**EcoDriving for Electric Vehicles**

Diogo Reis e Tiago Magalhães

Faculdade de Engenharia da Universidade do Porto

Email: {up201505472, up201607931}@fe.up.pt

**KEYWORDS**

Electric Cars, Environment friendly,

String matching, exact string matching, approximate string matching, Knuth-Morris-Pratt Algorithm, Dynamic programming matrix .

**ABSTRACT**

Our work seeks to create a trip planning system, that considers an electric car needs this being the existence of points to recharge the cars battery. In this project, we string matching algorithms , for both exact matching and approximate matching with the intent of filtering through the recharge points of a given network.

**INTRODUCTION**

This project attempts to address the issues of shortest path calculation in a weighted graph between two points or nodes in the graph network with an electric car taking into account into need to battery resupply. In our network a node represents a point in a map with x, y and z coordinates in the xyz plane, each node can also have a resupply point or not. Each arc or arrow represents a road that connects the two nodes. When a node has a higher elevation that the previous it will cost to more there as it will expend more battery, likewise if a point has a lower elevation that the previous it will cost less to move there as the regenerative brake system will kick in regenerating the cars battery. So, for this we use the expended battery as the weight of each edge taking into consideration that when going to a slope the weight should not be negative because there is still travel time which is still important for calculating an efficient route. For this work we use only directed graphs as a given road may or may not have lanes for traversal both ways.

**STUDY’S GOAL**

The objective of this project is to create a system through which we learn to better work the graph structure and its traversal methods.

**MAIN DEFINITIONS**

* **Node**
* **Edge**
* **Graph**
* **Path**
* **Section**

**Implementation**

1. **Input –** The input to this system will be the map or set of nodes and connection which we want to move along, as well as the nodes we want to move from to, indicated by their relative ID’s as show by the graphViewer software.
2. **Output –** The output will be the list of nodes in succession that when moved through well get us to the destination in the shortest amount of battery spent.

**Weight Function**

Given a set of two nodes i and j, where I and j are connected, the weight to traverse from i to j is given by Euclidean absolute distance formula, the we take that value and if j has a higher altitude than I then the weight is the distance multiplied by 1.5. If j’s altitude is lower than i’s then the weight is the distance – distance multiplied by 0.5, finally if the altitude is the same the weight is simply the distance.

W(i,j)=distance (if j altitude == i altitude)

Or

W(i,j)=distance x 1.5 (if j altitude > i altitude)

Or

W(i,j)=distance/2 (if j altitude < i altitude)

**Dijkstra’s Algorithm**

This algorithm allows us to find the shortest path from one node to all other nodes that are connected to it either directly or indirectly. To this effect, the algorithm first sets the distance from the origin to itself to 0 and to all other nodes to infinity or in this case INT\_MAX a macro that defines the maximum value possible of being represented by a 32-bit unsigned integer. The we mark all nodes that are not the origin point as unvisited and the origin as visited, and put all unvisited nodes into a set. Then for the current nodes all its unvisited neighbors have their weight to the current node calculated and if this value is lower than the previous value then substitute the previous with the new one, we do this for all the nodes in the unvisited set. Then we simply travel backwards using the nodes of smallest value that are connected to the end.

Pseudo Code:

1 **function** Dijkstra(*Graph*, *source*):

2

3 create vertex set Q

4

5 **for each** vertex *v* in *Graph*: *// Initialization*

6 dist[*v*] ← INFINITY *// Unknown distance from source to v*

7 prev[*v*] ← UNDEFINED *// Previous node in optimal path from source*

8 add *v* to *Q* *// All nodes initially in Q (unvisited nodes)*

9

10 dist[*source*] ← 0 *// Distance from source to source*

11

12 **while** *Q* is not empty:

13 *u* ← vertex in *Q* with min dist[u] *// Node with the least distance will be selected first*

14 remove *u* from *Q*

15

16 **for each** neighbor *v* of *u*: *// where v is still in Q.*

17 *alt* ← dist[*u*] + length(*u*, *v*)

18 **if** *alt* < dist[*v*]: *// A shorter path to v has been found*

19 dist[*v*] ← *alt*

20 prev[*v*] ← *u*

21

22 **return** dist[], prev[]

Dijkstra runs in square time, that scales with the amount of nodes in the graph O(||) , where V is the number of nodes in the graph.

Dijkstra’s Algorithm can be considered a perfect example of a problem solved using dynamic programming due to its heavy usage of values calculated in previous iterations.

One possible improvement to this algorithm, is processing only nodes that actually get us closer to the destination and not processing nodes that get us away from it, this type of heuristic is used in an algorithm that is considered a subset of Dijkstra by the name of A\*. A\* combines the advantages of Dijkstra and Best First Search to create an algorithm capable of adapting itself to a large array of situations, that can find the same result as Dijkstra but in much faster way, as it processes a lot less nodes.

**Breadth First Search**

This algorithm allows us to traverse a graph starting at a random point and reaching all nodes connected to it directly or indirectly, for this it simply starts at a point and for all the nodes connected to that point it goes to the nodes connected to those, until it has reached a dead end or traversed the entire graph. Pseudo Code:

Breadth-First-Search(Graph, root):

create empty set S

create empty queue Q

root.parent = NIL

add root to S

Q.enqueue(root)

**while** Q is not empty:

current = Q.dequeue()

**if** current is the goal:

**return** current

**for** each node n that is adjacent to current:

**if** n is not in S:

add n to S

n.parent = current

Q.enqueue(n)

The breadth first search algorithm run in linear time scaling with the number of nodes and edges in the graph.

O(|V|+|E|), where V is the number of nodes in the graph and E is the number edges in the graph. This algorithm can be used to determine if a graph is connected by running it on a regular graph and on a graph with its edges reversed, if for both cases the number returned by this algorithm is equal to the number of edges in the graph then the graph is strongly connected.

**Program Logic Flow**

We first start by parsing the whole dataset into a hash table where they key for each entry will be its node identifier.

Then we will add all those edges to the graph and the graph viewer software, after this we all in all the edges with their respective weight.

Then we select the nodes we want to obtain the path between and run the path calculation algorithm indicating on the console and on graph viewer the result. The we calculate where the graph is connected or not as described previously and this is the end of the logic flow.

**Conclusion and Future Work**

After this project, we concluded that this field of programming is one with many nuances and lots of different challenges to overcome , for one the search for an efficient path finding algorithm, currently as far as we are aware the best existing algorithms are all subsets of the A\* algorithm , and while A\* is much more efficient than its predecessor Dijkstra , it is still a brute force algorithm and is by nature not efficient, while working on this project we used two different subsets of data , one with 1050 nodes and about 900 ways , and the time it took to run that with Dijkstra was fairly short , but we also tried a set of over 130000 nodes and that took us over 4 minutes to run on a basic version of the whole program, and while Dijkstra’s algorithm wasn’t the only culprit for that , it still took a fair amount of time to run.

Overall one of the greatest challenges we were able to identify in this study area was the overall lack of efficiency in terms of the existing algorithms, especially when compared to other study fields in computer science. One definite possibility for improvement is a larger use of concurrency, while this would be hard to implement it would surely create noticeable results, but it would not change the fact that these algorithms are fundamentally brute force algorithms.

The main difficulty we found was how to implement a way for Dijkstra to take into account the cars need to stop and recharge we ended up going with an implementation that simple called Dijkstra multiple times. However it must be said that this solution is currently not working due to a nullptr exception that we are not being able to track. Therefore this code segment is not commented but is disabled from working.

In the end this project was a big eye opener to some of the many challenges that existing in the very important field of study.

**References**

<https://en.wikipedia.org/wiki/Breadth-first_search>

<https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm>

**Contribution**

Diogo Reis – 50%

Tiago Magalhães – 50%