Problem Set 6

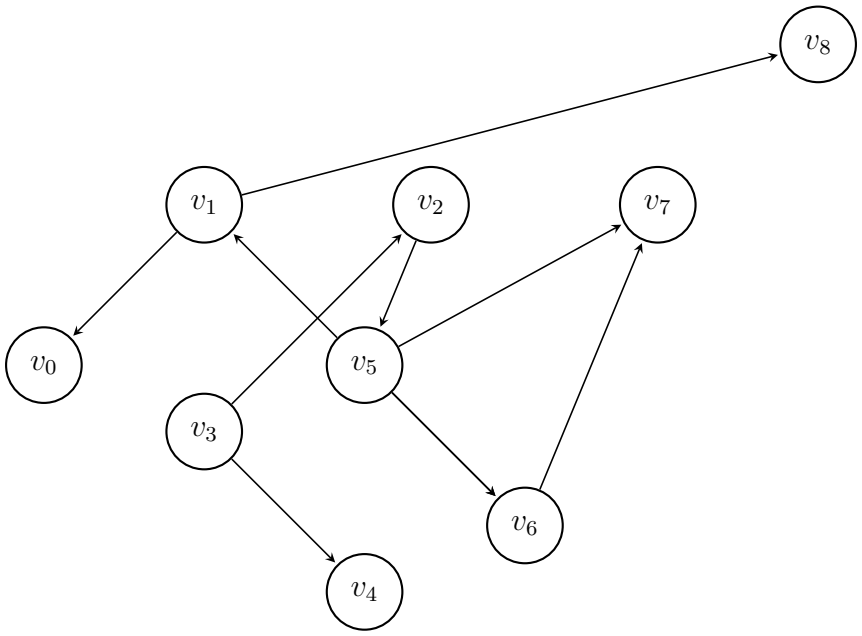
Instructions

All parts are due on April 8, 2021 at 10PM. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct. Solutions should be submitted on Gradescope, and any code should be submitted for automated checking on `https://alg.mit.edu`.

Please take note of the new collaboration policy as described in `https://canvas.mit.edu/courses/7477/discussion_topics/76837`. Unless explicitly stated otherwise, to get full credit you must prove the runtime and correctness of any algorithm you give.

Problem 6-1. [15 points] **Warm-up problems**

(1) [5 points] **Topological Sort** Use DFS to find a topological sort for the following graph. We explore nodes in increasing index order. Starting at the node v_3, for each iteration of DFS, indicate the `parent` and `order` lists maintained, and return the final topological sort that DFS finds.



Solution: Based off how an iteration of DFS is interperated, there are two possible solutions. The first is the one where an iteration is defined as when we call DFS on a vertex that hasn't been explored yet.

1. `order` = [], `parent` = [None, None, None, v_3, None, None, None, None, None]
2. `order` = [], `parent` = [None, None, v_3, v_3, None, None, None, None, None]
3. `order` = [], `parent` = [None, None, v_3, v_3, None, v_2, None, None, None]
4. `order` = [], `parent` = [None, v_5, v_3, v_3, None, v_2, None, None, None]
5. `order` = [v_0], `parent` = [v_1, v_5, v_3, v_3, None, v_2, None, None, None]
6. `order` = [v_0, v_8, v_1], `parent` = [v_1, v_5, v_3, v_3, None, v_2, None, None, v_1]
7. `order` = [v_0, v_8, v_1] `parent` = [v_1, v_5, v_3, v_3, None, v_2, v_5, None, v_1]
8. `order` = [v_0, v_8, v_1, v_7, v_6, v_5, v_2], `parent` = [v_1, v_5, v_3, v_3, None, v_2, v_5, v_6, v_1]
9. `order` = [v_0, v_8, v_1, v_7, v_6, v_5, v_2, v_4, v_3], `parent` = [v_1, v_5, v_3, v_3, v_3, v_2, v_5, v_5, v_1]

We also accept solutions when an iteration is counted as when we update the parent list.

1. `order` = [], `parent` = [None, None, None, v_3, None, None, None, None, None]

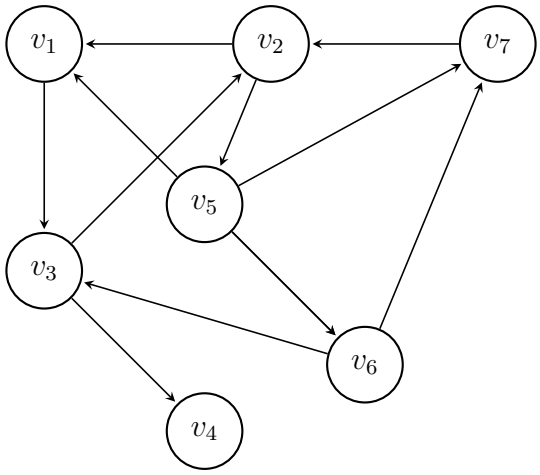
2. `order` = [], `parent` = [None, None, v_3, v_3, None, None, None, None, None]
3. `order` = [], `parent` = [None, None, v_3, v_3, None, v_2, None, None, None]
4. `order` = [], `parent` = [None, v_5, v_3, v_3, None, v_2, None, None, None]
5. `order` = [], `parent` = [v_1, v_5, v_3, v_3, None, v_2, None, None, None]
6. `order` = [v_0], `parent` = [v_1, v_5, v_3, v_3, None, v_2, None, None, v_1]
7. `order` = [v_0, v_8, v_0], `parent` = [v_1, v_5, v_3, v_3, None, v_2, v_5, None, v_1]
8. `order` = [v_0, v_8, v_0], `parent` = [v_1, v_5, v_3, v_3, None, v_2, v_5, v_6, v_1]
9. `order` = [v_0, v_8, v_0, v_7, v_6, v_5, v_2], `parent` = [v_1, v_5, v_3, v_3, v_3, v_2, v_5, v_6, v_1]
10. Final `order` = [v_0, v_8, v_0, v_7, v_6, v_5, v_2, v_4, v_3]

One possible topological sort is the reverse of the `order` list, or [v_3, v_4, v_2, v_5, v_6, v_7, v_1, v_8, v_0]

Rubric:

1. -0.5 for each wrong `order` or `parent` list, but no double jeopardy.
2. -0.5 for wrong topological sort based off of final `order`

(2) [10 points] **Edge Classification** Apply DFS to the following graph starting from vertex v_1. Explore nodes in increasing index order. For each node, state its `discover` and `finish` times. Here, we'll use the convention that once every node's neighbors are finished, the node is finished at the same iteration. Classify each edge as either `tree`, `backward`, `forward` or `cross`. Explain your rationale based on the `discover` and `finish` times of the nodes.



Solution: Note that these iterations start at 0, but if students start at 1, just add 1 to every time.

1. v_1: `discover` time: 0; `finish` time: 6
2. v_2: `discover` time: 2; `finish` time: 5
3. v_3: `discover` time: 1; `finish` time: 6

4. v_4: `discover` time: 6; `finish` time: 6
5. v_5: `discover` time: 3; `finish` time: 5
6. v_6: `discover` time: 4; `finish` time: 5
7. v_7: `discover` time: 5; `finish` time: 5

1. (v_1, v_3) is a tree edge, because v_3 was discovered after v_1 and they both haven't finished yet.
2. (v_3, v_2) is a tree edge, because v_2 was discovered after v_3 and they both haven't finished yet.
3. (v_2, v_1) is a back edge, because v_1 has already been discovered but isn't finished yet.
4. (v_2, v_5) is a tree edge, since v_5 hasn't been discovered yet.
5. (v_5, v_1) is a back edge, because v_1 has been discovered but hasn't finished yet
6. (v_5, v_6) is a tree edge, because v_6 hasn't been discovered yet.
7. (v_6, v_3) is a back edge, because v_3 has already been discovered but hasn't finished yet.
8. (v_6, v_7) is a tree edge, because v_7 hasn't been discovered yet.
9. (v_7, v_2) is a back edge, because v_2 has already been discovered and hasn't finished yet
10. (v_5, v_7) is a forward edge, since v_7 was discovered after v_5 and finished before v_5
11. (v_3, v_4) is a tree edge, since v_4 hadn't been discovered yet.

Problem 6-2. [20 points] **The Purples** The Purples are planning a fund-raiser. The up-and-coming party has a list of n "luminaries" that they want to invite. Each luminary is bringing a significant other to the party. The problem is that said luminaries, or their SO's, are of the jealous sort. Some luminaries, for reasons unknown, are jealous of some of the others. To make matters worse, some of the SO's are also jealous of some of the other SO's. Luckily, the planning committee knows these "issues" ahead of time and they have a list of the luminary problematic relationships as well as a list of the SO's. We write $s_i \to s_j$ to mean that SO number i is jealous of SO number j and $l_i \to l_j$ to mean luminary i is jealous of luminary j.

Why do the Purples need your help? Well, they are planning an arm-wrestling tournament where all luminaries will each arm-wrestle all the SO's. Each match can be written as (l_i, s_j) for some i, j, to mean that l_i is playing s_j. They ask you to come up with an ordering of the matches so that:

(i) If $l_i \to l_j$ and $s_i \to s_j$ for some i, j, then (l_i, s_i) is scheduled before (l_j, s_j)

(ii) If $l_i \to l_j$, then for every luminary s, (l_i, s) is scheduled before (l_j, s).

(iii) If $s_i \to s_j$, then for every luminary l, (l, s_i) is scheduled before (l, s_j).

Note that luminaries are only jealous of other luminaries and SO's are only jealous of other SO's. You can assume that all matches are played in a sequential order (so that no matches are played at the same time).

(1) [15 points] Help the Purples design an algorithm to create a suitable match schedule, or let them know that no such schedule is possible. Make sure to analyze its performance and correctness.

Hint: Try to show that a suitable schedule exists if and only if there is no cycle of jealousies.

Solution: Construct a graph such that we have a node for each luminary and each SO. We will create an edge from luminary i to luminary j iff i is jealous of j (and will do the same to determine edges between SOs). We have two cases, one where there is a cycle of jealousies in the graph and one where there is not, and we analyze them separately.

1. If the jealousies are cycle-free, there is an ordering of the luminaries $l_1^*, l_2^*, \ldots, l_n^*$ and of the SOs $s_1^*, s_2^*, \ldots, s_n^*$ such that luminary l_i is not jealous of luminary l_j^* for any $j > i$. This is because the graph is acyclic, so there is a topological sort of each graph (and we know how to find it via DFS), which produces such an ordering.
   Find this ordering and to make the schedule, have the first n matches be l_1^* against each of the s_i^*, in the order of this topological sort. Then do the same for l_2^*, and so on. Return this list of matches.
2. If the graph contains a cycle, return that the task is impossible.

Correctness: We will analyze correctness in each case individually.

1. We see that condition (ii) is met for this schedule because l_i^* plays all its games before any l_j^* for $j > i$ and the sort is such that l_i^* is only potentially jealous of the luminaries that come after it in the sort. Additionally, to meet condition (iii), we see that for each s_i^*, we get that for a given l_k^*, it plays l_k^* before any s_j^* with $j > i$, which are the only SO's it might be jealous of.
   We also note that conditions (ii) and (iii) together imply condition (i). Assume $l_i \to l_j$ and $s_i \to s_j$. Then condition (ii) implies (l_i, s_j) must occur before (l_j, s_j). Condition (iii) implies that (l_i, s_i) must occur before (l_i, s_j). Thus, if we meet the last two conditions, then we must have (l_i, s_i) before (l_j, s_j). This means that conditions (ii) and (iii) imply condition (i) and thus our ordering satisfies condition (i).
   Thus, this is a valid schedule that we returned.
2. In the case the graph has a cycle, we want to prove that there is no valid scheduling. We assume WLOG that there is a cycle among the luminaries. Let $(x, y) > (a, b)$ mean that match (x, y) *must* occur before (a, b). Denote the items in this this cycle by $l_0^*, l_2^*, \ldots, l_{k-1}^*$ such that $l_i^* \to l_{i+1 \mod k}^*$. Take a given SO, s. By condition (ii), we know that the game for $(l_i^*, s) > (l_{i+1}^*, s)$ for each l_i^*. Transitively, this implies that $(l_0^*, s) > (l_{k-1}^*, s)$. However, since $l_{k-1}^* \to l_0^*$, we also know $(l_{k-1}^*, s) > (l_0^*, s)$, a contradiction. Thus, we are correct that there is no valid game schedule in this case.

Runtime: Constructing the graph takes linear time in n and the number of jealousies. Getting a tolopogical sort also takes linear time in that graph size, and then getting an ordering from that topological sort requires iterating over the second list once for each item in the first list, for $O(n^2)$ time to return that list. (Side note: since the output must have size n^2, this is an optimal runtime.)

Rubric:

+4 Runs topo sort on graph of luminaries on that of SOs

+4 Gets a valid schedule from the two topo sorts

+5 Valid correctness argument for correct solution

+2 Valid runtime argument for correct solution

Some students have solutions that are $O(n^4)$ time algorithms in which they create a graph on all matches and then draw edges from one match to another if the first match must come before the second one. This solution can receive up to 10 points [+5 for correct algorithm, +3 for correctness argument, +2 for *correct* runtime $O(n^4)$]

(2) [5 points] Say we change the requirements for (i) to the following:

(i*) If $l_i \to l_j$ *or* $s_i \to s_j$ for some i, j, then (l_i, s_i) is scheduled before (l_j, s_j)

and we additionally drop requirements (ii) and (iii). Show that the hint from the previous part does not hold. That is, either give an example where there is a cycle among either the luminaries or the SOs (or both), and yet one can find a suitable schedule that

obeys the new requirement (i*) or give an example where there is no cycle but there does not exist a suitable schedule.

Solution: Say $n = 2$ and $l_1 \to l_2$ and $s_2 \to s_1$ (and these are the only edges so the graph has no cycles). The new condition and the fact that $l_1 \to l_2$ tells us that we must have $(l_1, s_1) > (l_2, s_2)$. The new condition and the fact that $s_2 \to s_1$ implies that $(l_2, s_2) > (l_1, s_1)$, a contradiction.

Rubric:

+5 for correct example

Problem 6-3. [30 points] **Do you hear the people sing?** Arius and Menjolras are living in Paris and they've decided there is "One day more" until revolution. Ahead of time, they get a map of the city, which consists of streets connecting plazas throughout the city (a single street connects a single pair of plazas and a plaza may have an arbitrary number of streets leaving it).

Thematic music (optional)

Arius and Menjolras have just enough people and furniture to block a single street. Their goal is to be as disruptive as possible. They want to make sure that there exist at least two plazas, x and y such that people cannot go from x to y without passing through their barricades.

We can view this problem as a graph with nodes that correspond to plazas and edges between two nodes iff there is a street between the corresponding plazas. What they are looking for is a *bridge*, or an edge that can be removed from a graph to increase the number of connected components. In this problem, we will walk you through how to find bridges in a graph (as promised in Lecture 1).

We will work off of the articulation points algorithm given in the recitation notes. Let $v.in$ be the discovery time of vertex v. For a given run of DFS, we will calculate $v.low$ for each vertex v in the graph. We define $v.low$ such that $v.low := \min\{v.in, \min_x\{x.in\}\}$ where the inner minimum is taken over x such that wx is a back edge and w is a descendant of v. We know from lecture and recitation that these values can be calculated in $O(|E|)$ time. You can assume throughout this question that the graph is *undirected and connected*.

(1) [5 points] Prove that if uv is a bridge, then all paths from u to v must go through edge uv.

Solution: If uv is a bridge, then we know that removing uv disconnects the graph. If u still has a path to v after this happens (ie there was a path in the original graph that didn't use uv), then any paths in the original graph that used uv can now use this path to replace that edge and thus it is not disconnected.

Rubric:

+5 for correct proof

(2) [5 points] Prove that if uv is a bridge, then uv must be a tree edge in the DFS tree.

Solution: If an edge is a bridge, then it must appear in any spanning tree of the graph (including a DFS tree) or else we could use the path from u to v in the tree itself to find a path from u to v that does not use edge uv and thus we do not have a bridge.

Rubric:

+5 for correct proof

(3) [5 points] From the previous claim, we know that if uv is a bridge, it must be in the DFS tree. Thus, we can assume that v is a child of u in this tree. Prove that $u.in < v.low$.

Solution: Say $u.in \geq v.low$, then we know that v has a descendant w that has a back edge to some ancestor of v whose discovery time was either $u.in$ or earlier. We call the node that w has this back edge to x. Since w is a descendant of v and thus of u,

any of its ancestors with discovery time $u.in$ or earlier are either u or an ancestor of u, so there is a path through the tree from x to u that does not use uv. Since w is a descendant of v, we can say the same about a path from v to w. Thus, if we combine these two paths with the edge wx, we have found a path from u to v that does not use uv and thus by the first claim, uv is not a bridge.

Rubric:

+5 for correct proof

Actually, as it turns out, the condition on edge uv in Part (3) is not only necessary for uv to be a bridge, but also sufficient. So, uv is a bridge *iff* uv is a tree edge (where v is u's child) and $u.in < v.low$. As a challenge, You may want to prove this on your own (we will provide a proof when we release solutions).

Solution: Proof of the claim. (You did not have to give this proof.)

Say uv is not a bridge but fits these properties. Then there must exist a path from u to v that does not go through uv (or else they would not be in the same connected component when this edge is removed). This path composed with uv forms a cycle in the graph, and since there is a cycle, there is at least one edge in the cycle that does not exist in the tree, and since it's undirected that edge must be a back edge. Let P be such a simple path and let wx be the first back edge in P. Since this is the first back edge in P, we know that there is a path from v to w that uses only tree edges, making w a descendant of v. Then we have that by definition of $v.low$, $v.low \leq x.in$. Additionally, since this is a back edge, we know that x is an ancestor of w, which means that it must be either: equal to u, an ancestor of u, equal to v, or a descendant of v. In the first two cases, $x.in \leq u.in$, which implies that $v.low \leq x.in \leq u.in$, which contradicts $u.in < v.low$. In the second two cases, we know that the tree path from v to w is through x, which means that P is not simple after all.

Thus, each case is impossible and we have reached a contradiction.

(4) [5 points] Give a linear time algorithm to find all bridges in the graph. Don't forget to analyze runtime and correctness. (You may use the above claims whether you proved them or not.)

Solution: Find $v.low, v.in$ for all v as discussed in recitation. Iterate over all edges uv in the tree and check if $u.in < v.low$, counting it as a bridge if it is.

Correctness: From the previous parts, we have that uv is a bridge iff it is a DFS tree edge with $u.in < v.low$. These are exactly the edges we find.

Runtime: From recitation, we can find the $v.low, v.in$ values in $O(|V| + |E|)$ time. Then we iterate over edges and compare these values in constant time per edge, for overall $O(|V| + |E|)$ (linear) time.

Rubric:

+1 for correct algorithm

+2 for correct correctness analysis

+2 for correct runtime analysis

(5) [10 points] Arius and Menjolras learn that Jinspector Avert plans to lead his troops from a particular plaza s to another plaza t, and they want to place their barricade such that they cannot make this journey without going through the barricade. In linear time (in the number of streets/plazas), help them find a list of all streets e such that blocking e (and only e) will achieve this. Remember to analyze runtime and correctness. *Hint:* For your correctness argument, you may find useful the fact proved in Part (1).

Solution: First run the algorithm from the previous part. This tells us all bridges. Then run BFS or DFS to find a simple path from s to t. Check if any of the edges on this path are bridges (using the information we got in the run of the previous algorithm). Return all such edges.

Correctness: If removing an edge removes all paths from s to t, then by definition it appears on all paths from s to t. Additionally, such an edge must be a bridge because removing it puts s and t in different connected components.

If an edge uv in a simple path from s to t is a bridge, then we know that u and v are in different connected components if that edge is removed (or else any pair of nodes that previously had a path between them still have a path between them). Since s still has a path to u once this edge is removed and v still has a path to t (as the path is simple so uv appears only once on it), we conclude that they are in different connected components if uv is removed and thus each edge we find is a valid edge for our criteria.

Thus, the set of edges that they can use is exactly the edges that appear on some simple path from s to t and form bridges in the graph.

Runtime: The previous algorithm runs in linear time. If instead of returning a list, we just marked the bridges in that part, as we ran DFS/BFS, we could mark any edges in the DFS/BFS tree that are bridges, and thus when we get a path between s and t we will have all bridges on that path marked and we can iterate over it in linear time to figure out the list of bridges in that path.

Rubric:

+3 for correct algorithm

+6 for correct correctness analysis

+1 for correct runtime analysis

Problem 6-4. [35 points] **pip install** You want to run a python file you found online, but it uses a lot of bizarre imported libraries, some of which depend on other libraries. You'll need a correct order in which to pip install the libraries so that a library is only installed if every library it depends on is already installed. You have the list of L libraries that you want to install (labeled by the numbers 0 through $L-1$) and the adjacency list, `Dependencies`, showing which libraries each library is dependent upon. There are D dependencies.

(1) [8 points] Describe an $O(L+D)$ time algorithm to return a correct order in which to install the libraries, so they can be installed one at a time, or return `None` if no such order exists.

Example: With dependency list `[[1,2], [], [1], [1]]`, one possible order is `[1, 2, 0, 3]`.

Solution: Construct a directed graph with a vertex for each library and an edge from library a to b if b depends on a. Check for cycles using DFS in $O(L+D)$, and if there's a cycle return `None`. Otherwise, use topological sort and return that order.

Correctness: Cycle detection and topological sort proven in class.

Runtime: From class topological sort is known to be $O(L+D)$.

(2) [12 points] You now have access to a multicore machine that can install an unlimited amount of libraries at the same time. That is to say, in one install session you can install multiple libraries in parallel, but each library must still already have all of its dependencies installed. Describe an $O(L+D)$ time algorithm to determine an order to install libraries that uses the least number of install sessions. You should return a list of sets indicating which libraries to install in each session.

Example: With dependency list `[[1,2], [], [1], [1]]`, the ordering is `[{1}, {2, 3}, {0}]`.

Solution: As in part **a)**, we check for cycles and obtain an order using topo sort. We set up a DAA where each libraries index in the array holds a session number. Iterate through the topo sort list and set the session number of library a to be one plus the max of the session numbers of each incoming neighbor (obtainable from the dependency list).

Correctness: Cycle detection and topological sort proven in class. Inductively, if the session numbers of each library are correct up to library of index i, then we have the correct session numbers for all the libraries that $L[i+1]$ depends upon. The earliest session that library $i+1$ can be in is one after the latest session that any of its dependencies is in; therefore $1+\max(\text{dependencies' sessions})$ is the earliest session possible, limiting the number of install sessions.

Runtime: Topological sort is known to be $O(L + D)$. Iterating through the topo sort list takes L time, and we iterate through each node's dependency list once to get the max session number, for a total of D operations. Overall this is still $O(L + D)$

(3) [15 points] Code the function `determine_install_order` described in the previous part (but now returning number of sessions rather than an ordering of libraries) using the outline provided on `alg.mit.edu`. The portion of the outline given below shows all functionality that you must implement (denoted by YOUR CODE HERE).

```
def determine_install_order(dependencies):
    """
    Args:
        dependencies (list): adjacency list showing which libraries
                                            each library is dependent on

    Output:
      int: The minimum number of sessions needed to install all libraries
           or None if there is no valid install order
    """

    ##################
    # YOUR CODE HERE #
    ##################

    return None
```