ALGORITHMS: ASSIGNMENT-7

KAVYA SHAMSUNDARA

1 (a) BFS Algorithm : Waterjug problem

- The waterjug object is initially in the state (0,0,0)
- Add this object (the one representing the initial state of the problem) to the queue.
- Do while the queue is not empty
- Begin
- Check if object at the start of the queue is required goal.
- If yes,
- Display or print the search path ie; the visited nodes.
- Exit.
- Else
- Create all possible valid states of child to current state
- Add them to the queue.
- Add this current state to the visited nodes array.
- This array stores all the traversed nodes.
- Deque the current object
- End If
- End

1 (b) Correctness of the algo

- Since the algorithm explores all possible nodes, we will get a finite count of the number of nodes
- The states of the algorithm will be finite
- Since the states is finite, a graph can be plotted.
- The application of BFS is to visit all node in the graph.
  Hence, we get a correct graph.
- In case of impossible state (where the traversal stops abruptly) it is not possible to plot the graph.

(1)(c) Time required

The worst case time complexity is all the numbe of nodes visited.
If the nodes exists, then time required will the last node visited.

∴ The time complexity is

O (waterjug limit)

In this case,

$$\boxed{O(C_1 C_2 C_3)}$$

KAVYA SHAMSUNDARA

(3) Problem 23-4 on pg 641
Alternative minimum-spanning-tree algorithms

Solution:

(a) MAYBE-MST-A

- MAYBE-MST-A removes edges in non-increasing order as long as the graph remains connected, and the resulting T is a minimum spanning tree.
- To show correctness, let S be a MST of G. Each time we remove ~~the~~ an edge e, either $e \in S$ or $e \notin S$.
- If $e \notin S$, then we can just remove it.
- If $e \in S$, removing e disconnects S (but not the graph) into two trees.
- ~~There~~ There cannot be another edge connecting these trees with smaller weight than e, because by assumption S is a MST, & if a larger edge existed that connected the trees the algorithm would have removed it before removing 'e'. We know a path exists since the graph is still connected, so it must be there is another edge with equal weight that hasn't been discovered yet, so we can remove 'e'.

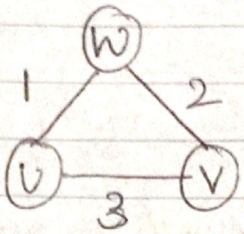
Implementation:

Use an adjacency list representation for T. We can sort the edges in $O(E \log E)$ using MERGE-SORT. BFS or DFS can be used in $O(V+E)$ to check if $T - \{e\}$ is a connected graph. The edges are sorted once and BFS/DFS is performed E times, once for each edge. The total running time is

$$O(E \log E + E(V+E)) = O(E^2).$$

(b) MAYBE-MST-B

- MAYBE-MST-B will not give a minimum spanning tree. We prove this by counter-example. Consider the following graph G:



- The MST of G would have edges (w,u) & (v,w) with weight 3. Since MAYBE-MST-B takes edges in arbitrary order, it could add edges (u,v) & (v,w) to T, then try to add (w,u) which forms a cycle, then return T (weight 5).

Implementation

- This implementation is similar to Kruskal's algorithm. Use a UNION-FIND data structure to maintain T. For each vertex 'v' we need to MAKE-SET(v). For each edge e=(u,v), if ~~FINDSET~~ FINDSET(u) ≠ FINDSET(v) then there is no cycle in T ∪ {e} & we UNION-SET (u,v).

- In total there are V MAKE-SET operations, 2E FIND-SET operations, & V-1 UNION-SET operations. Using an improved UNION-FIND data structure we can perform FIND-SET in O(1) & UNION-SET in O(log V) time amortized to give a running time of:

$$O(V) + O(E) + O(V \log V) = O(V \log V + E).$$

KAVYA
SHAMSUNDARA

(c) MAYBE-MST-C

- MAYBE-MST-C gives a minimum spanning tree
MAYBE-MST-C adds edges in arbitrary order to T & if a cycle c is detected removes the maximum-weight edge on c.
- Each time we add an edge e, either we form a cycle c or we do not. Suppose e is added to some tree T' in T & a cycle c is formed. Then we remove a maximum-weight edge e' from c & T'-e'+e is of less or equal weight than T' (because e' is of greater or equal weight than e).
- If a cycle is not formed we either just add e to some tree in T or e connects two trees in T.
In the second case if 'e' is not the smallest weight edge that ~~conve~~ connects the trees then we haven't discovered it yet, & when we eventually form a cycle we have already shown MAYBE-MST-C performs correctly.

Implementation:

- Use an adjacency list representation for T. For each edge, we add it to T & then check T ∪ {e} for cycles. There can be at most one cycle, so we can use DFS to detect the cycle & output it.
If there are no cycles, then we are done with this edge. Otherwise, we need to output the cycle, find the maximum-weight edge, & delete it from T.
- Adding an edge takes O(1). DFS takes O(V+E) = O(V) in this case (as soon as T has a cycle we break it, so the number of edges in T at any point is no greater than V). Finding the maximum-weight edge on the cycle takes O(V) & deleting an edge is O(V). For each edge we add it to T, perform DFS, & possibly find a cycle, so the total running time is O(EV).

(5) (6)

(2) A bipartite graph $G = (V, E)$ is a graph whose vertices can be partitioned into 2 sets ($V = V_1 \cup V_2$ & $V_1 \cap V_2 = \phi$) such that there are no edges between vertices in same set

Formally,

$$u, v \in V_1 \Rightarrow (u, v) \notin E$$
$$u, v \in V_2 \Rightarrow (u, v) \notin E$$

Input: Graph G
Output: return true if the graph is bipartite
false otherwise

```
for all v ∈ V:
  visited (v) = false
  color (v) = GREY
while ∃s ∈ V: visited (s) = false
   visited (s) = true
   color (s) = WHITE
S = [s]  (stack)
   while S is not empty
     u = pop (S)
     for all edges (u, v) ∈ E:
       if visited (v) = false
          visited [v] = true
          push (S, v)
       if color (v) = GREY
          if color (u) = BLACK
             color (v) = WHITE
       else if color (v) = WHITE:
          if color (u) ≠ BLACK:
             return false.
     return true.
```