CSOR W4231 Analysis of Algorithms I - Spring 2021
Homework #3

Joseph High - `jph2185`

March 17, 2021

Problem 1

Two modified versions of the DFS algorithm are constructed and used to traverse the input tree, iteratively matching unmatched nodes. The first modified DFS algorithm (`DFS-Leaves`), traverses the tree to match all leaf nodes and the neighbor of each leaf node provided that they have not yet been matched. The second modified version of DFS algorithm (`DFS-Match`) traverses the tree to match the remaining internal nodes that have not yet been matched. The output of the algorithm is either `True`, indicating that the tree has a perfect matching, or `False`, indicating that no perfect matching exists.

The main procedure, `PerfectMatchTree`, initializes an array *matched* that records which nodes have been matched, using a binary $\{0, 1\}$ indicator (0 if node is not matched, 1 if matched). Next, the algorithm evaluates the tree and its attributes to determine whether it has a perfect matching. If $|V| = 0$, the algorithm returns `True`, since a graph on zero nodes vacuously contains a perfect matching. If $|V|$ is odd, the algorithm returns `False` since a perfect matching can only exist if the graph has an even number of vertices. Otherwise, the algorithm runs `DFS-Leaves` first and then `DFS-Match`. If the sum of all entries in the *matched* array is $|V|$ then a perfect matching exists and the algorithm returns `True`; otherwise, a perfect matching does not exist and the algorithm returns `False`.

Pseudocode and correctness provided on the following two pages.

Running Time: Referring to the run-time comments in the pseudocode (in blue), lines 2 and 3 require constant time, the for-loop to initialize the *matched* array (lines 4-6) requires $O(n)$ time, and lines 7 through 10 requires constant time. The `DFS-Leaves` procedure requires the usual $O(n+m)$ run time for the original DFS procedure and $O(n)$ time to match all leaves, for a total run time of $O(n+m)$ for `DFS-Leaves`. The `DFS-Match` procedure also requires the usual $O(n+m)$ run time for the original DFS procedure and $O(m)$ time for the for-loop at lines 37-42 to run for *each* node; indeed, for each node u, the for-loop iterates $deg(u)$ times. The run time of `DFS-Match` is therefore $O(n+m)$. Hence, the total run time is

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

```
1:  PerfectMatchTree(T = (V, E))
2:      n = size(V)                          // Define number of nodes in T. Running time: O(1)
3:      matched = [ ]                        // Initialize empty array. Running time: O(1)
4:      for u ∈ V do                         // Initialize each entry in array to False. Running time: O(n)
5:          matched[u] = 0
6:      end for
7:      if n == 0 then                                                  // Running time: O(1)
8:          return 'True'
9:      else if n is odd then                                           // Running time: O(1)
10:         return 'False'
11:     else
12:         Run DFS-Leaves(T)                                           // Total running time: O(n + m)
13:         Run DFS-Match(T)                                            // Total running time: O(n + m)
14:         if sum(matched) == n then                                   // Running time: O(n)
15:             return 'True'
16:         else
17:             return 'False'
18:         end if
19:     end if
20: end
21:
22: DFS-Leaves(T = (V, E))                  // Only the modified portion shown
23: //Add the following to the beginning of the Search procedure in the DFS pseudocode
24:     if degree(u) == 1 and matched[u] == 0 then                      // Running time: O(1)
25:         if matched[neighbor(u)] == 0 then
26:             matched[u] = 1
27:             matched[neighbor(u)] = 1
28:         else
29:             return 'False'
30:         end if
31:     end if
32:     return matched                      // Add to the end of the Search procedure
33: end
34:
35: DFS-Match(T = (V, E))                   // Only the modified portion shown
36: //Add the following to the beginning of the Search procedure in the DFS pseudocode
37:     for (u, v) ∈ E do                   // Iterates deg(u) times for each u ∈ V
38:         if matched[u] == 0 and matched[v] == 0 then
39:             matched[u] = 1
40:             matched[v] = 1
41:         end if
42:     end for
43:     return matched                      // Add to the end of the Search procedure
44: end
```

Because trees are such that $|E| = |V| - 1$, the run time is $O(n + m) = O(2n - 1) = O(n)$.

Correctness: For $n = 0$, the algorithm returns `True`, since a graph on zero nodes vacuously contains a perfect matching. If $|V|$ is odd, the algorithm returns `False` since a perfect matching can only exist if the graph has an even number of vertices. Otherwise, the algorithm first matches all leaf nodes and their neighbors, provided that they have not yet been matched. Because leaf nodes have exactly one neighbor, a perfect matching necessarily includes all edges incident on leaf nodes. If the neighbor of any unmatched leaf node has already been matched then a perfect matching does not exist. The algorithm adds a 1 to $matched[k]$ if vertex $V[k]$ is matched, and a 0 otherwise. If the sum of all entries in the *matched* array is $|V|$ then a perfect matching exists and the algorithm returns `True`; otherwise, a perfect matching does not exist and the algorithm returns `False`. Indeed, a graph has a perfect matching if and only if *every* node in V is matched by exactly one edge.

Alternative Method

An alternative approach to this problem is to use a "greedy" or DP approach (which is likely what we were intended to do, but I did not realize it until it was too late). In particular, an alternative algorithm that evaluates whether sub-trees of the original tree have a perfect matching, where the set of sub-trees consists of nodes with neighboring leaf nodes, all of their adjacent leaf nodes, and their edges. The algorithm partitions the tree into sub-trees consisting of all the nodes with neighboring leaf nodes and their adjacent leaf nodes, and their edges. The algorithm deletes all leaves and their neighbors from the tree and stores them to be evaluated later. Nodes with degree > 1 are stored with all of their leaf neighbors. All edges incident to the nodes deleted from the tree are also deleted. If the remaining graph has nodes and edges after deleting the first round of sub-trees, the algorithm repeats the above. Indeed, it necessarily has leaf nodes since it was a tree to begin with. Once there are no nodes to partition, the algorithm evaluates each sub-tree for a perfect matching, and storing the results. The sub-trees have a perfect matching if and only if the sub-trees have exactly one leaf node; if any node has more than one leaf node, then a perfect matching cannot exist because the parent node can be matched at most once Thus, if a sub-tree has more than one leaf node, return `False`. If all sub-trees have a perfect matching, the algorithm returns `True`.

Problem 2

A DP approach can be used to solve this problem. At a high-level, the algorithm counts the number of symbols in S' that sequentially match S (in order). If the number of sequentially matched symbols is equal to the number of symbols in S' ($= m$), the algorithm returns `True`. Otherwise, the algorithm returns `False`. The subproblems consist of comparing the symbol $S'[k]$ to the symbol $S[t]$ for fixed k and t, where $k \leq t$. The subproblems are iteratively solved one by one, in increasing order. If $S'[k] = S[t]$ for some fixed values k and t, then we add 1 to the count, which is stored in an array M of size $n + 1$ (includes the boundary condition), from the bottom up. In particular,

$$M[i] = \begin{cases} M[i-1] + 1 \ , & \text{if } S[i] = S'[j] \\ M[i-1] \quad\quad , & \text{if } S[i] \neq S'[j] \end{cases}$$

Boundary condition: $M[0] = 0$. Pseudocode is provided below.

```
1:  SubSequence(S, S')
2:      n = length(S)
3:      m = length(S')
4:      if m > n then
5:          return 'False'
6:      end if
7:      if m == 0 then
8:          return 'True'
9:      end if
10:     Let M = array(n + 1)
11:     M[0] = 0
12:     i = 1
13:     j = 1
14:     while i ≤ n and j ≤ m do
15:         if S[i] == S'[j] then
16:             M[i] = M[i − 1] + 1
17:             i = i + 1
18:             j = j + 1
19:         else if S[i] ≠ S'[j] then
20:             M[i] = M[i − 1]
21:             i = i + 1
22:         end if
23:     end while
24:     if M[n] == m then
25:         return 'True'
26:     else return 'False'
27:     end if
28: end
```

Running Time: The symbols in the sequence S' are compared to the symbols in S over a maximum of n iterations. Indeed, there are n symbols in S, and if $m > n$ then no comparison is made since then S' cannot be a subsequence of S. Therefore, the running time is $O(n)$.

Space: The algorithm requires an additional $\Theta(n)$ space for the array M. The array M is filled from the bottom up.

Correctness: If $m = 0$, the algorithm returns `True` regardless of the size of S since an empty sequence is a subsequence of all sequences. Otherwise, the algorithm sequentially and iteratively compares the symbols in S' to the symbols in S, adding 1 to the number of matched symbols if a given subproblem has a truth value of `True`. If $S'[k] = S[t]$ for some fixed values k and t, the algorithm increments both S' and S to their next symbols so that $S'[k+1]$ is compared to $S[t+1]$; otherwise, it increments to the subproblem comparing $S'[k]$ to $S[t+1]$. Therefore, if elements S' are all present in the same order in S, the algorithm will account for it.

Problem 3

Construct a modified version of Dijkstra's algorithm such that it accounts for both the edge weights and the vertex costs. In particular, for each $(u, v) \in E$ where $u \in S$ and $v \in V - S$, pick v such that $dist[u] + w_{uv} + c_v$ is a minimum among all nodes in $V - S$. In particular, modify the `Update` procedure such that $dist[v]$ maintains the minimum weight and cost from s to u, for all $u \in V - S$. Additionally, modify the `Initialize` procedure to initialize $dist[s] = c_s$. Last, `Dijkstra` is modified to return the $dist$ array.

Pseudocode:

```
1: Note: Only the modified portion of each procedure is shown
2: Dijkstra-Modified(G = (V, E, w, c), s):
3:     Initialize(G, s)
4:     ⋮
5:     ...Update(u, v)
6:     ⋮
7:     return dist
8: end
9:
10: Initialize(G, s)
11:     ⋮
12:     dist[s] = c_s
13: end
14:
15: Update(u, v):
16:     if dist[v] > dist[u] + w_uv + c_v then
17:         dist[v] = dist[u] + w_uv + c_v
18:         prev[v] = u
19:     end if
20: end
```

Running Time: The running time is the same as the running time of Dijkstra's algorithm. That is, the run time is $O(n \log n + m \log n)$.

Correctness: Correctness follows from the correctness/validity of Dijkstra's algorithm, which computes, for all $u \in V$, the weight of the shortest $s - u$ path. Therefore, under the modified definition of the weight of a path, this modified version of Dijkstra's algorithm computes and returns the weight of the shortest $s - u$ path, for all $u \in V$.

Problem 4

This can be solved using a greedy approach. In particular, we can progressively choose a locally optimal position for each of the guards. In doing so, the position of all guards will be globally optimal. That is, the minimum number of guards will be used to protect all paintings (to be shown in proof of correctness). The first guard should be placed at either $x_1 + 1$ or $x_n - 1$ so that paintings on both his left and his right side, within a distance of 1, are protected (if any paintings exist in those positions); this ensures that as many paintings as possible are being protected by the first guard. Without loss of generality, the first guard will be placed at position x_1+1 on the line L. If any paintings are positioned before x_1+2 on the line, we do not need to place any additional guards to protect those paintings since they are already being protected by the first guard. Suppose the positions of paintings $x_2, \ldots, x_k$ are less than $x_1 + 2$ (i.e., protected by the first guard), with $k \in \{2, \ldots, n\}$. The second guard should be placed using a similar approach. That is, place the second guard at position $x_{k+1} + 1$ so that he protects all paintings positioned on the interval $[x_{k+1}, x_{k+1} + 2]$, on the line L. Continue in this way until all paintings are guarded.

```
1:  MinNumGuards({x_1, x_2, ..., x_n})
2:      X = array(n)
3:      for i = 1 to n do
4:          X[i] = x_i
5:      end for
6:      guardPositions = MinGuardPlacement(X)
7:      minNumGuards = length(guardPositions)
8:      return guardPositions, minNumGuards    // Returns positions and # of guards
9:  end
10:
11:  MinGuardPlacement(X)
12:     guard = [ ]                          // Initialize empty array for guard positions.
13:     guard[1] = X[1] + 1
14:     j = 1                                // j used to used to iterate through each guard.
15:     for k = 2 to n do
16:         if X[k] > guard[j] + 1 then
17:             guard[j + 1] = X[k] + 1      // Place next guard at locally optimal position.
18:             j = j + 1                    // Increment j for next guard position.
19:         end if
20:     end for
21:     return guard                         // Returns guard positions
22: end
```

Running Time: Constructing the array X in the `MinNumGuards` procedure requires $O(n)$ time. The for-loop in the `MinGuardPlacement` procedure iterates $n - 1$ times and each of the entries of X and *guard* are evaluated exactly one time. Thus, the `MinGuardPlacement` procedure requires $O(n)$ time. Therefore, the total run-time is $O(n)$.

Correctness: Let g_i denote the position of guard i for $i \in \{1, \ldots, n\}$. We induct on n .
Base Case ($n = 1$): Exactly one guard is used to protect the painting at position x_1, and is placed at position $g_1 = x_1 + 1$. Clearly, the minimum number of guards required to protect the one painting is 1.
Induction Hypothesis: Assume the algorithm outputs the minimum number of guards required to protect $n - 1$ paintings at positions $x_1, x_2, \ldots, x_{n-1}$.
Inductive Step: Let K denote the minimum number of guard to protect the $n - 1$ paintings. That is, suppose that paintings at positions $x_i, \ldots, x_{n-1}$ are protected by the K^{th} guard, for $i \in \{2, \ldots, n - 1\}$. If $x_n \leq g_K + 1$, then the number of guards required to protect all n paintings output by the algorithm is K, which is the minimum number of guards required to protect the first $n - 1$ paintings, from the inductive hypothesis. Thus, the minimum number of guards required for all n paintings when $x_n \leq g_K + 1$ is K. Therefore, the algorithm outputs the minimum number of guards required for all n paintings when $x_n \leq g_K + 1$. On the other hand, if $x_n > g_K + 1$, the algorithm places a guard at position $x_n + 1$ since the painting at position x_n is too far from the guard at position g_k to be able to protect it. That is, an additional guard must be used to protect the n^{th} painting. Then the algorithm outputs $K + 1$ and the positions of each of the guards. Because K is the minimum number of guards required for the first $n - 1$ paintings, then $K + 1$ is the minimum number required for all n paintings when $x_n > g_K + 1$. Therefore, the algorithm outputs the minimum number of guards in all cases.

Problem 5

This can be solved using a DP approach. Let $OPT(j)$ be the minimum total cost to store books in some libraries. We know that a copy is always stored at L_n. Suppose i is the largest index less than n such that a copy is stored at L_i in the optimal solution, then $OPT(n) = c_n + OPT(n-1)$.

Let the subproblems be such that we must choose $L_1, \ldots, L_j$ to store copies of the book at. There are two cases: one in which we incur a storage cost c_j for storing at library L_j, or there is a user delay penalty $(k-j)$ if a copy is not stored at L_j. That is,

$$OPT(j) = \min \begin{cases} c_j + OPT(j-1) & , \text{ if a copy is stored at } L_k \\ (i-j) + OPT(j-1) & , \text{ if a copy is not stored at } L_k \text{ for some } i > j \end{cases}$$

Let $d_{j,i} = i - j$ denote the user delay when a book is not available at L_j, but available at L_i, for $i > j$. We then have

$$OPT(j) = \min\{c_j + OPT(j-1),\ d_{j,i} + OPT(j-1)\}$$

Set a boundary condition $OPT(0) = 0$. Use an array M to store the values of $OPT(j)$.

Pseudocode provided on the next page - does not fit on this page.

Running Time: There are n subproblems. The bottleneck of the algorithm are the two nested for-loops, each of which iterates n times, in the `MinStorageCost` procedure. Therefore, the run time is $O(n^2)$.

Space: Additional space requirement is $\Theta(n)$ for the array M.

Correctness: The recurrence provided in the algorithm computes the minimum cost between storing a copy at library L_j or not storing a copy at L_j, for $j \in \{1, \ldots, n\}$. The minimal cost at iteration j depends on both the cost at L_j, or the user delay at server j and on the minimum cost at iteration $j-1$. Because each $M[j-1]$ returns the minimum cost, $M[j$ must also return the minimum cost. To show this, induct on j :
Base case ($j = 0$): $M[0] = 0$ is trivially the minimum cost for $j = 0$.
Induction Hypothesis: Suppose $M[j-1]$ returns the minimum cost for $L_1, \ldots, L_{j-1}$.
Inductive Step: $M[j] = \min\{c_j + M[j-1],\ d_{j,i} + M[j-1]\}$. From the induction hypothesis, $M[j-1]$ returns the minimum cost, then computing the minimum reduces to $\min\{c_j, d_{j,i}\}$ which will return the minimum. Thus, $M[j]$ returns the minimum total cost for $L_1, \ldots, L_j$. Similarly, $M[n]$ returns the minimum total cost for $L_1, \ldots, L_n$

Pseudocode provided on the next page.

Algorithm 1 Computes the min total cost and the set of libraries that achieve the min.

```
1:  MinStorageCost(n, C = [c_1, ..., c_n]):          // Computes the total min cost
2:      Initialize array M[0, ..., n]
3:      Set M[0] = 0
4:      for j = 1 to n do
5:          for i ≥ j do
6:              d_{j,i} = i − j                      // Compute and store all possible user delay values
7:          end for
8:      end for
9:      for j = 1 to n − 1 do                        // Computes OPT(j) for L_1, ..., L_{n−1}.
10:         M[j] = min{c_j + M[j − 1], d_{j,i} + M[j − 1]}
11:     end for
12:     M[n] = c_n + M[n − 1]                        // Since a copy is always stored at L_n
13:     return M[n]
14: end
15:
16:  OptLibraries(j, M):                            // Computes the set of libraries that achieve the min
17:     if j == 0 then return
18:     else
19:         if M[j] == c_j + M[j − 1] then
20:             Output L_j
21:             OptLibraries(j − 1)
22:         else if M[j] == d_{j,i} + M[j − 1] then
23:             OptLibraries(j − 1)
24:         end if
25:     end if
26: end
```