CS 577: HW 3

Daniel Ko

Summer 2020

§1 Kleinberg Chapter 6, Q14

§1.1 Suppose it is possible to choose a single path P that is an $s-t$ path in each of the graphs $G_0, G_1, \cdots, G_b$. Give a polynomial-time algorithm to find the shortest such path.

We define the set of edges that exist in all points in time as

$$E^* = \bigcap_{i=0}^{b} E_i$$

Since our graph is unweighted, we can perform breadth first search to find the shortest path from s to t on $G^* = (V, E^*)$. This will be $O(n)$, where n is the number of vertices in graph G.

§1.2 Give a polynomial-time algorithm to find a sequence of paths $P_0, P_1, \cdots, P_b$ of minimum cost, where P_i is an $s-t$ path in G_i for $i = 0, 1, \cdots, b$.

For an $s-t$ path P in one of the graphs G_i, we define the length of P to be simply the number of edges in P, and we denote this to be $\ell(P)$. Formally, we define changes$(P_0, P_1, \ldots, P_b)$ to be the number of indices i $(0 \leq i \leq b-1)$ for which $P_i \neq P_{i+1}$. Fix a constant $K > 0$. We define the cost of the sequence of paths $P_0, P_1, \ldots, P_b$

$$\text{cost}\left(P_0, P_1, \ldots, P_b\right) = \sum_{i=0}^{b} \ell\left(P_i\right) + K \cdot \text{changes}\left(P_0, P_1, \ldots, P_b\right)$$

§1.2.1 Set up the recursive formula and justify its correctness.

We define MinPath(i) to be the recurrence for the a sequence of paths of minimum cost in $G_0, \cdots, G_i$. Let the set of edges that exist from graphs $G_a, \cdots, G_b$ be defined as

$$E_{a,b} = \bigcap_{i=a}^{b} E_i$$

Let $\ell_E(E_{a,b})$ be the length of the shortest $s-t$ path in $E_{a,b}$. If a path does not exist we define $\ell_E(E_{a,b}) = \infty$.
The main idea behind MinPath is that there are two major cases for the minimum sequence of paths. The first major case is if there exists a single path for all graphs from G_0 to G_i. Then, we just calculate the length of this single path and multiply it by the amount of graphs we are considering. Note that since there are no changes we do not have to worry about the cost of changing paths, i.e. the K value in the cost function. The second major case is if we need to change paths. We

look at all places where the sequence of paths changes to a path that is contained in G_i. This path change happens between the graph ϕ and $\phi+1$. We recursively calculate the minimum cost of paths between 0 and ϕ and compute the cost of the path from $\phi+1$ to i, which includes the cost of changing paths, K. We also add in a special condition when we consider this second case which is $i > 0$. This is because if $i = 0$, there would be no switching paths as there is only one graph to consider. We need at least 2 graphs for switching paths to make sense. We formally define MinPath as

$$\text{MinPath}(i) = \min\left(\ell_E\left(E_{0,i}\right)\cdot\left(i+1\right), \min\left(\left\{\text{MinPath}(\phi)+\ell_E\left(E_{\phi+1,i}\right)\cdot\left(i-\phi\right)+K \mid (0 \le \phi < i)\wedge(i>0)\right\}\right)\right)$$

We assume that the min function for an empty set will return an empty set. Additionally, if one of parameters of the min function is an empty set, it will return the other parameter.

Proof. We show by strong induction that our recurrence relation is correct. Let $P(i)$ be the predicate, "MinPath correctly computes the sequence of paths of minimum cost in $G_0, \cdots, G_i$". We define $i \in \mathbb{N}$.
Base Case: When $i = 0$, it is trivial to see that the correct minimum cost is $\ell_E(E_{0,0})$ as there is only one graph to consider. We see that value in the left parameter of the min function evaluates to $\ell_E(E_{0,0}) \cdot (0+1) = \ell_E(E_{0,0})$. The right parameter never evaluates. Thus, the min of these two values is $\ell_E(E_{0,0})$, which is the correct minimum cost. Therefore, $P(0)$ holds.
Inductive step: Suppose $P(i)$ holds for all $0 \le i \le k$. We show that $P(k+1)$ holds. We have two major cases. The first case is if there exists a single minimum path from graphs $G_0, \cdots, G_{k+1}$. Then the cost of this sequence will be $\sum_{i=0}^{k+1} \ell_E(E_{0,k+1}) = \ell_E(E_{0,k+1})((k+1)+1)$, which is exactly what is in the first parameter of our min function. The second case is if the minimum path at G_{k+1} is not the same as all the paths before it. We check by exhaustion where the optimal change is, where ϕ is the index of the last graph where its minimum path is different than the minimum path at graph $k+1$. The cost of the entire path would be the cost of the sequence of minimum paths from graph 0 to graph ϕ, which would be MinPath(ϕ), plus the length of the minimum path in graph $k+1$ times the amount of graphs this path shows up in plus some constant K. If no such path exists from $\phi+1$ to $k+1$, then $\ell_E(E_{\phi+1,k+1}) = \infty$ as defined. Since $\phi \le k$, by our strong inductive hypothesis MinPath(ϕ) is correct. We do this for all values of ϕ and pick the smallest cost. We choose the smaller of both cases and return the minimum cost.
By strong induction, we have proven that $P(i)$ holds for all $i \in \mathbb{N}$. Therefore, MinPath is correct. □

§1.2.2 Write the pseudocode for the iterative version of the algorithm to find the minimum cost. You are not required to write pseudocode to find the shortest path.

Data: A list G, containing graphs $G_0, \cdots, G_b$
Result: The minimum cost of the sequence of paths in $G_0, \cdots, G_b$
def *minCost(G)*:
  dp ← a list contain the minimum cost for the sequence of paths from $G_0, \cdots, G_i$
  for *i from* 0 *to b*:
    dp[i] =
    $\min\left(\ell_E(E_{0,i})\cdot(i+1), \min(\{\text{dp}[\phi]+\ell_E(E_{\phi+1,i})\cdot(i-\phi)+K \mid (0 \le \phi < i)\wedge(i>0)\})\right)$
  return dp[b]

§1.2.3 Analyze the computing complexity.

We claim that the computing complexity is $O(b^3n^2)$.

Proof. Our for loop iterates b times. Let the number of vertices in G be n. The amount of edges in each graph is at most n^2. Computing $E_{\alpha,\beta}$ take at most bn^2 time, i.e. comparing the intersection of b graphs with n^2 edges. Computing the ℓ_E function takes n time if we use breadth first search. In total, computing $\ell_E(E_{\alpha,\beta})$ takes $bn^2 + n$. Inside the for loop, we compute $\ell_E(E_{\alpha,\beta})$ at most b times. The rest of the computations take constant time. Hence our total time complexity is $O(b(b(bn^2 + n))) = O(b^3n^2)$. □

§2 Given a rooted tree $T = (V, E)$ and an integer k, find the largest possible number of disjoint paths in T, where each path has length k.

§2.1 Set up the recursive formula and justify its correctness.

We define MaxPath(v) to be the recurrence for the maximum number of disjoint paths of size k in a sub tree of T with root v. We consider two major cases, the maximum number of disjoint paths may or may not contain v.

$$\text{MaxPath}(v) = \max\Big(\text{MaxDoesNotContain}(v), \text{MaxContains}(v, 0)\Big)$$

In the case where maximum number of disjoint paths does not contain v, we define MaxDoesNotContain(v) to be the sum of the maximum number of disjoint paths of size k for each sub tree generated by the children of v, i.e. the sub trees having c as the root where $(v, c) \in E$. We will define the set of child nodes of a vertex v as

$$C_v = \Big\{c \mid (v, c) \in E\Big\}$$

Adding up all the maximum paths for each sub tree gives us the total amount of maximum paths for the entire tree.

$$\text{MaxDoesNotContain}(v) = \sum_{c \in C_v} \text{MaxPath}(c)$$

In the case where the maximum number of disjoint paths contains r, there are a couple of subcases. We define MaxContains(v, δ) to be the maximum number of disjoint paths of size k in a sub tree of T with root v, given that v is currently a part of a path of size δ, where δ is bounded by $0 \leq \delta \leq k$. The first subcase is if v is a leaf node and the current size of the path it's on is less than k, i.e $\delta < k$, there would be 0 disjoint paths of size k in this sub tree. The second subcase is when v is the last node in a path of size k, i.e. $\delta = k$. Then, the maximum paths will be the sum of the maximum paths on the sub trees generated by the children of v, or equivalently MaxDoesNotContain(v), plus the path that v is on. The third subcase covers when v is on some path that is not complete nor trivially ends on v. The maximum number of disjoint paths in the sub tree of root v currently a part of a path of size δ, will be the maximum number of disjoint paths of some sub tree $c \in C_v$ which would be part of a path of size $\delta + 1$, plus the sum of the maximum number of disjoint paths for the rest of the children of v. We try out all combinations of c and pick the maximal one.

$$\text{MaxContains}(v, \delta) = \begin{cases} 0 & \textbf{when } \delta < k \textbf{ and } v \text{ is a leaf} \\ \text{MaxDoesNotContain}(v) + 1 & \textbf{when } \delta = k \\ \max\left(\left\{\text{MaxContains}(c, \delta + 1) + \displaystyle\sum_{c' \in C_v \setminus \{c\}} \text{MaxPath}(c') \mid c \in C_v\right\}\right) & \textbf{otherwise} \end{cases}$$

Proof. We show by strong induction that our recurrence relation is correct. Let $P(n, k)$ be the predicate, "MaxPath correctly computes the maximum number of disjoint paths of size k in a sub tree of T which has n number of nodes and where v is the root node". We assume that the size of a path must be at least 1, as a path of 0 would lead to infinite amount of paths, and that a sub tree must contain at least one node, as our sub tree has a root. Hence, we define $n, k \in \mathbb{N}^+$.
Base Case: When $n = 1, k = 1$, MaxPath returns 0. MaxContains$(v, 0)$ returns 0 because $0 < 1$ and v is a leaf. MaxDoesNotContain(v) returns 0 because v has no children. The max of $\{0, 0\}$ equals 0. It is clear to see that there are no paths for a tree that consists of a single node. Hence, $P(1, 1)$ holds.
Inductive step for number of nodes: Suppose $P(\eta, \kappa)$ holds for all $n \leq \eta$ and $k \leq \kappa$. We show that $P(\eta + 1, \kappa)$ holds. MaxPath(v) calls MaxContains$(v, 0)$ and MaxDoesNotContain(v). Since v is not a leaf and $0 \neq k$, third condition holds in MaxContains. Notice that the number of nodes the sub tree where the root is c will have strictly less nodes than $n+1$, which is amount of nodes in the sub tree where the root is v because c is a child of v. Hence, MaxContains$(c, 1)$ holds by our inductive hypothesis. Without loss of generality, MaxPath(c') and MaxDoesNotContain(v) holds by our inductive hypothesis. Therefore, $P(\eta + 1, \kappa)$ holds.
Inductive step for length of path: Suppose $P(\eta, \kappa)$ holds for all $n \leq \eta$ and $k \leq \kappa$. We show that $P(\eta, \kappa + 1)$ holds. MaxPath(v) calls MaxContains$(v, 0)$ and MaxDoesNotContain(v). We continue with the third condition in MaxContains$(v, 0)$. Notice that MaxContains$(c, 1)$ is equivalent to MaxContains$(c, 0)$ if we fix $k = \kappa$. Since $\kappa \leq \kappa + 1$, MaxContains$(c, 0)$ where $k = \kappa$ and subsequently MaxContains$(c, 1)$ where $k = \kappa + 1$ holds by our inductive hypothesis. Without loss of generality, MaxPath(c') and MaxDoesNotContain(v) holds by our inductive hypothesis because they will eventually reach MaxContains$(v', 1)$. Therefore, $P(\eta, \kappa + 1)$ holds.
By strong induction, we have proven that $P(n, k)$ holds for all $n, k \in \mathbb{N}^+$. Therefore, MaxPath is correct. □

§2.2 Write the pseudocode for the iterative version of the algorithm to find the maximum number of players that can play at the same time.

The main idea of this algorithm is to compute bottom up MaxPath. We iterate over all vertices in the tree using post order traversal. We compute all MaxContains values for that vertex and possible size of the paths that they are on. All the recursive values needed in the recurrence relationship will be computed by the time its needed.

```
Data: T = (V, E), a tree with root r; k, length of path
Result: Maximum number of disjoint paths of size k in the tree T
def max(T, k):
    contains ← a map representing memoized values of MaxContains, where < v, δ > is the
     key
    notContains ← a map representing memoized values MaxDoesNotContain, where node
     v is the key
    maxPaths ← a map representing memoized values MaxPath, where node v is the key
    for v ∈ V in post order traversal:
        #No paths unless this is the last edge needed to create a path
        if v is a leaf:
            maxPaths.insert(v, 0)
            notContains.insert(v, 0)
            for i from 0 to k - 1:
                contains.insert(<v,i>, 0)
            contains.insert(<v,k>, 1)
        # Computing MaxDoesNotContain(v) and MaxContains(v,δ)
        else:
            notContainVal = 0
            for each child c of v:
                notContainVal+=maxPaths.get(c)
            notContains.insert(v,notContainVal)
            for i from 0 to k-1:
                containsVal ← a empty list
                for each child c of v:
                    #Where c' ∈ C_v \ {c}
                    val = contains.get(c,i+1) + ∑ maxPaths.get(c')
                    containsVal.insert(val)
                contains.insert(<v,i>, max(containsVal))
            contains.insert(<v,k>, 1+ notContains.get(v))
            maxPaths.insert(v, max(contains.get(<v,0>),notContains.get(v)))
    return maxPaths.get(r)
```

§2.3 Analyze the computing complexity.

We claim the time complexity for max is $O(nk)$.

Proof. Let n be the number of vertices in T. The outer for loop runs n times. Inside the if statement, there is a for loop that runs k times. All other computations in the if statement runs in constant time. This leads to computing complexity of $O(nk)$ for the for loop and the if statement.

In the else statement, we access the child nodes of a vertex. Notice that no child vertex will be accessed twice, leading to a complexity of $O(n)$ for the for loop that goes through all the

vertices and the for loop accessing the children. The total complexity of outer for loop and the else statement is $O(2n + nk) = O(nk)$, where $2n$ comes from the 2 for loops for accessing the children and nk comes from the outer for loop, n, and the inner for loop, k.
In either the if statement or the else statement, the time complexity is $O(nk)$. Therefore, the time complexity for max is $O(nk)$. □