Problem Set 5

All parts are due on October 24, 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 .

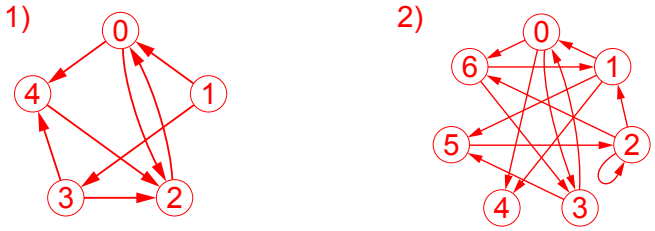
Problem 5-1. [30 points] **Graph Practice** A graph $G = (V, E)$ has two common representations. An *adjacency matrix* is a $|V| \times |V|$ matrix of boolean values, where the matrix element in row i and column j is 1 if edge (V_i, V_j) is in E, and 0 otherwise. An *adjacency list* is a $|V|$ length list of lists, where the ith list contains index j if edge (V_i, V_j) is in E.

(a) [8 points] **Representation to Graph:** Draw the **directed graph** associated with each of the following graph representations. (1) is an adjacency matrix representation and (2) is an adjacency list representation.

```
(1) = [[0 0 1 0 1],
       [1 0 0 1 0],
       [1 0 0 0 0],
       [0 0 1 0 1],
       [0 0 1 0 0]]
```

```
(2) = [[3,4,6],
       [0,4,5],
       [1,2,6],
       [0,5],
       [],
       [2],
       [1,3]]
```

Solution:

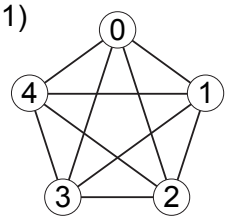


Rubric:

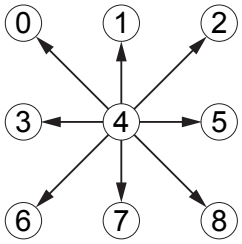
- 4 points for each correct drawing
- -1 point for each incorrect edge (max -4 for each graph)

(b) [16 points] **Graph to Representation:** Write down adjacency matrix and adjacency list representations for each of the following graphs.

1)



2)



Solution:

```
(1 matrix) = [[0 1 1 1 1],
              [1 0 1 1 1],
              [1 1 0 1 1],
              [1 1 1 0 1],
              [1 1 1 1 0]]

(2 matrix) = [[0 0 0 0 0 0 0 0 0],
              [0 0 0 0 0 0 0 0 0],
              [0 0 0 0 0 0 0 0 0],
              [0 0 0 0 0 0 0 0 0],
              [1 1 1 1 0 1 1 1 1],
              [0 0 0 0 0 0 0 0 0],
              [0 0 0 0 0 0 0 0 0],
              [0 0 0 0 0 0 0 0 0],
              [0 0 0 0 0 0 0 0 0]]
```

```
(1 list) = [[1,2,3,4],
            [0,2,3,4],
            [0,1,3,4],
            [0,1,2,4],
            [0,1,2,3]]

(2 list) = [[],
            [],
            [],
            [],
            [0,1,2,3,5,6,7,8],
            [],
            [],
            [],
            []]
```

Rubric:

- 4 points for each correct representation
- -1 point for each incorrect index (max -4 per representation)

(c) [6 points] **Simply Connected:** A graph is *connected* if there is a path between every pair of vertices. A graph is *simple* if every edge connects different vertices, and no two distinct edges exist between the same pair of vertices. What is the maximum and minimum number of edges in a connected simple undirected graph on $|V|$ vertices? What about for simple undirected graphs that are not connected? What about for connected undirected graphs that are not simple?

Solution:

If an undirected graph is connected, edges must form at minimum a spanning tree of the vertices. If it is simple and connected, the most edges occurs in a complete graph, with an edge between every pair of vertices. If the graph is simple but not connected, no edge is necessary. However, the maximum number of edges is less than the connected case as the graph must contain at least two distinct connected components, and is maximized when one of the components is a single vertex. If the graph is connected but not simple, a spanning tree must exist, in addition to one more edge to make the graph non-simple; but maximum number of edges is unbounded.

Connected	Simple	Min	Max
Y	Y	$\|V\| - 1$	$\|V\|(\|V\| - 1)/2$
N	Y	0	$(\|V\| - 1)(\|V\| - 2)/2$
Y	N	$\|V\|$	∞

Rubric:

- 1 point per correct min or max

Problem 5-2. [30 points] **Flying Feline**

Jon is planning a long trip with his emotional support animal, Mr. G, a fat orange cat who likes to drink coffee and eat lasagna. Jon is obsessive about trip planning and has collected up-to-date timetables from all c airline companies, containing information on all r flight routes, between the a airports worldwide. For each of the following independent scenarios, help Jon and Mr. G plan a flight itinerary consisting of multiple flights from their home airport to their destination airport. Make sure to specify the vertices and edges for any graphs you might construct.

(a) [5 points] **Short Flights:** Jon hates traveling on long flights with Mr. G. He prefers *short* flights that are under 1 hour. Describe a linear time algorithm to find a flight itinerary with the fewest flights, all of them being short, or to determine that no such itinerary exists.

Solution: Construct a directed graph G with one vertex per airport and a directed edge from one airport to another exactly when there exists a short route between them. This graph takes $O(a+r)$ time to build, containing a vertices and at most r edges. Run BFS on this graph, starting at the home airport. If BFS finds the destination airport, return the path found, which by BFS will be a shortest path. Otherwise no route exists using only short flights, so return that none exists. BFS runs in $O(|V| + |E|)$, so total runtime is $O(a + r)$ as desired.

Rubric:

- 5 points for a correct $O(a + r)$ algorithm
- Partial credit may be awarded

(b) [5 points] **Coffee:** Mr. G loves to drink coffee, but unfortunately not all flights serve coffee. Describe a linear time algorithm to find a flight itinerary with the fewest flights that do not serve coffee, or to determine that no such itinerary exists.

Solution: Construct a directed graph G with one vertex per airport and a directed edge from one airport to another exactly when there exists a route between them. This graph has size $O(a + r)$ and can be constructed in as much time. Run a modified BFS on the graph, starting at the home city. When processing an airport v in the frontier, run BFS locally starting at v, exploring only edges representing routes that serve coffee, adding all newly explored airports to the frontier, and adding parent pointers to the original BFS tree. If the modified BFS finds the destination airport, return the path found. Otherwise, no route exists, so return that none exists. To see that this algorithm returns an itinerary with the fewest flights that do not serve coffee, observe that all airports reachable with exactly k flights not serving coffee are processed consecutively with this algorithm, so if the destination city is found for the first time, the path is guarenteed to be a path with fewest flights not serving coffee. Each edge is traversed at most once, so the modified BFS also runs in $O(a + r)$ as desired.

Rubric:

- 5 points for a correct $O(a + r)$ algorithm
- Partial credit may be awarded
- Due to the ambiguity of the problem, correct algorithms that return itineraries with fewest non-coffee flights that do not have the minimum number of total flights among possible iteneraries should be awarded full points.

(c) [10 points] **Lasagna:** Mr. G is very grumpy when he hasn't eaten lasagna for a while. Jon wants to make sure his itinerary is *lasagna dense*: for any airport on his itinerary, at least one of the next, current, or previous airports on the itinerary contains an Italian restaurant. Let d be the maximum number of routes out of any airport. Describe a $O(d^3a)$ time algorithm to find a lasagna dense flight itinerary with the fewest number of flights, or to determine that no such itinerary exists. You may assume that the home and destination airports both contain Italian restaurants.

Solution: Construct the same graph G as part (b). For each airport $v \in V$ containing an Italian restaurant, run BFS from v, but only searching airports within three flights of v. Store a list P_v of all $O(d^3)$ paths of flights starting from airport v with length at most 3 that ends at another Italian restaurant. Each BFS can be performed in $O(d^3)$ time for, each of the at most a airports containing an Italian restaurant. Then construct a new graph G' in the following way. Add a vertex for each airport containing an Italian restaurant. Then for each vertex, add a new path from v to u for each path in P_v that ends in airport u. Run BFS on G' starting at the home airport, searching for the destination airport. If BFS finds the destination airport, return the path found. Otherwise, no route exists, so return that none exists. This graph G' has at most $a + 2d^3a$ vertices and at most $3d^3a$ edges, so can be constructed and searched with BFS in $O(d^3a + a)$ time.

Faster Solution: A more efficient $O(da + a)$ algorithm also exists. Construct the same graph G as part (b). Then construct a new graph G' containing $3a$ vertices, with three vertices for each airport: $(a_i, 0)$, $(a_i, 1)$, and $(a_i, 2)$. Let node (a_i, j) represent landing in airport i having last been at an airport with an Italian restaurant j airports before. For all edges (v, u) in G, add edges to G' in the following way. If u does not contain an Italian restaurant, add an edge in G' from $(v, 0)$ to $(u, 1)$ and an edge from $(v, 1)$ to $(u, 2)$. Alternatively, if u contains an Italian restaurant, add an edge to $(u, 0)$ from (v, j) for $j \in [0, 2]$. Let s be the starting airport and t be the destination airport. Run BFS on G', beginning at $(s, 0)$ and ending at $(t, 0)$, under the assumption that s and t both contain an Italian restaurant. If BFS finds the destination airport, return the path found. Otherwise, no route exists, so return that none exists. The runtime for constructing the graph and running BFS is $O(da + a)$, as there exists $3a$ vertices and at most $3da$ edges in G'.

Rubric:

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

(d) [10 points] **Sassy Tabby:** Mr. G often offends the flight attendants onboard the aircraft, so it can be embarrassing for Jon to fly on the same airline two flights in a row. Describe a $O(c(a+r+1))$ time algorithm to find a flight itinerary with the fewest flights containing no two consecutive flights on the same airline, or to determine that no such itinerary exists.

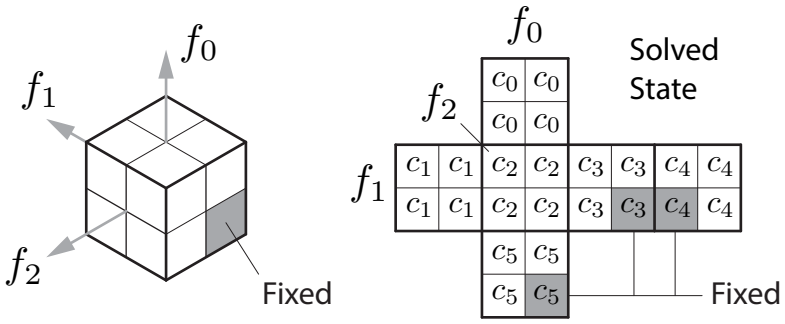
Solution: Construct a graph G on ca vertices, one for each airport/airline pair (a_i, c_j). Connect an edge from (a_i, c_j) to (a_k, c_l) if there is a flight from airport a_i to airport a_j on airline c_l, and $c_j \neq c_l$. Add an additional vertex v representing the home airport and connect it to each vertex (v, c_j) for $j \in [1, c]$. This graph has $ca + 1$ vertices and $(c-1)r + c$ edges and can be constructed in $O(c(a + r + 1))$ time. Run BFS starting from vertex v, searching for any vertex associated with the destination airport. If BFS finds the destination airport, return the path found. Otherwise, no route exists, so return that none exists. This algorithm is correct because any edge traversed corresponds to a valid route, and no two consecutive edges are flights on the same airline by construction. BFS runs in $O(|V| + |E|)$, so total runtime is $O(c(a + r + 1))$ as desired.

Faster Solution: A more efficient $O(a+r)$ algorithm also exists. The key observation is that if Jon can land in an airport using two different airlines, he will be able to traverse all outgoing edges from that airport. So each airport needs to be visited at most twice in any shortest path. Construct the same graph G as in part (b). Run a slightly modified BFS on this graph, starting at the home airport. During the BFS, we allow each airport to be visited and searched once or twice. Each airport stores up to two parent routes associated with different airlines used to visit the airport. When running BFS, we add an airport to the queue after arriving there on route r if the airport has not yet been visited, or the airport has been visited once via a route run by a different airline than route r. If BFS finds the destination airport, return the path by traversing back along the parent flight information stored at each node. Each airport is only visited twice, therefore each route is traversed at most twice, so total runtime is $O(a + r)$.

Rubric:

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

Problem 5-3. [40 points] **Pocket Cube:** Gill Bates attended the 6.006 lecture on breadth-first-search (BFS) and is super excited. He heard that a Rubik's cube can be modeled as a graph where each vertex is a configuration of the cube. Gill received a variant of a Rubik's cube for his birthday, a $2 \times 2 \times 2$ Pocket Cube[1]. Help Gill write a computer program to quickly solve a scrambled Pocket Cube in the fewest moves possible.



(a) [2 points] **Configurations:** A Pocket Cube consists of eight corner cubes, each with a different color on its three visible faces. A solved configuration is one in which each 2×2 face of the Pocket Cube is monochromatic. We reference each color c_i with an index $i \in [0, 5]$. Without loss of generality, we fix the position and orientation of one of the corner cubes and only allow clockwise rotations of the three faces of the Pocket Cube $\{f_0, f_1, f_2\}$ that do not contain the fixed corner cube. Argue an upper bound on the number of possible configurations of a Pocket Cube under this restriction (try to get as tight a bound as you can using combinatorics).

Solution: If one of the corner cubes is fixed, the remaining seven corner cubes may exist in $7!$ permutations, while each corner cube may rotate independently to any of three rotations. So the number of configurations is upper bounded by $7!3^7 = 11022480$.

Rubric:

- 2 points for a correct upper bound acheived by combinatorial means. The bound does not have to be tight.

(b) [2 points] **Implicit Graph:** Gill chooses not to store the Pocket Cube graph explicitly. Instead, from a given configuration, he will compute a list of adjacent configurations accessible in a single move. He considers a *move* to be rotation of one face by any amount. The *degree* of a vertex in an undirected graph is the number of edges incident to the vertex. What is the maximum and minimum degree of vertices in the Pocket Cube graph?

Solution: Three sides may be rotated, and each may be rotated by $90°$, $180°$, or $270°$. So each configuration has exactly $3 \times 3 = 9$ configurations neighboring it.

[1] `http://en.wikipedia.org/wiki/Pocket_Cube`

Rubric:

- 1 point each for a correct upper and lower bound

(c) [6 points] **Code Review:** Gill has written some code to explore the Pocket Cube graph using BFS. He searches the entire graph, building a BFS tree by storing parent pointers to preceding configurations in a hash map (Python dictionary). After searching the entire graph, he returns a sequence of moves that solves the Pocket Cube. Unfortunately, his solver is very slow. Run the code provided in the template file. Please note that the code requires a couple minutes and considerable memory (over 400 Mb) to complete.

1. How many configurations does Gill's BFS search? How does this number compare to your upper bound from part (a)?

   Solution: Gill's BFS searches 3674160 configurations. This is exactly one third of the configurations estimated by part (a). In fact, the space of configurations turns out to be three disconnected components of equal size. A representative configuration contained in each component can be achieved by rotating a single corner cube of the solved state to each of its three rotations.

   Rubric:

   - 1 point for number of configurations
   - 1 point for comparison

2. What is the *diameter* of the Pocket Cube graph, the longest distance between any two configurations?

   Solution: This follows directly from the code output: the diameter is equal to the number of frontiers visited minus 1, i.e. eleven.

   Rubric:

   - 2 points for a correct diameter

3. How many edges are in the Pocket Cube graph?

   Solution: There are 3674160 configurations, each with degree 9. So by the handshaking lemma, the number of edges must be $3674160(9/2) = 16533720$.

   Rubric:

   - 2 points for a correct number of edges

(d) [10 points] **Two-Way BFS:** Gill realizes that his code is quickly able to explore vertices of the graph that are within half the diameter from a given configuration, but it takes a long time to explore vertices that are more than half the diameter away. Let d be the maximum degree of the Pocket Cube graph, and let w be its diameter. Describe an algorithm to find a shortest sequence of moves to solve a given Pocket Cube configuration by visiting no more than $2d^{\lceil w/2 \rceil}$ configurations.

Solution: Run BFS from both the query configuration and the solved configuration, but alternating exploring frontiers from each. Store parent pointers to the configuration preceded by each explored configuration. After exploring each frontier, check if a configuration in the new frontier has been explored by the other BFS. If it has been explored by the other BFS, construct the path from the query configuration to the solved configuration through the overlapping node, by following parent pointers from one BFS to the query configuration, and following parent pointers from the other BFS to the solved configuration. Because we alternate exploring frontiers, the lengths of the paths found by each BFS differ by at most one, so each has length at most $\lceil w/2 \rceil$. Further, a frontier at distance i from its source contains at most d^i, so the number of configurations explored is at most $d^{\lceil w/2 \rceil}$ for each of the two BFSs, as desired.

Rubric:

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

(e) [20 points] **Implement:** Implement the `solve_faster` function stub in Gill's code, based on your algorithm from part (d). Please submit code for this problem in a single file to alg.csail.mit.edu. ALG will only import your implementation of `solve_faster`, though you may call any other function from Gill's code base from within your function.

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

```
# Gill's Pocket Cube Code
# -----------------------
# A Pocket Cube configuration is represented by a string of 24 color indices.
# Here is a string that represents the solved configuration:
SOLVED = '000011223344112233445555'

# Color locations correspond to a Latin Cross unfolding of a cube.
# print_config(SOLVED) returns:
#   00
#   00
# 11223344
# 11223344
#   55
#   55
def print_config(c):
    'Prints a configuration using a Latin Cross unfolding.'
    print('\n'.join(['  ' + c[:2], '  ' + c[2:4], \
        c[4:12], c[12:20], '  ' + c[20:22], '  ' + c[22:]]))

def spin_cw(config, side, turns):
    'Spins side #{side} of configuration {config} clockwise by #{turns} turns.'
    c = list(config)
    if side == 0:
        shift(c, [0,1,3,2], turns)
        shift(c, [11,10,9,8,7,6,5,4], 2 * turns)
    elif side == 1:
        shift(c, [4,5,13,12], turns)
        shift(c, [0,2,6,14,20,22,19,11], 2 * turns)
    elif side == 2:
        shift(c, [6,7,15,14], turns)
```

```
        shift(c, [2,3,8,16,21,20,13,5], 2 * turns)
    return ''.join(c)

def shift(A, ps, d):
    'Circularly shifts values in {A} at indices from {ps} by {d} positions'
    values = [A[p] for p in ps]
    for i, p in enumerate(ps):
        A[p] = values[(i - d) % len(ps)]

def neighbors(config):
    'Returns a generator of the neighbors of configuration {config}'
    for side in range(3):
        for turns in range(1, 4):
            c = spin_cw(config, side, turns)
            yield (side, turns, c)

def relate(c1, c2):
    'Returns the side and turns needed to transform {c1} into {c2} with one move'
    for (side, turns, c) in neighbors(c1):
        if c == c2:
            return (side, turns)
    return None

def explore_frontier(frontier, parent):
    'Explores {frontier}, adding new configs to {parent} and {new_frontier}'
    new_frontier = []
    for f in frontier:
        for (side, turns, c) in neighbors(f):
            if c not in parent:
                parent[c] = f
                new_frontier.append(c)
    return new_frontier

def path_to_config_from_parent_map(config, parent):
    'Returns a path of configurations from root of {parent} to {config}'
    path = [config]
    while path[-1] is not None:
        path.append(parent[path[-1]])
    path.pop()
    path.reverse()
    return path

def moves_from_path(path):
    'Given {path} of adjacent configurations, returns list of moves relating them'
    moves = []
    for i in range(len(path) - 1):
        moves.append(relate(path[i], path[i + 1]))
    return moves

def solve_BFS(config):
    'Solves {config} using BFS, with verbose output'
    print('Attempting to solve from configuration: ' + config)
    parent = {config: None}
    frontier = [config]
    while len(frontier) != 0:
        print('Exploring frontier containing # configs: ' + str(len(frontier)))
        frontier = explore_frontier(frontier, parent)
    print('Explored a total of ' + str(len(parent)) + ' configurations!')
    if SOLVED in parent:
        print('Path to solved state found!')
        path = path_to_config_from_parent_map(SOLVED, parent)
        return moves_from_path(path)
    else:
        print('Path to solved state not found... :(')
```

```
        return None

def solve_faster(config):
    'Solve {config} faster!'
    ############################
    #  Implement me! Part (e)  #
    ############################
    return None
```

Solution:

```
def solve_faster(config):
    'Solve {config} faster!'
    def check(frontier, parent):
        for f in frontier:
            if f in parent:
                return f
        return None
    parent_s = {SOLVED: None}
    frontier_s = [SOLVED]
    parent_c = {config: None}
    frontier_c = [config]
    middle = check(frontier_c, parent_s)
    while middle is None:
        frontier_s = explore_frontier(frontier_s, parent_s)
        middle = check(frontier_s, parent_c)
        if middle is not None:
            break
        frontier_c = explore_frontier(frontier_c, parent_c)
        middle = check(frontier_c, parent_s)
    if middle is not None:
        path_s = path_to_config_from_parent_map(middle, parent_s)
        path_s.reverse()
        path_c = path_to_config_from_parent_map(middle, parent_c)
        path_c.pop()
        return moves_from_path(path_c + path_s)
    return None
```