CMP201 – Data Structures and Algorithms 1 Coursework

JOSEPH LEE – 1903399

Comparison Between Lee and A* Pathfinding Algorithms For Pathfinding With Seabed Bathymetry Data

Problem

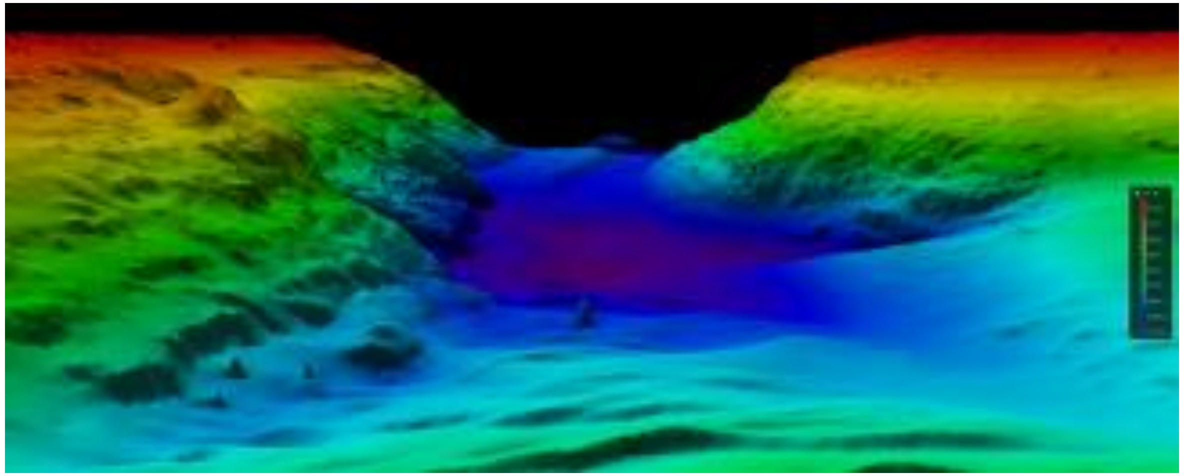
The Real-world Problem

In the offshore tidal energy industry there is often a requirement to route subsea cables on the seabed. These routes need to factor in both the distance of a cable run and smoothness of the seabed for this run as the cables experience excess fatigue and movement on the seabed if they have any unsupported spans.

This project set out to implement a method to calculate these routes between a selected start point and endpoint.

Bathymetry Data

Bathymetry Data



National Centers for Environmental Information (2020). Available at https://www.ncei.noaa.gov/sites/default/files/styles/max_325x325/public/sites/default/files/bathymetry-image-1200x480.png?itok=82IoqMAs

Bathymetry Data is a 3D representation of the seabed gathered from a set of depth and position results. When interpolated and plotted as a contour or surface they give a 3D representation of the seabed.

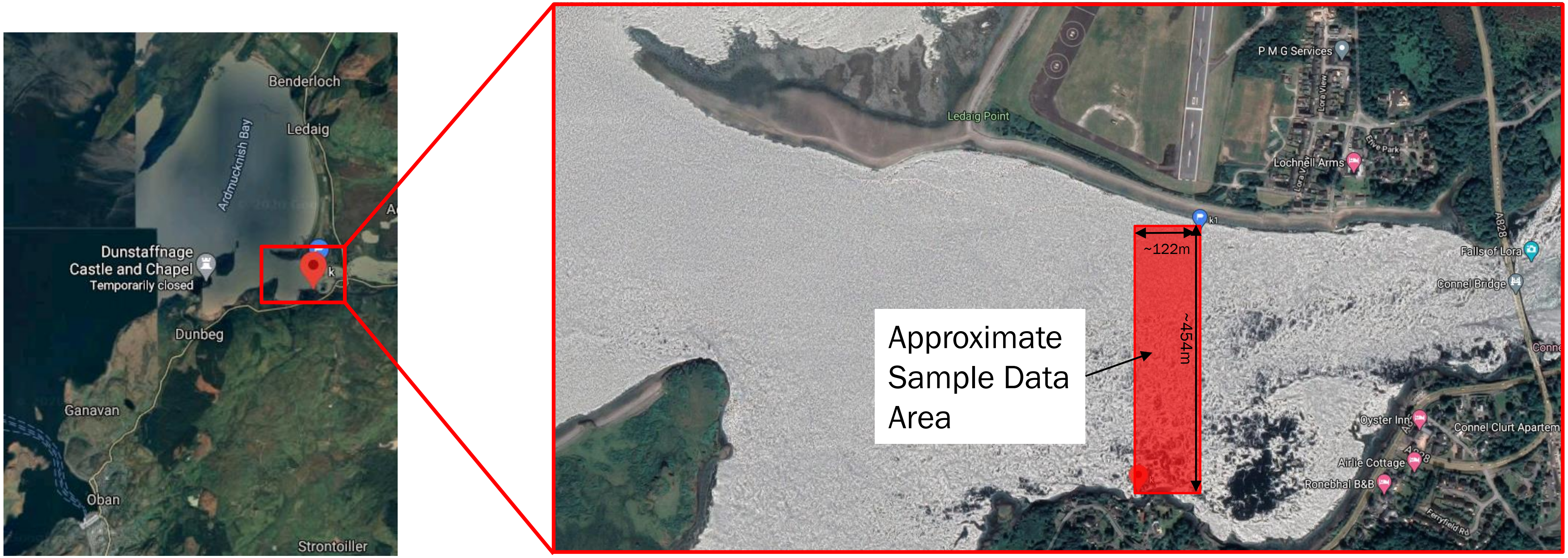
Bathymetry Data

The data used for this project is a small sample of the bathymetry data for the North Connel River, UK Hydrographic Office Survey (2014 HI1441 Loch Etive 2m SDTP).

(Admiralty Marine Data Portal (2020). Available at: https://datahub.admiralty.co.uk/Bathy_Data/Prodbathy/Bathymetry/2014%20HI1441%20Loch%20Etive%202m%20SDTP.bag)

Data converted from .bag to .csv using commercially available Global Mapper software

Bathymetry Data

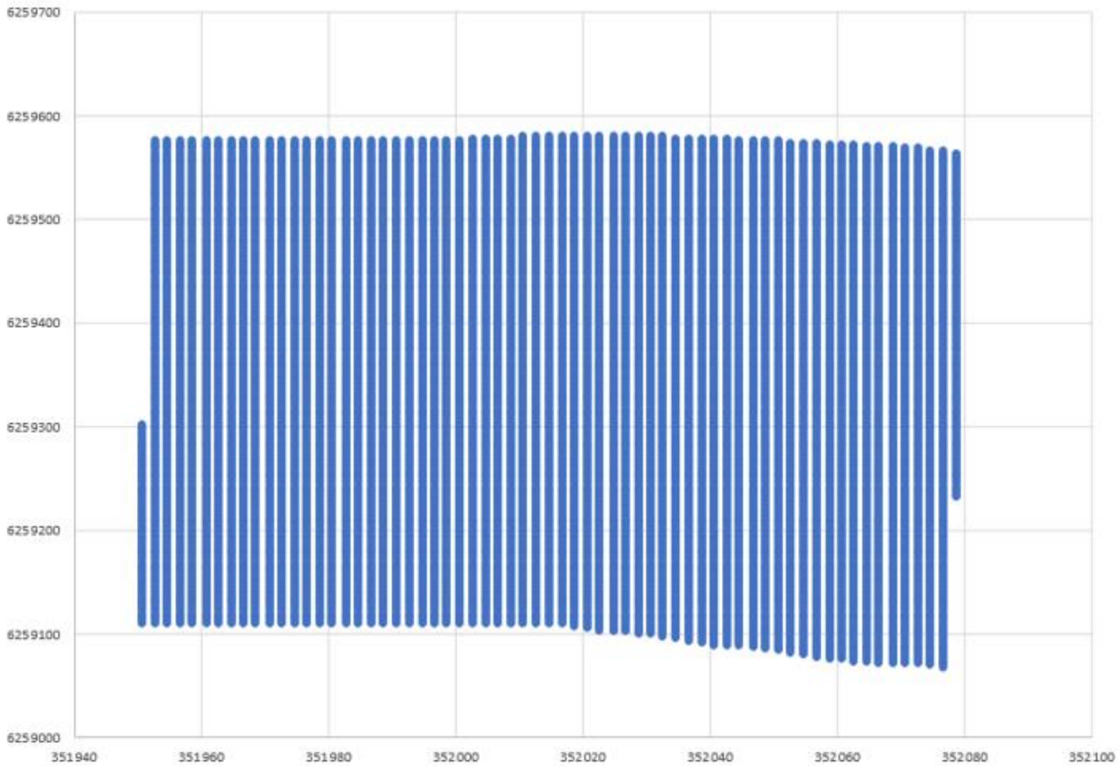


Approximate Sample Data Area

Google Maps(2020). Available at https://www.google.com/maps?q=google+maps+connel&rlz=1C1CHBF_en-GBGB792GB792&um=1&ie=UTF-&sa=X&ved=2ahUKEwjwgp3Bs4DuAhXIh1wKHSG-AGAQ_AUoAXoECAYQAw

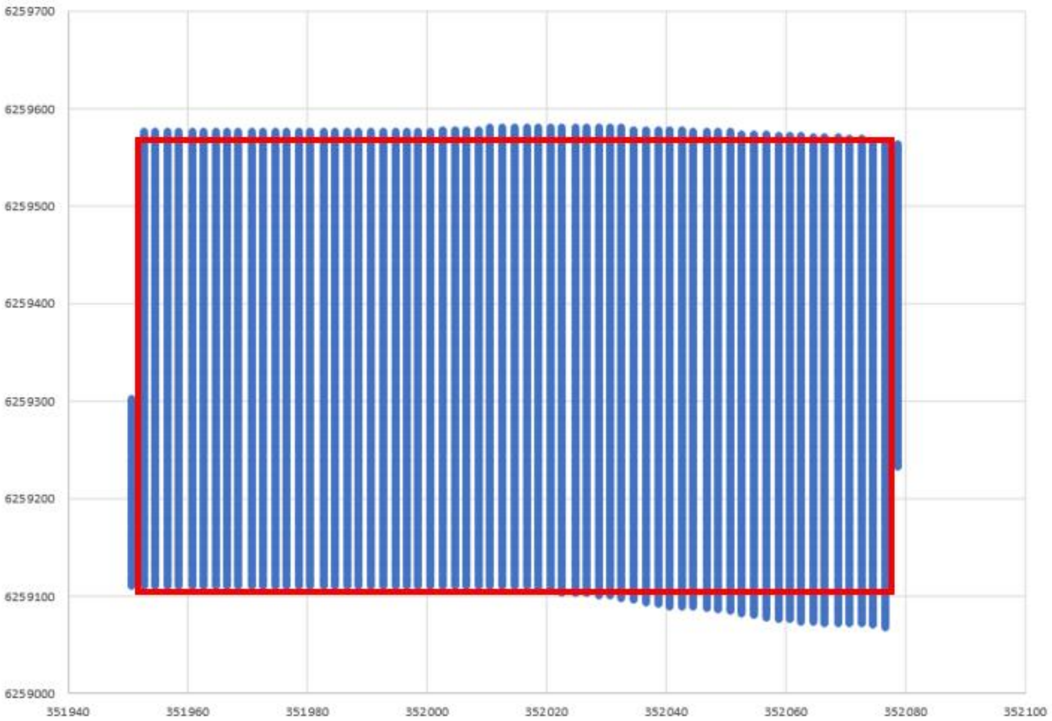
Bathymetry Data

Data is a grid of discrete point in rows and columns as shown below:



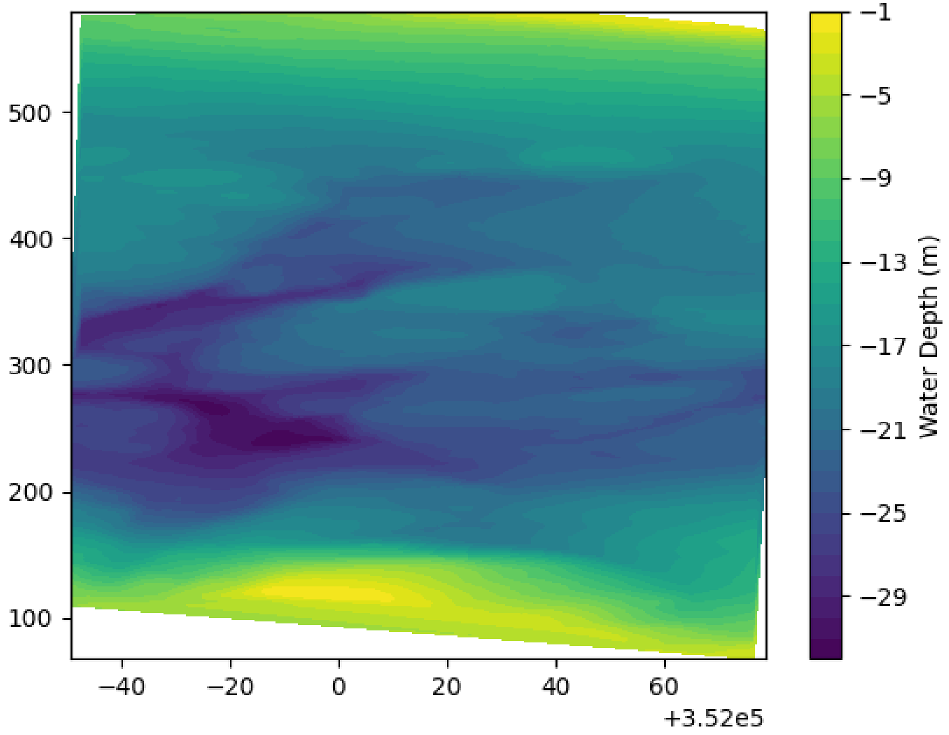
Bathymetry Data

To make the data easier to process for this program the dataset has been trimmed:

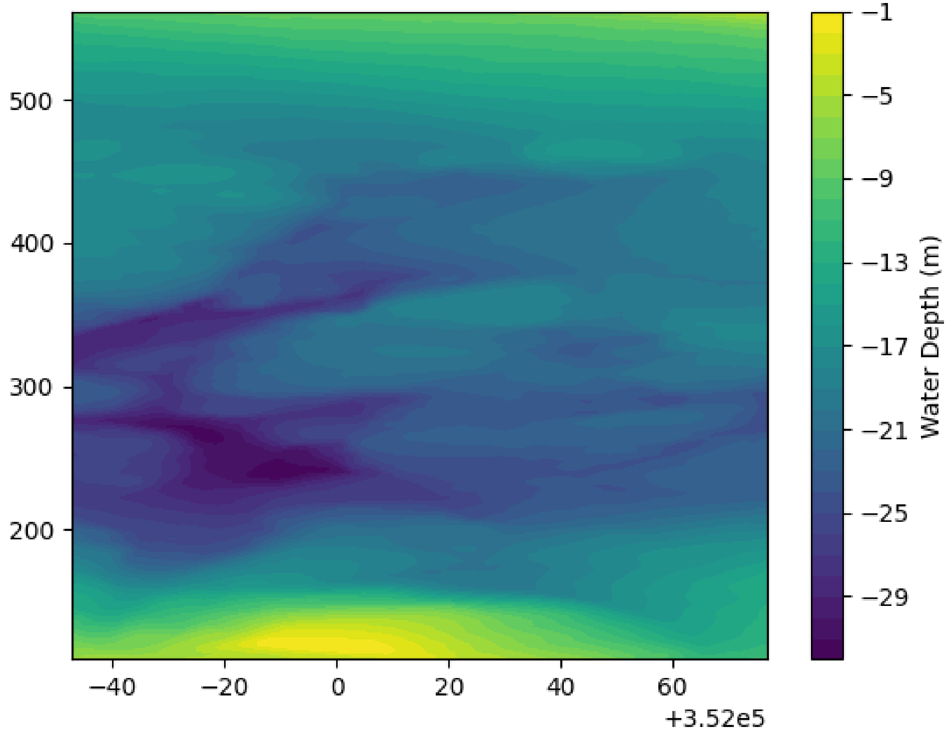


Bathymetry Data

Full Dataset



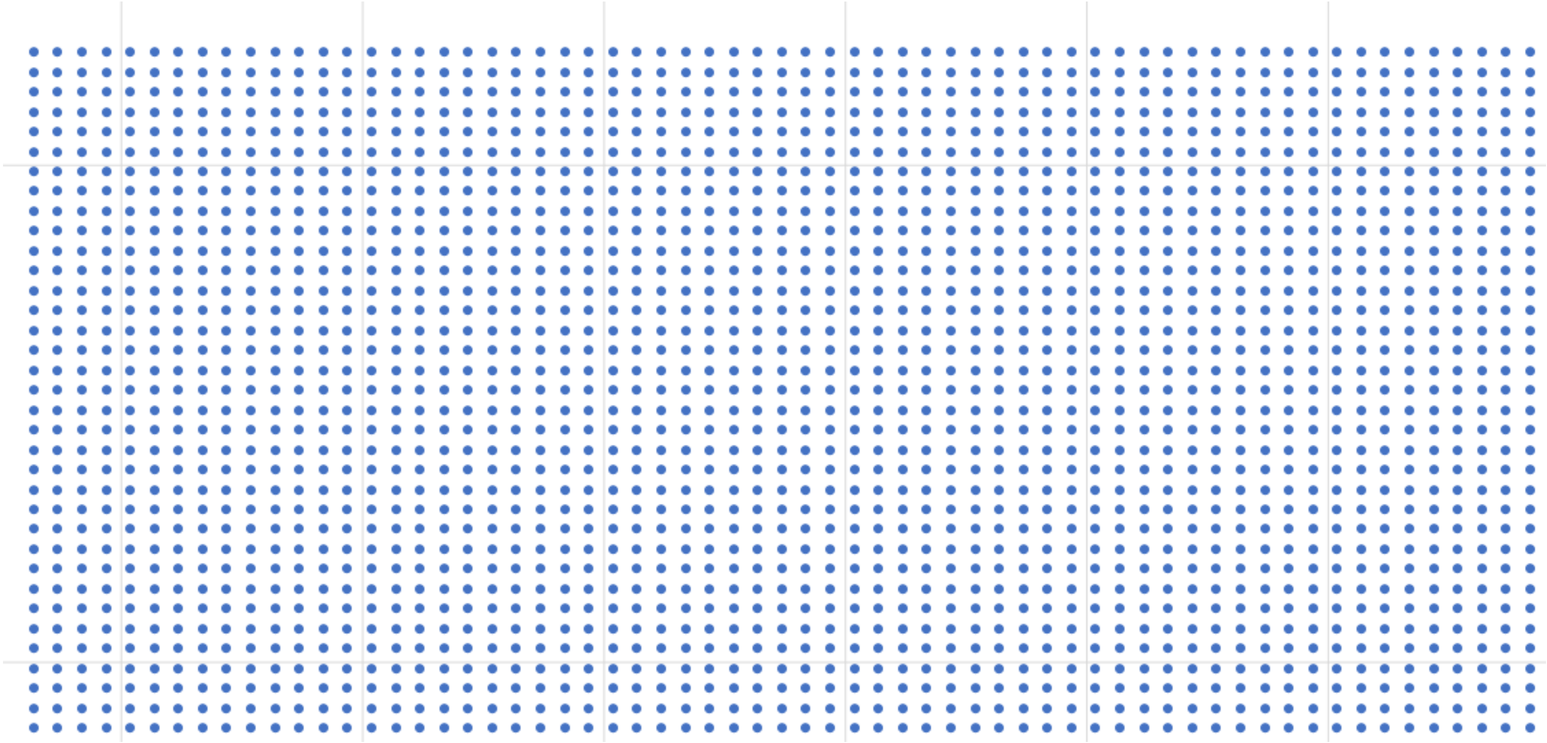
Trimmed Dataset



All Bathymetry plots except where referenced are plotted using the PlotterV1.3.py python script using Mathplotlib included in source code directory.

Bathymetry Data

Zoomed data showing the grid of point spaced by 2m



Representation of Data

As the bathymetry data itself is a grid of discrete points the choice was made to represent the data as a grid for the pathfinding algorithm. As the size of the data grid is not fixed the decision was made to use a **2D vector. std::vector<std::vector<node>>,** with the node being the relevant node type for each pathfinding algorithm.

The **population of this grid has a time complexity O(N ^{2})** however, this operation only needs to take place one at the start of the pathfinding function and no additional items need to be inserted into the 2D vector at a later stage.

One advantage of this approach is that to **access each element for pathfinding has a time complexity of O(1).**

A* Heuristic

The heuristic selected for this A* algorithm has was the Manhattan Distance because of this the route movement has been limited to **4 directions (up, down, left, right).**

The Manhattan Distance is calculated:

Manhattan Distance = minimum distance between nodes X (absolute ∆x between this node and goal + absolute ∆y between this node and goal)

A* Open & Closed Lists

Open List

A custom implementation of a priority queue has been implemented for the A* open list, **PriorityQueue.** This queue acts as a standard queue for **Push()** and **Pop()** but it will insert the Nodes in order of F-Cost of each node when the Push() is used**.** This means you have a worst-case insertion complexity of **O(N)** where N is the number of items in the queue and a **Pop()/Lowest()** complexity of **O(1).** Push() resolves conflicts by adding the most recent at the end of the block of equal items, this is to maintain the FIFO characteristics of the Queue**.**

A* Open & Closed Lists

Closed List

An std::unordered_set has been used to store the closed list nodes, this has been selected because the order of this set is not important for the algorithm. The unordered set is based on hashtables so all operations on the unordered set are O(1).

‘Smoothness’ Weighting

To account for adapting the route to find the shortest smoothest route each algorithm includes a weighting base on the difference in depth between the current node and the node being assessed.

For the Lee algorithm this is calculated as the difference in depth and added to the Distance value in the **LeeNode**, this summed value in saved in the **WeightedDistance.** It is this **WeightedDistance** that is used to find the shortest path in back track phase.

For the A* algorithm this difference in depth is added to the **G_Cost** for each **AStarNode**, so the **G_Cost** of each node is calculated as:

G_Cost = Distance from this node to start point + abs(previous node depth – this node depth)

AStarNode

Each AStarNode contains the below data:

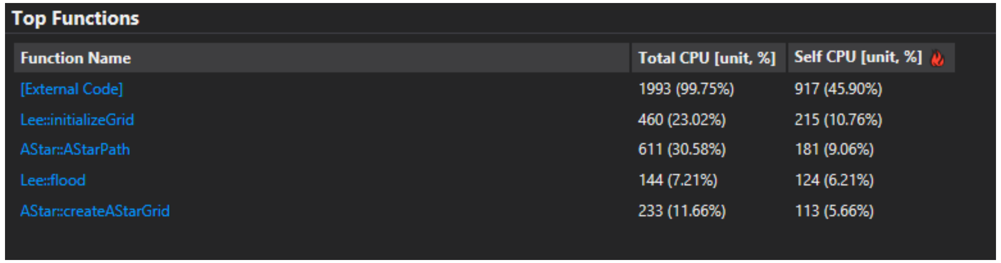
- G_Cost – Distance from this node to the start node
- H_Cost – Estimated distance from this node to the goal
- F_Cost – Total Node Cost (G_Cost + H_Cost)
- Depth – Depth of node based on bathymetry
- Distance – Distance between this node and the previous node
- Position – Relative position of node within the AStarGrid
- UtmPosition – Real-world UTM position of this node
- ParentNode – A pointer to the parent node of this node
- Visited – Set to true when a node has been visited and added to the closed set

LeeNode

Each LeeNode contains the below data:

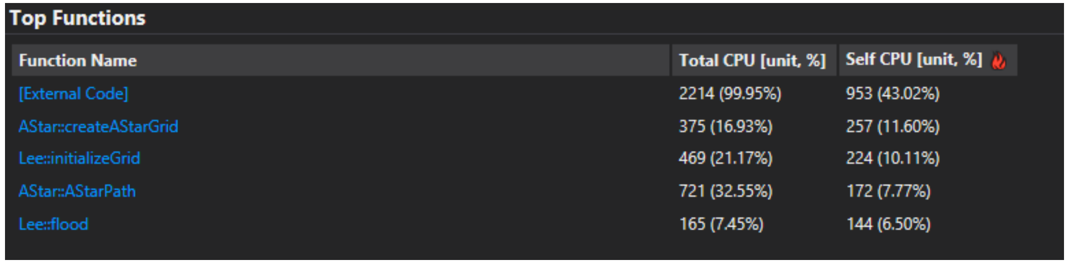
- Position - Relative position of node within the LeeGrid
- UtmPosition - Real-world UTM position of this node
- Distance – Distance from this node to the start node
- WeightedDistance – Distance adjusted for depth bias (Depth + Difference in depth between nodes).
- Depth - Depth of node based on bathymetry

Optimizations



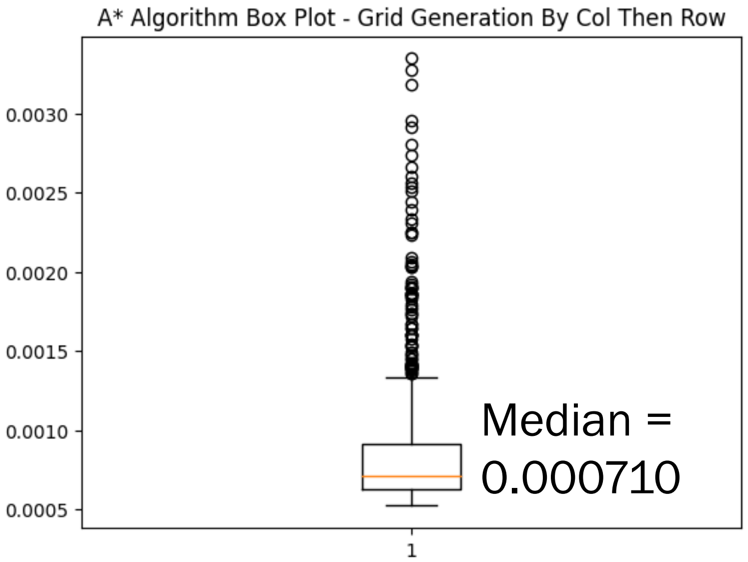
Top Functions

Function Name	Total CPU [unit, %]	Self CPU [unit, %]
[External Code]	1993 (99.75%)	917 (45.90%)
Lee::initializeGrid	460 (23.02%)	215 (10.76%)
AStar::AStarPath	611 (30.58%)	181 (9.06%)
Lee::flood	144 (7.21%)	124 (6.21%)
AStar::createAStarGrid	233 (11.66%)	113 (5.66%)



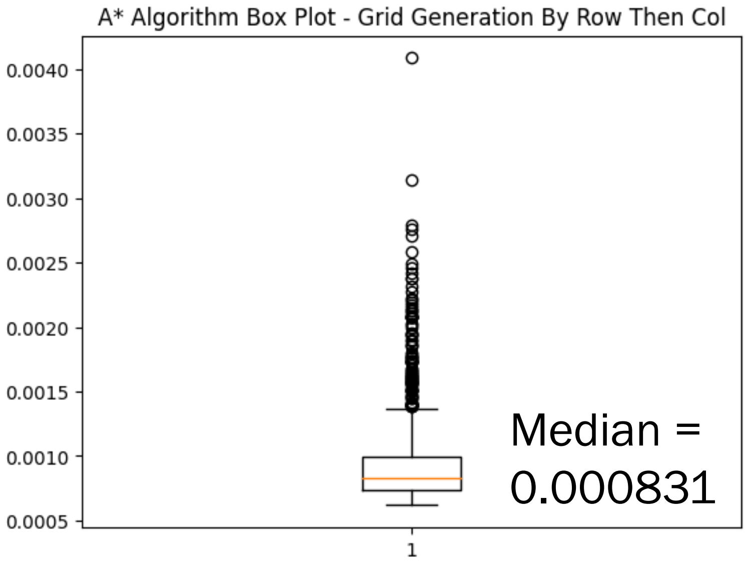
Top Functions

Function Name	Total CPU [unit, %]	Self CPU [unit, %]
[External Code]	2214 (99.95%)	953 (43.02%)
AStar::createAStarGrid	375 (16.93%)	257 (11.60%)
Lee::initializeGrid	469 (21.17%)	224 (10.11%)
AStar::AStarPath	721 (32.55%)	172 (7.77%)
Lee::flood	165 (7.45%)	144 (6.50%)



A* Algorithm Box Plot - Grid Generation By Col Then Row

Median = 0.000710



A* Algorithm Box Plot - Grid Generation By Row Then Col

Median = 0.000831

All Boxplots are plotted using the BoxPlotter.py python script using Matplotlib included in source code

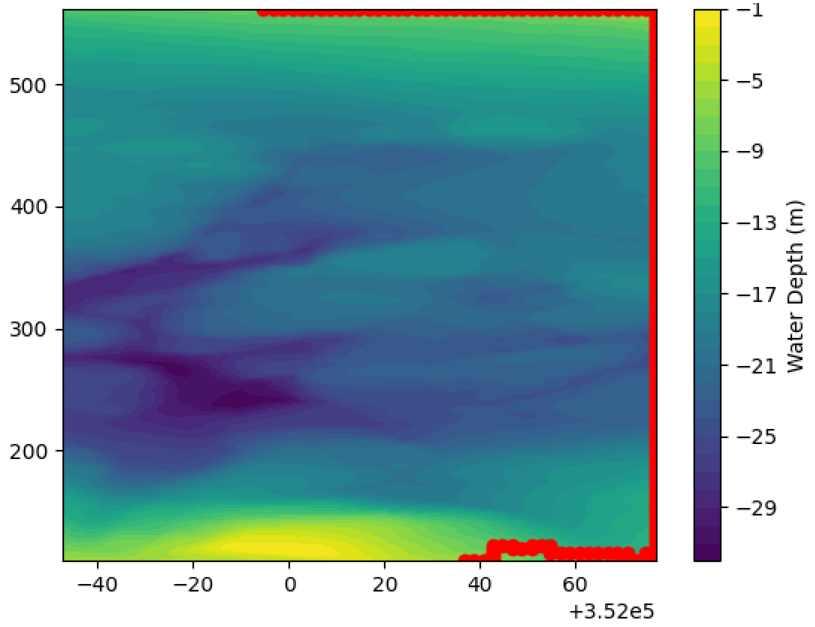
Results

Results

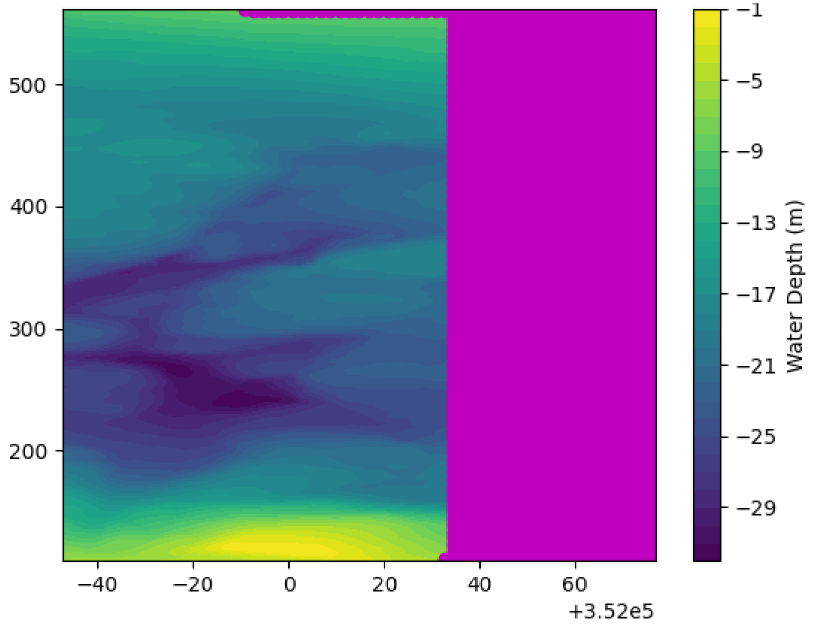
- All results have been collected over 1000 executions of the code and the median value used to account for an non-normal data distribution.
- All other programs were closed during performance tests
- Start point, end point and dataset were kept the same for each comparison run direct comparison
- All debug code was removed from pathfinding functions before performance tests and release compile used
- Only the pathfinding algorithm time is measured all .csv writing is done after timer is stopped.
- The Steady_clock was used to use monotonic time. This was used as intervals are constant and not affected by changes in the system clock, reducing errors introduced by changing system load.

Plot 1 - Lee

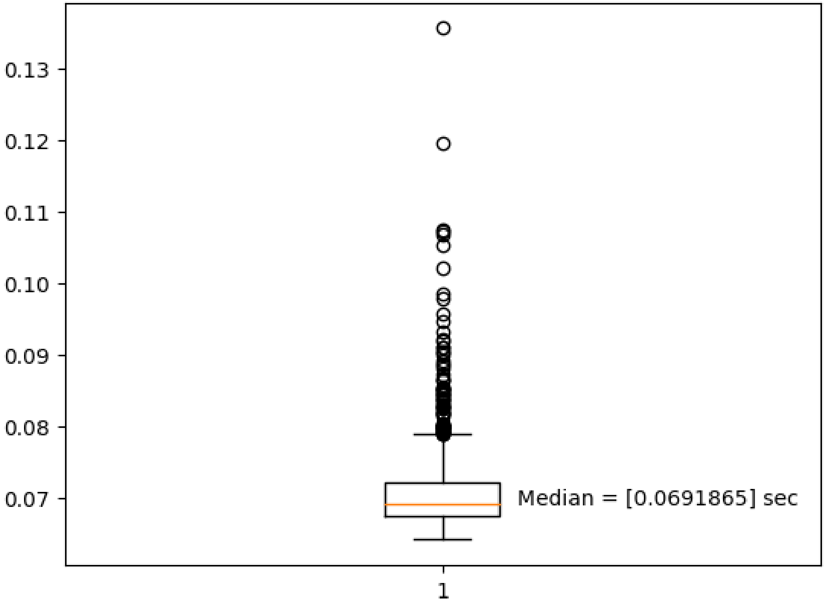
Full Data – Lee Path Plot 1



Full Data – Lee Visited Plot 1

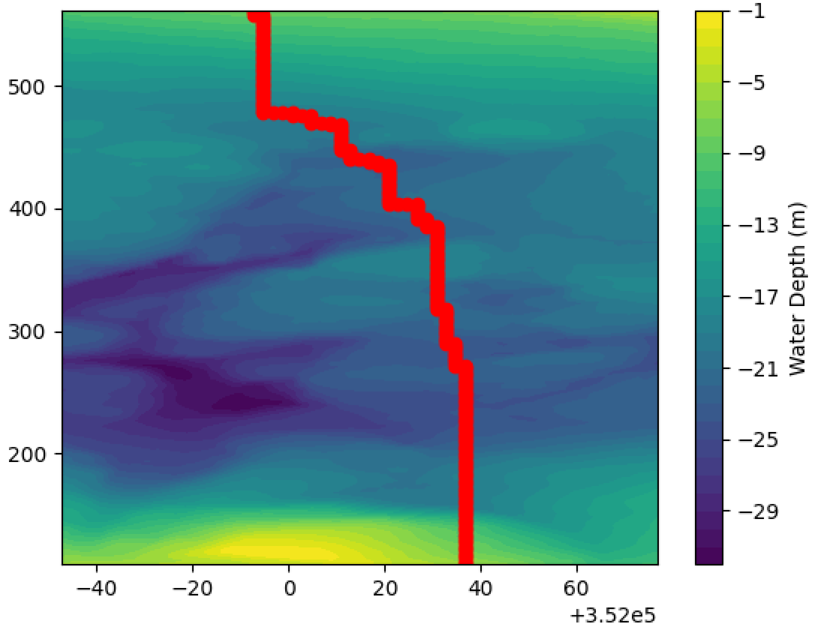


Full Data – Lee Performance Plot 1

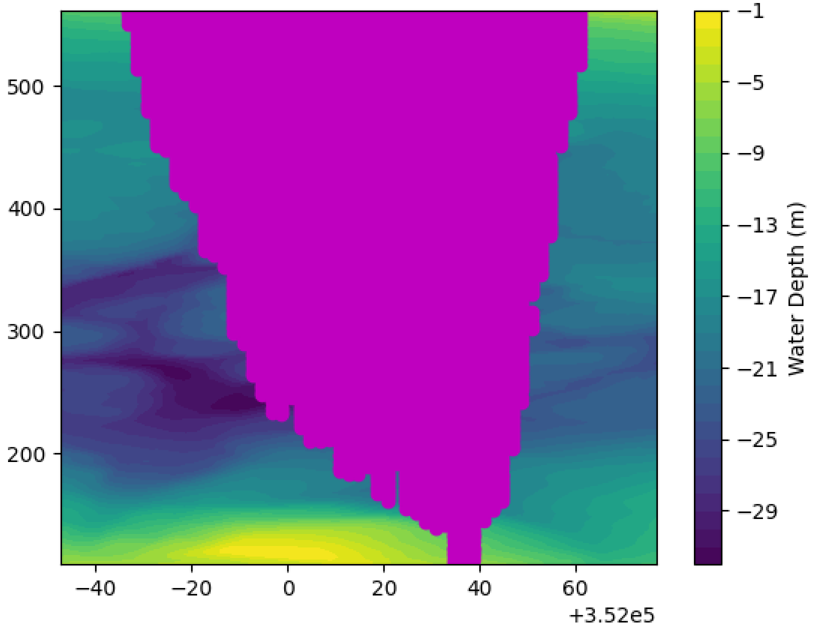


Plot 1 – A*

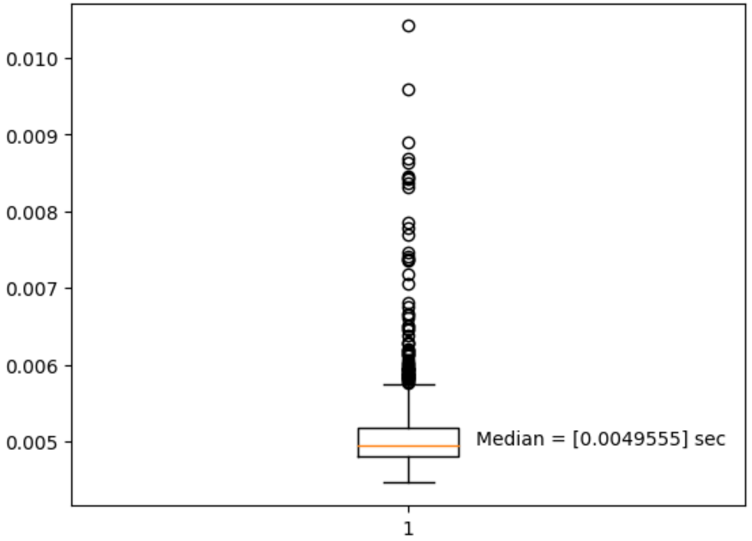
Full Data – A* Path Plot 1



Full Data – A* Visited Plot 1



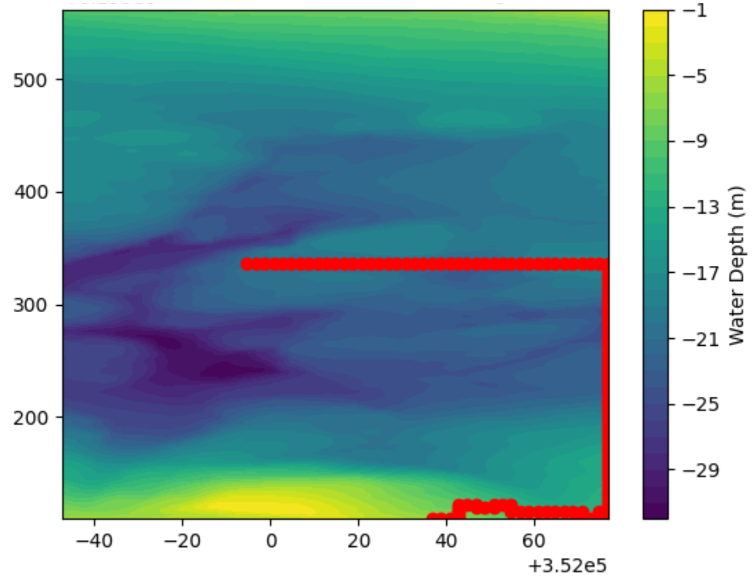
Full Data – A* Performance Plot 1



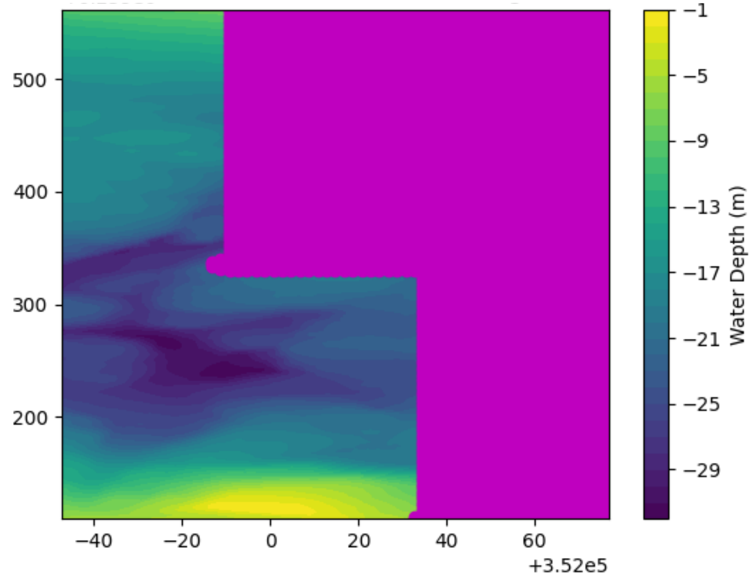
Median = [0.0049555] sec

Plot 2 – Lee

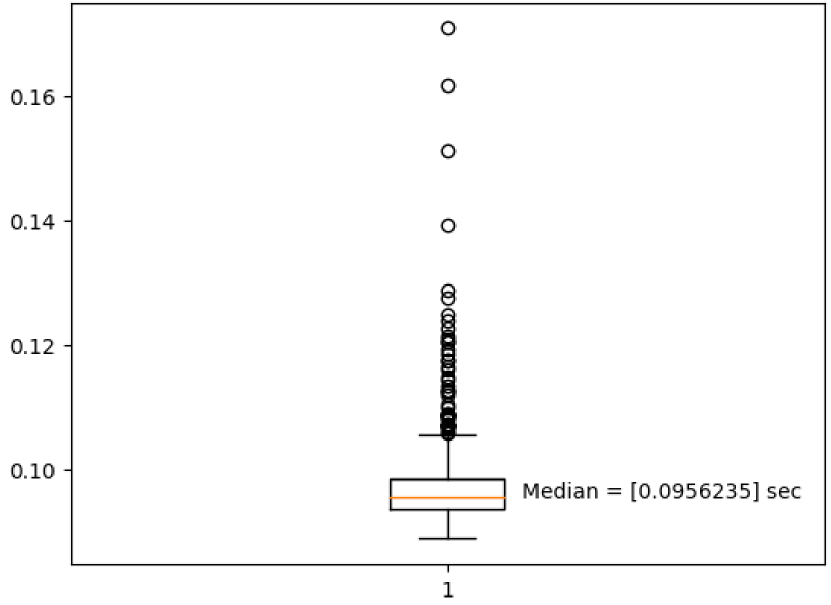
Full Data – Lee Path Plot 2



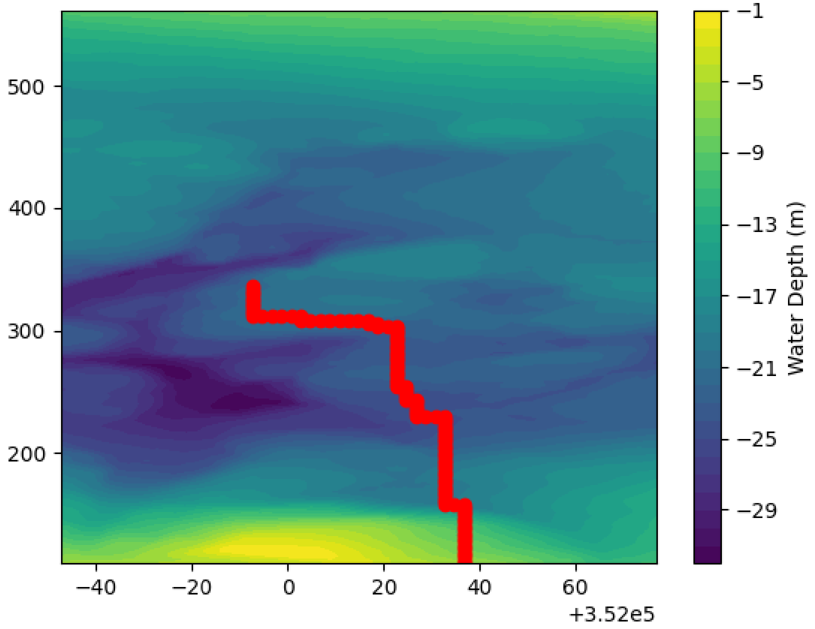
Full Data – Lee Visited Plot 2

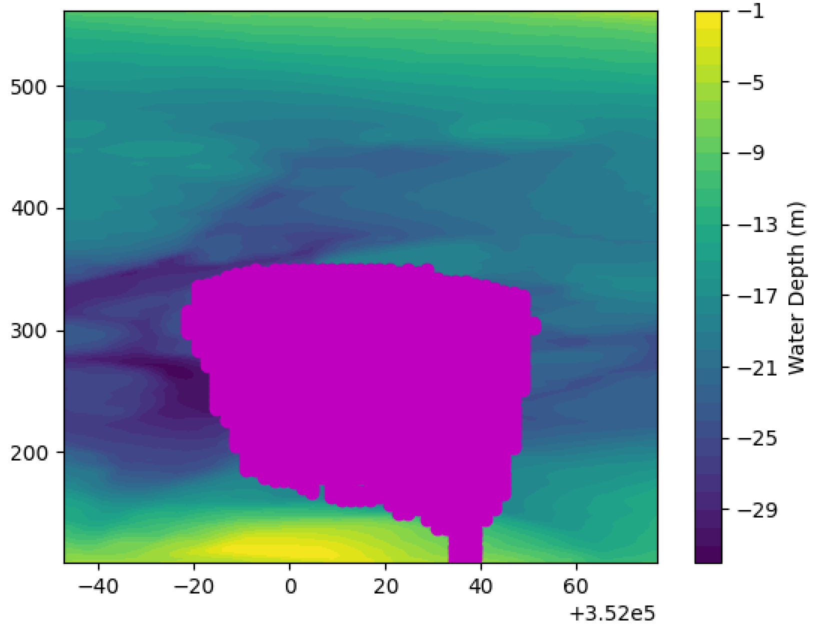


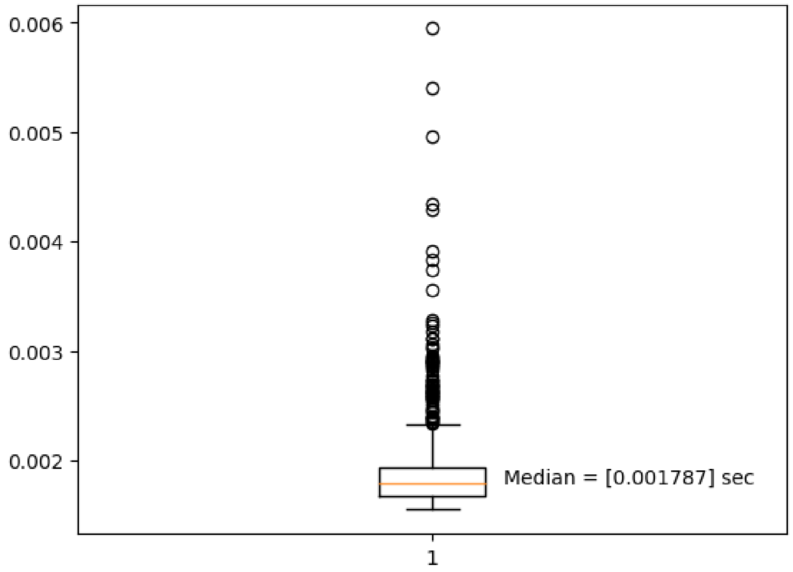
Full Data – Lee Performance Plot 2



Plot 2 – A*

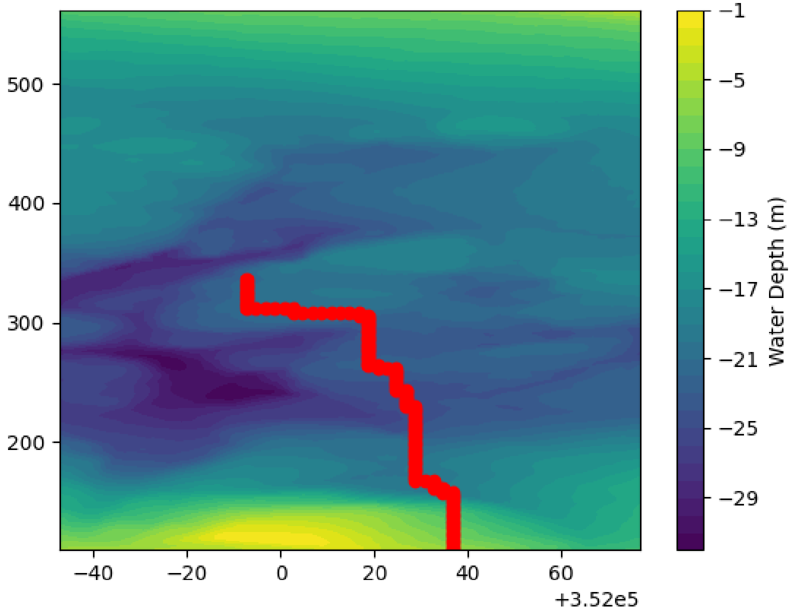

Full Data – A* Path Plot 2


Full Data – A* Visited Plot 2

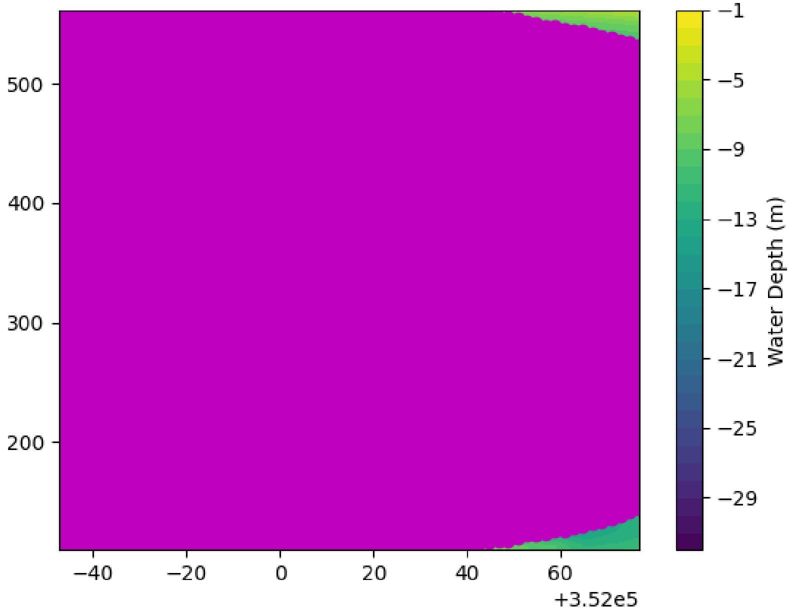

Full Data – A* Performance Plot 2

Plot 2 – Dijkstra

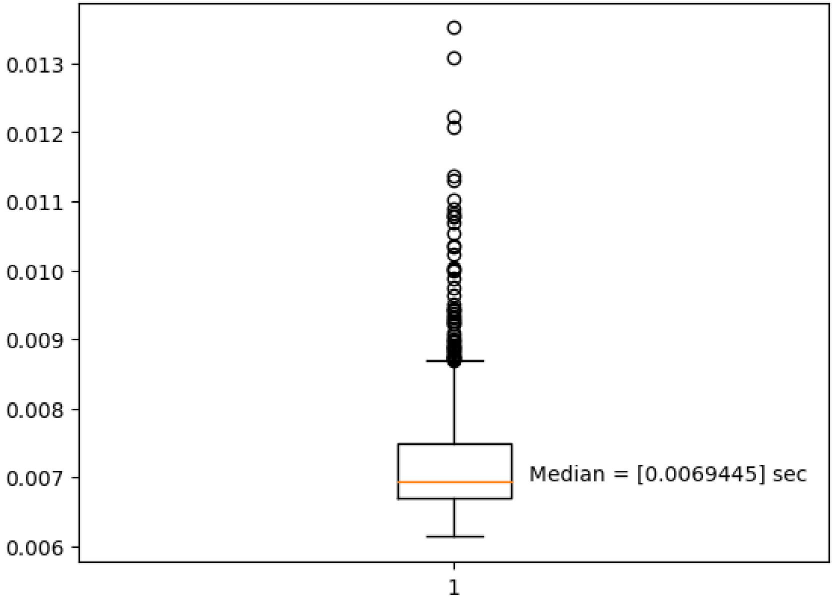
Full Data – Dijkstra Path Plot 2



Full Data – Dijkstra Visited Plot 2



Full Data – Dijkstra Performance Plot 2



This route is to confirm that the A* heuristic is finding the same route. There are some very minor differences but this does mainly match the A* path.

Summary

Algorithm	Plot 1 – Median Time (Seconds)	Plot 1 – Percentage Execution Time Increase Over A*	Plot 2 – Median Time (Seconds)	Plot 2 – Percentage Execution Time Increase Over A*
Lee	0.0691865	1296.16%	0.0956235	5251.06%
A*	0.0049555	▨	0.0017870	▨
Dijkstra	▨	▨	0.0069445	288.612%

For both Plot 1 & Plot 2 the A* algorithm was the fastest. The A* algorithm was faster than the Lee algorithm by 0.064231 sec (1296.16% increase over A*) for plot 1 and 0.0938365 sec (5251.06% increase over A*) for Plot 2.

For comparison for Plot 2 the Dijkstra algorithm was included, this was 0.0051575 sec (288.612% increase over A*).

A* Varying Data Size

Predicted Best Case:

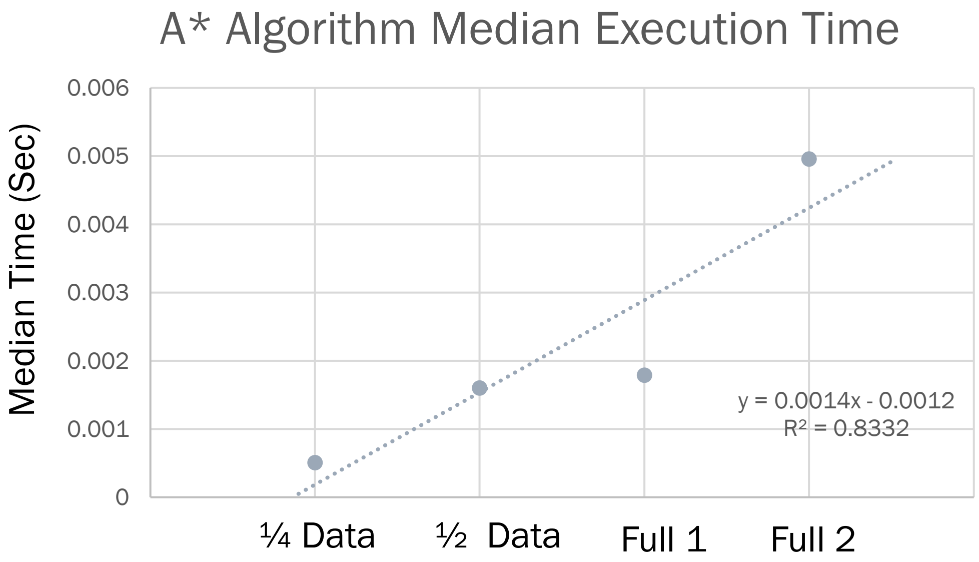
O(N)

Where N is path

Worst Case:

O(NM) (NM are grid width

And height)



This Implementation is between the 2 cases and is closer to the best case

Lee* Varying Data Size

Predicted Performance:

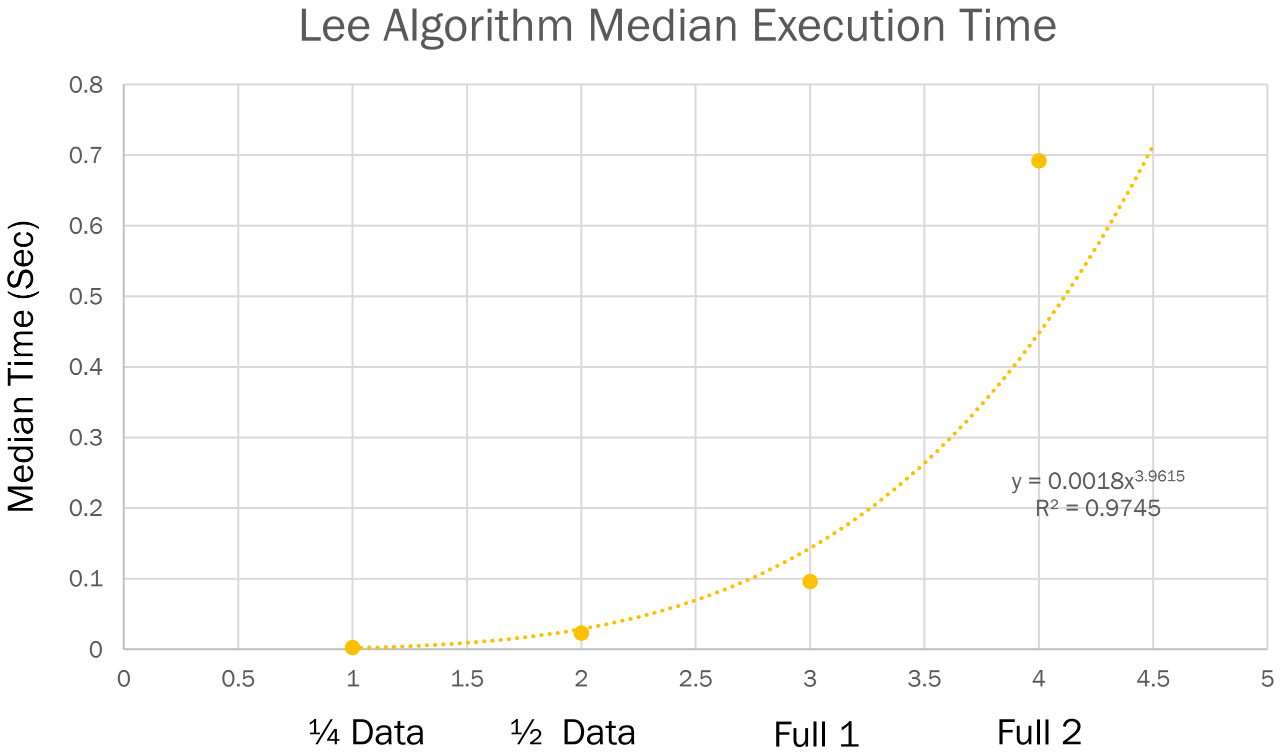
O(NM)

(NM are grid width

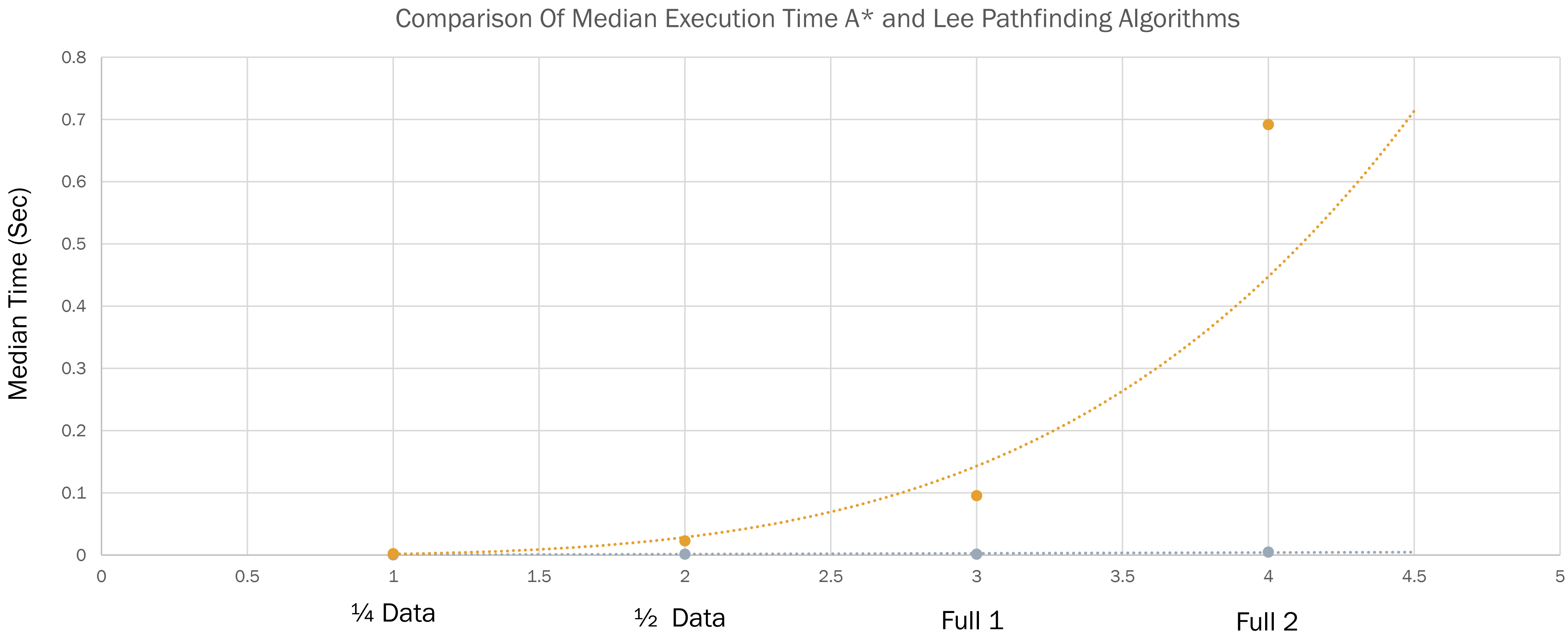
And height)

This Implementation slightly worse then the

predicted **O(NM)** especially for Plot 2



Combined Performance



Conclusion

Conclusion

The A* algorithm is the most efficient method of finding a path on the bathymetry. This implementation of the algorithm is close to the best case of O(N) when compared to Dijkstra's algorithm there is very small difference in the path found, this suggests that some further adjustments might be needed to the A* Heuristic to ensure it guarantees the shortest, smoothest path.

For certain cases to when start and end points are inline or at the dataset boundaries the Lee algorithm can be fairly efficient (e.g. Plot 1). However, as soon as the end point is at the centre of the grid (e.g. Plot 2) the algorithm runtime increases beyond the predicted O(NM).

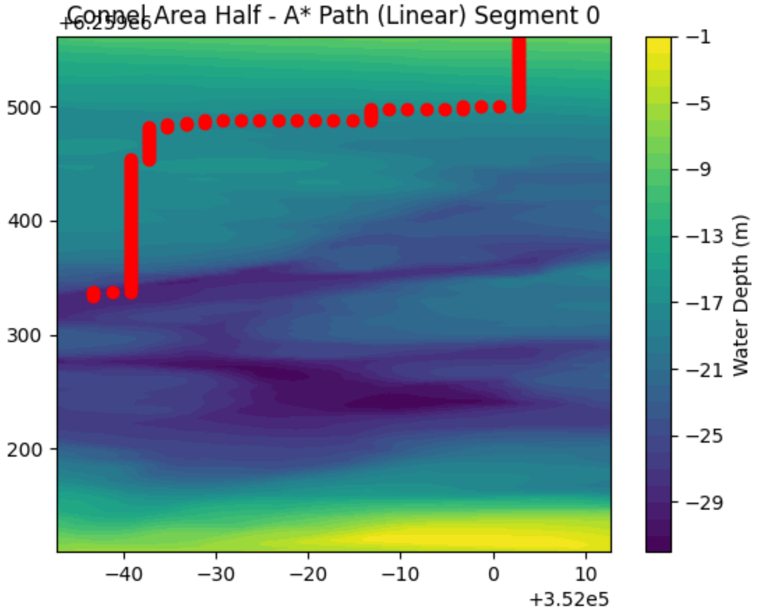
Reflection Points

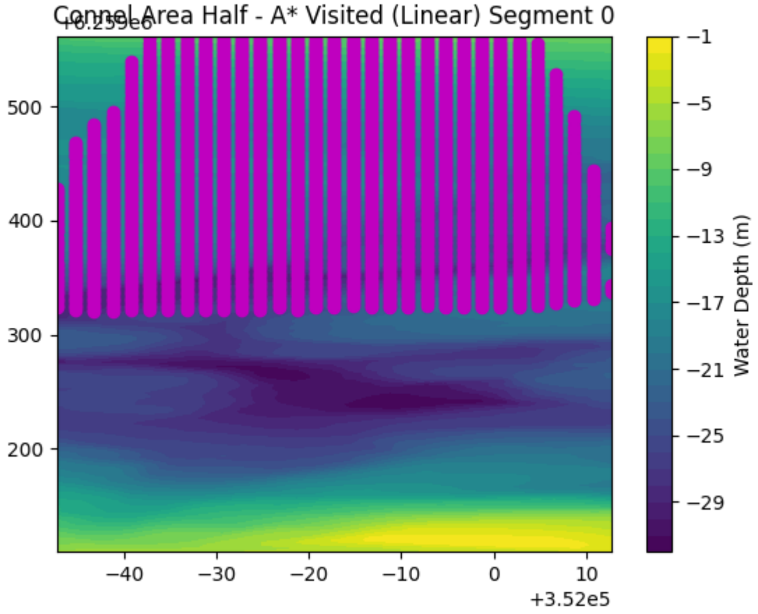
- Having the Lee algorithm terminate its search when it finds the end node was a choice to reduce the total number of nodes searched. However, this means that the guaranteed shorted path is not found.
- For data set where a path along the boundary is not wanted a weighting should be applied as part of the H-Cost calculation to check the distance from the dataset boundary.

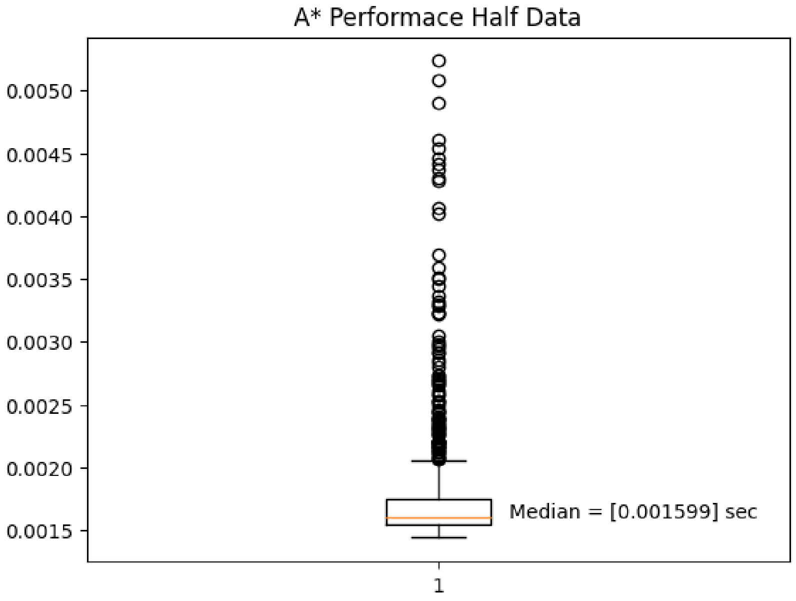
Questions

Appendices

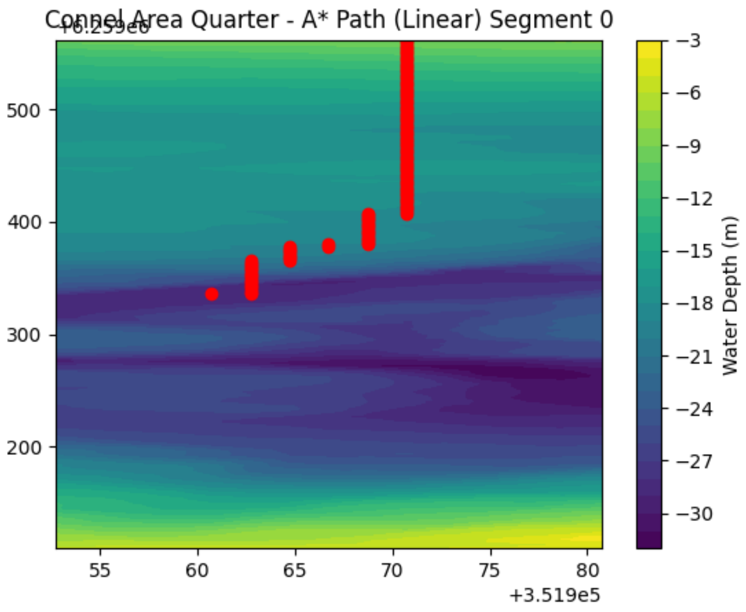
Half Dataset – A*

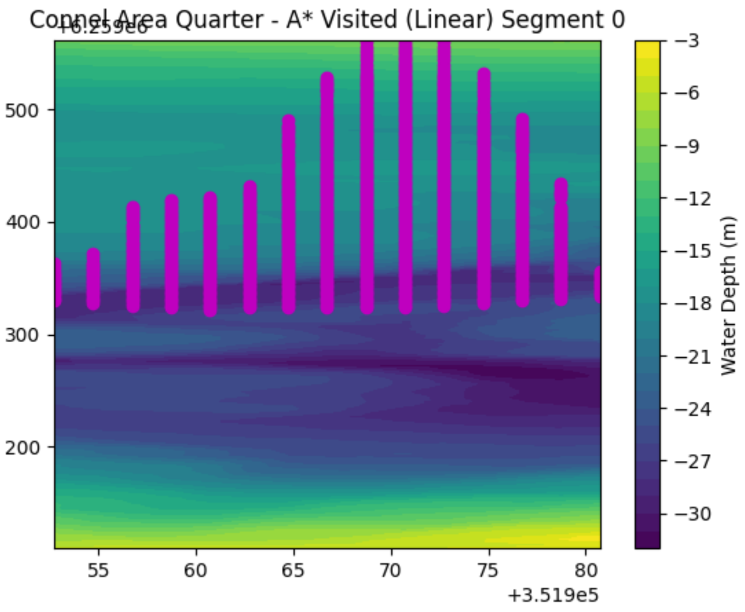

Connel Area Half - A* Path (Linear) Segment 0

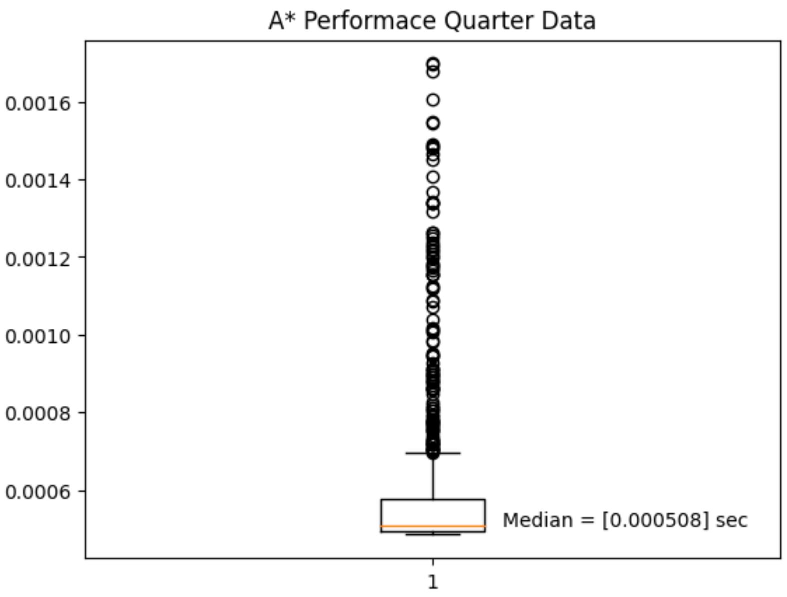

Connel Area Half - A* Visited (Linear) Segment 0


A* Performace Half Data

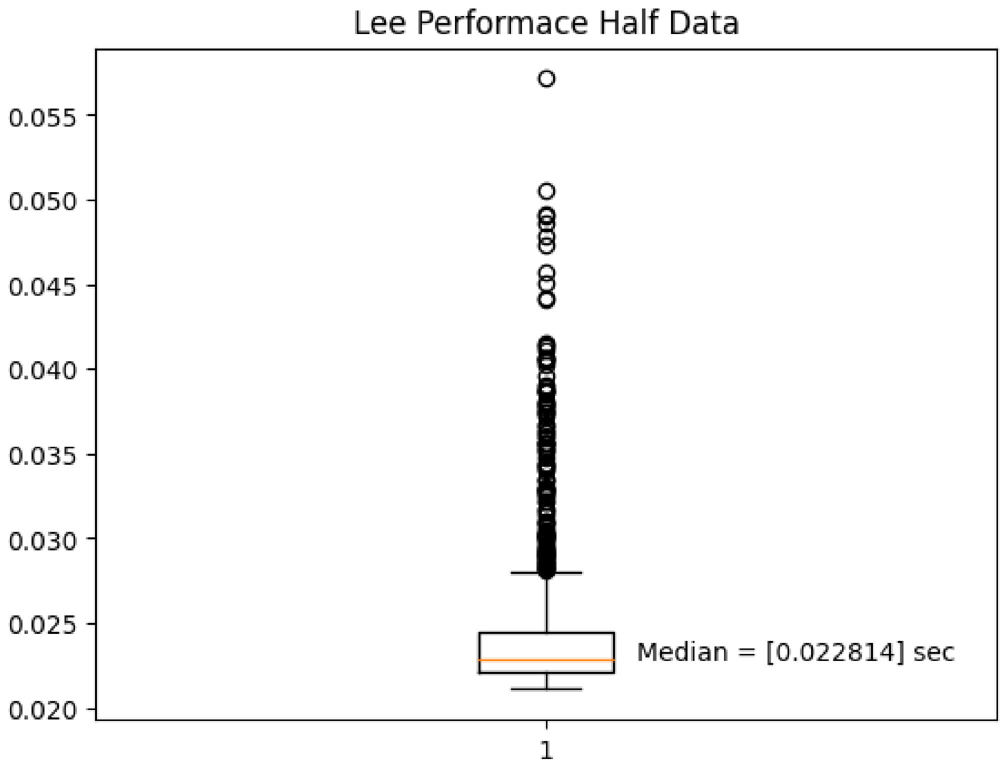
Quarter Dataset – A*


Connel Area Quarter - A* Path (Linear) Segment 0

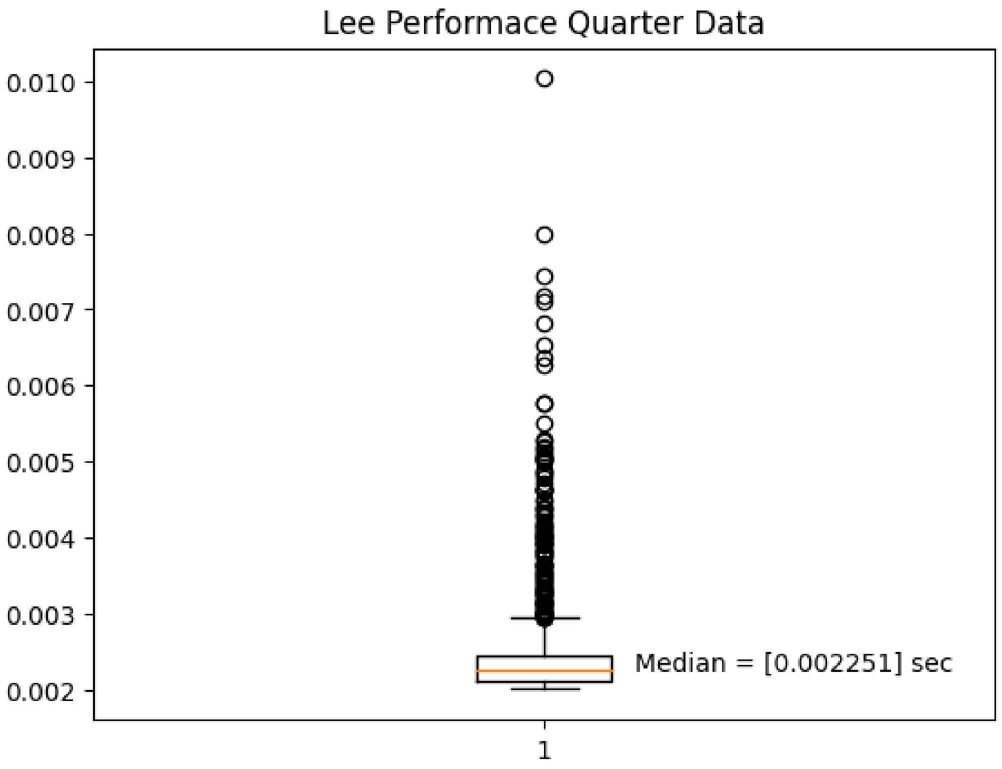

Connel Area Quarter - A* Visited (Linear) Segment 0


A* Performace Quarter Data

Half Dataset – Lee



Quarter Dataset – Lee


Lee Performace Quarter Data
Median = [0.002251] sec