Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — October 17, 2017
Instructors: Jason Ku, Muriel Medard, and Silvio Micali — Problem Set 5

Problem Set 5

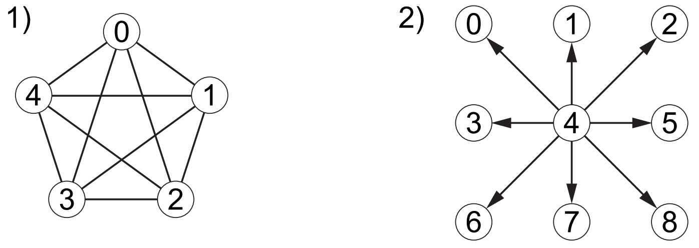
All parts are due on October 24, 2017 at 11:59PM. Please write your solutions in the LaTeX and Python templates provided. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct. Solutions should be submitted on the course website, and any code 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],        (2) = [[3,4,6],
       [1 0 0 1 0],               [0,4,5],
       [1 0 0 0 0],               [1,2,6],
       [0 0 1 0 1],               [0,5],
       [0 0 1 0 0]]               [],
                                  [2],
                                  [1,3]]
```

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



(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?

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.

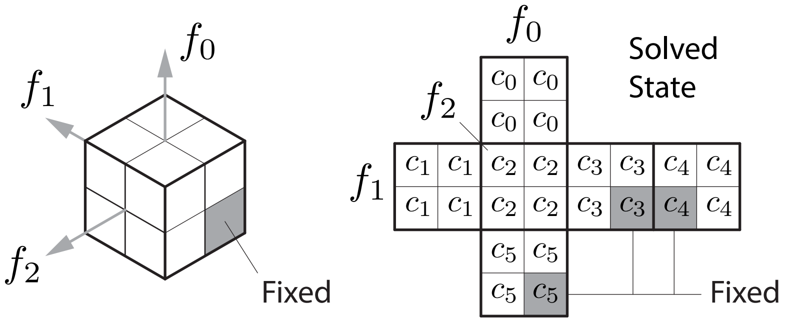
(a) [5 points] **Short Flights:** Jon hates traveling on long flights with Mr. G. He prefers *short* flights that are under 1 hour. Describe a linear time algorithm to find a flight itinerary with the fewest flights, all of them being short, or to determine that no such itinerary exists.

(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.

(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 $O(d^3 a)$ 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.

(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.

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).

(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?

(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)?

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

2. What is the *diameter* of the Pocket Cube graph, the longest distance between any two configurations?
3. How many edges are in the Pocket Cube graph?

(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.

(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.

```
# 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
```