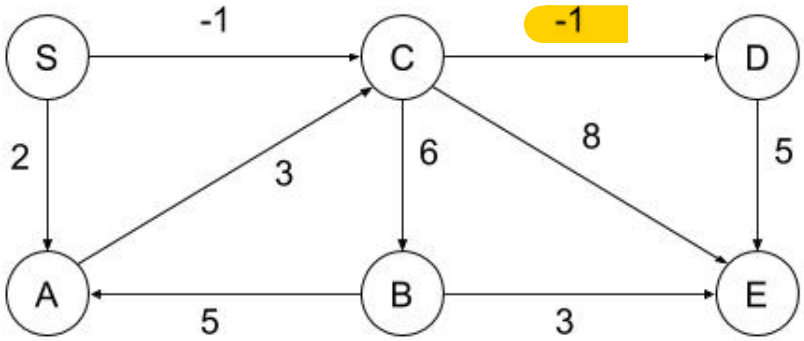
MIDTERM PRACTICE PROBLEMS

1. True/False: Dijkstra's algorithm will return the correct shortest paths from s to all other vertices when run on the graph given below.



Solution

True. Dijkstra's algorithm starting from vertex S returns $l(A) = 2, l(B) = 5, l(C) = -1, l(D) = -2, l(E) = 3$ which are the correct shortest distances from S to all the vertices.

2. True/False: If $f(n) = \mathcal{O}(g(n))$, then $f(5n)$ is $\mathcal{O}(g(5n))$

Solution

True. Since $f(n)$ is $\mathcal{O}(g(n))$, there exists constants c, n_0 such that for all $n \geq n_0$, $f(n) \leq c.g(n)$. This means that for all $n \geq \lceil n_0/5 \rceil$, $f(5n) \leq c.g(5n)$. This means that f(5n) is $\mathcal{O}(g(5n))$.

3. Suppose we are given an array A containing n elements such that the first p elements are in increasing order and the rest are in decreasing order. Give an algorithm to find the k^{th} smallest element in the array in $\mathcal{O}(log\ n)$ time.

   Solution

   Let A_1 be the sub-array from 1 to p and A_2 be the sub-array from p + 1 to n. Without loss of generality, we may assume that we are provided with two arrays A_1 and A_2 both of which are sorted in increasing order (since we do not change the contents of the array, we can do all operations in place).

```
1:  select(A, B, k):
2:  while k > 0 do
3:      m = (k - 1)/2
4:      if m ≥ len(A) then
5:          B = B[m + 1 :]
6:      else if m ≥ len(B) or A[m] ≤ B[m] then
7:          A = A[m + 1 :]
8:      else
9:          B = B[m + 1 :]
10:     end if
11:     k− = m + 1
12: end while
13: if len(A) == 0 then
14:     return B[0]
15: end if
16: if len(B) == 0 then
17:     return A[0]
18: end if
19: return min(A[0], B[0])
```

   The input size is reduced by half or one-fourth during every step of recursion. Hence the running time of the algorithm is $\mathcal{O}(log\ k)$. Since $k \leq n$, the running time is $\mathcal{O}(log\ n)$.

4. You are given a weighted directed graph $G = (V, E, w)$ and the shortest path distances $\delta(s, u)$ from a source vertex s to every other vertex in G. With this information, give an algorithm to find a shortest path from s to a given vertex t in $\mathcal{O}(V + E)$ time. Note the "a given vertex t" in the problem statement.

   Solution

   Start at u. Of the edges that point to u, at least one of them will come from a vertex v that satisfies $\delta(s, v) + w(v, u) = \delta(s, u)$. Such a v is on the shortest path. Recursively find the shortest path from s to v.

   Pseudocode:

```
FindPath(G,s,t)
  If t == s:  return prev
  For each edge (u,t):
    if δ(s,u) + w(u,t) = δ(s,t):
      prev[t]=u
  Findpath(G,s,u)
```

   The above algorithm will return a $prev$ array that will find a shortest path from s to t by backtracking from t.
   This algorithm hits every vertex and edge at most once, for a running time of $\mathcal{O}(V + E)$.

5. Given a directed acyclic graph $G = (V, E)$ and a vertex u, design an algorithm that outputs all vertices $S \subseteq V$ such that for all $v \in S$, there is an even-length simple path from u to v in G. (A simple path is a path with all distinct vertices.)

   Solution We will modify the DFS algorithm slightly to find all the vertices to which an even length path exists.

   Given that its a DAG, all the paths are simple paths, that is there are no cycles.

   a) We maintain 2 different values for each vertex.

   b) We will mark the vertex as visited only if it is at an even length. So have a boolean corresponding to the visit.

   c) The other boolean that we maintain is odd_visited, which is marked true if this vertex was at an odd distance when it is visited. If it is at odd distance we just visit its children as they will definitely be at an even distance.

   d) Also if we visit the same vertex again at odd distance we won't visit its children as they have already been visited when it was visited at an odd distance the last time. This is achieved by the odd_visited value we maintain at each vertex.

   The pseudo code for this:

```
1:  Initialize visited and odd_visited for all vertices to be false
2:  DFS(Vertex v):
3:  v.visited = true
4:  output v
5:  for each vertex i ∈ adj(v) do
6:      if i.odd_visited == false then
7:          i.odd_visited = true
8:          for each vertex j ∈ adj(i) do
9:              if j.visited == false then
10:                 DFS(j)
11:             end if
12:         end for
13:     end if
14: end for
```

   Proof of correctness:

   a) The algorithm only outputs vertices when we call DFS for that vertex. We call DFS only for vertices which are at an even distance. Therefore all vertices that are output by the algorithm are at even length.

   b) Now we need to show that the algorithm outputs all the vertices that are at even length.

      We can see that all the vertices in the connected component of u will be marked visited or odd_visited or both.

      Proof by contradiction

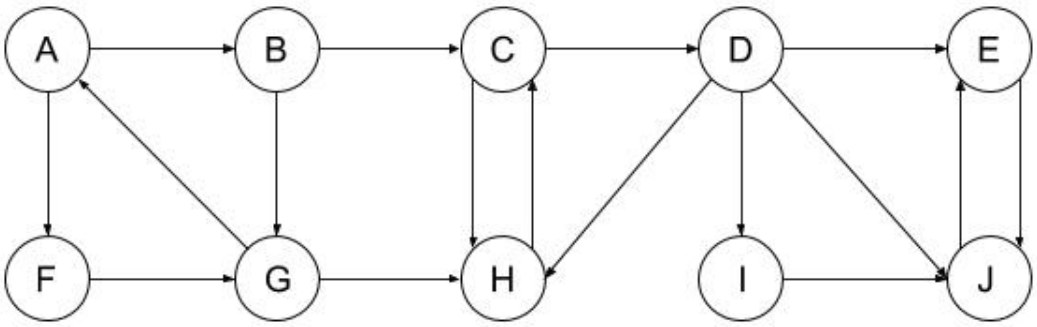
      Say there is a vertex w that is at an even distance but is not output by the algorithm. So w is not marked visited. So it's previous vertex, say x, on that path will not

be marked odd_visited. Because if it would have been marked odd_visited then w would have been marked visited. Therefore x would be marked visited on that path as it must be marked at least one of visited and odd_visited. That is x is at even distance on that path. But we assumed x to be at odd distance as it is one less than distance of w which is even. Therefore contradiction.

Time complexity:

This algorithm is a modification of the DFS algorithm. Here every vertex and edge is visited at most twice, once odd_visited and then visited. Therefore the time complexity is twice as much that of DFS. Therefore the time complexity is O(m + n), where m is the number of vertices and n is the number of edges.

6. Run the strongly connected components algorithm on the following directed graph. Whenever there is a choice of vertices to explore, always pick the one that is alphabetically first.



(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).

Solution

(a) The SCCs are found in the following order:
$EJ, I, CDH, ABFG$

(b) $ABFG$ is a source SCC, EJ is a sink SCC

(c) Metagraph