Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — Friday, October 11
Instructors: Jason Ku, Julian Shun, and Virginia Williams — Problem Set 5

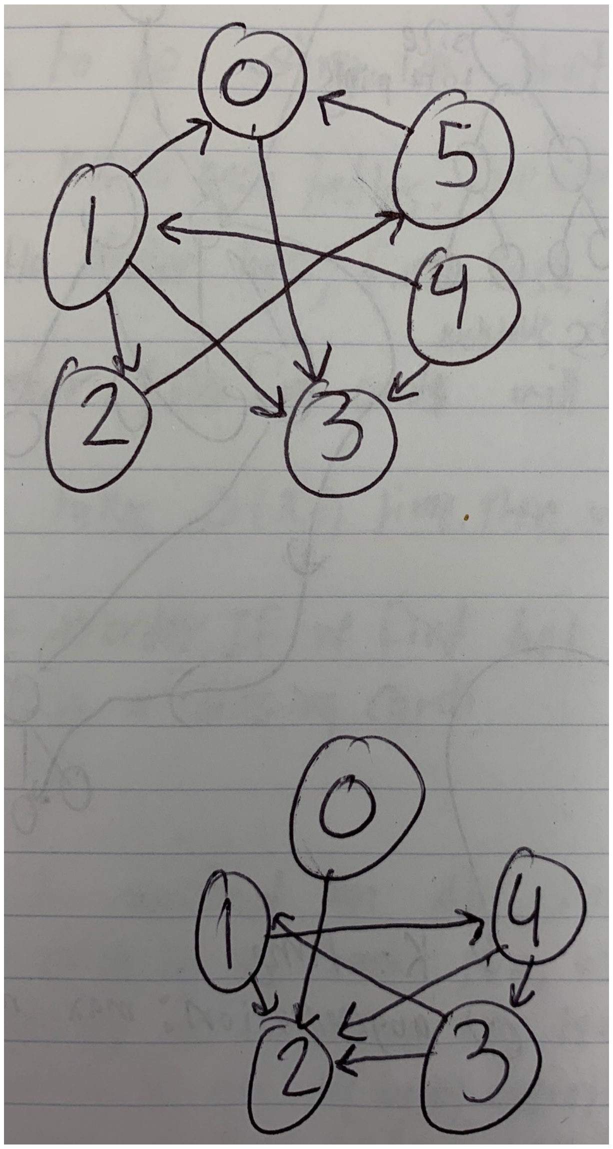
Problem Set 5

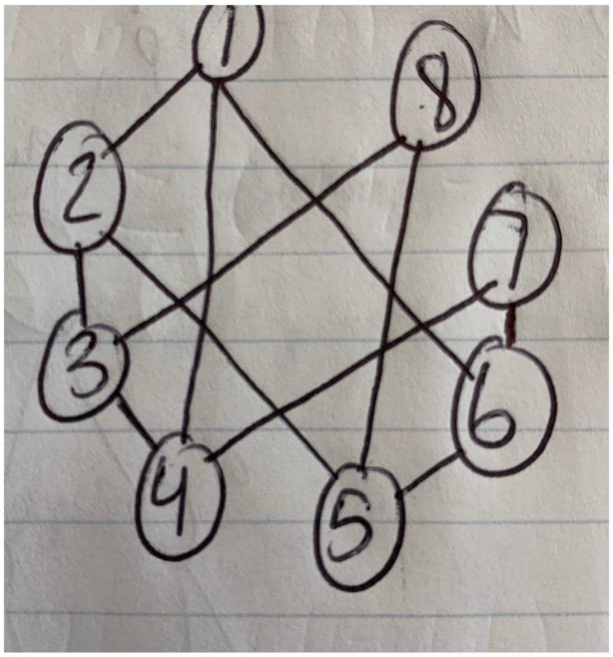
All parts are due Friday, October 18 at 6PM.

Name: Diego Escobedo

Collaborators: Mikael Nida, Noah Lee

Problem 5-1.

(a) 

(b) 

Direct Access Array: [None, $[2, 4, 6]$, $[1, 3, 5]$, $[2, 4, 8]$, $[1, 3, 7]$, $[2, 6, 8]$, $[1, 5, 7]$, $[4, 6]$, $[3, 5]$]

(c) DFS order visited: $[7, 4, 1, 2, 3, 8, 5, 6]$
BFS order visited: $[7, 4, 6, 1, 3, 5, 2, 8]$

Problem 5-2.

(a) For this problem, we are going to use BFS. We know that BFS, when run on a specific node, will allow us to find the Single Source Shortest Path from our node v to every other node u. From here, we can just simply find the the node u with the largest distance and store it. This distance is the shortest distance (guaranteed to be shortest because BFS guarantees SSSP) to the farthest vertex (because it has a maximal distance). Therefore, we can find the eccentricity for a single node in $O(|V| + |E|)$. We will repeat this process for the $|V|$ nodes, and simply find the smallest eccentricity to determine the radius. In terms of runtime, we are doing $|V|$ $O(|V| + |E|)$-time operations. However, we know that $|E| = O(|V|^2)$. Therefore, $|V| + |E| = O(|V|^2)$. So, if we run that $|V|$ times, our algorithm will be $O(|V|^3)$. However, since we said $|E| = O(|V|^2)$, we can substitute $|E|$ for $|V|^2$. Therefore, the runtime of our operation is $O(|V||E|)$.

(b) This problem is fairly simple. We can find an upper bound R^* for the graph by simply choosing a random point and considering its eccentricity. We know from above that calculating a single point's eccentricity is $O(|V| + |E|)$, but E dominates V so really it is $O(|E|)$. We know when we calculate the eccentricity that it is greater than or equal to the radius simply by the definition of what the radius is (the smallest eccentricity). However, we also need to prove that it less than 2 time R(G). Consider the worst-case scenario for the eccentricity of our randomly selected node: in the worst-case, the node is at one end of a chain, meaning that its eccentricity is equal to $|V|$. However, we know that there exists a node halfway along this long chain that has an eccentricity of $\frac{|V|}{2}$. We also know this node has the minimum eccentricity (and therefore $R(G) = \frac{|V|}{2}$, because any node that is not halfway long the chain will be closer to one end of the chain than the other, meaning its eccentricity is larger than $\frac{|V|}{2}$. Therefore, having an eccentricity of $|V|$ means you are at most, twice the radius, so the second inequality applies. Thus, we have proved using only an $O(|E|)$ calculation that a randomly selected point's eccentricity is greater than or equal to R(G) and less than or equal to 2*R(G).

Problem 5-3. For this problem, we are going to build a graph that simulates the MIT wifi router network. We are going to build this new graph by traversing every node in the schematic. What we are going to do is that if there is a connection of length l between node u and u v, we are going to build our graph with l-1 'ghost' nodes between u and v. Thus, we will have what is essentially an unweighted graph, since all the edges will be 1. As we build this graph, we also want to track the entry nodes and keep pointers to them. This algorithm will run in $O(r)$ because graph construction takes $O(|V| + |E|)$ time. The number of nodes is r, and as stated by the problem, the maximum number of edges is 100r. This means that the algorithm as a whole is 101r time, but 101 is a constant so the running time of constructing the graph is $O(r)$. Once we have this new graph, we are going to create a new node called 'great-grandpa'. This node will connect to all the entry nodes. We will then run BFS from the great-grandpa node. Our procedure will be that, once we reach a node, if it is not a ghost node and the node is located at level L, we are going to add L-1 to a running count of latency. We will add nothing if the node under consideration is a ghost node. This algorithm will work because we are simulating the weightedness of the graph with our ghost nodes and making sure that the 'level' is correct by using the ghost nodes to indicate distance from an entry node. The BFS runs in $O(|V| + |E|)$, which we proved is $O(r)$. Because we have two $O(r)$ sub-operations, the entire algorithm will run in $O(r)$.

Problem 5-4. Given a map of the labyrinth, we are going to create a graph G'(V,E). Each of the n rooms is going to be a vertex in V, and each non-enchanted connection between two rooms is going to be an edge in E (though we should still keep track of the enchanted edges).

Since we know that the maximum number of edges is 4 for each room, we know that the edges are a constant factor away from the number of nodes, meaning that we can run construct the graph and run BFS in O(n) time. This problem is essentially the same as connected components. What we are going to do is run BFS starting from what the problem calls the entry room. This is going to tell us every single room that is already reachable, without the enchantments. Then, we will pick a random unvisited vertex and start running BFS from it. We will repeat this process of picking a random unvisited node until we have reached all of the nodes. Therefore, the number of times we run BFS is going to equal the number of connected components in the graph. Because there is a constant number of enchanted edges that is less than n, we know that we will run BFS a finite number of times, and all our BFS operations will tehrefore run in O(n) time. Luckily for us, these disconnected components don't actually have 'no edges', they actually just have an enchanted edge. We also know that to connect k disconnected components, we have to create k-1 connections. Therefore, if we run BFS k times, the minimum number of enchanted doors we have to unlock is k-1.

Problem 5-5. We are going to create a graph G'(V,E).For each city c, we are going to create a vertex v_c. For each flight f connecting cities c_1 and c_2, we are going to create an edge f = $\{v_{c_1}, v_{c_2}\}$. Grpah construction will run in $O(|c| + |f|)$

We are going to analyze the 6 permutations of c_1, c_2, and c_3 to see which is best. We are going to run BFS from the start node, and find (without loss of generality) the shortest path from the start node to c_1. Then, we will run BFS to find the shortest path from c_1 to c_2, c_2 to c_3, and c_3 to the home node. Thus, we need to run BFS 4 times to determine the number of direct flights needed for that permutaiton of cities. If we do this for all 6 permutations, we can easily see which order of cities and itinerary is the best. So, we will run BFS a total of 24 times. Because 24 is a constant, and we are running two $O(|V| + |E|)$ operations (graph construction and BFS), the whole algorithm runs in $O(|c| + |f|)$.

Problem 5-6.

(a) In a Pocket Cube, each each sub-cube has a permutation for its position within the 8 cube arrangement, and also a permutation for the orientation of its faces. Since one of the cubes is fixed, the first cube can go into 7 different positions, the second cube into 6, etc. Therefore, there are $7!$ permutations on its location within the cube. We also know each cube can be oriented in three different ways, so that gives us 3^7 face orientation permutations. So, in total, there are $7! * 3^7 = 11,022,480$ distinct configurations.

(b) The max and min degree of this graph is going to be the same, because there are a limited number of moves but you can always make one of these moves. Because there are 3 movable faces, and we can move each of these faces either clockwise or counter-clockwise, there are 6 possible moves, and therefore the mindegree and maxdegree is 6.

(c) The search looks at $3,674,160$ different nodes of configurations, which is roughly a third of the upper bound we found in 6a.

(d) From a given starting point (node), the max number of moves needed to solve this graph is just the eccentricity of the node, because that's the optimal path to the node that's furthest away.Because this graph is perfectly symmetrical, and each node has the same 6 kinds of moves, then we know that the eccentricity of any vertex is the same (though the node that's the furthest away is not necessarily the solution). Thus, if we completely explore the graph, we could find out what an eccentricity for a node looks like. Luckily, that's exactly what our code in 6c did. We can see in the code that we explored 14'frontiers', or it took us 14 steps to traverse to the furthest nodes. Therefore, the eccentricity of that starting node, and any node, is 14. Therefore, in the worst case, it takes 14 steps to reach the max config.

(e) If we substitute in 6 for d and 14 for w, we get that we are allowed to visit $2 * 6^7 = 559,872$ configurations. To do this, what we can do is start two BFS from different points of the graph, and make sure each of them takes at most 7 steps (idea is that they meet in the middle). Thus, each BFS has nodes with degree 6, and each BFS would terminate after visiting 6^7 nodes, meaning the total nodes visited is going to be $2 * 6^7 = 559,872$. We should start our search at the given shuffled start node, and the other from the solved configuration. Since this is a connected graph, the two will collide at some point when they visit the same vertex. This works because of teh reasons explained above, where the maximum number of moves is 14, and in the worst case we need to make those 14 moves because our shuffled configuration's maximal distance node is the shuffled node. To make sure we get the shortest path, we just need to check at each step and break the loop if any collision is found. Thus, we know there has to be a collision after at most 7 moves.

(f) Submit your implementation to `alg.mit.edu`.