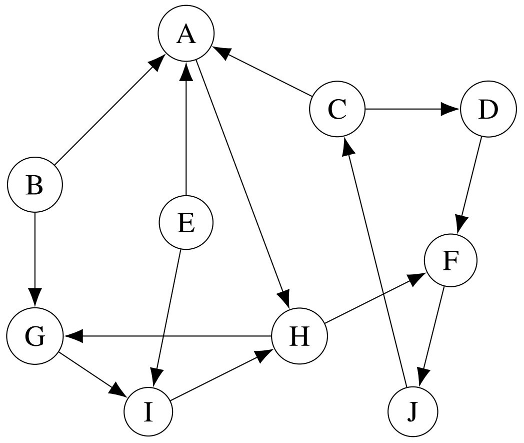
CMPSC 465 Data Structures & Algorithms
Fall 2025 Ke Chen and Yana Safonova

HW 5

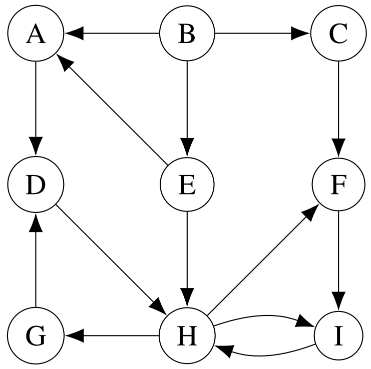
1. (? pts.) **Strongly Connected Components**

Run the strongly connected components algorithm on the following directed graphs. When doing DFS on the reverse graph G^R: whenever there is a choice of vertices to explore, always pick the one that is alphabetically first.

(i)



(ii)



In each case answer the following questions.

(a) In what order are the strongly connected components (SCCs) found?

(b) Which are source SCCs and which are sink SCCs?

(c) Draw the "metagraph" (each "meta-node" is an SCC of G).

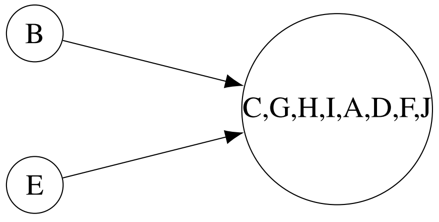
(d) What is the minimum number of edges you must add to this graph to make it strongly connected (namely, consisting of just a single SCC)?

Answer:

(i) (a) The strongly connected components are found in the order $\{C,G,H,I,A,D,F,J\}$, $\{E\}$, $\{B\}$.

(b) The source SCC's are $\{E\}$ and $\{B\}$, while $\{C,G,H,I,A,D,F,J\}$ is a sink SCC.

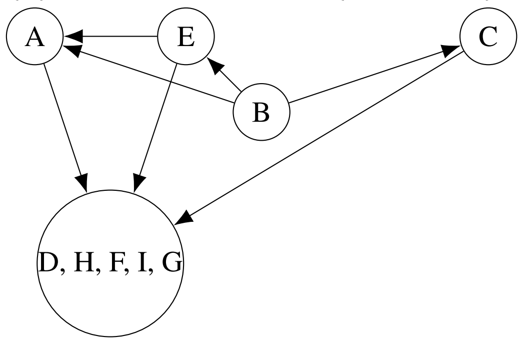
(c)



(d) It is necessary to add two edges to make the graph strongly connected, e.g. by adding $C \to B$ and $B \to E$.

(ii) (a) The strongly connected components are found in the order $\{D,F,G,H,I\}$, $\{C\}$, $\{A\}$,{E}, $\{B\}$.

(b) $\{B\}$ is a source SCC, while $\{D,F,G,H,I\}$ is a sink.

(c) 

(d) In this case, adding one edge from any vertex in the sink SCC to any vertex in the source SCC makes the metagraph strongly connected and hence the given graph also becomes strongly connected.

2. (? pts.) **Strongly Connected Components.** Give an efficient algorithm which takes as input a directed graph $G = (V,E)$ and determines whether or not there is a vertex $s \in V$ from which all other vertices are reachable. Explain why your algorithm is correct and analyze its running time.

Answer:

Consider any graph G. In the metagraph of G, if there are two (or more) source strongly connected components, then there can never be a vertex which can reach both of the source SCCs. So, if G is one-way connected, then there must be only one source SCC, and the special vertex v must be in this source SCC.

Moreover, any vertex u in this source SCC can reach v, and so all vertices in G are also reachable from u. So, if G is one-way connected, then all vertices can be reached from any vertex in the source SCC of G.

This gives the following algorithm - run a DFS on G from any starting node, and find the vertex v with the highest post number (which must be in a source SCC). Then run a DFS from v and check that all vertices are reachable from v. There are 2 DFS's, so this takes $O(|V|+|E|)$ time.

Clearly, if the algorithm returns true, then G is one-way connected (it actually finds the special vertex v). If the algorithm returns false, we know that there is a vertex in a source SCC of G which cannot reach all vertices, and by the preceding discussion this means that G is not one-way connected.

3. (? pts.) **Unique Shortest Path.** Shortest paths are not always unique: sometimes there are two or more different paths with the minimum possible length. Show how to solve the following problem in $O((|V|+|E|)\log|V|)$ time. Explain why your algorithm is correct and justify its running time.

Input: An undirected graph $G = (V,E)$; edge lengths $l(u,v) > 0$ for all $(u,v) \in E$; a starting vertex $s \in V$.

Output: A Boolean array `usp[·]`: for each node u, the entry `usp[`u`]` should be `true` if and only if there is a *unique* shortest path from s to u. (By convention, we set: `usp[`s`] = true`.)

Answer:

This can be done by slightly modifying Dijkstra's algorithm. The array `usp[·]` is initialized to `true` in the initialization loop. The main loop is modified as follows:

This will run in the required time when the heap is implemented as a binary heap.

Proof of Correctness. Assume for contradiction that for some vertex v, the algorithm outputs true but there exist two distinct shortest paths from s to v. Let u and w be the last vertices on these two paths. By

Algorithm 1 Dijkstra's Algorithm with Path Tracking

```
while $H$ is not empty do
    $u \leftarrow$ DELETEMIN$(H)$
    for $(u,v) \in E$ do
        if DIST$(v) >$ DIST$(u) + l(u,v)$ then
            DIST$(v) \leftarrow$ DIST$(u) + l(u,v)$
            usp$[v] \leftarrow$ usp$[u]$
            DECREASEKEY(H, v)
        else if DIST$(v) =$ DIST$(u) + l(u,v)$ then
            usp$[v] \leftarrow$ false
        end if
    end for
end while
```

the correctness of Dijkstra's algorithm, both dist$[u]$ and dist$[w]$ are less than dist$[v]$, so both u and w are extracted before v. The algorithm would then have seen both shortest paths via the edges (u,v) and (w,v) and set `usp[v]` = false, which is a contradiction. On the other hand, if `usp[v]` is false but only one shortest path exists, no second adjacent vertex can produce the same distance, so `usp[v]` would remain true, which is a contradiction. Therefore, `usp[v]` is true if and only if there is a unique shortest path from s to v.□

4. (20 pts.) **Dijkstra's Algorithm.** Consider a directed graph in which the only negative edges are those that leave s; all other edges are positive. Can Dijkstra's algorithm, started at s, fail on such a graph? Assume that s is a source node, i.e., s has no incoming edges. Prove your answer.

Answer:

Dijkstra's algorithm works correctly in this situation.

Proof: Consider the proof of Dijkstra's algorithm. The proof depended on the fact that if we know the shortest paths for a subset $S \subseteq V$ of vertices, and if (u,v) is an edge going out of S such that v has the minimum estimate of distance from s among the vertices in $V \backslash S$, then the shortest path to v consists of the (known) path to u and the edge (u,v). We can argue that this still holds even if the edges going out of the vertex s are allowed to negative. Let (u,v) be the edge out of S as described above. For the sake of contradiction, assume that the path claimed above is not the shortest path to v. Then there must be some other path from s to v which is shorter. Since $s \in S$ and $v \notin S$, there must be some edge (i,j) in this path such that $i \in S$ and $j \notin S$. But then, the distance from s to j along this path must be greater than that the estimate of v, since v had the minimum estimate. Also, the edges on the path between j and v must all have non-negative weights since the only negative edges are the ones out of s. Hence, the distance along this path from s to v must be greater than the estimate of v, which leads to a contradiction. □

5. (? pts.) **BFS on Directed Graph.** Consider a directed graph $G = (V,E)$. Depth first search allows us to label edges as tree, forward, back and cross edges. We can make similar definitions for breadth first search on a directed graph:

1. A BFS tree edge is an edge that is present in the BFS tree.
2. A BFS forward edge leads from a node to a non-child descendant in the BFS tree.
3. A BFS back edge leads to an ancestor in the BFS tree.
4. All other edges are BFS cross edges.

(a) Explain why it is impossible to have BFS forward edges.

(b) Give an efficient algorithm that classifies all edges in G as BFS tree edges, back edges, or cross edges.

Answer:

(a) Assume it is possible to have BFS forward edges. Assume one of this type of edges is (u,w), which leads a node u to its non-child descendant w. As u's descendant, w is visited after u is visited. Due to the edge (u,w), $dist[w] = dist[u]+1$. So, w is u's child which contradicts with the earlier assumption that w is u's non-child descendant. Therefore, it is impossible to have BFS forward edges.

(b) Algorithm:

- Run BFS on G and obtain the BFS tree. The edges in the BFS tree are tree edges.
- Intuition: both BFS back edges and DFS back edges require the tail to lead to its ancestor as the head. Also, both BFS cross edges and DFS cross edges require the tail to lead to a node that's neither its ancestor nor its descendant as the head. The only difference between BFS edges and DFS edges are that one is regarding the BFS tree and the other is regarding the DFS tree. So, the key to classify BFS back edges and BFS cross edges is to determine the ancestor/descendant relationship between the tail and the head of the edges. We can make use of pre-visit number and post-visit number of each node in a tree to achieve this, and those numbers can be obtained by running DFS on the tree.
- Run DFS on the BFS tree starting from the root to obtain the pre-visit number and post-visit number for each vertex on the BFS tree.
- To classify the edges that are not in the BFS tree (non tree edges), for an edge (w,v), if $pre[v] < pre[w] < post[w] < post[v]$, then it's a back edge. If $pre[v] < post[v] < pre[w] < post[w]$ or $pre[w] < post[w] < pre[v] < post[v]$, then it's a cross edge.

Run time analysis:

- BFS takes $O(|V|+|E|)$ time.
- DFS takes $O(|V|+|E|)$ time.
- Going through the edge set takes $O(|E|)$ time.

Thus, overall, the algorithm takes linear time, $O(|V|+|E|)$.

Rubric:

Problem 1, ? pts

?

Problem 2, ? pts

?

Problem 3, ? pts

?

Problem 4, ? pts

?

Problem 5, ? pts

?