Assignment 4 - Hady Ibrahim (400377576)
SFWRENG 2CO3: Data Structures and Algorithms–Winter 2023

Deadline: March 19, 2023

Department of Computing and Software
McMaster University

Please read the *Course Outline* for the general policies related to assignments.

Plagiarism is a *serious academic offense* and will be handled accordingly.
All suspicions will be reported to the *Office of Academic Integrity*
(in accordance with the Academic Integrity Policy).

This assignment is an *individual* assignment: do not submit work of others. All parts of your submission *must* be your own work and be based on your own ideas and conclusions. Only *discuss or share* any parts of your submissions with your TA or instructor. You are *responsible for protecting* your work: you are strongly advised to password-protect and lock your electronic devices (e.g., laptop) and to not share your logins with partners or friends! If you *submit* work, then you are certifying that you have completed the work for this assignment by yourself. By submitting work, you agree to automated and manual plagiarism checking of all submitted work.

Late submission policy. Late submissions will receive a late penalty of 20% on the score per day late (with a five hour grace period on the first day, e.g., to deal with technical issues) and submissions five days (or more) past the due date are not accepted. In case of technical issues while submitting, contact the instructor *before* the deadline.

Problem 1. Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$. Breadth-first search and depth-first search both have a runtime complexity of $\sim|\mathcal{N}| + |\mathcal{E}|$ if we represent the graph with an adjacency list.

P1.1. What is the runtime complexity of breadth-first search and depth-first search when we use the matrix representation? Explain your answer.

For BFS, we typically use a queue to store the vertices to be processed. In each iteration, we remove a vertex from the front of the queue, add its neighbors to the back of the queue if they have not been visited yet, and mark the removed vertex as visited. To determine the neighbors of a vertex, we need to examine the corresponding row of the adjacency matrix. Thus, for each vertex, we need to scan the corresponding row of the adjacency matrix to find its neighbors. Since there are $|N|$ vertices, each with $|N|$ entries in the adjacency matrix, the time complexity of BFS using the adjacency matrix representation is $O(|N|^2)$.

For DFS, we typically use a stack to store the vertices to be processed. In each iteration, we remove a vertex from the top of the stack, add its unvisited neighbors to the stack, and mark the removed vertex as visited. To determine the neighbors of a vertex, we need to examine the corresponding row of the adjacency matrix. Thus, for each vertex, we need to scan the corresponding row of the adjacency matrix to find its neighbors. Since there are $|N|$ vertices, each with $|N|$ entries in the adjacency matrix, the time complexity of DFS using the adjacency matrix representation is also $O(|N|^2)$.

P1.2. Consider the *ordered edge list* representation that represents a graph by an *ordered* array A that holds $|\mathcal{E}|$ edges. The order of edges is as follows: edge $(n_1, m_1) \in A$ comes before $(n_2, m_2) \in A$ if $\text{id}(n_1) < \text{id}(n_2)$ or if $\text{id}(n_1) = \text{id}(n_1) \land \text{id}(m_1) \leq \text{id}(m_2)$. What is the runtime complexity of breadth-first search and depth-first search when we use the *ordered edge list* representation? Explain your answer.

In the ordered edge list represenation, since edges are sorted, we know all edges that start at n_1 will be at the start of the array, followed by n_2 and so on. We can then use binary search to find the first edge of the list of edges that start at n_x where x is the node number.

For BFS, we know that we traverse every node once and every edge once (covered in class). Basically, we start by visiting the start node and then all its neighbours. To visit its neighbours, we need to search for the edges that have the starting node as the first node in the tuple. To find these nodes, it takes $\sim log_2|E|$ time where the length of the list to search is $|E|$. We repeat this process for every node, meaning we visit every node once and every edge once and need to call a $\sim log_2|E|$ operation once per node. This gives a complexity of $\sim|N|log_2|E| + |E|$.

For DFS, we also know that we traverse every node and edge once. We do this by starting at the start node and recursively visiting all its neighbours. Again, to visit the neighbours will require to find the edges in the ordered list, which requires the binary search of $\sim log_2|E|$. We repeat this process for all nodes that are visited, thus meaning we use the binary search once per node. Since this is the case, the overall runtime complexity is also $\sim|N|log_2|E| + |E|$ as we traverse every node and edge once and for every node, need to perform a $\sim log_2|E|$ complexity search.

Problem 2. Consider a directed acyclic graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ with a source node $m \in \mathcal{N}$ and target node $n \in \mathcal{N}$.

P2.1. Provide an algorithm that computes the number of paths starting at m and ending at n.

Assuming an adjacency representation of the graph.

```
Algorithm DFS($start$, $dest$, $storedPaths$):
 1: if $start = 1$ then
 2:     return 1
 3: end if
 4: $path = 0$
 5: for $n \in adj[start]$ do
 6:     if $(storedPaths.containsKey(n))$ then
 7:         $path+ = storedPaths.get(n)$
 8:     else
 9:         $paths+ =$DFS($start$, $dest$, $storedPaths$)
10:     end if
11: end for
12: $storedPaths.putIfAbsent(v, path)$
13: return $paths$
```

Call this via

```
Algorithm Main():
 1: $graph := newGraph()$ “initialized graph with edges and nodes already set”
 2: $Map < Integer, Integer > storedPaths = newHashMap <> ()$
 3: print DFS($start\_node\_id, destination\_node\_id, storedPaths$)
```

Graph representation: adjacency representation. The algorithm runs DFS from the start node. For each node in its adjacency list (neighbours), it runs another DFS. This happens recursively. This algorithm goes over what is considered a "marked node" in normal DFS as we want to account for every single path, not only unique ones. Each DFS counts the number of paths to the destination node by summing the number of paths to the destination node all of its children have. The base case is when reaching the destination node, we have 1 path. Using this recursion, we can sum all the paths to the destination node and return that.

As I use dynamic programming to keep track of the solution to "subproblems" (a subproblem being finding the paths to the destination node from specific node), we only ever need to compute the number of paths from a single node to the destination once. After that we can fetch that result from the storedPaths map. This means that for every node, we perform a DFS. We know DFS using adjacency represenation is $\sim|N| + |E|$, so if we do this for each node, the complexity is $\sim|N|^2 + |N||E|$

P2.2. We say that the directed acyclic graph $\mathcal{G}$ has a *bottleneck* if there is a node b, distinct from source m and target n, such that all paths from m to n go through node b. Write an algorithm that returns `true` if and only if $\mathcal{G}$ has such a bottleneck. You may assume that there is at-least one path from m to n.

```
Algorithm BOTTLENECK($m, n$):
1: $G = Graph(V, E)$ “Initialized graph with all nodes V and edges E”
2: $S = G.DFS(m)$ “All nodes reachable from m”
3: $G_{rev} = (V, e \in G | (n_1, n_2) \in E : (n_2, n_1))$ “Same graph, but with all edges flipped”
4: $T = G_{rev}.DFS(n)$ “All nodes that can reach n”
5: $B = Intersection(S, T)$ “All nodes that are reachable from m and can reach n”
6: for $x \in B$ do
7:     if $G.AllPathsContain(m, n, x) == false$ then
8:         $B.remove(x)$ “Traverse all paths from m to n, and if it doesn’t pass x, remove x from B”
9:     end if
10: end for
11: if $B.empty$ then
12:     return FALSE
13: end if
14: return TRUE
```

I am using adjacency representation again.

DFS on line 2: Is a $O(\sim|N| + |E|)$ as it is a normal DFS where we keep track of the nodes reachable from m and return them. As mentioned in lecture, DFS is $O(\sim|N| + |E|)$.

Reversing the edges on line 3: Is a $O(\sim|E|)$ as you need to swap the nodes for every edge.

DFS on line 4: With the reverse graph, do the same as line 2 where you find all reachable nodes. This will get all nodes that can reach n in $O(\sim|N| + |E|)$.

Intersection on line 5: Goes through all of S and T and adds nodes that are in both sets. This is $O(\sim|N|)$ as worst-case, all nodes are in both sets, so we must traverse the sets fully.

Traverse paths of lines 6-10: For every node in B, which we determined worst-case is $|N|$ nodes, we must traverse all paths, which can again be accomplished using DFS, which is $O(\sim|N| + |E|)$. Thus the complexity here is $O(\sim|N|^2 + |N||E|)$

Removing from B is an $O(\sim|B|)$, which is worst-case $O(\sim|N|)$, and checking if B is empty is $O(\sim 1)$.

Thus we can see, lines 6-10 are the largest complexity, so the overall complexity for this algorithm is $O(\sim|N|^2 + |N||E|)$

For each question, explain why your algorithm is correct, what the complexity of your algorithm is, and which graph representation you use.

Problem 3. Let n be a positive integer and consider two $n \times n$ matrices M_1 and M_2. The *Boolean matrix product* of M_1 and M_2, denoted by $M' := M_1 \otimes M_2$, is the $n \times n$ matrix M' in which $M'[i, j]$ is:

$$M'[i, j] = (M_1[i, 0] \wedge M_2[0, j]) \vee (M_1[i, 1] \wedge M_2[1, j]) \vee \cdots \vee (M_1[i, n-1] \wedge M_2[n-1, j]).$$

Now consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ implemented via the matrix representation M with $n = |\mathcal{N}|$.

P3.1. Let $M' = M \otimes M$. What does the value $M'[i, j]$ represent (when is it `true` and when is it `false`)?

The value $M'[i, j]$ represents whether there exists a path of length 2 from node i to node j in the directed graph G = (N, E) represented by matrix M.

In other words, $M'[i, j]$ is true iff there exists a node k such that $M[i, k]$ and $M[k, j]$ are true. This node k is an intermediate node that connects i and j.

If there is no node k that can connect i and j, then $M'[i, j]$ is false.

P3.2. Consider paths of length k in graph G. Provide an algorithm that uses the Boolean matrix product operations to compute a matrix M_k such that, for every pair of nodes $n, m \in \mathcal{N}$, $M_k[\text{id}(n), \text{id}(m)]$ is `true` if and only if there is a path of length exact-k from node n to node m. Explain why your algorithm is correct and provide the complexity of your algorithm in terms of the number of Boolean matrix product operations used.

Hint. One can design an algorithm that performs at-most $\sim \log_2(k)$ Boolean matrix product operations.

Assuming the graph is in matrix representation.

Base case: If k = 1, then just return M where $M_1[i, j]$ is 1 if there is an edge from node i to j.

General case: If k > 1, recursively compute $M_{\lfloor k/2 \rfloor}$, and then set M_k to be the Boolean matrix product of $M_{\lfloor k/2 \rfloor}$ and itself ($M_k = M_{\lfloor k/2 \rfloor} \otimes M_{\lfloor k/2 \rfloor}$). If $k/2$ had a remainder, than after computing $M_{k2} = M_{\lfloor k/2 \rfloor} \otimes M_{\lfloor k/2 \rfloor}$, compute $M_k = M_{k2} \otimes M_1$.

In other words, the Boolean matrix product of $M_p \otimes M_r = M_{p+r}$.

```
Algorithm BooleanProduct($M, k$):
 1: if $k == 1$ then
 2:     return $M$
 3: end if
 4: $h = k//2$
 5: $M_h$ = BooleanProduct($M, h$)
 6: $result = M_h \otimes M_h$ “Get Boolean Product of matrix”
 7: if $h\%2 == 1$ then
 8:     $result \otimes M$ “If needed to floor the value, then compute the result with $M_1$ to add 1 to path length”
 9: end if
10: return $result$
```

Complexity: The complexity is $\sim log_2 k$ as you just divide the problem into 2 recursively until hitting the base case of 1.

When k is even, we compute $M_{\frac{k}{2}}$ by splitting the paths of length k into two subpaths of length k/2. When k is odd, we first compute M_{k-1} for to find paths of length k-1, and then multiplying by M_1 to extend the paths by one edge.

P3.3. The *transitive closure* of $\mathcal{G}$ is a graph $\mathcal{G}' = (\mathcal{N}, \mathcal{E}')$ such that $(n, m) \in \mathcal{E}'$ if and only if there is a path from node n to node m in graph $\mathcal{G}$. Provide an algorithm that uses the Boolean matrix product operations to compute a matrix M' that represents graph $\mathcal{G}'$. Explain why your algorithm is correct and provide the complexity of your algorithm.

1. Create results matrix N (initialized to all false) and OR it with matrix M. By OR, it means $N[i, j] = N[i, j] | M[i, j]$

2. Compute the Boolean matrix product $M' = M \otimes M$.

3. Store edges of M' in the results Matrix N.

4. Repeat the matrix multiplication $M' = M' \otimes M$ until M' stops changing. Keep storing the edges in results Matrrix between each multiplication.

5. The results Matrix M' is the transitive closure of G.

Complexity is as large as the longest path since each matrix multiplication adds 1 to the path. This means $\sim O(|N-1|)$. This is added to the complexity of getting all edges in M' and storing them in a resulting Matrix. Transferring this data is $\sim O(|N|^2)$ since you need to check every edge in M'.

WRONG BC THERE COULD BE CYCLES

Problem 4. Consider a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and assume we have a weight function $weight : \mathcal{E} \rightarrow \{1, \ldots, W\}$ with W some maximum integer value W.

P4.1. Assume $W = 1$ (all edges have weight 1). Given node $n \in \mathcal{N}$, provide an algorithm that can compute the shortest path from node n to any other node m in $\sim|\mathcal{N}| + |\mathcal{E}|$. In this case, the shortest path is the path with the fewest edges.

Algorithm SSSP(G, s)**:**

```
1:  $predecessors = \{n- > ?|n \in \mathcal{N}\}$
2:  $distance[s] = s$
3:  $Q = aqueue$
4:  $ENQUEUE(Q, s)$
5:  while $!EMPTY(Q)$ do
6:      $n = DEQUEUE(Q)$
7:      for all $(n, m) \in \mathcal{E}$ do
8:          if $predecessors[m] = ?$ then
9:              $predecessors[m] = n$
10:             $ENQUEUE(Q, m)$
11:         end if
12:     end for
13: end while
14: return $predecessors$
```

Complexity: We inspect each node once and traverse each edge once making it $\sim|\mathcal{N}| + |\mathcal{E}|$ complexity.

P4.2. Given node $n \in \mathcal{N}$, provide an algorithm that can compute the shortest path (in terms of the sum of the weights of edges on the path) from node n to any other node m in $\sim|\mathcal{N}| + W|\mathcal{E}|$.

P4.3. Given node $n \in \mathcal{N}$, provide an algorithm that can compute the shortest path (in terms of the sum of the weights of edges on the path) from node n to any other node m in $\sim W|\mathcal{N}| + |\mathcal{E}|$.

For each question, explain why your algorithm is correct, why your algorithm achieves the stated complexity, and which graph representation you use.

Assignment Details

Write a report in which you solve each of the above problems. Your submission:

1. must be a PDF file;
2. must have clearly labeled solutions to each of the stated problems;
3. must be clearly presented;
4. must *not* be hand-written: prepare your report in LaTeX or in a word processor such as Microsoft Word (that can print or exported to PDF).

Submissions that do not follow the above requirements will get a grade of zero.

Grading

Each problem counts equally toward the final grade of this assignment.