Linear Pseudo-Polynomial Factor Algorithm for Automaton Constrained Tree Knapsack Problem

Soh Kumabe[1,2], Takanori Maehara[2(✉)], and Ryoma Sin'ya[3]

[1] The University of Tokyo, Tokyo, Japan
sou.kumabe@riken.jp
[2] RIKEN Center for Advanced Intelligence Project, Tokyo, Japan
takanori.maehara@riken.jp
[3] Akita University, Akita, Japan
ryoma@math.akita-u.ac.jp

Abstract. The *automaton constrained tree knapsack problem* is a variant of the knapsack problem in which the items are associated with the vertices of the tree, and we can select a subset of items that is accepted by a tree automaton. If the capacities or the profits of items are integers, it can be solved in pseudo-polynomial time by the dynamic programming algorithm. However, this algorithm has a quadratic pseudo-polynomial factor in its complexity because of the max-plus convolution. In this study, we propose a new dynamic programming technique, called *heavy-light recursive dynamic programming*, to obtain algorithms having linear pseudo-polynomial factors in the complexity. Such algorithms can be used for solving the problems with polynomially small capacities/profits efficiently, and used for deriving efficient fully polynomial-time approximation schemes. We also consider the k-subtree version problem that finds k disjoint subtrees and a solution in each subtree that maximizes total profit under a budget constraint. We show that this problem can be solved in almost the same complexity as the original problem.

Keywords: Knapsack problem · Dynamic programming
Tree automaton

1 Introduction

1.1 Background and Motivation

The knapsack problem seeks a set of items that maximizes total profit under a budget constraint. The problem is one of the most fundamental combinatorial optimization problems [12] and has many real-world applications such as scheduling [9], network design [15], and natural language processing [7]. The problem is NP-hard; however, if the profits or the weights of items are integers, the problem can be solved using the dynamic programming (DP) that runs in

G. K. Das et al. (Eds.): WALCOM 2019, LNCS 11355, pp. 248–260, 2019.
https://doi.org/10.1007/978-3-030-10564-8_20

pseudo-polynomial time. This algorithm is the basis for the fully-polynomial time approximation scheme (FPTAS) of the knapsack problem [9,13].

Here, we consider the *automaton constrained tree knapsack problem*, which is defined as follows. Let $T = (V(T), E(T))$ be a rooted tree where $V(T)$ is the set of vertices and $E(T)$ is the set of edges, $\mathcal{F}(\mathcal{A}) \subseteq 2^{V(T)}$ be a feasible domain represented by a top-down tree automaton (see Sect. 2.1 for details). We denote by $n = |V(T)|$ the number of vertices in T. Each $u \in V(T)$ has profit $p(u) \in \mathbb{R}_{\geq 0}$ and weight $w(u) \in \mathbb{R}_{\geq 0}$. For a vertex subset $X \subseteq V(T)$, we define $p(X) = \sum_{u \in X} p(u)$ and $w(X) = \sum_{u \in X} w(u)$. Let $C \in \mathbb{R}_{\geq 0}$ be the capacity. Then, the task is to solve the following optimization problem:

$$\text{maximize} \;\; p(X) \;\; \text{subject to} \;\; w(X) \leq C, \quad X \in \mathcal{F}(\mathcal{A}), \tag{1}$$

This is a quite general problem since any constraint on a tree specified by a monadic second-order logic formula is represented by a tree automaton [18]. For example, the precedence constrained problem [14], the connectivity constrained problem [8], and the independent set constrained problem [16] are particular cases of this problem (See Examples 1, 2, and 3).

As in the case of the standard knapsack problem, the automaton constrained tree knapsack problem can be solved by DP. If the tree automaton has a *polynomially bounded diversity of transitions* (see Sect. 2.1), the complexity of the algorithm is $O(\text{poly}(n)C^2)$ time if the weights are integers, and $O(\text{poly}(n)P^2)$ time if the profits are integers, where P is an upper bound of the optimal value (see Sect. 2.2). Several existing studies have considered particular cases of the problem and derived the corresponding realization of this algorithm [8,14,16].

In this study, we focus on the *pseudo-polynomial factors* C or P in the complexity. The quadratic pseudo-polynomial factors of the standard DP come from merging solutions to the subtrees, which is implemented by the max-plus (or min-plus) convolution, whose current best complexity is $O(N^2 \log\log N / \log^2 N)$, where N is the length of the arrays [2]. It is conjectured that the max-plus convolution requires $\Omega(N^{2-\delta})$ time for any $\delta > 0$ [1,2,6]. However, quadratic pseudo-polynomial factors are sometimes unacceptable. For example, in practice, we often encounter the case that C is polynomially greater than n (e.g., $n = 100$ and $C = 100,000$). In this case, quadratic pseudo-polynomial factors are not desirable. For another example, when we derive a FPTAS from the DP, we take $P \propto 1/\epsilon$; thus, a smaller degree in P implies a faster algorithm with the same accuracy. The purpose of this study is to derive algorithms for the problem that run in $O(\text{poly}(n)C)$ or $O(\text{poly}(n)P)$ time.

Thus far, the only studies that have addressed this issue are those on the precedence constrained knapsack problem. Johnson and Niemi [11] proposed a technique, called left-right DP, which runs in $O(nC)$ time. Cho and Shaw [4] proposed a variant of the left-right DP, called depth-first DP, which also runs in $O(nC)$ time. However, we do not know what kinds of constraints (other than the precedence constraint) admit algorithms with complexity that is linear in pseudo-polynomial factors.

1.2 Our Contribution

In this study, we introduce a new DP technique, called *heavy-light recursive dynamic programming (HLRecDP)*. This technique is motivated by Chekuri and Pal's recursive greedy algorithm for the s-t path constrained monotone submodular maximization problem [3] and its generalization to the logic constrained monotone submodular maximization problem [10]. It also generalizes the left-right DP and depth-first DP for precedence constrained problem to the automaton constrained problem. Formally, by using this technique, we obtain the following theorem. From now on, we denote the logarithm of base two by log.

Theorem 1. *Let $T = (V(T), E(T))$ be a tree with n vertices and $\mathcal{A}$ be a nondeterministic top-down tree automaton with the diversity of transitions $\delta(n)$. Let $p \in \mathbb{R}^V_{\geq 0}$, $w \in \mathbb{Z}^V_{\geq 0}$, and $C \in \mathbb{Z}_{\geq 0}$. Then, there is an algorithm for problem (1) that runs in $O(n^{\log(1+\delta(n))}C)$ time. In particular, if $\delta(n) = O(1)$, the algorithm runs in $O(poly(n)C)$ time.*[1]

This theorem gives a sufficient condition for admitting (pseudo-)polynomial time algorithms with linear pseudo-polynomial factors. By applying this theorem to the precedence constrained problem, we obtain $O(nC)$ time algorithm that is equivalent to the existing left-right DP [11] and depth-first DP [4] (Example 2).

We then consider the *k-subtree* version problem. Let $k = O(1)$ be an integer. Then, the problem is to find k disjoint subtrees of the given tree and a feasible solution in each subtree such that the total profit is maximized under the total budget constraint. For example, the k connected component constrained problem is the k-subtree version of the precedence constrained problem. By using the property of the algorithm of Theorem 1 and divide-and-conquer techniques, we show that this problem can be solved in almost the same time complexity as the original problem.

Theorem 2. *Suppose that $\mathcal{A}$ is a prefix-closed top-down tree automaton with the bounded diversity of transitions, and the automaton constrained tree knapsack problem with $\mathcal{A}$ can be solved in $f(n)$ time by Algorithm 1. Let $k = O(1)$. Then, there exists an algorithm for the corresponding k-subtree version problem that runs in the following complexity:*

- $k = 1$. *$O(f(n) \log n)$ if $f(n) = O(nC)$, and $O(f(n))$ time if $f(n) = O(n^e C)$ for some $e > 1$.*
- $k \geq 2$. *$O(f(n)(\log n)^{\log k})$ time if $f(n) = O(n^e C)$ for some $e > 1$; the hidden constant is a polynomial in k.*

This theorem gives an $O(n \log nC)$ time algorithm for the connectivity constrained problem, and an $O(n^e C)$ time algorithm for any $e > 1$ for the k connected component constrained tree knapsack problem.

[1] For simplicity, we only consider the case in which the weights are integers. The same result is obtained when the profits are integers.

Organization of the Paper

The paper is organized as follows. In Sect. 2.1, we introduce top-down tree automata. In Sect. 2.2, we introduce the standard DP using a top-down tree automaton. In Sect. 3, we prove Theorem 1 by introducing the HLRecDP. In Sect. 4, we prove Theorem 2 using the divide-conquer technique with HLRecDP.

2 Preliminaries

2.1 Tree Automaton

A *non-deterministic top-down tree automaton ("automaton" for short)* [5] is a tuple $\mathcal{A} = (Q, \Sigma, Q_{\mathrm{init}}, \Delta)$, where Q is the set of states, Σ is a set of alphabets, $Q_{\mathrm{init}} \subseteq Q$ is the set of initial states, and Δ is a set of rewriting rules of the form

$$Q \times \Sigma \ni (q, \sigma) \mapsto (q_1, \ldots, q_d) \in Q \times \cdots \times Q. \tag{2}$$

We assume that the number of states of the automaton is constant, $|Q| = O(1)$. The automaton is *prefix-closed* if $(q, \sigma) \mapsto (q_1, \ldots, q_d)$ is in Δ then $(q, \sigma) \mapsto (q_1, \ldots, q_{d-1})$ also in Δ.

The *run* of the automaton is defined as follows. Let $T = (V(T), E(T))$ be a rooted tree, and $\sigma : V(T) \to \Sigma$ be labels on the vertices. The automaton first assigns an initial state $q \in Q_{\mathrm{init}}$ to the root of the tree. Then it processes the tree from the top (root) to the bottom (leaves). If vertex $u \in V(T)$ has state $q \in Q$, we choose a rewriting rule $(q, \sigma(u)) \mapsto (q_1, \ldots, q_d)$ and assign the states $q_1, \ldots, q_d$ to the children $v_1, \ldots, v_d \in V(T)$ of u, respectively. Note that, if no rule is applicable to u and q, the run fails. The automaton accepts a labeled tree if there is at least one run from the root to the leaves in which the state of the root is in Q_{init}.

To represent a substructure of a tree using an automaton, we choose the alphabet $\Sigma = \{0, 1\}$ and identify the subgraph $X \subseteq V(T)$ as the labels $\sigma_X : V(T) \to \Sigma$ such that $\sigma_X(u) = 1$ for $u \in X$ and $\sigma_X(u) = 0$ for $u \notin X$. Then, the family of subsets $\mathcal{F}(\mathcal{A}) \subseteq 2^{V(T)}$ represented by this automaton is specified by

$$\mathcal{F}(\mathcal{A}) = \{X \subseteq V(T) : \mathcal{A} \,\text{accepts}\, T \,\text{with label}\, \sigma_X\}. \tag{3}$$

To evaluate the complexity of DP, we introduce the following quantity $\delta(n)$, called the *diversity of transitions*.

$$\delta(n) = \max_{m \le n} \Big| \bigcup_{\text{“}(q,\sigma) \mapsto (q_1, \ldots, q_m)\text{”} \in \Delta} \{(q_1, \ldots, q_m)\} \Big|. \tag{4}$$

By definition, $\delta(n)$ is monotone in n. Intuitively, $\delta(n)$ is the maximum number of subproblems in DP; see Sect. 2.2 below. There is an automaton with exponentially large diversity of transitions, i.e., $\delta(n) = \Theta(|Q|^n)$, and in such case, it looks impossible to obtain $O(\mathrm{poly}(n))$ time algorithm. Therefore, we assume some boundedness of $\delta(n)$. Note that, even $\delta(n) = O(1)$, we can represent some interesting examples, such as independent set constraint (Example 1) and precedence constraint (Example 2).

2.2 Quadratic Pseudo-Polynomial Factor Algorithm

Here, we introduce the standard DP that solves the problem in $O(\mathrm{poly}(n)C^2)$ time if the automaton has a polynomially bounded diversity of transitions [8,14,16]. We regard this as a baseline algorithm for the problem.

Let $T = (V(T), E(T))$ be a rooted tree. We denote by T_u the subtree of T rooted by $u \in V(T)$. The algorithm computes array $x_{u,q}$ of length $C+1$ for each $u \in V(T)$ and $q \in Q$, such that

$$x_{u,q}[c] = \max\{p(X) : X \subseteq V(T_u), w(X) = c, \text{ subtree } T_u \text{ with labels } \sigma_X \text{ is accepted by } \mathcal{A}, \text{ where the initial state is } q\}. \tag{5}$$

Once the array for the root vertex $r \in V(T)$ is obtained, the optimal value is computed by $\max_{q \in Q_{\mathrm{init}}, c \in \{0,\dots,C\}} x_{r,q}[c]$ in $O(|Q_{\mathrm{init}}|C) = O(C)$ time.

We compute these arrays using the bottom-up DP as follows. For each leaf, the array is immediately computed in $O(\delta(0)C) = O(C)$ time. Consider a vertex $u \in V(T)$ with children $v_1, \dots, v_d \in V(T)$, such that the arrays $x_{v,q}$ are computed for all $v \in \{v_1, \dots, v_d\}$ and $q \in Q$. Then,

$$x_{u,q}[c] = \max\{x_{v_1,q_1}[c_1] + \cdots + x_{v_d,q_d}[c_d] + w(u)\sigma : (q,\sigma) \mapsto (q_1, \dots, q_d) \in \Delta, c_1 + \cdots + c_d + w(u)\sigma = c\}. \tag{6}$$

Here, we identify symbol $\sigma =$ "0" and "1" as integer 0 and 1, respectively. The maximization with respect to $c_1, \dots, c_d$ is evaluated by the max-plus convolution; thus, it costs about $O(nC^2)$ time. For the maximization with respect to $(q,\sigma) \to (q_1, \dots, q_d) \in \Delta$, we only have to evaluate the formula for distinct $(q_1, \dots, q_d)$. Therefore, the complexity of evaluating (5) is $O(n\delta(n)C^2)$ time, and the total complexity is $O(n^2\delta(n)C^2) = O(\mathrm{poly}(n)C^2)$.

3 Heavy-Light Recursive Dynamic Programming

In this section, we present the HLRecDP for obtaining an $O(n^{\log(1+\delta(n))}C)$ time algorithm. In Sect. 3.1, we first propose the recursive dynamic programming (RecDP) technique for balanced trees. To handle non-balanced trees, in Sect. 3.2, we combine the heavy-light decomposition to the RecDP.

3.1 Recursive Dynamic Programming for Balanced Trees

Our goal is to compute arrays $\{x_{r,q}\}_{q \in Q_{\mathrm{init}}}$ for the root $r \in V(T)$ of the tree, where $x_{r,q}$ is defined in (5). To avoid quadratic pseudo-polynomial factors, we call the recursive procedure for the children multiple times, instead of merging subtree solutions.

Formally, we design procedure $\textsc{RecDP}(u, q, a)$, where $u \in V(T)$, $q \in Q$, and a is an array of size $C+1$. It computes array $y_{u,q,a}$ defined by

$$y_{u,q,a}[c] = \max\{p(X) + a[c'] : X \subseteq V(T_u), w(X) + c' = c, \text{ subtree } T_u \text{ with labels } \sigma_X \text{ is accepted by } \mathcal{A}, \text{ where the initial state is } q\}. \tag{7}$$

The difference between (5) and (7) is that (7) contains the array parameter a, which corresponds to the "initial values" of the DP. More intuitively, it returns an array that is obtained by "adding" items in the subtree T_u optimally to the current solution represented by a. By calling $\textsc{RecDP}(r, q, [0, -\infty, \ldots, -\infty])$, where $r \in V(T)$ is the root of the tree and $q \in Q_{\mathrm{init}}$, we obtain the desired solution $x_{r,q}$.

Here, $\textsc{RecDP}(u, q, a)$ is implemented as follows. If $u \in V(T)$ is a leaf, we can compute (7) in $O(C)$ time. Consider a vertex $u \in V(T)$ that has children $v_1, \ldots, v_d \in V(T)$. For each rewriting rule $(q, \sigma) \mapsto (q_1, \ldots, q_d)$, we first call $\textsc{RecDP}(v_1, q_1, a)$ to obtain array $y_1 = y_{v_1,q_1,a}$. Then, we call $\textsc{RecDP}(v_2, q_2, y_1)$ to obtain array $y_2 = y_{v_2,q_2,y_1}$, i.e., we use the returned array y_1 as the initial values of the DP to the subtree rooted by v_2. By iterating this process to the last child, we obtain array $y_d = y_{v_d,q_d,y_{d-1}}$. The solution corresponds to this rewriting rule is then obtained by

$$z_{u,q,a}[c] = y_d[c - \sigma w(u)] + \sigma p(u), \quad c \in \{0, \ldots, C\}. \tag{8}$$

By taking the entry-wise maximum of the solutions on all of the rewriting rules, we obtain the solution to $\textsc{RecDP}(u, q, a)$.

The correctness of the above procedure is easily checked. We evaluate the time complexity. Let $f(n)$ be the complexity of the procedure. Let $n_1, \ldots, n_d$ be the number of vertices on the subtrees rooted by $v_1, \ldots, v_d$. Then we have $n_1 + \cdots + n_d = n - 1$. Because the algorithm calls the procedure recursively to each subtree at most $\delta(n)$ times, the complexity satisfies[2]

$$f(n) \leq \delta(n)(f(n_1) + \cdots + f(n_d)) + O(C). \tag{9}$$

If the tree is balanced, i.e., $n_j \leq n/2$ for all $j = 1, \ldots, d$, this already provides the desired complexity: Without loss of generality, we can assume that $f(n)$ is convex in n. Then, the maximum of the right-hand side is attained at $n_1 = \lceil (n-1)/2 \rceil$, $n_2 = \lfloor (n-1)/2 \rfloor$, and $n_3 = \cdots = n_d = 0$. Therefore,

$$f(n) \leq \delta(n)\left(f(\lceil (n-1)/2 \rceil) + f(\lfloor (n-1)/2 \rfloor)\right) + O(C). \tag{10}$$

By solving this inequality, we have $f(n) = O((2\delta(n))^{\log n} C) = O(n^{1+\log \delta(n)} C)$.

3.2 Heavy-Light Recursive Dynamic Programming

To obtain an $O(\mathrm{pseudopoly}(n)C)$ time algorithm for general (i.e., non-balanced) trees, we have to make the depth of the recursion to $O(\log n)$. The HLRecDP achieves this by using the *heavy-light decomposition* [17].

First, we introduce the heavy-light decomposition. Let $T = (V(T), E(T))$ be a rooted tree whose edges are directed toward the leaves. An edge $(u, v) \in E(T)$ is a *heavy edge* if v has more descendants than other children of u do

[2] The additive term is naturally $O(dC)$; however, it is separated and included in the recursive terms.

(ties are broken arbitrary). An edge is a *light edge* if it is not a heavy edge[3]. $v \in V(T)$ is a *heavy child* of $u \in V(T)$ if (u, v) is a heavy edge. A *light child* is defined similarly. A subtree rooted by a light child is referred to as a *light subtree*. The set of heavy edges forms disjoint paths, called *heavy paths*. The tree is decomposed into the heavy paths, which is referred to as a *heavy-light decomposition*. The heavy-light decomposition is computed in linear time by depth-first search. The most important property of a heavy-light decomposition is that for each $u \in V(T)$, the number of descendants of a light child is at most $|V(T_u)|/2$.

Recall algorithm $\textsc{RecDP}(u, q, a)$ defined in the previous section. We observe that all recursive calls of $\textsc{RecDP}(v_1, q_1, a)$ to the first child has the same initial array a for different q_1. Thus, we can "gather" all recursive calls for the first child into a single recursive call. The HLRecDP sets the heavy child as the first child to avoid an excessive number of recursive calls this child.

Formally, we define procedure $\textsc{HLRecDP}(u, a)$. This returns a set of arrays $\{y_{u,q}\}_{q \in Q}$, where $y_{u,q}$ is defined in (7). For $v_1, \ldots, v_d \in V(T)$ and $q_1, \ldots, q_d \in Q$, we define $\textsc{HLRecDP}(v_1, \ldots, v_d, a)_{q_1,\ldots,q_d}$ as a shorthand notation of the sequential evaluation

$$\textsc{HLRecDP}(v_d, \textsc{HLRecDP}(v_{d-1} \cdots \textsc{HLRecDP}(v_1, a)_{q_1} \cdots)_{q_{d-1}})_{q_d}. \tag{11}$$

Now we describe the procedure. Let $v_1, \ldots, v_d \in V(T)$ be the children of u, where v_1 is the heavy child. First, we call $\textsc{HLRecDP}(v_1, a)$ and store the resulting arrays for all $q \in Q$. Then, for each rewriting rule $(q, \sigma) \mapsto (q_1, \ldots, q_d)$, we call $\textsc{HLRecDP}(v_2, \ldots, v_d, \textsc{HLRecDP}(v_1, x)_{q_1})_{q_2,\ldots,q_d}$ and add item u if $\sigma = 1$ to obtain the solution to the rewriting rule. By taking the entry-wise maximum over the rewriting rules, we obtain the desired solution; see Algorithm 1.

By construction, HLRecDP gives the same solution as RecDP; thus, it correctly solves the problem. We evaluate the complexity as follows. Let $n_1, \ldots, n_d$ be the number of vertices on the subtrees rooted by $v_1, \ldots, v_d$. As same as the analysis of RecDP, the complexity $f(n)$ of the algorithm satisfies

$$f(n) \leq f(n_1) + \delta(n)\,(f(n_2) + \cdots + f(n_d)) + O(C). \tag{12}$$

By the convexity of $f(n)$ and the heavy-light property, i.e., $n_j \leq n/2$ $(j = 2, \ldots, d)$, the maximum of the right-hand side is attained at $n_1 = \lceil (n-1)/2 \rceil$, $n_2 = \lfloor (n-1)/2 \rfloor$, and $n_3 = \cdots = n_d = 0$. Thus, we have

$$f(n) \leq f(\lceil (n-1)/2 \rceil) + \delta(n)) f(\lfloor (n-1)/2 \rfloor) + O(C). \tag{13}$$

By solving this inequality, we have $f(n) = O(n^{\log(1+\delta(n))} C)$. □

[3] Our definition of the heavy edge is slightly different to the original one: In [17], (u, v) is said to be "heavy" if $2 \times \mathrm{size}(v) > \mathrm{size}(u)$, where $\mathrm{size}(v)$ is the number of descendants of v. Thus, their heavy edge is always our heavy edge, but the converse is not. In particular, in their definition, any internal vertex has *at most* one heavy edge, but in our definition, any internal vertex has *exactly* one heavy edge.

Algorithm 1. Heavy-Light Recursive Dynamic Programming

procedure HLRecDP(u, a)
  $y_{u,q}[c] = -\infty$ for all $c \in \{0, \dots, C\}$, $q \in Q$
  Let $v_1, \dots, v_d$ be the children of u, where v_1 is the heavy child
  Call HLRecDP(v_1, a) and store the arrays for $q \in Q$
  for "$(q, \sigma) \mapsto (q_1, \dots, q_d)$" $\in \Delta$ **do**
    Let $z = \text{HLRecDP}(v_2, \dots, v_d, \text{HLRecDP}(v_1, a)_{q_1})_{q_2,\dots,q_d}$
    for $c = 0, \dots, C$ **do**
      $y_{u,q}[c] \leftarrow \max\{y_{u,q}[c], z[c - \sigma w(u)] + \sigma p(u)\}$
    end for
  end for
  return $\{y_{u,q}\}_{q \in Q}$
end procedure

Remark 1. There is a gap of the tractable classes between the standard DP (Sect. 2.2) and the HLRecDP. The analysis in Sect. 2.2 implies that we can obtain $O(\text{poly}(n)C^2)$ time algorithm if $\delta(n)$ is polynomially bounded. On the other hand, the analysis in this section implies that if $\delta(n)$ is polynomially bounded (rather than bounded by a constant), we can only obtain an algorithm with quasi-polynomial time complexity, i.e., $n^{O(\log n)}C$.

Here, we derive several results for particular cases using our method.

Example 1 (Independent Set Constrained Problem). Let us consider the *independent set constrained* tree knapsack problem whose feasible set contains no adjacent vertices. This constraint is represented by an automaton $\mathcal{A} = (Q, \Sigma, Q_{\text{init}}, \Delta)$, where $Q = Q_{\text{init}} = \{\text{s}, \text{x}\}$ and

$$(\text{s}, 0) \mapsto (\text{s}, \dots, \text{s}), \quad (\text{s}, 1) \mapsto (\text{x}, \dots, \text{x}), \quad (\text{x}, 0) \mapsto (\text{s}, \dots, \text{s}). \tag{14}$$

Here, s means the vertex can be selected and x means the vertex cannot be selected. The diversity of transitions is $\delta(n) = 2$ because the rules for $(\text{s}, 0)$ and $(\text{x}, 0)$ have the same right-hand side; therefore, we can solve the independent set constrained tree knapsack problem in $O(n^{\log(1+\delta(n))}C) = O(n^{\log 3}C) = O(n^{1.585}C)$ time.

Example 2 (Precedence Constrained Problem). Let us consider the *precedence constrained* tree knapsack problem whose feasible set is precedence closed, i.e., if a vertex is contained in a solution, all the precedences are also contained in the solution. This constraint is represented by an automaton $\mathcal{A} = (Q, \Sigma, Q_{\text{init}}, \Delta)$, where $Q = Q_{\text{init}} = \{\text{s}, \text{x}\}$ and

$$(\text{s}, 0) \mapsto (\text{x}, \dots, \text{x}), \quad (\text{s}, 1) \mapsto (\text{s}, \dots, \text{s}), \quad (\text{x}, 0) \mapsto (\text{x}, \dots, \text{x}). \tag{15}$$

Here, state s means the vertex can be selected and state x means the vertex cannot be selected. Since the diversity of transitions is $\delta(n) = 2$, the algorithm runs in $O(n^{\log(1+\delta(n))}C) = O(n^{1.585}C)$ time.

This complexity can be improved further. If a vertex has state x, we cannot select all of the descendants of the vertex; thus we obtain the solution for this case without calling the procedure recursively. Thus, the required number of recursive calls is at most one, which is for $(\mathrm{s}, 1)$. Therefore, the algorithm runs in $O(n^{\log(1+1)}C) = O(nC)$ time. Note that this algorithm "coincides" with the left-right DP [11] and the depth-first DP [4] in the sense that these perform the same manipulations.

Example 3 (Connectivity Constrained Problem). Let us consider the *connectivity constrained* tree knapsack problem whose feasible set forms a connected subgraph of a given tree. This constraint is represented by an automaton $\mathcal{A} = (Q, \Sigma, Q_{\mathrm{init}}, \Delta)$, such that $Q = \{\mathrm{s}, \mathrm{o}, \mathrm{x}\}$, $Q_{\mathrm{init}} = \{\mathrm{s}\}$ and

$$\begin{aligned}
&(\mathrm{s}, 0) \mapsto (\mathrm{s}, \mathrm{x}, \ldots, \mathrm{x}), \quad (\mathrm{s}, 0) \mapsto (\mathrm{x}, \mathrm{s}, \ldots, \mathrm{x}), \quad \ldots, \quad (\mathrm{s}, 0) \mapsto (\mathrm{x}, \mathrm{x}, \ldots, s),\\
&(\mathrm{s}, 1) \mapsto (\mathrm{o}, \mathrm{o}, \ldots, \mathrm{o}), \quad (\mathrm{o}, 0) \mapsto (\mathrm{x}, \mathrm{x}, \ldots, \mathrm{x}), \quad (\mathrm{o}, 1) \mapsto (\mathrm{o}, \mathrm{o}, \ldots, \mathrm{o}),\\
&(\mathrm{x}, 0) \mapsto (\mathrm{x}, \mathrm{x}, \ldots, \mathrm{x}). \qquad (16)
\end{aligned}$$

Here, state s means the vertex can be selected, state o means the vertex is now selecting, and state x means that the vertex cannot be selected. Note that $\mathcal{A}$ is non-deterministic because there are d rules for $(\mathrm{s}, 0)$. Thus, the diversity of transitions is $\delta(n) = n$, which is not bounded by a constant. Thus, the theorem gives only quasi-polynomial time algorithm.

To improve the performance, we make the similar observation to the precedence constraint (Example 2). Then, the number of recursive calls to each subtree is at most twice; one is for $(\mathrm{s}, 0)$ and the other is for $(\mathrm{s}, 1)$ and $(\mathrm{o}, 1)$. Therefore, the algorithm runs in $O(n^{\log(1+2)}C) = O(n^{1.585}C)$ time.

Example 4 (k Connected Component Constrained Problem). Let us consider k *connected component constrained* tree knapsack problem whose feasible solution is k connected components. By using the same technique as the connectivity constrained problem (Example 3), we obtain $n^{O(\log k)}C$ time algorithm for the problem. Note that, if we handle k as a kind of weight, we can derive $O(kn^eC) = O(n^eC)$ time algorithm for some universal constant e.

4 k-Subtree Version Problems

In this section, we consider the k-subtree version problems and prove Theorem 2. We introduce two auxiliary problems: The first one is the *for-all-subtree* problem that requires to solve the problem on each subtree T_u of T rooted by $u \in V(T)$. The second one is the *for-all-subtree-complement* problem that requires to solve the problem on each subtree-complement $T \setminus T_u$ of T for all $u \in V(T)$. These problems can be solved in almost the same time complexity as follows.

Lemma 1. *Suppose that the automaton constrained tree knapsack problem with tree automaton $\mathcal{A}$ can be solved in $f(n) = O(n^eC)$ time by Algorithm 1. Then, the corresponding for-all-subtree version problem can be solved in $O(f(n)\log n)$ time if $e = 1$ and $O(f(n))$ time if $e > 1$.*

Proof. Let us fix a heavy path $u_1, \ldots, u_l$ that starts from the root of the tree, i.e., u_1 is the root and u_l is a leaf. First, we call HLRecDP $(u_1, [0, -\infty, \ldots, -\infty])$ to the root u_1 of the tree. Then, it recursively calls HLRecDP$(u_i, [0, -\infty, \ldots, -\infty])$ to the vertices $u_2, \ldots, u_l$ on the heavy path. This means that this single call gives the solutions to the subtrees rooted by the vertices on the heavy path.

After this computation, we call the procedure recursively to the light subtrees adjacent to the heavy path. The total complexity $g(n)$ satisfies

$$g(n) \leq g(n_1) + \cdots + g(n_s) + f(n), \tag{17}$$

where $n_1, \ldots, n_s$ are the sizes of the subtrees. By definition, $n_1 + \cdots + n_s \leq n - 1$. Also, by the heavy-light property, $n_j \leq n/2$ $(j = 1, \ldots, s)$. Therefore, the maximum of the right-hand side is attained at $n_1 = \lceil (n-1)/2 \rceil$, $n_2 = \lfloor (n-1)/2 \rfloor$, and $n_3 = \cdots = n_s = 0$. Thus,

$$g(n) \leq g(\lceil (n-1)/2 \rceil) + g(\lfloor (n-1)/2 \rfloor) + f(n), \tag{18}$$

By solving this inequality we obtain the desired result. □

Lemma 2. *Suppose that the automaton constrained tree knapsack problem with tree automaton $\mathcal{A}$ can be solved in $f(n) = O(n^e C)$ time by Algorithm 1. Then, the corresponding for-all-subtree-complement version problem can be solved in $O(f(n)(\log n)^2)$ time if $e = 1$ and $O(f(n) \log n)$ time if $e > 1$.*

Proof. For vertex $u \in V(T)$, we define array $x_{u,q}$ of length $C+1$ that represents the solution on $T \setminus T_u$, where the parent of u has state $q \in Q$. We compute the arrays for all the vertices. We define $x_{r,q} = [0, -\infty, \ldots, -\infty]$ for the root $r \in V(T)$ and all $q \in Q$. Let us fix a heavy path $u_1, \ldots, u_l$ that starts from the root of the tree. We compute the arrays for the vertices on the heavy path, and for the vertices adjacent to the heavy path separately.

Vertices on the heavy path. Suppose that we have $\{x_{u_{i-1},q}\}_{q \in Q}$. Let $v_1, \ldots, v_d$ be children of u_{i-1}, where $v_1 = u_i$. For each rewriting rule $(q, \sigma) \mapsto (q_1, \ldots, q_{d-1}) \in \Delta$, which is a rule of length $d-1$, which will match to $v_2, \ldots, v_d$, the array corresponds to this rule is obtained by calling HLRecDP$(v_2, \ldots, v_d, x_{u_{i-1},q})_{q_1,\ldots,q_{d-1}}$ and by adding u_i if $\sigma = 1$. By taking the entry-wise maximum of the arrays for different rules, we obtain $\{x_{u_i,q}\}_{q \in Q}$. Since this computation process pays the same computational effort as HLRecDP(u_1, x), the complexity is $f(n)$.

Vertices adjacent to the heavy path. We compute $\{x_{v,q}\}_{q \in Q}$ for all light child v adjacent to the heavy path. It is obtained by calling HLRecDP to all the subtrees except T_v; however, this method involves redundant computations. We reduce the complexity by storing intermediate results by a segment tree-like divide-and-conquer technique.

First, we compute arrays y_{i,j,q_i,q_j} for $i = 0, \ldots, l-1$, $j = i+1, \ldots, l$, and $q_i, q_j \in Q$. This stores the vector obtained by calling HLRecDP with initial

array $x_{u_1,q}$ for some q to the subtree except the light children of $u_{i+1}, \ldots, u_j$, where the states of u_i and u_i are q_i and q_j, respectively. Initially, we set $y_{0,l,q_0,q_l} = x_{u_1,q_0}$ for all $q_0, q_l \in Q$. If we have $\{y_{i,j,q_i,q_j}\}_{q_i,q_j \in Q}$ for $i+1 < j$, we can compute $\{y_{i,m,q_i,q_m}\}_{q_i,q_m \in Q}$ where $m = \lfloor (i+j)/2 \rfloor$ by calling HLRecDP to the light subtrees of $u_{m+1}, \ldots, u_j$ with initial array z_{i,j,q_i,q_j}. Similarly, we can compute $\{y_{m,j,q_m,q_r}\}_{q_m,q_r \in Q}$. The complexity of computing all the arrays is $O(f(n) \log n)$ since HLRecDP is called to subtree T_{u_i} at most $O(\log n)$ times.

Next, for each u_k ($k = 1, \ldots, l-1$) on the heavy path, we consider the children $v_1, \ldots, v_d$ of u_k, where v_1 is the heavy child (i.e., $v_1 = u_{k+1}$). For each rewriting rule $(q, \sigma) \mapsto (q_1, \ldots, q_{d-1}) \in \Delta$, we compute arrays z_{i,j,q,q_1} for $i = 1, \ldots, d-1$ and $j = i+1, \ldots, d$. This stores the vector obtained by calling HLRecDP with initial array y_{k-1,k,q,q_1} to the subtrees except $v_{i+1}, \ldots, v_j$, and is computed by the same technique as y. Once the arrays are obtained, we can retrieve $x_{v_i,q}$ by taking the entry-wise maximum of z_{i-1,i,q,q_1} with respect to q_1. Thus, the total complexity of this part is $O(f(n) \log n)$.

After this computation, we call the procedure recursively to the light subtrees adjacent to the heavy path. The total complexity $g(n)$ satisfies

$$g(n) \leq g(n_1) + \cdots + g(n_s) + O(f(n) \log n), \tag{19}$$

where $n_1 + \cdots + n_s \leq n-1$ and $n_j \leq n/2$ ($j = 1, \ldots, s$). By solving this inequality as similar to Lemma 1, we obtain the desired result. □

Now we provide an outline of the proof of Theorem 2.

Proof (of Theorem 2, outline). We design algorithm $\textsc{Conn}(u, k, x)$ that computes arrays $x_{u,q,b,l}$ where $u \in V(T)$, $q \in Q$, $b \in \{0,1\}$, and $l \in \{0, \ldots, k\}$. The array represents the solution to the subtree T_u such that the root ($= u$) has state q and is included by a subtree if $b = 1$, and l subtrees are selected. If $k = 0$, the solution is $[0, -\infty, \ldots, -\infty]$. If $k = 1$, we can solve the problem by solving for-all-subtree version problem since the automaton is prefix closed. Thus, in the following, we consider $k \geq 2$. Let $g(n, k)$ be the complexity of the algorithm for n vertices with parameter k. We derive the recursive relation of g. We fix a heavy path, and consider light subtrees adjacent to the heavy path.

Case 1: There is a light subtree T_v that contains at least $k/2$ components. In this case, the subtree complement $T_u \setminus T_v$ contains at most $k/2$ components. Thus, we guess such subtree T_v and solve the problem on $T_u \setminus T_v$ and T_v separately. We can solve all the subtree complements simultaneously by calling the subtree-complement version of $\textsc{Conn}(u, k/2, *)$. Also, we can solve each subtree by calling $\textsc{Conn}(v, k, *)$. The complexity of this approach is $g(n_1, k) + \cdots + g(n_s, k) + O(g(n, k/2) \log n)$.

Case 2: Otherwise; i.e., all the light subtrees contain at most $k/2$ components. We call $\textsc{Conn}(v, k/2, *)$ for all subtrees v, sequentially. The complexity of this part is given by $g(n_1, k/2) + \cdots + g(n_s, k/2) \leq g(n, k/2)$.

The total complexity $g(n,k)$ of the algorithm satisfies

$$g(n,k) \leq g(n_1,k) + \cdots + g(n_s,k) + O(g(n,k/2)\log n). \tag{20}$$

By using $n_1 + \cdots n_s \leq n-1$ and $n_j \leq n/2$ $(j = 1,\ldots,s)$, we obtain $g(n,k) \leq h(k)f(n)(\log n)^{\log k}$, where $h(k)$ is a polynomial in k. □

Example 5 (Connected Component Constrained Problem (again)). By using this technique, the connectivity constrained problem can be solved in $O(n \log nC)$ time, and the k connected component constrained problem can be solved in $O(n^{1+e}C)$ time for any $e > 0$, since $(\log n)^k = O(n^e)$ for any $e > 0$.

Acknowledgment. We thank the anonymous reviewers for their helpful comments.

References

1. Backurs, A., Indyk, P., Schmidt, L.: Better approximations for tree Sparsity in nearly-linear time. In: Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2017), pp. 2215–2229 (2017)
2. Bremner, D., et al.: Necklaces, convolutions, and $X + Y$. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 160–171. Springer, Heidelberg (2006). https://doi.org/10.1007/11841036_17
3. Chekuri, C., Pál, M.: A recursive greedy algorithm for walks in directed graphs. In: Proceedings of the 46th Annual Symposium on Foundations of Computer Science (FOCS 2005), vol. 2005, pp. 245–253 (2005)
4. Cho, G., Shaw, D.X.: A depth-first dynamic programming algorithm for the tree knapsack problem. J. Comput. **9**(4), 431–438 (1997)
5. Comon, H., et al.: Tree automata techniques and applications (2007)
6. Cygan, M., Mucha, M., Węgrzycki, K., Włodarczyk, M.: On problems equivalent to (min,+)-convolution. arXiv:1702.07669 (2017)
7. Hirao, T., Yoshida, Y., Nishino, M., Yasuda, N., Nagata, M.: Single-document summarization as a tree knapsack problem. In: Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing (EMNLP 2013), pp. 1515–1520 (2013)
8. Hochbaum, D.S., Pathria, A.: Node-optimal connected k-subgraphs. University of California, Berkeley, Technical report (1994)
9. Ibarra, O., Kim, C.: Fast approximation algorithms for the knapsack and sum of subset problems. J. ACM **22**(4), 463–468 (1975)
10. Ishihata, M., Maehara, T., Rigaux, T.: Algorithmic meta-theorems for monotone submodular maximization. arXiv:1807.04575 (2018)
11. Johnson, D.S., Niemi, K.: On knapsacks, partitions, and a new dynamic programming technique for trees. Mathe. Oper. Res. **8**(1), 1–14 (1983)
12. Kellerer, H., Pferschy, U., Pisinger, D.: Knapsack Problems. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-24777-7
13. Lawler, E.L.: Fast approximation algorithms for knapsack problems. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS 1977), vol. 4(4), pp. 339–357 (1977)
14. Lukes, J.A.: Efficient algorithm for the partitioning of trees. IBM J. Res. Dev. **18**(3), 217–224 (1974)

15. Van der Merwe, D., Hattingh, J.M.: Tree knapsack approaches for local access network design. Eur. J. Oper. Res. **174**(3), 1968–1978 (2006)
16. Pferschy, U., Schauer, J.: The knapsack problem with conflict graphs. J. Graph Algorithms Appl. **13**(2), 233–249 (2009)
17. Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. J. Comput. Syst. Sci. **26**(3), 362–391 (1983)
18. Thatcher, J.W., Wright, J.B.: Generalized finite automata theory with an application to a decision problem of second-order logic. Mathe. Syst. Theor. **2**(1), 57–81 (1968)