CS5800: Algorithms — Spring '21 — Virgil Pavlu

Homework 10
Submit via Gradescope

Name: Ketaki Kolhatkar
Collaborators: Ashwini Kumar

Instructions:

- Make sure to put your name on the first page. If you are using the LaTeX template we provided, then you can make sure it appears by filling in the `yourname` command.
- Please review the grading policy outlined in the course information page.
- You must also write down with whom you worked on the assignment. If this changes from problem to problem, then you should write down this information separately with each problem.
- Problem numbers (like Exercise 3.1-1) are corresponding to CLRS 3^{rd} edition. While the 2^{nd} edition has similar problems with similar numbers, the actual exercises and their solutions are different, so make sure you are using the 3^{rd} edition.

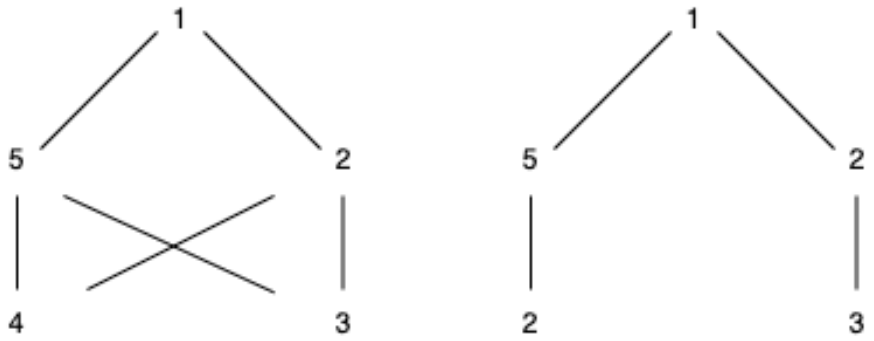
1. ***(15 points)*** *Exercise 22.1-5.*

Solution:
The adjacency representation of a graph G = (V, E) is given as the square of the [V]. If we square the matrix , then the matrix would be bounded by an edge of vertices separated by a path of exactly 2. G[u, v] should also represent node with an edge 1 included. The most efficient algorithm for multiplying n*n matrices has the runtime of $O(n^2.376)$. The best lower bound is $O(n^2)$. Square of a matrix could be done by the SQUARE-MATRIX-MULTIPLY method.
Level 1 has immediately adjacent vertices. Level 2 has distance 2 from vertices from the source. We add the vertices to the adjacency list. We recursively do BFS on the vertices.

```
For each vertex v in V:
    Implement BFS with v as source vertex:
        For all vertices u at distance of 2 from v:
            add u to the v's adjacency list
            termiate BFS
```

2. ***(15 points)*** *Exercise 22.2-6.*

Solution: With the given graph G' = (V, E), with 1 as the vertex, E can never be produced by running



BFS on the graph. 2 comes before 5 in the adjacency list of list1, hence 2 would be dequeued before . 3 and 4 would be equal to 2. But this is not the case. This tells us that 5 comes before 2 in the adjancency list. This again means that the 3 and 4 both must be equal to 5. This again is not true. The unique simple path in G' from to a vertex is the shortest path in G.

3. ***(15 points)*** *Exercise 22.2-7.*

Solution:

```
1. Make a graph of the wrestlers and mark (colour) the vertices based on which type of
wrestler - "babyfaces" ("good guys") or the "heels" ("bad guys"). Since we have different
marking we maintain the property of rivarly between the two groups.

2. We calculate the d of each vertex each of the connected components of the graph by
running the Breadth First Traversal. Based on the d value, if we get odd - we mark those
```

vertices as the "heels" and when we get an even value we mark the vertex as the "babyface".

3. The d values of v and and v would be consecutive, making them one odd and one even, hence the marking scheme would be maintained of no "babyface" being connected to "heels".

4. We check the edges for verification.

The BFS takes a time complexity of O(n+r), while the checking takes a time complexity of O(n), giving us an overall time complexity of O(n+r).

4. ***(10 points)*** *Exercise 22.3-7.*

Solution:

```
DFS_STACK(G):
    for u in {G.V}:
        u.colour = White
        #Breadth First Tree
        u.pi = NIL
    end for
    time = 0
    #Setting up an empty stack
    S = {}
    while u in G == White:
        S.push(u)
        while S != 0:
            v = S.pop
            #white vertex u has been discovered
            time ++
            v.d = time
            for all neighbours w of v:
                if w.colour == White:
                    w.colour == Gray
                    w.pi = v
                    S.push(w)
                end if
            end for
            #gray vertex u has been explored
            time ++
            v.f = time
        end while
    end while
```

5. ***(10 points)*** *Exercise 22.3-10.*

Solution:
For printing the edge and its type we would need to modify the DFS-VISTED-PRINT method by adding a print method to each edge type.

```
DFS-VISITED-PRINT(G, u):
time++
u.d = time
u.colour = Gray
for v in G.Adj[u]:
    if v.colour == White:
        print "(u,v) is an edge tree"
        v.pi = u
        DFS-VISITED-PRINT(G, v)
    else if v.colour == Gray:
        print "(u,v) is a back edge"
    else if v.d > u.d:
        print "(u, v) is a forward edge"
    else:
        print "(u, v) is a cross edge"
    end if
end for
u.colour = Black
time++
u.f = time
```

6. ***(15 points)*** *Exercise 22.3-12.*

Solution:
The modification we need to do are - Start at time 0 at any one of the white vertex and 1 would be the component number. Then we enter the descendants of the graph from vertex 1. We do this using the $DFS_C C$ method where the when the vertices are visited they are marked gray, then connected component number is updated to the value k. The vertex is then coloured black after it has been visited. Once all the vertices within first connected component are visited, we then increment the value k and start with a white vertex from another component.

```
DFS_CC(G):
    for u in G.V:
        u.colour = White
        u.pi = NIL
    end for
    time = 0
    for u in G.V:
        if u.colour == White:
            u.cc = k
            k++
```

```
            DFS_VISIT_CC(G, u)
        end if
    end for

DFS_VISIT_CC(G, u):
    time++
    u.d = time
    u.colour = Gray
    for each u in G.Adj[u]:
        v.cc = u.cc
        if v.colour == White:
            v.pi = u
            DFS_VISIT_CC(G, v)
        end if
    end for
    u.colour = Black
    time ++
    u.f = time
```

7. ***(20 points)*** *Exercise 22.4-5.*

Solution:
We count the number of incoming edges that all the vertices have and we store it in a dictionary with counts keyed corresponding to the vertices. We get a time complexity of O(V+E) due to this reason. With a in degree 0, we know that the vertex has 0 incoming edges, which we store in the dictionary. Next we check the outgoing edges at the vertex and decrease it from the count of the incoming edges. When this count becomes equal to 0, the vertex is then moved to the zero vertex dictionary. G having cycles will not be included in the dictionary since it already has an incoming edge. Hence we decide to move it to the zero vertex dictionary.

8. ***(15 points)*** *Two special vertices s and t in the undirected graph G=(V,E) have the following property: any path from s to t has at least* $1+|V|/2$ *edges. Show that all paths from s to t must have a common vertex v (not equal to either s or t) and give an algorithm with running time O(V+E) to find such a node v.*

Solution:
Each path from s to t has $1+|V|/2$ edges. The level of node t ¿ than that of $1+|V|/2$. When we count the level from 1 to $|V|/2$, we get a total of $|V|-2$ levels excluding the two levels. Each of these has >= 2 nodes in each level , hence the total nodes in G is more than that of $|V|$. Therefore, there must be a node in these levels, which when deleted will lead to no path between s and t.

```
Modified BFS():
    BFS(G, s, L):
        for u in G.V:
            u.colour = white
            u.pi = NIL
```

```
            u.d = inf
        end for
        s.colour = gray
        s.d = 0
        L[0].append(s)
        s.pi = NIL
        Q = none
        Enqueue(Q, s)
        while Q != none:
            u = Dequeue(Q)
            for v in G.adj[u]:
                if u.colour == white:
                    v.colour = gray
                    v.d = u.d + 1
                    L[v.d].append(v)
                    v.pi = u
                    Enqueue(Q, v)
                end if
            end for
        end while
        u.colour = black

CommonNode(G, s, t):
    BFS(G, s, L)
    i = 1
    n = |L|/2
    while i <=n:
       if |L[i]| == 1:
            print(L[i][0])
        end if
        i+=1
```

The time complexity of the algorithm would be equal to O(V+E).

9. ***(Extra Credit)*** *Problem 22-3.*

Solution:

10. ***(Extra Credit)*** *Problem 22-4.*

Solution:

11. ***(25 points)*** *Exercise 23.1-3.*

Solution:

We consider a nil set of edges. Considering the ideal case, if we make a cut with an edge (u,v)crossing it, and we know that this is the edge with the minimum weight, we infer that

(u, v) is the light edge of the cut. We can hence add it to the minimum spanning tree.
By contradiction, when we consider 2 trees which we would obtain by disconnecting the minimum weight edge from the minimum spanning tree. If we have two vertices cut by the (u, v) edge from the two trees, and there was another edge with even less weight than the (u, v) edge in the cut which becomes the minimum weight, we would choose this edge to create the minimum spanning tree. Doing this would give us a minimum spanning tree with a weight less than that with the edge (u, v) contradicting the MST property. (u, v) is hence is a light edge crossing a cut of the graph.

12. ***(25 points)*** *Exercise 23.2-2.*

Solution:
To find the adjacent vertices of a vertex u we have to search for the row of u in the adjacent matrix. The matrix stores the edge weights. For the edges that are not connected, the value of the weights written is 0.

```
PRIM(G, vertex):
    for each u in V[G]:
        P[u] = NIL
        key[u] = NIL
    end for
    key[vertex] = 0
    tempQ = V[G]
    while tempQ != 0:
        u = Minimum(tempQ)
        for each v in V[G]:
            if A[u, v] != 0 and v belongs to tempQ and A[u, v] < key[v]:
                P[v] = u
                key[v] = A[u, v]
            end
        end
    end
```

The time complexity of this algorithm is $O(V^2)$.

13. ***(25 points)*** *Exercise 23.2-4.*

Solution:
We can use the Counting sort method to sort the edges. This would take a time complexity of /*theta*(V+E). The Krushkal's algorithm sorts the edges in a non decreasing order by their weights. Apart from the sorting process the Krushkal's algorithm takes O(Vlogv) to run and hence the overall time complexity would be O(V + E + VlogV). We could write that as O(E + VlogV).
Here, we have been given that the numbers are in the range of 1 to W, then we would get the time complexity as O(W+ E + VlogV).

14. ***(25 points)*** *Exercise 23.2-5.*

Solution:
When we consider the functions extract min, decrease key and addition of a vertex to a different list

to its new value, the time complexity of all adds up to constant runtime. Extracting minimum edge weight is again a constant runtime where we check the linked list and consider all the indices with non zero lists. We get an overall time complexity of O(VlogV + E) since we have a constant time complexity of all the other functions. The size of the array for constant W integers would be W+1. Every index has a linked list with the priority as its index. It uses the last index, w+1th position for vertex with infinite priority. It takes O(V) for building the priority queue. VlogV is the time taken for the EXTRACT-MIN loop operation while ElogV is for the implicit DECREASE-KEY operation. These both could be written as edge-weight loop with a loop of O(W). For removing the vertex from the list and adding it to another list, it takes O(1) time. Hence the modifies Prim's algorithm takes O(WV + E) time.

15. ***(Extra Credit)*** *Problem 23-1.*

Solution:

16. ***(Extra Credit)*** *Exercise 23.1-11.*

Solution:

17. ***(Extra Credit)*** *Write the code for Kruskal algorithm in a language of your choice. You will first have to read on the disjoint sets datastructures and operations (Chapter21 in the book) for an efficient implementation of Kruskal trees.*

Solution: