CS5800: Algorithms — Spring '21 — Virgil Pavlu

Homework 10
Submit via Gradescope

Name:Shri Datta Madhira (NUID: 001557772)
Collaborators:

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 condition given for E^2 as mentioned in the text book is, $G^2 = (V, E^2)$ such that $(u, v) \in E^2$ if and only G contains a path with at most two edges between u and v. That means G should have a path that should contain ≤ 2 edges between u and v.

So, G^2 graph contains the all the edges E in graph G plus paths where the number of edges between $(u, v) \leq 2$. The adjacency-matrix and adjacency-list calculation algorithms for such a graph is as follows,

Algorithm 1: Algorithm to calculate adjacency-matrix.

```
Function ADJACENCY-MATRIX(G):
    G'[1...V, 1...V] = 0
    For u = 1 to V
        For v = 1 to V
            For w = 1 to V
                If G[u,w] == 1 AND G[w,v] == 1 :
                    G'[u,v] = 1
```

Algorithm 2: Algorithm to calculate adjacency-list.

```
Function ADJACENCY-LIST(G):
    G'[1...V] = NULL
    For u in G.V
        For v in G.Adj[u]
            For w in G.Adj[v]
                G'[u] = w
```

Time Complexity for adjacency-list is $O(V.E^2)$.
Time Complexity for adjacency-matrix is $O(V^3)$.

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

Solution:

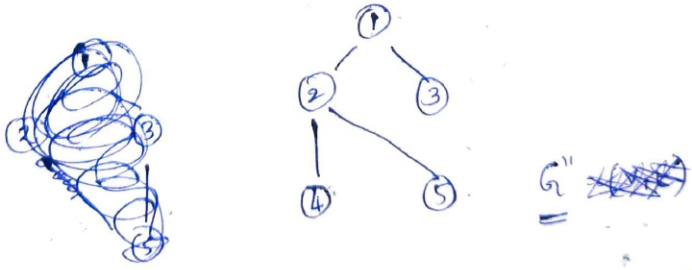
The simple path E_π from s to v in G should not be produced by BFS. The example graph we can use is,

22.2-6 :-

$G = (V, E)$ $G' = (V, E_\pi)$

- Here, $G' = (V, E_\pi)$ can never be produced using BFS because, when we run BFS both 4.π & 5.π will either be '2' or '3' (whichever comes first in the Queue).

So, the graph will look more like,



G''

As we can see, clearly not the same graph as G'.

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

Solution:

We have to divide the wrestlers into two group such that no two wrestlers of the same category have a rivalry. No babyface-babyface rivalry, or no heel-heel rivalry. That means, in the graph, if we assume wrestlers are vertices and rivalries are edges, there should not be an edge between same type of vertices.

If we take a vertex, all its adjacent vertices should be of different type. To do that we can assign a new attribute to the graph G called $type$. It stores the type of wrestler, $babyface$ or $heel$. Since babyface means good I will represent it with 1 and heel with 0. So, $babyface\ is\ 1$ and $heel\ is\ 0$.

Now, we apply normal BFS routine. We pick a source vertex and assign it o or 1 for the type. We go wave by wave and assign types such that no two adjacent vertices have the same type.

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

Solution:

Algorithm 3: Algorithm for DFS using stack.

```
Function DFS(G):
    For u in G.V
        u.color = WHITE
        u.π = NIL
    time = 0
    For u in G.V
        If u.color == WHITE :
            DFS-VISIT-STACK(G,u)
Function DFS-VISIT-STACK(G, u):
    stack = [u]
    While !stack.isEmpty()
        s = stack.POP()
        s.color = GRAY
        time += 1
        s.d = time
        For v in G.Adj[s]
            If v.color == WHITE :
                v.π = s
                stack.PUSH(v)
        s.color = BLACK
        time += 1
        s.f = time
```

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

Solution:

Algorithm 5: Algorithm to print DFS for Directed graph.

```
Function DFS(G):
    For u in G.V
        u.color = WHITE
        u.π = NIL
    time = 0
    For u in G.V
        If u.color == WHITE :
            DFS-VISIT-DIRECTED(G,u)

Function DFS-VISIT-DIRECTED(G, u):
    stack = [u]
    While !stack.isEmpty()
        s = stack.POP()
        time += 1
        s.color = GRAY
        s.d = time
        For v in G.Adj[s]
            If v.color == WHITE :
                v.π = s
                stack.PUSH(v)
                print(v,"is a Tree Edge.")
            ElseIf s.d < v.d :
                print(v,"is a Forward Edge")
            ElseIf v.color == GRAY :
                print(v,"is a Back Edge")
            Else
                print(v,"is a Cross Edge")
        s.color = BLACK
        time += 1
        s.f = time
```

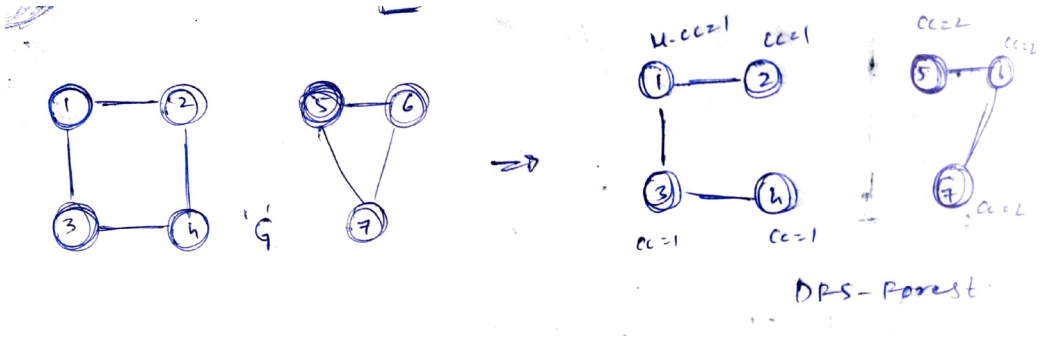
Algorithm 6: Algorithm to print DFS for undirected graph.

```
Function DFS-VISIT-UNDIRECTED(G, u):
    stack = [u]
    While !stack.isEmpty()
        s = stack.POP()
        s.color = GRAY
        time += 1
        s.d = time
        For v in G.Adj[s]
            If v.color == WHITE :
                v.π = s
                stack.PUSH(v)
                print(v,"is a Tree Edge.")
            Else
                print(v,"is a Back Edge")
        s.color = BLACK
        time += 1
        s.f = time
```

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

Solution:

We can use DFS and get as many trees in the DFS forest as we have connected components in the graph G. Let us assume we have a graph G with two connected components as shown in the image below. If we have to write the adjacency lists for these nodes, we will not get any of the nodes in the second component because none of the nodes in first component have connections with the nodes in the second component. So, when we run DFS we have to call DFS-VISIT() two times for two connected components. So, we will get two trees in the DFS forest. That means, we will have 2 trees in the DFS forest for 2 connected components in main graph G.



Generalizing this, we can say that, as we are calling DFS-VISIT() the same number of times as the number of connected components, there will be as many number of trees in the DFS forest as there are connected components in DFS.

Algorithm 7: Algorithm for DFS using stack.

```
Function DFS(G):
    For u in G.V
        u.color = WHITE
        u.π = NIL
    time = 0
    k = 0
    For u in G.V
        If u.color == WHITE :
            k += 1
            DFS-VISIT-DIRECTED(G,u,k)
```

```
Function DFS-VISIT(G, u, k):
    time += 1
    u.d = time
    u.color = GRAY
    u.cc = k
    For v in G.Adj[u]
        If v.color == WHITE :
            v.π = u
            v.cc = k
            DFS-VISIT(G,v)
    u.color = BLACK
    time += 1
    u.f = time
```

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

Solution:

Algorithm 8: Algorithm for Topological Sort using In-degree of vertices.

```
Function TOP-SORT-IN-DEGREE(G):
    in-degree[1...V] = 0
    For u in G.V
        For v in G.Adj[u]
            in-degree[v] += 1
    For u in G.V
        If in-degree[u] == 0 :
            Top-sort.add(u)
            For v in G.Adj[u]
                in-degree[u] -= 1
            del u from G.V
    Return top-sort
```

The above algorithm runs in O(V+E) time.
If the graph G has cycles, the topological sort will not output the vertices with the cycles. Because, if two vertices have cycles, both of them have in-degree > 0. So, it just skips past them.

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:

In BFS, the length of shortest path from vertex 's' to 't' is the level at which 't' occurs. From this we will also get the vertices involved in the path from 's' to 't'.

So, we run BFS on the graph G. Then the vertex 'v' should occur at level between 0 and $1+|V|/2$, i.e., $0 < l < 1 + |V|/2$. At least one of these levels should have a single vertex because if all these levels have ≥ 2 vertexes, then $2(|V|/2)$ is already the number of vertices.

Therefore, we have proved that there is one layer with only one vertex. That means, if we delete that vertex in the single vertex level, we delete all the paths from 's' to 't'.

Algorithm 9: Algorithm to find vertex v.

```
Function FIND-v(G):
    levels = BFS-Modified(G,s)
    For l in levels
        If l.length == 1 :
            Return l[0]
Function BFS-Modified(G,s):
    s.color = GRAY
    s.d = 0
    s.π = NIL
    s.level = 0
    level[1...G.V] = []
    level[0] = s
    q = Queue()
    q.PUT(s)
    While !q.isEmpty()
        s = q.GET()
        For v in G.Adj[s]
            If v.color == WHITE :
                v.color = GRAY
                v.d = u.d+1
                v.π = u
                v.level = s.level + 1
                level[v.level] = v
    Return level
```

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

Solution:

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

Solution:

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

Solution:

Let us assume T is the minimum spanning tree that is formed by including the edge (u,v).

Let us also assume there is another edge (x,y) which is an edge also going through the cut and is also included in the path from the vertex u to v forming a cycle.

(*Refer Figure* 23.3 *from CLRS textbook Page No.* 628)

For (x,y) to be the light edge, **w(x,y) < w(u,v)**.

Say that a tree T′ is formed by including (x,y) instead of (u,v). Then, the total weight of T′ will be less than the total weight of T, **w(T′) < w(T)**. Then T′ will be the new minimum spanning tree which brings a contradiction to the statement given in the question that said the tree that includes (u,v) is the minimum spanning tree.

Therefore, (u,v) is a light edge crossing some cut of the graph.

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

Solution:

Algorithm 10: Algorithm to find vertex v.

```
Function PRIM-MATRIX(G):
    isPresent[1...V] = false
    weights[1...V] = float("inf")
    parent[1...V] = 0
    weights[0] = 0
    For u = 1 to V
        min = float("inf")
        For v = 1 to V
            If isPresent[v] == false AND weights[v] < min :
                min = weights[v]
                minIndex = v
        isPresent[minIndex] = true
        For v = 1 to V
            If G[minIndex][v] < weights[v] AND !isPresent(v) :
                weights[v] = G[minIndex][v]
                parent[v] = minIndex
    Return parent, weights
```

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

Solution:

Kruskal's Algorithm takes $O(E \lg E)$ for sorting the edges. If the edges are in the range of 1 to $|V|$, then we can use counting sort to sort the edges in $O(V + E) = O(E)$ time. So, the total running time will be $O(V + E + E\alpha(V)) = O(E\alpha(V))$. This will not significantly speed up the existing algorithm because we wont use it anywhere other than in the sorting step.

If the edge weights are in the range 1 to W for some constant W, we can still use the method mentioned above and get $O(E + W) = O(E)$. So, the total running time would be $O(V + E + E\alpha(V)) = O(E\alpha(V))$. Therefore, no change.

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

Solution:

Prim's algorithm uses queue operations with edge weights as keys. So, we can implement the queue as an array of edge weights, where each edge weight has the corresponding vertices in a double linked list. For the EXTRACT-MIN operation we maintain another hash map that returns the memory address of a vertex in O(1) time.

If the edge weights are in the range of 1 to $|V|$, then the EXTRACT-MIN() step takes $\Theta(V)$ time. So, the total running time will be dominated by this step and the running time will be $\Theta(V^2)$. Which makes the current implementation with the Fibonacci Heap the better one.

If the edge weights are in the range of 1 to W where W is some constant, then both the EXTRACT-MIN operation take $O(W) = O(1)$ time, since W is a constant. So, the total running would be $O(E)$.

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: