Team ID: G3T2
Team Members: NG JUN HONG ALEX, VIVIAN LAI WAN YIN

Algorithm Design

In essence, this algorithm attempts to traverse nodes and build a path to the destination based on highest node value per time unit [node value / (travel time to node + node time)] that can be obtained, while trying to maximise the time budget. This node value per time unit metric is used such that nodes with a higher metric can be considered first. Having a higher metric would mean that we are maximising the value we can obtain every time we travel to another node.

To track if a node has been visited, a 2D array is used, where row is the "from" node and column is the "to" node. E.g. After choosing node 4, node 5 is chosen – visited[4][5] = true. This way, we do not eliminate the possibility of arriving at node 5 from other nodes.

A final_path array will be created with the origin pushed in as the starting point. A while-loop is used to continuously seek for nodes until destination is found, or all nodes have been visited, whichever is earlier. Every iteration of this loop will attempt to select a node, if any, to be pushed into final_path.

In the while loop, all neighbours of the last node (last_node variable) pushed into final_path will be placed into a hash, with key being neighbouring node and value being the node's value per time unit. This hash will then be sorted according to values in descending order, where neighbours_values_arr will be obtained.

This neighbours_values_arr will then be iterated and a node will be chosen only if it is not visited and adding the node does not exceed the time_limit; all neighbours will be considered first before considering the destination node (if destination happens to be a neighbour of last_node). If a node is chosen, it will be pushed into final_path and marked as visited – this is so that we do not find ourselves taking the same paths over and over again. If no node can be chosen, the last node in final_path will be popped, and last_node will then be set to the current last node in final_path. This is done to backtrack and find another path to destination.

Once destination is finally found, the while loop will terminate. final_path will be returned if it does not exceed time limit, if it does, nil is returned.

Complexity Analysis

In the worst case, the while loop will be iterating through all nodes, therefore complexity here would be O(n). Within this while loop, we first iterate through each node's neighbours to calculate node value per time unit and place into a hash, with worst case being O(n - 1). **Current complexity: $O(n) * O(n) = O(n^2)$**

A sort is then performed on the resulting hash to obtain neighbours_values_arr, with worst case complexity of $O(n^2)$. **Current complexity: $O(n) * [O(n) + O(n^2)] = O(n^3)$**

The neighbours_values_arr will then be iterated through to find a suitable node, with worst case being traversing the whole array, therefore O(n). **Current complexity = $O(n) * [O(n) + O(n^2) + O(n)] = O(n^3)$**

Algorithm complexity: $O(n^3)$

Special Efforts

Compared to running a shortest path algorithm from origin to destination and then trying to perform a local search to put in more nodes, this algorithm is able to consider all nodes within a shorter execution time. Furthermore, the complexity of implementing local search will be raised due to the fixed origin and destination nodes.