Full Name : Yavuz Selim Yesilyurt
Id Number : 2259166

Part 1 - Proposed Algorithm and Pseudocode

To accomplish this homework's task, I have basically created one fundamental and one helper function. In fundamental function (*saveBunny)*, I have used Dijkstra's idea to find shortest path from designated source to destination. Basically what *saveBunny* does is, it calculates the path with maximum amount ammo from given source node(room) to given destination node(room) according to given time informations timesteps. To achieve that, it uses Dijsktra's Shortest Path algorithm to find distances and extends it with the calculation of timeSteps and ammo nodes.

My submission contains *Graph* class and its methods to implement graph data structure as adjacency list. It maintains graph as a vector of vectors of pairs where pair's first element is "neighbour node number" and the second is "that edge's(vertice - neighbour) weight". It has a method for adding edge to the graph, it has a method for Printing the graph, I also have overloaded the "Indexing operator" ([]) for graph to access elements smoothly. There is also an *edgeComparetor* structure for handling of the comparison between edges of the graph.

The other code segments in *the3.cpp* consists of **saveBunny**, **choiceHandler** and **main** function of the homework. In **main** I have opened file with "Ifstream" and read whole input accordingly. Then I have called **choiceHandler** which calls **saveBunny** function inside.

For solution of the problem, I have used *choiceHandler* function to exhaust all the possibilities of ammo nodes (which can be 2 in number at most), to find the best optimized solution. In total there are 19 cases in which ammo nodes can be interchangeably used and not used. Since this function basically calls *saveBunny* function for 19 distinct cases, (rather than writing the whole thing) I will specify its pseudocode with writing the cases it handles.

In all of these cases, function calls *saveBunny* function for each part of path (i.e. from key to scientist, or from scientist to bunny) with decent time, ammo amount information and arranging which nodes to pass for that part of path, and at the end, it gathers all parts of a shortest path from initial position to bunny (with accomplishing the aims accordingly of course) and constructs the path and it stores the path with remaining ammo amount to a container for later usage. After exhausting all choices and doing the same for path calculation, function simply finds the path with maximum amount of ammo and Prints that. If in from any call of saveBunny function, it gets a value which is same as *failVector*, it just passes the remaining computation of that choice.

In *saveBunny* function (The function that I used as a backbone for path calculations) I simply used Dijkstra's shortest path algorithm with time constraints for a given source to a given destination. I have used a priority queue for keeping track of the nodes with edgeComparetor" (which I mentioned above) and I have used a distance container for keeping track of the weight and parent information, then till queue gets empty I have traversed all neighbours of all nodes with checking 2 conditions; "If node is visitable (not in

roomsToPass)" and "If node is not locked in next time Step". On my traversal, I have used the same logic as Dijkstra for updates on the path costs. Before function return, I constructed the shortest path If there is any and decreased the ammo amount accordingly and I returned the path. If there is not a path from src to dst I simply returned a container with value "-1". Below is the Pseudocodes for **saveBunny** and **choiceHandler** function:

```
function saveBunny(src, dst, time, ammoAmount, oddLockedNodes, evenLockedNodes, roomsToPass, graph)
        Initialize pQueue
        Initialize distVector with (INT_MAX,INT_MAX)
        Push ((src,0),time) to pQueue
        Set distVector[src - 1].first to 0
        Set finished to True
        while pQueue is not empty:
                Set destNode to pQueue.top().first.first
                Set timeStep to pQueue.top().second
                If destNode equals to dst:
                        Set finished True
                        Set time to timeStep
                        break
                Pop node from pQueue
                // Traverse all neighbours of destNode with an iterator it
                For  it = graph[destNode].begin() ; it != graph[destNode].end() ; ++it:
                        If (*it).first is not in roomsToPass:
                                If timeStep % 2 equals to 1 and (*it).first is not in evenLockedNodes or
                                  timeStep % 2 equals to 1 and (*it).first is not in oddLockedNodes:
                                        If distVector[(*it).first - 1].first > distVector[destNode - 1].first +
                                        (*it).second
                                            Set distVector[(*it).first - 1].first to distVector[destNode - 1].first +
                                            (*it).second
                                            Set distVector[(*it).first - 1].second to destNode
                                            Push (((*it).first, distVector[(*it).first - 1].first), timeStep + 1) to
                                            pQueue

        If finished is True
                Set dest to dst
                Initialize path with dst;
                For i = 0; dst != src; ++i:
                        Append distVector[dst - 1].second to path
                        Set dst to distVector[dst - 1].second
                Reverse path
                Decrement ammoAmount by distVector[dest - 1].first
                Return path
        Else
                Return {-1}

function choiceHandler(input parameters, graph)
        Case - 1  Find shortest path to bunny room without going to ammo room with saveBunny
        Case - 2  Find shortest path to bunny room with going to the first ammo room before going to the skull
        key's room with saveBunny
        Case - 3 Find shortest path to bunny room with going to the first ammo room before going to the
        scientist's room with saveBunny
        Case - 4  Find shortest path to bunny room with going to the first ammo room before going to the
        bunny's room with saveBunny
```

Case - 5 Find shortest path to bunny room with going to the second ammo room before going to the skull key's room with saveBunny
Case - 6 Find shortest path to bunny room with going to the second ammo room before going to the scientist's room with saveBunny
Case - 7 Find shortest path to bunny room with going to the second ammo room before going to the bunny's room with saveBunny
Case - 8 Find shortest path to bunny room with going to the first ammo room before going to the skull key's room and going to the second ammo room before going to the scientist's room with saveBunny
Case - 9 Find shortest path to bunny room with going to the second ammo room before going to the skull key's room and going to the first ammo room before going to the scientist's room with saveBunny
Case - 10 Find shortest path to bunny room with going to the first ammo room before going to the scientist's room and going to the second ammo room before going to the bunny's room with saveBunny
Case - 11 Find shortest path to bunny room with going to the second ammo room before going to the scientist's room and going to the first ammo room before going to the bunny's room with saveBunny
Case - 12 Find shortest path to bunny room with going to the first ammo room before going to the skull key's room and going to the second ammo room before going to the bunny's room with saveBunny
Case - 13 Find shortest path to bunny room with going to the second ammo room before going to the skull key's room and going to the first ammo room before going to the bunny's room with saveBunny
Case - 14 Find shortest path to bunny room with going to the first ammo room and then going to the second ammo room before going to the skull key's room with saveBunny
Case - 15 Find shortest path to bunny room with going to the second ammo room and then going to the first ammo room before going to the skull key's room with saveBunny
Case - 16 Find shortest path to bunny room with going to the first ammo room and then going to the second ammo room before going to the scientist's room with saveBunny
Case - 17 Find shortest path to bunny room with going to the second ammo room and then going to the first ammo room before going to the scientist's room with saveBunny
Case - 18 Find shortest path to bunny room with going to the first ammo room and then going to the second ammo room before going to the bunny's room with saveBunny
Case - 19 Find shortest path to bunny room with going to the second ammo room and then going to the first ammo room before going to the bunny's room with saveBunny

Part 2 - Complexity Analysis

For saveBunny:

- The implementation is done with adjacency list and priority queue.
- $O(V)$ operations for Initializing distVector with INT MAX.
- Statements in the inner loop are executed $O(V+E)$ times.
- *roomsToPass* size can only be at most 2 at a time, so searching it for each neighbour will make $V*2$ in the worst case.
- *oddLockedNodes* size can only be at most V-1 at a time, so searching it for each neighbor will make $V*(V-1)$ in the worst case.
- *evenLockedNodes* size can only be at most V-1 at a time, so searching it for each neighbor will make $V*(V-1)$ in the worst case.
- Construction of path can take V time in worst case.
- So in total, this function has a time complexity $O(V^2)$.

For choiceHandler:

- Time complexity of this function is basically nothing but *constant* * V^2, since it calls *constant* many times *saveBunny* function for distinct cases and does nothing in addition. So in total, this function has also a time complexity of $O(V^2)$