Algorithm Design and Analysis (Fall 2023)
Assignment 3
Deadline: Dec 12, 2023

1. (25 points) Given a constant $k \in \mathbb{Z}^+$, we say that a vertex u in an undirected graph *covers* a vertex v if the distance between u and v is at most k. In particular, a vertex u covers all those vertices that are within distance k from u, including u itself. Given an undirected *tree* $G = (V, E)$ and the parameter k, consider the problem of finding a minimum-size subset of vertices that covers all the vertices in G. Design a polynomial time algorithm for this problem. Prove the correctness of your algorithm and analyze its running time. You will receive 15 points if you can solve the problem for $k = 1$.

Solution:

First, consider the problem for $k = 1$. The algorithm for $k = 1$ is below.

Algorithm 1 Cover all vertices with a minimum-size subset with $k = 1$

```
1: Boolean vectors $covered[\,]$ and $marked[\,]$ are initialized to $False$.
2: $subset = [\,]$                              ▷ Record the resultant vertex found by algorithm.
3: Apply BFS on the tree $G$ and obtain an array $a[1, ..., n]$ which sorts all vertices by their
   depths incrementally, and an array $pre[1, ..., n]$ recording their parent vertex.
4: for $i = n$ to 1 do
5:     if $covered[i] = True$: continue
6:     $u = pre[i]$
7:     $marked[u] = True$, $covered[u] = True$, $subset$.append($u$)
8:     for $v$ in $N(u)$ do                     ▷ $N(u)$ denotes children vertices of $u$
9:         $covered[v] = True$
10: return $subset$
```

This algorithm repeatedly finds the deepest "uncovered" vertex i, marks its parent vertex u and then covers all vertices within distance $k = 1$ from u. I'll show the correctness of this algorithm by induction. Claim that the subset found by the algorithm is one of the optimal solution.

Base step:

Let $u = subset[1]$, i.e., u is the parent vertex of the deepest vertex $a[n]$ and will be marked by algorithm first, and let $N(u)$ denotes the set containing all children vertices of u. Suppose u is not chosen by the optimal solution, then we must choose all vertices in $N(u)$ since these vertices are deepest vertex and $k = 1$. If $|N(u)| > 1$, we can instead mark u to get a better solution, which leads to contradiction. If $|N(u)| = 1$, then we can alternatively mark u to get another optimal solution.

Inductive step:

Suppose $subset[1, ..., i]$ are vertices chosen by algorithm in first i steps, and all of them are in one optimal solution. We need to prove $subset[i+1]$ is also in one optimal solution.

Let v be the deepest "uncovered" vertex in the $i+1$-th round, u be the parent vertex, and $N'(u)$ be the set containing all "*uncovered*" children vertices of u. By algorithm, we will choose u to be $subset[i+1]$. Suppose u is not chosen by optimal solution, similar to base step, if $|N'(u)| > 1$, we can instead mark u to get better solution, which leads to contradiction. If $|N'(u)| = 1$, then we can alternatively mark u to get another optimal solution.

The algorithm consists of two parts: the time complexity of BFS is $O(|V| + |E|)$; the first loop (outer loop) traverses all vertices and the second loop (inner loop) traverses all edges independently, so the time complexity for this part is also $O(|V| + |E|)$. The overall is $O(|V| + |E|)$.

Now, consider the problem for general k. A natural idea is repeatedly finding the deepest "uncovered" vertex i, marking its k-th ancestor vertex u and then covering all vertices within distance k from u. The algorithm is below.

Algorithm 2 Cover all vertices with a minimum-size subset with general k

```
1: Boolean vectors covered[ ] and marked[ ] are initialized to False.
2: $subset$ =[ ]                    ▷ Record the resultant vertex found by algorithm.
3: Apply BFS on the tree $G$ and obtain an array $a[1, ..., n]$ which sorts all vertices by their
   depths incrementally, and an array $pre[1, ..., n]$ recording their parent vertex.
4: for $i = n$ to 1 do
5:     if $covered[i] = True$: continue
6:     $u = i$
7:     for $j = 1$ to $k$ do
8:         $u = pre[u]$
9:     $marked[u] = True$, $covered[u] = True$, $subset$.append($u$)
10:    Set all vertices $v$ within distance $k$ from $u$ $covered[v] = True$.
11: return $subset$
```

I'll show the correctness by induction extremely similar to the previous part.

Base step:

Let u be the k-th ancestor vertex of the deepest vertex v and will be chosen by algorithm first, and let $N(u)$ denotes the set containing all children vertices of u within distance of k. An important observation is that for any vertex in $N(u)$, the range it covers must be a subset of that of u due to the tree structure.

Suppose u is not chosen by the optimal solution, then at least one vertex in $N(u)$ will be chosen. If we choose more than two such vertices, then we can choose u instead to get better solution which leads to contradiction. If only one such vertex is chosen, then we can alternatively choose u to get another optimal solution since u will cover all vertices the original vertex covers based on our observation.

Inductive step:

Suppose $subset[1, ..., i]$ are vertices chosen by algorithm in first i steps, and all of them are in one optimal solution. We need to prove $subset[i+1]$ is also in one optimal solution.

Let v be the deepest "uncovered" vertex in the $i + 1$-th round, u be the k-th ancestor vertex, and $N'(u)$ be the set containing all "*uncovered*" children vertices of u within the distance of k. By algorithm, we will choose u to be $subset[i + 1]$. Suppose u is not chosen by optimal solution, the left part is same to the base step, so I'll omit it here.

The time complexity of BFS is $O(|V| + |E|)$; the outer loop traverses all vertices, and for each we need to mark k-th ancestor with cost of $O(|V|)$, so the time complexity for this part is $O(|V|^2)$. The overall is $O(|V|^2)$.

2. (25 points) Suppose you are a driver, and you plan to drive from A to B through a highway with distance D. Since your car's tank capacity C is limited, you need to refuel your car at the gas station on the way. We are given n gas stations on the highway with surplus supply. Let $d_i \in (0, D)$ be the distance between the starting point A and the i-th gas station. Let p_i be the price for each unit of gas at the i-th gas station. Suppose each unit of gas exactly supports one unit of distance. The car's tank is empty at the beginning, and the 1-st gas station is at A. Design efficient algorithms for the following tasks.

   (a) (5 points) Determine whether it is possible to reach B from A.

   (b) (20 points) Minimized the gas cost for reaching B.

Solution:

(a) We only need to check if the distance between each two adjacent gas station less than or equal to the capacity C. That is, for any $i \in (1, n]$, if $d_i - d_{i-1} \le C$, then it is possible to reach B, since we can refuel the car to C at each gas station which is enough for the car to reach the next one. Otherwise, even we fill up the car at some station, the car cannot reach next one. The time complexity is trivially $O(n)$.

(b) The algorithm is below.

Algorithm 3 Minimize the gas cost for reaching B

```
function GetNewCost($start$, $end$, $gas$):                ▷ Global $cost$ is initialized to 0.
 1: Find $i$ to minimize $p_i$ where $i \in [start + 1,\ end - 1]$.
 2: if $d_i - d_{start} \le gas$ then                   ▷ Station $i$ is reachable from $start$ with $gas$.
 3:     $gasLeft = gas - (d_i - d_{start})$
 4:     if $C - gasLeft < d_{end} - d_i$ then          ▷ Station $end$ is unreachable from $i$ with $C$.
 5:         $cost = cost + (C - gasLeft) * p_i$                                  ▷ Fill up the car.
 6:         GetNewCost($i$, $end$, $C$)
 7:     else
 8:         $cost = cost + (d_{end} - d_i) * p_i$        ▷ Refuel the car so that it can just reach $end$.
 9: else
10:     if $C < d_{end} - d_i$ then
11:         $cost = cost + C * p_i$
12:         GetNewCost($i$, $end$, $C$)
13:         GetNewCost($start$, $i$, $gas$)
14:     else
15:         $cost = cost + (d_{end} - d_i) * p_i$
16:         GetNewCost($start$, $i$, $gas$)
```

This algorithm recursively operates on a interval from station $start$ to station end with initial gas at $start$. We first want to find the lowest price p_i in this interval and then greedily cover as far as possible. Note that the first call is GetNewCost(1, $n+1$, 0), and the final $cost$ is the result we want.

I'll prove the correctness of this recursive algorithm by induction. That is: 1. Prove it is optimal in the case of not calling itself; 2. Suppose the result it gets from calling itself is optimal, prove algorithm of previous layer is optimal.

Base step:

The case the algorithm doesn't call itself is that (line 8 of the algorithm): i is reachable from $start$ with gas, and end is also reachable from i with $d_{end} - d_i + gasLeft$. This is trivially optimal since the total extra gas we need is $d_{end} - d_{start} - gas = d_{end} - d_i + (d_i - d_{start} - gas) = d_{end} - d_i + gasLeft$, and the algorithm buys this amount of gas with minimum cost per unit p_i.

Inductive step:

There are three scenarios here:

1. Station i is reachable from $start$ with gas, but end is not reachable from i with C (line 4).
   Here the cost of the problem can be divided into two parts: the cost of refueling from station $start + 1$ to i, and the cost of GetNewCost(i, end, gas), where gas is a variable. First, claim that if a certain amount of gas g is added to the car in some stations in $[start + 1, i]$, refueling at station i is optimal since p_i is the minimum among p_j, $j \in [start + 1, end - 1]$. Second, claim that if we only add g ($g \in [0, C - gasLeft]$) at station i, $g = C - gasLeft$ gives the optimal total cost. Otherwise suppose $g_0 < C - gasLeft$ is optimal. To reach B, in the distance between d_{end} and $d_{end} + C$, at least $C - (g_0 + gasLeft)$ gas will be added at some station at a higher unit price than p_i. We can instead add these $C - (g_0 + gasLeft)$ add at station i (i.e., add $C - gasLeft$ at station i) to get a better solution, which leads to contradiction.
2. Station i is not reachable from $start$ with gas, but end is reachable from i with $d_{end} - d_i$ (line 13).
   The analysis here is similar to previous part. The cost can be divided into two parts: the cost of refueling from station i to $end - 1$, and the cost of GetNewCost($start$, i, gas). According to our assumption, GetNewCost($start$, i, gas) is optimal. This implies that the second part is a fixed value, which is the minimum cost from $start$ to i, and the gas left at station i will be 0 according to base step. For the first part, only add $d_{end} - d_i$ gas at station i

will trivially minimize the cost from i to *end*. Therefore, overall is optimal.

3. Neither Station i is reachable from *start* with *gas*, nor *end* is reachable from i with C (line 10).
   This scenario is just the combination of the previous two, so the proof is omitted here.

For an interval with length a, finding it's minimum value takes $O(a)$, and after the algorithm performs on this interval, at least one station can be "cut down", introducing 0 to 2 smaller intervals. The worst case is that we need to "cut down" n times, introducing n layers, each layer consists of many intervals, and the sum of their length is less than or equal to n. So the worst time complexity is $O(n^2)$.

3. (25 points) Given a ground set $U = \{1, \dots, n\}$, a *set function on* U is a function $f : \{0,1\}^U \to \mathbb{R}$ that maps a subset of U to a real value. A set function f is *submodular* if

$$f(S \cup \{v\}) - f(S) \geq f(T \cup \{v\}) - f(T)$$

holds for any $S, T \subseteq U$ with $S \subseteq T$ and any $v \in U \setminus T$. We make the following assumptions on a submodular set function f:

- Nonnegative: $f(S) \geq 0$ for any $S \subseteq U$; you can assume $f(S)$ is always a rational number.
- Monotone: $f(S) \leq f(T)$ for any $S, T \subseteq U$ with $S \subseteq T$.
- $f(S)$ can be computed in a polynomial time with respect to $n = |U|$.

Given a positive integer $k > 0$ as an input, the goal is to find $S \subseteq U$ that maximizes $f(S)$ subject to the cardinality constraint $|S| \leq k$. Design a polynomial time $(1 - 1/e)$-approximation algorithm for this maximization problem. Prove that the algorithm you design runs in a polynomial time and provides a $(1 - 1/e)$-approximation.

Solution:

The greedy algorithm is below, which is highly similar to the approximation solution of Max-k-Coverage problem introduced in the lecture.

Algorithm 4 Find the set to maximize the submodular function

```
1: Initialize the resultant array $A =[\ ]$ and $visited[1,...,n] = False$.
2: Find the element $v$ that maximizes $f(\{v\})$.
3: $visited[v] = True$
4: for $i = 1,...,k-1$ do
5:     Find the elemnet $v$ that maximizes $f(A \cup \{v\}) - f(A)$ and $visited[v] = False$.
6:     $A$.append($v$), $visited[v] = True$
7: return $A$
```

Now I will show this is a $(1 - \frac{1}{e})$-approximation algorithm. Let $A_l = \{a_1, a_2, ..., a_l\}$ denotes the set selected by algorithm after l rounds, and a_i is selected in i-th round, and $O = \{o_1, o_2, ..., o_k\}$ denotes the optimal set. Let $\Delta(u \mid S) = f(S \cup \{u\}) - f(S)$.

First, claim that after l rounds, $f(A_l) \geq (1 - (1 - \frac{1}{k})^l) \cdot f(O)$. I'll show this by induction.

Base step:

For $l = 0$, $1 - (1 - \frac{1}{k})^l = 0$, $f(\emptyset) \geq 0$ holds trivial.

Inductive step:

Suppose $f(A_l) \geq (1-(1-\frac{1}{k})^l)\cdot f(O)$. We need to show $f(A_{l+1}) = f(A_l \cup \{a_{l+1}\}) \geq (1-(1-\frac{1}{k})^{l+1})\cdot f(O)$.

By the nature of greedy, we have

$$\Delta(a_{l+1} \mid A_l) \geq \Delta(o_i \mid A_l), \quad i \in [1,\ k],$$

then applying the averaging trick we have

$$\Delta(a_{l+1} \mid A_l) \geq \frac{1}{k}\sum_{i=1}^{k}\Delta(o_i \mid A_l).$$

For any $i \in [1,\ k]$, let $S = A_l$, $T = A_l \cup \{o_1, ..., o_{i-1}\}$. Then by the definition of submodular function, we have

$$\Delta(o_i \mid A_l) = f(A_l \cup \{o_i\}) - f(A_l) \geq f(A_l \cup \{o_1, ..., o_i\}) - f(A_l \cup \{o_1, ..., o_{i-1}\}).$$

Note that o_i may in A_l which violates the definition, in this case, the inequality still holds since both two sides equal to 0.

Therefore, by this inequality, we have

$$\begin{aligned} f(A_{l+1}) - f(A_l) = \Delta(a_{l+1} \mid A_l) &\geq \frac{1}{k}\sum_{i=1}^{k}\Delta(o_i \mid A_l) \\ &\geq \frac{1}{k}\sum_{i=1}^{k} f(A_l \cup \{o_1, ..., o_i\}) - f(A_l \cup \{o_1, ..., o_{i-1}\}) \\ &= \frac{1}{k}(f(A_l \cup O) - f(A_l)) \geq \frac{1}{k}(f(O) - f(A_l)). \end{aligned}$$

Then we have that

$$\begin{aligned} f(A_{l+1}) &\geq \frac{1}{k}f(O) + (1-\frac{1}{k})f(A_l) \\ &\geq \frac{1}{k}f(O) + (1-\frac{1}{k})(1-(1-\frac{1}{k})^l)f(O) \\ &= (1-(1-\frac{1}{k})^{l+1})f(O), \end{aligned}$$

which proofs the claim. After k rounds, with inequality $1-\frac{1}{k} < e^{-\frac{1}{k}}$, we have

$$f(A_k) \geq (1-(1-\frac{1}{k})^k)f(O) > (1-\frac{1}{e})f(O).$$

Suppose the time complexity of computing $f(S)$ is $O(n^m)$. For each round, we need to traverse all n elements. So the overall time complexity is $O(k \cdot n^{m+1})$, which implies the algorithm runs in a polynomial time.

4. (25 points) Given an undirected graph $G = (V, E)$, a *matching* M is a subset of edges such that no two edges in M share an endpoint. The *maximum matching problem* takes the graph $G = (V, E)$ as the input and outputs a matching M with the maximum size $|M|$. Consider the following greedy algorithm.

```
• initialize $M \leftarrow \emptyset$
• while there exists $e \in E$ such that $M \cup \{e\}$ is a valid matching:
    – update $M \leftarrow M \cup \{e\}$
• endwhile
• return $S$
```

(a) (20 points) Prove that this is a 0.5-approximation algorithm.

(b) (5 points) Provide a tight example showing that this is not a $(0.5+\varepsilon)$-approximation algorithm for any $\varepsilon > 0$.

Solution:

(a) Let M_{OPT} denotes the optimal matching, and M_{ALG} denotes the subset given by the approximation algorithm. For a vertex $u \in M_{ALG}$, there are two possible scenarios:

- $u \in M_{OPT}$: u makes same contribution to both optimal solution and algorithm solution, and it is independent of other edges in M_{OPT} and M_{ALG}.
- $u \notin M_{OPT}$: Let $u = AB$. There are obviously at most two edges in M_{OPT} sharing endpoint with u, recorded as AA' and BB'. The worst case is that A' and B' are not covered in M_{ALG} (since we are analysing the lower bound of the approximation, we are not care about whether this worst case is possible).

The worst case is all edges in M_{ALG} belongs to the worst case of the second type, in other words, each edge in M_{ALG} will introduces at most two edges in M_{OPT}. Therefore, $2|M_{ALG}| \leq |M_{OPT}|$, which implies the algorithm is 0.5-approximation.

(b) Consider the graph shown below. There are four vertices, the optimal solution is $M_{OPT} = \{AB,\ CD\}$ with size of 2. However, if the algorithm choose AC first, then no other edges can be chosen, which produces $M_{ALG} = \{AC\}$ with size of 1.

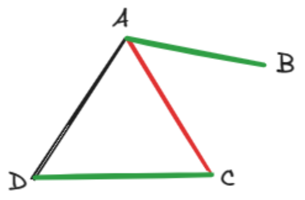

Figure 1: A tight example.

5. How long does it take you to finish the assignment (including thinking and discussion)? Give a score (1,2,3,4,5) to the difficulty. Do you have any collaborators? Please write down their names here.

   Solution:

   These four questions cost 4h, 6h, 2h, 1h respectively (typing also included).

   The difficulty scores are 4, 5, 3, 2 respectively.

   I complete this assignment independently.