CS5800: Algorithms — Spring '21 — Virgil Pavlu

Homework 10
Submit via Gradescope

Name: Xuran Feng
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.*

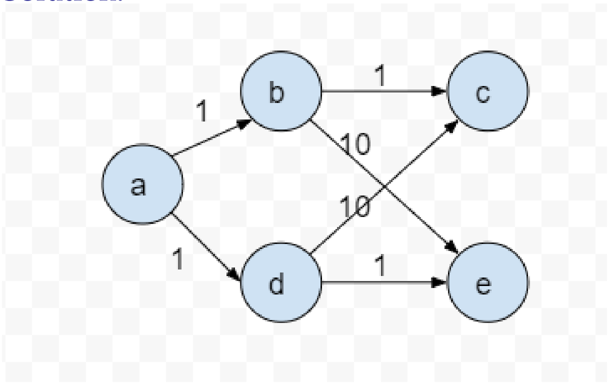
Solution: For each vertex v, iterate all neighbours, and for each neighbour n of v, mark all neighbours of n for v. It's like doing BFS of each vertex but with a limit to 2 waves.

```
For each vertex v of G:
    for each neighbour n1 of v:
        mark edge(v, n1)
        for each neighbour n2 of n1:
            mark edge(v, n2)
```

The running time of the graph square using adjacency matrix is $O(V^3)$ because we need to traverse neighbour of neighbour of each vertex; the running time of the graph square using adjacency list is $O(|V||E|)$ because for each vertex we may need to go through every edge.

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

Solution:



A valid set of tree edges are edge(a,b), edge(a,d), edge(b,c) and edge(d,e). This is why BFS fails to find shortest path when edges have weights. From the graph above, if starting from vertex a, the first wave of BFS will add b and d into the queue and edge(a,b) and edge(a,d) will be employed; however when b gets popped, edge(b,c) and edge(b,e) will be employed, c and e are added into the queue; when d gets popped, edge(d,c) and edge(d,e) will never be considered because c and e are already added to the queue. The problem is edge(b,c) and edge(d,e) are for sure the shortest path to c and e respectively but BFS will not employ edge(b,c) and edge(d,e) at the same time because edge(b,e) must be employed with edge(b,c) together, edge(d,c) and edge(d,e) must be employed together. So BFS can never produce a tree like this.

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

Solution: We can see all the wrestlers as vertex and the pair relation as an edge, so now a tree has been formed. Then we can do BFS search, we randomly start from a vertex and assign a number A to it meaning this vertex is already a babyface or heel, then mark this vertex as visited, for all neighbours of this vertex, assigning a number B to each meaning the neighbours must be in different parties and add each neighbour to the queue. Continuing the BFS, when a node n gets dequeued, for all neighbours of this node, if the neighbour already has a number and the number is same as n's number, return false; else if the neighbour's number is different from n's, continue checking other neighbours; else assign an opposite number to its neighbour and enqueue this neighbour.

```
BFS(G,s)
    for each vertex u:
        u.visited=false
    s.Number=A
    ENQUEUE(Q,s)
    while(Q.size>0)
        u=DEQUEUE(Q)
        for(each v that belongs to G.Adj[u])
            if v.visited=true then continue;
            if v.number!=null and v.number==u.number
                return false;
            if v.number!=null and v.number!=u.number
                continue;
            assign an opposite number to v
            ENQUEUE(v)
        u.visited=true
```

The running time of this algorithm is O(n+r) because it is a modified algorithm based on BFS and BFS has a time complexity of O(n+r).

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

Solution:

```
DFS(G)
    for each vertex u:
        u.color=WHITE
        u.p=NIL
    time=0
    stack={}
    for each vertex u:
        if u.color==WHITE
            stack.PUSH(u)
            DFS-VISIST(G,stack)
DFS-VISIT(G,stack)
    u.d=++time
    u.color=GRAY
    while(stack.size>0)
        u=stack.peek()
        while(u.hasWhiteNeighbour()):
            v=u.getWhiteNeighbour()
            v.d=++time
            v.color=GRAY
            v.p=u
            stack.push(v)
            u=v
        u=stack.pop()
        u.color=BLACK
        u.f=++time

hasWhiteNeighbour(u)
    for vertex v that are adjacent to u
        if v.color==WHITE
            return true
    return false
getWhiteNeighbour(u)
    for vertex v that are adjacent to u
        if v.color==WHITE
            return v
    return null
```

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

Solution:

```
DFS(G)
    for each vertex u:
        u.color=WHITE
        u.p=NIL
    time=0
    for each vertex u:
        if u.color==WHITE
            DFS-VISIST(G,u)
DFS-VISIT(G,u)
    u.d=++time
    u.color=GRAY
    for each v that is adjacent to u:
        if v.color==WHITE
            v.p=u
            output(”tree edge from u to v”)
            DFS-VISIT(G,v)
        else if v.color==GRAY
            output(”back edge from v to u”)
        else if v.d>u.d
            output(”forward edge from u to v”)
        else output(”cross edge from u to v”)
    u.color=BLACK
    u.f=++time
```

If G is undirected, then there is no cross edge because DFS will allow search as far as possible; back edge and forward edge are actually pointing to same edge because the undirected edge can be seen as a bidirectional edge, but the tree edge serves the same function.

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

Solution:
Because the graph now is undirected, the undirected relation between two vertex can be seen as a bidirectional relation, as long as there is an edge from u to v, then there must be an edge from v to u at the same time. So as long as the DFS can continue going, all vertex that have been traversed must be reachable for each other. However, all vertex may not be connected together, there could be several trees can form a forest, so the total times of running DFS before all vertex are all marked finished means in fact the number of separate trees in this forest.

```
DFS(G)
    for each vertex u:
        u.color=WHITE
        u.p=NIL
    time=0
    cc=1
    for each vertex u:
        if u.color==WHITE
            u.cc=cc
            DFS-VISIST(G,u)
            cc++
DFS-VISIT(G,u)
    u.d=++time
    u.color=GRAY
    for each v that is adjacent to u:
        if v.color==WHITE
            v.p=u
            v.cc=u.cc
            DFS-VISIT(G,v)
    u.color=BLACK
    u.f=++time
```

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

Solution: Use a queue to continue enqueuing vertex that has an in-degree 0(out-degree 1) and every time we dequeue a vertex from the queue, we iterate all its neighbours and decrement neighbour's in-degree by 1, if any neighbour reaches a 0 in-degree, enqueue it. Continue until the queue is empty.

```
TopoSort(G)
    queue={}
    for each vertex u:
        if u.in-degree==0
            queue.ENQUEUE(u)
    while(queue.size>0):
        u=queue.DEQUEUE()
        for each v that is adjacent to u:
            v.in-degree=v.in-degree - 1
        if v.in-degree==0
            queue.ENQUEUE(v)
```

Because the iterative TopoSort is actually modified from BFS with just different enqueuing and dequeuing orders, since BFS runs in time O(V+E), this topological sort also runs in O(V+E) time. If G has cycles, this algo will end too early because there are still vertex left to be traversed but since there is a deadlock so no vertex can first reach an in-degree 0 to be enqueued, the queue will become empty while some vertex are still unvisited.

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: Any path from s to t has at least 1+$|V|/2$ edges meaning if we do a BFS from s, there must be at least $|V|/2$ waves before t will be visited; if for each wave, there are two vertices, then total number of vertices will be $|V|$, however, s and t are still not counted, so if s and t are counted, the total number of vertices will exceed $|V|$ and thus a contradiction. So there must be a wave which only has one vertex and this vertex will be the only pass that every path needs to go through else it cannot form a path. The algorithm is BFS and the common vertex will belong to the wave which has only one vertex.

```
BFS(G,s)
    for each vertex u:
        u.color=WHITE
        u.d=∞
    s.color=GRAY
    s.d=0
    queue={}
    ENQUEUE(Q,s)
    while queue.size>0
        u=DEQUEUE(Q)
        for each v that belongs to u's neighbours:
            if v.color==WHITE
                v.color=GRAY
                v.d=u.d+1
                ENQUEUE(Q,v)
        u.color=BLACK
FIND-WAVE-HAS-ONE-VERTEX:
    bucket[]=Integer[$|V|/2$]
    for each vertex u:
        bucket[u.d]++
    for each bucket:
        if bucket slot==1 then return bucket key
```

The total running time is O(V+E) because the dominant part is to summrize the wave information by doing BFS and since BFS has a running time O(V+E), this is also 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:
Since the edge is in a MST, we can find the two separate cuts this edge(u,v) connects, if this edge is not the light edge that connects these two cuts, to exchange argument, then there must be another edge that has less weight than this edge(u,v); so in this case we definitely will replace (u,v) with that lighter edge since in this way we can get a MST for sure, then the edge(u,v) will not even appear in the MST, so there is a contradiction.

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

Solution: To achieve a $O(V^2)$ time complexity, we need to update each neighbour's key of a vertex after this vertex got visited and choose the shortest edge as next candidate. We use three arrays to keep track of status of all vertices. minEdge[] to track the minimum edge to that vertex, prev[] to track parent, visited[] to track whether this vertex has been visited.

```
Prim(G,u):
    visited[u]=true
    for each vertex v in G:
        if v!=u:
            prev[v]=u
            minEdge[v]=G[u][v]
            visited[v]=false
    for i=1 to n-1:
        tmp=INF, t=u
        for each vertex j in G:
            if(visited[j]==false&&minEdge[j]¡tmp):
                t=j
                tmp=minEdge[j]
        visited[t]=true
        for each vertex v in G:
            if(visited[j]==false&&G[t][j]¡minEdge[j]):
                minEdge[j]=G[t][j]
                prev[j]=t
```

This algo doesn't employ a min-heap structure to help finding a minimum next edge, instead it just brute force every possible edge and compare to find a minimum edge. The total running time is $O(V^2)$ because there is an outer loop and an inner loop nested with each other. And the loop times are both O(V) because every vertex gets queried all the time.

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

Solution: For Kruskal algorithm, the dominant part is union-find process and the quickest union-find is to do union by rank and path compression, the edges are only need to be sorted at first and waiting to be selected in ascending weights until all vertices are connected. So if the edge weights are limited to $|V|$, the original sorting time O(ElgE) can be reduced to O(V+E) using counting sort, since the graph is connected and V=O(E), so the sorting time is reduced to O(E). The total running time is O(V+E+Ea(V))=O(Ea(V)), left is just about disjoint-set process itself because edge weights are not used next.
If the edge weights are up to constant W, the sorting time is still O(E+W)=O(E) and this won't affect the O(Ea(V)) time complexity.

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

Solution: The time complexity is dominated by how to extract the minimum edge weight and decrease edge weight in an efficient way, according to textbook, if a Fibonacci heap is used, it takes O(E+VlgV) running time. To improve the running time, we can use an array and linked lists to achieve O(1) instead of O(lgv). If the edge weights are up to constant W, then the array is an array with W+1 slots and each slot has key 0,..W, each slot will have a linked list to store pair of vertice of edge whose weight equals to this key. To extarct minimum edge, we simply iterate all slots and find slot that has a non-empty linked list, this takes O(W)=O(1) time; to decrease edge weight we can simply remove it from current linked list and re-insert it into the new key list, the total running time now is O(E+VW)=O(E+V)=O(E).
If the weights are up to $|V|$, the total running time is O(E+V*V)=$O(E+V^2)$=$O(V^2)$ because extract minimum takes $\theta(V)$ time now. So we better switch back to Fibonacci heap.

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: