Washington University in St. Louis

Washington University Open Scholarship

McKelvey School of Engineering Theses & Dissertations

McKelvey School of Engineering

5-14-2024

Tree Recovery by Dynamic Programming

Gustavo Alberto Gratacos
Washington University – McKelvey School of Engineering

Follow this and additional works at: https://openscholarship.wustl.edu/eng_etds

Part of the Computer Sciences Commons

Recommended Citation

Gratacos, Gustavo Alberto, "Tree Recovery by Dynamic Programming" (2024). *McKelvey School of Engineering Theses & Dissertations*. 1056.
https://openscholarship.wustl.edu/eng_etds/1056

This Dissertation is brought to you for free and open access by the McKelvey School of Engineering at Washington University Open Scholarship. It has been accepted for inclusion in McKelvey School of Engineering Theses & Dissertations by an authorized administrator of Washington University Open Scholarship. For more information, please contact digital@wumail.wustl.edu.

WASHINGTON UNIVERSITY IN ST. LOUIS

McKelvey School of Engineering
Department of Computer Science and Engineering

Dissertation Examination Committee:
Nathan Jacobs, Chair
Jeremy Buhler
Ayan Chakrabarti
Tao Ju
Ulugbek Kamilov

Tree Recovery by Dynamic Programming

by

Gustavo Gratacós

A dissertation presented to
the McKelvey School of Engineering
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

May 2024
St. Louis, Missouri

© 2024, Gustavo Gratacós

Table of Contents

List of Figures

Acknowledgments

I'm grateful to Tao Ju for guiding me throughout my degree. I am also grateful to Ayan Chakrabarti for mentoring me during the initial years, as well. I would like to thank all my lab members for all of their valuable feedback for my presentations and projects, and my office mates, who have helped me stay grounded throughout my graduate degree. I would like to thank my family, who have never ceased to give me their full support.

Furthermore, I would like to thank Chris Topp at Donald Danforth Plant Science Center for providing the CT volumes of maize root crowns, Rolando Estrada for providing the implementation of [10] and helping us with its use, and Nadav Dym and Shahar Kovalsky for helpful discussions.

This work was supported in part Washington University's Dean's Select PhD Fellowship, and by the NSF under Grants DBI-1759836 and EF-192172.

Gustavo Gratacós

Washington University in St. Louis
May 2024

ABSTRACT OF THE DISSERTATION

Tree Recovery by Dynamic Programming

by

Gustavo Gratacós

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2024

Professor Nathan Jacobs, Chair

Tree-like structures are common, naturally occurring objects that are of interest to many fields of study, such as plant science and biomedicine. Analysis of these structures is typically based on skeletons extracted from captured data, which often contain spurious segments or cycles that need to be removed. We propose a dynamic programming algorithm which seeks to recover directed trees from these noisy skeletons. Our method recovers trees by removing edges and duplicating nodes while adhering to edge-label constraints. Our algorithm proceeds by iteratively merging graph nodes, such that the solution on the original graph can be obtained from those on the contracted graphs. Furthermore, we show that our method can recover trees in many settings, such as from rice root images, retinal fundus images, 3D corn root images, and 3D grapevine images.

Chapter 1

Introduction

Many biological shapes are like *trees* - they consist of tubular branches and generally have no “loops”. Some examples are plants, roots, lung airways, blood vessels, neurons, to name a few. Characterizing the branching pattern of such shapes is important for domains such as biology and medicine.

Analysis of tree-like shapes often makes use of a graph representation, known as *skeletons*, which captures the branching structure as a curve network. A skeleton can be extracted from a variety of inputs, such as 2D or 3D images, point clouds, and meshes. However, skeletons extracted from real-world data often contain spurious cycles. These cycles may arise in two different ways. Firstly, the skeleton may have a *deficiency* of nodes, where multiple nodes are merged as one. Cycles caused by a deficiency of nodes may be seen in figure 1.1 (a), where a 2D projection of a 3D object causes nodes to be merged in “cross overs”. Secondly, the skeleton may have an *excess* of nodes and edges, where nodes and edges not belonging to the underlying object cause cycles. Figure 1.2 shows multiple graphs that suffer from an excess of nodes and edges.

While simple methods such as Minimum Spanning Tree (MST) or k-MST may yield trees from these skeletons, they have many setbacks. Firstly, they only address issues when there is an excess of nodes and edges. Figure 1.4 shows how MST would not handle problems that require vertices to be duplicated, while duplicating nodes would. Figure 1.3 shows practical applications of this on a real-life dataset. Another problem with MST-based methods is that they use edge-based costs, which are insufficient to encode many tree-topological and geometrical features. For example, edge based costs cannot make use of the resulting tree’s node degrees. Being able to consider these features is essential to obtain results of good quality.

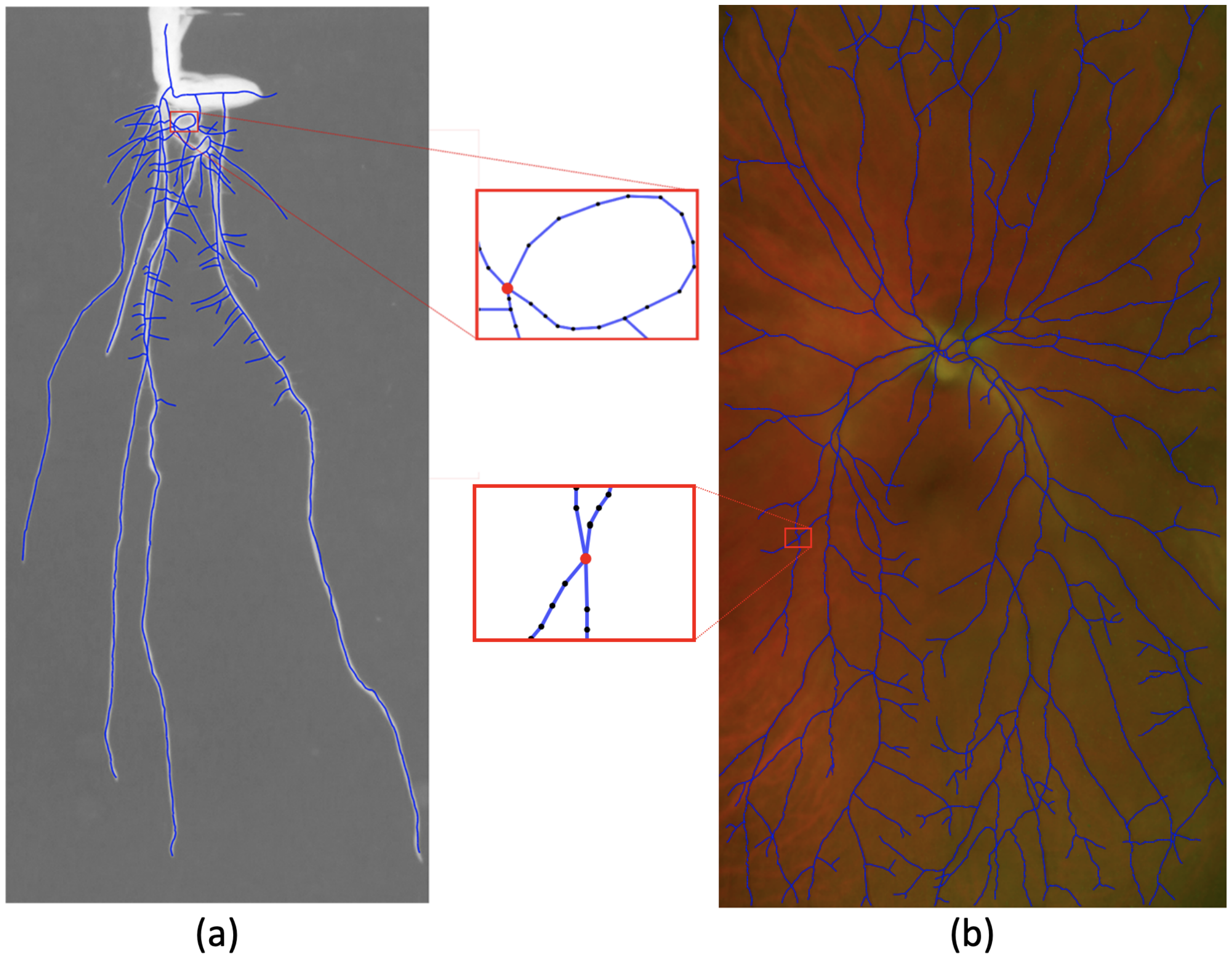

Figure 1.1: (a) A 2D image of a rice root. Branches "cross over" and share a vertex in the resulting skeleton. (b) The same thing happens with 2D retinal fundus images.

The general goal of this work is to remove the cycles from a skeleton graph and recover a tree that best describes the branching pattern of the shape, while making use of the prior knowledge and ample datasets available to us. This work is divided into 2 parts:

Firstly, we consider the NP-hard, graph-based formulation proposed by Estrada et al. [10], which removes the cycles by *partitioning* the nodes. Specifically, a node (e.g., red node in Figure 1.4 (a)) may be duplicated into multiple nodes (e.g., blue nodes in Figure 1.4 (b)), each incident to a subset of edges incident to the original node. Given a cost associated with each possible partition of a node, the problem seeks a set of partitions, one at each node, with the least total costs that result in a tree.

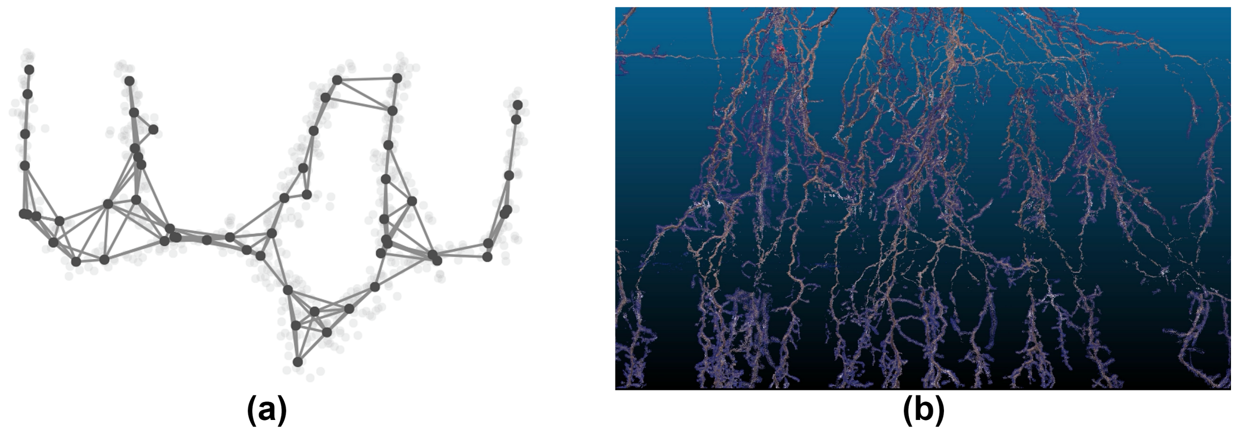

Figure 1.2: (a) A skeleton obtained from an Upright Shooting Offshoot cherry trees, with an excess of edges and nodes that should be removed from the skeleton. (b) A highly noisy point cloud obtained from switch grass.

Secondly, we propose an extension of this problem that addresses the excess of edges and nodes by removing nodes and edges, while also addressing the deficiency of nodes. We also propose to add label constraints. These label constraints are useful to enforce a tree hierarchy on root systems, or label blood vessels as either arteries or veins.

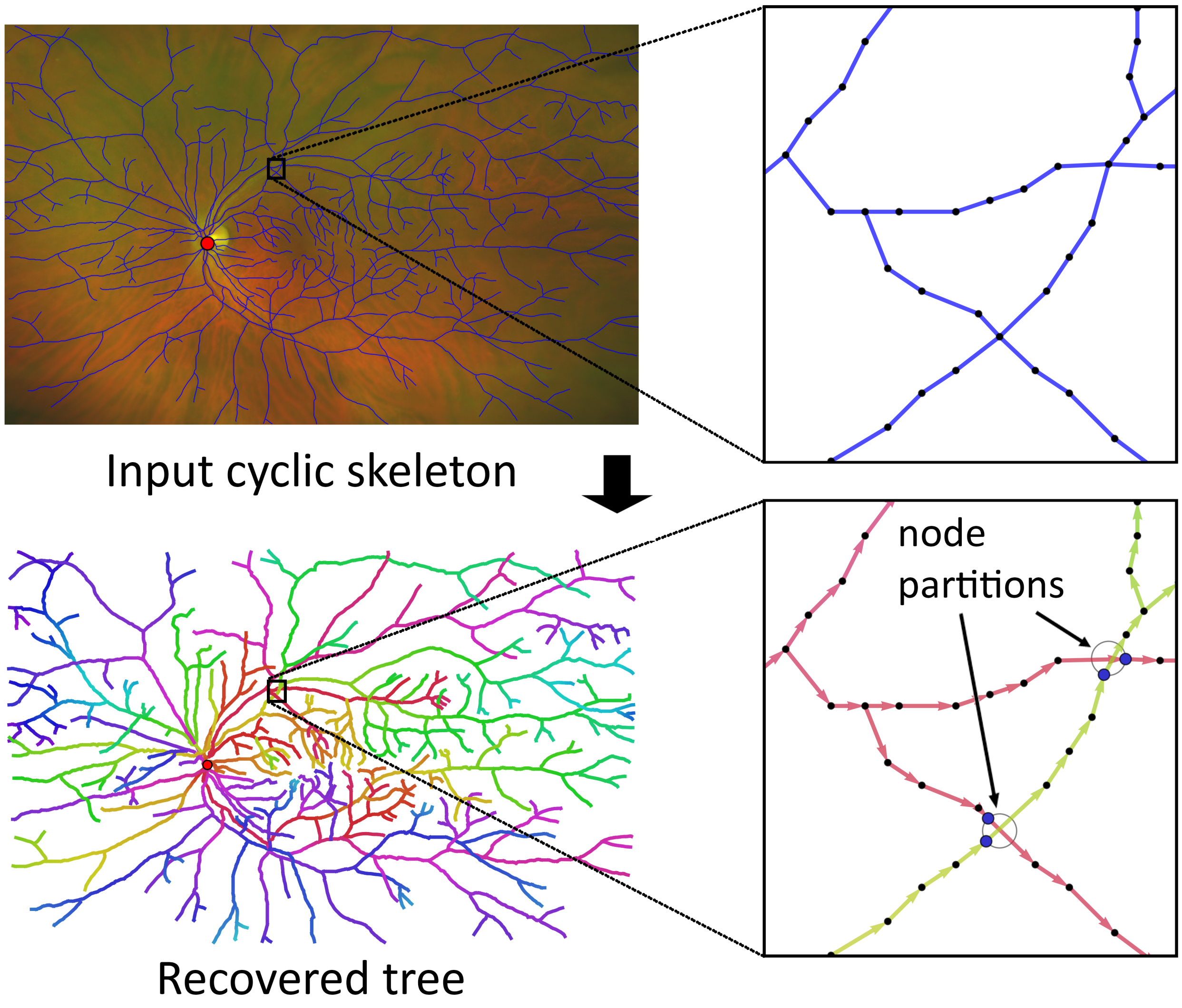

Figure 1.3: Top: A retinal vessel image and its skeleton (from the WIDE dataset in [10]). The close-up shows one of the many skeleton cycles caused by crossing branches. Bottom: A directed tree recovered by our algorithm that retains all skeleton edges and partitions some skeleton vertices into multiple nodes (large blue nodes in the close-up). Adjacent skeleton edges on the recovered tree are assigned similar colors.

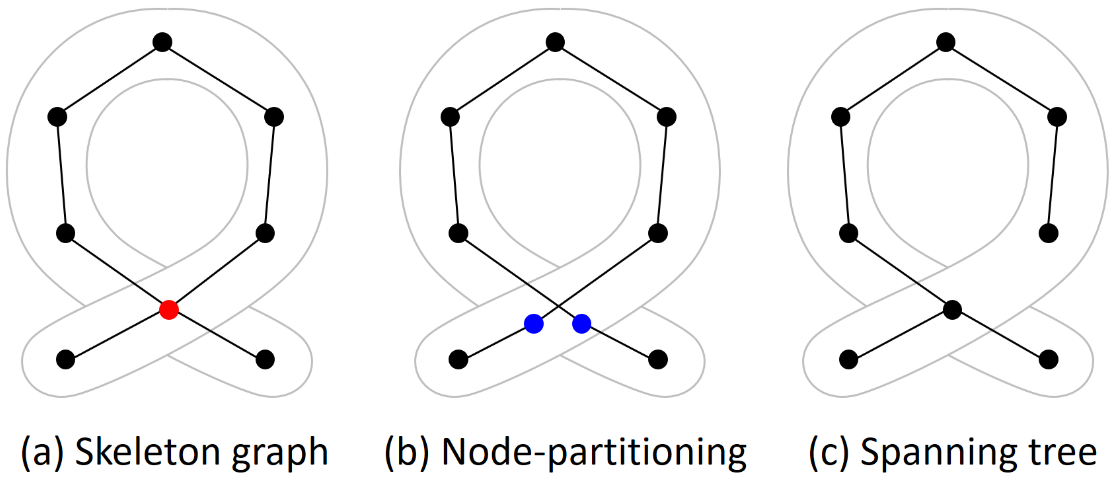

Figure 1.4: (a): skeleton graph with a cycle due to crossing-over branches. (b): tree recovered by partitioning the red node into two blue nodes. (c): a spanning tree. Node partitioning maintains the continuity of the branches, but the spanning tree does not.

Chapter 2

Tree Partitioning

2.1 Related Work

2.1.1 Tree recovery

There is a rich literature on using graph-based methods to analyze tree-like structures, such as neurons [7, 18, 23, 24, 27, 28, 32], blood vessels [6, 7, 10, 23, 28], lung airways [2, 11, 20], and plants (shoots and roots) [10, 17, 28]. These methods usually start with an initial, possibly cyclic graph, which is typically obtained by either skeletonizing [6, 7, 10, 17, 18, 28] a segmented shape or connecting paths between salient points or segments [2, 11, 23, 24, 27]. These methods then go on to extract a tree from the graph. Our review will focus on this tree extraction step.

Many methods extract the minimum spanning tree (MST) from the graph. Given an graph where costs are associated with edges, the MST is the least-cost subgraph spanning all nodes. The MST can be simply and efficiently (in polynomial time) computed, which makes it an ideal choice in practice, whether it is for pruning cycles or connecting components [2, 11, 20, 25, 32].

While an MST spans all nodes, graphs created from noisy inputs may contain redundant nodes that need to be removed. To this end, several formulations of trees that span a subset of the graph nodes have been investigated. Graham et al. [11] and Bauer et al. [2] prune the MST by formulating a *tree knapsack* problem [15]. Given costs and values on the nodes of a directed tree, the problem seeks a maximum-valued subtree whose total costs is within a given limit. Turetken et al. [24] explores *k-MSTs*, which are minimum-cost trees that span k nodes for a pre-defined value of k. Turetken et al. [23] formulates a generalized version of the *minimal arborescence* problem, which seeks the minimum-cost arborescence (directed rooted tree) in a directed graph [8]. You et al. [31] aims to recover tree skeleton trees from point cloud data. This involves finding a tree subset of an initial, noisy skeleton graph. In chapter 3, we propose a method that makes use of You et al's data. Unlike MST, these formulations are all NP-hard on general graphs. Typical optimization techniques include dynamic programming [2, 11], network flow [24], and integer programming [23].

In many cases, cycles are introduced by branches that share common skeleton nodes. Such cycles are best resolved by splitting the nodes and associating them with distinct branches, rather than removing edges as done in MST and the above-mentioned formulations (e.g., compare Figure 1.4 (b,c)). To this end, the authors of [6, 17] apply heuristics locally at each node to determine whether it should be split and, if so, how the incident edges should be grouped. However, these local heuristics lack a global objective. Methods such as dominant set clustering [28] and the matrix forest theorem [7] can solve the splitting problem globally, but they cannot guarantee the connectedness of the output (i.e., it may be a forest). Turetken et al. [23] modifies the minimal arborescence problem to allow a vertex to be used twice and weights to be associated with edges as well as edge-pairs. However, their formulation is tailored to a specific class of partitions and nodes.

Estrada et al. [10] proposed a general and global formulation for tree recovery by node partitioning. Unlike [23], their formulation (see Section 2.2) accommodates arbitrary partitions at any node. The authors show that the problem is NP-hard and describes a heuristic search algorithm. The search is over the space of edge orientations that form a directed acyclic flow graph (or *flow-DAG*), which is a strict subspace of all possible tree-forming edge orientations. In addition, the search algorithm, which is based on best-first exploration, terminates at only locally optimal trees in the subspace. In chapter 1, we propose a new algorithm for solving the tree recovery problem with improved optimality and efficiency.

2.1.2 Dynamic programming on graphs

Dynamic programming (DP) is well-suited for solving graph optimization problems that can be decomposed into sub-problems on subgraphs. Different types of hierarchical decompositions of a graph G have been explored. A *branch-decomposition*, a tree whose leaf nodes represent edges of G, is useful for solving call routing problems [21] and TSP [5]. A *carving-decomposition*, whose leaf nodes represent nodes of G, has been used for finding max-path-coloring [1]. A *tree-decomposition*, where each node represents a group of nodes of G, is useful for vertex partitioning [22], edge colouring [4] and solving Steiner tree problems [16]. A survey of different types of decompositions and their use in DP algorithms can be found in [19].

The complexity of a DP algorithm is typically polynomial on the size of the graph but exponential on some *width* parameter of the graph decomposition. For graphs that admit decompositions with low width, DP can therefore find optimal solutions efficiently, even if the problem is NP-hard. However, determining the smallest decomposition width for a given graph, and finding such decompositions, are both NP-hard problems themselves for general graphs [21], although polynomial-time algorithms exist for planar graphs [3, 12, 13, 21]. Practical heuristics for computing low-width decompositions are surveyed in [14].

Our main contribution is demonstrating that the tree recovery problem is another instance that DP is well-suited for. Our DP algorithm iteratively reduces the graph size by merging pairs of nodes, effectively performing a traversal of a carving-decomposition of the input graph. Building on existing algorithms for computing low-width decompositions [5], we introduce new algorithms to optimize merging sequences for our DP algorithm and to further improve the efficiency of DP at the cost of optimality. We also propose an extension of the algorithm that addresses the removal of nodes and edges, using the datasets from point-clouds provided by You et al.

In this chapter, we present a dynamic-programming (DP) algorithm for finding the optimal tree as formulated in [10]. The algorithm iteratively reduces the size of the problem, such that the optimal solution of the larger problem can be recovered from that of the smaller problem, until the problem becomes trivial to solve. Each reduction involves contracting the graph by merging two nodes and the partitions at those nodes. The algorithm is simple to implement and guaranteed to find the optimal solution. To be able to process even larger inputs, we introduce an approximate variant of the DP algorithm using beam search. This variant offers a parameter (beam width) that balances the opposing needs of optimality and efficiency.

2.2 Problem Formulation

We shall formally state the tree recovery problem introduced by Estrada et al. [10]. The original formulation seeks the most likely directed tree that *projects* to a given undirected cyclic graph, in the sense that multiple tree nodes may project to the same graph node. For the purpose of describing our dynamic programming algorithm, we shall re-state the problem in the other direction, which seeks the most likely way that the nodes in the given graph are *partitioned* to form a tree.

2.2.1 Node partition

Consider an undirected, cyclic graph $G = \{V, E\}$. A *partition* of a node $v \in V$ consists of one or more *instances* of v, each associated with one or more edges incident to v (denoted by E_v), such that each edge in E_v is associated with exactly one instance of v. A partition also assigns a direction to each edge. Example partitions of a degree-3 node are shown in Figure 2.1.

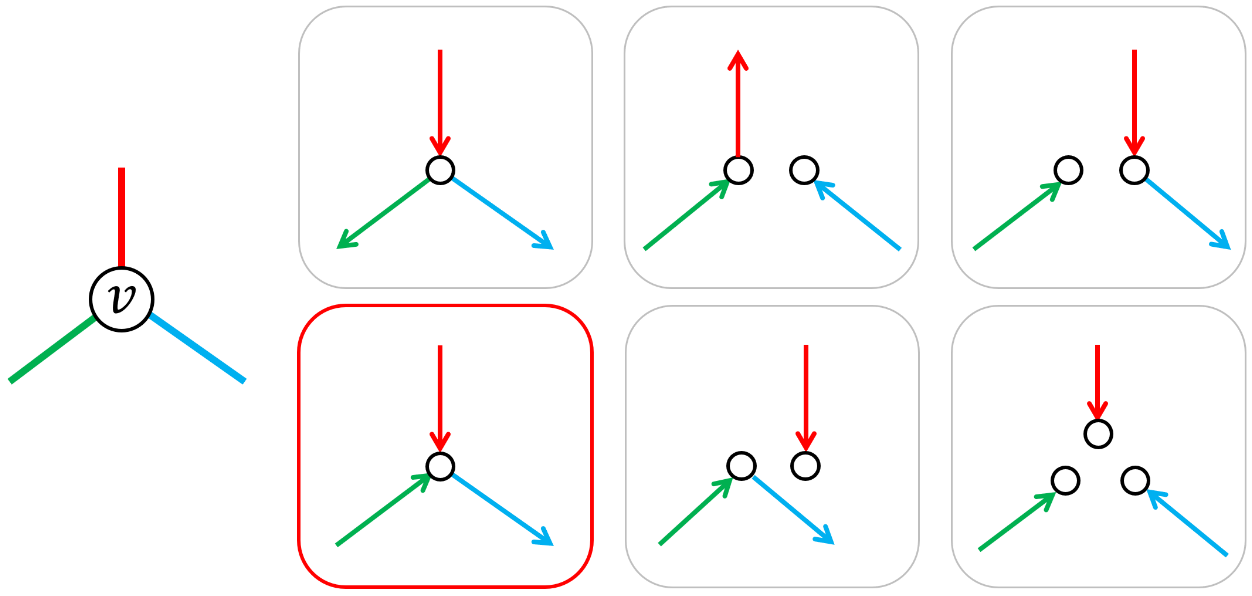

Figure 2.1: A degree-3 node v (left) and several possible partitions (right), all of which are valid except the one outlined in red. Corresponding edges are identified by colors.

Our goal is to recover a directed tree (i.e., an *arborescence*) rooted at some prescribed node $r \in V$. As a result, we are only interested in partitions at a node that might be part of an arborescence. We call such partitions *valid*. A valid partition at v can be characterized as follows:

1. If v is not the root node r, each instance of v must be associated with one incoming edge (directed towards that instance) and zero or more outgoing edges (directed away from that instance).

2. If $v = r$, one instance of v must be associated with only outgoing edges (none if $V = \{r\}$) whereas every other instance is associated with one incoming edge and zero or more outgoing edges.

All partitions shown in Figure 2.1 are valid (assuming the node v is not the root) except for the highlighted one.

2.2.2 Partition set

A *partition set* P, or a *p-set*, consists of one partition at each node $v \in V$, denoted by P_v. A p-set is said to be *consistent* if each edge $(u, v) \in E$ is assigned the same direction by partitions P_u and P_v. A consistent p-set P defines a *partition-graph* (or *p-graph*) of G, denoted by G_P, whose nodes and edges are instances and directed edges in P. In a p-graph, directed edges belonging to partitions of two adjacent nodes that correspond to the same edge in E appear as one edge. Figure 2.2 shows examples of p-graphs defined by 4 consistent p-sets.

A p-set is said to be *valid* if it is consistent and its p-graph G_P is an arborescence rooted at an instance of the root node r. Among the four p-graphs shown in Figure 2.2, the top two are arborescent, and hence their corresponding p-sets are valid. A valid p-set consists of only valid partitions. However, the inverse is not true. For example, G_{P_4} in Figure 2.2 is not an arborescence, and hence the corresponding p-set P_4 is not valid, although it consists of valid partitions at all nodes.

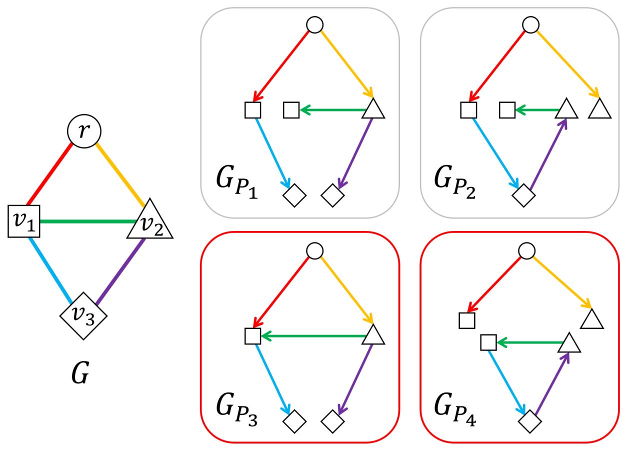

Figure 2.2: A graph G (left) and the p-graphs G_{P_i} of four consistent p-sets P_i ($i = 1, \dots, 4$) (right). The two p-graphs on the top are arborescent but the bottom two (outlined in red) are not. Corresponding edges are identified by colors, and instances of the same node in G are identified by their shape (e.g., $\square$ is an instance of v_1).

2.2.3 Optimal partition set

To assess the goodness of a p-set, we assume that a partition cost function h is given so that $h_v(p)$ is the cost of a partition p at node v. The cost of a p-set P is the total costs over all its partitions,

$$h(P) = \sum_{v \in V} h_v(P_v). \tag{2.1}$$

The *tree recovery problem* seeks the valid p-set of G with the minimum cost. We call such p-set the *optimal p-set* of G under the cost function h, and its corresponding p-graph the *optimal tree*.

2.3 Dynamic Programming

The key idea of our algorithm is to iteratively reduce the problem size until the problem becomes trivially solvable. Each reduction step ensures that the solution of the original problem can be recovered from that of the reduced problem. We will describe the reduction step in Section 2.3.1 and the full algorithm in 2.3.2.

2.3.1 Graph contraction

Given a graph $G = \{V, E\}$ and a partition cost function h, we shall create a smaller graph $G' = \{V', E'\}$ and the corresponding cost function h', such that the optimal p-set of $\{G, h\}$ can be obtained from the optimal p-set of $\{G', h'\}$.

The new graph G' is obtained by merging a pair of nodes of G into one. Let $u, w \in V$ be the nodes being merged and $x \in V'$ the merged node. As illustrated in Figure 2.3 (a), merging removes edges connecting u and w and reconnects those edges incident to either u or w with x. The root node r' of G' is set to be r if $r \notin \{u, w\}$, and x otherwise. Note that merging may result in a multi-graph where multiple edges connect the same pair of nodes, although there is no loop edge that connects a node to itself.

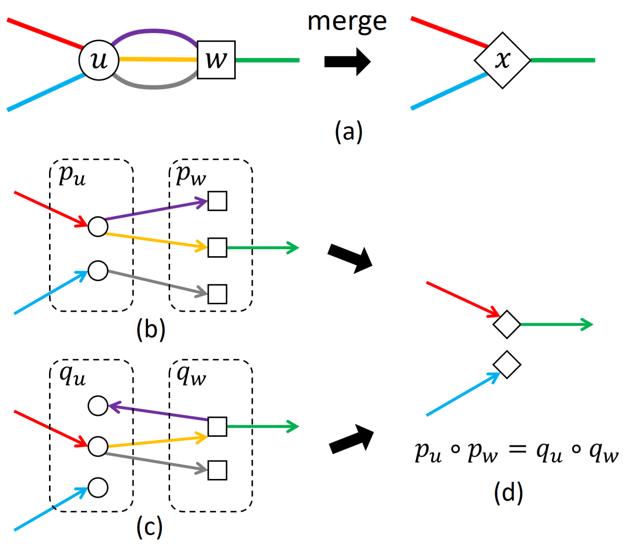

Figure 2.3: (a): Graph contraction by merging u, w into x. (b,c): The p-graphs of two pairs of consistent partitions of u and w, $\{p_u, p_w\}$ and $\{q_u, q_w\}$. (d): The partition of x that both pairs merge into.

The new partition costs h' are identical to h at those nodes that are not affected by merging. That is,

$$h'_v = h_v, \forall v \in V \setminus \{u, w\} \tag{2.2}$$

The partition costs h'_x at the merged node x are defined by the costs of pairs of partitions at u and w. We say a pair of partitions, p_u of u and p_w of w, is *consistent* if they assign the same direction to each edge connecting u and w. Similar to a consistent p-set, a consistent partition pair $\{p_u, p_w\}$ defines a partition graph (or p-graph), which we denote by $p_{u,w}$. Figure 2.3 (b,c) show the p-graphs of two consistent pairs of partitions at nodes u and w in (a).

We can construct a partition of x by *merging* a consistent pair of partitions $\{p_u, p_w\}$ as follows: create one instance of x for each connected component of the p-graph $p_{u,w}$, and associate that instance with all edges of E' incident to x in that component of $p_{u,w}$. We denote this merged partition by $p_u \circ p_w$. As examples, both pairs of partitions in Figure 2.3 (b,c) merge into the same partition of x shown in (d).

The cost of a partition p of x is the minimum cost of any consistent pair of *valid* partitions of u and w that merge into p. That is,

$$h'_x(p) = \min_{\substack{p_u, p_w \text{ are valid} \\ \{p_u, p_w\} \text{ is consistent,} \\ p_u \circ p_w = p}} h_u(p_u) + h_w(p_w) \tag{2.3}$$

Note that not all partitions of x can be merged from some pair of consistent and valid partitions of u and w. We set $h'_x(p) = \infty$ for such partitions p.

We call a partition pair $\{p_u, p_w\}$ that attains the minimal cost on the righthand side of Equation 2.3 a *generator* of the partition p. As shown below, generators are the key for recovering the optimal p-set of $\{G, h\}$ from that of $\{G', h'\}$:

Proposition 1. *Consider the graph G' contracted from G by merging u, w into x, and the partition costs h' as defined by Equations 2.2 and 2.3. If P' is an optimal p-set of $\{G', h'\}$, then*

$$P = P' \setminus \{P'_x\} \cup \{p_u, p_w\} \tag{2.4}$$

is an optimal p-set of $\{G, h\}$, where $\{p_u, p_w\}$ is a generator of the partition P'_x of x.

2.3.2 Algorithm

Our algorithm iteratively contracts the graph and updates the costs, until the graph contains a single node (the root r). The optimal p-set of such a graph can be trivially found, because the only valid partition of r is a single instance with no incident edges. The optimal p-set of the original graph can then be recovered from the generators computed during graph contraction (following Equation 2.4) in a backward traversal of the contracted graphs. The pseudo-code of the entire algorithm and the recovery routine (implemented using recursion) are shown in Algorithms 1 and 2.

The DP algorithm (Algorithm 1) takes an additional input, S, that prescribes the sequence of node pairs to be merged at each step of graph contraction. The choice of S does not affect the optimality of the algorithm, but it may have a significant impact on the time complexity. In the next section, we will analyze the complexity of our DP algorithm as a function of the merging sequence S, and compute optimized merging sequences with low DP complexity.

Algorithm 1: Dynamic programming

Input: undirected graph $G = \{V, E\}$; partition costs h_v at each $v \in V$; sequence S of node index pairs to be merged.

Output: optimal p-set of $\{G, h\}$.

```
/* initialization                                              */
$nodes \leftarrow \emptyset$ ;
forall $v \in V$ do
    $x \leftarrow$ new $node$ ;
    $x.nodes \leftarrow \emptyset$ ;                              /* original nodes */
    $x.cost \leftarrow h_v$ ;                                    /* partition costs */
    $x.gen \leftarrow null$ ;                                       /* generators */
    $nodes.add(x)$;
end
/* iterative contraction                                       */
forall $\{i, j\} \in S$ do
    $u \leftarrow nodes[i]$ ; $w \leftarrow nodes[j]$ ;
    $x \leftarrow$ new $node$ ;
    $x.nodes \leftarrow \{u, w\}$ ;
    $x.cost \leftarrow \infty$ ;
    $x.gen \leftarrow null$ ;
    forall $valid\ partitions\ p_u\ of\ u\ and\ p_w\ of\ w$ do
        if $p_u$ and $p_w$ are consistent then
            $p = p_u \circ p_w$ ;
            $s = u.cost(p_u) + w.cost(p_w)$ ;
            if $x.cost(p) > s$ then
                $x.cost(p) = s$;
                $x.gen(p) = \{p_u, p_w\}$ ;
        end
    end
    $nodes.add(x)$;
end
/* recovering the p-set                                        */
$x \leftarrow nodes.last()$ ;
$p \leftarrow$ only valid partition of $x$ ;
return $recover(x, p)$;
```

Algorithm 2: *recover*(x, p)

Input: node x (either an original or a merged node) and a valid partition p of x.
Output: p-set of original nodes merged into x whose merged partition is p.

```
if $x.nodes = null$ then
    /* $x$ is an original node                                         */
    return $p$;
else
    /* $x$ is a merged node                                            */
    $P_u \leftarrow recover(x.nodes[0], x.gen(p)[0])$;
    $P_w \leftarrow recover(x.nodes[1], x.gen(p)[1])$;
    return $P_u \cup P_w$;
end
```

2.4 Optimizing merging sequence

We start by analyzing the complexity of our DP algorithm in relation to the merging sequence (Section 2.4.1). We will then introduce an alternative representation of the merging sequence as a binary tree, called the *merge-tree* (Section 2.4.2), and describe a heuristic approach for optimizing the merge-tree in terms of the complexity of the DP algorithm (Section 2.4.3).

2.4.1 DP complexity

Each step of DP enumerates all pairs of valid partitions of the two nodes being merged (blue line in Algorithm 1). As proven in Appendix B, the number of valid partitions at a node is an exponential function of its degree d (i.e., number of incident edges),

$$\Pi(d) = \sum_{k=1}^{d} \binom{d}{k} k^{d-k}. \tag{2.5}$$

Merging nodes u and w, whose degrees are d_u and d_w, thus takes $O(\Pi(d_u)\Pi(d_w)(d_u + d_w))$ time, where $(d_u + d_w)$ is for checking the consistency of and merging a pair of partitions of u and w. Let d_S be the maximum degree of a node in G or a merged node in a merging sequence S. Since DP performs $|V| - 1$ merge operations, its complexity using S is $O(|V|\Pi(d_S)^2 d_S)$.

While the complexity is linear to the graph size $|V|$, it is exponential to the maximum node degree d_S. As a result, finding merging sequences with low node degrees is crucial for lowering the complexity of DP.

2.4.2 Merge-trees

To obtain low-degree merging sequences, it is convenient to think of the merging process as performing hierarchical clustering, where each merging operation clusters two groups of nodes in V into a larger group. We can represent the hierarchy as a rooted binary tree, which we call a *merge-tree* (or simply *m-tree*). Each leaf of an m-tree represents a node in V, and each interior node x of the m-tree represents a subset of V, denoted by V_x, that is the union of the two subsets represented by the two children of x. Two examples of m-trees are shown in Figure 2.4 (c,d) for the graph in 2.4 (a).

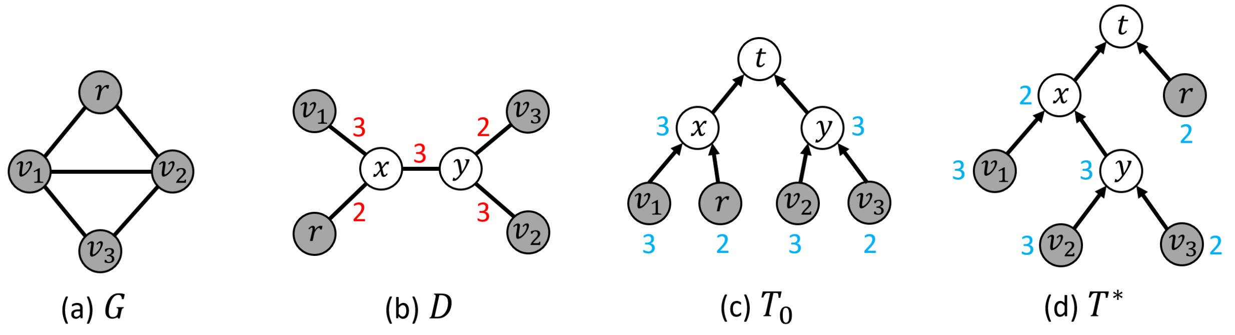

Figure 2.4: Steps in creating an m-tree of a graph G (a): a low-width carving-decomposition D (b), an initial m-tree T_0 obtained by inserting a root node t to D (c), and the optimized m-tree T^* after iterative restructuring (d). Numbers indicate edge capacities (red) and node orders (blue).

Compared to merging sequences, m-trees are a more compact representation. Multiple merging sequences can be derived from an m-tree T by a bottom-up traversal. Furthermore, these sequences all lead to the same DP complexity. This is because, for each node x of T, the degree of its corresponding node in any merging sequence is the same. This degree is the number of edges in E that connect between V_x and $V \setminus V_x$. We call this number the *order* of x in T, denoted by $o_T(x)$ (shown in blue in Figure 2.4 (b,c)). To find merging sequences with low node degrees, we instead look for m-trees with low node orders.

Multiple m-trees may have the same maximum node order, which leads to the same maximum node degree in the derived merging sequences. For optimization purposes, we measure the goodness of an m-tree using a fine-grained complexity of DP. This measure, which we call the *merge-cost* (or *m-cost*) of T, is a summation over all interior nodes of T,

$$F(T) = \sum_{x \in T \setminus V} f(o_T(x.left), o_T(x.right)), \tag{2.6}$$

where $\{x.left, x.right\}$ are the children nodes of x and $f(n, m)$ is a proxy of the time needed for merging two nodes with degrees n and m,

$$f(n, m) = \Pi(n)\Pi(m) * (n + m). \tag{2.7}$$

2.4.3 Computing merge-trees

We do not know the complexity of computing the m-tree with the least m-cost, although we suspect it to be NP-hard. The algorithm presented here does not guarantee to find the optimal m-tree, but it was shown in our experiments to be more effective in lowering the DP complexity than other simpler alternatives.

The algorithm utilizes a slightly different tree structure known as the *carving-decomposition* [21]. A *carving-decomposition* of a graph $G = \{V, E\}$ is an un-rooted tree where all interior nodes have degree 3 and each leaf node corresponds to a node in V. An example carving-decomposition is shown in Figure 2.4 (b) for the graph in 2.4 (a). The *capacity* of an edge e in a carving-decomposition D (shown in red in 2.4 (b)) is the number of edges in E that connect two nodes in V whose corresponding leaf nodes in D are connected by a path containing e. The maximum edge capacity in D is called the *width* of D, and the minimal width of any carving-decomposition of G is called the *carving-width* of G.

An m-tree can be obtained from a carving-decomposition D by inserting a root node on any edge of D. For example, inserting the root node to the edge (x, y) of the carving-decomposition in Figure 2.4 (b) results in the m-tree in 2.4 (c). The order of a node x in such an m-tree T is the same as the capacity of the edge in D that connects x with its parent node in T. As a result, a low-width carving-decomposition yields m-trees with low maximum node orders.

Our algorithm proceeds in three steps, as illustrated in Figure 2.4. We start by computing a carving-decomposition D (2.4 (b)) with low width for the input graph G. We adopt the eigenvector heuristic of [5] for this step, which was shown to be effective for dynamic programming algorithms on graphs. We then insert a root node to an edge of D to create an initial m-tree T_0 (2.4 (c)). The edge where the root is inserted is selected, among all edges of D, to yield the m-tree with the minimal m-cost. Finally, an optimized m-tree T^* (2.4 (d)) is obtained from T_0 via iterative restructuring. As the last two steps – root insertion and tree optimization – are our novel contributions, they will be described next. We give details of the first step in Appendix C.

Root insertion

Given a carving-decomposition D, we obtain an initial m-tree by inserting a degree-2 root node into one of the edges of D. There are $2|V|-3$ edges in a carving-decomposition of G, and hence the same number of possible m-trees obtained by root insertion. We seek among them the m-tree with the least m-cost.

A naive algorithm would construct all $2|V|-3$ m-trees from D and evaluate the m-cost of each using Equation 2.6, which in turn evaluates f (Equation 2.7) at each of the $|V|-1$ interior nodes of the m-tree. This algorithm thus runs in $O(|V|^2)$ time.

We can reduce the time complexity to $O(|V|)$ using the following observation: if we know the m-cost of the m-tree rooted on an edge incident to some node x of the carving-decomposition D, the m-cost of the m-tree rooted on any other edge incident to x can be obtained in $O(1)$ time. Specifically, as illustrated in Figure 2.5, let s_a, s_b, s_c be the capacities of edges incident to x in D, and T_a and T_b the m-trees constructed by inserting the root node t to the edge with capacities s_a and s_b, respectively. Note that the node orders of T_a and T_b differ only at x, changing from s_a in T_a to be s_b in T_b. Accordingly, the m-costs of the two m-trees satisfy:

$$F(T_b) = F(T_a) - f(s_c, s_b) - f(s_a, s_a) + f(s_c, s_a) + f(s_b, s_b) \tag{2.8}$$

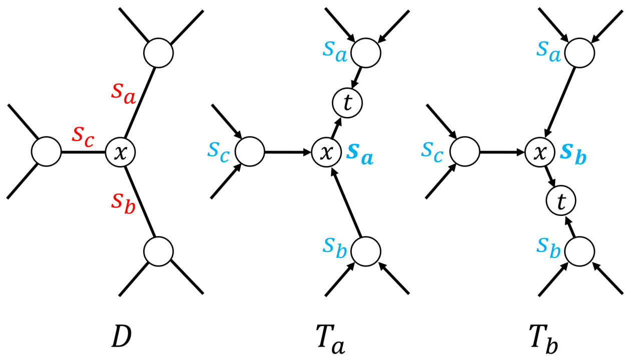

Figure 2.5: A portion of a carving-decomposition D and m-trees T_a and T_b obtained from D by inserting a root node t to two different edges, both incident to node x. Edge capacities are shown in red and node orders in blue.

To compute the m-costs of all $2|V|-3$ m-trees, we first compute the m-cost of one m-tree rooted at an arbitrary edge of D using Equation 2.6. Then we propagate the m-costs to m-trees rooted at adjacent edges using Equation 2.8. Both steps take $O(|V|)$ time.

Tree optimization

The m-tree T_0 constructed in the previous step minimizes the m-cost among those m-trees obtainable by root insertion. We next attempt to further reduce the m-cost of T_0 by modifying its structure. While there are many ways to do so, we found the following iterative scheme to be simple and effective.

A single step in our iterative algorithm, which we call a *pivot*, is illustrated in Figure 2.6. Given an m-tree T, pivoting considers a pair of nodes, $\{x, y\}$, such that x is the sibling of y's parent. We call $\{x, y\}$ an *aunt-niece pair*. Let the sibling of y be z. Pivoting creates a new m-tree, T', by swapping the locations of the subtrees rooted at x and y, thus making x a sibling of z and $\{y, x\}$ the new aunt-niece pair. Whereas a merging sequence derived from T merges z first with y and then with x, a sequence derived from T' merges z first with x and then with y.

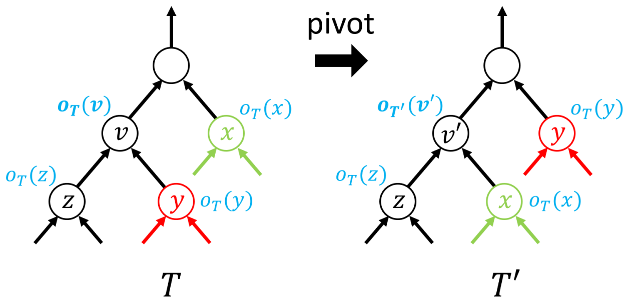

Figure 2.6: Pivoting an aunt-niece pair $\{x, y\}$. Node orders are shown in blue.

Pivoting maintains the orders of all nodes in T except the parent node of y and z, denoted by v, which is replaced by the parent node of x and z in T', denoted by v'. The m-cost of T' can be obtained from the m-cost and node orders of T, as well as the order of v' in T', $o_{T'}(v')$, as:

$$F(T') = F(T) - f(o_T(y), o_T(z)) - f(o_T(v), o_T(x)) + f(o_T(x), o_T(z)) + f(o_{T'}(v'), o_T(y)) \tag{2.9}$$

We show in Appendix D that the space of all m-trees is connected by pivoting. That is, for any two m-trees of the same graph G, there exists a sequence of pivots that turn one m-tree into another. We explore this space in a greedy manner to find an m-tree whose m-cost is locally minimal. Starting from the initial m-tree T_0, we create a sequence of m-trees $T_1, \ldots, T_k$, such that T_{i+1} (for $i = 0, \ldots, k-1$) is obtained from T_i by pivoting the aunt-niece pair whose pivot leads to the largest positive reduction in m-cost among all aunt-niece pairs of T_i. The last m-tree in the sequence, T_k, satisfies that no pivots would reduce its m-cost, and it is returned as the (locally) optimal m-tree, T^*.

To implement the algorithm efficiently, we first compute the potential m-cost reduction resulted from pivoting each aunt-niece pair of the initial m-tree T_0. This step takes $O(|V|^3)$ time, as there are $O(|V|)$ aunt-niece pairs, and evaluating Equation 2.9 requires computing the order of the new node v', which has the worse-case complexity of $O(|V|^2)$. After performing the pivot in each iteration, we only update the potential m-cost reduction for aunt-niece pairs affected by the pivot. As there are a constant number of affected aunt-niece pairs, each update takes $O(|V|^2)$ time. The overall complexity for iterative optimization is thus $O(|V|^3 + k|V|^2)$, where k is the number of iterations.

2.5 Beam search

The algorithm in the previous section can produce merging sequences that result in efficient DP on many real-world graphs (see Section 2.6). However, the inherent exponential complexity of DP makes it unsuited for larger graphs, such as those with high carving-widths.

To be able to process complex graphs efficiently, we propose a variant of our DP algorithm based on *beam search*. Specifically, instead of enumerating *all* pairs of valid partitions of nodes u, w in each step (the blue line in Algorithm 1), we only pair up the (up to) K least-cost valid partitions at both u and w. Only valid partitions with finite costs are considered. This reduces the complexity of our DP algorithm to $O(|V|K^2 d_S)$, where d_S is the maximum node degree in the merging sequence S. The parameter K, known as the *beam width*, balances the opposing needs for efficiency and optimality. Smaller K leads to lower time complexity, whereas larger K considers more partitions and hence the solution is likely to be closer to the optimum.

There is one issue with the beam search described above. When K is too small, it is possible that none of the pairs of the K least-cost valid partitions of the nodes u, w is consistent and merging into a valid partition of the merged node x. Hence all valid partitions of x will have an infinite cost, and the algorithm cannot proceed because x can no longer be merged with any other node.

We modify the beam search so that every merged node has at least one valid partition with finite cost. To do so, we first pre-compute a valid p-set of $G = \{V, E\}$ using the greedy strategy introduced in [10]. Specifically, assuming that edges E are equipped with length, we compute the shortest distance from the root r to each node in V, and orient each edge in the direction of increasing distance. Then we compute the least-cost valid partition at each node with those edge directions. As shown in [10], such partitions form a valid p-set of G. We mark these partitions as *constrained*. During dynamic programming, and as we merge u, w into x, a partition of x is also labeled as constrained if it is merged from two constrained partitions of u and w. Furthermore, a constrained partition is always considered in the pair-wise enumeration even if it is not among the K least-cost valid partitions. This modification ensures that each merged node has at least one valid partition with finite cost (i.e., the constrained partition) and the algorithm can always proceed.

2.6 Results

We evaluate our algorithms on synthetic graphs as well as skeleton graphs of 2D and 3D images. Our algorithm is implemented in Python, and all experiments were performed on a PC with Intel i9-10900 CPU and 64GB of RAM.

2.6.1 Synthetic graphs

To obtain inputs with varying complexity, we created a suite of synthetic graphs using Voronoi diagrams (VD). Each graph is constructed by randomly sampling points (*Voronoi sites*) within a square and computing their VD restricted within the square. To make non-planar graphs, we randomly select pairs of non-adjacent VD vertices and connect them by edges (called *bridges*). An arbitrary VD vertex is picked as the root. Let n_s and n_b be the number of Voronoi sites and bridges, the resulting graph thus has $n_s + n_b$ cycles. See Figure 2.7 (a) for an example with $n_s = 20$ and $n_b = 5$.

Running time of DP

We evaluate the efficiency of our full DP algorithm (Section 2.3) on the synthetic graphs, with a focus on its dependency on the merging sequences. In these experiments, we adopted a random partition cost function h, as the choice of h does not affect the DP complexity.

We consider the following two baseline strategies for creating merging sequences:

- *Random*: Randomly selecting two edge-adjacent nodes to merge at each DP iteration.
- *Greedy*: Selecting the pair of edge-adjacent nodes whose merged node has the least degree at each DP iteration (ties are broken randomly).

We also consider three variations of our own algorithm for computing the m-trees (Section 2.4.3):

- *Ours-Carving*: Only performing the first step of our algorithm, namely computing a carving-decomposition using [5]. The carving-decomposition is then converted to an m-tree by inserting the root node to the first edge created by [5] (see details in Appendix C).
- *Ours-Rooting*: Performing the first two steps, including applying our root-insertion method (Section 2.4.3) to convert the carving-decomposition to an m-tree.
- *Ours*: Performing all three steps, including the final step of pivoting (Section 2.4.3).

The m-costs (Equation 2.6) of the m-trees (or merging sequences) computed by these strategies as the complexity of the graphs (n_s or n_b) increases are plotted in Figure 2.7 (b,c). For each distinct value of n_s or n_b, computations are performed on 100 randomly generated graphs. Observe that the baseline strategies, *Random* and *Greedy*, produce significantly higher m-costs and faster m-cost growth than variants of our algorithm. As to our variants, the m-costs reduce monotonically as more steps of our algorithm are used. It is also interesting to see

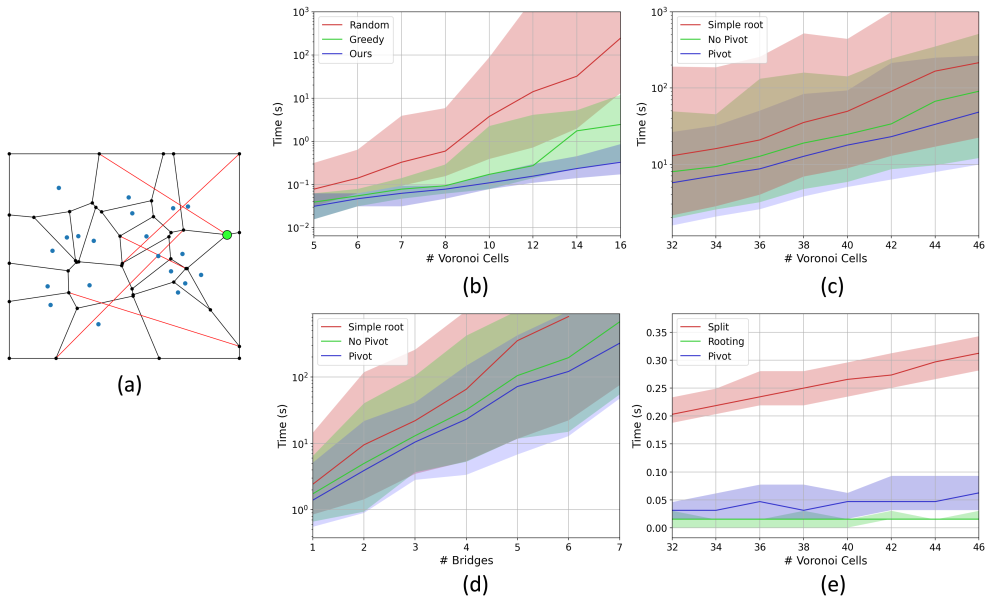

Figure 2.7: (a): A synthetic graph with Voronoi sites (blue dots), bridges (green edges), and root (red dot). (b,c): The median (colored curves) and standard deviation (shaded regions) of m-cost (Equation 2.6), in log-scale, of the merging sequences created by various strategies as a function of n_s (number of Voronoi sites) or n_b (number of bridges). (d): The log-scale running times of DP using three different merging sequences, as well as the time for computing the merging sequence *Ours*, as a function of n_s. (e): The timing of the different stages of computing a merge sequence. *Split* denotes the step of computing the carving decomposition. *Rooting* denotes the step of inserting a root to obtain a merge tree. *Pivot* denotes the step of iteratively improving the merge tree afterwards.

that adding bridges in the graph, and thus making the graph more non-planar, triggers a more significant increase in the m-costs than adding Voronoi sites for all merging sequence strategies.

We plot in Figure 2.7 (d) the wall-clock running times of DP using the merging sequences computed by the three variants of our algorithm as n_s increases. Our complete algorithm (*Ours*) results in faster DP than the other variants, achieving an almost 10-fold speedup over *Ours-Carving* at $n_s = 46$. The same plot also shows the time taken to compute the merging sequence of *Ours*, which is negligible compared with the DP running time.

Optimality and running time of beam search

We next evaluate the optimality and efficiency of the beam search strategy (Section 2.5) on the synthetic graphs. We measure the *error* of a p-set P found by beam search as the ratio $h(P)/h(P^*)-1$, where P^* is the optimal p-set found by the full DP algorithm. An error of 0 indicates an optimal solution.

We consider a number of factors, including the beam width K, the choice of the merging sequence, and the partition cost function h. To examine the latter, we define h to simulate real-world costs with ambiguity. We first compute a valid p-set P using the greedy strategy described in Section 2.5. The cost $h_v(p)$ for a partition p at each node v is then sampled from a normal distribution with standard deviation σ (a tunable parameter) centered at 0 if $p \in P$ and at 1 otherwise. A larger σ leads to more ambiguity in separating partitions in P from those that are not.

We plot the relationship between the errors of beam search and K, under different ambiguity level of h, in Figure 2.8 (a,b). The ambiguity is created either by increasing the standard deviation σ at all nodes, shown in (a), or by increasing the percentage of nodes with a high $\sigma = 0.8$ while keeping the remaining nodes at a low $\sigma = 0.4$, shown in (b). Each data point on the plot represents the median error on 300 random graphs with $n_s = 20$ and $n_b = 3$. Observe that the optimality of beam search quickly improves as K increases, while higher ambiguity in h leads to higher errors.

Besides beam width and partition cost, the optimality of beam search is also affected by the merging sequence. Since the number of partitions at a node is a function of its degree, merging sequences with lower node degrees lead to fewer partitions being excluded from beam search, which increases the odds of finding a low-cost solution. As shown in Figure 2.8 (a,b), beam search using merging sequences obtained by the variant *Ours-Rooting* (dotted curves), which does not perform pivoting, produces larger errors than *Ours*, particularly at lower beam widths where a greater number of partitions are excluded during the search.

Finally, Figure 2.8 (c) plots the running time of beam search as K increases. We set $\sigma = 0.6$ and use the merging sequence computed by *Ours*. Due to the low beam widths, we did not notice any significant difference in the running time for different h or merging sequences. The running time exhibits a log-like growth, which is consistent with our analysis of the

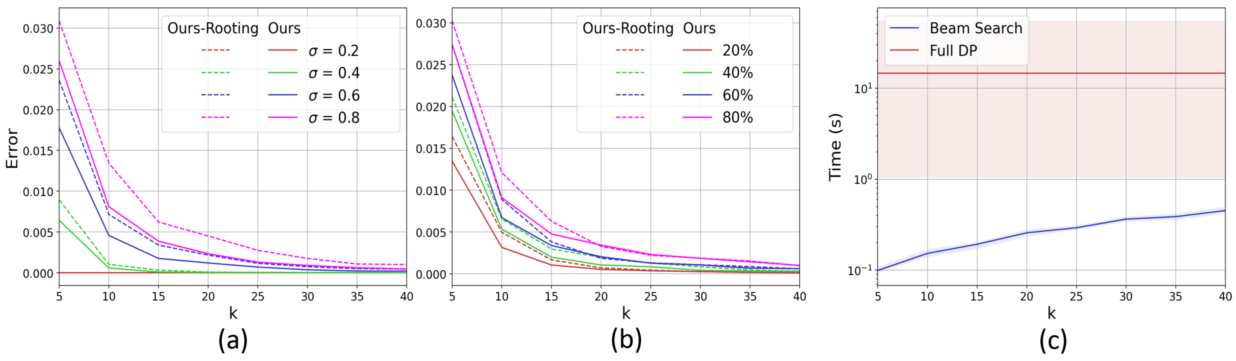

Figure 2.8: (a,b): Error of beam search at increasing beam width K and increasing noise in the partition cost (identified by color). The latter is achieved by either increasing the standard deviation σ at all nodes (a) or the percentage of nodes with a high σ (b). Results are shown using merging sequences created by two strategies, *Ours-Rooting* (dotted curves) and *Ours* (solid curves). (c): Running time of beam search at increasing K compared with that of full DP (which does not depend on K).

complexity of beam search (i.e., polynomial on K), and is significantly faster than the full DP algorithm.

2.6.2 Skeleton graphs

We considered skeleton graphs of 2D and 3D images for two types of biological structures, blood vessels (2D) and plant roots (2D and 3D). Details of the graphs are provided below. We adopt the tree-growth model proposed by Estrada et al. [10] to define the partition cost function h for these graphs (see Appendix D).

We tested both the full DP algorithm, denoted as *DP*, and beam search with beam width $K = 40$, denoted by *DP*-40. As demonstrated on the synthetic graphs (Figure 2.8), this beam width hits a reasonable balance between efficiency and optimality. We also compared with the best-first search heuristic proposed by Estrada et al. [10], denoted as *Estrada-X*, where X is the number of trees explored by their heuristic divided by the number of cycles in the graph. Similar to [10], we tested $X = 50$ and 100. We used the Matlab implementation kindly provided by the authors.

2D skeleton graphs

We used two sets of 2D skeleton graphs provided by [10], one computed from 18 rice-root images (RICE) and another from 15 wide-field retinal vessel images (WIDE). The number of cycles of each graph of RICE and WIDE (as a proxy of its complexity) are reported in Figure 2.9 top. Note that the WIDE graphs are more complex than those of RICE. An example from WIDE is shown in Figure 1.3.

We report the running times and solution errors of various algorithms in Figure 2.9 middle and bottom, respectively. Observe that *DP* finished within seconds on most graphs, and within 15 minutes on all graphs. Compared with *Estrada-50* and *Estrada-100*, which routinely produce sub-optimal solutions, *DP* is not only optimal but also more efficient on all but the few most complex graphs. On the other hand, beam search (*DP*-40) strikes a nice balance between speed and optimality: it is significantly faster than either *DP* or *Estrada-50/100*, and the solution errors are significantly lower than *Estrada-50/100*. In particular, *DP*-40 produces optimal solutions for all but one graph (#14 in RICE).

We visually compared the results of different algorithms on one example of RICE in Figure 2.11. This example contains a branch that loops back onto itself, as shown in the closed-up views on the right. The heuristic of [10] cannot recover such looping branches, since the flow direction along the branch does not form a DAG. Our algorithms (*DP* and *DP*-40) have no such restrictions and recover the branch correctly.

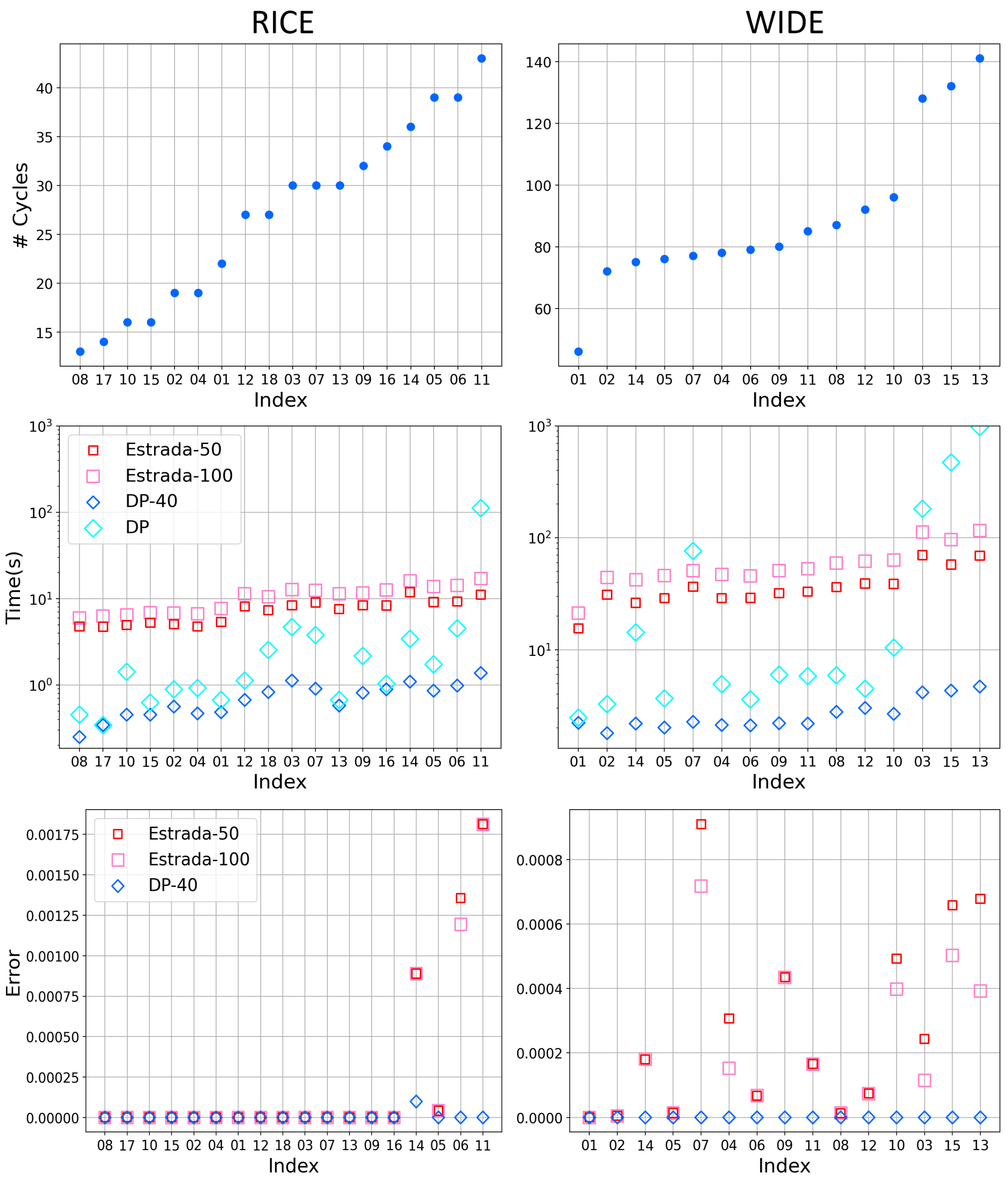

Figure 2.9: Number of graph cycles (top), running times of various algorithms (middle) and their solution errors (bottom) on RICE (left) and WIDE (right) datasets. Horizontal axes are labelled by the original graph indices in each dataset.

3D skeleton graphs

Our 3D dataset, CORN, consists of skeleton graphs of 10 CT volumes of excavated maize (corn) root crowns. The samples were imaged at the Donald Danforth Plant Science Center in St. Louis, MO. The resolution of each volume varies but is approximately 500^3. After thresholding each grayscale volume to obtain a binary volume, we use the Voxel Core method [29] to compute an approximate medial axis, from which a curve skeleton is extracted using Erosion Thickness [30]. An example volume and its skeleton are shown in Figure 2.10 (a).

The CORN graphs are much more complex than those in RICE or WIDE in terms of the number of cycles, as reported in Figure 2.10 (b, top). Furthermore, unlike the planar graphs in RICE and WIDE, the CORN graphs are generally non-planar. The full DP algorithm failed to terminate on any CORN graph after our cut-off time (3000 seconds). In contrast, beam search DP-40 only takes tens of seconds on most graphs, as shown in 2.10 (b, middle). *Estrada-100* was not able to process these large graphs due to memory overflow (using the authors' implementation). Hence we tested *Estrada-10* and *Estrada-50* instead (the latter also encountered memory overflow for the three largest samples). Observe in 2.10 (b, middle) that both heuristics were about 10 times slower than DP-40. To evaluate optimality of these methods, and since we do not have the optimal solution, we ran beam search at a very large beam width (1000), denoted by DP-1000, and measured the solution errors of each recovered tree relative to DP-1000. As shown in 2.10 (b, bottom), DP-40 results in significantly lower error than both *Estrada-10* and *Estrada-50*. These results show that our beam search algorithm has a clear advantage over [10] in both optimality and efficiency on large graphs.

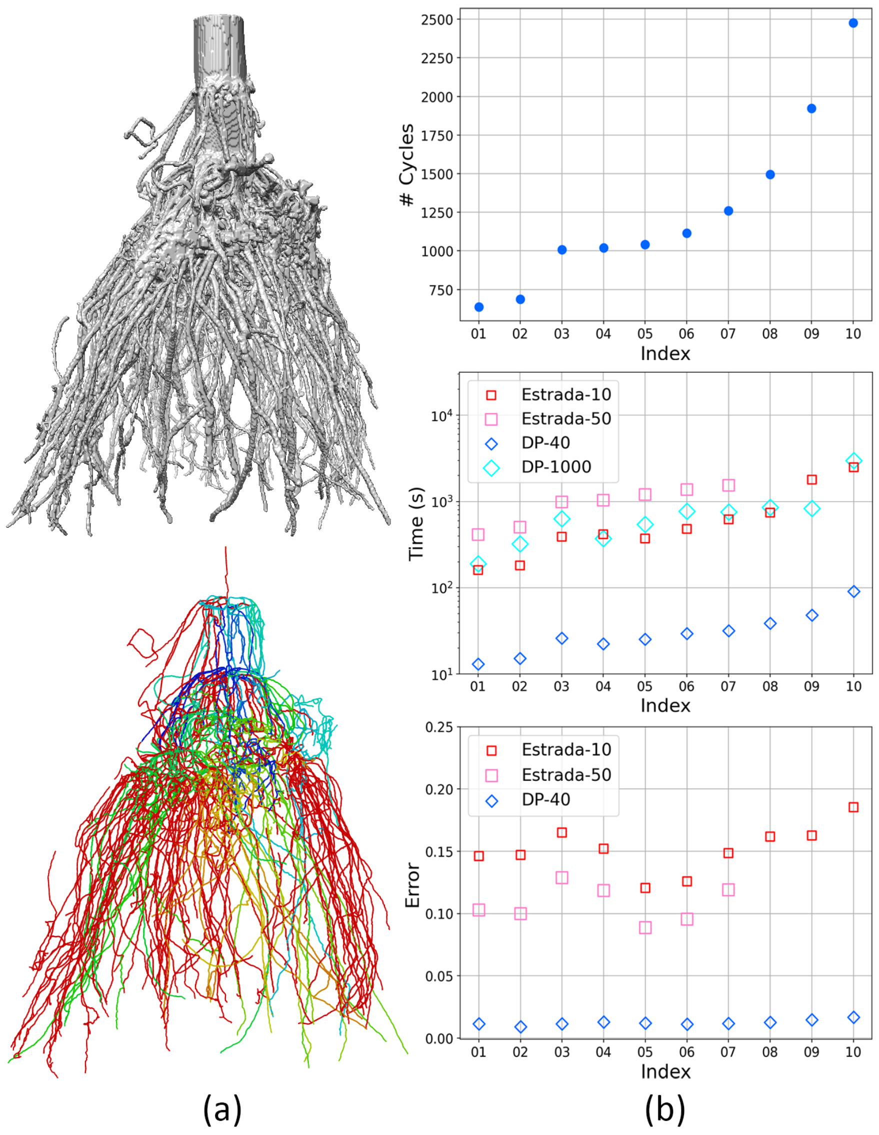

Figure 2.10: (a): A CT volume and its 3D skeleton from our CORN dataset. Skeleton edges are colored by proximity on the recovered tree using our *DP*-40 method. (b): Number of cycles (top), running times of various algorithms in log-scale (middle), and their solution errors (relative to *DP*-1000) (bottom) for all CORN examples. Horizontal axes are graph indices.

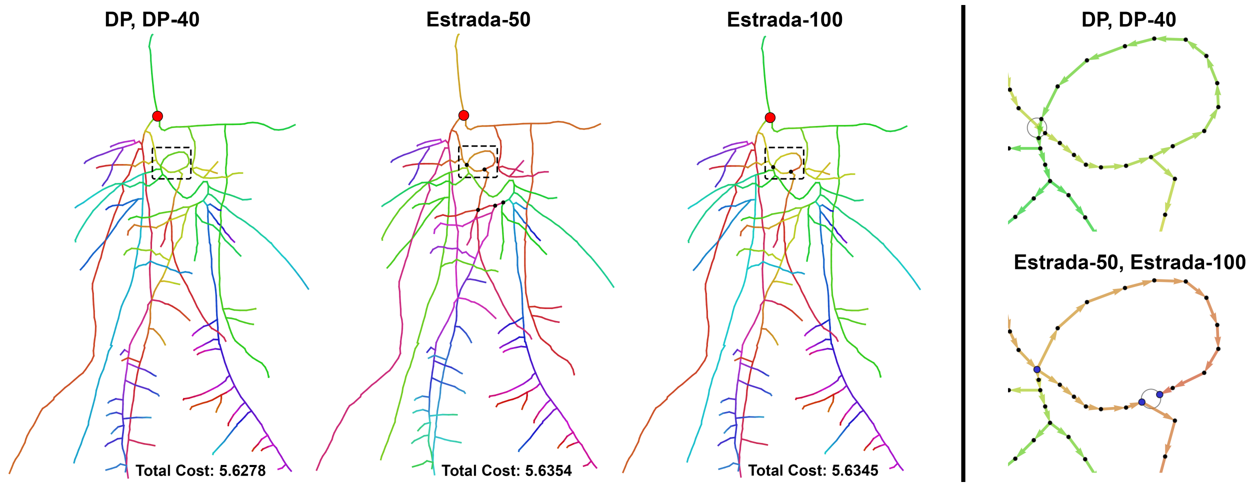

Figure 2.11: Left: Trees recovered by various algorithms on one RICE graph and their costs (both *DP* and *DP*-40 recover the same, optimal tree). Similar colors are assigned to adjacent edges on the tree. The red dot indicates the root, and black dots are nodes whose partitions differ from the optimal tree. Right: Close-up views of a branch that loops back onto itself. Arrows indicate edge directions in the recovered rooted tree, and dots drawn on the same circle are instances of the same node.

2.7 Discussion

In this chapter, we introduced a dynamic programming algorithm for optimally solving an NP-hard tree-recovery problem. The algorithm iteratively reduces the graph size by merging pairs of nodes. By optimizing the merging sequence, our algorithm can efficiently process many medium-size skeleton graphs, where optimal solutions were not known previously. We also explored variants of our algorithm using beam search for processing large inputs, and they were found to offer a better trade off between efficiency and optimality than an existing heuristic [10].

In the following chapters, we propose to extend our dynamic programming algorithm to solve more general tree-recovery problems. In particular, adding the ability to resolved merged edges and removing edges, and adding the ability to label edges.

Chapter 3

Edge Omission and Labeling

In chapter 2, we discussed the problem of recovering trees by duplicating nodes. While this formulation is suitable for addressing the WIDE and Rice datasets provided by [10], it may be insufficient for a wider range of applications. In particular, data such as the 3D switch grass images shown in figure 1.2 (b) and the Upright Shooting Offshoot cherry trees shown in figures 1.2 (a) and 3.1 will generate a lot of noise, yielding skeletons with many unwanted edges, as seen in figure 3.2 (b). Furthermore, colliding branches in 3D plant root images may yield additional, unwanted edges in the skeleton.

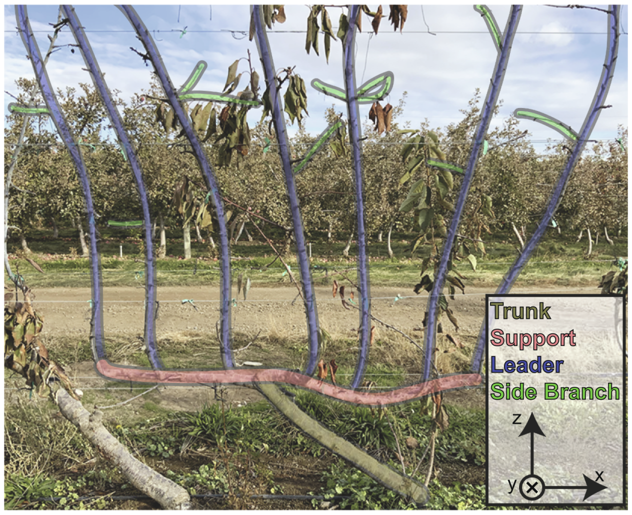

Figure 3.1: An example Upright Shooting Offshoot (UFO) cherry tree, color coded by its different levels of hierarchy.

Additionally, many problems require us to *label* edges based on biological characteristics such as:

- Labeling edges as arteries or veins based on oxygen flow.
- Labeling edges in accordance to a plant's structural hierarchy, such as in 3.1

Labeling edges requires the addition of new consistency constraints and cost functions that support edge-label assignments, none of which are covered in chapter 2.

For this chapter, we will generalize the problem presented in chapter 2 to fit a wider range of tree-recovery problems, and propose a DP algorithm to address them. In particular, we propose the following:

1. The ability to omit edges and nodes.
2. The addition of user-defined edge *labels*.

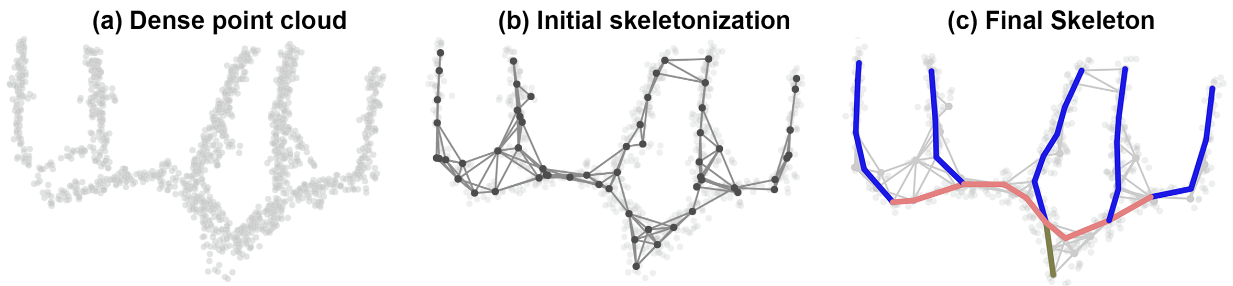

Figure 3.2: (a) The point cloud, input to the proposed method to [31]. (b) Their resulting, noisy initial skeletonization. (c) The desired result with the trunk illustrated as brown, the support branches in pink, and leader branches in blue (right). These images were obtained from [31].

3.1 Related Work

As mentioned in section 2.1, there are many works that utilize MST to retrieve a tree [2, 11, 20, 25, 32], and are not able duplicate nodes to resolve cycles. This also applies to methods that use tree-knapsack [2, 11], k-MST [24], minimum-cost arborescence [23], or other techniques [31] to eliminate edges and nodes that don't belong to the underlying object. However, these methods are able to eliminate nodes and edges, while our proposed method in chapter 2 cannot. In this chapter, our proposed work will address the elimination of nodes and edges, while simultaneously adding label-constraints.

Since this is an extension of the work done in chapter 2, methods that don't address all the problems addressed in that chapter also don't address the problems addressed in this one.

This includes methods that use node duplication, but lack a global objective, don't guarantee connectedness, or are only tailored towards a specific class of partitions [6, 7, 17, 23, 28]. Moreover, Estrada et al. [10] and our formulation in chapter 2 do address these problems, but they cannot remove nodes and edges and cannot resolve merged edges. Furthermore, these methods do not take edge labels into consideration. In this chapter, we will address the problem shown in chapter 2, originally proposed by Estrada et al., while also addressing these new changes.

3.2 Problem Formulation

We will adopt an extended version of our previous formulation from chapter 2, introduced by Estrada et al. [10]. The extension consists of the following:

1. A valid partition may have no instances of a given edge e. If no instances are used, we say that the *capacity* of the edge is zero $\mu(e) = 0$, otherwise it is 1 $\mu(e) = 1$. Furthermore, if $\mu(e) = 0 \ \forall e \in E_v$, then v is removed from the resulting partition graph.
2. The edges will each have an integer-valued label $l(e)$. A valid partition has to abide by a set of user-defined rules based on the edge labels. These labels are meant to represent a tree's hierarchy.

Consider an undirected, cyclic graph $G = \{V, E\}$. A *partition* of a node $v \in V$ consists of one or more node instances of v, each associated with zero or one *edge instances* of E_v, such that each edge instance is associated with exactly one node instance of v. A partition also assigns a direction to each edge instance. An edge is considered to be *omitted* if there are no edge instances associated with it. Example partitions of a degree-3 node are shown in Figure 3.3.

Our goal is to recover a directed tree (i.e., an *arborescence*) rooted at some prescribed node $r \in V$, that may not necessarily span all of the nodes and edges in G. A valid partition at v can be characterized as follows:

1. If v is not the root node r and has 1 or more instances, each instance of v must be associated with one incoming edge (directed towards that instance of v) and zero or more outgoing edges (directed away from that node instance).

2. If $v = r$, one instance of v must be associated with only outgoing edges whereas every other node instance is associated with at most one incoming edge and zero or more outgoing edges.

All partitions shown in Figure 3.3 are valid (assuming the node v is not the root) except for the highlighted ones.

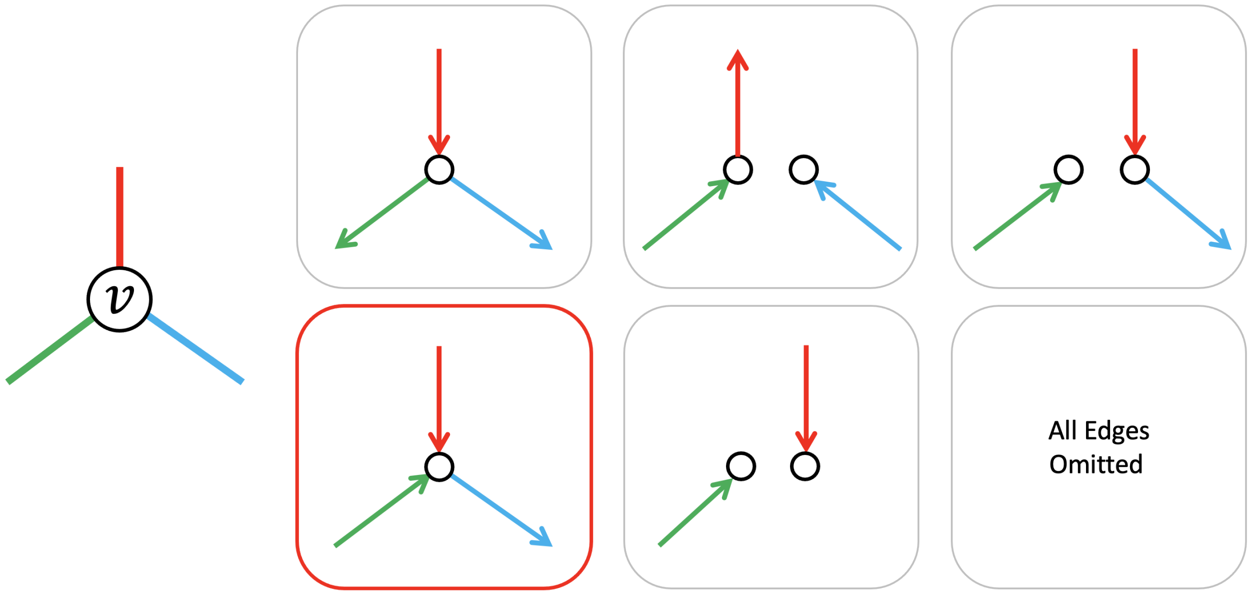

Figure 3.3: A degree-3 node v (left) and several possible partitions. Notice how the blue edge can now be omitted (bottom middle). All partitions here are valid except the ones outlined in red. The last one illustrates the fact that we can omit all edges to exclude a node.

A p-set P_v is said to be *consistent* if each edge (u, v) is in partitions P_u and P_v, and either assigned the same direction by the partitions, or omitted in both partitions. Furthermore, each edge must be assigned the same label in P_u and P_v. A consistent p-set P defines a *partition-graph* (or *p-graph*) of G, denoted by G_P, whose nodes and edges are node instances and directed edges in P. In a p-graph, directed edges belonging to partitions of two adjacent nodes that correspond to the same edge appear as one edge. Furthermore, nodes whose adjacent edges are all omitted in P are not present in G_P. Figure 3.4 shows an example of a p-graph defined by a consistent p-set.

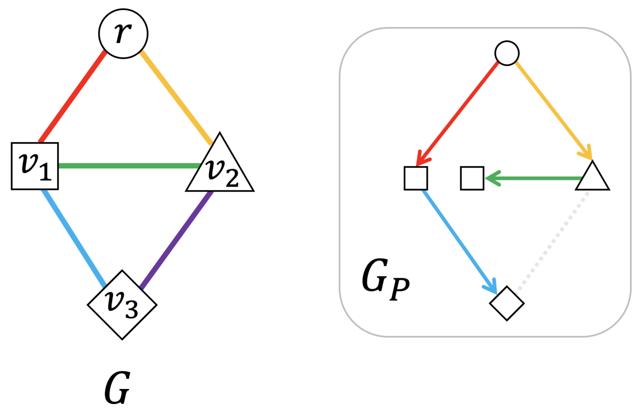

Figure 3.4: An input graph G and a p-graph defined by a valid p-set. In G_P, the purple edge is omitted.

3.3 Extended Dynamic Programming

In chapter 2.3, we discussed a dynamic programming algorithm to optimally recover a tree from a skeleton by duplicating nodes, given the problem formulation used in chapter 2.2. For this chapter, we will assume the problem formulation of section 3.2, and claim that the DP algorithm may remain unchanged. That is, no changes to the algorithm are required, aside from assuming the new definitions of *valid partition* and *partition consistency* from section 3.2.

The main contribution in this front is instead the proof of optimality, this is not immediately obvious, because despite using the same general DP algorithm, we have changed the problem formulation and added more allowable valid partitions.

3.3.1 Optimality

Proof. Our proof of optimality is similar to the proof discussed in Appendix A, with two small modifications, we can adapt this proof of optimality to the extended case.

Firstly, for lemma 1's proof, the enriched p-graph $p_{u,w}$ does not include exterior nodes for any edge e such that $\mu(e) = 0$. Figure 3.5 shows an enriched p-graph using the new formulation. Furthermore, the labels assigned to each edge do not affect the structure of the tree, as such, they are irrelevant for the discussion of lemma 1.

Secondly, for Part I of the proof of proposition 1, it remains to show that P is a consistent p-set, since the definition of consistency has been modified to accommodate for labels. However, it is given that, p_u and p_w assign both the same directions *and* labels as the exterior edges of $P'_{u,w}$, so it follows that P is consistent.

With these modifications to the proof, we have showed that optimality is still retained for the Extended Dynamic Programming algorithm.

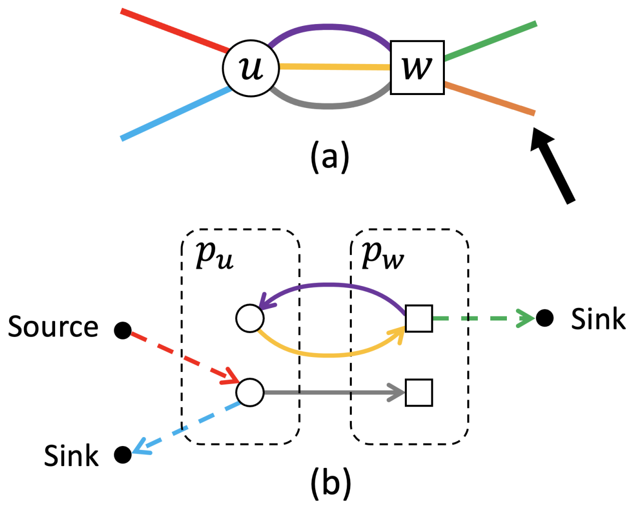

Figure 3.5: (a): Two nodes u, w to be merged. (b): An enriched p-graph of two consistent and valid partitions p_u, p_w of u, w. Interior nodes of the p-graph are instances of u ($\bigcirc$) and w ($\square$), and exterior nodes ($\bullet$) are classified as either source or sink. Exterior edges $E_u \ominus E_w$ are dashed. Note that the orange edge is removed in p_w, so no source or sink is provided for that edge.

$\square$

3.4 Extended Merge Sequence

The computation of the merge sequence remains unchanged. However, the merge cost will be different. There are many more possible valid partitions for a given degree-d node. Let Π be the function described in equation 2.5, then the number of possible partitions for a degree d node in our new formulation is given by:

$$\Pi'(d) = (1 - r_{\min}) + \sum_{i=r_{\min}d}^{d} \binom{d}{i} \Pi(i) L^i \tag{3.1}$$

This function is prohibitively complex, however, as we will see in section 3.5, the complexity will be significantly lower in practical applications.

3.5 Use Cases

The function described in equation 3.1 has a prohibitive complexity with respect to an increasing d. However, we argue that such a complexity will rarely be needed due to several factors. We note that not all possible partition, direction, and label combinations are considered in most practical applications. For this reason, we may decide which partitions may be most useful as the *input* to the DP method.

Equation 3.1 assumes an L^i number of labels at each partition. However, in practical applications, a user should only provide partitions as input with labels that abide by practical constraints. For example, the AV-WIDE blood-vessel dataset will only require as input partitions where each edge group has the same label. Only at the root may edges be labelled as both arteries or veins in the same edge group.

Furthermore, as we will shortly discuss, some datasets do not require the removal of edges, others don't require more than 1 edge-group per node. These restrictions at the input will greatly reduce the complexity of equation 3.1.

In this section, we will describe different applications for the proposed method. In particular, there are two things that will differ for each use case:

- The allowed valid partitions (I.E. The input to the EDP algorithm). Whether they allow crossings, edge-removal, whether we will label edges, etc.
- The complexity of the $\Pi'(d)$ function. That is, an upper-bound of the number of partitions we may observe at a degree d node.

- The cost function.

3.5.1 Artery-Vein Datasets

The artery-vein dataset is almost identical to the aforementioned WIDE dataset. However, this time, we are interested in labelling each edge e as either being an *artery* ($l(e) = 0$) or a *vein* ($l(e) = v$)).

Input

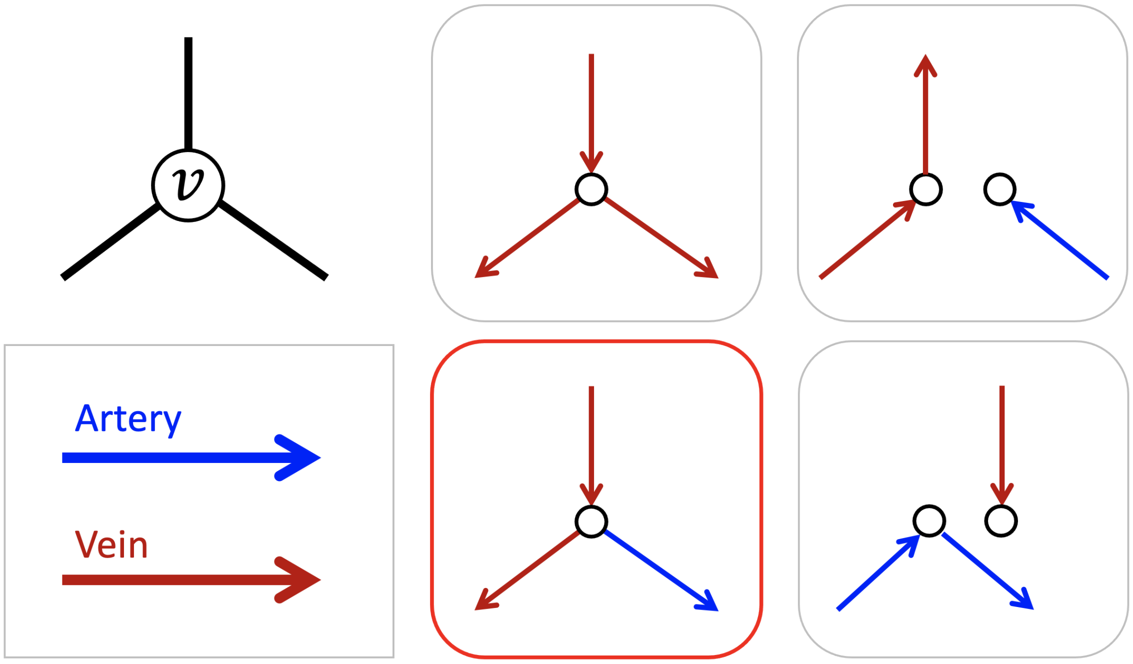

Figure 3.6: Different ways to label & partition a given node v of the AV-WIDE dataset. The bottom left example is not allowed (and in turn, not provided as input to the algorithm) because arteries and veins may not share the same node.

Since this dataset is similar to the WIDE dataset, and the skeletons are of high quality, removing edges will not be needed. Furthermore, since arteries and veins are disjoint in all visible blood vessels, we will only allow them to share the same edge-group at the root. Figure 3.6 shows an example of some edge-labelings we will allow as the input.

Complexity

We have a uniform label for each edge group. As such, the number of partitions Π_{AV} is the same as $\Pi(d)$, with an added coefficient to adjust for the 2 labels among k edge groups:

$$\Pi_{\mathrm{AV}}(d) = \sum_{k=1}^{d} \binom{d}{k} \binom{k}{2} k^{d-k} \tag{3.2}$$

Cost

This cost function is inspired by [9]. Firstly, we want to penalize when similarly labelled edge groups overlap. The idea is that veins rarely overlap other veins, and arteries rarely overlap other arteries. Let $N_a(p)$ be the number of edge groups labeled as arteries at partition p, and $N_v(p)$ the number of edge groups labeled as veins. The *overlap cost* is given by:

$$h_o(p) = -\frac{N_a(p)N_v(p)}{(N_a(p) + N_v(p))^2} \tag{3.3}$$

Secondly, we want to promote the image intensity of arteries to be greater than the image intensity of veins. Let v be a vertex with a degree greater than 2. For a partition p_v of vertex v, we may create a subraph G_v with v, all the degree 2 "vertex chains" adjacent to v, and all the associated edges. Let E_A be the edges in G_V labelled as arteries, and E_V the edges labelled as veins. Furthermore, let $\rho_e = \{\rho_1, \rho_2, \cdots, \rho_n\}$ and $\omega_e = \{\omega_1, \omega_2, \cdots, \omega_n\}$ be the sets of all pixels and weights acquired by running Xiolin Wu's line-drawing algorithm on the edge $e \in G_v$, respectively [26]. Given a color channel ch $\in \{R, G, B\}$, we may compute the following value:

$$l_{\mathrm{ch}}(p) = \sum_{e \in E_A} \rho_e^T \omega_e - \sum_{e \in E_V} \rho_e^T \omega_e \tag{3.4}$$

Then, the color cost is computed as follows:

$$h_c(p) = -\max(\frac{l_R(p), l_G(p), l_Bp}{3}, 0) \tag{3.5}$$

Finally, the cost of a given partition p for vertex v is given by:

$$h_v(p) = \lambda_g h_g(p) + \lambda_o h_o(p) + \lambda_c h_c(p) \tag{3.6}$$

Where h_g is the cost given by Appendix E.

3.5.2 Cherry Tree Pruning

Another application of the tree-recovery problem is in the automation of dormant pruning for fresh market fruit trees, discussed in You et al. [31]. In this setting, there is a robot, equipped with a stereo camera, that aims to prune old, diseased, or unproductive branches from Upright Shooting Offshoot cherry trees. An example cherry tree can be seen in 3.1. For this to work, the robot must:

1. accurately sense the environment,
2. create a robust representation of the environment,
3. decide where to cut, and
4. plan a collision-free motion to move a cutting implement to desired cut point.

The focus of You et al. is on step (2), where there is a point-cloud as input, and a skeleton tree must be recovered, while abiding by label-topological constraints. One of the main challenges of this problem is that the initial skeletonization has many cycles due to false-positive edges that don't correctly represent the underlying tree. An example initial skeletonization can be seen in figure 3.2 (b).

Input

For this problem, we will allow edge removal, since the grapevine skeletons are heavily overepresented. There are $L = 4$ possible labels, but not all combinations are allowed, since they have to adhere to a specific hieararchy. Furthermore, since these are 3D skeletons, crossings are rare or nonexistent, so we will not allow more than one edge group.

The hierarchy that the labels have to follow is described by [31]. The edges are labelled as one of the following:

- **Leader branches**, which are vertical, 3 meter branches where the tree bears cherries during growing season.
- Small, slender **side branches** that grow from leaders.
- Horizontal **support branches** (cordons), which are two strong, horizontal branches growing from trunk.
- **Trunk**, or central branch that offshoots from the ground.
- None.

Furthermore, they propose the following Label-topology constraints to abide by the structure of cherry trees:

- **Label progression**: the trunk must be succeeded by support branches. The support branches must be succeeded by leader branches, and leader branches must be succeeded by side branches.
- **Label linearity**: degree-3 nodes cannot have same label for all adjacent edges.
- **Trunk-support split**: only one or two support edges may spawn from the trunk, and it must happen in a junction with only one trunk edge.

Since we want to label edges at a given partition with these constraints, then valid and invalid partitions look like those in figure 3.7.

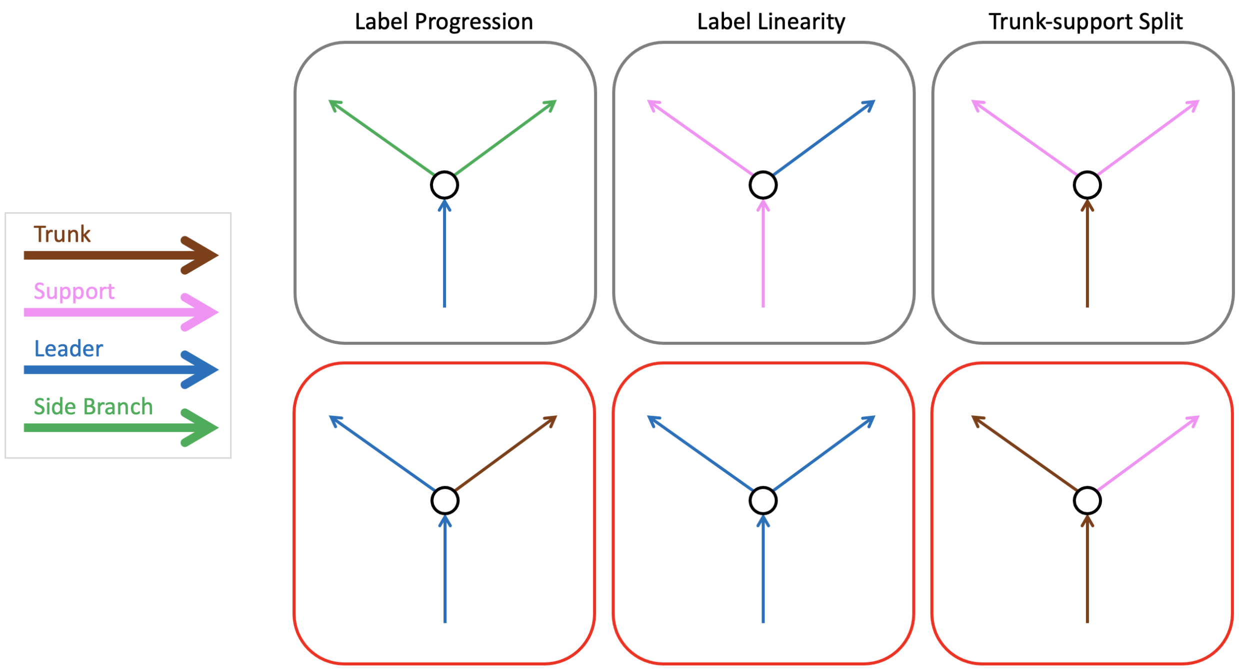

Figure 3.7: Examples of valid and invalid partitions in accordance with You et al.'s label constraints [31]. The edges are color coded in accordance to the legend on the left. For the label progression constraint, the trunk must be succeeded by support branches. The support branches must be succeeded by leader branches, and leader branches must be succeeded by side branches. For the label linearity constraint, 3 branches of the same label may not be connected. For the trunk-support split constraint, a trunk can only be succeeded by up to 2 support branches at the endpoint of a trunk.

Complexity

For the complexity of this use-case, we remove any terms associated with the number of partitions over the edges, while leaving in any term that enumerates directions and labels. For the labels, we assume that any combinations of labels can be had, so long as each label is equal or 1 plus the label of the incoming edge.

$$\Pi_{\text{cherry}}(d) = 1 + \sum_{i=0}^{\max(d,4)} \binom{d}{i} (6i^2 + i) \tag{3.7}$$

Cost

In a cherry tree, trunks and leader branches are expected to go upwards from the x-axis, while support branches are expected to go perpendicular to the x-axis. As such, we have an h_{axis} cost that penalizes edges that deviate from the expected direction of a given label, μ_l.

$$h_{\text{axis}}(p) = \sum_{(u,v)\in p} \log \frac{p_\theta(\text{arctan}_2(\mathbf{v}_y - \mathbf{u}_y, \mathbf{v}_x - \mathbf{u}_x))}{p(\mu_{l((u,v))})} \tag{3.8}$$

Furthermore, when a trunk branch ends, usually one or two support branches spawn perpendicularly from it. Let v be the vertex where the trunk ends, and the support spawns, t be the incoming vertex at the trunk, and s_1, s_2 be the support branches sorted in clockwise order with respect to v. Then the new cost is as follows:

$$h_{\text{TS}}(p) = \log(p(\angle tvs_1|\mu_{\text{TS}})) + \log(p(-\angle tvs_2|\mu_{\text{TS}})) \tag{3.9}$$

Where $\angle abc$ is the concave angle created by following the vertices a, b, c. Finally, the cost of a given partition p for vertex v is given by:

$$h_v(p) = \lambda_{\text{l}} \sum_{e\in P} ll(e) + h_{\text{axis}}(p) + h_{\text{TS}}(p) \tag{3.10}$$

where $\lambda_l = 2.6$ is a balancing parameter and $ll(e)$ is the edge-likelihood determined by the Neural Network. The *axis cost* h_{axis} is determined by the following:

3.5.3 3D Cornroot Images

The Cornroot dataset consists of 3D images obtained from CT, MRI, or CAT scans of plant roots. In this scenario, an edge collisions will create an additional edge between two branches, causing a cycle in the skeleton.

Input

The Cornroot dataset consists of 3D images obtained from CT, MRI, or CAT scans of plant roots. In this scenario, an edge collisions will create an additional edge between two branches, causing a cycle in the skeleton. As a result, the removal of edges is allowed. However, node-removal is not allowed, so we require each node to have one edge adjacent to it. This is because, contrary to the Grapevine dataset, most nodes in the skeleton are associated to the root system. Furthermore, this dataset will have no labeling of the edges, and only one edge-group per node.

Complexity

The complexity of this one is simple, since we only have to concern ourselves with directions and capacity 0 edges.

$$\Pi_{\text{cornroot}}(d) = \sum_{i=1}^{d} \binom{d}{i} i \tag{3.11}$$

Cost

We take the cornroot dataset from chapter 1, and update it to allow edge removal, and prohibit multiple edge groups. This cost is straightforward to adapt to this case, as it is the cost described in Appendix E, but with only one edge group. Removed edges in p are merely discarded for the computation of the cost.

3.6 Extended Results

We evaluate the extended DP algorithm on the three aforementioned datasets: The AV-WIDE dataset, the Cornroot dataset, and the Grapevine dataset. Contrary to the results of chapter 2, we implemented the extended DP algorithm in C++, which gives us a significant speedup over the Python implementation. All of our experiments were performed on a PC with Intel i9-10900 CPU and 64 GB of RAM.

3.6.1 AV-WIDE Dataset

The AV-WIDE dataset consists of skeleton graphs of 2D fundus images of retinal blood vessels. Both the images and skeletonized graphs are provided by the dataset. We adopt the tree-growth model proposed by Estrada et a. [9], which provides us costs with both partitioning crossings, and labeling the edges as either arteries or veins.

We tested the full extended DP algorithm without capacity-0 edges, since the input skeletons are of high-quality. Labeling, however, is allowed. A label of 0 represents an artery edge, and 1 represents a vein. Moreover, for the input, each set in a partition is provided with at most 1 label, and only the root may have 2 labels in the same set.

We report the size of the graphs in figure 3.8 (a), note that, due to the high variance in size, only the graphs marked in red are able to run optimally without beam search. Furthermore, we report the timing of the DP algorithm in figure 3.8 (b). Note that most runs are able to run within seconds with beam search. We also report the errors in figure 3.8 (c). Note that most graphs have a substantial improvement over the greedy solution.

We also compare how well our solutions match those of human experts, provided by the AV-WIDE dataset. Figure 3.9 demonstrates that visually, our results label arteries and veins very similarly to the human-graded experts.

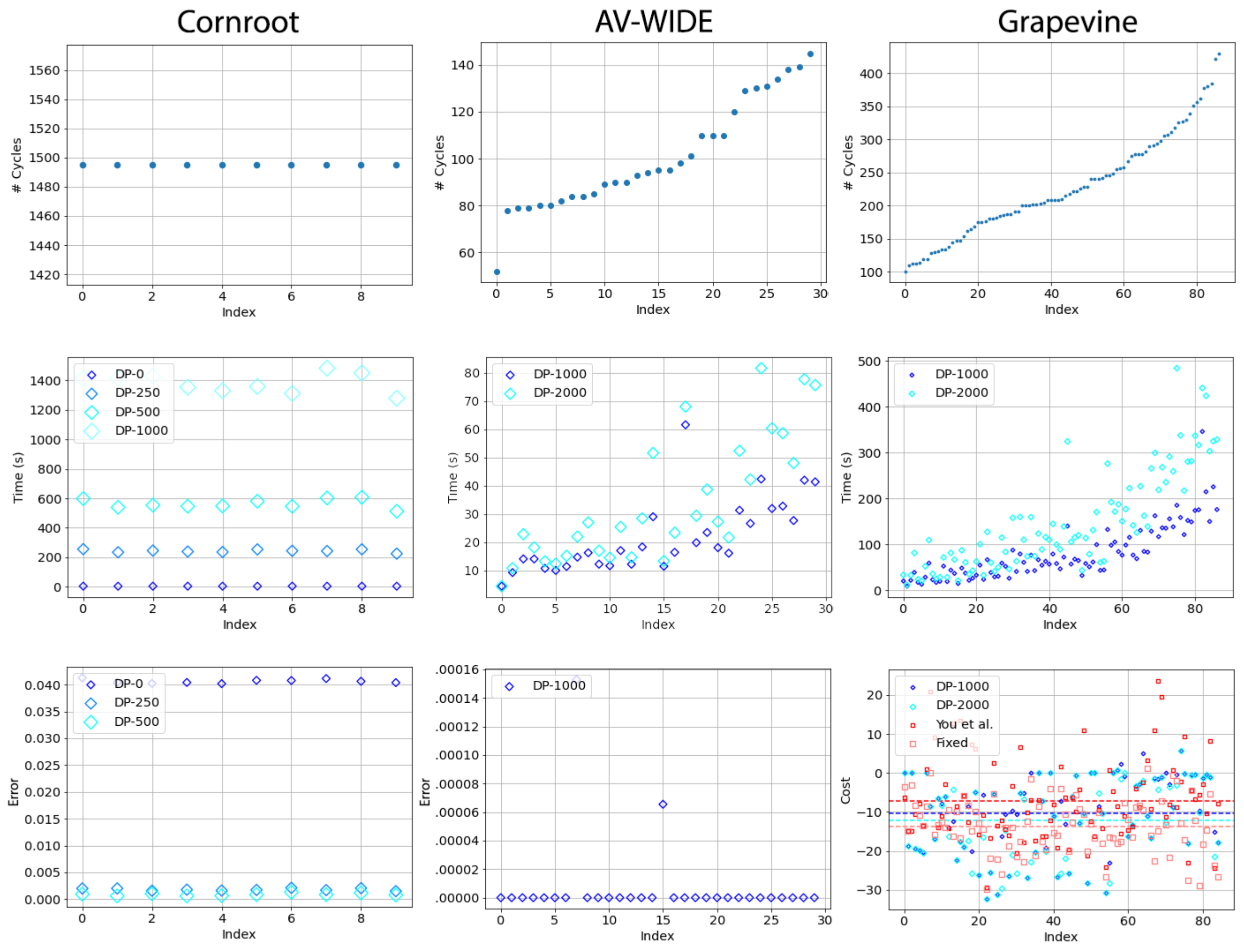

Figure 3.8: Quantitative results for the three datasets used on the extended DP method. The left column illustrates the results for the Cornroot dataset, the middle column illustrates the results for the AV-WIDE dataset, and the right column illustrates the results for the grapevine dataset. The top row shows the number of cycles, the middle row shows the execution time of the DP algorithm, and the bottom row shows the error for each graph in the dataset. For the grapevine dataset, we only the cost, rather than the error.

3.6.2 Cornroot Dataset

The cornroot dataset is described in chapter 1. However, we argue that a more appropriate way to handle the dataset is using capacity 0 edges, whilst ensuring that no node is removed. Finally, we no longer split vertices, since we are looking at a 3D skeleton, rather than a projection.

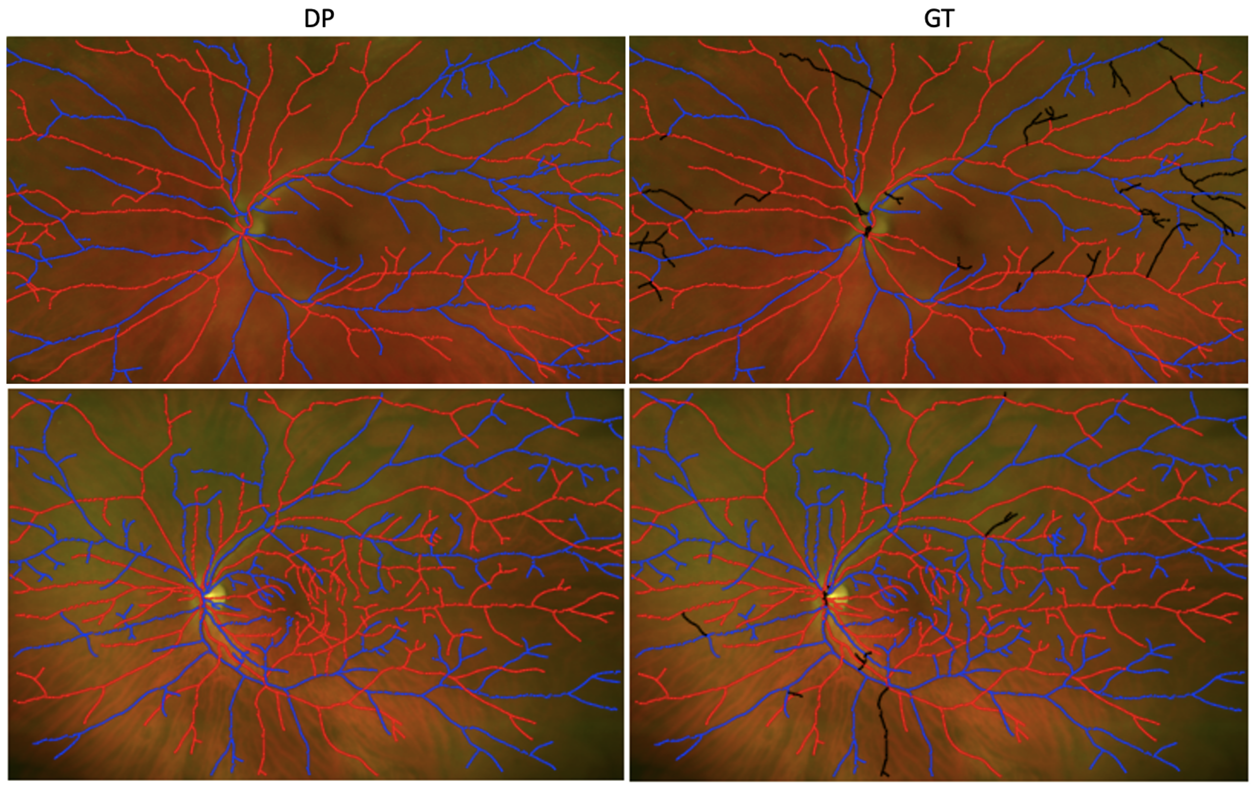

Figure 3.9: (Left) The EDP optimal results. (Right) Human-labeled results.

Figure 3.8 demonstrates the complexity and runtime of the dataset. We can see that the extended DP methods runs within the confines of the partitioning method, and the Estrada-10 and Estrada-50 methods reported in chapter 3.8. Furthermore, the error, normalized by DP-1000, improves greatly over the greedy, Minimum Spanning Tree solution.

Finally, we can visually justify the use of removing edges, rather than splitting nodes in figure 3.10, where we can see that node-splitting provides us with unnatural looking solutions, when compared to removing edges.

3.6.3 Grapevine Dataset

For the Grapevine dataset, we allow the removal of nodes and edges. Furthermore, since the dataset consists of 3D graphs that look mostly planar, we do not allow node-splitting.

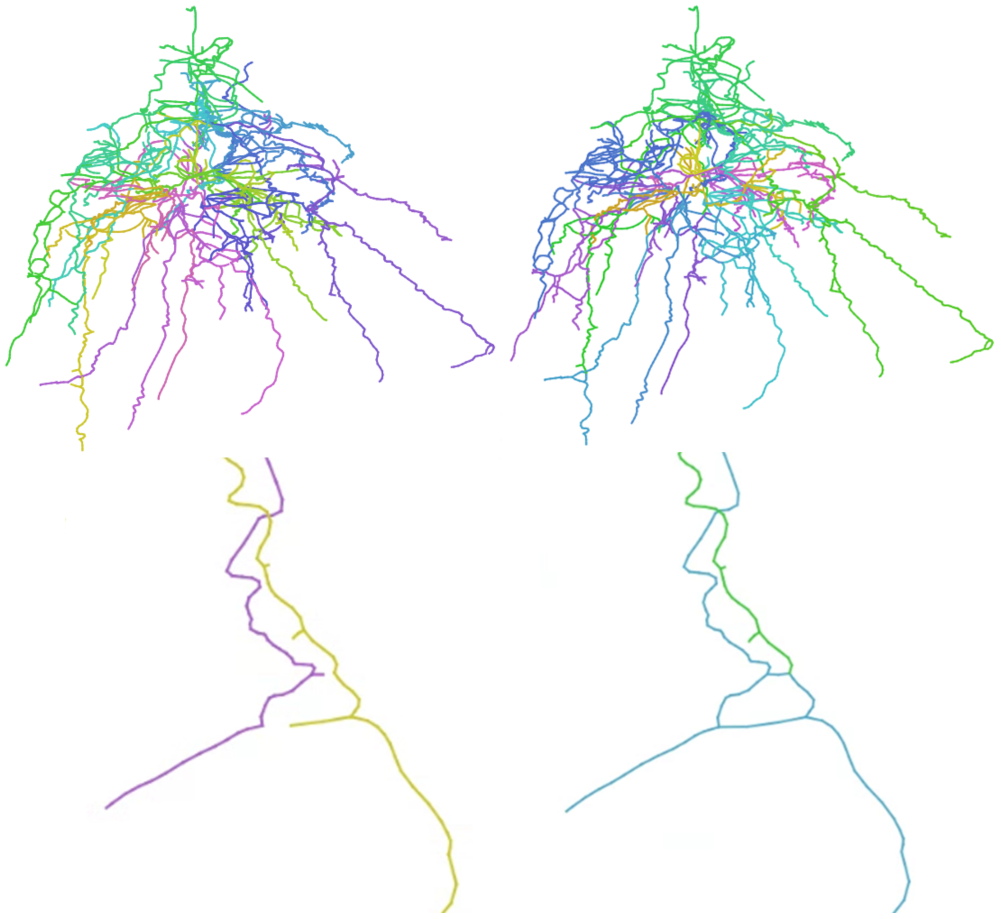

Figure 3.10: (Left) The results with edge removal. (Right) The results with node duplication. On the bottom row is a zoomed-in example that demonstrates how edge removal is more suitable for this dataset than node duplication.

Finally, our DP algorithm is made to label each edge as trunk, support, or leader. Each constraint described in [31] is enforced in the input.

The grapevine dataset consists of graphs generated from dense point-clouds. As such, the graphs in this dataset contain many cycles, and an optimal solution (DP) cannot be run. We instead run DP-1000 and DP-2000. We compare our method against [31] we denote as You et al. and are provided with a manually-fixed dataset we denote as Ground Truth.

In figure 3.8, we can see both the complexity of the graphs and the timing of our DP-1000 and DP-2000 methods. Note that [31] did not provide us with timing information in their paper, so we are not able to compare the results.

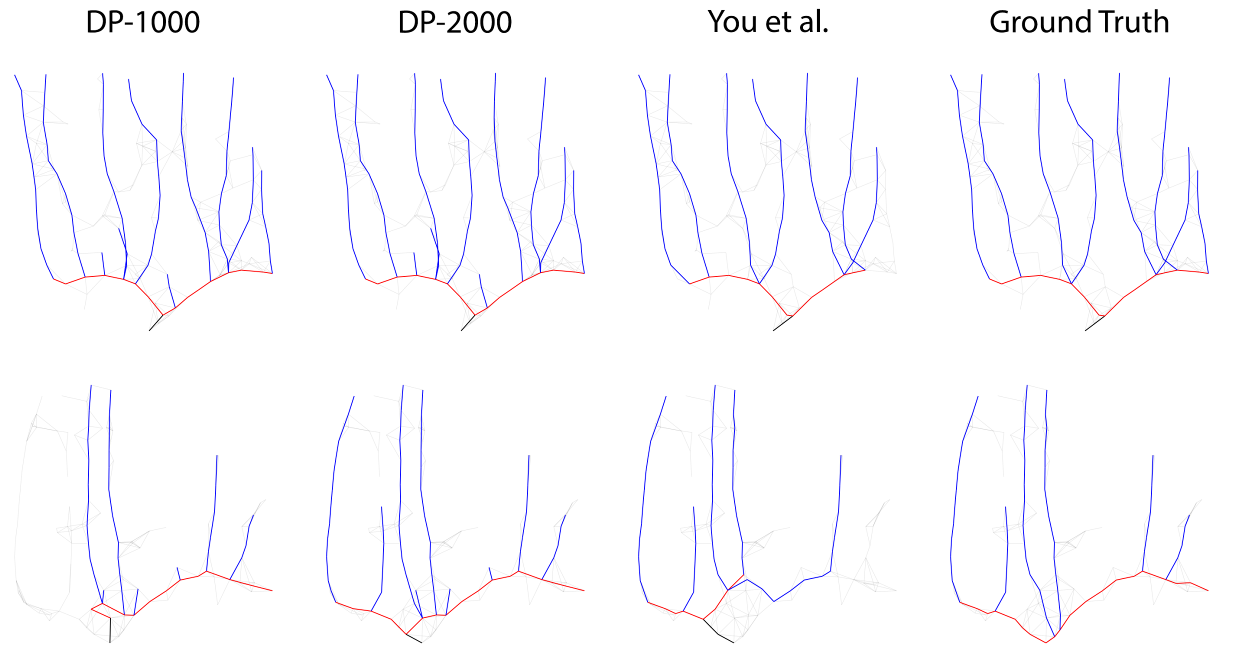

Figure 3.11: Visual results for the grapevine dataset. In both cases, DP-2000 provides a better result than You et al.

Our cost function has a coefficient which determines the importance of the edge-likelihood term λ_l. We found that at 2., the graphs with low cost look close to the ground truth, while the graphs with high cost deviate further from the ground truth. As such, we chose the coefficient to be 2. A plot of the cost function with respect to the competing method and the ground truth is found at figure 3.8.

Finally, figure 3.11 shows us some qualitative results against You et al. and Ground Truth. Note that our method can generate results that are close to Ground Truth.

3.7 Interactivity

In practice, a user may want to modify the resulting tree from the DP algorithm, either from inaccuracies of the cost function, aberrant or unusual input graphs, or the greedy nature of beam search. In this section, we discuss a minor addendum to the DP algorithm that allows users to re-run the DP algorithm with their own constraints. We show that, by reusing the

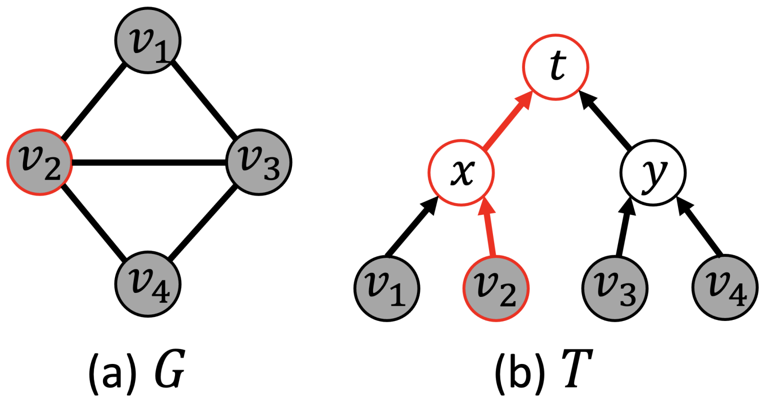

Figure 3.12: Suppose we want to rerun DP on graph G while constraining node v_2. Then we'd only need to run the merge sequence for the nodes highlighted in red in the merge tree T. That is, we'd only have to merge v_2 and v_1, and x and y, while other nodes in T remain unaffected.

merge tree T, and the generators from the first run of the DP algorithm, we may re-run the DP algorithm with new constraints while saving time.

3.7.1 Interactive Method

Once the merge sequence and DP algorithms are run, two important data structures are generated. Firstly, the merge tree, indicating how each node will be merged with other nodes throughout the DP algorithm. Secondly, the generators. For each node in the merge sequence, we store a list of partitions $P_{u,w}$ for that node, and for each partition $P_{u,w}$, we store the optimal two partitions p_u, p_w that merge into $P_{u,w}$.

Now, suppose the user runs the DP algorithm and decides to constrain node v to use partition P_v. Then, the naive option would be to re-run the entire algorithm, but provide P_v as the only allowable partition at v. However, a faster method is to only rerun the DP algorithm for the merges associated with v. That is, only run DP for v and the predecessors of v in the merge tree T, as shown in figure 3.12 (b).

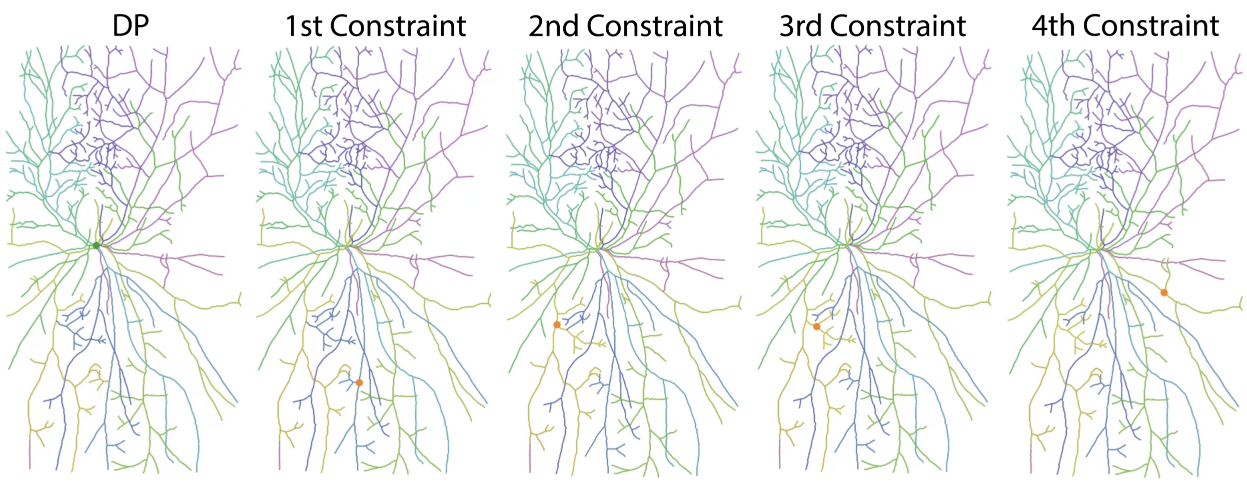

Figure 3.13: Sequential reruns used to fix a WIDE graph.

3.7.2 Interactive Results

We test user-defined constraints on two different datasets. We test our DP algorithm from chapter 2 on the WIDE dataset, and our extended DP algorithm from chapter 3 on the grapevine dataset.

For these tests, the resulting trees from the DP algorithm were laid out in a 2D plane. The user may select a node v with their own partition P'_v, and re-run the DP algorithm while constraining v's partition to be P'_v. The DP algorithm will be re-run on v and all its predecessors in the merge tree T. A new tree will be generated with v having the partition P'_v.

For the WIDE graph, we selected the result, and sequentially ran 4 different constraints. The results of this can be seen in figure 3.13. We can see that the results were fairly localized. However, when one branch is fixed, the DP algorithm tends to fix several of the subsequent branches next to it. The timing can be seen on the top plot of figure 3.15. The re-runs take less than half the time of the original algorithm.

For the Grapevine graph, we sequentially tested 2 different constraints that only involved removing nodes (or “omitting” all of its edges). The last constraint was the only one that involved the user selecting the desired edges and labels. The visual results of this can be seen in figure 3.14. While the results are favorable, the bottom plot of figure 3.15 shows that each

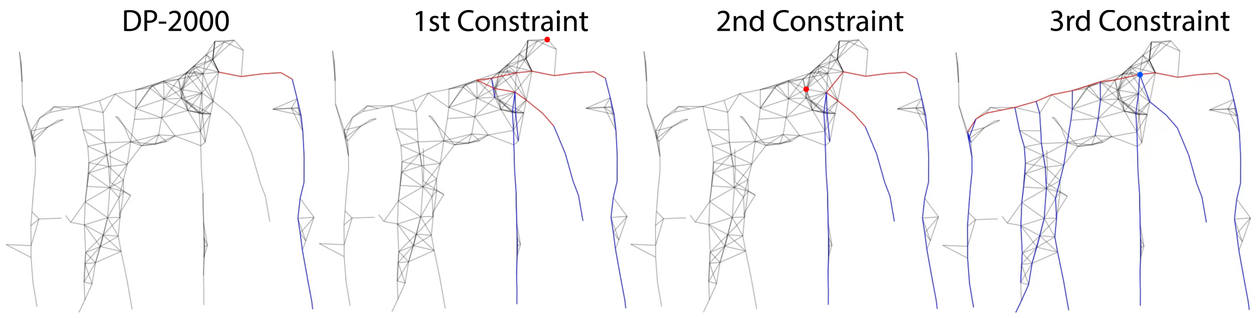

Figure 3.14: Sequential reruns used to fix a grapevine graph. Points in red are constraints performed on nodes to be removed. Points in blue are constraints performed on nodes that contain edges.

re-run took over 20 seconds. Despite being significantly less than the full DP algorithm, it is still a significant amount of time for an interactive method.

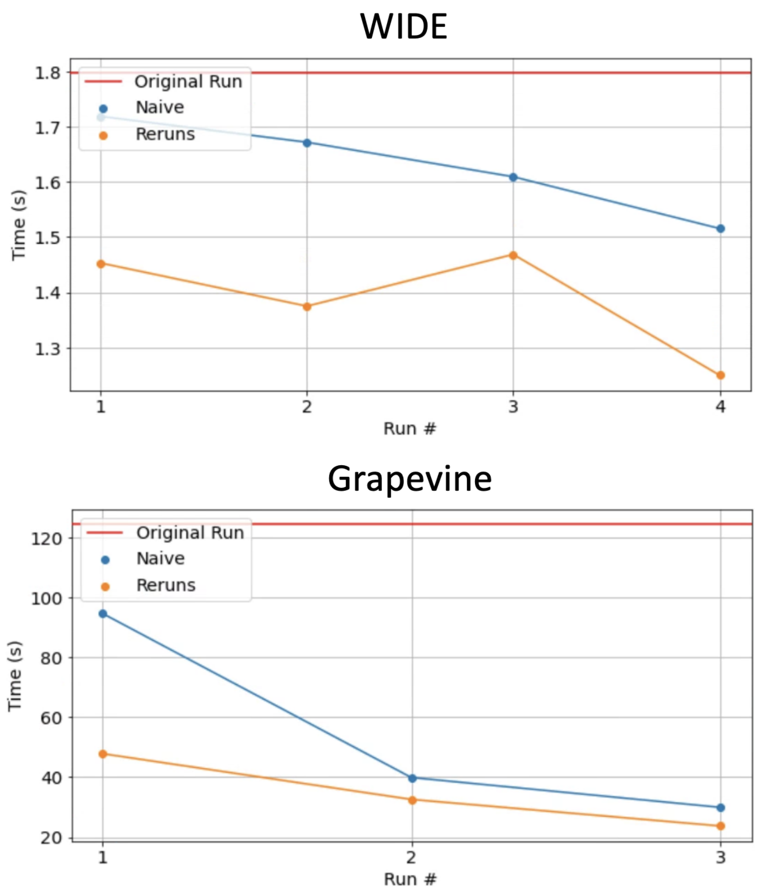

Figure 3.15: The timing it takes to perform a rerun. The original DP algorithm is shown as a red horizonal line. Each consequent rerun is a point in the plot.

3.8 Discussion

The work in this chapter extends chapter 2's work to a general setting. As such, independently of the kind of phenomena that yields cycles, the imaging modality, or use case, our method is able to duplicate nodes and remove edges to produce an accurate tree in accordance to the input cost function. Our method is also able to label edges, a requirement in many use cases.

We also demonstrate the validity of an interactive method based on our DP algorithm. The inclusion of an interactive element makes this method feasible in cases where the cost function is not enough to obtained the desired result. While in complex dataset, the interactive element still runs at a slow speed, we argue that there are ways to alleviate this: by allowing the user to select as many incorrect nodes in the first go, or allowing the user to visualize the complexity $\Pi(d)$ of each node in the DP algorithm to prioritize constraints.

Chapter 4

Conclusion

4.1 Discussion

We have constructed an algorithm, using the formulation proposed in [10], that uses Dynamic Programming (DP) to optimally partition the nodes of a noisy skeleton and produce a tree. This method iteratively merges graph nodes in such a way that the solution can be obtained from the merged nodes. We provide a novel algorithm to determine an efficient merge order, and provide a fast approximate version of the DP algorithm. Finally, we show that our method is superior to the competition on both speed and optimality on several datasets.

For chapter 2, we extend this algorithm to more general cases that require edge and node removal, and edge labelling. We show that this method still achieves optimality. Furthermore, by carefully selecting the input partitions, we show that our extended method can feasibly run on several datasets and provide great results. We finally show that our method can run in an interactive setting.

4.2 Future Work

4.2.1 Expanding the problem formulation

In chapter 2, we addressed the challenge of addressing merged nodes. However, there may be cases where two branches cross or collide with each other across multiple points, and the skeletonization algorithm produces merged edges. This happens rarely in the cornroot dataset, where branches may collide with each other throughout multiple edges (figure 4.1), and in 2D skeletonization algorithms at crossovers that span more than one point.

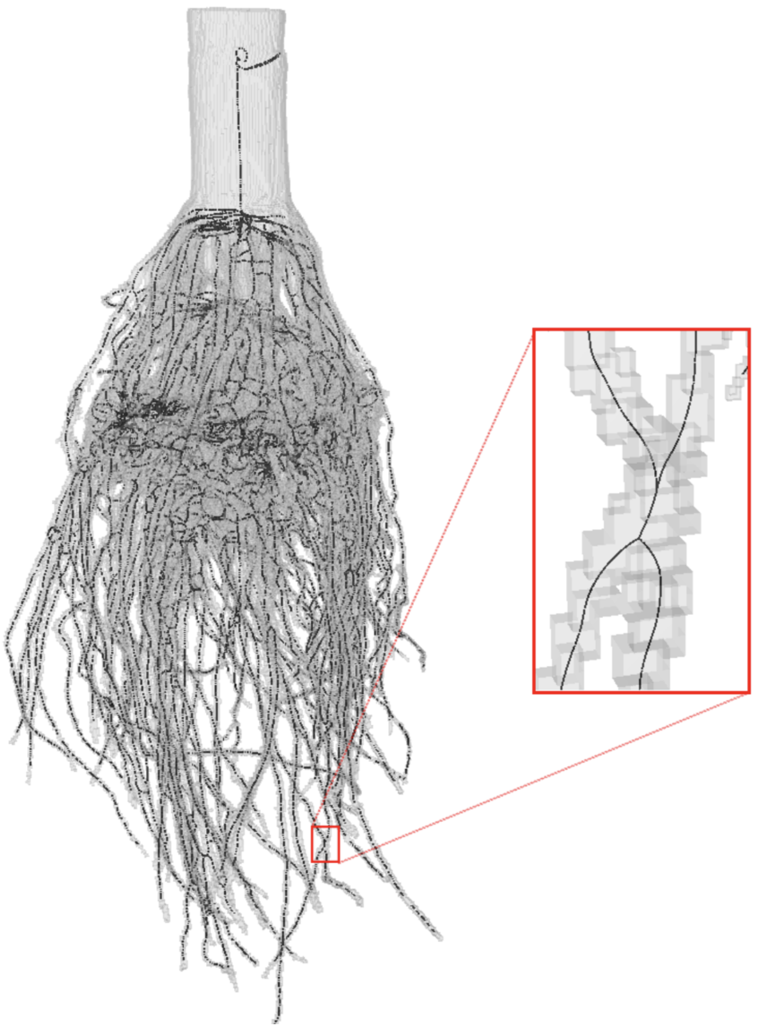

Figure 4.1: A cornroot volume and its skeleton (left), and several consecutive edges that have been merged together due to branches colliding (right).

To address this, we may extend the definition of valid partitions and partition consistency to allow for edges to be duplicated. Rather than the *edge-removal* parameter, we may have two parameters: $r_{\min}$ and $r_{\max}$ to limit the edge capacity. That is, the number of *instances* of an edge will be between $r_{\min}$ and $r_{\max}$. A valid partition will be one whose number of edge *instances* will be between $r_{\min}$ and $r_{\max}$. A new definition of partition consistency also has to be formulated. One that requires each adjacent partition to assign the same capacity for the same edge.

Also, our DP algorithm may work for many cases not covered in this thesis. Many problems, such as minimum spanning tree, the traveling salesman problem, etc. can be addressed when using our algorithm on different input partitions. There are innumerable possibilities that can be addressed by the DP algorithm within and outside of biological structures, and thus many opportunities for novel research.

4.2.2 Improving the Efficiency

For medium to large datasets, our DP algorithm is not able to run optimally. Furthermore, depending on the use case, our beam search may struggle to find a good cost solution for difficult datasets. This is why improving the efficiency of the algorithm is one of the most important aspects of extending it and making sure it is generalizable.

One way to improve the efficiency is to extend the method of creating the merge sequence. Right now, our merge-sequence computation algorithm, detailed in chapter **??**, improves upon a rooted carving decomposition by pivoting until the cost function can no longer improve. However, a less restrictive way of improving the rooted carving decomposition would be to add the ability for pivoting to ”swap” any given sub-tree of the merge tree, until the cost can no longer be improved. This tree ”grafting” method will be less likely to be stuck in local optima, when optimizing for a low merge cost.

In terms of efficiency, beam search may also be improved. Our current method only allows for a fixed beam width. However, we may want a beam width that can change variably depending on the cost or the complexity of each merge operation. At the end of each merge operation, we may filter out partitions that are not consistent with the partitions of nearby nodes before discarding solutions for beam search.

References

[1] Mehwish Bashir and Qian-Ping Gu. “Carving-decomposition based algorithms for the maximum path coloring problem.” In: *2012 IEEE International Conference on Communications (ICC)*. 2012, pp. 2977–2982. DOI: 10.1109/ICC.2012.6363762.

[2] Christian Bauer, Michael Eberlein, and Reinhard R. Beichel. “Graph-Based Airway Tree Reconstruction From Chest CT Scans: Evaluation of Different Features on Five Cohorts.” In: *IEEE Transactions on Medical Imaging* 34 (2015), pp. 1063–1076.

[3] Zhengbing Bian, Qian-Ping Gu, and Mingzhe Zhu. “Practical algorithms for branch-decompositions of planar graphs.” In: *Discrete Applied Mathematics* 199 (2016). Sixth Workshop on Graph Classes, Optimization, and Width Parameters, Santorini, Greece, October 2013, pp. 156–171. DOI: https://doi.org/10.1016/j.dam.2014.12.017.

[4] Hans L Bodlaender. “Polynomial algorithms for graph isomorphism and chromatic index on partial k-trees.” In: *Journal of Algorithms* 11.4 (1990), pp. 631–643.

[5] William Cook and Paul Seymour. “Tour Merging via Branch-Decomposition.” In: *INFORMS Journal on Computing* 15 (Aug. 2003), pp. 233–248. DOI: 10.1287/ijoc.15.3.233.16078.

[6] Behdad Dashtbozorg, Ana Maria Mendonça, and Aurélio Campilho. “An Automatic Graph-Based Approach for Artery/Vein Classification in Retinal Images.” In: *IEEE Transactions on Image Processing* 23.3 (2014), pp. 1073–1083. DOI: 10.1109/TIP.2013.2263809.

[7] Jaydeep De et al. “A Graph-Theoretical Approach for Tracing Filamentary Structures in Neuronal and Retinal Images.” In: *IEEE Transactions on Medical Imaging* 35.1 (2016), pp. 257–272. DOI: 10.1109/TMI.2015.2465962.

[8] Christophe Duhamel, Luis Gouveia, Pedro Moura, and Mauricio Souza. “Models and heuristics for a minimum arborescence problem.” In: *Networks* 51.1 (2008), pp. 34–47. DOI: https://doi.org/10.1002/net.20194. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/net.20194.

[9] Rolando Estrada, Michael J. Allingham, Priyatham S. Mettu, Scott W. Cousins, Carlo Tomasi, and Sina Farsiu. “Retinal Artery-Vein Classification via Topology Estimation.” In: *IEEE Transactions on Medical Imaging* 34.12 (2015), pp. 2518–2534. DOI: 10.1109/TMI.2015.2443117.

[10] Rolando Estrada, Carlo Tomasi, Scott C. Schmidler, and Sina Farsiu. “Tree Topology Estimation.” In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 37.8 (2015), pp. 1688–1701. DOI: 10.1109/TPAMI.2014.2382116.

[11] Michael W. Graham, Jason D. Gibbs, Duane C. Cornish, and William E. Higgins. "Robust 3-D Airway Tree Segmentation for Image-Guided Peripheral Bronchoscopy." In: *IEEE Transactions on Medical Imaging* 29.4 (2010), pp. 982–997. DOI: 10.1109/TMI.2009.2035813.

[12] Qian-Ping Gu and Hisao Tamaki. "Optimal Branch-Decomposition of Planar Graphs in O(N3) Time." In: *ACM Trans. Algorithms* 4.3 (July 2008). DOI: 10.1145/1367064.1367070.

[13] Illya V. Hicks. "Planar Branch Decompositions I: The Ratcatcher." In: *INFORMS Journal on Computing* 17.4 (2005), pp. 402–412. DOI: 10.1287/ijoc.1040.0075. eprint: https://doi.org/10.1287/ijoc.1040.0075.

[14] Illya V. Hicks, Arie M. C. A. Koster, and Elif Kolotoglu. "Branch and Tree Decomposition Techniques for Discrete Optimization." In: 2005.

[15] D. S. Johnson and K. A. Niemi. "On Knapsacks, Partitions, and a New Dynamic Programming Technique for Trees." In: *Mathematics of Operations Research* 8.1 (1983), pp. 1–14.

[16] Ephraim Korach and Nir Solel. *Linear time algorithm for minimum weight Steiner tree in graphs with bounded tree-width.* Tech. rep. Computer Science Department, Technion, 1990.

[17] Luis Lopez, Yuanyuan Ding, and Jingyi Yu. "Modeling Complex Unfoliaged Trees from a Sparse Set of Images." In: *Comput. Graph. Forum* 29 (Sept. 2010), pp. 2075–2082. DOI: 10.1111/j.1467-8659.2010.01794.x.

[18] Brian Matejek, Daniel Haehn, Haidong Zhu, Donglai Wei, Toufiq Parag, and Hanspeter Pfister. "Biologically-Constrained Graphs for Global Connectomics Reconstruction." In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).* June 2019.

[19] Johan M. M. van Rooij, Hans L. Bodlaender, Erik Jan van Leeuwen, Peter Rossmanith, and Martin Vatshelle. "Fast Dynamic Programming on Graph Decompositions." In: *CoRR* abs/1806.01667 (2018). arXiv: 1806.01667.

[20] Raghavendra Selvan, Thomas Kipf, Max Welling, Antonio Garcia-Uceda Juarez, Jesper H Pedersen, Jens Petersen, and Marleen de Bruijne. "Graph refinement based airway extraction using mean-field networks and graph neural networks." In: *Medical Image Analysis* 64 (2020), p. 101751.

[21] Paul D. Seymour and Robin Thomas. "Call routing and the ratcatcher." In: *Combinatorica* 14 (1994), pp. 217–241.

[22] Jan Arne Telle and Andrzej Proskurowski. "Algorithms for vertex partitioning problems on partial k-trees." In: *SIAM Journal on Discrete Mathematics* 10.4 (1997), pp. 529–550.

[23] Engin Turetken, Fethallah Benmansour, Bjoern Andres, Przemysław Głowacki, Hanspeter Pfister, and Pascal Fua. "Reconstructing Curvilinear Networks Using Path Classifiers and Integer Programming." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 38.12 (2016), pp. 2515–2530. DOI: 10.1109/TPAMI.2016.2519025.

[24] Engin Turetken, Germán González, Christian Blum, and Pascal Fua. "Automated Reconstruction of Dendritic and Axonal Trees by Global Optimization with Geometric Priors." In: *Neuroinformatics* 9 (Sept. 2011), pp. 279–302. DOI: 10.1007/s12021-011-9122-1.

[25] Yu Wang, Arunachalam Narayanaswamy, and Badrinath Roysam. "Novel 4-D open-curve active contour and curve completion approach for automated tree structure extraction." In: *CVPR 2011*. IEEE. 2011, pp. 1105–1112.

[26] Xiaolin Wu. "An efficient antialiasing technique." In: *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '91. New York, NY, USA: Association for Computing Machinery, 1991, pp. 143–152. DOI: 10.1145/122718.122734.

[27] Hang Xiao and Hanchuan Peng. "APP2: Automatic Tracing of 3D Neuron Morphology Based on Hierarchical Pruning of Gray-Weighted Image Distance-Trees." In: *Bioinformatics (Oxford, England)* 29 (Apr. 2013). DOI: 10.1093/bioinformatics/btt170.

[28] Jianyang Xie, Yitian Zhao, Yonghuai Liu, Pan Su, Yifan Zhao, Jun Cheng, Yalin Zheng, and Jiang Liu. "Topology Reconstruction of Tree-Like Structure in Images via Structural Similarity Measure and Dominant Set Clustering." In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019, pp. 8497–8505. DOI: 10.1109/CVPR.2019.00870.

[29] Yajie Yan, David Letscher, and Tao Ju. "Voxel cores: Efficient, robust, and provably good approximation of 3d medial axes." In: *ACM Transactions on Graphics (TOG)* 37.4 (2018), pp. 1–13.

[30] Yajie Yan, Kyle Sykes, Erin Chambers, David Letscher, and Tao Ju. "Erosion thickness on medial axes of 3D shapes." In: *ACM Transactions on Graphics (TOG)* 35.4 (2016), pp. 1–12.

[31] Alexander You, Cindy Grimm, Abhisesh Silwal, and Joseph R. Davidson. "Semantics-guided skeletonization of upright fruiting offshoot trees for robotic pruning." In: *Computers and Electronics in Agriculture* 192 (2022), p. 106622. DOI: https://doi.org/10.1016/j.compag.2021.106622.

[32] Ting Zhao, Jun Xie, Fernando Amat, Nathan Clack, Parvez Ahammad, Hanchuan Peng, Fuhui Long, and Eugene Myers. "Automated reconstruction of neuronal morphology based on local geometrical and global structural models." In: *Neuroinformatics* 9.2 (2011), pp. 247–261.

Appendix A

Optimal recovery

A.0.1 Proof Of Optimality

Before proving our main result, Proposition 1, we make an observation on a pair of valid partitions that merge into another valid partition.

Lemma 1. *Let $\{p_u, p_w\}$ be a consistent pair of valid partitions of nodes $u, w \in V$ whose merged partition $p_u \circ p_w$ is also a valid partition of the merged node $x \in V'$. Then each connected component of the p-graph $p_{u,w}$ is an arborescence.*

Proof. For convenience of discussion, we shall enrich the structure of the p-graph $p_{u,w}$ by additional nodes as follows. Let E_u and E_w be edges incident to u and w prior to merging. We call the set $E_u \ominus E_w$ *exterior edges*, where $\ominus$ is the disjunctive union. For each capacity-1 exterior edge in $p_{u,w}$, we attach an *exterior node* to the end that is not an instance of u or w. Accordingly, we call an instance of u or w in $p_{u,w}$ an *interior node*. We call an exterior node a *sink* if its incident exterior edge is directed towards that node, and a *source* otherwise. Figure A.1 (b) shows an enriched p-graph of a consistent pair of valid partitions of nodes u, w in A.1 (a).

As both p_u and p_w are valid partitions, every (interior or exterior) node of $p_{u,w}$ has at most one incoming edge, and only an instance of the root $r \in V$ or a source exterior node has no incoming edges. Let V_M, E_M be the nodes and edges of a connected component M of $p_{u,w}$. Since no two edges in E_M are directed towards the same node, we can count the number of nodes without incoming edges, k_M, as

$$k_M = |V_M| - |E_M|.$$

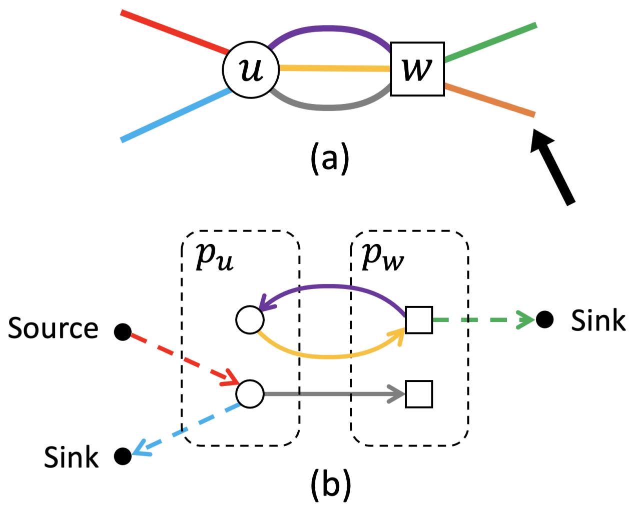

Figure A.1: (a): Two nodes u, w to be merged. (b): An enriched p-graph of two consistent and valid partitions p_u, p_w of u, w. Interior nodes of the p-graph are instances of u (◯) and w (□), and exterior nodes (•) are classified as either source or sink. Exterior edges $E_u \ominus E_w$ are dashed.

On the other hand, the number of undirected cycles in M, denoted by g_M, is related to the number of nodes and edges by

$$g_M = |E_M| - |V_M| + 1.$$

Combining the two equations yields

$$k_M = 1 - g_M.$$

Both k_M and g_M are non-negative integers. Hence we only have two possibilities to consider.

We first consider $\{g_M = 0, k_M = 1\}$. In this case, the undirected graph of M is acyclic and there is a single node without incoming edges. This node, denoted by r_M, can be a source exterior node or an instance of r. Since every other node of M has exactly one incoming edge, M is an arborescence rooted at r_M. An example of M is the bottom component in Figure A.1 (b), which is rooted at a source exterior node.

Next consider $\{g_M = 1, k_M = 0\}$. In this case, M has one undirected cycle and every node has one incoming edge. It follows that every exterior node of M (if there are any) is a sink. An example of M is the top component in Figure A.1 (b). We will show that this case cannot arise if the merged partition $p_u \circ p_w$ is valid. As M has no source exterior nodes, its corresponding instance of x in $p_u \circ p_w$, denoted by x_M, has no incoming edges. The validity of $p_u \circ p_w$ implies that x is the root of the contracted graph, and hence $r \in \{u, w\}$. Since both p_u and p_w are valid, there is exactly one interior node of $p_u \circ p_w$ (an instance of r), denoted by i, that has no incoming edges. Note that i cannot lie in M, because every node of M has an incoming edge. Let M' be the component of $p_u \circ p_w$ containing i. Then M' falls into the case above, and it is an arborescence. Since the arborescence is rooted at an interior node i, M' has no source exterior node, and hence its corresponding instance of x in $p_u \circ p_w$, denoted by $x_{M'}$, has no incoming edges. Therefore there exist two instances of x in $p_u \circ p_w$, x_M and $x_{M'}$, that have no incoming edges, which contradicts the validity of $p_u \circ p_w$.

As only the first case $\{g_M = 0, k_M = 1\}$ is possible, every component of $p_u \circ p_w$ must be an arborescence.

□

We now prove Proposition 1. The proposition states that the optimal p-set of a graph $G = \{V, E\}$ and partition costs h can be recovered by Equation 2.4 from the optimal p-set of the graph $G' = \{V', E'\}$ after merging $u, w \in V$ into $x \in V'$ and its costs h' defined by Equations 2.2 and 2.3.

Proof of Proposition 1. The proof consists of two parts.

Part I We first prove that P constructed in Equation 2.4 is a valid p-set of G. We start by showing that P is a consistent p-set. Since the pair of partitions $\{p_u, p_w\}$ is a generator and P'_x is their merged partition, p_u and p_w must be consistent and they assign the same directions as P'_x to exterior edges $E_u \ominus E_w$. As P'_x is consistent with all other partitions $P_v = P'_v$ for $v \in V \setminus \{u, w\}$, so are p_u and p_w, and hence P is consistent.

It remains to show that the p-graph of P, G_P, is an arborescence. G_P can be constructed by replacing the partition P'_x of x in the p-graph $G'_{P'}$ by the p-graph $p_{u,w}$. Note that $G'_{P'}$ is an arborescence, since P' is a valid p-set of G', and each connected component of P'_x (a valid partition of x) is an arborescence. By Lemma 1, each component of P'_x corresponds to a component of $p_{u,w}$ that is also an arborescence. Hence replacing P'_x in $G'_{P'}$ by $p_{u,w}$ results in another arborescence, G_P.

Part II We next show that $h(P) \leq h(\overline{P})$ for any other valid p-set $\overline{P}$ of G. Consider the partitions $\overline{P}_u$ and $\overline{P}_w$ of u and w, and denote their merged partition $\overline{P}_u \circ \overline{P}_w$ as p_x. We can define a p-set of G' from $\overline{P}$ as

$$\overline{P}' = \overline{P} \setminus \{\overline{P}_u, \overline{P}_w\} \cup p_x.$$

We will show that $\overline{P}'$ is a valid p-set of G'. $\overline{P}'$ is consistent because $\overline{P}$ is consistent and p_x retains the directions of exterior edges $E_u \ominus E_w$ in $\overline{P}_u$ and $\overline{P}_w$. The p-graph $G'_{\overline{P}'}$ of $\overline{P}'$ can be obtained from the p-graph $G_{\overline{P}}$ of $\overline{P}$ by replacing the p-graph $\overline{P}_{u,w}$ of the partition pair $\{\overline{P}_u, \overline{P}_w\}$ with the merged partition p_x. As $G_{\overline{P}}$ is an arborescence, and since each component of p_x contains a single node that not only corresponds to a component of $\overline{P}_{u,w}$ but also maintains the component's exterior edge directions, $G'_{\overline{P}'}$ is an arborescence too.

By the definition of partition cost h'_x in Equation 2.3, and since the partition pair $\{\overline{P}_u, \overline{P}_w\}$ is consistent and valid, we know that $h'_x(p_x) \leq h_u(\overline{P}_u) + h_w(\overline{P}_w)$ and hence $h'(\overline{P}') \leq h(\overline{P})$. Due to optimality of P', and since $\overline{P}'$ is a valid p-set of G, $h'(\overline{P}') \geq h'(P')$. On the other hand, the pair $\{p_u, p_w\}$ being the generator of P'_x implies $h'_x(P'_x) = h_u(p_u) + h_w(p_w)$, and hence $h'(P') = h(P)$. Combining these results yields

$$h(P) = h'(P') \leq h'(\overline{P}') \leq h(\overline{P}),$$

which concludes the proof. □

Appendix B

Counting partitions

We shall prove that:

Proposition 2. *The number of all possible valid partitions of a degree-d non-root node is* $\sum_{k=1}^{d} \begin{pmatrix} d \\ k \end{pmatrix} k^{d-k}$.

Proof. The number of decompositions of d elements into k groups, with x_i number of variations for each group of i elements, is known as the *Bell polynomial* $B_{d,k}(x_1, \ldots, x_{d-k+1})$. For a non-root node, each instance in a valid partition has exactly one incoming edge, and hence there are i possible assignments of the incoming edge for an instance with i incident edges. Therefore the total number of valid partitions with k instances is $B_{d,k}(1, \ldots, d-k+1)$. This number is also known as the *idempotent number* and has the form $\begin{pmatrix} d \\ k \end{pmatrix} k^{d-k}$. The total number of valid partitions, for any number of instances, is the sum of the idempotent number for all $k = 1, \ldots, d$. $\square$

Note that the root node r has fewer valid partitions than a non-root node, because one of its instances is incident to only outgoing edges and thus has fewer variations than other instances (where any incident edge may be an incoming edge).

Appendix C

Computing low-width carving decompositions

We detail the first step of our algorithm for computing the m-tree (Section 2.4.3), which computes a carving-decomposition of a graph $G = \{V, E\}$ with low width using the heuristic of [5]. The heuristic was originally designed for computing a different type of decomposition known as the *branch-decomposition*, whose leaf nodes are edges E instead of nodes V. We adapt the heuristic to carving-decompositions.

The heuristic works by iteratively splitting nodes. We define a *partial carving-decomposition* of G as a tree where each leaf corresponds to a node in V and each interior node has a degree of *at least* 3. Starting from an initial partial carving-decomposition where a single interior node connects to all $|V|$ leaf nodes, the heuristic iteratively splits an interior node whose degree is greater than 3 into two interior nodes with lower degrees. As illustrated in Figure C.1, splitting an interior node z of a partial carving-decomposition D involves replacing z by two nodes $\{x, y\}$ and connecting them by an edge. In addition, z's incident edge set Z in D is partitioned into two groups, X and Y, each connected to x or y.

To produce a carving-decomposition with low width, we wish to find a split of z so that the new edge (x, y) has low capacity. In addition, to avoid making short-sighted decisions, both $|X|$ and $|Y|$ cannot be too small. These two goals are achieved by performing a *balanced cut* of the edge set Z.

Specifically, define the *cut set* C_e of an edge e of a partial carving-decomposition D as the set of edges in E that connect two leaf nodes of V whose corresponding leaf nodes in D are connected by a path containing e. Note that the cardinality of C_e, $|C_e|$, is the capacity of e. Consider a weighted complete graph $\mathcal{Z}$ whose nodes correspond to edges in Z, and define the weight of an edge between two nodes $\{a, b\}$ to be $|C_a \cap C_b|$. The capacity of the edge

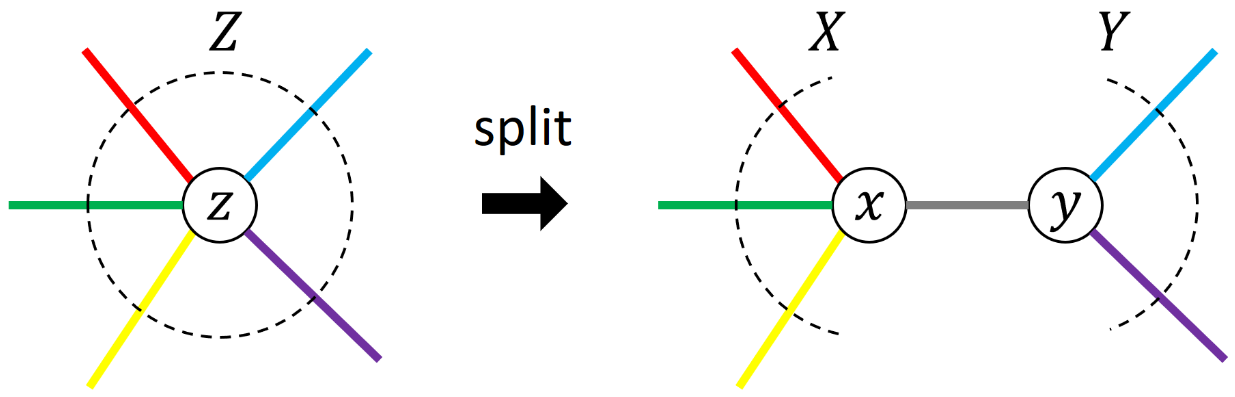

Figure C.1: Splitting an interior node z of a partial carving-decomposition into two interior nodes $\{x, y\}$, each connected with a subset (X or Y) of the incident edges Z of z.

(x, y) for edge groups X and Y can be interpreted as the total weights of edges of $\mathcal{Z}$ that connect between node groups X and Y. We then compute the eigenvector corresponding to the second smallest eigenvalue of the Laplacian matrix of $\mathcal{Z}$, sort the elements of the eigenvector, and compute the min-cut of $\mathcal{Z}$ that partitions Z into two components X, Y such that X (resp. Y) contains the first (resp. last) $\lceil \frac{|Z|}{3} \rceil$ elements of the sorted eigenvector.

Appendix D

Pivoting

We shall prove that the space of all m-trees are connected by the pivoting move introduced in Section 2.4.3.

Proposition 3. *Given two m-trees T, T' of the same graph $G = \{V, E\}$, there exists a finite sequence of pivoting moves that start from T and end with T'.*

Proof. We call the leaf nodes in the subtree of T rooted at an interior node v the *leaves of* v, denoted as $T(v)$. We label a node v "clean" if there is a node v' in T' such that $T(v) = T'(v')$, and "dirty" otherwise. We call v' the *twin* of v. We can label clean and dirty nodes in T' symmetrically. It is easy to see that $T = T'$ if and only if all nodes of T (or T') are clean.

Our proof constructs a sequence of pivots from T that reduce its number of dirty nodes. Consider a pair of clean nodes $\{x, y\} \subseteq T$ such that their twins $\{x', y'\} \subseteq T'$ (also clean) are siblings whose parent w is dirty. Such pair always exists, because all leaf nodes of T' are clean by definition, and hence the dirty node in T' with the shallowest subtree must have two clean children. Let l_x and l_y be the paths from x and y to the root of T, and u be the first node shared by both paths. Denote all nodes on l_x (resp. l_y) between x (resp. y) and u as Q_x (resp. Q_y) (see Figure D.1 (a)).

We now show that all nodes in $Q_x \cup Q_y$ are dirty. This is because the leaves of each such node include either $T(x)$ or $T(y)$ but not both. On the other hand, all nodes of $T' \setminus \{x', y'\}$ whose leaves include $T(x')$ or $T(y')$ must include both sets (since x', y' are siblings). Thus no node in $Q_x \cup Q_y$ has a twin in T'. Note that either Q_x or Q_y may be empty, but $Q_x \cup Q_y \neq \emptyset$ (otherwise x and y would be siblings, making u the twin of w and contradicting that w is dirty).

We will construct a sequence of pivots from T that consists of two sub-sequences:

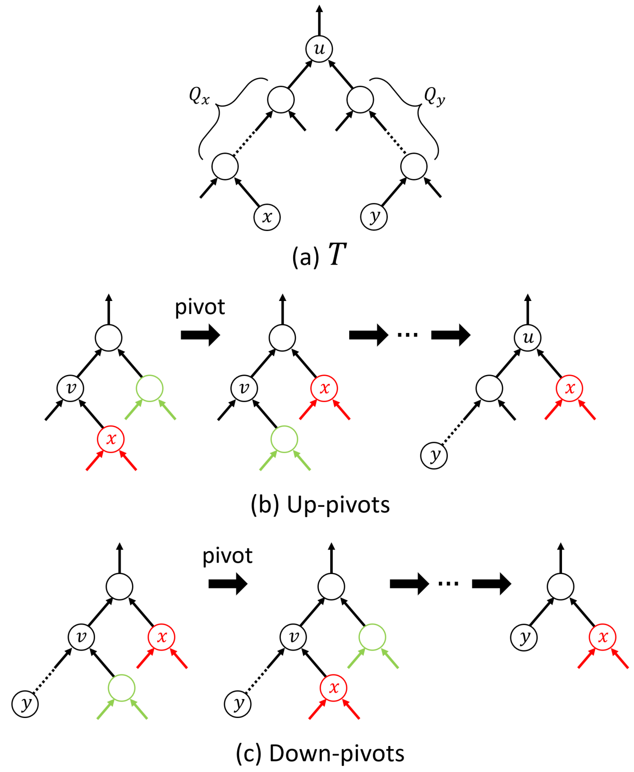

Figure D.1: Illustrations for the proof.

Up-pivots (Figure D.1 (b)), iteratively pivoting x with its (unique) aunt, until x is a direct child of u, and down-pivots (Figure D.1 (c)), iteratively pivoting x with one of its two nieces that is not an ancestor of y, until x is a sibling of y.

This sequence always ends with x and y being siblings. Their new parent is therefore the twin of w and hence is clean. The only node whose leaves changes is either the parent of x in an up-pivot (e.g., v in Figure D.1 (b)) or an ancestor of y in a down-pivot (e.g., v in Figure D.1 (c)). In either case, such a node is an ancestor of one of x and y, but not both, before the respective pivot, and hence it must be one of $Q_x \cup Q_y$. As a result, the leaves of the rest of the nodes, $T \setminus (Q_x \cup Q_y)$, remains unchanged after the sequence. As all of $Q_x \cup Q_y$ were originally dirty before the sequence, and at least one new clean node is created (parent of x and y) as a result of the sequence, the sequence reduces the number of dirty nodes of T by at least one. Hence performing a finite number of such sequences will eventually remove all dirty nodes, producing T'.

$\square$

Appendix E

Partition costs

We provide the definition of the partition cost function h used for the skeleton graphs in our experiments (Section 2.6.2). The definition on 2D skeleton graphs is identical to that in Estrada et al. [10], which we review first and then extend to 3D skeleton graphs.

Consider a 2D skeleton graph $G = \{V, E\}$, and a valid partition p of a node $v \in V$ with incident edges E_v. Suppose p consists of $m \geq 1$ instances of v, each associated with an edge group $g_i \subseteq E_v$ for $i = 1, \ldots, m$. Recall that, since p is valid, each group g_i must contain exactly one incoming edge and zero or more outgoing edges. We denote the set of out-going edges in group g_i as g_i^+ and the incoming edge as g_i^-. The partition cost of p is defined as

$$h_v(p) = -\sum_{i=1}^{m}(w_1 \log d_1(|g_i^+|) + w_2 \log d_2(g_i)), \tag{E.1}$$

where w_1, w_2 are balancing weights of two log-likelihood terms, d_1 and d_2. The first term, $d_1(c)$, evaluates the likelihood of a tree node having c outgoing edges. It is defined by a Poisson distribution with expected value (and variance) λ,

$$d_1(c) = \begin{cases} \frac{1}{1+\alpha}(\text{Pois}(c;\lambda) + \alpha) & \text{if } c = 1 \\ \frac{1}{1+\alpha}\text{Pois}(c;\lambda) & \text{otherwise} \end{cases} \tag{E.2}$$

where $\alpha >> \text{Pois}(1;\lambda)$ acts to encourage the continuation of a branch instead of splitting. The second term, $d_2(g)$ where g is an edge group of p, is the sum of the likelihood of the direction of each outgoing edge in g^+ with respect to the incoming edge g^-,

$$d_2(g) = \sum_{e_i \in g^+} \text{VMF}(e_i; (\mu_i(|g^+|, g^-) + F), \kappa). \tag{E.3}$$

Here VMF is the von Mises-Fisher distribution (restricted to a unit circle), κ is the concentration, F is a global force vector (e.g., gravity), and $\mu_i(c, v)$ is the expected direction of the i-th outgoing edge for a tree node with c outgoing edges and incoming edge direction v. In the implementation of [10] (as well as in our experiments), the parameters $(w_1, w_2, \lambda, \alpha, F, \kappa)$ were learned from a training set of 2D skeleton graphs, and the expected direction $\mu_i(c, v)$ is defined by sampling c vectors uniformly on a circle so that one of them is aligned with the incoming edge direction v.

We apply the same cost definition (Equations E.1, E.2, E.3) and parameters to the 3D skeleton graphs from our CORN dataset, after making two adjustments. First, we use the gravity direction of each root example as the force vector F. Second, to obtain the expected direction $\mu_i(c, g^-)$ for an edge group g, we compute the optimal alignment between an 1-to-c 3D branching template and g. To do so, we first align the incoming edge of the template with g^-. We then compute, for each permutation of the outgoing edges of the template, the rotation of those edges around g^- that aligns with the corresponding edges in g^+ with the least error (measured by the sum of dot products). This can be done using the technique of Singular Value Decomposition. Finally, we take the permutation that yields the least-error rotation.