HW3 REPORT

Adil Kaan Akan

2171155

1 Proposed Algorithm & Pseudocode

My idea is thinking graph as directed graph and then use a shortest path algorithm to find optimal paths between start point and key, key and scientist, and scientist and bunny. First build the graph as an undirected graph, then make it directed according to rewarding vertices. Making directed graph part consist of the parts. In the first part, I build the directed adjacency matrix and adjacency list just making copies of the all edges. In other words, replace one edge between any two vertices with two edges directed to each other. Then, in the second part, by looking rewarding vertices, I change the weight of the edges that goes to some rewarding vertex by subtracting the reward value at that vertex. For example, if we have an edge E goes to the some vertex V with weight W and vertex V contains some reward R, then the edge E will have a weight W - R. By this way, we can consider the rewarding vertices as normal vertices. After making it directed, I use 3 shortest path. First is from start vertex which is 1 to the vertex that contains key, second is from the vertex that contains key to the vertex that contains scientist and third is from the vertex that contains scientist to the vertex that contains bunny. I used dijkstra algorithm to find shortest paths between these vertices. Dijkstra algorithm cannot calculate the graph with negative weight cycles, but when we visit a vertex with reward we update its weight the value before making it directed. Therefore, we will not have a problem about negative weight cycles. My dijkstra algorithm uses priority queue like the original one. The only difference is when we get the next vertex, we need to look that the vertex is locked or not at that time. I solve it by iteratively taking the next element from priority queue if we have locked vertex. For example, the priority queue contains vertices 5,8,3 at that time. We will get 5 at the beginning of the each cycle. If the vertex 5 is locked at that time, then my algorithm tries to find the next vertex which is not locked. My algorithm tries to get vertex 8 and look for whether or not vertex 8 is locked. I solve the odd-even lock problem by using this method. After choosing the right vertex, I check whether or not the vertex is end vertex that we look for. If so, I break the loop. Otherwise, I check the all vertices that is adjacent to vertex, and update the distances between them, update the parents of vertices, and finally put the updated vertices to the priority queue. The loop terminates when we get to the end vertex. After loop terminates, by using parents of vertices, I formed the visited array. At the end the shortest path algorithm returns the visited array, the distance between start and end vertices. By calling this shortest path algorithm 3 times for between start vertex 1 and vertex that contains key, between vertex that contains key and vertex that contains scientist, and between vertex

that contains scientist and vertex that contains bunny. At the end we get 3 distance between these vertices, and visited vertices list.

As a data structures, my algorithm keeps undirected adjacency matrix, directed adjacency matrix, directed adjacency list, vertices that are locked in odd times, vertices that are locked in even times, and vertices that contains reward.

2 Complexity Analysis

The complexity analysis of my algorithm as follows.

- Read the needed values from input file
- Make the graph directed graph
- 3 shortest path calls

2.1 Reading from input file

This process is bounded by number of edges. Since we will get first 6 line and then the number of edges and then all edges in each line. Therefore, this process is bounded by number of edges. $O(E)$.

2.2 Making graph directed

This process is bounded by maximum value of the number of vertices and the number of edges. First, adjacency matrix is updated, total of V operations. Then, adjacency list is updated, total of E operations since we look whether or not each edge goes to the vertex that contains reward. We make these updates the number of vertices that contains reward. However, the number of vertices that contains reward is bounded by 2. Therefore, we get the asymptotic complexity of $O(E + V)$.

2.3 3 shortest path calls

The normal dijkstra runs at $O(E.logV)$. However, we make some additional works. By checking whether or not vertex is locked at that time, we might look for all vertices at the worst case. Therefore, it gives a load of V. When we get a rewarding vertex, we need to update it, therefore it gives a load of max(E+V) since we update all edges. Since the work the algorithm done out of the loop is not important as the work made in loop, we can ignore the small term. This gives us a asymptotic complexity of $O(E.V^2.logV)$ at the worst case.

2.4 Space complexity

Since the algorithm stores adjacency list and adjacency matrix, we get the space complexity of $O(V^2)$ since the number of edges is also bounded by $O(V^2)$.