    DOUBLELINKEDLIST_STRUCT cedges;       /* @FIELD List of celledge endpoints
                                            *     connected to this cellnode
                                            */
    struct cellgraph_edgeend_struct *temp_ptr;
                                            /* @FIELD Pointer to store temporary
                                            *     information during the
                                            *     merging of two cells
                                            */
    DOUBLELINKEDLIST_POSITION *list_ptr;  /* @FIELD Position of this node
                                            *     in the cellgraph's
                                            *     cellnode list
                                            */
} CELLGRAPH_NODE_STRUCT;
/ * @STRUCT CELLGRAPH_EDGEEND_STRUCT | 
* This structure describes the endpoint of a celledge 
*/
typedef struct cellgraph_edgeend_struct
{
    CELLGRAPH_NODE_STRUCT *cell_ptr; /* @FIELD The cell connected to this 
        *  endpoint 
        */
    struct cellgraph_edge_struct *edge_ptr; /* @FIELD The edge of which this 
        *  is an endpoint 
        */
    struct cellgraph_edgeend_struct *otherend_ptr; /* @FIELD The endpoint on 
        *  the other side 
        */
    LINKEDLIST_STRUCT bnodes;        /* @FIELD Boundary nodes of cell 
        *  to which the endpoint 
        *  is connected 
        *  This is a list of 
        *  DOUBLELINKEDLIST_POSITIONs 
        *  that indicate the position 
        *  of the boundary node in the 
        *  cell's bnodes list. 
        */
    long nr_bnodes;                  /* @FIELD Number of elements in the 
        *  bnodes list 
        */
    DOUBLELINKEDLIST_POSITION *me_ptr; /* @FIELD This edgeend's position in 
        *  cell's cedges list 
        */
} CELLGRAPH_EDGEEND_STRUCT;

/* @STRUCT CELLGRAPH_EDGE_STRUCT | 
* This is a celledge, representing the relation between two cells 
*/
typedef struct cellgraph_edge_struct
{
    CELLGRAPH_EDGEEND_STRUCT end1;  /* @FIELD Information about the 
        *  connection to one of the 
        *  cells in the relation 
        */
    CELLGRAPH_EDGEEND_STRUCT end2;  /* @FIELD Information about the 
        *  connection to the other 
        *  cell in the relation 
        */
    long nr_edges;                   /* @FIELD Number of edges between 
        *  cell1 and cell2 
        */
double priority; /* @FIELD Priority of to merge cell1 and cell2 together */

long pos; /* @FIELD Position of this edge in the cellgraph's celledge list */

int tag_add; /* @FIELD Tag for storing extra information.
* Upon merging the new celledge's tag will be the sum of its predecessors */

} CELLGRAPH_EDGE_STRUCT;

/* @STRUCT CELLGRAPH_STRUCT |
* This is a cellgraph, representing a division of a graph into cells */
typedef struct cellgraph_struct
{
    DOUBLELINKEDLIST_STRUCT cellnodes; /* @FIELD List of cells in the cellgraph */

    CELLGRAPH_EDGE_STRUCT **celledges_ptr_ptr; /* @FIELD List of celledges in the cellgraph */

    long nr_cells; /* @FIELD Number of cells */
    long nr_celledges; /* @FIELD Number of celledges */
    long nr_nodes; /* @FIELD Total number of nodes */

    double objective_value; /* @FIELD Value of this particular partition */

} CELLGRAPH_STRUCT;

typedef double CELLGRAPH_PRIORITY_FUNCTION(CELLGRAPH_STRUCT *,
                                          CELLGRAPH_EDGE_STRUCT *);

/* @NAME
* CELLGRAPH_destroy
*
* @DESCRIPTION
* Frees the memory used by a cellgraph
*
* @RETURNS
* Nothing

D.4. Cellgraph

*  *  @GLOBAL
*  *  @PRE
*  *  @POST
*     The cellgraph is destroyed
*  *  @REFERENCES
*  *  @REMARKS
*/
void CELLGRAPH_destroy(
   CELLGRAPH_STRUCT *cellgraph_ptr
   /* @PARAM_IN_OUT | The cellgraph to destroy */
)

/*@NAME
*   CELLGRAPH_select
*  *
* @DESCRIPTION
*   Selects the edge between the two cells that have to
*   be merged
*  *
* @RETURNS
*   The edge between the two cells that have to
*   be merged
*  *
* @GLOBAL
*  *
* @PRE
*  *
* @POST
*  *
* @REFERENCES
*  *
* @REMARKS
*/

CELLGRAPH_EDGE_STRUCT *CELLGRAPH_select(
   CELLGRAPH_STRUCT *cellgraph_ptr
   /* @PARAM_IN | The cellgraph from
   *   which to pick
   */
)

/*@NAME
CELLGRAPH_merge

* @DESCRIPTION
* Merges two cells in the celgraph into one
* *
* @RETURNS
* The new cell
* *
* @GLOBAL
* *
* @PRE
* *
* @POST
* The specified cells have been merged into one
* *
* @REFERENCES
* *
* @REMARKS
*/

CELLGRAPH_NODE_STRUCT *CELLGRAPH_merge(
   CELLGRAPH_STRUCT *cellgraph_ptr,
   /* @PARA IN_OUT | The celgraph in which
   * the cells reside
   */
   CELLGRAPH_EDGE_STRUCT *edge_ptr
   /* @PARA IN_OUT | The edge between the
   * two cells that
   * will be merged
   */
)

/* Pointer to the priority function */
CELLGRAPH_PRIORITY_FUNCTION *CELLGRAPH_priority_function_ptr;

The I/O routines are used for loading and storing a celgraph. The unit containing these
routines also introduces a data type for storing the input file (called an IDB file). By loading
the file into this data type the data remains in memory and can be used multiple times(when
loading and when saving the boundary graph). There is also a function for loading the file
into the data type and converting it into a celgraph simultaneously, which is faster than
doing it in sequence.

The reason the input file is loaded into memory is that it contains information that is not
needed by the partitioning algorithm, but which do need to be written to the output file.

/* @NAME
* CELLGRAPH_IO_read_idbfile
D.4. Cellgraph

* * @DESCRIPTION
* Creates a CELLGRAPH_IO_IDBFILE_STRUCT of the file
* *
* @RETURNS
* The new struct
* *
* @GLOBAL
* *
* @PRE
* The file exists and can be read
* *
* @POST
* *
* @REFERENCES
* *
* @REMARKS
*/
CELLGRAPH_IO_IDBFILE_STRUCT *CELLGRAPH_IO_read_idbfile(
    char *filename_ptr
    /* @PARAM IN | Filename of the idb file */
)

/* @NAME
 * CELLGRAPH_IO_term_idbfile
 *
 * @DESCRIPTION
 * Destroys a CELLGRAPH_IO_IDBFILE_STRUCT
 *
 * @RETURNS
 * Nothing
 *
 * @GLOBAL
 * *
 * @PRE
 * The struct exists
 *
 * @POST
 * *
 * @REFERENCES
 * *
 * @REMARKS
*/
void CELLGRAPH_IO_term_idbfile(
    CELLGRAPH_IO_IDBFILE_STRUCT *idbfile_ptr
    /* @PARAM IN_OUT | The struct to destroy */
)
CELLGRAPH_STRUCT *CELLGRAPH_IO_load_idb(
    CELLGRAPH_IO_IDBFILE_STRUCT *idbfile_ptr
    /* @PARA IN | The idb file */
)

CELLGRAPH_STRUCT *CELLGRAPH_IO_load_and_read_idbfile
    /* @PARA IN | The idb file */

CELLGRAPH_STRUCT *CELLGRAPH_IO_load_and_read_idbfile
    /* @PARA IN | The idb file */
D.4. Cellgraph

* This is a combination of the load_idbfile
* and read_idbfile functions
*/
CELLGRAPH_STRUCT *CELLGRAPH_I0_load_and_read_idbfile(
    char *filename_ptr,
    // @PARA IN | Filename of the idb file */
    CELLGRAPH_I0_IDBFILE_STRUCT **cache_ptr
    // @PARA OUT | Cache of the file
    * // for later use
    */


/* @NAME
* CELLGRAPH_I0_load
*
* @DESCRIPTION
* Creates a cellgraph by loading it from the specified file
*
* @RETURNS
* A newly allocated cellgraph, read from the file
*
* @GLOBAL
*
* @PRE
* The file was created by the CELLGRAPH_I0_save function
* The binary file has been opened for reading
* at the beginning
*
* @POST
*
* @REFERENCES
*
* @REMARKS
*/
CELLGRAPH_STRUCT *CELLGRAPH_I0_load(
    FILE *file_ptr           // @PARA IN | The file */
)

/* @NAME
* CELLGRAPH_I0_save
*
* @DESCRIPTION
* Saves the cellgraph structure to the file
* indicated by the file_ptr parameter
*
* @RETURNS
* Nothing
void CELLGRAPH_IO_save(
    CELLGRAPH_STRUCT *cellgraph_ptr,
    /* @PARA IN | The cellgraph
    *     | structure to save
    */
    FILE *file_ptr /* @PARA IN | The file */
)

/* @NAME
 * CELLGRAPH_IO_save_boundary
 *
 * @DESCRIPTION
 * Saves the boundary graph derived from the
 * cellgraph structure to a file
 *
 * @RETURNS
 * Nothing
 *
 * @GLOBAL
 *
 * @PRE
 * The file can be created/opened for writing
 * and there is enough disk space to contain the file
 * The idbfile was the one used to originally create
 * the cellgraph
 *
 * @POST
 * The boundary graph has been written to file
 *
 * @REFERENCES
 *
 * @REMARKS
D.5 Flexfile

Flexfiles can be used to store information in a flexible way. Records in a flexfile can be extended with fields while still allowing older programs to read new files or new programs read older files.

For a more detailed description of flexfiles see Appendix C.

/*
 * TYPE DEFINITIONS
 */

/* Base types */
typedef enum {
   FLEXFILE_FIXED = 1,
   FLEXFILE_VARIABLE = 2
} FLEXFILE_SIMPLE_TYPE;

/* Complex types */
typedef enum {
   FLEXFILE_RECORD = 11,
   FLEXFILE_ARRAY = 12
} FLEXFILE_COMPLEX_TYPE;

/* Indicates whether a field is volatile or not */
typedef enum {
   FLEXFILE_NONVOLATILE = 1,
   FLEXFILE_VOLATILE = 2 /* Volatile information is lost if the program does not know the field and alters any part of the data structure */
} FLEXFILE_VOLATILE_ATTRIBUTE;
typedef enum { FLEXFILE_UNKNOWN_DEFAULT = 0,
               FLEXFILE_UNKNOWN_KEEP = 1,
               FLEXFILE_UNKNOWN_DISCARD = 2,
               FLEXFILE_UNKNOWN_DISCARD_VOLATILE = 3,
               FLEXFILE_UNKNOWN_DISCARD_NONVOLATILE = 4
               } FLEXFILE_UNKNOWN_ACTION;

/* Callback functions for simple types
 * The first char * parameter indicates the name of the type
 * being read.
 * If your callback function checks the name of the type it
 * is possible to speed things up by comparing pointers;
 * the char * points to the same address as the
 * format_ptr->types_ptr[x].name_ptr.
 * (Where x is the * typeid of the type)
 * The second char * parameter points to the name of the
 * field that contains this type (when reading a record,
 * NULL otherwise) the same as for the first char * holds;
 * Name comparison can be speeded up by comparing to the
 * pointers obtained with FLEXFILE_format_get_fieldname.
 * (Each pointer is unique, making it possible to
 * distinguish fields of different record types that have
 * the same name and type)
 * The long parameter indicates the index of the type
 * (when reading an array,
 * when reading a record it is the fieldid)
 * The void pointer is a pointer that was returned to the
 * parser when it called one of the
 * FLEXFILE_RECORD_START, FLEXFILE_READ_ARRAY_START,
* FLEXFILE_WRITE_ARRAY_START
* functions when it started reading/writing a certain
* array or record.
* The parser only uses this pointer to pass it to its
* callback functions. If the callback function
* was not assigned then the parameter does not
* change.
*
* The functions should return the address where the data
* can be written to (in case of reading a file) or the
* address from where the data can be read (in case of
* writing a file).
*
* When reading an element of variable size, the size of
* the element is passed as an extra parameter to the
* function.
* When writing an element of variable size the function
* must return the size of the data by means of the output
* variable size_t.
*
* The FLEXFILE_AFTER_SIMPLE function can be used to
* perform some operation after a fixed or variable type
* has been read or written.
* The extra size parameter contains the size of the data
* (even for fixed size types) and the extra void *
* parameter contains the address where the data was
* written (in case of reading a file) or where it was
* read from (in case of writing a file).
*
* Unknown data is handled through special functions.
* The read function passes an extra void *, this is the
* one that has to be returned by the write function to
* restore unknown data.
* So to write back the unknown data the same void * must
* be returned by the write function and the
* FLEXFILE_FORMAT_STRUCT used for loading the file
* must be used for writing the file
* The parameters passed to the function are the same as
* the ones passed to the FLEXFILE_RECORD_START function
* for that record.
* (This is because the unknown data may have more than
* one field name or index).
*/
typedef void *(FLEXFILE_HANDLE_FIXED)(const char *,
    const char *,
    long, void *);
typedef void *(FLEXFILE_READ_VARIABLE)(const char *,
    const char *,
    long, void *,
    long, void *);
const char *,
long, void *,
size_t);
typedef void *(FLEXFILE_WRITE_VARIABLE)(const char *,
const char *,
long, void *,
size_t *);

typedef void (FLEXFILE_READUNKNOWN)(const char *,
const char *,
long, void *,
void *);

typedef void *(FLEXFILE_WRITEUNKNOWN)(const char *,
const char *,
long, void *);

/* Callback functions for complex types
 * The first four parameters are the same as for
 * simple types.
 *
 * The fifth parameter for arrays indicates the
 * length of the array.
 *
 * The start functions return a void pointer that
 * is passed as the fourth parameter to each callback
 * function until a new complex type starts or a
 * complex type ends.
 * If a complex type ends the fourth parameter reverts
 * to the value it had before the complex type started.
 *
 * This reverting of the fourth parameter happens after the
 * call to an END function. I.e. the FLEXFILE_RECORD_END
 * and FLEXFILE_ARRAY_END still have the value returned by
 * the START function as their fourth parameter.
 *
 * Leaving elements (of an array or record) out is not
 * possible.
 * Should a callback function be unassigned or return NULL
 * then depending on the type of the elements the following
 * happens:
 * fixed size types are filled with 0.
 * variable size types are set to size 0.
 * record types are saved with 0 fields.
 * array types are saved with 0 elements.
 */
typedef void *(FLEXFILE_RECORDSTART)(const char *,
const char *,
long, void *);
typedef void (FLEXFILE_RECORD_END)(const char *,
    const char *,
    long, void *);
typedef void *(FLEXFILE_READ_ARRAY_START)(const char *,
    const char *,
    long, void *,
    long);
typedef void *(FLEXFILE_WRITE_ARRAY_START)(const char *,
    const char *,
    long, void *,
    long *);
typedef void (FLEXFILE_ARRAY_END)(const char *,
    const char *, long,
    void *
);

/* @UNION FLEXFILE_CALLBACK_STRUCT
 * Union to store a callback function in
 */
typedef union flexfile_callback_struct
{
    struct {
        FLEXFILE_HANDLE_FIXED *read;
        FLEXFILE_HANDLE_FIXED *write;
        FLEXFILE_AFTER_SIMPLE *after_read;
        FLEXFILE_AFTER_SIMPLE *after_write;
    } fixed;
    struct {
        FLEXFILE_READ_VARIABLE *read;
        FLEXFILE_WRITE_VARIABLE *write;
        FLEXFILE_AFTER_SIMPLE *after_read;
        FLEXFILE_AFTER_SIMPLE *after_write;
    } variable;
    struct {
        FLEXFILE_RECORD_START *read_start;
        FLEXFILE_RECORD_START *write_start;
        FLEXFILE_READ_UNKNOWN *read_unknown;
        FLEXFILE_WRITE_UNKNOWN *write_unknown;
        FLEXFILE_RECORD_END *read_end;
        FLEXFILE_RECORD_END *write_end;
    } record;
    struct {
        FLEXFILE_READ_ARRAY_START *read_start;
        FLEXFILE_WRITE_ARRAY_START *write_start;
        FLEXFILE_ARRAY_END *read_end;
        FLEXFILE_ARRAY_END *write_end;
    } array;
} FLEXFILE_CALLBACK_STRUCT;
/* @STRUCT FLEXFILE_FIELD_STRUCT */
* This structure represents a field of a record type */
typedef struct flexfile_field_struct
{
    char *field_name_ptr; /* @FIELD Field name */
    long field_type; /* @FIELD Field type */
    FLEXFILE_VOLATILE_ATTRIBUTE vol; /* @FIELD Indicates
                * whether this
                * field is
                * volatile */
} FLEXFILE_FIELD_STRUCT;

/*@STRUCT FLEXFILE_TYPE_STRUCT */
* This represents a type in a flexfile */
typedef struct flexfile_type_struct
{
    char *name_ptr; /* @FIELD Name of
                * this type */
    long field_type; /* @FIELD Indicates
                * the type of
                * the field */

    union {
        size_t fixed_size; /* @FIELD Size of
                * a fixed
                * size
                * type */
        long array_subtype; /* @FIELD Type of
                * the elements
                * of the array */
    }

    struct {
        long nr_fields; /* @FIELD Number of
                * fields */
    } record;
} info;
} FLEXFILE_TYPE_STRUCT;

/* @STRUCT FLEXFILE_FORMAT_STRUCT */
* The format of a flexfile and how to interpret it.
*/
typedef struct flexfile_format_struct
{
    long nr_types;            /* @FIELD Number of
                                 *    types in
                                 *    this file
                                 */
    long nr_complex_types;    /* @FIELD Number of
                                 *    complex
                                 *    types
                                 *    (array, record)
                                 *    in this file
                                 */
    FLEXFILE_TYPE_STRUCT *types_ptr;  /* @FIELD Array of
                                          *    types
                                          */
    long **known_fields_ptr_ptr;    /* @FIELD Indicates
                                         *    how the program
                                         *    using this file
                                         *    knows certain fields.
                                         *    -1 = Program
                                         *    does not
                                         *    know the type
                                         *    0 = Program
                                         *    knows the
                                         *    type as it is in
                                         *    the file
                                         *    x = Program
                                         *    knous the
                                         *    type as x, while it
                                         *    is something
                                         *    different in the
                                         *    file.
                                         *    Values of x are
                                         *    not yet possible
                                         */
    FLEXFILE_CALLBACK_STRUCT *callback_ptr;  /* @FIELD Callback functions
                                               *    for each type
                                               */
    int error;
    char error_description[255];
} FLEXFILE_FORMAT_STRUCT;

/*
 * EXTERN FUNCTION DECLARATIONS
 */

/* Functions for creating and destroying a flexfile */
extern FLEXFILE_FORMAT_STRUCT *FLEXFILE_create(char *);
extern void FLEXFILE_destroy(FLEXFILE_FORMAT_STRUCT *);
extern void FLEXFILE_destroy_unknown(FLEXFILE_FORMAT_STRUCT *,
                              void *);

/* Functions for modifying the types in a flexfile */
extern long FLEXFILE_format_add_fixed_type(
    FLEXFILE_FORMAT_STRUCT *,
    char *, size_t);

extern long FLEXFILE_format_add_variable_type(
    FLEXFILE_FORMAT_STRUCT *,
    char *);

extern long FLEXFILE_format_add_record_type(
    FLEXFILE_FORMAT_STRUCT *,
    char *);

extern long FLEXFILE_format_add_array_type(
    FLEXFILE_FORMAT_STRUCT *,
    char *, char *);

extern long FLEXFILE_format_add_field_to_record(
    FLEXFILE_FORMAT_STRUCT *,
    char *, char *,
    FLEXFILE_VOLATILE_ATTRIBUTE);

/* Functions to get some information about types */
extern long FLEXFILE_format_get_typeid(
    const FLEXFILE_FORMAT_STRUCT *,
    const char *);

extern long FLEXFILE_format_get_fieldid(
    const FLEXFILE_FORMAT_STRUCT *,
    const long, const char *);

extern char *FLEXFILE_format_get_typename(
    const FLEXFILE_FORMAT_STRUCT *,
    const char *);

extern char *FLEXFILE_format_get_fieldname(
    const FLEXFILE_FORMAT_STRUCT *,
    const char *, const char *);

/* Function to validate the format of the flexfile */
extern int FLEXFILE_format_validate(FLEXFILE_FORMAT_STRUCT *);
/* Functions for loading a flexfile */
extern FLEXFILE_FORMAT_STRUCT *FLEXFILE_load_header(FILE *);
extern void FLEXFILE_assign_fixed_read(
    FLEXFILE_FORMAT_STRUCT *, char *,
    FLEXFILE_HANDLE_FIXED *,
    FLEXFILE_AFTER_SIMPLE *);
extern void FLEXFILE_assign_variable_read(
    FLEXFILE_FORMAT_STRUCT *, char *,
    FLEXFILE_READ_VARIABLE *,
    FLEXFILE_AFTER_SIMPLE *);
extern void FLEXFILE_assign_array_read(
    FLEXFILE_FORMAT_STRUCT *, char *,
    FLEXFILE_READ_ARRAY_START *,
    FLEXFILE_ARRAY_END *);
extern void FLEXFILE_assign_record_read(
    FLEXFILE_FORMAT_STRUCT *, char *,
    FLEXFILE_RECORD_START *,
    FLEXFILE_RECORD_UNKNOWN *,
    FLEXFILE_RECORD_END *);
extern void FLEXFILE_known_field(
    FLEXFILE_FORMAT_STRUCT *, char *,
    char *);
extern void FLEXFILE_load_body(
    FLEXFILE_FORMAT_STRUCT *,
    FLEXFILE_UNKNOWN_ACTION,
    FILE *, void *);

/* Functions for saving a flexfile */
extern void FLEXFILE_assign_fixed_write(
    FLEXFILE_FORMAT_STRUCT *, char *,
    FLEXFILE_HANDLE_FIXED *,
    FLEXFILE_AFTER_SIMPLE *);
extern void FLEXFILE_assign_variable_write(
    FLEXFILE_FORMAT_STRUCT *, char *,
    FLEXFILE_WRITE_VARIABLE *,
    FLEXFILE_AFTER_SIMPLE *);
extern void FLEXFILE_assign_array_write(
    FLEXFILE_FORMAT_STRUCT *, char *,
    FLEXFILE_WRITE_ARRAY_START *,
    FLEXFILE_ARRAY_END *);
extern void FLEXFILE_assign_record_write(
    FLEXFILE_FORMAT_STRUCT *, char *,
    FLEXFILE_RECORD_START *,
    FLEXFILE_WRITE_UNKNOWN *,
    FLEXFILE_RECORD_END *);
extern void FLEXFILE_save(FLEXFILE_FORMAT_STRUCT *,
    FLEXFILE_UNKNOWN_ACTION,
Bibliography


