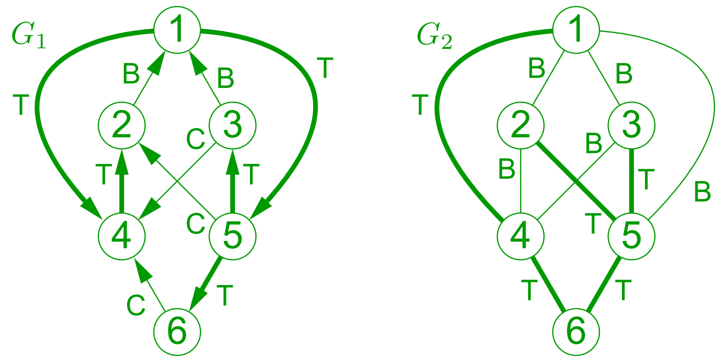
Problem Set 7

All parts are due on April 12, 2018 at 11PM. 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 `py.mit.edu/6.006`.

Problem 7-1. [30 points] **Graph Practice II**

(a) [20 points] For graphs drawn and embedded in the plane, it is common to fix an order to search outgoing edges: for example, counterclockwise order from the edge from which you entered a vertex. We will refer to a depth-first search using such an ordering as a CCW DFS. For direct graph G_1 and undirected graph G_2 **as drawn** below, perform a depth-first search from vertex 1 using CCW DFS, starting the search along the edge from vertex 1 to vertex 4. Then for each graph, (i) label each edge as either a **tree**, **back**, **forward**, or **cross** edge, and (ii) list the graph vertices in a topological sort based on the finishing times of your depth-first search, or argue that a topological sort does not exist.

Solution:



Neither graph admits a topological sort as each contains an unorderable cycle.

Rubric:

- 10 points for each graph:
- $\max(0, 8 - k)$ points for edge labeling: k number of incorrect labels.
- 2 points for argument that topological sort does not exist.

(b) [5 points] Prove that for any connected undirected graph G on n vertices and any positive integer $k < n$, either G has some path of length at least k, or G has fewer than $k \cdot n$ edges.

Solution: Note that this problem statement is ambiguous as to whether the desired path must be **simple**. If non-simple paths are allowed, the following argument holds. If there is a cycle in the graph, then there is a path of length n, and if there is not, the undirected graph is a tree, and has at most $n - 1$ edges.

If non-simple paths are disallowed, the following argument holds. Run depth-first search from some node in the graph and consider the longest path in the DFS tree. If it has length at least k, we are done. Otherwise, since G is undirected, there are no cross edges with respect to the DFS tree, so each edge of the graph connects an ancestor and a descendant in the DFS tree. This bounds the number of edges in the graph by counting the total number of ancestors of each descendant; but if the longest path is shorter than k, each descendant has at most $k - 1$ ancestors. So there can be at most $(k - 1)n$ edges.

Rubric:

- 5 points for a correct proof (using either assumption).
- Partial credit may be awarded.

(c) [5 points] An *articulation point* of a connected undirected graph is any vertex that, if removed from the graph would make the graph no longer connected. For example, if only one path of vertices connects vertices u and v in a graph, then every vertex in the path between u and v would be an articulation node. Given an undirected graph, describe a $O(|V||E|)$ time algorithm to output a list of articulation points in a graph. **Challenge:** listing all articulation points of an undirected graph can be accomplished in $O(|V| + |E|)$ time.

Solution: For each vertex, check if removing it would disconnect the graph. The check can be done by running either BFS or DFS on any remaining vertex of the graph with the selected vertex removed. The running time is $O(|V|(|V| + |E|))$. Since the graph is connected, $|E| = \Omega(|V|)$, so the running time simplifies to $O(|V||E|)$.

The challenge finds articulation points in linear time. Run depth-first search from an arbitrary root vertex r. During the DFS, label each graph edge as a directed tree or back edge, and label each vertex v with its depth $d(v)$ in the DFS tree. Then for each vertex v, compute the lowest depth $D(v)$ of any vertex reachable from v via a (possible empty) path of tree edges followed by at most one back edge. If $T(v)$ and $B(v)$ are the sets of out-going tree and back edges from v respectively, then

$$D(v) = \min(\{D(u) | \, u \in T(v)\} \cup \{d(u) | \, u \in B(v)\}).$$

Because each edge is labeled out-going from at most one vertex, these $D(v)$ values can be computed in $O(|E|)$ time by computing in reverse topological sort order of the vertices. Any vertex v that is not the root of the DFS tree, will be an articulation point if and only if none of its children in the DFS tree can reach a vertex with depth lower than $d(v)$ (a rigorous proof of this statement is omitted). So, for each non-root vertex v with DFS tree children $C(v)$, v is an articulation point if and only if $d(v) \geq \min_{u \in C(v)} D(u)$. Again, all such evaluations can be computed in $O(|E|)$ time since each vertex appears as a child of at most one vertex, and each such relation corresponds to an edge of the graph. As a special case, the root of the DFS tree will be an articulation point only when it has more than one child in the DFS tree. Since $|E| = \Omega(|V|)$, this algorithm runs in $O(|E|)$ time.

Rubric:

- 4 points for a correct algorithm
- Partial credit may be awarded
- 1 point for $O(|V||E|)$ running time analysis

Problem 7-2. [20 points] **Buzzword Bearing**

As an MIT student you naturally want to take 19.006, Introduction to Quantum Machine Learning on the Blockchain with Space Applications. Assume you adhere strictly to prerequisites, taking no classes before having satisfied the relevant prerequisite. You've scraped from the registrar, a list of all classes taught at MIT and all their prerequisite classes. For this problem, you may assume that every class is taught every semester at MIT.

(a) [10 points] Given a list of classes you've already taken, describe a linear time algorithm to determine the earliest semester you can take 19.006, assuming you can take as many classes as you want in any semester.

Solution: Construct a graph with a vertex for each of the n classes at MIT. Add a directed edge from class a to class b if class a is a prerequisite to class b and you have NOT yet taken class a. Let m be the number of such prerequisites. Then add auxiliary vertex s and connect it to every class. We assume that this graph is acyclic (that no cycle of prerequisite dependencies exists), or else no one would be allowed to take a class in that cycle. Then the length of the longest path from s to class 19.006 in this graph, will be the minimum number of semesters you need to take.

For each class/vertex v in the graph, we will compute the minimum number of semesters $m(v)$ needed to take that class starting from node s. Let $N(v)$ be the set of prerequisites to v. Since v cannot be taken before any prerequisite, then $m(v) = 1 + \max_{u \in N(v)} m(u)$. Compute $m(v)$ in topological sort order of the vertices from s, with $m(s) = 0$. All of these can be computed in $O(n + m)$ time as each edge appears in at most one prerequisite list $N(v)$.

Alternatively, one can assign each edge of the graph a weight of -1 and run topological sort relaxation to find a path with lowest weight from s to 19.006 in $O(n + m)$ time. The negative weight of this path will then be the number of semesters required. Note that we can negate weights to find longest paths because the graph is acyclic. This strategy would fail if the graph contained positive weight cycles.

Rubric:

- 8 points for a correct algorithm
- Partial credit may be awarded
- 2 points for linear running time analysis

(b) [10 points] Unfortunately, being able to take as many classes as you want isn't realistic, and your adviser is upset at you for proposing such a schedule. You've been given a credit limit of only 1 class per semester. Since you don't want to pay tuition indefinitely, design an algorithm, that returns a valid schedule for every semester that minimizes the number of semesters you're here.

Solution: First, we identify set of classes that must be taken prior to 19.006, and then we schedule those classes in a valid order. Begin with the same graph constructed for part (a), but remove any vertices from the graph not reachable from 19.006 following perquisite edges in the opposite direct. We can identify such vertices by running breadth- or depth-first search from 19.006 in $O(n+m)$ time, traversing edges only in the backward direction. Then, compute a topological sort order on the classes using DFS, and then schedule classes in the order of the topological sort, also in $O(n+m)$ time.

Rubric:

- 8 points for a correct algorithm
- Partial credit may be awarded
- 2 points for running time analysis (slow and correct OK)

Problem 7-3. [20 points] **Louis' Mansion**

One day, a vacuum salesman named Louis inherited his grandparent's mansion, which was said to be haunted. When he arrives to the mansion, trickster ghosts swooped through the hallways and opened up every single door! Louis needs to lock all the doors of the mansion as soon as possible to ensure that Browser, a large spider monster who lurks within the mansion cannot escape.

Every hallway connects exactly two rooms. Each room has an initially-open door to all of its hallways, and each door can either be open, closed, or locked. Doors can be locked from either side, but once a door is locked, even Louis can't open it. Louis has a very poor memory, and has no map of the mansion, but is very good at following instructions. Write a set of instructions for Louis that ensures that he locks every door in the mansion, starting and ending at the front door (and as always, be sure to analyze correctness and running time).

Solution: Model the mansion as a fully-connected undirected graph, where nodes are rooms and edges are hallways connecting two rooms. With each edge store additional data indicating the state of the two doors (open, closed, or locked) at the ends of the hallway corresponding to the edge. Initialize all door states to open at the start. We need to make sure they are all marked locked by the end. The approach will be to perform a DFS of this room-hallway graph. Each time Louis enters a room, Louis should follow these instructions:

- Case 1: If Louis enters a room for which all the doors in the room are open, then Louis followed a tree edge into a new room. Tell Louis to close the door he entered through (leaving it unlocked for now) then choose an open door to pass through into another room. Closed doors will represent parent pointers in DFS tree.
- Case 2: If Louis enters a room containing a closed door and does not lock the door he entered through, then Louis followed a back edge into a room visited previously. Tell Louis to go back via the hallway he just came from, locking both doors on his way out.
- Case 3: If Louis enters a room containing a closed door and locks the door he entered through, then Louis has just backtracked along a tree or back edge. If there is an open door, tell Louis to go through it, continuing the exploration. Otherwise, all other doors are locked, so tell Louis go through through the closed door hallway, locking its doors behind him.

By the properties of DFS, Louis will traverse every hallway (edge) once in both directions, thus because the second traversal of any hallway results in him locking both doors, Louis will lock every door in the mansion. Let the mansion have n rooms and m hallways. Since Louis will pass through every door twice, he will perform $O(m)$ instructions during his DFS. Each time he enters a room r, Louis must check the state of each door in the room (worst-case) in order to know what to do next, thus each instruction takes $O(\deg(r))$ time (and Louis only ever needs to remember the door in which he entered a room). Therefore, the total time is $O\left(md\right)$ where d is the maximum degree of the graph. Since $d = O(n)$, Louis' procedure takes $O(nm)$ time.

Rubric:

- 16 points for a correct algorithm
- Partial credit may be awarded
- 4 points for running time analysis (slow and correct OK)

Problem 7-4. [40 points] **Solving Sudoku**

A **Sudoku** puzzle consists of a 9 x 9 grid, with each grid square possibly a numerical digit from 1 to 9 inclusive. In an **unsolved** puzzle, some cells of the grid are pre-filled with digits, but many do not, as shown in the example below. The goal of the puzzle is to fill in each un-filled cell with a digit between 1 and 9 inclusive, such each row, each column, and each 3×3 sub-grid (with highlighted boundaries), contains the digits 1 through 9 exactly once. In this problem, we will develop an algorithm to return a fully filled, **solved** Sudoku given an unsolved Sudoku as an input.

(a) [5 points] **Implicit Graph:** Given a Sudoku configuration C with some filled cells, let the neighbors of C be the configurations which fill the first unfilled cell with a valid digit. A valid digit is one that does not cause any of the conditions to be violated. We will define the *first unfilled cell* to be the first empty cell when scanning each row left to right starting from the top. What is the minimum and maximum number of neighbors of any Sudoku puzzle configuration? Describe how to compute all neighbors of a given configuration.

Solution: The minimum number of neighbors of a configuration is 0, if there are no valid digits for the next unfilled cell (or if there are no unfilled digits). The maximum number of neighbors of a configuration is 9, if all the digits 1 through 9 are valid for the first unfilled cell. To identify all neighbors of a given configuration, initialize an array of length 9, corresponding to each possible valid digit 1-9. Then iterate through the row, column, and sub-grid containing the first unfilled cell. Mark any digit already appearing in any of these locations, since a conflict would result if such a digit was included in the unfilled cell. We can then iterate through the array to identify all remaining (unmarked) digits, which are valid choices for the unfilled cell. Finding these configurations takes time proportional to the number of squares in the board ($\sim 9^2$). Since 9 is constant, neighbors can be computed in constant time.

Rubric:

- 5 points for a correct algorithm
- Partial credit may be awarded
- As a constant sized problem, asymptotic running time analysis not required

(b) [10 points] **Sudoku Solve:** Given an input unsolved Sudoku puzzle, describe an algorithm to output a solved Sudoku, using the implicit graph defined in part (a). If there are multiple solutions, your algorithm should return the solution which leads to the smallest number when concatenating all the digits in the initially empty cells, in order from the top row to the bottom row. Your algorithm should use only $O(d)$ space, where d is the number of unfilled cells in the initial Sudoku. (**Hint:** can the graph from (a) contain cycles?)

Solution: We can perform depth-first search on the given Sudoku puzzle starting from the initial unsolved puzzle. The neighbors of any configuration can be identified in constant time using the algorithm designed in part (a). At each node, we explore the neighbors in ascending order to ensure that the correct solution is returned when multiple solutions exist. Once a configuration with no empty cells is reached, then the first valid solution has been found and can be returned. If no configuration without empty cells is reached through the DFS, then None is returned. Note that normally DFS would need to store every configuration previously explored to efficiently identify non-tree edges. However, the Sudoku configuration space built in this way cannot contain cycles, so no configuration can be processed more than once. Thus DFS only needs to store information about the configuration currently being evaluated, which requires space proportional to the number of unfilled cells in the initial Sudoku. This algorithm potentially searches $(9^2)!$ configurations on an unsolved board, so in the worst case runs in time proportional to that number.

Rubric:

- 8 points for a correct algorithm
- Partial credit may be awarded
- 2 points for argument that space usage is proportional to d

- As a constant sized problem, asymptotic running time analysis not required

(c) [25 points] **Implement:** Write the Python function `solve_sudoku(config)` that implements your puzzle solving algorithm. An input configuration will be a list of lists, each of length 9, corresponding to each row of the Sudoku board. Each element in a row will be an integer from 0 to 9 inclusive, where 0 corresponds to an empty cell, e.g., the first row of the unsolved Sudoku shown above would be represented as `[0,2,0,5,0,1,0,9,0]`. Your function should return a solved Sudoku using the algorithm described in part (b). The output should be in the same format as the input, but containing no 0 entries. If no valid solution exits, your function should return `None`. Submit your code online at `py.mit.edu/6.006`.

Solution:

```
def nextValid(config, row, col):
    seen = [True for i in range(10)]     # keep track of conflicting
    for i in range(9):
        gr, gc = 3 * (row // 3) + (i // 3), 3 * (col // 3) + (i % 3)
        seen[config[gr][gc]] = False     # check grid
        seen[config[row][i]] = False     # check row
        seen[config[i][col]] = False     # check column
    for i in range(1, 10):
        if seen[i] == True:
            yield i                      # return if not conflicting

def solve_sudoku(config):
    for i in range(9):                   # find first empty square
        for j in range(9):
            if config[i][j] == 0:                       # check if empty
                for choice in nextValid(config, i, j):  # neighbors
                    config[i][j] = choice               # make neighbor
                    solved = solve_sudoku(config)       # recursive call
                    if solved is not None:              # check if found
                        return solved                   # solution found!
                config[i][j] = 0
                return None              # no solution found :(
    return config                        # no empty entries
```