Name: [Mohamed Ghanem]

CS 3510: Design & Analysis of Algorithms 01/26/2023

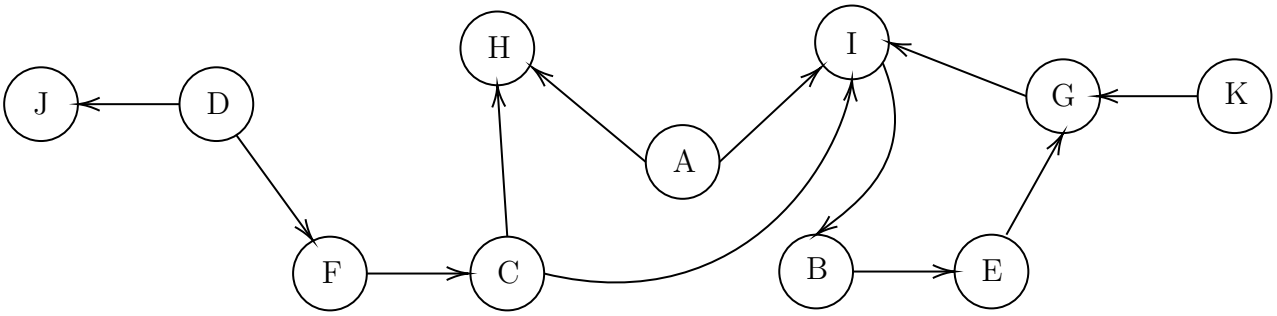
Problem Set 3: BFS, DFS, Topological Sort & SCC

Abrahim Ladha Due: 02/09/2023 11:59pm

- **Please TYPE your solutions using Latex or any other software. Handwritten solutions *won't* be accepted.**
- **Your solutions must be in plain English and mathematical expressions.**
- **Unless otherwise stated, the fastest (and correct) algorithms will receive more credit. If we ask for a specific running time, a correct solution achieving it will receive full credit even if a faster solution exists.**
- **Do not use pseudocode. Your answer will receive zero credit even if the pseudocode is correct.**
- **You can use the following algorithms as black-box for this problem set i.e. you don't have to explain the algorithm while using it. For example, you can write "Using DFS, compute the post labels":**
  - **BFS**
  - **DFS**
  - **Explore(v)**
  - **Toposort**
  - **SCC**
  - **Construct a reverse graph**
  - **Construct the metagraph**
- **When we say "design an efficient algorithm", we are asking for a detailed description of the algorithm in English, i.e., without any code or psuedocode. Fastest (and correct) solutions are worth more credit.**

Problem 1
(20 points)

Consider the graph below:



Part A

For this part, in the event of a tie, choose the node that is alphabetically first.

a.) Run DFS on this graph and list the pre and post labels of each vertex.

b.) Use the labels from (a.) to topologically sort this graph. List the vertices in topological order.

Solution:

a.) $A(1,12) \rightarrow H(2,3) \rightarrow I(4,11) \rightarrow B(5,10) \rightarrow E(6,9) \rightarrow G(7,8) \rightarrow C(13,14) \rightarrow D(15,20) \rightarrow F(16,17) \rightarrow J(18,19) \rightarrow K(21,22)$

b.) The graph does not have a topological order because it contains a cycle:

$$I \rightarrow B \rightarrow E \rightarrow G \rightarrow I$$

Definition of Topological Sort (Wikipedia)

Topological sort is a linear ordering of vertices of a directed graph such that for every directed edge (u, v), vertex u comes before vertex v in the ordering.

Explanation

If a graph contains a cycle, it is impossible to find a linear ordering that satisfies the definition above. A cyclic graph, must contain a back edge that connects some vertex v to another u where $v > u$ in the sorted order, this contradicts the definition of topological sort. Therefore, a topological sort is not possible for graphs that is cyclic and only works on Directed Acyclic Graphs (DAGs).

Part B

For this part, we run the SCC algorithm on the above graph. When running DFS on the reverse graph, use the alphabetical order of vertices.

a.) Write the strongly connected components in the order you found them.

b.) Label all the sink components and all the source components.

c.) Give the minimum number of edges that must be added to the graph above to make it strongly connected.

Solution:

a.)

$$(J) \rightarrow (H) \rightarrow (B, E, G, I) \rightarrow (K) \rightarrow (C) \rightarrow (F) \rightarrow (D) \rightarrow (A)$$

b.) Sinks Components:

$$(J), (H), (B, E, G, I)$$

Source Components:

$$(D), (K), (A)$$

c.) The minimum number of edges that must be added to the graph so that every vertex is reachable from any other vertex (one strongly connected component), is 3. Three possible edges to add are (J, K), (B, D), and (H, A). This was done by constructing and examining the metagraph, and adding edges from sink components to source components.

Problem 2
(20 points)

Part A

(10 points) Let $G = (V, E)$ be a weighted Directed Acyclic Graph (DAG). Design an efficient algorithm to find the shortest path from a source vertex s to all other vertices. Make sure to provide proof of the correctness of your algorithm and the time complexity.

Note: You cannot use Dijkstra's Algorithm as a black-box for this problem.

Solution:

Proposed Algorithm

1. Run Toposort on the graph to obtain the topological ordering. The first element in the topological ordering is the source vertex (assuming the input graph contains only one source).
2. Create a map M that maps each vertex in the graph to its shortest path from the source. Initialize all the distances to infinity, except for the source to 0.
3. Begin traversing the graph in topological ordering starting from the source S.
4. For every vertex in the topological sort v iterate over every vertex adjacent to it u (every incoming edge).
5. At each iteration perform the following:
   i.) Compute the minimum between $M[v]$, the current shortest path to v and $M[u] + W(u, v)$, the shortest path to the adjacent vertex u in addition to the weight of the edge from u to v.
   ii.) Update the current distance from S to v, $M[v]$ to the new current computed minimum.

Time Complexity

TopoSort has a time complexity of $\mathcal{O}(|V| + |E|)$. To calculate the shortest path we iterate over every vertex in the graph and every incoming edge to that vertex (every adjacent vertex), this also has a time complexity of $\mathcal{O}(|V| + |E|)$. Therefore, the overall time complexity of the proposed algorithm is $\mathcal{O}(|V| + |E|)$.

Proof of Correctness

The algorithm calculates the shortest path between the source and every other vertex in the graph by linearizing the graph using TopoSort, then computing the shortest path for every vertex one at a time in topological ordering. This guarantees that at every iteration

over the topological ordering the shortest path between the current vertex and the source is found. This is then used going forward to find the shortest path for other vertices. For example, if we have already iterated over the first n vertices v in the topological order, then the shortest path between the source and $v_0, v_1, ..., v_{n-1}$ has already been computed. These computed distances can then be used to calculate the shortest path between the source and v_n because the shortest path is either the direct path from the source to v_n or some shortest path from the source to a vertex v_m in addition to the path from v_m to v_n where $m < n$. Therefore, the path at v_n will also be the shortest path from the source. This algorithm is correct because if there exists a path between the source and some vertex in the graph the algorithm will find the shortest path.

Part B

(10 points) Suppose a wedding is hosted at your house and you have invited n guests. However, each of the guests has a list of close friends and relatives with that they must sit at the same table. Since some of these tables might have a lot of people sitting at them, we want to make sure the tables we buy are large enough. Design an algorithm to determine the maximum number of people sitting at any one table. Make sure to provide proof of the correctness of your algorithm and the time complexity.

Solution:

Proposed Algorithm

1. Create a directed graph using the guest list where every guest is a vertex. Add an edge between two guests only if the two guests want to sit together.
2. Create two variables max and $currentCount$. The max variable will be initialized to 0 and will keep track of the current maximum number of people sitting at any one table. The $currentCount$ variable will be reset to 0 at every iteration and will keep track of the current number of people sitting at one table.
3. Run DFS on the graph.
4. Before DFS calls Explore reset $currentCount$ to 0.
5. Every time Explore is called recursively increment $currentCount$ by 1.
6. When Explore returns to DFS update the max variable to be the maximum value between max and $currentCount$.
7. When the algorithm terminates the value in max will be the maximum number of people sitting at any one table.

Time Complexity

The proposed algorithm consists of a DFS call with some additional constant time operations. Therefore, the overall time complexity of the proposed algorithm is the time complexity of DFS which is $\mathcal{O}(|V| + |E|)$.

Proof of Correctness

The graph is a disconnected, every Explore call made by DFS is a single table of guests. The proposed algorithm keeps track of the number of reachable nodes by a Explore call from DFS, which is equal to the number of guests sitting on one table. The algorithm iterates through all the disconnected components of the graph, which corresponds to the all the tables on the guest list, finding the number of guests on each table and updating the current maximum number of guests on a single table whenever a new maximum is found.

Therefore, the algorithm is correct because it is able to determine the maximum number of people sitting at one table.

Problem 3
(20 points)

Suppose Mr. Kartik is hosting a party with international delegates, where all the guests speak unique languages. However, each guest can understand (but cannot speak) some of the other languages spoken at the party. In this way, two guests that don't understand each other can have a conversation by using another guest (or more) as a "translator". You're given a list of guests and which languages they understand.
As an example, if A is trying to communicate with C, and B understands A, and C understands B, then C can effectively understand A. However, A can't necessarily understand C.

a.) (15 points) You want to form the minimum number of tables so that everyone at a table is able to converse with everyone else at the table (that is, for any two people at the table, they can both understand each other). Design an efficient algorithm for this. Make sure to provide proof of the correctness of your algorithm and the time complexity.

b.) (5 points) Now you want to pass a message around to as many tables as possible, by starting it at some table(remember that some guests can understand guests at other tables). Explain your approach to get the maximum number of tables your message can reach.
Note: No need to provide a formal algorithm, but a small paragraph explaining your approach is sufficient.

Solution:

a.)

Proposed Algorithm

1. Create a directed graph with the guests as vertices. Add an directed edge from one guest A to another B only if A understands B.
2. Run SCC on the graph. SCC will return all the strongly connected components in the graph.
3. Return the number of strongly connected components as the result.

Time Complexity

The proposed algorithm consists of just one SCC call, which has a time complexity of $\mathcal{O}(|V| + |E|)$. Therefore the overall time complexity of the algorithm is $\mathcal{O}(|V| + |E|)$.

Proof of Correctness

In the graph a strongly connected component is formed only when every guest on the table is able to converse with every other guest on the same table. If in the graph, a guest A is reachable from some other guest B and guest B is reachable from guest A, then A and B can converse. This means there must exist a cycle that connects guests A and B. Therefore, A

and B are in the same strongly connected component. The proposed algorithm finds all the strongly connected components, the number of strongly connected component corresponds to the minimum number of tables required so that everyone at each table is able to converse with everyone else at the same table. Therefore, the algorithm is correct because it finds the minimum number of tables given the above requirement using SCC.

b.)

Create a metagraph using the strongly connected components returned from SCC with edge weight of 1. Assume we are not given a start table (strongly connected component). We are trying to find the longest path in the metagraph which corresponds to the maximum number of tables that a message can travel. Run TopoSort to find the topological ordering of the metagraph. Create a map to track the current maximum distance traveled to each vertex in the metagraph. Iterate over the metagraph in topological order. For each vertex iterate over its adjacent vertices. For each adjacent vertex add the edge weight of 1 to the current maximum distance in the map for that adjacent vertex, then take the maximum between the result and the current iteration vertex. After the topological sort iteration, the maximum value in the map is the maximum number of tables that a message can travel.