Problem Session 5

Problem 5-1. Graph Radius

In any undirected graph $G = (V, E)$, the **eccentricity** $\epsilon(u)$ of a vertex $u \in V$ is the shortest distance to its farthest vertex v, i.e., $\epsilon(u) = \max\{\delta(u, v) \mid v \in V\}$. The **radius** $R(G)$ of an undirected graph $G = (V, E)$ is the smallest eccentricity of any vertex, i.e., $R(G) = \min\{\epsilon(u) \mid u \in V\}$.

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

(b) Given connected undirected graph G, describe an $O(|E|)$-time algorithm to determine an upper bound R^* on the radius of G, such that $R(G) \leq R^* \leq 2R(G)$.

Problem 5-2. Internet Investigation

MIT has heard complaints regarding the speed of their WiFi network. The network consists of r routers, some of which are marked as **entry points** which are connected to the rest of the internet. Some pairs of routers are directly connected to each other via bidirectional wires. Each wire w_i between two routers has a known length ℓ_i measured in a positive integer number of feet. The **latency** of a router in the network is proportional to the minimum feet of wire a signal from the router must pass through to reach an entry point. Assume the latency of every router is finite and there is at most $100r$ feet of wire in the entire network. Given a schematic of the network depicting all routers and the lengths of all wires, describe an $O(r)$-time algorithm to determine the sum total latency, summed over all routers in the network.

Problem 5-3. Quadwizard Quest

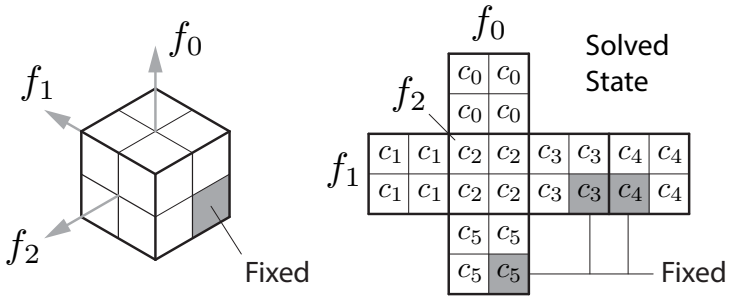
Wizard Potry Harter and her three wizard friends have been tasked with searching every room of a Labyrinth for magical artifacts. The Labyrinth consists of n rooms, where each room has at most four doors leading to other rooms. Assume all doors begin closed and every room in the Labyrinth is reachable from a specified entry room by traversing doors between rooms. Some doors are protected by evil enchantments that must be **disenchanted** before they can be opened; but all other doors may be opened freely. Given a map of the Labyrinth marking each door as enchanted or not, describe an $O(n)$-time algorithm to determine the minimum number of doors that must be disenchanted in order to visit every room of the Labyrinth, beginning from the entry room.

Problem 5-4. Purity Atlantic

Brichard Ranson is the founder of Purity Atlantic, an international tour company that specializes in planning luxury honeymoon getaways for newlywed couples. To book a customized tour, a couple submits their home city, and the names of three touring cities they would like to visit during their honeymoon. Then Purity will arrange all accommodations, including a **flight itinerary**: a sequence of flights from their home to each touring city (in any order), then returning back to their home. Unfortunately, it's not always possible to fly directly between any two cities, so multiple flights may be required. While cost and time are not a factor, couples prefer to minimize the number of direct flights they will have to take during their honeymoon. Given a list of c cities and a list of all f available direct flights, where each direct flight is specified by an ordered pair of cities (origin, destination), describe an efficient algorithm to determine a flight itinerary for a given couple that minimizes the number of direct flights they will have to take.

Problem 5-5. Pocket Cube

A Pocket Cube[1] is a smaller $2 \times 2 \times 2$ variant of the traditional $3 \times 3 \times 3$ Rubik's cube, consisting of eight corner cubes, each with a different color on its three visible faces. The **solved** configuration is when each 2×2 face of the Pocket Cube is monochromatic. We reference each color c_i with an index $i \in \{0, \ldots, 5\}$. Without loss of generality, we fix the position and orientation of one of the corner cubes and only allow single-turn rotations about the normals of the three faces of the Pocket Cube $\{f_0, f_1, f_2\}$ that do not contain the fixed corner cube; specifically, a **move** is described by tuple (j, s) corresponding to a single-turn rotation of face f_j, clockwise when $s = 1$ and counterclockwise when $s = -1$. Breadth-first search can be used to solve puzzles like the Pocket Cube by searching a graph whose vertices are possible configurations of the puzzle, with an edge between two configurations if one can be reached from the other via a single move. Instead of storing these adjacencies explicitly, one can compute the neighbors of a given configuration by applying all possible single moves to the configuration.



(a) Argue that the number of distinct configurations of a Pocket Cube is less than 12 million (try to get as tight a bound as you can using combinatorics).

(b) State the max and min degree of any vertex in the Pocket Cube graph.

(c) In your problem set template is code that fully explores the Pocket Cube graph from a given configuration using breadth-first search, and then returns a sequence of moves that solves the Pocket Cube (assuming the solved configuration is reachable). However, this solver is very slow[2]. Run the code provided and state the number of configurations the search explores. How does this number compare to your upper bound from part (a)?

(d) State the **max number of moves** w needed to solve any solvable Pocket Cube.

(e) Let N_i be the number of Pocket Cube configurations reachable within i moves of the a particular configuration. The code provided visits N_w configurations (which is larger than 3 million). Describe an algorithm to find a shortest sequence of moves to solve any Pocket Cube configuration (or return no such sequence exists) that visits no more than $2N_{\lceil w/2 \rceil}$ configurations (which is less than 90 thousand).

(f) Rewrite the `solve(config)` function in the template code provided, based on your algorithm from part (e).

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

[2] Please note that the code requires a couple minutes and considerable memory (over 400 Mb) to complete.

```
# ---------------------------------- #
# REWRITE SOLVE IMPLEMENTING PART (e) #
# ---------------------------------- #
def solve(config):
    # Return a sequence of moves to solve config, or None if not possible

    # Fully explore graph using BFS
    parent, frontier = {config: None}, [config]
    while len(frontier) != 0:
        frontier = explore_frontier(frontier, parent, True)
    print('Searched %s reachable configurations' % len(parent))

    # Check whether solved state visited and reconstruct path
    if SOLVED in parent:
        path = path_to_config(SOLVED, parent)
        return moves_from_path(path)
    return None

# ------------------------------------- #
# READ, BUT DO NOT MODIFY CODE BELOW HERE #
# ------------------------------------- #
# Pocket Cube configurations are represented by length 24 strings
# Each character repesents the color of a small cube face
# Faces are layed out in reading order of a Latin cross unfolding of the cube

SOLVED = '000011223344112233445555'

def config_str(config):
    # Return config string representation as a Latin cross unfolding
    return """
             %s%s
             %s%s
          %s%s%s%s%s%s%s%s
          %s%s%s%s%s%s%s%s
             %s%s
             %s%s
          """ % tuple(config)

def shift(A, d, ps):
    # Circularly shift values at indices ps in list A by d positions
    values = [A[p] for p in ps]
    k = len(ps)
    for i in range(k):
        A[ps[i]] = values[(i - d) % k]

def rotate(config, face, sgn):
    # Returns new config by rotating input face of input config
    # Rotation is clockwise if sgn == 1, counterclockwise if sgn == -1
    assert face in (0, 1, 2)
    assert sgn in (-1, 1)
    if face is None:    return config
    new_config = list(config)
    if face == 0:
        shift(new_config, 1*sgn, [0,1,3,2])
        shift(new_config, 2*sgn, [11,10,9,8,7,6,5,4])
```

```
    elif face == 1:
        shift(new_config, 1*sgn, [4,5,13,12])
        shift(new_config, 2*sgn, [0,2,6,14,20,22,19,11])
    elif face == 2:
        shift(new_config, 1*sgn, [6,7,15,14])
        shift(new_config, 2*sgn, [2,3,8,16,21,20,13,5])
    return ''.join(new_config)

def neighbors(config):
    # Return neighbors of config
    ns = []
    for face in (0, 1, 2):
        for sgn in (-1, 1):
            ns.append(rotate(config, face, sgn))
    return ns

def explore_frontier(frontier, parent, verbose = False):
    # Explore frontier, adding new configs to parent and new_frontier
    # Prints size of frontier if verbose is True
    if verbose:
        print('Exploring next frontier containing # configs: %s' % len(frontier))
    new_frontier = []
    for f in frontier:
        for config in neighbors(f):
            if config not in parent:
                parent[config] = f
                new_frontier.append(config)
    return new_frontier

def path_to_config(config, parent):
    # Return path of configurations from root of parent tree 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 configurations, return list of moves relating them
    # Returns None if any adjacent configs on path are not related by a move
    moves = []
    for i in range(1, len(path)):
        move = None
        for face in (0, 1, 2):
            for sgn in (-1, 1):
                if rotate(path[i - 1], face, sgn) == path[i]:
                    move = (face, sgn)
                    moves.append(move)
        if move is None:
            return None
    return moves
```

```
111 def path_from_moves(config, moves):
112     # Return the path of configurations from input config applying input moves
113     path = [config]
114     for move in moves:
115         face, sgn = move
116         config = rotate(config, face, sgn)
117         path.append(config)
118     return path
119 
120 def scramble(config, n):
121     # Returns new configuration by appling n random moves to config
122     from random import randint
123     for _ in range(n):
124         ns = neighbors(config)
125         i = randint(0, 2)
126         config = ns[i]
127     return config
128 
129 def check(config, moves, verbose = False):
130     # Checks whether applying moves to config results in the solved config
131     if verbose:
132         print('Making %s moves from starting configuration:' % len(moves))
133     path = path_from_moves(config, moves)
134     if verbose:
135         print(config_str(config))
136     for i in range(1, len(path)):
137         face, sgn = moves[i - 1]
138         direction = 'clockwise'
139         if sgn == -1:
140             direction = 'counterclockwise'
141         if verbose:
142             print('Rotating face %s %s:' % (face, direction))
143             print(config_str(path[i]))
144     return path[-1] == SOLVED
145 
146 def test(config):
147     print('Solving configuration:')
148     print(config_str(config))
149     moves = solve(config)
150     if moves is None:
151         print('Path to solved state not found... :(')
152         return
153     print('Path to solved state found!')
154     if check(config, moves):
155         print('Move sequence terminated at solved state!')
156     else:
157         print('Move sequence did not terminate at solved state... :(')
158 
159 if __name__ == '__main__':
160     config = scramble(SOLVED, 100)
161     test(config)
```

6.006 Introduction to Algorithms

Spring 2020