Mr. Little Z's problem

Dijkstra's algorithm

Michail Livieratos

1 Understanding the problem

1.1 Code structure and implementation concept

The goal of this problem is to find a way to transverse across node 0 (Earth) to node n (Zearth), while choosing the safest path. In addition, once the safest path is chosen then the maximum distance is chosen from the safest path and hence producing the maximum distance within the safest path.

The problem is a classic application of Dijkstra's algorithm, however with some modification. It is worth going through a short review of Dijkstra's algorithm and then suggesting the modification.

```
1   Set C(x) = ∞ for all x ≠ x_I, and set C(x_I) = 0
2   Q.Insert(x_I)
3   while Q not empty do
4       x ← Q.GetFirst()
5       forall u ∈ U(x)
6           x' ← f(x, u)
7           if C(x) + l(x, u) < min{C(x'), C(x_G)} then
8               C(x') ← C(x) + l(x, u)
9               if x' ≠ x_G then
10                  Q.Insert(x')
```

Figure 1: A generalization of Dijkstra's algorithm, which upon termination produces an optimal plan (if one exists) for any prioritization of Q, as long as X is finite. Q is usually a priority queue or a linked list (or an array) , C is the cost value at given node or state, l(x,u) is the edge cost from x to u and x_G is the goal state

There are three main variables to define, that is the path weight, visited and previous

Variables	Definition
Path weight	maps each node to total weight of the path.
Previous	Maps from each node the previous one in along the shortest path
visited	A priority queue of all node in the graph, each nodes priority is defined by the path weight

Table 1: Key variables in the algorithm

A priority queue is an abstract data type that has an object and a key and the goal is to remove the object with the smallest key. This implementation can be done in two way either with an array or with an a min heap.

(i) Select a node from visited priority queue with the lowest path weight value (this is node S)

(ii) For each neighbouring node compare path weight to path weight+edge(S,x). Update path weight cost of x with the lowest value of the two as it represents the optimal path

(iii) Remove S from visited

(iv) Repeat all previous steps until visited is an empty array

The time complexity of the algorithm depends on how the priority queue is implemented. To elaborate, if an array is used then visited will be called O(n) times and it would take O(n) to find the appropriate key (hence O(n^2)).Furthermore, the path weight and previous will need to be updated at most once for every edge and hence O(e).

In the case of a min heap it will take O(log n) time for updating the keys of the priority queue (binary search for example). However this will happen for each node O(n) and hence O(n log n).In addition, for each edge the priority queue will be called to find the wait and hence O(e logv), resulting in O((v+e)log v).
In our case the best is the array since in our case the graph is always fully connected and hence $e = \frac{v(v-1)}{2}$ and hence 0(v^2)<0($v^2 log v$).

Finally the key thing to notice in Dijkstra's is how the path weight is updated (look at step (ii)), that is effectively comparing the local path from the immediate neighbours to the global path calculated.However, in our case the safest path is necessary and hence only the shortest path is necessary since a high weight value implies high risk. In other words, the algorithm has to greedily choose the lowest neighbouring weight as it is the most safe one for every node. In code this can happen by simple looking at the neighbouring nodes and picking the shortest one.

The overall run-time was implemented via an array since the graph will always be dense (you can always teleport, it is just more risky) and hence it is 0(v^2), in order to find the path and maximum distance of the path. In addition to that however the graph construction also takes 0(v^2) (note additional features where added for the user such as graph information that increase the run-time). Consequently the run-time is at 0(v^2)

2 Improvements

One way to improve upon the implementation is to implement the priority queue with a Fibonacci heap, this due to the fact that Fibonacci Heap needs O(1) time to find the smallest key instead of O(log n) or O(n). This leads to a complexity O(e + n Log n), for the case of a dense graph 0(v^2), but for sparse graphs O(n log n) and hence faster for sparse graphs.