Karlsruhe Institute of Technology

Parallel Optimal Longest Path Search

Master 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, 9th November 2018

Abstract

This thesis presents a parallel algorithm that solves the longest path problem for weighted undirected graphs. The algorithm uses dynamic programming and graph partitioning in order to find the longest path between two vertices. The algorithm finds the optimal longest path and not an approximation of it like many other approaches for NP-complete problems. We first present the serial algorithm and then how it can be parallelized. Through experiments we measured the speedup of the parallel algorithm compared to its serial counterpart. We achieved reasonable speedups. We also demonstrate that the algorithm has possible applications in solving the Hamiltonian cycle problem.

Deutsche Zusammenfassung

Diese Arbeit präsentiert einen parallelen Algorithmus, der das Problem des längsten Weges (longest path problem) für gewichtete ungerichtete Graphen löst. Der Algorithmus macht sich das Partitionieren von Graphen und das Prinzip der dynamischen Programmierung zu Nutzen, um den längsten Pfad zwischen zwei Knoten des Graphen zu finden. Im Gegensatz zu vielen anderen Algorithmen für NP-vollständige Probleme liefert der Algorithmus den optimalen längsten Pfad und nicht nur eine Approximation davon. Zuerst stellen wir den sequenziellen Algorithmus vor und zeigen dann wie er parallelisiert werden kann. Anhand von Experimenten messen wir die Beschleunigung, die durch die parallele Aufführung erreichbar ist. Dabei wurden akzeptable Werte gemessen. Außerdem zeigen wir, dass der Algorithmus zum Lösen bestimmter Fälle des Hamiltonkreisproblems geeignet ist.

Contents

1. Introduction

The longest path problem (LP) is the problem of finding a simple path of maximum length between two given vertices of a graph. A path is simple if it does not contain any vertex more than once. The length of a path can be defined as the number or the cumulative weight of its edges. LP is NP-complete [GJ79]. LP has applications in the design of circuit boards, where the length difference between wires has to be kept small [OW06a] [OW06b]. 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].

In our previous work ([Fie16]) we presented the "Longest Path by Dynamic Programming" (LPDP) algorithm. LPDP makes use of graph partitioning and dynamic programming. Unlike many other approaches for NP-complete problems LPDP is not an approximation algorithm. It finds an optimal longest path. In [Fie16] we compared LPDP to other LP algorithms. LPDP performed far better than the other algorithms. This thesis presents an improved version of LPDP. We then show how we parallelized the algorithm. Through experiments we evaluate the speedups achieved by the parallel algorithm. Additionally we examine the worst case performance of LPDP and possible applications in solving the Hamiltonian cycle problem.

2. Preliminaries

2.1 Related Work

The paper "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 algorithms that are usually used to solve the shortest path problem (SP) to the longest path problem (LP). They first make clear why LP is so difficult compared to SP. Then they present several algorithms that are frequently used to solve SP or other minimization search problems, which are then modified in order to be able to solve LP. The search algorithms are part of three categories. First "uninformed" searches, which do not require any information other than what is already given in the definition of the problem. An example for these algorithms were Dijkstra's algorithm or Depth First Branch and Bound (DFBnB) without a heuristic. Modifying these algorithms for LP led to bruteforce algorithms, which means that they still had to look at every possible path in the search space. The second category is that 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. The well known algorithms DFBnB and A* were modified in order to solve the longest path problem with the help of a heuristic function. These algorithms had an advantage over bruteforce searches. The third category is that of suboptimal searches. Stern et al. [SKP+14] looked at a large number of these algorithms that only find approximations of a longest path. We used solvers and benchmarks from this paper in this thesis and the bachelor thesis [Fie16]. This is explained below.

In the bachelor thesis "Using Graph Partitioning to Accelerate Longest Path Search" [Fie16] we first presented the Longest Path by Dynamic Programming (LPDP) algorithm. This original version of the algorithm is different to its current iteration. In the experiments section of this thesis we compare the two versions. In [Fie16] we compared LPDP to other longest path algorithms. We did this with the two heuristic searches DFBnB and A* from Stern et al. [SKP+14] that were mentioned above. Additionally, we used a bruteforce solver. For the tested problems LPDP performed significantly better than the other algorithms. Our problem benchmark consisted of two problem classes that were also used in [SKP+14]. Specifically the two problem types "grids" and "roads". In this thesis we also make use of these problem types. What they are and how they are generated is explained in section 6.5. Additionally, [Fie16] looked at the influence that different partitions have on the runtime

of LPDP. The runtime of LPDP was found to be dependent on the quality of the partition. Investing more time into the partitioning of the graph paid off, as it decreased the runtime of LPDP.

Problems that are related to LP are the Hamiltonian path/cycle problem and the snake-in-the-box problem:

A Hamiltonian path is a path that visits each vertex of the graph exactly once. A Hamiltonian cycle is a cycle that does the same. The Hamiltonian path/cycle problem is the problem of determining if an undirected graph contains a Hamiltonian path/cycle. In a graph with n vertices finding a Hamiltonian path is equivalent to finding a longest path with $n-1$ edges between any two vertices. Later in this thesis we also show a way to transform the Hamiltonian cycle problem (HCP) to the longest path problem. We then solve transformed HCP instances with the LPDP algorithm.

The snake-in-the-box problem was first presented by Kautz in 1958 [Kau58]. It is the problem of finding the longest simple path in an n-dimensional hypercube. Additionally, it is required that any two vertices of the path that are adjacent in the hypercube also have to be adjacent in the path. This additional constraint is what makes snake-in-the-box different to LP and why LPDP is not suitable to solve snake-in-the-box instances.

2.2 Definitions

Graph

A **vertex** is a fundamental unit that the graph is made of. Another name for vertex is node. We usually label vertices with natural numbers (including zero). An **edge** is a connection between two vertices x and y. An edge that serves as a two way connection between x and y is represented by the set $\{x, y\}$ and is called **undirected**. An edge that only serves as a one way connection from x to y is represented by the tuple (x, y) and is called **directed**.

We define a **graph** $G := (V, E)$. V is a set of vertices. Usually $V := \{0, 1, 2, \ldots, |V| - 1\}$. E is a set of edges. G is a **directed graph** if $E \subseteq \{(x, y) | x, y \in V\}$ is a set of directed edges. G is an **undirected graph** if $E \subseteq \{\{x, y\} | x, y \in V\}$ is a set of undirected edges. In this thesis we only look at undirected graphs. Additionally, a graph can be **weighted** if we associate a weight with each edge of the graph. This is represented by a weight-function w: E $\mapsto$ X where X is a set of weights. In this thesis the weights are considered to be real numbers, so $X = \mathbb{R}$. Any unweighted graph can be interpreted as a weighted graph where all edges have a weight of 1.

A subset of an undirected graph's vertices is a **clique** if there is an edge between any two distinct vertices of the subset.

A **matching** is a subset of edges where no two edges have a common vertex.

Path

A **path** in an undirected graph $G := (V, E)$ is defined as a sequence of its edges $p = e_0, e_1, \ldots, e_n$ where $e_i = \{v_i, v_{i+1}\} \in E$. A path can be seen as a way to traverse the graph from vertex v_0 to v_{n+1}. The **length** or weight of a path in a weighted graph is the cumulative weight of its edges. In an unweighted graph it is the number of its edges. An alternative representation for p is the vertex sequence $v_0, v_1, \ldots, v_{n+1}$. In this thesis we also allow paths that only consist of a single vertex and no edges. Such a path can only be represented as the vertex sequence v_0. This path has a weight of 0.

A path is **simple** if it does not contain a vertex more than once.

A **cycle** is the same as a simple path except that the start- and end-vertex are the same ($v_0 = v_{n+1}$) and that n has to be 2 or higher.

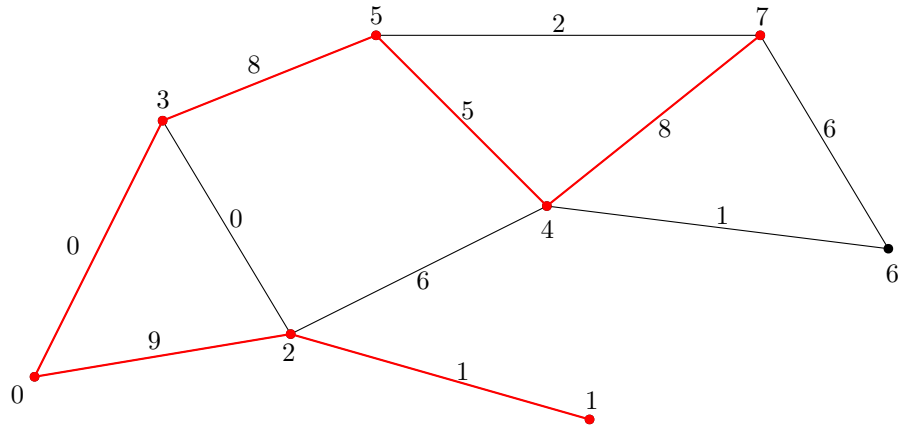

Figure 2.1: Example of a weighted and undirected graph. The longest path between the vertex 1 and 7 is shown in red. Its weight is 31.

Longest Path Problem (LP)

The **longest path problem** for a given undirected 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)**. An example can be seen in Figure 2.1

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 transformed into an instance of the previous definition by introducing the vertices s and t 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. We call such a subset a **block**.

Usually the intent is to distribute the vertices of a graph evenly into a number of blocks while minimizing the number or the combined weight of the edges between the vertices of different blocks.

Array

An **array** is a collection of elements that can be accessed 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 $\in \{0, 1, \ldots, size - 1\}$. An empty bracket [] behind a variable's name indicates that it is an array ($arrayName[]$).

Hash table/map

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.

3. Longest Path by Dynamic Programming

This section introduces our algorithm called "Longest Path by Dynamic Programming" (LPDP). The algorithm solves the longest path problem (LP) for *weighted undirected graphs*. LPDP is based on the principles of dynamic programming. Later in the thesis we will show how to parallelize this algorithm.

3.1 Exhaustive Depth First Search

A simple way to solve the longest path problem is *exhaustive depth-first search* [SKP+14]. In regular depth-first search (DFS) a vertex has two states: marked and unmarked. Initially, all vertices are unmarked. The search starts by calling the DFS procedure with a given vertex as a parameter. This vertex is called the root. The current vertex (the parameter of the current DFS call) is marked and then the DFS procedure is recursively executed on each unmarked vertex reachable by an edge from the current vertex. The current vertex is called the parent of these vertices. Once the recursive DFS calls are finished we backtrack to the parent vertex. The search is finished once DFS backtracks from the root vertex.

Exhaustive DFS is a DFS that unmarks a vertex upon backtracking. Pseudocode can be seen in Figure 3.1. In that way every simple path in the graph starting from the root vertex is explored. The LP problem can be solved with exhaustive DFS by using the start vertex as the root. During the search the length of the current path is stored and compared

```
Search exhDFS(v)
    if v is unmarked then
        mark v
        foreach {v, w} ∈ E do
            exhaustiveDFS(w)
        end
        unmark v
    end
```

Figure 3.1: Exhaustive depth first search. In order to solve LP we start this search from the start vertex and update the best found solution each time the (unmarked) target vertex is found.

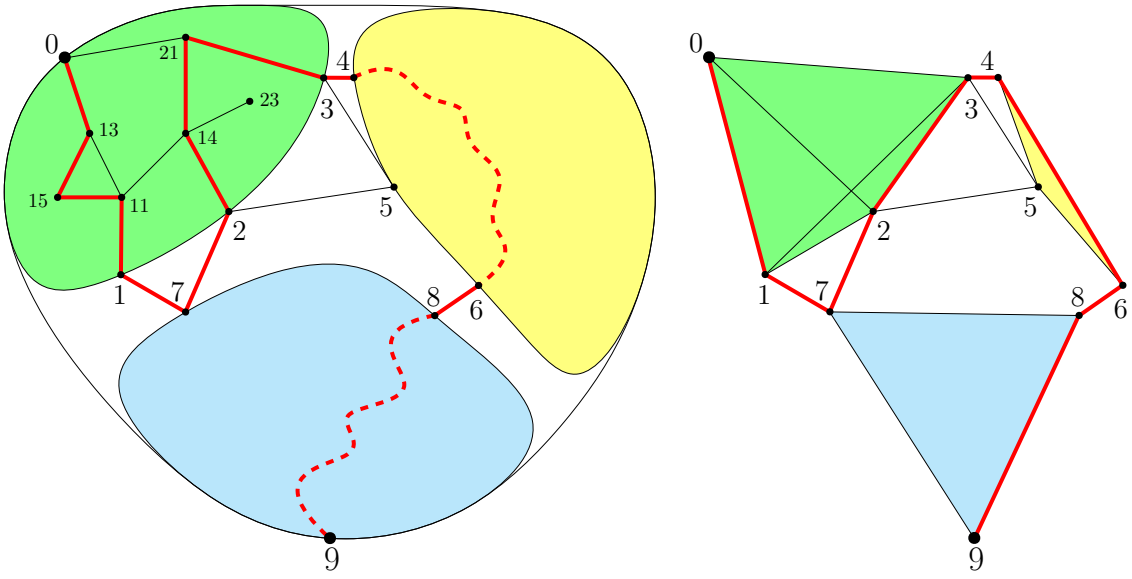

Figure 3.2: On the left we see an example graph $G = (V, E)$. We want to calculate the longest path between $s = 0$ and $t = 9$. The initial block $B = V$ is partitioned into three subblocks B_{green}, B_{yellow} and B_{blue}. The vertices $0, 1, \ldots, 9$ are the boundary vertices of these subblocks. The edges between the subblocks are boundary edges. The right graph is the auxiliary graph that is constructed by the LPDP algorithm. A search path of LPDP can be seen on the right. Its induced boundary vertex pairs for the subblocks are: $P_{green} = \{\{0, 1\}, \{2, 3\}\}$, $P_{yellow} = \{\{4, 6\}\}$, $P_{blue} = \{\{7, 7\}, \{8, 9\}\}$. The corresponding candidate for the longest path is shown on the left in red.

to the previous best solution each time the target vertex is reached. If the current length is greater than that of the best solution, it is updated accordingly. When the search is done a path with maximum length from s to t has been found. If we store the length of the longest path for each vertex (not just the target vertex), then all the longest simple paths from s to every other vertex can be computed simultaneously.

It is easy to see, that the space complexity of exhaustive DFS is the same as regular DFS – linear in the size of the graph. However, the time complexity is much worse. In the worst case – for a clique with n vertices – the time complexity is $\mathcal{O}(n!)$ since every possible simple path is explored, which corresponds to all permutations of the vertex set. If the maximum degree of the graph is d, then the running time can be bound by $\mathcal{O}(d^n)$, where n is the number of vertices.

3.2 Algorithm Overview

Dynamic programming requires us to be able to divide LP into subproblems. In order to do this we first generalize the problem:

Given: A graph $G = (V, E)$, $s, t \in V$, $B \subseteq V$ and $P \subseteq \{\{a, b\} \mid a, b \in b(B)\}$
where $b(B) := \{v \in B \mid v = s \vee v = t \vee \exists \{v, w\} \in E : w \notin B\}$ are the *boundary vertices* of B.

Problem: Find a simple path from a to b in the subgraph induced by B for every $\{a, b\} \in P$.
Find these paths in such a way that they do not intersect and have the maximum possible cumulative weight.

We explain an example of the problem with Figure 3.2. Later this figure will be used to explain the LPDP algorithm further, but for now we only look at the green area on the left. This green area represents a subgraph of a graph G. This subgraph is induced by the set $B = \{0, 1, 2, 3, 11, 13, 14, 15, 21, 23\}$. The vertices 0 and 9 in the figure are the vertices s and t of the problem's definition. This results in the boundary vertices $b(B) = \{0, 1, 2, 3\}$. If we restrict the figure to the green subgraph, there are two paths shown in red. These paths represent a solution to the problem if $P = \{\{0, 1\}, \{2, 3\}\}$. This solution has a weight of 7 as the graph is unweighted and the two paths consist of 7 edges.

We make the following observations about the problem:

Observation 1 (The structure of P)**.** *The set P of a solvable problem can contain pairs of the form $\{v, v\}$. The problem is impossible to solve if P contains two pairs $\{a, b\}$ and $\{b, c\}$.*

Proof. A pair $\{v, v\} \in P$ results in a path of weight 0 that consists of a single vertex (v) and no edges. Two pairs $\{a, b\}, \{b, c\} \in P$ would result in two intersecting paths as they would have the same start- or end-vertex b. □

Observation 2 (Number of solvable Ps)**.** *There are at most $\#P(n) := \sum_{k=0}^{\lfloor n/2 \rfloor} \frac{n!2^{n-3k}}{(n-2k)!k!}$ possible, solvable P. $\#P(n) \in \mathcal{O}(2^n n^{n/2} e^{\sqrt{n}})$. If the subgraph induced by B is a clique, all $\#P(n)$ sets are solvable. So in this case $\#P(n)$ is not an upper bound but the exact number of solvable Ps.*

Proof. We calculate an upper bound on the number of all solvable P the following way: We transform P into two sets (M, X): $\{x, y\} \in P \wedge x \neq y \iff \{x, y\} \in M$ and $\{x, x\} \in P \iff x \in X$. We imagine a clique consisting of the boundary vertices $b(B)$. This way we can interpret M as a set of edges in this clique. It follows from observation 1 that M represents a matching (set of edges without common vertices) in that clique. The numbers of all possible matchings in a clique of size n are also known as the telephone numbers or involution numbers [Knu73]: $T(n) := \sum_{k=0}^{\lfloor n/2 \rfloor} \frac{n!}{2^k (n-2k)!k!}$. Each element of the sum equals the number of matchings with k edges. Any of the other $n - 2k$ boundary vertices are either in X or not. This leads to 2^{n-2k} possibilities for X per k-edge matching M. This means that there are at most $\#P(n) := \sum_{k=0}^{\lfloor n/2 \rfloor} \frac{n!2^{n-3k}}{(n-2k)!k!}$ possible, solvable P.

We also know that $T(n) \sim (\frac{n}{e})^{n/2} \frac{e^{\sqrt{n}}}{(4e)^{1/4}}$ [CHM51]. Additionally, we calculate $\#P(n) = \sum_{k=0}^{\lfloor n/2 \rfloor} \frac{n!2^{n-2k}}{2^k (n-2k)!k!} \leq 2^n T(n)$. From this we can say that $\#P(n) \in \mathcal{O}(2^n n^{n/2} e^{\sqrt{n}})$

□

Observation 3. *If the problem is unsolvable for a P, it is also unsolvable for any $P' \supseteq P$.*

The proof is trivial.

Observation 4 (Subproblems)**.** *A solution of the problem for B and P also induces a solution to the problem for any $B' \subseteq B$ and a P': Restricting the solution to vertices of B' results in non-intersecting, simple paths in the subgraph of B'. These paths start and end in boundary vertices of B' inducing the set P'. These paths are the solution to the problem for B' and P'.*

Proof. Otherwise we could take the solution for B and P, remove all paths in the subgraph of B' and replace them with the solution for B' and P'. We would obtain a solution for B and P with a higher cumulative weight than before. This is impossible. □

LP is a special case of this problem where $B = V$ and $P = \{\{s, t\}\}$. Observation 4 is the basis of the LPDP algorithm as it allows us to recursively divide LP into subproblems. LPDP requires a hierarchical partitioning of the graph. Level 0 represents the finest level of partitioning. On each higher level we combine a group of blocks from the lower level into a single larger block. On the highest level we are left with a single block $B = V$. We solve our problem for each of these blocks and any possible P: We start by calculating the solutions for each block of level 0. We then calculate the solutions for a block on level 1 by combining the solutions of its level 0 subblocks. This is repeated level by level until we calculated all solutions for the block $B = V$, namely the solutions for $P = \{\{s, t\}\}, \{\}, \{\{s, s\}\}, \{\{t, t\}\}$ and $\{\{s, s\}, \{t, t\}\}$. The latter four are trivial (see observation 1) and do not have to be calculated. With solution for $P = \{\{s, t\}\}$ we also solved LP as it is the longest path from s to t.

The next section shows how we calculate the solutions for one block B with the help of its subblocks from the level below. The initial solutions for the blocks on level 0 can be calculated with the same algorithm. In order to do this we interpret each vertex v as a separate subblock. We know the solutions for each of these subblocks ($P = \{\}$ or $\{\{v, v\}\}$). So we can use the same algorithm to calculate solutions for the blocks on level 0.

3.3 Combining Solutions

Let P_S be the set of boundary vertex pairs for a set of vertices S. Given is a subset of vertices $B \subseteq V$ and a partition $B_1,\ldots,B_k$ of B ($B_1 \cup \cdots \cup B_k = B$ and $B_i \cap B_j = \emptyset$ for $i \neq j$). We assume that we already solved the problem for each B_i and every possible P_{B_i}. We calculate the solution for B and every possible P_B with the following algorithm:

We construct an auxiliary graph $G' = (V', E')$ with $V' = \bigcup_{i=1}^{k} b(B_i)$. E' contains all edges $\{v, w\} \in E$ where $v \in b(B_i)$ and $w \in b(B_j)$ (with $i \neq j$). We call these edges boundary edges. They keep the weight they had in G. We also create a clique out of the boundary vertices of every B_i. These new edges have a weight of 0. An example of this can be seen in Figure 3.2. The graph on the left is partitioned into three blocks B_{green}, B_{yellow} and B_{blue}. On the right we can see the constructed auxiliary graph. For now we can ignore the path that is shown in red.

In order to calculate the solutions for B we start a modified version of the exhaustive DFS on every boundary vertex of B. Pseudocode of this search algorithm is called LPDP-Search and is shown in Figure 3.3. Compared to exhDFS LPDP-Search works with multiple paths. This way the search algorithm resembles a nested exhDFS. The first path starts from a starting boundary vertex b_1. Once another boundary vertex b_2 of B is reached, LPDP-Search has two options. It can traverse the graph's edges as usual, but it also is able to jump to other unused boundary vertices. If LPDP-Search jumps to another boundary vertex b_3, we completed a path from b_1 to b_2 and a new path from b_3 is started. This induces the pair $\{b_1, b_2\} \in P_B$. At any point of the search P_B is equivalent to the boundary vertex pairs induced by the completed paths. The sets P_{B_i} are maintained the following way: The paths contain an edge $\{v, w\}$ of the B_i-clique $\iff \{v, w\} \in P_{B_i}$. If the paths contain a vertex $v \in B_i$ but no edge $\{v, w\}$ of the B_i-clique: $\{v, v\} \in P_{B_i}$. During the search we do not traverse an edge that would induce a P_{B_i} without a solution. Further traversal of a path with an unsolvable P_{B_i} only leads to $P'_{B_i} \supseteq P_{B_i}$ which is still unsolvable (as already mentioned in observation 3).

Each time we complete a path we calculated a candidate for the solution to B and P_B. The weight of this candidate is the weight of the solution of each block B_i and the induced P_{B_i} plus the weight of all boundary edges in the paths. Until now no P_B found by the search contains a pair $\{v, v\}$ as we do not allow a path to end in its starting boundary vertex. This way P_B is equivalent to a M and $X = \emptyset$ according to the representation in

```
1  Search LPDP-Search(v)
2      if v is unmarked & ∀i : there exists a solution for B_i and P_{B_i} then
3          mark v
4          if v ∈ b(B) then
5              if already started some {a,.}-path then
6                  if v > a then
7                      P_B ← P_B ∪ {{a,v}}              // completes the {a,v}-path
8                      update_solutions(0)
9                      foreach w ∈ b(B) where w > a do
10                         LPDP-Search(w)                  // starts a {w,.}-path
11                     end
12                     P_B ← P_B \ {{a,v}}          // resume search with {a,.}-path
13                 end
14             end
15         end
16         foreach {v,w} ∈ E do
17             LPDP-Search(w)
18         end
19         unmark v
20     end
```

Figure 3.3: Basic search algorithm that LPDP uses to search the auxiliary graphs. Figure 3.4 shows the update_solutions(.) operation.

the observation 2. So when we complete a path we additionally go through all possible sets X (while modifying the sets P_{B_i} accordingly) and update the best found solution for these candidates as well. Figure 3.4 shows how this can be done. LPDP-Search in Figure 3.3 calls this function in line 8.

An example can be seen in Figure 3.2. In the auxiliary graph on the right we traversed a path from 0 to 9. We found a solution for $P = \{\{0, 9\}\}$. The induced boundary vertex pairs for the subblocks are $P_{green} = \{\{0, 1\}, \{2, 3\}\}$, $P_{yellow} = \{\{4, 6\}\}$, $P_{blue} = \{\{7, 7\}, \{8, 9\}\}$. When we look up these solutions for the subblocks we obtain the path that is shown on the left.

An important optimization which can be seen in Figure 3.3 in line 6 is that we only allow a path to end in a boundary vertex with a higher ID than its starting boundary vertex. Additionally, a path can only start from a boundary vertex with a higher ID than the starting vertex of the previous path (line 9). The first optimization essentially sorts the two vertices of each pair $\{x, y\} \in P$. The second then sorts these pairs. Resulting in an order and a direction in which we have to search each path in P. This avoids unnecessary symmetrical traversal of the graph.

3.4 Examples

Figure 3.5 and 3.6 show examples of how the LPDP algorithm works. Figure 3.5 shows how a level 0 block is solved. Figure 3.6 shows how we solve a block from a higher level. In both figures the red path is the current search path. If we start a new path from another boundary vertex, the previous path becomes gray. Each graph in the figures represents the state of the search while LPDP-Search(v) is called. If LPDP-Search(v) calls itself n times LPDP-Search(w_1), LPDP-Search(w_2), ..., LPDP-Search(w_n), there are n arrows going out from the LPDP-Search(v). Each one points to the state LPDP-Search(w_i) for

```
  // u_j := j-th unmarked boundary vertex of B
1 Operation update_solutions(index)
2 |  update solution for B and P_B
3 |  foreach j ≥ index do
4 |  |  Choose k so that u_j ∈ B_k
5 |  |  P_{B_k} ← P_{B_k} ∪ {{u_j, u_j}}
6 |  |  if solution for B_k and P_{B_k} exists then
7 |  |  |  P_B ← P_B ∪ {{u_j, u_j}}
8 |  |  |  update_solutions(index+1)
9 |  |  |  P_B ← P_B \ {{u_j, u_j}}
10|  |  end
11|  |  P_{B_k} ← P_{B_k} \ {{u_j, u_j}}
12|  end
```

Figure 3.4: LPDP-Search (Figure 3.3) uses this operation to update the bock's solutions.

one $1 \leq i \leq n$. If a LPDP-Search()-call updates any solutions the corresponding sets of boundary vertex pairs are written below the state. In order to reduce the size of the figures the examples show a fully optimized LPDP-Search that prevents a lot of unnecessary LPDP-Search()-calls. This means that the figures do not correspond completely with the pseudocode in Figure 3.3. Some of the optimizations may not have been presented until now. They are explained in section 5.2.

3.4.1 Solving a Block on Level Zero

Figure 3.5 shows how a level 0 block is solved. As mentioned we can use LPDP-Search() to solve level 0 blocks. For this we simply interpret each vertex v as a clique of a single vertex. This way each clique/vertex represents a subblock with one boundary vertex. Only two solutions exist for such a subblock. These solutions are for $P_v = \{\}$ and $\{\{v, v\}\}$. Both solutions always exists, are trivial and have a weight of 0. They do not effect the search. This way LPDP-Search() essentially degenerates into a nested exhaustive depth first search. The example will explain this further.

In Figure 3.5 we solve a level 0 block with 5 vertices. The block has 4 boundary vertices labeled 0, 1, 2 and 3. The unlabeled vertex is a non-boundary vertex. In order to solve the block completely we would have to run LPDP-Search from the boundary vertices 0, 1 and 2. We would not start a search from boundary vertex 3 as a $\{3, .\}$-path can only be completed in a higher boundary vertex. As 3 is the boundary vertex with the highest ID we could never complete the path. Meaning that LPDP-Search started from 3 could never find a solution. The example in Figure 3.5 shows the LPDP-Search that is started from boundary vertex 0. The initial state represents this LPDP-Search(0) call. We see the {0,.}-path that was started in red. LPDP-Search(0) traverses the graph and calls itself twice. One time for each of the two edges that are adjacent to 0. We look at the branch on the right that took the edge $\{0, 2\}$ in more detail.

As we reached another boundary vertex we check if the vertex ID is higher than that of the starting vertex of the path. With $0 < 2$ this is the case. This means that the current path $0, 2$ represents a possible solution for $P = \{\{0, 2\}\}$ with the weight of 1. Since this is an unweighted graph we count the number of edges in the path. We update any previous solution that was found for $P = \{\{0, 2\}\}$ accordingly. Additionally, there are two unused boundary vertices: 1 and 3. Any unused boundary vertex v can also serve as a $\{v, v\}$-path. Such a path has a weight of 0. This means that the solution for $P = \{\{0, 2\}\}$ that we found

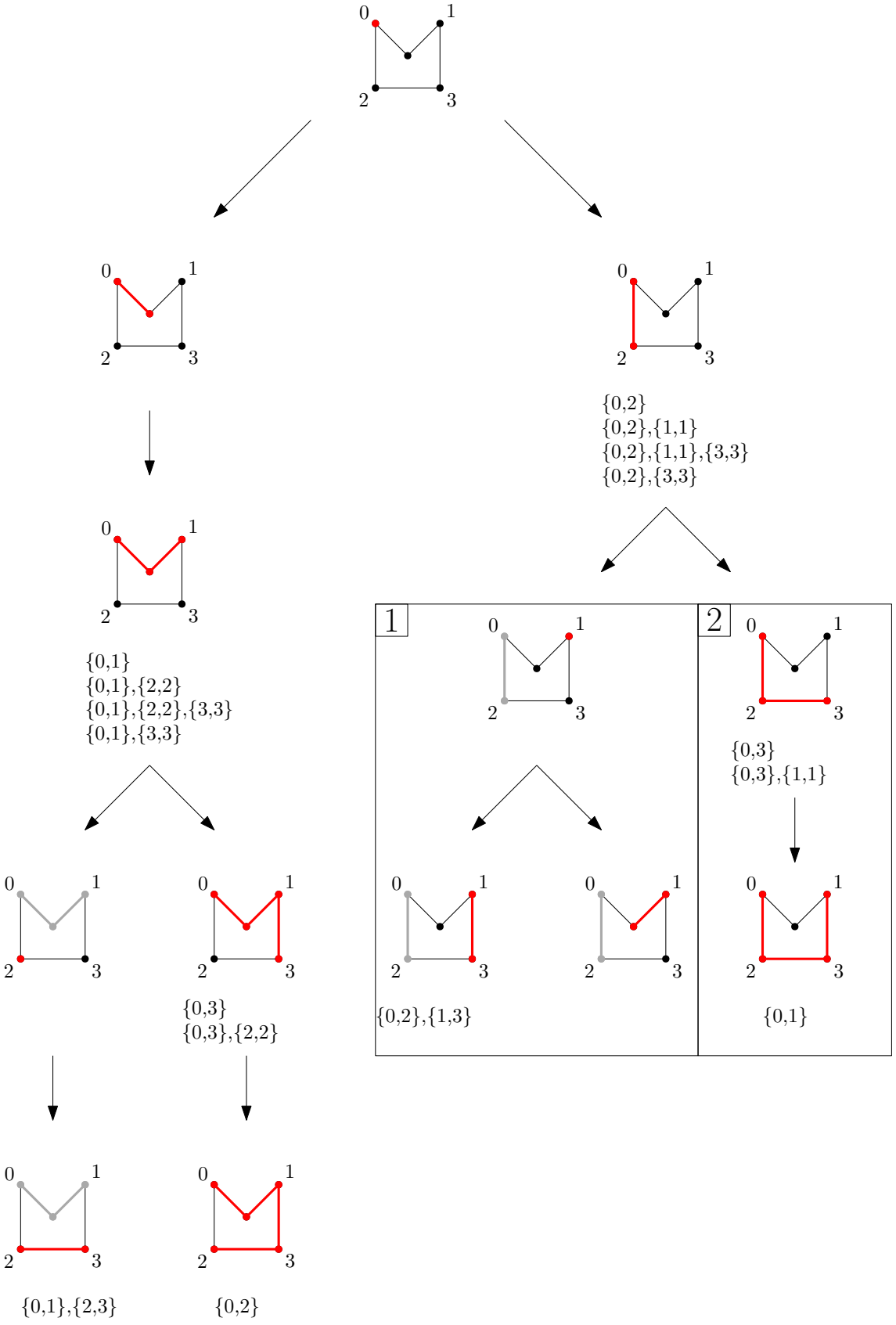

Figure 3.5: See section 3.4

also serves as a solution for $P = \{\{0,2\},\{1,1\}\}, \{\{0,2\},\{3,3\}\}$ and $\{\{0,2\},\{1,1\},\{3,3\}\}$. We also update these values accordingly. As we just finished a path LPDP-Search does two different things. First we start new paths from other unused boundary vertices that have a higher ID than the starting vertex of the current path. The candidates for this are 1 and 3. Both are larger than 0, but because of the reason already mentioned above we never start a path from the highest boundary vertex. This leaves the boundary vertex 1. In the picture we can see this branch of the search in the box that is labeled "1". In its first state we can see the completed $\{0,2\}$-path in gray and the newly started {1,.}-path in red. After this LPDP-Search also continues the search with the open $\{0,.\}$-path. This is seen in the box labeled "2" as LPDP-Search traverses the edge $\{2,3\}$.

We first look at the branch in box 1. Here we continue to search the graph as before. This means that we first look at the edge $\{1,3\}$. As $1 < 3$ this also allows us to complete the second path. This results a solution for $P = \{\{0,2\},\{1,3\}\}$ with the weight of 2. There are no other boundary vertices. This means that this is the only solution we have found and that we cannot start any new paths. We also cannot traverse the {1,.}-path any further. The recursion ends here. We backtrack to the first state in box 1. Here we can also traverse the edge from 1 to the unlabeled vertex. This vertex is not a boundary vertex and we cannot traverse the graph any further. This state is a dead end, which also ends the recursion.

We now look into the search branch in box 2 that continued with the {0,.}-path. We first traverse the edge $\{2,3\}$. As $0 < 3$ we can complete a {0,3}-path. This path is 0,2,3 and has a weight of 2. We update the solution for $P = \{\{0,3\}\}$ accordingly. As the boundary vertex 1 still remains unused we can also update the solution to $P = \{\{0,3\},\{1,1\}\}$. Again LPDP-Search can start new paths and continue with the current one. Here another optimization comes into play. According to the pseudocode in Figure 3.3 we would start a new {1,.}-path, but this path could never be completed as no other boundary vertices are left. The optimized LPDP-Search keeps track of this and does not start this path. So the only other state is the result of traversing the edge $\{1,3\}$. As $0 < 1$ we found a complete {0,1}-path and a solution 0,2,3,1 with the weight of 3 for $P = \{\{0,1\}\}$. There are no other boundary vertices. This means that we cannot start a new path or complete the {0,.}-path if we continue the search. The optimized LPDP-Search ends the recursion here.

3.4.2 Merging Two Blocks

Figure 3.6 shows how LPDP-Search solves a block of a level > 0. In green and yellow we can see the two boundary vertex cliques of the block's two subblocks. The green subblock has four boundary vertices: 0, 1, 4 and 5. The yellow subblock has three boundary vertices: 2, 3 and 6. The boundary vertices of each subblock form a clique. The block that we want to solve has the four boundary vertices: 0, 1, 2 and 3. In the figure a vertex that is the boundary vertex of both the block and a subblock is shown as a dot. A vertex that only is the boundary vertex of a subblock, but not of the block is shown as a circle. In the following text "boundary vertex" only describes the boundary vertices of the current block. The other vertices of the subblocks (4, 5 and 6) are considered normal vertices.

As with the level 0 block above we also would have to execute LPDP-Search from each boundary vertex (except the highest) in order to solve the block completely. Here we only show the search that is started from 0. Initially this search can traverse the edges $\{0,1\}$, $\{0,4\}$ and $\{0,5\}$. In order to reduce the complexity of the figure we also omit the two branches of the search that follow the edges $\{0,1\}$ and $\{0,5\}$. Instead we only look at the search branch that initially traverses the $\{0,4\}$ edge. In Figure 3.6 this is the second state from the top. It is to note that traversing this graph also continuously modifies the set P_{green} and P_{yellow}. At the beginning these sets are empty. Now that we traversed

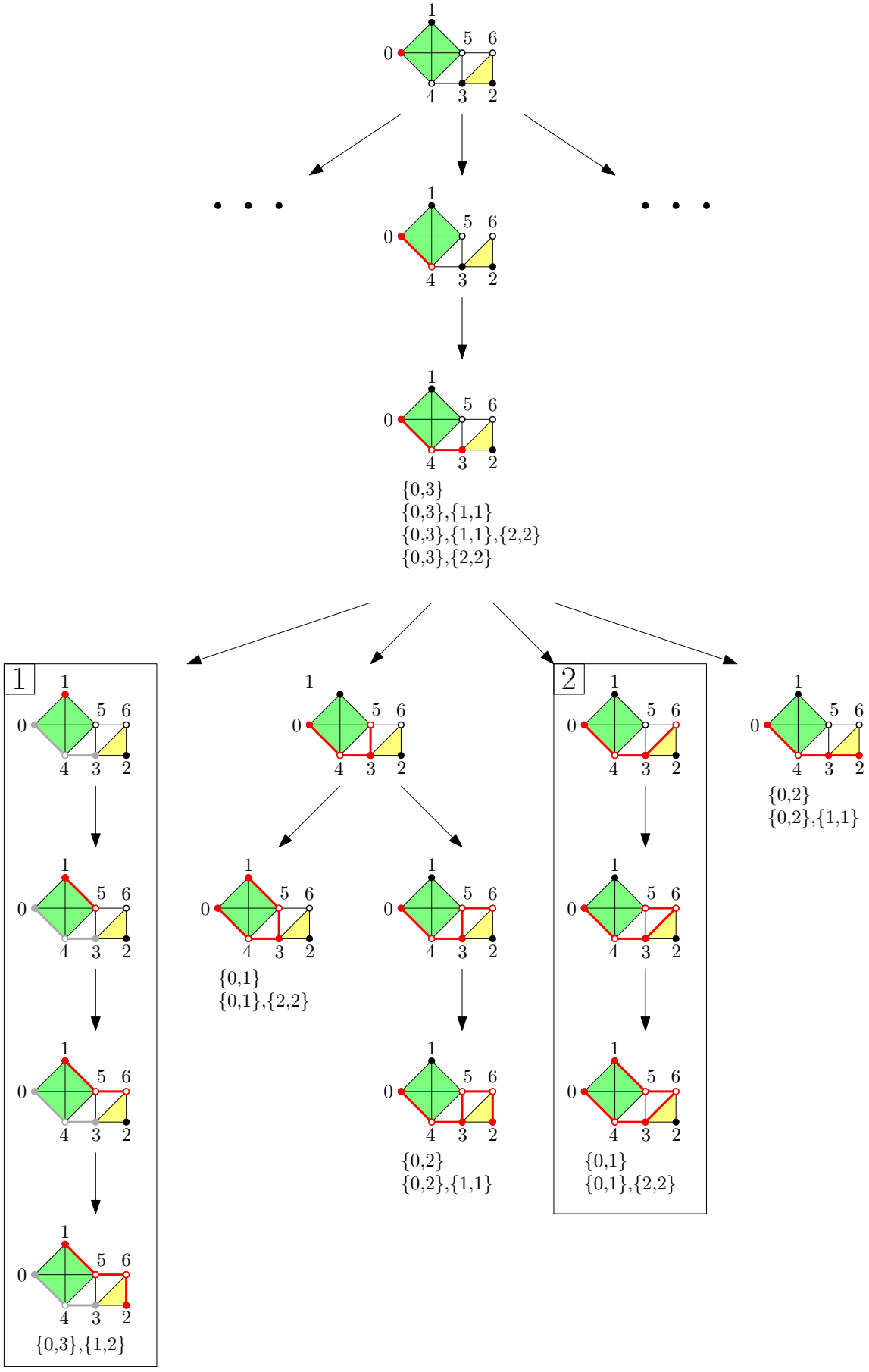

Figure 3.6: See section 3.4

the edge $\{0,4\}$ we add $\{0,4\}$ to P_{green}. This means that $P_{green} = \{\{0,4\}\}$. Before even traversing the edge we lookup if this solution for the green subblock exists. If it doesn't, the search would already end here. In this example we are going to assume that every possible solution for the subblocks exists. The edges that we can traverse next are $\{1,4\}$ and $\{3,4\}$. The edge $\{1,4\}$ is part of the green clique. Traversing this edge would result in $P_{green} = \{\{0,4\},\{1,4\}\}$. This P_{green} is unsolvable as it implies two non-intersecting paths: One path from 0 to 4 and one from 1 to 4. As they share the common vertex 4 these paths always intersect. We already discussed this in the observation 1. The optimized version of LPDP-Search that is presented later in section 5.2 simply prevents these cases by never traversing two clique-edges consecutively. Whether we use the optimized version or simply check if the solution exists the result is the same: We cannot traverse the edge $\{1,4\}$.

We can traverse the boundary edge $\{3,4\}$. As $0 < 3$ and 3 is a boundary vertex of our block we can complete a {0,3}-path. But ending the path here would induce the set $P_{yellow} = \{\{3,3\}\}$. We first have to check if this solution for the yellow subblock exists. As mentioned above we assume that it exists. We then look up the weight for each subblock's solution (for the current P_{green} and P_{yellow}). Let w_{green} and w_{yellow} be these weights. This means that we have found a solution for $P = \{\{0,3\}\}$ with the weight of $w_{green} + w_{yellow} + 1$. The 1 is the weight of the boundary edge $\{3,4\}$ that was used. Additionally, the boundary vertices 1 and 2 are unused. This means that we also found possible solutions for $P = \{\{0,3\},\{1,1\}\}, \{\{0,3\},\{2,2\}\}$ and $\{\{0,3\},\{1,1\},\{2,2\}\}$. It is to note that each of these P also modifies P_{green} and P_{yellow}. For example $P = \{\{0,3\},\{1,1\},\{2,2\}\}$ induces $P_{green} = \{\{0,4\},\{1,1\}\}$ and $P_{yellow} = \{\{3,3\},\{2,2\}\}$. We have to check again if these subsolutions even exist and what their weights w_{green} and w_{yellow} are.

After this LPDP-Search starts new paths and also continues the search with the current {0,.}-path. There are two unused boundary vertices 1 and 2. As a {2,.}-path cannot be completed we only start a new {1,.}-path. This branch of the search can be seen in box 1 in Figure 3.6. When continuing with the {0,.}-path we can traverse the edges $\{3,2\},\{3,5\}$ and $\{3,6\}$. We look further into the branch of the search that traverses the $\{3,6\}$ edge. This is shown in box 2.

But first we look into box 1. The finished {0,3}-path is shown in gray. We start a new {1,.}-path in red. We can only traverse the edge from 1 to 5. This induces the set $P_{green} = \{\{0,4\},\{1,5\}\}$. From there we continue with the boundary edge $\{5,6\}$ and then the clique-edge $\{6,2\}$. Traversing the latter one induces the set $P_{yellow} = \{\{2,6\},\{3,3\}\}$. As $1 < 2$ we also are able to complete the path. This means we have found a solution for $P = \{\{0,3\},\{1,2\}\}$. The are no unused boundary vertices. The search ends here as we cannot start new paths or continue with the current one.

Now we also look closer into the search branch in box 2. Here the LPDP-Search took the $\{3,6\}$ edge instead of starting a new {1,.}-path. This induces the set $P_{yellow} = \{\{3,6\}\}$. From there LPDP-Search can only traverse the edge to the vertex 5 as we cannot traverse two clique-edges consecutively. After this we traverse the edge $\{1,5\}$. This induces set $P_{green} = \{\{0,4\},\{1,5\}\}$. As $0 < 1$ and the boundary vertex 2 is still unused we have found solutions for $P = \{\{0,1\}\}$ and $\{\{0,1\},\{2,2\}\}$. The recursion ends here. We do not start a new path as 2 is the only boundary vertex left. A {2,.}-path can not be completed.

3.5 Space & Time Complexity

The worst case scenario for LPDP is a complete graph. If we partition such a graph, every block on every level is a clique. As long as there is more than one block in a level every vertex is also a boundary vertex. According to observation 2 this means we have to store

$\#P(n) = \sum_{k=1}^{\lfloor n/2 \rfloor} \frac{n!2^{n-3k}}{(n-2k)!k!}$ solutions for each block with n (boundary) vertices. Note that we start the sum with $k = 1$. The element of the sum with $k = 0$ represents the number of all P that do not contain any $\{x, y\}$ where $x \neq y$. These solutions always exist and have weight 0. We do not have to store them. This can be used to estimate the space complexity of the algorithm.

We did not find a good estimation of the time complexity of LPDP, but $\#P(n)$ also gives us a lower bound of it as we have to look at least once at each solution. We can also see that LPDP degenerates into exhDFS without the use of partitioning (a single level with one block $B = V$). This means that LPDP can always achieve the time complexity of exhDFS. We further look into the time complexity of LPDP for partitioned complete graphs in section 6.8. We show empirically that LPDP has a smaller time complexity than exhDFS.

3.6 Differences to Previous Work

The original LPDP from [Fie16] worked differently to its current form. Here we will explain the differences between the presented algorithm and the original. It is to note that our presentation of the original LPDP is different from [Fie16]. It still remains the same algorithm, but allows easier comparison to the current form of LPDP.

First "boundary vertices" are different: They are auxiliary vertices that are introduced in the graph to separate the partition blocks from each other. This is done by removing any boundary edge $\{x, y\}$ with the weight w and inserting a boundary vertex z and the edges $\{x, z\}$ and $\{z, y\}$ with the weights w and 0. We also attach such a vertex to the start- and target-vertex with an edge weight of 0. These boundary vertices can be seen in Figure 3.7. The figure shows the same example as Figure 3.2, but for the original LPDP. As seen in the right picture of Figure 3.7 we build an auxiliary graph the same way as before: We turn the boundary vertices of each block into a clique. The resulting auxiliary graph is a multigraph as there can be multiple edges between two vertices. An edge belongs to the block for which it was created. The induced boundary vertex pairs for the blocks are: $P_{green} = \{\{40, 41\}, \{42, 45\}\}$, $P_{yellow} = \{\{45, 46\}\}$, $P_{blue} = \{\{41, 42\}, \{46, 47\}\}$.

This is not a complete description of the original algorithm, but we can already see the important differences. For the same blocks the original algorithm creates an equal or greater amount of boundary vertices. So generally the original algorithm has to store more solutions per block and creates more complex auxiliary graphs. The experiments in section 6.6 compare the runtimes of the algorithms. The current version of LPDP performs significantly better.

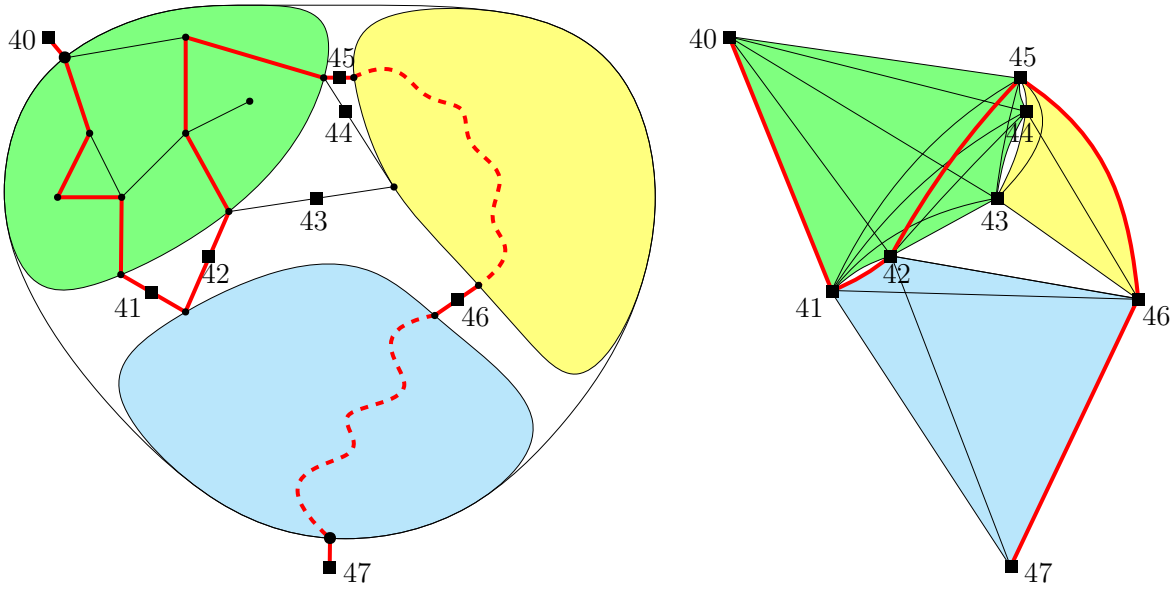

Figure 3.7: This figure shows the same example as Figure 3.2, but for the original LPDP version of [Fie16]. The boundary vertices that are inserted by the algorithm are shown as black squares. They are labeled with the numbers 40 to 47. A possible longest path between 40 and 47 is shown in red. The auxiliary graph created by the original LPDP is shown on the right. The boundary vertex pairs induced by the path are: $P_{green} = \{\{40, 41\}, \{42, 45\}\}$, $P_{yellow} = \{\{45, 46\}\}$, $P_{blue} = \{\{41, 42\}, \{46, 47\}\}$

4. Parallelization

The parallelization of the LPDP algorithm was done in two ways. First multiple blocks can be solved at the same time. Additionally, LPDP-Search from Figure 3.3, which is used to solve a block, can also be parallelized.

4.1 Solving Multiple Blocks

A block is only dependent on its subblocks. We can solve it once all of its subblocks have been solved. No information from any other block is needed. This allows us to solve multiple blocks independently of each other. How this form of parallelism was implemented is explained in section 5.3.

The question is how effective this form of parallelism can be. For example: If p% of a problem's serial runtime is spent solving a single block, solving multiple blocks in parallel cannot achieve a speedup higher than $\frac{100}{p}$. In order to test this we ran the serial LPDP solver on the set of problems that is used in the experiment section (6.5.4). LPDP had a time limit of 64 minutes to solve a problem. We only looked at problems that took more than 5 seconds to solve.

For each of the solved problems the plot in Figure 4.1 shows the percentage of the problem's runtime that is spent solving its most difficult block. The problems are sorted from lowest to highest percentage. From this plot we can see that achieving speedups over 2 would be impossible for most of the problems. In fact for almost half of the shown problems a single block makes up over 80% of their runtime. This means that parallelizing the execution of a single block is far more important than solving multiple blocks at the same time. The next section explains how this was done.

4.2 Parallelizing LPDP-Search

In order to solve a block LPDP starts the search shown in Figure 3.3 from each boundary vertex of the block (except the last). We could parallelize the solving of a block simply by running multiple of these searches at the same time. There are multiple problems with this approach: We could only use up to $n - 1$ threads when solving a block with n boundary vertices. This limits the speedup that could be achieved to $\frac{1}{n-1}$. Additionally, because of the optimization explained in section 3.3, the searches that start from boundary vertices with lower ID's usually take much longer than those started from higher boundary vertices. This would lead to bad load balancing and limit the possible speedup even further.

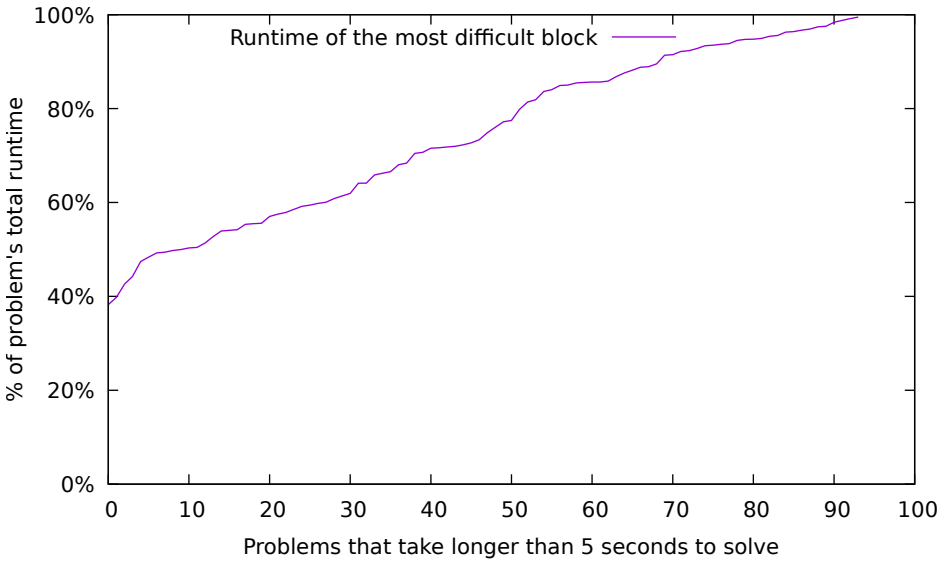

Figure 4.1: The plot is restricted to the problems that took the serial LPDP solver more than 5 seconds. For each of these problems the plot shows the percentage of the problem's runtime that is spent on solving its most difficult block. The problems are sorted from lowest to highest percentage.

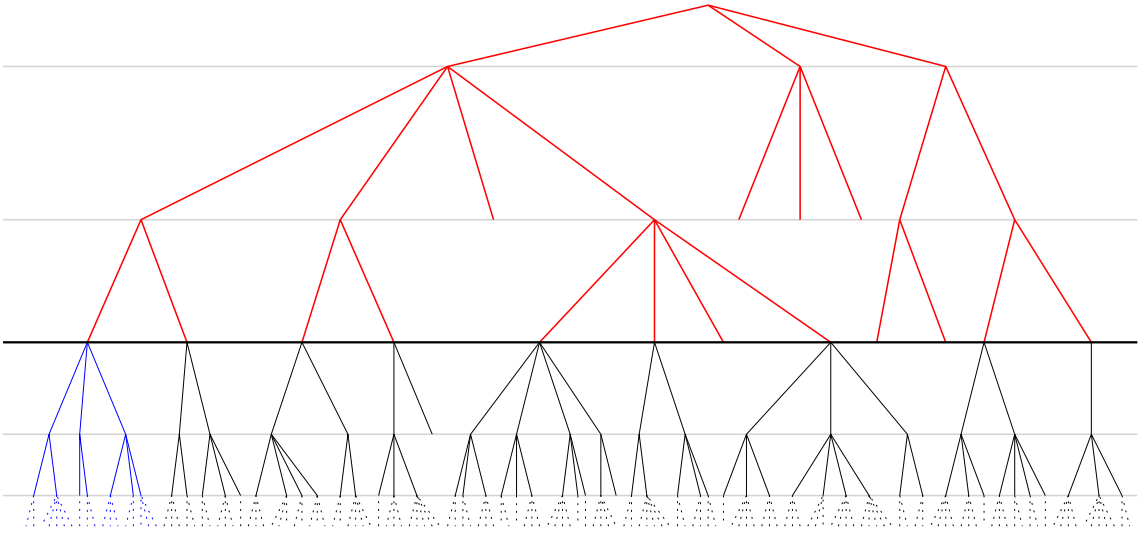

Figure 4.2: This tree is an example that is generated through the recursion of a LPDP-Search. The horizontal lines are the different levels of recursion. The root of the tree is the initial call of LPDP-Search(). Every edge represents a fuction call to LPDP-Search(). For example the initial LPDP-search() calls itself 3 times. This results in the 3 edges to the first horizontal line (first level of recursion). The call represented by the left edge calls LPDP-Search() 4 times. This results in 4 branches to the second horizontal line. The whole tree represents a full LPDP-Search. In red we can see a LPDP-Search that is limited to 3 levels of recursion.

An approach that does not have these problems is inspired by the "Cube and Conquer" approach for SAT-Solving that was presented by Heule et al. [HKWB11]. In this paper a SAT formula is partitioned into many subformulas. These subformulas can be solved in parallel. We can do the same for LPDP by partitioning the search space of LPDP-Search into many disjunct branches. We do this by running LPDP-Search from each boundary vertex with a limited recursion depth. Every time the search reaches a certain level of recursion the search stores its current context in a list and returns to the previous recursion level. Figure 4.2 shows an example of this. We can see a LPDP-Search limited to 3 recursions in red. A stored context represents all the data that allows us to continue the search at this point later on. On higher recursion levels the search works the same as before. The created list of contexts is then used as a queue. Each element of the queue represents a branch of the search that still has to be executed. One such branch can be seen in blue in Figure 4.2.

We execute these branches in parallel. Each time a thread finishes one branch it receives the next branch from the top of the queue. This automatically results in a form of load balancing as threads that execute faster branches simply end up executing more branches. In order for this to work well we need a large number of branches. But generating the branches should also not take too long. We have to choose the recursion depth limit accordingly. This could be done by initially choosing a small recursion depth limit. If the resulting list of branches is too small, we repeat the process with a higher and higher limit until the list has the necessary size. These repeats should not take too long as a small list of search branches would also indicate a short runtime of the restricted search. If we want to prevent the repeats, we could also continue the restricted LPDP-Search from each of the branches in the list until a higher recursion depth is reached. In the experiments of chapter 6 none of this was done. Instead the recursion depth limit was set to the fixed value 5. This has proven to be sufficient. Generating the branches only took a negligible amount of time but still resulted in good load balancing.

This form of parallelization requires a certain amount of synchronization between the threads. The serial implementation of LPDP uses hash tables to store the solutions for a block. Several threads can try to modify the same hash table entry at the same time. Introducing a concurrent hash table solves this problem.

5. Implementation

This chapter details our implementation of the LPDP algorithm. We implemented LPDP in C++. The code can be compiled as C++11 or C++14 code. Other vesions of C++ might also work, but were not tested. The parallelization of LPDP is based on a shared memory model. We mainly utilized OpenMP 4.5 [Ope15] in order to parallelize the algorithm. As an implementation of a concurrent hash table was needed we also made use of the Intel® Threading Building Blocks (TBB) library [Rei07] (libtbb-dev 2017∼U7∼8).

5.1 Data Structures

5.1.1 Block Hierarchy

If we want to solve multiple blocks of the partition at the same time, a single graph data structure is undesirable. Instead we represent each block as its own object. It only contains the data that is needed in order to solve the block. This includes pointers to all subblocks (the blocks of the lower partitioning level that make up the current block) and a hash table that contains the results after solving the block. This creates a tree-like data structure of blocks. The root of this tree is the singular block that contains the whole graph. When solved its hash table will contain the solution to the longest path problem. A block can only be solved once all of its subblocks have been solved. Because of this we solve the tree in a bottom up approach.

This data structure has the advantage that each thread only has knowledge of/access to the data of the block that it is currently solving. This allows the threads to not interfere with each other and increases cache locality. In order to implement this hierarchy efficiently we often have to transform the vertex IDs. For example a block of the hierarchy that contains n boundary vertices always labels them with $0, 1, \ldots, n-1$. We omit this in this thesis and just address every vertex with the ID that it has in the underlying graph.

5.1.2 Storing the Results

In order for our algorithm to work we need an efficient data structure to store the results of our search. When searching a block B we need to store one solution for each possible set of boundary node pairs P_B. There are multiple ways to do this. One way would be with a large array. This would require a fast way of calculating an array index for any given P_B. A big drawback of this method is that a block with n boundary nodes would always require an array of size $\#P(n) = \sum_{k=1}^{\lfloor n/2 \rfloor} \frac{n!2^{n-3k}}{(n-2k)!k!}$ (as mentioned in the observation 2

in section 3.2). However, if LPDP works on sparse graphs, there often exists no solution to a lot of the possible P_Bs. In this case large parts of the array would be empty, which would waste a lot of memory.

We also looked at a tree data structure as a way to store the solutions. As it is quite complex we only explain the basic idea and why we ultimately did not use it. The LPDP-Search algorithm always modifies P_B in a specific pattern. This would have allowed us to build a tree structure in such a way that LPDP-Search only keeps track of a single node of the tree. This node stores the solutions for the current P_B. Any time LPDP-Search modified P_B it only has to switch to one of the children nodes or to the parent node. The advantage of the tree would have been that we would not have to reserve space for unsolvable P_B. These nodes of the tree simply would not have existed. The problem is that LPDP-Search does not look up the solutions for the subblocks in the same pattern. Instead it requires random access to them. This is not possible with a tree. So the tree would have been good when solving a block B, but solving the parent block of B requires us to look up solutions in the tree of B. These lookups would have been extremely inefficient.

Instead LPDP stores the results in a hash table. A hash table stores $(key, value)$-pairs. In our case the key is a P_B and the value is the corresponding solution. Similar to the tree the hash table also has the advantage that we do not reserve space for P_Bs that do not have a solution. In addition we can randomly access the stored P_B. Something that is not possible with a tree. The next section talks about this hash table in more detail.

5.1.3 Concurrent Hash Table

For the reasons mentioned above we use a hash table to store $(P_B, solution)$-pairs for a set of boundary node pairs P_B from a block B. This section explains the hash table implementation and how the algorithm interacts with it.

For a block B with n boundary nodes we represent a set P_B as an array A[.] of n integers. Initially this array is filled with some number representing an empty entry (for example n). When adding a pair $\{x, y\}$ to the set P_B we assign A[x] = y and A[y] = x. Example: $n = 7$ and $P_B = \{\{0, 3\}, \{1, 1\}, \{2, 6\}\}$ is represented as the array $[3, 1, 6, 0, -, -, 2]$ where "$-$" is an empty entry.

The solutions have to be represented in two different ways. With a level 0 block the solution corresponding to a P_B is the paths in the underlying graph that connect the boundary node pairs given by P_B. For higher levels the solution is a P_{B_i} for each subblock B_i of B and a set of boundary edges (edges between these subblocks). We store both types of solutions as arrays of integers. Additionally, we also have to store the weight of each solution regardless of the level of the block. An example of such hash tables can be seen in Table 5.1. It shows possible hash tables for the graph from Figure 3.2. Figure 3.2 shows a longest path from the vertex 0 to 9. This path induces the following boundary vertex pairs in the subblocks $P_{green} = \{\{0, 1\}, \{2, 3\}\}$, $P_{yellow} = \{\{4, 6\}\}$, $P_{blue} = \{\{7, 7\}, \{8, 9\}\}$. Additionally, the path uses the following boundary edges: {1,7},{2,7},{3,4},{6,8}. The first table of 5.1 shows how we store this information in our hash table. For the sake of readability we do not use the array representations from above. The key of the hash table entry is the boundary vertex pair $P_B = \{\{0, 9\}\}$. As the block only has two boundary vertices the hash table only has this one entry. The solutions for $P_B = \{\}, \{\{0, 0\}\}, \{\{9, 9\}\}$ and $\{\{0, 0\}, \{9, 9\}\}$ are trivial and do not have to be stored. The weight of 19 is the cumulative weight of every subblock's solution plus the weights of the boundary edges. The hash tables for the subblocks can also be seen in Table 5.1. In this case the subblocks are level 0 blocks which is why their hash tables only store weights and paths. The cumulative weight of the three solutions is $7 + 3 + 5 = 15$. With our 4 boundary edges this results in a longest path of the length/weight $15 + 4 = 19$.

Key	Solution		
	Weight	Subblock-keys	Boundary edges
{{0,9}}	19	{{0,1},{2,3}}	{1,7},{2,7},{3,4},{6,8}
		{{4,6}}	
		{{7,7},{8,9}}	

Key	Solution	
	Weight	Paths
...		
{{0,1},{2,3}}	7	0,13,15,11,1 2,14,21,3
...		

Key	Solution	
	Weight	Paths
...		
{{4,6}}	3	4,22,24,6
...		

Key	Solution	
	Weight	Paths
...		
{{7,7},{8,9}}	5	8,27,31,32,33,9
...		

Table 5.1: Possible hash tables for the example given in Figure 3.2. Figure 3.2 shows a longest path from vertex 0 to 9. The first table shows how this path is stored under the key $P = \{\{0, 9\}\}$. The boundary vertex pairs that the path induces in each subblock can be seen in the column "Subblock-Keys". Additionally, we store all boundary edges that the path uses. The paths weight of 19 is the cumulative weight of every subblock's solution plus the weights of the boundary edges. The hash tables for the three subblocks are seen below. The subblocks are level 0 blocks which is why their hash tables only store weights and paths. These paths do not follow from Figure 3.2. The cumulative weight of the three solutions is $7 + 3 + 5 = 15$. With our 4 boundary edges this results in a longest path with the weight of 19.

If we only wanted to calculate the length of the longest path, we would only store the weights. Any other data about a solution is only necessary to reconstruct the longest path at the end of the algorithm. This would save a lot of memory.

The parallelization of LPDP as described in section 4.2 requires our hash table to allow concurrent access. We used the concurrent hash table implementation *concurrent_hash_map* from Intel® Threading Building Blocks (TBB) [Rei07]. The concurrent_hash_map can be accessed with a read- or a write-lock. A read-lock allows a thread reading access to a $(key, value)$-pair at the same time as other readers. A write-lock gives a thread exclusive read- and write-access. LPDP only interacts with concurrent_hash_map in two ways. First when solving a block we need to look up the weight of solutions in the hash table of the subblocks. This is done with a read-lock. When solving a block the hash tables of its subblocks are already completed and do not get modified anymore. This means synchronization is not even necessary at this point. Additionally, we need a way to insert a new solution into the hash table of the current block or update an already existing one. This happens in line 2 of Figure 3.4. Pseudocode of this operation can be seen in Figure 5.1. Hereby P is the set of boundary node pairs and S' a possible solution for it. The operation insert_or_update(P, S') starts by attempting to acquire a read-lock for P. If a pair (P, S) already is in the hash table, we acquire the read-lock and possibly have to update S accordingly. If no such pair exists, the attempt fails and we have to insert the pair (P, S'). In the first case we need to override the old solution S with the new one S' if the weight of S' is higher than that of S. In order to do this we first compare the weights. If we have to update S, we release the read-lock and acquire a write-lock. As S can be modified during the switch from read- to write-lock we have to compare the weights of S and S' a second time. If S still has a lower weight, we override it with the better solution S'.

In a previous implementation we just acquired a write-lock from the beginning. This would have circumvented having to check the weights twice, but led to problems if threads often called insert_or_update$(P, .)$ for the same P. The best example of this is the highest level block as only one possible P exists. Write-locks require exclusive access so the threads constantly blocked each other even if their solutions had the same or lesser weight than the one already in the hash table. This led to longer runtimes for certain problem-instances the more threads were used. Usually such an instance's runtime was largely spent solving the highest level block. Acquiring a read-lock first solved this problem.

If we call insert_or_update(P, S') and no (P, S)-pair exists in the hash table, acquiring the read-lock fails. In this case we have to insert the pair (P, S') into the table. We acquire a write-lock for P as it allows us to do this. We check if a pair (P, S) exists as another thread could have inserted it in the meantime. If it does, we update S the same way as above. If not, we finally insert the new pair (P, S') into the table.

Instead of using a concurrent hash table we could also use multiple non-concurrent hash tables. In this case every thread has its own separate hash table. After all threads have finished their search we would merge the hash tables into a one. In early testing this was not as effective as one concurrent hash table. It sometimes even led to increased runtimes compared to the sequential algorithm. Nevertheless this option could be useful when implementing a LPDP solver for a computer system in which each processor has its own private memory. In this case we might use a hybrid model: Each processor has one concurrent hash table. All threads of the processor store their results in this table. After the search is done all tables are merged into one.

```
Operation insert_or_update(P, S')
    get read-lock for P
    if (P, S)-pair exists in the hash table then
        if S.weight < S'.weight then
            release read-lock for P
            // here S could change, so S.weight has to be checked again
            get write-lock for P
            if S.weight < S'.weight then
                S ← S'
            end
            release write-lock for P
        end
    else // no read-lock acquired as (P, S)-pair did not exist
        get write-lock for P
        if (P, S)-pair exists now then
            if S.weight < S'.weight then
                S ← S'
            end
        else
            insert (P, S')-pair into hash table
        end
        release write-lock for P
    end
```

Figure 5.1: P is the set of boundary node pairs and S' a possible solution for it. The operation insert_or_update(P, S') describes how we insert or update an entry in a concurrent hash table. For more detail see section 5.1.3.

5.2 LPDP-Search

The pseudocode of LPDP-Search shown in Figure 3.3 was kept simple. This section will explain additional detail of how LPDP-Search was implemented and optimized. Line 7 in the pseudocode completes the current path from a to v. The loop in line 9 iterates through all boundary vertices $w > a$ and starts a new path from w. This new path can only be completed in another (unmarked) boundary vertex $z > w$. If such a boundary vertex does not exist, LPDP starts a path that cannot be finished. This results in a lot of unnecessary searching of the graph. We prevent this by first finding the largest unmarked boundary vertex $b > a$. If no such vertex exists: $b = a$. We then start new paths as in line 9, but only iterate through boundary vertices w where $b > w > a$. This way any path that is started has a possible endpoint in b.

Another problem arises if we think about the following case. The boundary vertex v is the only viable endpoint that is left for the current $\{a, .\}$-path. If we visit v, the current path is finished in line 7. But in line 12 we resume the search with the $\{a, .\}$-path. Now the path contains all of its possible endpoints, but we still continue the search for others in line 16. We prevent this by aborting the search in line 12 if $b = a$. This way determining b not only prevents us from starting unnecessary paths, but also allows us to abort the current path if no viable boundary vertex is left.

We also optimized the loop in line 16. The auxiliary graph exists of two different types of edges. First the boundary edges that connect boundary vertices of different blocks. Additionally, all boundary vertices of a block are connected to each other via clique-edges. Traversing a clique-edge $\{x, y\}$ of a block B_i induces the boundary vertex pair $\{x, y\} \in P_{B_i}$. If we traverse two such edges consecutively, this results in a set $\{x, y\}, \{y, z\} \in P_{B_i}$. According to observation 1 in section 3.2 there exists no solution for this set P_{B_i}. Therefore we prevent traversing two clique-edges consecutively by keeping track of the last edge in the current path. If it is a clique-edge, we only iterate over boundary edges in line 16. If not, we iterate over all edges.

Finally we discuss the If-statement in the first line of the pseudocode as it does not exists in our implementation. First the statement checks if the current vertex v is unmarked. In our implementation we check this before calling LPDP-Search(). This means we check if w is unmarked in line 10 and 17 before calling LPDP-Search(w). This prevents an "empty" function call and the overhead that is associated with it. The second part of the If-statement checks if solutions for all the P_{B_i}s exist. Instead our implementation only checks a P_{B_i} before we traverse the graph in a way that would modify it. If the hash table for block B_i does not contain a solution for the resulting P_{B_i}, we prevent the traversal. This way we make sure that LPDP-Search() is never called with an unsolvable P_{B_i}.

An optimization can be made to the update_solutions(.)-pseudocode in Figure 3.4. Here we do not have to calculate certain solutions. A pair $\{v, v\} \in P_B$ represents a path that enters and leaves the block B through the boundary vertex v. But usually this is only possible if at least two edges $\{a, v\}$ and $\{v, b\}$ with $a, b \notin B$ exist. There is an exception to this if v is the start s or target t vertex. In this case a path can start or end in v. This means that $\{v, v\} \in P_B$ is possible as long as a single edge $\{a, v\}$ with $a \notin B$ exists. In all other cases a solution with $\{v, v\} \in P_B$ cannot be part of the longest path. We can prevent calculating and storing these useless solutions by modifying the definition of u_j in Figure 3.4 accordingly. We simply ignore the u_j for which the conditions above do not hold true.

5.3 Parallelization

The first form of parallelism that was discussed in section 4 is solving multiple blocks independently of each other. This form of parallelism was implemented level-wise. We start by solving all the block on the lowest level. All of them can be solved in parallel as they do not have any subblocks. Once they are solved we switch to the next level. All blocks on this level can also be solved in parallel as any subblocks have already been solved on the previous level. We repeat the process until all blocks are solved. We have implemented this with two for-loops. The outer loop iterates over the levels. The inner loop iterates over all blocks of the current level and solves them. We parallelize the inner for-loop with the OpenMP directive "#pragma omp parallel for schedule(dynamic, 1)". This directive parallelizes the loop with the dynamic scheduling strategy. First the scheduler assigns one loop iteration to each thread. When a thread is finished it retrieves the next unassigned iteration. We chose this strategy as iterations can have vastly different runtimes. It could lead to a high workload imbalance if we assigned fixed chunks of the iterations to the threads beforehand. Dynamic scheduling automatically acts as a form of load balancing.

This level-wise approach is not an optimal way to implement this form of parallelism as we have to wait for all blocks of a level to be solved before switching to the next level. Instead we could already start to solve blocks from the higher levels once their subblocks have been processed. We decided not to implement this for reasons of simplicity and because the speedup that can be achieved through this form of parallelism is very limited anyways (see section 4).

The second form of parallelism is the solving a single block with multiple threads. The idea behind it was already explained in detail in section 4.2. The parallel execution of the generated search branches is also done with the "#pragma omp parallel for schedule(dynamic, 1)"-directive. Here we talk about how we combined both forms of parallelism in our solver: We still work with the level-wise approach that was explained above. When solving a level we first try to predict how difficult each block will be to solve. We do this by counting the boundary vertices of the block's subblocks. If the number of boundary vertices is above a certain threshold (20 in the experiments), we consider a block to be "difficult". Otherwise the block is considered "easy". If a block is difficult, all threads are used to solve it. Once no difficult blocks are left on the current level we use the first form of parallelism to solve the easy blocks in parallel (1 thread per block).

5.4 Reconstructing the Longest Path

After we solved all blocks the hash table of the block on the highest level contains a single entry with the key {{s,t}} (if a path from s to t exists). This entry represents our longest path from s to t. Its weight also is the length of the longest path. An example of this can be seen in Table 5.1. The longest path is stored under the key {{0,9}} and has a weight of 19. But we still have to reconstruct the actual path / sequence of vertices. We do this through recursive hash table lookups. We start with the above mentioned hash table entry. We then look up its stored subblock-keys in their corresponding hash tables. We repeat this process until we reach hash tables of level 0 blocks. During these lookups we store the boundary edges and paths of each visited hash table entry. Hereby any boundary edge {x,y} turns into a path x,y. For Table 5.1 this results in the paths:

$1, 7$
$2, 7$
$3, 4$
$6, 8$
$0, 13, 15, 11, 1$

$2, 14, 21, 3$
$4, 22, 24, 6$
$8, 27, 31, 32, 33, 9$

These paths are all segments of the longest path. They all start and end in a boundary vertex. As the graph is undirected a path segment can be read forwards or backwards as needed. We reconstruct the longest path from these path segments. The longest path start with the segment that begins with the start-vertex s. Let $s, v_1, ..., v_n$ be this segment. We search for the one other segment that ends with v_n. We add this segment $v_n, w_1, ..., w_m$ to the existing path, which results in the path $s, v_1, ..., v_n, w_1, ..., w_m$. In this way we continue adding segments to the path until all path segments are used up. This results in the longest path $s, ..., t$. The following is an example for the path segments listed above:

$0, 13, 15, 11, 1$
$0, 13, 15, 11, 1, 7$
$0, 13, 15, 11, 1, 7, 2$
$0, 13, 15, 11, 1, 7, 2, 14, 21, 3$
$0, 13, 15, 11, 1, 7, 2, 14, 21, 3, 4$
$0, 13, 15, 11, 1, 7, 2, 14, 21, 3, 4, 22, 24, 6$
$0, 13, 15, 11, 1, 7, 2, 14, 21, 3, 4, 22, 24, 6, 8$
$0, 13, 15, 11, 1, 7, 2, 14, 21, 3, 4, 22, 24, 6, 8, 27, 31, 32, 33, 9$

The main difficulty of this process is finding the path segment that ends with a certain boundary vertex. We do this by constructing an array $A[]$ beforehand. $A[v]$ contains pointers to the segments that end in v. If no such segment exists, $A[v]$ is empty. If $v = s$ or t, $A[v]$ only contains one pointer. Otherwise $A[v]$ contains two pointers. This means that $A[]$ allows us to find a segment in constant time.

5.5 Other Optimizations

Currently LPDP does unnecessary calculations for parts of the graph that are unreachable from s or t. For example: If two boundary vertices of a block are unreachable, we still calculate a path between them. We prevent these unnecessary calculations by changing the definition for boundary vertices. A vertex now also has to be reachable from s (and t) in order to be a boundary vertex. We can check this by running a depth first search from s before starting LPDP. Any vertex that is visited by the search is reachable. Before we implemented this optimization we already did this to check if t is reachable. If t is unreachable, we are finished as no longest path exists. Otherwise we start LPDP with the reachable boundary vertices. Now a block might still contain unreachable vertices, but LPDP executes no calculations for them.

We also could have done the same by deleting any unreachable vertices. This optimization had a great impact on the runtime of grids in our experiments, especially on the grids with 40% obstacles. As we randomly delete 40% of the vertices large parts of the graph can be unreachable. Otherwise the optimization had no effect as the other problems have no unreachable vertices by construction.

The necessity for another optimization reveals itself when looking at a block that only has one subblock. When we solve such a block we just end up with a copy of the subblock's hash table. Instead we just pass on a pointer to the hash table of the subblock.

6. Experiments

We used GCC (GNU Compiler Collection) 7.3.0 to compile our code. The LPDP code was compiled as C++14 code. For optimization we used the compiler options "-O3" and "-march=native". The Intel® Threading Building Blocks (TBB) library [Rei07] was provided with the libtbb-dev package (2017∼U7∼8). We used OpenMP 4.5 [Ope15].

6.1 Hardware

The experiments were run on two different machines. The specifications of the computers are listed below. Unless otherwise mentioned any experiment was run on the computer A.

Computer A:

- 4 Intel® Xeon® E5-4640 processors (2.4 GHz with 8 cores each)
- 32 cores / 64 threads
- 512 GB RAM
- Ubuntu 18.04.1 LTS (64 bit)

Computer B:

- AMD Ryzen^{TM} 7 1800x processor (3.6 GHz with 8 cores)
- 8 cores / 16 threads
- 64 GB RAM
- Ubuntu 18.04.1 LTS (64 bit)

6.2 Plots and Tables

This section explains some of the plot and table types that are used to illustrate the results of the experiments.

Cactus plot:
Example: Figure 6.3. A cactus plot shows the execution times of a set of problems in ascending order. The plot has two axes. The first shows the index of a problem. The

second axis shows the execution time of a problem. A point (x, t) means that the x-th fastest solved problem was solved in the time t.

Speedup plot:
Example: Figure 6.5. Same as a cactus plot, but shows the speedups of parallel algorithms over their serial counterpart instead of execution times. A point (x, s) means that s was the x-th largest speedup that was measured.

Speedup table:
Example: Table 6.1. A speedup table presents the speedups achieved by LPDP over its serial version when using multiple threads. The first column shows the number of threads used for the parallel execution. The "Parallel Solved" column gives the number of all instances that were solved by the parallel algorithm within a given time limit. The "Both Solved" column does the same for the instances that were solved by both the parallel and the serial algorithm. The next three columns show the average, total and median speedups of the parallel algorithm for all problems that were solved by both algorithms. The last three columns do the same, but only for "big" problems. When using n threads a problem is considered big if the serial solver took more than $n \cdot t$ seconds (where t is some arbitrary time). This is done because of the additional overhead involved in using multiple threads. Parallelization might only pay off for more difficult problems. With a larger number of threads this overhead is also expected to be larger. The threshold of $n \cdot t$ seconds allows us to filter out the problems that do not warrant a certain level of parallelization.

6.3 Partitioning

The hierarchical partitioning required for LPDP was generated with KaHIP - Karlsruhe High Quality Partitioning - [SS13]. We used KaHIP version 1.00. Given a weighted graph and a number n KaHIP partitions the vertices of the graph into n blocks. KaHIP tries to find these blocks in such a way that they contain an equal number of vertices, while minimizing the cumulative weight of all edges between vertices of different blocks.

For LPDP we want a hierarchical partitioning with a small number of edges between the blocks. We create this hierarchical partitioning using a bottom up approach: In order to minimize the number of edges between the blocks we set all edge-weights of the original graph to 1. We then partition the graph with KaHIP into a large number of blocks. This represents our lowest level of partitioning. The next level of partitioning is calculated with an auxiliary graph. This auxiliary graph contains one vertex for each block of the current level. If there are n edges between two blocks in the original graph, we insert an edge with weight n in the auxiliary graph between the two corresponding vertices. We then partition the auxiliary graph with KaHIP, combining blocks of the current level into larger blocks. This creates the next level of the partitioning. We repeat this process until only a single block is left.

In the experiments the number of blocks in the initial partition was chosen based on the graph. For higher levels we then partitioned every auxiliary graph into $\frac{\#vertices}{2}$ blocks. This results in a partition hierarchy that halves the number of blocks with each level. This means that a block usually is created by combining two lower level blocks.

For partitioning we use KaHIP 1.0 with the parameters –preconfiguration=strong –imbalance=0.1. The first parameter results in higher quality partitions in exchange for longer execution time. The second allows higher imbalance between the number of vertices in each block. These parameters have given good results in earlier experiments [Fie16]. Additionally, there is an option to give KaHIP a specific duration that it will spend partitioning the graph. In order to make use of this we first partitioned the graph with the parameters mentioned above. Then we multiplied the KaHIPs runtime by a factor of 5

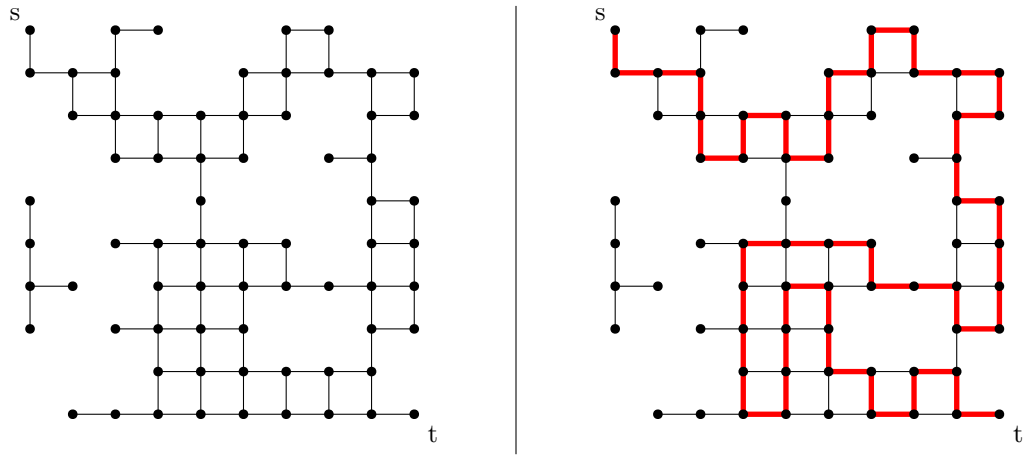

Figure 6.1: Example of a grid of size 10. 30% of the vertices have been deleted. The picture on the right shows a longest path between s and t in red.

and ran KaHIP again with the forced additional execution time. This has also been shown in [Fie16] to improve the quality of the partition.

In this thesis the runtime of KaHIP is not counted towards the total runtime of LPDP. In [Fie16] we have shown that especially for more difficult problems the runtime of KaHIP is negligible compared to the runtime of LPDP. We precalculate a partitioning for each problem. This way the serial and parallel LPDP solvers do not use different partitions for the same problem. This allows us to accurately measure parallel speedups.

6.4 The LPDP-Solver

The LPDP solver has several possible parameters. Whether a block is solved serially or in parallel is decided with a *threshold* parameter. If the subblocks of a block have fewer total boundary vertices than the threshold, the block is solved serially. Otherwise the block is solved in parallel. This threshold was set to 20 boundary vertices.
If a block is solved in parallel, the search space is partitioned as it is explained in section 4.2. The number of *steps* that the solver takes for this was set to 5.
The third parameter is the number of *threads* that the solver utilizes.

6.5 Benchmark Problems

This section introduces the classes of problems that were used to evaluate the LPDP algorithm.

6.5.1 Grids

Grids represent two dimensional mazes. In order to generate a grid instance we need two parameters: the size of the grid $n \geq 0$ and the obstacle-probability $0 \leq p \leq 1$. We start with an undirected graph that represents a $n \times n$ grid of vertices. Each vertex is connected with an edge to each of its up to four neighboring vertices. All edges have a weight of 1. We call the top left vertex s and the bottom right vertex t. We then randomly delete $\lfloor p \cdot n^2 \rfloor$ vertices. These missing vertices represent obstacles/blocked passages in the maze. Afterwards we check if a path from s to t exists. If it exists, the graph and the start and target vertices s and t represent our generated LP instance. If not, we repeat the process. Figure 6.1 shows a grid of size 10 with $p = 0.3$. On the right a possible solution can be seen in red.

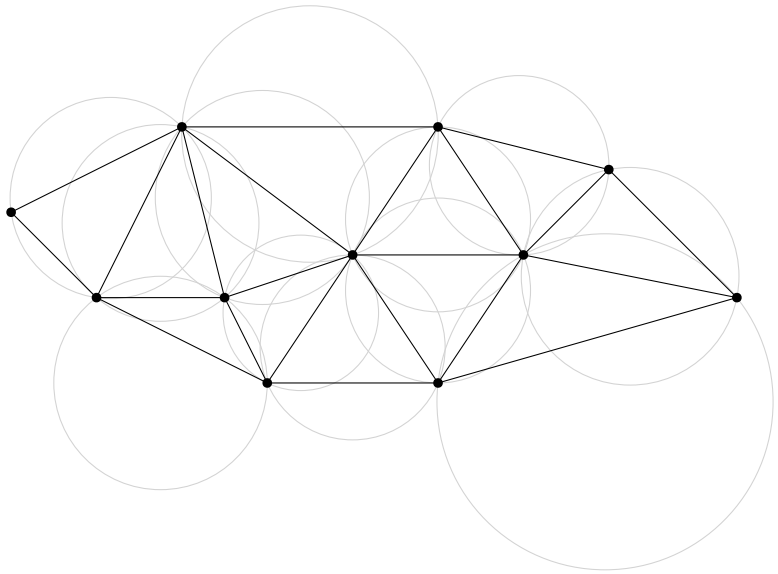

Figure 6.2: Example of a Delaunay triangulation of size 11. We can see a circle for every triangle that passes through its three vertices. The condition for a Delaunay triangulation is met as none of the circles contains a vertex of another triangle.

In [SKP+14] Stern et al. used grids in order to evaluate their LP solvers. We also use grids as we assume that they are easy to partition in a way that is suitable for the LPDP algorithm. In fact in [Fie16] we compared LPDP to the LP solvers from [SKP+14]. During this evaluation we measured the biggest runtime advantage for LPDP when we solved grids.

6.5.2 Delaunay Triangulations

A triangulation of a set of points in a two dimensional plane is a Delaunay triangulation if all triangles fulfill the Delaunay condition. A triangle fulfills the Delaunay condition if the unique circle that passes through all three of its vertices contains no other points. An example can be seen in Figure 6.2. We use an implementation of the Bowyer–Watson algorithm [Bow81] [Wat81] to generate random Delaunay triangulations:

In order to generate a random Delaunay triangulation of size n we first generate n points. Each point has a random float coordinate $(x, y) \in [0, 1000] \times [0, 1000]$. We initialize the plane with a super triangle that completely contains the area $[0, 1000] \times [0, 1000]$. As the plane contains no other points at this moment this super triangle fulfills the Delaunay condition. The algorithm then incrementally inserts points into the plane. Each time we insert a point we remove all triangles that violate the Delaunay condition. This creates a convex polygonal cavity inside the triangulation. We connect each point of this polygon with the newly inserted point. This creates new triangles that satisfy the Delaunay condition. After n points have been inserted we remove the vertices of the initial super triangle.

We obtain a LP problem instance by assigning every edge a weight of 1 and choosing two random vertices as start- and target-vertex.

It is to note that Delaunay triangulations do not represent a worst case for planar graphs. The exterior face of a Delaunay triangulation is not necessarily a triangle. In that case more edges could be added without the graph losing its planarity.

6.5.3 Roads & Word Association Graphs

Roads and Word Association Graphs are subgraphs the road network of New York [SKP+14] and the word association graph from [LoWA]. The road network is a weighted graph, while the word association graph is unweighted. We extract a subgraph of size n from these

graphs by starting a breadth-first search from a random vertex. The search runs until n vertices have been visited. The subgraph is induced by these vertices. The start-vertex of the LP instance is the origin of the search. The target-vertex is chosen at random.

6.5.4 Benchmark

The main set of problems used in this thesis consists of the four different problem classes that were explained above. For grid instances we used two different obstacle-probabilities. One set of grids with 30% obstacles and one with 40%. The first set contains grids of the sizes $10, 15, 20, 25, ..., 50$. There are 10 grids per size. The second set has the sizes $10, 15, ...130$ with the same number of instances per size. So overall 340 grids. The set of Delaunay triangulations contains 98 triangulations with $3, 4, 5, ..., 100$ vertices. We also tested 150 road graphs with $2, 4, 6, ..., 300$ vertices and 60 word association graphs. The word association graphs are subgraphs of $10, 20, ..., 60$ vertices with 10 instances per size class. This results in 648 LP instances overall.

6.6 Performance Compared to Previous Implementations

This section illustrates the increased performance of our current LPDP implementation compared to previous implementations. There are three different implementations of LPDP that also represent three different solvers. We name the solvers: old1, old2 and cur.

"old1" represents the original implementation from the bachelor thesis [Fie16]. This implementation introduces boundary vertices for every boundary edge like it is described in section 3.6 and should be the slowest. Later this implementation was updated to work like the current algorithm from section 3.2. The solver "old2" represents this version of the implementation. For this thesis LPDP was implemented from scratch. The "cur" solver represents this current implementation of LPDP. Compared to "old2" it also contains all optimizations that are mentioned in this thesis. The "cur" solver does not use any parallelization for this evaluation as we want to compare the serial implementations.

We tried to solve the set of problems presented in section 6.5.4 with all three solvers. Each solver was given a time limit of 32 minutes per problem. The results can be seen in Figure 6.3. The plot shows the sorted runtimes for the three solvers. Our current implementation of LPDP clearly is superior to the previous ones even without the use of parallelization. Overall "cur" solved 614 of the 648 problems. "old2" solved 573 and "old1" only solved 533 problems.

6.7 Parallel Speedups

We ran the benchmark described in section 6.5.4. In order to measure the speedups achieved through parallelization we ran the solver multiple times on each problem. We first ran the serial solver (1 thread) and then doubled the number of threads with each additional run. This was done until 64 threads, the maximum number of threads that the hardware can support, were reached. It is to mention that the computer (see section 6.1) only has 32 cores. Simultaneous multithreading is used to support 2 simultaneous threads per core. This should reduce the maximum achievable speedup as two threads have to share the resources of one core. The serial solver had 64 minutes to solve each problem. The parallel solvers only had a time limit of 32 minutes.

Table 6.1 gives a first overview of the results. In the second column of the table we see the number of problems that were solved by each of the parallel solvers. As expected this number increases as we increase the number of threads. With the 64 thread solver 15 problems remain unsolved. The third column shows the number of problems that were also solved by the serial algorithm. With 2 threads we initially solve fewer problems than

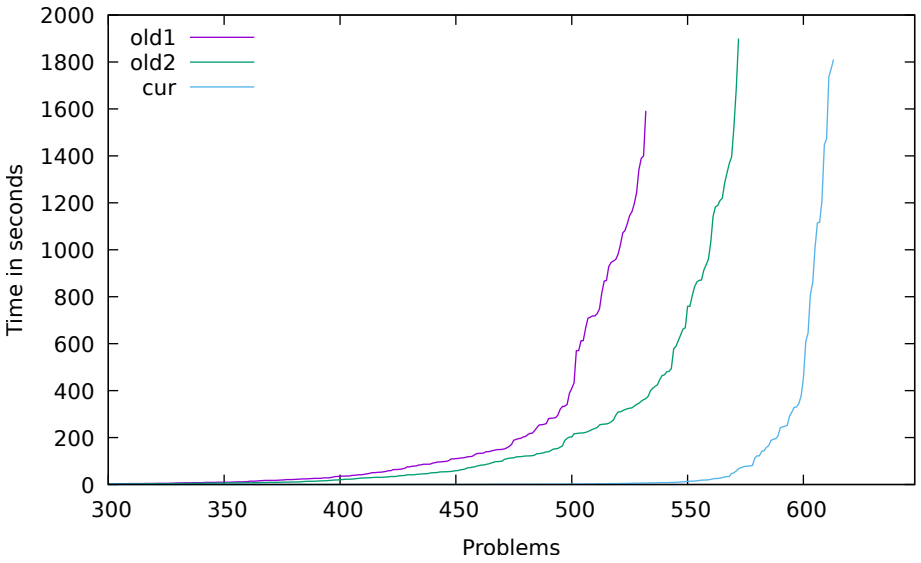

Figure 6.3: A cactus plot as it is explained in section 6.2. This plots shows the sorted runtimes for each of the serial LPDP solvers old1, old2 and cur. old1 is the original implementation of LPDP. old2 is an updated variant of it. Finally cur is the current implementation of LPDP used in this thesis.

Threads	Parallel Solved	Both Solved	Speedup All			Speedup Big		
			Avg.	Tot.	Med.	Avg.	Tot.	Med.
2	618	618	1.038	1.360	1.031	1.335	1.363	1.331
4	624	621	1.392	2.623	1.224	2.542	2.638	2.564
8	627	621	1.788	4.833	1.189	4.707	4.913	4.720
16	628	621	2.257	8.287	1.127	8.097	8.569	8.208
32	629	621	2.344	10.714	0.987	11.272	11.553	11.519
64	633	621	2.474	11.691	0.889	15.180	13.512	14.665

Table 6.1: Speedup table as it is defined in section 6.2. The serial version of LPDP had a time limit of 64 minutes for each problem. All parallel versions had a time limit of 32 minutes. A solver with n threads considers a problem "big" if the serial solver took more than $n \cdot 5$ seconds to solve it.

with one. This is because of the decreased time limit given to the parallel solvers. From now on we will only look at this subset of problems as only they can be used to calculate the speedups achieved through parallelization. The average, total and median speedups can be seen in column 4, 5 and 6. The total speedup of a solver is how much faster it solved all the problems compared to the serial solver. For 2 threads we see a total speedup of 1.360. Doubling the thread count results in an increase of the total speedup by 92.9%, 84.3%, 71.5%, 29.3% and 9.1%. We see that the initial jump from 2 to 4 threads almost doubles the speedup. The speedup gain per jump then slightly decreases until 16 threads are reached. At this point the total speedup is roughly half the number of threads (8.287). Further increasing the number of threads has a smaller effect on the total speedup. 32 threads still result in a gain of 29.3%. Especially the final jump from 32 to 64 threads with 9.1% only does little. We end with a total speedup of 11.691.

When looking at the average speedup per problem we see that it stays below 2.5 for all solvers. This is vastly different from the total speedup. This indicates that many problems in our benchmark set are relatively easy to solve. The overhead of the parallelization makes up a large part of the runtime for these problems. This keeps the average speedup small. The total speedup on the other hand is dominated by a smaller number of difficult problems which make up a large part of the total runtime of the benchmark.

The same thing can be seen when looking at the median speedups. Initially there is almost no speedup. With 4 threads the median speedup increases, but then starts to drop again. The 32 and 64 thread solver even have a median speedup below 1. This means that they are slower than the single threaded solver for at least half of the problems. From the data we know that none of these are difficult problems. LPDP solves them so fast that they do not warrant the necessary runtime overhead of parallelization.

In order to filter out these problems that are too easy to benefit from parallelization we restrict ourselves to "big" problems. For a solver with $n > 1$ threads we call a problem big if the serial solver took more than $5 \cdot n$ seconds to solve it. So 10, 20, 40, 80, 160 and 320 seconds for 1, 2, 4, 8, 16 and 64 threads. The threshold for a big problem increases with the number of threads as a higher thread count only pays off for more difficult problems. This can be seen in the steady decrease in the median speedup from 2 threads onwards.

The average, total and median speedups for big problems can be seen in column 7, 8 and 9 of the table. The total speedups for big problems are higher overall. The percentage gains when doubling the number of threads also are higher: 93.5%, 86.2%, 74.4%, 34.8% and 17.0%. Especially the last jump from 32 to 64 threads now increases the total speedup by 17.0% compared to the 9.1% before. The biggest difference can be seen for the average and median speedups. Now they are relatively similar to the total speedups. An interesting point is that for all numbers of threads except for 64 the average and median speedup is slightly lower than the total speedup. For 64 threads both are higher than the total speedup. This means that some of the easier big problems give us higher speedups than the more difficult ones.

The plots in Figure 6.4 show the runtimes for all solvers. The first plot further illustrates the low average and median speedups that were measured. We can clearly see the increased overhead induced by a higher number of threads. For the fastest solved problems each increase of the thread count leads to higher runtimes. For the most difficult problems this is reversed. This can be seen in more detail in the second plot. We can also see that the curves for 32 and 64 threads are very similar for large problems. This is expected as the total speedup between 32 and 64 threads only increased by 9.1%.

The first plot in Figure 6.5 shows the sorted speedups per solver. The second plot only shows the speedups restricted to the "big" problems. In the first plot we see that the solvers with fewer threads result in relatively flat curves. They grow very slowly and seem

to approach a certain value. For solvers with higher thread counts the curves stay far below their measured total speedup for most of the problems, but have a steep increase for the last 100 or so problems.

The second plot shows the speedup for the "big" problems. We see a similar pattern as before. For lower thread counts we see a relatively flat curve near the measured total speedup. For higher thread counts this curve starts far below their total speedup and ends far above it. This might mean that the effectiveness of our parallel solver highly dependents on the given problem. For 64 threads the lowest speedup in the plot is just 2.679. The highest is 31.236. One speculation is that the lower speedups are the result of problems where the threads constantly block each other as they try to update the same result in the concurrent hash table. The necessary synchronization would slow down the solver significantly. A better implementation of the hash table might mitigate this problem. But even then the speedup of 2.679 is an outlier. The runtimes for this outlier are $377.41, 280.47, 146.45, 87.151, 57.74, 76.88$ and 140.90 seconds in order of increasing thread count. We can see that the speedup is the greatest for 16 threads. After this the runtimes increase again. This is the result of a single block. As we increase the number of threads the runtimes of the other blocks decrease as expected. This is not the case for this block. The runtime for this block is 122.48 seconds out of the 140.90 seconds for 64 threads. We assume that this is because we do not partition the search space of this block efficiently. Modifying the number of *steps* that we take to partition the search space may help.

6.8 Cliques

Something that was not previously done is to look at the worst case performance of LPDP. Cliques represent such a worst case for LPDP. If we partition a clique into multiple blocks, every node is a boundary node. If we do not use partitioning and view the clique as a single block, LPDP degenerates into the bruteforce approach of exhaustive DFS. We ran LPDP and exhDFS on cliques of various sizes with a time limit of one hour. A clique's edges had random integers between 0 and 100 as their weights. The start- and target-vertices were also chosen at random (except that they could not be the same vertex). When using LPDP we partition the clique into two blocks. The first block consists of $\lfloor size \cdot p \rfloor$ vertices. The other vertices make up the second block. We tested LPDP with both $p = 0.5$ and 0.75. One block contains the start-vertex. The other contains the target-vertex. We measured the runtimes and the number of steps that the algorithms take. Hereby an algorithm makes one step each time it traverses an edge. As LPDP can update multiple solutions when traversing a single edge we also count each update as an additional step. This experiment was conducted on the computer B from section 6.1. The results can be seen in the tables 6.2. We first look at the results for p=0.5: While exhDFS is faster for all tested clique sizes the execution time of exhDFS increases faster than that of LPDP. For 8 vertices the runtime of exhDFS is only 14.8% that of LPDP. This percentage continuously increases until 84.2% is reached for the 15 vertex clique. With larger cliques we can assume that LPDP will eventually be faster than exhDFS. The rising step based speedup in the second table also indicates this. We actually measure fewer steps for LPDP than exhDFS in all cases. The higher runtimes of LPDP are explained by the fact that a step of LPDP requires time consuming hash table lookups. A step of exhDFS does not and is therefore much faster.

When looking at LPDP (p=0.75) we see a better results: This time LPDP is faster than exhDFS for cliques of size 11 and higher. For the 15 vertex clique LPDP is 13.167 times as fast as exhDFS. Additionally, LPDP solves the cliques 16 and 17. It is to note that the runtime speedup actually slightly decreases from clique 11 to 12. For 11 vertices the speedup is 1.098. This decreases to 1.095 for 12 vertices. This might be an outlier. For

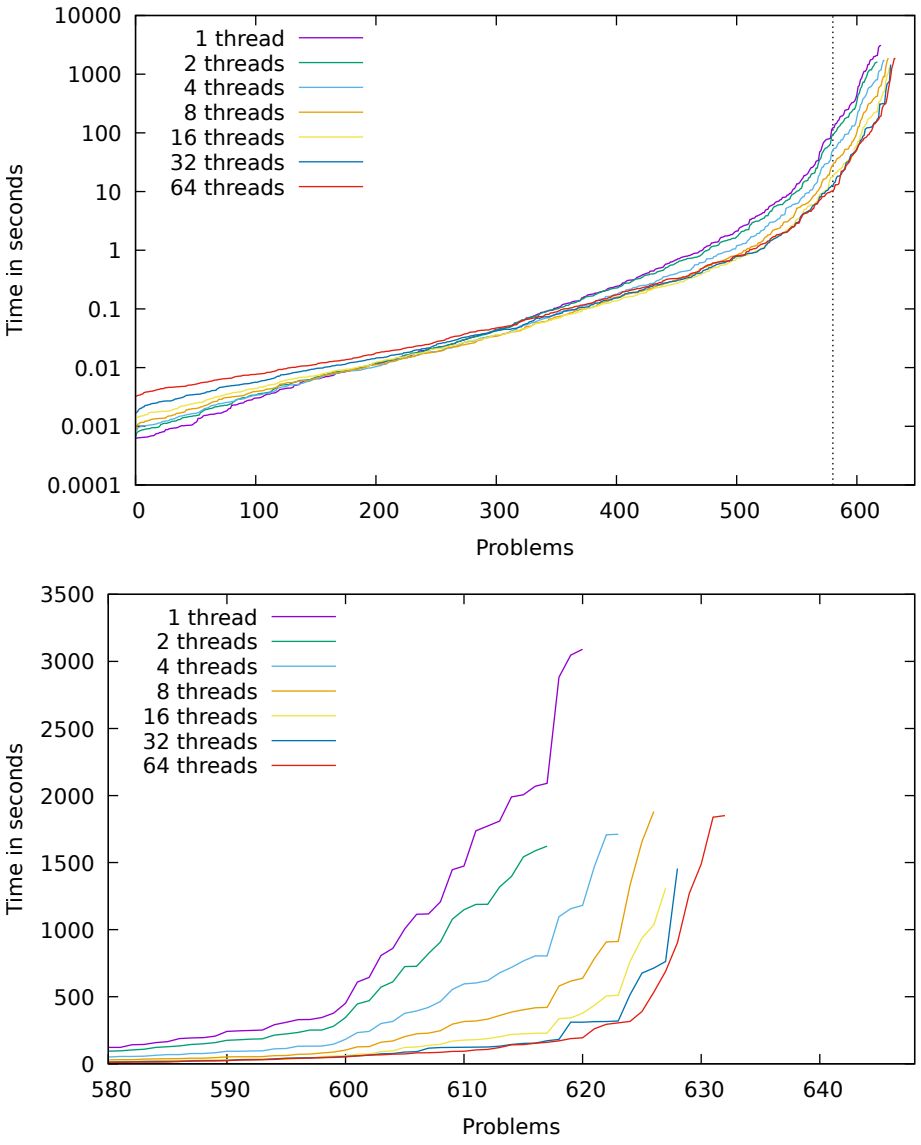

Figure 6.4: Cactus plots as they are explained in section 6.2. The sorted runtimes for the LPDP solver with 1, 2, 4, 8, 16, 32 and 64 threads are shown. The set of 648 problems is given in section 6.5.4. The solver with 1 thread had a time limit of 3840 seconds / 64 minutes per problem. All other solvers were given a time limit of 1920 seconds / 32 minutes. The first plot has a logarithmic scale and shows the runtimes for all solved problems. The part of the plot to the right of the vertical dotted line can be seen in the second plot with a linear scale.

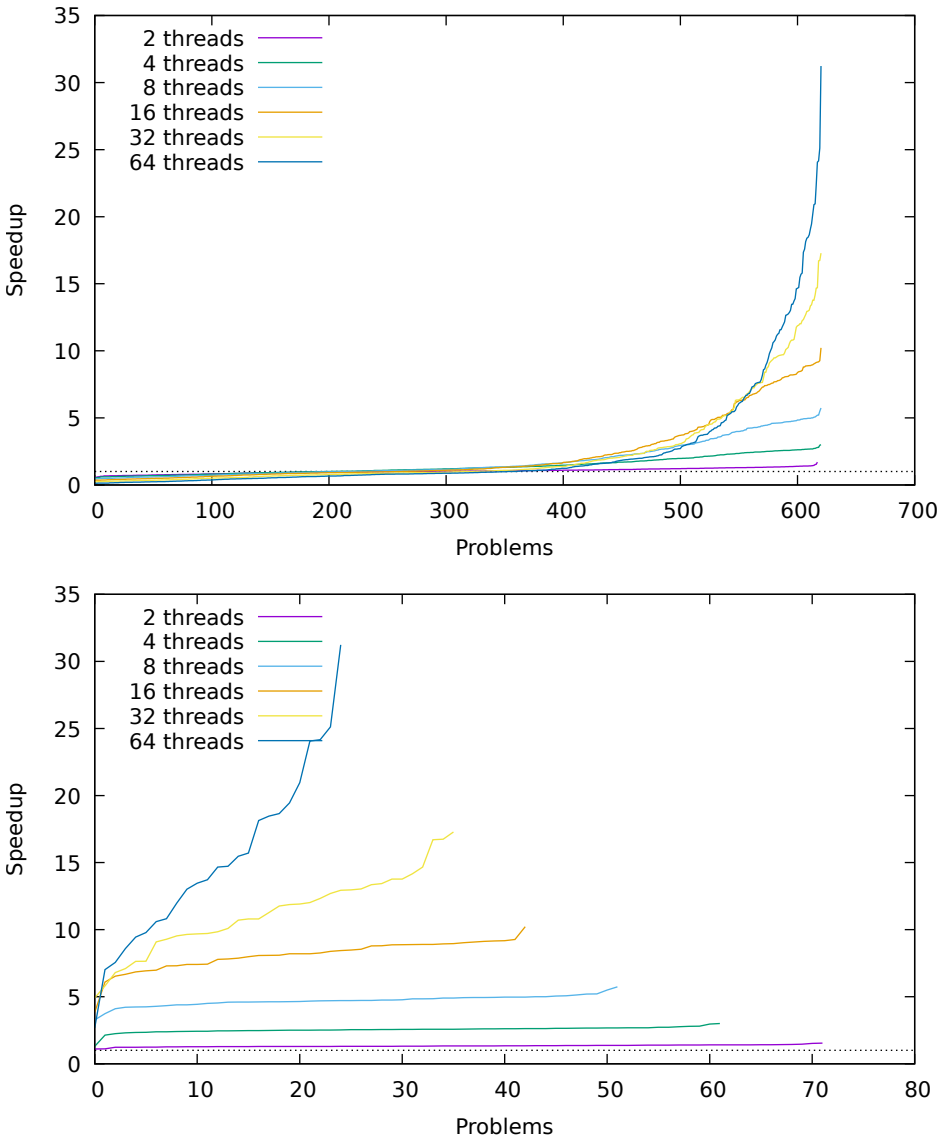

Figure 6.5: The plots show the sorted speedups per solver for the problems given in section 6.5.4. The first plot does this for all problems. The second plot is restricted to "big" problems. A solver with n threads considers a problem "big" if the serial solver took more than $n \cdot 5$ seconds to solve it. The dotted line represents a speedup of 1. We ran the LPDP solver with 1, 2, 4, 8, 16, 32 and 64 threads. The serial solver had a time limit of 64 minutes per problem. Otherwise LPDP had 32 minutes.

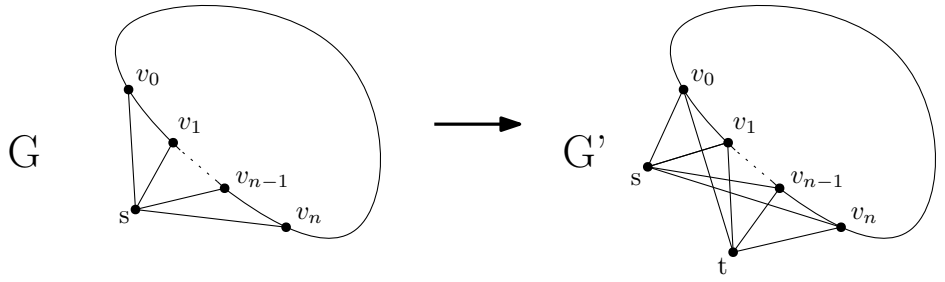

Figure 6.6: Example of our transformation from the Hamiltonian Cycle Problem to LP. We chose a vertex s in the graph $G = (V, E)$. We then build the graph G' by introducing the vertex t as shown above. $G' := (V \cup \{t\}, E')$ where $E' := E \cup \{\{t, v\} \mid \{s, v\} \in E\}$

p=0.5 these two values (0.426 and 0.428) are also very close to each other. Otherwise the speedups are steadily increasing. When looking at the step counts we also see better speedups for p=0.75 than 0.5. For 8 vertices LPDP(p=0.75) actually uses more steps than exhDFS. For all cliques after that LPDP(p=0.75) gives far better step based speedups than LPDP(p=0.5). For 15 vertices we measure a step based speedup of 45.840 compared to the 4.6 for p=0.5. This means that LPDP(p=0.75) only requires about a tenth of the number of steps that LPDP(p=0.5) does.

While LPDP(p=0.75) seems to be far better it is also to mention that it has a higher memory usage than LPDP(p=0.5). As explained in section 3.5 we need to store $\#P(n) = \sum_{k=1}^{\lfloor n/2 \rfloor} \frac{n!2^{n-3k}}{(n-2k)!k!}$ solutions in the hash table of a block with n boundary vertices. For example the 15 vertex clique and p=0.5 requires us to store $\#P(7) + \#P(8) = 1850 + 7193 = 9043$ solutions for the two blocks. With p=0.75 we already need to store $\#P(11) + \#P(4) = 538078 + 43 = 538121$ solutions. This means that searching for an optimal partitioning for cliques probably requires us to find a balance between the time and space complexity of the algorithm.

6.9 Hamiltonian Cycle Problem

The Hamiltonian Cycle Problem (HCP) is the problem of finding a Hamiltonian cycle in a given graph $G := (V, E)$. A Hamiltonian cycle is a cycle that contains each vertex of the graph exactly once. Here the graphs are undirected and do not contain loops. We can solve HCP instances with the LPDP algorithm by transforming them into LP instances. We do this the following way:
Choose a arbitrary vertex $s \in V$. Construct the graph $G' := (V \cup \{t\}, E')$ where $E' := E \cup \{\{t, v\} \mid \{s, v\} \in E\}$. This can be seen in Figure 6.6.
(Except for one special case) Searching for a Hamiltonian cycle in G is equivalent to searching a longest path $s, v_1, v_2, ..., v_{|V|-1}, t$ ($v_i \in V$) in G'. The corresponding Hamiltonian cycle to this path is $s, v_1, v_2, ..., v_{|V|-1}, s$. There is one exception to this: One edge alone does not constitute a cycle. Therefore a graph with $|V| = 2$ cannot contain a (Hamiltonian) cycle. Yet one could still find a longest path s, v_1, t in the transformed graph.

In order to test LPDP capabilities to solve HCP instances we tried to solve the FHCP Challenge Set [Hay18]. The FHCP Challenge Set is a collection of 1001 HCP instances. These instances have been specifically designed to be difficult to solve with common HCP approaches. Each one of the three approaches transforms the instance into an instance of the Traveling Salesman Problem. Then Concorde [ABCC95], the Snakes and Ladders Heuristic [BEF+14] and the Lin-Kernighan Traveling Salesman Heuristic [LK73] are used to solve the instance. If at least two of the algorithms struggled to solve an instance, it was

clique size	exhDFS	LPDP (p=0.5)		LPDP (p=0.75)	
	runtime [s]	runtime [s]	speedup	runtime [s]	speedup
8	<0.001	0.001	0.148	0.001	0.102
9	<0.001	0.002	0.209	0.002	0.262
10	0.005	0.015	0.369	0.007	0.758
11	0.034	0.080	0.426	0.031	1.098
12	0.278	0.650	0.428	0.254	1.095
13	3.107	5.641	0.551	0.527	5.895
14	37.761	61.869	0.610	3.682	10.255
15	535.457	635.768	0.842	40.667	13.167
16	N/A	N/A	N/A	568.151	N/A
17	N/A	N/A	N/A	1081.140	N/A

clique size	exhDFS	LPDP (p=0.5)		LPDP (p=0.75)	
	steps	steps	speedup	steps	speedup
8	3 914	3 608	1.085	5 828	0.672
9	27 400	19 241	1.424	11 532	2.376
10	219 202	133 934	1.637	62 552	3.504
11	1 972 820	926 395	2.130	478 234	4.125
12	19 728 202	8 207 542	2.404	4 760 073	4.145
13	217 010 224	69 521 926	3.121	8 659 923	25.059
14	2 604 122 690	735 066 046	3.543	66 442 136	39.194
15	33 853 594 972	7 359 580 750	4.600	738 510 944	45.840
16	N/A	N/A	N/A	9 995 560 186	N/A
17	N/A	N/A	N/A	16 468 631 539	N/A

Table 6.2: The results of running exhDFS and LPDP on cliques of size 8 to 17. We measured the runtime and number of steps that the algorithms take. We used a simple partition for LPDP: We divide the clique into two blocks. The start and target vertex are not in the same blocks. The first block consists of $\lfloor size \cdot p \rfloor$ vertices. The other vertices make up the second block. We tested LPDP with $p = 0.5$ and 0.75. The first table shows the runtimes of the algorithms and the speedups achieved by LPDP over exhDFS. The second table does the same with the number of steps and the step based speedup between LPDP and exhDFS.

included in the FHCP Challenge Set. There was a one year long competition to solve the set that ended on the 30th September 2016. The top five submissions were able to solve 385, 464, 488, 614 and 985 of the 1001 graphs.

We transformed the problems in the FHCP set into LP instances and tried to solve them with the LPDP solver. LPDP used 64 threads and had a time limit of 5 minutes per instance. This resulted in 403 of 1001 instances being solved. This would have granted LPDP the 5th place in the competition. It is to note that all solved instances were solved within a few seconds. 358 of the instances were solved in under one second. Only four instances were solved in over two seconds. Still these four runtimes were far below the time limit with 2.032, 2.079, 3.001 and 6.700 seconds. This could indicate the LPDP has applications in solving certain classes of HCP instances, but is ineffective for others.

For comparison we also tried to solve FHCP with Concorde. We did not compile Concorde, but used the executable version of Concorde. We ran Concorde with default parameters, except for a fixed seed ("-s 99"). Concorde also had a time limit of 5 minutes per problem. We transformed the HCP instances into TSP instances by turning the HCP's graph into a clique. Every edge that existed in the original graph is assigned a weight of 0. Any edge that had to be inserted has a weight of 1. Concorde solved 56 problems. Only 26 problems were solved by Concorde and LPDP. The runtimes for these problems can be seen in table 6.3. We can see that LPDP often had a large advantage over Concorde for these problems. There are 30 problems that were solved by Concorde, but not by LPDP.

Overall we can say that LPDP and Concorde have very different applications. LPDP itself might be effective in solving some HCP classes, but certainly does not represent an universal HCP solver.

Graph ID	LPDP [s]	Concorde [s]
1	0.017	1.300
2	0.029	0.060
3	0.020	21.650
4	0.071	1.740
5	0.016	185.270
6	0.017	0.330
8	0.052	6.400
9	0.020	205.550
10	0.023	3.170
14	0.035	0.140
18	0.036	0.170
23	0.025	0.200
27	0.035	0.240
31	0.027	0.260
35	0.067	15.480
39	0.051	7.340
47	0.058	0.510
52	0.063	11.680
56	0.046	0.460
61	0.051	0.500
78	0.065	0.610
93	0.055	7.780
114	0.066	93.420
125	0.070	47.500
138	0.101	27.090
184	0.103	55.170

Table 6.3: Runtimes for the 26 FHCP problems that were solved both by LPDP and Concorde within 5 minutes.

7. Conclusion

This thesis presented an improved LPDP algorithm and how it can be parallelized. In the experiments our the sequential LPDP solver performed far better than previous implementations. Our parallel version of LPDP achieved reasonable speedups compared to the sequential solver. We executed the parallel solver with 2, 4, 8, 16, 32 and 64 threads. The initial doubling of the thread count nearly doubled the achieved speedup. As expected each doubling after that had a smaller effect on the speedup. This is the result of the synchronization that is needed between the threads. We also looked into the worst case performance of LPDP. Clique graphs represent this worst case of LPDP. Partitioning a clique still gives LPDP an advantage over the bruteforce approach. We also presented a formula that allows us to calculate LPDP's worst case space complexity. Additionally, we discovered possible applications of LPDP in solving the Hamiltonian cycle problem (HCP). LPDP could perform well for HCP instances that are considered difficult for common HCP approaches.

7.1 Future Work

In our current implementation the majority of the synchronization happens through the concurrent hash tables that our solver uses. We could experiment with different concurrent hash table implementations in order to improve the speedups that can be achieved. In order to decide if a block is solved in parallel we currently simply count the number of boundary vertices. Finding a better way to estimate the difficulty of a block could also improve the solver's performance. We could assign a different number of threads based on the block's difficulty.

In section 4.2 we already presented a way of ensuring that the search space of LPDP-Search is partitioned into a large enough number of search branches. Additionally, we may want to predict the possible solutions that a branch can find. For example: If two branches will find solutions for the same sets of P, they will also access the same hash table entries. We probably should not solve these two branches at the same time as this would require a lot of synchronization. In this manner we could schedule the execution of the search branches in a way that minimizes the necessary synchronization between the threads.

Something that was not mentioned in this thesis is that LPDP could be well suited for solving multiple LP queries on the same graph. For example: We first run the algorithm without any start- or target-vertex. Then we want to calculate the longest path between two vertices s and t. We already calculated the solutions for any block that does not

contain s or t. If a block contains s or t, we only have to recalculate it if the vertices were not boundary vertices of the block already. Additionally, we probably can recalculate these blocks faster with the solutions that we already calculated. For multiple queries $(s_1, t_1), (s_2, t_2), \ldots, (s_n, t_n)$ this could lead to a large speedup compared to rerunning LPDP for every query.

One could also implement a version of LPDP for distributed memory systems. We already explained an approach to doing this in the end of section 5.1.3.

Finally we could look into other problems that are related to the longest path problems. LPDP has already shown some potential for the Hamiltonian cycle problem. LPDP might find applications for related problems. Other than that determining the time complexity of LPDP is also important.

Bibliography

[ABCC95] David Applegate, Robert Bixby, Vašek Chvátal, and William Cook. *Finding cuts in the TSP (A preliminary report)*, volume 95. Citeseer, 1995.

[BEF+14] Pouya Baniasadi, Vladimir Ejov, Jerzy A. Filar, Michael Haythorpe, and Serguei Rossomakhine. Deterministic "snakes and ladders" heuristic for the hamiltonian cycle problem. *Mathematical Programming Computation*, 6(1):55–75, Mar 2014.

[Bow81] A. Bowyer. Computing dirichlet tessellations*. *The Computer Journal*, 24(2):162–166, 1981.

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

[CHM51] S. Chowla, I. N. Herstein, and W. K. Moore. On recursions connected with symmetric groups i. *Canadian Journal of Mathematics*, 3:328–334, Feb 1951.

[Fie16] K. Fieger. Using Graph Partitioning to Accelerate Longest Path Search. Bachelor's Thesis, Karlsruhe Institute of Technology, 2016.

[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.

[Hay18] M. Haythorpe. Fhcp challenge set: The first set of structurally difficult instances of the hamiltonian cycle problem. *Bulletin of the ICA*, 83:98–107, 2018.

[HKWB11] Marijn Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: guiding cdcl sat solvers by lookaheads, 12 2011.

[Kau58] William H. Kautz. Unit-distance error-checking codes. *IRE Trans. Electronic Computers*, 7:179–180, 1958.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., 1973.

[LK73] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Oper. Res.*, 21(2):498–516, April 1973.

[LoWA] University of Milano Laboratory of Web Algorithms. Datasets.

[Ope15] OpenMP Architecture Review Board. OpenMP application programming interface version 4.5, November 2015.

[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.

[Rei07] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[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.

[Wat81] D. F. Watson. Computing the n-dimensional delaunay tessellation with application to voronoi polytopes*. *The Computer Journal*, 24(2):167–172, 1981.

[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.