

Computer Games Development CW208

SRS and Project Report

Year III

|  |  |
| --- | --- |
|  |  |
| [Paul Nolan]  [C00213034]  [04/09/20] | |

Contents

[Acknowledgements 2](#_heading=h.gjdgxs)

[Project Abstract 2](#_heading=h.30j0zll)

[Project Introduction 2](#_heading=h.1fob9te)

[Background 2](#_heading=h.3znysh7)

[Project Description 2](#_heading=h.2et92p0)

[Overview 3](#_heading=h.tyjcwt)

[Philosophy 3](#_heading=h.tyjcwt)

[Define the Application 5](#_heading=h.3dy6vkm)

[What is the application supposed to do 5](#_heading=h.1t3h5sf)

[Who is going to be using this application 5](#_heading=h.4d34og8)

[Context Diagram and Use Cases 5](#_heading=h.2s8eyo1)

[Metrics 5](#_heading=h.17dp8vu)

[Is there a precedent for this application? (Your inspiration): 5](#_heading=h.3rdcrjn)

[Project Milestones 6](#_heading=h.26in1rg)

[Project Review and Conclusions 6](#_heading=h.lnxbz9)

[References 6](#_heading=h.35nkun2)

# **Acknowledgements**

I would like to thank the following people who assisted in completing this project including:

Ross Palmer of Carlow IT who supervised this project.

# **Project Abstract**

This project focused on creating an A\* algorithm that is more applicable to real world applications, in this case a vehicle-based application/game. A\* is a pathfinding algorithm that combines a breadth-first approach that also takes into account the target position, so it can determine the shortest path from the starting point to the end point. However, the issues that remain with it are its ability to handle dynamic variables and create realistic long-term movement. Therefore, this algorithm, though widely used, is not perfectly suitable to creating games. Building an A\* variant that can take into account resources and user attributes (fuel, width etc.) can become a useful starting point for creating an algorithm that is suitable for games.

It was found that the created algorithm was slower than the A\*, but much more realistic as it chose viable paths. Future work will focus on creating more input variables, and on efficient path creation.

# **Project Introduction**

The goal of this project is to create a variant of A\* that can be more applicable to real world situations. In this case, a vehicle-based application/game was created to try incorporate a basic GPS-like system into the pathfinder. However, a GPS in the past would not take into account an attribute such as fuel or vehicle width. Therefore, this project focused on creating an algorithm, (a variant of the A\* algorithm), that takes into account the variables of the user such as fuel, width etc. It will try to implement two separate solutions.

The first solution will implement variables that remain static such as width. Therefore, any path that goes over a node that contains a minimum value that falls under a maximum value will be turned off. This solution should save time due to removing unnecessary elements of the map before trying to create the path.

The second solution will also apply diminishing values such as fuel directly to the nodes. The algorithm will start off with a pre-determined value to represent the fuel that will be applied to the start node. The algorithm will then check the start node’s neighbours, and apply a cost which equates to the fuel value minus the path cost to them. If the calculated cost falls below zero, that node will not be visited regardless of whether it was the most suitable one under normal conditions. Certain nodes will be able to increase the value given to them once. Previously calculated values for neighbours of the node in a previous iteration that have been assigned a lower fuel value than the value just calculated will be checked again. It is expected that this will increase time for the pathfinder to complete, but will also enable it to investigate if previously dismissed options are now valid.

A secondary objective is to implement dynamic elements into the map such as turning off or on certain arcs. To start this, the first action would be to calculate the point to start working from. When the arc is turned off, it will be checked to see if it interferes with the current path. If it does interfere, then one of the side nodes will be selected as a starting point. In cases where the arc is turned on both of its connected nodes will be checked to see if they were expanded into by the algorithm. If one was, a for-loop will be started to find the member of the path it can be linked to. After both instances are started, the current path will be split in two with the selected node as the current node, and all the following nodes being moved to a separate list. If the algorithm were to check the removed nodes again it will determine whether the vehicle can make the journey. If it can, the removed nodes will be added back into the path.

The reason for creating two situations for this to occur is to save time. It would be faster to determine whether a turned-off arc would affect the path than it would for a turned-on arc, as the latter will take far more iterations to determine if it will affect the path. Additionally, storing the removed nodes since if a member were to be selected for the new path it can quickly be determined if the vehicle can make that journey.

# **Background**

## Graph Theory

Before any pathfinders can be used, the game’s geometry has to be encoded as a map, which produces a Navigation Mesh or a set of Waypoints (Graham Hugh McCabe Stephen Sheridan, 2003). These waypoints are all connected, like nodes in a graph. Therefore, a pathfinder’s goal is to navigate the graph until it finds the endpoint.

A graph G is composed of two sets, and can be written as G = (V, E) (Graham Hugh McCabe Stephen Sheridan, 2003) where:

* V – Vertices: A set of points in n-space.
* E – Edges (arcs): Edges are connections between the points. These connections can be either directed or undirected (more on this later).

## Pathfinding

In many computer games, AI is used to move around agents in a virtual world. Developing complex systems for high-level decision making will be made redundant if an agent can’t find its way around a set of obstacles to implement that decision (Graham Hugh McCabe Stephen Sheridan, 2003). This is where pathfinders come in useful. Pathfinding is defined as the search for the shortest path between two nodes in a graph (Carlos Gonçalves Adaixo & Abel João Padrão Gomes, n.d.), given a set of constraints (Graham Hugh McCabe Stephen Sheridan, 2003).

A path connects two nodes, A and B, through a set of arcs and nodes. A node is a component that is used to represent a state and position, while an arc is defined as a connection between the two nodes (Carlos Gonçalves Adaixo & Abel João Padrão Gomes, n.d.)

## Approaches to pathfinding

As mentioned earlier, Pathfinding can be divided into two distinct groups: Undirected and Directed.

#### Undirected

Undirected algorithms blindly search the map until they find the goal. An example of an Undirected Algorithm is the breadth-first search, which searches all of the nodes connected to the start point before going on to search the children of those nodes, which will continue until the goal is found. This is detailed in (Stuart Russel, 2010).

#### Directed

Directed algorithms differ from undirected ones in that they do not move blindly in the graph. They are capable of assessing how much progress they are making towards the goal (Graham Hugh McCabe Stephen Sheridan, 2003). This assessment calculates the Cost of travelling to the next node, which is generally seen as the distance. An example would be the A\* algorithm which assigns a value to each node based on its distance from the goal with the nodes with smaller values taking priority.

There are two main strategies for directed pathfinders (Graham Hugh McCabe Stephen Sheridan, 2003):

* Uniform cost search g(n) modifies the search to always choose the lowest cost to the next node. This minimises the Cost of the path up to its current point, but it does not take into account future successor nodes.
* Heuristic search h(n) goes one step further and estimates the cost of the successor node after the goal node.

The two most popular directed pathfinders are Dijstra’s and A\*. Dijkstra utilizes the Uniform cost search to find the optimal path while A\* combines both (Graham Hugh McCabe Stephen Sheridan, 2003).

### Dijkstra’s Algorithm

The Dijkstra algorithm is a directed algorithm first presented in a paper published by Edsger W. Dijkstra in 1959(Dijjkstra, 1959). It introduced the method to find the shorted path between two nodes of a weighted graph, where each edge would contain a cost (Amador & Gomes, 2018).

The algorithm explores the whole graph while keeping track of the cost value of each node from the starting node (Amador & Gomes, 2018). This cost value C provides critical information on the extent of what has been explored from the previous node. Multiple paths can exist on a node, which means that a node might already have a calculated G cost. In this case an evaluation is performed to check if a better path can be created. If the new evaluation calculates a path with a lower cost value C, then the path is updated (Amador & Gomes, 2018).

#### Workings

The way the Dijkstra algorithm works is as follows:

1. A start node and goal node are selected.
2. At the start each node is assigned an infinite value. The start nodes value is made to zero.
3. The distances between the start node and its neighbours are found and are then added together with the start nodes value after which they are made into that nodes respective value. These nodes are then added to a list.
4. The previous step is then repeated with the first node on the list replacing the start node. The main difference in this step is that when assigning values to the neighbours if the neighbour already has a non-infinite value it and the new value are compared and if the new value is lower the latter will replace the former. The checked neighbours with the exception of nodes which were previously checked and weren’t assigned a new value and nodes that are already present are then added to the back of the list. The current node is then removed from the list.
5. The previous step will repeat itself until the list is completely empty.

### A\*

The A\* algorithm is a directed pathfinding algorithm based on the Dijkstra algorithm, that was first put forward by (P. E. Hart, 1968). The main difference between the two algorithms is that A\* will try to determine which nodes have the highest possibility of reaching the end point of the path (heuristics as mentioned before).

#### Workings

The way the A\* algorithm works is as follows.

1. First each node is assigned a heuristic value, where each arc is assigned a cost value. This is found by calculating the straight-line distance between the node, the goal and the two nodes the arc connects respectively.
2. Next the start node is selected, has a g and f value assigned to it and is added to a queue. The g value represents the accumulated cost that it takes to get from the start node to the node currently being examined. The f value is the sum of the nodes g value and heuristic value.
3. Next the neighbours of the starting node are examined and have the g and f values that would be applied to them if the current node being checked was their previous node. These values are then compared to the values the nodes already have and if they are less than the nodes value the current node is made into the previous node for it. These neighbours are then added to the queue if they haven't been already.
4. The previous step repeats itself until the end node is found. After this the path is created from the lists of nodes connecting the start and end node.

## Limitations of traditional approaches

Traditional approaches to pathfinding have certain limitations.

The first limitation is that most approaches cannot handle dynamic environments. This is due to most of them not being able to take into account a change in the environment that would block a path they created. While it would be possible to run the algorithms again to create a new path that would take up far more time then it would in a non-dynamic environment (Graham Hugh McCabe Stephen Sheridan, 2003).

The second limitation is non-realistic movement (Graham Hugh McCabe Stephen Sheridan, 2003). This refers to how objects moving under the algorithms move in straight lines when moving from one point to another. This issue becomes more apparent the lower the number of nodes an environment has and can be somewhat solved by increasing the number of nodes which would as a cost increase the time taken to create paths.

## Combining Algorithms

While most algorithms are developed from scratch or developed from a single algorithm some are developed by taking the elements of two separate algorithms. This method when used successfully will result in an algorithm that is capable of handling all the situations its parents were designed for while suffering less flaws. An example of this would be the integration of a short-term dynamic algorithm into the a\* algorithm. While the latter is capable of creating long term paths it can’t take into account dynamic elements efficiently and while the former can handle collision avoidance it cannot create a long-term path. These two algorithms can be used together with the a\* algorithm handling standard movement while the dynamic algorithm will take over if a collision is determined to be imminent.

# Project Description

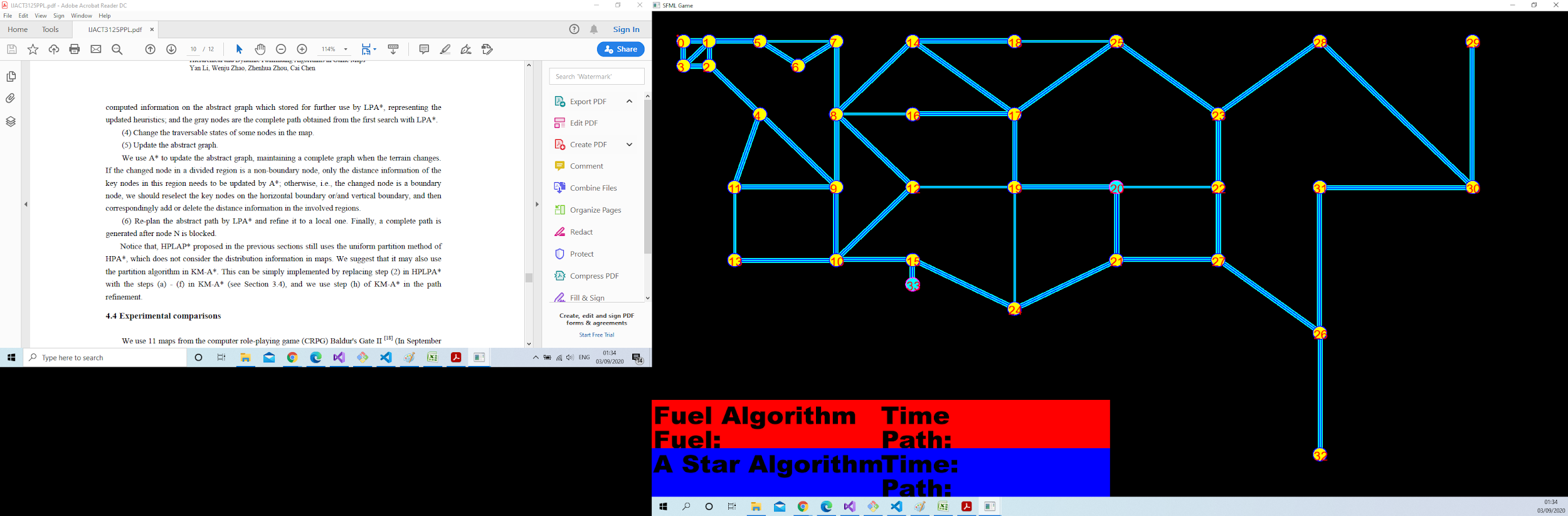
## Methodology

The following methodology was used to run the project algorithm:

1. Start-up program. Data is loaded at this point from a Yaml file. This is used to construct the map.
2. Once the program is started, a menu is displayed on the screen. 3 options are given to the user.
3. Once an option is selected, the user is redirected to a screen displaying the map and the car created.
4. If a town is selected, the custom algorithm and the A\* algorithms are executed. Results such as time and paths are sent to the hud (UI element that displays information for the player).
5. The object representing the vehicles moves to demonstrate the path created by the custom algorithm.
6. If a road is clicked while a path is being run the algorithm will be called with the road’s id passed to it. If it is determined that the change can affect the path the algorithm is rerun to adjust the path.

# Map

Below outlines the map that was created for this project.



The map can be broken down into three basic components:

* **Nodes**: These are the yellow points on the map with numbers assigned to them. The algorithm uses these nodes along with the fuel nodes to create the path that the vehicle uses to travel through the map.
* **Fuel Nodes**: These are similar to the regular nodes but with two differences. First, they are cyan coloured with a purple outline to differentiate them from the regular nodes. Second, when the vehicle passes through them their current fuel is increased.
* **Roads:** These are the blue lines shown above. The roads are used to represent the arcs used in the algorithm. The active state of each road can be changed with the roads that are not active not being taken into account by the algorithm.

## Algorithm

The Algorithm created by this project is based on the A\* with a number of differences. These differences include how it searches through the map, variables it takes into account, dynamic elements and other additions to maximise the positives, and then minimise the negatives of the changes.

# The first difference from the base A\* algorithm (which uses breadth first search), is that this algorithm uses depth first search and will only end when its associated queue is empty. This is done to ensure a valid path is created as quickly as possible, after which the paths current total cost is recorded. This cost is then used as a limiter for the number of nodes checked by comparing it against the current total cost of each member of the queue, their heuristics and the estimated total cost of nodes being checked in addition to their heuristic. In the case of the former, the process for that member is skipped and is then removed. In the case of the latter, its examination will be ended prematurely. This is done because the current total cost of a node, added to its heuristic, reveals the current minimum distance of the path that it is currently a part of. If said distance is higher than the current recorded distance, and it is confirmed the path has no possibility of being shorter than the current completed one, no further time needs to be spent investigating it.

The second difference is that three variables are added onto the nodes of the map. These are:

* A list of previous ids
* Current fuel & fuel value
* Number of stations passed.

### Variables

#### List of Previous ID’s

The list of previous ids is used to store and determine the current path being checked by the algorithm. This is used to allow the possibility of paths having two instances of a node if necessary. This could be required if the node were to only record its immediate predecessor, such a path wouldn't be able to form due to the previous instance of the node being overwritten.

#### Current Fuel

The current fuel is used to determine if the possible path at the moment is still viable by checking if the estimated current fuel of a checked node is above zero. If it isn't that node won't be added to the queue, otherwise that node is added to the queue. Then its current fuel is made equal to its fuel value plus the estimated current fuel, which is calculated as the current fuel of the previous town minus the cost of the road.

If the fuel value is above zero, it is used to increase the current fuel of its current path, therefore extending its life span. In addition to this, it also enables the possibility of backtracking and the re-examination of previously checked nodes due to the increased fuel capacity. Therefore, allowing the possibility of a previously failed path now being successful. If so, these nodes are added to the queue.

#### Number of Stations

# The number of stations passed is used to keep track of the number of nodes passed with a fuel value higher than zero. This, together with the previous ids list, is used to determine if a node with a non-zero fuel value has previously been checked by the current path. This is determined using a for loop, that is run through until the number of stations found is equal to the current value. If that is the case, then no further checks are required and the node is added to the path. This avoids creating an infinite loop. It is also used to determine if a node added to the queue early on, is more fuel efficient than a neighbour that was checked after it was added. This is done by finding the difference between their respective values, which is used to determine how much extra fuel the later node has received. The latter node’s current fuel, minus the extra fuel it has received, is then compared to the earlier node. If the earlier node has a higher current fuel, it is determined to be more efficient and the later node is re added to the queue.

### Struct Queue

To prevent the potential backtracking from overwriting the data of nodes, that are yet to be checked, the node queue was replaced by a struct queue. The idea of this was to save space by storing only one queue instead of two. The struct is used to store the non-static values of a specific instance of a node, in order to prevent the data itself from being overwritten and erased. These values include:

* the id of the node
* the id of its immediate predecessor
* its accumulated cost
* its current fuel
* the number of stations passed.

These values are then reapplied to their respective node once the struct is called on.

### Dynamic Elements

The dynamic elements of the algorithm are as follows. If a change to one of the arcs of the map were to occur while the vehicle is travelling, the algorithm will be recalled with the arc id passed to it. The status of the arc is checked and the algorithm proceeds based on whether it was turned off or on.

For when the arc is turned off, it is checked to determine if the two nodes connected to the path, is already a part of the path in its current state. If it wasn't, the algorithm is stopped early due to the change not affecting the path. If it is, each node after the breaking point is removed from the path, with the node before the breaking point being the first node the algorithm checks. Due to the path now being broken, the path that will be created will almost certainly have a higher total cost. This means the current path cost value will have to be reset.

For when the arc is turned on, it is checked if the minimum possible distance of a path, including the two connected nodes, is lower than the current paths cost, and if one of the nodes had previously been checked for that specific path. The minimum possible distance of the path is calculated by adding the heuristic of the node closest to the end point, the distance between the start point and the member of the pair closest to it, and the path cost (distance) between the pair. If both conditions mentioned above are met, the break point is found by checking if a member of the pair is part of the path. If that is not the case, the member of the path closest to the break point is made head of the queue. Since the path has not been broken, the current path cost is still usable, and does not need to be reset. This reduces the time needed to find a new potential one. If the conditions are not met, the algorithm will be ended early and the paths are left intact.

After both cases the struct queue is then prepared with the data of each member of the path being applied to its node. This to ensure the previous instance of the algorithm does not interfere. The struct queue is then prepared with the earliest node of the path, and its data is pushed to the start. The latest member will then be pushed to the back where it will be immediately examined.

### Missing functionality of the design

One part of the original design that wasn't fully included was the use of a proper level rather than just a series of arcs and nodes. This wasn’t included due to time constraints and difficulty determining how to determine the border and exit points between the nodes and their roads. It was however included to an extent with the roads which are composed of tiles.

## Learning

### Technical & Personal Achievements

The achievements for this project would be as follows:

* Create an algorithm that successfully takes in the factors of the user into account. In this case it was fuel.
* Create an algorithm that incorporated dynamic elements into it. This will then take into account paths becoming impassable or passable, to determine if a new path would be necessary.
* Create an algorithm that combines features from existing algorithms in order to decrease known flaws of them. Such as using the dynamic elements of the LPA\* algorithm and applying to this one.
* Incorporate techniques such as back-tracking into the algorithm. This means the algorithm can, if necessary, go down a dead-end to acquire a resource and return, without the need to create a new path.

# **Overview**

## ***Philosophy***

Philosophical point #1

This application is trying to create an algorithm that is fully capable of taking into account the resources and attributes of the user.

Common Questions

What is the game?

The game is a simulator designed to run and test a custom algorithm meant to generate the shortest viable path available to the user.

Why create this game?

I created this game in order to show that pathfinding algorithms are capable of taking into account the resources and attributes of the user.

What do I control?

You will be in control of a car which navigates its way through the map by following the algorithm created for it.

How many characters do I control?

There is only one character in this game.

What is the main focus?

The main focus of the game is to run the created algorithm and compare its performance to the original A\* algorithm.

What’s different?

Unlike most other pathfinding algorithms this one is capable of taking into account the users attributes and resources when creating the optimal path for them.

# **Define the Application**

What is the application supposed to be?

The application is suppose to calculate the shortest viable route available to the user based on their resources.

What the product is.

The product is a simulation demonstrating the algorithm created to accomplish the above task.

# **What is the application supposed to do**

* To run the algorithm and create the shortest viable path available.
* To run the a\* algorithm and compare its results and performance to the new algorithm.
* To quickly adjust the path whenever a change occurs in the map.
* To go into the map when selecting a car size.

# **Who is going to be using this application**

Drivers planning out long trips.

# **Metrics**

I will consider the project a success if the path is able to create a shortest possible viable path between two points for a user at a fast rate. If the algorithm were to produce a path that has negative fuel values then it will be considered a failure.

# **Is there a precedent for this application? (Your inspiration):**

This application is functionally similar to a gps which is designed to plan out the shortest path available to a driver.

It differs in how it tries to take into account the resources the driver has such as fuel when creating the path. This will help decrease the chances of a driver being running out of certain resources such as fuel due to not properly researching the region they are travelling through.

# **Design Manual**

Model the Application.

Design and describe how the application will be used: e.g. Navigation sequence in Web based project.

The application has two games states which are class selection and gameplay. The game starts off in class selection where the player selects a class after which the game state is changed to the gameplay state. When a part of the map is selected the shortest viable path to that part will be calculated.

This may necessitate some high level design in order to convey how the application may be used to access its functionality.

## UI

The user interface is broken down into two rectangles which are each used to represent a pathfinding algorithm. These rectangles display the paths created by the algorithms and the time taken to create the path. For the custom algorithm there is an extra section used to display the fuel of the user as they navigate through the map.

# 

# **Project Review and Conclusions**

It was found that the A\* algorithm generally performed better in terms of time take to complete the path. However, these results were found to be unviable due to the negative fuel values. Therefore, it can be concluded that the custom algorithm, whilst being slower than the A\*, is more suitable to the problem involved. The main reasons for the slower time to complete the path is due to the increase in path cost that resulted from the fuel points that were present on the map, i.e. the custom algorithm was aware of declining fuel values and the need to then refuel. This awareness was not present in the A\* algorithm.

The second conclusion that was made was that the proximity of the fuel points to the start point of the algorithm made a huge difference to the finishing time of the pathfinder. When the fuel point was closer, the path was less complex and therefore the cost of it was lower. For these instances, it meant that the vehicle only needed to refuel the once, which helped to reduce the finishing time.

There are aspects of this algorithm that could be improved on. One would be to add more cumulative variables. For this, fuel was used as a diminishing variable, i.e. it could never go below 0. So, it would be worth exploring the addition of another variable that has a maximum limit. In this case the addition of rest points, determined by time taken between the start point or the last rest point passed. This should help calculate a more realistic result, as it takes into account more variables. To do this however, a more efficient process would be required to check the variables, update them and keep them within reasonable limits.

Two things to improve on in this algorithm would be to use cumulative variables that must not get too high and to efficiently manage multiple different resources. An example of the former would be time taken in driving before taking a break with nodes that represent stop points where the time is reset and an example of the latter would be to integrate that variable into the current algorithm without making it any more complex as that would only increase the amount of time taken.

Additionally, another improvement is the introduction of another method that could effectively reduce the time taken in creating long term paths. As demonstrated in the results, the current finished path cost loses its effectiveness as a limiter, the longer the path is. A possible method would be to determine if the current node being investigated can potentially get to the target, or failing that, make it to a fuel point node with the fuel it currently has assigned to it. If it's the latter, it would also need to be determined if the potential path that includes the node could have a lower cost than the current finished path cost. This could be used to replace the current method, which checks to see if the path runs out of fuel on the next node. Ideally it should be able to determine that much earlier in the algorithm.

# 

# **References**

 Amador, G. P., & Gomes, A. J. P. (2018). XTrek: An Influence-Aware Technique for Dijkstra’s and A Pathfinders. *International Journal of Computer Games Technology*, *2018*. https://doi.org/10.1155/2018/5184605

Carlos Gonçalves Adaixo, M., & Abel João Padrão Gomes, D. (n.d.). *UNIVERSIDADE DA BEIRA INTERIOR Engenharia Influence Map-Based Pathfinding Algorithms in Video Games*.

Dijjkstra, E. W. (1959). A Note on Two Problems in Connexion with Graphs. In *NUMERISCHE MATHEMATIK* (Vol. 1, Issue 1). https://doi.org/10.1016/0042-6989(66)90039-3

Ezra Sidran, by D. (n.d.). *An Analysis of Dimdal’s 1 “An Optimal Pathfinder for Vehicles in Real-World Terrain Maps” Spring 2005*.

Graham Hugh McCabe Stephen Sheridan, R. (2003). Issue 2 Article 6 2003 Part of the Computer and Systems Architecture Commons. *The ITB Journal*, *4*(2). https://doi.org/10.21427/D7ZQ9J

P. E. Hart, N. J. N. and B. R. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. In *IEEE Transactions on Systems Science and Cybernetics* (pp. 100–107). IEEE.

Sazaki, Y., Primanita, A., & Syahroyni, M. (2018). Pathfinding car racing game using dynamic pathfinding algorithm and algorithm A∗. *Proceedings - ICWT 2017: 3rd International Conference on Wireless and Telematics 2017*, *2017*-*July*, 164–169. https://doi.org/10.1109/ICWT.2017.8284160

Stuart Russel, P. N. (2010). Artificial intelligence: A Modern Approach. In *2010 The 2nd International Conference on Computer and Automation Engineering, ICCAE 2010* (Vol. 4). https://doi.org/10.1109/ICCAE.2010.5451578

Yan Li, Wenju Zhao, Zhenhua Zhou, C. C. (2013). Hierarchical and Dynamic Pathfinding Algorithms in Game Maps. *International Journal of Advancements in Computing Technology*, *5*(11), 87–98. https://doi.org/10.4156/ijact.vol5.issue11.10