Partitioning of Binary Cactus Graphs with Two Weight Functions

Birgit Hillebrecht

A thesis presented for the degree of
Master of Science

to

Faculty of Mathematics and Computer Science
FernUniversität Hagen

on April 17, 2021

supervised by Prof. Winfried Hochstättler
in the subject area of discrete mathematics and optimization

Acknowledgement

Before heading to the content, I would like to thank my supervisor, Professor Winfried Hochstättler, for posing this interesting topic for my degree thesis, insightful discussions, which were invaluable for the successful investigation of the given problem, and the foundation created by various classes and seminars.

Contents

Revisions

[2021-07-18]

- Correction of definition 2.5: proper subgraph definition.
- Insertion of definition 2.23 and lemma 2.24 for usage in proof of lemma 2.25.
- Lemma 2.25:
  - renaming for readability: $r_C^{1,2}$ to $r_C, r_{C\prime}$
  - replaced erroneous line $\mathbb{P}(r, r_C) \cap \mathbb{P}(r, r_{C\prime}) \neq \emptyset$ by $\forall P \in \mathbb{P}(r, r_C)$ and $P\prime \in \mathbb{P}(r, r_{C\prime})$: $P \cap P\prime \neq \emptyset$
- Naming in definition 3.2 adjusted to avoid double indices.
- Theorem 4.1/4.2: Naming consistency of graph $T = (V, E)$ established

Chapter 1

Introduction

The general graph partitioning problem is part of many real-world applications, e.g., for logistic problems, numerical simulations, social studies [1] [2], parallelization of computations [3] or genome comparisons [4]. Research efforts are targeted to cover certain specializations of the problem such as balancing problems [5][6] or k-way cut problems [7] [8].
Buchin and Selbach [7] extended the algorithm proposed by Ito et al. [9] k-way to handle cactus trees in pseudopolynomial time. This strategy of augmenting a partitioning algorithm suitable for trees to one suitable for cactus trees is a promising approach for other algorithms devised only for trees.
Since the cactus tree is the first intuitive generalization of trees that admits only simple cycles in the graph under consideration, this approach is auspicious for providing a better understanding of graph partitioning problems for general graphs based on insights from tree partitioning.
Partitioning of trees under multiple constraints has been shown to be solvable in polynomial time for fixed outdegree of the tree and a fixed number of constraints [10]. The polynomial running time bound for binary trees under two constraints is especially promising for extension. In the following, the algorithm proposed by Hamacher et al. [10] will be extended for binary cactus trees.

Chapter 2

Preliminaries

To allow a concise investigation of the partitioning problem, a firm understanding of elementary concepts and notation is required. The concepts presented in this chapter are only a small subset of those required for a thorough understanding of graph theory but sufficient for subsequent studies. A complete introduction might be found in the book 'Graph Theory' by Diestel [11], which also serves as the foundation for the formulation of the following definitions.

2.1 Graphs

Definition 2.1. Graph A pair $G = (V, E)$ is called a graph when it consists of a set of vertices V and a set of edges $E \subset \{\{u, v\} \mid u, v \in V\}$. Thus, edges are unordered pairs of vertices. [1]

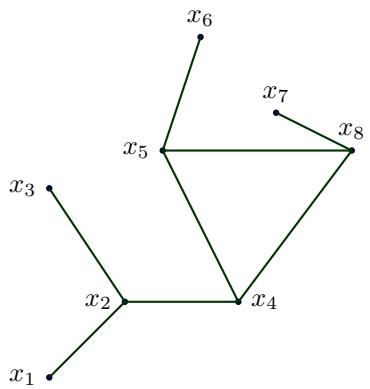


Figure 2.1: Example for a graph.

An example for a graph is shown in fig. 2.1. Therein, we have $G = (V, E)$ given by

$$V = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$$
$$E = \{\{x_1, x_2\}, \{x_2, x_3\}, \{x_2, x_4\}, \{x_4, x_5\}, \{x_4, x_8\}, \{x_5, x_6\}, \{x_5, x_8\}, \{x_7, x_8\}\}$$

[1] In general, one distinguishes between directed and undirected graphs. The definition here corresponds to an undirected graph since the edges are unordered pairs of vertices.

In the following, we generally assume that the graphs under consideration are non-empty, thus, $V \neq \emptyset$.

Definition 2.2. Incident Edges For a graph $G = (V, E)$ and an edge $e \in E$ with the endpoints $u, v \in V$ we call the edge e incident in u and v. We denote the set of all incident edges on a vertex as

$$E(v) := \{e \in E | e \text{ is incident in v}\}$$

Definition 2.3. Degree of a Vertex For a given vertex v in a graph $G = (V, E)$, we define the degree of v as

$$deg(v) := |E(v)|$$

Definition 2.4. Neighbor of a Vertex For a given vertex v in a graph $G = (V, E)$, we define the neighbors of v as those which have a common incident edge

$$N(v) := \{w \in V \mid \exists\, e \in E \text{ s.th. e is incident in v and w}\}$$

Returning to example 2.1, we find that for vertex x_4 the set of incident edges is given by

$$E(x_4) = \{\{x_2, x_4\}, \{x_4, x_5\}, \{x_4, x_8\}\}$$

so that $deg(x_4) = 3$. The neighbors are then given by $N(x_4) = \{x_2, x_5, x_8\}$.

Definition 2.5. Subgraph For a given graph $G = (V, E)$, we call a graph $G' = (V', E')$ a subgraph of G if $V' \subseteq V$ and $E' \subseteq E$. We call it a proper subgraph if $V' \subset V$ or $E' \subset E$.
Let a subset $V' \subset V$ of vertices be given, then we call the graph

$$G_{V'} = (V', \bigcup_{v \neq w \in V'} [E(v) \cap E(w)])$$

the graph, which is induced by V'.

Definition 2.6. Path Let $G_P = (V_P, E_P)$ be a graph. G_P is called a path if

$$V_P = \{x_1, ..., x_n\}$$
$$E_P = \{\{x_1, x_2\}, ..., \{x_{n-1}, x_n\}\}$$

wherein the x_i are distinct. x_1 and x_n are called the end vertices of the path G_P. The other vertices are called inner vertices. We call such a path an x_1-x_n-path.
Since a path is defined completely by the sequence of vertices, we introduce an alternative notation

$$P = (x_1, ..., x_n)$$

Definition 2.7. Subpath Let $G_P = (V_P, E_P)$ be a path. A subgraph $G_{P'} = (V_{P'}, E_{P'})$ of G_P is called a subpath of G_P if it is a path. For P being described by

$$P = (x_1, ..., x_n)$$

the subpath P' can be written as

$$P' = (x_{m_1}, ..., x_{m_n}) \quad \text{for some } 1 \leq m_1 \leq m_2 \leq n$$

Furthermore, we represent it equivalently by $P' = P|_{x_{m_1}, x_{m_2}}$.

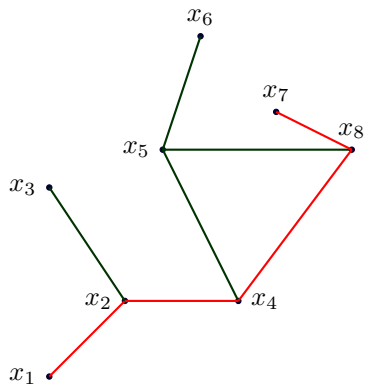


Figure 2.2: Example for a path (red) as subgraph to the graph of the previous example.

In fig. 2.2, a x_1-x_7-path is highlighted in red. It can be written as $G_P = (V_P, E_P)$ with

$$V_P = \{x_1, x_2, x_4, x_8, x_7\}$$
$$E_P = \{\{x_1, x_2\}, \{x_2, x_4\}, \{x_4, x_8\}, \{x_8, x_7\}\}$$

or alternatively as

$$P = (x_1, x_2, x_4, x_8, x_7)$$

Definition 2.8. Cycle A graph $G_C = (V_C, E_C)$ is called a cycle if

$$V_C = \{x_1, ..., x_n\}$$
$$E_C = \{\{x_1, x_2\}, ..., \{x_{n-1}, x_n\}, \{x_n, x_1\}\}$$

wherein the x_i are distinct and $n \geq 3$. The circumference or size of the cycle is defined as $|G_C| := |V_C|$. Similarly to the path definition, a cycle can be represented equivalently by a sequence of vertices

$$C = (x_1, ..., x_n)$$

The example, which was discussed previously, contains a cycle of circumference 3 (a 3-cycle) marked in red in fig. 2.3.

Definition 2.9. Connected Graph A graph $G = (V, E)$ is called connected if for all vertices $s, t \in V$ there is a s-t-path, which is a subgraph to G.

Definition 2.10. Cut Vertex In a connected graph $G = (V, E)$ a vertex v is called **cut vertex** if removing the vertex and all incident edges disconnects the graph.

Definition 2.11. Cluster/Component In a graph $G = (V, E)$ a maximal connected subgraph $G' = (V', E')$ is called a cluster or component of G.

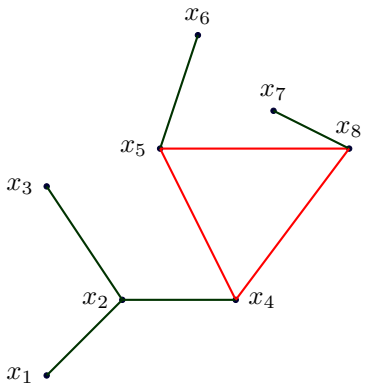


Figure 2.3: Example for a cycle (red) as subgraph to the graph of the previous example.

Definition 2.12. Block For a graph G, a maximal connected subgraph of G, which contains no cut vertices, is called a block.

In fig. 2.3, the graph induced by $V' = \{x_4, x_5, x_8\}$ is a block: None of the three vertices is a cut vertex, as removing one does not disconnect the subgraph. Thus, the graph induced by V' fulfills the condition that it does not contain cut vertices. However, extending V' by x_7 would make x_8 a cut vertex for the induced subgraph. The same holds for x_2 and x_6 so that V' is maximal under the condition that it does not contain any cut vertices.

2.2 Trees

Definition 2.13. Forest A graph not containing any cycles (an acyclic graph) is called a forest.

Definition 2.14. Tree A connected forest is called a tree. All vertices of degree 1 are called leaves. The set of all trees is represented by

$$\mathbb{T} = \{G\,(V,\, E) |\, G \text{ is a tree}\}$$

As stated and proven in [11] (p.14, theorem 1.5.1),

Theorem 2.15. For a graph $G = (V,\, E)$, the following are equivalent

- G is a tree
- there is a unique path connecting any two vertices $u, v \in V$

Definition 2.16. Rooted Tree If one vertex is designated as root vertex r, we call the tree a rooted tree.

Definition 2.17. Parent/Child nodes For a given vertex $v \neq r$ we call the (unique) neighbor $w \in V$ of v, which lies on the (unique) path from v to the root vertex r, the parent of v. We write $w = p(v)$. Vice versa, we call v a child of w.

Definition 2.18. Descendants We define the descendants of v as all nodes $w \in V$ so that v lies on the (unique) path from v to the root vertex r.

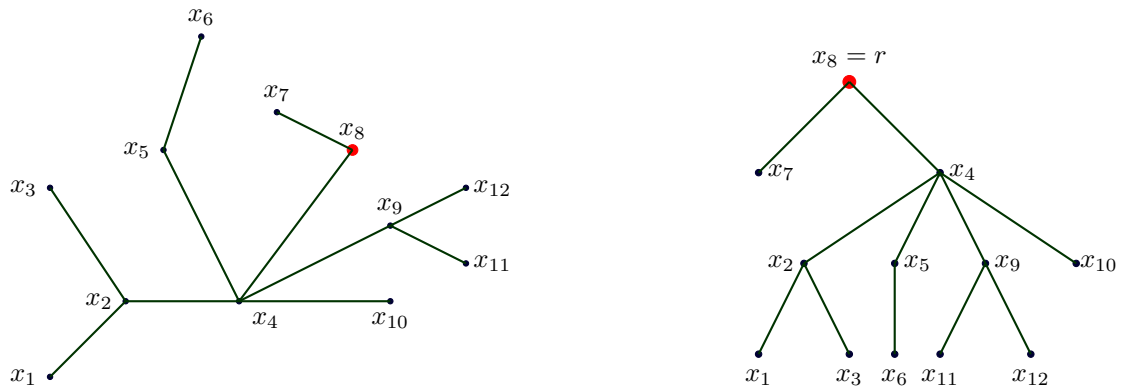


Figure 2.4: Example for a rooted tree. The node, which is used as root is highlighted in red. The right-hand side depicts the same tree as on the left but rearranged as commonly done for trees with the root on top and leaves in the bottom.

In fig. 2.4, a tree is depicted and rearranged to a rooted tree. Deploying the definitions of parents and children, x_5, x_2, x_9 and x_{10} are children of x_4.

Definition 2.19. Binary Tree A rooted tree for which all nodes have at most two children is called a binary tree.

Definition 2.20. Subtree Given a rooted tree $G = (V, E)$ and a designated node $v \in V$ we call the tree T_v, which is induced by v and its descendants, the subtree induced by v.

In fig. 2.4, the subtree induced by x_9 is given by

$$T_{x_9} = (\{x_9, x_{11}, x_{12}\}, \{\{x_9, x_{11}\}, \{x_9, x_{12}\}\})$$

2.3 Cactus Trees

Definition 2.21. Cactus Tree A graph $G = (V, E)$ is called a cactus tree (or cactus graph, or cactus) if it is connected and each block is either an edge or a cycle [12]. If one node is pointed out as the designated root node, we'll call it a rooted cactus tree. We denote the set of all cactus trees by

$$\mathbb{C} = \{G = (V, E) | G \text{ is a cactus tree}\}$$

Cactus trees are defined in the literature in various ways. The stated one is one of the most prominent definitions. However, there are some equivalent definitions: A graph $G = (V, E)$ is called a cactus

1. if each edge is at most part of one cycle.

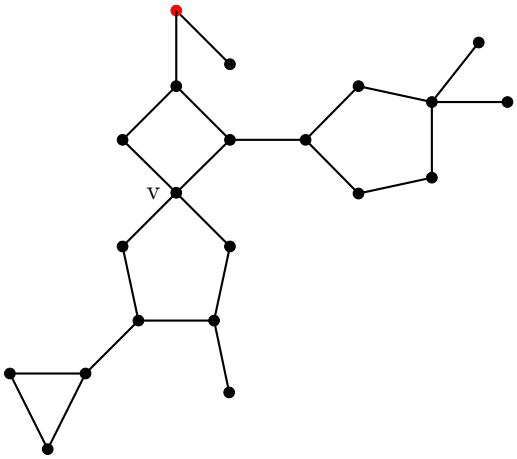


Figure 2.5: Rooted binary cactus tree, the root is marked in red

2. if two cycles have at most one common vertex [7].

It is easily verifiable, that these definitions are equivalent to the one given in def. 2.21. An example of a (rooted) cactus tree is shown in fig. 2.5.
In the previous section, we stated that the path from an arbitrary vertex v to the root r is unique in a tree. Even if we can specify a dedicated root node, not even the shortest path of a node to the root node must be uniquely determined in a cactus graph (consider e.g. the vertex labeled 'v' in fig. 2.5).

Definition 2.22. Cycle Root Let a rooted cactus tree $G = (V, E)$ be given. We call for each cycle $C = (x_1, ..., x_n)$ the node $r_c \in \{x_1, ..., x_n\}$, which has the shortest path to the (cactus-wide) root, the cycle root.

We define the set of shortest paths from a node to the cactus root as follows:

Definition 2.23. Let a cactus $G = (V, E)$ with a dedicated root node $r \in V$ be given and a node $v \in V$. Then the set of all shortest-lengths path from the cactus root r to v are denoted by

$$\mathbb{P}(r, v) = \{P \mid P \text{ is a shortest-length path from } r \text{ to } v\}$$

Then, the following holds

Lemma 2.24. For a rooted cactus tree $G = (V, E)$ with root $r \in V$ and a cycle $C \in G$, $C = (x_1, ..., x_n)$ with a cycle root r_C

$$\forall P \in \mathbb{P}(r, r_C)\ \forall x_i \in C \setminus \{r_C\}\ :\ x_i \notin P \tag{2.1}$$

Proof. If there would be such an $x_i \in C \setminus \{r_C\}$ and such a $P \in \mathbb{P}(r, r_C)$, then $|P|_{r,x_i}| < |P|$. This would contradict the assumption, that r_C is the cycle root. □

Lemma 2.25. For every cycle in a rooted cactus tree $G = (V, E)$, the cycle root exists and is unique.

Proof. By definition of a cactus tree, it is connected so that the cycle root must exist. Let the (global) root of the cactus tree be called r. Now, assume that for a given cycle $C = (x_1, ..., x_n)$ there are two cycle roots $r_C \neq r_{C\prime} \in C$. There exist two paths H_1 and H_2 from r_C to $r_{C\prime}$ using only cycle edges and vertices

$$C = (r_C = \underbrace{w_1, .., w_m}_{:=H_1} = r_{C\prime} = \underbrace{v_k, ..., v_1}_{:=H_2} = r_C) \quad m, k \in \mathbb{N} : m + k - 2 = n$$

Due to the reason that the path to the root is not unique, we have two sets of shortest length paths $\mathbb{P}(r, r_C)$ and $\mathbb{P}(r, r_{C\prime})$ as defined above. It holds by the previous lemma that

$$\forall\, P \in \mathbb{P}(r, r_C) \ : \quad r_{C\prime} \notin P$$

and

$$\forall\, P\prime \in \mathbb{P}(r, r_{C\prime}) \ : \quad r_C \notin P\prime$$

However, there exist vertices, which are common for at least two paths (since the paths end at the same node).

$$\forall P \in \mathbb{P}(r, r_C) \text{ and } P\prime \in \mathbb{P}(r, r_{C\prime}) \ : \ P \cap P\prime \neq \emptyset$$

Since the paths can be interpreted as ordered sequences of nodes from r_C or $r_{C\prime}$ to r, there exists the first intersection for any two of these paths. We call this first common node of paths P and P' $v_{intersect,\ P,\ P'}$.
Knowing $v_{intersect,\ P,\ P'}$ for two fixed paths

$$P \in \mathbb{P}(r, r_C) \text{ and } P' \in \mathbb{P}(r, r_{C\prime})$$

the two graphs

$$C^1_{r_C, r_{C\prime}} := H_1 \cup P|_{r_C, v_{intersect}} \cup P'|_{r_{C\prime}, v_{intersect}}$$

$$C^2_{r_C, r_{C\prime}} := H_2 \cup P|_{r_C, v_{intersect}} \cup P'|_{r_{C\prime}, v_{intersect}}$$

are cycles in G, which share all edges in H_1 (H_2 respectively) with C. Consequentially, C is not a simple cycle which contradicts the assumption that G is a cactus graph. □

Using the definition of blocks, as given earlier, we can transform the input cactus tree to a tree by mapping each block to a vertex and connecting them by edges.

Definition 2.26. Block Transformation The block transformation

$$\begin{array}{rccc} b : & \mathbb{C} & \to & \mathbb{T} \\ & G = (V, E) & \to & G' = (V', E') \end{array}$$

of a cactus tree with m blocks $b_1, ..., b_m$ is defined by the following transformation of vertices and edges:

- each block $b_i \in G$ is mapped to a vertex $w_i \in V'$. w_i is then called either **edge block** or **cycle block** depending on the nature of b_i
- each vertex $v_i \in V$, which is part of multiple blocks b_j $j \in J$, maps to $v'_i \in V'$ and edges $\{v'_i, w_j\} \in E'$. These nodes v'_i are called **connection blocks**. A connection block, which has a non-cycle parent block, is called a **vertex block**.

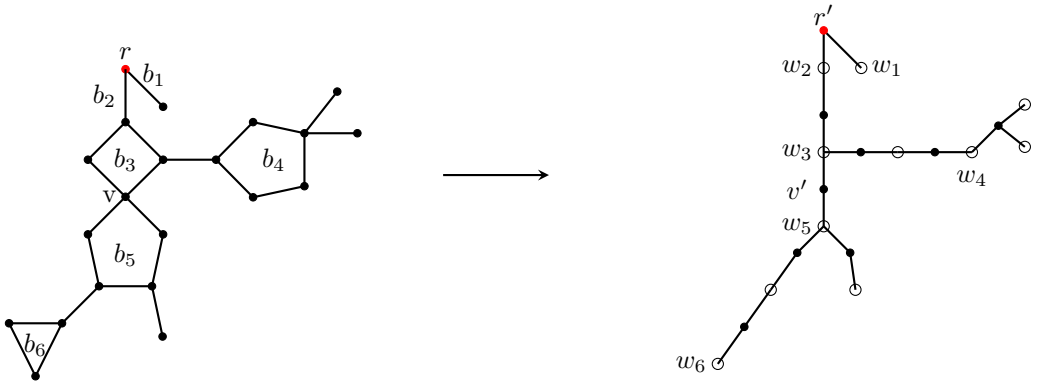


Figure 2.6: A rooted binary cactus tree, in which the root is marked in red (left), and the corresponding tree, which is created by block-transforming the cactus tree on the left-hand side (right).

This transformation is well-defined for cactus trees (even for generally connected graphs [11], Prop. 3.1.2).
This suggests the definition of a binary cactus tree as follows

Definition 2.27. Binary Cactus Tree A cactus tree (V, E), which can be block-transformed and rooted to be a binary tree, is called a **binary cactus tree**.

2.4 Problem Definition

Definition 2.28. Partitioning under Contraints, PuC Let $G = (V, E)$ be a graph and $w : V \to \mathbb{N}^k$ a mapping associating k weights to each vertex. The vertex weight of v can thereby be found via $w(v)$. Additionally, weight constraints $R \in \mathbb{N}^k$ are given.
The objective is to find a minimal set of edges $E' \subset E$ so that the clusters $V_1, ..., V_n$ of $G \setminus E'$ fulfill the constraints

$$\forall i \in \{1, .., n\} \ \ \forall j \in \{1, .., k\} \quad \sum_{v \in V_i} w_j(v) \leq R_j$$

The most relevant graph restricted problems of PuC for this work are listed in table2.1. The last restriction yields the problem targeted in the following chapters: partitioning of binary cactus graphs under two constraints.

Shortcut	Restrictions on Graphstructure	Restrictions on Constraints
TPuC	Trees	-
CTPuC	Cactus Trees	-
BCTPuC	Binary Cactus Trees	-
TP (target problem)	Binary Cactus Trees	k=2

Table 2.1: Restricting PuC to the target problem: Partitioning of binary cactus trees under two constraints.

For the following sections, we generally assume that $k = 2$. Even though the algorithm itself can be easily extended for other $k > 1$, most of the results on performance do not apply in general.

Chapter 3

Partitioning Algorithm

In the following, the partitioning algorithm of Hamacher et al. [10] for trees with two weight functions is elucidated shortly before it is extended to cactus graphs and its correctness is proven.

3.1 Partitioning of Trees with Two Weight Functions

3.1.1 The Algorithm

The idea of the algorithm for TPuC described in [10] is to solve the problem recursively by dynamic programming. Starting at the leaves all partitioning possibilities of a subtree rooted at some vertex v are evaluated by using the solutions, which were previously determined for each child. This is formalized in the following algorithm

Algorithm 3.1. Dynamic Program for Trees [10]

Input:

Let a tree (V, E) with root v_0, a set of weights $(w_1(v), w_2(v))$ for all vertices $v \in V$ and restrictions $(R_1, R_2) \in \mathbb{N}$ be given.

Initialization:

$W(v) := \{w(v)\} = \{(w_1(v), w_2(v))\}$ for all leaves v.
$k := 0$.

Program:

```
while (∃ p ∈ V : p not processed) {
        Select one such p, which has only processed children p_1, ..., p_Δ
        Process p according to 3.4 returns (W(p),j)
        k = k + j
 }
```

Before we continue elucidating how to process a vertex p, a short remark on the used variables is in order.
Firstly, k indicates the number of the currently cut edges in E'. Thus, each time we decide to

cut an edge, k is increased. The algorithm aims at keeping k as small as possible, which means minimizing the number of cut edges.
Secondly, each $W(v)$ contains a set of weight vectors. Every element of $W(v)$ corresponds to the weight of the cluster containing v in a valid solution of TPuC on T(v).
To achieve an optimal solution, there is no need to store all options for valid cluster weights containing v but it is sufficient to store the 'lightest' of those. The 'lightest' weight vector is not a trivial notion, since we have two weight functions

Definition 3.2. Dominating Vectors [10] Let $w, w\prime \in \mathbb{N}^2$, then w **dominates** $w\prime$, when

$$w_1 \geq w\prime_1 \quad \wedge \quad w_2 \geq w\prime_2$$

is fulfilled. Notation: $w \succeq w\prime$. In case $w \neq w\prime$, we say that w dominates $w\prime$ strictly and write $w \succ w\prime$.

Each time when a set $W(v)$ for some vertex v is determined, dominating weights are dropped directly by using the following set operations:

Definition 3.3. Vector Set Operations [10] Let $L_1, L_2 \subseteq \mathbb{N}^q$. Addition $\oplus$ and multiplication $\otimes$ are defined as

$$\begin{aligned} L_1 \oplus L_2 = &\quad \{l \in L_1 \cup L_2 \mid \quad \nexists\, l' \in L_1 \cup L_2 : l \succeq l'\} \\ L_1 \otimes L_2 = &\quad \{l \in \bar{L} \mid \quad \nexists\, l' \in \bar{L} : l \succeq l'\} \\ \text{with } \bar{L} = &\quad \{l = l_1 + l_2 \mid \quad l_i \in L_i \text{ for } i = 1, 2 \ \wedge \ R \succeq l\} \end{aligned} \tag{3.1}$$

The algorithm by Hamacher et al. [10] processes nodes as follows:

Algorithm 3.4. Node Processing

Input:

Given an vertex p with Δ children $p_1, ..., p_\Delta$, which have weight sets $W(p_1), ..., W(p_\Delta)$ assigned.

Initialization:

$j := 0$

Program:

1 $W(p, 1, 0) = W(p_1) \otimes \{w(p)\}$;
2 $W(p, 1, 1) = \{w(p)\}$;
3 For $(i = 2;\ i \leq \Delta;\ i++)$ {
4 $\quad W(p, i, 0) = W(p, i-1, 0) \otimes W(p_i)$;
5 $\quad W(p, i, i) = \{w(p)\}$;
6 }
7 While $(W(p, \Delta, j) = \emptyset)$ {
8 $\quad j++$;
9 $\quad$ For $(i = j + 1\,..\,\Delta)$ {
10 $\quad\quad W(p, i, j) = W(p, i-1, j-1) \oplus (W(p, i-1, j) \otimes W(p_i))$;
11 $\quad$ }
12 }
13 Return $(W(p, \Delta, j), j)$;

3.1.2 The Interpretation

The algorithm 3.1shows clearly which node may be processed next. However, the node processing itself, as described in 3.4, is not that easily comprehensible. The following paragraphs aim at providing a better understanding of the algorithm employing an example.

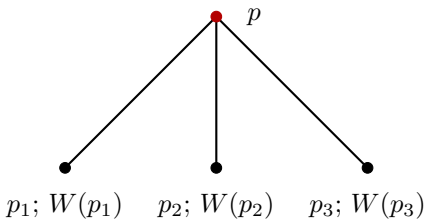


Figure 3.1: Initial setup for algorithm 3.4: The vertex p needs to be processed and its three children have been processed before (and have accordingly the sets $W(p_i)$ set).

The example consists of a simple structure with four nodes. The currently processed node p has three children p_1, p_2, p_3 (see fig. 3.1), which have been processed previously according to 3.4.
We start by initializing the variable j, which counts the number of cuts necessary to merge and extend the partitions of $T(p_1)$, $T(p_2)$, $T(p_3)$ by the connecting parent node p. It will be added to the required cuts in algorithm 3.1, thus to the size of the partition.
$W(p, i, j)$ has three parameters:

- The first parameter p highlights the currently processed node.
- The second parameter i indicates the subtree we consider at that point. For all $W(p,\ 1,\ j)$ we consider the tree induced by $p \cup T(p_1)$, for $W(p,\ 2,\ j)$ the one induced by $p \cup T(p_1) \cup T(p_2)$, etc. So in general, $W(p,\ k,\ j)$ is a weight set on p considering only the tree induced by $p \cup \bigcup_{l=1}^{i} T(p_l)$.
- The third indicates the number of cuts j, which are currently considered. The node processing algorithm terminates at the smallest j, which allows a solution obeying the restrictions.

Knowing this, we can interpret the algorithm. The beginning (ln. 1-6) could be considered an extended initialization, as we assign only the things we know trivially as shown in fig. 3.2.

- For 0 cuts (ln 1, ln 4) we try to fit all child nodes p_i and the parent node p in one cluster by multiplying the weight sets.
- When cutting off the first i edges, we'll be left at $W(p,\ i,\ i)$ with p in the cluster and thus $W(p,\ i,\ i) = \{w(p)\}$.

After ln. 6, we would have $W(p, \Delta, j = 0) \neq \emptyset$, if there were weights

$$w_i \in W(p_i)\ \forall i \in \{1, .., \Delta\}$$

so that

$$R \succeq w(p) + \sum_{i=1}^{\Delta} w_i$$

Thus, the sum of weights would suffice the restrictions and we would need no cut when merging the solutions for $p_1, ..., p_\Delta$ at the node p (see fig. 3.2 in the right)

Now, in the second part of the algorithm (ln.7-12), we increase the number of cuts, as long as there is no valid solution. A valid solution exists as soon as $W(p,\, \Delta,\, j) \neq \emptyset$. We add one cut at the smallest possible i for which we can cut off at least j children and still add the node itself, meaning at $i = j + 1$ (reflecting the iteration boundaries in ln. 9).
There are two contributions to $W(p,\, i,\, j)$;

- $W(p,\, i-1,\, j-1)$ the (j-1)-cut solution of previous considerations for $i-1$ and cutting off the node i itself
- $W(p,\, i-1,\, j) \otimes W(p_i)$ is the j-cut solution of $i-1$ and merging child p_i to the cluster of p.

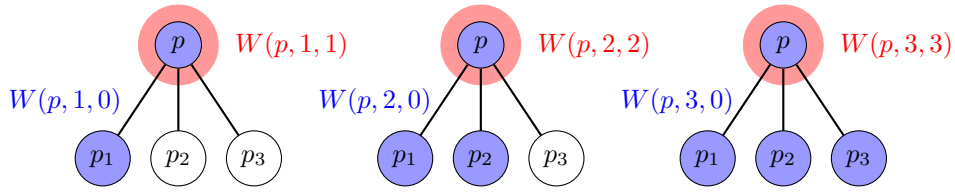


Figure 3.2: Example for the first part of the algorithm. The considered vertex has three children, which were processed previously. In lines 3-8 of the algorithm 3.4, the sets of weights are initialized. The clusters, which are represented by these weight sets are colored accordingly.

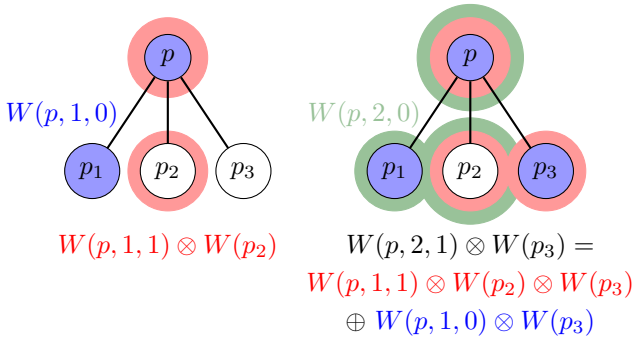


Figure 3.3: First iteration of the second loop of algorithm 3.4. We assume, that $W(p, \Delta = 3, 0) = \emptyset$ and start in ln. 9 of the algorithm with $j = 1$

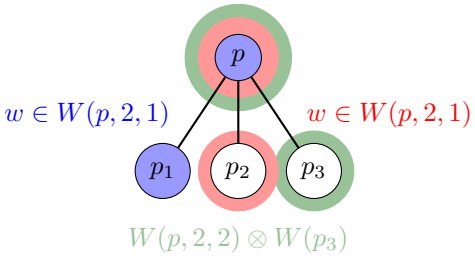


Figure 3.4: Second iteration of the second loop of algorithm 3.4. We assume, that $W(p, \Delta = 3, 1) = \emptyset$ and start in ln. 9 of the algorithm with $j = 2$.

This is shown for $j = 1$ in the example in fig. 3.3. Starting at $i = 2$, we either have a 0-cut solution for $p \cup T(p_1)$ and cut $\{p, p_2\}$ (blue) or cut $\{p, p_1\}$ to have p and p_2 in one small cluster (not yet being said, that it suffices the restrictions). The results of $i = 2$ are then needed for $i = 3$, since a solution with $j = 1$ cut for $p \cup T(p_1) \cup T(p_2)$ with no cut at $\{p, p_3\}$ can yield a valid solution

as well - this is marked, similarly to the $i = 2$ picture, in red and blue. Additionally, the green indicates using the 0-cut solution of $p \cup T(p_1) \cup T(p_2)$ and cutting $\{p, p_3\}$. Two cuts require only processing of $i = j + 1 = 3$, as shown in fig. 3.4.
Lines 7 -12 can be understood to compensate for the bias of the algorithm due to the arbitrary ordering of the children.

3.1.3 Proven Results and Properties

As proven in [10], the worst-case running time for processing a node is given in this theorem:

Theorem 3.5. Let $T = (V, E, w, R)$ be an instance of TPuC. Assume a vertex v_0 has only Δ processed children. Let p be the maximum size of a list $W(v_0, i, j)$ occurring at v_0. The algorithm 3.4 at v_0 can be implemented in $O(\Delta^2 p^3)$ worst-case running time.

3.2 Extension for Cactus Trees

3.2.1 Algorithm

To extend the algorithm 3.1for cactus graphs, we consider the block-transformed tree (see initialization of algorithm 3.6). Thereby it is easy to distinguish between a simple parent, as usually encountered in a tree, and a parent node, which is part of a cycle. The algorithm introduced in the following is inspired by the algorithm presented by Buchin and Selbach [7] for a related problem.

Algorithm 3.6. Dynamic Program for Cactus Trees

Input:

Let a cactus tree (V, E) with root v_0, a set of weights $(w_1(v), w_2(v))$ for all vertices $v \in V$ and restrictions $(R_1, R_2) \in \mathbb{N}$ be given.

Initialization:

$W(v) = \{w(v)\} = \{(w_1(v), w_2(v))\}$ for all leaves $v \in V$.
$(V', E') = b((V, E))$
$A = \{v' \in V' \mid v' \text{ is leaf}\}$; $k = 0$.

Program:

```
1  while (A ≠ ∅) {
2      Select v ∈ A
3      Switch (v) {
4          case (vertex block):
5              Call algorithm 3.10 for b^{-1}(v), returns (W(b^{-1}(v)), k_{b^{-1}(v)})
6              Set k = k + k_{b^{-1}(v)}
7              break
8          case (cycle block):
9              Preprocess the cycle b^{-1}(v) with algorithm 3.9 [a]
10             If (v_0 ∈ b^{-1}(v) ∧ deg(v_0) == 2 )
11                 Process v_0 with algorithm 3.10, returns (W(v_0), k_{v_0})
12                 Set k = k + k_{v_0}
13             break
14         case (edge block):
15             If (v_0 ∈ b^{-1}(v) ∧ deg(v_0) == 1 )
16                 Process v_0 with algorithm 3.10, returns (W(v_0), k_{v_0})
17                 Set k = k + k_{v_0}
18             break
19         default:
20             break
21     }
22     A = A \ {v}
23     If (all children of p(v) are processed)
24         A = A ∪ {p(v)}
25 }
```

[a] corresponds to processing all cycle nodes except the cycle root node completely

We keep track of the active blocks, by which those blocks are meant, which have only processed children but themselves are not processed yet, in set A. An easy observation is then

Lemma 3.7. Each node of the block-transformed graph (V', E') in the algorithm 3.6is added and removed exactly once to A.

Proof. We initialize A as a non-empty set so that the algorithm does not terminate immediately. Each time a node is processed, it is removed from A and eventually, its unique parent node is added. Since the block-transformed graph is a tree, no node can be added twice. By definition of a tree, it is connected, so that each node is added to A at least once. □

Before continuing, a short example shall increase the intuitive understanding of this algorithm. In figures 3.5-3.9 the first steps of the algorithm are shown exemplarily for a small cactus tree. The explanations are given completely in the captions. All alternatives for processing blocks, except the special treatment for the root node are covered by this example.

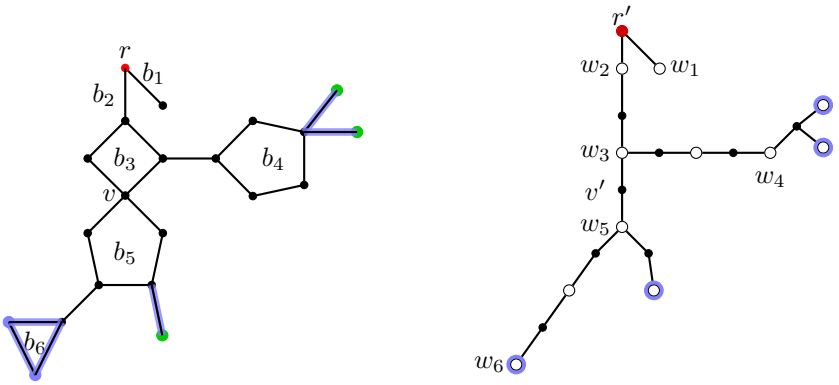


Figure 3.5: Initial State of a call to algorithm 3.6. In the left graph, the original input graph is shown with the root highlighted in red. The already processed vertices, so all, which are leaves, are shown as larger green nodes. In the right graph, the block transformed graph is shown. All active blocks (blocks in set A) are highlighted in blue. There are no processed blocks yet, which shall be shown highlighted in green in the following pictures. However, transfering the active blocks back to the original graph, results in the three edges and the 3-cycle being active in the next processing step.

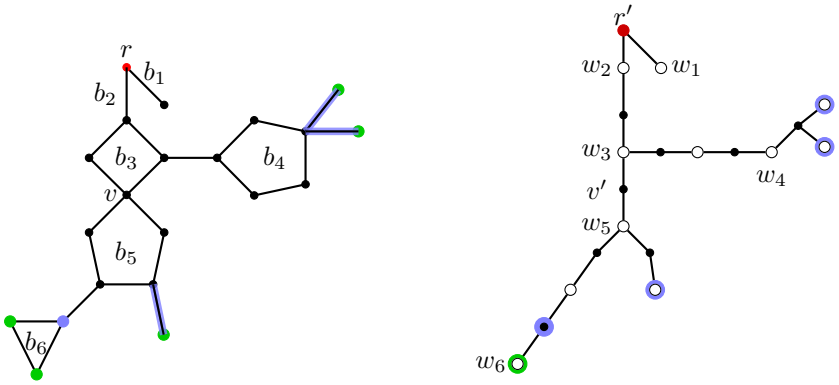


Figure 3.6: After processing the active cycle block w_6, so after preprocessing the cycle b_6, all but nodes of b_6 except the cycle root are processed. The cycle root is not processed yet and now active because the parent of w_6, the vertex block, is active.

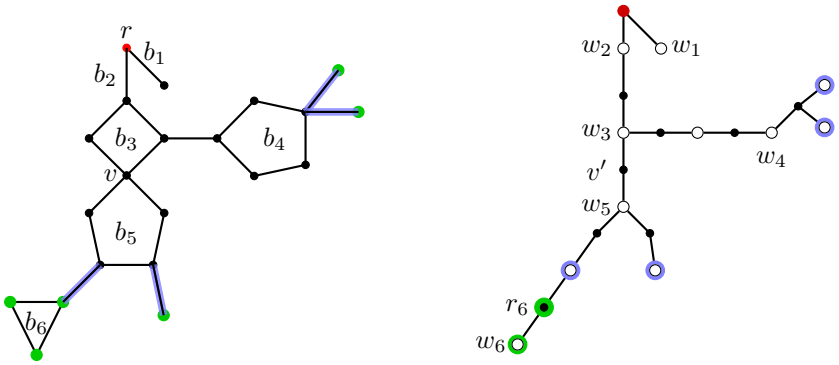


Figure 3.7: Starting from fig. 3.6, we select the parent of w_6, here marked as r_6, to be processed in the next iteration. It is a vertex block and will be processed according to the case in ln.4 of algorithm 3.6.

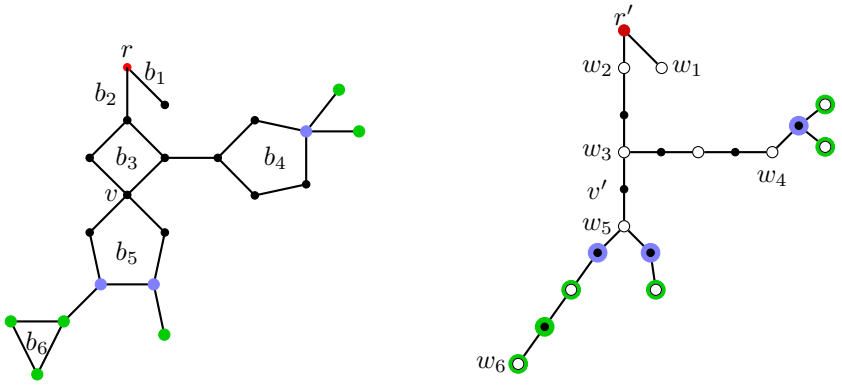


Figure 3.8: After reaching the state shown in fig. 3.7, only edge blocks are active. Those are only processed non-trivially if they are the root block. So in this case, we iterate and remove them from A and add their parents to A. Stepping 4 iterations forward in which we process all these edge vertices, we arrive at the state depicted here.

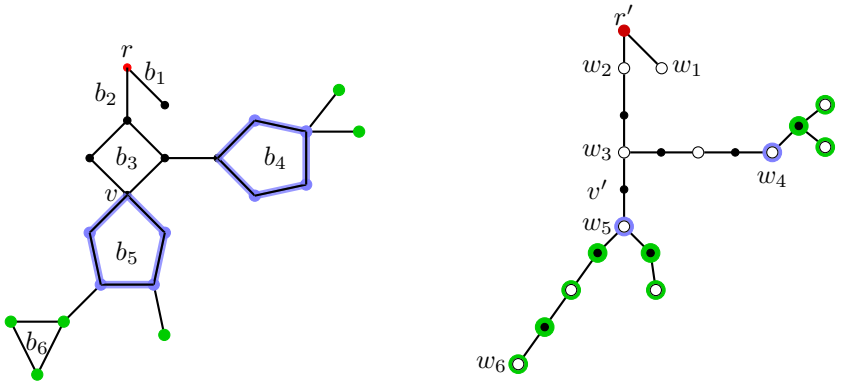


Figure 3.9: After reaching the state shown in fig. 3.8, all active blocks are of the same type. They are connection blocks but no vertex blocks. Thus, they are processed trivially in algorithm 3.6. Three iterations later, the blocks are processed but their according vertices not. The new active blocks are the cycle blocks w_5 and w_4.

Figure 3.10: In case, the root node has only one direct child, it'll be transformed to be part of an edge block. The edge (left) and its corresponding block (right) are marked in blue. The child of the root is transformed to a vertex block with two edge blocks as children. Edge blocks are again white with black surrounding.

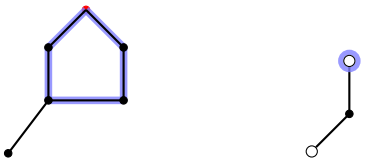


Figure 3.11: In this case, the root node (red) is part of a cycle and has no more children than the two given by the cycle. In the block transformed graph (right), the root node is only part of the cycle block which is marked in blue and needs to be processed as part of the processing of the cycle node.

The extra treatment of the root node is a consequence of the block transformation. In the case, which is depicted in the example, the root node is a vertex node itself. This is always the case when the root node either has more than one simple, no-cycle child or is part of multiple cycles or a combination of those.

In algorithm 3.6ln. 10 and ln. 15 assert that the root node is processed even if it block-transforms not as a dedicated vertex block. In case it is only part of an edge block, as depicted in fig. 3.10, the required special treatment is recognized by investigating whether the root node is part of an edge block and has only one child. Having more children, the root node would occur in the pre-image of a vertex block again. Similar argumentation yields that as part of a cycle the root might have at most two children to need special treatment (see fig. 3.11).

The examples made it clear that by processing each block each vertex from the original graph is processed before the algorithm terminates. This is formalized in the following lemma.

Lemma 3.8. Each node of the graph (V, E) in algorithm 3.6is processed exactly once.

Proof. We have two kinds of nodes $v \in V$, which are part of exactly one block

Vertex Type	Processed
Leaf	by initialization
Inner cycle node (not cycle root) - no neighbours outside its cycle	in cycle preprocessing (ln. 8)

We have two kinds of nodes $v \in V$, which are part of multiple blocks and would translate to connection nodes in the block transformation

Vertex Type	Processed
Vertex with parent vertex - can have multiple vertex children - can be cycle root for some cycles	transforms to a vertex block processed in ln. 5
Inner cycle node - can have multiple vertex children - can be cycle root for some other cycles	in cycle preprocessing (ln. 8)

The root node needs to be treated more explicitely than any other vertex

Vertex Type	Processed
Root connects multiple blocks	processed by common logic (ln. 5)
Root is only part of one cycle $\rightarrow$ does not connect multiple blocks	processed in ln. 10-11
Root has exactly one child $\rightarrow$ does not connect multiple blocks	processed in ln. 14-15

□

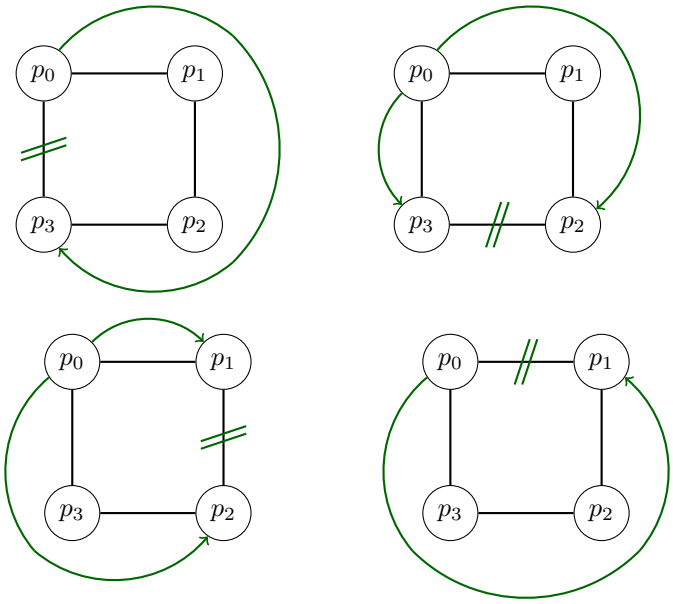


Figure 3.12: Cutting cycles displayed for a 4 node cycle rooted in p_0.

3.2.2 Preprocessing Cycle Blocks

Buchin et al. [7] processed cycles in context k-(l,u)-partitioning by 'cutting open the cycles' at each possible edge and collecting all found solutions. An example of this procedure is shown in fig. 3.12, analogously to the explanations in [7]. The restriction to exactly k clusters, which is not given in the problem investigated here, makes it easy to keep or drop configurations. However, the aim here is to find a partition that uses as few clusters as possible, which results in a demand for a more careful investigation of these intermediate solutions.

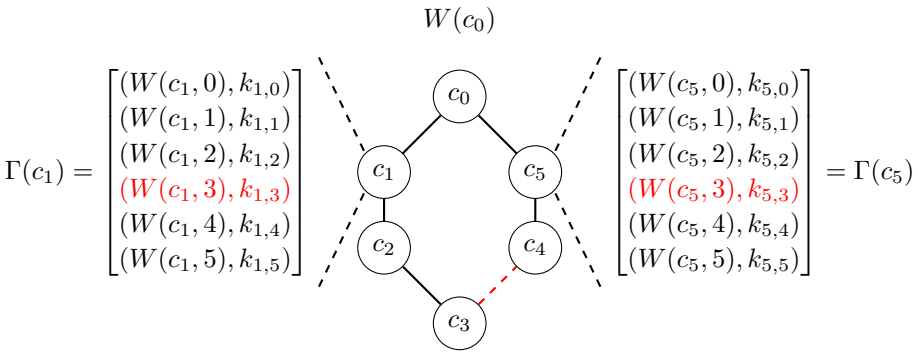


Figure 3.13: Ordered weight storage vectors after the cycle preprocessing. The red entry marks the entry generated by the initial cut between $c_3 - c_4$. When we process the cycle root, we can decide for an initial cut and store the so found weight vectors in $W(c_0)$.

Usually, as explained in subsection 3.1.1, we would store a set of weights $W(v)$ on each vertex v. Now, we want to preprocess the cycle with each initial opening cut (as displayed exemplarily in

fig. 3.12) and store the information on needed cuts and resulting weight vectors in ordered arrays of the two direct children of the cycle root. An example is shown in fig. 3.13.

The cycle preprocessing algorithm is described formally in algorithm 3.9. It is worth highlighting beforehand that the cycle root is not processed yet, explaning the naming. The cycle root is processed e.g. in ln. 10 of algorithm 3.6 when it is the global root as well. Another option to process the cycle root would occur when the cycle root is part of the vertex block and thus be processed in ln. 5. Similarly, if the cycle root is an inner node of some other cycle, it is processed in in the preprocessing of that.

Algorithm 3.9. Dynamic Program for Cycle Preprocessing

Input:

Let a rooted cycle be given $C = (V_C, E_C)$ with a designated root node c_0. The size of the cycle is denoted by Ω. The cycle nodes are numbered as $(c_0, c_1, ..., c_{\Omega-1})$. C is a subgraph of a larger cactus graph G. All neighbors $v \in N(c_i)$ for $i = 1, .., \Omega - 1$, which are not in C, are processed.

Initialization:

$\Gamma(c_1, 0) = ((0, 0), 0)$
$\Gamma(c_{\Omega-1}, \Omega) = ((0, 0), 0)$

Program:

1 For($\omega = 0$; $\omega < \Omega$; $\omega + +$){
2 For($i = \omega + 1$; $i < \Omega$; $i + +$){
3 Process c_i
4 • having children $N(c_i) \setminus V_C \cup \{c_{i-1}\}$ using $W(c_{i-1}) = \Gamma(c_{i-1}, \omega)$ for $i - 1 \neq \omega$
5 • having children $N(c_i) \setminus V_C$ for $i - 1 = \omega$
6 returns $(W(c_i), k_{c_i})$
7 Store result in $\Gamma(c_i, \omega) = (W(c_i), k_{c_i})$
8 }
9 For($i = \omega$; $i > 0$; $i - -$){
10 Process c_i
11 • having children $N(c_i) \setminus V_C \cup \{c_{i+1}\}$ using $W(c_{i+1}) = \Gamma(c_{i+1}, \omega)$ for $i \neq \omega$
12 • having children $N(c_i) \setminus V_C$ for $i = \omega$
13 returns $(W(c_i), k_{c_i})$
14 Store result in $\Gamma(c_i, \omega) = (W(c_i), k_{c_i})$
15 }
16 }
17 Return $(\Gamma(c_1, \omega))_{\{\omega=0,...,\Omega\}}$, $(\Gamma(c_{\Omega-1}, \omega))_{\{\omega=0,...,\Omega\}}$

Results computed for various initial cuts, which are indicated by the variable ω, are stored at each cycle node c_i in a vector $\Gamma(c_i)$. During initialization, the two trivial values are set: When we cut the 0th-edge of the cycle, the first neighbor will be cut off. For sake of symmetry and to avoid more case distinctions, we'll set the weight set encompassing only zero-weight and not needing any more cuts. The same holds vice versa for the Ω-th-edge and the $(\Omega - 1)$-th node.

The outer loop of the program iterates the initial cut. For fixed ω, the edge $c_\omega - c_(\omega + 1)$ is cut.

Then we have two branches originating at c_0 (except in the cases $\omega = 0$ and $\omega = \Omega$), one containing the path $(c_1, ..., c_\omega)$ and the other $(c_{\omega+1}, ..., c_{\Omega-1})$. The algorithm processes these paths as usually done for trees:

- at the beginning of each iteration, we consider all $c_i \in V_C$ as unprocessed
- since all $N(c_i) \setminus V_C$ are processed, at the beginning of each iteration, only the nodes c_ω for the first branch and $c_{omega+1}$ for the second branch have only processed children. Thus, we begin at those (see for-loops in ln. 2 and 9)
- they are processed as described in the next section (see ln. 5, 12)
- the result is stored in Γ (ln.7, 14)
- in the following iteration, the next node on each path is unprocessed having only processed children. By ln.4, 11 we use the correct weight for the in-cycle children of each node on the path.

This can be seen more clearly by investigating an example. As shown in fig. 3.14, the cycle is cut open iteratively on the edge between c_ω and $c_{\omega+1}$. The processing order is always from the nodes next to the cut to the cycle node.

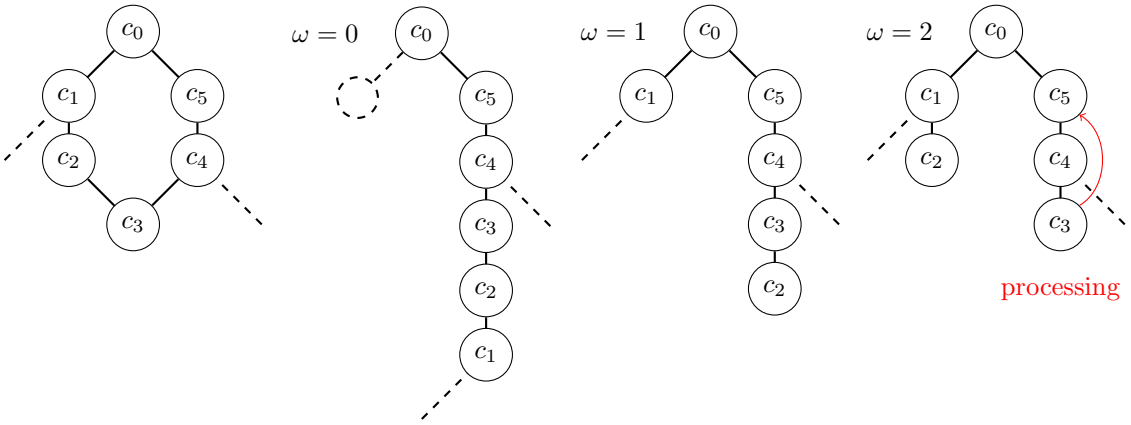


Figure 3.14: Left: Input cycle for the cycle preprocessing algorithm. For generality, the dashed lines starting from c_4 and c_1 are to indicate that these nodes have an additional child outside the cycle. The other figures show configurations that are considered when ω is selected as indicated in the outer loop of the algorithm. The dashed node in $\omega = 0$ indicates the "imaginary" child that does not require an additional cut and has no weight. This balances the $[\Gamma(c_1)]_0$ entry. In the rightmost image, the processing order is indicated by the red arrow.

3.2.3 Processing Vertex Nodes

In the following paragraphs, we extend the common node processing as used in algorithm 3.4 for trees, to cope with the additional complexity encountered when treating a node, which serves as a cycle root to one or more cycles.

Algorithm 3.10. Node Processing in Cactus Graphs

Input:

Given an vertex p with Δ children $p_1, ..., p_\Delta$. Thereof, the first α children are simple vertex children, which have weight sets $W(p_1), ..., W(p_\alpha)$ assigned. The latter $\Delta - \alpha = 2 \cdot \beta$ children belong to β Ω-cycles for which p is the cycle root, each having a vector $(\Gamma(p_i, \omega))_{\omega=0,..,\Omega}$ assigned. With $\Gamma(p_i, \omega) = (W(p_i, \omega), j_{p_i,\omega})$ [a].

Initialization:

$j_{min} := \infty$
$W := \emptyset$

Program:

1 Forall($i \in \{0, .., \Omega\}^\beta$){
2 Process p according to algorithm 3.4 with child node weights
3 $(W(p_1), ..., W(p_\alpha), \Gamma(p_{\alpha+1}, i_1).W, \Gamma(p_{\alpha+2}, i_1).W, ..., \Gamma(p_{\alpha+2\beta}, i_\beta).W)$
4 returns $(W_i(p), j_i)$
5 If($j_i + \sum_{l=1}^{\beta} c_{close}(p_{\alpha+2\cdot l-1}, p_{\alpha+2\cdot l}) < j_{min}$) {
6 $W = W_i(p)$
7 $j_{min} = j_i + \sum_{l=1}^{\beta} c_{close}(p_{\alpha+2\cdot l-1}, p_{\alpha+2\cdot l})$
8 }
9 Else if ($j_i + \sum_{l=1}^{\beta} c_{close}(p_{\alpha+2\cdot l-1}, p_{\alpha+2\cdot l}) = j_{min}$) {
10 $W = W \cup W_i(p)$
11 }
12 }
13 If ($\beta = 0$) {
14 Process p according to algorithm 3.4, which directly returns (W, j_{min})
15 }
16 Return (W, j_{min});

[a] Here, we address first and second component of the tuple via .W or .j

In the algorithm, we use the function $c_{close}(p_m, p_{m+1})$ to determine, the actual number of cuts needed to partition the child cycle consisting of the child nodes p_m, p_{m+1} belonging to one initial cut $i \in \{0, ..., \Omega\}^\beta$:

$$c_{close}(p_m, p_{m+1}) = \begin{cases} 0 & \text{neither } p_m \text{ nor } p_{m+1} \text{ was cut} \\ & \text{and } \Gamma(p_m, i).j + \Gamma(p_{m+1}, i).j = 0 \\ \Gamma(p_m, i).j + \Gamma(p_{m+1}, i).j + 1 & \text{else} \end{cases} \quad (3.2)$$

This accounts for the case, where an initial cut is made, but no branch needs any more cuts and both branches can be added to the cluster of the top node. Then the initial cut wasn't necessary

and may not be added.
The presented algorithm shows a rather intuitive way of evaluating the different possibilities arising from different initial cuts of the cycles. Sadly, one can state from the first glance, that this implementation must show bad performance for increasing Ω.
This algorithm works the same way, as the algorithm which processes nodes in trees. However, it accounts for the additional variability coming in from child cycles but covers it in an outer forall-loop. Therefore, an example is skipped.

3.2.4 Correctness

As shown in [10][13] the algorithm for trees fulfills the following correctness theorem

Theorem 3.11. [10] For the algorithm 3.1 using algorithm 3.4, the following holds

1. If $W(p, i, j)$ was set in the algorithm 3.4, either by setting the trivial values in ln. 1f, ln. 4f, or while incrementing j in ln. 10, it contains all non-dominating weight vectors of the p-containing cluster of a partition of $T(p)$, which has j cuts at p. Additionally, for all induced partitions on $T(p_l)$ for all children p_l of p with $1 \leq l \leq i$, the partitions are minimal.

2. If $W(p) \neq \emptyset$, it contains all non-dominating weight vectors of the p-containing cluster of a minimal partition of $T(p)$.

3. After each iteration of a loop in algorithm 3.1, ln. 1, the variable k contains the minimal number of edges to cut to obtain a feasible partition.

A proof of this theorem can be found in [10] [13].
We can show a similar result for algorithm 3.6 for cactus trees, making use of the sub-algorithms 3.9, 3.10 and 3.4. We start by showing that the algorithm for cactus trees performs exactly the same way under the assumption that the input cactus tree is a tree.

Lemma 3.12. Algorithm 3.10 reduces to one call to algorithm 3.4 when $\Delta = \alpha$, thus, when a node has no cycle nodes.

Proof. $\Delta = \alpha$ implies $\beta = 0$, thus the forall-loop in algorithm 3.10, ln. 1-9 is skipped and instead a direct call to 3.4 is executed in ln. 11. □

Lemma 3.13. In case, a tree serves as input for algorithm 3.6, the algorithm reduces to algorithm 3.1.

Proof. In that case, there are only vertex and edge blocks present in the block transformed graph. The leaves are initialized as previously. All nodes except the root node are processed being vertex blocks with algorithm 3.10, directly forwarding to algorithm 3.4 and updating k accordingly. The root node is either processed as vertex block as well or as edge block. Both times algorithm 3.4 is called directly. Thus, for trees, the algorithm 3.6 behaves analogously to 3.1. □

Lemma 3.14. Calling algorithm 3.6 for a tree, the following holds

1. If $W(p) \neq \emptyset$, it contains all non-dominating weight vectors of the p-containing cluster of a minimal partition of $T(p)$.

2. After each iteration of a loop in algorithm 3.6, ln. 1, the variable k contains the minimum number of edges to cut to obtain a feasible partition.

Proof. Due to lemma 3.13, the algorithm performs the same way as3.1 and we can transfer the result 3.11 directly to algorithm 3.6 when applied to trees. □

Seeing that the algorithm works for the special case of trees as desired, we prove the correctness for the case of a cactus tree following a similar strategy as [7]. First, we show that every partition of a cycle can be found in one configuration induced by one initial cut:

Lemma 3.15. Let C be a cycle consisting of nodes $v_0, ..., v_{\Omega-1}$, each minimal partition can be found in the partitions computed for one of the trees created by opening the cycle with one of the initial cuts $i = 1, ..., \Omega$.

Proof. Assuming there is no cut needed, in the cycle, thus, we can create one cluster containing all nodes $v_0, ..., v_{\Omega-1}$ fulfilling the restrictions, all initial cuts cause the same resulting $W(v_0)$ at the cycle root node. Thus, the partition can be found in the one result partition computed for all initial cuts.
When the partition of the graph parts the cycle C in at least two clusters, there need to be at least two cycle edges cut, one of which may be taken as the first cut. Thus, the partition can be found in the partition created by processing the subtree after opening the cycle at some edge. □

Theorem 3.16. For algorithm 3.6, the following holds

1. For a node p, which is not an inner node of a cycle and which has been processed so that $W(p) \neq \emptyset$, $W(p)$ contains all non-dominating weight vectors of the cluster containing p of a minimal partition of $T(p)$.

2. For a node p, which is an inner node of a cycle and which has been processed as part of algorithm 3.9, $\Gamma(p)$ contains a vector of sets of non-dominating weight vectors and respectively required cuts for minimal partitions of the subtree induced by an initial cut. The initial cut index indicates the index of the component in the vector $\Gamma(p)$.

3. After each iteration of a loop in algorithm 3.6, in which no cycle remains preprocessed without the root node being processed, the variable k contains the minimum number of edges to cut to obtain a feasible partition for the processed subgraph.

Proof. The first two statements are given by construction. For the last statement, there are two ways by which we can have no preprocessed loops after an iteration of the algorithm. Either we close a cycle by processing the cycle root node v. In that case, k is updated in ln. 6, 12, or 17. The return value of the call to algorithm 3.10 computes the number of cuts actually needed for the cycles attached to v is computed and updated according to algorithm 3.10, ln. 7. Or the conditions for the last statement were fulfilled in one iteration and are still fulfilled in the next iteration of the algorithm because either a vertex node or an edge node was processed. □

As a consequence, after the algorithm terminates, a minimal partition for the CTPuC under two constraints is found.

3.2.5 An Initial Glance at the Performance

The algorithm described so far is fairly independent of the characteristic numbers Δ, which denotes the maximum number of children a node can have, and Ω, which describes the maximal size of a cycle involved.
Keeping those two numbers fixed for an input graph $G = (V, E)$ with $n := |V|$ and assuming, that processing one node in algorithm 3.10 ln. 3 can be described by some function $p(n)$ (here not necessarily polynomial), we get

- one call to algorithm 3.10 costs $O(\Omega^{\Delta/2} \cdot (p(n) + \Delta/2))$ time
- one call to algorithm 3.9 costs $O(\Omega^2 \cdot (\Omega^{\Delta/2} \cdot (p(n) + \Delta/2)))$ time
- one call to algorithm 3.6 costs $O(n \cdot \Omega^2 \cdot (\Omega^{\Delta/2} \cdot (p(n) + \Delta/2)))$ time

Lemma 3.17. The maximum number of children, which have to be processed in algorithm 3.10 during the execution of algorithm 3.6 for a binary cactus tree is limited by $\Delta_{max} = 5$.

Proof. The critical blocks are the connection blocks, which correspond directly to a node in the original graph. Since it is a binary cactus graph, these connection blocks can have at most two-block children. Each child block can add at most two-node children to the connection vertex when the child block is a cycle block. And at most one node child, if the child block is an edge block. During the processing of a cycle, we get an additional child on each inner cycle node, as soon as we don't put the cut next to the currently processed node. The various cases are displayed in fig. 3.15. This yields:

$$\Delta \leq \underbrace{2}_{\text{child 1}} + \underbrace{2}_{\text{child 2}} + \underbrace{1}_{\text{cycle neighbour}} = 5 = \Delta_{max} \tag{3.3}$$

□

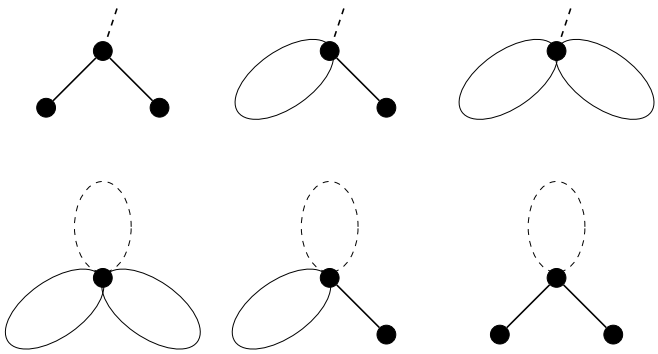


Figure 3.15: Nodes in binary cactus trees can be part of up to three cycles in different configurations. Either one or both child blocks are cycles and or the parent block.

That means that we would have the overall time consumption for the processing of a binary cactus tree limited by

$$O(n \cdot \Omega^{4.5} \cdot p(n))$$

Since Ω is limited trivially by $O(n)$, we can limit it further by

$$O(n^{5.5} \cdot p(n))$$

Chapter 4

Efficiency

At the end of the last chapter, we analyzed the number of elementary calls to algorithm 3.4. The time needed to compute one such call was hidden in $p(n)$, dependent on the size of the graph. This chapter aims to determine this unknown factor following an analogous procedure as presented in [10] [13].

4.1 Efficiency Estimations for TPuC

Let an instance of TPuC be given, wherein the input tree (not necessarily binary) is $T = (V, E)$. Furthermore, each node $v \in V$ was assigned a 2-vector of weights. The tree shall be partitioned so that each component fulfills restrictions R_i on each weight component i.

Theorem 4.1. [10] Let $T = (V, E)$ with weight functions w and restrictions R be an instance of TPuC. Assume T is rooted at v_0 and the maximal outdegree is Δ. Let $p(n)$ denote the size of the longest list $W(v)$ that occurs during the running time of the algorithm. The algorithm can be implemented with a worst-case running time of

$$O(n\Delta^2 p(n)^3)$$

The heart of the reasoning in [10] is to limit the size of $W(v)$ by a polynomial in the number of vertices in the subtree rooted in v (here only given for two weight functions, as well proven in general in [13]):

Theorem 4.2. [13] Let $T = (V, E)$ with weight functions w and restrictions R be an instance of TPuC, where the degree of T is bounded by Δ and $w : V \to \mathbb{N}^2$ and $v \in V$. The number of list items in $W(v)$ can be estimated by

$$O\left(|V(T(v))|^{2\cdot(2\cdot\Delta^{\Delta}-1)}\right)$$

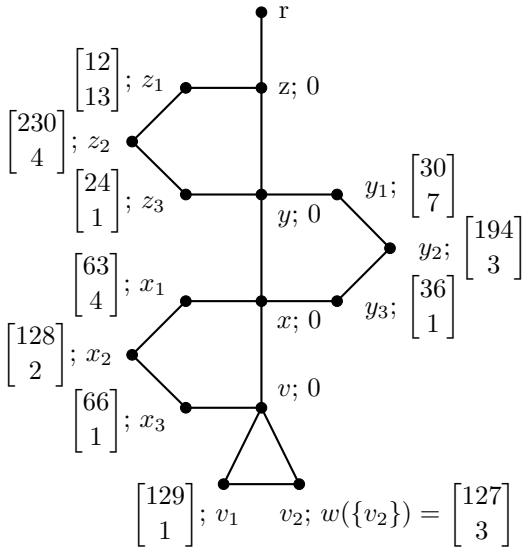


Figure 4.1: Instance of a TPuC with exponential growth of stored weight vectors per node. The weights of the nodes are noted beside the node name.

4.2 Efficiency Estimations for CTPuC

4.2.1 Estimating the Number of Stored Weight Vectors

Estimating the algorithmic complexity of the p-(l,u)-partitioning algorithm in [7] is based on limiting the maximum number of stored tuples per node by $O(pu)$, where p is the maximum number of nodes in the cluster and u the upper weight limit.
This would be tractable for the present problem, however, the algorithm being only pseudopolynomial then, the approach taken by Hamacher et al. [10] was evaluated. In the case of binary trees with two weight functions, a strong bound on stored weight vectors was proven: $O(n^2)$, with $n = |T_v|$ being the number of nodes of the subtree rooted in v. It is desirable to extend this limit to the case of (binary) cactus trees (with some modifications).
Sadly, this is not possible in general - a counterexample is given in the first subsection hinting at a possible exponential growth of the number of stored weight vectors. A general construction instruction is then given in the second subsection. This is followed by a subsection covering rendering possibilities for other parameters. The special cases of cactus trees containing only 3-cycles or 4-cycles are handled in the last subsections.

4.2.2 Counterexample using 5-Cycles

The CTPuC instance shown in fig. 4.1 has the following restrictions

$$R = \begin{bmatrix}255\\27\end{bmatrix} \tag{4.1}$$

It is obvious that the second restriction does not exclude any node. It assures only that no dominating vectors occur. The nodes, which are shared between two cycles are chosen with trivial weight, so that they may be included always.
Let's consider cycle by cycle bottom up:

V-Cycle The V-cycle (containing the nodes, v, v_1, v_2) needs to be divided to fulfill the restrictions, since

$$w(\text{ V-cycle }) = w(v_1) + w(v_2) + w(v) = \begin{bmatrix} 256 \\ 4 \end{bmatrix}$$

wherein the first component would violate the constraints.

$$w(\text{ V-cycle })_1 = 256 > 255 = R_1$$

Excluding exactly one node suffices to comply to the restrictions. The trivially weighted node v can always be included in every adjoining cluster of a feasible solution.

$$W(v) = \left\{ \begin{bmatrix} 127 \\ 3 \end{bmatrix}; \begin{bmatrix} 129 \\ 1 \end{bmatrix} \right\}$$

X-Cycle By construction, the X-cycle allows only three nodes to be combined sufficing the restrictions. The possible cuts are depicted in fig. 4.2

$$W(x) = \left\{ \underbrace{\begin{bmatrix} 191 \\ 6 \end{bmatrix}}_{x-x_1-x_2}; \underbrace{\begin{bmatrix} 190 \\ 7 \end{bmatrix}; \begin{bmatrix} 192 \\ 5 \end{bmatrix}}_{x_1-x-v}; \underbrace{\begin{bmatrix} 193 \\ 4 \end{bmatrix}; \begin{bmatrix} 195 \\ 2 \end{bmatrix}}_{x-v-x_3}; \right\}$$

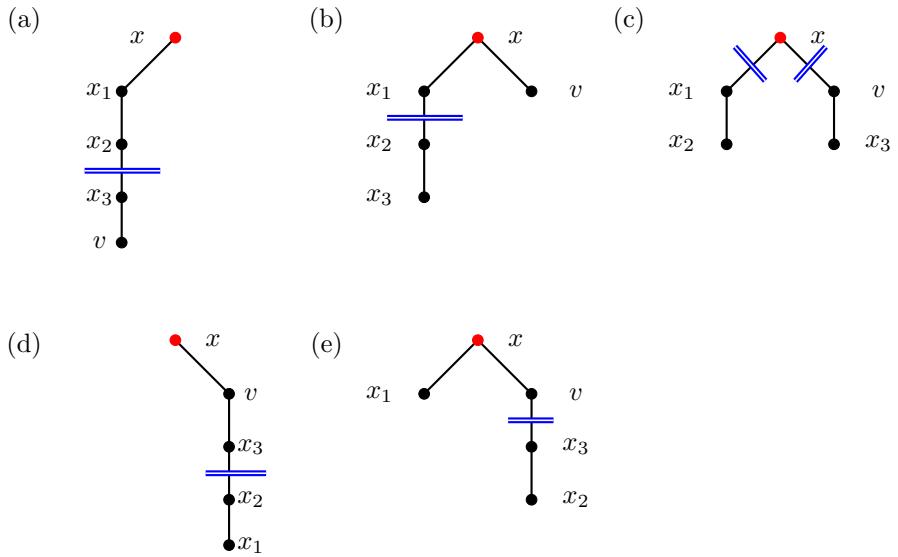


Figure 4.2: Partitioning options for the X-cycle in the example depicted in fig. 4.1. The top node of the subproblem is marked in red. The second cut, which is needed to fulfill the restrictions, is shown in a double stroken blue line (if there are two double stroken lines each of them can be selected but are not required at the same time).

Y-Cycle The Y-cycle is constructed analogously to the X-cycle. The resulting cluster weights at y are

$$W(y) = \left\{ \underbrace{\begin{bmatrix}224\\10\end{bmatrix}}_{y-y_1-y_2}; \underbrace{\begin{bmatrix}220\\14\end{bmatrix}; \begin{bmatrix}221\\13\end{bmatrix}; \begin{bmatrix}222\\12\end{bmatrix}; \begin{bmatrix}223\\11\end{bmatrix}; \begin{bmatrix}225\\9\end{bmatrix}}_{x-y-y_1}; \underbrace{\begin{bmatrix}226\\8\end{bmatrix}; \begin{bmatrix}227\\7\end{bmatrix}; \begin{bmatrix}228\\6\end{bmatrix}; \begin{bmatrix}229\\5\end{bmatrix}; \begin{bmatrix}231\\3\end{bmatrix}}_{y_3-x-y}; \right\}$$

Z-Cycle Consequentially, we have many possible clusters of various weights at z

$$W(z) = \left\{ \underbrace{\begin{bmatrix}242\\17\end{bmatrix}}_{z-z_1-z_2}; \underbrace{\begin{bmatrix}232\\27\end{bmatrix}; \begin{bmatrix}233\\26\end{bmatrix}; \begin{bmatrix}234\\25\end{bmatrix}; \begin{bmatrix}235\\24\end{bmatrix}; \begin{bmatrix}236\\23\end{bmatrix}; \begin{bmatrix}237\\22\end{bmatrix}; \begin{bmatrix}238\\21\end{bmatrix}; \begin{bmatrix}239\\20\end{bmatrix}; \begin{bmatrix}240\\19\end{bmatrix}; \begin{bmatrix}241\\18\end{bmatrix}; \begin{bmatrix}243\\16\end{bmatrix}}_{y-z-z_1}; \right.$$
$$\left. \underbrace{\begin{bmatrix}244\\15\end{bmatrix}; \begin{bmatrix}245\\14\end{bmatrix}; \begin{bmatrix}246\\13\end{bmatrix}; \begin{bmatrix}247\\12\end{bmatrix}; \begin{bmatrix}248\\11\end{bmatrix}; \begin{bmatrix}249\\10\end{bmatrix}; \begin{bmatrix}250\\9\end{bmatrix}; \begin{bmatrix}251\\8\end{bmatrix}; \begin{bmatrix}252\\7\end{bmatrix}; \begin{bmatrix}252\\6\end{bmatrix}; \begin{bmatrix}254\\4\end{bmatrix}}_{y_3-x-y}; \right\}$$

In summary, we observe the following increase of the size of $W(p)$ for this extensible setup

$$\begin{aligned} |W(z)| &= 23 \\ &= 2 \cdot |W(y)| + 1 \\ &= 2 \cdot (2 \cdot |W(x)| + 1) + 1 \\ &= 2 \cdot (2 \cdot (2 \cdot |W(v)| + 1) + 1) + 1 \end{aligned}$$

This suggests that the existence of subsequent cycles can cause exponential growth of $|W(v)|$.

4.2.3 Construction Instruction for Cactus Trees with Exponential Growth of $|W(v)|$

To construct a cactus tree for the counterexample in the most simple way, the construction obeys the following restrictions:

- Only directly connected cycles are contained in the construction. As a consequence, all nodes are part of at least one cycle.
- The construction is a chain of cycles: each cycle has only one cycle child.
- All nodes, which connect two cycles (here called junction nodes), have trivial weight (0, 0).
- The junction nodes are always placed directly neighboring. That means that the root of cycle i, called r_i, and the root of cycle i-1, called r_{i-1}, have a common incident edge (r_i, r_{i-1})
- The bottom cycle is a 3-cycle.

We perform the construction using N_{cycle} 5-cycles and one 3-cycle. The number of nodes contained in the cactus is then

$$n = 3 + N_{cycle} \cdot 4$$

wherein the second summand is the number of nodes of each cycle minus the junction node to the subsequent cycle. This construction is extensible to use m-cycles ($m > 5$) resulting in a larger base for the exponential growth of the size of the set of allowed weight vectors $|W(v)|$. However, a simple extension as provided in the next subsection can be used as well to proof exponential growth for cactus trees with larger Ω.

Step 1: Determine the Weight Scaling and the Restrictions Observe in the example the difference in weights of x_1 and x_3, y_1 and y_3 and z_1 and z_3:

$$|w(z_1) - w(z_3)| = \begin{bmatrix}12\\12\end{bmatrix}$$
$$|w(y_1) - w(y_3)| = \begin{bmatrix}6\\6\end{bmatrix}$$
$$|w(x_1) - w(x_3)| = \begin{bmatrix}3\\3\end{bmatrix}$$

In each cycle the needed difference between the weights of the introduced weights doubles, as the number of allowed configurations does.
This required weight distance is utilized to construct a staggered weight system.

$$\begin{array}{rrrr} (12) & 12 & +\,24 & =36 \\ (6) & 30 & +\,36 & =66 \\ (3) & 63 & +\,66 & =129 \\ (2) & 127 & +\,129 & =256 \end{array}$$

The number in brackets in the beginning of each line describes the additional weight difference needed for each cycle from root to leaf. The calculation for the example is described, in which we set the restriction to $R_1 = 255$, the value of the last line decremented by 1.
In general, the staggering can be computed by

$$\begin{array}{rlll} (3\cdot 2^{N_{cycle}-1}) & \underbrace{3\cdot 2^{N_{cycle}-1}}_{t_{N_{cycle}}} & +\underbrace{2\cdot(3\cdot 2^{N_{cycle}-1})}_{s_{N_{cycle}}} & =: s_{N_{cycle}-1} \\ (3\cdot 2^{i}) & \underbrace{(s_{i+1}-3\cdot 2^{i})}_{t_{i+1}} & +s_{i+1} & =: s_i \qquad \text{for } i=(N_{cycle}-1),...0 \\ (2) & (s_0-2) & +s_0 & =: s_{base} \end{array} \tag{4.2}$$

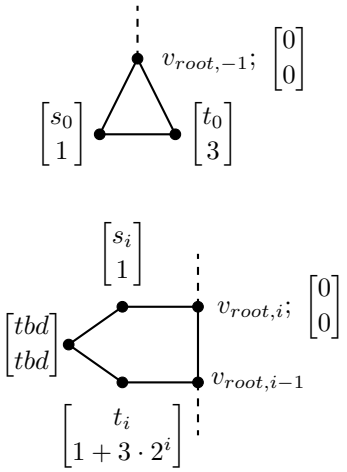
Choosing R as

$$R = \begin{bmatrix} s_{base}-1 \\ N+3\cdot 2^{N_{cycle}} \end{bmatrix} = \begin{bmatrix} \sum_{l=0}^{i} t_l + s_i - 1 \\ N+3\cdot 2^{N_{cycle}} \end{bmatrix} \text{ for } i \in \{0,...,N_{cycle}-1\} \tag{4.3}$$

will then result in at least one node to be excluded from the cycle root cluster in each cycle.

Step 2: Treeconstruction Construct the tree bottom to top:

1. Create a 3-Cycle as follows:



2. Construct the subsequent 5-cycles:
   The entries, where the weight needs to be determined (marked by 'tbd'), need to be 'filled by hand'. Considering the list of non-dominating weight vectors, which are collected in $|W(v_{root,i-1})|$, there will always be a gap in this list of weights from which a suitable weight can be chosen. This gap is elucidated more thoroughly later.

Now, we have a chain of cycles causing an exponential increase in the number of allowed weight vectors.

Lemma 4.3. The bottom 3-cycle has to be divided into multiple clusters to fulfill the restrictions.

Proof. By construction (eq. 4.3)

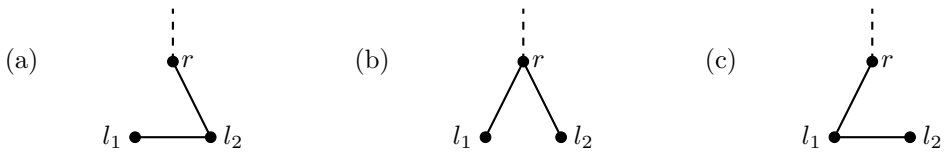
$$R_1 = s_{base} - 1 = s_0 + t_0 - 1 < s_0 + t_0$$

holds, thus, the two non-trivial nodes in the bottom 3-cycle can not be in the same cluster in a valid CTPuC solution. □

Lemma 4.4. For the root node of the bottom 3-cycle the following holds

$$W(v_{root,-1}) = \left\{ \begin{bmatrix}s_0\\1\end{bmatrix}; \begin{bmatrix}t_0\\3\end{bmatrix} \right\}$$

Proof. Following the algorithm, there are three opening-cuts of the cycle, which need to be considered: In configuration (a), starting from bottom (l_1) to the root node r, we'll get a cut at l_2 to its



child l_1, since the combination violates R_1. Since r is trivially weighted, l_2 and r are joined in a set.

Thus the weight vector $(t_0, 3)^T$ is added to $W(r)$. Analogous argumentation for (c) adds the weight vector $(s_0, 1)^T$ to $W(r)$. (b) does not add any new information and processing of configuration (b) creates the same result as processing (a) and (c).
The two weight vectors are non-dominating, since $t_0 = s_0 - 2 < s_0$, thus

$$\begin{bmatrix} t_0 \\ 3 \end{bmatrix} \leqslant \begin{bmatrix} s_0 \\ 1 \end{bmatrix}$$

which proofs the claim. □

Definition 4.5. Gap Let W be a set of non-dominating weight vectors. W has a gap if

$$\begin{aligned} &\exists\, w_1, w_2 \in W \ : \\ &\quad w_{1,1} < w_{2,1} - 1 \\ &\quad w_{1,2} > w_{2,2} + 1 \end{aligned}$$

and

$$\begin{aligned} &\nexists\, v \in W \setminus \{w_1, w_2\} \\ &\quad w_{1,1} < v_1 < w_{2,1} \\ &\quad w_{1,2} > v_2 > w_{2,2} \end{aligned}$$

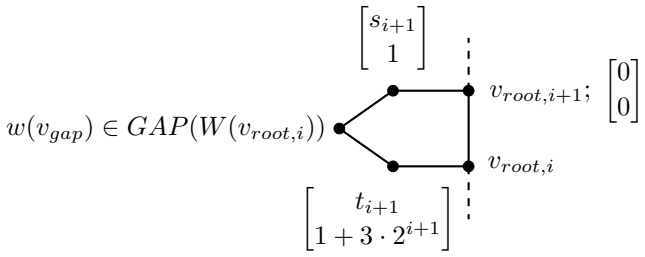
A vector $v \in \mathbb{N}^2$, so that

$$\begin{aligned} &w_{1,1} < v_1 < w_{2,1} \\ &w_{1,2} > v_2 > w_{2,2} \end{aligned}$$

is called **gap weight** for W [1]. The set of non-dominating gap weights for W is denoted by

$$GAP(W) = \{v \in \mathbb{N}^2 \mid \text{v is a gap weight for W}\}$$

Theorem 4.6. For each 5-cycle ($i \in \{0, ..., N_{cycle} - 1\}$) constructed as described previously $\forall i \in \{-1, ..., N_{cycle} - 1\}$ the following holds



1.
$$GAP(W(v_{root,i})) \neq \emptyset$$

2.
$$\forall w \ \in W(v_{root,i}) \quad : \quad \sum_{j=0}^{i+1} t_j \leq w_1 \leq \sum_{j=0}^{i+1} s_j$$

[1] $v = (w_{2,1} - 1, w_{2,2} + 1)$ would be a suitable gap weight

3.
$$\forall w \in W(v_{root,i}) \quad : \quad i+1 \leq w_2 \leq \begin{cases} 3 & \text{for } i = -1 \\ i+3+\sum_{j=0}^{i-1} 3 \cdot 2^j & \text{else} \end{cases}$$

4.
$$\forall w \in W(v_{root,i}) \quad : \quad R_1 - w_1 < s_i$$

5.
$$\forall w \in W(v_{root,i}) \quad : \quad R_1 - w_1 > s_{i+1}$$

6.
$$|W(v_{root,i})| = \begin{cases} 2 \cdot |W(v_{root,i-1})| + 1 & i \geq 0 \\ 2 & i = -1 \end{cases}$$

Proof. By induction.
Using the previous lemma 4.4, we find, that

$$GAP(W(v_{root,-1})) = \left\{ \begin{bmatrix} s_0 - 1 \\ 2 \end{bmatrix} \right\} \neq \emptyset \quad \text{and} \quad |W(v_{root,-1})| = 2$$

Thus for $i = -1$, claim 1 and 6 is proven. Claim 2 for $i = -1$ holds by construction:

$$\forall w \in W(v_{root,-1}) = \left\{ \begin{bmatrix} s_0 - 2 \\ 3 \end{bmatrix}; \begin{bmatrix} s_0 \\ 1 \end{bmatrix} \right\} \quad : \quad s_0 - 2 \leq w_1 \leq s_0$$

For claim 2 and 3 observe $\forall w \in W(v_{root,-1})$

$$\begin{aligned} \sum_{j=0}^{(-1)+1} t_j = t_0 = 127 \leq \quad & w_1 \leq \quad 129 = s_0 = \sum_{j=0}^{(-1)+1} s_j \\ 0 + 1 = 1 \leq \quad & w_2 \leq \quad 3 \end{aligned}$$

Claim 5 and 4 can be proven in general depending on claim 2:

Lemma 4.7. For $i \in \{0, ..., N_{cycle} - 1\}$: If claim 2 holds

$$\forall w \in W(v_{root,i}) \quad : \quad \sum_{j=0}^{i+1} t_j \leq w_1 \leq \sum_{j=0}^{i+1} s_j$$

then with $R_1 = s_{base} - 1$ claim 4 and 5 hold

$$\begin{aligned} \forall w \in W(v_{root,i}) \quad : \quad & R_1 - w_1 < s_i \\ & R_1 - w_1 > s_{i+1} \end{aligned}$$

Proof. 1.

$$\begin{aligned} R_1 - w_1 = \quad & (\sum_{l=0}^{i} t_l + s_i - 1) - w_1 \\ \leq \quad & (\sum_{l=0}^{i} t_l + s_i - 1) - (\sum_{l=0}^{i} t_l) \\ = \quad & s_i - 1 \\ < \quad & s_i \end{aligned}$$

2.

$$\begin{aligned} R_1 - w_1 = & \quad (\sum_{l=0}^{i} t_l + s_i - 1) - w_1 \\ \geq & \quad (\sum_{l=0}^{i} t_l + s_i - 1) - (\sum_{l=0}^{i} s_l) \\ = & \quad s_i - 1 - \sum_{l=0}^{i} (s_l - t_l) \\ = & \quad s_i - 1 - (\sum_{l=1}^{i} 3 \cdot 2^{l-1} + 2) \\ = & \quad s_i - 3 \cdot \frac{1-2^i}{1-2} - 3 \\ = & \quad s_i - 3 \cdot 2^i + 6 \\ = & \quad s_{i+1} + t_{i+1} - 3 \cdot 2^i + 6 \\ > & \quad s_{i+1} \end{aligned}$$

wherein the last inequality holds, since due to the construction, all t_{i+1} are larger than $3 \cdot 2^{N_{cycle}}$.

□

Now, let $i \in \{0, ..., N_{cycle} - 1\}$ be fixed. By hypothesis, $GAP(W(v_{root,})) \neq \emptyset$ and $W(v_{root,i})$ is given with

$$\forall w \in W(v_{root,i}) \quad : \quad \sum_{j=0}^{i+1} t_j \leq w_1 \leq \sum_{j=0}^{i+1} s_j$$

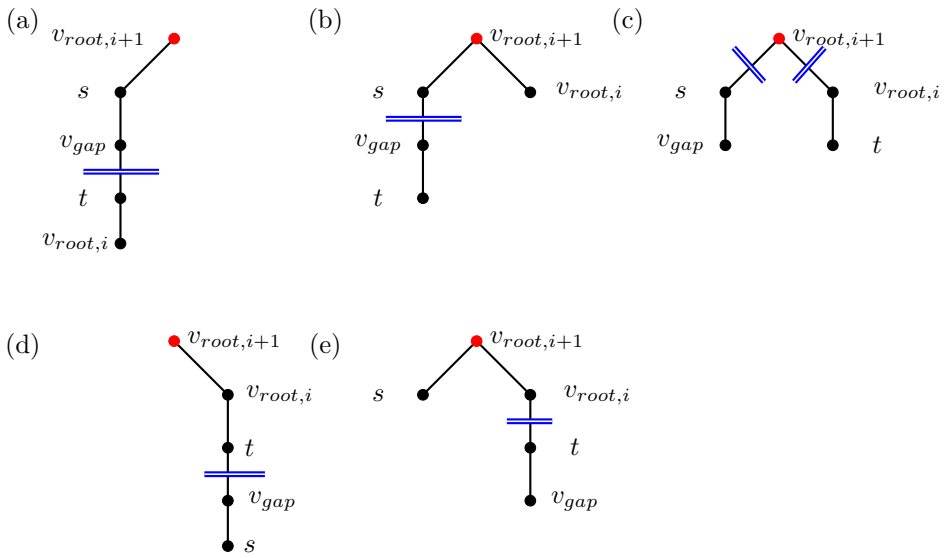


Figure 4.3: Cut options for the 5-cycle in the construction rule. The two cuts in figure (c) are an either or.

Then select $w_{gap} \in GAP(W(v_{root,i-1}))$ and assign it to node v. Before considering the concrete problem, we observe that from the cycle we can combine (independently of connecting edges) $v_{root,i}$

or v_{gap} with either t or s, but never with both (by hypothesis claim 4 and 5). Consequentially, since $w(v_{gap}) > s_{i+1} + t_{i+1}$, we can never combine v_{gap} and $v_{root,i}$.
Processing the cycle as shown in fig. 4.3, we yield processing bottom-up

(a) t can stay connected to $v_{root,i}$. Since v_{gap} and $v_{root,i}$ in one cluster would exceed R_1, we cut the edge between v_{gap} and t. Then no further cut is needed.

(b) v_{gap} remains connected to t, but adding s would exceed R_1. Making the cut, we can fit s and $v_{root,i}$ together with $v_{root,i+1}$ in one cluster.

(c) s stays connected to v_{gap} and $v_{root,i}$ to t, but then we need to cut off either one of the children at $v_{root,i+1}$ and can keep one.

(d) analogously to (a)

(e) analogously to (b)

Hence,

$$W(v_{root,i+1}) = [W(v_{root,i}) \otimes w(t)] \oplus [W(v_{root,i}) \otimes w(s)] \oplus [w(v_{gap}) \otimes w(s)]$$

Since the weight of v_{gap} is somewhere in between the extreme values of $W(v_{root,i})$,

$$\begin{aligned}
&\min \{w_1 | w \in W(v_{root,i+1})\} \\
=&\min \{w_1 | w \in W(v_{root,i})\} + \min \{s_{i+1}, t_{i+1}\} \\
=&\sum_{l=0}^{i} t_l + t_{i+1} = \sum_{l=0}^{i+1} t_l
\end{aligned}$$

and

$$\begin{aligned}
&\max \{w_1 | w \in W(v_{root,i+1})\} \\
=&\max \{w_1 | w \in W(v_{root,i})\} + \max \{s_{i+1}, t_{i+1}\} \\
=&\sum_{l=0}^{i} s_l + s_{i+1} = \sum_{l=0}^{i+1} s_l
\end{aligned}$$

proving claim 2. Similarly, we prove claim 3:

$$\begin{aligned}
&\min \{w_2 | w \in W(v_{root,i+1})\} \\
=&\min \{w_2 | w \in W(v_{root,i})\} + \min \{1, 1 + 3 \cdot 2^{i+1}\} \\
=&(i+1) + 1
\end{aligned}$$

(using the hypothesis in the last line) and

$$\begin{aligned}
&\max \{w_2 | w \in W(v_{root,i+1})\} \\
=&\max \{w_2 | w \in W(v_{root,i})\} + \max \{1, 1 + 3 \cdot 2^{i}\} \\
=&\left(i + 3 + \sum_{l=0}^{i-1} 3 \cdot 2^l \right) + 1 + 3 \cdot 2^i \\
=&(i+1) + 3 + \sum_{l=0}^{(i+1)-1} 3 \cdot 2^l
\end{aligned}$$

To prove the existence of the gap and the size of W we make use of the following lemma

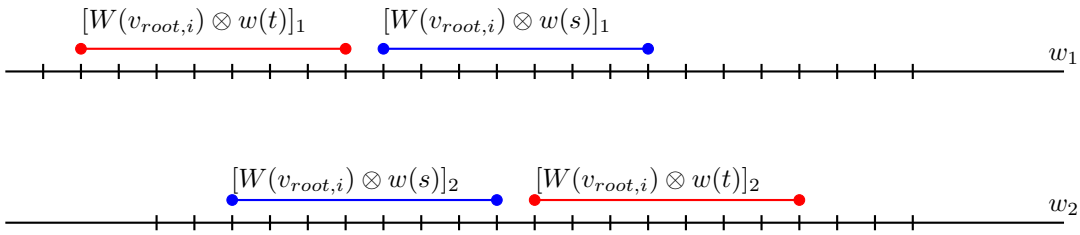


Figure 4.4: Illustration of the two components of the constructed cluster weights

Lemma 4.8. The two sets $W(v_{root,i}) \otimes w(t)$ and $W(v_{root,i}) \otimes w(s)$ do not intersect

$$[W(v_{root,i}) \otimes w(t)] \cap [W(v_{root,i}) \otimes w(s)] = \emptyset$$

and all elements are non-dominating, thus

$$\forall w_t \in [W(v_{root,i}) \otimes w(t)] \;\; \forall w_s \in [W(v_{root,i}) \otimes w(s)] \quad : \quad w_t \lneqq w_s$$

Proof. We proof the first claim by showing, that the difference between t_{i+1} and s_{i+1} is larger than the span of the first components of $W(v_{root,i})$

$$\begin{aligned}
&\max \{w_1 \,|w \in W(v_{root,i})\} - \min \{w_1 \,|w \in W(v_{root,i})\} \\
=&\sum_{l=0}^{i} s_l - \sum_{l=0}^{i} t_l = \sum_{l=0}^{i} (s_l - t_l) \\
=&\sum_{l=0}^{i} 3 \cdot 2^{l-1} = 3\frac{1-2^i}{1-2} + 2 \\
=&3 \cdot 2^i - 1 = s_{i+1} - t_{i+1} - 1
\end{aligned}$$

Thereby the first component fulfills

$$\forall w_t \;\in [W(v_{root,i}) \otimes w(t)] \;\; \forall w_s \in [W(v_{root,i}) \otimes w(s)] \quad : \quad w_{t,1} < w_{s,1}$$

Proceeding in the same way for the second component the following holds

$$\begin{aligned}
&\max \{w_2 \,|w \in W(v_{root,i})\} - \min \{w_2 \,|w \in W(v_{root,i})\} \\
=&(i + 3 + \sum_{l=1}^{i} 3 \cdot 2^i) - (i+1) = 2 + 3\frac{1-2^i}{1-2} = 3 \cdot 2^i - 1
\end{aligned}$$

Thus, the difference between the second components of the weights of s and t is larger than the span of the second components of the weights in $W(v_{root,i})$. Graphically shown, we'll have the situation as depicted in fig. 4.4. The conclusion is that no dominating weight vector are contained in the union of the two sets, or equivalently that

$$\forall w_t \in [W(v_{root,i}) \otimes w(t)] \;\; \forall w_s \in [W(v_{root,i}) \otimes w(s)] \quad : \quad w_t \lneqq w_s$$

□

Since all weight vectors in

$$[W(v_{root,i}) \otimes w(t)] \cup [W(v_{root,i}) \otimes w(s)] \cup W(v_{gap}) \otimes w(s)$$

are non-dominating and the sets are pairwise disjoint we conclude claim 6.
$W(v_{root,i})$ has (by hypothesis) at least one gap at w_{gap} so that $[W(v_{root,i}) \otimes w(t)] \oplus [W(v_{root,i}) \otimes w(s)]$ has two gaps, one at $w_{gap} + t_{i+1}$ and the other one at $w_{gap} + s_{t+1}$. When the latter one is filled by the weight vector $[w(v_{gap}) \otimes w(s)]$ the first one remains. Thus, $GAP(W(v_{root,i+1})) \neq \emptyset$ (claim 1). Given these points, all claims are proven for $i+1$ assuming that they hold for i. By induction this holds for all $i \in \{0, .., N_{cycle} - 1\}$. □

4.2.4 Construction Rendering for Cycles with more than 5 Nodes

The construction can easily be rendered for n-cycles with $n > 5$ by introducing trivially weighted nodes as shown in fig. 4.5 exemplary for 9-cycles (in blue the additional nodes). using this polynomial reduction of problems with 5-cycles to problems with n-cycles, it is clear that the problem increases in difficulty with increasing Ω.

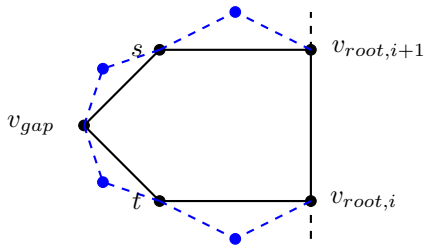


Figure 4.5: Extension of the example for the CTPuC with exponential growth of $|W(v)|$ for 9 nodes per cycle.

4.2.5 General Limitations for the Number of Stored Weight Vectors

As shown in the previous subsections, we can not extend the strong limits, as given in [10], to the case of cactus trees. However, we can formulate a general limit leading to a pseudopolynomial algorithm similar to [7]

Lemma 4.9. Executing algorithm 3.6, the set of stored weight vectors $W(v)$ at a node v is limited by

$$|W(v)| \leq min\{R_1, R_2\}$$

Proof. This limitation is the result of the elimination of dominating weight vectors. W.r.o.g. let $min\{R_1, R_2\} = R_1$. Assuming that $|W(v)|$ has more than R_1 elements there must be two weight vectors w_1, w_2 so that $w_{1,1} = w_{2,1}$. Thus, one weight dominates the other and can be eliminated □

4.2.6 Special Case: Only 3-Cycles

The case in which a cactus tree contains at most 3-cycle is special because the cycle root is directly connected to both other cycle nodes.

Lemma 4.10. Partitioning a binary cactus tree with maximal cycle size $\Omega = 3$ with algorithm 3.6, the size of the set $W(v)$ can be limited by

$$|W(v)| \in O(n^{2(2\Delta^{\Delta}-1)})$$

wherein n denotes the number of nodes in the subtree T_v rooted in v.

Proof. The presence of the cycle closing edge does not add any more options of creating clusters in comparison to a simple tree, since both child nodes of a cycle root are already connected to the cycle root. Thus, the result from theorem 4.2 can be transferred directly to the case of cactus trees with only 3-cycles. □

4.2.7 Special Case: Only 4-Cycles

For the case of 4-cycles, we consider all cut options to conclude. However, it is only relevant to compare cut options with the same number of cuts. The vertex naming of the 4-cycle is shown in fig. 4.6.

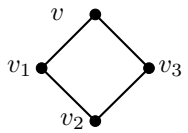


Figure 4.6: Labelling of nodes in 4-cycles for evaluations of cut options.

The three trivial options are

- no cut at all needed - then we can pose the same limit as for the tree case since we can open the circle at an arbitrary position and treat the root node as a normal tree node

$$W(v) = W(v) \otimes W(v_1) \otimes W(v_2) \otimes W(v_3) \neq \emptyset$$

- one cut is not possible in cycles. If one cut is sufficient, no cut is needed at all.
- four cuts needed. No two neighboring nodes of the cycle can fit in one set together according to the restrictions R, thus

$$\begin{aligned} W(v) \otimes W(v_1) &= \emptyset \\ W(v_1) \otimes W(v_2) &= \emptyset \\ W(v_2) \otimes W(v_3) &= \emptyset \\ W(v_3) \otimes W(v) &= \emptyset \end{aligned}$$

  We'll end up with one element in

$$W(v) = \{w(v)\}$$

  Again, this is the same result as if we would consider a tree with the cycle cut open at an arbitrary position.

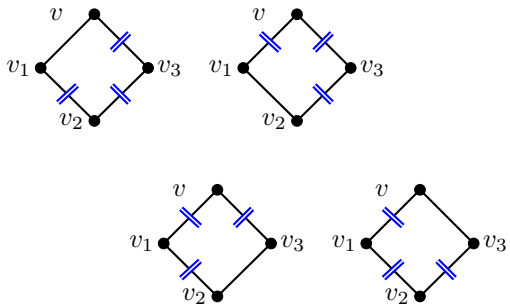


Figure 4.7: Options for 3 cuts in a 4-cycle

Another easily treatable option is the one with three cuts (see figure 4.7). The top node has the same options as in the case of a binary tree: either both children v_1 and v_3 are cut, or one of both is kept. In this case, no node is in two valid partitions in one set with v so that the cycle presence does not increase the size of $W(v)$.

It remains to consider the case, where two cuts are needed to create a valid partition. All options are shown in table 4.1.

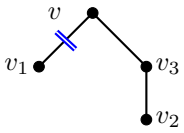
W.r.o.g. we can assume that the nodes are have non-zero weights. Then we can investigate which configurations can contribute significantly to $W(v)$ at the same time. E.g. when the algorithm finds configuration 1, all other valid configurations dominate the weight of configuration 1 and are thus eliminated. When configuration 2 contributes non-trivially to $W(v)$, configuration 1 is not feasible since 2 would dominate 1 and, thus, would not contribute. A summary of all combination options is shown in table 4.2.

Now, we can see the cases, which are dangerous for the growth of $|W(v)|$. When configurations 4 and 6, 5 and 6 or 4 and 5 occur at the same time v_i for $i \in \{1, 2, 3\}$ would be in multiple valid partitions in one set with v. The worst case would then result in $|W(v)| > 2 \cdot |W(v_i)|$ for $i \in \{1, 2, 3\}$.

All the other configuration combinations can be mapped to the case of trees. E.g if configurations 2 and 5 occur together and contribute significantly to $W(v)$, then configurations 1 and 3 can not contribute significantly. This is because they would be dominated by 2 and 5 and thus contributions from configurations 2 and 5 would be eliminated from $W(v)$. Furthermore, 4 and 6 would dominate configuration 2 and would thus not contribute to $W(v)$. So the relevant partition would be equivalent to the partition of the subtree without edge (v_2, v_3).

In the following, we'll reason that configurations 4, 5, and 6 can never occur at the same time (excluding partitions where we need more than two cuts because they are not relevant to the algorithm as long as we have a valid one with two cuts).

Case 4: Initial Cut Option (a) Here, we start with cut option (a) for case 4. This means that the initial cut was made at edge (v_1, v_2) and the second cut at (v, v_1). This leads to the conclusion



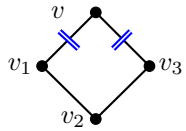
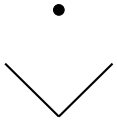
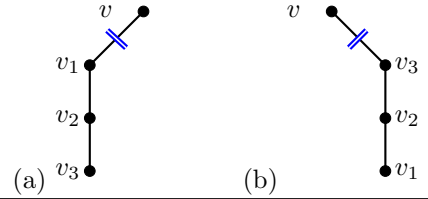
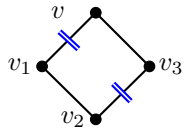
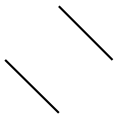
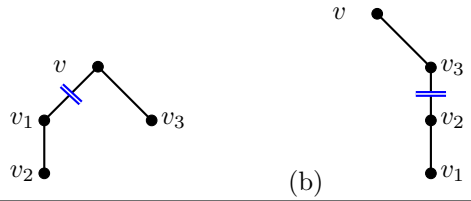
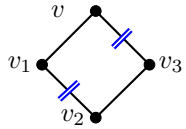
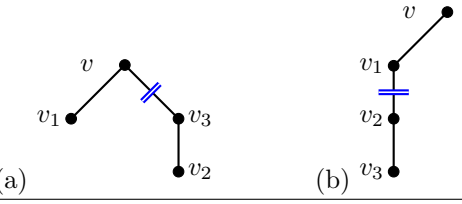
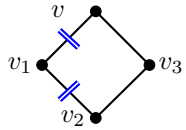
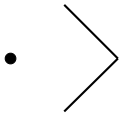
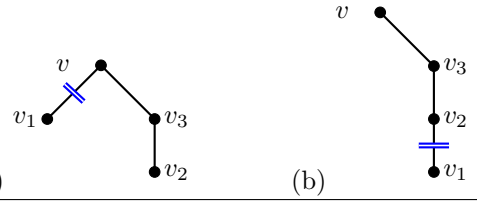
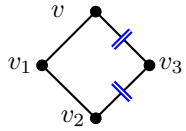
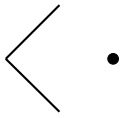
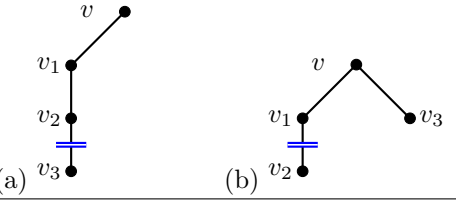
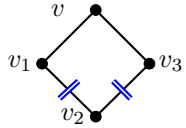
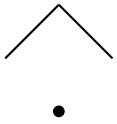
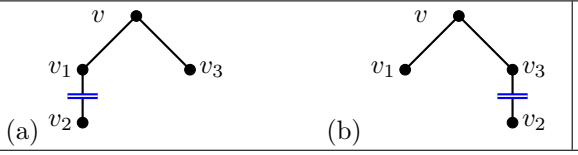
No	2-Cut Configuration	Schematic	Initial Cut Configuration	Comb.
1				-
2				3, 5
3				2, 4
4				3,5,6
5				2,4,6
6				4,5

Table 4.1: Possible 2-cuts and corresponding initial cuts in the algorithm for four cycles. The last column indicates, which configurations can contribute at the same time to $W(v)$ (excluding dominating configurations)

that

$$W(v) \otimes (W(v_2) \otimes W(v_3)) \neq \emptyset$$
$$(W(v) \otimes W(v_2) \otimes W(v_3)) \otimes W(v_1) = \emptyset$$

When processing the children of v, we have two options

1. $W(v) \otimes W(v_1) = \emptyset$ Starting from the opening cut at (v_2, v_3), we could not join v and v_1 in the same cluster and would need an additional cut at (v, v_1). This leaves us with configuration 2, which would dominate 4. Since we assumed that configuration 4 contributes significantly,

Configuration	Dominates at v	Is dominated at v by	Can contribute simultaneously to $W(v)$ with
1	-	2,3,4,5,6	-
2	1	4,6	3,5
3	1	5,6	2,4
4	1,2	-	3,5,6
5	1,3	-	2,4,6
6	1,2,3	-	4,5

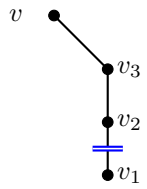
Table 4.2: Relations between configurations and possible simultaneous contributions to $W(v)$

this can not be the case. Starting from the initial opening cut at (v, v_3), we would end up (by the same reasoning) at configuration 1. This in turn leads us to a contradiction to the assumption that case 4 contributes significantly to $W(v)$. So this can not be true.

2. $W(v) \otimes W(v_1) \neq \emptyset$ must hold, since the last option led to contradictions. Configuration 3 is a valid partition as well and configurations 5 and 6 would dominate 3.

Summarized: Assuming that configuration 4 (created by the initial cut (a)) contributes significantly to $W(v)$ configurations 5 and 6 can not contribute.

Case 4: Initial Cut Option (b) Starting with we conclude, that



$$\begin{aligned} W(v_1) \otimes W(v_2) &= \emptyset \\ W(v_2) \otimes W(v_3) \otimes W(v) &\neq \emptyset \end{aligned}$$

When considering the initial cut at (v, v_3) and knowing that $W(v_2) \otimes W(v_3) \neq \emptyset$, we would need to cut (v_1, v_2). This creates the cycle partition 3 (unless more edges are cut). Since 5 and 6 dominate 3, $W(v)$ can not contain significant contributions of 5 and 6. We can only avoid a valid configuration 3 if we assume that

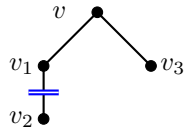
$$W(v_1) \otimes W(v) = \emptyset$$

(which would prevent cutting edges yielding configurations 5 or 6) or if we assume that the weight vectors of configuration 3 dominate the ones of configuration 4

$$(W(v_1) \otimes W(v)) \oplus (W(v_3) \otimes W(v_2) \otimes W(v)) = W(v_3) \otimes W(v_2) \otimes W(v)$$

which would result in possible contributions of partitions 5 and 6 to be dominating as well. Consequentially, when processing the 4-cycle, we can not get significant contributions from configuration 4 at the same time as 5 or 6.

Case 5 Case 5 with the two initial cut options can be shown symmetrically to case 4.



Case 6: Initial Cut Option (a) Starting with the opening cut at (v_2, v_3) and getting significant contributions to $W(v)$ by configuration 6, we'll need a second cut at (v_1, v_2) so that the following must hold

$$W(v_1) \otimes W(v_2) = \emptyset$$
$$W(v_1) \otimes W(v) \otimes W(v_3) \neq \emptyset$$

When we start with the initial cut at (v_1, v_2), we either get configuration 6 again, if

$$W(v_2) \otimes W(v_3) = \emptyset$$

or else configuration 3. Since the latter alternative would cause configuration 6 to dominate and thus its contributions being eliminated from $W(v)$ while excluding dominating vectors, we assume the first case. Starting with initial cuts at either (v, v_1) or (v, v_3), we would then need two more cuts (at (v_1, v_2) and at (v_2, v_3)) to comply to the restrictions. These three cuts would not be considered in our solution, since we have a valid 2-cut solution.

Case 6: Initial Cut Option (b) Analogously to option (a).
This leads us to formulate the following lemma

Lemma 4.11. Partitioning a binary cactus tree with maximal cycle size $\Omega = 4$ with algorithm 3.6, the size of the set $W(v)$ can be limited by

$$|W(v)| \in O(n^{2(2\Delta^{\Delta}-1)})$$

wherein n denotes the number of nodes in the subtree T_v rooted in v. Δ is the maximum outdegree of every ode limited by $\Delta_{max} = 5$.

Proof. The presence of the cycle closing edge does not add any more options for creating clusters in comparison to a simple tree, as elucidated in the previous discussion. Thus, the result from theorem 4.2 can be transferred directly to the case of cactus trees with only 4-cycles. □

Chapter 5

Conclusion

This generalization of the partitioning algorithm proposed by Hamacher et al. [10] to cactus trees has been shown to be computable in polynomial time for cactus trees, which have at most 4-cycles. Sadly, it could not be shown that the generalization results in a polynomial algorithm for all cactus trees and is only pseudopolynomial following intuitive reasoning.
The algorithm for CTPuC is open for improvements, e.g. the similarity of solutions for cycles hints that there might be a more efficient way of computation, storage, and processing. Especially an extension of the method of storing possible solutions in intervals and processing those, as done in [9] and [7], is promising to diminish the maximum size of $W(v)$. Albeit it is unclear how the interval view can be extended to multiple constraints the idea points to the existence of a polynomial-time algorithm even for cactus trees with larger cycles.
An alternative to showing that the algorithm is extensible to cactus trees with larger cycles and thus showing that the problem is solvable in polynomial time is to show that CTPuC with two constraints is NP-complete for the cycle degree $\Omega \geq 5$. This requires more thorough investigations on the basis, which was given in this thesis.

Index

Bibliography

[1] Per-Olof Fjällström. *Algorithms for graph partitioning: A survey*, volume 3. Linköping University Electronic Press Linköping, 1998.

[2] David A Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. *Graph partitioning and graph clustering*, volume 588. American Mathematical Society Providence, RI, 2013.

[3] Bruce Hendrickson and Tamara G Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26(12):1519–1534, 2000. Graph Partitioning and Parallel Computing.

[4] Jin Kim, Inwook Hwang, Yong-Hyuk Kim, and Byung-Ro Moon. Genetic approaches for graph partitioning: a survey. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 473–480, 2011.

[5] Konstantin Andreev and Harald Racke. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.

[6] George Karypis and Vipin Kumar. Multilevel algorithms for multi-constraint graph partitioning. pages 28– 28, 12 1998.

[7] Maike Buchin and Leonie Selbach. Partitioning algorithms for weighted cactus graphs, 2020.

[8] Carlos Eduardo Ferreira, Alexander Martin, C Carvalho de Souza, Robert Weismantel, and Laurence A Wolsey. The node capacitated graph partitioning problem: a computational study. *Mathematical programming*, 81(2):229–256, 1998.

[9] Takehiro Ito, Takao Nishizeki, Michael Schröder, Takeaki Uno, and Xiao Zhou. Partitioning a weighted tree into subtrees with weights in a given range. *Algorithmica*, 62:823–841, 04 2012.

[10] Anja Hamacher, Winfried Hochstättler, and Christoph Moll. Tree partitioning under constraints — clustering for vehicle routing problems. *Discrete Applied Mathematics*, 99(1):55 – 69, 2000.

[11] Reinhard Diestel. *Graph Theory.* Springer, 5th edition, 2017.

[12] Kalyani Das. Some algorithms on cactus graphs. *Annals of Pure and Applied Mathematics*, 2(2):114–128, 2012.

[13] Anja Hamacher. Baumzerlegungen unter nebenbedingungen - ein clusterverfahren zur lösung praktischer vehicle-routing-probleme. *Ph.D. Thesis, Universität zu Köln*, 1997.