A polynomial-time partitioning algorithm for weighted cactus graphs

Maike Buchin and Leonie Selbach

Ruhr-University Bochum
maike.buchin@rub.de, leonie.selbach@rub.de

Abstract

Some graph partitioning problems are known to be NP-hard for certain graph classes and polynomially solvable for others. We study the problem of partitioning the vertex set of a weighted cactus graph into clusters with bounded weights and present a polynomial-time algorithm that solves the problem with the optimal as well as some fixed number of clusters.

1 Introduction

Let $G = (V, E)$ be a cactus graph, that is a graph where every two simple cycles have at most one vertex in common. Every vertex v in G is assigned a non-negative integer weight $w(v)$. A *(vertex) partition* of G is a partition of the vertex set V into pairwise disjoint clusters such that the induced subgraphs are connected. Given two non-negative integers l and u (with $l \leq u$), we call a partition (l, u)*-partition* if the weight of each cluster is at least l and at most u. We consider different variants of this problem: For the *minimum/maximum* (l, u)*-partition problem* we want to find a (l, u)-partition of a graph with the minimal resp. maximal number of clusters. And for the p-(l, u)*-partition problem* the goal is to find a partition with exactly p clusters (see Fig. 1).

Dyer and Frieze studied the complexity of different graph partitioning problems [3]. They showed that partitioning a general graph into connected components (or trees) of a given size is NP-complete, even for planar bipartite graphs – but polynomially solvable for both series-parallel graphs and trees. Cordone and Maffioli consider partitioning an edge-weighted graph into trees with different optimization goals [2]. They proved that this problem is NP-hard for all cases if the graph has additional vertex weights and a weight constraint – as a lower and upper weight bound for each tree – is added. The related tree-equipartition problem, where the partition should consist of clusters with weight as equal as possible, proved to be NP-complete for trees as well as 2-spiders, but can be solved in polynomial time for other graph classes, such as stars, worms and caterpillars [6]. Ito et al. showed that the (l, u)-partition problems are NP-hard for series-parallel graphs, but pseudo-polynomial algorithms exist [5]. Ito et al. proved that the decision variant of those problems can be solved in polynomial time if the graph is a weighted tree [4]. In this paper we present a (l, u)-partitioning algorithm for weighted cactus graphs whose runtime can be reduced to solve the given problems in polynomial time. Because of length constraints we excluded the proofs, these can be found in the full version of this paper [1].

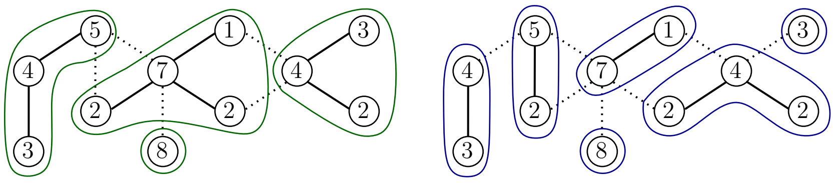


Figure 1 Left: Minimum $(3, 12)$-partition. Right: 6-$(3, 12)$-partition.

36th European Workshop on Computational Geometry, Würzburg, Germany, March 16–18, 2020.

2 Preliminaries

Let $G = (V, E, w)$ be a weighted cactus graph. We want to find a partition of G into clusters V_i such that the weight of each cluster $w(V_i) = \sum_{v \in V_i} w(v)$ lies between the two given bounds: $l \leq w(V_i) \leq u$. For our algorithm we represent the graph G as a tree T_G using depth first search (DFS) on an arbitrary vertex. This vertex becomes the root of the tree (see. Fig. 2). Every vertex of G is a node in T_G and every cycle degenerates to a path. We store a cycle by its path $\langle x, y \rangle$ in T_G and define x as the *start node* and y as the *end node* of this cycle. We denote the cycle by $C(x, y)$. If two cycles are adjacent, their common node is part of both cycles. When building the tree T_G, this node will become the start node of at least one of them.

▶ **Lemma 2.1.** *Let v be a node in T_G. If $v \in C(x, y)$ and $v \in C(x', y')$ for $(x', y') \neq (x, y)$, then v equals either x or x' (or both).*

▶ **Corollary 2.2.** *For each node v there exists at most one cycle in T_G which contains v but where v is not the start node.*

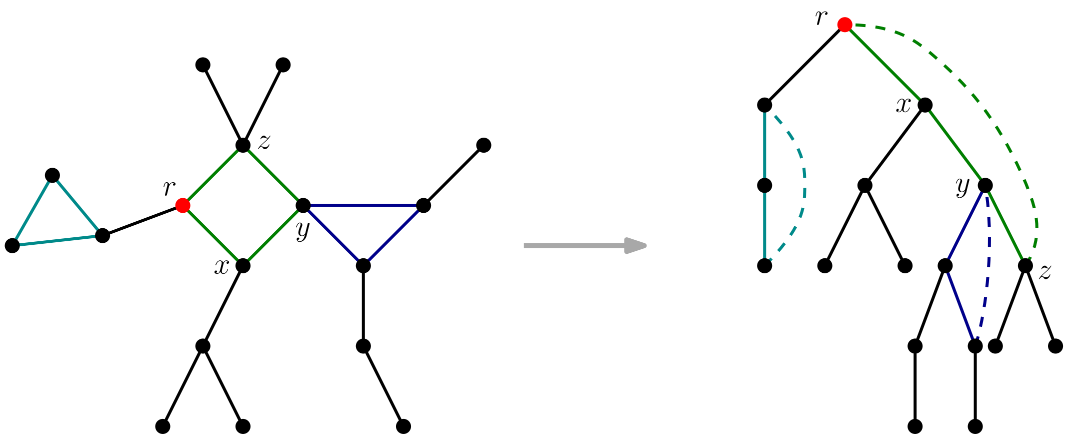


Figure 2 Turning a cactus graph G into a tree T_G by starting the DFS with the red vertex r. The cycle containing the node r is represented as a path $\langle r, x, y, z \rangle = C(r, z)$ with start node r and end node z.

▶ Remark. Let v be a node with s children $v_1, v_2, \ldots, v_s$ and let v be part of a cycle $C(x, y)$ with $v \neq x$. Because of Cor. 2.2 there is at most one such cycle. If $v \neq y$, there exists a child v' of v with $v' \in C(x, y)$ as well. W.l.o.g. we sort the children of v such that $v' = v_s$. This allows easier definitions for the algorithm since these nodes have to be treated differently.

Let us define some further notations. Let T_v be the subtree rooted in the node v. We have $T = T_r$ for r being the root of T. We define T_v^i as the subtree with root v and its first i children (see Fig. 3). If v has s children, we have $T_v^s = T_v$. Consider a partition P of the graph. Let $|P|$ be the size of the partition, meaning the number of clusters in P. For a node v we denote the cluster of P that contains v by C_v.

Let P be a (l, u)-partition of a general tree. If the lower weight bound l equals zero, the partition P induces a feasible $(0, u)$-partition of any subtree T_v. This is not necessarily the case for a general (l, u)-partition since the weight of the cluster containing the node v might not exceed the lower weight bound l. We call such a partition of a subtree T_v– where $w(C_v) \leq u$ and $l \leq w(C') \leq u$ for every cluster $C' \neq C_v$ – an *extendable* (l, u)-partition.

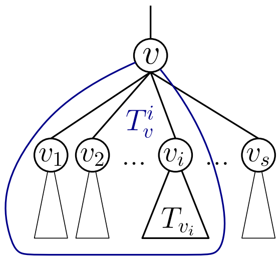


Figure 3 Definition of a subtree T_v^i of a node v.

By adding further nodes to the cluster C_v we can extend such a partition to be a feasible (l, u)-partition. For a subtree T_v (or resp. T_v^i) we define a set $S(T_v)$ as follows:

$$S(T_v) = \{(x, k) \mid \exists \text{ extendable } (l, u)\text{-partition } P \text{ of } T_v \text{ such that } |P| = k \ \wedge \ w(C_v) = x\}$$

Thus, a tuple (x, k) corresponds to a possible partition of T_v, where x is the weight of the cluster containing the node v and k is the number of clusters.

▶ **Lemma 2.3.** *A rooted tree T has a p-(l, u)-partition if and only if there exists a tuple $(x, p) \in S(T)$ such that $l \leq x \leq u$.*

Ito et al. described a method for solving the partition problems for weighted trees by computing the sets S for every subtree T_v^i bottom-up. With an adjustment they can decide the partition problem in time $\mathcal{O}(p^4 n)$. Cactus graphs, however, present the challenge of having cycles and therefore considerably more partition possibilities. We show how to extend their algorithm to cactus graphs while retaining polynomial time.

3 Partitioning Algorithms

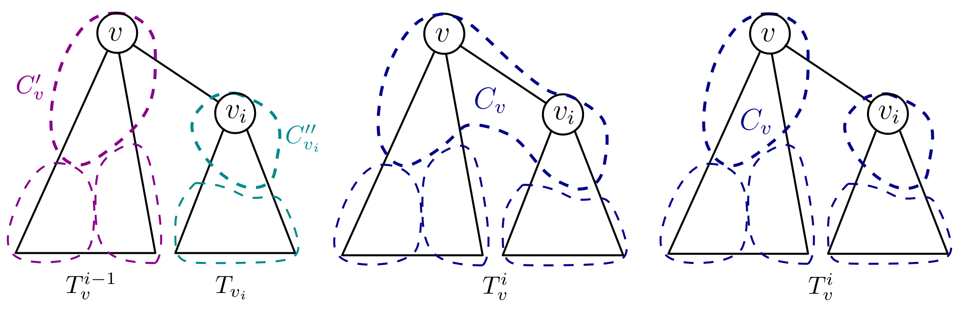
First, let us consider the decision of the p-(l, u)-partition problem, that is deciding if there exists a (l, u)-partition of a given cactus graph G into p clusters. Using the tree representation of G described in Sec. 2, we follow the general idea of Ito et al. but include a procedure that deals efficiently with cycles. Similary, we present a general algorithm which we adjust to solve the problems in polynomial time.

For each subtree T_v^i we compute all extendable (l, u)-partitions with at most p clusters. This can be done by combining the computed partitions of T_v^{i-1} and T_{v_i} as follows. First, let us consider the case that v and v_i are not part of a cycle. For each combination of partitions for the given subtrees we have two options: Either we merge the clusters containing v resp. v_i or we do not (see Fig. 4). Obviously, we keep only the generated partitions that are extendable (l, u)-partitions with size less than p. We have to make sure that in option 1 (Fig. 4b) the weight of the new cluster C_v does not exceed the upper bound u. In option 2 (Fig. 4c) P'' has to be a feasible (l, u)-partition – not just an extendable one. Therefore, we have to make sure that the weight of the cluster C''_{v_i} fulfills the lower weight constraint.

We compute those partitions using the sets S as defined in the previous section. We use a specific set-operation denoted by $\oplus$. For two sets $A = S(T_v^{i-1})$ and $B = S(T_{v_i})$ we define:

$$\begin{aligned} A \oplus B = &\{(x_1, k_1 + k_2) \mid l \leq x_2, k_1 + k_2 \leq p, (x_1, k_1) \in A, (x_2, k_2) \in B\} \qquad (1) \\ &\cup \{(x_1 + x_2, k_1 + k_2 - 1) \mid x_1 + x_2 \leq u, k_1 + k_2 - 1 \leq p, (x_1, k_1) \in A, (x_2, k_2) \in B\}\,. \end{aligned}$$

The second line of this definition corresponds to all feasible partitions we obtain by merging the clusters and the first line corresponds to those we get without merging. We assume



(a) Partitions P' and P''. **(b)** Option 1: $v_i \in C_v$. **(c)** Option 2: $v_i \notin C_v$.

Figure 4 Combining the two partitions P' of T_v^{i-1} and P'' of T_{v_i} into a partition P of T_v^i.

that $w(v) \leq u$ for all nodes v – otherwise a (l,u)-partition of G would not exist. We set $S(T_v^0) = \{(w(v), 1)\}$ and compute $S(T_v^i) = S(T_v^{i-1}) \oplus S(T_{v_i})$ iteratively for all $1 \leq i \leq c_v$ where c_v is the number of children of the node v.

Now, let us consider cycles. Deleting one edge of a cycle leads to a path which we can partition. For a cycle of length m there are m different paths. If we take the union over all partitions of these paths, we obtain all possible partitions of the cycle. This remains true even if we declare one node of the cycle as the "root" and consider the paths as trees (see. Fig. 5). We call these the different *configurations* of a cycle which we number from 1 to m. To find all partitions the m-th configuration is not necessary.

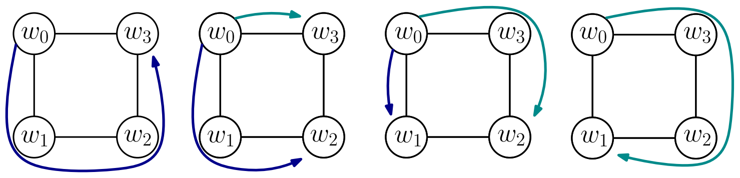


Figure 5 Configurations in a cycle of length four.

Let C be a cycle in G of length m and $C(w_0, w_{m-1}) = \langle w_0, w_1, \ldots, w_{m-1}\rangle$ be the corresponding cycle in T_G. Let each node w_i have c_i children and let w_1 be the k-th child of w_0. Remember that for all other nodes w_{i+1} is always the c_i-th, i.e. the last, child of w_i (Rem. 2). Similarly, we consider different configurations of a cycle in our tree representation (see. Fig 6). The first one is the original subtree as in T_G. For the second configuration we introduce the node w_{m-1} with its corresponding subtree as an additional child of w_0 and remove it from w_{m-2}. We continue this procedure of removing and adding nodes and their corresponding subtrees. Thus, in configuration j the node w_{m-j+1} is no longer the child of w_{m-j} but of w_{m-j+2}.

▶ **Lemma 3.1.** *Let C be a cycle in G and $C(w_0, w_{m-1})$ the corresponding cycle in T_G. A partition P of C can be found in one of the $m-1$ configurations of $C(w_0, w_{m-1})$.*

Let us transfer this to the sets $S(T_v^i)$ as computed by the algorithm. We introduce additional sets $S_j(T_{w_i})$ for each node w that is part of a cycle. Each set $S_j(T_w)$ contains tuples (x, k) corresponding to an extendable (l,u)-partition of T_w in its configuration j. As before, such a tuple corresponds to a partition of size k where the cluster containing the node w has weight x. We can define the different computations for $S(T_{w_i})$ using the $\oplus$-operation as above. First, let us consider the case where $i = m-1$. Until the third configuration its subtree (and

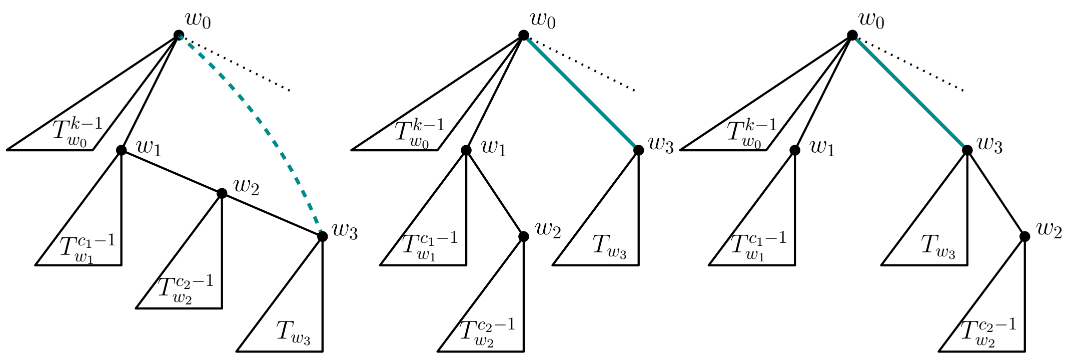


Figure 6 The different considered configurations for a cycle of length four in the subtree of w_0.

therefore the computation) stays the same. Then, the node obtains a additional child w_{m-2} whose subtree (and hence partition set) changes with each configuration.

$$S_j(T_{w_{m-1}}) = \begin{cases} S(T_{w_{m-1}}) \oplus S_j(T_{w_{m-2}}) & \text{for } j > 2, \\ S(T_{w_{m-1}}) & \text{otherwise.} \end{cases} \tag{2}$$

When $0 < i < m-1$, that is w_i is neither the start nor end node of the given cycle, we have to consider three cases: In configuration $m-i$ and $m-i+1$ the subtree T_{w_i} is equal to $T_{w_i}^{c_i-1}$ because the last (c_i-th) child is removed. If $j < m-i$, w_i still has w_{i+1} as a child but with a different subtree. If $j > m-i+1$, w_i does not have w_{i+1} as its last child but w_{i-1} instead. This can be formalized in the following way:

$$S_j(T_{w_i}) = S_j(T_{w_i}^{c_i}) = \begin{cases} S(T_{w_i}^{c_i-1}) \oplus S_j(T_{w_{i+1}}) & \text{for } j < m-i, \\ S(T_{w_i}^{c_i-1}) \oplus S_j(T_{w_{i-1}}) & \text{for } j > m-i+1, \\ S(T_{w_i}^{c_i-1}) & \text{otherwise.} \end{cases} \tag{3}$$

Now let us consider the start node w_0 of the cycle. For the first configuration we have to combine the partitions of the subtree $T_{w_0}^{k-1}$ with the partitions of the subtree rooting in w_1. In case $j > 1$ the subtree of w_1 has changed and we treat w_{m-1} as an additional child of w_0.

$$S_j(T_{w_0}^k) = \begin{cases} S(T_{w_0}^{k-1}) \oplus S_j(T_{w_1}) & \text{for } j = 1, \\ \left(S(T_{w_0}^{k-1}) \oplus S_j(T_{w_1})\right) \oplus S_j(T_{w_{m-1}}) & \text{otherwise.} \end{cases} \tag{4}$$

To compute the set $S(T_{w_0}^k)$ we take the union over all configurations: $S(T_{w_0}^k) = \bigcup_{j=1}^{m-1} S_j(T_{w_0}^k)$. Note that the computations change only for nodes that are part of the cycle. In each configuration we delete only one edge and include a new one. This has no effect on the remaining subtrees of each w_i, i.e. $T_{w_i}^{c_i-1}$, and neither on the computations for those subtrees. For the algorithm we use the bottom-up approach, set $S(T_v^0)$ as above and compute $S(T_v^i)$ for every $1 \leq i \leq c_v$. If there is a cycle $C(v,w)$ with v as the start node and its i-th child $v_i \in C(v,w)$, we use (4) with $v = w_0$, $v_i = w_1$ and $w = w_{m-1}$ to compute $S(T_v^i)$, using (3) resp. (2) recursively. In the case that either v is not the start node of a cycle or v_i is not part of it, we can compute $S(T_v^i) = S(T_v^{i-1}) \oplus S(T_{v_i})$ as described before.

▶ **Theorem 3.2.** *Given a weighted cactus graph G, a positive integer p and two non-negative integers l and u (with $l \leq u$). The p-(l,u)-partition problem can be decided in time $\mathcal{O}(u^2p^2n^2)$ using Algorithm 1.*

Proof sketch. Every set S resp S_j has size $\mathcal{O}(up)$ and therefore every $\oplus$-operation takes $\mathcal{O}(u^2p^2)$ time. For each node v the set $S(T_v)$ (resp. $S_j(T_v)$) contributes to exactly one $\oplus$-operation. Since the number j is in $\mathcal{O}(n)$, we have $\mathcal{O}(n^2)$ $\oplus$-operations and hence an overall runtime of $\mathcal{O}(u^2p^2n^2)$. ◀

Algorithm 1: Algorithm for deciding the p-(l,u)-partition problem for a cactus graph.

```
forall $v \in V$ bottom up do
    $S(T_v^0) = \{(w(v), 1)\}$
    for $1 \le i \le c_v$ do
        if there is a cycle $C(v,w)$ with $v_i \in C(v,w)$ then
            $m = length(C(v,w))$
            for $1 \le j \le m-1$ do
                Compute $S_j(T_v^i)$ using (4) (with $v = w_0$, $v_1 = w_1$, $w = w_{m-1}$) and
                recursively (2) and (3)
            $S(T_v^i) = \bigcup_{j=1}^{m-1} S_j(T_v^i)$
        else
            $S(T_v^i) = S(T_v^{i-1}) \oplus S(T_{v_i})$
if $(x,k) \in S(T_r)$ such that $l \le x \le u$ and $k = p$ then return yes
```

▶ Remark. If we store for each tuple in the partition sets how it was computed – meaning if we did or did not merge and therefore did or did not delete an edge – we can use backtracking to compute the actual p-(l,u)-partition of the graph (assuming there is one).

Polynomial-time Algorithm

Analogously to Ito et al. we can use intervals to reduce the size of computed sets $S(T_v)$. The idea is the following: Instead of storing partitions as tuples (x,k), we store them as $([a,b], k)$. Each interval $[a,b]$ is a maximal d-consecutive subset of weights, where $d = u - l$ is the difference between the upper and lower weight bound. If we adjust the $\oplus$-operation and Alg. 1 to compute interval sets $I(T_v^i)$ (resp. $I_j(T_v^i)$) instead of $S(T_v^i)$ (resp. $S_j(T_v^i)$), we obtain an algorithm with the same number of $\oplus$-operations as before. Since the size of a set $I(T_v^i)$ is $\mathcal{O}(p^2)$ (instead of $\mathcal{O}(up)$), the overall runtime for solving the p-(l,u)-partition problem for cactus graphs is reduced to $\mathcal{O}(p^4n^2)$. Ito et al.'s work is missing a convincing polynomial-time method for the computation of a partition. See the full version of our paper for details about the time reduction and a method for the computation which has polynomial time and space and is applicable for both trees and cactus graphs [1].

Minimum and Maximum Partition Problem

We showed that for a given number p we can partition a cactus graph into p clusters such that the weight of each cluster lies between the lower bound l and upper bound u. Now we consider the problem of finding a (l,u)-partition with the minimal resp. maximal number of clusters. Ito et al. showed that the minimal resp. maximal number of clusters can be found in $\mathcal{O}(n^5)$ for trees. Similar reasoning and our computation method leads us to the following result for cactus graphs.

▶ **Theorem 3.3.** *Given a weighted cactus graph G and l and u as above. The minimum and maximum (l, u)-partition problem can be solved in time $\mathcal{O}(n^6)$.*

References

1 Maike Buchin and Leonie Selbach. A polynomial-time partitioning algorithm for weighted cactus graphs, 2020. `arXiv:2001.00204`.

2 Roberto Cordone and Francesco Maffioli. On the complexity of graph tree partition problems. *Discrete Applied Mathematics*, 134:51–65, 01 2004. `doi:10.1016/S0166-218X(03)00340-8`.

3 M.E. Dyer and A.M. Frieze. On the complexity of partitioning graphs into connected subgraphs. *Discrete Applied Mathematics*, 10(2):139 – 153, 1985. `doi:https://doi.org/10.1016/0166-218X(85)90008-3`.

4 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(3-4):823–841, April 2012. `doi:10.1007/s00453-010-9485-y`.

5 Takehiro Ito, Xiao Zhou, and Takao Nishizeki. Partitioning a graph of bounded tree-width to connected subgraphs of almost uniform size. *Journal of Discrete Algorithms*, 4(1):142 – 154, 2006. `doi:10.1016/j.jda.2005.01.005`.

6 Caterina De Simone, Mario Lucertini, Stefano Pallottino, and Bruno Simeone. Fair dissections of spiders, worms, and caterpillars. *Networks*, 20(3):323–344, 1990. `doi:10.1002/net.3230200305`.