CS 344
Fall 2022

Problem Set 3
Due: Nov 21, 2022, 11:59 p.m.

Name: (Dong Shu) **NetID**: (ds1657)

Honor Code: Students may discuss and work on homework problems in groups, which is encouraged. However, each student must write down their solutions independently to show they understand the solution well enough to reconstruct it by themselves. Students should clearly mention the names of the other students who offered discussions. We check all submissions for plagiarism. We take the honor code seriously and expect students to do the same.

Instruction for Submission: This homework has a total of 100 points, it will be rescaled to 10 points as the eventual score. We encourage you to use LaTex to write your answer, because it's particularly suitable to type equations and it's frequently used for writing academic papers. We have provided the homework3.tex file for you, you can write your answer to each question in this .tex file directly after the corresponding question, and then compile the .tex file into a PDF file for submission. As a quick introduction to LaTex, please see this *Learn LaTeX in 30 minutes* [1]. Compiling a .tex file into a PDF file needs installing the LaTex software on your laptop. It's free and open source. But if you don't want to install the software, you can just use this website `https://www.overleaf.com/`, which is also free of charge. You can just create an empty project in this website and upload the homework1.zip, and then the website will compile the PDF for you. You can also directly edit your answers on the website and instantly compile your file.

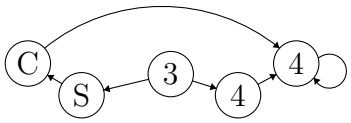
You can also use Microsoft Word or other software if you don't want to use LaTex. If so, please clearly number your answers so that we know which answer corresponds to which question. You also need to save your answers as a PDF file for submission.

Finally, please submit your PDF file only. You should name your PDF file as "Firstname-Lastname-NetID.pdf".

Late Policy: The homework is due on 11/21 (Monday) at 11:59pm. We will release the solutions of the homework on Canvas on 11/25 (Friday) 11:59pm. If your homework is submitted to Canvas before 11/21 11:59pm, there will no late penalty. If you submit to Canvas after 11/21 11:59pm and before 11/25 11:59pm (i.e., before we release the solution), your score will be penalized by 0.9^k, where k is the number of days of late submission. For example, if you submitted on 11/24, and your original score is 80, then your final score will be $80 * 0.9^3 = 58.32$ for $24 - 21 = 3$ days of late submission. If you submit to Canvas after 11/25 11:59pm (i.e., after we release the solution), then you will earn no score for the homework.

Drawing graphs: You might try `http://madebyevan.com/fsm/` which allows you to draw graphs with your mouse and convert it into LaTeX code:

[1] `https://www.overleaf.com/learn/latex/Learn_LaTeX_in_30_minutes`



You can also draw by hand and insert a picture.

Make best use of picture in Latex: If you think some part of your answer is too difficult to type using Latex, you may write that part on paper and scan it as a picture, and then insert that part into Latex as a picture, and finally Latex will compile the picture into the final PDF output. This will make things easier. For instructions of how to insert pictures in Latex, you may refer to the Latex file of our homework 1, which includes several examples of inserting pictures in Latex.

Discussion Group (People with whom you discussed ideas used in your answers if any):

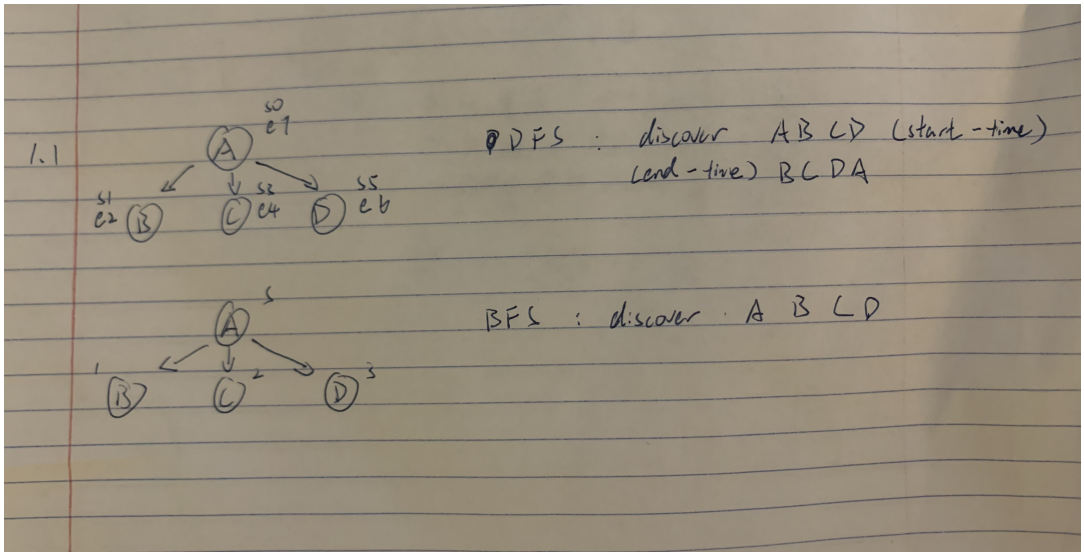
TianLe Chen

I acknowledge and accept the Honor Code. Please type your initials below:
Signed: (DS)

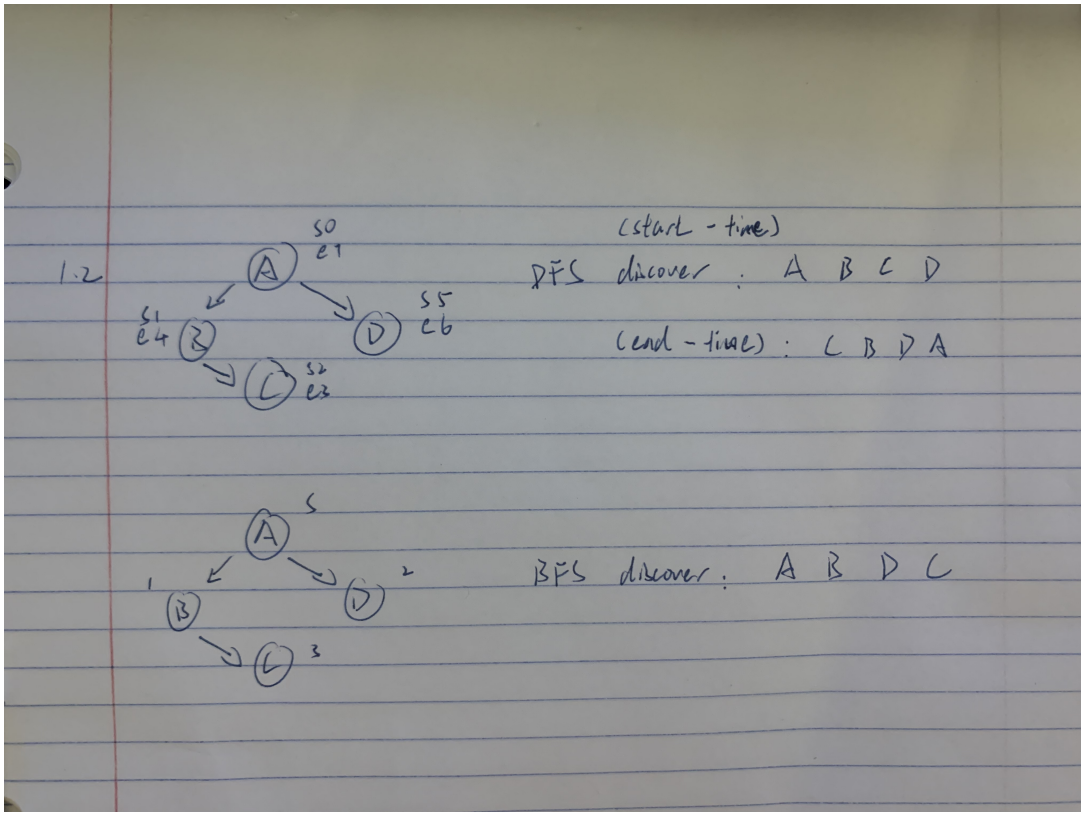
---

1. **Warmup with DFS/BFS.** (10 points)

   (1) Give one example of a directed graph on four vertices, A, B, C, and D, such that both depth-first search and breadth-first search discover the vertices in the same order when started at A.



(2) Give one example of a directed graph where DFS and BFS discover the vertices in a different order when started at A.



"Discover" means the time that the algorithm first reaches the vertex, referred to as `start_time` during lecture. Assume that both DFS and BFS iterate over outgoing neighbors in alphabetical order.

[We are expecting a drawing of your graphs and an ordered list of vertices discovered by DFS and BFS.]

2. **Warmup with Dijkstra.** (20 points)

   Let $G = (V, E)$ be a weighted directed graph. For the rest of this problem, assume that $s, t \in V$ and that **there exists a directed path from s to t.**

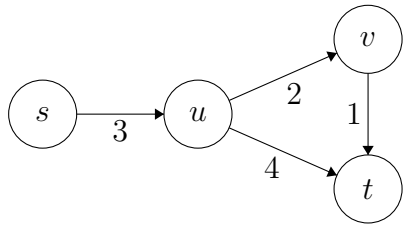
   For the rest of this problem, refer to the implementation of Dijkstra's algorithm given by the pseudocode below.

```
dijkstra_st_path(G, s, t):
  for all v in V, set d[v] = Infinity
  for all v in V, set p[v] = None

  // we will use p to reconstruct the shortest s-t path at the end
  d[s] = 0
  F = V
  D = []
  while F isn't empty:
    x = vertex v in F such that d[v] is minimized
    for y in x.outgoing_neighbors:
      d[y] = min( d[y], d[x] + weight(x,y) )
      if d[y] was changed in the previous line, set p[y] = x
    F.remove(x)
    D.add(x)

  // use p to reconstruct the shortest s-t path
  path = [t]
  current = t
  while current != s:
    current = p[current]
    add current to the front of the path
  return path, d[t]
```

The variable `p` maintains the "**p**arents" of the vertices in the shortest s-t path, so it can be reconstructed at the end.

Step through `dijkstra_st_path`(G, s, t) on the graph G shown below. Complete the table below to show what the arrays `d` and `p` are at each step of the algorithm, and indicate what path is returned and what its cost is.

G



[We are expecting the table below filled out, as well as the final shortest path and its cost. No further justification is required.]

	d[s]	d[u]	d[v]	d[t]	p[s]	p[u]	p[v]	p[t]
When entering the first while loop for the first time, the state is:	0	∞	∞	∞	None	None	None	None
Immediately after the first element of D is added, the state is:	0	3	∞	∞	None	s	None	None
Immediately after the second element of D is added, the state is:	0	3	5	∞	None	s	u	None
Immediately after the third element of D is added, the state is:	0	3	5	7	None	s	u	u
Immediately after the fourth element of D is added, the state is:	0	3	5	6	None	s	u	v

3. **Fun with Reductions.** (15 points) Suppose the economies of the world use a set of currencies $C_1, \ldots, C_n$; think of these as dollars, pounds, Bitcoin, etc. Your bank allows you to trade each currency C_i for any other currency C_j, and finds some way to charge you for this service. Suppose that for each ordered pair of currencies (C_i, C_j), the bank charges a flat fee of $f_{ij} > 0$ dollars to exchange C_i for C_j (regardless of the quantity of currency being exchanged).

   Describe an algorithm which, given a starting currency C_s, a target currency C_t, and a list of fees f_{ij} for all $i, j \in \{1, \ldots, n\}$, computes the cheapest way (that is, incurring the least in fees) to exchange all of our currency in C_s into currency C_t. Also, justify the its runtime.

   [We are expecting a description or pseudocode (either is OK) of your algorithm, as well as a brief justification of its runtime. You can use any algorithm we have learned in class.]

   answer:
   Following code is a modify version of the Dijkstra's code that provide in question 2. Since we only need to know the smallest fee, so we can ignore the 'parent' parameter. After we update every d[y] in the graph, we can just return the d[Ct], which will be our smallest fee, because at every step we are using min(d[y], d[x] + f_{ij}(x,y)) to find the cheapest path.
   The running time for this algorithm will be $O(|E|+|V|\log|V|)$. Because for each vertex it can connect to (V-1) vertices, and we have total of V vertices. Also, during the algorithm we need to go through every edge. Therefore, our running time for Dijkstra algorithm will be $O(|E|+|V|\log|V|)$.

```
Currency(f_ij, Cs, Ct):
    for all v in Graph, set d[v] = Infinity

    // we will use Dijkstra algorithm to find the smallest fee
```

```
d[Cs] = 0
F = all_vertices
while !F.isEmpty():
    x = vertex v in F such that d[v] is minimized
    for y in x.outgoing_neighbors:
        d[y] = min(d[y], d[x] + f_ij(x,y))
    F.remove(x)
return d(Ct)
```

4. **Social engineering.** (20 points)

Suppose we have a community of n people. We can create a directed graph from this community as follows: the vertices are people, and there is a directed edge from person A to person B if A would forward a rumor to B. Assume that if there is an edge from A to B, then A will always forward any rumor they hear to B. Notice that this relationship isn't symmetric: A might gossip to B but not vice versa. Suppose there are m directed edges total, so $G = (V, E)$ is a graph with n vertices and m edges.

Define a person P to be *influential* if for all other people A in the community, there is a directed path from P to A in G. Thus, if you tell a rumor to an influential person P, eventually the rumor will reach everybody. You have a rumor that you'd like to spread, but you don't have time to tell more than one person, so you'd like to find an influential person to tell the rumor to.

In the following questions, assume that G is the directed graph representing the community, and that you have access to G as an array of adjacency lists: for each vertex v, in $O(1)$ time you can get a pointer to the head of the linked lists v`.outgoing_neighbors` and v`.incoming_neighbors`. Notice that G is not necessarily acyclic. In your answers, you may appeal to any statements we have seen in class, in the notes, or in CLRS.

(a) (10 points) Show that all influential people in G are in the same strongly connected component, and that everyone in this strongly connected component is influential.

[Hint: You need to refer the definition of strongly connected component, and you can prove using either induction or contradiction.]

answer:

During the class we know that strongly connected components means for all pairs of vertices u and v, there's a path from u to v and a path from v to u.

We can prove all influential people in G are in the same SCC with contradiction.

If not all influential people in G are in the same strongly connected component. Then that means there exist other influential people in the outside of this SCC. However, the SCC metagraph is a directed acyclic graph, which means the influential person that is outside of the SCC can either talk to the SCC or hear from the SCC, but not both, because there is no cycle path between the SCC and the outside influential person. Otherwise, they can merge into one SCC. In this case, it will leave us with two cases, the first one is the outside influential person can only talk to the SCC. In

this case, the original SCC will no longer have influential people, because it violate the definition of influential people, which is that for all people A, there is a directed path from P to A in G. Since the SCC cannot talk to the outside influential person, so the people in SCC can no longer be influential. In this case, the outside influential person will form its own SCC, and this new SCC will be influential, because it can talk to the original SCC, which means the new SCC has directed path for all people. Since the outside influential person forms its own SCC, and of course this new SCC contains all the influential people. The other case is when the outside influential person can only hear from the SCC. Then this outside person will not be an influential person, because he cannot talk to this SCC, which violates the definition of influential people. Therefore, if the outside influential people can only talk to the SCC, then the SCC will not be influential and the outside people will form its own SCC. If the outside influential people can only hear from the SCC, then outside people will not be influential. However, if a outside influential person can both talk and hear from the SCC, then the outside people and SCC can collapse into one SCC. Therefore, the contradiction above shows that all influential people in G are in the same SCC.

We can prove everyone in this strongly connected component is influential by using contradiction.
If not everyone in this SCC is influential. Then it means that there exist not influential people in this SCC. In this case, this not influential person will not have a directed path from u to all other people v in G. Which means in this SCC, for all pairs of vertices u and v, there may not be a path from u to v, and may not be a path from v to u. In this case, it violate the definition of SCC, which is for all pairs of vertices u and v, there's a path from u to v and a path from v to u. Therefore, this SCC should only contain influential people and all un-influential people should not be in this SCC. In this case, our contradiction above proved that everyone in this SCC is influential.

(b) (10 points) Suppose that an influential person exists. Give an algorithm that, given G, finds an influential person.

[We are expecting a description or pseudocode of your algorithm and a short argument about the runtime. You can borrow any algorithm that we have learned in class.]

answer:
I will use Kosaraju's algorithm to find the influential person. We first use DFS to go through every node, and store the start and end time for each node. After this process, we will have a list of nodes from smallest end_time to biggest end_time. It will give us running time $O(|V| + |E|)$. Then we reverse the graph and erase the status of the node, which will give us running time $O(|V| + |E|)$. Then we do DFS again, and this time we will start at the node with highest end_time, which is the last node of the list. Once we find a complete SCC, then that means we find the influential person, then we will return it. It will give us running time $O(|V| + |E|)$. Thus, we will have total running time of $O(3(|V| + |E|)) = O(|V| + |E|)$.

```
cur_time = 0
list = []
algorithm dfs(u, cur_time):
    u.start_time = cur_time
    cur_time += 1
    u.status = 'in_progress'
    for v in u.neighbors:
        if v.status is 'unvisited':
            cur_time = dfs(v, curtime)
            cur_time += 1
    u.end_time = cur_time
    u.status = 'done'
    list.append(u)
    return cur_time

algorithm reverse_edge(G):
    for all edge in G:
        reverse edge
    for all node in G:
        empty status
    return G

scc = []
algorithm dfs_scc(u, G):
    u.status = 'in_progress'
    for v in u.neighbors:
        if v.status is 'unvisited':
            scc.append(dfs_scc(v, G))
    u.status = 'done'
    return scc

algorithm Kosaraju(G)
    u = Random_choose_node(G)
    dfs(u, cur_time)
    G = reverse_edge(G)
    last_node = list.length()
    return dfs_scc(list[last_node - 1], G)
```

5. **Job Sequencing Problem.** (15 points)

   In this problem we have n jobs $j_1, j_2, ..., j_n$, each has an associated deadline $d_1, d_2, ..., d_n$ and profit $p_1, p_2, ..., p_n$. Profit will only be awarded or earned if the job is completed before the deadline. We assume that each job takes 1 unit of time to complete. The objective

is to earn maximum profit when only one job can be scheduled or processed at any given time.

Provide the pseudocode of an algorithm to find the sequence of jobs to do with the maximum total profit. Also describe the main idea of your algorithm using plain language.

[Hint: You can select the jobs in a greedy way. You can use the following example to help your analysis.]

Job	J1	J2	J3	J4	J5
Deadline	2	1	3	2	1
Profit	60	100	20	40	20

The best job sequence would be $J2 \rightarrow J1 \rightarrow J3$.

answer:
First we go through every job in the Jobsss, and then put jobs which have deadline 1 into the job[], and then delete those jobs from Jobsss. After we find every job that has deadline 1, then we will find which job in deadline 1 has biggest profit, and then put this job into the answer[]. After that we will empty the job[] to store jobs in next deadline, and increment deadline by 1. Then we will repeat the process. Our code will end when Jobsss is empty, which means we have already find every biggest profit job in every deadline. By that time we will find our answer

```
Job_Sequencing(Jobs):
    answer = []
    Jobsss = Jobs
    deadline = 1
    job = []
    answer = []
    while(!Jobsss.isEmpty):
        for(int i = 0; i<Jobsss.length(); i++):
            if(Jobsss.Deadline = deadline):
                job.append(Jobsss[i])
                Jobsss.remove(i)
        find the biggest profit 'j' in job[]
        answer.append(j)
        empty job[]
        deadline = deadline + 1
    return answer
```

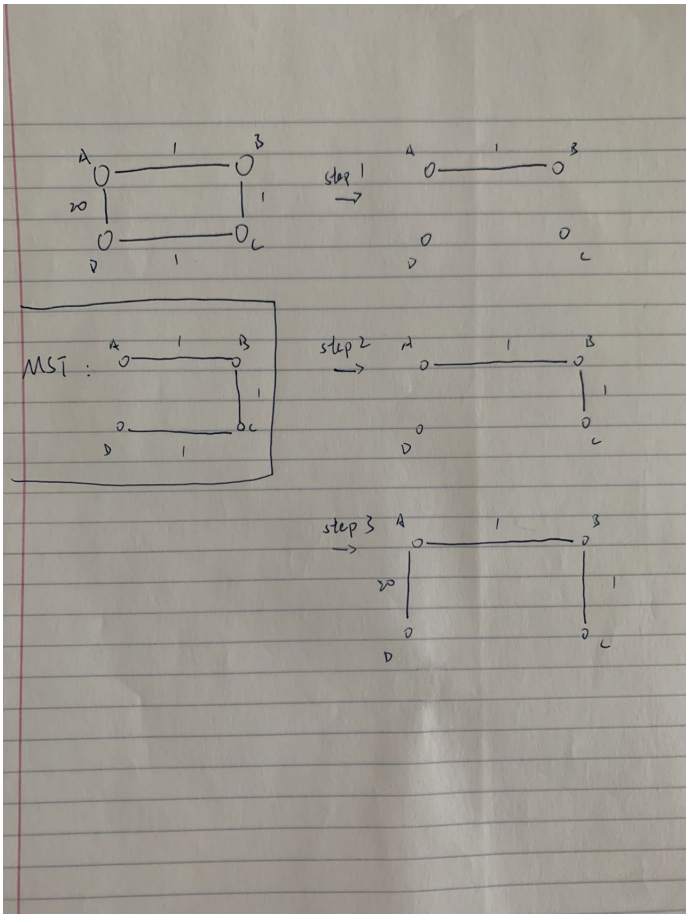
6. **Alternative Minimum Spanning Trees (23.4 CLRS).** (20 points) In this problem, we give pseudocode for two different algorithms. Each one takes a connected graph and a weight function as input and returns a set of edges T. For each algorithm, either show that T is a minimum spanning tree (give a brief justification of the algorithm) or show that T is not a minimum spanning tree (give a counter-example).

a. (10 points) **MAYBE-MST-A** (G, w)
1. sort the edges into non-increasing order of edge weights w
2. $T = E$
3. for each edge $e \in E$, taken in non-increasing order by weight
4. if $T - \{e\}$ is a connected graph
5. $T = T - \{e\}$
6. return T

answer:
This algorithm will give us a minimum spanning tree. Because on line 1, it sorts the edge into a non-increasing (from biggest to smallest) order. Then on line 4, he is saying after I delete this edge, will the tree still be connected. If yes then I will delete this edge, if no then go to next edge. Since, we are checking the biggest edge to smallest edge, so we will delete the unnecessary big edge first, and only include those essential big edges. In the end, the algorithm will return after we delete every unnecessary edge from biggest to smallest. In this case, even if we do have big edge, this edge will be essential for the graph. So our spanning tree weight will be smallest,

b. (10 points) **MAYBE-MST-B** (G, w)
1. $T = null$
2. for each edge e, taken in arbitrary order
3. if $T \cup \{e\}$ has no cycles
4. $T = T \cup \{e\}$
5. return T

answer:
This algorithm cannot give us a minimum spanning tree. Because on line 3, it is saying if I include this edge, will this graph have cycle, if yes go to the next edge, if no then this graph will add this edge. This algorithm will give us a spanning tree, but it will not give us a minimum spanning tree, because it did not consider the weight at all, since it take edge e arbitrarily.
An counter example will be the picture below. At first we will have a graph ABCD. AB edge = 1, BC edge = 1, CD edge = 1, and AD edge = 20. When we pick the first edge, we arbitrary picked AB, and we find out there has no cycles, so we include this edge. Then we arbitrary picked BC, and we find out there has no cycles, so we include this edge. Then we arbitrary picked AD, and we find out there has no cycle, so we include this edge. Therefore, our spanning tree will looks like AD, AB, BC, and it has total weight of 22. However, the true minimum spanning tree should looks like AB, BC, CD, and the weight is 3.


A
B
C
D
1
20
step 1
MST :
step 2
step 3