CS 344: Design and Analysis of Computer Algorithms

Instructor: Kangning Wang

Homework 4: Due on **April 7, 2025**

Acknowledgment: **Our team members are Alexander Lin (al1655), Pranav Tikkawar (pt422), and Ivan Zheng (iz72)** .

Problem 1

We covered the offline caching problem in class and stated that the farthest-in-future algorithm is optimal. Formally prove its correctness using the exchange argument. You can refer to our textbook [KT, Section 4.3] to see how the proof is done.

Solution. Let S be an optimal strategy for the given request sequence. Let FiF be the strategy followed by the Farthest-in-Future algorithm. Assume that S and FiF behave identically up to a certain point where they differ in their eviction choices. We'll show that we can modify S to align with FiF at that point without increasing the number of misses.

- Consider the first time S and FiF differ in their eviction decisions. Let this be at request r_i where item x is requested, and it's not in the cache.
- At this point, both S and FiF must evict an item to bring in x (since the cache is full and x is not present).
- Suppose FiF evicts item a (the one farthest in the future or never used again) and S evicts item b (b $\neq$ a).
- Since FiF chose a, this means a is used farther in the future than b, or a is not used again while b is used sooner.
- Define S' to be the same as S up to request r_i.
- At r_i, S' evicts a (like FiF) instead of b.
- After r_i, S' behaves optimally but ensures that it doesn't incur more misses than S.
- Case 1: a is never requested again. Evicting a is optimal since keeping b might lead to an earlier eviction of b later. S' doesn't incur extra misses compared to S.
- Case 2: a is requested later than b. By evicting a instead of b, b will be in the cache when it's next needed before a's next request. Thus, S' might avoid a miss on b that S would have had, or at least not do worse.

- In both cases, S' does not have more misses than S.
- Repeat this process for all points where S and FiF differ. Each exchange either keeps the number of misses the same or reduces them. Eventually, S is transformed into FiF without increasing the number of misses.
- Since any optimal strategy S can be transformed into FiF without increasing the number of misses, FiF must itself be optimal.

Problem 2

We have n boxes. The box i has weight w_i and weight limit ℓ_i. A box will break if the sum of weight above it exceeds its weight limit. Each box has the same size of $1 \times 1 \times 1$.

What is the tallest tower we can build by stacking a subset of boxes on top of each other? We can freely choose the order. Give a greedy algorithm and formally prove its correctness using the exchange argument.

Solution. We have n boxes. The i-th box has weight w_i and weight limit ℓ_i. A box will break if the sum of weights above it exceeds its weight limit. Each box has the same size of $1 \times 1 \times 1$. The goal is to determine the tallest tower we can build by stacking a subset of boxes on top of each other, such that no box breaks. We can freely choose the order in which the boxes are stacked. Not that this problem is quite similar to the interval scheduling problem.
To solve this problem, we use a greedy algorithm based on sorting boxes by the sum of their weight limit and weight $(\ell_i + w_i)$ in non-increasing order. The algorithm proceeds as follows:

1. **Sort all boxes** in non-increasing order of $(\ell_i + w_i)$.
2. **Process boxes in reverse order** (from last to first in the sorted list):
    - Maintain a cumulative weight counter initialized to 0.
    - For each box, if its weight limit ℓ_i is greater than or equal to the cumulative weight, include it in the tower and add its weight w_i to the cumulative counter.

The total number of boxes included in the tower is the height of the tallest possible tower.
We prove the correctness of the greedy algorithm using an exchange argument. Assume an optimal solution O exists that differs from the greedy solution G. If there are two adjacent boxes i and j in O, where i is below j, and $(\ell_i + w_i) < (\ell_j + w_j)$, swapping them preserves validity:

- In O, $\ell_i \geq w_j + S$ and $\ell_j \geq S$, where S is the cumulative weight above box j.
- After swapping, $\ell_j \geq w_i + S$ and $\ell_i \geq S$. Since $(\ell_i + w_i) < (\ell_j + w_j)$, it follows that $\ell_j - w_i > \ell_i - w_j \geq S$, ensuring validity.

By iteratively swapping such pairs, O can be transformed into G without reducing the height of the tower. This proves that G is optimal.

Problem 3

Given a weighted undirected graph $G = (V, E)$, decide whether its minimum spanning tree is unique. Give an algorithm that runs in $O(m \log m)$ time and prove its correctness. [Hint: You can modify Kruskal's algorithm.]

Solution. The algorithm to determine whether the minimum spanning tree (MST) of a weighted undirected graph $G = (V, E)$ is unique involves the following steps:

1. Compute the initial MST T using Kruskal's algorithm. Let W be its total weight.
2. Modify edge weights:
   - For each edge $e \in T$, assign a new weight $(w_e, 0)$.
   - For each edge $e \notin T$, assign a new weight $(w_e, 1)$.
3. Compute a second MST T' using Kruskal's algorithm with lexicographic ordering on the modified weights.
   - Compare edges first by original weight w_e, then by the second component (0 or 1).
   - This prioritizes edges in T when weights are equal.
4. Check uniqueness:
   - If T' has the same original total weight W but differs from T, the MST is not unique.
   - Otherwise, it is unique.

Time Complexity: $O(m \log m)$, where m is the number of edges. The algorithm involves sorting the edges, which takes $O(m \log m)$ time. The Kruskal's algorithm runs in $O(m \log m)$ time as well.

We can prove the correctness of the algorithm as follows:

- The modified weights ensure that edges in the original MST T are prioritized over those not in T when they have the same weight.
- If T' is different from T but has the same total weight, it means that there exists at least one edge in T' that is not in T, indicating that the MST is not unique.
- If $T' = T$, then the MST is unique.

Problem 4

In this problem, we will design an algorithm to compute the minimum spanning tree in a given graph in $O(m \log \log n)$ time, where m is the number of edges and n is the number of vertices in the given graph. For example, if $m = \Theta(n)$, then the running time of this algorithm

($\Theta(n \log\log n)$) is better than that ($\Theta(n \log n)$) of Prim's algorithm, Kruskal's algorithm, and Borůvka's algorithm. Here are the ideas.

- In the original Borůvka's algorithm, we might need $\Theta(\log n)$ rounds to merge all vertices into one group, and each round takes $O(m)$ time. The twist is that we now only run Borůvka's algorithm for r rounds.
- After running Borůvka's algorithm for r rounds, we obtain a new graph with at most $\frac{n}{2^r}$ vertices and at most m edges. We run Prim's algorithm (with a Fibonacci heap) on it to find its MST.

How should we choose the parameter r so that the total running time becomes $O(m \log\log n)$?

Solution. The proposed algorithm combines Boruvka's algorithm with Prim's algorithm. Run Boruvja's for r rounds, after which the number of remaining components is at most $\frac{n}{2^r}$. Each round takes $O(m)$ time, so r rounds take $O(r \cdot m)$. Then construct the contracted graph, where the vertices are the remaining components after r rounds. The edges are the original edges connecting these components (at most m). Run Prim's algorithm on the contracted graph. Using a Fibonacci heap, Prim's runs in $O(m + klogk)$, where k is the number of vertices. Here, $k \leq \frac{n}{2^r}$, so time is $O(m + \frac{n}{2^r} log \frac{n}{2^r})$.

- The total time is the sum of Boruvka's r rounds ($O(r \cdot m)$) and Prim's on the contracted graph ($O(m + \frac{n}{2^r} log \frac{n}{2^r})$). Assuming m dominates (as $m \geq n - 1$ for connected graphs), the time simplifies to $O(r \cdot m + \frac{n}{2^r} logn)$.
- To choose r, we set $r \cdot m + \frac{n}{2^r} logn = O(mloglogn)$. Assuming $m = \Theta(n)$, this becomes $r \cdot n + \frac{n}{2^r} logn = O(nloglogn)$. Dividing both sides, $r + \frac{logn}{2^r} = O(loglogn)$.
- To balance the terms, set r = loglogn: where first term is loglogn and second term is $\frac{logn}{2^{loglogn}} = \frac{logn}{logn} = 1$. Thus, $loglogn + 1 = O(loglogn)$.
- For general m, set r = loglogn. Boruvka's time is $O(mloglogn)$ and Prim's time is $O(m + \frac{n}{2^{loglogn}} log \frac{n}{2^{loglogn}})$.
- $2^{loglogn} = logn$, so $\frac{n}{logn} log(\frac{n}{logn}) \approx \frac{n}{logn}(logn - loglogn) = n(1 - \frac{loglogn}{logn}) = O(n)$. Thus, Prim's time is $O(m+n)$, which is $O(m)$ since $m \geq n-1$. Total time is $O(mloglogn+m) = O(mloglogn)$.
- Therefore, to get runtime $O(mloglogn)$, set the number of Boruvka rounds r to loglogn.

Problem 5

We have an undirected graph $G = (V, E)$. Each edge has a positive length, and there is at most one edge between each pair of vertices. Find the length of the shortest simple cycle in G. Your algorithm should run in $O(n^3)$ time, where $n = |V|$. [Hint: You can modify the Floyd–Warshall algorithm. Recall that $f(k, u, v)$ is the length of the shortest path from u to v using vertices with

indices of at most k. Consider a cycle where k is the largest index among its vertices and u and v are the two neighbors of k in the cycle.]

Solution. Every simple cycle in the graph has a highest-indexed vertex k. The cycle can be decomposed into:

- Two edges: (u, k) and (k, v), where u and v are neighbors of k.
- A path from u to v that uses only vertices with indices less than k.

The algorithm explicitly checks all such cycles by iterating over all possible values of k, and for each k, it considers all pairs of neighbors u, v of k. The shortest path from u to v, which avoids using vertex k, is stored in the distance matrix $D[u][v]$. This allows the algorithm to compute the length of the cycle efficiently.

Completeness: The algorithm considers all possible simple cycles in the graph because:

- Every simple cycle must have a highest-indexed vertex, say k.
- For each such vertex k, the algorithm examines all pairs of neighbors u, v of k.
- The algorithm computes the length of the cycle formed by combining:
  - The shortest path from u to v that uses only vertices with indices less than k, which is stored in $D[u][v]$.
  - The edges (u, k) and (k, v).

Thus, no cycle is missed.

Optimality: The algorithm ensures that the shortest simple cycle is found because:

- At each step, the distance matrix $D[u][v]$ contains the length of the shortest path from u to v that uses only vertices with indices less than or equal to the current value of $k - 1$. This is guaranteed by the Floyd-Warshall update rule:

  $$D[i][j] = \min(D[i][j], D[i][k] + D[k][j]).$$

  This ensures that when considering a cycle involving vertex k, the path from u to v is optimal.

- For each pair of neighbors u, v of k, the algorithm computes the total length of the cycle as:

  $$D[u][v] + w(u, k) + w(k, v),$$

  where:
  - $D[u][v]$: Shortest path from u to v, avoiding vertex k,
  - $w(u, k) + w(k, v)$: Weights of edges connecting vertex k.

By taking the minimum over all such cycles, the algorithm guarantees that it finds the shortest simple cycle.

Complexity: The time complexity of the algorithm is as follows:

- The outer loop iterates over all vertices ($k = 1, 2, ..., n$).
- For each vertex k, we consider all pairs of neighbors (u, v) of vertex k. This takes at most $O(n^2)$.
  - For each pair, computing the candidate cycle length takes constant time.
  - Updating the distance matrix using Floyd-Warshall also takes at most $O(n^2)$ per iteration.

Thus, the total time complexity is:

$$O(n^3),$$

which matches the desired bound.