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.

Solution: Compute $R(G)$ directly: run breadth-first search from each node u and calculate $\epsilon(u)$ as the maximum of $\delta(u, v)$ for each $v \in V$. Then return the minimum of $\epsilon(u)$ 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.

(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 $R(G) \leq R^* \leq 2R(G)$.

Solution: Use breadth-first search to find $\epsilon(u)$ for any $u \in V$ in $O(|E|)$ time. We claim that $R(G) \leq \epsilon(u) \leq 2R(G)$, so we can choose $R^* = \epsilon(u)$. First, $\epsilon(u) \geq R(G)$ because $R(G)$ is the minimum $\epsilon(v)$ over all $v \in V$. Second, $2R(G) \geq \epsilon(u)$ because if a vertex x has eccentricity $R(G)$, we can construct a path from u to any other vertex v by concatenating a shortest path from u to x and a shortest path from x to v, both of which have length at most $R(G)$; thus $\epsilon(u)$ cannot be greater than $2R(G)$.

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.

Solution: Construct undirected unweighted graph G in the following way. Construct r vertices, one associated with each router in the network. Then, for each wire w_i connecting between routers a_i and b_i with length ℓ_i, add an unweighted chain of ℓ_i edges between the vertices associated with routers a_i and b_i. For each wire w_i, this process adds ℓ_i edges and $\ell_i - 1$ vertices to the graph. Now, if there were exactly one entry point, we could run breadth-first search from it to every other node in the graph, and the shortest path from the entry point to each router would be equal to the router's latency, by definition.

In order to search from all entry points at once, we add an additional node s (sometimes called a super node) to the graph with an edge to every entry point, and compute shortest paths from s using breadth-first search. Now, the shortest path from s to a router is one more than the latency at the router, so to return the total latency, we can sum the shortest path distance over all routers and subtract r. The graph G has at most $r + 100r + 1 = O(r)$ nodes and at most $100r + r = O(r)$ edges, so the graph takes $O(r)$ time to construct.

Breadth-first search runs in linear time in the size of the graph, so also takes $O(r)$ time. Lastly, computing the sum just requires looping over the $O(r)$ vertices and summing shortest paths to vertices that are routers. Thus the algorithm runs in $O(r)$ time.

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.

Solution: Construct a graph G with a vertex associated with each of the n rooms in the Labyrinth and an edge between two rooms if there is a door that is **not** enchanted connecting them. For any room in this graph, if the wizards can reach a room corresponding to vertex v, the wizards can search the rooms associated with every vertex in v's connected component without having to disenchant any door. Since every room is reachable from the entry room, it is suffices to identify the connected components of G, and then repeatedly disenchant doors that would connect two disconnected components. If there are k connected components of G, the wizards must disenchant $k - 1$ doors to visit every room. So, run either Full breadth-first search or Full depth-first search to count the number of connected components of G and return one less. G has n vertices and at most $4n$ edges, so counting the number of connected components in G will take at most $O(n)$ time for either algorithm.

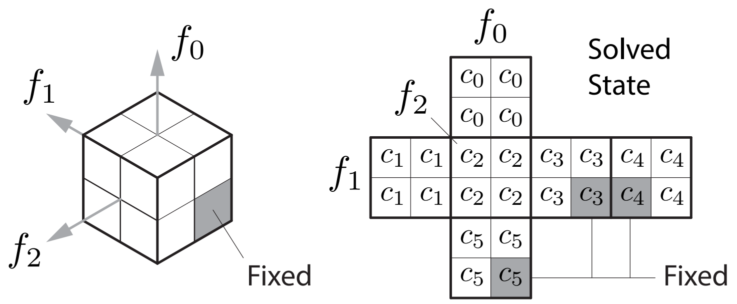
Problem 5-4. Purity Atlantic

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.

Solution: Let the directed **distance** from city a to city b be the fewest direct flights needed to reach b from a (or infinite if b is not reachable from a). Given the home city and three touring cities, if we could compute the $2\binom{4}{2} = 12 = O(1)$ pairwise directed distances between all pairs of the four input cities, then we could find an itinerary of that minimizes the number of direct flights starting at home, visiting the three cities, and returning in $O(1)$ time, by comparing the fewest flights needed for each of the $3! = 6 = O(1)$ permutations of touring cities, each of which can be computed in $O(1)$ time (if none yield a finite distance, then return that no itinerary is possible). To compute these pairwise directed distances, create a graph G with a vertex associated with each of the c cities and a directed edge for each of the f flights from the vertex associated with the origin to the vertex associated with the destination. Then, for each of the four input cities, breadth-first search to find the directed distance between it and the other three cities, storing parent pointers to remember a shortest path for each pair. Each of the four breadth-first searches takes $O(c + f)$ time, so the 12 pairwise distance can be all computed in $O(c + f)$ time, and a minimizing itinerary can be returned by concatonating the corresponding remembered shortest paths. Thus the entire algorithm also runs in $O(c + f)$ time. This algorithm is efficient because any correct algorithm must examining the entire graph.

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 $i \in \{0, \dots, 5\}$. 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 configruation 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).

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 $7!3^7 = 11022480$.

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

Solution: Three sides may be rotated, and each may be rotated clockwise or counterclockwise. So each configuration has exactly $3 \times 2 = 6$ configurations neighboring it.

(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)?

Solution: The provided BFS searches 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

[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.

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.

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

Solution: This can be found directly from the code output: the diameter is equal to the number of frontiers visited minus 1, i.e., 14.

(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).

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 whether 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 $\lceil w/2 \rceil$. Then each BFS visits at most $N_{\lceil w/2 \rceil}$ configuration, as desired.

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

Solution:

```python
def solve(config):
    # Return a sequence of moves to solve config, or None if not possible
    def check(frontier, parent):
        for f in frontier:
            if f in parent:
                return f
        return None
    parent_c, frontier_c = {config: None}, [config]
    parent_s, frontier_s = {SOLVED: None}, [SOLVED]
    middle = check(frontier_c, parent_s)
    while middle is None:
        frontier_c = explore_frontier(frontier_c, parent_c)
        middle = check(frontier_c, parent_s)
        if middle: break
        frontier_s = explore_frontier(frontier_s, parent_s)
        middle = check(frontier_s, parent_c)
    if middle:
        path_c = path_to_config(middle, parent_c)
        path_s = path_to_config(middle, parent_s)
        path_s.pop()
        path_s.reverse()
        return moves_from_path(path_c + path_s)
    return None
```