A search for faster algorithms for the capacitated vertex cover problem.

Bas van Rooij
ICA-4155572

Supervisors:
dr. J.M.M. van Rooij
Prof. dr. H.L. Bodlaender

Department of Information and Computing Sciences,
Utrecht University,
The Netherlands

June 29, 2018

Abstract

The vertex cover problem is a well studied problem. A natural extension to the vertex cover problem is the capacitated vertex cover problem (CVC). In this problem each vertex has a predefined capacity which indicates the total amount of edges that vertex can cover, if the vertex is included in the cover. The CVC problem can be solved trivially in $O^*(2^n)$ time by enumerating all possible covers. For the CVC problem there is no known exact exponential algorithm which solves the problem in $O^*(c^n)$ time for some constant $c < 2$.

In this thesis we study the CVC problem and search for faster algorithms which solve the capacitated vertex cover problem on specific graph classes. We show the complexity of the problem with several NP-completeness proofs of the CVC problem on bipartite graphs. We will show a linear time algorithm for the CVC problem on trees and show a path- and treewidth algorithm. We will also study exponential algorithms which solve the CVC problem in $O^*(c^n)$ time on some specific graphs classes, for some constant $c < 2$. The general case of the CVC problem in $O^*(c^n)$ time will be discussed and we give some results on the difficulty of the problem with respect to the Strong Exponential Time Hypothesis.

Contents

1 Introduction

In the last couple of decades, exact exponential algorithms for NP-hard problems gained more attention. It is expected that there are no algorithms which solve NP-complete problems in polynomial time. Therefore, we must try different approaches, such as approximation algorithms, local search and exact exponential algorithms. Many NP-hard problems have a trivial exponential algorithm in $O^*(2^n)$ time, where the O^* notation hides a polynomial time factor in the runtime. For these problems people have tried to come up with algorithms that improve on this trivial bound and solve the problems in $O^*(c^n)$ time for some constant $c < 2$.

A subset of the NP-hard problems are the vertex subset problems. If there exists a polynomial time check to test if a given subset is a solution to the given problem, these problems can be solved trivially in $O^*(2^n)$ time by checking every possible subset of vertices. Many of these subset vertex problems, such as Vertex Cover [1] and Dominating Set [2], can be solved in $O^*(c^n)$ time for some constant $c < 2$.

A natural extension for these two problems is to add a capacity constraint. Each vertex v in the graph gets a predefined capacity $c(v)$, which indicates the maximum amount of edges/vertices it can cover/dominate, if v is chosen to be in the subset. These Capacitated Vertex Cover problem and Capacitated Dominating Set problem are more difficult as even checking whether a given subset is valid cover or dominating set requires a max flow or matching algorithm. For both these problems, the question for an algorithm in $O^*((2-\epsilon)^n)$ time for some $\epsilon > 0$, were stated in the IWPEC 2008 open problems by van Rooij [3]. The Capacitated Dominating Set problem was first solved by Cygan et al. [4], who gave an algorithm in $O^*(1.89^n)$ time. This runtime was later improved by Liedloff et al. [5] to $O^*(1.8463^n)$.

An algorithm for the Capacitated Vertex Cover problem (CVC) in $O^*((2-\epsilon)^n)$ time is still unknown. Before we discuss previous work on this problem, we introduce a formal definition.

Definition 1 (Capacitated Vertex Cover Problem)**.** *Given a graph $G = (V, E)$ with capacities $c(v)$ and an integer k. A subset of vertices $V' \subseteq V$ is a capacitated vertex cover if there exists a function $f : E \to V'$ such that for every edge $e = \{v, w\} \in E : f(e) \in \{v, w\}$ and for every vertex $v \in V' : |f^{-1}(v)| \leq c(v)$. The* Capacitated Vertex Cover *problem asks whether there exists a capacitated vertex cover of size $\leq k$.*

In prior work, Guo et al.[6] proved that the capacitated vertex cover problem is in FPT with respect to the solution size k and can be solved in $O^*(1.2^{k^2})$ time. Later, Michael Dom et al. [7] improved this bound to $O(2^{3klog(k)})$ using a treewidth algorithm and they proved that the CVC problem is $W[1]$-hard when parameterized by treewidth. Guha et al. [8] showed a primal-dual based approximation algorithm with an approximation ratio of 2. They also proved that the problem restricted to trees can be solved in $O(n \log n)$ time.

This work:

In this work we will show in Section 2 that the capacitated vertex cover problem is NP-complete on tree convex and modular graphs, on which the normal vertex cover problem is solveable in polynomial time. We also show that if the Exponential Time Hypothesis is true, the CVC problem on bipartite graphs cannot be solved in subexponential time.

In Section 3 we will focus on trees. We will give a linear time algorithm to solve the CVC problem on trees, improving the algorithm of Guha et al. [8], which solved a more general variant of the capacitated vertex cover problem in $O(n \log n)$ time. After that we will show a path- and treewidth algorithm which runs in $O^*((m+1)^{tw})$ time, where m is the maximum capacity over all

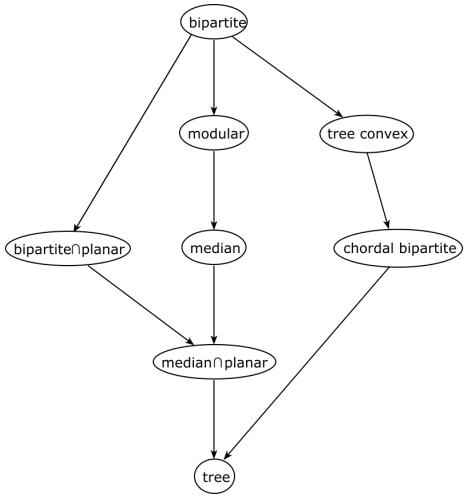

Figure 1: Inclusion graph for some bipartite graph classes.

vertices in the graph. The treewidth algorithm will altered to run in $O^*((k+1)^{tw})$ time where k is the solution size, which improves the bound from Dom et al.[7].

After that in Section 4 we will will introduce a series of algorithms which solve the capacitated vertex cover problem on some specific graph classes in $O^*((2-\epsilon)^n)$ time. We will show an algorithm for the CVC problem on graphs which contain a degree bounded spanning tree, which includes the graphs with bounded degree. We generalize this algorithm to solve the CVC problem on graphs with a significantly sized matching and finally we will combine this algorithm with the pathwidth algorithm to solve the CVC problem on graphs with $\leq cn$ edges for some constant c.

Finally in Section 5 will discuss the graph classes for which we cannot solve the CVC problem in $O^*((2-\epsilon)^n)$ time with the results of this paper. We will argue why these cases are difficult and give a reduction rule which shows that we can decide in polynomial time whether a capacity one vertex has to be included in a minimum capacitated vertex cover.

2 Difficulty of the problem

In this section we will look at several NP-completeness proofs for the capacitated vertex cover problem. There is a trivial reduction from vertex cover. For any instance of the vertex cover problem, we can make an instance of the capacitated vertex cover by setting all the capacities equal to the degree of the vertices, which is equivalent to the vertex cover problem. Therefore, the capacitated vertex cover problem is NP-complete on any graph class on which the vertex cover problem is NP-complete. There are however, more specific graph classes on which, unlike the normal vertex cover problem, the CVC problem is NP-complete. We will focus on bipartite graphs, for which the vertex cover problem can be solved in polynomial time. See Figure 1 to get an overview of the inclusion structure for the studied graph classes.

Before we give the reductions to the capacitated vertex cover problem on bipartite graphs, we will show in Section 2.1 that the problem is in NP. There we will give a polynomial time check to test whether a given subset of vertices is a valid capacitated vertex cover. After that in Section

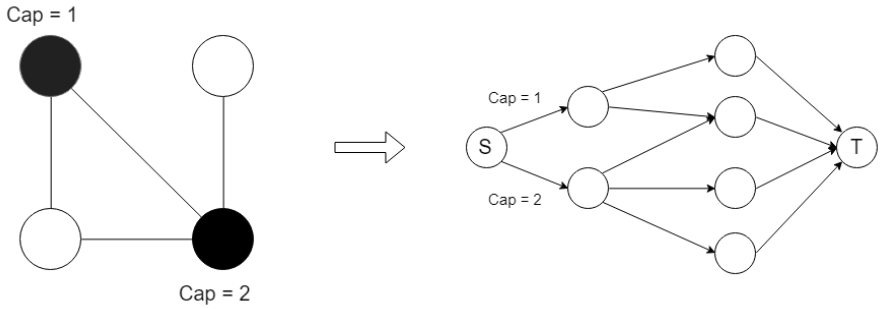

Figure 2: A bipartite flow graph. On the left the original graph G with two selected vertices (black) in a potential capacitated vertex cover. On the right the flow graph which has a flow of size 4 if and only if the selection is a valid cover. We note that the max flow is equal to 3, thus selection is not a valid capacitated cover.

2.2 we will give reductions from NP-complete problems to the CVC problem on planar bipartite graphs with maximum degree 3, modular graphs and tree convex graphs.

2.1 Polynomial time check

To show that the CVC problem is in NP, we will give a polynomial time check to test whether a given subset of vertices is a valid capacitated vertex cover. For an instance of the CVC problem on a graph $G = (V, E)$ and a given subset of vertices $V' \subseteq V$, we will construct a directed bipartite flow graph $G' = (X, Y, A)$, which has a max flow of size $|E|$ if and only if V' is a valid capacitated cover.

Lemma 1. *Given an instance of the capacitated vertex cover problem on a graph $G = (V, E)$ with capacities $c(v)$. It can be decided in polynomial time whether a given vertex subset $V' \subseteq V$ is a valid capacitated vertex cover.*

Proof. Given instance of the capacitated vertex cover on a graph $G = (V, E)$ and a subset $V' \subseteq V$. We will construct a bipartite flow graph $G' = (X, Y, A)$ which has a max flow of size $|E|$ if and only if V' is a valid capacitated vertex cover.

For every vertex $v \in V'$ we add a node to X. For every edge $e \in E$ we add a node to Y. We connect two vertices $v \in X, w \in Y$ with an arc from v to w if and only if v is an endpoint of the corresponding edge of w in G. We give every such arc a capacity of 1. Next, we add a source which is connected to all the vertices in X. For every arc to a vertex $v \in X$, we set the capacity equal to the capacity of the corresponding vertex in G. Finally we add a sink which is connected to all the vertices in Y. All the arcs from a vertex in Y to the sink get capacity 1. See Figure 2 for an example graph.

We will show that G' has a max flow of size $|E|$ if and only if V' is a valid capacitated vertex cover. For V' to be a valid capacitated vertex cover, every edge in G must be covered by a vertex in V' such that the capacity constrains are satisfied.

Assume the given set V' is a valid capacitated cover. We will show that the graph G' has a flow of size $|E|$. Every edge in E can be covered by a vertex in V' while respecting the capacities of the

vertices in the cover. In particular, for every node $v \in Y$ representing an edge, there exists a node $w \in X$ which corresponds to the vertex covering the corresponding edge of v. We note that by the construction of G', there exists an arc from w to v. We can send a flow from the source through w to v and to the sink. We can add this flow for every node $v \in Y$ and have a max flow of $|E|$. We note that for any node in X, we can send at most the capacity of the corresponding vertex in flow through the node. Because V' is a valid cover, our added flows we will not exceed this limit. Thus the constructed flow is valid and has size $|E|$.

Now assume that G' has a flow of size $|E|$. We will show that the original subset V' is a valid cover. Because the largest flow possible is at most $|E|$, we know that every vertex in Y must have a flow going through them. We note that this flow must come from a node in X which is an endpoint of the corresponding edge in G. Because the nodes in X can have at most the capacity of flow through them, we note that every edge in G can be covered by a vertex in V' such that the capacity are satisfied.

The construction of G' and the max flow can be performed in polynomial time and thus the capacitated vertex cover problem is in NP. □

For an instance of the CVC problem on a graph $G = (V, E)$, the check given in the previous Lemma performs a max flow on a graph with $O(n + m)$ vertices. In Section 5.2 we will see how we can perform this check with a max flow on a graph with $O(n)$ vertices, if we have an initial cover with an edge assignment.

2.2 Reductions

Next we will show several reduction from NP-complete problems to the capacitated vertex cover problem on several bipartite graph classes. We start with a reduction to planar bipartite graphs with maximum degree three. After that we will show a reduction from SAT to the CVC problem on bipartite graphs. This reduction is important with respect to the Exponential Time Hypothesis. Finally we will extend the SAT reduction to convex tree graphs and modular graphs. But first we will look at the planar bipartite graphs.

Lemma 2. *The capacitated vertex cover problem is NP-complete on planar bipartite graphs with maximum degree* 3*.*

Proof. We use the fact that the vertex cover problem is NP-complete on planar graphs with maximum degree 3 [9]. Given an instance of the vertex cover problem on a planar graph $G = (V, E)$ with maximum degree 3 and an integer k. We will create a new graph G', which has a capacitated vertex cover of size at most $|E| + k$ if and only if G has a vertex cover of size at most k.

We start with $G' = G$. Next we will substitute every edge in G' with the gadget in Figure 3. For every edge $\{v, w\} \in E$, we replace the edge with a copy of the gadget from Figure 3. We note that the endpoints of the edge remain in the graph and become the endpoints of the gadget. These endpoint vertices all get a capacity equal to their degree.

We note that the new graph G' stays planar and that every vertex in G' has at most degree 3. To show that G' is bipartite, we will partition the graph in two independent sets. We put every capacity 2 vertex of every gadget in one independent set and the remaining vertices in the other set. Because every edge is replaced with the gadget, all the vertices in the original graph are separated by the capacity two vertices in the gadgets. We also note that no two capacity two vertices of any two gadgets are adjacent to each other. Therefore, the given partitions are two independent sets and we conclude that G' is also bipartite.

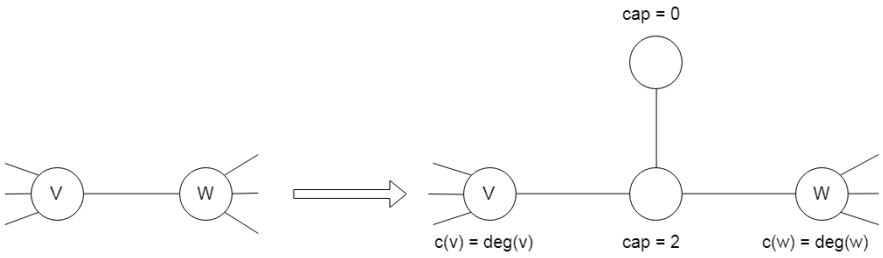

Figure 3: The planar graph gadget. On the left is the original edge e in G, on the right the gadget in G' after we have replaced e. Note that the endpoint vertices v and w remain in the graph and become the endpoints of the gadget.

Now we will show that the original vertex cover problem on G has a solution of size at most k if and only if the capacitated vertex cover in G' has a solution of size at most $|E| + k$. First we note that any capacity zero vertex is never in an optimal capacitated cover. Therefore, the zero capacity vertex in every gadget is never in the optimal cover and thus forces the capacity 2 vertex of the same gadget in the cover. This gives us at least $|E|$ vertices in any capacitated cover on G'. We note that the capacity two vertex in every gadget can cover at most one other edge in the gadget. The last edge in each gadget must be covered by one of the original endpoints.

Assume we have a solution for the vertex cover problem of size at most k in graph G. We will construct a capacitated vertex cover in G' of size at most $|E| + k$. By the previous statement, we must add the capacity two vertex in every gadget. Next we will include all the gadget endpoint vertices which are in the original solution of the vertex cover. Because this is a vertex cover, every edge in the original graph can be covered by one of the endpoints. We just saw that each gadget needed to have one capacity from one of its endpoints, which can be satisfied by the fact that the at most k added vertices form a vertex cover. We conclude that we have a capacitated vertex cover of size at most $|E| + k$.

The other way, assume we have a capacitated vertex cover of size at most $|E| + k$ on graph G'. We will construct a vertex cover of size at most k in G. We can change the gadgets in G' back to the original edges. Because the gadgets contain exactly $|E|$ vertices in the capacitated vertex cover, the remaining graph will have at most k vertices which where part of the capacitated vertex cover. We note that these vertices could cover at least one edge from every gadget, which is equivalent to covering all the edges in the original graph. Thus these remaining vertices form a vertex cover of size at most k in G. □

The previous Lemma implies that the capacitated vertex cover problem on bipartite graphs in general is also NP-complete. Later we will see that we can solve the CVC problem on planar graphs in subexponential time using a treewidth algorithm. If we only have this reduction, it would imply we might be able to solve the CVC problem on bipartite graphs in subexponential time. We can however, make a reduction from SAT to the CVC problem on bipartite graphs. The reductions from SAT is important with respect to the Exponential Time Hypothesis (ETH). This hypothesis states that $3SAT$ cannot be solved in subexponential time.

Definition 2 (Exponential Time Hypothesis)**.** *There exists an $\epsilon > 0$ such that $3SAT$ cannot be solved in $O^*(2^{\epsilon n})$ time.*

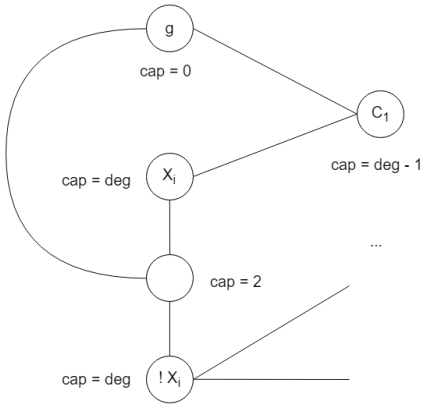

Figure 4: SAT variable gadget.

If the ETH is true, it implies that if there is a reduction from $3SAT$ to an instance of the capacitated vertex cover of size $O(n)$, the capacitated vertex cover can also not be solved in subexponential time. Of course, the reduction from SAT which we will show is also a reduction from $3SAT$. Our reduction from SAT produces a $O(n+m)$ sized bipartite graph, not $O(n)$. However, with the Sparsification Lemma, one can prove the following Theorem [14, Thm 11.8].

Theorem 1. *[14, Thm 11.8] The $3SAT$ problem is solvable in subexponential time with respect to the number of variables n if and only if it is solvable in subexponential time with respect to the number of clauses m.*

This implies that if the ETH is true and there is a reduction of either size $O(n)$ or $O(m)$, the reduced problem cannot be solved in subexponential time. In particular, if the ETH is true, our $O(n+m) \leq O(2\max\{n,m\})$ reduction shows that the capacitated vertex cover problem on bipartite graphs cannot be solved in subexponential time. We will now give the reduction from SAT.

Lemma 3. *If the ETH is true, the capacitated vertex cover on bipartite graphs cannot be solved in subexponential time.*

Proof. We will make a reduction from SAT to the capacitated vertex cover problem on bipartite graphs with $O(n+m)$ vertices. Given an instance of SAT. Let $x_1, \ldots, x_n$ be the variables of the SAT instance and let $C_1, \ldots, C_m$ be the clauses. We will construct a bipartite graph $G = (X, Y, E)$ with capacities $c(v)$, which has a capacitated vertex cover of size $2n + m$ if and only if the SAT instance is satisfiable.

We start by adding a single vertex g with capacity 0 to X. For every clause C_i we add a vertex to Y, which is connected to g. These clause vertices in Y get a capacity equal to their degree minus one. For every variable x_i we will create a gadget from Figure 4. The gadget consists of a capacity two vertex and two other vertices representing either a true or false assignment for the variable x_i. If we choose one of these vertices in the capacitated vertex cover, we effectively set the variable to true or false. The two vertices representing a true and false assignment will be connected to the clause vertices in Y which represent the clauses which contain the true and false literals. The

vertex g and the vertices c_i in the Figure are the vertices described before, they are not part of the gadget.

Now we show that the SAT instance is satisfiable if and only if the minimum capacitated vertex cover on G has size $2n + m$.

Assume we have a capacitated cover of size at most $2n + m$ in G. We note that the zero capacity vertex g is connected to all vertices in Y. Therefore, all vertices in Y must be included in the minimum capacitated vertex cover. We note that every clause c_i adds a node to Y and every gadget for a variable x_i adds a node to Y. This results in a cover of size at least $n + m$. Every variable gadget forces two vertices in the cover. The first vertex is the capacity two vertex which is connected to g. Next we note that this capacity two vertex is connected to the vertices representing a true and false assignment for the variable x_i. The capacity two vertex can cover at most one such edge, which forces at least one of the assignment vertices in the cover. This results in a cover of size at least $m + 2n$. Now every clause vertex in Y needs at least one edge to be covered by a neighboring vertex, which are exactly the vertices representing the literals included in the clause. If we take such a literal vertex in the cover, we effectively set it to true and complete all the clause vertices which contain that vertex. Therefore, if we have a capacitated vertex cover of size $2n + m$, we must have picked exactly one true or false assignment per variable x_i and every clause node in Y has at least one selected neighbor. In particular, the chosen variable assignment are in all the clauses of the SAT instance. Thus these assignments form a truth assignment to the SAT instance.

The other way, assume that we have a truth assignment to the SAT instance. As described before, all the vertices in Y must be included. For the variable vertices, we can choose exactly the n literal vertices corresponding to a truth assignment to complete the cover of size $m + 2n$. We note that any clauses vertex in G' can cover all but one of it's edges. Because we chose the literal vertices which give a truth assignment for the SAT instance, every clause vertex will have a neighbor which is included in the cover. Therefore, the capacitated vertex cover is a valid cover of size at most $m + 2n$.

If the ETH is true, we cannot solve the $3SAT$ problem in subexponential time. We note that our reduction from SAT is also a valid reduction from $3SAT$. Our reduction gives an instance of the capacitated vertex cover of size $O(n + m)$. Without loss of generality, we can assume that $m \geq n$ and bound the size by $O(n + m) = O(2m) = O(m)$. By Theorem 1 we can conclude that if the ETH is true, we cannot solve the capacitated vertex cover problem on bipartite graphs in subexponential time. □

The SAT reduction from Lemma 3 can be used for two more specific bipartite graph classes. The first class are the tree convex graphs.

Definition 3. *A bipartite graph $G = (X, Y, E)$ is called tree convex if there exists a tree $T = (X, A)$ such that for any $v \in Y$, $N(v)$ is a connected subtree in T. [12]*

Lemma 4. *The capacitated vertex cover problem is NP-complete on tree convex graphs and if the ETH is true, it cannot be solved in subexponential time.*

Proof. We will show that the graph G' constructed in the proof of Lemma 3 is a convex tree. To construct the tree $T = (X, A)$, we select g as the root of T and every other vertex in X is a leave of g. Because all the vertices in Y are connected to the vertex g, no matter what other neighbors they might have, they will form a connected subtree in T. □

We can extend the reduction of Lemma 3 to construct a modular graph.

Definition 4. *A bipartite graph G is modular if for every triplet of vertices x, y, z, there exists a vertex on three shortest paths from x to y, y to z and from x to z. [13]*

Lemma 5. *The capacitated vertex cover on modular bipartite graphs is NP-complete and if the ETH is true, it cannot be solved in subexponential time.*

Proof. Given an instance of SAT and let $G' = (X', Y', E')$ be the bipartite graph constructed with the SAT reduction described in Lemma 3. We will extend G' into a new graph $G'' = (X'', Y'', E'')$ such that G'' is a modular graph. We start with $G'' = G'$ and we add a new vertex h to Y'' which is connected to all vertices in X''. We give h maximum capacity, thus $c(h) = deg(h)$. Because h is connected to g, it has to be included in the capacitated vertex cover and because it has maximum capacity, it can cover all its edges. We note that the remaining graph is equivalent to G' and the proof that G'' has a cover of size $2n + m + 1$ if and only if the SAT instance is satisfiable is the same as in Lemma 3. We will now show that the new graph G'' is a modular graph.

We will prove that for any triplet $x, y, z \in V''$, there exists a vertex on three shortest paths. Without loss of generality we will look at four cases.

- $x, y, z \in X''$: Every vertex can reach any other vertex with a shortest path through h, thus h lies on three shortest paths.
- $x, y \in X'', z \in Y''$ and the vertices are independent: A shortest path from x to y goes through h. Because z is not connected to x and y, a shortest path from z to x has to pass through a third vertex $w \in X''$, which is a neighbor of z. From w we can go through h and reach both x and y. Thus h lies on three shortest paths.
- $x, y \in X'', z \in Y''$ and z is connected to both x and y. Now z is on both the shortest paths from z to x and z to y. There also exists a shortest path from x to y through z, thus z lies on three shortest paths.
- $x, y \in X'', z \in Y''$ and, without loss of generality, z is connected to x. The shortest path from z to x and from y to x all go through x. A shortest path from z to y goes through x followed by h to y. Thus x lies on three shortest paths.

All other combination of x, y, z, where two or more vertices are from Y'' can be reduced to these four cases. We only have to change the vertex h with g in the arguments.

Just like Lemma 3, because we have an $O(n+m)$ sized reduction from SAT to the CVC problem on modular graphs, if the ETH is true, we cannot solve the capacitated vertex cover on modular graphs in subexponential time. □

3 Trees, pathwidth and treewidth

In this Section we will focus on trees. In Section 3.1 we will give a linear time algorithm which solves the CVC problem on trees. After that in Section 3.2 we will introduce a pathwidth algorithm which runs in $O^*((m + 1)^{pw})$ time, where m is the maximal capacity over all the vertices in the graph. In Section 3.3 we will extend the pathwidth algorithm to a treewidth algorithm which runs in $O^*((m + 1)^{tw})$ time. We will also alter the treewidth algorithm to run in $O^*((k + 1)^{tw})$ time, where k is the solution size. This improves the running time of the treewidth algorithm of Dom et al.[7].

3.1 Linear time algorithm on trees

Guha et al.[8] have shown that the capacitated vertex cover problem on trees can be solved in $O(n \log n)$ time. They solved the more general case of the minimum weighted capacitated vertex cover. In that problem each vertex also has a weight w_v and we search for the smallest weighted capacitated vertex cover. We will show that a slight modification to their algorithm can solve the capacitated vertex cover on trees in $O(n)$ time.

We give a dynamic programming algorithm over a given tree $T = (V, E)$. For every node $v \in V$, with T_v we denote the subtree of T rooted at v. For every node $v \in V$ we will compute two values $f(v)$ and $g(v)$. Here $f(v)$ is the size of the minimum capacitated vertex cover of T_v such that v covers the edge to its parent node. The value $g(v)$ indicates the minimum capacitated vertex cover on T_v such that v does not cover the edge to its parent. The main difference with the algorithm in [8] is that in our case the gap $f(v) - g(v)$ can have at most three different value, namely 0, 1 and ∞.

Theorem 2. *The capacitated vertex cover problem on trees can be solved in $O(n)$ time.*

Proof. Let $T = (V, E)$ be a tree and let $r \in V$ be the root. For every vertex $v \in V$, let T_v denote the subtree rooted at v. Let e_v denote the edge from v to it's parent node. For every $v \in V$ we will calculate two values $f(v)$ and $g(v)$. The value $f(v)$ represent the smallest capacitated vertex cover on T_v, where v covers the edge e_v. The value $g(v)$ represent the smallest cover on T_v, not covering e_v. We note that the value $g(r)$ is the size of the minimum capacitated cover on the tree T.

The pseudocode of the algorithm can be seen in Figure 3.1. The Figure contains three procedures. To get the size of the minimum capacitated vertex cover for a tree T, we need to call the procedure CalculateGValue on the root of T.

To calculate the values $g(v)$, we need to check two possibilities. The case where v is excluded from the cover, in which case all children need to cover the edges to v. The other case is when v is included, in which case we can cover $c(v)$ edges to the child nodes. The most efficient edges to cover are those of the child nodes w for which the gap $f(w) - g(w)$ is the largest. We note that in our case this difference is either 0, 1 or ∞ for any child node. This allows us to greedily pick the edges to the child vertices with a gap of ∞ followed by the vertices with a gap of 1. For the child nodes where the gap is 0, it is always beneficial if the child node covers the edge to v.

For the value $f(v)$, we need to include v in the cover and spend 1 capacity to cover the edge to the parent of v. The rest is the same as in $g(v)$, where we use the remaining capacity to cover the edges to the children where the gap $f(w) - g(w)$ is the largest.

For every leaf node $v \in V$ we have $g(v) = 0$ and $f(v) = 1$ if $c(v) > 0$, otherwise we have $f(v) = \infty$. It may happen that we get a node v with $g(v) = \infty$. In this case there does not exists a capacitated cover and we can stop the algorithm. It may also happen that a node v with capacity $c(v)$ has more than $c(v)$ children with $f(w) = \infty$. In this case we also get that there does not exists a capacitated vertex cover.

We note that per node v, the values $f(v)$ and $g(v)$ can be computed with one pass over the child nodes. In particular, every node is visited at most once by it's parent node, which gives us a running time of $O(n)$.

□

The main difference between this algorithm and the algorithm of Guha et al.[8] is that in our case for all nodes $v \in V$, the difference between $f(v)$ and $g(v)$ has at most three different values. It is either 0, 1 or infinity. For any node v which is included in the cover, we need to decide which

Algorithm 1 Linear time algorithm for the Capacitated Vertex Cover Problem on trees

Input: A tree $T = (V, E)$ with capacitties $c(v)$.

Output: The value CalculateGValue on the root of T returns the size of the smallest capacitated vertex cover on T.

```
1: procedure CalculateCostGivenCapacity(node, cap)
2:     RemainingCap ← cap
3:     Result ← 0
4:     Create a bucket B_1 for nodes with a gap of 1.
5:     for all Children c of node do
6:         Gap ← CalculateFValue(c) − CalculateGValue(c)
7:         if Gap is 0 then
8:             Include ← Include + CalculateFValue(c)
9:         end if
10:        if Gap is ∞ then
11:            if RemainingCap is 0 then
12:                return ∞
13:            end if
14:            RemainingCap ← RemainingCap − 1
15:            Result ← Result + CalculateGValue(c)
16:        end if
17:    end for
18:    for all nodes c in B_1 do
19:        if RemainingCap is 0 then
20:            Result ← Result + CalculateFValue(c)
21:        else
22:            RemainingCap ← RemainingCap − 1
23:            Result ← Result + CalculateGValue(c)
24:        end if
25:    end for
26:    return Result
27: end procedure
28:
29: procedure CalculateGValue(node)
30:    if node is a leaf node then
31:        return 0
32:    end if
33:    Exclude ← CalculateCostGivenCapacity(node, 0)
34:    Include ← 1 + CalculateCostGivenCapacity(node, capacity of node)
35:    Result ← min{Exlude, Include}
36:    if Result is ∞ then
37:        Stop the algorithm, it is impossible to cover the given subtree
38:    end if
39:    return Result
40: end procedure
41:
42: procedure CalculateFValue(node)
43:    if node is a leaf node then
44:        if capacity of node > 0 then
45:            return 1
46:        else
47:            return ∞
48:        end if
49:    end if
50:    return 1 + CalculateCostGivenCapacity(node, capacity of node − 1)
51: end procedure
```

edges to the child nodes it covers. The three different gap values allows to pick these child nodes greedily, starting with those who have a gap value of ∞. After that we pick the nodes which have a gap of 1. This observation speeds up the algorithm as we can leave out a sorting step. In the weighted capacitated vertex cover problem in [8], the gap value could be any value, which requires a sorting step over the gap values of the child nodes to pick the most optimal edges to cover. Because we can leave this sorting step out, we improve the run time from $O(n \log n)$ to $O(n)$ time.

3.2 Pathwidth

In this Section we will introduce a pathwidth algorithm. In the Section 3.3 we will extend this algorithm to a treewidth algorithm. Path- and treewidth are commonly used algorithmic techniques which can solve difficult problems on graphs with small path- or treewidth efficiently. For the capacitated vertex cover problem, Dom et al.[7] have shown a treewidth algorithm which runs in $O^*(k^{3tw})$ time, where k is the solution size. In Section 3.3 we will introduce a treewidth algorithm which runs in $O^*((m+1)^{tw})$ time, where m is the maximum capacity over all vertices in the graph. We will also alter the treewidth algorithm such that it runs in $O^*((k+1)^{tw})$ time, improving the result by Dom et al.[7].

Before we give the treewidth algorithm in Section 3.3, we start with a pathwidth algorithm which runs in $O^*((m+1)^{pw})$ time, where $m = \max_{v \in V} c(v)$ is the maximum capacity over all vertices. We will first introduce the pathwidth of a graph.

Definition 5. *A path decomposition for a graph $G = (V, E)$ is a path P where each node represents a* bag *$X_i \subseteq V$, such that:*

- *For every $v \in V$, there exists a bag X_i such that $v \in X_i$.*
- *For every edge $\{v, w\} \in E$, there exists a bag X_i such that $v, w \in X_i$.*
- *For every $v \in V$, all the bags which contain v form a connected subpath of P.*

The width *of a path decomposition is the size of the largest bag minus one. The pathwidth of a graph G, $pw(G)$, is the minimum width of a path decomposition over all path decompositions of G.*

Path decompositions can be used to create fast dynamic programming algorithms with a running time of $O(f(pw) \cdot poly(n))$ for some function $f(pw)$. Typically, $f(pw)$ is exponential in pw. This implies that if we have graphs with a constant pathwidth, we can obtain polynomial time algorithms for some NP-hard problems on those graphs. Also, if we have graphs with a pathwidth $\leq \delta n$ for some $\delta < 1$, we can achieve running times of $O^*((2-\epsilon)^n)$ for some $\epsilon > 0$ for some NP-hard problems.

Our pathwidth algorithm will use a nice path decomposition with edge introduce bags. With edge introduce bags every edge in the graph gets added to the path decomposition in their own separate bag.

Definition 6. *A path decomposition with edge introduce bags is called nice if every bag X_i is one of the following types:*

- *Start node: $X_0 = \{v\}$ for some vertex v.*
- *End node: $X_{last} = \{v\}$ for some vertex v.*
- *Vertex introduce node: $X_i = X_{i-1} \cup \{v\}$, for some vertex v.*

- *Edge introduce node:* $X_i = X_{i-1} \cup \{e\}$*, for some edge* e*.*
- *Vertex forget node:* $X_i = X_{i-1} \setminus \{v\}$*, for some vertex* v*.*

We require that every edge is introduced exactly once in the path decomposition.

If we have a path decomposition of width k, we can make a nice path decomposition with edge introduce bags of width at most k in $O(n^2)$ time [14]. We define the graph $G_i = \cup(X_j, E_j), j \leq i$ as the graph consisting of all the vertices and edges added in bags up to and including the bag X_i.

We will now introduce the pathwidth algorithm for the capacitated vertex cover problem. The algorithm will compute a table per bag X_i, which contains the size of the optimal solutions for the graph G_i, for different initial values. As we will see, every table on a bag X_i is only dependent on the values of the table X_{i-1}. This allows us to build the tables one by one, starting with X_0. Because we have at most $O(n^2)$ number of bags, the running time of the algorithm in O^* only depends on the time it takes to compute a table for a single bag. We will see that it takes $O^*((m+1)^{pw})$ time to compute the table per bag X_i.

Theorem 3. *Given an instance of the capacitated vertex cover problem on a graph* $G = (V, E)$ *with capacities* $c(v)$ *and given a path decomposition of width at most* k*. The capacitated vertex cover problem can be solved in* $O^*((m+1)^k)$ *time where* $m = \max\{c(v) : v \in V\}$ *is the maximum capacity over all vertices in the graph.*

Proof. First we construct a nice path decomposition P with edge introduce bags, thus every edge will be added in it's own bag. For every bag X_i we compute the values $c_i(f)$, which indicates the smallest capacitated vertex cover over G_i, given the capacity function f. The capacity function $f : X_i \to \mathbb{N}$ indicates the total amount of capacity a certain vertex has used in the bags $X_j, j < i$. In other words, the capacity function tells how much capacity a vertex has spend in all the bags up to the current bag. The optimal solution of a graph is the smallest value over all the values $c_m(f)$, where X_m is the last bag in the path decomposition, for all the possible capacity functions f.

From the capacity function we can derive whether a certain vertex is included in the cover. If a vertex has 0 spend capacity in a forget node, we know that it has not spend any of it's capacity in the capacitated cover and we can leave it out of the cover. Otherwise it has covered an edge somewhere in the previous bags and the vertex must be included in the cover.

We will show the recursive formulas to calculate the table $c_i(f)$ for a given bag X_i with a given capacity function f. We give the formula for every type of bag.

- **Start node:** Let $X_0 = \{v\}$ with v a vertex. Because we just added the vertex v, it cannot have spend any capacity in previous bags. Thus any capacity function where v has spend capacity is impossible. This gives the following recurrence.

$$c_0(f) = \begin{cases} \infty \text{ if } f(v) > c(v) \\ 0 \text{ if } f(v) = 0 \end{cases}$$

  This step takes $O(1)$ time per capacity function f.

- **Vertex introduce node:** Let $X_i = X_{i-1} \cup \{v\}$ with v a vertex. Here we do the same as the start node. The vertex cannot spend any capacity in any previous bags, as it is not included in those bags.

$$c_i(f) = \begin{cases} \infty \text{ if } f(v) > 0 \\ c_{i-1}(f') \text{ if } f(v) = 0 \end{cases}$$

where $\forall w \in X_{i-1} : f'(w) = f(w)$.

This step takes $O(1)$ time per capacity function f.

- **Vertex forget node:** Let $X_i = X_{i-1} \setminus \{v\}$ for some vertex v. Because the vertex is removed from the path decomposition, we can check if it has to be added to the cover. If vertex v has spend capacity in the previous bags, it has to be included in the cover and needs to be counted. Otherwise, if the vertex has 0 spend capacity, we can leave it out of the cover. We will check for the best solution over all the possible amounts of spend capacity.

$$c_i(f) = \begin{cases} \min_{f'(v)=1,\dots,c(v)}\{1 + c_{i-1}(f')\} \\ c_{i-1}(f') \text{ where } f'(v) = 0 \end{cases}$$

where $\forall w \in X_{i-1} : f'(w) = f(w)$. This takes $O(n)$ time per capacity function f.

- **Edge introduce node:** Let $X_i = X_{i-1} \cup \{e\}$, with e an edge with endpoints $v, w \in X_i$. When we introduce an edge, it has to be covered by one of it end points. We get

$$c_i(f) = \begin{cases} \min\{c_{i-1}(f'), c_{i-1}(f'')\} \\ \infty \text{ if there is no valid cover covering } e \end{cases}$$

where $f'(v) = f(v) - 1$ and $f''(w) = f(w) - 1$, the rest is equal to f. We set the value equal to ∞ if there is no remaining capacity, thus $f(w) = 0$ and $f(v) = 0$.

This step takes $O(1)$ time per capacity function f.

The total amount of time spend per bag in the pathwidth algorithm depends on the maximum number of capacity functions per bag. We note that for any vertex $v \in X_i$, we have $c(v)+1$ different values a capacity function can assign. This results in a simple bound of $(m+1)^k$ different capacity functions, where $m = \max_{v \in V} c(v)$. We get a running time of $O^*((m+1)^k)$. □

Later we will see an example where we can create a better bound on the amount of capacity functions per bag under a constrain on the number of edges in the graph. We can use the pathwidth algorithm to solve cubic graphs efficiently.

Definition 7. *A graph $G = (V, E)$ is a cubic graph if all vertices $v \in V$ have a degree* 3.

Fomin and Høie [11] proved the following theorem on pathwidth for cubic graphs.

Theorem 4. *Cubic graphs have a pathwidth of $(\frac{1}{6} + \epsilon)n$, for any $\epsilon > 0$. [11]*

We note that any vertex in a cubic graph has a capacity of at most 3. Therefore, we can apply the pathwidth algorithm on cubic graphs and solve the capacitated vertex cover in $O^*(4^{(\frac{1}{6}+\epsilon)n})$ time for any $\epsilon > 0$. We note that the worst case in this algorithm is achieved when we have a bag whose vertices all have a capacity equal to three. In this case each vertex has four states, namely

all the possibilities on capacity used in previous bags. However, if a vertex has capacity three in a cubic graph, then that vertex has maximum capacity. Thus we either take the vertex and cover all the edges, or we do not take the vertex. In particular, we can reduce the amount of states of a capacity three vertex to two. Now each vertex has at most 3 different states, which is achieved when the vertex has a capacity of 2. This improves the worst case to a bag where each vertex has three different states, which results in a run time of $O^*(3^{(\frac{1}{6}+\epsilon)n})$.

Corollary 1. *The capacitated vertex cover problem on cubic graphs can be solved in* $O^*(3^{(\frac{1}{6}+\epsilon)n})$ *time for any* $\epsilon > 0$.

3.3 Treewidth

Next we will extend the pathwidth algorithm of Section 3.2 to a treewidth algorithm. The treewidth of a graph is defined similar to pathwidth. The main difference is that for a tree decomposition, the decomposition structure can be a tree instead of a path.

Definition 8. *A tree decomposition for a graph* $G = (V, E)$ *is pair* (T, X) *with* $T = (I, F)$ *a tree and* $X = \{X_i | i \in I\}$ *a family of subsets of* V*, called bags, one for each node of* T *such that:*

- *For every* $v \in V$*, there exists a bag* X_i *such that* $v \in X_i$.
- *For every edge* $\{v, w\} \in E$*, there exists a bag* X_i *such that* $v, w \in X_i$.
- *For every* $v \in V$*, all the bags* X_i *which contain* v *form a connected subtree of* T.

The width *of a tree decomposition is the size of the largest bag minus one. The treewidth of a graph G, tw(G), is the minimum width of a tree decomposition over all tree decompositions of G.*

Note that any path decomposition is a valid tree decomposition, which gives us that $tw(G) \leq pw(G)$. Just like the pathwidth algorithm, the treewidth algorithm will work on a nice tree decomposition with edge introduce bags. This adds a new type of bag, the join node.

Definition 9. *A tree decomposition of a graph G with edge introduce bags is called nice if every bag* X_i *is one of the following type:*

- *Join node:* X_i *has two children* X_j *and* X_k *and we have that* $X_i = X_j = X_k$.
- *Vertex introduce node: The bag* X_i *has one child bag* X_j *and we have that* $X_i = X_j \cup \{v\}$ *for some vertex* v.
- *Edge introduce node: The bag* X_i *has one child bag* X_j *and we have that* $X_i = X_j \cup \{e\}$ *for some edge* e.
- *Vertex forget node: The bag* X_i *has one child bag* X_j *and we have that* $X_i = X_j \setminus \{v\}$.
- *Leaf node: The bag* X_i *has no child nodes and contains one vertex,* $X_i = \{v\}$ *for some vertex* v.

We also require that every edge is introduce exactly once in the tree decomposition.

A nice tree decomposition with edge introduce bags can be calculated in $O(n^2)$ time from a given tree decomposition of width k [14]. The algorithm on a tree decomposition works similar to the path decomposition algorithm. For a tree decomposition we define G_i as the graph consisting of all vertices and edges added in the bags in the subtree rooted at X_i. The capacity functions f for a bag X_i indicate how many capacity a certain vertex spends in the graph G_i. The table we will calculate per bag and the described bag types from Theorem 3 stay the same for the treewidth algorithm. We will only describe the additional join node and leaf node operation.

Theorem 5. *For a graph $G = (V, E)$ with a given treewidth decomposition of width at most k and a maximum capacity of $m = \max\{c(v) : v \in V\}$, the capacitated vertex cover can be solved in $O^*((m+1)^k)$.*

Proof. First we construct a nice tree decomposition T with edge introduce bags. For each bag we will calculate the values $c_i(f)$ for all capacity functions as in Theorem 3. The node introduce, forget and edge introduce nodes are the same as the pathwidth algorithm. We will now show the leaf and join node.

- **Leaf node:** Let $X_i = \{v\}$ be a leaf node with some vertex $v \in V$. We calculate the table the same as the begin node of the pathwidth algorithm.

$$c_0(f) = \begin{cases} \infty \text{ if } f(v) > 0 \\ 0 \text{ if } f(v) = 0 \end{cases}$$

  This takes $O(1)$ per capacity function f.

- **Join node:** Let $X_i = X_j = X_k$ where X_j, X_k are the child bags of X_i. For a join node we need to add the used capacity from both child nodes. We get

$$c_i(f) = \min_{g,l}\{c_k(g) + c_j(l)\} \tag{1}$$

  where we take the minimum over all capacity functions g, l such that $\forall v \in X_i : g(v) + l(v) = f(v)$. If we give the same upper bound of $(m+1)^{tw}$ to the number of capacity functions, we can compute this join node in $O^*((m+1)^{tw})$ time using the techniques described in [17], which are outside the scope of this paper.

We note that the additional bag types can be computed in $O^*((m+1)^{tw})$ time for all capacity functions f. Therefore, the treewidth algorithm runs in $O^*((m+1)^{tw})$ time as well.

□

We can use the treewidth algorithm to solve the capacitated vertex cover problem on planar graphs in subexponential time. It is know that any planar graph has a treewidth of $O(\sqrt{n})$. Therefore, the treewidth algorithm will run in $O^*((n+1)^{O(\sqrt{n})}) = O^*(2^{O(\sqrt{n}log_2(n))})$ time on planar graphs.

Corollary 2. *The capacitated vertex cover problem on planar graphs can be solved in $O^*(2^{O(\sqrt{n}log_2(n))})$ time.*

We can alter the treewidth algorithm to obtain a FPT algorithm with respect to the solution size k. We obtain a running time of $O^*((k+1)^{tw})$, which improves the time given by Dom et al. [7]. To do this we will use the same counting argument as described in [7]. Instead of keeping track of the capacity used by vertices with capacity functions, we keep track on how many edges are covered by the neighbors of the vertices. Thus the new capacity functions f indicate how many edges will be covered by neighboring vertices in previous bags. We note that this number can be at most k for any vertex, otherwise we would have a solution of size $k+1$. The recursions given previously stay mostly the same. The main difference is that in the edge introduce bags we have to adjust the capacity function for the other neighbor. To check whether a vertex v has to be included in the cover we have to check whether the amount of edges covered by neighbors is equal to the degree of v. If this is the case, we can leave v out of the cover, otherwise v covers an edge and must be included in the cover. Finally to check if a vertex doesn't spend more capacity than it has available, we check the difference between the amount of edges covered by neighbors and the degree of the vertex. If this difference is larger than the capacity of v, we have spend more capacity than we have and we have an invalid solution.

Corollary 3. *Given an instance of the capacitated vertex cover problem and an integer k. We can check if there exists a solution of size at most k in $O^*((k+1)^{tw})$ time.*

4 Faster exponential time algorithms

In this section we will introduce a series of algorithms which solve the capacitated vertex cover problem in $O^*((2-\epsilon)^n)$ time for some specific graph classes. First in Section 4.1 we will introduce the different techniques we use to determine the running time of the algorithms. After that in Section 4.2, we introduce an algorithm which solves the CVC problem on graphs which contain a bounded degree spanning tree. In Section 4.3 we will generalize the main argument of this algorithm to solve the CVC problem on any graph which contains a significantly sized matching, that is, a matching of size $\geq \epsilon n$ for some $\epsilon > 0$. After that in section 4.4, we will solve the CVC problem on graphs which contain significantly many twin vertices. Finally, in Section 4.5, we will combine these algorithms with the pathwidth algorithm to solve the CVC problem on graphs with several edge or capacity restrictions.

Before we give the first algorithm, we will introduce some notation and techniques to determine the running time of the algorithms.

4.1 Runtime analysis

For the capacitated vertex cover problem, if a graph G contains multiple connected components, we can solve the problem on each component individually and combine the optimal results to solve the instance of the CVC on G. Because the algorithms discussed in this section have an exponential running time, the running time of an algorithm on such a graph G is dominated by the largest connected component. In the remaining of this paper, we will assume that all graphs are connected.

In Section 4.2, we will introduce a branching algorithm. For an instance of the CVC problem on a given graph G, the algorithm will branch into two or more subproblems, which the algorithm will solve recursively. This creates a search tree $T(n)$, where each node in the tree represents a subproblem encountered while branching. In order to determine the running time of a branching

algorithm, we want to bound the number of nodes of a searchtree $T(n)$, where n is the amount of vertices of a graph G.

Branching algorithms contain one or more branching rules, which indicate under what circumstances the problem needs to be split into which subproblems, and how to make the split. To bound the number of nodes in a search tree, we will look at the branching vectors of the branching rules. A branching vector of a branching rule which splits into m different branches, is a vector $(x_1, \ldots, x_m)$, with $x_i \in \mathbb{R}_{>0}$. Each x_i in the branching vector indicates the amount of vertices we lose in branch i. The following Theorem [14, thm. 2.1] allows us to calculate a running time for a branching algorithm based on the branching vector.

Theorem 6. *[14, thm. 2.1] Let $(x_1, \ldots, x_m)$ be a branching vector for a branching rule* b. *If a branching algorithm uses only branching rule* b*, then the runtime of the algorithm is $O^*(\alpha^n)$ where α is the unique positive root of the equation*

$$1 - \sum_{i=1}^{m} x^{-x_i} = 0$$

For a given vector $(x_1, \ldots, x_m)$, we denote the value α with $\tau(x_1, \ldots, x_m)$. This function is often called the τ-function.

If we have a search tree consisting entirely out of a singe branching rule b, we get the running time described in Theorem 6. It may happen that a branching algorithm has more than one branching rule, which results in more than one branching vector. In order to bound the running time over all possible search trees, we will take the worst $\tau(b)$ value over all the different branching rules b. For a more detailed explanation about branching vectors, we refer to [14].

Next, we will bound the running time of a combinatoric algorithm which checks $\binom{n}{\alpha n}$ subsets, for some alpha $0 < \alpha < 1$, and spends polynomial time per subset. In this case one can bound the running time using Stirling's formula [14, lm. 3.13].

Lemma 6. *For any $\alpha \in (0, 1)$, we have that*

$$O^*(\binom{n}{\alpha n}) \leq O^*((\frac{1}{\alpha^\alpha (1-\alpha)^{1-\alpha}})^n).$$

We note that for any $\alpha \neq \frac{1}{2}$ the runtime is of the form $O^*((2-\epsilon)^n)$, for some $\epsilon > 0$.

4.2 Graphs with bounded degree

In this section we will introduce a branching algorithm which solves the capacitated vertex cover problem on graphs with a bounded degree spanning tree, which includes graph of bounded degree. The algorithm is based on two observations. First, every valid capacitated vertex cover is also a valid vertex cover on the same graph. The second is that a vertex cover for a graph remains a valid vertex cover even after we remove edges from the graph. In particular, every capacitated vertex cover is a vertex cover for a spanning tree of the graph.

The algorithm will enumerate all vertex covers for a spanning tree T of G. For every vertex cover on T, it checks if it is a valid capacitated vertex cover on G, and in the end, the algorithm returns the smallest valid capacitated vertex cover. To prove the running time of the algorithm, we will need the following Lemma about enumerating vertex covers on trees.

Table 1: Running times for graphs with bounded degree

Max degree	2	3	4	5	6	7	8	9	10
Running time	1.619	1.710	1.733	1.763	1.791	1.816	1.836	1.853	1.867

Lemma 7. *All vertex covers of a tree of bounded degree $k \geq 3$ can be enumerated in $O^*((1+2^{k-1})^{\frac{n}{k}})$ time. All vertex covers of a tree with bounded degree 2, which are paths, can be enumerated in $O^*(1.619^n)$ time.*

Proof. Let T be a tree with bounded degree k. Choose an node of the tree as the root such that each node has at most $k-1$ child nodes. We will branch on the maximum depth leaf node v. We branch on v, the parent node of v and all the sibling nodes of v. Let d be the number of vertices branched on, thus $d = 1 + |children(parent(v))|$. The branches will consist of all possibilities, under the constrain that for every edge at least one endpoint must be included. In particular, if we leave out the parent node, we have must include all the child nodes in the cover. If we include the parent node, we get 2^{d-1} possibilities for the child nodes. In each branch we remove all the decided vertices.

We will measure the running time of this branching rule with respect to the number of vertices removed in each branch. The branching has a τ value of $\tau(d, \ldots, d)$, where the τ-function has $1 + 2^{d-1}$ arguments. By Theorem 6, the runtime is the root of the equation

$$\begin{aligned} 1 - \sum_{1+2^{d-1}} x^{-d} &= 0 \\ 1 - (1 + 2^{d-1}) \cdot x^{-d} &= 0 \\ x^{-d} &= \frac{1}{1+2^{d-1}} \\ x &= (\frac{1}{1+2^{d-1}})^{-\frac{1}{d}} = (1+2^{d-1})^{\frac{1}{d}} \end{aligned} \tag{2}$$

This results in a running time of $O^*((1+2^{d-1})^{\frac{n}{d}})$. We note that $d \leq k$. To determine the running time of the algorithm, we take the worst running time over all the possible values of d. For all $k \geq 4$, the running time of the algorithm is dominated in the case where $d = k$. This is because the function $(1+2^{k-1})^{\frac{n}{k}}$ is a strictly increasing function on the domain $k \in [4, \infty)$ and because the case where $d = 4$ dominates the run time of $d \leq 3$. In particular, we have that $(1+2^{k-1})^{\frac{n}{k}} < (1+2^k)^{\frac{n}{k+1}}$ for any $k \geq 4$. This results in a running time of $O^*((1+2^{k-1})^{\frac{n}{k}})$ time.

For the case $k = 3$, the running time is dominated by the case where a parent node has one child, instead of two. Thus $\tau(2,2,2) > \tau(3,3,3,3,3)$. When the parent node has only one child node, we can make a different branch. We either take the child node, or discard it and take the parent node. This results in a run time of $\tau(1,2) < O^*(1.619^n)$. This improves the running time for the cases where $k = 2$ and $k = 3$. For $k = 3$ we get a runtime of $O^*((1+2^{k-1})^{\frac{n}{k}}) = O^*(1.733^n)$. For $k = 2$ we get a runtime of $O^*(1.619^n)$ time.

It may happen that the algorithm ends with a single root node, without any children left to branch on. In this case simple branch on both possibilities, because this occurs at most once in each branch of the search tree, it will not affect the run time in the big O^*. □

In Table 1 you can see the run time for the first 10 values of k. For any given constant k,

this runtime is of the form $O^*((2-\epsilon)^n)$ for some $\epsilon > 0$. Any graph with bounded degree k has a spanning tree with bounded degree k. This allows us to solve the capacitated vertex cover problem on bounded degree graphs in the running time stated by Lemma 7.

Corollary 4. *The capacitated vertex cover problem on graphs with bounded degree k for any constant $k \geq 3$ can be solved in $O^*((1+2^{k-1})^{\frac{n}{k}})$ time.*

In Section 5.3 we will see how we can solve the CVC problem on graphs with bounded degree 2 in polynomial time. In the next Lemma we will extend the spanning tree algorithm to solve the CVC problem on any graph having a bounded degree spanning tree.

Lemma 8. *The capacitated vertex cover problem on graphs with a spanning tree of bounded degree k, $k \geq 2$, can be solved in $O^*((1+2^k)^{\frac{n}{k+1}})$ time.*

Proof. Given an instance of capacitated vertex cover on a graph $G = (V, E)$ with a k degree bounded spanning tree. As stated before, every valid capacitated vertex cover in G is also a normal vertex cover for every spanning tree of G.

To solve the instance of the capacitated vertex cover problem, we will construct a spanning tree over G. Because G has a bounded degree spanning tree, by Lemma 7 we can enumerate all possible vertex covers of the spanning tree in $O^*((1+2^{k-1})^{\frac{n}{k}})$ time. However, finding the optimal minimum degree spanning tree is NP-hard. For our purpose we can use a polynomial time approximation which results in a spanning tree of maximum degree at most $k+1$ [10]. We will enumerate all vertex covers on this approximated spanning tree and return the smallest capacitated vertex cover. □

We note that the previous Lemma assumed we do not know the minimum degree spanning tree of the given graph. Therefore, we had to use an approximation which costs us in the running time of the algorithm. If the minimum degree spanning tree is given, we can improve the running time.

Corollary 5. *Given an instance of the capacitated vertex cover problem on a graph which contains a spanning tree with bounded degree $k \geq 3$ and given such spanning tree. We can solve the CVC problem in $O^*((1+2^{k-1})^{\frac{n}{k}})$ time. If the graph has a spanning tree with bounded degree 2, a hamiltonian path, we can solve the CVC problem in $O^*(1.619^n)$ time.*

We will give an indication of the difference for the CVC problem on graphs which have a hamiltonian path. These graphs have a spanning tree of maximum degree 2.

Corollary 6. *The capacitated vertex cover problem on graphs which contain a hamiltonian path can be solved in $O^*(\sqrt{3}^n)$ time.*

Corollary 7. *The capacitated vertex cover problem on graphs which contain a hamiltonian path and where the hamiltonian path is given can be solved in $O^*(1.619^n)$ time.*

4.3 Matchings

The main observation which allows us to improve on the $O^*(2^n)$ time barrier on graphs with a bounded degree spanning tree, is that for every edge at least one endpoint needs to be in the cover. This gives us three possibilities per set of edge endpoints, instead of four. We can make a branching rule which, for a given edge, branches in the three possibilities in $\tau(2,2,2)$ time. This rule requires that both endpoints are undecided. If we take an edge where one of the endpoints is included in the cover, we get a $\tau(1,1)$ rule on the other vertex, which will not break the $O^*(2^n)$ barrier.

However, if we have a significantly large set of disjoint edges, i.e. a matching, we can use the faster branching rule on those independent edges and the result will be an algorithm which breaks the $O^*(2^n)$ time barrier. The next Lemma formalizes the algorithm.

Theorem 7. *Let G be a graph. If G contains a matching M with $|M| \geq \epsilon n$ for some $\epsilon > 0$, we can solve the capacitated vertex cover on G in $O^*(\sqrt{3}^{2\epsilon n} 2^{(1-2\epsilon)n})$ time.*

Proof. Given an instance of the capacitated vertex cover on a graph $G = (V, E)$. Let M be a matching in G with $|M| \geq \epsilon n$ for some $\epsilon > 0$. For every edge $e \in M$ at least one of the endpoints needs to be in the cover. For every edge we branch over all three possibilities with a branching vector of $\tau(2, 2, 2)$. These branches remove exactly 2ϵ vertices and takes $\tau(2, 2, 2) < O^*(\sqrt{3}^{2\epsilon n})$ time. For every branch we will brute force the remaining vertices in $O^*(2^{(1-2\epsilon)n})$ time, which results in the total running time of $O^*(\sqrt{3}^{2\epsilon n} 2^{(1-2\epsilon)n})$. □

This matching algorithm allows us to solve the CVC problem on any graph which contains a significantly sized matching. In particular, we can solve regular graphs.

Definition 10. *A graph $G = (V, E)$ is regular if all vertices have the same degree. A graph G is called k-regular if all vertices have a degree k.*

We note that k-regular graphs have a bounded degree of k. Therefore, by Lemma 4 we can solve the CVC problem on k-regular graphs with $k \geq 3$ in $O^*((1 + 2^{k-1})^{\frac{n}{k}})$ time. In order to solve regular graphs in general, we need the following Theorem by Henning and Yeo [15] which states that k-regular graphs have a significant large matching.

Theorem 8. *Let G be a connected k-regular graph with $k \geq 3$. If k is even, G has a maximum matching of size at least $\min\{(\frac{k^2+4}{k^2+k+2})\frac{n}{2}, \frac{n-1}{2}\}$. If k is uneven, G has a maximum matching of size at least $\frac{(k^3-k^2-2)n-2k+2}{2(k^3-3k)}$. [15]*

With this Theorem we can give a lower bound on the size of the maximum matching over all regular graphs. We can use this lower bound to solve the CVC problem on regular graphs in general using the matching algorithm.

From Theorem 8 we can conclude that for even k, the minimum lower bound of the size of the maximum matching over all connected k-regular graphs is achieved for both $k = 4$ and $k = 6$, which gives a lower bound of $\frac{10n}{22}$. For odd $k \geq 3$, the minimum lower bound on the size of the maximum matching is achieved for $k = 3$, which attains a value of $\frac{4n}{9}$. We conclude that for any k-regular graph with $k \geq 3$, we have a lower bound on the maximum matching of size $\frac{4n}{9}$. For any $k \geq 3$, we get a maximum matching of size at least $\frac{4n}{9}$. We note that if $k = 1$, we have a perfect matching and if $k = 2$, we get a cycle, which contains a matching of size $\lfloor \frac{n}{2} \rfloor$. In Section 5.3 we will see how we can solve regular graphs with $k = 1$ and $k = 2$ in polynomial time. We conclude that a regular graph has a matching of size at least $\frac{4n}{9}$, which leads to the following corollary.

Corollary 8. *Let $G = (V, E)$ be a regular graph. We can solve the capacitated vertex cover problem in $O^*(1.733^{\frac{8}{9}n} 2^{\frac{1}{9}n}) = O^*(1.760^n)$ time.*

Although the matching algorithm allows us to solve the CVC problem on a number of graphs classes, it is not enough to solve the general capacitated vertex cover problem. If we take an arbitrarily sized star graph where the middle vertex has a capacity of $\frac{1}{2}n$ and the leaves all have

capacity 1, we have $O(2^n)$ different minimum capacitated vertex covers, which the matching algorithm will all check. This shows that the matching algorithm cannot solve the capacitated vertex cover problem in $O^*((2-\epsilon)^n)$ time on arbitrary graphs.

Although the star graph used in this example is a tree and they can be solved in linear time, there are other graphs which have $O(2^n)$ different minimal vertex cover. In the next section, we will introduce a different algorithm which uses twin vertices to skip valid capacitated covers and solves the star graph in linear time.

4.4 Twin vertices

In this section we will introduce another algorithm to solve the capacitated vertex cover. It is based on the observation that vertices with the same neighborhood, twin vertices, can be added in order of capacity. Guo et al. [6] used the same observation to obtain a FPT algorithm with respect to the solution size k in $O^*(1.2^{k^2})$ time.

Definition 11. *Given a graph $G = (V, E)$. Two vertices $x, y \in V$ are called* twin *vertices if they have the same neighborhood. If x and y are neighbors, they are called true twins, if they are not neighbors, they are called false twins.*

The next Lemma shows the main argument used in this algorithm.

Lemma 9. *Given a capacitated vertex cover problem on a graph $G = (V, E)$ with capacities $c(v)$. If there exists a minimum capacitated vertex cover C, such that two twin vertices $x \in C$ and $y \notin C$ have $c(x) \leq c(y)$, then there exists a minimum capacitated vertex cover C' such that $x \notin C'$ and $y \in C'$.*

Proof. To construct C' we simply switch vertex x in C with y. Thus $C' = \{y\} \cup C \setminus \{x\}$ for which we will prove that it is also a minimum capacitated vertex cover.

First we note that $|C'| = |C|$, thus it has the minimum size. Because C is a valid cover and $y \notin C$, we know that all edges with y as endpoint are covered by the vertices in $N(y)$. Because $N(x) = N(y)$, in C' we can use these capacities to cover all the edges with x as endpoint. On the other hand, in C vertex x has covered at most $c(x)$ edges. The rest of the edges is again covered by $N(x)$. In C' the vertex y can cover the at least the $c(x)$ corresponding edges because $c(y) \geq c(x)$. The other edges can be cover by the same vertices as in C. We note that it doesn't matter whether x and y are true or false twins. The switch can be performed in both cases. □

The above Lemma allows us to introduce a new branching rule. If we have two twin vertices x and y with $c(x) \geq c(y)$, we do not have to check the cover where y is included and x is excluded, if we already check the cover with x and without y. We can extend this observation to a set of twin vertices, which are all twin to each other. If we have a set of twin vertices, it only matter how many vertices from that set we pick. By Lemma 9 we can pick the vertices with the most capacity.

This reduces the complexity for a set of twin vertices from exponential time to linear time. The algorithm is now straight forward. For every set of twin vertices, it tries all possibilities for the twin vertices in order of their capacity.

The algorithm branches on the twin vertices faster than a trivial brute force. But for every branch without any twin vertices left, the remaining graph will be brute forced. Therefore, the runtime will only break the $O^*(2^n)$ barrier if we have a significant amount, ϵn for some $\epsilon > 0$, of twin vertices. Before we can analyze the final runtime, we will need the following Lemma.

Lemma 10. *Given a $m \geq 2$ and a constant k. Over all sets of positive numbers $x_1, \ldots, x_m$ such that $\sum_{i=1}^{m} x_i = k$, the maximum value of $\prod_{i=1}^{m}(x_i + 1)$ is obtained for the set where $x_1 = \ldots = x_m = \frac{k}{m}$.*

Proof. Proof by induction over m.

Base case: We have that $m = 2$ and $x_1 + x_2 = k$, which gives $x_2 = k - x_1$. We want to maximize the product $(x_1 + 1)(x_2 + 1)$. Rewriting the product gives us

$$(x_1 + 1)(x_2 + 1) = x_1 \cdot x_2 + x_1 + x_2 + 1 = x_1 \cdot (k - x_1) + k + 1 \tag{3}$$

To find the maximum value, we take the derivative to x_1 and set it equal to zero:

$$\begin{aligned} k - 2x_1 &= 0 \\ x_1 &= \frac{k}{2} \end{aligned} \tag{4}$$

which gives us that the optimal is obtained if $x_1 = k - x_2 = \frac{k}{2}$, and thus $x_2 = \frac{k}{2}$.

Induction step: Assume the lemma holds for some $q \geq 2$. We are going to prove it also holds for $q + 1 \geq 3$.

Assume we have a set of positive numbers $x_1, \ldots, x_{q+1}$. To maximize the product we also need to maximize the product $\Pi_{i=1}^{q} x_i + 1$ with $\sum_{i=1}^{q} x_i = k - x_{q+1}$. By our induction hypothesis, we know that the values $x_1, \ldots, x_q$ must be equal. The same way we can show that $x_2, \ldots, x_{q+1}$ must be equal. Because $q + 1 \geq 3$ we have that $|\{x_1, \ldots, x_q\} \cap \{x_2, \ldots, x_{q+1}\}| \geq 1$. In particular, we get that $x_1 = x_2 = \ldots = x_{q+1} = \frac{k}{q+1}$. □

Now we can prove the running time of the algorithm.

Theorem 9. *Given a graph G with ϵn twin vertices for some $\epsilon > 0$, not necessarily all twin to each other. We can solve the capacitated vertex cover problem in $O^*(1.737^{\epsilon n} 2^{(1-\epsilon)n})$ time.*

Proof. Let $X_1, \ldots, X_m$ be the sets of all twin vertices. A set X_i contains all vertices $v, w \in V$ such that $N(v) = N(w)$. Let $k = \epsilon n = \sum_{i=1}^{m} |X_i|$. For each set we will try all possible combinations of vertices for that set. By Lemma 9, we can add the vertices in order of capacity. This gives that any set X_i has $|X_i| + 1$ possibilities. Thus the total amount of possibilities for all sets is equal to $\prod_{i=1}^{m} |X_i| + 1$. We also have that $\sum_{i=1}^{m} |X_i| = k$. By Lemma 10 the maximum amount of different possibilities we have to try over all the twin vertices is obtained when $|X_1| = \ldots = |X_m| = \frac{k}{m}$. This gives an upper bound to the maximum number of combination of:

$$(\frac{k}{m} + 1)^m = (\frac{k+m}{m})^m = l^{\frac{k+m}{l}}, \text{ where } l = \frac{k+m}{m}. \tag{5}$$

Because we only remove twin vertex sets if we have at least two vertices, we have that $m \leq \frac{k}{2}$, thus this can be bounded by $l^{\frac{1.5k}{l}}$. This value takes a maximum for $l = e$ and attains a value of $1.44^{1.5k} = 1.737^k$.

Thus trying every possible combination of twin vertices takes $O^*(1.737^k) = O^*(1.737^{\epsilon n})$ time. The rest of the graph will be brute forced for a final running time of $O^*(1.737^{\epsilon n} 2^{(1-\epsilon)n})$ time.

□

4.5 Capacity and edge restrictions

Thus far we have seen several separate algorithms to solve the capacitated vertex cover problem on specific graph classes. In this section we will combine the matching algorithm with the pathwidth algorithm to solve the capacitated vertex cover in more cases. In particular, we will look at instance of the CVC who have their capacities bounded by some constant k, followed by the CVC problem on graphs with restrictions on the number of edges. Lastly, we will briefly look at the connection between the number of edges and the total amount of capacity in a graph and see under what circumstances we can break the $O^*(2^n)$ time barrier.

First we will introduce a Lemma from which we can derive a connection between the size of a maximal matching and the pathwidth of a graph.

Lemma 11. *For any graph G with a maximal matching M, there exists an independent set of size at least $n - 2|M|$.*

Proof. Given a graph G and let M be a maximal matching of G. If any two vertices, which are not an endpoint of an edge in M, are connected to each other, we can add that edge to the matching and create a larger matching M', which is a contradiction with the assumption that M is maximal. Therefor, all these vertices must form an independent set and there are exactly $n - 2|M|$ such vertices.

To show that this is a tight bound we will look at the graph which consist of arbitrarily many disjoint triangles. For the maximal matching we can take one edge from each triangle and for the independent set we also can take at most one vertex per triangle. This results in a maximal matching M of size $\frac{n}{3}$ and a maximum independent set I of size $\frac{n}{3}$. In particular we have that $|I| = n - 2|M|$. □

With this Lemma we can always create a path decomposition of width at most $2|M|$. First we create a bag which contains all the vertices in M. As before in Lemma 11, the rest of the vertices must form an independent set. We can add and remove all the vertices of the independent set to the decomposition one by one, which will create a path decomposition of width $2|M|$.

Corollary 9. *For any graph G with a maximal matching M, the pathwidth of G is at most $2|M|$.*

This observation allows us to first check a graph G for a maximum matching of size $\geq \epsilon n$. If such a matching exists, we will use the matching algorithm to solve the capacitated vertex cover problem. Otherwise, by Corollary 9, we know that the pathwidth is at most $2\epsilon n$. This observation holds for any $\epsilon > 0$. The next couple of Lemma's will use this observation and choose the $\epsilon > 0$ carefully in order to solve the CVC problem on some more specific graph classes in $O^*(c^n)$ time for some constant $c < 2$.

We start with an algorithm for graphs whos capacities are bounded by some constant k.

Lemma 12. *Given an instance of capacitated vertex cover on a graph $G = (V, E)$, whose capacities $c(v)$ are bounded by some constant $k \geq 2$. We can solve the problem in $O^*(2^{\frac{1}{1-log_{k+1}(\frac{1}{2}\sqrt{3})}n})$ time.*

Proof. First we check if G contains a matching of size δn for a yet to be determined δ. If G contains such a matching, we solve the problem in $O^*(\sqrt{3}^{2\delta n}2^{(1-2\delta)n})$ time using Theorem 7. Otherwise, we know by Corollary 9 that G has a pathwidth of at most $2\delta n$. By the assumption that every vertex $v \in V$ has a capacity of at most k, we can use the pathwidth algorithm in $O^*((k+1)^{2\delta n})$ time.

The total running time of this algorithm will be the maximum of the matching runtime and the pathwidth runtime.

To optimize the runtime, we will choose δ such that the runtime of the matching algorithm and the pathwidth algorithm are equal.

$$\begin{aligned}
(k+1)^{2\delta} &= \sqrt{3}^{2\delta} 2^{1-2\delta} \\
(k+1)^{2\delta} &= 2(\frac{1}{2}\sqrt{3})^{2\delta} \\
k+1 &= 2^{\frac{1}{2\delta}} \frac{1}{2}\sqrt{3} \\
2^{\frac{1}{2\delta}} &= \frac{k+1}{\frac{1}{2}\sqrt{3}} \\
\frac{1}{2\delta} &= \log \frac{k+1}{\frac{1}{2}\sqrt{3}} \\
2\delta &= \frac{1}{\log \frac{k+1}{\frac{1}{2}\sqrt{3}}} \\
\delta &= \frac{1}{2\log \frac{k+1}{\frac{1}{2}\sqrt{3}}}
\end{aligned}$$

We can fill this δ into one of the sides to obtain the final running time.

$$\begin{aligned}
(k+1)^{2\frac{1}{2log(\frac{k+1}{\frac{1}{2}\sqrt{3}})}n} &= (k+1)^{\frac{1}{log(\frac{k+1}{\frac{1}{2}\sqrt{3}})}n} \\
&= 2^{log(k+1)\frac{1}{log(k+1)-log(\frac{1}{2}\sqrt{3})}n} \\
&= 2^{\frac{1}{1-\frac{log(\frac{1}{2}\sqrt{3})}{log(k+1)}}n} \\
&= 2^{\frac{1}{1-log_{k+1}(\frac{1}{2}\sqrt{3})}n}
\end{aligned}$$

We note that for any $k \geq 2$, this is of the form $O^*((2-\epsilon)^n)$ for some $\epsilon > 0$.

□

Next we will look at graphs with restrictions on the number of edges. We start with graphs who have $> \epsilon n^2$ edges for some $\epsilon > 0$. We will show that these graphs always have a significantly sized matching of size at least $\frac{1}{2}\epsilon n$ and that we can use the matching algorithm. Assume we would not have such a sized matching. By Lemma 11 we have an independent set of size at least $(1-2\epsilon)n$. The total amount of edges we can have in such a graph is equal to $\epsilon n(1-\epsilon)n + (\epsilon n)^2 = \epsilon n^2$. We note that we had strictly more edges, thus a contradiction and we must have a matching of size $\frac{1}{2}\epsilon n$. This gives us the following corollary.

Corollary 10. *The capacitated vertex cover problem on a graph $G = (V, E)$ with $|E| > \epsilon n^2$, for some $\epsilon > 0$, can be solved in $O^*(\sqrt{3}^{\epsilon n} 2^{(1-\epsilon)n})$ time.*

Next we will look at sparse graphs with $\leq cn$ edges, for some constant $c \geq 1$. The restriction on the number of edges allows us to improve the running time of the pathwidth algorithm. Theorem 3 states a running time of $O^*((m+1)^{pw})$, where m is the maximum capacity in the graph. We will show that we can reduce this m in the runtime to a constant which is dependent on c, which allows us to choose an $\delta > 0$ for the matching algorithm such that the final runtime is of the form $O^*((2-\epsilon)^n)$ time.

Lemma 13. *The capacitated vertex cover problem on a graph $G = (V, E)$ with $|E| \leq cn$ for some constant $c \geq 1$ can be solved in $O^*((2-\epsilon)^n)$ time.*

Proof. First we check if we have a matching of size $\frac{1}{2}\delta n$. The exact value of δ will be determined later. If such a matching exists, we use the matching algorithm of Theorem 7. Otherwise, we will use the pathwidth algorithm of Theorem 3.

We note that we have at most $2cn$ total capacity in the entire graph. By Corollary 9 every bag in the path decomposition has a size of at most δn . This allows us to bound the size of the table $c_i(f)$ we have to calculate for each bag. We are going to use these bounds to bound the maximum amount of capacity functions per bag.

Let X_i be a bag. The total amount of capacity functions is equal to $\prod_{x_i \in X_i}(c(x_i)+1)$ under the constraint $\sum_{x_i \in X_i} c(v) \leq 2cn$ and $|X_i| \leq \delta n$. By Lemma 10 we know that the maximum of the product is obtained if every vertex has the same maximum capacity. This results in a table size of $(\frac{2cn}{|X_i|}+1)^{|X_i|}$ per bag X_i. We note that this is an increasing function with respect to $|X_i|$. Therefore, it takes a maximum value if $|X_i|$ is as large as possible. We note that the maximum value for $|X_i|$ is δn. The runtime of our pathwidth algorithm becomes $O^*((\frac{2cn}{\delta n}+1)^{\delta n}) = O^*((\frac{2c}{\delta}+1)^{\delta n})$ time. For any constant c we can find a small enough constant δ such that $(\frac{2c}{\delta}+1)^{\delta} < 2$.

□

For a given instance of the CVC problem, the next two Lemmas give a relation between the total amount of capacity in the instance and the number of edges in the graph. The maximum amount of capacity in a graph G is obtained when all the vertices have their degree as capacity. This results in a total capacity of $2|E|$. We will show that if we have near maximum capacity, the problem is similar to the normal vertex cover problem, in which case we can break the $O^*(2^n)$ time barrier.

Lemma 14. *Given an instance of the capacitated vertex cover problem on a graph $G = (V, E)$, such that $\sum_{v \in V} c(v) - |E| \geq |E| - cn$ for some constant $c < \frac{1}{2}$. We can solve the capacitated vertex cover problem in $O^*((2-\epsilon)^n)$ time.*

Proof. First we check if we have a matching of size at least δn with $\delta < \frac{1}{2} - c$. If it exists, we solve the problem with the matching algorithm of Theorem 7. Otherwise, by Corollary 11, G has an independent set of size at least $n - 2\delta n$.

In this graph we can enumerate all possible minimal vertex covers in $O^*(2^{2\delta n})$ time by brute forcing the vertices in the maximum matching M. For every configuration of M, we can complete the vertex cover in polynomial time by adding the vertices in the independent set which are connected to an excluded vertex.

For every minimal vertex cover, we will add excluded vertices to find the smallest capacitated vertex cover which contains the minimal vertex cover. For a minimum capacitated vertex cover, every vertex in the cover will cover at least one edge. We note that by our assumption, a minimal vertex cover misses at most cn capacity to cover all the edges. Therefore, we have to add at most

cn extra vertices to the minimal vertex cover to create a capacitated vertex cover. We try every such possibility per minimal vertex cover for a total running time of $O^*(2^{2\delta n}\binom{n-2\delta n}{cn})$ By Lemma 6 and our choice of δ, this is of the form $O^*((2-\epsilon)^n)$ for some $\epsilon > 0$. □

We can make a similar argument for when we have just enough capacity for all the edges. The minimum amount of capacity we need is equal to the number of edges in the graph. If we have only slightly more, we can show that we need almost all vertices in the cover, which results in an algorithm which breaks the $O^*(2^n)$ time barrier.

Lemma 15. *Given an instance of the capacitated vertex cover problem on a graph $G = (V, E)$ with $\sum_{v \in V} c(v) - |E| \leq cn$ for some constant $c < \frac{1}{2}$. We can solve this instance of the capacitated vertex cover problem in $O^*((2-\epsilon)^n)$ time for some $\epsilon > 0$.*

Proof. We note that we need almost all vertices to cover all the edges. We start with all the vertices in the capacitated cover. For every vertex we leave out, the total amount of capacity in the cover decreases with at least one. The cover need to have at least $|E|$ capacity in total. This means that by our assumption we can leave out at most cn vertices, which gives a running time of $O^*(\binom{n}{cn})$. We note that by Lemma 6, for any $c < \frac{1}{2}$ this has a running time of $O^*((2-\epsilon)^n)$ for some $\epsilon > 0$. □

5 A discussion on the general case

We have tried to find an algorithm which solves the general CVC problem in $O^*((2-\epsilon)^n)$ time, but have not found such an algorithm. In Section 5.1 we will discuss the cases of the CVC problem we cannot solve with the results of this paper and argue why these cases are difficult with some proofs with respect to the Strong Exponential Time Hypothesis. After that in Section 5.2 we will introduce notation and definitions we used to try and solve the general case. We use this to make a faster polynomial time check which checks whether a given subset is a valid capacitated cover. Finally in Section 5.3 will use the theory of Section 5.2 to show a reduction rule which allows us to determine in polynomial time whether a capacity one vertex has to be included in a minimum capacitated vertex cover. We will use this rule to solve the CVC problem on graphs of bounded degree 2 in polynomial time.

5.1 Difficult cases

Here we will discuss the instances of the CVC problem we cannot solve in $O^*((2-\epsilon)^n)$ time yet and argue why these cases are difficult.

From our results, a promising begin to a general algorithm would be to use the matching algorithm. First we check for a matching of size $\geq \delta n$ and if it exists, we use the matching algorithm. If it does not exists, by Lemma 11 we get an independent set of size at least $n - 2\delta n$. We note that this holds for any $\delta > 0$. These remaining graphs are the graphs we cannot solve with the results in this paper. In particular, for any $\delta > 0$, the CVC problem on graphs which contain an independent set of size at least $(1-\delta)n$ cannot be solved in $O^*((2-\epsilon)^n)$ time with the results of this paper. We note that this includes bipartite graphs where one side has $\leq \delta n$ vertices and that we cannot solve the CVC problem on bipartite graphs in general.

Because the graphs contain an independent set of size at least $n-2\delta n$ for any $\delta > 0$, a promising begin to a general algorithm would be to brute force the vertices outside the independent set and per configuration we will decide the smallest amount of vertices from the independent set we need

to add to get a capacitated vertex cover. The brute force will take $O^*(2^{2\delta n})$ time for an arbitrarily small $\delta > 0$, which leaves a lot of room to solve the remaining problem with the independent set. We will show that this remaining is not as easy as it looks. First we define an extension to the capacitated vertex cover problem, namely the Partial Capacitated Vertex Cover (PCVC) problem.

Definition 12 (Partial Capacitated Vertex Cover problem)**.** *Given a graph $G = (V, E)$ with capacities $c(v)$ and an integer k. Given a subset of vertices $V' \subseteq V$ such that $I = V \setminus V'$ is an independent set. For the vertices $v \in V'$ it is decided whether v is included or excluded from the capacitated vertex cover. The goal is to decide whether there exists a capacitated vertex cover, given the configuration of V', of size at most k. The runtime is calculated with respect to the amount of undecided vertices $|I|$.*

We will show that if the Strong Exponential Time Hypothesis is true, we cannot solve the PCVC problem in $O^*((2-\epsilon)^n)$ time for any $\epsilon > 0$. We first define the Strong Exponential Time Hypothesis.

Definition 13 (Strong Exponential Time Hypothesis)**.** *There does not exists an $\epsilon > 0$ such that SAT in general can be solved in $O^*((2-\epsilon)^n)$ time.*

The SETH can be used in a similar way as the ETH. If the SETH is true and we can make reduction from SAT to the CVC problem of size n, we can conclude that the PCVC problem cannot be solved in $O^*((2-\epsilon)^n)$ time.

We have already seen a reduction from SAT to the CVC problem in Lemma 3. We can extend this reduction to make an instance of the PCVC problem. For a given instance of SAT, we make the reduced graph $G' = (V, E)$ as described in Lemma 3. For the PCVC problem, we make the predefined set V' equal to all the forced vertices in G'. We note that the remaining vertices form an independent set I of size $2n$. Now if the SETH is true, we cannot solve the PCVC problem in $O^*((2-\epsilon)^n) = O^*((2-\epsilon)^{\frac{1}{2}|I|}) = O^*(\sqrt{2-\epsilon}^{|I|})$ time for any $\epsilon > 0$.

We can improve this bound if we make a reduction from the Hitting Set problem.

Definition 14 (Hitting Set Problem)**.** *Given a universe $\mathbf{U} = \{u_1, \ldots, u_n\}$, a family of sets $\mathbf{S} = \{s_1, \ldots, s_m\}$ with $s_i \subseteq \mathbf{U}$ and an integer k. We are asked to determine if there exists a subset $U \subseteq \mathbf{U}$ such that $\forall s_i \in \mathbf{S} : s_i \cap U \neq \emptyset$ with $|U| \leq k$. The runtime is measured with respect to the size of the universe.*

A trivial algorithm for the Hitting Set problem tries every possible subset $U \subseteq \mathbf{U}$. This takes $O^*(2^n)$ time. Cygan et al.[16] have shown that if the SETH holds, Hitting Set cannot be solved in $O^*((2-\epsilon)^n)$ time for any $\epsilon > 0$.

Theorem 10. *If the SETH is true, the Hitting Set problem cannot be solved in $O^*((2-\epsilon)^n)$ time for any $\epsilon > 0$. [16]*

We will make a reduction from Hitting Set to the PCVC problem of size n. With this reduction and by Theorem 10, we can conclude that if the SETH is true, the PCVC problem cannot be solved in $O^*((2-\epsilon)^n)$ time for any $\epsilon > 0$.

Lemma 16. *If the SETH is true, the partial capacitated vertex cover problem cannot be solved in $O^*((2-\epsilon)^{|I|})$ time for any $\epsilon > 0$.*

Proof. We will make a reduction from Hitting Set to an instance of the partial capacitated vertex cover problem on a bipartite graph $G = (X, Y, E)$. The reduction is similar to the SAT reduction of Lemma 3.

Given an instance of the Hitting Set problem. Let $\mathbf{U} = \{u_1, \ldots, u_n\}$ be the universe elements and let $\mathbf{S} = \{s_1, \ldots, s_m\}$ be the family of sets. We start by adding a zero capacity vertex g to X. For every set s_i we add a node to Y, which is connected to g. Every such node gets a capacity equal to their degree minus one. Finally, for every element u_i we add a node to X, and connect it to the set nodes $s_i \in Y$ for which $u_i \in s_i$. The element nodes $u_i \in X$ get maximal capacity. To complete the instance of the PCVC problem, we define the predefined vertex set V' as all the vertices in Y and the zero capacity vertex g. The remaining vertices $I = V \setminus V'$ consist of all the element vertices in X and form an independent set I of size n.

Now we will show that we have a solution to the hitting set problem of size at most k if and only if we have a minimum capacitated vertex cover of size $m + k$ on the graph G. First we note that any set node in Y is connected to the zero capacity vertex $g \in X$, and thus has to be included in the minimum capacitated cover. We note that the predefined vertices V' contain all these vertices in Y. This forces all the vertices in V' to be in the capacitated vertex cover. In particular, the only vertices which still need to be decided are the independent vertices $V \setminus V'$. We note that every set node in Y has a capacity equal to their degree minus one, thus they need at least one edge to be covered by a neighbor.

Assume we have a hitting set of size at most k. In addition to the forced nodes in V' described before, we can include the element nodes in X corresponding to the solution of size at most k in the capacitated cover. This results in a cover of size at most $m+k$. We saw before that to complete the capacitated vertex cover on G, every set node in Y needed one edge to be covered by its neighbor. Because the chosen vertices are a hitting set, each set node in Y has a neighbor which is included in the cover. Because the element vertices in X have maximum capacity, they can cover all their edges. Thus we have a capacitated vertex cover of size at most $m + k$ in G.

Now assume we have a capacitated vertex cover of size at most $m + k$. We note that we have m forced vertices in Y. Therefore, we have at most k vertices in X in the capacitated cover. We note that again every set vertex in Y must have at least one neighbor in the cover in order to form a capacitated vertex cover. In particular, the at most k selected vertices in X hit every node in Y. Thus the corresponding elements in $\mathbf{U}$ hit every set in $\mathbf{S}$ and we have a hitting set of size at most k.

By Theorem 10, if the SETH is true, the hitting set problem cannot be solved in $O^*((2-\epsilon)^n)$ time for any $\epsilon > 0$. Because we have a n sized reduction to the partial capacitated vertex cover problem, the PCVC problem can also not be solved in $O^*((2-\epsilon)^{|I|})$ time, if the SETH is true. □

Lemma 16 provides some evidence why the described algorithm in the beginning of this section will be difficult. After we have brute forced the vertices outside the independent set, we are left with an instance of the PCVC cover. By Lemma 16 we know that if the SETH is true, this cannot be solved in $O^*((2-\epsilon)^{|I|})$ time for any $\epsilon > 0$. Although the previous observation does not indicate that the algorithm is not possible at all, it does indicate it is not as trivial as it seems. A standard branching algorithm will encounter instances of the PCVC problem. In order to achieve a runtime of $O^*((2-\epsilon)^n)$ time, we probably need to use more sophisticated techniques, such as measure and conquer, to prove that even if the branching algorithm hits the difficult problems of the PCVC problem, the overall algorithm runs in $O^*((2-\epsilon)^n)$ time.

5.2 Capacity flow

Thus far we have mostly used the structure of a graph to develop algorithms. In this section we will look at the capacities of vertices. With the following theory we tried to solve the CVC problem in general in $O^(2-\epsilon)^n)$ time. Although we didn't solve the general case, we can use the theory to improve the running time of the polynomial time check to test whether a given subset is a valid capacitated vertex cover. In Section 5.3 we will show a reduction rule which decides in polynomial time whether a capacity one vertex has to be included in the minimum capacitated vertex cover.

An important concept for this theory is the edge assignment function. This is a function which assigns to each edge in the graph a vertex in the cover which covers the edge. We will start with the definition.

Definition 15. *Given a graph $G = (V, E)$ with capacities $c(v)$ and a capacitated vertex cover C. An edge assignment function $\gamma_C : E \to C$ is a function which assigns to every edge in the graph a vertex from the cover. We say that a vertex v* covers *an edge e if $\gamma_C(e) = v$. A function γ_C is a valid edge assignment function if $\forall e \in E : \gamma_C(e)$ is an endpoint of e and $\forall v \in C : |\gamma_C^{-1}(v)| \leq c(v)$. When the cover is clear from context, we will omit it in notation.*

When we have a cover C with an edge assignment γ_C, the amount of capacity a vertex $v \in C$ uses to cover edges is called the *used* capacity.

Definition 16. *Let C be a capacitated vertex cover with an edge assignment γ_C. The* used capacity *of a vertex $v \in C$ is defined as $uc(v) = |\gamma_C^{-1}(v)|$ and indicates the amount of capacity used by v to cover edges in the graph.*

It may happen that the total amount of capacity in the cover is larger than the number of edges it has to cover. In this case, for an edge assignment γ_C, some vertices have unused capacity. We will call this *overcapacity*.

Definition 17. *Let C be a capacitated vertex cover with an edge assignment γ_C. The* overcapacity *of a vertex $v \in C$ is defined as $oc(v) = c(v) - |\gamma_C^{-1}(v)|$ and indicates the amount of capacity not used by v to cover edges.*

For a cover C with an edge assignment γ_C, this overflow can be moved through the graph by means of *edge flips*. Given a vertex $v \in C$ with $oc(v) > 0$. Now v must have an incident edge which is covered by a neighbor $w \in C$. Because v has an overcapacity, it could cover this edge instead of w. When we do this, we say we *flip* the assignment of the edge to the other endpoint. This results in vertex v losing an overcapacity and vertex w gaining one overcapacity. Effectively, we have moved an overcapacity from v to w. We can repeat this procedure and move the overcapacity from w to another vertex and so forth.

We are going to introduce some notation to formalize this idea.

Definition 18. *Given a graph $G = (V, E)$ and a capacitated vertex cover C with an edge assignment function γ_C. With an* edge flip *we change the vertex assignment $\gamma_C(e)$ of an edge $e \in E$ to the other endpoint and create a new assignment γ'_C. When both γ_C and γ'_C are valid edge assignments, we call the edge flip* valid*, otherwise, it is called* invalid*.*

We will mostly focus on valid edge flips. For a given cover C with an edge assignment function γ_C, we can encode every possible valid edge flip in a flow network. We start with a definition.

Definition 19. *Given a graph $G = (V, E)$ and a cover C with an edge assignment γ_C. We define the directed flow graph G_{γ_C} of G as follows. Let $V_{G_{\gamma_C}} = V$ and for every edge $vw = e \in E$, we add an arc in G_{γ_C} between v and w pointing towards $\gamma_C(e)$.*

In the flow graph G_{γ_C} of a graph G for a cover C, each arc corresponds to an edge in G and points towards the vertex which covers this edge. This allows us to move the overcapacity through the graph. If we want to check if a vertex $v \in C$ with $oc(v) > 0$ can send an overflow to a vertex $w \in C$, we only have to check if there exists a path in the flow graph G_{γ_C} from v to w. If such a path exists, we can flip every edge on the path which effectively moves an overcapacity from v to w. We note that after flipping all the edges on the path, the nodes lying on the path between v and w will have used the same capacity as before. The only nodes which have their total used capacity changed are v and w.

We can perform this check with a max flow. For this we will set the capacity of every arc in G_{γ_C} to 1. If we set v to be the source and w to be the sink, we check for more than one overcapacity at once. Each flow in the graph G_{γ_C} corresponds to a series of edge flips. We note that because the capacity of each arc is one, we cannot get a double edge flip. Because the flow is in an integer flow network, we get an integer flow. Now if there exists an integer max flow of size k from v to w in G_{γ_C}, we can move k overflow from v to w by means of edge flips.

Besides edge flips, we can also perform cycle flips. If we have an edge assignment γ for which the flow graph G_γ contains a cycle, we can change the direction of the cycle.

Definition 20. *Given a graph G and a capacitated vertex cover C with an edge assignment γ_C. If G_{γ_C} contains a cycle, we can perform a* cycle flip*. When we perform a cycle flip, we change the direction of a cycle in G_{γ_C} and create a new edge assignment γ'_C. Note that if γ_C is a valid assignment, the new edge assignment γ'_C is also a valid assignment.*

The next Lemma states that for any given cover C with edge assignment γ_C, we can make any other valid edge assignment γ'_C by applying a set of cycle flips followed by valid edge flips. In other words, we can reach any other valid edge assignment by these two operations without invalidating the edge assignment somewhere along the line.

Lemma 17. *Given a graph G and a capacitated vertex cover $C \subseteq V$. For any two edge assignments γ and γ', there exists an order of cycle flips followed by valid edges flips such that those operation applied to γ give γ'.*

Proof. We note that there always exists a set of (not necessarily valid) edge flips $S = \{e_1, \ldots, e_m\}$, such that these applied to γ result in γ'. Each edge is flipped at most once in this set. We are going to create an order of operations F, consisting of cycle flips followed by valid edge flips, from this set. The operation of F applied to γ will give us γ'.

First we are going to convert any cycle c in G with $\forall e \in c : e \in S$ to a cycle operation c_j. We remove the edge flips from S and include a corresponding cycle operation in F. Repeat until all cycles in S are removed.

Next, we will cover the graph G_γ with directed paths, such that each path corresponds with an overflow move from one vertex to another. For this we create the paths in G_γ from the remaining edge flips in S. We start with a single edge path for each of the remaining flips. If there are two paths p and q such the the endpoint of p is the startpoint of q, we merge the paths together by attaching path q to the end of p. Repeat this step until no more path merges are possible. For every resulting path, we add the corresponding edge flips in path order to F. We will now show that F consist of only cycle flips and valid edge flips.

We only have to show that all the edge flips are valid. For each created path, it will effectively move an overcapacity from the starting point to the endpoint, relative to the assignment γ. Because the operation of F applies to γ give γ', which are both valid edge assignments, and because there are no more paths p and q with the startpoint of q as endpoint of p, the startpoint of any remaining path must have an overflow in γ. Because we added all the edge flips in path order, each flip will be a valid edge flip. In particular, a path of edge flips will never contain an invalid edge flip. Therefore, all the edge flips in F are valid and we have constructed an order of cycle flips and valid edge flips to create the edge assignment γ' from γ. □

The flow graph and Lemma 17 allows us to check whether a given set $V' \subseteq V$ is a valid capacitated vertex cover using a max flow on a graph of with $O(n)$ vertices. This requires an initial edge assignment γ for the cover $C = V$. We note that any edge assignment for any cover is also a valid edge assignment for $C = V$, thus any possible valid edge assignment will do.

We note that this check has a better runtime than the check given in Lemma 1, which performed a max flow over a graph with $O(n+m)$ vertices, where m is the amount of edges in the CVC graph.

Lemma 18. *Given a instance of the CVC problem on a graph $G = (V, E)$. If we have an edge assignment γ for the cover $C = V$, we can check whether a given subset $V' \subseteq V$ is a valid capacitated vertex cover with a max flow on a graph of with $O(n)$ vertices.*

Proof. We will construct a flow graph G' over G_γ. Any arcs over which a flow will be send corresponds to an edge flip in γ. The graph will be constructed in a way such that the max flow will try to move the available overflow in the cover V' to all the vertices $v \notin V'$. In other words, we try to make an edge assignment γ' for which none of the vertices $v \notin V'$ use a capacity. If such an assignment exists, V' is a valid capacitated cover.

First we construct the flow graph $G_\gamma = (V_\gamma, A_\gamma)$. Next we add the source and the sink. For every vertex $v \notin V'$, we add an arc to the sink with a capacity equal to $|\gamma^{-1}(v)|$. Any flow which goes through such an arc corresponds to a capacity which is no longer used by v in the new edge assignment γ'. Next for any $v \in V'$ we add an arc from the source to v. The arc gets a capacity of $c(v) - |\gamma^{-1}(v)|$. Each flow over this arcs correspond to an edge cover by v which uses a capacity unused in the edge assignment γ.

Now V' is a valid cover if we have a max flow equal to $\sum_{v \notin V'} |\gamma^{-1}(v)|$. If we have such a flow, we know that all the capacity used by the vertices not in V' in the original assignment γ are unused in the new assignment γ'. To show that such a flow exists if and only if V' is a valid capacitated cover, we will use Lemma 17.

Assume V' is a valid capacitated vertex cover. There exists a valid edge assignment $\gamma_{V'}$ for the cover V'. We note that this assignment is also a valid assignment for the cover $C = V$. By Lemma 17 we can go from the assignment γ to $\gamma_{V'}$ by a series of cycle flips followed by valid edge flips. Next we note that any integer flow in the graph G_γ corresponds to a combination of cycle flips and valid edge flips. In particular, the series of cycle flips and valid edge flips from Lemma 17 corresponds to a valid flow in G' and this flow must have a size of $\sum_{v \notin V'} |\gamma^{-1}(v)|$. We note that the max flow will be able find this flow if it exists.

Now assume V' is not a valid capacitated vertex cover. We note that there does not exists a flow size $\sum_{v \notin V'} |\gamma^{-1}(v)|$, otherwise V' would be a valid capacitated vertex cover.

We note that the constructed flow graph G' has size $n + 2 = O(n)$, with respect to the graph G. Thus the check performs a max flow over a graph with $O(n)$ vertices. □

5.3 Removing single capacity vertices

Next we will introduce a reduction rule which decides in polynomial time whether a capacity one vertex must be included in a minimum capacitated cover. Let v be a vertex with capacity one. If v is part of a minimum capacitated cover C, we will try to switch it with a different vertex $w \notin C$, by including w in the cover and removing v. If this is possible, we have a minimum capacitated cover without v. We will show that it is possible to decide in polynomial time whether such a switch exists, without actually knowing the minimum capacitated cover C.

If we have a minimum capacitated vertex cover C, with a vertex $v \in C$ with capacity one, and an edge assignment γ_C, we can check if there exists a $w \notin C$ which can be switched with v in C using the flow graph G_{γ_C}. For each vertex $w \notin C$ with $c(w) > 0$, we check if there exists a path from w to v in G_{γ_C}. If there exists such a path, we can add w to the cover and send an overflow from w to v. In the new assignment v will have zero used capacity. In particular, after adding w to C we can remove v from C.

The next Lemma shows that if there is an edge assignment γ_C such that there exists a $w \notin C$ which can be switched with v, then there exists a $w' \notin C$ which can be switched with v for every valid edge assignment γ'_C.

Lemma 19. *Given an instance of the capacitated vertex cover on a graph $G = (V, E)$ with capacities $c(v)$ and a capacitated vertex cover D with edge assignment γ_D. Assume we have a vertex $v \in D$ with capacity 1, such that the set $D \setminus \{v\}$ is not a valid capacitated cover. If there exists a vertex $w \notin D$ with $c(w) > 0$ such that there exists a path in G_{γ_D} from w to v, then for all possible valid edge assignments γ'_D of the cover D, there exists a $w' \notin D$ with $c(w') > 0$ which has a path from w' to v in $G_{\gamma'_D}$.*

Proof. Assume we have a capacitated vertex cover D with $v \in D$ such that v cannot be removed from D without invalidating the cover. Thus $D \setminus \{v\}$ is not a valid capacitated vertex cover. Assume that for an edge assignment γ_D, there does not exists a $w \notin D$ which can be switched with v. We will show that there does not exists such a $w \notin D$ for any edge assignment γ'_D.

Assume that there exists a different edge assignment γ'_D where there does exists a $w \notin D$ such that we can switch v with w. We will show by contradiction that such an assignment γ'_D cannot exists.

By Lemma 17 there exists a set of cycle flips followed by valid edge flips such that those operation applied to γ_D gives us γ'_D. We note that in the flow graph of γ'_D there must exists a path from w to v. Let e_i be the last edge flip performed on that path from w to v. Because e_i is a valid edge flip, it will effectively move one overcapacity from one of endpoints to the other. In particular, there exists an overflow on the path from w to v. But because e_i was the last edge flip performed on the path from w to v, after flipping e_i, there must exists a path from e_i to v. In particular, we can move the overflow on the path all the way to v. After this, v will have zero used capacity and we can remove v from D without adding w. This is a contradiction with the assumption v could not be removed from D without invalidating the cover, thus the edge assignment γ'_D cannot exist.

We conclude that if there does not exists a $w \notin D$ which can be switched with v for some edge assignment γ_D, then any other possible edge assignment will also not have a $w \notin D$. □

The previous Lemma allows us to check only one edge assignment for a vertex $w \notin C$ which can be switched with v. Next we will show that if there does not exists a w who can switch with v, then it is not possible to remove v from the cover even if we added all vertices $w \notin D$ to the cover.

Lemma 20. *Given an instance of the capacitated vertex cover on a graph $G = (V, E)$ and given a capacitated cover D with $v \in D, c(v) = 1$ and such that $D \setminus \{v\}$ is not a valid capacitated vertex cover. If there does not exists a $w \notin D$ such that $D' = D \cup \{w\} \setminus \{v\}$ is a valid capacitated vertex cover, then the set $V \setminus \{v\}$ is not a valid capacitated vertex cover.*

Proof. Assume we have a cover D with a capacity one vertex $v \in D$ and v cannot be removed without invalidating D. Also assume that we have an edge assignment γ_D and that there does not exists a $w \notin D$ with $c(w) > 0$ which has a path from w to v in G_{γ_D}. If we add a random vertex $x \notin D$ to D, we get a new cover D' from which v cannot be removed without invalidating the cover. By Lemma 19, if there does not exists a $w \notin D'$ with $c(w) > 0$ which has a path to v in $G_{\gamma_{D'}}$ for some edge assignment $\gamma_{D'}$, it does not exists for any edge assignment for D'. Note that γ_D is still a valid edge assignment for D'. Thus we can use the same flow graph G_{γ_D} for the new cover D'. By our assumption, there does not exists a $w \notin D$ with $c(w) > 0$ which has a path to v in G_{γ_D}. Because we use the same flow graph, there will still be no $w \notin D'$ with $c(w) > 0$ which has a path from w to v. In particular, there is no $w \notin D'$ which can be switched with v in the new cover D'. We can repeat this argument until we have added all the vertices $w \notin D$ to the cover D and conclude that we cannot remove v from the cover consisting of all vertices V. □

The last Lemma allows us to state our reduction rule. We note that if we have a minimum capacitated vertex cover C which contains a vertex $v \in C$ with $c(v) = 1$, we can apply Lemma 20. We note that we cannot remove v from C without invalidating the cover, because C is a minimum cover. By Lemma 20, if there does not exists a vertex $w \notin C$ with $c(w) > 0$ which can be switched with v, then we cannot remove v from the cover consisting of all the vertices V. This last statement is easy to check. If we can remove v from the cover $D = V$, there must exists a $w \notin C$ which can be switched with v. If we cannot remove v, there is no possible cover without v and v has to be included in the minimum capacitated vertex cover.

Corollary 11. *Let $G = (V, E)$ be a graph. For any vertex $v \in V$ with $c(v) = 1$, we can decide in polynomial time whether there exists a minimum capacitated vertex cover C with $v \notin C$.*

Corollary 11 shows an interesting observation. The polynomial time check indicates that a capacity one vertex v is so inefficient in covering edges, that if there exists a capacitated cover without v, there must exists a minimum capacitated vertex cover without v.

Corollary 11 allows us to solve the capacitated vertex cover problem on graphs with bounded degree 2 in polynomial time.

Lemma 21. *Graphs with bounded degree 2 can be solved in polynomial time.*

Proof. These graphs consists of paths and cycles. Every line can be solved by repeatedly applying Corollary 11 to the endpoints. For the circles, if it contains a capacity one vertex, apply Theorem 11 and solve the resulting line as described. We only have a circle with only capacity two vertices left. This is equal to the normal vertex cover problem, which has a $\lceil \frac{k}{2} \rceil$ cover for a k length cycle. □

6 Conclusion

We have shown several NP-completeness proofs, showing that the capacitated vertex cover problem is NP-complete on bipartite graphs, on which the normal vertex cover problem can be solved in

polynomial time. We showed that if the ETH is true, the CVC problem on bipartite graphs cannot be solved in subexponential time.

We have introduced an algorithm for the CVC problem restricted to trees, which improves the time of Guha et al.[8] from $O(n \log n)$ time to $O(n)$ time. After that we showed a path- and treewidth algorithm. The treewidth algorithm runs in $O^*((k+1)^{tw})$ time where k is the solution size, which improved the time of Dom et al.[7].

We have introduced a series of algorithms which break the $O^*((2-\epsilon)^n)$ time barrier for instances of the capacitated vertex cover problem on specific graph classes. The main results were the matching algorithm and the twin vertices algorithm. We have combined the matching algorithm with the pathwidth algorithm to solve the CVC problem on more graph classes in $O((2-\epsilon)^n)$ time.

In the end we haven't found a general algorithm which solves the CVC problem in $O^*((2-\epsilon)^n)$ time. We have shown which cases of the CVC problem cannot be solved yet and argued why these cases are difficult. An algorithm which solves the CVC problem in general in $O^*((2-\epsilon)^n)$ time remains an open problem and is an interesting topic for future research. We have shown that the PCVC cannot be solved in $O^*((2-\epsilon)^n)$ time if the SETH is true. For future work it may be interesting to search for other variants of the CVC problem which are hard, which may give us the insight we need to find a general algorithm in $O^*((2-\epsilon)^n)$ time.

References

[1] Fomin, Fedor V., Fabrizio Grandoni, and Dieter Kratsch. *Measure and conquer: a simple $O(2^{0.288n})$ independent set algorithm* In Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm, pp. 18-25. Society for Industrial and Applied Mathematics, 2006.

[2] Fomin, Fedor V., Fabrizio Grandoni, and Dieter Kratsch. *Measure and conquer: domination–a case study.* In International Colloquium on Automata, Languages, and Programming, pp. 191-203. Springer, Berlin, Heidelberg, 2005.

[3] Fomin, Fedor V., Kazuo Iwama, Dieter Kratsch, Petteri Kaski, Mikko Koivisto, Lukasz Kowalik, Yoshio Okamoto, Johan van Rooij, and Ryan Williams. *08431 Open Problems–Moderately Exponential Time Algorithms.* In Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2008.

[4] Cygan, Marek, Marcin Pilipczuk, and Jakub Onufry Wojtaszczyk. *Capacitated domination faster than $O^*(2^n)$.* Scandinavian Workshop on Algorithm Theory. Springer, Berlin, Heidelberg, 2010.

[5] Liedloff, Mathieu, Ioan Todinca, and Yngve Villanger. *Solving capacitated dominating set by using covering by subsets and maximum matching.* International Workshop on Graph-Theoretic Concepts in Computer Science. Springer, Berlin, Heidelberg, 2010.

[6] Guo, Jiong, Rolf Niedermeier, and Sebastian Wernicke. *Parameterized complexity of generalized vertex cover problems.* Workshop on Algorithms and Data Structures. Springer, Berlin, Heidelberg, 2005.

[7] Dom, Michael, Daniel Lokshtanov, Saket Saurabh, and Yngve Villanger. *Capacitated domination and covering: A parameterized perspective.* International Workshop on Parameterized and Exact Computation. Springer, Berlin, Heidelberg, 2008.

[8] Guha, Sudipto, Refael Hassin, Samir Khuller, and Einat Or. *Capacitated vertex covering with applications.* Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, 2002.

[9] Garey, Michael R., and David S. Johnson. *The rectilinear Steiner tree problem is NP-complete.* SIAM Journal on Applied Mathematics 32, no. 4 (1977): 826-834.

[10] Fürer, Martin, and Balaji Raghavachari. *Approximating the minimum degree spanning tree to within one from the optimal degree.* In Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms, pp. 317-324. Society for Industrial and Applied Mathematics, 1992.

[11] Fomin, Fedor V., and Kjartan Høie. *Pathwidth of cubic graphs and exact algorithms.* Information Processing Letters 97, no. 5 (2006): 191-196.

[12] Jiang, Wei, Tian Liu, and Ke Xu. *Tractable feedback vertex sets in restricted bipartite graphs.* In International Conference on Combinatorial Optimization and Applications, pp. 424-434. Springer, Berlin, Heidelberg, 2011.

[13] Bandelt, H-J. *Hereditary modular graphs.* Combinatorica 8, no. 2 (1988): 149-157.

[14] Fomin, Fedor V., and Petteri Kaski. *Exact exponential algorithms.* Communications of the ACM 56, no. 3 (2013): 80-88.

[15] Henning, Michael A., and Anders Yeo. *Tight lower bounds on the size of a maximum matching in a regular graph.* Graphs and Combinatorics 23, no. 6 (2007): 647-657.

[16] Cygan, Marek, Holger Dell, Daniel Lokshtanov, Dániel Marx, Jesper Nederlof, Yoshio Okamoto, Ramamohan Paturi, Saket Saurabh, and Magnus Wahlström. *On problems as hard as CNF-SAT.* ACM Transactions on Algorithms (TALG) 12, no. 3 (2016): 41.

[17] Cygan, Marek, and Marcin Pilipczuk. *Exact and approximate bandwidth.* Theoretical Computer Science 411, no. 40-42 (2010): 3701-3713.