CS5800: Algorithms — Spring '21 — Virgil Pavlu

Homework 10
Submit via Gradescope

Name: Zerun Tian

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.*

The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$ such that $(u, v) \in E^2$ if and only G contains a path with at most two edges between u and v. Describe ... G^2 from G for both the adjacency-list and adjacency-matrix of G. Analyze the running times of your algorithms.

Solution:

For the adjacency-list, we do the following steps to generate G^2,

```
1:  function GraphSquaredAdjList(G)
2:      init G2.Adj as an array of size |G.V|
3:      for u in G.Adj do
4:          init added-set to be an empty set      ▷ to prevent duplicate edges being added to the list
5:          for v in G.Adj[u] do                   ▷ (u, v) is an edge
6:              ListInsert(G2.Adj[u], v)
7:              added-set.add(v)
8:              for w in G.Adj[v] do               ▷ (v, w) is an edge, thus (u, w) is a path of 2 edges
9:                  if w not in added-set then
10:                     ListInsert(G2.Adj[u], w)
11:     return G2
```

For every vertex u in $G.Adj$, we scan through every vertex v that is adjacent to u. $G2$ will have (u, v) as an edge. Moreover, for every v, we find each of its adjacent vertices w, which represents an edge (v, w). We know that (u, w) is a path of two edges, which qualifies to belong to E^2. We want to avoid adding duplicate vertices in $G2.Adj[u]$, so a set data structure called *added-set* is used to take care of this. The for loops at line 5 run at most $O(E)$ iterations, each of which has constant amount of work to do, so the runtime of this algorithm is $O(|V| \cdot |E|)$.

For the adjacency-matrix representation, we do the following steps,

```
1:  function GraphSquaredAdjMatrix(A)              ▷ adjacency matrix A of graph G
2:      A2 = A × A                                 ▷ matrix multiplication
3:      for i in 1 to |V| do
4:          for j in 1 to |V| do
5:              if A[i, j] == 1 then
6:                  A2[i, j] = 1
7:              if A2[i, j] ≥ 1 then
8:                  A2[i, j] = 1
9:      return A2
```

We can easily find paths of two edges by multiplying the matrix A of G by itself. For some index k ($1 \le k \le |V|$), if (i, k) and (k, j) are both 1s' in A, it means there is a path of 2 edges between i and j through the vertex k. After multiplication, the entry (i, j) in $A2$ will be at least 1 if we can find such ks'. To not confuse entry values ≥ 1 with edge weights, we replace values that are greater than 1 with 1. Furthermore, we copy edges of G into G^2 by filling 1 to entry $A2[i, j]$ whenever $A[i, j]$ is 1. The matrix multiplication can be done in $O(|V|^{\lg 7})$ using Strassen's algorithm shown in book page 82. The for loops take $O(|V|^2)$-time. Overall, the algorithm runs in $O(|V|^{\lg 7}) \approx O(|V|^{2.807})$-time.

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

Give an example of a directed graph $G = (V, E)$, a source vertex $s \in V$, and a set of tree edges $E_\pi \subseteq E$ such that for each vertex $v \in V$, the unique simple path in the graph (V, E_π) from s to v is a shortest path in G, yet the set of edges E_π cannot be produced by running BFS on G, no matter how the vertices are ordered in each adjacency list.

Solution:

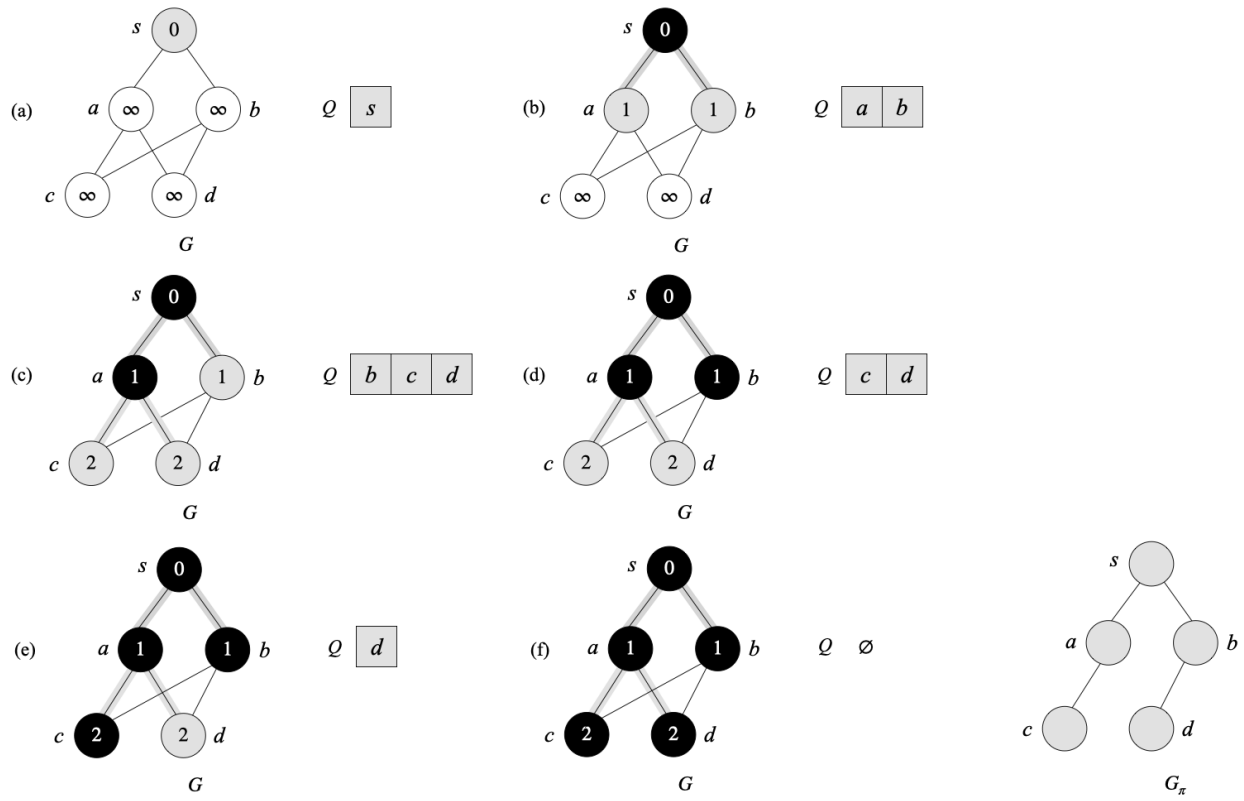

Diagram (a) shows the graph G with the start vertex added to the Q. In (b), we show an example where a appears before b in the adjacency list of vertex s. In (c), we pop the vertex a from the Q, then we subsequently add its children c and d to the queue (order doesn't matter). It is obvious now that c and d's predecessor is a. On the other hand, suppose in (b), we have vertex b appears before a in the adjacency list of vertex s. Consequently, the predecessor of c and d will be b. In either case, we won't obtain the paths shown in G_π which yet shows another way we can reach shortest path for every pair of s and v.

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

There are two types of professional wrestlers: "babyfaces" ("good guys") and "heels" ("bad guys"). Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose we have n professional wrestlers and we have a list of r pairs of wrestlers for which there are rivalries. Give an $O(n+r)$-time algorithm that determines whether it is possible to designate some of the wrestlers as babyfaces and the remainder as heels such that each rivalry is between a babyface and a heel. If it is possible to perform such a designation, your algorithm should produce it.

Solution:

Let's first model this problem as a graph. Each wrestler is a vertex in the graph, and each pair of rivalries is an edge in the graph. We are able to find a designation if the resulting graph exhibits bipartite property which allows us to split vertices into two groups where no edge is present within a group. If so, we can assign one group of vertices to be babyfaces, and the other to be heels, or the other way around.

To check if the graph is bipartite, we are going to run BFS on a vertex. Meanwhile, we mark vertices with two colors such that vertices of the same wave is colored the same, and vertices of two consecutive waves have alternate colors, say Red and Black. During the process, if we are about to color a vertex that is already colored using a different color from what it currently is, then we conclude that the graph is not a bipartite. After we have run through all vertices and edges, if we did not witness such a case, the graph is a bipartite. Indeed, the graph might not be connected, so we need to run this procedure for all disconnected components of the graph. In the end, we designate all Red vertices as "babyfaces" and the rest as "heels".

In this algorithm, we must go through all edges (rivalries) once. In addition, we visit each vertex at least once yet the total number of visits is upper bounded by number of edges r. Thus, the runtime of the algorithm is $O(n+r)$.

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

Rewrite the procedure DFS, using a stack to eliminate recursion.

Solution:

```
1:  function DFS-HELPER(G)
2:      let S be an empty stack
3:      time = 0
4:      for each vertex u ∈ G.V do
5:          u.color = WHITE
6:          u.π = NIL
7:      for each vertex u ∈ G.V do                ▷ in case there are disconnected components
8:          if u.color == WHITE then              ▷ the vertex is not yet processed
9:              time = time + 1
10:             u.d = time                        ▷ u is discovered
11:             u.color = GRAY
12:             S.push(u)
13:             while S is not empty do
14:                 cur = s.top()                 ▷ peek the top elm without removing it
15:                 child = NIL
16:                 for each vertex c in G.Adj[cur] do   ▷ find the first adjacent vertex that is white
17:                     if c.color == WHITE then
18:                         child = c
19:                         break
20:                 time = time + 1
21:                 if child == NIL then          ▷ all of cur's adjacency vertices are finished
22:                     cur.f = time              ▷ u is finished
23:                     cur.color = BLACK
24:                     S.pop()
25:                 else
26:                     child.π = cur
27:                     child.d = time
28:                     child.color = GRAY
29:                     S.push(child)
```

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

Modify the pseudocode for depth-first search so that it prints out every edge in the directed graph G, together with its type. Show what modifications, if any, you need to make if G is undirected.

Solution:

We modify the DFS-VISIT function such that it prints the edges accordingly, assuming the DFS function shown on page 604 stays as it is

```
1:  function DFS-VISIT(G, u)
2:      time = time + 1
3:      u.d = time
4:      u.color = GRAY
5:      for each v ∈ G.Adj[u] do
6:          if v.color == WHITE then
7:              print "(" + u + ", " + v + ") is a tree edge"
8:              v.π = u
9:              DFS-VISIT(G, v)
10:         else if v.color == GRAY then
11:             print "(" + u + ", " + v + ") is a back edge"
12:         else                                          ▷ v is black
13:             if u.d < v.d then                         ▷ u is discovered first
14:                 print "(" + u + ", " + v + ") is a forward edge"
15:             else
16:                 print "(" + u + ", " + v + ") is a cross edge"
17:     u.color = BLACK
18:     time = time + 1
19:     u.f = time
```

We don't need to do anything particular for G that is undirected.

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

Show that we can use a depth-first search of an undirected graph G to identify the connected components of G, and that the depth-first forest contains as many trees as G has connected components. More precisely, show how to modify depth-first search so that it assigns to each vertex v an integer label $v.cc$ between 1 and k, where k is the number of connected components of G, such that $u.cc = v.cc$ if and only if u and v are in the same connected component.

Solution:

It is relatively trivial to identify connected components in an undirected graph. Vertices completely severed from other groups would be considered a separate connected component. We modify the code as follows,

```
1:  function DFS(G)
2:      for each vertex u ∈ G.V do
3:          u.color = WHITE
4:          u.π = NIL
5:      time = 0
6:      k = 1
7:      for each vertex u ∈ G.V do
8:          if u.color == WHITE then      ▷ found an unvisited vertex after the last DFS-VISIT if any
9:              u.cc = k
10:             DFS-VISIT(G, u)                    ▷ all vertices reachable from u are explored
11:             k = k + 1
```

Notice the line 6, 9, and 11 in the above pseudocode.

DFS-VISIT explores all vertices of a connected component given the start vertex u. Their labels should be consistent with that of the start vertex u.

```
1:  function DFS-VISIT(G, u)
2:      time = time + 1
3:      u.d = time
4:      u.color = GRAY
5:      for each v ∈ G.Adj[u] do
6:          if v.color == WHITE then
7:              v.cc = u.cc
8:              v.π = u
9:              DFS-VISIT(G, v)
10:     u.color = BLACK
11:     time = time + 1
12:     u.f = time
```

Notice the line 7 in the above pseudocode.

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

Another way to perform topological sorting on a directed acyclic graph $G = (V, E)$ is to repeatedly find a vertex of in-degree 0, output it, and remove it and all of its outgoing edges from the graph. Explain how to implement this idea so that it runs in time $O(V + E)$. What happens to this algorithm if G has cycles?

Solution:

```
1:  function TYPOLOGICAL-SORT(G)
2:      let indegree be an array of zeros of size |G.V|
3:      for each vertex u ∈ G.V do                          ▷ compute in-degree for every vertex
4:          for each vertex v ∈ G.Adj[u] do
5:              indegree[v] = indegree[v] + 1
6:      let Q be a queue for storing vertices               ▷ only store vertices of degree 0
7:      for each vertex u ∈ G.V do
8:          if indegree[u] == 0 then
9:              Q.enqueue(u)
10:     let ans be an empty array to store the sorted vertices
11:     while Q is not empty do
12:         cur = Q.dequeue()
13:         ans.push(cur)
14:         for each vertex u ∈ G.Adj[cur] do
15:             indegree[u] = indegree[u] − 1
16:             if indegree[u] == 0 then
17:                 Q.enqueue(u)
18:     for each vertex u ∈ G.V do
19:         if indegree[u] > 0 then
20:             error "failed to do typological sort due to a cycle"        ▷ raises an error
21:     return ans
```

From line 2 to 5, we compute the in-degree of every vertex which takes $O(V + E)$-time. From line 6 to 9, we store those vertices of 0 in-degree in a queue. This step takes $O(V)$-time. From line 10 to 17, we iteratively get a vertex with 0 in-degree from the Q and "remove" all its edges by decrementing the in-degree of the respective vertices. This step takes $O(V + E)$-time. After there is no vertex left in the Q, we use $O(V)$-time to check if every vertex's in-degree is zero. If so, we can output the vertices in sorted order; otherwise, we claim there is a cycle that refrains us from doing typological sort. Overall, the algorithm runs in $2 \cdot O(V + E) + 2 \cdot O(V) = O(V + E)$-time.

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:

Because there are at least $1+|V|/2$ edges from s to t, there are at least $1+|V|/2$ waves. When there are more than $|V|/2$ waves, we have to have some waves with only 1 vertex because we don't have enough vertices to put 2 vertices into all waves, and each wave needs to have at least 1 vertex. Therefore, a common vertex v for all paths from s to t can be found in a wave that has only one vertex.

We devise an algorithm that basically modifies BFS to trace such a v. Since BFS visits nodes wave by wave, we can maintain a counter to record the number of vertices it has seen in a wave. Specifically, we initialize two variables before the while loop: an integer to count the number of vertices in a wave and a pointer to the previous vertex which BFS visited (dequeued). The counter is initialized as 0, and the pointer is initialized as NIL. At every iteration of the while loop, we update the counter and pointer accordingly. As soon as we dequeued a vertex whose wave is 1 larger than the current wave, we reset the counter. Before we do the reset, we check if the counter is 1, meaning there is only one vertex present in that wave. We also make sure we are not in the initial wave that contains s. If so, a common vertex v is pointed by that pointer we maintained, then we can just return it to caller. In the worst cast, a v is present at the last wave before t. The runtime is thus the same as BFS, which is $O(V+E)$.

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

Solution:

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

Solution:

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

Show that if an edge (u, v) is contained in some minimum spanning tree, then it is a light edge crossing some cut of the graph.

Solution:

Let's remove the edge (u, v) from the minimum spanning tree. Then, we have two minimum spanning trees: one contains the vertex u and the other contains the vertex v. Now, we consider a cut that respects the set of edges in both trees and find a cross edge (u', v') that is shorter than (u, v). The (u', v') edge is a safe edge we should pick to construct a minimum spanning tree that does not include (u, v). This makes a contradiction that the (u, v) edge is contained in the minimum spanning tree. Therefore, we proved that the edge (u, v) is indeed a light edge crossing some cut of the graph.

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

Suppose that we represent the graph $G = (V, E)$ as an adjacency matrix. Give a simple implementation of Prim's algorithm for this case that runs in $O(V^2)$ time.

Solution:

```
1:  function MST-PRIM-ADJ-MATRIX(G, w, r)
2:      for each u ∈ G.V do                          ▷ clean up the parent pointer for each vertex
3:          u.π = NIL
4:      let C be an array of ∞ of size |G.V|  ▷ C stores the min cost to connect the i-th vertex to tree
5:      for i = 1 to |G.V| do                        ▷ update the costs for the root's neighbors
6:          if G.A[r, i] == 1 then
7:              C[i] = w(r, i)
8:              i.π = r
9:      for i = 1 to |G.V| − 1 do
10:         k = 1                                    ▷ k is the index to the min value of C
11:         min-c = ∞
12:         for j = 1 to |V| do
13:             if C[j] < min-c then
14:                 min-c = C[j]
15:                 k = j
16:         for u = 1 to |G.V| do                    ▷ update k's neighbors in C
17:             if G.A[k, u] == 1 and w(k, u) < C[u] then
18:                 C[u] = w(k, u)
19:                 u.π = k
```

The algorithm implicitly maintains the minimum spanning tree A of G. In the end, we have the tree A as a set of edges that satisfy,

$$A = \{(v, v.\pi) : v \in V - \{r\}\}$$

It is trivial to see that the for loops of line 2-3, and line 5-8 runs in $O(V)$-time respectively. The for loop of lines 12-15 runs in $O(V)$-time as it iterates through the array C to find the index of the minimum value. The for loop of lines 16-19 has V iterations each of which does constant amount of work. These two loops run in $O(V)$-time within in the for loop at line 9, which results in $O(V \cdot (V-1)) = O(V^2)$-time. Overall, this algorithm runs in $2 \cdot O(V) + O(V^2) = O(V^2)$-time.

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

Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Kruskal's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W?

Solution:

Because edge weight is in a range of values that is finite and known, we could use counting sort to arrange the edges in non-decreasing order. This pre-kruskal step thus takes $O(E)$ time. The for loop of lines 5-8 in the MST-KRUSKAL still runs in $O(E\alpha(V))$ time per the explantation in page 633 of the book. Overall, the algorithm runs in $O(E) + O(E\alpha(V)) = O(E\alpha(V))$ time. If the edge weights are integers in the range from 1 to W, we still want to use counting sort to speed up the pre-kruskal part to $O(n+k) = O(E+W)$ time. In this case, the algorithm runs in $O(E+W) + O(E\alpha(V)) = O(E\alpha(V))$ time because W is a constant.

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

Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Prim's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W?

Solution:

We devise a data structure to replace the queue in the original MST-PRIM while keeping the structure of the algorithm. Our goal is to have a new data structure and some auxiliary data to make EXTRACT-MIN and DECREASE-KEY run as fast as possible.

```
1:  function MST-PRIM(G, w, r)
2:      for each u ∈ G.V do
3:          u.key = ∞
4:          u.π = NIL
5:      let in-tree be an empty hash set
6:      let Q be an empty array of size |V|
7:      ADDVERTEX(Q, r, 1)                    ▷ adding the root vertex to the list at index 1 of the Q
8:      np = Linkedlist(1)                    ▷ np maintains indices to non-empty lists of Q
9:      while Q ≠ ∅ do
10:         u = EXTRACT-MIN(Q, np)
11:         in-tree.add(u)
12:         for each v ∈ G.Adj[u] do
13:             if v.key == ∞ then
14:                 v.key = w(u, v)
15:                 ADDVERTEX(Q, v, w(u, v))          ▷ O(1)-time to add a vertex to Q
16:             else if v ∉ in-tree and w(u, v) < v.key then
17:                 v.π = u
18:                 v.key = w(u, v)
19:                 DECREASE-KEY(Q, v, w(u, v), np)   ▷ move v to the linked list of index w(u, v)
```

The data structure Q is designed as follows. Let's have an array of size $|V|$ where each element in the array stores a list of vertices that have distance to the prim tree matching the array index.

To extract a min, we call EXTRACT-MIN(Q, np) which finds the first non-empty list in Q through np and removes the first node of that list. This is done in $O(1)$ time. The removed node is returned which is added to the prim tree. In addition, for its neighbors whose keys are still ∞, we add them to Q. Otherwise, for neighbors who are not already in the prime tree, we update their keys and locations in Q to maintain the invariant of our data structure when $w(u, v)$ is indeed smaller than the current $v.key$. Thus, DECREASE-KEY is done in $O(1)$ time to move a node v from where it was to the list at index $w(u, v)$ by just changing several pointers. The ADDVERTEX function is adding a vertex to the list at given index of Q, which is done in $O(1)$ time. And notice that, np, the linked list of indexes to non-empty lists of Q, could be managed in $O(1)$-time.

It's trivial to see that the for loop in lines 2-4 runs in $O(V)$ time. Some initializations in lines 5-8 runs in $O(1)$ time. The for loop in lines 12-19 executes $O(E)$ time altogether because the sum of the lengths of all adjacency lists is $2|E|$. Within the for loop, we only have several constant-time

operations as detailed earlier. The line 11 and 12 are constant time operations within the while loop at line 9, so $O(V)$ time for that.

Overall, the algorithm runs in $O(V) + O(1) + O(E) + O(V) = O(E + V)$ time.

Similarly, for integer edge weights in the range from 1 to some constant W, we use the same algorithm but initialize Q to be of size $|W|$ to maintain the list of vertices. So, its runtime would also be $O(E + V)$.

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: