Problem Set 5

All parts are due on March 22, 2019 at 6PM. 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.mit.edu`.

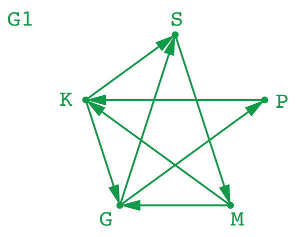
Problem 5-1. [16 points] **Graph Mechanics**

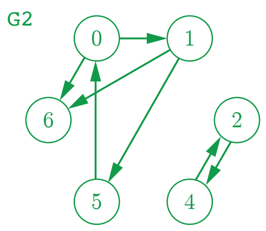
(a) [4 points] Draw the **directed** graph associated with each of the following graph representations.

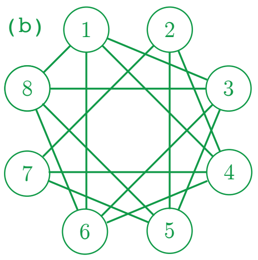
```
G1 = {"StringMander": ["MITsaur"],
          "KoiQueen": ["StringMander", "GeoBro"],
            "GeoBro": ["StringMander", "PikaBoo"],
           "MITsaur": ["KoiQueen", "GeoBro"],
           "PikaBoo": ["KoiQueen"]}

G2 = [[1,6], [5,6], [4], None, [2], [0], []]
```

Solution:







Rubric:

- 2 points for any graph drawing depicting correct vertices for G_1
- -1 point for each incorrect or missing edge in G_1, minimum zero points
- 2 points for any graph drawing depicting correct vertices for G_2
- -1 point for each incorrect or missing edge in G_2, minimum zero points

(b) [4 points] Let the ***k*-prime-difference graph** be the undirected graph on vertices $\{1, \ldots, k\}$ having an undirected edge between distinct vertices u and v if and only if $|u - v|$ is a prime number. For example, the 5-prime-difference graph contains edges $\{5, 3\}$ and $\{2, 5\}$ (and others), but not edges $\{4, 5\}$ or $\{5, 1\}$. Draw a picture of the 8-prime-difference graph, and write its graph representation as a direct access array of adjacency lists, as in `G2` above. Hint: your graph should have 15 edges.

Solution:

```
G_8 = [None, [3,4,6,8], [4,5,7], [1,5,6,8],
       [1,2,6,7], [2,3,7,8], [1,3,4,8], [2,4,5], [1,3,5,6]]
```

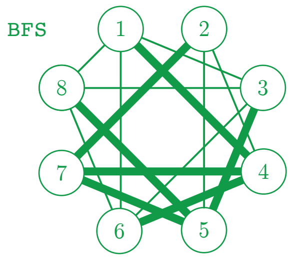
Rubric:

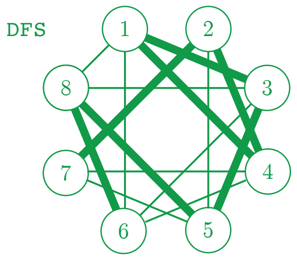
- 2 points for any graph drawing depicting correct vertices
- -1 point for each incorrect or missing edge, minimum zero points
- 2 points for any array of adjacency lists
- -1 point for each mistake in lists, minimum zero points

(c) [8 points] Run breadth-first search and depth-first search on the 8-prime-difference graph, starting from node 7. While performing each search, visit the neighbors of each vertex in increasing order. For each search, draw the parent tree constructed by the search, and list the order in which nodes are first visited.

Solution:

```
BFS: [7, 2, 4, 5, 1, 6, 3, 8]
DFS: [7, 2, 4, 1, 3, 5, 8, 6]
```





Rubric:

- 2 points for each parent pointer tree on correct vertices
- -1 point for each incorrect tree edge, minimum zero points per graph
- 2 points for each first-visited list
- -1 point for each mistake in a list, minimum zero points per list

Problem 5-2. [10 points] **Graph Diameter**

In any undirected graph $G = (V, E)$, the **eccentricity** $e(u)$ of a vertex $u \in V$ is the shortest distance to its farthest vertex v, i.e., $e(u) = \max\{\delta(u, v) \mid v \in V\}$. The **diameter** $D(G)$ of an undirected graph $G = (V, E)$ is the largest eccentricity of any vertex, i.e., $D(G) = \max\{e(u) \mid u \in V\}$. The diameter of a graph is then the largest distance between two nodes.

(a) [5 points] Given connected undirected graph G, describe an $O(|V||E|)$-time algorithm to determine the diameter of G.

Solution: Compute $D(G)$ directly: run breadth-first search from each node u and calculate $e(u)$ as the maximum of $\delta(u, v)$ for each $v \in V$. Then return the maximum of $e(u)$ for all $u \in V$. Each vertex is connected to an edge because G is connected so $|V| = O(|E|)$. Each BFS takes $O(|E|)$ time, leading to $O(|V||E|)$ time in total.

Rubric:

- 4 points for description of a correct algorithm
- 1 point for a correct analysis of running time analysis
- Partial credit may be awarded

(b) [5 points] Given connected undirected graph G, describe an $O(|E|)$-time algorithm to determine an upper bound D^* on the diameter of G, such that $D(G) \le D^* \le 2D(G)$.

Solution: Use breadth-first search to find $e(u)$ for any $u \in V$ in $O(|E|)$ time. We claim that $D(G) \le 2e(u) \le 2D(G)$, so we can choose $D^* = 2e(u)$. First, $e(u) \le D(G)$ because $D(G)$ is the maximum $e(v)$ over all $v \in V$. Second $D(G) \le 2e(u)$ because we can construct a path between any two vertices (x, y) by concatenating a shortest path from x to u and a shortest path from u to v, both of which has length at most $e(u)$; thus diameter $D(G)$ cannot be greater than $2e(u)$.

Rubric:

- 4 points for description of a correct algorithm
- 1 point for a correct analysis of running time analysis
- Partial credit may be awarded

Problem 5-3. [8 points] **Social Optimization**

LinkedOUT is a new social network. Every pair of LinkedOUT users is either **acquainted** or not acquainted with each other. Two users u_1 and u_2 are **approachable** if there exists a sequence of users starting with u_1 and ending with u_2 where each adjacent pair of users in the sequence are acquainted. New user Weff Jeiner has just signed up for LinkedOUT and is not acquainted with anyone. Given a list of all pairs of users that are acquainted on LinkedOUT, describe an efficient algorithm to determine the minimum number of users with whom Weff needs to become acquainted in order to be approachable by every user on LinkedOUT.

Solution: You need to assume something to solve this problem: either you assume that every user exists in some input pairs, or you assume you have access to a list of all users. We make the former assumption, and let n be the number of input pairs (so that the number of users is also at most n).

The LinkedOUT network can be represented by an undirected graph on the users, with an undirected edge between users a and b if they are acquainted. This graph may have one or more connected components. For Weff to be approachable by every user requires that Weff be acquainted with at least one user from every connected component, so the solution to this problem is equivalent to counting the number connected components in the graph. To count the number of connected components, repeatedly choose an arbitrary unmarked user and run a graph search (either BFS or DFS can be used) to mark all users reachable from that user, i.e., marking that user's connected component. Then return the number of distinct searches that were made. Because each user is contained in at most one search, and each search runs in linear time with respect to the connected component it explores, this full graph search takes $O(n)$ time.

The above solution is sufficient for full points. There is a small detail of whether this running time is expected or worst-case. If users are not representable as integers in a linearly bounded range, you may have to use a hash table to access adjacency lists. Otherwise, you can use a direct access array providing worst-case bounds. We do not require you to differentiate worst-case vs. expected bounds for graph problems; either will suffice.

Rubric:

- 1 points for description of graph
- 4 points for description of a correct algorithm
- 2 points for a correct argument of correctness
- 1 points for correct analysis of running time
- Partial credit may be awarded

Problem 5-4. [21 points] **Beaver Break**

The snow and ice have **dam**pened the spirits of Tim the Beaver. He's super excited to spend spring break on a beach in Miami, but he forgot to book flights which are now very expensive. Tim wonders whether he can save money traveling by bus. The Silverdog bus company offers bus routes around the country, where each bus route is a pair (c_1, c_2) representing a non-stop trip from city c_1 to city c_2. Assume it is possible to reach Miami from Boston on the Silverdog bus network.

(a) [4 points] Silverdog offers big discounts to students! A student ticket on any bus route costs exactly one dollar. Given a list of the n bus routes serviced by Silverdog, describe an $O(n)$-time algorithm to determine the minimum number of dollars Tim needs to spend on bus tickets to get from Boston to Miami.

Solution: Construct a graph on all cities serviced by Silverdog, with a directed edge from c_1 to c_2 for every bus route (c_1, c_2). This graph has n edges and fewer vertices, so constructing this graph takes $O(n)$ time. Then we can run a breadth-first search from Boston to find the length of a shortest path to Miami in this graph, also in $O(n)$ time. Each edge traversed costs one dollar, so returning the length of the shortest path is equivalent to the minimum number of dollars that Tim needs to spend.

Rubric:

- 1 point for description of graph
- 2 points for description of a correct algorithm
- 1 point for correct analysis of running time
- Partial credit may be awarded

(b) [7 points] Tim is despondent. When he goes to buy bus tickets, Tim discovers that the discount only applies to human students, so he will sadly have to pay full price. The full price of each bus ticket varies, but is always a positive integer number of dollars. Describe an $O(kn)$-time algorithm to determine whether there exists a sequence of bus routes that Tim can take from Boston to Miami that costs less than k dollars.

Solution: Construct a graph on all cities, but this time with a a-edge directed chain of edges from c_1 to c_2 for every bus route (c_1, c_2) having price $a \le k$ (and no chain of connections if the bus route costs more than k dollars). This graph has at most kn edges and fewer vertices, so constructing this graph takes $O(kn)$ time. Then we can run breadth-first search from Boston to find the length ℓ of a shortest path to Miami in this graph, also in $O(kn)$ time. If $\ell \le k$, then Tim can reach Miami spending at most k dollars, and cannot reach Miami within that budget if $\ell > k$.

Rubric:

- 3 points for description of graph
- 2 points for description of a correct algorithm
- 2 point for correct analysis of running time
- Partial credit may be awarded

(c) [10 points] Tim complains on social media that Silverdog's business practices are discriminatory. The resulting public outrage prompts Silverdog to lift the restriction, allowing Tim to buy dollar student tickets! But in order to compensate for the loss in revenue, Silverdog places the following restriction on all student tickets: if a passenger enters a city on a bus using a student ticket, they may not use a student ticket to leave that city (so must pay full price). Describe an $O(kn)$-time algorithm to determine whether there exists a sequence of bus routes that Tim can take from Boston to Miami that costs less than k dollars subject to these new restrictions.

Solution: To solve this problem, we combine the ideas of parts (a) and (b). We construct a graph having two vertices per city c: (1) vertex c^s corresponding to being at city c and able to leave on either a regular or a student ticket, and (2) vertex c^r corresponding to being at city c only able to leave on a regular ticket. Then for each bus route (c_1, c_2) having price $a < k$, add a single directed edge from vertex c_1^s to c_2^r, and two a-edge directed chains of edges: one from c_1^s to c_2^s, and another from c_1^r to c_2^s. This graph has at most $3kn$ edges and fewer vertices, so constructing this graph takes $O(kn)$ time. Then any path from Boston to Miami in this graph corresponds to visiting a sequence of cities without taking two bus routes in a row at the student rate. So we can run breadth-first search from Boston to find the length ℓ of a shortest path to Miami in $O(kn)$ time. As in part (b), Tim can reach Miami by spending at most k dollars if and only if $\ell \le k$.

Rubric:

- 4 points for description of graph
- 4 points for description of a correct algorithm

- 2 point for correct analysis of running time
- Partial credit may be awarded

Problem 5-5. [45 points] **Peg Solitaire**

Peg solitaire is a single player game played on a **game board**: a square grid having R rows and C columns. The board begins in a given **initial configuration** where each grid square either contains a peg, or is empty and does not contain a peg. Then given any **straight line** of three adjacent grid squares (x, y, z) where squares x and y contain pegs and square z is empty, a player can make a **move** by moving the peg at square x into square z and removing the peg at square y from the board, resulting in a new board configuration containing one fewer peg. Specifically, a move can be specified by triple $m = (r, c, d)$ corresponding to moving the peg in row r and column c in the cardinal direction $d \in \{N, E, S, W\}$, if the move is possible. A board configuration is **solved** if exactly one of its grid squares contains a peg. An initial configuration is **solvable** if the player can make a sequence of moves to transform the board into a solved configuration.

In this problem, we represent a board configuration B as a length R tuple of length C tuples, where $B[r][c] = 1$ if the grid square in row r and column c contains a peg, and $B[r][c] = 0$ otherwise. Given a board configuration B and triple $m = (r, c, d)$, let `make_move(`$B, m,$`"forward")` be the board configuration B' resulting from making move m on B (or `None` if move m is not a valid move). Further, if $B' =$ `make_move(`$B, m,$`"forward")`, we call the transformation of B' into B the **inverse** application of move m, where `make_move(`$B', m,$`"backward")`$= B$. We have provided an implementation of `make_move` in your code template.

(a) [3 points] Apply moves $((0, 1, S), (1, 3, W), (1, 1, S))$ in sequence to the board configuration below, and draw the resulting board after each move.

```
          c   0   1   2   3   4
                                     r              |
     B = ((0,  1,  0,  0,  0),       0              N
           (0,  1,  1,  1,  0),      1          --W   E--
           (0,  0,  0,  0,  0),      2              S
           (0,  0,  1,  0,  0))      3              |
```

Solution:

```
(0, 1, S) =>            (1, 3, W) =>            (1, 1, S) =>
   ((0, 0, 0, 0, 0),       ((0, 0, 0, 0, 0),       ((0, 0, 0, 0, 0),
    (0, 0, 1, 1, 0),        (0, 1, 0, 0, 0),        (0, 0, 0, 0, 0),
    (0, 1, 0, 0, 0),        (0, 1, 0, 0, 0),        (0, 0, 0, 0, 0),
    (0, 0, 1, 0, 0))        (0, 0, 1, 0, 0))        (0, 1, 1, 0, 0))
```

Rubric:

- 1 point per correct move

(b) [3 points] Inversely apply moves $((3, 4, W), (3, 1, E), (3, 3, N))$ in sequence to the board configuration above, and draw the resulting board after each move inverse.

Solution:

```
inv(3, 4, W) =>      inv(3, 1, E) =>      inv(3, 3, N) =>
   ((0, 1, 0, 0, 0),    ((0, 1, 0, 0, 0),    ((0, 1, 0, 0, 0),
    (0, 1, 1, 1, 0),     (0, 1, 1, 1, 0),     (0, 1, 1, 0, 0),
    (0, 0, 0, 0, 0),     (0, 0, 0, 0, 0),     (0, 0, 0, 1, 0),
    (0, 0, 0, 1, 1))     (0, 1, 1, 0, 1))     (0, 1, 1, 1, 1))
```

Rubric:

- 1 point per correct inverse move

(c) [4 points] The **accessibility** of an initial board configuration B is the number of board configurations that can be reached via a sequence of moves from B. If N is the accessibility of an initial board configuration having R rows and C columns containing exactly k pegs, prove that $\log N = O(\min(RC, k \log k))$.

Solution: The total number of configurations for a board with R rows and C columns is 2^{RC}, where each configuration has a binary choice at each square (peg or no), so $N \leq 2^{RC}$, implying $\log N = O(RC)$. Further, given configuration B containing k pegs, the number of configurations reachable in one move from B is at most $4k$ (since each peg may be moved in one of four directions). Then the number reachable in two moves is at most $4k + (4k)(4(k-1))$, and the number reachable in i moves is at most $4k + \ldots + 4^i k!$. Thus sequence increases super-geometrically, so it is asymptotically dominated by its last term, so $N < 4^{i+1}k!$. Then by Stirling's approximation, $\log N = O(k \log k)$ as desired.

Rubric:

- 2 points for a correct argument for each bound in the min

(d) [10 points] Given an initial board configuration B with R rows and C columns and accessibility N, describe an $O(kRCN)$-time algorithm to return a sequence of moves that solves B, or returns `None` if no such sequence exists.

Solution: Our approach will be to explore the graph on all board configurations that are accessible from B having k pegs. Then, for each board configuration, we can tell whether it is solved by seeing whether the sum of its entries equals one, naively in $O(RC)$ time. To perform this exploration, we can use breadth-first-search or depth-first search. For each explored configuration, we store a parent move: the last move made before reaching the configuration during the search. At each step of the search, we can loop through each of the $O(k)$ neighbors of a given configuration in $O(RC)$ time by checking all possible peg moves using the `make_move` function provided. Since the graph search explores at most N configurations, and does at most $O(kRC)$ work per configuration, this exploration takes $O(kRCN)$ time. If during the search, a solved configuration is reached, reconstruct the sequence of moves by repeatedly applying `make_move(`$B, m,$`"backward")` to parent moves until the original configuration is reached in $O(kRC)$ time; otherwise, return `None` since a solved configuration is not reachable from B via a sequence of moves.

Rubric:

- 3 points for description of graph
- 4 points for description of a correct algorithm
- 3 point for correct analysis of running time
- Partial credit may be awarded

(e) [25 points] Write a Python function `peg_solve(B)` that implements your algorithm. You can download a code template containing some test cases from the website. Submit your code online at `alg.mit.edu`.

Solution:

```
DIRECTIONS = {"N": (-1, 0), "E": ( 0, 1), "S": ( 1, 0), "W": ( 0,-1)}
def make_move(B, m, orientation):
    if m is None: return None
    r, c, d = m
    dr, dc = DIRECTIONS[d]
    R, C = len(B), len(B[0])
    if (0 <= r + 2*dr < R) and (0 <= c + 2*dc < C):
        before, after = (1, 1, 0), (0, 0, 1)
        if orientation is "backward":
            before, after = after, before
        for i in range(3):
            if B[r + i*dr][c + i*dc] != before[i]:
                return None
        B_ = [[p for p in row] for row in B]
        for i in range(3):
            B_[r + i*dr][c + i*dc] = after[i]
        return tuple(tuple(p for p in row) for row in B_)

def is_solved(B):
    return sum(sum(row) for row in B) == 1

def neighbors(B):
    R, C = len(B), len(B[0])
    for r in range(R):
        for c in range(C):
            for d in DIRECTIONS:
                m = (r, c, d)
                B_ = make_move(B, m, "forward")
                if B_:
                    yield m, B_

def make_path(B, moves):
    path = []
    while B in moves:
        if moves[B]:
            path.append(moves[B])
        B = make_move(B, moves[B], "backward")
    path.reverse()
    return path

def peg_solve(B):                     # DFS Implementation
    moves, stack = {B: None}, [B]
    while len(stack) > 0:
        P = stack.pop()
        for m, B_ in neighbors(P):
            if B_ not in moves:
                moves[B_] = m
                stack.append(B_)
                if is_solved(B_):
                    return make_path(B_, moves)
```