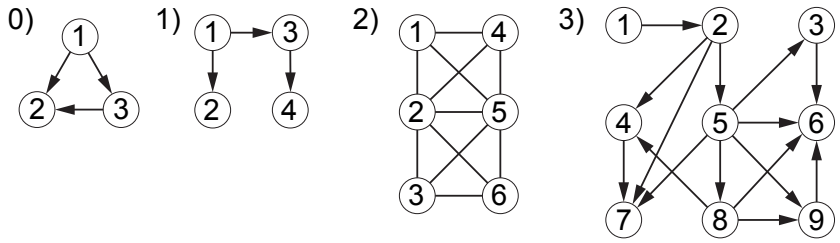
Problem Set 6

All parts are due on November 2, 2017 at 11:59PM. Please write your solutions in the LaTeX and Python templates provided. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct. Solutions should be submitted on the course website, and any code should be submitted for automated checking on alg.csail.mit.edu .

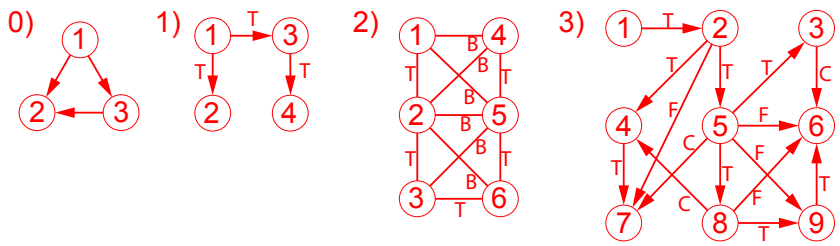
Problem 6-1. [25 points] DFS Practice

A depth first search (DFS) of a graph explores vertices along an unsearched path until reaching a vertex that has already been explored, at which point the search backtracks and continues along the next unsearched path. Which unsearched path is *next* is arbitrary and different choices can lead to different DFS trees. For graphs embedded in the plane, it is common to fix the order to search outgoing edges, for example in counterclockwise order from the edge from which you entered the node. We will refer to DFS using this ordering as a **CCW DFS**. Answer the following questions by performing depth first search on the following graphs, starting your search along the edge from vertex 1 to vertex 2, using CCW DFS.



(a) [12 points] During DFS, *tree* edges are edges traversed to a previously unexplored node and together form a DFS tree. Edges traversed to previously explored nodes have different labels: a *back* edge extends from a node to one of its ancestors in the DFS tree; a *forward* edge extends from a node to one of its descendants; and a *cross* edge is any other edge. Label edges in graphs G_1, G_2, and G_3 as either tree (T), back (B), forward (F), or cross (C), using a CCW DFS. For example, CCW DFS on G_0 has a single cross edge, $(3, 2)$, with all other edges being tree edges.

Solution:



Rubric:

- 2 points for correct labeling of G_1
- 4 points for correct labeling of G_2
- 6 points for correct labeling of G_3
- -1 for each incorrect labeling

(b) [6 points] For graphs G_1, G_2, and G_3, list vertices in a topological sort based on the finishing times of a CCW DFS, or argue that a topological sort does not exist. The topological sort of G_0 based on CCW DFS is $[1, 3, 2]$.

Solution:

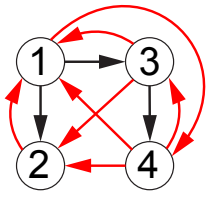
- G_1: $[1, 3, 4, 2]$
- G_2: Undirected graph contains back edges, thus cycles: no topological sort.
- G_3: $[1, 2, 5, 3, 8, 9, 6, 4, 7]$

Rubric:

- 2 points for each graph order or argument
- Partial credit may be awarded

(c) [7 points] Graph G_1 is actually the CCW DFS tree of a larger **simple** directed graph G. In fact, among all graphs admitting a CCW DFS tree equivalent to G_1, G is the graph with the most edges. Provide an argument for any edge (i, j) that does not exist in G, and draw G with vertices labeled.

Solution: It is easier to characterize the edges that are not in G. G cannot contain edges $(2, 3)$ or $(2, 4)$; if it did, those edges would be tree edges as they would be traversed from 2 and first discover 3 and 4 in a CCW DFS. Edge $(1, 4)$ is tricky. Drawing $(1, 4)$ inside the square would force 4 to be discovered directly from 1 in CCW DFS. However, drawing $(1, 4)$ outside the square is consistent with a CCW DFS as 3 will be explored before 4.



Rubric:

- 3 points for drawing
- 2 points for each missing edge correctly argued
- Partial credit may be awarded

Problem 6-2. [30 points] **Spy Networks**

Harriet works for the CIA and is monitoring terrorists with the help of an inside informant. Help Harriet bring down the terrorists. Time is of the essence, so you will be judged based on the speed of your solutions.

(a) [10 points] **Who's the Boss:** Harriet's informant has given her details on various terrorists and who they take orders from. A terrorist's *influence* is the set of terrorists he can give orders to, directly or indirectly; a terrorist will never take orders from someone within their influence, but may take orders directly from multiple people. A *terrorist cell leader* is a terrorist who takes orders from no one; a leader's influence is called a *terrorist cell*. Based on her informant's information, Harriet needs to generate a list of terrorist cells, and for each cell, construct a list of the cell's members so that a terrorist within the influence of another appears lower down on the list. Describe an algorithm to help Harriet generate these lists as quickly as you can.

Solution: Construct a directed graph G with a vertex associated with each terrorist, and an edge from terrorist a to terrorist b if b takes orders from a. Because a terrorist does not take orders from someone within their influence, G is acyclic. Identify cell leaders by looping over vertices, and for each check whether any incident edge terminates at the vertex. This identification process loops over vertices and evaluates each edge at most twice, so leaders can be identified using this algorithm in $O(|V| + |E|)$ time. Then for each cell leader, run BFS to find the subgraph of vertices within their influence. Then, topologically sorting vertices within the same cell produces an ordering for which a terrorist within the influence of another occurs later in the sort. The influence of each cell may contain $O(V)$ vertices and $O(E)$ edges, and there are at most $O(V)$ cells, so these lists can be generated in $O(|V|(|V| + |E|))$ by the above procedure.

Rubric:

- 10 points for a correct algorithm
- Partial credit may be awarded

(b) [10 points] **Informant Rescue:** Harriet's informant has been discovered by a terrorist cell. He's been captured and is being held, chained in an elaborate system of underground tunnels. Fortunately, the guards do not patrol the tunnels at night, so Harriet has been flown in to rescue him. Unfortunately, Harriet has no information about the tunnels or where in the tunnels her informant is being held, except that she will be able to reach him from the tunnel's only entrance. Inside the tunnels, there is a lantern hanging at each intersection to light the way. Time is short, but Harriet thinks there will be enough time to scratch a single arrow on the ground at each intersection. Describe an algorithm to help Harriet rescue her informant as quickly as you can.

Solution: Explore the tunnels using depth first search. When entering an intersection containing no markings, draw an arrow pointing to the tunnel from which you en-

tered. Then search tunnels in a consistent ordering e.g. counterclockwise with respect to the tunnel last searched. Harriet must also take care to remember if she is actively searching or backtracking the tunnel she is currently in. If she enters an already arrowed intersection and she is actively searching, she backtracks along the same tunnel she came in. If she enters an already arrowed intersection and she is backtracking, she should explore the next counterclockwise tunnel. If an arrow points to the next counterclockwise tunnel, then backtrack along that tunnel. This is standard DFS from a local perspective, so takes time linear in the number of tunnels plus the number of intersections.

Rubric:

- 10 points for a correct algorithm
- Partial credit may be awarded

(c) [10 points] **Targeted Capture:** Now that the informant is safe, the CIA is ready to take action. Harriet needs to recommend to her superiors a list of candidate terrorists for capture to help break up the cells. Cell leaders are very difficult to capture, so she has been asked to prioritize non-leader terrorists whose individual capture would decrease the size of a cell leader's influence by more than one. Assume that only one terrorist from the list will be pursued at this time. Describe an algorithm to help Harriet identify a list of candidate targets for capture as quickly as you can.

Solution: Construct the same graph G as in part (a), with each vertex labeled with the cells in which the terrorist associated with the vertex appears, and each cell leader containing the number of terrorists within their influence. The following brute force algorithm can identify candidate terrorists in $O(|V|^2(|V|+|E|))$ time. For each terroist t and for every cell leader c, remove t from G and run BFS from c on the new graph to count the number of nodes reachable from c in the modified graph. If c's influence is exactly one less than c's influence in G, then t is not a candidate; otherwise c is a candidate. This brute force algorithm can be slow, but it gets the job done.

A more sophisticated algorithm can identify candidates in $O(|V|(|V| + |E|))$ time. We notice that a non-leader t is a candidate (i.e. individual capture would decrease the size of the influence of leader c by more than one) if and only if t has an outgoing edge to a vertex v, whose only incoming edge from a vertex within c's influence is vertex t. The reverse implication is trivial: removing t from the graph would disconnect v from c because t was the only vertex within c's influence with an edge to v. Alternatively, suppose t's capture decreases the size of the influence of leader c by more than one. Let V be the nonempty set of t's influence that would be disconnected from c upon the removal of t, and let v be the first $v \in V$ occuring in a topological sort of G. We argue that v's only parent contained in c's influence is t. Suppose for contradiction that v has another parent p within c's influence that is not t. Vertex p must be in V or else v would not be disconnected by removal of t. But then p would come before v in a topological order, and we have reached a contradiction.

Then for each vertex v in cell leader c's influence, it suffices to check the number of incoming edges to v from vertices within c. If the number is exactly one, mark v's only parent as a candidate for capture. This algorithm checks each vertex and every edge, once per cell, so the algorithm runs in $O(|V|(|V| + |E|))$ time.

Rubric:

- 10 points for a correct algorithm
- Partial credit may be awarded
- Max 6 points for an algorithm asymptotically slower than $O(|V|(|V| + |E|))$

Problem 6-3. [45 points] **Linear Equations:** A linear equation on n variables $\{x_1, \ldots, x_n\}$ specified by sequence of coefficients $C_i = (d_i, c_{i1}, \ldots, c_{in})$ has the following form:

$$0 = d_i + \sum_{j=1}^{n} c_{ij} x_j$$

Given a set of linear equations, an assignment of the variables to real numbers is said to *solve* the set of equations if the assignment applied to each equation makes the equation true. You may assume that sets of linear equations encountered in this problem will have a single unique solution, with the number of equations equal to the number of variables.

(a) [9 points] An algorithm called Gaussian Elimination can solve a set of linear equations, but requires $\Omega(n^3)$ time in the worst case. Some sets of linear equations can be solved faster. Call a set of linear equations *delegated* if $c_{ii} = -1$ for all $i \in [1, n]$. Equation C_i from a delegated set of linear equations can be rewritten as:

$$x_i = d_i + \sum_{\substack{j=1 \\ i \neq j}}^{n} c_{ij} x_j$$

We say that x_i *depends on* x_j in a delegated set if $c_{ij} \neq 0$. A delegated set is *special* if there exists a permutation $\pi : [1, n] \to [1, n]$ such that x_i depends on x_j only when $\pi(i) < \pi(j)$, for all $i, j \in [1, n]$. Describe a simple directed graph on n vertices based on a delegated set of linear equations that will be acyclic if and only if the set of equations is special.

Solution: Construct a graph G on n vertices with vertex i associated with linear equation C_i. For each linear equation C_i, add directed edge (v_i, v_j) to G if $c_{ij} \neq 0$ and $i \neq j$. If the equations are special, then there exists a permutation π such that G contains no directed edge $(i, j) \in [1, n]^2$ where $\pi(j) < \pi(i)$. This implies that $\pi(i)$ is a topological sort on the vertices; G then has no back edges under this ordering, so G is acyclic. Alternatively, if G is acyclic, we can compute a topological sort on the vertices π'. Edge (i, j) was added to G only when $c_{ij} \neq 0$ and $i \neq j$, so $c_{ij} = 0$ when $\pi'(i) < \pi'(j)$ because no back edges exist in G with respect to topological sort $\pi'(i)$, completing the proof.

Rubric:

- 3 points for description of the graph
- 3 points for argument that acyclic implies special
- 3 points for argument that special implies acyclic
- Partial credit may be awarded

(b) [8 points] Describe an algorithm to determine whether a delegated set of linear equations is special in $O(n^2)$ time. Note that the input to the problem consists of $\Omega(n^2)$ coefficients, so $O(n^2)$ running time is linear in the size of the input.

Solution: Construct the graph G from part (b) based on the set of linear equations, which will be acyclic if and only if the equations are special. We look for directed cycles by running DFS. If a back edge is encountered during the search, the graph has a cycle, and thus is not special. Alternatively, DFS completes certifying that G is acyclic, and thus special. Graph G has n nodes and at most $O(n^2)$ edges so may be constructed and searched via DFS in $O(n^2)$ time as desired.

Rubric:

- 8 points for a correct algorithm
- Partial credit may be awarded

(c) [8 points] Describe an algorithm that returns a solution to a special set of linear equations in $O(n^2)$ time. You may assume that the size of coefficient values in the problem are small, i.e. have $O(1)$ bit complexity so that any single arithmetic operation takes $O(1)$ time.

Solution: Construct the graph G from part (b) based on the special set of linear equations. By (b), G is acyclic, so find a topological sort order π on the vertices using DFS. Solve the equations in reverse topological sort order. Because each equation C_i depends only on variables further down in the topological sort, we can compute variable x_i directly from previously computed variables. The topological sort takes $O(n^2)$ time, while each of the n variables may require $O(n)$ operations to compute, leading to $O(n^2)$ time.

Rubric:

- 8 points for a correct algorithm
- Partial credit may be awarded

(d) [20 points] Implement function `solve_special` that takes as input a list of lists representing a set of linear equations and returns a list representing a solving assignment for the variables from x_1 to x_n if the equations are delegated and special, and `None` otherwise. Please submit code for this problem in a single file to alg.csail.mit.edu. ALG will only import your implementation of `solve_special`.

```
def solve_special(C):
    """
    Given a set of linear equations C, returns a solving assignment to variables
    if the set of equations is delegated and special, and None otherwise
    Input:
        C is a list of n lists, each with length n + 1
        C = [C_0, ... C_n]
        C_i = [d_i, c_i1, c_i2, ... c_in]
        C_i represents equation:  0 = d_i + \sum_j c_ij * x_j
    Output:
        x = [x_1, ... x_n]
        solving assignment of variables
    """
    ############################
    #  Implement me! Part (d)  #
    ############################
    x = None
    return x
```

Solution: See below. This problem is graded by ALG.

```
def solve_special(C):
    def dfs_topo_sort(A, i = None, order = None, parent = None):
        'Runs DFS and topological sort on adjacency matrix'
        n = len(A)
        if i is None:
            order = [-2 for i in range(n)]
            parent = [None for i in range(n)]
            for i in range(n):
                if order[i] == -2:
                    dfs_topo_sort(A, i, order, parent)
        else:
            order[i] = -1
            for j in range(n):
                if (A[i][j] != 0) and (order[j] == -2):
                    parent[j] = i
                    dfs_topo_sort(A, j, order, parent)
            order[i] = max(order) + 1
        return order, parent

    def cyclic(A, order, parent):
        'Find back edges from DFS tree'
        n = len(A)
        for i in range(n):
            for j in range(n):
                if (A[i][j] != 0) and (order[i] < order[j]):
                    # edge is either back or cross
                    node = parent[i]
                    while node is not None:
                        if node is j:
                            # ancestor! so found a back edge
                            return True
                        node = parent[node]
        return False

    n = len(C)
    A = [c[1:] for c in C]
    for i in range(n):
        if A[i][i] != -1:
            return None
        A[i][i] = 0
    order, parent = dfs_topo_sort(A)
    if cyclic(A, order, parent):
```

```
    return None
x = [C[i][0] for i in range(n)]
for o in range(n):
    i = order.index(o)
    for j in range(n):
        if A[i][j] != 0:
            assert(order[j] < order[i])
            x[i] += A[i][j] * x[j]
return x
```