Karlsruhe Institute of Technology

Using Graph Partitioning to Accelerate Longest Path Search

Bachelor Thesis of

Kai Fieger

At the Department of Informatics
Institute of Theoretical Informatics, Algorithmics II

Advisors: Dr. Tomáš Balyo
Prof. Dr. rer. nat. Peter Sanders

Statement of Authorship

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, 7th November 2016

Abstract

This thesis presents an optimal algorithm that solves the longest path problem for undirected graphs. The algorithm makes use of graph partitioning and dynamic programming. It's performance was evaluated in a number of benchmarks and compared to other known algorithms. The runtime of the algorithm was shown to be significantly faster on the tested graphs.

Deutsche Zusammenfassung

Diese Arbeit präsentiert einen optimalen Algorithmus, der das Längest Wege Problem (longest path problem) für ungerichtete Graphen löst. Der Algorithmus macht sich dabei die Partitionierung von Graphen und dynamische Programmierung zu Nutzen. Seine Laufzeit wurde anhand von mehreren Experimenten evaluiert und mit anderen bekannten Algorithmen vergleichen. Dabei stellte sich der Algorithmus auf den getesteten Graphen als wesentlich schneller heraus.

Contents

1. Introduction

1.1 Problem/Motivation

The shortest path problem (SP) is a well known problem of finding a path of minimum length between two given vertices of a graph. The minimum length can be defined by the number of edges that the path consists of. Another possibility with a weighted graph is to try to minimize the summed up weight of the path's edges. SP can be solved optimally and in polynomial time with Dijkstra's Algorithm [Dij59] if the graph is unweighted or only contains non-negative edge-weights or with the Bellman-Ford-Algorithm if the graph contains negative weights [Bel58] [For56].
A similar problem to SP is the longest path problem (LP), which is often used (also in this thesis) as a synonym to the more accurate longest single path problem (LSP). LP is identical to SP except that a simple path of maximum length is searched for. A path is called simple if it doesn't contain a vertex of the graph more than once. While being very similar, LP is NP complete [GJ79]. LP for example has applications in the design of circuit boards, where the length difference between wires has to be kept small [OW06] [**?**]. LP manifests itself when the length of shorter wires is supposed to be increased. Additionally the longest path is relevant to project planning/scheduling as it can be used to determine the least amount of time that a project could be completed in [Bru95]. There are also applications in the information retrieval in peer-to-peer networks [WLK05] and patrolling algorithms for multiple robots in a graph [PR10].

1.2 Content

This thesis concentrates on the longest path problem (LP) for undirected graphs. An optimal algorithm for LP based on using graph partitioning and dynamic programming is presented. The partitioning was done with KaHIP - Karlsruhe High Quality Partitioning - [SS13]. Oriented at a paper from Stern, Kiesel, Puzis, Feller and Ruml [SKP$^+$14], that provided a large amount of optimal and suboptimal algorithms for LP, experiments and benchmarks were created. The algorithm was compared with paper's algorithms and turned out to be significantly faster than the other optimal algorithms for the tested instances. Additionally the algorithm's runtime was compared with different partition-qualities to find a balance between the time spent partitioning and the runtime of the actual algorithm.

2. Preliminaries

2.1 Definitions

Vertex

A fundamental unit that graphs are made of. Here a vertex will be represented by a (natural) number. Other name: node.

Edge

Connection between two vertices x and y.
An edge that serves as a two way connection between x and y is called an **undirected edge** and is represented as the set {x, y}.
An edge that only serves as a one way connection from x to y is called a **directed edge** and is represented as the tuple (x, y).

Graph := (V, E)

A graph represents a set of vertices V whose elements are interconnected by the edges in the set E. If $E \subset \{ (x, y) \mid x, y \in V \}$, meaning directed edges/vertex tuples, G is a directed graph. While G is called an undirected graph if $E \subset \{ \{x, y\} \mid x, y \in V \}$. Additionally a graph can be weighted if its edges have weights associated with them. This can be represented with a weight function w: $E \mapsto X$, where X is a set of numbers. In this thesis the weight function will usually not be explicitly stated and weights are considered to be real numbers, $X := \mathbb{R}$.

Path

A way to traverse a graph G := (V, E) from a start- to an end-vertex. For a directed graph G a path P can be defined as a sequence of directed edges: $P := (e_1,e_2,...,e_k)$, where $e_i := (v_i,v_{i+1})$ and $e_i \in E$. A path can be defined the same way for an undirected graph G, where $\{v_i,v_{i+1}\}$ has to be element of E instead of e_i.
A path is called **simple** if it does not contain a vertex more than once, meaning $v_i \neq v_j$ holds if $i \neq j$.

Longest path problem (LP)

The longest path problem for a given graph G := (V, E) and the start and target vertices $s, t \in V$ is to find the longest simple path from s to t. Also, more accurately, called **longest simple path problem (LSP)**. The length of a path is either defined as the number of its edges or the sum of their weight.
Another definition of LP is to find the longest simple path in a graph G between any two of its vertices. Any instance of this definition of LP can also be made into an

instance of the previous definition by introducing a s and t vertex and edges with weight 0 from them to all other vertices in G.

Partitioning

A graph G := (V, E) is partitioned by dividing its vertices V into the subsets V_1,V_2,...,V_k. These subsets are disjointed ($\forall$ i $\neq$ j : $V_i \cap V_j = \emptyset$) and contain all vertices of the graph ($\bigcup_{i=1}^{k} V_i = V$). The subsets are a **partition** of G.

Usually the intent is to distribute the vertices of a graph evenly amongst a certain number of subsets while trying to minimize the number or the combined weight of the edges between vertices of different subsets.

Clique

A subset of an undirected graph's vertices is called a clique if the graph contains an edge between every two distinct vertices of the subset.

Matching

A matching is a subset of the edges of a graph where no two edges have vertices in common.

Array

An array is a collection of elements that can be identified with an index. The size of an array is the number of its elements. $arrayName$[i] describes the (i+1)th element of the array $arrayName$ where i $\in \{0..size - 1\}$. An empty bracket [] behind a variable's name can indicate that it is an array ($arrayName$[]).

Hash table/map

An array basically stores (key, value) pairs, where the key is the index. Arrays can be an ineffective way of storage if the keys do not already represent valid indices. A hash table can be used in these cases. A hash table consists out of a large array of "buckets" and a hash function H. A (key, value) pair is stored in the bucket with the index H(key). A value can then be looked up in the table through its key. With certain assumptions about the hash table, like that the function H almost uniformly and randomly distributes the pairs over the array, the hash table can present an effective way to store and look up arbitrary (key, value) pairs.

2.2 Related Work

The paper with the title "Max Is More than Min: Solving Maximization Problems with Heuristic Search" from Stern, Kiesel, Puzis, Feller and Ruml [SKP$^+$14] mainly focuses on the possibility of applying the algorithms that are normally used to solve the shortest path problem (SP) on the longest path problem (LP). [SKP$^+$14] first makes clear why LP is so difficult compared to SP. Then a number of algorithms are presented that are frequently used to solve SP or other minimization search problem, which are then modified in order to be able to solve LP. The search algorithms can be said to be part of three categories. The, for this thesis, most important category is the one of the heuristic searches. A heuristic can provide extra information about the graph or the type of graph. The heuristic searches of [SKP$^+$14] require a heuristic function that can estimate the remaining length of a solution from a given vertex of the graph. This can give important information that helps to speed up the search depending on the heuristic. It was shown that heuristic searches can be used efficiently for LP. The algorithms Depth-First-Branch-and-Bound (DFBnB) and A* that were modified in order to solve LP and which heuristics were used will be explained in more detail below, since they were used in the experiments of this thesis as comparison. Another category represents the "uninformed" searches, which don't require any information other than what is already given in the definition of the problem.

An example for these algorithms were Dijksra's algorithm or DFBnB without a heuristic. Modifying these algorithms to fit LP basically lead to brute force algorithms, which means that they still had to look at every possible path in the search space. No uninformed search strategy was found that could be used beneficially for LP. The last category are the suboptimal searches. [SKP+14] looked at a large number of these algorithms that only find approximations of a longest path. They are not that important to this thesis since the presented algorithm is an optimal algorithm.
Grids and roads are the two graph types that are also used in this thesis. The only thing that currently matters is that roads-graphs are weighted and grids are not. A road-graph also can only have edges with a positive weight. A more detailed explanation is in section 4.1.1. The heuristic searches of the paper use following heuristic to estimate the remaining length of a longest path from a given vertex.

Grids:
G_v describes a subgraph of G for a path v = (v_1,v_2,...,v_k). G_v only contains the vertices that could be part of a path that starts from v_k and ends in the goal vertex. This path also cannot intersect with v itself (except for v_k). The vertices of G_v can be calculated with a Depth-First-Search starting from v_k. It can easily be seen that G_v for the current search path of an algorithm only contains the vertices that could be part of a longest path (except for v_k). In the best case a longest path starting with v connects all vertices of G_v, which would require $|G_v|$-1 edges. The heuristic returns this value as an estimation of the remaining length of a longest path from v_k, since grids are unweighted/only contain edges of the weight 1. This estimation always upper bounds the highest possible remaining length.

Roads:
A similar heuristic is used for road-graphs. The heuristic function cannot simply return $|G_v|$-1, because the graphs edges are not all weighted 1. Furthermore the edge-weights are always positive. The length of the remaining longest path is estimated by calculating the weight of the maximum spanning tree of G_v. A spanning tree is a subgraph of G_v that represents a connected, acyclic graph that contains all vertices of G_v. This means that there exists one and only one path between any two vertices of the subgraph. The maximum spanning tree is the one with the highest combined weight of its edges. The weight of this maximum spanning tree also represents an estimation that upper bounds the highest possible remaining length of a longest path.

Depth-First-Branch-and-Bound (DFBnB) basically represents a Depth First Search (DFS) that continues after the goal has been found and keeps a record of the best current solution. Once the search is finished the incumbent solution represents the longest path. DFBnB additionally uses a heuristic function $h(.)$ to prune paths during the search. Let $curLength$ be the length of the current search path and $bestLength$ the length of the incumbent solution. The heuristics from above always represent an upper bound to the remaining length of a longest path, which is why every search path with $curLength + h(.) \leqslant bestLength$ does not have to be pursued any further and can be pruned.

A* defines an f value for a path that equals the length of the path plus the value of the heuristic function $h(.)$. A* keeps a priority queue of all "open" paths, which initially contains a single path consisting of the start vertex. In each step A* removes the path (v_1,v_2,...,v_k) with the highest f value, meaning the path with the highest estimated length, from the queue. It inserts a path (v_1,v_2,...,v_k,v_{k+1}) back into the queue, for each vertex v_{k+1} that has an edge to v_k and also isn't already part of the original path. This continues,

if a path to the goal even exists, until a path gets removed from the queue where v_k is the goal vertex. This is a longest path, which is returned as the result of the algorithm. Since the used heuristic's value is 0 for this path, f equals the actual length of the path. All remaining paths in the queue have an f-value equal or lower to this path. They could never result in longer paths and have effectively been pruned since the used heuristic always upper bounds their longest remaining length.

We are not aware of any recent work other than [SKP+14] about this topic.

3. Our Algorithm (LPDP)

Our algorithm will be called "Longest Path Dynamic Programming" or LPDP as it is based on principles of dynamic programming. LPDP solves the longest path problem (LP) for undirected graphs. The graphs are exclusively weighted, since an unweighted, undirected graph can simply be seen as a graph that only contains edges with a weight equal to 1. The start and target vertex are called **s** and **t**. Since

3.1 The Basic Approach

The naive way to solve the longest path problem is something called exhaustive depth-first search by [SKP+14]. Normal depth-first search (DFS) is started from a root vertex in the graph and every visited vertex is marked as such. DFS recursively calls itself for each unmarked vertex that is reachable by an edge from the current vertex v. Additionally v is said to be the parent of these vertices. Once it has done this it backtracks to its parent. The search is finished once DFS tries to backtrack to the parent of the root vertex.
Exhaustive DFS is simply DFS that unmarks a node upon backtracking. In that way every simple path in the graph starting from the root vertex is explored. LP can be solved with exhaustive DFS by starting it from the start vertex. During the search the length of the current search path is stored and compared to the previous best solution if the target vertex is found. If the current length is greater than that of the best solution, it is updated accordingly. If the search is done, a path with maximum length from s to t is found.
The idea of LPDP is to partition the graph G := (V, E) (with V:={0, 1, ..., |V|-1}) into multiple subsets, run a search similar to exhaustive DFS on them and then combine the results into a single longest path for G.

3.1.1 Step 1 - Partitioning and preprocessing

The graph's vertices are partitioned into different subsets like shown in Fig.3.1. Then every edge {x,y} running between different subsets is replaced by introducing a new node k and the edges {x,k} and {k,y} (Fig.3.2). Only one of these edges retains the original's weight, the other's is set to 0. Additionally a new node with an edge to the start vertex is created. This edge also has a weight of 0. The same is done again for the target vertex. All newly generated vertices are called border-nodes, since they, except for the latter two, resemble the borders between different subsets. Now new graphs $G_i := (V_i, E_i)$ can be created with V_i being the vertices of subset i and all border-nodes connected to them and E_i all edges that run between V_i in G. All graphs G_i that are created from the graph in

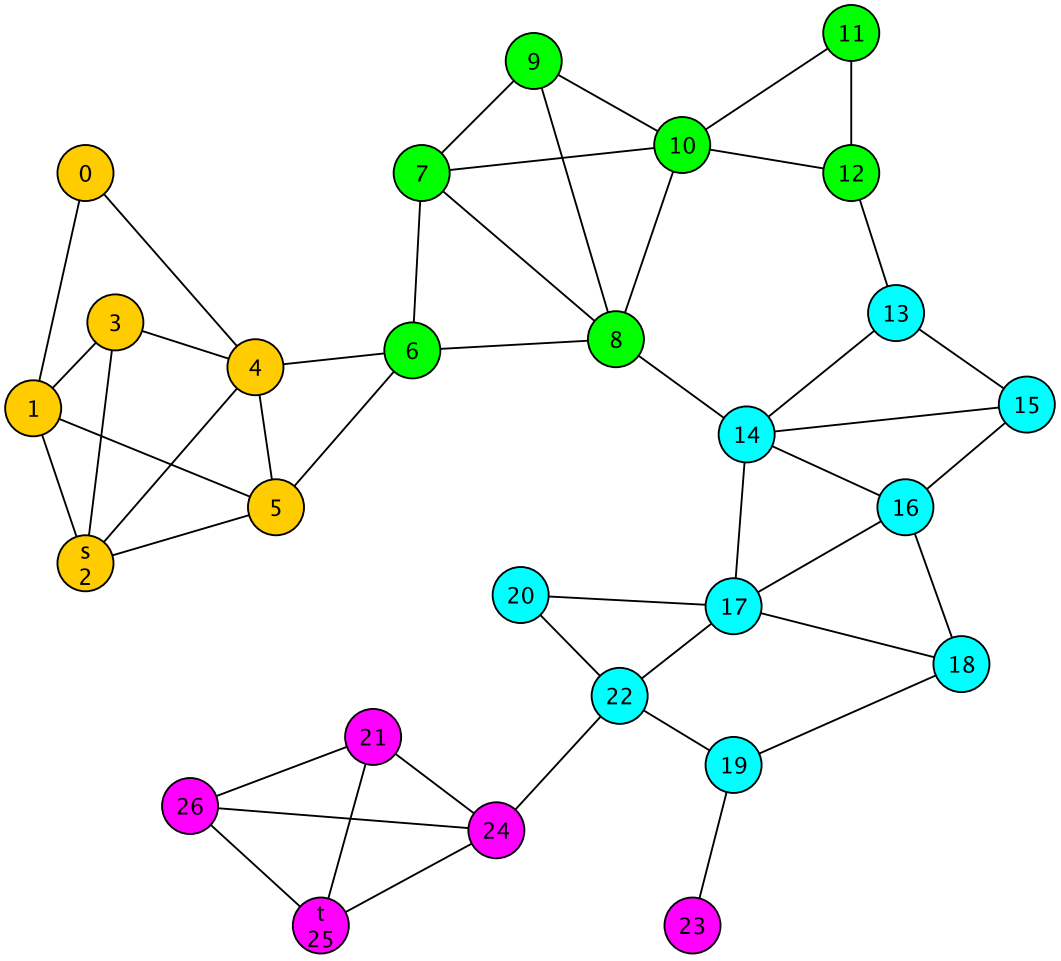

Figure 3.1: Example graph. A partition of the graph is shown by the different colors. An edge's weight is only shown if it connects two different parts of the partition. Vertex 2 and 25 are the start and target vertices for the longest path problem and are marked with the letters s and t.

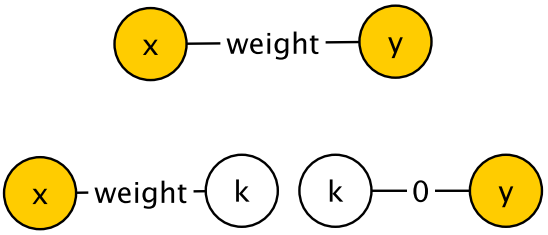

Figure 3.2: Example of how edges that run between different components of a partition are split

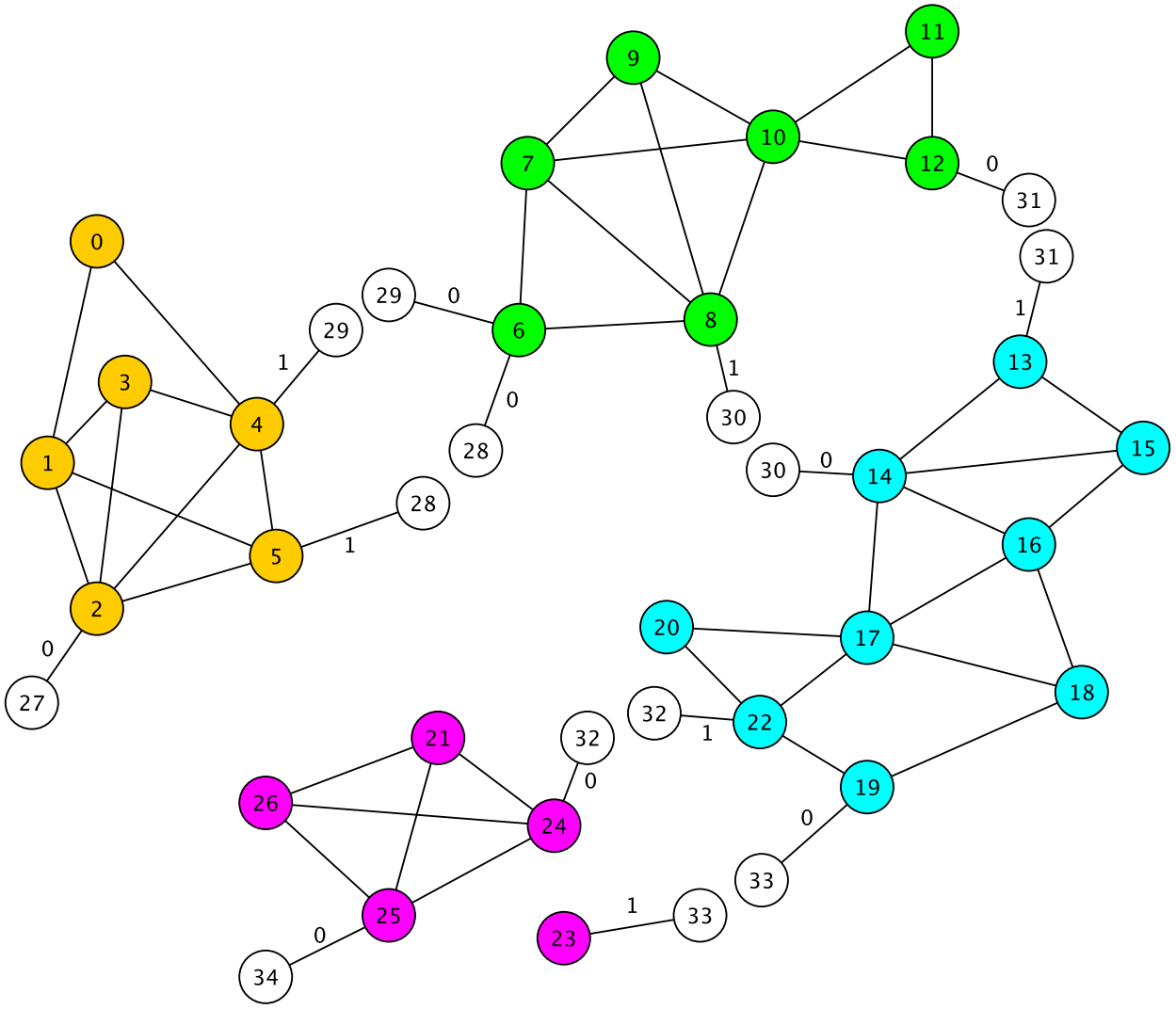

Figure 3.3: Graph from Fig.3.1 split into multiple graphs according to its partition like shown in Fig.3.2

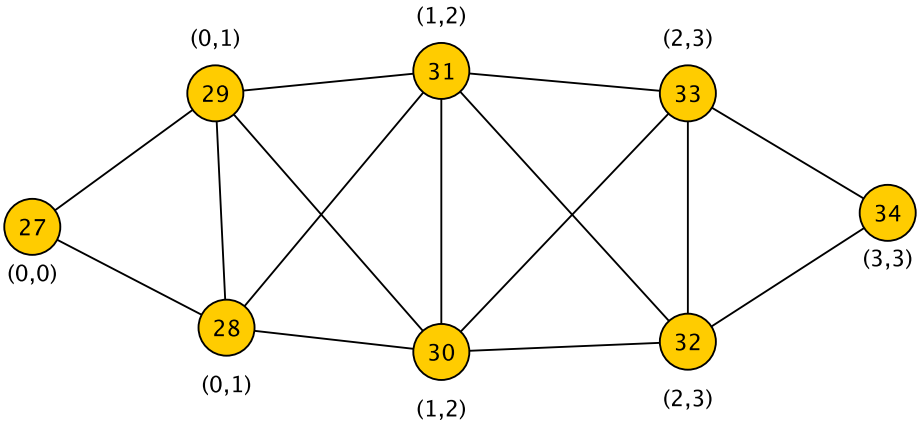

Figure 3.4: Higher level graph. Every vertex v corresponds to the border-node v in Figure 3.3. Every graph G_i is represented as a clique of its border-nodes. A vertex also stands for a connection of two subsets. This connection is shown as a pair (x,y) near the vertex, where x and y are the numbers of the subsets that are connected. The vertices that represent the previous start and target vertices are seen as connecting their subset to itself, which means x = y.

Fig.3.1 are shown in Fig.3.3. As seen the border-nodes have the numbers |V|, |V|+1, ... with the border-node of the start- and target-vertex having the lowest and highest number respectively.
Now we are looking for example at the blue G_i graph in Fig.3.3. For the calculation of the longest simple path in G these border-nodes function as entry and exit points for their subset of the partition. A longest simple path from s to t can only enter and exit this subset through its border-nodes or more specifically the original edge that is associated with this node. Since neither s nor t lie within the blue subset, it is clear that every time the path enters this subset, it also has to leave it again, connecting its border-nodes in pairs of two. This means, without knowing anything about the rest of the graph, that a longest simple path from s to t can connect the border-nodes pairs {{30,31}}, {{30,32}}, {{30,33}}, {{31,32}}, {{31,33}}, {{32,33}}, {{30,31},{32,33}}, {{30,32},{31,33}}, {{30-33},{31,32}} or none of them {}. These sets of border-node-pairs for any subset s are equivalent to the matchings that exist for a clique-graph that consists of the border-nodes of s. The pairs would have to be connected by non-intersecting simple paths. The longest of these connections for each of the cases can be found through a modified version of exhaustive DFS, which will be shown later. The same will be done for all other graphs G_i. The fact that the border-nodes that represent the start- or target-vertex always have to be connected to another border-node won't matter for the algorithm.

3.1.2 Step 2 - Combining the paths

Now it is possible to search for the longest (simple) path of G in a graph that only contains its border-nodes. In this graph two border-nodes are connected with an edge if they belonged to the same graph G_i. Every subset of the original partition is now represented by a clique of its border-nodes. The border-nodes are seen as regular vertices of the new graph. An example of this graph can be seen in Fig.3.4.
Every vertex in this graph represents a connection between two subsets. In the case of start- and target-border-node they will be seen as a connection to their own subset. The connection

of a vertex v is shown in Fig.3.4 as two values (a,b) (=:connection(v)) near the node. An edge {v,w} can be said to be part of a subset s if s ∈ connection(v) ∧ s ∈ connection(w). This edge than represents a path located in subset s or rather in G_s that connects its two corresponding border-nodes. While this condition for an edge could be true for two different subsets, it is only allowed to be part of one subset at the same time. Otherwise the edge would represent a circle.
In order to solve the longest path problem another modified version of exhaustive DFS, which starts from the lowest numbered vertex as it represents the original start vertex, can be used. Broadly speaking this version creates a set of border-node-pairs for every subset s, which is called $Pairs_s$, from its search path. In order to do this it sees every edge in the current search path as a part of a fixed subset/G_i graph. If the edge {v,w} belongs to the search path as part of the subset s, the pair {v,w} is an element of $Pairs_s$. The pair {v,w} ∈ $Pairs_s$ represents a connection of the corresponding border-nodes of v and w in G_s through a simple path. The simple paths of all these pairs in $Pairs_s$ cannot intersect with each other. The best possibility to do this and the combined length of these paths has already been calculated in step 1 for the different G_s. To only receive valid $Pairs_s$ the following conditions are followed while trying to append new edges to the current search path:

- For every subset s a solution for $Pairs_s$ has to exist. This can simply be looked up, since the best possible solutions were already calculated in step 1.
- The new edge has to be part of a different subset than the previous edge. Otherwise {a,b},{b,c} ∈ $Pairs_s$ would be possible, which would mean that the two paths in G_s would intersect since they share the border-node b (and the vertex of G_s that has an edge to b)

Every time the highest numbered vertex, the border-node of the original target-vertex, has been found, the paths in G_s for every $Pairs_s$ are looked up and their weight summed up. At the end the different $Pairs_s$ with the highest combined weight are returned. This weight is the weight/length of the longest simple path in G. The actual longest simple path can be constructed by looking up all the precalculated paths in G_s for the given connections of its border-nodes $Pairs_s$. All of these paths, now called segments, start and end with a border-node. The start- and target-border-node appears exactly once, the others twice. The paths can be concatenated the following way (since G is an undirected graphs, paths can be reversed at will):

- a-v_1-...-v_k-b and b-w_1-...-w_k-c ⇒ a-v_1-...-v_k-w_1-...-w_k-c
- a-v_1-...-v_k-b and c-w_1-...-w_k-b ⇒ a-v_1-...-v_k-w_k-...-w_1-c

This is done until only a single segment a-v_1-...-v_k-b, with a being the start- and b the target-border-node, is left. Now v_1-v_2-...-v_k is a longest simple path in G.

3.1.3 Improvement through hierarchical partitioning

While the calculations for the different G_i graphs maybe can be done fast and even in parallel, the graph of border-nodes from the second step of the algorithm still has to be searched as one complete piece with a variant of the naive brute-force approach. This means that the possible acceleration is relatively limited, once this graph becomes more complex. The next logical step is to try to avoid this by applying the same principles that we used to accelerate exhaustive DFS on this variant of it. Of course it isn't possible to split up edges like on a normal graph. The cliques representing the different subsets have to stay intact. Nodes now serve as dividers instead of edges. New border-nodes are

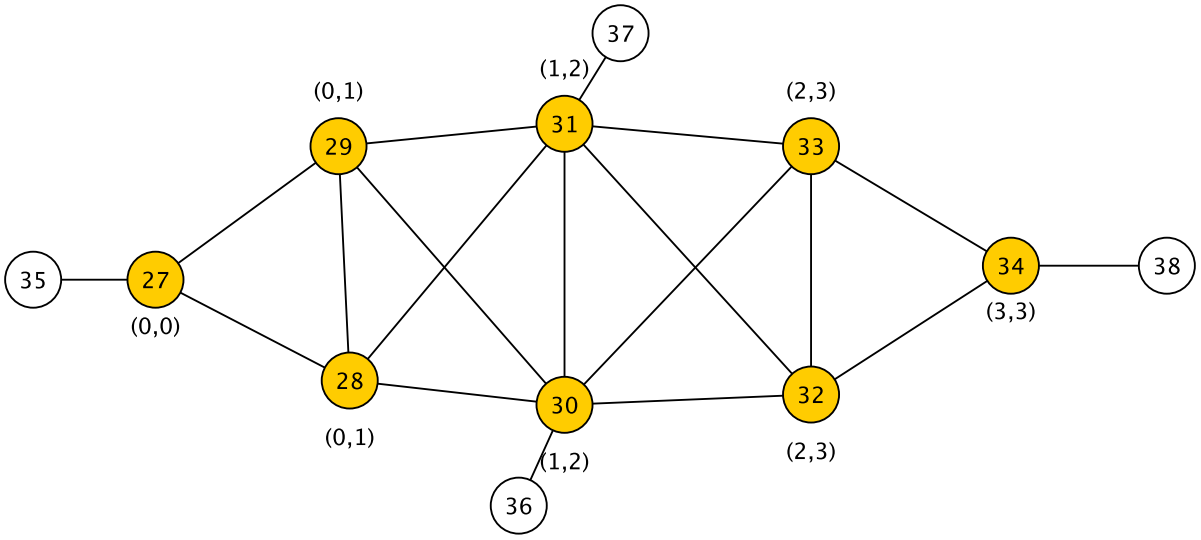

Figure 3.5: Example of how border-nodes would be introduced in the graph of Figure 3.4 in order to create two new subsets according to the next partitioning level. The first is created by combining the lower level subsets 0 and 1. 2 and 3 for the second subset. Since 1 and 2 will be part of different subsets, border-nodes have been introduced for the two vertices that represent a connection (1,2). The same was done for the vertices 27 and 34 since they represent the start and goal vertices.

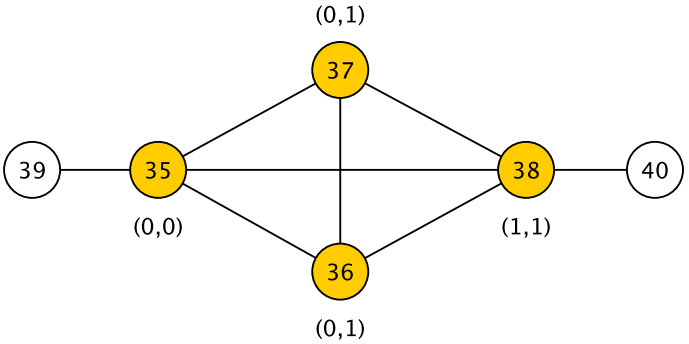

Figure 3.6: Even higher level graph consisting out of the border-nodes of the graph of Figure 3.5. Only two subsets, named 0 and 1, exist. The only two border-nodes are the start and target vertex, combining the last two subsets into one. A search in this graph will determine the longest path and complete the algorithm.

connected directly to them and to the start and target vertices like it is shown in Fig.3.5. The point is to combine a group of subsets and the paths that were calculated for them into a single new subset. This is again done with a variant of exhaustive DFS between the new subsets border-nodes. Fig.3.5 shows the subsets 0 and 1 got combined into one, also 2 and 3. For simplicity we did not actually split the graph into multiple parts like we did with G, but it was made sure that only vertices and edges of the subsets, that are supposed to be combined, are used during the search. When this is done another graph like in Fig.3.6 can be created. In this way Step 2 becomes a recursive call on its own resulting graph. To achieve this the partitioning of the original graph G has to be hierarchical. The partitioning specifies a high number of small subsets that the original graph is partitioned into. These smaller subsets are subsequently combined step by step into larger ones, until only one is left and a longest path of G is calculated.

3.2 Implementation

3.2.1 Data structures

The graph **G = (V, E)** is the given, undirected and weighted graph for the algorithm in which the longest simple path is calculated. As defined previously V is its set of vertices and E its set of edges. The vertices are represented and named sequentially with the numbers 0 to |V|-1. Meaning: V = {0, 1, ..., |V|-1}. **s** and **t** are vertices of G and are the start- and target-node of the longest simple path.
The hierarchical partitioning of G is given through a two-dimensional array **partitions[][]**, which is built up as follows: The first array (partitions[0]) represents the lowest, finest level of partitioning, assigning single vertices to the subsets that the graph is partitioned in. That means if v ∈ V then partitions[0][v] is the number of the subset that v is a part of. These subsets are numbered similar to the vertices with 0,1,2,... . Each higher level of partitions[][] combines the subsets of the previous level. This leads to a hierarchical partitioning of G. Level 0 represents the underlying partition. Its subsets get combined further and further with higher levels, until only a single "subset" is left that contains all vertices. The highest level of partitions[][] always is an array containing a single element 0. For level l the subset that a given vertex v is part of can be calculated with the following function:

$$p(l,v) := \begin{cases} partitions[l][p(l-1,v)] & \text{for } l > 0 \\ partitions[0][v] & \text{for } l = 0 \end{cases}$$

Example used for the graph in Figure 3.1 throughout this thesis:
partitions[0] = [0,0,0,0,0,0,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2,3,2,3,3,3,3]
partitions[1] = [0,0,1,1]
partitions[2] = [0,0]
partitions[3] = [0]

The algorithm uses two major custom data structures. The different paths between the border-nodes of a subset are stored in a map. There is a map for every subset. This data structure is implemented as a hash table and is simply called **map**. This map stores instances of a **Key** and a **Paths** data structure as key-value-pairs.
The **Paths** data structure stores paths between border-nodes, but also has a length attribute for the combined length of all the paths associated with it. How paths are stored changes in the algorithm. In the first step of the algorithm **Paths** stores the actual paths in a subset. A path is stored as the sequence of its vertices, starting and ending in a border-node.
In the second step of the algorithm **Paths** are stored differently. If a step of the algorithm combines subset 1, 3 and 5 into one, it also combines the paths in the maps

```
1: function LPDP(G = (V, E); s,t ∈ V; partitions[][])
2:     initialize G_i graphs, borderNodes and pathMaps
3:     for i ← 0 to partitions[1].size() do
4:         for each bNode ∈ borderNodes[i] do
5:             searchLowestLevel(G_i, borderNodes[i], pathMaps[i], bNode)
6:     Paths result := solveHigherLevels(partitions, borderNodes, pathMaps, 1)
7:     combine paths from result into one valid path of G
8:     return combined path & weight of result
```

Figure 3.7: Basic pseudocode of LPDP. The used variables and methods are explained in the text.

$pathMap_i$ (i ∈ {1, 3, 5}) into one new map $keyMap$. To do this $keyMap$ doesn't save the actual paths, but the identifiers under which the paths can be looked up in the different $pathMap$s. This means **Paths** still stores paths, but only as their identifier or key (**Key**). **Key** is a unique identifier of a **Paths** instance. **Key** is a sequence of nodes that resembles the $Pairs_s$ from the section above, meaning the set of border-node-pairs that are connected through the **Paths** instance. The elements of the sequence **Key**[2i] and **Key**[2i+1] resemble a pair {**Key**[2i],**Key**[2i+1]} ∈ $Pairs_s$ for all possible i ≥ 0. To keep **Key** unique, one could for example swap **Key**[2i] and **Key**[2i+1] without changing the $Pairs_s$ set it resembles, the following conditions are introduced for a valid **Key** instance:

- **Key** has to have an even number of elements (0 included)
- **Key**[i] ≠ **Key**[j] for i ≠ j
- **Key**[2i] < **Key**[2i+1] for i ≥ 0
- **Key**[2i] < **Key**[2i+2] for i ≥ 0

3.2.2 Pseudocode

The first part of the algorithm (Figure 3.7) splits the given graph G into the different G_i graphs, based on the partitions[0] array, like it was explained in section3.1.1 and shown in Fig.3.1 and Fig.3.3. borderNodes[i] describes an array of the border-nodes of G_i in ascending order. It then calls the searchLowestLevel() function for every border-node of a graph G_i, which completes the first step of the algorithm (section 3.1.1) by filling up the graphs pathMap. Once this has been done for all G_i, the second step of the "improved" algorithm (section 3.1.2 and 3.1.3) is executed by calling the solveHigherLevels() function. Its result is a set of path segments selected from all pathMaps that can form a longest simple path in G. Additionally the result contains the combined weight of all segments. All of this can also be saved in the **Paths** data structure. These path segments are then combined into the single path in G as shown at the end of section 3.1.2.

searchLowestLevel() describes a modification of exhaustive DFS, that is run with a border-node as root. The search algorithm divides its current search path into different path segments. It traverses the vertices of the graph as normal with the exception of the border-nodes. The first segments starts from the root border-node. The segment is completed once a different border-node is reached. If this happens, the algorithm starts a new segment by jumping to an other border-node, as if they were connected by an edge, and continues traversing the graph as before. This way each segments starts and ends in a border-node. The start- and endpoints of all segments resemble the border-node pairs

Key	**Paths**	
	Segments/Nodes	Length
(30,31)	(30,14,17,20,22,19,18,16,15,13,31)	9
(30,32)	(30,14,13,15,16,17,18,19,22,32)	8
(30,33)	(30,14,13,15,16,18,17,20,22,19,32)	8
(31,32)	(31,13,14,15,16,17,18,19,22,32)	9
(31,33)	(31,13,14,15,16,18,17,20,22,19,33)	9
(32,33)	(32,22,20,17,14,13,15,16,18,19,33)	9
(30,31,32,33)	(30,14,15,13,31) (32,22,20,17,16,18,19,33)	9
(30,32,31,33)	(30,14,17,20,22,32) (31,13,15,16,18,19,33)	9

Table 3.1: Example of a finished $pathMap$ of the blue subset from Figure 3.3

$Pairs_i$. The best current result for each possible $Pairs_i$ is stored and updated if necessary every time a path segment is completed. To avoid unnecessary traversal of the graph a path segment is only allowed to end in a node higher than its start. Additionally a path segment can only start from a border-node, if this node is higher than all other starting nodes in the current search path. This way the border-nodes in the search path also automatically induce a valid **Key** instance, which is unique for every possible $Pairs_i$. The longest way of connecting each possible $Pairs_i$ can be looked up in the $pathMap$ under its corresponding **Key**, once solveLowestLevel() is completed. An example of the $pathMap$ for the blue subset of the graph in Figure 3.3 can be seen in Table 3.1.

3.2.2.1 Solving the higher levels

The self-recursive method seen in Figure 3.8 takes the $pathMaps[]$ calculated for the subsets of lowest partitioning level 0 and combines them step by step until only one subset/map is left. The method is initially called with $level = 1$, upon which it combines the results of level 0 into a valid result for the partition given through the array partitions[1]. After this the method increases the *level* by 1 and recursively calls itself, until only one map is left. level $\in \{1,2,...\}$. $partitionNodes[][]$ is the $borderNodes[][]$ array of the previous level. This means that $partitionNodes$[i] is an array that contains all border-nodes of the subset i (in ascending order).
The method creates a new graph G := (V, E) as follows:
V is the set of all vertices that occur in $partitionNodes[][]$. An edge between two vertices only exists if an i exists, where $partitionNodes[i]$ contains both vertices. E := { {v,w} $\in$ V^2 | $\exists$ i : v,w $\in$ partitionNodes[i] }
The currently lowest and highest vertex are representatives of the start- and goal-vertex. Every vertex, except these two, is an element of exactly two subsets i and j. This means that the vertex stands for a connecting edge between those two subsets. This connection is shown in the example graph in Figure 3.5 as the pair (i,j). A start- or goal-vertex only occurs in a single $partitionNodes$[i], which is represented as a connection (i,i). The connection for every vertex v is stored in $connectionsPerNode$[v]. Now new border-nodes are introduced. A new border-node with an edge to the start-vertex is created. Then the same is done for the goal-vertex and every vertex with the connection (i,j), where $partitions[level][i] \neq partitions[level][j]$. These border-nodes are numbered in such a way that the lowest border-node is the one that is connected to the start-vertex and the highest border-node connected to the goal-vertex.
searchHighLevel() is version of searchLowestLevel() that is slightly modified in order to work on the higher level graphs. searchHighLevel() can only search the graph in a way

```
1:  function SOLVEHIGHERLEVELS(partitions[][], partitionNodes[][], pathMaps[], level)
2:      if pathMaps.size() == 1 then
3:          if pathMaps[0] is not empty then
4:              it can only contain a single entry (Key,Paths)
5:              return the Paths instance
6:          else
7:              return empty Paths instance with weight = 0
8:
9:      initialize graph G, borderNodes and keyMaps
10:     for all i ← 0 to partitions[level+1].size() do
11:         for each bNode ∈ borderNodes[i] do
12:             SEARCHHIGHLEVEL(G, borderNodes[i], pathMaps, keyMaps[i], bNode)
13:
14:     Paths keys := SOLVEHIGHERLEVELS(partitions, borderNodes, keyMaps, level+1)
15:     Paths result
16:     result.weight = keys.weight
17:     for i ← 0 to keys.size() do
18:         for j ← 0 to keyMaps[i].get(keys[i]).size() do
19:             if keyMaps[i].get(keys[i])[j] not empty then
20:                 result[j] = keyMaps[i].get(keys[i])[j]
21:     return result
```

Figure 3.8: solveHigherLevels() recursively combines the paths that were calculated for the partition of the lowest level, finds the longest path and returns it. The used variables and methods are explained in the text.

Key	**Paths**		
	Keys	Length	
(35,36)	(27,28)	6	
	(28,30)	5	=11
	()	0	
	()	0	
(35,37)	(27,28)	6	
	(28,31)	6	=12
	()	0	
	()	0	
(36,37)	()	0	
	(30,31)	7	=7
	()	0	
	()	0	

Key	**Paths**		
	Keys	Length	
(36,37)	()	0	
	()	0	
	(30,31)	9	=9
	()	0	
(36,38)	()	0	
	()	0	
	(30,32)	8	=11
	(32,34)	3	
(37,38)	()	0	
	()	0	
	(31,32)	9	=12
	(32,34)	3	

Table 3.2: Example of the *keyMap*s from Figure 3.5. The left table is *keyMaps*[0] and combines the *pathMap*s 0 and 1 of the previous level. The right table shows *keyMaps*[1], which combines the subsets/*pathMap*s 2 and 3. pathMaps[3] can be seen in Table 3.1

that induces correct paths in the original, underlying graph. To achieve this the edges of the graph have to be treated differently. The following only applies to edges {v,w} where neither v nor w is a border-node, while all other edges are treated as before.
An edge {v,w} can be said to be part of a subset s if s $\in$ *connectionsPerNode*[v] $\wedge$ s $\in$ *connectionsPerNode*[w]. This edge than represents a path located in subset s that connects the two corresponding border-nodes of the graph that lies one level below. Each edge in the current search path has to be assigned a valid subset. It also has to be made sure that two successive edges in the search path cannot be part of the same subset. Since the search is done on the whole graph and not on separate parts like with searchLowestLevel(), one also has to make sure that edges can only be assigned to those subsets that searchHighLevel() is currently trying to combine. Section 3.3 explains how it could be done in a different manner, but this is the way the algorithm was implemented.
Additionally the **Key**-**Paths**-*map*, called *keyMap*, that searchHighLevel() creates is build up differently. While the **Key**s are created as before, **Paths** now contains **Key**s of *pathMaps* from the previous partitioning level. If the previous level had n *pathMap*s every **Paths** value in the *keyMap* consists of n different segments. The xth segment represents a **Key** in the xth *pathMap* or subset. As before the **Key** of a subset s represents the connected border-node-pairs ($Pairs_s$). During the search $Pairs_s$ equals the set of all edges in the search path that are assigned to subset s. Additionally a search path does not have to be further pursued once a subset s exists whose *pathMap* doesn't contain a entry for the current $Pairs_s$. If it is impossible to connect $Pairs_s$ in subset s, it will also be impossible for all pairs X with $Pairs_s \subset$ X.
An example for the two *keyMap*s of Figure 3.5 can be seen in Table 3.2

3.3 Parallelization

This section only contains considerations and thoughts for the future. They or parallelization in general were not actually implemented.
The way it is currently presented, the algorithm could be easily parallelized by simply parallelizing the loops that call the *searchLowestLevel*()/*searchHighLevel*() methods for the different subsets. This is because all subsets of a level can be solved independently from each other. This could also be done without having to know the complete graph. At

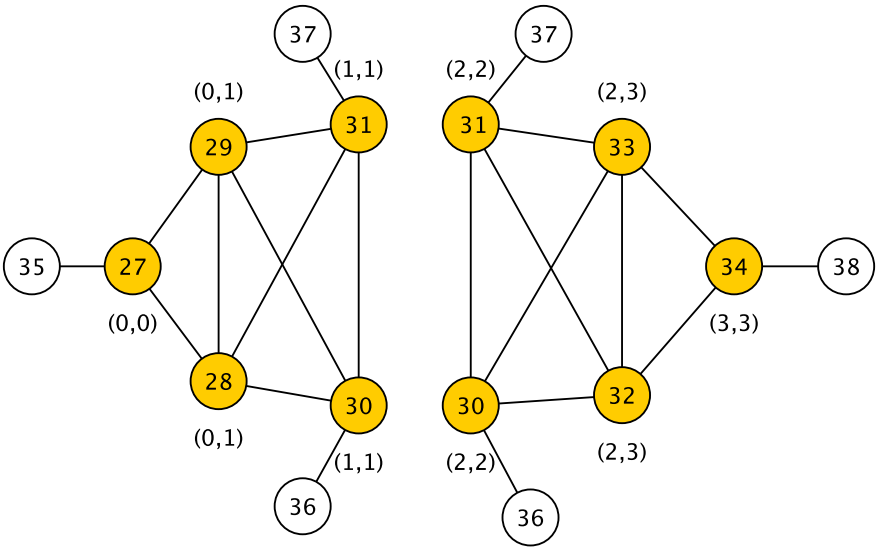

Figure 3.9: Example of how the graph from Figure 3.5 could be split into separate graphs, where each one represents a new subset. The left one combines the subset 0 and 1 into one. The right does the same with 2 and 3. The vertices that they have in common and the edges between those vertices have been duplicated. These vertices also had their (x,y)-pairs adjusted. The duplicate of the vertex in the graph that combines x now connects, similar to start- or goal-vertices, (x,x). A (y,y) connection is ascribed to the equivalent of this vertex in the other graph.

the lowest level the graph was already split into a G_i-graph for each subset i, which is the only thing that is needed to gather the possible paths for a subset. This wasn't done for the higher partitioning levels. The algorithm here and the implementation of it used for the experiments keep a single graph for all subsets of a higher level and simply check each time an edge is supposed to be traversed if it doesn't lead out of the current subset. Splitting these graphs could be done in a way that can be seen in Figure 3.9.
The current algorithm also has to wait for all the calculations of a level to finish until the next can be started. It could be changed in a way that would allow a higher level subset to be calculated once the subsets that it is combined from are finished, instead of having to wait for all of them.

3.4 Partitioning procedure and the xN-solver

xN (where N is a number, which will be explained later) will describe our LP-solver that was created based on the LPDP algorithm. LPDP needs a previous partitioning of the graph. This was done with a bottom up approach and KaHIP - Karlsruhe High Quality Partitioning - [SS13]. KaHIP partitions a graph G = (V, E) into a given number of subsets n. It tries to create subsets of a similar size ($\frac{|V|}{n}$), while trying to minimize the total number of edges running between different subsets or these edge's combined weight. How the number of subsets n was chosen will be explained later in the experiments. After the original graph is partitioned, we combine its subsets step by step, also with the help of KaHIP. For this a graph that represents the partition is created. Each of its vertices represents a subset. Edges between two subsets are represented by an edge between their corresponding vertices in the new graph. The weight of this edge is the total number of edges between the subsets in the original graph. This new graph is again partitioned by KaHIP. After that a new graph is created in a similar manner as above and the process is

repeated. This is done until only one partition is left. This process induces the hierarchical partitioning that is needed for the algorithm. While the original graph is partitioned in n subsets, the others are always partitioned into $\frac{numberOfNodes}{2}$ subsets. This means that the total number of subsets gets halved with every step and that most of the subsets of a higher level get created by combining 2 subsets of the lower level. Halving the number of subsets with every step gave decent results in almost all previous tests, which is why it was used for all experiments. In this way the only change between the different experiments is the starting number of subsets n that the solver uses. Additionally, KaHIP has an option to set a time limit that it will spend to search for a partition. Giving a higher time limit than the normal runtime of the KaHIP-call should give a partition of a higher quality. In the following experiments the time limit is given as a multiple of the standard KaHIP-call "xN", which means N-times as much time as a standard call. Thus the partitioning procedure of the "xN"-solver simply calls KaHIP once without time limit and measures its runtime. The KaHIP-call that actually determines the partitioning is then simply run with a time limit of "N * measured runtime". This of course isn't the case for the solver without time limit, called "x1".

4. Evaluation

4.1 Experiments

The experiments were run on computers that had two Intel® Xeon® Processors X5355 (2.66 GHz with 4 cores) and 24 GB RAM. The computers ran the 64-bit version of Ubuntu 14.04.4 LTS.

4.1.1 Benchmarks

4.1.1.1 Grids

A grid represents a maze like it is shown in Fig 4.1. The maze is a NxN grid of square fields with a given start and target field. One can only move to adjacent fields horizontally or vertically. Fields of the maze can be obstacles that cannot be navigated through. They are marked as black in Fig 4.1. The goal is to find the longest simple path from the start field to the target field. A grid can be represented by a so called grid graph. It contains a vertex for every free field in the grid. There exists an edge between any two vertices, whose fields in the grid are horizontally or vertically adjacent to each other. All edges have a weight of 1.
In [SKP+14] Stern, Kiesel, Puzis, Feller and Ruml used grids for some of their benchmarks and had solvers for them. Grids were also used here. Not only to allow an easy comparison to [SKP+14], but also since it was assumed that grids could be easily partitioned in a way that is suited for the LPDP algorithm. The grids for these experiments were also generated in the same manner. The top left and bottom right field are the start and target fields. Then random fields of the grid are consecutively made into obstacles until a certain percentage of all fields is reached. Afterwards a path between the start and target is searched for to make sure that a solution of the longest path problem exists.

4.1.1.2 Roads

Roads are subgraphs of a large weighted graph that represents the road network of the USA. These are also adopted from the benchmarks of [SKP+14]. They are created by appointing a random vertex as start-vertex and part of the road instance. A breadth-first search from the start-vertex finds other members of the road instance until a certain number of vertices is reached. One of them is declared the target-vertex. The road-instance for the longest path problem consists of these vertices and all the edges that run between them in the road network graph.

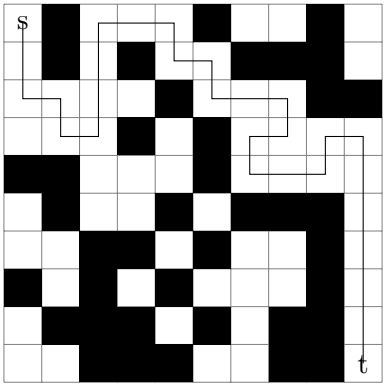

Figure 4.1: An example of a 10x10 grid. 40% of its fields are obstacles (black). s and t are the start and target vertices. The longest path between them has a length of 28.

4.1.2 Used plots and tables

The following is an explanation of the plots and tables that are used to illustrate the results of the experiments.

Scatter plot:
See Figure 4.3 for an example.
Scatter plots were used to compare the runtimes of two different solvers. The horizontal axis at the bottom is the x-axis and shows the time scale for the first solver. The vertical (y-)axis to the left does the same for a second solver. The axes are restricted to a time interval [a, 600 sec] (with $a < 600$ seconds). a can differ per axis. 600 seconds or 10 minutes is the time limit used for the experiments. A problem was considered to be unsolved by a certain solver if it took longer. The time limits form the right and upper border of the plot. A mark with the coordinates (x,y) in this coordinate system stands for a problem that the first solver solved in the time x and the other in the time y. If x or y is 600 seconds, meaning directly on the upper or right border, it is unsolved by the respective solver. Additionally the line x = y is plotted, which divides the plot into two parts. Each mark (x,y) in the top part represents a problem where the x-axis-solver was superior, since x < y. The opposite is true for the bottom part.

Cactus plot:
See Figure 4.2 for an example.
A cactus plot has an axis for the number of problems in the experiment, which is plotted against the runtime-axis. The plot shows the runtimes of the problems, which were sorted in ascending order. The point (x, t) on a curve means that the xth fastest solved problem was solved with the runtime t. Problems that were not solved within the time limit are not shown.

Table of runtime averages:
See Table 4.1 for an example.
This table shows runtime averages of NxN grid graphs. The first column to the left contains values of N. The first row from the top contains the name of the algorithms. Any other field of the table that is in column c and row r corresponds to the size class NxN with N being given in row r of the first column and the algorithm given in column c of the first row. The field contains two values. The first value is the runtime average for all grids of size NxN that could be solved with the corresponding algorithm within the time limit of

10 minutes. The other value is the number of these problems that could be solved by the algorithm.

4.1.3 Solvers

The **xN** solvers work as specified in the section 3.4, which mentions the number of subsets n that the graph is partitioned in. The variable *nodesPerSubset* describes the average number of nodes that a subset should consist of. Once we choose this number, n can be calculated with $n = \frac{|V|}{nodesPerSubset}$. *nodesPerSubset* was simply set to 10 for all of the road-graphs. The *nodesPerSubset* for grids were calculated with an interpolation between two given values based on the grid's "edge-density" compared to an obstacle-free grid of the same size. This led to about 15 nodes per subset for grids with 40% obstacles and 12/13 for grids with 30%.
Partitioning the grids with less obstacles into smaller subsets stems from the idea that these grids contains larger, in both dimensions more "open" regions, which quickly increase the number of possible paths through these areas. Decreasing the size of the subsets could at least limit the calculation time of the lowest level partition. Subsets of the same size in a grid with a higher percentage of obstacles mostly should have fewer edges between vertices and also fewer edges connecting it with the rest of the grid, which would allow these grids to be partitioned into larger subsets. Aside from this the exact numbers that have been chosen as subset-sizes have also been shown to lead to decent results in early testing. This was less of the case for road-graphs, since a good n-value was harder to determine.

Exhaustive DFS is the naive brute-force approach that was talked about in 3.1. This algorithm simply looks at all simple paths starting from the start-vertex and returns the path with the highest cost ending in the target-vertex. The implementation of the solver was taken from [SKP+14].

The **A***- and **DFBnB**-solvers also were taken from [SKP+14] and represent heuristic searches. They and their heuristics are explained in section 2.2.

4.1.4 Results

4.1.4.1 Grids (40%)

These experiments are NxN grids with 40% of its fields being obstacles. The benchmark's data consists of the grid sizes 10x10, 15x15, ..., 120x120 with N increasing in steps of five. There are 10 grid instances per size class, which leads to 230 grids in total.
First the LPDP-solver was tested with the different partitioning times x1, x5, x10, x15 and x20. This was done to evaluate the performance of the algorithm with higher quality partition (in respect to the partitioning procedure and its parameters). The runtime of the solver for the grid instances that were solved within an upper time limit of 10 minutes or 600 seconds per instance can be seen in the first plot of Fig.4.2 in ascending order. The second plot of Fig.4.2 only shows the runtime of the actual LPDP-algorithm (without counting the partitioning time). A trend can be seen in Figure 4.2 that, maybe except for the x15-results, an increased time spent on partitioning results in a faster runtime for the algorithm. This validates the used partitioning procedure to some degree. It can be assumed from Figure 4.2 that a higher partitioning time only yields a profit for the overall runtime of the solver with larger or in general more difficult grids, while the basic solver (x1) is faster for a majority of the problems. Additionally it can be seen that the xN solvers with a higher N-number only have a slowly increasing runtime with sudden peaks at the end. Since the runtimes are ordered in ascending order, these peaks could simply be

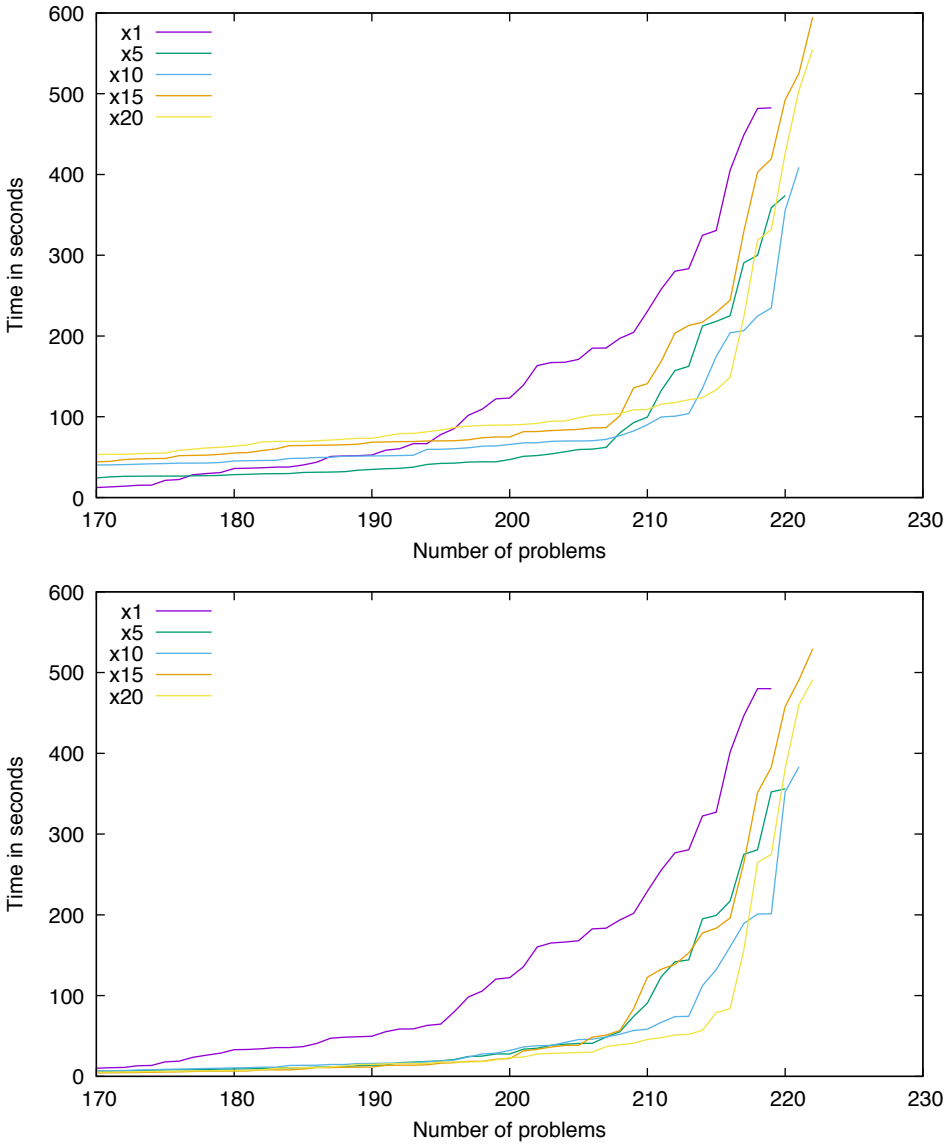

Figure 4.2: Cactus plots (see section 4.1.2) for the 230 grid problems that were solved with the LPDP-solver. The different curves show the runtime with different partitioning times that the solver used. The first plot shows the complete runtime (including the time spent partitioning). The second plot is only the runtime of the actual algorithm as if the partitioning was already given (excluding the partitioning time).

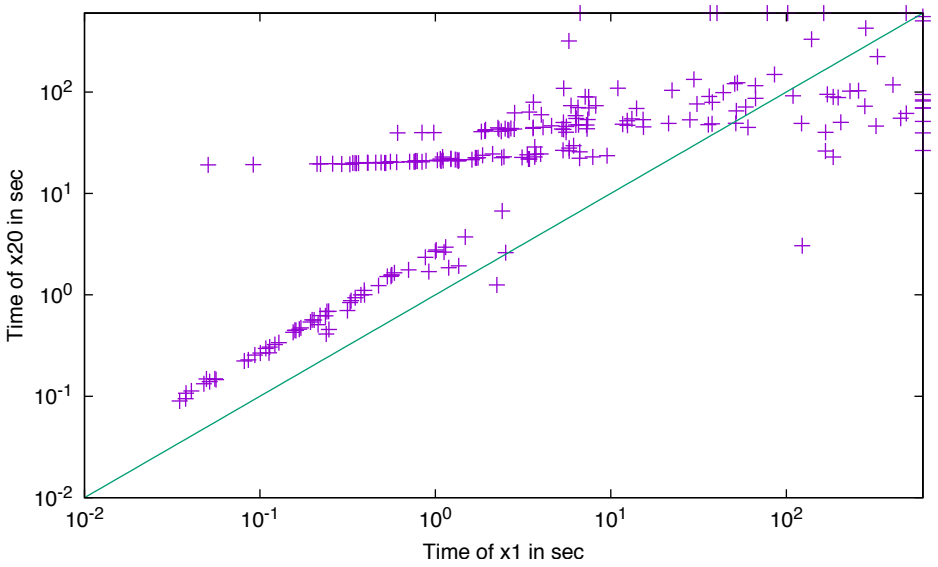

Figure 4.3: Scatter plot (see section 4.1.2) of the x1 and x20 for the grid (40%) instances.

outliers. An outlier could either be a grid that simply is especially difficult to solve by the algorithm compared to most randomly generated instances of the same size or a grid that could be solved faster through the right partitioning, but the used partitioning procedure was ill-suited.

Figure 4.3 is used to further examine the runtime difference between different partitioning times. Figure 4.3 compares x1 and x20. Each mark in the top left half of the plot shows a grid that x1 could solve faster than x20. It is the other way around for the bottom right half. This figure clearly shows that the runtime profit of x20 only exists for high runtimes and with this larger more difficult grids. This acceleration might not look that impressive because of the logarithmic scale, but still means that x20 could solve a lot of instances in under 100 seconds that took x1 between 100 and 600 seconds. In general it has to be said that the results are still pretty similar and the overall number of solved instances didn't change that much. This can be seen more clearly in the following table:

solver	x1	x5	x10	x15	x20	A*	DFBnB	exh.DFS
solved problems	220	221	222	223	223	24	24	20

This difference in 3 grids between x1 and x20 can also be seen more clearly in Fig.4.3. Markings at the top and/or right border represent grids that couldn't be solved within the time limit of 600 seconds. x20 was able to solve 10 more grids in time, because of the better partitioning, while it also couldn't solve 7 grids that x1 could, because the acceleration of the actual algorithm didn't make up for the additional time spent partitioning.
Furthermore Figure 4.3 doesn't have a mark on the intersection of the top and right border. Which means that x1 and x20 together were able to solve all grids of this benchmark set, proving that the algorithm would be able to solve all instances with the right partitioning procedure.

Figure 4.4 shows the runtimes of the x1 solver and the optimal algorithms from [SKP+14] in ascending order. x1 not only solves far more instances than the other solvers, but its runtime also increases slower with larger grid sizes. A* and DFBnB seem to perform almost the same and only slightly better than the "naive" brute force approach of exhaustive DFS.

Additional information about the runtime of the algorithms can be seen in Table 4.1.

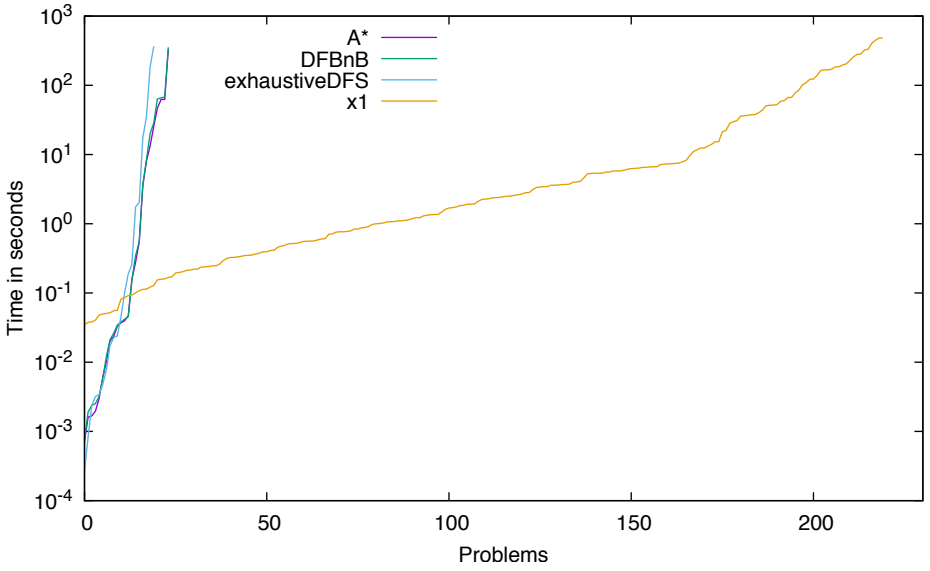

Figure 4.4: Cactus plot (see section 4.1.2) for the different optimal algorithms that were evaluated. The LPDP-solver is only represented with x1, as the other xN are pretty similar to it and would only reduce the clarity of the plot. The runtime of x1 includes the time that was spent partitioning the graph.

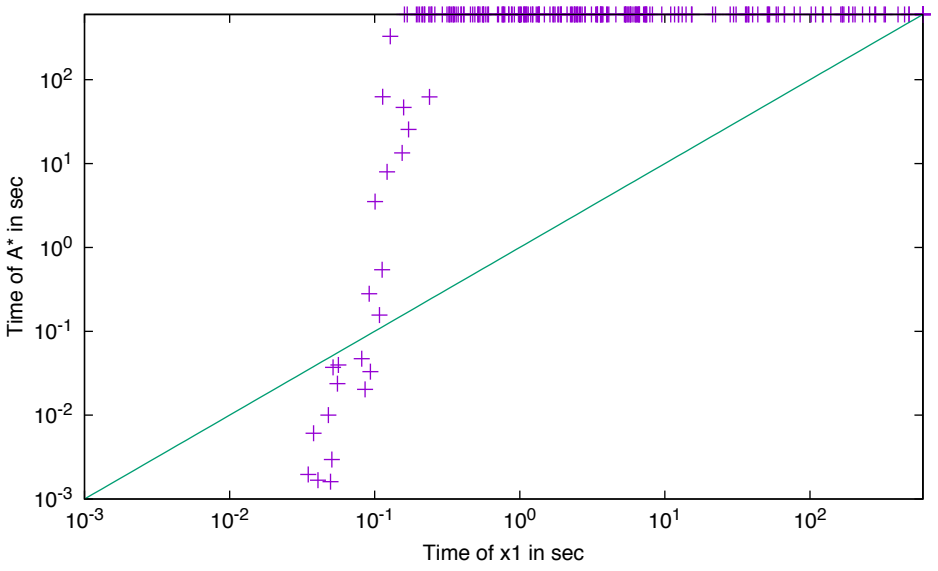

Figure 4.5: Scatter plot (see section 4.1.2) of the x1 and A* for the grid (40%) instances.

	x1	x5	x10	x15	x20	A*	DFBnB	exh.DFS
10	0,046251 10	0,125383 10	1,02198 10	0,125245 10	2,02575 10	0,0125595 10	0,0135098 10	0,0164589 10
15	0,103686 10	0,273619 10	0,299249 10	0,274637 10	2,17323 10	40,4125 10	43,4765 10	46,9462 9
20	0,212964 10	0,487714 10	1,38576 10	0,455555 10	4,24751 10	37,0106 4	44,4601 4	186,371 1
25	0,223155 10	2,19029 10	2,40384 10	6,19008 10	6,30692 10	N/A 0	N/A 0	N/A 0
30	0,349825 10	2,09939 10	4,50234 10	5,11015 10	8,49127 10	N/A 0	N/A 0	N/A 0
35	0,642843 10	3,17047 10	6,54547 10	8,11579 10	14,4023 10	N/A 0	N/A 0	N/A 0
40	0,624812 10	3,90416 10	5,12334 10	9,96432 10	11,0545 10	N/A 0	N/A 0	N/A 0
45	1,06923 10	3,503 10	5,51139 10	13,0241 10	13,2416 10	N/A 0	N/A 0	N/A 0
50	1,1688 10	6,80114 10	10,7783 10	14,8612 10	21,6473 10	N/A 0	N/A 0	N/A 0
55	1,24492 10	5,95532 10	7,37345 10	13,7989 10	17,8037 10	N/A 0	N/A 0	N/A 0
60	15,2127 10	11,7823 10	15,7523 10	14,3927 10	21,6463 10	N/A 0	N/A 0	N/A 0
65	22,6005 10	48,0453 9	51,5173 10	23,9584 9	30,9666 9	N/A 0	N/A 0	N/A 0
70	44,2266 10	38,1816 10	28,3478 10	31,9899 10	39,0062 10	N/A 0	N/A 0	N/A 0
75	52,7414 10	12,0956 9	23,2856 10	27,0736 9	38,145 8	N/A 0	N/A 0	N/A 0
80	3,2441 10	11,4768 10	23,3298 10	34,6903 10	41,3709 10	N/A 0	N/A 0	N/A 0
85	25,5702 9	39,3021 9	39,4046 10	45,3084 10	48,0658 10	N/A 0	N/A 0	N/A 0
90	35,1536 9	25,3597 10	56,48 9	116,298 10	80,267 10	N/A 0	N/A 0	N/A 0
95	108,499 10	47,3925 10	60,8619 10	69,8176 9	53,2011 10	N/A 0	N/A 0	N/A 0
100	130,142 9	59,9476 9	59,8605 9	119,78 9	112,296 9	N/A 0	N/A 0	N/A 0
105	62,2706 8	50,0802 9	114,649 8	95,1448 8	165,909 9	N/A 0	N/A 0	N/A 0
110	36,1491 8	54,6478 10	46,2932 9	82,5595 10	72,3537 10	N/A 0	N/A 0	N/A 0
115	118,364 7	107,262 9	56,3912 9	163,928 9	103,418 9	N/A 0	N/A 0	N/A 0
120	118,496 10	48,0512 7	88,904 8	113,035 10	129,429 9	N/A 0	N/A 0	N/A 0

Table 4.1: Table of runtime averages (see section 4.1.2) for all NxN-grid (40%) instances and the different algorithms.

4.1.4.2 Grids (30%)

This benchmark is similar to the one before, but used grids that had 30% of its fields converted to obstacles instead. This results in a higher number of possible paths compared to the previous benchmark's grids of the same size. The benchmark only consists of the grid sizes 10x10, 15x15, ..., 40x40 because of this. There are again 10 grids per size class, which leads to 70 grids in total.
First the LPDP-solver was tested with the different partitioning times x1, x5, x10, x15 and x20. This was done to evaluate the performance of the algorithm with higher quality partition (in respect to the partitioning procedure and its parameters). The runtime of the solver for the grid instances that were solved within an upper time limit of 10 minutes or 600 seconds per instance can be seen in the first plot of Fig.4.6 in ascending order. The second plot of Fig.4.6 only shows the runtime of the actual LPDP-algorithm (without counting the partitioning time). These figures show very similar results compared to the previous benchmark with 40% grids. Higher partitioning time tends to slightly accelerate the algorithm, but there are no major changes in the overall runtime of the solver. It can also be seen that the 30% grids are much harder to solve than the previous ones, since the xN-solvers could still solve 120x120 grids in the previous benchmark, while they are now having trouble with 40x40 sized grids. The number of solved problems for the tested algorithms can be seen in the following table.

solver	x1	x5	x10	x15	x20	A*	DFBnB	exh.DFS
solved problems	54	61	63	60	62	11	11	10

Increased partitioning time made the algorithm solve a lot more problems compared to the previous benchmark. The x1 solver was only able to solve 54 of 70 problems. This number increased by 6-9 with longer partitioning times. This stands in contrast to the 220 of 230 problems that could be solved in the previous benchmark, where higher partitioning times only increased this number by 1-3. This could be because the quality of the partitioning is far more important for grids with a lower percentage of obstacles and with this also for graphs with a higher density of edges in general. Another possibility would be that the maximal grid size of 120x120 was not large enough and the 10 unsolved problems from previous benchmark were outliers, especially hard grids were better partitioning wouldn't necessarily make a difference. Maybe the higher partitioning times would make more of a difference for even larger grids. A point that speaks against this is that x1 and x20 together were able to solve all 230 problems of the previous benchmark. This means that the longer partitioning times did make a lot more instances solvable (than just 1-3), while others were rendered unsolvable since the partitioning took much longer, while only insufficiently accelerating the algorithm.

Figure 4.7 again illustrates the runtimes for different optimal algorithms. The LPDP-algorithm is represented by x10 since it solved the most problems. The x10 solver is again much better than all other algorithms from [SKP+14]. A* and DFBnB perform even more similar than in the previous benchmark. It can be seen in the [SKP+14] paper that A* and DFBnB are related algorithms. A* basically is a Best-First-Branch-and-Bound algorithm compared to Depth-First-Branch-and-Bound (DFBnB). While they work in a different manner, they most likely traversed nearly all the same paths at the end of the search. Something noteworthy, which is further shown in Figure 4.8, is that A* (and also DFBnB) exclusively performed better than the brute force approach that is exhaustive DFS. Previously exh.DFS was faster for the smallest grid sizes. Now this has changed, further showing the difficulty of these grids compared to the previous ones.

Additional information about the runtime of the algorithms can be seen in Table 4.2.

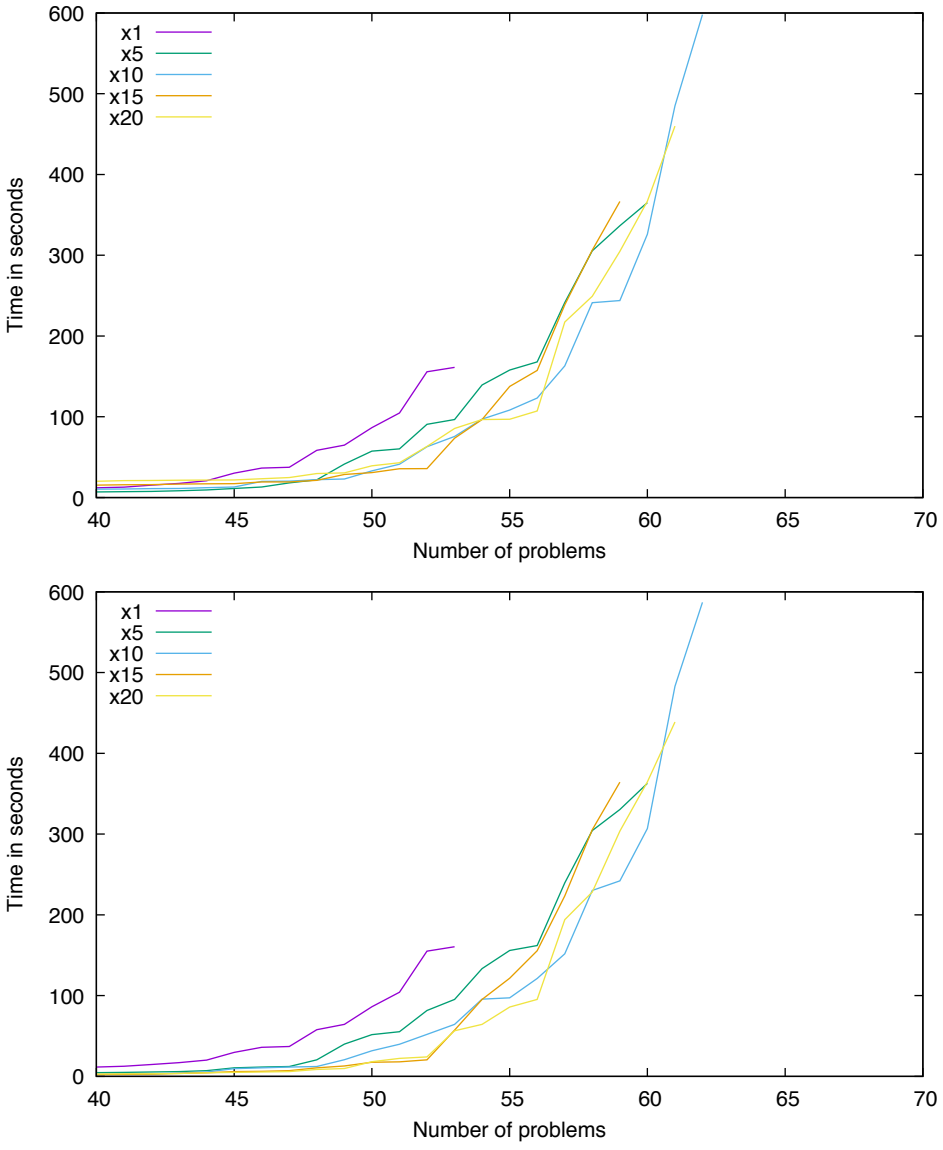

Figure 4.6: Cactus plots (see section 4.1.2) for the 70 grid problems that were solved with the LPDP-solver. The different curves show the runtime with different partitioning times that the solver used. The first plot shows the complete runtime (including the time spent partitioning). The second plot is only the runtime of the actual algorithm as if the partitioning was already given (excluding the partitioning time).

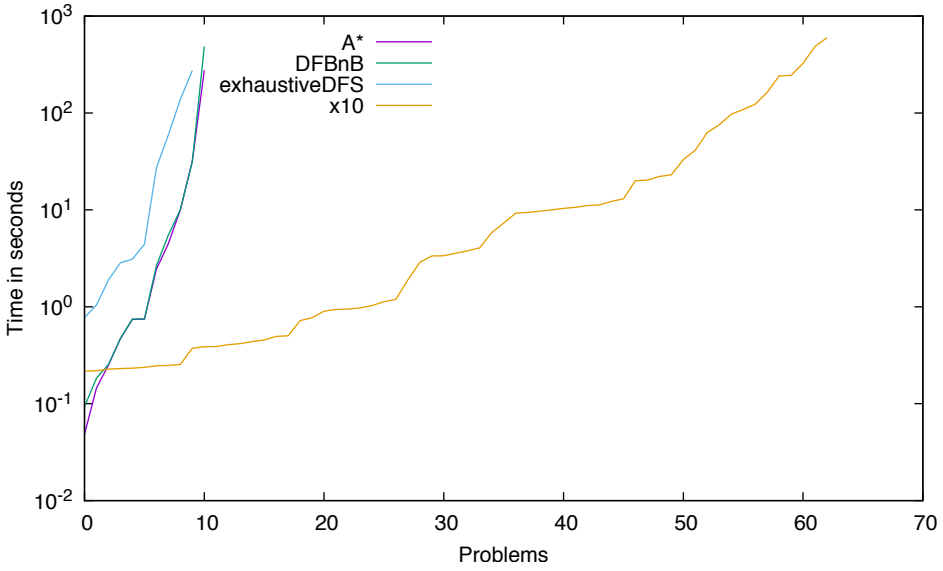

Figure 4.7: Cactus plot (see section 4.1.2) for the different optimal algorithms that were evaluated. The LPDP-solver is only represented with x10, as the other xN are pretty similar to it and would only reduce the clarity of the plot. The runtime of x10 includes the time that was spent partitioning the graph.

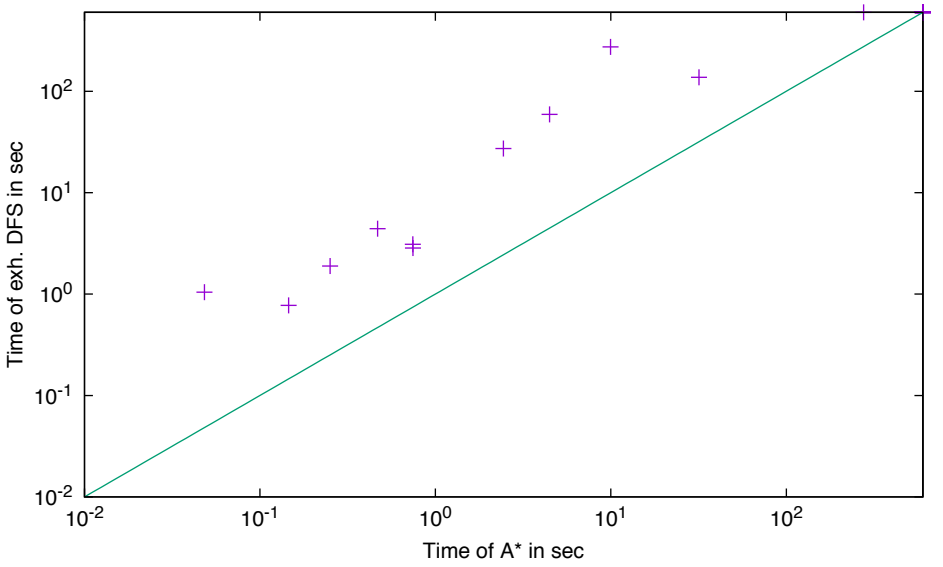

Figure 4.8: Scatter plot (see section 4.1.2) of the A* and exhaustive DFS for the grid (30%) instances. A* exclusively performed better.

	x1	x5	x10	x15	x20	A*	DFBnB	exh.DFS
10	0,0890115 10	1,03739 10	1,13457 10	0,236855 10	2,13764 10	5,10719 10	5,20982 10	51,2043 10
15	0,200694 10	0,434824 10	1,32732 10	1,83663 10	2,34593 10	275,837 1	485,575 1	N/A 0
20	0,998142 10	3,13388 10	3,72064 10	1,93556 10	8,91817 10	N/A 0	N/A 0	N/A 0
25	13,2708 9	40,1293 10	36,6412 10	38,0422 10	39,3882 10	N/A 0	N/A 0	N/A 0
30	40,4099 9	28,9649 10	26,6574 10	45,3826 10	31,7437 9	N/A 0	N/A 0	N/A 0
35	60,9708 6	79,5184 8	173,16 9	54,0274 8	47,0712 7	N/A 0	N/A 0	N/A 0
40	N/A 0	289,876 3	147,096 4	220,06 2	227,576 6	N/A 0	N/A 0	N/A 0

Table 4.2: Table of runtime averages (see section 4.1.2) for all NxN-grid (30%) instances and the different algorithms.

4.1.4.3 Roads

The road benchmark consists of 150 random subgraphs of a large, weighted graph that represents the road network of the USA. The first subgraph only consists of 2 vertices. The next subgraph always contains two more vertices than the previous one until 300 vertices are reached.

solver	x1	x5	x10	x15	x20	A*	DFBnB	exh.DFS
solved problems	132	135	132	134	133	76	76	65

Even less of a difference in the actual runtime of the algorithm can be seen in Figure 4.9. The only big difference exists between x1 and the others. Any xN with $N > 1$ seems to give partitions of a similar quality, which practically doesn't increase further with higher N.
Additionally Figure 4.10 suggests that the advantage that LPDP or at least the xN-solver has over the other solvers from [SKP+14] has gotten smaller. In fact the xN could only solve about twice as many instances as the others. This could have a multitude of reasons. The first is that road-instances are random subgraphs of the US road network. This means that the number of its vertices is rather insignificant to the overall difficulty of an instance, which was seen in the benchmark as, for example, x1 solved some of the biggest instances (300,290,286,...) in less than a second, while others of similar size were unsolvable in the complete 10 minutes. This occurs because random subgraphs consisting of n vertices vary far more in their overall difficulty. This is less of an issue for grids since its obstacles get distributed at random and uniformly over the whole grid. Any area in the grid can be expected the have the same percentage of obstacles as the whole grid. The road graph's unpredictability is a problem for the partitioning procedure that is used. First off KaHIP itself, which is the program that is used to partition a graph, could be less fit to partition road graphs. Grid graphs have useful characteristics (each vertex can only have up to 4 edges, ...) that could help to partition it and also make sure that better partitions exist. This isn't the case for the road graphs. An additional problem is that KaHIP tries to partition the graph into subsets of a similar size, while minimizing the number or weight of the edges between them. The wanted size has to be given to KaHIP as a parameter beforehand. This parameter was simply set to 10 for all roads. This is obviously not optimal. A better approach would be to determine this number based on the specific graph's properties (e.g. edge density). Additionally the partitioning procedure surrounding

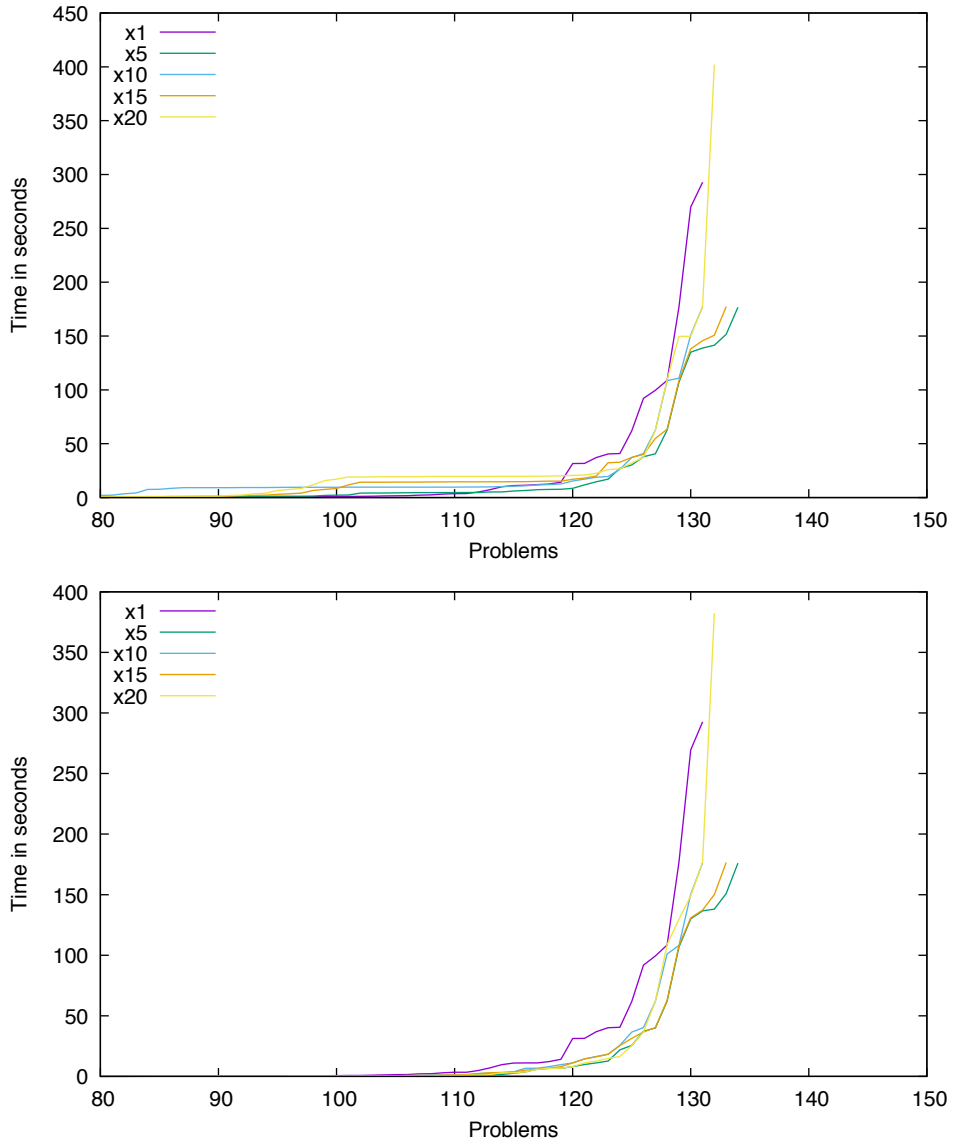

Figure 4.9: Cactus plots (see section 4.1.2) for the 150 road graphs that were solved with the LPDP-solver. The different curves show the runtime with different partitioning times that the solver used. The first plot shows the complete runtime (including the time spent partitioning). The second plot is only the runtime of the actual algorithm as if the partitioning was already given (excluding the partitioning time).

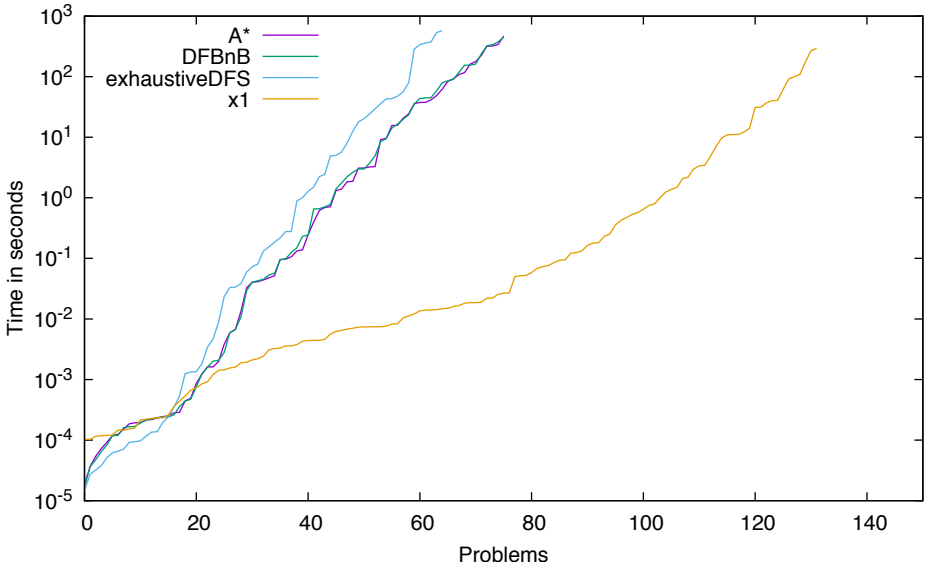

Figure 4.10: Cactus plot (see section 4.1.2) for the road graphs that were evaluated. The LPDP-solver is only represented with x1, as the other xN are pretty similar to it and would only reduce the clarity of the plot. The runtime of x1 includes the time that was spent partitioning the graph.

KaHIP, as explained in section 3.4, could also be flawed (which would also affect grid graphs).

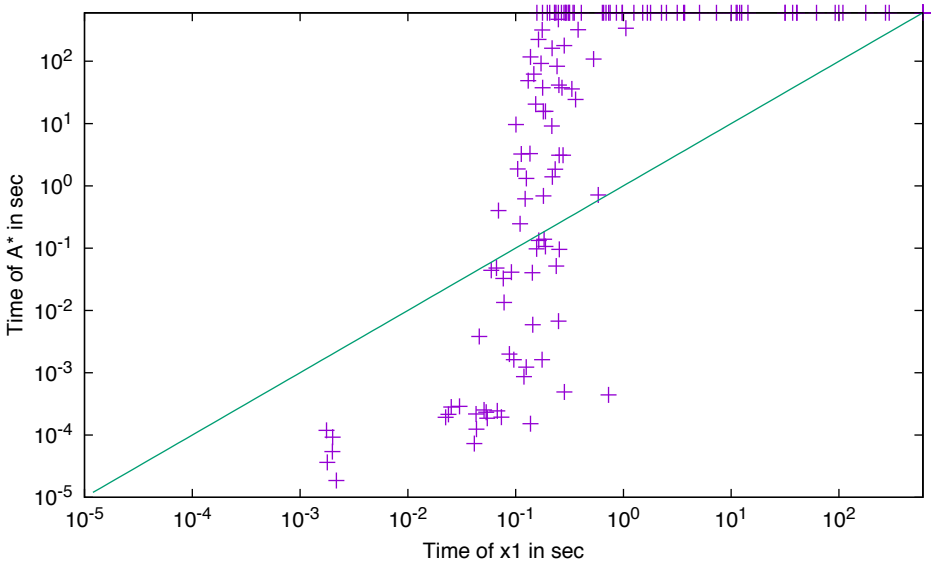

Figure 4.11: Scatter plot (see section 4.1.2) of the x1 and A* for the road instances.

5. Conclusion

Longest Path Dynamic Programming (LPDP), the optimal algorithm for the longest path problem that was presented in this thesis, is based upon graph partitioning and dynamic programming. A hierarchical partition has to be precalculated and given to the algorithm. The acceleration of the algorithm compared to a brute force approach is dependent on the quality of this partition. LPDP was tested on certain graph types and has been shown to be significantly faster than other known algorithms. This was the case even though the partitioning procedure certainly didn't use KaHIP to the fullest extend that would have been possible. The parameters used in the current procedure could be calculated in a better way. Additionally other partitioning programs and approaches could be tested.
The graphs that were tested all had a rather low amount of edges compared to their number of vertices and allowed for a fitting partitions. The algorithm's usefulness could strongly vary if such good partitions do not exist or are harder to find. Other types of graphs, especially ones for which practical applications of the longest path problem exist, could still be tested. The algorithm could be suitable for most planar graphs as grid graphs also are planar and all graphs of the benchmarks had around 2 to 3 times as many edges as vertices (a planar graph with n vertices can only have a maximum of $3n - 6$ edges). Additionally planar graphs are probably easier and better to partition in a manner that is fitting for the algorithm.
Possible improvements to the algorithm in regards to parallelization have already been given in the section 3.3. Another improvement might be to cull unreachable parts of the graph before the partitioning. This could be useful for graphs that consist of many separate components. Unnecessary calculations take place if a component of the graph that is unreachable from the start or goal vertices is divided into multiple subsets. The algorithm then calculates possible longest paths between their borders only to discard them later. This was the case for grid graphs. Road graph were constructed in a way that always resulted in a single connected component. Certain calculations for subsets that contain the start or target vertex are also unnecessary, since the start and target vertices always have to be connected with another border-node. The algorithm, as it was implemented, calculates all possible connections within a subset, even if they don't include the start/target vertex.

Bibliography

[Bel58] R.E. Bellman. "On a routing problem". In *Quarterly of Applied Mathematics*, pages 87–90, 1958.

[Bru95] Peter Brucker. *Scheduling Algorithms*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1995.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *NUMERISCHE MATHEMATIK*, 1(1):269–271, 1959.

[For56] L. R. Ford. Network flow theory, 1956.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[OW06a] M. M. Ozdal and M. D. F. Wong. Algorithmic study of single-layer bus routing for high-speed boards. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(3):490–503, March 2006.

[OW06b] M. Mustafa Ozdal and M. D. F. Wong. A length-matching routing algorithm for high-performance printed circuit boards. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(12):2784–2794, Dec 2006.

[PR10] David Portugal and Rui Rocha. Msp algorithm: Multi-robot patrolling based on territory allocation using balanced graph partitioning. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 1271–1276, New York, NY, USA, 2010. ACM.

[SKP+14] Roni Stern, Scott Kiesel, Rami Puzis, Ariel Feller, and Wheeler Ruml. Max is more than min: Solving maximization problems with heuristic search. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search (SoCS 2014)*, 2014.

[SS13] Peter Sanders and Christian Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *LNCS*, pages 164–175. Springer, 2013.

[WLK05] Wan Yeung Wong, Tak Pang Lau, and Irwin King. Information retrieval in p2p networks using genetic algorithm. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, WWW '05, pages 922–923, New York, NY, USA, 2005. ACM.