

SAT I - Electric Car Rally

By Alexander Xu

Milestone 1

The Problem

Features

Exceptions

ADTs Storing Data

Justification and Specifications

Visual Representation

Milestone 2

Possible Algorithmic Designs

Path-Finding Algorithms

Charge-Determining Algorithms

Full Algorithm

ADT Shortcomings

Milestone 3

Creating The Model

Algorithm Justification and Limitation

Low Level Pseudocode

High Level Pseudocode

Milestone 1

The Problem

The assignment requires a car rally to be designed. This includes modelling various aspects of the race such as roughness, lengths and speed limits of roads, classifications of and charger efficiencies at locations and car battery and consumption. The following assumptions are made:

- The length of each road between locations is between 30km - 300km.
- The speed limit on the road is between 60 km/hr - 100 km/hr.
- The charging station will have between 40 km - 100 km of range per hour of charging. (This is mostly followed but efficiency is measured by kW in this project. Hence due to the different car consumption rates sometimes the overall charge per hour will get the car just over 100km. Additionally, in this model, some locations do not have chargers at all.)
- Types of electrical cars (see <https://rac.com.au/car-motoring/info/small-electric-cars-australia>)

Each car will effectively be experiencing the same race but in a different order (The race is a cycle of all 14 locations where each of the 12 cars start at a random position and end back up in that same position). Hence, the race will be fair. This predetermined cycle order will be known to the event organisers but not the drivers. The problem further requires an algorithm to be designed that follows a greedy format, trying to optimise its path at each step. This is because this is how the drivers will experience the race. The algorithm will take in the cycle (represented by a directed graph) along with data about the locations, roads, car and starting location. It will then return the path it chose to take as if it was with the driver and only given the next node (e.g. only the next goal node, not the entire cycle). Both time to complete the rally and electric charge left in the car after the rally will be considered, so each driver will need to try their best to maximise these. In the end, the scoring will follow:

$$\text{score} = (10^k) * (B/B_{\max}) * (1/t)$$

Where k is some integer power (that makes the score differences much more obvious), B is the battery left at the end, B_{\max} is the maximum battery capacity and t is the time taken to complete the race in days.

Hence, the scoring system heavily rewards being on as high of a battery percentage as possible.

Features

Locations

The 14 locations are taken from South Australia and reflect the actual position and some features of the real-life place. Each location may also have a charging station where cars can stop to charge up.

Location classification

Locations will be classified as either urban, suburban or rural depending on how they are classified in real life.

Charger efficiency

Similarly, larger cities will have chargers with greater efficiencies, while smaller cities will have less efficient (or no) chargers. More efficient chargers will charge the car quicker.

Location position

The relative position of the location is useful when visualising how this rally would be in real life. Hence, they are recorded based on a screenshot of a map which contains all 14 locations. The relative coordinates are then used to position the nodes in the graph.

Location information table

Location	Classification	Charger Efficiency Index
Adelaide	Urban	3
Gawler	Suburban	2
Murray Bridge	Suburban	2
Victor Harbor	Rural	1
Port Pirie	Rural	1
Whyalla	Urban	2
Port Augusta	Urban	1
Kadina	Rural	2
Peterborough	Rural	0
Kimba	Rural	0
Mount Gambier	Urban	2
Renmark	Rural	0
Naracoorte	Rural	0
Kingston SE	Rural	0

Roads/routes

The 30 roads between locations are randomly picked to create various routes between locations. All routes are unique (e.g. there are not two roads between the same two locations) and work in both directions.

Road distance

The distances of routes will reflect that of routes that exist between the two locations in real life. These are all between 30 and 300km and are how far the car will need to travel to get to the connected location.

Road roughness

This will be an integer between 0 and 2 which is influenced by how big the cities that the road connects are. 0 means the road is very smooth and 2 means the road is very rough. Rougher roads mean that the car is less efficient. Thus, if the car is less efficient, it will use up more energy.

Road speed limit

Roads will have a speed limit between 60 and 100km/hr randomly assigned to it. It is assumed that the car will always be travelling at this exact speed limit when using the road.

Road information table

Location 1	Location 2	Distance (km)	Roughness Index	Speed Limit (km/h)
Adelaide	Gawler	44	0	100
Adelaide	Murray Bridge	76	0	90
Adelaide	Victor Harbor	83	0	70
Adelaide	Port Pirie	223	1	80
Adelaide	Renmark	257	1	100
Gawler	Kadina	132	1	80
Gawler	Port Pirie	198	1	70
Gawler	Murray Bridge	97	0	100
Murray Bridge	Victor Harbor	71	0	90
Murray Bridge	Renmark	204	1	70
Murray Bridge	Naracoorte	193	1	60
Port Pirie	Kadina	103	0	90
Port Pirie	Whyalla	91	0	60
Port Pirie	Port Augusta	93	0	100
Port Pirie	Peterborough	114	1	80
Port Augusta	Whyalla	76	0	70
Port Augusta	Kimba	156	1	100
Port Augusta	Renmark	295	2	90
Whyalla	Kimba	130	1	80
Whyalla	Peterborough	227	1	100
Mount Gambier	Naracoorte	98	1	70
Mount Gambier	Kingston SE	163	1	90
Renmark	Naracoorte	289	2	60
Renmark	Kingston SE	270	2	100

Kingston SE	Victor Harbor	238	2	70
Kingston SE	Murray Bridge	186	1	90
Peterborough	Kadina	190	1	80
Peterborough	Renmark	265	1	100

Car model

This is the model of the electric car. Each car is different, particularly in battery capacity and energy consumption.

Car battery

The total energy capacity of the electric car is recorded in kWh.

Car consumption

This describes the energy efficiency of the electric vehicle in kWh/100km.

Car information table

Car Model	Battery (kWh)	Consumption (kWh/100km)
GWM Ora	48	15
GWM Ora High Battery	63	15
BYD Dolphin	44.9	15
BYD Dolphin High Battery	60.48	15
MG4	64	13
MG4 HB	77	15.2
Nissan Leaf	40	16.6
Nissan Leaf High Battery	62	18.5
Fiat 500e	42	14.4
Mini Cooper E Electric	41	14.3
Mini Cooper E Electric High Battery	54	14.7
Cupra Born	82	17

Exceptions

Due to time restraints, some features are simplified or excluded. For example, location classification and roughness of road are only classified into 3 categories when in reality these are much more detailed. Additionally, things such as the speed limit and roughness of road do not exactly reflect the real-life situation. Due to limitations present, it will be assumed that the race planner has decided to set these features randomly themselves (e.g. they have chosen roads with random roughness between locations and set them to be the valid routes for the rally). Furthermore, road inclines are not considered, and the cities are simply thought of as a position rather than an entire area.

ADTs Storing Data

- **Circuit (graph)**
 - Node: Location Name [String]
 - Edge: Road Connections (Custom Immutable Set)
 - o Element: Location 1 [String]
 - o Element: Location 2 [String]
- **Location Information (Dictionary)**
 - Key: Location Name [String]
 - Value: Charger Efficiency Integer [Integer]
- **Road Information (Dictionary)**
 - Key: Road Connections (Custom Immutable Set)
 - o Element: Location 1 [String]
 - o Element: Location 2 [String]
 - Value: Road Features (Array Length 3)
 - o Index 0: Distance [Integer]
 - o Index 1: Roughness Integer [Integer]
 - o Index 2: Speed Limit [Integer]
- **Car Information (Dictionary)**
 - Key: Car Model [String]
 - Value: Car Features (Array Length 2)
 - o Index 0: Car Battery [Integer]
 - o Index 1: Car Consumption [Integer]
- **Predetermined Cycle (Graph)**
 - Nodes: Location Name [String]
 - Edges (directed): Next Location Connection (Array Length 2)
 - o Element: Location (From) [String]
 - o Element: Location (To) [String]

Justification and Specifications

The circuit will be stored as a graph because it allows relationships between locations and roads to be easily accessed and represented (e.g. neighbours or end_points). In this context, using a graph is superior to using other ADTs such as dictionaries as these would either need to store roads and locations separately, store them in a messy way or require additional logic which the graph ADT already implements. Therefore, using a graph will simplify the implementation of the algorithm.

For the features of the various locations, a dictionary is best. This is because it allows direct relation from the location name to its feature. There are also no issues with unique keys as all the

locations of the circuit all have different names. The value of the dictionary is the charger efficiency index as this is the only feature the algorithm needs to access. Other ADTs can be used in the place of this dictionary; however, there will likely need to be multiple of these ADTs combined to function the same way. Hence, by using a dictionary, the charger efficiency is easily accessed through the location name.

To store the road information, a dictionary of Custom Immutable Sets (keys) and arrays (values) is used. The array is length 3 and the key is a Custom Immutable Set, a custom ADT which functions much like the set, but where instead of being able to add and remove elements, it is created using a set and cannot be changed after. This allows it to be the key of the dictionary as it cannot be changed. Furthermore, the Custom Immutable ADT was chosen to be a set as the locations that the road connects have no order, hence the ADT does not need to be ordered. A Custom Immutable Array would work just the same, it would just be a bit messier to use as it would have the locations ordered, meaning that every time information about the features of the road needs to be extracted, the specific order of the two locations in the Custom Immutable Array would need to be the same as the dictionary key. As with the dictionary and array combination, the justification of the location dictionary and array combination applies.

For the car information dictionary, a dictionary where the values are arrays of length 2 was used. Again, this allows direct association between the car model and its features and since the indexing of the features is known, getting this information is easy to implement. Whilst a dictionary instead of the inner array offers a slightly more direct correlation between the feature names and the actual feature rating/value, this would require the same keys for the features for every location, which is completely unnecessary and messy. Instead, indexing in this case is better as each index is already known to store a specific rating/value. Hence whatever needs to be accessed can be easily (If readability needed to be improved, a different dictionary could be added to go from the feature name (key) into the array index (key) which would then be used in the array).

For the predetermined cycle, a graph was chosen because it allows for a cycle to occur, meaning that regardless of where you start from, there is no start or end of the ADT. All the while, direction is also maintained and the graph is extremely easy to traverse. If another ADT like the queue was used, there would be both a start and an end, making it much more confusing to go through the cycle of locations that need to be visited in order.

Specifications of ADTs (from above):

- Graph:

create	: \rightarrow graph
add_node	: graph x element \rightarrow graph
add_edge	: graph x element x element \rightarrow graph
remove_node	: graph x element \rightarrow graph
remove_edge	: graph x element \rightarrow graph
adjacent	: graph x element x element \rightarrow boolean
neighbours	: graph x element \rightarrow set
nodes	: graph \rightarrow set
edges	: graph \rightarrow set
end_points	: element \rightarrow array

- Dictionary:

create	: \rightarrow dictionary
has_key	: dictionary x element \rightarrow boolean
add	: dictionary x element x element \rightarrow dictionary
update	: dictionary x element x element \rightarrow dictionary

- | | | |
|---|-----------------------|--|
| | remove | : dictionary x element → dictionary |
| | get_value | : dictionary x element → element |
| | get_keys | : dictionary → set |
| - | Array: | |
| | create | : integer → array |
| | set_value | : array x integer x element → array |
| | get | : array x integer → element |
| | length | : array → integer |
| - | Custom Immutable Set: | |
| | create | : set → Custom Immutable Set |
| | size | : Custom Immutable Set → integer |
| | contains | : Custom Immutable Set x element → boolean |

Visual Representation

Figure 1: Graph Representation of Rally Circuit (actually used in algorithm)

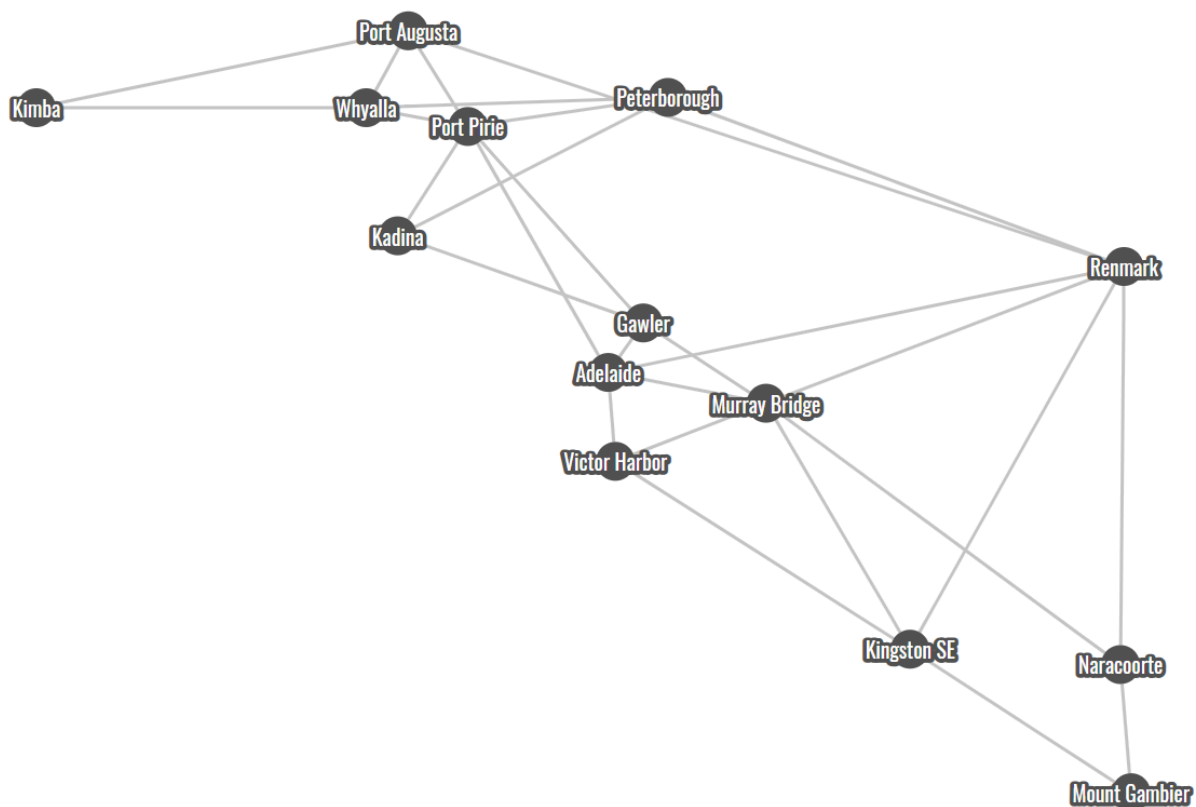







Figure 2: Visual Graph Representation of Rally Circuit (Includes more information visually)

Legend

- Medium: Regular road
- Thick: Rough road

Edge Colour:

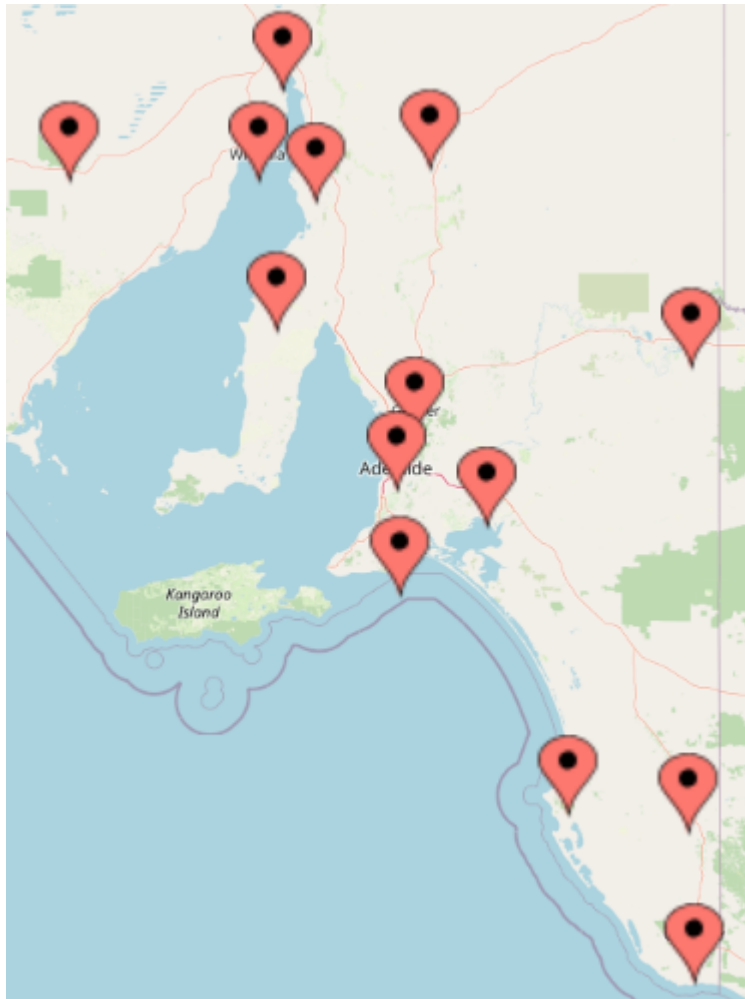
Edge Colour	Speed Limit (km/hr)
	100
	90
	80
	70
	60

Node Size:

- Large: Urban
- Medium: Suburban
- Small: Rural

Node Positions:

- Reflects real life (map of the actual positions below for comparison)



Milestone 2

Possible Algorithmic Designs

Path-Finding Algorithms

Bellman-Ford Algorithm:

Given the track and a starting node, this algorithm can find the minimum distance from the starting node to every other node in the graph. While it could be run at every step of the race to find the shortest path between the current node and the next goal node, all computations which are not related to the shortest path between the starting node and specifically the goal node will be completely unnecessary. Hence, this algorithm is unsuitable for the problem.

A* Search Algorithm:

The a* algorithm can be used to find the shortest path between any two nodes in a weighted graph. Hence, this can be run for every single step of the race, after the electric car has reached the goal node and is given its next goal node. While this algorithm is faster and more time efficient, it is not necessarily always optimal. In this context, finding the optimal path from one

node to the next is far more important than the time complexity of the algorithm. Hence, this algorithm is not the best and will not be used.

Floyd-Warshall Algorithm:

The Floyd-Warshall Algorithm can be used to generate all the lowest weighted paths between two nodes. This algorithm could be slightly altered to also keep track of the order of these lowest weighted paths (so that they can be taken). From here, the information can be stored and referred to when attempting to find the shortest path between the current node and the next goal node in the race. However, this finds all the shortest paths between all possible pairs of nodes, and, because this is not needed, there is a lot of unnecessary computation.

Nevertheless, Floyd-Warshall has an advantage: it allows comparison of all the shortest distances between all pairs of nodes. In some cases, such as in the final iteration of this algorithm, this advantage can be useful. Nevertheless, there are still much better ways of achieving what needs to be done in this final iteration (find the shortest edge that is connected to a given node). Hence, this algorithm will not be used in this section or the main body of the designed algorithm.

Dijkstra's Algorithm:

Dijkstra's algorithm can find the optimal path between any two nodes. This allows it to be run at every step of the graph (from current node to next goal node). As the algorithm follows a greedy format and also only finds the shortest path between the two nodes it is given, it is more efficient than the other options while being optimal.

"Shortest Road To" Algorithm:

This algorithm takes in a target node and finds the shortest edge connected to that node. It does this by iterating through every edge that is connected to the target node and storing the shortest edge up to that iteration as a temporary shortest edge. Once every one of these edges has been through this, the temporary shortest edge is the shortest edge connected to the target node.

While this algorithm isn't designed for the main body of the algorithm, in this circumstance, it has a particular use as the car needs to be as high battery as possible when it completes the race. Hence, this algorithm can be used in the last iteration of determining the path to ensure the final road taken is the shortest one and will therefore be used in the final part of the algorithm.

Charge-Determining Algorithms

"Always Charge" Algorithm:

As the scoring system values being on as high of battery as possible at the end of the race, this algorithm makes the car charge at every possible charging station to full battery. While this ensures the car will be at the highest possible charge, it does not consider any other details such as the efficiency of the charger, details of the car or roughness/length of the road whatsoever. Therefore, it wastes time choosing to charge at slower chargers when a faster charger is on route and able to be reached. Though this design is easy to understand and implement, it is not at all optimal and hence will not be chosen.

“Always Minimum Charge” Algorithm:

The “Always Minimum Charge” algorithm will only charge the minimum amount needed to get to the next location in the path given by the path finding algorithm. Hence, it will save time by not stopping for more than it needs to at each charger. This design has many flaws, however, for example it will always be on 0 charge, even at the end of the race, dramatically affecting the final score. Additionally, it will also not take advantage of good chargers by charging as much as possible at these and instead will have to charge at bad chargers when it gets to them. This algorithm would be easy to implement; however, it is not at all optimal. Hence it will not be used.

“Best Charge” Algorithm:

This algorithm will use the path deemed as optimal. It will always charge up to full battery at level 3 chargers and will charge only the bare minimum for all other chargers to get to the next better or equal charger. If there are no better or equal chargers in front of it before the end of the race, it will charge to max and continue to the next repeating the algorithm until it reaches the final location. Hence, it will return the optimal charging times at each location.

Full Algorithm (Almost... Optimal Score)

Altered Dijkstra’s Algorithm + “Best Charge” Algorithm + “Shortest Road To” Algorithm (Chosen):

In this algorithm, the race will first be split into 14 segments - one for each pair of locations (current node and goal node). First an altered version of Dijkstra’s will be run for each step of the race to find an almost optimal path. Within each of these segments, it will alternate between taking 2 roads and claiming a location. When it is on the turn of taking roads, it will look at all claimed nodes and choose the best road connected to any of these nodes by running Dijkstra’s on the roads, where it considers the time it takes to traverse the road by using its roughness and distance. If there is no other road that can be taken at the time due to a lack of edges connected to claimed nodes, it will simply skip the turn. On the claim node turn, it will claim the node with the most efficient charger. It will continue this process until the goal node is reached (the goal node will have a temporary infinity value, so it will be claimed immediately).

The “Best Charge” algorithm will then be used to determine when and how much to charge. Given the sub-path, it will find when and how much to charge to optimize the score by the end of the sub-path. Together, these will create an almost optimal solution for the partial-path. This will be repeated for all 14 pairs of locations except for the last one.

Because the ending battery is so highly valued in the scoring system, in order to optimise score, this final battery must be optimised. Hence, a different approach is taken for this final step. Instead of simply using the altered Dijkstra’s, “Shortest Road To” algorithm will first be used to determine the shortest distance road between a given location and any other location. To do this, the algorithm will iterate through every edge connected to the given node and keep track of the minimum. At the end, the road with the shortest distance will be returned as a new temporary goal node. After this, the altered Dijkstra’s will be run from the original current location to the new temporary goal location to find a relatively optimal path. The final pair of locations obtained from running the “Shortest Road To” algorithm will be added to this path and then the “Best Charge” algorithm will be run to optimise charging.

At the end, the score as well as order of locations traversed will be returned.

Note: The algorithm is done in this way so that it is given the information the driver is given when it is given. Hence, the charging is not optimal. Additionally, as this algorithm has a high affinity for efficient chargers, it does not really consider situations where going through locations with worse chargers would lead to a shorter path in terms of score. It is very difficult to create an algorithm that perfectly finds every road that needs to be taken and the exact amount of time to charge at each location to create the shortest path in terms of time while ending on as high of battery as possible. Hence, due to the time restriction, a different algorithm will be created to give an almost optimal solution.

Alternate Algorithmic Design - Dijkstra's + "Best Charge" Algorithm:

A different algorithm could instead be used where, like the chosen algorithmic design, the problem is broken down into its 14 parts, but regular Dijkstra's is run for the distances of the road. In this design, the "Best Charge" algorithm will still be used in each of the steps, attempting to create optimal charging. However, this design does not take into account charging efficiency at all. Hence, it will waste lots of time as it will likely be forced to charge at slower chargers. Due to this drawback, this algorithm has not been chosen.

Alternate Algorithmic Design - Altered Dijkstra's + "Best Charge" Algorithm (Different Problem Interpretation):

In a different interpretation of the problem, where the driver is given the entire cycle of locations at the beginning of the race, the algorithm could take advantage of this. By generating the entire path first (still using the altered Dijkstra's algorithm discussed above) rather than just one part at a time, the algorithm could then use the "Best Charge" part of the algorithm for the entire path at once, allowing the charging to be much more efficient. Nevertheless, as this is not the interpretation of the problem used, this algorithm will not be implemented.

ADT Shortcomings

Adding and accessing information in nested ADTs:

Right now, there are no custom operations that allow information to be easily added or accessed; all nested ADTs currently require both outer and inner ADT to be navigated. Additionally other custom operations can be added to make querying or manipulating the data easier. Hence, the following will be added:

- **Dictionary:**
 - set_feature : dictionary x element x integer x element → dictionary
 - get_feature : dictionary x element x integer → element
- **Graph:**
 - next : graph x element → element
 - num_nodes : graph → integer

Explanations:

set_feature: takes in the dictionary, the key (element), the index (integer) corresponding to any given feature stored in the array which is the value of the dictionary, and the updated rating for the feature (element) and returns the updated/new dictionary.

`get_feature`: takes in the dictionary, the key (element) and the index (integer) corresponding to any given feature stored in the array which is the value of the dictionary and returns the rating of that feature for that key (element).

`next`: given the graph is a directed single cycle (which will be true for the cycle of locations graph), this operation takes in the graph and the current node (element) and returns the node (element) adjacent in the direction of the edge to it.

`num_nodes`: given a particular graph, returns how many nodes there are.

Milestone 3

Creating the Model

When initially creating the model, an attempt was made to incorporate as many different factors as possible into the problem. In the end charger efficiencies, road distances, road roughnesses, road speed limits, car consumption and car battery (all different for each location, road and car respectively) were included. This was a success as these were all the major factors that were discussed in milestone 1 and originally meant to be included in the algorithm. Nevertheless, even with all of these factors, the model still cannot model the real world perfectly as there are almost infinite factors that all have some effect on an uncontrolled rally race in the real world such as the one described in the problem. Even with the included factors, the problem is already extremely complex, especially to attempt to solve. Adding more factors would make creating an algorithmic design even more difficult to create.

Some problems and difficulties were discovered during the further development and pseudocode writing of the algorithm. These are discussed below.

Altered Dijkstra's:

Originally, the altered Dijkstra's that was created alternated between claiming nodes and claiming edges in a 1:1 ratio; however, when coding it up and visualising this algorithm taking place, a problem arose. From the start, only the starting node is claimed and then an edge is claimed. After this, however, only one known node could be claimed and so it would be claimed. This cycle would repeat itself in the next iteration. Hence the original altered Dijkstra's worked in the same way as the regular Dijkstra's: it did not consider charger efficiencies. By changing the ratio of node claiming and edge claiming to 1:2, the algorithm now actually considers the charger efficiency at the node. Luckily, this was realised and fixed and parts of the report have also been changed to reflect.

Algorithm Creating Process:

A difficulty that became evident almost immediately was actually creating the pseudocode for the chosen algorithmic design. It was hard to even get started as the process of the algorithm had only been thought of as a few sentences that describe generally how it functioned. What allowed the algorithm to be properly implemented in the end was the modularisation and (mostly) temporary abstraction of the algorithm. Breaking the algorithm up into its 3 major sub-algorithms, made these parts much easier to implement as their own separate functions. Though this caused some inaccuracy as they were being considered as separate rather than part of the full algorithm, these were much easier to fix after, as a large part of the algorithm had been completed after the writing of the sub-functions. Furthermore, abstracting particular difficult to break down parts of the sub-algorithms made writing them much more efficient, as a or a few (in most cases) temporary

and more high-level lines of pseudocode allowed these steps to be temporarily skipped and then come back to later after the rest of the function had been written.

Algorithm Justification and Limitation

While other algorithmic designs different to the one below can be used, this one was chosen as it is able to generate a mostly optimal path given only the information it is given in the real world problem. It follows a greedy approach, trying to balance optimising charging and weighted distance simultaneously for each of the individual 14 iterations all while only actually knowing the current next goal node and none of the other future goal nodes. Hence, this algorithm is able to use just the information that would be given in a real world scenario.

Additionally, this complete algorithm is relatively simple to explain and implement, and, due to the time constraint, this was necessary.

Furthermore, the separate parts of the algorithm are quite optimal when compared to others that aim to achieve the same thing. Compared to the other shortest path algorithms, Dijkstra's is just as accurate (or more) and also does not compute unnecessary information. The "Best Charge" algorithm is also very accurate and performs much better than other algorithms discussed. To further improve the score, "Shortest Road To" algorithm is also used in the final iteration, and this algorithm is able to achieve its aim quite easily.

It is the need to combine both the optimal charging and optimal travelling that forces inaccuracy to occur. Regular Dijkstra's algorithm has been modified so that it considers the ability to charge at different nodes, and while this may overcompensate the need for good chargers, running regular Dijkstra's in its place would likely result in a much less secure path as it does not consider chargers whatsoever. The fact that the problem needs to be broken up into steps also gives rise to many issues, as it is way easier to optimise charging for one continuous path, where every next goal node is known in comparison to 14 subpaths separately, where the charger after the current goal node may be good or bad.

Often, brute force algorithms give the most accurate results when compared to other algorithmic designs. In this problem, however, this such design is not at all feasible as the system of charging cannot be brute forced as there are way too many possibilities for how much a car can charge at a particular charger. The problem is also broken down into smaller steps in order to mimic the real world problem as accurately as possible, making it even more impossible to solve the problem with a brute force algorithm. Hence, a greedy approach is the best design given the context.

Low Level Pseudocode:

// Dijkstra's but where, during every second iteration, the node with the highest charger efficiency at the ends of Dijkstra's tree is claimed and in every iteration, an edge that is connected to a claimed node is claimed

// -> The final time to travel the path and the final path are returned

function run_altered_dijkstras(track, first, second)

let unseen be a set

let previous be a dictionary

for each node n in the track:

 set the value of n to infinity

 set the value of key n to None in previous

 add n to unseen

let claimed be a set

 set the value of first to 0

 add first to claimed

 set counter to be 1

while unseen is not empty

 // Claiming an edge

try do

 set u to be the node in unseen that is adjacent to a node in claimed with the lowest value

 remove u from unseen

except do

 pass

if counter is divisible by 2 then

 // Claiming a node

 set v to be the node with the highest efficiency not in unseen and not in claimed

 add v to claimed

 // Breaks once the goal node is reached

if v is second then

break

 // Calculate new edge weights

for each neighbor n of v still in unseen do

 set alt to be the sum of the value of node v and the weighted distance (roughness incorporated) of the edge all divided by the speed limit of the edge between nodes v and n

if alt is less than the value of n then

 set the value of n to alt

 set the previous node of n to v

 set counter to be the sum of counter and 1

let path be a list

 set current to be second

 // Reconstruct the path

```

while current is not None
    prepend current to path
    set current to the value of key current in previous

return the value of second, path

```

// An algorithm that fills up to full battery if the car is at the best charger or cannot reach a better charger even with max fuel, otherwise it fills up just enough to get to the next equal or better charger

// -> Total charging time, individual charging times and final battery are returned

```

function run_best_charge(track, path, initial)

```

```

    let charging be a queue
    set totalTime to be 0
    set full to be full battery

```

```

for each node n in path do

```

```

    // Best charger means charge to max

```

```

    if the charger efficiency index at n is 3 then
        set time to be the amount of time to charge from initial to full using the
        charger efficiency at n
    else then

```

```

        set best to be the next charger with an equal or better charger efficiency as
        n

```

```

    if best is not reachable on full charge then

```

```

        if the quotient of battery and consumption all multiplied by 100 is
        less than or equal to the weighted distance to best then

```

```

            // Charge however much is needed to get to the next
            better/equal charger

```

```

            set final to be the weighted distance divided by 100 all
            divided by consumption

```

```

            set time to be the amount of time to charge from initial to
            final using the charger efficiency at n

```

```

            // Don't charge if the current battery can make it to the next
            better/equal charger

```

```

        else then

```

```

            set time to be 0

```

```

        // Charge to full if a better/equal charger cannot be reached even on full
        charge

```

```

        else then

```

```

            set time to be the amount of time to charge from initial to full using
            the charger efficiency at n

```

```

        set totalTime to be the sum of totalTime and time

```

```

        add time to charging

```

```

        set initial to be final

```

```

set finalBattery to be initial

```

```

return totalTime, charging, finalBattery

```

// An algorithm that iterates through every edge that is connected to the node "final" and compares them to get the shortest one.
// -> The other node connected to the shortest edge as well as the time to travel the weighted distance of this shortest edge are returned

```
function run_min_edge(track, final)
    let tempMin be an array of size 2 containing None, infinity
    for each edge e connected to final node in track do
        set t to be weighted distance of e divided by the speed limit
        if t is less than the second element of tempMin then
            set tempMin to other node connected to e, t

    return tempMin
```

// Algorithm body code

// -> The time for the full race, the path and the individual charging times are returned

```
function algorithm(predeterminedCycle, starting, chosenCar)
    let track be a graph of the track where nodes represent locations and edges represent roads
    let locInfo be a dictionary where the keys are locations and the values are charger efficiencies
    let roadInfo be a dictionary where the keys are immutable sets containing location 1 and location 2 and values are arrays size 3 of road distance, road roughness and speed limit
    let carInfo be a dictionary where the keys are cars and the values are arrays size 2 of battery and consumption

    set first to be starting
    set second to be None
    set totalTime to be 0

    set initialBattery to be the battery of chosenCar
    let allCharging be a queue
    let fullPath be a queue

    // First 13 iterations
    // Run altered djikstra's then best charge
    while starting is not equal to second
        set second to be the next node after first in predeterminedCycle
        set roadTime, path to be run_altered_djikstras(track, first, second)
        set chargeTime, charging, finalBattery to be run_best_charge(track, path, initialBattery)

        append every element in charging to allCharging
        remove the head element of path and append the element to fullPath until path is empty
        set totalTime to the sum of totalTime, roadTime and chargeTime

        set initialBattery to be finalBattery
        set first to be second
```

```
// Last iteration
// Set the new temporary goal node to the one closest to the actual final node then do
the same thing as other iterations
tempFinal, finalNodeRoadTime be run_min_edge(track, second)
path, partialRoadTime be run_altered_dijkstras(track, first, tempFinal)
append second to path
set chargeTime, charging, finalBattery to be run_best_charge(track, path, initialBattery)
append every element in charging to allCharging
remove the head of path and append the elements to fullPath until path is empty
set totalTime to the sum of totalTime, finalNodeRoadTime, partialRoadTime and
chargeTime

return totalTime, fullPath, allCharging
```

```
// Predetermined parameters that can be set
let predeterminedCycle be a graph of the cycle order of locations to visit
set starting to be the starting location of the car
set chosenCar to be the chosen car
// Run algorithm body code and get results
set time, path, charging to be algorithm(predeterminedCycle, starting, chosenCar)
```

High Level Pseudocode:

// Dijkstra's but where, during each iteration, the node with the highest charger efficiency at the ends of Dijkstra's tree is claimed and only edges that are connected to claimed nodes are considered and can be taken in the next iteration

// -> The final time to travel the path and the final path are returned

function run_altered_dijkstras(track, first, second)

 set all node values to infinity

 set all the previous node of all nodes to None

 set the value of first to 0 and claim first

 alternate between claiming 2 edges by using the lowest node value and claiming the node with the highest charger efficiency connected to a claimed edge, updating the values of each node between every iteration (skip edge claiming **if** there are none)

break once "second" is claimed

 reconstruct the path by using the previous node of each node

return the value of "second" and the path

// An algorithm that fills up to full battery if the car is at the best charger or cannot reach a better charger even with max fuel, otherwise it fills up just enough to get to the next equal or better charger

// -> Total charging time, individual charging times and final battery are returned

function run_best_charge(track, path, initial)

for each node n in path do

 charge to max **if** the charger efficiency of "n" is 3 (max), storing the individual charging time and final battery

else charge to the bare minimum of whatever battery is needed to make it to the next better/equal charger, storing the individual charging time and final battery

else charge to full **if** a better/equal charger is not reachable even on full charge, storing the individual charging time and final battery

 add up the total time for all the charging

return the total time, the individual charging times and the final battery

// An algorithm that iterates through every edge that is connected to the node "final" and compares them to get the shortest one.

// -> The other node connected to the shortest edge as well as the time to travel the weighted distance of this shortest edge are returned

function run_min_edge(track, final)

 find the shortest edge to traverse in time that is connected to "final"

return the other node connected to this edge and the time it takes to traverse this edge

// Algorithm body code

// -> The time for the full race, the path and the individual charging times are returned

function algorithm(predeterminedCycle, starting, chosenCar)

let track be a graph of the track where nodes represent locations and edges represent roads

let locInfo be a dictionary where the keys are locations and the values are charger efficiencies

let roadInfo be a dictionary where the keys are immutable sets containing location 1 and location 2 and values are arrays size 3 of road distance, road roughness and speed limit

let carInfo be a dictionary where the keys are cars and the values are arrays size 2 of battery and consumption

// First 13 iterations

// Run altered djikstra's then best charge

run altered djikstra's then best charge then update overall time, overall path, overall individual charging and current battery **for** the first 13 pairs of nodes in the predeterminedCycle

// Last iteration

// Set the new temporary goal node to the one closest to the actual final node then do the same thing as other iterations

run min edge then altered djikstra's then best charge then update overall time, overall path, overall individual charging and current battery with the last pair of nodes in the predeterminedCycle

return the total time to complete the track, the full path and all the individual charging information

// Predetermined parameters that can be set

let predeterminedCycle be a graph of the cycle order of locations to visit

// Run algorithm body code and get results

run algorithm