# CSOR W4246 - Summer 2021
# Homework #2

Joseph High - `jph2185`

June 1, 2021

## Problem 1

Let $G = (V, E)$ be a graph with $e = (x, y) \in E$. The algorithm runs BFS on a modified graph $G' = (V, E')$, where $E' = E \setminus e$, starting at any one of the end-point nodes incident to $e$. *W.l.o.g.*, suppose we run BFS from $x$. When the BFS procedure terminates, the position of $y$ in the *discovered* array utilized in the BFS algorithm is evaluated. In particular, if $discovered[y] = 1$, then it indicates that $y$ is reachable from the input node. Because the edge $(x, y)$ was deleted, this implies that there must be a cycle in the original graph that contains the edge $(x, y)$. On the other hand, if $discovered[y] = 0$, then $y$ was not explored by BFS, indicating that $y$ is not reachable from $x$, implying that no cycles containing the edge $(x, y)$ exist in the original graph.

Pseudocode:

```
CycleDetect($G = (V, E),\ e = (x, y)$):
    $n = size(V)$                          // Define number of nodes in $G$.
    $E' = E \setminus \{(x, y)\}$          // Delete $e$ from the original graph
    $G' = (V, E')$                         // Construct new graph from previous graph with $e$ removed.
    if $n == 2$ then                       // Since the graph must have at least 2 nodes
        return False                       // If $|V| = 2$, then $G$ cannot have a cycle since $G$ is simple
    else
        Run BFS($G', x$)
        if $discovered[y] == 1$ then
            return True
        else
            return False
        end if
    end if
end
```

Running Time: Deleting an edge from $E$ in line 2 requires constant time. Constructing a new graph (with $e$ removed) in line 3 requires $O(n + m)$ time. The BFS procedure requires

$O(n+m)$ time. Lines 5 through 9 require constant time. Therefore, the total running time of the algorithm is $O(n+m)$.

Correctness: Let $G=(V,E)$ be a graph where $e=(x,y)\in E$. Suppose $G$ has a cycle that contains $e=(x,y)$, and let $(x,y,v_1,v_2,\dots,v_k,x)$ be a cycle in $G$ containing the edge $(x,y)$. The cycle contains two $x-y$ paths: a path from $x$ to $y$ using the edge $(x,y)$, and a path from $y$ to $x$ using the path $(y,v_1,v_2,\dots,v_k,x)$. Now consider the graph $G'=(V,E')$ where $E'=E\setminus e$, the original edge set with the edge $e=(x,y)$ removed. The modified graph $G'$ still contains one $x-y$ path: $(y,v_1,v_2,\dots,v_k,x)$. In particular, since $G$ is undirected, then the node $y$ is still reachable from $x$ using the path $(x,v_k,\dots,v_2,v_1,y)$. Therefore, $y$ would be explored by `BFS`$(G',x)$ since $y$ is reachable from $x$ using the path $(x,v_k,\dots,v_2,v_1,y)$. Hence, the algorithm returns `True`, since, by the above argument, this indicates that the graph has a cycle containing $e$.

Now suppose that $G$ has no cycles containing the edge $e=(x,y)$, and again let $G'=(V,E')$ with $E'=E\setminus e$. Claim: $G'$ has no $x-y$ paths. To prove the claim, suppose to the contrary that there is an $x-y$ path in $G'$ and let $P_{xy}=(x,u_1,u_2,\dots,u_l,y)$ be this path. Now consider augmenting $P_{xy}$ with an (undirected) edge between $x$ and $y$. That is, $P_{xy}\cup e=(x,u_1,u_2,\dots,u_l,y,x)$ which is a cycle containing the edge $(x,y)$. This implies that $G$ has a cycle containing the edge $(x,y)$, contradicting the assumption that $G$ has no cycles containing the edge $(x,y)\implies y$ is not reachable from $x$ in $G'$, and vice versa (since $G$ is undirected). Therefore, $y$ will not be explored by `BFS`$(G',x)$. Hence, the algorithm returns `False`, since, by the above argument, indicates that no cycles containing $e$ exist in $G$.

## Problem 2

**(a)** Because $G$ is a directed acyclic graph (DAG), then G has a topological ordering. Apply the topological sort algorithm on $G$ to obtain the topological ordering of the vertices in $G$. Using the linked list output from the `Topological-Sort` procedure, we compute the minimum reachable price starting at the end of the list. That is, we start by computing the minimum reachable price for nodes with no outgoing edges, which should be the price of itself; indeed, no nodes other than itself are reachable from nodes with no outgoing edges. By computing it backward in this way, computing the minimum price for adjacent nodes (in-neighbors) only requires comparing the price of itself and the `cost` value for all nodes it is adjacent to (out-neighbors).

Pseudocode:

```
 1: MinReachCost-DAGs($G$, $p = [p_1, p_2, \dots, p_n]$)
 2:     $n = size(V)$                         // Define number of nodes in $G$.
 3:     $L$ = Topological-Sort($G$)       // Store linked list output from Topological-Sort
 4:     for $u \in V$ do                                  // Initialize cost array.
 5:         cost[$u$] = 0
 6:     end for
 7:     if $n == 1$ then
 8:         for $v \in V$ do
 9:             cost[$v$] = $p_v$      // Only one node $v \in V$, so min cost is the price of itself.
10:         end for
11:     else
12:         for $i = n$ to 1 do
13:             if $outdeg(L[i]) == 0$ then            // If node has no outgoing edges
14:                 cost[$L[i]$] = $p[L[i]]$      // then the min cost equal to price of itself
15:             else
16:                 cost[$L[i]$] = $\min\{p[L[i]], \min_{u \in \texttt{neighbors}(L[i])}$ cost[$u$]$\}$
17:             end if
18:         end for
19:     end if
20:     return cost
21: end
```

Running Time: The running time to initialize $n$ is $O(1)$. The `Topological-Sort` procedure requires linear time: $O(n+m)$. Because we are working with the word RAM model, initializing the `cost` array at lines 4 through 6 requires $O(n)$ time. Lines 7 through 9 takes $O(1)$ time. The for-loop at lines 10 through 16 fills the rest of the `cost` array, computing the minimum price according to the minimum cost values of adjacent nodes; therefore, for a fixed $u \in V$, the (worst-case) running time to compute `cost`$[u]$ is $O(deg(u))$. Iterating over all $n$ nodes, the (worst-case) time to compute each entry in `cost` is $n \cdot O(deg(u)) = O\left(\sum_{i=1}^{n} deg(u)\right) = O(m)$. Therefore, the total running time is:

$$O(1) + O(n+m) + O(n) + O(1) + O(m) \;=\; O(n+m)$$

Correctness: Induct on $n$.

**Base case** ($n = 1$): Let $v$ be the one node in $V$. Referring to the pseudocode above: when $n = 1$, the algorithm sets the value of $\texttt{cost}[v]$ equal to $p_v$, the price of itself, which is correct. Indeed, if $V = \{v\}$, the only node reachable from $v$ is itself. Therefore, $\texttt{cost}[v] = p_v$.

**Induction Hypothesis**: For all $1 \leq k \leq n-1$, assume `MinReachCost-DAG` correctly computes the minimum price reachable from all nodes, and thus correctly filling in the `cost` array.

**Inductive Step**: Let $G = (V, E)$ with $V = V_{cost} \cup V_{rem}$ and $E = E_{cost} \cup E_{rem}$, where

- $V_{cost} = \{v \in V : \texttt{cost}[v] \text{ already computed}\}$ (i.e., the set of nodes for which `cost` has already been computed)
- $V_{rem} = \{u \in V : \text{cost}[u] \text{ not yet computed and } indeg(u) = 0\}$ (i.e., the set of nodes such that `cost` has not yet been computed)
- $E_{cost} = \{(u, v) : u, v \in V_{cost}\}$
- $E_{rem} = \{(u, v) : u \in V_{rem} \text{ and } v \in V_{cost}\}$,

Specifically, the set $V_{rem}$ consists of the nodes at the beginning of the topological ordering with no incoming edges, and $V_{cost}$ consists of all nodes to the right of the nodes in $V_{rem}$. Note that $|V_{rem}| \geq 1$. Then $|V_{cost}| \leq n - 1 \implies \forall v \in V_{cost}$, $\texttt{cost}[v]$ has been correctly computed, by the induction hypothesis. For $u \in V_{rem}$, computing the minimum reachable price reduces to computing the minimum of the price of the node itself and $\texttt{cost}[v]$ for all $v \in N(u)$ (out-neighbors of $u$). That is, for $u \in V_{rem}$, $\texttt{cost}[u] = \min\{p_u,\ \min_{v \in N(u)} \texttt{cost}[v]\}$. Because $\texttt{cost}[v]$ has been computed correctly for all $v \in V_{cost}$, the minimum reachable price is selected for all $u \in V_{rem}$.

**(b)** The algorithm computes the minimum reachable price for any directed graph: cyclic or acyclic. If the input graph is acyclic (a DAG), the algorithm calls the algorithm proposed in part (a) of this problem since it solves the problem for directed acyclic graphs. Otherwise, if the graph contains at least one cycle, the algorithm runs the SCC procedure covered in class to compute all SCCs in the graph. We can then run the algorithm from part (a) on the meta graph since it is a DAG. However, we must first adjust prices of the nodes in each SCC. For each SCC, the algorithm determines the minimum price among all nodes in the SCC, and subsequently replaces their current price with the minimum price in the SCC since all nodes in the SCC are reachable from each other. That is, the meta-graph is constructed so that the price of each SCC node is the minimum reachable price of all nodes within that SCC. At this point, the algorithm runs the algorithm from part (a) on the meta-graph with the adjusted prices.

**Pseudocode, running time analysis, and proof of correctness are all provided on the following page.**

```
MinReachCost-All($G$, $p = [p_1, p_2, \ldots, p_n]$)
    Run DFS($G$)
    if $G$ does not have a back edge then                    // If $G$ is a DAG
        Run MinReachCost-DAGs($G, p$)                        // Run algorithm from part (a).
    else if $G$ has a back edge then                         // If $G$ contains a cycle
        $SCC$ = SCC($G$)                                     // Storing output from SCC algorithm.
        for $T \in SCC$ do                                   // $T$ denotes each SCC tree from DFS
            for $u \in T$ do                                 // For all nodes in a given SCC
                $p[u] = \min_{v \in T} p_v$      // Since all $v$ in SCC are reachable from all other $u$
            end for
        end for
        Run MinReachCost-DAGs($G_{meta}, p$)                 // Run part (a) alg. on meta-graph
    end if
    return cost
end
```

Running Time: If $G$ is a DAG, the running time of the algorithm is $O(n+m)$, which was argued in part (a). If, however, $G$ has a cycle the running time is as follows: The running time of the `SCC` procedure is $O(n+m)$. The for-loop iterates at most $n$ times, and for each iterate the minimum cost is computed for each node in the given SCC tree. The algorithm will only iterate $n$ times in the case that there are $n$ SCCs, in which case each SCC has exactly one node. On the other hand, if there is only one SCC, the for-loop iterates once, but the time to compute `cost` is $O(n)$, which only needs to be done once since all nodes in the SCC are reachable from each other. That is, the worst-case running-time of the for-loop is $O(n)$. The total running time is then $O(n+m) + O(n) = O(n+m)$.

Correctness: It suffices to show that the algorithm correctly computes the minimum cost for directed graphs with cycles since correctness has already been proven for the `MinReachCost-DAGs` algorithm in part (a). All nodes in an SCC are reachable from one another; therefore, the minimum reachable price for one node is the minimum reachable price for all nodes in the SCC. The algorithm constructs the meta-graph so that the price of each SCC node is minimum reachable price of all nodes within that SCC; indeed, if an alternative price was used for an SCC, the price would not be a minimum. The proof of correctness from part (a) applies to the remainder of the algorithm since all that is left is to run the algorithm from part (a) on the meta-graph, which is a DAG. As shown in part (a), this gives the minimum cost for each position.

## Problem 3

We can solve this using a memoization technique used in DP, not including the optimization aspect of DP. In particular, let $M[i, j]$ be the probability that the $j^{th}$ head occurs on the $i^{th}$ coin flip, where $M$ is an $(n+1) \times (k+1)$ matrix, which we will use to store the probabilities. That is, the subproblems consist of computing the probability of obtaining $j$ heads when $i$ distinct coins are tossed for $j \in \{1, \ldots, k\}$ and $i \in \{1, \ldots, n\}$. Fill in $M$ from the top down, left to right such that all sub-problems needed for entry $M[i, j]$ have already been computed when we compute $M[i, j]$. The solution we are looking for is $M[n, k]$. The probability of obtaining $k$ heads in $n$ coin tosses (with $n$ distinct biased coins) is then

$$M[n, k] = p_n \cdot M[n-1, k-1] + (1-p_n) \cdot M[n-1, k]$$

In general, the probability that the $j$ heads are obtained in $k$ coin tosses is:

$$M[i, j] = p_i \cdot M[i-1, j-1] + (1-p_i) \cdot M[i-1, j]$$

Indeed, the probability of the $j^{th}$ head being obtained on the $i^{th}$ coin toss depends on all coin tosses that occurred before that. In particular, if the $j^{th}$ head is obtained on the $i^{th}$ coin toss or if it was obtained before the $i^{th}$ coin toss.

Boundary conditions:

- $M[0, 0] = 1$ , since the event of not obtaining a heads when no coins are tossed is a sure event.
- If $i < j$, then $M[i, j] = 0$. Indeed, obtaining a greater number of heads than there are coin tosses is impossible.

Pseudocode:

```
CoinTossProb($n, k, p = \{p_1, p_2, \ldots, p_n\}$)
    $M$ = matrix($(n+1) \times (k+1)$)                // Initializing DP matrix/table
    Initialize $M[0, 0] = 1$                              // Boundary condition
    for $i = 1$ to $n$ do
        for $j = 1$ to $k$ do
            if $(i < j)$ or $(j < 0)$ then
                $M[i, j] = 0$
            else
                $M[i, j] = p_i \cdot M[i-1, j-1] + (1-p_i) \cdot M[i-1, j]$
            end if
        end for
    end for
    return $M[n, k]$
end
```

Space: The algorithm has an additional space requirement of $\Theta(nk)$ for the $(n+1) \times (k+1)$ matrix $M$.

Running Time: Initializing the $(n+1)\times(k+1)$ matrix at line 2 takes $O(nk)$ time. Line 3 takes $O(1)$ time. The for-loop at lines 4 through 12 requires $O(nk)$ time since there are $k$ nested iterations within a for-loop that iterates $n$ times, where each iteration requires only constant time since multiplying and adding two numbers takes $O(1)$ time and looking up pre-computed probabilities also only requires constant time. Therefore, the total running time is

$$O(nk) + O(1) + O(nk) \; = \; O(nk)$$

Correctness: Induct on $n$ and $k$.
**Base case** ($n = k = 0$ and $n = k = 1$): For $n = k = 0$ , `CoinTossProb`$(0, 0)$ returns 1, which aligns with the probability of the event of not obtaining a heads when no coins are tossed. For $n = k = 1$, the algorithm returns

$$M[1,1] = p_1 \cdot \underbrace{M[0,0]}_{=1} + (1-p_1)\cdot \underbrace{M[0,1]}_{=0\ (i<j)} = p_1$$

$$\implies M[1,1] = p_1$$

That is, it returns the probability of obtaining a head on the first coin toss, which is the definition of $p_1$.

**Induction Hypothesis**: For all $0 \le i \le n-1$ and all $0 \le j \le k-1$ assume the algorithm correctly computes the probability of obtaining $j$ heads in $i$ coin tosses.

**Inductive Step**: Now consider running `CoinTossProb` on inputs $n$ and $k$ (i.e., `CoinTossProb`$(n, k, \{p_1, p_2, \dots, p_n\})$). The algorithm computes the following:

$$M[n,k] = p_n \cdot \underbrace{M[n-1,k-1]}_{\text{Correct by induc. hyp.}} + (1-p_n)\cdot M[n-1,k]$$

$$= p_n \cdot \underbrace{M[n-1,k-1]}_{\text{Correct by induc. hyp.}} + (1-p_n)\cdot \left(p_{n-1}\underbrace{M[n-2,k-1]}_{\text{Correct by induc. hyp.}} + (1-p_{n-1})M[n-2,k]\right)$$

$$= p_n \underbrace{M[n-1,k-1]}_{\text{Correct by induc. hyp.}} + (1-p_n)\Big(p_{n-1}\underbrace{M[n-2,k-1]}_{\text{Correct by induc. hyp.}} +$$

$$+ (1-p_{n-1})(p_{n-2}\underbrace{M[n-3,k-1]}_{\text{Correct by ind. hyp.}} + (1-p_{n-2})M[n-3,k])\Big)$$

$$\vdots \qquad\qquad \vdots \qquad\qquad \vdots$$

. . . and so on. Note that the terms in blue do not satisfy the Induction Hypothesis since the second dimension index in each is greater than $k-1$; this index must be less than $k$ for the induction hypothesis to hold. However, as the computation continues, the only term left that will not satisfy the induction hypothesis will be such that first dimension index is less than the second dimension index. That is, it will eventually reach $M[n-l,k]$ for some integer $l > 0$ where $n-l < k$. From our argument above, this probability is 0 since the event of obtaining more heads than there are tosses is impossible. Once this term vanishes, all terms will satisfy the induction hypothesis, thus giving a correct computation of $M[n,k]$.

## Problem 4

*I provide two algorithms for this problem. An alternative algorithm is provided after this one on page 10.*

This can be solved using a DP approach. In particular, let $OPT(i)$ be the maximum expected profit optimized over the first $i$ of the locations, i.e. the subset $\{1, 2, \ldots, i\}$. If a hotel is opened at the $n^{th}$ location, a profit of $p_n$ is gained and the $n^{th}$ location and all locations that are a distance of less than $k$ miles away are extracted from the set of feasible locations (locations that satisfy the given constraints). We then proceed to optimize over the remaining feasible locations. On the other hand, if a hotel is not opened at location $n$, then no profit is realized, extract the $n^{th}$ location and continue to optimize over the remaining $n-1$ locations.

A matrix $F$ will be used to indicate which of the locations satisfy the constraint that no two hotels should be a distance of less than $k$ miles apart. In particular, $F[i,j] = 1$ indicates that locations $i$ and $j$ are at least $k$ miles apart, while $F[i,j] = 0$ indicates that locations $i$ and $j$ are less than $k$ miles apart. That is,

$$F[i,j] = \begin{cases} 1 \;\; , & |m_i - m_j| \geq k \\ 0 \;\; , & \text{otherwise} \end{cases}$$

If location $i$ is selected, the algorithm will subsequently maximize over the remaining locations that satisfy the constraints. Specifically, it optimizes over all locations $j$ such that $F[i,j] = 1$. The recursive DP algorithm is then

$$OPT(i) = \max \begin{cases} OPT(i-1) & , \text{ if hotel is not opened at location } i \\ \max\limits_{\forall j : F[i,j]=1} \{p_i + OPT(j)\} & , \text{ if hotel is opened at location } i \end{cases}$$

More succinctly,

$$OPT(i) = \max \left\{ OPT(i-1), \max_{\forall j : F[i,j]=1} \{p_i + OPT(j)\} \right\}$$

Set a boundary condition $OPT(0) = 0$ , since a profit cannot be made with no available locations. An array $M$ will be used to store the values of $OPT(i)$, which will be filled in from left to right.

**Pseudocode, space requirement analysis, and running time analysis are all provided on the following page.**

```
1:  Opt-Locations($n, k, m = [m_1, m_2, \ldots, m_n], p = [p_1, p_2, \ldots, p_n]$)
2:      $M$ = array($n + 1$)                         // Define and initialize DP array
3:      $F$ = matrix($n, n$)                         // Define and initialize feasibility matrix
4:      Initialize $M[0] = 0$                        // Boundary condition
5:      for $i = 1$ to $n$ do
6:          for $j > i$ and $j \leq n$ do
7:              if $|m_i - m_j| \geq k$ then
8:                  $F[i, j] = 1$
9:              else
10:                 $F[i, j] = 0$
11:             end if
12:         end for
13:     end for
14:     for $i = 1$ to $n$ do
15:         $M[i] = \max\left\{M[i-1], \max_{\forall j : F[i,j]=1} \{p_i + M[j]\}\right\}$
16:     end for
17:     return $M[n]$
18: end
```

Space: The algorithm has space requirement of $\Theta(n)$ for the DP array $M$ of size $n + 1$ and additional space requirement of $\Theta(n^2)$ for the $n \times n$ matrix $F$ used to indicate whether constraints are satisfied.

Running Time: Initializing the DP array (line 2) takes $O(n)$ time. Initializing the indicator matrix (line 3) takes $O(n^2)$ time. Initializing the boundary condition (line 4) takes constant time. The nested for-loop (lines 5-10) requires $O(n^2)$ time to run. The recursion at lines 12-14 iterates $n$ times, each of which takes constant time since each of the values were computed in earlier steps. The total running time of the algorithm is then $O(n^2)$.

***Alternative Algorithm/Pseudocode for Problem 4*:**
This alternative algorithm is almost identical to the one provided above, except for tracking of the constraints - which was the bottleneck of the previous algorithm. Here, we use an array instead of the matrix and update it for every iteration. Everything else is the same. $F$ is defined:

$$F[i] = \begin{cases} 1 \;, & |m_i - m_j| \geq k \\ 0 \;, & \text{otherwise} \end{cases}$$

The entries of $F[i]$ are replaced in every iteration. Everything else is identical to the previous algorithm (above).

```
Opt-Locations($n, k, m = [m_1, m_2, \ldots, m_n], p = [p_1, p_2, \ldots, p_n]$)
    $M$ = array($n + 1$)                         // Define and initialize DP array
    $F$ = array($n$)                             // Define and initialize feasibility array
    Initialize $M[0] = 0$                        // Boundary condition
    for $i = 1$ to $n$ do                        // if-statement updates the feasibility array for each $i$
        if $|m_i - m_j| \geq k$ then
            $F[i] = 1$
        else
            $F[i] = 0$
        end if
        $M[i] = \max\left\{M[i-1],\ \max_{\forall j:F[i]=1}\{p_i + M[j]\}\right\}$
    end for
    return $M[n]$
end
```

Space: This alternative algorithm has space requirement of $\Theta(n)$ for the DP array $M$ of size $n + 1$ and additional space requirement of $\Theta(n)$ for the array $F$ used to indicate whether constraints are satisfied.

Running Time: Initializing the DP array takes $O(n)$ time. Initializing the indicator array $F$ takes $O(n)$ time. Initializing the boundary condition takes constant time. The for-loop requires $O(n)$ time to run. The recursion at line 11 iterates $n$ times, each of which takes constant time since each of the values were computed in earlier steps. Therefore, the total running time of the *alternative* algorithm is then $O(n)$.

## Problem 5

This can be solved using a DP approach. In particular, let $OPT(r,i)$ denote the maximum sum starting at the top of the pyramid and ending at row $r$ and position $i$ in the pyramid. Let $P$ denote the pyramid input. The traversal of the pyramid resembles that of a binary tree, so we can relate this problem as such. Supposing $P[r,i]$ is the current optimal position in the pyramid, the algorithm decides whether to move to the position directly below the current position (to $P[r+1,i]$) or move to the diagonally to the right (to $P[r+1,i+1]$). The recursive algorithm can then be expressed as

$$OPT(r,i) = P(r,i) + \max\{OPT(r+1,i), OPT(r+1,i+1)\}$$

$M$ will be used to store the values of $OPT(r,i)$, which will be filled in from the top down. Boundary condition: $M[1,1] = P[1,1]$

```
MaxSum($P$)
    $n = depth(P)$
    $M = P$
    $path = [\ ]$                                   // declare array to record optimal path
    $path[1] = (1,1)$                     // initialize 1st path position to top of pyramid
    Initialize $M[1,1] = P[1,1]$                                  // Boundary condition
    for $r = 1$ to $n$ do
        for $i = 1$ to $r$ do
            $M[r,i] = P[r,i] + \max\{M[r+1,i], M[r+1,i+1]\}$
            if $M[r,i] == P[r,i] + M[r+1,i]$ then   // if-statement records optimal path
                $path[i] = (r+1,i)$
            else if $M[r,i] == P[r,i] + M[r+1,i+1]$ then
                $path[i] = (r+1,i+1)$
            end if
        end for
    end for
    return $M[n,r]$ , $path$                     // Returns max sum and the optimal path
end
```

Space: Additional space requirement for for $M$ is $\Theta(n^2)$ and the space requirement for the *path* array is $\Theta(n)$. The overall space requirement is then $\Theta(n^2)$.

Running Time: Duplicating the pyramid takes $O(n^2)$ time. and the nested for-loop takes $O(n^2)$ time. Therefore, the total running time is $O(n^2)$.