2025 Algorithm hw2

109062327 Hsu, Hung-Che

April 2025

Problem 1: Maximum Spanning Tree

Let $G = (V, E)$ be a connected, undirected graph with edge weights and let $n = |V|$, $m = |E|$. A *maximum spanning tree* (Max-ST) is a spanning tree with maximum total weight. We now design a greedy algorithm to compute a Max-ST.

Algorithm Design

A simple method is to modify Kruskal's algorithm by sorting the edges in *non-increasing* order. The algorithm is as follows:

Algorithm: MaximumSpanningTree(G)

1: Sort the edges E in non-increasing order by weight.
2: Initialize a forest F where each vertex is a separate tree.
3: **for** each edge $e = (u, v)$ in the sorted order **do**
4: **if** the endpoints u and v are in different trees in F **then**
5: Add edge e to F (i.e., union the trees).
6: **end if**
7: **end for**
8: **return** the set of edges in F, which forms the maximum spanning tree.

Time Complexity Analysis

- Sorting the m edges takes $O(m \log m)$ time.
- The union-find data structure is used to check and merge trees. With path compression and union by rank, the total cost for m `find` and `union` operations is $O(m\,\alpha(n))$, where α is the inverse Ackermann function.

Thus, the overall time complexity is

$$O(m \log m + m\,\alpha(n)) = O(m \log m).$$

Proof of Correctness

We use a greedy-choice and cut property argument:

1. **Greedy Choice:** At each step, the algorithm selects the available edge with maximum weight that does not create a cycle. Suppose the selected edge is $e = (u, v)$. Consider the cut $(S, V \setminus S)$ where $u \in S$ and $v \in V \setminus S$. Since e is the heaviest edge across the cut (by the ordering), by the *cut property* it must belong to some maximum spanning tree.

2. **Safe Extension:** Adding such edges repeatedly preserves optimality, and when the forest becomes connected, the resulting tree is a spanning tree with maximum total weight.

Thus, the algorithm is correct.

Problem 2: Optimal 2-Way Merge Tree Property

Statement: There exists an optimal 2-way merge tree in which the two leaf nodes with the minimum sizes are siblings, and their parent is an internal node at maximum distance from the root.

Proof

We prove the property by an exchange argument (similar in spirit to the proof for Huffman coding):

Let T be an optimal 2-way merge tree for a set of files (or items) with sizes. Suppose in T, the two smallest files (say with sizes a and b, where $a \leq b$) are not siblings. Then, there exist two siblings x and y such that at least one of these files is not among the two smallest. We can exchange the positions of a (or b) with one of x or y, forming a new tree T'.

- Since merging smaller files first minimizes the accumulated cost (or sum of merge operations), the cost in T' is no greater than that in T.

- Moreover, by repeatedly applying such exchanges, we eventually obtain an optimal merge tree where the two smallest files are merged at the last possible merge (i.e., they are siblings at the deepest level).

Hence, there is an optimal merge tree in which the two leaves with minimum sizes are siblings, and their parent is as far from the root as possible. This property is crucial in proving the optimality of the greedy strategy for the 2-way merge problem.

Problem 3: Shortest Paths in Graphs with Negative Edge Costs

Let $G = (V, E)$ be an arbitrary directed, edge-weighted graph with negative cost edges but no negative cycles. We design an algorithm to compute the shortest path from a source vertex s to every other vertex.

Algorithm: Bellman-Ford

1: Initialize: for all $v \in V$, set $d(v) = \infty$, and set $d(s) = 0$.
2: **for** $i = 1$ **to** $|V| - 1$ **do**
3: **for** each edge $(u, v) \in E$ **do**
4: **if** $d(u) + w(u, v) < d(v)$ **then**
5: Set $d(v) = d(u) + w(u, v)$.
6: **end if**
7: **end for**
8: **end for**
9: **for** each edge $(u, v) \in E$ **do**
10: **if** $d(u) + w(u, v) < d(v)$ **then**
11: Report that a negative cycle exists (this case is not possible by assumption).
12: **end if**
13: **end for**
14: **return** $d(v)$ for all $v \in V$.

Time Complexity Analysis

The algorithm performs:

- $|V| - 1$ iterations, and in each iteration, all $m = |E|$ edges are relaxed.
- Hence, the running time is $O(|V| \cdot |E|)$.

Proof of Correctness

The Bellman-Ford algorithm is based on the following observation:

- Any shortest path from s to a vertex v contains at most $|V| - 1$ edges.
- By relaxing all edges $|V| - 1$ times, we ensure that the distance estimates converge to the actual shortest path distances.
- The final pass detects negative cycles; however, under the given assumption there are none.

Thus, the algorithm correctly computes the shortest paths from s.

Problem 4: Fractional Knapsack Problem

Given positive integers $P_1, P_2, \ldots, P_n$ (profits), $W_1, W_2, \ldots, W_n$ (weights), and a knapsack capacity M, we wish to select fractions $X_1, X_2, \ldots, X_n$ with $0 \leq X_i \leq 1$ such that the total profit

$$\sum_{i=1}^{n} P_i X_i$$

is maximized subject to the weight constraint

$$\sum_{i=1}^{n} W_i X_i \leq M.$$

Greedy Algorithm Design

For each item i, compute the profit-to-weight ratio $r_i = \frac{P_i}{W_i}$.
Sort the items in non-increasing order of r_i.
Set the remaining capacity $R = M$.
for each item i in sorted order **do**
  if $W_i \leq R$ **then**
    Set $X_i = 1$.
    Update $R = R - W_i$.
  else
    Set $X_i = \frac{R}{W_i}$.
    Update $R = 0$.
    break.
  end if
end for
return $\{X_i\}_{i=1}^{n}$.

Time Complexity Analysis

- Computing the ratios for n items takes $O(n)$.
- Sorting the items takes $O(n \log n)$.
- The subsequent loop takes $O(n)$.

Thus, the overall time complexity is $O(n \log n)$.

Proof of Correctness

The correctness of the greedy algorithm for the fractional knapsack problem follows from the *greedy-choice property*:

1. Assume that there exists an optimal solution that does not fully utilize the item with the highest ratio available. Then by exchanging a portion of the item with a lower ratio for a corresponding amount of the item with the highest ratio, the total profit increases without violating the capacity constraint.

2. Hence, it is always optimal to take as much as possible of the item with the highest profit-to-weight ratio.

3. Repeating this argument for the remaining capacity shows that the greedy solution is optimal.