

A Fast and Correct Gather Algorithm

2026-03-14 · Updated 2026-03-20

#Botting #Math #polygen

This document formally describes an optimal algorithm to solve the Maximum Weight Bounded-Size Subtree problem. By leveraging the properties of the Depth-First Search (DFS) sequence (preorder traversal), the algorithm flattens the tree into a 1D array. This transformation reduces the traditional complex tree-based backpack merging process into a standard sequence Dynamic Programming (DP) algorithm, achieving a deterministic $O(|V| \cdot N)$ time complexity.

Definition 1 (Maximum Weight Bounded-Size Subtree)

Let $T = (V, E)$ be a rooted tree with a designated root node $r \in V$. Each node $v \in V$ is assigned a real-valued weight $w(v) \in \mathbb{R}$. Given a capacity integer N ($1 \leq N \leq |V|$), the objective is to find a subset of vertices $S \subseteq V$ such that:

- $r \in S$.
- The subgraph induced by S is connected in T .
- $|S| \leq N$.

The goal is to maximize the total weight of the selected subset:

$$W(S) = \sum_{v \in S} w(v).$$

Note that the connectivity constraint, combined with the requirement that $r \in S$, implies a strict topological dependency: for any node $v \in S \setminus \{r\}$, its parent $p(v)$ must also be in S .

The algorithm transforms the tree structure into a one-dimensional array using a Preorder Traversal (DFS sequence). Let $|V| = M$.

Let $v_1, v_2, \dots, v_M$ be the sequence of vertices in the order they are visited during a DFS preorder traversal, where $v_1 = r$. Let $\text{sz}(v)$ denote the total number of nodes in the subtree rooted at v .

We define a 2D dynamic programming table D . Let $D_{i,j}$ be the maximum weight obtained by selecting exactly j nodes from the suffix of the DFS sequence $v_i, v_{i+1}, \dots, v_M$, assuming that the dependency path from the root down to the parent of v_i has already been selected.

For a given index $i \in [2, M]$ and size $j \in [1, N]$, we have two choices regarding the current node $u = v_i$:

- Include u : We gain its weight $w(u)$, consume 1 unit of capacity, and proceed to evaluate the very next node in the DFS sequence v_{i+1} . Thus, the maximal weight is $D_{i+1,j-1} + w(u)$.
- Exclude u : If we do not select u , the connectivity dependency dictates that we cannot select any node in the subtree of u . Since the subtree of u occupies exactly the contiguous block from index i to $i + \text{sz}(u) - 1$ in the preorder sequence, we must skip this entire block. The next available valid node is precisely at index $i + \text{sz}(u)$. Thus, the maximal weight is $D_{i+\text{sz}(u),j}$.

Combining these, the optimal sub-structure is given by the Bellman equation:

$$D_{i,j} = \max\{D_{i+1,j-1} + w(v_i), D_{i+\text{sz}(v_i),j}\}$$

By definition, the root $r = v_1$ *must* be included. Therefore, we do not have the option to exclude it:

$$D_{1,j} = D_{2,j-1} + w(v_1) \quad \text{for } 1 \leq j \leq N$$

For any index $i \in [1, M + 1]$, choosing $j = 0$ nodes yields a weight of 0: $D_{i,0} = 0$. For any conceptually invalid state, the table is initialized to $-\infty$.

Theorem 2 (Correctness)

The D tables correctly compute the maximum weight subtree satisfying the connectivity and capacity constraints.

Proof. By the properties of DFS preorder traversal, a parent node u always appears before any of its descendants. Furthermore, the descendants of u form a contiguous subarray $[i, i + \text{sz}(u) - 1]$ in the sequence.

The connectivity constraint dictates that if a node u is not selected, no node in its subtree can be selected. The transition explicitly models this: exclusion of node v_i forces the state transition to $i + \text{sz}(v_i)$, completely bypassing all descendants of v_i . Inclusion of v_i evaluates the contiguous node $i + 1$ (allowing the selection of its children). Since the root v_1 is forcibly included, the subgraph generated by this sequence of decisions is guaranteed to form a single connected component containing the root. By Bellman's Principle of Optimality, evaluating both choices ensures global topological optimization. $\square$

Theorem 3 (Complexity)

The proposed algorithm has a time complexity of $O(|V| \cdot N)$ and a space complexity of $O(|V| \cdot N)$.

Proof. The DFS traversal to generate arrays v_i and sz takes $O(|V|)$ time. The DP table has $(|V| + 2)$ rows and $(N + 1)$ columns. Each cell $D_{i,j}$ is computed in $O(1)$ time by comparing at most two previously computed values. Therefore, filling the table takes $O(|V| \cdot N)$ operations. Finding the maximum value among $D_{1,j}$ ($1 \leq j \leq N$) takes $O(N)$. Overall time complexity is $O(|V| \cdot N)$.

The algorithm strictly maintains an array D of size $(|V| + 2) \times (N + 1)$ and auxiliary arrays of size $O(|V|)$. Thus, the memory footprint is bounded by $O(|V| \cdot N)$. $\square$