Bachelor Thesis

Winter Semester 2016/17

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Hitting Set Problems on Graphs

TU Kaiserslautern
Department of Mathematics
January 7, 2017

Handed in by: Oliver Bachtler
Student No.: 388409
Mail: obachtle@rhrk.uni-kl.de
Phone: +49 176-47791009

Supervisor: Prof. Dr. Sven O. Krumke
Optimization Group
TU Kaiserslautern

Abstract

Given a universe U and a collection $\mathcal{S}$ of subsets of U, a hitting set is a set $A \subseteq U$ such that $A \cap S \neq \emptyset$ for all $S \in \mathcal{S}$. The hitting set problem is a well known $\mathcal{NP}$-complete problem that asks to find a minimum hitting set for a given instance $(U, \mathcal{S})$. If the universe U is the set of nodes of a graph and a set $S \in \mathcal{S}$ corresponds to the set of nodes on a path of the graph with $|S| = k$, we call this problem a k-path cover. The aim of this thesis is to regard these special cases of the hitting set problem in detail.

To this end the first part formally introduces the hitting set and path cover problems and summarises some known results. Next the so called primal-dual method, and how it can be used to approximate hitting set problems in general, is described. It is subsequently applied to the path cover problems on paths, trees, and general graphs and the resulting approximation guarantees are determined. Afterwards the complexity of these problems is analysed and they are either shown to be $\mathcal{NP}$-hard or a polynomial time algorithm is presented. The final part covers a few possible restrictions that are sufficient to make $\mathcal{NP}$-hard instances on trees solvable in polynomial time.

It turns out that, not too surprisingly, k-path covers on general graphs are hard and the primal-dual method gives a k-approximation for these. While there are instances for trees where an approximation factor of k is reached as well, these problems can even be solved optimally in several cases, especially when allowing a few additional restrictions. The case of paths leads to an approximation guarantee of two (or even slightly better). However, these problems can be solved to optimality in linear time.

Contents

1. Introduction

Regard the following problem in E-mobility: Electric vehicles have a limited cruising range, often less than 200 kilometres. Hence, to be able to travel from a certain location A to a destination B, battery loading stations are required at regular intervals along the path from A to B. If we model the area we are regarding, Germany for example, as a graph, we would like to ensure that all paths in the graph that are longer than the cruising range of such a vehicle contain a battery loading station. This would make it possible to drive from any point to any other on any desired route. Seeing as installing battery loading stations is an expensive task, the aim is to find locations such that they cover all of these paths and such that they minimise the installation costs.

In this context it seems useful to regard the hitting set problem. Given a universe U of elements and a collection $\mathcal{S}$ of subsets of U, a hitting set is a subset $A \subseteq U$ that hits every set $S \in \mathcal{S}$, so $A \cap S \neq \emptyset$. A solution to the hitting set problem is a minimum hitting set, either inclusion-wise minimal or minimal with respect to some weight function. Special cases of the hitting set problem are the k-path cover problems presented in [FNS14a], in which the universe is the set of nodes of a graph and a set in $\mathcal{S}$ corresponds to the set of nodes on a path containing k nodes in the graph. The paths in question differ depending on the specific instance of the problem, for example, they can be be all paths between all pairs of vertices or just the shortest paths between certain distinguished pairs.

Armed with these problems we take another look at the problem of finding locations for battery loading stations: With the somewhat simplifying assumption that the nodes are evenly spaced along the paths and $k-1$ is approximately the cruising range divided by the distance of two nodes, one of the long paths described previously corresponds to a path of k or more nodes in the graph. Consequently an optimal placement of the loading stations just corresponds to a solution of the k-path cover problem where $\mathcal{S}$ contains the node sets of all paths with k nodes. If we restrict the problem to the shortest path version, it will still be possible to drive from any point to any other, however, not necessarily along every possible route. As a result the amount of stations required will probably be less, but it most likely will no longer be possible to drive along the most scenic route between two locations as this probably is not the shortest route and might not contain a sufficient amount of loading stations.

Goemans and Williamson presented an approximation algorithm based on the primal-dual method for the general hitting set problem in [GW97] and applied it to several network design problems. It has been shown in [FNS14a] that these k-path cover problems are $\mathcal{NP}$-hard and that the primal-dual method provides a k-approximation.

In this thesis we take a look at the k-path cover problems on paths, trees and general graphs. We apply the primal-dual method to these instances and determine whether the k-approximation is best possible or if the method, in fact, leads to better results. We also take a look at which of these problems are actually $\mathcal{NP}$-hard and which can be solved in

polynomial time. Afterwards we regard a few slight restrictions that reduce the complexity of certain instances considerably.

In Chapter 2 we formally introduce the hitting set and path cover problems as well as show or cite several known results from [APMS+99], [Fei98], [RS97], [Sto15], and the aforementioned [FNS14a]. In Chapter 3 we present the primal-dual method, from [GW97], with its approximation guarantees and apply it to the path cover problem on paths, trees, and general graphs in Chapter 4. Determining which of these problems are actually $\mathcal{NP}$-hard and which of them can be solved optimally in polynomial time is the content of Chapter 5. Finally Chapter 6 takes a closer look at the problem on trees, describing some bad properties as well as two modifications that simplify an $\mathcal{NP}$-hard version enough to make it solvable in polynomial time, using results from [KMP+02], [Gav74] and [KN09].

2. The Hitting Set Problem

To begin we give an overview of the hitting set problem including its complexity and a basic approximation algorithm. We then describe a different model, called the *successive hitting set model*, as shown in [Sto15]. Finally we introduce the hitting set variants we are going to take a closer look at in later chapters. These are based on [FNS14a].

2.1. Basics

First we formally define the hitting set problem and introduce a few variations. Afterwards we show that the decision problem is $\mathcal{NP}$-complete and take a look at the standard greedy algorithm as well as some other approximation results.

We begin with the definition of the hitting set problem.

2.1 Definition. Given a set system $(U, \mathcal{S})$ where U is a universe of elements and $\mathcal{S}$ is a collection of subsets of U, the aim of the hitting set problem is to find the smallest subset $A \subseteq U$ such that A hits every set in $\mathcal{S}$, more precisely, for all $S \in \mathcal{S}$ we have $S \cap A \neq \emptyset$.

The weighted version of the hitting set problem includes a function $w\colon U \to \mathbb{R}_+$ that associates a weight with every element in U. The goal of the weighted version is to find a hitting set A that minimises $w(A) = \sum_{x \in A} w(x)$. Another version not only requires that every set is hit, but that a set $S \in \mathcal{S}$ is hit at least $f(S)$ many times where $f\colon \mathcal{S} \to \mathbb{N}$. Note that for $f(S) = 0$ we could also remove the element from the set $\mathcal{S}$ or we could extend the function f to $\mathcal{P}(U)$ by setting $f(S') = 0$ for $S' \notin \mathcal{S}$. We call this the *extended hitting set problem*.

The decision version of the hitting set problem asks whether there exists a hitting set A of cardinality at most k.

2.2 Theorem. The decision version of the hitting set problem is $\mathcal{NP}$-complete.

Proof. The problem is clearly in $\mathcal{NP}$ as it is easy to check whether a given set A has size at most k and hits every subset in $\mathcal{S}$. To show that it is $\mathcal{NP}$-hard we reduce the vertex cover problem to it. The vertex cover problem asks whether there exists a subset C of k (or less) vertices in a graph G such that every edge in G is incident to a vertex in C. Let $G = (V, E)$ together with k be an instance of vertex cover. We set $U = V$ and choose $\mathcal{S}$ as the set containing all the sets $\{u, v\}$ for $(u, v) \in E$. Then there exists a hitting set A of size at most k in the instance $(U, \mathcal{S})$ if and only if there exists a vertex cover C of size at most k in G. □

Clearly the decision problems of our other two versions are also $\mathcal{NP}$-complete as they are more general than the basic version and so it can be reduced to them.

Another well known problem is the set cover problem defined in Definition 2.3. We will show that it is, in fact, equivalent to the hitting set problem.

2.3 Definition. Given a universe V and a collection $\mathcal{T}$ of subsets of this universe, the set cover problem aims to find a minimum sub-collection $\mathcal{T}'$ of $\mathcal{T}$ such that $\bigcup_{T\in\mathcal{T}'} T = V$.

To show that this problem is equivalent to the hitting set problem we transform an instance of set cover to an instance of hitting set and vice versa. In doing so we ensure that the solutions translate. This would allow us to transform an instance of hitting set to one of set cover, use an approximation algorithm for the new problem, and transform it back, letting us transfer results from one problem to the other.

The transformation can be done as follows: Let $(V, \mathcal{T})$ be an instance of set cover. Define the instance $(U, \mathcal{S})$ of hitting set by choosing $U := \{t_i \mid T_i \in \mathcal{T}\}$ and $\mathcal{S} := \{V_i \mid v_i \in V\}$ where $V_i = \{t_j \mid v_i \in T_j\}$. Then $\mathcal{T}' \subseteq \mathcal{T}$ solves the instance of set cover if and only if $A = \{t_i \mid T_i \in \mathcal{T}'\} \subseteq U$ solves the instance of hitting set. This is true as

$$\begin{aligned}\mathcal{T}' \text{ solves set cover } &\Leftrightarrow \forall v_i \in V \ \exists T_j \in \mathcal{T}' \colon v_i \in T_j \\ &\Leftrightarrow \forall V_i \in \mathcal{S} \ \exists t_j \in A \colon t_j \in V_i \\ &\Leftrightarrow A \text{ solves hitting set.}\end{aligned}$$

The reverse transformation works in much the same way: Let $(U, \mathcal{S})$ be an instance of the hitting set problem. Define the instance $(V, \mathcal{T})$ of set cover by choosing $V := \{s_i \mid S_i \in \mathcal{S}\}$ and $\mathcal{T} := \{U_i \mid u_i \in U\}$ where $U_i = \{s_j \mid u_i \in S_j\}$. Again we get $A \subseteq U$ solves the instance of hitting set if and only if $\mathcal{T}' = \{U_i \mid u_i \in A\} \subseteq \mathcal{T}$ solves the instance of set cover.

Applying the above transformation to the greedy algorithm for set cover, which picks the set that covers the most new elements of V in every iteration, it translates to the greedy algorithm for the hitting set problem shown in Algorithm 2.1. This one picks the element in U that hits the most new sets in every iteration.

Algorithm 2.1: Greedy Algorithm for the Hitting Set Problem

Input: The hitting set instance $(U, \mathcal{S})$.
Output: A feasible solution A.
procedure:

```
1  begin
2      A := ∅;
3      S' := S;
4      while S' ≠ ∅ do
5          Find the element u ∈ U that is contained in the most sets of S';
6          A := A ∪ {u};
7          S' := S'\{S ∈ S' | u ∈ S};
8      end
9      return A;
10 end
```

The greedy algorithm has an approximation guarantee of $\log n + 1$ where $n = |\mathcal{S}|$ as shown by [APMS+99]. Also the set cover and hence the hitting set problem has an inapproximability bound of $(1 - o(1)) \log n$ unless $\mathcal{NP} \subseteq \text{TIME}(n^{\mathcal{O}(\log\log n)})$ where n denotes the number of elements in the universe V or the amount of sets to be hit, $|\mathcal{S}|$, respectively. This was proved

in [Fei98]. Another result of this type can be found in [RS97] stating that the set cover problem is not approximable within $c \log n$ for some $c > 0$ unless $\mathcal{P} = \mathcal{NP}$.

2.2. The Successive Hitting Set Model

Here we state a different model for this problem, called the successive hitting set model, and show how the greedy algorithm can be extended to it.

Notice that the greedy algorithm is less simple than it sounds as the number of sets contained in $\mathcal{S}$ can be very large, up to $2^{|U|}$. This means that it can be very costly to store all these sets and in the early iterations of the algorithm most of them need to be regarded to find the element contained in most. So the question arises whether we can solve the problem without having all the information available from the start. In this case we will not have access to all elements in $\mathcal{S}$ at every point in time. The successive hitting set model does this by giving us access to all the sets that contain a certain element and is defined below.

2.4 Definition. In the successive hitting set model we are given an instance $(U, \mathcal{S})$ of the hitting set problem and a generator $G\colon U \to \mathcal{S}$ where $G(u)$ reveals the collection of all sets in $\mathcal{S}$ that contain the element $u \in U$.

Clearly this model does not change the nature of the hitting set problem as we could just call the generator for every element in U and reconstruct $\mathcal{S}$. It does, however, give us the opportunity to be content with only knowing a sub-collection of $\mathcal{S}$ and saving storage space and computation time. We now extend the greedy algorithm with the help of this model to make it more efficient and check how this affects the approximation guarantee.

The new greedy algorithm is displayed in Algorithm 2.2. It ensures that the generator is only called once for every element of U. This means that a set cannot be dropped once it is generated unless it has been hit, giving us an easy way to determine correctness: We have hit every set once the generator has been called for every element and the set $\mathcal{S}'$ is empty.

To analyse the approximation guarantee of this algorithm let c denote the size of the optimal solution A^* and assume that the sets in $\mathcal{S}$ are closed under intersection, meaning there exists no partition of $\mathcal{S}$ into $\mathcal{S}_X$ and $\mathcal{S}_Y$ such that there is no element in U that is contained in both a set of $\mathcal{S}_X$ and $\mathcal{S}_Y$. Should this be the case, we can divide the problem into two subproblems and solve the smaller problems individually.

2.5 Theorem. The successive greedy algorithm finds a hitting set A with $|A| \leq c(\log m + 2)$ where $m = |\mathcal{S}|$.

Proof. First we assign a cost to every set as follows. Assume we order the sets in the optimal solution A^* in some way and write $\mathcal{S}(u)$ for the collection of sets first hit when the element u is added to A^*. In this case we assign the cost $s_u = 1/|\mathcal{S}(u)|$ to every set $S \in \mathcal{S}(u)$. We do the same thing for the greedy algorithm and write $c(S)$ for the cost of a set S in this case. To be able to distinguish between the collections of sets first hit by an element u in the optimal solution and by the greedy algorithm we denote the latter by $\mathcal{S}_g(u)$. This gives us that

$$|A| = \sum_{u \in A} 1 = \sum_{u \in A} \sum_{S \in \mathcal{S}_g(u)} c(S).$$

Algorithm 2.2: Successive Greedy Algorithm for the Hitting Set Problem

Input: The hitting set instance $(U, \mathcal{S})$.
Output: A feasible solution A.
procedure:

```
1  begin
2      A := ∅;
3      Let S' contain all sets generated by G(u) for some u ∈ U;
4      C := {u};
5      while S' ≠ ∅ do
6          Find the element u' ∈ U that is contained in the most sets of S';
7          A := A ∪ {u'};
8          X := {S ∈ S' | u' ∈ S};
9          S' := S'\X;
10         for S ∈ X do
11             for u ∈ S do
12                 if u ∉ C then
13                     C := C ∪ {u};
14                     Call generator G(u);
15                     if generated set is hit by A then
16                         Add the set to S';
17                     end
18                 end
19             end
20         end
21         if S' = ∅ and C ≠ U then
22             Add all sets generated by G(u) for some u ∈ U\C to S';
23         end
24     end
25     return A;
26 end
```

By showing that for every $u \in A^*$ the inequality

$$\sum_{S \in \mathcal{S}(u)} c(S) \leq H_{s_u - 1} + 1 \leq \log(s_u - 1) + 2 \leq \log m + 2 \tag{2.1}$$

holds we get that

$$|A| = \sum_{u \in A} \sum_{S \in \mathcal{S}_g(u)} c(S) = \sum_{u \in A^*} \sum_{S \in \mathcal{S}(u)} c(S) \overset{(2.1)}{\leq} \sum_{u \in A^*} \log m + 2 = (\log m + 2)|A^*|$$

and the approximation guarantee that was claimed. Note that the second equality is just a reordering of the terms.

Hence we only need to prove the inequalities in (2.1). The last two follow from the fact that $H_n < \log n + 1$ and $s_u \leq m$ leaving the first inequality. So let $u \in A^*$ and u_0 be the first element of A that hits a set in $\mathcal{S}(u)$. As the sum of the costs of all sets in $\mathcal{S}_g(u_0)$ is equal to

one, the sum of those that are also contained in $\mathcal{S}(u)$ is at most one. After adding u_0 to A all sets containing the element u have been generated. Consequently all sets in $\mathcal{S}(u)$ are either available or have already been hit (and for at least one set the latter is the case). From this point forward we could choose u as the next element in the greedy algorithm, hitting all the remaining sets of $\mathcal{S}(u)$ and adding costs equal to at most one, which satisfies our inequality as long as they do not exceed costs of H_{s_u-1} at any point. However, if u is not chosen, the algorithm finds an element that hits more sets than u. Let u_1 be the next element chosen that hits a set in what is left of $\mathcal{S}(u)$. If it is not u, then any set it hits has a cost of at most $1/s_u-k$ where $k \geq 1$ is the amount of sets already hit. We can apply this argument iteratively to the worst case, in which u is never picked and in every iteration exactly one new set is hit. This is, in fact, the worst case as sets covered later have higher cost. This gives us that

$$\sum_{S \in \mathcal{S}(u)} c(S) \leq 1 + \sum_{i=1}^{s_u-1} \frac{1}{s_u - i} = 1 + \sum_{i=1}^{s_u-1} \frac{1}{i} = H_{s_u-1} + 1,$$

completing the proof. □

2.3. Path Covers

We now introduce a couple of specific hitting set problems called *path covers* that we are going to look at later. As a motivation we primarily use the example from the introduction, but also give two additional applications afterwards.

Recall that in the introduction we wanted to enable an owner of an electric car to be able to travel to any location in a given region. To do so, battery loading stations need to be installed at regular intervals since electric vehicles have a limited cruising range. As installing such loading stations is expensive, we want to minimise the amount required. One could also consider the case that not all locations are equal since it might be more difficult to install a loading station in certain areas. In this case we want to minimise the total installation cost. Of course the first problem is just a special case of this one if uniform weights are used.

If we model the region in question as a graph and make the simplified assumption that the nodes are evenly spaced, then, if $k-1$ is approximately the cruising range divided by the distance between two nodes, the problem described above corresponds to the k-AVAPC problem that we define now.

2.6 Definition. Given a graph $G = (V, E)$, $k \in \mathbb{N}$, and a set $\mathcal{V} \subseteq V \times V$, let $\mathcal{S}$ be the sets of nodes on simple subpaths $P = (v_1, \ldots, v_k)$ of s-t paths in G where $(s, t) \in \mathcal{V}$. The aim of the k-$\mathcal{V}$-all-path cover problem (k-$\mathcal{V}$-APC) is to select a minimum subset $C \subseteq V$ such that we have $C \cap S \neq \emptyset$ for all $S \in \mathcal{S}$. If $\mathcal{V} = V \times V$, we call this the k-all-vertex-all-path cover problem (k-AVAPC) and if $\mathcal{V} = \{(s, t)\}$, we call it the k-two-vertex-all-path cover problem (k-TVAPC).

Note that these problems are clearly hitting set problems if we choose the universe to be the set of vertices and $\mathcal{S}$ as in the definition.

Let us quickly check that a solution to the k-AVAPC problem, in fact, solves our problem of finding locations for battery loading stations: If we are given a solution, then it selects

a node on every path that contains k nodes. By our assumption such a path has length approximately equal to the cruising range of an electric vehicle. As a result any feasible solution to the k-AVAPC problem installs a loading station on every path of length greater than or equal to the cruising range. The converse also holds true and consequently the optimal cover is exactly what we were looking for.

Next we modify the above problem a bit. One of our previous solutions might be more expensive than expected, undoubtedly it will be more costly than desired, but perhaps it actually is more expensive than necessary. This could be the case as we wanted to ensure that a driver can take any route to get from his location to a given destination, no matter how out of the way this route might be. If we reduce our ambitions slightly and only require that a driver should be able to get from one point to another efficiently, instead of in whatever way he pleases, we arrive at the definition of k-shortest-path covers.

2.7 Definition. Given a graph $G = (V, E)$, $k \in \mathbb{N}$, and a set $\mathcal{V} \subseteq V \times V$, let $c\colon E \to \mathbb{R}_+$ be costs on the edges of G and $\mathcal{S}$ be the sets of nodes on subpaths $P = (v_1, \ldots, v_k)$ of the shortest s-t path in G where $(s, t) \in \mathcal{V}$. The aim of the k-$\mathcal{V}$-shortest-path cover problem (k-$\mathcal{V}$-SPC) is to select a minimum subset $C \subseteq V$ such that we have $C \cap S \neq \emptyset$ for all $S \in \mathcal{S}$. If $\mathcal{V} = V \times V$, we call this the k-all-vertex-shortest-path cover problem (k-AVSPC) and if $\mathcal{V} = \{(s, t)\}$, we call it the k-two-vertex-shortest-path cover problem (k-TVSPC).

This again corresponds to the hitting set problem if we choose the universe to be the set of vertices and $\mathcal{S}$ as in the definition. We add that, whenever we regard this problem later on, we will assume that the shortest paths are uniquely determined, justifying us calling it "the shortest path" in the definition.

Again we quickly check that this new definition has the desired effect. As we only require shortest paths to be hit, it will most likely not be necessary to hit all paths, making our solution cheaper. Additionally, the driver can still efficiently get to his destination as the shortest path is still a possible route. However, it is clear that it will (probably) no longer be possible to take any route, so the most scenic route might not be an option any more.

As promised we get to two more applications now. For the first assume that we have some street network at our disposal as well as knowledge of certain points of interest and their locations. This can be anything from shopping malls or gas stations to actual tourist attractions or amusement parks. We would like to create a feature that can, given a route, return all points of interest along it. Let us say we are planning a weekend trip, perhaps visiting some relatives, and have decided on how to get there and now want to find out whether there are any other activities we could pursue without needing to take significant detours. Another possibility is that, in analogy to the above, we might just want to know where gas stations along the route are. Of course we could solve this by just creating a data structure $\mathcal{D}$ that, given a point v, returns all points of interest close to it. We could then query $\mathcal{D}$ for all nodes on the route and return the result. This has two obvious disadvantages. The first being that longer paths require a lot of queries, making them expensive very quickly. This is easy to remedy by just checking every k-th node. The other disadvantage is that $\mathcal{D}$ needs to be defined for every node in the graph, possibly (or probably) making it a complicated data structure and resulting in long computation times. Only running queries on every k-th node does not directly help here. However, if we compute a path cover C, more precisely a k-AVAPC, we can just compute $\mathcal{D}$ for the nodes in C and can still return the points of interest of at least

every k-th node on any given path. By adding weights we could make it cheaper to omit nodes with few or no points of interest, but penalise not including nodes with many.

For the second application we take a look at a personalised route planner that in addition to the current location and destination also takes information concerning the driving style or personal preferences into account. This can include the driver's typical driving speed or whether lower travel time or reduced gas usage is preferred. These would then affect the objective function used to determine the optimal route. Clearly, this makes the computation very complicated as we need to compute weights for all edges before even beginning to look for a solution. Again we want to make use of a path cover and as before a k-AVAPC C will be helpful. Here we would compute the optimal route on the smaller graph induced by the nodes of C instead of on the actual graph. As we consequently need to compute the weights on fewer edges, this could speed up the computation significantly.

3. The Primal-Dual Method

In this chapter we first want to describe the classical primal-dual method and show how it can be applied to approximate the $\mathcal{NP}$-hard hitting set problem as shown in the second and third sections of chapter four of [GW97].

3.1. The Classical Primal-Dual Method

The classical primal-dual method is used to solve linear programs. To show how it works we will need a few results from linear programming. First consider the linear program

$$\begin{aligned} \min \quad & c^T x \\ & A\,x \geq b, \\ & x \geq 0 \end{aligned}$$

and its dual

$$\begin{aligned} \max \quad & b^T y \\ & A^T\, y \leq c, \\ & y \geq 0. \end{aligned}$$

Here we have $A \in \mathbb{Q}^{m \times n}$, $c, x \in \mathbb{Q}^n$ and $b, y \in \mathbb{Q}^m$. We write A_i for the i-th row and A^j for the j-th column of A. We recall the complementary slackness conditions for a linear program and its dual. On the one hand they state that

$$x_j > 0 \Rightarrow A^j y = c_j,$$

which we call the *primal complementary slackness conditions*. On the other hand we get the *dual complementary slackness conditions*

$$y_j > 0 \Rightarrow A_i x = b_i.$$

In the following we assume that $c \geq 0$, in which case the solution $y = 0$ is feasible for the dual. Let $J \coloneqq \{j \mid A^j y = c_j\}$ and $I = \{i \mid y_i = 0\}$ for feasible x and y. Then by the definition of these two sets we have $x_j > 0$ implies that $j \in J$ and $A_i x \neq b_i$ requires that $i \in I$.

Now we describe the primal-dual method. It starts with a feasible solution for the dual, so we can start with $y = 0$, and iteratively computes either a solution x for the primal which satisfies the complementary slackness conditions or a new dual solution y' with a greater objective value. In the first case we have found an optimal solution to the primal, whereas the second case gives us a better lower bound. If we have a feasible dual solution y, we can rephrase the problem of finding a feasible primal solution x such that complementary slackness

conditions hold as an optimisation problem. This can be done, for example, by the problem below, called the *restricted primal*:

$$\begin{aligned} z_{\text{INF}} = \min \sum_{i \notin I} s_i + \sum_{j \notin J} x_j & \\ A_i x \geq b_i, & \quad i \in I, \\ A_i x - s_i = b_i, & \quad i \notin I, \\ x, s \geq 0. & \end{aligned}$$

As $s \geq 0$, the first two constraints ensure that $Ax \geq b$ and consequently x is feasible for the primal. If $z_{\text{INF}} = 0$ for the solution (x, s), we get that x is a feasible solution for the primal that fulfils $A_i x = b_i$ for all $i \notin I$ and $x_j = 0$ for all $j \notin J$. As a result x and y fulfil both complementary slackness conditions and are as such optimal. However, if we have $z_{\text{INF}} > 0$, we need to construct a better solution to the dual. To do this we regard the dual of the restricted primal:

$$\begin{aligned} z_{\text{INF}} = \max \ b^T y & \\ A^j y \leq 0, & \quad j \in J, \\ A^j y \leq 1, & \quad j \notin J, \\ y_i \geq -1, & \quad i \notin I, \\ y_i \geq 0, & \quad i \in I. \end{aligned}$$

As the primal had an optimal solution of positive value, so does the above dual. This means that we can find a feasible solution y'' with $b^T y'' > 0$. We claim that we can find an $\varepsilon > 0$ such that $y' := y + \varepsilon y''$ is a feasible dual solution. Its value is an improvement of our previous solution as $b^T y' = b^T y + \varepsilon b^T y'' > b^T y$. But how do we find ε? Observe that $y''_i \geq 0$ for $i \in I$, so only the entries y'_i for $i \notin I$ can possibly violate the constraint $y'_i \geq 0$. By choosing $\varepsilon \leq \min\{-y_i/y''_i \mid i \notin I, y''_i < 0\} > 0$ we can ensure $y' \geq 0$. Also as $A^j y'' \leq 0$ for $j \in J$ we get that $A^j y' \leq c_j$ for these indices. Again by choosing $\varepsilon \leq \min\{c_j - A^j y / A^j y' \mid j \notin J, A^j y'' > 0\} > 0$ we can ensure that $A^T y' \leq c$ and we have found an ε as required. So in every step of the primal-dual method we either find an optimal solution or increase the value of our dual.

One should note that the method solves a linear program by calculating the optimal solution to another linear program in every step. This cannot be helpful unless the subproblems are easier to solve than the original one. However, the new dual no longer contains the vector c and can often be solved combinatorially.

The above restricted primal required primal feasibility and measured the violation of the complementary slackness conditions. Another option is to set up a restricted primal that requires the complementary slackness conditions and measures the degree to which x violates feasibility. This can be achieved, for example, by the following linear program

$$\begin{aligned} z_{\text{INF2}} = \min e^T s + e_{\bar{I}}^T t & \\ A_i^J x_J + s_i \geq b_i, & \quad i \in I, \\ A_i^J x_J + s_i - t_i = b_i, & \quad i \notin I, \\ x_J \geq 0, & \\ s, t \geq 0 & \end{aligned}$$

where $\bar{I}$ denotes the set of indices not contained in I. Here we have set $x_j = 0$ for all $j \notin J$, enforcing primal complementary slackness. In this case the dual has the form

$$\begin{aligned} z_{\text{INF2}} = \max \quad & b^T y \\ & A_J^T y \leq 0, \\ & y \leq 1, \\ & y_i \geq -1, \quad i \notin I, \\ & y_i \geq 0, \quad i \in I. \end{aligned}$$

3.2. The Primal-Dual Method for the Hitting Set Problem

To begin we formulate the hitting set problem from Definition 2.1 as an integer linear program and proceed to adapting the method presented in the previous section to achieve an approximation algorithm for this problem.

In the following we usually apply the hitting set problem to a graph $G = (V, E)$ where the universe is E and the collection of sets to be hit is a set of cuts $T_i = \delta(S_i)$, so we have $U = E$ and $\mathcal{S} = \{T_1, \ldots, T_p\}$. The hitting set problem can now be formulated as an integer program:

$$\begin{aligned} \min \quad & \sum_{e \in E} c_e x_e \\ & \sum_{e \in T_i} x_e \geq 1, \quad i = 1, \ldots, p, \\ & x_e \in \mathbb{B}, \quad e \in E. \end{aligned}$$

Here the vector x represents the incidence vector for the selected set A. Relaxing and dualising leads to the primal

$$\begin{aligned} \min \quad & \sum_{e \in E} c_e x_e \\ & \sum_{e \in T_i} x_e \geq 1, \quad i = 1, \ldots, p, \\ & x_e \geq 0, \quad e \in E \end{aligned}$$

and the dual

$$\begin{aligned} \max \quad & \sum_{i=1}^{p} y_i \\ & \sum_{i : e \in T_i} y_i \leq c_e, \quad e \in E, \\ & y_i \geq 0, \quad i = 1, \ldots, p. \end{aligned}$$

If x is an incidence vector of a set A and y is a feasible solution to the dual, the primal complementary slackness conditions are

$$e \in A \Rightarrow \sum_{i : e \in T_i} y_i = c_e, \tag{3.1}$$

while the dual conditions read

$$y_i > 0 \Rightarrow |A \cap T_i| = 1.$$

Next we modify the primal-dual method to enforce the primal and relax the dual complementary slackness conditions. If we regard the set $A \coloneqq \{e \mid \sum_{i:e\in T_i} y_i = c_e\}$ for a feasible solution y, then there is no feasible set that satisfies (3.1) if A is infeasible. As in the classical method we want to increase our dual solution. To do so we notice that by the infeasibility of A we can find a set T_k such that $A \cap T_k = \emptyset$, which we call *violated*. By increasing the dual variable y_k the dual solution improves (as all the y_i have a coefficient of one) and the maximum value y_k can take without losing feasibility is

$$y_k = \min \left\{ c_e - \sum_{i\neq k:e\in T_i} y_i \;\middle|\; e \in T_k \right\} > 0. \tag{3.2}$$

Note that y_k is, in fact, positive as none of the elements in T_k are contained in A. After increasing y_k by this amount at least one new element of E, namely the argmin in (3.2), will satisfy $\sum_{i:e\in T_i} y_i = c_e$ and can be added to A. We can repeat this procedure until A is feasible. This is the first version of the primal-dual algorithm and is displayed in Algorithm 3.1.

Algorithm 3.1: Primal-Dual Algorithm, First Version

Input: The hitting set instance $(E, \{T_1, \ldots, T_p\})$.
Output: A feasible solution A (and a feasible dual solution y).
procedure:

```
1 begin
2     y ≔ 0;
3     A ≔ ∅;
4     while ∃k : A ∩ T_k = ∅ do
5         Increase y_k until there exists an e ∈ T_k : Σ_{i:e∈T_i} y_i = c_e;
6         A ≔ A ∪ {e};
7     end
8     return A (and y);
9 end
```

Clearly the algorithm returns a feasible solution, but what can we say about its quality?

3.1 Theorem. The first version of the primal-dual method shown in Algorithm 3.1 returns a feasible solution of cost at most $\alpha \sum_{i=1}^{p} y_i \leq \alpha z_{OPT}$ if α satisfies that

$$|A \cap T_i| \leq \alpha \quad \forall i : y_i > 0. \tag{3.3}$$

Proof. The cost of the solution returned by the algorithm is $c(A) = \sum_{e\in A} c_e$. Any element $e \in A$ is only added if $\sum_{i:e\in T_i} y_i = c_e$, allowing us to rewrite this as $c(A) = \sum_{e\in A} \sum_{i:e\in T_i} y_i$. As y_i appears in this sum once for every $e \in A$ that is contained in T_i, we get that

$$c(A) = \sum_{i=1}^{p} |A \cap T_i| y_i.$$

Finally, by the feasibility of y for the dual, we get that the value $\sum_{i=1}^{p} y_i$ is a lower bound for the optimum value of the primal. Consequently Algorithm 3.1 is an α-approximation as claimed. □

Clearly $\alpha = \max\{|T_i| \mid i = 1, \dots, p\}$ satisfies the condition (3.3). Applying this to the vertex cover problem, where the universe is the set of nodes and the T_i are the endpoints of the edges, we get a maximum cardinality of two and as a result a 2-approximation algorithm.

This first version of the algorithm leaves a few open ends when it comes to selecting the violated set. First of all there could be an exponential amount of sets that need to be hit, meaning we need an efficient way of finding a violated one instead of just iterating over all of them. For this, however, we assume that we have a so called violation oracle which returns the information that the solution is feasible or a violated set if one should exist. Also, generally, there will be multiple violated sets that can be chosen. In this case a good selection rule is the *minimal violated set rule* which picks an inclusion-wise minimal violated set.

One problem of the above method can be seen by regarding the shortest-path problem. This can also be formulated as a hitting set problem by choosing the set of edges as the universe and the set of all (s,t)-cuts as the sets to be hit. If we want to apply Algorithm 3.1 with the minimal violated set rule to this problem, we can actually efficiently compute the set our violation oracle would return by choosing the violated set $\delta(S)$ where S is the connected component containing s. As in many cases this will not be so simple, we just assume that we are given such an oracle. The resulting algorithm returns a solution similar to that of Dijkstra's algorithm. The difference is that the solution does not solely consist of a shortest (s,t)-path, but instead it is a shortest path forest out of s. So we have added edges that are unnecessary for our final solution. This motivates the adding of a delete step in which superfluous elements are again discarded, called the *reverse delete step*, as the elements to be dropped are considered in the opposite order in which they are added. The algorithm that makes use of this can be seen in Algorithm 3.2.

Algorithm 3.2: Primal-Dual Algorithm, Second Version

Input: The hitting set instance $(E, \{T_1, \dots, T_p\})$.
Output: A feasible solution A (and a feasible dual solution y).
procedure:

```
1  begin
2      y := 0;
3      A := ∅;
4      l := 0;
5      while ∃k : A ∩ T_k = ∅ do
6          l := l + 1;
7          Increase y_k until there exists an e_l ∈ T_k : Σ_{i:e_l∈T_i} y_i = c_{e_l};
8          A := A ∪ {e_l};
9      end
10     for j := l downto 1 do
11         if A\{e_j} is feasible then
12             A := A\{e_j};
13         end
14     end
15     return A (and y);
16 end
```

Next we take a look at the performance guarantee of this algorithm.

3.2 Theorem. The second version of the primal-dual method shown in Algorithm 3.2 returns a feasible solution of cost at most $\beta \sum_{i=1}^{p} y_i \leq \beta z_{OPT}$. Here β is given by

$$\beta = \max_{\substack{A \subseteq E \\ \text{infeasible}}} \max_{\substack{B \text{ minimal aug-} \\ \text{mentation of } A}} |B \cap T(A)| \tag{3.4}$$

where $T(A)$ denotes the violated set selected by the algorithm when confronted with A and a minimal augmentation B of A is a superset of A that is feasible such that no element in $B \backslash A$ can be removed without losing feasibility.

Proof. Let A_f be the set returned by Algorithm 3.2 and choose an i such that $y_i > 0$. Furthermore, let e_j be the edge added when y_i was increased. Because of the reverse delete step, none of the edges e_k for $k < j$ have been removed yet. Let B be the set of elements in A right after e_j was considered for removal, then $B = A_f \cup \{e_1, \ldots, e_{j-1}\}$. This means that B is a minimal augmentation of $A = \{e_1, \ldots, e_{j-1}\}$ as it is a superset, feasible and $B \backslash \{e\}$ is not feasible for any $e \in B \backslash A$. Clearly, because B is a superset of A_f, also $|A_f \cap T_i| \leq |B \cap T_i|$ holds. Combining these results we get that

$$|A_f \cap T_i| \leq |B \cap T_i| \leq \max_{\substack{B' \text{ minimal aug-} \\ \text{mentation of A}}} |B' \cap T_i| \leq \beta$$

as A is infeasible (else the algorithm would not have regarded e_j) and $T_i = T(A)$. □

Notice that the proof actually shows a possibly stronger bound than β. It technically suffices to regard infeasible sets A that the algorithm can be faced with for some instance and it is enough to check minimal augmentations B that the algorithm can create. This is of use if we know that any solution constructed by the algorithm has certain properties and these translate to the augmentations.

The third version only varies slightly from the previous one and shares its performance guarantee under certain circumstances. We now wish to add sets to the collection of those that need to be hit. More precisely we want to add sets $T_{p+1}, \ldots, T_q$ that the algorithm will need to hit when constructing the feasible solution A, but which it will not need to hit any more when it comes to the reverse delete step. This version is described in Algorithm 3.3.

3.3 Theorem. The third version of the primal-dual method shown in Algorithm 3.3 returns a feasible solution A_f of cost at most $\beta \sum_{i=1}^{p} y_i \leq \beta z_{OPT}$ with β as in equation (3.4) if we can guarantee that $A_f \cap T_i = \emptyset$ for $i = p + 1, \ldots, q$.

Proof. The algorithm constructs a dual solution y. Let y^p be the first p components of this solution, then y^p is feasible for the original dual. This is the case as y satisfies

$$\sum_{i \in \{1,\ldots,q\}: e \in T_i} y_i \leq c_e \text{ for all } e \in E \text{ and } y \geq 0$$

which implies

$$\sum_{i \in \{1,\ldots,p\}: e \in T_i} y_i \leq c_e \text{ and } y^p \geq 0.$$

Algorithm 3.3: Primal-Dual Algorithm, Third Version

Input: The hitting set instance $(E, \{T_1, \ldots, T_p\})$ and additional sets $T_{p+1}, \ldots, T_q$.
Output: A feasible solution A (and a feasible dual solution y).
procedure:

```
1  begin
2      y := 0;
3      A := ∅;
4      l := 0;
5      while ∃k ∈ {1,...,q} : A ∩ T_k = ∅ do
6          l := l + 1;
7          Increase y_k until there exists an e_l ∈ T_k : Σ_{i:e_l∈T_i} y_i = c_{e_l};
8          A := A ∪ {e_l};
9      end
10     for j := l downto 1 do
11         if A\{e_j} is feasible then
12             A := A\{e_j};
13         end
14     end
15     return A (and y);
16 end
```

Therefore, $\sum_{i=1}^{p} y_i$ is a lower bound for the optimum value and it can be compared to the cost of the solution, which is $\sum_{i=1}^{q} |A_f \cap T_i| y_i$. By assumption $A_f \cap T_i = \emptyset$ for $i = p+1, \ldots, q$, so this simplifies to $\sum_{i=1}^{p} |A_f \cap T_i| y_i$. Again we need to show that $|A_f \cap T_i| \leq \beta$ for all $i \leq p$ where $y_i > 0$. With the same notation as in the proof of Theorem 3.2 let B be a minimal augmentation of A such that the sets $T_1, \ldots, T_p$ are hit. By the increase of e_j, A was infeasible and $T(A) = T_i$ is one of the sets in $T_1, \ldots, T_p$. So, as before, $|A \cap T_i| \leq |B \cap T_i| \leq \beta$. □

The final design rule comes from regarding the minimum spanning tree problem and Kruskal's algorithm. The universe here would again be the set of edges and the sets to be hit would be all possible cuts. However, a minimal augmentation of an infeasible set A induces a spanning tree on the component graph of (V, A) (the graph where we contracted all connected components) and could possibly have $k - 1$ edges in $A \cap T_i$ if there are k connected components. Unlike Kruskal's algorithm, this is far from optimal. The design rule that results from this observation is the following. Instead of increasing the dual variable of a violated set, we increase the dual variables of multiple such sets at the same speed. This is called the *uniform increase rule*. In the case of the minimum spanning tree problem increasing the dual variables corresponding to all minimal violated sets, which are the connected components of the graph (V, A), leads to the algorithm due to Kruskal. This suggests using all minimum violated sets. This algorithm is displayed as Algorithm 3.4 where VIOLATION is an oracle that returns a collection of violated sets. It could for example return all minimal violated sets as used previously and we will again assume that we are given such an oracle.

The proof of the approximation guarantee is similar to the proof of Theorem 3.2.

Algorithm 3.4: Primal-Dual Algorithm, Fourth Version

Input: The hitting set instance $(E, \{T_1, \ldots, T_p\})$.
Output: A feasible solution A (and a feasible dual solution y).
procedure:

```
1  begin
2      y := 0;
3      A := ∅;
4      l := 0;
5      while A is not feasible do
6          l := l + 1;
7          V := VIOLATION(A);
8          Increase y_k uniformly for all T_k ∈ V until there exists an e_l ∉ A : Σ_{i:e_l∈T_i} y_i = c_{e_l};
9          A := A ∪ {e_l};
10     end
11     for j := l downto 1 do
12         if A\{e_j} is feasible then
13             A := A\{e_j};
14         end
15     end
16     return A (and y);
17 end
```

3.4 Theorem. The fourth version of the primal-dual method shown in Algorithm 3.4 returns a feasible solution of cost at most $\gamma \sum_{i=1}^{p} y_i \leq \gamma z_{OPT}$ if γ satisfies that for any infeasible set A and any minimal augmentation B of A

$$\sum_{T_i \in \mathcal{V}(A)} |B \cap T_i| \leq \gamma |\mathcal{V}(A)| \tag{3.5}$$

where $\mathcal{V}(A)$ denotes the collection of violated sets returned by VIOLATION for the input A.

Proof. Again the aim is to compare the value $\sum_{i=1}^{p} y_i$ of the dual to that of the output solution A_f. In this case we can make use of the fact that the algorithm increases the dual variables for all sets returned by the oracle uniformly to rewrite the value $\sum_{i=1}^{p} |A_f \cap T_i| y_i$ of the solution. To this end let $\mathcal{V}_j$ denote the collection of violated sets returned by VIOLATION in iteration j and ε_j be the increase of the corresponding dual variables. As a result we get that $y_i = \sum_{j:T_i \in \mathcal{V}_j} \varepsilon_j$. This allows us to rewrite the value of the dual solution as

$$\sum_{i=1}^{p} y_i = \sum_{j=1}^{l} |\mathcal{V}_j| \varepsilon_j$$

and the cost of A_f as

$$\sum_{i=1}^{p} |A_f \cap T_i| y_i = \sum_{i=1}^{p} |A_f \cap T_i| \sum_{j:T_i \in \mathcal{V}_j} \varepsilon_j = \sum_{j=1}^{l} \left(\sum_{T_i \in \mathcal{V}_j} |A_f \cap T_i| \right) \varepsilon_j.$$

By comparing both expressions the claim now follows if the below inequality holds:

$$\sum_{T_i \in \mathcal{V}_j} |A_f \cap T_i| \leq \gamma |\mathcal{V}_j|.$$

With the notation from the proof of Theorem 3.2 the set A in iteration j is again infeasible and $B = A_f \cup A$ is a minimal augmentation of A as well as $|A_f \cap T_i| \leq |B \cap T_i|$. This leads to

$$\sum_{T_i \in \mathcal{V}_j} |A_f \cap T_i| \leq \sum_{T_i \in \mathcal{V}_j} |B \cap T_i| \leq \max_{\substack{B' \text{ minimal augmentation of } A}} \sum_{T_i \in \mathcal{V}_j} |B' \cap T_i| \leq \gamma |\mathcal{V}_j|$$

as required. □

As before it is possible for the oracle VIOLATION to return sets which do not need to be hit without affecting the performance guarantee. The algorithm is shown as Algorithm 3.5.

Algorithm 3.5: Primal-Dual Algorithm, Fifth Version

Input: The hitting set instance $(E, \{T_1, \ldots, T_p\})$ and additional sets $T_{p+1}, \ldots, T_q$.
Output: A feasible solution A (and a feasible dual solution y).
procedure:

```
1  begin
2      y := 0;
3      A := ∅;
4      l := 0;
5      while A ∩ T_k = ∅ for some k ∈ {1, ..., q} do
6          l := l + 1;
7          V := VIOLATION(A);
8          Increase y_k uniformly for all T_k ∈ V until there exists an e_l ∉ A : Σ_{i:e_l∈T_i} y_i = c_{e_l};
9          A := A ∪ {e_l};
10     end
11     for j := l downto 1 do
12         if A\{e_j} is feasible then
13             A := A\{e_j};
14         end
15     end
16     return A (and y);
17 end
```

3.5 Theorem. The fifth version of the primal-dual method shown in Algorithm 3.5 returns a feasible solution A_f of cost at most $\gamma \sum_{i=1}^{p} y_i \leq \gamma z_{OPT}$ if γ satisfies that for any infeasible set A and any minimal augmentation B of A

$$\sum_{T_i \in \mathcal{V}(A)} |B \cap T_i| \leq \gamma c \tag{3.6}$$

where $\mathcal{V}(A)$ denotes the collection of violated sets returned by VIOLATION for the input A and c denotes the number of sets in $\mathcal{V}(A)$ which need to be hit.

Proof. As in the proof of Theorem 3.4 we have $\sum_{i=1}^{p} y_i = \sum_{j=1}^{l} |\mathcal{V}_j|\varepsilon_j$. Consequently we get

$$\sum_{i=1}^{q} |A_f \cap T_i| y_i = \sum_{j=1}^{l} \left(\sum_{T_i \in \mathcal{V}_j} |A_f \cap T_i| \right) \varepsilon_j.$$

This, however, results in

$$\sum_{i=1}^{p} y_i = \sum_{j=1}^{l} c_j \varepsilon_j$$

where c_j denotes the number of sets in $\mathcal{V}_j$ that need to be hit. Comparing the expressions shows that it suffices to prove that

$$\sum_{T_i \in \mathcal{V}_j} |A_f \cap T_i| \leq \gamma c_j \text{ for all } j.$$

So again we look at iteration j. With the notation as in Theorem 3.2 we get that the set A is infeasible, $B = A \cup A_f$ is a minimal augmentation of A and $|A_f \cap T_i| \leq |B \cap T_i|$. As before we get

$$\sum_{T_i \in \mathcal{V}_j} |A_f \cap T_i| \leq \sum_{T_i \in \mathcal{V}_j} |B \cap T_i| \leq \max_{\substack{B' \text{ minimal augmentation of } A}} \sum_{T_i \in \mathcal{V}_j} |B' \cap T_i| \leq \gamma c_j$$

as required. □

4. The Primal-Dual Method for Path Covers

In this chapter we want to apply our algorithms from Chapter 3, in particular the Algorithms 3.2 and 3.4, to the path cover problems from Definitions 2.6 and 2.7 and determine what kind of approximation guarantees they yield. After checking the results on general graphs, we will look at the cases of paths and trees as well in the hope of getting better results on these classes.

Notice that the special cases discussed in [GW97] require that the function f fulfils the maximality condition, which in the case of $\{0,1\}$-functions implies that the union of disjoint non-violated sets cannot be violated. This is not satisfied in our case as the union of a path with $k-1$ nodes and one with a single node can be violated even though the components were not. So none of these results apply here.

First of all we recall that Algorithm 3.2 has an approximation guarantee of β, which is defined by

$$\beta = \max_{\substack{A \subseteq E \\ \text{infeasible}}} \max_{\substack{B \text{ minimal aug-} \\ \text{mentation of } A}} |B \cap T(A)|. \tag{3.4}$$

As all of our violated sets have cardinality k, we clearly get that for any infeasible set A and minimal augmentation B of A the cardinality of $B \cap T(A)$ is at most k. This gives us that $\beta \leq k$ for all our path cover problems. In the same way we get that Algorithm 3.4 has an approximation guarantee of at most γ, which is given by

$$\sum_{T_i \in \mathcal{V}(A)} |B \cap T_i| \leq \gamma |\mathcal{V}(A)|, \tag{3.5}$$

and $\gamma \leq k$.

We now want to find out whether the value k for the two bounds can be attained for our six problems and three graph classes or if we can get a better bound. We begin with general graphs and after little luck finding better bounds than k we proceed to trees. The only better result we get is for the case of paths, which correspond to the TVSPC problem on general graphs under the assumption that the shortest path is unique. The results for general graphs are summarised in Table 4.1 which shows the best attainable bound for these cases. It turns out that there is almost always an example where the bound k is attained for the value β and where the value γ is arbitrarily close to k. Even with the restriction to trees these values do not improve significantly (to be more precise, the only improvement to be found was in the AVAPC problem in the case $k = 2$, where the bound improves to $\gamma = 3/2$).

To prove that the bound of k can be attained or that $k - \varepsilon$ can be attained for any $\varepsilon > 0$ we construct graphs that fulfil this property. This will always be done for the infeasible set $A = \emptyset$ and a minimal augmentation B. We claim that the lemma below gives a criterion for a minimal augmentation.

Bound	Problem					
	AVAPC	AVSPC	TVAPC	TVSPC	$\mathcal{V}$-APC	$\mathcal{V}$-SPC
β	k	k	k	2	k	k
γ	k	k	k	$2 - 2/(k+1)$	k	k

Table 4.1.: Approximation guarantees of the primal-dual method for path covers.

4.1 Lemma. The set B is a minimal augmentation of A if and only if the following two conditions hold:

(i) $B \cap S \neq \emptyset$ for all $S \in \mathcal{S}$;
(ii) For any $v \in B \backslash A$ there exists a path $P = (v_1, \ldots, v_k)$ with $v \in \{v_1, \ldots, v_k\}$ and $u \notin \{v_1, \ldots, v_k\}$ for all $v \neq u \in B$.

Proof. Let B be a minimal augmentation of A. Then by feasibility of B the condition (i) holds. By minimality we get that for any $v \in B \backslash A$, $B \backslash \{v\}$ is infeasible. This means that there exists a path P in G with k nodes that contains no node in $B \backslash \{v\}$. As B is feasible, this path must contain v and as such it fulfils condition (ii).

Conversely let conditions (i) and (ii) hold. By (i) B is feasible and by (ii) it is minimal as removing any node in $B \backslash A$ results in the creation of a path containing k nodes that is not hit. □

We will be using the above condition a lot and to save the need to reference this lemma constantly we give the second part a name: A path that fulfils condition (ii) for a node $v \in B$ is called an *indicator path* for v. So to show that B is a minimal augmentation we will check that every node in B has an indicator path and that no path with k nodes exists which has only nodes not contained in B. This is then sufficient by the lemma.

Note that the above lemma also shows that the value γ will never actually need to attain the value k in our case as VIOLATION returns all violated sets (they are all minimal). As a result there are at least $|B|$ many sets of cardinality one contained in the sum which consequently can never quite reach $k|\mathcal{V}(A)|$.

One final remark should be made concerning the $\mathcal{V}$-APC and $\mathcal{V}$-SPC.

4.2 Remark. The AVAPC is a special case of the $\mathcal{V}$-APC just as the AVSPC is a special case of the $\mathcal{V}$-SPC. Consequently any example for the AVAPC or AVSPC is also an example for the $\mathcal{V}$-APC or $\mathcal{V}$-SPC respectively. This means that if we show that β can be equal to k in the AVAPC and AVSPC then this is also true for the $\mathcal{V}$-APC and $\mathcal{V}$-SPC. The same is true for the value of γ, so it suffices to look at the special cases in the following.

4.1. Path Covers on General Graphs

As promised we begin with general graphs. We are going to find graphs (dependent on k) that reach a β-value of k and a γ-value that is arbitrarily close to k for all cases except the TVSPC.

We begin by regarding the β-values. For the AVAPC problem we construct two graphs that show that β can attain the value k. For the first we begin with a path $P_1 = (v_1, \dots, v_k)$ with k nodes and a path $P_2 = (u_1, \dots, u_{k-1})$ with $k-1$ nodes. We will then add edges from every vertex v_i in P_1 to the vertex u_1. We can then choose every vertex in the path P_1 as part of the minimal augmentation B as every node has an indicator path (by taking the nodes in P_2) and B is feasible (only $k-1$ nodes are not in B in total). The resulting graph is shown in Figure 4.1. In all our graphs the coloured nodes are the ones in B.

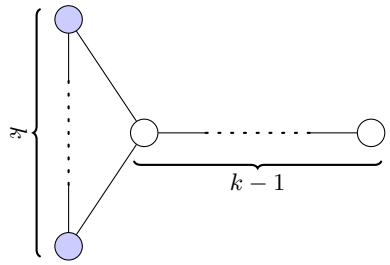


Figure 4.1.: First example with $\beta = k$ for the AVAPC problem.

Our second example differs from this only slightly. We replace the shared path P_2 by k paths with $k-1$ vertices as shown in Figure 4.2. Clearly the nodes in P_1 still form a minimal augmentation.

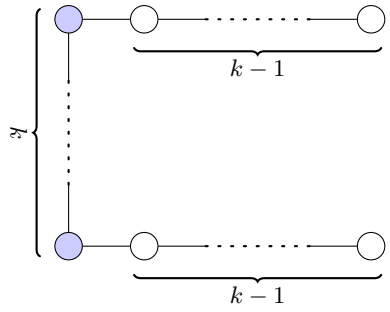


Figure 4.2.: Second example with $\beta = k$ for the AVAPC problem.

This example also covers the case of the AVSPC. For the TVAPC we construct a graph similar to the one in Figure 4.1 to show that $\beta = k$. It can be seen in Figure 4.3. Let $P_1 = (v_1, \dots, v_k, v_{k+1})$ and $P_2 = (u_1, \dots, u_{k-2})$ and choose $s = v_1$ and $t = v_{k+1}$. We add an edge from s to u_1 and from u_{k-2} to v_i for $i = 2, \dots, k+1$. Then the s-t paths consist of the path P_1 and the paths from s via P_2 to a v_i and from there to t. If we choose the set B to be $\{v_2, \dots, v_{k+1}\}$, this is a minimal augmentation as again every node in B has an indicator path and only $k-1$ nodes are not in B. The indicator path for v_i is the path from s over P_2 to v_i which has length k and is a subpath of an s-t-path and as such a violated set.

We summarise what we have done so far in the next lemma.

4.3 Lemma. For the AVAPC, AVSPC and TVAPC problems we get an approximation guarantee of $\beta = k$ for Algorithm 3.2.

Now just the TVSPC problem remains where we want to show that $\beta \leq 2$ and that this value can be attained. Here the violated sets consist of subpaths of a single s-t-path P as

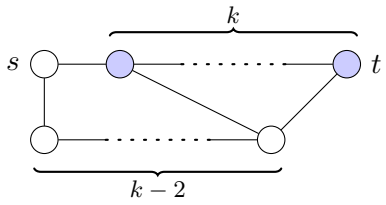


Figure 4.3.: Example with $\beta = k$ for the TVAPC problem.

we have assumed the shortest path to be unique. Let A be infeasible and B any minimal augmentation of A. We claim that $B \cap T(A)$ contains at most two elements. To see this notice that $P' = T(A)$ is a subpath of P that does not contain an element of A. Assume there are three vertices $v_1, v_2, v_3 \in (B \backslash A) \cap P'$. We can assume without loss of generality that they appear in the path in that order. The situation described is displayed in Figure 4.4.

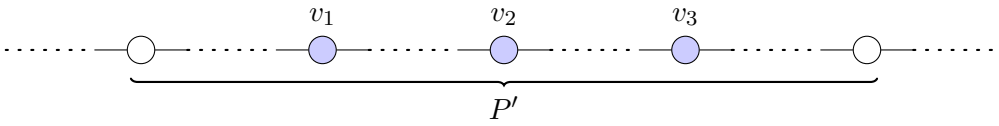


Figure 4.4.: Visualisation of the path P'.

By minimality we get that v_2 must have an indicator path, so there must be a subpath of P of length k that contains only v_2 as a node in B. This, however, is impossible as there are at most $k-3$ nodes that are not in B on a subpath containing v_2. Consequently we have $\beta \leq 2$. The graph in Figure 4.5 shows an example where $\beta = 2$ and the nodes in B clearly form a minimal augmentation.

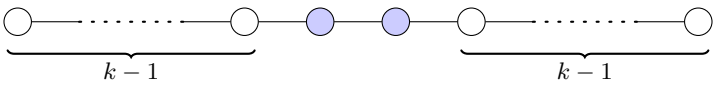


Figure 4.5.: Example with $\beta = 2$ for the TVSPC problem.

We have now proved the lemma below.

4.4 Lemma. For the TVSPC problem we get an approximation guarantee of $\beta = 2$ for Algorithm 3.2.

It should be noted that this problem is easy to solve optimally as we will see in Chapter 5.

Now just the γ-values remain. In the following we will say a path or violated set has value i if it contains i nodes that are in the minimal augmentation B. Also to simplify things we generally refer to the violated sets as paths and not as the nodesets of paths, which would be more accurate. The general idea is the following. We know that for every node in B we get a path with just a single node that is hit, so a path of value one. This means that we need to find examples that have lots of paths of value k compared to those of lower value. For the AVAPC we proceed as follows. As usual we have a path $P = (u_1, \ldots, u_{k-1})$, but here we also need a clique C_m of size m. Let every node in C_m have an edge to u_1 and we choose B to be all elements of the clique. This situation is shown in Figure 4.6.

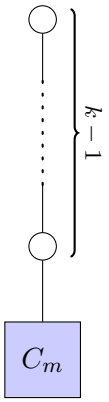


Figure 4.6.: Example with γ arbitrarily close to k for the AVAPC problem.

B is a minimal augmentation as every node in the clique is the only node in B on the path from it to u_1 and from there to u_{k-1}, its indicator path, and again only $k-1$ nodes are not in B. So let us count the amount of nodesets on paths of length k. To make things easier we will count them by their value. The paths of value one must include the entire path P and are as such exactly the paths described above, so we get $m = \binom{m}{1}$ many of these. Paths of value two must contain all the nodes in P up to u_{k-2} and two nodes from the clique, giving us $\binom{m}{2}$ such paths and in general we get $\binom{m}{i}$ paths of value i. Note that this uses the fact that we regard the nodesets of such paths and the order in which the nodes in the clique are visited is irrelevant. So in this case we get

$$\gamma = \frac{\sum_{i=1}^{k} \binom{m}{i} \cdot i}{\sum_{i=1}^{k} \binom{m}{i}}.$$

We now show that $\gamma \to k$ for $m \to \infty$ and as such we will not be able to find a better guarantee than k. First we get that for $1 \leq i \leq k-1$ and $m \geq 2k$ (which we can assume as we are interested in large m) the inequality

$$\frac{k-i}{\sum_{j=1}^{k} \frac{i!(m-i)!}{j!(m-j)!}} \leq \frac{k-1}{\sum_{j=1}^{k} \frac{(k-1)!(m-k+1)!}{j!(m-j)!}} \leq \frac{k-1}{\binom{m-k+1}{k}} = \frac{k(k-1)}{m-k+1} \tag{4.1}$$

holds. The first inequality is true by our assumptions on i and m as the choice of $i = 1$ maximises the nominator and $i = k-1$ minimises the denominator. The second inequality is clear as we only drop all but the k-th term of the sum. We can use this to prove that $\gamma \to k$:

$$\gamma = \frac{\sum_{i=1}^{k} \frac{m!}{i!(m-i)!} \cdot i}{\sum_{j=1}^{k} \frac{m!}{j!(m-j)!}} = k + \frac{\sum_{i=1}^{k} \frac{i}{i!(m-i)!} - \sum_{i=1}^{k} \frac{k}{i!(m-i)!}}{\sum_{j=1}^{k} \frac{1}{j!(m-j)!}} =$$

$$k - \frac{\sum_{i=1}^{k-1} \frac{k-i}{i!(m-i)!}}{\sum_{j=1}^{k} \frac{1}{j!(m-j)!}} = k - \sum_{i=1}^{k-1} \frac{k-i}{\sum_{j=1}^{k} \frac{i!(m-i)!}{j!(m-j)!}} \overset{(4.1)}{\geq} k - \sum_{i=1}^{k-1} \frac{k(k-1)}{m-k+1} =$$

$$k - \frac{k(k-1)^2}{m-k+1} \xrightarrow{m \to \infty} k.$$

For $k = 2$ the example also works in the case of shortest paths. However, for larger k, this is no longer the case as the direct connections in the clique are shorter than visiting all other

nodes first. So we need a new graph for the AVSPC problem. As we could not find a simple example, we will not give one here. Instead, we refer to the result on trees shown in Corollary 4.14 in the next section. As a tree clearly is a special case of a general graph, this suffices.

So we have shown (or will show) the next lemma.

4.5 Lemma. For the AVAPC and AVSPC problems we cannot get an approximation guarantee of $\gamma < k$ for Algorithm 3.4.

So we are left with the two vertex problems. The TVAPC problem is easy now as the example from Figure 4.6 does the trick here as well if we choose $s = u_1$ and t to be any element in C_m. Clearly all the nodesets we regarded before are then also nodesets of subpaths of an s-t-path, simply by adding all nodes in P and visiting the node t last if it is contained and adding it if it is not. Notice that this is the second time where the distinction between paths and nodesets of paths has been relevant, letting us choose t to be the last node on the path instead of it having a fixed position. To summarise we have:

4.6 Lemma. For the TVAPC problem we cannot get an approximation guarantee of $\gamma < k$ for Algorithm 3.4.

Before we get to the proof that the value of γ is never worse than $2 - \frac{2}{k+1}$, we show that this value is actually obtained for our example in Figure 4.5. This is rather simple as all we need to do is count the paths of value one and two. There are exactly two paths of value one and $k-1$ paths of value two which results in

$$\gamma = \frac{2 + 2(k-1)}{2 + (k-1)} = \frac{2k}{k+1} = 2 - \frac{2}{k+1}.$$

Now let us assume we have a shortest s-t-path P with l nodes and a minimal augmentation B of A and we want to show that γ is at most $2 - \frac{2}{k+1}$. We claim that we can assume A to be the empty set. If A is not equal to the empty set, the violated sets never contain elements of A and are consequently subpaths of P of length k with no nodes in A. Let $P_1, \dots, P_n$ be the subpaths of P created by removing the nodes in A, then the violated sets correspond to all subpaths of the P_i of length k. Let P_i contain x_i violated sets and let the violated path P_j^i have value v_j^i. The inequality $\gamma \leq 2 - \frac{2}{k+1}$ is true for all these paths which gives us that

$$\gamma_i = \frac{\sum_{j=1}^{x_i} v_j^i}{x_i} \leq 2 - \frac{2}{k+1} \text{ for all } i$$

or equivalently that

$$\sum_{j=1}^{x_i} v_j^i \leq \left(2 - \frac{2}{k+1}\right) \cdot x_i. \tag{4.2}$$

If we look at the value of γ on the entire path P, we get that

$$\gamma = \frac{\sum_{i=1}^{n} \sum_{j=1}^{x_i} v_j^i}{\sum_{i=1}^{n} x_i} \overset{(4.2)}{\leq} \frac{\sum_{i=1}^{n} \left(2 - \frac{2}{k+1}\right) \cdot x_i}{\sum_{i=1}^{n} x_i} = \left(2 - \frac{2}{k+1}\right) \cdot \frac{\sum_{i=1}^{n} x_i}{\sum_{i=1}^{n} x_i} = 2 - \frac{2}{k+1}.$$

So we can assume without loss of generality that $A = \emptyset$. For the next part a bit of notation will be helpful: We have a path $P = (v_1, \dots, v_l)$ and let $B = \{v_{i_1}, \dots, v_{i_n}\}$ be a minimal

augmentation of the empty set where $i_1 < \ldots < i_n$. In a path with l nodes there are $l-k+1$ violated sets as the first node up to the $(l-k+1)$-st node are valid starting points for paths with k vertices. We divide these $l-k+1$ paths into two sets S_1 and S_2, where S_i contains all the subpaths of value i. With this notation we get

$$\gamma = \frac{|S_1| + 2|S_2|}{l-k+1}$$

and $|S_1| + |S_2| = l-k+1$. Next we make a few useful observations (and call them a lemma as some might require a proof).

4.7 Lemma. (i) For all i the following statement holds: If $v_i \in B$, then there exists a subpath of length k such that v_i is the only node of B on this path.
(ii) For all i the following statement holds: The node v_i is contained in B or there is a node $v_j \in B$ with $|i-j| \leq \frac{k-1}{2}$.
(iii) In the first and last k nodes of P exactly one node is contained in B.
(iv) On any path of length k at most two nodes are chosen.

Proof. (i) The path required is just the indicator path.
(ii) Assume this is not the case, then there are at least $\frac{k-1}{2}$ nodes to both sides of v_i that are not in B. Together with v_i these form a path of at least length k with no node hit, a contradiction to B being feasible.
(iii) The claim follows as at least one must be chosen by feasibility of B and more than one cannot be in B as any path of length k containing the outer one would also contain the inner, contradicting (i).
(iv) We proved this when showing that β is at most two, see Figure 4.4. □

Now to the problem at hand. First we look at an easy case. We assume $l \geq k$ as there are no violated paths otherwise. If $k \leq l \leq 2k-1$, the solution consists of one or two nodes by Lemma 4.7(iii). If just one is selected, $\gamma = 1$ as $S_2 = \emptyset$ and all violated paths contain exactly one node in B. If two are selected, there are at least the two indicator paths which only contain one node of B. It follows that there are at most $l-k-1$ paths of value two and we get

$$\gamma \leq \frac{2 + 2(l-k-1)}{l-k+1} = \frac{2(l-k)}{l-k+1} = 2 - \frac{2}{l-k+1} \overset{l<2k}{<} 2 - \frac{2}{k+1}.$$

We are just left with the case $l \geq 2k$ and consequently we have $n \geq 2$. By Lemma 4.7(iii) the first path only contains the node v_{i_1} of the nodes in B. Until v_{i_2} is reached all following paths also only contain this node. As a result there are $(i_2 - 1) - (k-1)$ paths of value one with the node v_{i_1}. To see this, notice that $i_2 - 1$ is the last end point of a path that will still have value one as v_{i_2} is reached afterwards. Furthermore, $k-1$ is the last invalid end point as it does not correspond to a path of length k. We then get paths of value two that contain the two nodes v_{i_1} and v_{i_2} and of these there are $(i_1 + k - 1) - (i_2 - 1) = k - i_2 + i_1$ many. This follows by a similar argument: Here the last end point that does not include v_{i_2} is $i_2 - 1$ and the last endpoint that still includes v_{i_1} is $i_1 + k - 1$. Continuing in this fashion gives us $(i_3 - 1) - (i_1 + k - 1) = i_3 - i_1 - k$ paths that only include the node v_{i_2} and so on. So in general we get $i_{j+1} - i_{j-1} - k$ paths that only include the node v_{i_j} and $k - i_{j+1} + i_j$ many paths that include the two nodes v_{i_j} and $v_{i_{j+1}}$. Here we set $i_0 = 0$ and $i_{n+1} = l + 1$. These

values are all non-negative, the first kind as there are at least k nodes between $v_{i_{j+1}}$ and $v_{i_{j-1}}$ as v_{i_j} must have an indicator path and the second kind by feasibility.

Summing up over all these values yields $|S_2| = \sum_{j=1}^{n-1} k - i_{j+1} + i_j$ and $|S_1| = \sum_{j=1}^{n} i_{j+1} - i_{j-1} - k$. Simplifying gives us $|S_2| = (n-1)k + i_1 - i_n$ and $|S_1| = (i_{n+1} + i_n - i_1 - i_0) - nk = l + 1 - nk + i_1 - i_n$. If we use this in our calculation of γ, we get

$$\begin{aligned}\gamma &= \frac{[l+1-nk+i_1-i_n]+2[(n-1)k+i_1-i_n]}{l-k+1} \\ &= \frac{2(n-1)k+i_1-i_n+l+1-nk}{l-k+1} \\ &= \frac{l-k+1+(n-1)k+i_1-i_n}{l-k+1} \\ &= 1+\frac{(n-1)k+i_1-i_n}{l-k+1}.\end{aligned} \tag{4.3}$$

The description above shows us that the value of γ is larger for increasing n and if i_1 and i_n are less far apart. Somewhat surprisingly it does not directly depend on anything else, like the locations of all the other v_{i_j}.

We first take a look at what the maximal amount of nodes a minimal augmentation B can possibly contain is. We then show that the value γ attains its maximum (which is what we are interested in) for such an augmentation.

4.8 Theorem. Any minimal augmentation B with a maximal amount of nodes has $n = 2x$ nodes for $x = \left\lfloor \frac{l}{k+1} \right\rfloor$ if $l \neq c(k+1)-1$ for $c \in \mathbb{N}$ and it has $n = 2x+1$ nodes if $l = c(k+1)-1$ for $c \in \mathbb{N}$.

Proof. We show that the amount of nodes in B on a path of length $l = x(k+1)+r$, where $r \leq k$, is exactly the same as in the set B' shown in Figure 4.7, in which we choose B' to contain the nodes $v_{i(k+1)-k}$ and $v_{i(k+1)}$ for $i = 1, \ldots, x$ and the node v_{l-k+1} if $r = k$. We do this by taking an arbitrary minimal augmentation B with a maximal amount of nodes and transforming it into B' in a series of steps, after each of which we still have a new minimal augmentation with the same amount of nodes.

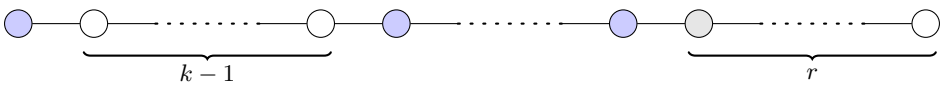


Figure 4.7.: Generic minimal augmentation B' with maximal amount of nodes.

The transformation works in the following way: Regard the first node v_{i_1} in B. If it is not the node v_1, we check how many nodes to its right are also not contained in B, let this amount be x_1. Then we remove the node v_{i_1} and add the node $v_{i_1-(k-1)+x_1}$, so we move the picked node $(k-1)-x_1$ nodes to the left. Note that it is always possible to move the node in this manner as v_{i_1} is the unique node in B among the first k nodes by Lemma 4.7(iii). As a result $i_2 \geq k+1$ and $k-1-x_1 = k-1-(i_2-i_1-1) \leq i_1-1$. Consequently we have $i_1-(k-1)+x_1 \geq 1$. The result is still feasible as we ensured that there are at most $x_1+(k-1)-x_1 = k-1$ nodes between v_{i_1} and v_{i_2} and all other subpaths consisting of nodes that are not in B either remain unchanged or have a reduced length. It is also still minimal as all nodes aside from v_{i_1} still

have the same corresponding indicator path and for v_{i_1} the path consisting of the node itself and the next $k-1$ nodes to its right does the job. We call this step a *feasible move* for future use.

In the next step we check whether v_{i_1} is now the first node. If this is still not the case, we remove all nodes from B and for every removed node v_j we add the node v_{j-i_1+1}. This can be described as shifting all nodes in B to the left by $i_1 - 1$ nodes and we refer to it as a *shift move*. Afterwards we finally have $v_{i_1} = v_1$. Again the result gives us a feasible set, the only critical path would be the last k nodes. As B contained the maximal amount of nodes, we must have a node in the new B on that path or we would get a minimal augmentation with more nodes than B. It clearly remains minimal as every path keeps its corresponding indicator path.

Now we are in the situation that B contains the nodes $i_1 = 1$ and $i_2 = k+1$ as we would like. We repeat these two steps for the next node as long as at least $k+1$ nodes remain. For the feasible move we needed the third condition of Lemma 4.7 which only holds for the start of the path. However, it also holds for every new node we regard as the node $v_{j(k+1)}$ is contained in B and by minimality we cannot select two nodes in the first k as the middle one would not have a corresponding subpath with $k-1$ nodes not in B (again see Figure 4.4).

Once we have done this we get a set B which coincides with B' on the first $2x$ nodes. If $r < k$, we are in the case $l \neq c(k+1) - 1$ (with $c = x+1$) and we need to show that there are, in fact, no further nodes in B. This is clearly the case as any node selected in the last r nodes would be superfluous, a contradiction to minimality. If $r = k$, then $l = (x+1)(k+1) - 1$ and we need to show that B contains another node and that it can be transformed into B' if the additional node is not v_{l-k+1}. Again this is clear as it is possible to select the node v_{l-k+1}, so by the fact that B contains the maximal amount of nodes it has to contain another node. Replacing the node it contains with the node v_{l-k+1} does not affect feasibility or minimality, completing the proof. □

To get the estimate for γ we also need to look at the minimal distance in the case that B contains the maximal amount of nodes. We begin with the first case where $l \neq c(k+1) - 1$ for $c \in \mathbb{N}$.

4.9 Theorem. In any minimal augmentation B with a maximal amount of nodes the distance $i_n - i_1$ is at least $2x(k+1) - (l+1)$ where $x = \left\lfloor \frac{l}{k+1} \right\rfloor$ if $l \neq c(k+1) - 1$ for $c \in \mathbb{N}$.

Proof. As in the previous proof we show the statement by transforming an arbitrary minimal augmentation B with maximal amount of nodes into the generic one, B', shown in Figure 4.8. For a path with l nodes we know that B contains $2x$ nodes by the previous theorem. Let the nodes in B' be the nodes $v_{j(k+1)-k+a}$ and $v_{j(k+1)}$ for $j = 1, \ldots, x$, where $a = l - x(k+1)$ and $b = k - 1 - a$.

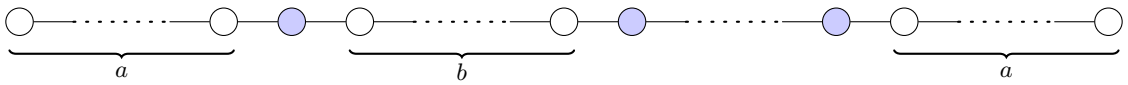


Figure 4.8.: Generic minimal augmentation B' with minimal distance of the outer nodes.

Again every transformation step leads to a new minimal augmentation, but here we also ensure that the distance $i_n - i_1$ never increases. This means that if we start with an instance which minimises the distance it is preserved and the distance in the generic example must be minimal as well.

The transformation works as follows: To begin we fix the node v_{i_1} and set $a' = i_1 - 1 \leq k - 1$ to be the amount of nodes to its left. Then there must be at least $b' = k - 1 - a'$ many nodes to its right that are also not contained in B by the existence of an indicator path. If the following node is not contained in B, we proceed as in the previous proof by using a feasible move followed by a shift move on all but the first node v_{i_1} to ensure that it is picked afterwards. Observe that a shift move decreases the distance between i_n and i_1 as i_n is decreased whereas i_1 remains unchanged and the feasible move creates no change here as it is not applied to either i_1 or i_n. Consequently, if we have an instance with a minimal distance, the shift move will never be necessary. Then again, as an indicator path exists, we have at least a' nodes to the right of the new v_{i_2} that are not in B and we ensure that the next node is picked by the same use of a feasible move and a shift move (now applied to all nodes in B other than v_{i_1} and v_{i_2}). This leads to the situation shown in Figure 4.9 where we have the path with a' nodes followed by a node in B and b' nodes followed by one in B exactly x times and some rest nodes r.

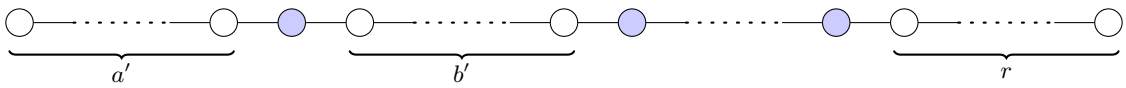


Figure 4.9.: The minimal augmentation B after the first step of the transformation.

As the first part accounts for $x(k+1)$ nodes, we get that $r = l - x(k+1) = a$. We also get that $a \geq a'$ as the node v_{i_n} has an indicator path. We fix the last node and apply the previous transformations in the opposite direction, creating the generic instance.

So this instance has the lowest possible value for $i_n - i_1$. As $i_1 = a + 1$ and $i_n = x(k+1)$, we get that $i_n - i_1 = x(k+1) - (l - x(k+1) + 1) = 2x(k+1) - (l+1)$ as claimed. □

We now claim that γ attains its maximum for an augmentation that contains the maximal amount of nodes possible (in the case $l \neq c(k+1) - 1$). The maximal difference between the distances of i_1 and i_n we can achieve in two minimal augmentations is $2k - 2$ by Lemma 4.7(iii). Consequently no augmentation with two or more nodes less than the maximal amount can be one where γ is maximal (as the n is weighted with k). So it suffices to show that for any augmentation B with one node less than the maximal amount we can find an augmentation B' with at least the same value of γ but with an additional node. First we take any augmentation B'' that contains one further node. If $i_n - i_1$ increases by at most k in B'', the new value of γ is the same or a higher. So we can assume that the distance in B'' increases by more than k. This means that there are more than k nodes that are not contained in B at the start and end of the path, or more formally $i_1 - 1 + l - i_n > k$. We transform the solution B, exactly as in the above proof, into the form shown in Figure 4.8. Note that the last node is not in the new set B' as we have exactly one node less. So B' is no longer feasible, but the distance between i_1 and i_n has not increased. We now remove all nodes v_{i_j} from B' where j is odd and for each removed node we add the new node $v_{i_j - a}$. This corresponds to the application of a feasible move to all odd v_{i_j}. The result is the generic augmentation for the maximal amount

of nodes from Figure 4.7, without the final node. The path ends in $k+r$ nodes that are not contained in B', adding the node v_{i_n+k} to B' then creates an instance as required.

As it suffices to look at augmentations with the maximal amount of nodes, using the previous two theorems in equation (4.3) yields

$$\begin{aligned}\gamma &= 1 + \frac{(n-1)k + i_1 - i_n}{l-k+1} \leq 1 + \frac{(2x-1)k + (l+1-2x(k+1))}{l-k+1} \\ &= 1 + \frac{2xk - k + l + 1 - 2xk - 2x}{l-k+1} = 1 + \frac{l-k+1-2x}{l-k+1} \\ &= 2 - \frac{2x}{l-k+1} = 2 - \frac{2\left\lfloor \frac{l}{k+1} \right\rfloor}{l-k+1}.\end{aligned} \tag{4.4}$$

If we want to find the maximum for this ratio, we only need to look at the cases where $l = c(k+1) - 2$ for some $c \in \mathbb{N}$ as the nominator is fixed for all values in the intervals $[c(k+1), c(k+1)-1]$ and the denominator increases. As we excluded the values $l = c(k+1)-1$, the maximum must be attained for $l = c(k+1) - 2$. Plugging it into the inequality in (4.4) we get that

$$\gamma \leq 2 - \frac{2\left\lfloor \frac{c(k+1)-2}{k+1} \right\rfloor}{c(k+1) - 2 - k + 1} = 2 - \frac{2(c-1)}{(c-1)(k+1)} = 2 - \frac{2}{k+1}.$$

This seems very promising as the only case left is that when $l = c(k+1) - 1$ for some $c \in \mathbb{N}$. We already know the maximal amount of nodes by Theorem 4.8 so just the minimal distance remains to be checked.

4.10 Theorem. In any minimal augmentation B with a maximal amount of nodes the distance $i_n - i_1$ is at least $x(k+1)$ where $x = \left\lfloor \frac{l}{k+1} \right\rfloor$ and $l = c(k+1) - 1$ for $c \in \mathbb{N}$.

Proof. The proof is very similar to the last one, the main difference is that we have an uneven amount of nodes as $n = 2x + 1$. We apply the first step of the procedure from the proof of Theorem 4.9 to the minimal augmentation B. This results in the same situation as before, which is shown in Figure 4.9.

Here we get that $r = l - x(k+1) - a' - 1$ and, as $l = (x+1)(k+1) - 1$, it simplifies to $r = k - 1 - a' = b'$. Notice that, unlike in the previous proofs, this is not generic as it depends on the augmentation B (since we did not change i_1). We can remedy this by applying a shift move to ensure that $v_1 \in B$, but this is not necessary. We calculate the distance in this instance: As $i_1 = a' + 1$ by definition and $i_n = x(k+1) + a' + 1$, the distance $i_n - i_1 = x(k+1)$ is as claimed. □

Here it is very easy to see that the maximal value for γ is attained for an instance with a maximal amount of nodes. The minimal distance is $i_n - i_1 = (l-k+1) - k = l - 2k + 1$ and the distance in the instance above is $x(k+1) = l - k$. As a result their difference is $(l-k) - (l-2k+1) = k - 1 < k$.

Again using both these results in equation (4.3) and the fact that $x = c - 1$ for $l = c(k+1) - 1$ gives

$$\begin{aligned}\gamma &= 1 + \frac{(n-1)k + i_1 - i_n}{l-k+1} \leq 1 + \frac{(2x+1-1)k + (-x(k+1))}{l-k+1} \\ &= 1 + \frac{2xk - x(k+1)}{l-k+1} = 1 + \frac{x(k-1)}{c(k+1)-k} \\ &= 1 + \frac{(c-1)(k-1)}{(c-1)k+c} = 1 + \frac{(c-1)k + c - 2c + 1}{(c-1)k+c} \\ &= 2 - \frac{2c-1}{(c-1)k+c} = 2 - \frac{2-\frac{1}{c}}{\frac{c-1}{c}k+1} \leq 2 - \frac{2}{\frac{c-1}{c}k+1} \leq 2 - \frac{2}{k+1}.\end{aligned}$$

This finally finishes the last case and shows $\gamma \leq 2 - \frac{2}{k+1}$ and consequently proves the corollary below.

4.11 Corollary. For the TVSPC problem we get an approximation guarantee of $\gamma = 2 - \frac{2}{k+1}$ for Algorithm 3.4.

4.2. Path Covers on Trees

In this section we look at the case where the graphs are restricted to trees. Given this setting, the APC and SPC problems are identical as there is a unique path between every pair of nodes, so the (simple) paths and the shortest paths coincide. Additionally the results from the TVSPC problem from the previous section carry over to this case, giving us the same factors of $\beta = 2$ and $\gamma = 2 - \frac{2}{k+1}$. So just the AVSPC problem remains to be examined. The value of β stays the same, so $\beta = k$ as the example given in Figure 4.2 is a tree. To summarise we get:

4.12 Lemma. The bounds $\beta = 2$ and $\gamma = 2 - \frac{2}{k+1}$ for the TVSPC on general graphs carry over to the TVAPC and TVSPC problems on trees.

The example for the γ value is the only thing missing now. To complete the previous section we need one for $k \geq 3$, but we begin with $k \geq 4$. Regard the graph $G = G_{m,n}$ in Figure 4.10.

It consists of a central node v which is adjacent to m nodes $v_1, \ldots, v_m$ and connected to the graphs $G_m^1, \ldots, G_m^n$, where each G_m^j has the form shown in Figure 4.11.

More specifically, each G_m^j consists of a path of $k-3$ nodes $u_1^j, \ldots, u_{k-3}^j$, where u_1^j is adjacent to m nodes $v_1^j, \ldots, v_m^j$ and u_{k-3}^j is the node that is adjacent to v. Furthermore, each of these nodes is a node in B meaning each of them needs an indicator path. We remedy this by giving every node an edge to a path with $k-1$ nodes that are not in B, displayed as a node labelled p in both figures. As usual we count the paths by values, but first we make a general observation: Any path of value $i \neq 1$ begins with a nodes that are not in B followed by i nodes in B and completed by $k-i-a$ nodes not in B. As a result we can associate any path in $G_{m,n}$ with a path consisting of i nodes in the graph $G[B]$ induced by the nodes in B. For any such path in $G[B]$ we have $k+1-i$ paths in $G_{m,n}$, one for each possible value of a which range from 0 to $k-i$. Consequently it suffices to count the paths of value i in $G[B]$ and multiply those amounts by $(k+1-i)$. In the case that $i = 1$ we get exactly one path per node in B.

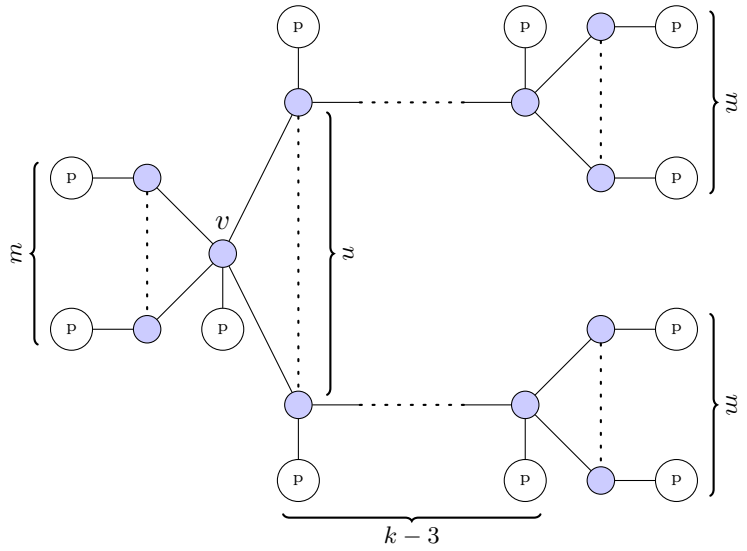


Figure 4.10.: Example with γ arbitrarily close to k for the AVSPC problem.

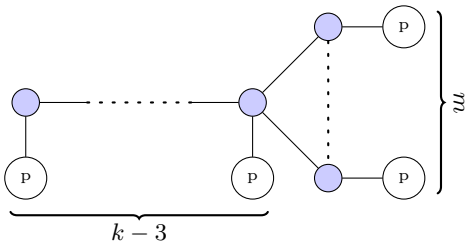


Figure 4.11.: The subgraph G_m^j.

As this example is significantly more complicated than any of our previous ones, we first give an intuition as to why it is a good idea to regard exactly this graph, before we begin counting the paths. The goal is, informally speaking, to find a graph that has many paths of value k compared to all other values such that the value of γ converges to k. The tree here does exactly that as only the paths of value k can have nodes in two of the subgraphs G_m^j and reach the nodes v_l^j in one of them. This gives a total of $\mathcal{O}(n^2 m)$ many paths of value k, something that cannot be achieved with any other length.

Now let us get to the counting. We break the example into several parts to give the analysis more structure and make it easier to follow. As a start it seems reasonable to count the paths of value i in the subgraphs G_m^j. It turns out that it is easier to do so if we add the node v to these subgraphs as well because it simplifies the results and saves a few case distinctions. For clarity this subgraph is shown in Figure 4.12 and the results are summarised in Table 4.2.

Let us begin with the paths of value one. As remarked above there is exactly one for every node in B and there are $k-3+m+1$ many of these, the $+1$ being the node v. However, this would lead to us counting the path with just the node v a total of n times if we just multiplied the results by n later. To keep this in mind we denote their value by $m+k-3(+1)$.

Up next are the values $2 \leq i \leq k-2$. In these cases we have paths that contain a node v_l^j. These are of the form $v_l^j, u_1^j, \ldots, u_{i-1}^j$, of which there are a total of m. In the case that $i = 3$

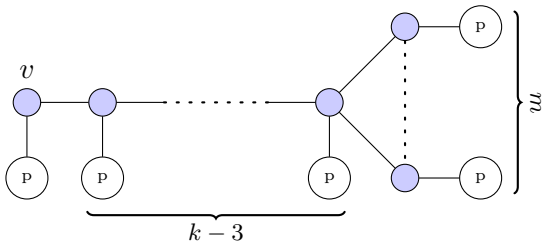


Figure 4.12.: The extended subgraph G_m^j.

value	amount
1	$m + k - 3(+1)$
$2 \leq i \leq k-2, i \neq 3$	$[m + k - 1 - i](k + 1 - i)$
3	$[\binom{m+1}{2} + k - 4](k - 2)$
$k - 1$	$2m$
k	0

Table 4.2.: The amount of paths of value $1 \leq i \leq k$ in the extended subgraph G_m^j.

they can also contain two such nodes and be of the form $v_l^j, u_1^j, v_{l'}^j$ with $l' \neq l$. This adds another $\binom{m}{2}$ new paths as they are defined by their two endpoints. The paths of length i that only contain nodes in the set $\{u_1^j, \dots u_{k-3}^j, v\}$ are the final set. Of these there are $k - 1 - i$ many: The paths have the form $u_l^j, \dots u_{l+i-1}^j$ for $l = 1, \dots, k - 2 - i + 1$, as for the final value $l + i - 1$ is equal to $k - 2$. Here we let $u_{k-2}^j = v$. This gives us a total amount of $m + k - 1 - i$ paths if $i \neq 3$ and $\binom{m+1}{2} + k - 1 - i$ if $i = 3$. Notice that we only required that $k \geq 4$ so the case $i = 3$ is not necessarily contained here at all. If we are, in fact, in the situation that $k = 4$, there are still $\binom{m+1}{2}$ paths that contain a node v_l^j as they correspond to a choice of two endpoints from the set $\{v_1^j, \dots, v_m^j, v\}$. However, there are now no paths that do not contain any of the nodes v_l^j, resulting in a total of $\binom{m+1}{2} = \binom{m+1}{2} + (k - 1 - 3)$. So the formula from the previous case still applies, even if $k = 4$.

We have already counted the paths of value $k - 1$ in the case $k = 4$ above, so we can assume that $k \geq 5$. The only option then are the paths that contain the nodes $v, u_{k-3}^j, \dots, u_1^j$ and a node v_l^j, so there are m such paths. Paths of value k do not exist at all in these subgraphs.

That being done we can now count all the paths in $G_{m,n}$ by taking the amounts we have calculated for the extended graphs G_m^j, multiplying them by n (with the exception of the path with just the node v as noted above), and, additionally, counting the missing paths. For these we distinguish between the following two cases: The paths that contain a node v_l and those that have nodes from two subgraphs G_m^j. These are the only two options as any path that contains no node v_l and does not contain nodes from multiple G_m^j must already be contained completely in one of the extended graphs regarded earlier.

Let us start with the simpler first case. For value one we get m such paths, one for each node v_l, and for value two another m, one for each path containing v_l and v. For the values 3 to $k - 1$ we then get the paths consisting of the vertices $v_l, v, u_{k-3}^j, \dots, u_{k-3-(i-2)+1}^j = u_{k-i}^j$.

These are uniquely defined by their first (one of the v_l) and final node (one of the u^j_{k-i}), giving us mn such paths. Finally for the value k the paths are of the form $v_l, v, u^j_{k-3}, \dots, u^j_1, v^j_{l'}$ of which there are m^2n many. In summary we get the results shown in Table 4.3.

value	amount
1	m
2	$m(k-1)$
$3 \leq i \leq k-1$	$mn(k+1-i)$
k	nm^2

Table 4.3.: The amount of paths of value $1 \leq i \leq k$ containing a node v_l.

We are left with the final case where we need to look at paths that contain nodes from two of the subgraphs G^j_m. First of all we pick two such subgraphs which leads to the situation shown in Figure 4.13. We count the paths in this graph and then multiply the result by the $\binom{n}{2}$ possible choices of two subgraphs, the results of which are shown in Table 4.4.

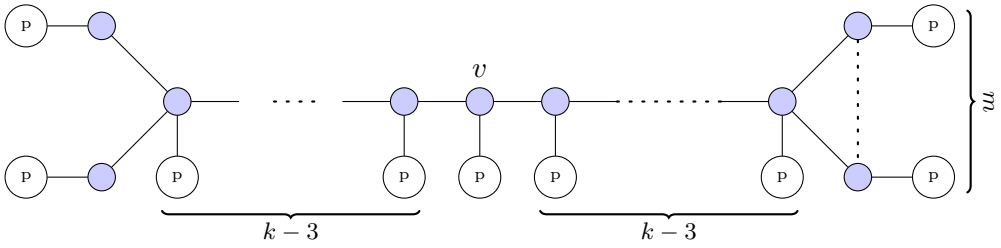


Figure 4.13.: The subgraph consisting of two G^j_m and v.

value	amount
$1 \leq i \leq 2$	0
$3 \leq i \leq k-1$	$\binom{n}{2}(i-2)(k+1-i)$
k	$\binom{n}{2}(2m+k-4)$

Table 4.4.: The amount of paths of value $1 \leq i \leq k$ containing a node in two subgraphs G^j_m.

Clearly, there are no such paths of value one or two as any such path contains at least the nodes $u^j_{k-3}, v, u^{j'}_{k-3}$. Let us regard a path of value i where $3 \leq i \leq k-1$. Then it has a nodes in G^j_m and consequently $i-1-a$ nodes in $G^{j'}_m$. We get both $a \geq 1$ and $i-1-a \geq 1$ or equivalently $a \leq i-2$. Hence we get $i-2$ such paths as they can never contain a node v^j_l. For value k this changes as $i-2 = k-2 > k-3$, so the nodes v^j_l can now be reached. More precisely, the choices for a now range from 1 to $k-2$ where the choices $a = 2, \dots, k-3$ correspond to one path as in the previous case and the choice of $a = 1$ or $a = k-2$ leads to m paths each. This results in a total of $2m+k-4$ such paths.

Combining all three cases leads to the results shown in Table 4.5.

We can finally take a look at the value of γ: Let x_i be the amount of paths of value i, then

value	amount	class
1	$n(m+k-3)+1+m$	$\Theta(nm)$
2	$[n(m+k-3)+m](k-1)$	$\Theta(nm)$
3	$[n[\binom{m+1}{2}+k-4]+mn+\binom{n}{2}](k-2)$	$\Theta(nm^2+n^2)$
$4 \leq i \leq k-2$	$[n(m+k-1-i)+mn+(i-2)\binom{n}{2}](k+1-i)$	$\Theta(nm+n^2)$
$k-1$	$[nm+mn+(k-3)\binom{n}{2}]2$	$\Theta(nm+n^2)$
k	$nm^2+\binom{n}{2}(2m+k-4)$	$\Theta(nm^2+n^2m)$

Table 4.5.: The amount of paths of value $1 \leq i \leq k$ in G_m^j.

we have

$$\gamma = \frac{\sum_{i=1}^{k} i x_i}{\sum_{i=1}^{k} x_i} = \frac{\sum_{i=1}^{k} \frac{x_i}{n^2 m} i}{\sum_{i=1}^{k} \frac{x_i}{n^2 m}}.$$

We now regard what happens to the values of $\frac{x_i}{n^2 m}$ when we let n go to infinity. For $i = 1, 2$ we get

$$\frac{x_i}{n^2 m} \xrightarrow{n \to \infty} 0.$$

For $i = 3, \ldots, k-1$ only the term from the last case we discussed plays a role, here we get that

$$\frac{x_i}{n^2 m} \xrightarrow{n \to \infty} \frac{i-2}{2m}(k+1-i).$$

Finally for the value k we have

$$\frac{x_i}{n^2 m} \xrightarrow{n \to \infty} \frac{2m+k-4}{2m} = 1 + \frac{k-4}{2m}.$$

If we let m go to infinity as well, we get that $\frac{x_i}{n^2 m} \to 0$ for all $i \neq k$ and $\frac{x_k}{n^2 m} \to 1$. Hence

$$\gamma \xrightarrow[m \to \infty]{n \to \infty} k.$$

As a result we have shown this theorem.

4.13 Theorem. For the AVAPC problem on trees we cannot get an approximation guarantee of $\gamma < k$ for Algorithm 3.4 in the case $k \geq 4$.

Now we are only missing the case $k = 3$ to complete the AVSPC problem for general graphs, and for trees we need to take a look at the case of $k = 2$ as well.

Let us start with the case where $k = 3$. Here we regard the following graph G_m which is similar to the previous subgraphs G_m^j and shown in Figure 4.14. It consists of a node v in B which has its indicator path and is adjacent to m further nodes $v_1, \ldots, v_m$ which are also in B (and consequently also have their respective indicator paths).

As in the general example, we count the paths, though it is a lot simpler in this case. There are clearly $m+1$ paths of value one and $2m$ paths of value two as each of the latter contains the node v and a node v_j which gives m possibilities with the usual multiplier of $k+1-i = 2$.

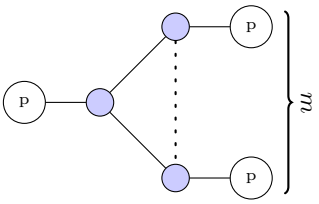


Figure 4.14.: The example graph where γ is arbitrarily close to three.

Furthermore, the paths of value three must contain v and are defined by their two endpoints amongst the v_j, so there are $\binom{m}{2}$ many. This leads to a γ-value of

$$\gamma = \frac{m+1+4m+3\binom{m}{2}}{m+1+2m+\binom{m}{2}} = \frac{3+\frac{7m+2}{m^2}}{1+\frac{5m+2}{m^2}} \xrightarrow{m\to\infty} 3.$$

We have now shown everything needed to complete the previous section.

4.14 Corollary. For the AVAPC problem on trees we cannot get an approximation guarantee of $\gamma < k$ for Algorithm 3.4 in the case $k \geq 3$.

This just leaves the case where $k = 2$ on trees. Here the result is, in fact, better as we claim that $\gamma \leq 3/2$ and again give an example where the value of γ is arbitrarily close to this bound. For the first part notice that a path with two vertices in a graph G just corresponds to an edge, so the paths of value two are just the edges in the graph induced by B, $G[B]$. As G was a tree, the induced graph must be a forest and as such it has at most $|B|-1$ edges giving us at most that many paths of value two. Because every node in B has an indicator path, we get at least $|B|$ paths of value one. Putting these two observations together gives us that

$$\gamma = \frac{x_1+2x_2}{x_1+x_2} \leq \frac{|B|+2(|B|-1)}{|B|+|B|-1} = \frac{3|B|-2}{2|B|-1} \xrightarrow{|B|\to\infty} \frac{3}{2}$$

where x_i denotes the amount of paths of value i. Note that the inequality holds as we can choose x_2 to be maximal as it only increases the value of the fraction, whereas x_1 can be chosen to be minimal as higher values decrease it because $\gamma > 1$.

The promised example is now clear, we just create the worst case situation described above. Just consider a path P_m of length $m-1$ where every node is in B and has an edge to a node not in B, to create an indicator path. From this we get the above γ-value, so $\gamma = \frac{3m-2}{2m-1}$ which converges to $3/2$ as required. So we have finished the case of trees and proved the lemma below.

4.15 Lemma. For the AVAPC problem on trees we get an approximation guarantee of $\gamma = 3/2$ for Algorithm 3.4 in the case $k = 2$.

5. Solving Path Covers Optimally

In this chapter we want to see in which situations the path cover problems can be solved to optimality. Note that in the case of paths and trees there is no difference between all paths and shortest paths. In the first section we prove that several of our stated problems are, in fact, $\mathcal{NP}$-hard. Afterwards we give an algorithm that solves the problem optimally on a path, giving us an optimal solution for the TVSPC problem as claimed in the last chapter. We then take a look at the case of trees where we give another algorithm that calculates an optimal solution in the AVAPC case. The results we have found are shown below in Table 5.1.

Problem	**Graph Type**					
	General Graph		Tree		Path	
AVAPC	$\mathcal{NP}$-hard	(5.1)	$\mathcal{P}$	(5.11)	$\mathcal{P}$	(5.9)
AVSPC	$\mathcal{NP}$-hard	(5.4)	$\mathcal{P}$	(5.11)	$\mathcal{P}$	(5.9)
TVAPC	$\mathcal{NP}$-hard	(5.5)	$\mathcal{P}$	(5.9)	$\mathcal{P}$	(5.9)
TVSPC	$\mathcal{P}$	(5.9)	$\mathcal{P}$	(5.9)	$\mathcal{P}$	(5.9)
$\mathcal{V}$-APC	$\mathcal{NP}$-hard	(5.3)	$\mathcal{NP}$-hard[1]	(5.7)	$\mathcal{P}$	(5.10)
$\mathcal{V}$-SPC	$\mathcal{NP}$-hard	(5.4)	$\mathcal{NP}$-hard[1]	(5.7)	$\mathcal{P}$	(5.10)

Table 5.1.: Complexity of the different path cover problems.

5.1. $\mathcal{NP}$-hard Path Cover Problems

We begin by showing that one of our most general problems is $\mathcal{NP}$-hard.

5.1 Theorem. The AVAPC problem on general graphs is $\mathcal{NP}$-hard.

Proof. To prove this we reduce the vertex cover problem to the k-AVAPC problem. For $k = 2$ these problems already coincide. For $k > 2$ let $(G = (V, E), x)$ be an instance of vertex cover, where we want to cover all edges by choosing at most x vertices. We create an instance of the k-AVAPC problem by simply adding an edge from every vertex to a path containing $k - 2$ nodes, giving each one something similar to an indicator path. The reduction also works in the case $k = 2$, in which the graph remains unchanged. We give the nodes on each of these new paths a weight of $x + 1$ and the nodes in G a weight of one. Then there exists a vertex cover with at most x nodes if and only if there exists a path cover of weight at most x in the AVAPC instance. This holds as a vertex cover of weight x translates to a solution of the new instance by just picking the same vertices. A solution in the augmented graph of weight at most x cannot contain any of the new nodes. It forms a vertex cover as any edge together with the "indicator path" of one of its endpoints form a path containing k nodes that is hit. So we get that the k-AVAPC problem is $\mathcal{NP}$-hard on general graphs. □

[1] These problems are $\mathcal{NP}$-hard in the case $k \geq 3$.

5.2 Remark. Notice that the above proof makes use of a weight function to artificially make vertices, which we do not want to select, too expensive to be part of a solution. However, with a bit more effort, we can remove this and actually show that the problem is already $\mathcal{NP}$-hard in the case of uniform weights ($c \equiv 1$). We can do this by replacing the "indicator path" with expensive nodes just by $x + 1$ paths with uniform costs. As a node must be selected from every path, we have to select at least $x+1$ nodes if we do not select one of those corresponding to a node in G. So the problem is already hard for uniform weights.

As the AVAPC problem is just a special case of the $\mathcal{V}$-APC problem, we get the following corollary.

5.3 Corollary. The $\mathcal{V}$-APC problem on general graphs is $\mathcal{NP}$-hard.

In fact, we get even more:

5.4 Corollary. The AVSPC and $\mathcal{V}$-SPC problems on general graphs are $\mathcal{NP}$-hard.

Proof. Notice that the above reduction also works for the AVSPC problem (assuming that the costs on the edges fulfil the triangle inequality, which we can guarantee by just choosing $d \equiv 1$). Consequently the AVSPC and $\mathcal{V}$-SPC problems on general graphs are $\mathcal{NP}$-hard as well. □

Next we prove that the TVAPC problem on general graphs is also $\mathcal{NP}$-hard by another reduction of vertex cover.

5.5 Theorem. The TVAPC problem on general graphs is $\mathcal{NP}$-hard.

Proof. As claimed we will do this by a reduction of vertex cover. However, we begin by reducing it to the 4-TVAPC problem. Again let $(G = (V, E), x)$ be an instance of vertex cover. We create an instance (G', x, s, t) of the 4-TVAPC problem by adding two new nodes s and t to G and an edge from all nodes in G to these two. So $G' = (V', E')$ with $V' = V \cup \{s, t\}$ and $E' = E \cup \{(s, v) \mid v \in V\} \cup \{(v, t) \mid v \in V\}$. Let $c(v) = 1$ for all $v \in V$ and $c(s) = c(t) = x + 1$. Then the instance of vertex cover has a solution of at most x nodes if and only if the instance of the 4-TVAPC problem has a solution of cost at most x. This holds as we can choose the nodes in the vertex cover as a solution of the 4-TVAPC problem: Any subpath of an (s, t)-path consisting of four nodes must contain an edge in E and is consequently hit by the vertex cover. For the converse direction we notice that a solution of the 4-TVAPC problem of cost at most x cannot contain s or t. So we can choose the same nodes for the vertex cover problem and they form a vertex cover as for any edge $e = (u, v) \in E$ we get a path $P_e = (s, u, v, t)$ containing four nodes, which must be hit. As neither s nor t is selected, either u or v must have been.

Now we need to generalise this to values of $k \neq 4$. For $k \geq 4$ this is very easy to do as we can simply add another $k - 4$ additional nodes to the graph which effectively replace t by a path of length $k - 4$, the last node of which is the new node t. The reduction works in the same way on this graph if we set the value of all new nodes to be $x + 1$ as before.

For the case $k = 2$ (see Figure 5.1) we add two more nodes s_0 and t_0 to G' which are situated between s or t and the graph. More precisely, s is adjacent to s_0 which is connected to all

nodes in G and the analogous holds for t and t_0 respectively. By setting the cost of the nodes s_0 and t_0 to zero we get that a solution of vertex cover corresponds to a solution of the 2-TVAPC problem by additionally choosing s_0 and t_0 and vice versa. This is true as a solution of vertex cover is clearly a solution in the new graph as all additional edges are incident to s_0 or t_0, both of which are chosen. Additionally, a solution of the 2-TVAPC problem in our graph must select a node incident to any edge in G as every such edge is on an (s, t)-path, meaning an endpoint must be hit.

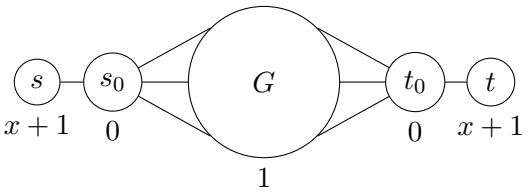


Figure 5.1.: Graph used in the reduction for the 2-TVAPC problem.

The case $k = 3$ then works by combining the two techniques, as in we only add the node s_0 but not t_0. Here a vertex cover together with s_0 is still a solution and every solution must be a vertex cover as any edge $e = (u, v) \in E$ is part of the path $P_e = (s, s_0, u, v, t)$, so the subpath $P'_e = (u, v, t)$ must be hit and either u or v selected. □

5.6 Remark. As for the previous proof, we want to point out that even though this proof makes use of the weights assigned to nodes, it is, in fact, unnecessary. A slight change to the reduction makes it work for uniform node weights and actually reduces the amount of case distinctions that are required. For the case $k \geq 3$ we use a graph similar to the one we used for $k \geq 4$ in the reduction above. The node s is connected to all nodes in the graph and the node t is now connected to $x + 2$ paths containing $k - 2$ nodes. As before the increase in the amount of paths replaces the artificially high weights. In this situation a vertex cover G of cost x corresponds to a path cover in G' of cost $x + 1$.

To see this let P_i be the paths containing $k - 2$ nodes for $i = 1, \ldots, x + 2$. Assume we have a path cover of cost $x + 1$. Then for every edge $e = (u, v) \in E$ we get $x + 2$ paths (u, v, P_i) that have to be hit, hence either u or v must be selected. Moreover, unless all nodes in G are selected, we must select t. This is the case as if $v \in V$ is not selected we get $x + 2$ paths (v, P_i, t) and we must pick t. If all nodes in G are selected, this is clearly a path cover and $x+1 \geq n$. Consequently there must be a vertex cover of at most $x \geq n-1$ nodes (just remove an arbitrary node). If we are given a vertex cover, we just select the same x nodes and t. Any path of length k containing an edge in G is hit. The only paths in the graph that contain no edge in G are the paths (v, P_i, t) for some $v \in V$ and $i \in \{1, \ldots, x + 1\}$. These are hit as well as t is selected.

Notice that the case $k = 2$ was not covered above. For this we again use the graph shown in 5.1, this time with uniform weights. We claim that in this case a vertex cover of G of cost x corresponds to a path cover of G' of cost $x + 2$. This holds as selecting the nodes in a vertex cover as well as s_0 and t_0 clearly results in a path cover. Conversely, given a path cover, the nodes in G that are selected form a vertex cover as any edge is part of an (s, t)-path and must be hit. Also at most x nodes in G are selected as either s or s_0 and t or t_0 must be chosen, so we get a vertex cover of x vertices.

The only problem missing that is still stated to be $\mathcal{NP}$-hard is the general path cover problem on trees. We show that this is already hard on star graphs for $k = 3$ and generalise the result to larger k, thus completing this section. For these larger k we need trees, but they are very similar to the star graph. Note that we have to assume that $k > 2$ as for $k = 2$ the problem is just identical to vertex cover on the forest that results from removing the edges that do not need to be hit from the tree. The vertex cover problem on trees is easy to solve (as we will also see in Section 5.3) and the subtrees can be solved independently.

5.7 Theorem. The $\mathcal{V}$-APC and $\mathcal{V}$-SPC problems on trees are $\mathcal{NP}$-hard for $k \geq 3$.

Proof. As already stated, these two problems coincide on trees. To show that the $\mathcal{V}$-APC problem on a tree is $\mathcal{NP}$-hard we (again) give a reduction from the vertex cover problem to this one. For $k = 3$ let $(G = (V, E), x)$ be an instance of vertex cover. We construct a star graph $G' = (V \cup \{c\}, E')$ where c is a new node, the centre node, and $E' = \{(c, v) \mid v \in V\}$. This gives us the star shown in Figure 5.2.

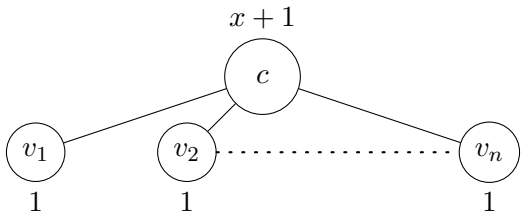


Figure 5.2.: Star graph used in the reduction for the 3-$\mathcal{V}$-APC problem.

We define the costs of all nodes except c to be one and set the cost of c to $x + 1$. Finally we choose the set $\mathcal{V}$ to be

$$\mathcal{V} = \{(u, v) \mid (u, v) \in E\}.$$

We now claim that there exists a vertex cover of size at most x if and only if there exists a solution to the $\mathcal{V}$-APC problem of cost at most x. A vertex cover translates to a path cover as, by picking the corresponding vertices in our star graph, any path specified by $\mathcal{V}$ is hit because one endpoint is part of the vertex cover. Conversely, a solution to the path cover problem of cost at most x cannot select c, so we can translate it to a set of nodes in the original graph. These indeed form a cover as an edge $(u, v) \in E$ directly corresponds to the path $P_{uv} = (u, c, v)$ containing three nodes in G' which by the definition of $\mathcal{V}$ must have been hit.

For $k > 3$ this can be generalised as follows: If k is odd, the same reduction can be done by introducing a path containing $\frac{k-1}{2}$ nodes instead of the single node from before (which in the case $k = 3$ is also the same as such a path) and setting all their costs to $x + 1$ as well. For even k we need to be a bit more careful. In this case the idea is to also introduce paths, but, as they cannot be evenly divided, we choose the shorter length for all nodes and introduce an additional dummy node d after each node v.

Formally let $(G = (V, E), x)$ be an instance of vertex cover. Then we construct the graph G' shown in Figure 5.3. We choose the set $\mathcal{V}$ to be

$$\mathcal{V} = \{(d_i, d_j) \mid (v_i, v_j) \in E\}.$$

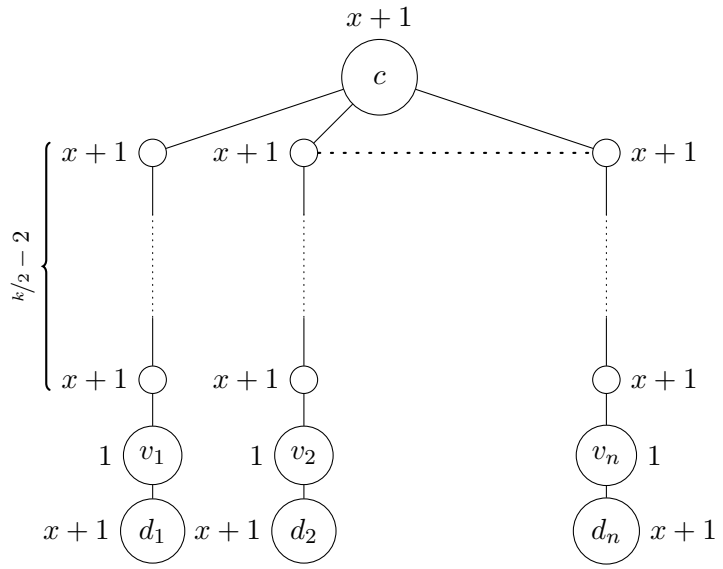


Figure 5.3.: Generalised graph used in the reduction for the k-$\mathcal{V}$-APC problem for even k.

There are exactly two paths of length k corresponding to such a pair (d_i, d_j) in the graph, namely the one starting at d_i and going to v_j via v_i and the one starting at v_i and ending at d_j, consequently passing the node v_j. Clearly, both are hit if either the node v_i or v_j are selected, so a vertex cover of cost at most x corresponds to a path cover of the same cost. The converse is again true as a solution to the path cover problem for the given $\mathcal{V}$ only contains nodes corresponding to nodes in v and still translates to a vertex cover as for any edge (v_i, v_j) the path from v_i to d_j must be hit. By the fact that all nodes on this path aside from v_i and v_j have cost $x + 1$, one of these two must have been selected, completing the reduction. □

We end with a final remark.

5.8 Remark. Notice that the above reduction relies heavily on the fact that we have a weighted path cover problem. Indeed, the problem on all the above graphs (and star graphs for $k = 3$) is actually trivial in the case of constant costs as the node c is a cover by itself, regardless of the choice of $\mathcal{V}$. We will take a look at this case on general trees in the next chapter and show that this is already a restriction that makes the problem easy to solve.

5.2. Path Cover Problems on a Path

Here we begin by giving an algorithm that solves the AVAPC (and consequently AVSPC) problem optimally on a path, letting us also solve the TVSPC problem optimally on general graphs as well as the TVAPC problem on paths and trees. Assume we are given the path P which contains the nodes $v_1, \ldots, v_l$, then this task is particularly simple if we have uniform costs for the nodes as in the previous chapter. In this case we can just choose every k-th node to get an optimal solution as any feasible solution contains at least $\lfloor l/k \rfloor$, the same amount that we get by just picking every k-th node. For an arbitrary non-negative cost function $c\colon V \to \mathbb{R}_+$ this will not work any more, but we can find the optimal solution by dynamic

programming. Notice that a hitting set must contain one of the first k nodes of the path. If it contains the node i, the subpaths that still need to be hit are exactly the subpaths of the path beginning at node $i+1$. So let P_i denote the subpath of P starting at node i and let $c(G)$ denote the cost of the optimal hitting set of the graph G, then $c(P) = c(P_1)$ and in general

$$c(P_i) = \begin{cases} \min\{c(v_{i+j}) + c(P_{i+j+1}) \mid j = 0, \dots, k-1\}, & \text{if } \ i \leq l-k+1, \\ 0, & \text{otherwise.} \end{cases}$$

This means that we can start with $c(P_{l+1}) = c(P_l) = \ldots = c(P_{l-k+2}) = 0$ and iteratively calculate $c(P_i)$ for $i = l-k+1, \dots, 1$, giving us the following result.

5.9 Lemma. The AVAPC and AVSPC problems on a path are in $\mathcal{P}$. So is the TVSPC problem on general graphs (and consequently on paths and trees). Additionally, the TVAPC problems on paths and trees are also solvable in polynomial time. In fact, the above algorithm allows us to solve these problems in linear time.

We complete this part by generalising the above procedure to an algorithm for the $\mathcal{V}$-APC (and $\mathcal{V}$-SPC) problem. To do so notice that on a path the sets to be hit are characterised by their start nodes. To start off we calculate the set S of such nodes, that is

$$\begin{aligned} S &= \{i \mid \text{the path } (i, i+1, \dots, i+k-1) \text{ must be hit}\} \\ &= \{i \mid \exists (s,t) \in \mathcal{V} : s \leq i \leq i+k-1 \leq t\}. \end{aligned}$$

We define $m(X) \coloneqq \min X$ and $S_i \coloneqq S \backslash \{1, \dots, i\}$ then clearly

$$c(P) = c\left(P_{m(S)}\right) = c\left(P_{m(S_0)}\right).$$

Similar to before we also get that

$$c(P_i) = \begin{cases} \min\left\{c(v_{i+j}) + c\left(P_{m(S_{i+j})}\right) \mid j = 0, \dots, k-1\right\}, & \text{if } \ i < \infty, \\ 0, & \text{otherwise.} \end{cases}$$

With this algorithm we have now seen that at least on a path the path cover problem is easy to solve and have proved the next theorem.

5.10 Theorem. The $\mathcal{V}$-APC and $\mathcal{V}$-SPC problems on a path are solvable in polynomial time.

To conclude this section we present another way of showing that we can solve path cover problems on a path. To do so we write the path cover problem as an integer program as already seen in Section 3.2:

$$\begin{aligned} (IP) \quad \min \quad & \sum_{v \in V} c_v x_v \\ & \sum_{v \in P} x_v \geq 1, \qquad P \in \mathscr{P}, \\ & x_v \in \mathbb{B}, \qquad v \in V. \end{aligned}$$

Here $\mathscr{P}$ denotes the set of paths that must be hit in the $\mathcal{V}$-APC problem. This is the same as the problem

$$\begin{aligned} (IP) \quad \min \quad & c^T x \\ & A\,x \geq e, \\ & x \in \mathbb{B}^V \end{aligned}$$

where e denotes the all-one vector and the matrix A is a $(|\mathscr{P}| \times n)$-matrix with an entry of one at $a_{P,v}$ if the node v is contained in the path P and zero everywhere else. As we are on a path, we can sort the nodes in V in the order in which they occur on the path. Then the ones in every row of A appear consecutively. By Theorem 4.18 in [Kru15] we get that the matrix A^T is totally unimodular and consequently (by Observation 4.10) A is also totally unimodular. It follows by the Integrality-Theorem of Hoffmann and Kruskal (Corollary 4.12) that the polyhedron $P = \{x \mid Ax \geq e, x \geq 0\}$ is integral and we can solve the problem optimally by solving the LP-relaxation. Note that Corollary 4.12 uses a slightly different polyhedron, namely it uses $P = \{x \mid Ax \leq e, x \geq 0\}$. However, replacing $Ax \geq e$ by $-Ax \leq -e$ brings our polyhedron into this form and the proof of the Consecutive Ones Theorem also works for a matrix with consecutive negative ones.

5.3. The AVAPC Problem on a Tree

Unlike paths the case of a tree is not quite so simple, but again we want to find the optimal solution with the help of dynamic programming. This will give us an algorithm for the AVAPC and AVSPC problem on trees. To this end let T be a tree with a non-negative cost function c on the vertices as before. We now root T at an arbitrary vertex v. Let its subtrees be the trees T_i rooted at v_i for $i = 1, \ldots, s$, giving us the situation shown in Figure 5.4.

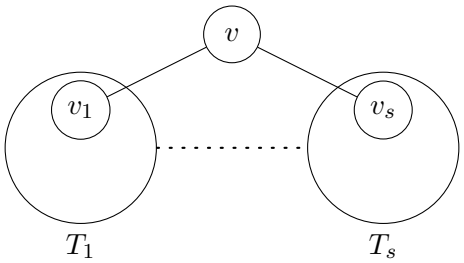


Figure 5.4.: The tree T rooted at v.

Notice that for an optimal path cover we have two options: It either contains the node v or it does not. The first case is particularly nice as it reduces the problem to that of finding optimal path covers for the subtrees. This is true as if the cover contains the node v the paths that still need to be hit are those contained in one of the subtrees. This lets us straightforwardly reduce the problem to the solving of subproblems.

The second case, where v is not part of the cover, is more problematic as it does not directly allow the transition to the subtrees. However, a path containing k vertices and the node v must contain nodes from one or two subtrees. As every path with k vertices must be hit, we cannot find two subpaths P_i in T_i and P_j in T_j which start at the node v_i and v_j respectively

that contain only nodes that are not part of the cover and fulfil $d_i + d_j \geq k - 1$. Here d_i denotes the amount of nodes on the path P_i and d_j the amount on P_j. If such paths did exist, they would form a violated set when connected by the node v. So for any two such paths we get that $d_i + d_j < k - 1$. We want to make use of this observation in order to get an algorithm for this problem.

To this end let $c(T, v, x)$ denote the cost of an optimal path cover in the tree T rooted at v with the restriction that any path starting at the node v containing only nodes not included in the path cover has at most x nodes. Clearly, the optimal cover we are looking for has the cost $c(T, v, k - 1)$. Now we just need to figure out how we can calculate these values.

Notice that if T consists only of its root node v that

$$c(T, v, x) = \begin{cases} 0, & \text{if } x \neq 0, \\ c(v), & \text{if } x = 0. \end{cases} \tag{5.1}$$

The next step will be to find a formula that calculates $c(T, v, x)$ from the values $c(T_i, v_i, x')$ of its subtrees. If the optimal hitting set contains the node v, we get that

$$c(T, v, x) = c(v) + \sum_{j=1}^{s} c(T_j, v_j, k - 1) \tag{5.2}$$

as described before (the additional condition for x is always satisfied here). We generalise our previous observation for the second case. If v is not part of the path cover that fulfils the required condition for the value x, we can look at the paths in the subtrees which start at the root but contain no nodes of the cover. Let d_i denote the amount of nodes on the longest such path in the subtree T_i. This situation is displayed in Figure 5.5 and obviously the inequality $d_i + d_j < k - 1$ must still be fulfilled as well as $d_i \leq x - 1$.

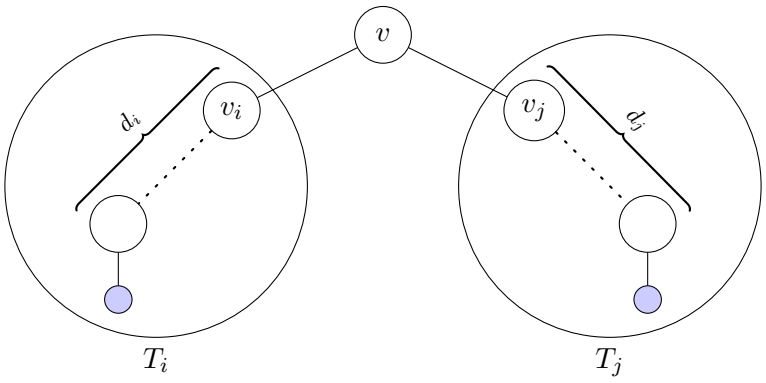


Figure 5.5.: The situation when v is not selected.

Notice that only one subtree can contain paths as described above with $a \coloneqq \left\lceil \frac{k-1}{2} \right\rceil$ or more nodes as otherwise we can find two subtrees such that $d_i + d_j \geq 2a \geq k - 1$. On the other hand, if all trees fulfil $d_i < a$, then $d_i + d_j \leq 2(a - 1) \leq 2\frac{k}{2} - 2 = k - 2 < k - 1$. This means that if $x - 1 < a$ we get

$$c(T, v, x) = \sum_{j=1}^{s} c(T_j, v_j, x - 1). \tag{5.3}$$

This holds as there cannot be a violated path containing v for any feasible solution of the subtrees as the condition for $x-1$ implies that any subpath beginning at the root contains less than a nodes not in the cover.

The only remaining case is the one where v is not in the path cover and $x-1 \geq a$. We stated before that only one subtree can have a value d_i that is greater than or equal to a. However, we need to know which or whether it is none of them, making this situation the most complicated one. Assume that in the optimal solution we have $a \leq d_i \leq x-1$, then $d_j \leq k-2-d_i < a \leq x-1$ and we can calculate $c(T,v,x)$ as follows:

$$c(T,v,x) = c(T_i,v_i,d_i) + \sum_{j\neq i} c(T_j,v_j,d_j) = c(T_i,v_i,d_i) + \sum_{j\neq i} c(T_j,v_j,k-2-d_i). \tag{5.4}$$

The last equality follows from the fact that $c(T_j,v_j,d_j) \geq c(T_j,v_j,k-2-d_i)$ as the latter is less restrictive and consequently permits more feasible solutions, but our solution was optimal. Moreover, it does not cause any infeasible solutions as $k-2-d_i < a \leq x-1$, so it fulfils all the requirements for the d_j. The last possibility is that all values d_j are at most $a-1$. Then we get

$$c(T,v,x) = \sum_{j=1}^{s} c(T_j,v_j,d_j) = \sum_{j=1}^{s} c(T_j,v_j,a-1). \tag{5.5}$$

Again this holds as $c(T_j,v_j,d_j) \geq c(T_j,v_j,a-1)$ and our solution is optimal. Furthermore, our solution remains feasible as $a-1$ is less than $x-1$ and a.

Combining all that we have shown so far, we get that we can initialise $c(T,v,x)$ as in equation (5.1) if T consists only of the node v. If this is not the case and we have $x-1 < a$, we get that

$$c(T,v,x) = \min\left\{ c(v) + \sum_{j=1}^{s} c(T_j,v_j,k-1), \sum_{j=1}^{s} c(T_j,v_j,x-1) \right\} \tag{5.6}$$

by combining the two possibilities from equations (5.2) and (5.3). If $x-1 \geq a$, we could be in the case where equation (5.2) or (5.5) applies. We could also be in the case that equation (5.4) applies for some subtree and some value $a \leq d_i \leq x-1$. For the sake of completeness, we note that this case can only occur if the value a is possible. This is the case as long as $a+1 \leq k-1$. Rearranging the inequality shows that this holds for $k \geq 3$. This leads to the formula (for $k \geq 3$):

$$c(T,v,x) = \min\left\{ \begin{matrix} c(v) + \sum_{j=1}^{s} c(T_j,v_j,k-1),\ \sum_{j=1}^{s} c(T_j,v_j,a-1), \\ c(T_i,v_i,d) + \sum_{j\neq i} c(T_j,v_j,k-2-d) \end{matrix} \middle| \begin{matrix} 1 \leq i \leq s, \\ a \leq d \leq x-1 \end{matrix} \right\}. \tag{5.7}$$

In the case $k=2$ the above minimum simply needs to be modified by removing the possibilities corresponding to equation (5.4).

Armed with the equations (5.1), (5.6) and (5.7), we can now calculate the value of an optimal solution. First of all we notice that the tree T_i is uniquely determined by its root v_i if we know T and v. Given our tree T, we first root it at an (arbitrary) node v and perform a breadth-first search starting at v. Let $v_1, v_2, \ldots, v_n$ be the nodes in the order that they are visited by the search, so $v_1 = v$. We then create a matrix A with n rows and k columns in which row i corresponds to node v_i (and the subtree rooted at this node) and column j

corresponds to an x-value of $j-1$. We wish to enter the value $c(T_i, v_i, j-1)$ into the matrix A at position (i, j).

We do this by filling the table from the bottom to the top. By the fact that we did a breadth-first search to order the nodes, the node v_n must be a leaf of the tree. Hence we can initialise the last row with the help of equation (5.1). Now, if we are in row i and wish to calculate $c(T_i, v_i, x)$, there are three options. If v_i is a leaf, we can again use equation (5.1). Otherwise, we need to use either equation (5.6) or (5.7), depending on the value of x. However, both of these can be calculated by using the values of their subtrees which are already contained in A as their roots have a higher value due to the fact that a breadth-first search visits them later.

We take a quick look at the running time. We need to fill in nk entries of the table. If we are in the leaf case, this can be done by a single comparison in constant time. In the case for low values of x we need $\mathcal{O}(s)$ many additions, where s is the amount of subtrees. If x is large, we need $\mathcal{O}(s^2k)$ many additions as $x-1-a+1 \leq k$. Using the somewhat brute force estimation $s \leq n$ we get a total running time of $\mathcal{O}(n^3k^2)$.

It should be noted that this method most likely calculates way more than is actually necessary, a recursive procedure that only computes the values that are needed might be more efficient. It would check the matrix to see whether the required value is already available and only calculate it should this not be the case. Regardless of the exact implementation we have proved the theorem below.

5.11 Theorem. The AVAPC and AVSPC problems on trees is solvable in polynomial time.

6. Further Results for Path Covers on Trees

6.1. Bad Properties

After we could use the fact that the constraint matrix of the $\mathcal{V}$-APC problem is totally unimodular nicely in the case of paths, we want to give an example of a tree where this is not the case. The example graph is shown in Figure 6.1.

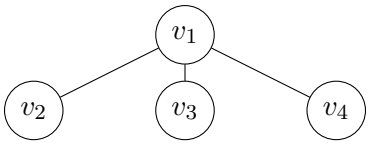


Figure 6.1.: Example of a tree where the constraint matrix is not totally unimodular.

The constraint matrix in the case of $k = 3$ has the form

$$A = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}.$$

The nodes are sorted in the same way as they are numbered and the paths with three nodes are just the paths $P_1 = (v_2, v_1, v_3)$, $P_2 = (v_2, v_1, v_4)$ and $P_3 = (v_3, v_1, v_4)$. The sub-matrix corresponding to the nodes v_2, v_3, v_4 has a determinant of -2 and consequently A is not totally unimodular.

Even worse, if we let the weight of v_1 be sufficiently large, say two, and assign a weight of one to the remaining three nodes, we get that an optimal solution to the problem has weight two (and either selects v_1 or two of the other nodes). However, the solution that chooses $x_{v_1} = 0$ and $x_{v_2} = x_{v_3} = x_{v_4} = 1/2$ is also feasible with a weight of $3/2 < 2$. So the polyhedron is actually not even integral.

6.2. The $\mathcal{V}$-APC Problem on Trees with Bounded Frequencies

In this section we want to illustrate a case in which the $\mathcal{V}$-APC problem can be solved in polynomial time on a tree. The results shown here are based on Section 4 of [KMP+02]. We begin by defining what is actually meant by the word *frequency* in this section's title.

6.1 Definition. Let $(G = (V, E), k, \mathcal{V})$ be an instance of the $\mathcal{V}$-APC problem. As before let $\mathscr{P}$ denote the set of paths that must be hit in this instance. For a node $v \in V$ we write

$$\phi_v := |\{P \in \mathscr{P} \mid v \in P\}|$$

for the frequency of v. By $\phi(\mathscr{P})$ we denote the maximum frequency of $\mathscr{P}$, that is

$$\phi(\mathscr{P}) \coloneqq \max_{v \in V} \phi_v.$$

The aim is now to show that, if the maximum frequency $\phi(\mathscr{P})$ is bounded, we can solve the $\mathcal{V}$-APC problem in polynomial time, or more precisely:

6.2 Theorem. Let $(G, k, \mathcal{V})$ be an instance of the $\mathcal{V}$-APC problem where $\mathscr{P}$ is the set of paths to be hit and $\phi(\mathscr{P}) \leq f$ for some constant $f \in \mathbb{N}$. Furthermore, let the costs c_v of the vertices be polynomially bounded, so $c_v \in \mathcal{O}(p(|V|))$ for some polynomial p. Then we can solve this instance of the $\mathcal{V}$-APC problem in polynomial time.

To prove this theorem we first need the notion of a *tree-decomposition*.

6.3 Definition. A tree-decomposition of a graph $G = (V, E)$ is a pair $D = (S, T)$ where $S = \{X_i \mid i \in I\}$ is a collection of subsets of V and T is a tree with node set isomorphic to S such that the following three conditions are satisfied:

(i) $\bigcup_{i \in V(T)} X_i = V$,
(ii) for all edges $(u, v) \in E$, there exists a subset $X_i \in S$ containing both vertices u and v,
(iii) for each vertex $v \in V$ the set of nodes $\{i \mid v \in X_i\}$ forms a subtree of T.

The width of the tree-decomposition D is defined to be $\max_{i \in I} |X_i| - 1$. The tree-width $\mathrm{tw}(G)$ of a graph G is the minimum width of a tree-decomposition of G.

We also need to define the *constraint graph* and the *interaction graph* for a given (for example integer or linear) program.

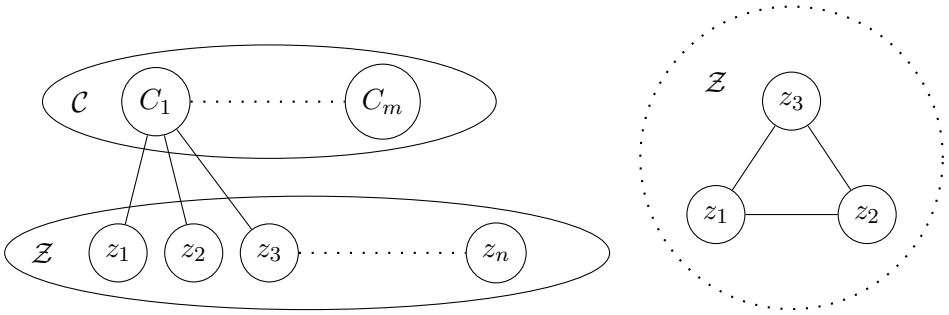
6.4 Definition. Let $\mathcal{Z}$ be a set of variables and $\mathcal{C}$ be a set of constraints on $\mathcal{Z}$.

(i) The constraint graph $CG(\mathcal{Z}, \mathcal{C})$ associated with $(\mathcal{Z}, \mathcal{C})$ is the bipartite graph consisting of a node for every $z \in \mathcal{Z}$ and every constraint $C \in \mathcal{C}$ where two nodes z and C are connected if and only if z appears in C.
(ii) The interaction graph $IG(\mathcal{Z}, \mathcal{C})$ for $(\mathcal{Z}, \mathcal{C})$ is the graph with vertex set $\mathcal{Z}$ and an edge from z_1 to z_2 if and only if they have a common neighbour in $CG(\mathcal{Z}, \mathcal{C})$, that is if they are both contained in the same constraint.

Before we go any further let us quickly illustrate these last two definitions with a small example. Assume $\mathcal{Z} = \{z_1, z_2, z_3, \ldots, z_n\}$ and $\mathcal{C} = \{C_1, \ldots, C_m\}$ with $C_1 = z_1 + z_2 + z_3 \leq 1$. Then the constraint graph has an edge from C_1 to z_1, z_2 and z_3 and to no other nodes as shown in Figure 6.2(a). Consequently in the interaction graph these three nodes form a clique as shown in Figure 6.2(b).

Recall that we formulated the following integer linear program for the $\mathcal{V}$-APC problem:

$$\begin{aligned} (IP) \quad \min \quad & \sum_{v \in V} c_v x_v \\ & \sum_{v \in P} x_v \geq 1, \qquad P \in \mathscr{P}, \\ & x_v \in \mathbb{B}, \qquad v \in V. \end{aligned}$$



(a) The constraint graph. (b) The interaction graph.

Figure 6.2.: Illustration of the constraint and interaction graph.

As we will need a maximisation program in the following that still has non-negative coefficients, we need to adjust our formulation. This can be done by reversing the meaning of the x_v. We have

$$x_v \coloneqq \begin{cases} 1, & \text{if node } v \text{ is not chosen,} \\ 0, & \text{otherwise.} \end{cases}$$

We want to maximise the costs of the nodes that are not part of the path cover. This yields the formulation

$$\begin{aligned} (IP') \quad \max \quad & \sum_{v \in V} c_v x_v \\ & \sum_{v \in P} x_v \leq k-1, \qquad P \in \mathscr{P}, \\ & x_v \in \mathbb{B}, \qquad\qquad v \in V. \end{aligned}$$

Notice that the constraint has changed from requiring at least one of the x_v to be set to one (and the path to be hit) to at most $k-1$ of the x_v are one. This in turn means that one of the x_v must be zero and the path is hit. In consideration of the above formulation we call the nodes of the corresponding constraint graph x_v or x_P respectively and denote it by $\mathrm{IG}(IP')$.

We now cite two more theorems also used in [KMP+02] which we need to prove Theorem 6.2. The first one can be found in [SI96], the second in [HMR+02].

6.5 Theorem. Let p be a polynomial. Let $\mathcal{Z}$ be a set of variables taking values from the domain $\{0, \ldots, K\}$, where $K \in \mathcal{O}(p(|\mathcal{Z}|))$, and let $\mathcal{C}$ be a set of constraints on $\mathcal{Z}$. Then, for any fixed $k \in \mathbb{N}$ and any non-negative vector $c = (c_z)_{z \in \mathcal{Z}} \in \{0, \ldots, p(|\mathcal{Z}|)\}^{\mathcal{Z}}$, the integer program of maximising $\sum_{z \in \mathcal{Z}} c_z \cdot z$ subject to the constraints $\mathcal{C}$, restricted to those instances where the interaction graph for $(\mathcal{Z}, \mathcal{C})$ has bounded tree-width at most k, can be solved in time $K^{O(k)}$.

6.6 Theorem. Let $\mathcal{Z}$ be a set of variables and $\mathcal{C}$ be a set of constraints on $\mathcal{Z}$. Suppose that each constraint contains at most k variables. Let $\mathrm{CG}(\mathcal{Z}, \mathcal{C})$ be the constraint graph associated with $(\mathcal{Z}, \mathcal{C})$ and $\mathrm{IG}(\mathcal{Z}, \mathcal{C})$ be the interaction graph. Then

$$\mathrm{tw}(\mathrm{IG}(\mathcal{Z}, \mathcal{C})) \in \mathcal{O}(k \cdot \mathrm{tw}(\mathrm{CG}(\mathcal{Z}, \mathcal{C}))).$$

The first theorem lets us solve an integer program whose interaction graph has a bounded tree-width in polynomial time if the coefficients of the objective function and the values of the variables are bounded by a polynomial. We assumed that the first is the case and the latter clearly holds for binary variables.

The second theorem states that the tree-width of the interaction graph is already bounded if the tree-width of the constraint graph is bounded and each constraint contains at most a constant amount of variables. In our case the second part is obviously true as every constraint has exactly k variables, one for each node on the path.

What remains to be shown is that the constraint graph has a bounded tree-width. For this we proceed analogously to the proof of Lemma 22 in [KMP+02].

6.7 Lemma. The constraint graph of (IP') has a bounded tree-width, in fact, it has a tree-width of at most $\phi(\mathscr{P})$.

Proof. To prove the lemma we construct a tree decomposition $D = (S, T)$ with the given bound. We define the sets $X_v := \{x_v\} \cup \{x_P \mid v \in P\}$ and the set $S := \{X_v \mid v \in V\}$. Let T be the tree given by the instance of the $\mathcal{V}$-APC instance (with the set X_v corresponding to the node v). We claim that this is a tree decomposition, so we need to check the three properties:

(i) For every $v \in V$ we get $x_v \in X_v \subseteq \bigcup_{u \in V} X_u$. For each $P \in \mathscr{P}$ there exists a $v \in V$ such that $v \in P$. Consequently $x_P \in X_v \subseteq \bigcup_{u \in V} X_u$ and $\bigcup_{u \in V} X_u = V(\mathrm{CG}(IP'))$ as required.

(ii) Let (x, y) be an edge of $E(\mathrm{CG}(IP'))$. Then $(x, y) = (x_v, x_P)$ for some $v \in V$ and $P \in \mathscr{P}$ such that $v \in P$. Consequently $x_v, x_P \in X_v$ are contained in a subset.

(iii) Now let $x_i \in V(\mathrm{CG}(IP'))$. If $x_i = x_v$ for some $v \in V$, the set $\{u \mid x_v \in X_u\} = \{X_v\}$ is clearly a subtree of T. If $x_i = x_P$ for a path $P \in \mathscr{P}$, the set $\{u \mid x_P \in X_u\} = \{u \mid u \in P\}$ is a path in T and as such also a subtree. So this condition holds as well.

As a result, D is a tree decomposition. To determine the tree-width we only need to look at the maximum cardinality of a set X_v. These, however, contain at most $\phi(\mathscr{P}) + 1$ many elements, giving us a tree-width of at most $\phi(\mathscr{P})$ as claimed. □

6.3. The $\mathcal{V}$-APC Problem on Trees with Uniform Weights

In this section we show that the $\mathcal{V}$-APC (and $\mathcal{V}$-SPC) problem on trees is easy to solve for uniform weights. We begin by giving an algorithm that constructs a feasible path cover and then prove that the solution it constructs is already optimal. Afterwards we take a look at its running time. We complete this section by giving an alternative proof of this fact.

The idea of the algorithm is a very simple greedy approach: First select a random node $v \in V$ and root the tree at this node. Do this via a breadth-first search and name the nodes $v_1, \ldots, v_n$ in the order in which they are visited by the search, so $v = v_1$. Next begin with the clearly feasible cover $C = V$ containing all the nodes. The algorithm iterates over the nodes in reverse order, beginning at v_n and ending with v_1, and for every node v_i it checks whether $C \backslash \{v_i\}$ is still a feasible path cover. If this is the case, it removes the node v_i from C, otherwise C does not change. The resulting algorithm is shown in Algorithm 6.1.

Algorithm 6.1: Uniform $\mathcal{V}$-APC Algorithm

Input: An instance of the k-$\mathcal{V}$-APC problem on a tree $T = (V, E)$ with uniform weights.
Output: A feasible (and optimal) path cover C.
procedure:

```
1  begin
2      Select an arbitrary node v ∈ V;
3      Perform a breadth-first search starting at v;
4      Label the nodes v_1, ..., v_n in the order in which they are visited;
5      for i := n downto 1 do
6          if C\{v_i} is a feasible path cover then
7              C := C\{v_i};
8          end
9      end
10     return C;
11 end
```

Now for the optimality:

6.8 Theorem. The cover returned by Algorithm 6.1 is optimal.

Proof. Let C_{ALG} be the cover returned by the algorithm and C_{OPT} be an optimal path cover. We already know that C_{ALG} is a feasible cover, so it remains to show that it contains the same amount of nodes as C_{OPT}. To do so we transform C_{OPT} into C_{ALG} step by step, exchanging a selected vertex and a non-selected vertex of C_{OPT} in each step in such a way that the cover remains feasible and consequently optimal. Let $v_1, \ldots, v_n$ be the nodes as ordered by Algorithm 6.1. It clearly suffices to prove the following claim for $i = 1$:

Claim: There exists an optimal cover C_{OPT} such that C_{ALG} and C_{OPT} coincide on all nodes $\{v_i, \ldots, v_n\}$.

We prove this by induction on i beginning with $i = n+1$ and step $i \to i-1$. The induction start for $n+1$ is clear as any optimal solution does the trick. By the induction hypothesis we have an optimal solution C_{OPT} such that C_{ALG} and C_{OPT} coincide on the nodes in $\{v_{x+1}, \ldots, v_n\}$ and $i = x$. If $v_x \in C_{\mathrm{OPT}}$ and $v_x \in C_{\mathrm{ALG}}$ or $v_x \notin C_{\mathrm{OPT}}$ and $v_x \notin C_{\mathrm{ALG}}$, the claim for x follows.

Assume that $v_x \in C_{\mathrm{ALG}}$ and $v_x \notin C_{\mathrm{OPT}}$. In this case, as v_x is selected by the algorithm, we know that the set $(C_{\mathrm{ALG}} \cap \{v_{x+1}, \ldots, v_n\}) \cup \{v_1, \ldots, v_{x-1}\}$ is not a feasible cover. Consequently, as $v_x \notin C_{\mathrm{OPT}}$, we get

$$C_{\mathrm{OPT}} \subseteq \{v_1 \ldots, v_{x-1}\} \cup (C_{\mathrm{OPT}} \cap \{v_{x+1}, \ldots, v_n\}) \overset{IH}{=} (C_{\mathrm{ALG}} \cap \{v_{x+1}, \ldots, v_n\}) \cup \{v_1, \ldots, v_{x-1}\}.$$

This would imply that C_{OPT} is not feasible which is a contradiction, thus this case cannot occur.

This leaves the last case in which we have $v_x \notin C_{\mathrm{ALG}}$ but $v_x \in C_{\mathrm{OPT}}$. Here we want to remove the node v_x and add its father node v_f. Note that, as long as we are not regarding the root node, v_f always exists. So let $C'_{\mathrm{OPT}} := C_{\mathrm{OPT}} \backslash \{v_x\} \cup \{v_f\}$. As v_f is the father node, we have $f < x$ and we get that C'_{OPT} coincides with C_{ALG} on $\{v_x, \ldots, v_n\}$ as required. So all we need to show is that C'_{OPT} is a feasible cover. Let P_x be the set of paths not covered by $C_{\mathrm{OPT}} \backslash \{v_x\}$.

Clearly, as C_{OPT} is feasible, any $P \in P_x$ must contain the node v_x. If P also contains v_f, it is hit by C'_{OPT}. So the only problematic case is that $v_f \notin P$ and we must show that this cannot happen. Assume such a path did exist, then none of the nodes in P are hit. Additionally, as v_f is not part of the path, it contains only vertices of the subtree rooted at v or equivalently all vertices v_j in the path fulfil $j \geq x$. This, however, means that we get

$$\emptyset = P \cap (C'_{\mathrm{OPT}} \backslash \{v_x\} \cap \{v_x, \ldots, x_n\}) = P \cap (C_{\mathrm{ALG}} \cap \{v_x, \ldots, v_n\}) = P \cap C_{\mathrm{ALG}}$$

where the second equality follows from the fact that $v_x \notin C_{\mathrm{ALG}}$ and the last one from the observation above. Hence no node on P is in C_{ALG} contradicting the feasibility of the cover. As a result C'_{OPT} is an optimal cover with the desired properties. Lastly, if $v_x = v_1$ and it is not included by the algorithm, it is not required for a cover. Therefore, it cannot be part of the the optimal solution that coincides on all vertices except the root either.

This shows the claim and completes the proof. □

It should be noted that the above proof, in fact, shows more. Indeed, the only time we made use of the fact that our sets are paths was in the transition to the father node, for which it would suffice if the sets were connected. Consequently the algorithm still returns an optimal solution if we allow subtrees, instead of just paths, to be part of the collection of sets that need to be hit.

Finally let us take a look at the running time. As breadth-first search runs in linear time, the algorithm has a total running time of $\mathcal{O}(nT_{\mathrm{feas}})$ where T_{feas} is the time it takes to check whether a given cover is feasible. This can be done in polynomial time by the brute force method: Regard the $\mathcal{O}(n^2)$ pairs in $\mathcal{V}$ and their paths and check whether any of them contains a subpath of k unpicked nodes. This gives us $T_{\mathrm{feas}} \in \mathcal{O}(n^3)$ and a running time in $\mathcal{O}(n^4)$ for the entire algorithm.

However, as we saw in the proof of optimality, the only paths that are violated by removing the node v_i if the father node v_f is selected (which is always the case in the algorithm as all the nodes of lesser index are still contained in the cover) are those with both endpoints in the subtree rooted at v. To be precise, the set $C\backslash\{v_i\}$ regarded by the algorithm is infeasible if and only if there exists a pair of nodes $(s,t) \in \mathcal{V}$ such that s and t are in two different subtrees of v (or one of them is v) and there is a subpath of this (s,t)-path containing k unpicked nodes. With a little bit of work we can use this solution to reduce our total running time to $\mathcal{O}(n^3)$.

To do so we first pre-process the set $\mathcal{V}$: We create an $n \times n$-matrix initialised with zeroes in which an entry corresponds to a pair of nodes. We then iterate over all $\mathcal{O}(n^2)$ pairs in $\mathcal{V}$ and for each such pair (s,t) we regard all subpaths of k nodes in the (s,t)-path and set the entries of the matrix corresponding to the start and endpoint of such a subpath to one. As any (s,t)-path contains at most $\mathcal{O}(n)$ subpaths this preprocessing can be done in $\mathcal{O}(n^3)$ time. We now want to use this matrix to check in $\mathcal{O}(n^2)$ time whether a given cover is infeasible, or in other words, whether a path as described above exists. If we look at a node v_i, we can perform a breadth-first search on all subtrees T_j of v_i which terminates when a node in C is reached. This results in a subset T'_j of T_j such that T'_j forms a subtree of T_j and $T'_j \cap C = \emptyset$. This can be done in linear time. We now only need to check whether the matrix has a one entry at a position corresponding to a pair of nodes (s,t) such that $s \in T'_{j_1}$ and $t \in T'_{j_2} \cup \{v_i\}$

with $j_1 \neq j_2$. This is some subset of all pairs, so it can be checked in $\mathcal{O}(n^2)$ time. This gives us the claimed running time of $\mathcal{O}(n^3)$ and proves the following theorem.

6.9 Theorem. The $\mathcal{V}$-APC problem on a tree can be solved optimally in polynomial time in the case of uniform weights. In fact, it can be solved in $\mathcal{O}(n^3)$ time.

Observe that even though the algorithm can solve the problem for subtrees optimally as well, determining whether a cover is feasible or not does not seem to be so easy. At least our above approach will not always work as in general there can be exponentially many such subtrees. Regard for example the star graph with $n+1$ nodes. It has n paths not containing the central node and 2^n paths that contain it. This is not problematic, in terms of polynomial running time at least, if we assume that the violated sets are explicitly given.

We want to complete this section by giving another, less elementary, way of proving Theorem 6.9, excluding the running time. To do so we need a few definitions and general results. We will not prove these and refer the reader to [KN09] for details.

For this proof we want to look at our problem from another perspective. We begin by defining what an *intersection graph* is.

6.10 Definition. Given a collection of subsets $\mathcal{S} = \{S_1, \ldots, S_m\}$ of a set U, the intersection graph of $\mathcal{S}$ has a vertex v_S for each set $S \in \mathcal{S}$ and an edge between v_{S_i} and v_{S_j} if $S_i \cap S_j \neq \emptyset$. We denote the intersection graph of $\mathcal{S}$ by $IS(\mathcal{S})$.

Now let us see how the intersection graph can be helpful. Let U be the set of nodes in our tree and $\mathcal{S}$ the set of subtrees we want to hit. The goal is to find an equivalent description of a minimum hitting set in the graph $IS(\mathcal{S})$. To this end we define a *clique decomposition* of a graph.

6.11 Definition. Given a graph $G = (V, E)$, a minimum clique decomposition is a partition $V = V_1 \cup \ldots \cup V_r$ such that the graph induced by the nodes in V_i is a clique for all $i = 1, \ldots, r$ and r is minimal.

With this definition let us return to the problem at hand. Picking an element $u \in U$ hits all sets that contain u. By construction these form a clique in the intersection graph as for any pair of sets u is contained in their intersection, so it is not empty. Hence a hitting set directly gives us a clique decomposition. We want to show that the converse is true as well, allowing us to solve the problem by computing a minimum clique decomposition in the intersection graph. For this to be true we need that the elements of $\mathcal{S}$ corresponding to any clique in $IS(\mathcal{S})$ can be hit by a single element of U. This just means that any set of subtrees that intersect pairwise must contain a common node. So we show that this is indeed the case.

6.12 Lemma. Given subtrees $T_1, \ldots, T_s$ of a tree T such that $T_i \cap T_j \neq \emptyset$ for all $i \neq j$, it holds that $T_1 \cap \ldots \cap T_s \neq \emptyset$.

Proof. We prove this by induction on $n = |V(T)|$. For $n = 1$ this is clear. Let the claim hold for $n - 1$ and regard the case where $n = |V(T)|$. As T has a leaf v_T, there are two options. If there exists an i such that $T_i = \{v_i\}$, then, as $T_i \cap T_j \neq \emptyset$ for all $j \neq i$, we get $v_T \in T_1 \cap \ldots \cap T_s \neq \emptyset$. If this is not the case, any subtree T_i that contains v_T also contains

its father node v_f. So by removing the node v_T we get subtrees $T'_1, \ldots, T'_s$ of T' where T' is the tree induced by the nodes in $V(T)\backslash\{v_T\}$. These still intersect pairwise as $v_T \in T_i \cap T_j$ gives us $v_f \in T'_i \cap T'_j$. By the induction hypothesis the T'_i have a non-empty intersection, so in particular the T_i do. □

All that is left to show is that we can compute such a minimum clique decomposition in $IS(\mathcal{S})$. For this we need the notion of *chordal graphs* and *perfect elimination orderings*.

6.13 Definition. A graph $G = (V, E)$ is chordal if every (simple) cycle of length four or more has a chord, that is an edge between two non-consecutive nodes of the cycle.

6.14 Definition. Let G be a graph. A perfect elimination ordering for G is a bijective map $\sigma\colon V \to \{1, \ldots, n\}$, an ordering of the nodes, such that for every $i \in \{1, \ldots, n\}$ the neighbourhood of $\sigma^{-1}(i)$ is a clique in the graph induced by the nodes in the set $\{\sigma^{-1}(i), \ldots, \sigma^{-1}(n)\}$.

By [Gav74] the intersection graph of subtrees is a chordal graph and a graph is chordal if and only if it has a perfect elimination ordering (Theorem 4.20 in [KN09]). In fact, such an ordering can be found in linear time. So let σ be a perfect elimination ordering for $IS(\mathcal{S})$ and let $v_1, \ldots, v_n$ be the nodes of the intersection graph ordered with respect to σ. We now construct a clique decomposition as follows. Let $I = \{1, \ldots, n\}$ and choose v_1 and all its neighbours to be V_1. By the properties of σ this is a clique. Next remove the indices of all nodes in V_1 from I, so $I \coloneqq I\backslash\{i \mid v_i \in V_1\}$. Then choose the node v_i such that $i = \min I$ and set V_2 to contain it and its neighbours. This clearly results in a clique decomposition $V = V_1 \cup \ldots \cup V_r$ and we only need to show that it is minimal. To see this notice that the cardinality of any independent set is a lower bound for r as none of these nodes are connected and at least one clique is required for each. Next let $v_{i_1}, \ldots, v_{i_r}$ be the nodes chosen to induce the cliques in the algorithm. These are already an independent set. If they were not, there would be an edge from v_{i_j} to v_{i_k} for some $j < k$. By construction, as v_{i_k} is in the neighbourhood of v_{i_j}, it would have been included in the clique v_{i_j} induces. This contradicts the fact that it was chosen to induce a clique as well. Hence the algorithm computes a minimum clique decomposition.

This completes the proof. We get the same result as before, namely a polynomial time algorithm if there are only polynomially many sets in $\mathcal{S}$ or if they are all explicitly given. So this is an alternative way of showing Theorem 6.9.

7. Concluding Remarks

In this thesis we have, after summarising results concerning the hitting set problem, introduced a more general version of the path covers regarded in [FNS14a]. The path covers regarded there are the AVAPC and AVSPC we defined here. The paper notes that the primal-dual method we described in the third chapter leads to a k-approximation. We have constructed examples to show that this factor is indeed tight and have also determined how well or badly the method behaves on the newly defined path covers.

We discovered that on general graphs the situation looks rather bleak as the problems are $\mathcal{NP}$-hard and the primal-dual method cannot achieve an approximation guarantee of less than k. However, somewhat surprisingly, the case of trees behaves much more nicely. While it is true that the approximation guarantee does not improve, it turns out that all but the most general version, the $\mathcal{V}$-APC (and $\mathcal{V}$-SPC), can be solved in polynomial time. Not unexpectedly, the problem on a path is easy to solve, even in linear time. Here the primal-dual method achieves a guarantee of 2 or $2 - \frac{2}{k+1}$ depending on which of the two presented versions is chosen. Additionally, we have seen that the $\mathcal{NP}$-hard $\mathcal{V}$-APC problem on trees can be solved in polynomial time after all if we can ensure that either the amount of paths that contain a certain node is bounded by a constant or we have uniform weights.

All in all we now know in which cases we can solve path covers to optimality, what guarantees the primal-dual method ensures and which problems are hard to solve.

Finally let us take a look at some possibilities for future work. With the knowledge that the problem is hard on general graphs and the approximation guarantee is k, it seems reasonable to look for different approaches to determine such a cover. For example one could develop heuristic approaches and evaluate their quality with the use of lower bounds. Some work on this was already done in [FNS14a].

In our main application we wanted to be able to get from a location to a destination first on any possible route, then on the shortest such path. Another possibility would be to just require any such path to exist. Whether this is useful in the given scenario is another question entirely, which we will not discuss in detail. It results in cheaper solutions at the cost of possibly expensive detours.

Lastly, again in the context of the electric vehicle application, one could drop the simplifying assumption that the nodes are evenly spaced. A possible way of modelling the new problem can be found in [FNS14b]. This leads to a more complicated model, so the case of general graphs will be hopeless, but better results might be achievable on trees or paths.

A. Appendix

A.1. Description of the Thesis

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Fachbereich Mathematik
AG Optimierung
Prof. Dr. Sven O. Krumke
Telefon: 0631/205-4808
Telefon (Sekretariat): -2740
Telefax: -4737
E-Mail: krumke@mathematik.uni-kl.de
URL: http://optimierung.mathematik.uni-kl.de/~krumke

Prof. Dr. S. O. Krumke · TU Kaiserslautern · D-67653 Kaiserslautern

Datum
7. September 2016

Bachelorarbeit „Hitting Set Probleme in Graphen“

Hitting Set Probleme in Graphen

Ziel der Bachelorarbeit ist es, Modelle und Algorithmen für sogenannte *Hitting Set Probleme* in Graphen zu entwickeln und die entstehenden Probleme auf ihre Komplexität und Approximierbarkeit hin zu untersuchen.

Bei der Planung von Netzwerken für elektrische Fahrzeuge tritt das Problem auf, dass im zugrundeliegenden Netzwerk Ladestationen installiert werden müssen, um sicherzustellen, dass die Fahrzeuge auf längeren Strecken geladen werden können. Da Ladestationen teuer sind, sollen möglichst wenige dieser Stationen gebaut werden.

Dies führt in einem Graphen $G = (V, E)$ auf die Problemstellung, für eine explizit oder implizit vorgegebene Menge von Wegen $\mathcal{P}$ in G eine (kosten-) minimale Auswahl an Ecken $S \subseteq V$ so zu wählen, dass jeder Weg $P \in \mathcal{P}$ „getroffen“ wird, d.h. mindestens eine Ecke aus P in S liegt.

Ausgangspunkt der Bachelorarbeit sind die Arbeiten [2, 1, 5], deren Ergebnisse zunächst (in Auswahl) aufgearbeitet werden sollen. In [3] haben Goemans und Williamson ein allgemeines algorithmisches Konzept für die Approximation von Hitting-Set Problemen vorgestellt. In der Bachelorarbeit soll dann Anwendbarkeit dieses Konzepts auf die Probleme geprüft werden: Folgende Fragen sind von Interesse:

- Welche Gütegarantien lassen sich für die Primal-Duale-Methode aus [3] im konkreten Fall in Abhängigkeit der Wegemenge $\mathcal{P}$ und der Struktur des Graphens G beweisen?
- Ergeben sich unterschiedliche Ergebnisse im „Single-Commodity“ und „Multi-Commodity“ Fall?
- Was passiert, wenn man zusätzliche Bedingungen über das „Treffen“ der Wege fordert? (Etwa: Mehrfaches Treffen bei zu langen Wegen)
- Welche weitere Modellierungen sind aus der Anwendung heraus sinnvoll?
- Gibt es andere Ansätze, etwa auf Basis von randomisierten Rundens, die beweisbare Gütegarantien liefern?
- Wie gut sind die Verfahren „in der Realität“?

Ausgewählte Verfahren können bei Interesse implementiert und praktisch evaluiert werden.

Literatur

[1] S. Funke, A. Nusser, and S. Storandt, *Placement of loading stations for electric vehicles: No detours necessary!*, Journal of Artificial Intelligence Research **53** (2015), 633–658.

[2] _____, *On k-path covers and their applications.*, VLDB Journal **25** (2016), no. 1, 103–123.

[3] M. X. Goemans and D. P. Williamson, *The primal-dual method for approximation algorithms and its application to network design problems*, in [4].

[4] D. S. Hochbaum (ed.), *Approximation algorithms for NP-hard problems*, PWS Publishing Company, 20 Park Plaza, Boston, MA 02116–4324, 1997.

[5] S. Storandt, *Approximation algorithms in the successive hitting set model.*, Proceedings of the 26th International Symposium on Algorithms and Computation (Khaled M. Elbassioni and Kazuhisa Makino, eds.), Lecture Notes in Computer Science, vol. 9472, Springer, 2015, pp. 453–464.

A.2. Acknowledgements

I would first like to thank my thesis supervisor Professor Dr. Sven O. Krumke for his support in writing this thesis, for letting me choose from an abundance of possible topics which I would like to examine in detail and for letting me work on it on my own, but always being there with an idea when I needed help.

Furthermore I would like to thank Meiko Volz for providing the template this thesis uses. Additionally my thanks go to him along with my mum, Sebastian Blauth and Thomas Schneider for proofreading this thesis and for all the helpful feedback they provided.

Bibliography

[APMS+99] G. Ausiello, M. Protasi, A. Marchetti-Spaccamela, G. Gambosi, P. Crescenzi, and V. Kann. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1999.

[Fei98] Uriel Feige. A threshold of ln n for approximating set cover. *J. ACM*, 45(4):634–652, July 1998.

[FNS14a] Stefan Funke, Andre Nusser, and Sabine Storandt. On k-path covers and their applications. *PVLDB*, 7(10):893–902, 2014.

[FNS14b] Stefan Funke, Andre Nusser, and Sabine Storandt. Placement of loading stations for electric vehicles: No detours necessary! In Carla E. Brodley and Peter Stone, editors, *AAAI*, pages 417–423. AAAI Press, 2014.

[Gav74] Fănică Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *Journal of Combinatorial Theory, Series B*, 16(1):47 – 56, 1974.

[GW97] Michel X. Goemans and David P. Williamson. Approximation algorithms for NP-hard problems. chapter The Primal-dual Method for Approximation Algorithms and Its Application to Network Design Problems, pages 144–191. PWS Publishing Co., Boston, MA, USA, 1997.

[HMR+02] Harry B Hunt, Madhav V Marathe, Venkatesh Radhakrishnan, S.S Ravi, Daniel J Rosenkrantz, and Richard E Stearns. Parallel approximation schemes for a class of planar and near planar combinatorial optimization problems. *Information and Computation*, 173(1):40 – 63, 2002.

[KMP+02] Sven O. Krumke, Madhav V. Marathe, Diana Poensgen, Sekharipuram S. Ravi, and Hans-Christoph Wirth. Budgeted maximal graph coverage. (02-24), 2002.

[KN09] S.O. Krumke and H. Noltemeier. *Graphentheoretische Konzepte und Algorithmen*. Leitfäden der Informatik. Vieweg+Teubner Verlag, 2009.

[Kru15] Sven O. Krumke. Integer Programming, Polyhedra and Algorithms, 2015.

[RS97] Ran Raz and Shmuel Safra. A sub-constant error-probability low-degree test, and a sub-constant error-probability PCP characterization of NP. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 475–484, New York, NY, USA, 1997. ACM.

[SI96] Richard E. Stearns and Harry B. Hunt III. An algebraic model for combinatorial problems. *SIAM Journal on Computing*, 25(2):448–476, apr 1996.

[Sto15] Sabine Storandt. Approximation algorithms in the successive hitting set model. In Khaled M. Elbassioni and Kazuhisa Makino, editors, *ISAAC*, volume 9472 of *Lecture Notes in Computer Science*, pages 453–464. Springer, 2015.

Declaration of Authenticity

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Oliver Bachtler Kaiserslautern, January 7, 2017