Toby William Towler

Registration number 100395626

2025

Robot Mower Mapping and Pathing

Supervised by Edwin Ren

University of East Anglia
Faculty of Science
School of Computing Sciences

Abstract

The "Robot Mower" is a self-contained lawn mower unit aimed at maintaining golf courses. This project is a redesign of the movement planning systems, enhancing and automating the map generation and complete coverage path planning techniques through external libraries and machine learning methods. The complete coverage path planning module adapts a powerful C++ library designed for agricultural use complete with obstacle avoidance, to the golf course use case. Map generation is now comprised of an orthophoto or drone image input to a custom MaskRCNN PyTorch machine learning model trained on golf courses from around the world to ensure consistent and accurate results correctly idenifying all classes. In conclusion, the complete coverage path planning and aerial image map generation successfully automates the route planning issues of the Robot Mower.

Acknowledgements

Edwin Ren, Eden Attlebourgh

Contents

List of Figures

1. Introduction

The robot mower is an already existing project developed by previous masters students from the University of East Anglia. Physically, the mower has 2 tracks for movement on the sides of a metal frame, it is controlled by raspberry pi 4 running the Robot Operating System (6). Sensor wise, the robot is equipped with a 4G dongle, lidar and a GPS chip that was upgraded to an RTK chip in this iteration of the project. The existing code base was mostly written in python with very small amounts of C++. Because of this, all of my code will be written in python to slot into existing code without issue.

My contribution to this project this year will be, regarding the overall movement and guidance of the robot. For ease of planning, I have broken this down into 3 sections:

1. Basic map generation
2. complete coverage path planning
3. map generation from an aerial image

These sections are all modular meaning they can be developed, tested and function independently but still easily be integrated together for the final product. The specified use case of the robot will now be to cut golf courses, this is particularly relevant to the aerial map generation section of my work which will likely use a machine learning model and require relevant training data, while the other sections are not concerned with a real world use case as their algorithms will work be able to tweaked for any applicable use case of this robot.

2. Background and Related Work

2.1. ECOVACS GOAT A3000

The GOAT A3000 RTK (ECOVACS) is a compact self contained lawn mower created by Ecovacs. It moves its 46 cm wide body on 4 wheels cutting 33cm wide areas at a time. Boundary detection is automated with dual Lidar and AI cameras allowing accuracy within 2cm, several maps and paths can be saved at once. This robot allows for complete edge detection covering a whole area with no user input required but can still connect to a mobile app to allow adding, editing, splitting and merging of virtual

maps. Other options in the mobile app include mowing height, travel speed, obstacle avoidance height and scheduling. Many U-shaped paths are joined together to form a grid pattern ensuring all grass is cut.

2.2. STIHL iMOW 5

This robotic lawn mower (STIHL) can cover 28cm at once under a 52cm body, also on 4 wheels its physical desigin is similar to the ECOVACS product. STIHL also provide a mobile app to connect to the robot, many features are similar with the addition of definable starting points and priority zones. Additionaly this product has a rain sensor, since moisture can affect the mowing process the robot can determine if it should mow or not based on preferences by the user. It is also equipped with anti theft technology. If the robot detects it is picked up or removed from the desired area it locks all controls and alerts the owner.

3. System Design

3.1. Map Generation

Map generation is an important part of testing this system, it is important to test on all scenarios that may occur in the real world. For this reason, random or parameter based map generation is very necessary to guarantee success in every environment. As the outputs of this section will mostly be used for testing the path planning algorithm on regions with differing area, number of corners and complexity, no excessive algorithmic complexity is needed. This program should also be able to create n obstacles within the main field, such areas would represent obstructions in the mowers desired path, for example trees or telephone poles in the real world. This means we need a function with 2 parameters:

- **K**, number of angles in the outer field
- **N**, number of obstacles within the field, since it would not be sensible to take a parameter for the number of corners for every hole, we can generate them randomly assuming 3–8 corners staying inline with the complexity of the rest of the field without being unreasonably overengineered.

3.1.1. Corners

The number and distance between corners could be thought to represent complexity of a shape. It is important that both the number and placement of corners in a field are variable to simulate all situations that may occur in the real world. This leads to robust and intense testing ensuring reliability in practice.

3.1.2. Obstacles

Obstacles or holes, can be thought to represent real world obstructions for example trees or telephone poles in a real field. Generation of obstacles can be completed using the same function as the outer field generation, simply with different parameters. The algorithm 2 takes 3 parameters, number of points in the shape, an origin point and the range new points. This allows for variable size, positioning and complexity.

3.1.3. Graham Scan

The Graham Scan (4) is an algorithm to find convex hulls, that is from a set of points the outline which contains all inner points. This algorithm does this by sorting the points by their polar angle to the lowest point, since this is always in the hull. There is then further calculations based on the angle between adjacent points to omit inner points from the outline, however for this use case that is not necessary since all points will be vertexes in a field. For this reason, the algorithm used is not strictly a Graham Scan but rather heavily based on the first stage, as shown in 3 Using this algorithm allows for consistent outlining of any set of points with no crossovers or intersections

3.2. Complete Coverage Path Planning

Complete coverage path planning(CCPP) is "the task of determining a path that passes over all points of an area or volume of interest while avoiding obstacles" (12) For this module, I have used the Fields2Cover library (10), this library is open source. During the course of this project, I contributed to his library, fixing a bug during the build process. The library splits the task up into several sub-tasks.

3.2.1. Robot Configuration

Robot sizing has 2 important factors, track width, how wide the machine itself is, and blade width, how wide the utility object is. For farming equipment the utility object is usually larger than the vehicle, for example a combine harvester's wheels being narrower than blade, however for this project the blade is within the tracks. For this reason the functions will compute slightly differently to its probable intended use case as the tracks are likely to overlap however this should not cause an issue and all outputs should function as needed.

The robot has 2 more parameters regarding its turning circle, these are minimum turning radius and maximum differential curvature. Both of these are important when calculating a path for the robot to follow as they ensure the robot can easily follow the path.

The robot is 22cm wide from outer edge to edge and the gap between the tracks is 17cm, with a track width of roughly 2.5cm.

3.2.2. Headland Generation

Headlands are the area in which a vehicle turns, think of the rough edges of a crop field, these exist so that agricultural vehicles do not trample their crops when turning. Although for the use case of a golf course this is not strictly needed as there are no "crops" or areas the robot cannot touch, so in this usecase headlands are not necessary. Headlands can also generated around obstacles to allow for suitable turning area around the obstacles if needed, this serves as an area to do a "U-turn", effectively the same as reaching the end of the field.

3.2.3. Swath Generation

Swaths are the individual straight lines that make up the complete path, they are often parallel to each other, but this can vary depending on how the shape of the field is segmented into smaller shapes however they will always be parallel while in the same sub area. There are 2 ways of determining the optimal angle; the sum of the lengths of all swaths, or the number of swaths, where lower is better for each. Both of these have pros and cons, for example lower total length offers better time efficiency assuming a near constant travel speed while lower swath count gives reduced turning movements

which can be better if turning requires a slower speed or other complexities. Similarly, lower sum length of course means less distance travelled and usually less fuel/energy consumed, however the opposite can be true if the acceleration and deceleration causes greater energy consumption compared to constant velocity. Swaths are generated within the bounds of the headlands to allow for independent handling of the turning and connection of swaths regardless of the rest of the generation process. To find the best angle, each angle is tested and compared. This brute force approach could be considered slow, but it is all the library offers and could be taken into consideration in a future release.

3.2.4. Route Planning

A route is the order in which swaths will be covered. These can be sorted in a number of ways:

- **Shortest route** to cover all swaths, order depends on field
- **Boustrophedon order**, a zigzag like pattern covering swaths in order (1,2,3...)
- **Snake order**, Similar to Boustrophedon but skipping 1 swath each time (1,3,5...)
- **Spiral order**, a spiral pattern around the field, first swath then last (1, n, 2, n-1...)
- **Custom order**, user defined

While they all ensure complete coverage, each of these has their benefits:

The shortest route method uses OR-Tools (Perron and Furnon) to compute the shortest route to cover all swaths, it considers headland paths and is the only method to do so. While it will always find the quickest path, it also has the longest computation time due to the extra, sometimes complex mathematics involved.

Boustrophedon is the most efficient for rectangular shapes, traversing swaths in order minimising the number of turns and therefore wasted time and energy.

Snake order is the same as boustrophedon but provides a better solution when turning radius is large or multiple passes are needed to completely cover the area.

Spiral order can be benificial in irregular or circular areas where back and forth patterns cause a large number of unnecessary turns. It also prevents going back over already covered areas which provides greater time efficiency.

Since all of these have their specific use cases it would not make sense to pick one for all appliactions, instead it should be considered for each generation by the user.

3.2.5. Path Planning

Path planning refers to the connection of all swaths in route order. There are a couple of ways the to link neighbouring swaths: the Dubins curve and the Reeds-Shepp curve.

Both of these compute the shortest path between the 2 points, with the deciding factor between the 2 being the robots ability to move backwards. Reeds-Shepp allows backwards movement which can be helpful when headlands are not needed, allowing for mowing right to the edges of an area.

There is a derivative of both of these featuring continous curvature, this substitutes a slightly longer path for a wider turning angle but higher average speed. The advantages of continous curvature are mainly for vehicles that have difficulty turning sharply.

However, since the robot moves on tracks, it can turn in place. This means the majority of benefits for path planning techniques are invalid, instead the most time is saved by having the shortest path and least angle to turn on the spot. So the best method for this use case is likely the Reeds-Shepp as it can work with no headland and produces nearly straight lines for its links between swaths.

3.2.6. Cell Decomposition

3.2.7. Final Setup

- **Mower size**, 0.22m track wide, 0.15m working width
- **Headland generation**, no headlands - full coverage
- **Swath generation**, brute force - optimal pathing
- **Path planning**, Reeds-Shepp curve - minimizes time off of course
- **Route planning**, shortest route - seems to minimize time out of area too

3.3. Aerial Map Generation

Previously, for the program to take in a real world map from an image of the real world, the user would have to upload the picture and manually trace the outline, this could cause some problems. For example, the outline is only as accurate as the user's mouse placement, likely skipping some smaller corners to save time or due to inability to be accurate to the necessary degree. Automating this labourious and time consuming task

would positively increase user experience and usability, this can be done using algorithmic and machine learning approaches.

3.3.1. Algorithmic Approaches

There are several algorithmic methods that try to achieve this task from several different librarys. A popular library for image processing is scikit-image.

OpenCV's Canny edge detection implementation (1) follows the classic algorithm developed by Canny (2). The algorithm works through several steps to detect edges in images:

- **Gaussian Blur**: First, the image is smoothed with a Gaussian filter to reduce noise, as edge detection is highly sensitive to noise.
- **Gradient Calculation**: The algorithm calculates the intensity gradients of the image using Sobel filters in both horizontal and vertical directions:
- **Non-maximum Suppression**: This step thins out the edges by keeping only the local maxima. For each pixel, it checks if it's a local maximum in the direction of the gradient. If not, it's suppressed (set to zero).
- **Double Thresholding**: The algorithm applies two thresholds:
  - High threshold: Pixels above this are considered "strong" edges
  - Low threshold: Pixels between low and high are considered "weak" edges
- Edge Tracking by Hysteresis: Finally, weak edges that are connected to strong edges are kept, while isolated weak edges are discarded. This helps ensure that the detected edges are continuous.

This approach works very well for high contrast images, particularly those black and white Figure 1. The clear separation between foreground and background elements allows the algorithm to detect distinct gradient changes and create clean, continuous edge lines.

As for the similarly coloured golf course images, it can often find the outlines but they are made up of many smaller lines Figure 2. This fragmentation occurs because the subtle color transitions between different areas of the golf course (like fairways,

roughs, and greens) create weaker gradient signals that may fall inconsistently above or below the detection thresholds.

It may appear there is a simple solution to this problem; join the connected lines into one, however this proves to be rather problematic in itself for a number of reasons:

1. The desired line does not form a complete shape - Gaps in the detected edges mean that even sophisticated line-joining algorithms may fail to connect all segments belonging to a single boundary

2. The line is connected to other lines outside the desired shape - Many detected edges from shadows, texture variations, or other regions of the golf course can intersect with the boundary edges required to be isolated, creating unwanted connection points

3. Once lines are connected, it is impossible to tell which collection of lines is needed - Without prior knowledge of the expected shape or additional contextual information, there's no reliable way to determine which edge segments represent meaningful boundaries, like the edge of a green compared to an outline of trees or a rough area.

All of these means use of canny edge detection can often cause more problems than it solves, while it may have more accurate outlines in places, they may be connected to unwanted areas, Or they may not be complete and still require further user input to chose a selection or finish a shape.

3.3.2. Machine Learning Approach

Another possible solution to this problem is to use a machine learning network to predict the areas of the golf course. PyTorch (7) is a powerful open-source machine learning library and has become one of the most popular frameworks for deep learning research and production applications since its initial release in 2016. It was developed to show that usability and speed can both be present in a python implementation of machine learning making it the perfect tool for this task.

Model

For this specific problem, a Mask R-CNN model is best. Mask R-CNN models offer strong instance segmentation meaning they are good for outline individual parts of images to within pixels of accuracy and therefore greater precision. This type of model is ideal for golf courses since they can distinguish between the similarly coloured parts of a golf course e.g. bunkers vs fairway.

Data Set

A models training data needs to be closely related to its real world input data, for the Robot Mower that is satelite images or orthophotos from above the desired mowing area. Training data also needs to be clearly and consistently annotated to produce accurate, reliable results in practice. The only orthophoto data set publicly available is the danish golf course data set, a dataset produced by the Danish government, however common annotations lack quality and consistency; the outlines do not cover the whole area of single sections and often are missing entirely. Upon testing the data with theses annotations, it produced faulty results including both true negative and false positives Figures 3 to 4.

Since the raw orthophotos are still highquality images, they are suitable but to avoid over training on specific regions a mix of golf course images will be used in training data this will include some images form the Danish date set, Google Earth and various other sources. This should also promote more diverse predictions from data with different shadow angles, grass colour and design. Creating this larger, more diverse dataset means custom annotations can be created for all images with more consistent and higher quality boundaries than those found on the Danish data set.

Labelme (wkentaro) is an open source python adaptation of an older tool produced originally by MIT allowing users to draw around parts of an image creating boundaries and saving them as a list of points in a **.JSON** format.

One prominent issue with this data set, and probably any data golf course image, is that bunkers are far more common than any other class. This can cause over prediction of bunkers and greater uncertainty for other classes. There are several possible fixes for this the most likely are more data or class weighting.

Doubling the data count caused slightly more consistency while adding class weights was a game changer. Class weights adjusted for the less frequent classes like fairways and greens but still not enough for rough as seen in Figure 5. The lack of rough is likely because in most cases, they are large single annotations unlike the frequent smaller

bunkers which are predicted with very low uncertainty. Breaking up roughs into smaller sections covering the same area increases the frequency and diversifies the chaerectistics more.

The model's performance also varied greatly depending on the shade of green in the area. For instance the Danish image above scores well as it shares lots of characteristics with the training data but for a golf course in England with darker greens it scores poorly in comparison Figure 7.

The fix for this was more diverse training data, the model must understand that different shades or textures can mean the same class is needed while also being able to distunguish between neighbouring classes.

Training

Each class' statistics are tracked such as number of occurances and total pixel area, these values are used to adjust the weighting during training to improve accuracy and reduce uncertainty of each class. To account for the less common classes such as water features, images containing these are trained on more often with their weights boosted to ensure each class is well represented in the learning. Similar weighting is used in the loss function of each class to further emphasise their rarity, this is innacurate prediction of less frequent classes hold a greater penalty than more frequent ones. Boosting exposure to rarer classes like this means the model is well equipped to identify them if they do occur in real world use. The accuracy and loss of each class is also taken into consideration to further adjust things if needed.

Random transformations are used to increase the robustness of the model, these can happen in a number of ways:

- Horizontal flipping
- Colour jitter
- Rotiation

Using the same images with slight adjustments increases reliability and should allow the model to work well on images with varying quality but also real world factors like shadowing or sun angle, without needing a great increase in the size of the dataset.

The model starts with PyTorch's **maskrcnn_resnet50_fpn** architecture with pretrained weights on the COCO (5) dataset. This model serves as a strong backbone

as it produces high accuracy in segmentation tasks while simultaneously being able to quickly learn on new domains. Its RoIAlign layer features pixel to pixel mapping between input images and masks predictions resulting in much finer boundaries that previous generations.

As is typical for maching learning, the dataset is split into 2 parts for each epoch. 80% for training and 20% for validation to get a score of the current parameters. This approach allows for constant iteration and adjustment on the fly resulting in faster training times and a better final model.

The model uses an early stop method to save more time when possible, as mentioned before the loss of each epoch is tracked if they loss does not improve over a set number of epochs, in this case **20**. This prevents unnecessary training cycles aiming for a set number of epochs.

4. Performance Evaluation

Performance is a key part of user experience therefore is crucial to the real world performance of this product. Performance in this case can be broken down into 2 key metrics:

- **Run time**: how long the process takes to complete in real time
- **Memory Usage**: how much system memory the process uses both average and maximum

Both of these can be measured in python consistently and reliably, meaning the code can be directly tested in the same file without needing system usage or other programs that may have their own overheads. Each module will be run with different parameters to see how they impact performance which then will be run 5 times, these 5 results will then be averaged and plotted against the chosen parameter providing clear graphs to visualise any impact varying inputs may have on the products performance. A baseline run will be provided using set parameters and run 10 times to show a good default.

Run time will be measured in milleseconds and track the total execution time from start to finish. The tool used to measure exectuion time is the **time** library and its **perf_counter** function. This is accurate to nanoseconds and causes negligable difference in runtime.

Memory Usage will be measured in MB and track the average and peak memory usage during the runtime of the program. Peak memory usage is tracked using the **tracemalloc** library. This library tracks the peak memory usage of the code exectuion.

For reference, these benchmarks were taken on a **Ryzen5 5600X** with **16GB** of R.A.M. at **3200 MT/s** and where relevent **GeForce RTX 3060 Ti**.

4.1. Map Generation

4.1.1. Run Time

BaseLine

Figure 8 represents the baseline and tested with 10 corner points, 0-400 range and 0 holes. The runtime is fairly consistent having a very small range of 0.2 milleseconds, showing a very strong baseline to run further testing on.

Number of points

Figure 9 shows the impact of varying the number of points on runtime. The tests were run with a range or 0-400 and 0 holes. Clearly points are added in linear respect to execution time but are efficient in doing so, taking around 0.0075 milleseconds per point added. Points are added with O(n) complexity

Range of points

Figure 10 visualises the time taken when varying limits are placed on the range of points, this can be thought to represent the Physical size of the field. These runs were carried out with 10 points and 0 holes. There is no clear correlation between the range of points and the run time of the program, the graph closely resembles the baseline, having the same range and very similar bounds at each side. The quickest runtime from this test and all baseline runs actually occurs with a range 1200, three times the range in the baseline. This is a very good indication on top of everything else that range has no effect on runtime meaing the program should work for any size of field in good time.

Number of holes

Figure 11 demonstrates how the number of holes translate to runtime. The runs were carried out with 10 points in the main field, a range of 400 and 5 points in each hole. Similarly to the number of points, the number of holes has a linear effect on runtime,

however runtime increaes in larger steps. This is likely because in this example, each hole has 5 points and an extra append operation, this is likely why the same number of points generated in the outside field, and in total considering holes differ in runtime. For instance, 200 points in a single field takes around 1.5ms but 210 points comprised of 10 in the outer field and 40, 5 point holes just under 2ms. This time increase does not line up and could be for a number of reasons such as the additional appending to the array or the random number generation process of the point generation function, or could be the extra overheads from the sorting of the points in each hole. Whichever of these is the cause, or if it is a combination of all 3, they are all needed operations so nothing can be done to improve this further, although they whole process still completes in an insignificant amount of time, even with 200 holes it only takes around 9 milleseconds. 200 holes is likely far more than any real world appliaction would need but still computes in excellent time.

4.1.2. Memory Usage

Baseline

Figure 12 is a baseline graph to show the memory usage of the map generation process with 10 points, 400 range and 0 holes. There memory usage is quite consisten except for the obvious outlier on run 1. There is no clear reason for this but it occured on every measurement taken, it is possible it is an issue with the measurement script although unlikely.

Number of points

Figure 13 shows the memory usage of the map generation process with variable points, 400 range and 0 holes. Points effect memory linearly which make sense as there is contasnt size need for each point. The amount of memory needed is very small and is almost guaranteed to be suitable for any computer system even a low end portable micro PC such as a raspberry Pi.

Range of points

Figure 14 draws the memory usage of the map generation process with 10 points, variable range and 0 holes. Increasing the range of points does have a positively correlated effect on memory when compared to the baseline. This could be for a number of reasons but essentially comes down to the higher number possible coordinate values. That

could be the data structure of the point takes more memory to store the greater value, because the random number generation process requires more memory or becuase the sorting algorithm needs to memory to sort the higher values.

Number of holes

Figure 15 demonstrates the memory usage of the map generation process with 10 points, 400 range and variable holes. As mentioned in the runtime section, holes are closely related to points, they effect memory linearly, but at a higher rate than points as they are a collection of points. Where memory differs to runtime however, is there are no overheads to having points as holes or a single field, since memory allocation is constant. This concludes the number of holes is irrelevant and the number of total points is all that needs to be considered.

4.2. Complete Coverage Path Planning

4.2.1. Runtime

Basaeline

Figure 16 shows a baseline runtime for the fields2cover library generating with:

- **Area**, $10^3\,m^2$ and 6 sides
- **Mower size**, 0.22m track wide, 0.15m working width
- **Headland generation**, constant function
- **Swath generation**, brute force
- **Route planning**, boustrophedon order
- **Path planning**, dubins curve
- **Cell decomposition**, none

These paremters are all default, first mentioned or most simplistic to give a stable baseline.

Map Size

Figure 17 shows how differing field size affects the runtime of algorithm. The graph shows an exponential relationship between the two, this makes sense because there are a number of components affected by field area:

- Area calculations
- Buffer operations, used to generate headland
- Swath generation and optimal angle

All of these cause longer runtimes on larger areas, mostly due to the increase of calculation complexity or simply needing more calculations to complete the process. The graph shows upto $10^5 m^2$ which is about the size of 14 football pitches, taking about 8 seconds. While this is a long time to wait, it is unlikely that the program will have to consider an area this large very often since most input images are going to be close up to capture detail.

Route Planning Method

Figure 18 shows how different route planning methods impact processing time. There is not much difference betwen the 3 known patterns, almost margin of error, but the shortest route method takes much longer. This is because, the 3 knowns requrie 0 calculations just simple array indexing in a set order. Where as the shortest route method uses a brute force approach and requires vast calculations to find the best sequence, this time increase is also likely to have a non linear affect when paired with larger fields.

Number of Holes

Figure 19 Shows how the number of holes in a field affect the overal run time, this test was conducted on an 80 x 80 square field with a varying number of holes. When holes are present, the **Shortest Route** planning method must be used to connect the isolated swaths that border obstacles and as seen above this going to impact run time on its own, so it is expected that runtime will be longer than before. Holes have a linear effect on run time, this makes sense as field generation and swath placement is linear.

4.2.2. Memory Usage

Baseline

The baseline graph Figure 20 indicates constant memory usage across several executions of the pathing method. This consistency in results indicates a stable process for path generation that should be reliable and produce identical results each time it is required.

Map Size

According to Figure 21, field area has very little impact on the memory consumption. There seems to be 3 levels of memory consumption but it is unclear which parameter causes the discrepancy since the results are not in distinct groups. The range is within 30 bytes of each other, this small variation is completely normal and is likely unavoidable due to the nature of python's memory allocation system. The results of this test are negligable and can be considered consistent within a working margin.

Route Planning Method Figure 22 displays the peak memory usage when using different route plannig methods. Yet again, memory usage is hardly changed by the selected method, this seems strange especially for the shortest route application since there are so many calculations and processing overheads. The short range is a big positive and allows the route planning decision can be made purely on runtime and physical factors without worrying about system limitations, aslong as processing runtime is not a huge concern, shortest route can be used bringing big benefits to the physical use.

Number of holes Figure 23 shows the peak memory vartation with differing number of holes in the field. As mentioned in the runtime test the map for this test is slightly different, taking place on a 80x80 sqaure field with a number of sqaure holes, this approach is necessary to ensure consistent placement of holes in the field. The results show 2 things; the number of obstacles in a field do not affect memory and memory consumption is not directly affected by the shape of the field. Both of these are good results and show memory is largely similar accross any selection of parameters, consistently low memory usage is a very good trait for an embedded system such as this, meaining computation can be carried out on a variety of computers with differing specfications, specifically low spec end micro computers.

4.3. Aerail Map Generation

y axis - accuracy x axis - image size

y axis - accuracy x axis - image resolution

y axis - accuracy x axis - training data size

y axis - accuracy x axis - training epochs

y axis - precision x axis - recall

5. Conclusion and Future Work

Another essential section that should keep its title as suggested. Briefly discuss your main findings, outcomes, results; what worked and what could have been done differently. Then summarise your work in a concluding statement by comparing your outcomes against the main and sub-objectives and/or MoSCoW requirements (if used) and suggest potential future work that could be done if more time would be available.

References

[1] Bradski, G., Kaehler, A., and Pisarevsky, V. (2023). Opencv: Open source computer vision library. *https://opencv.org/*. Accessed: 2025-04-13.

[2] Canny, J. (1986). A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, PAMI-8(6):679–698.

[ECOVACS] ECOVACS. Goat a3000 lidar.

[4] Graham, R. L. (1972). An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132–133.

[5] Lin, T.-Y. and Maire (2014). Microsoft coco: Common objects in context.

[6] Macenski, S., Foote, T., Gerkey, B., Lalancette, C., and Woodall, W. (2022). Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074.

[7] Paszke, A. et al. (2019). Pytorch.

[Perron and Furnon] Perron, L. and Furnon, V. Or-tools.

[STIHL] STIHL. imow 5.

[10] Team, F. (2023). Fields2cover: Mission planning for agricultural vehicles. *https://fields2cover.github.io/*.

[wkentaro] wkentaro. lableme.

[12] Zhao, S. and Hwang, S.-H. (2023). Complete coverage path planning scheme for autonomous navigation ros-based robots. *ICT Express*, 9(3):361–366.

A. Map Generation

Algorithm 1 Point Class Definition

```
1:  procedure CLASS POINT
2:      X ← −1                                  ▷ X-coordinate initialized to -1
3:      Y ← −1                                  ▷ Y-coordinate initialized to -1
4:      angle ← −10                             ▷ Angle initialized to -10
5:      procedure CONSTRUCTOR(x, y)
6:          this.X ← x
7:          this.Y ← y
8:          this.angle ← −10                    ▷ Default angle value
9:      end procedure
10: end procedure
```

Algorithm 2 Generate random points

```
1: function GENPOINTS(num, P, size)
2:     points ← []
3:     for i ← 0 to num − 1 do
4:         randX ← random_integer(P.X + 1, P.X + size)
5:         randY ← random_integer(P.Y + 1, P.Y + size)
6:         points.append(Point(randX, randY))
7:     end for
8:     return points
9: end function
```

Algorithm 3 Sort points by polar angle to origin

```
1:  function SortPoints(points, origin)
2:      hull ← [origin]
3:      Sort points by Y-coordinate
4:      for i ← 0 to length(points) − 1 do
5:          points[i].angle ← CalcAngle(origin, points[i])
6:      end for
7:      Sort points by angle
8:      Append points to hull
9:      return hull
10: end function
```

Algorithm 4 Main function

```
1: function Main
2:     hull ← []
3:     origin ← Point(20, 20)
4:     field ← GenPoints(20, origin, 400)
5:     hull.append(SortPoints(field, origin))
6: end function
```

Algorithm 5 Main function with holes in shape

```
1:  procedure Main
2:      hull ← empty list
3:      origin ← Point(20, 20)
4:      field ← GenPoints(20, origin, 400)
5:      add SortPoints(field, origin) to hull
6:      hole1Base ← Point(100, 100)
7:      hole1Points ← GenPoints(5, hole1Base, 50)
8:      add SortPoints(hole1Points, hole1Base) to hull
9:      hole2Base ← Point(150, 50)
10:     hole2Points ← GenPoints(3, hole2Base, 30)
11:     add SortPoints(hole2Points, hole2Base) to hull
12: end procedure
```

B. Aerial Mapping

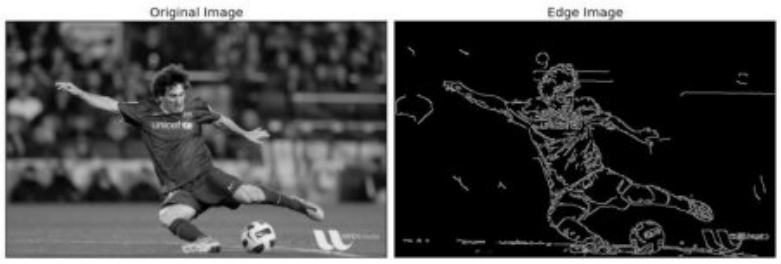

Figure 1: Opencv example of canny detection, source (1) docs

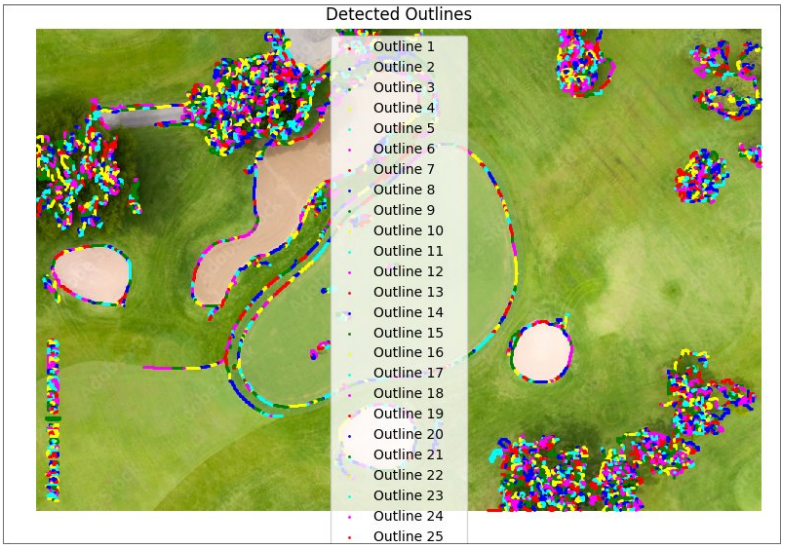

Figure 2: OpenCV canny edge detection on an image of a golf course

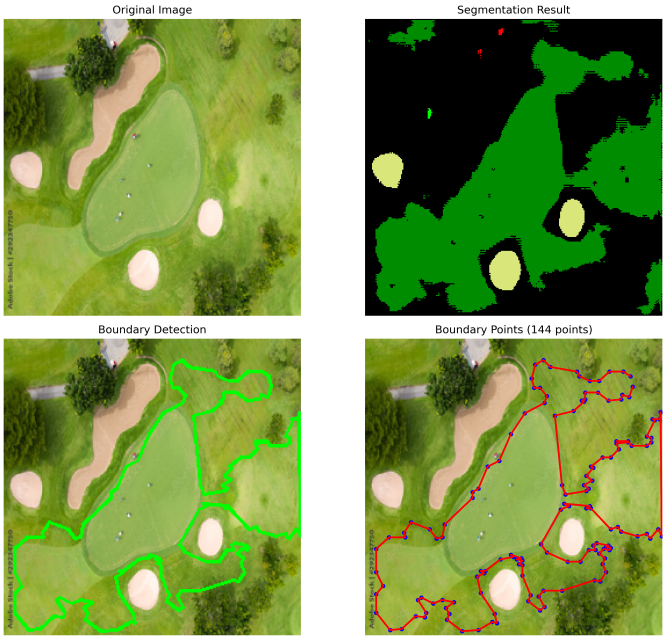

Figure 3: An example ouput of pytorch model trained on the Danish Golf Course data set

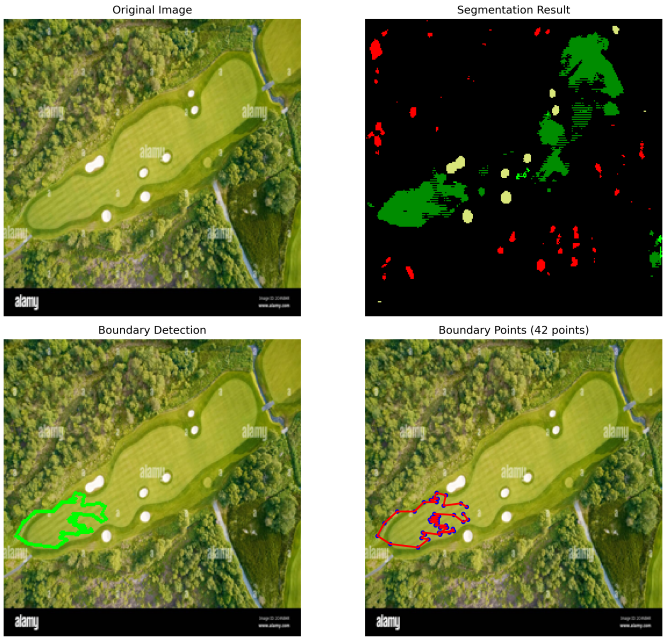

Figure 4: An example ouput of pytorch model trained on the Danish Golf Course data set

Figure 5: Output of model trained on 30 inputs with uncertainty at 0.7 and class weighting

Figure 6: Output of model trained on 30 inputs with uncertainty at 0.7 and class weighting

Figure 7: Output of model trained on 30 inputs with uncertainty at 0.7 and class weighting

C. Performance and Evaluation

C.1. Mapping

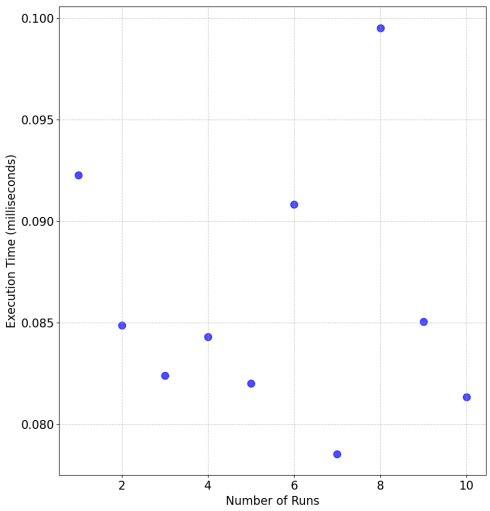

Figure 8: Runtime of the map generation algorithm with 10 points, 400 range, 0 holes

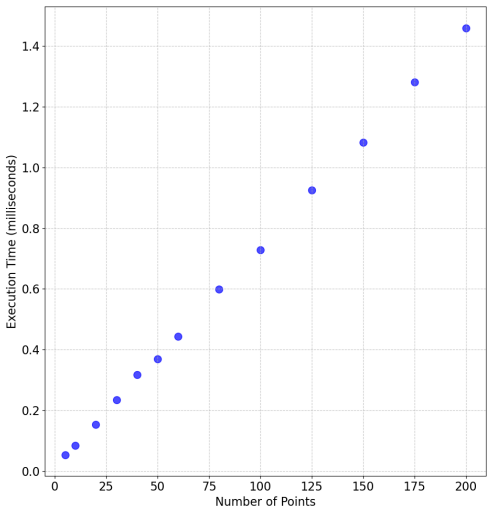

Figure 9: Runtime of the map generation algorithm with variable points, 400 range, 0 holes

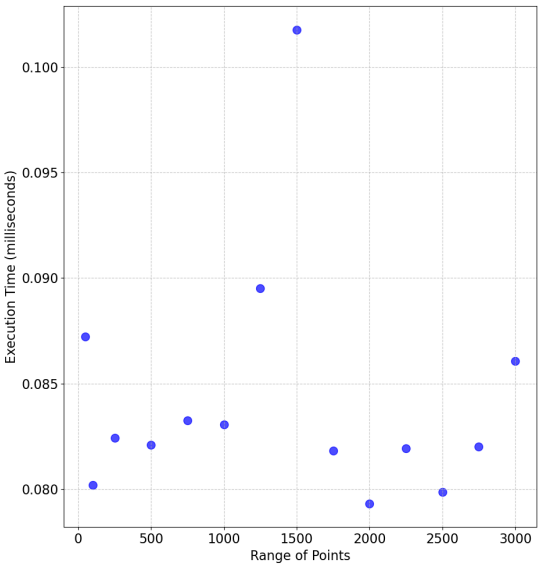

Figure 10: Runtime of the map generation algorithm with 10 points, variable range, 0 holes

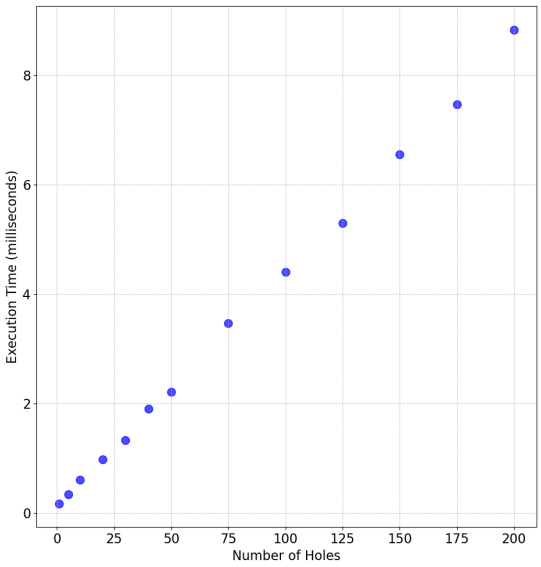

Figure 11: Runtime of the map generation algorithm with 10 points, 400 range, variable holes

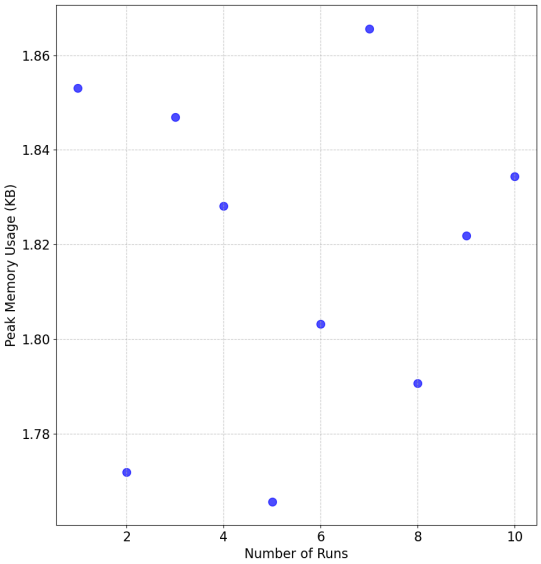

Figure 12: Memory usage of map generation algorith with 10 points, 400 range, 0 holes

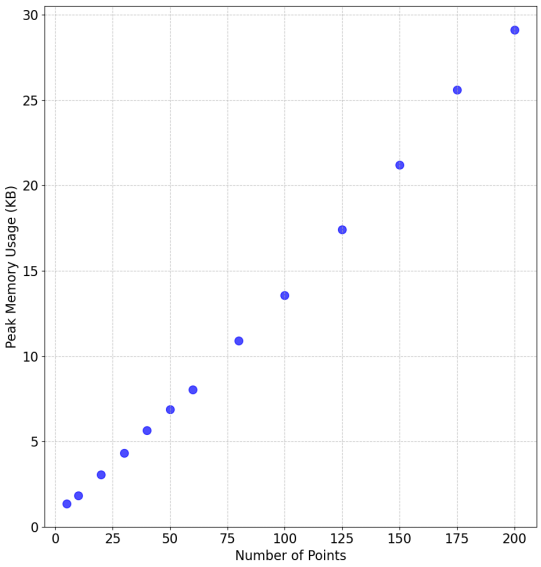

Figure 13: Memory usage of map generation algorith with variable points, 400 range, 0 holes

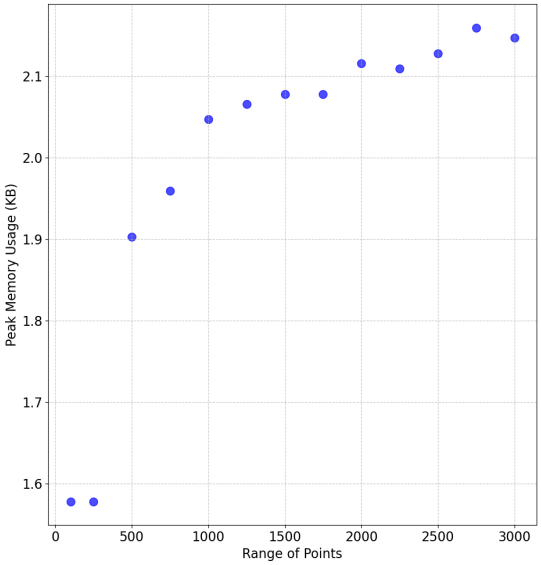

Figure 14: Memory usage of map generation algorith with 10 points, variable range, 0 holes

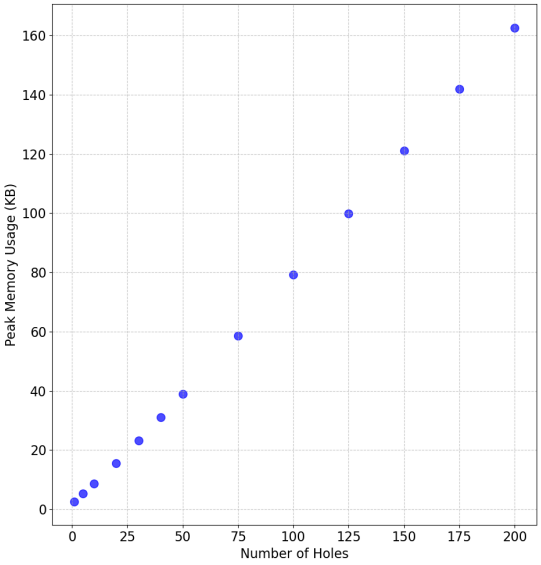

Figure 15: Memory usage of map generation algorith with 10 points, 400 range, variable holes

C.2. Pathing

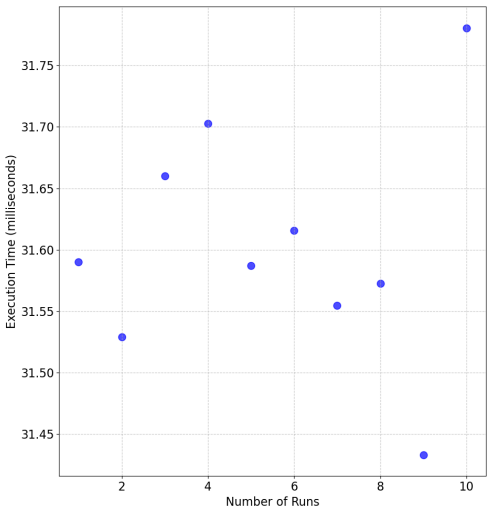

Figure 16: Runtime of path planning system with $10^3 m^2$ area, 6 corners, accurate robot size, constant headland, brute force swath gen, Reeds-Shepp pathing and shortest route planning

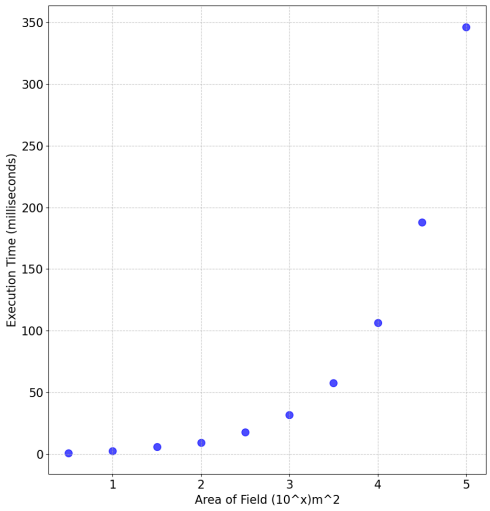

Figure 17: Runtime of path planning system with $10^3 m^2$ area, 6 corners, accurate robot size, constant headland, brute force swath gen, Reeds-Shepp pathing and shortest route planning

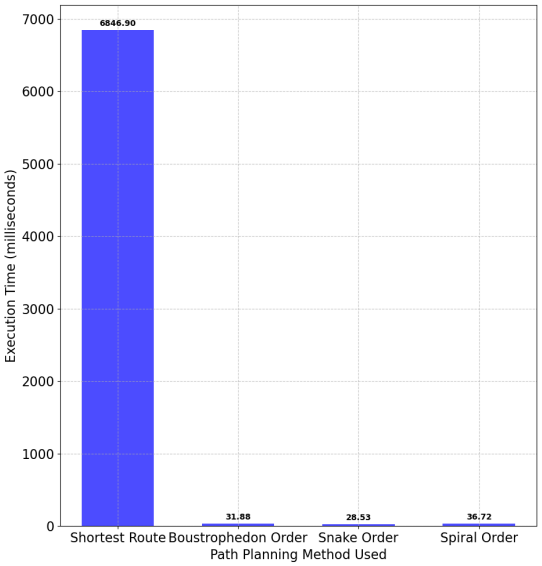

Figure 18: Runtime of path planning system with $10^3 m^2$ area, 6 corners, accurate robot size, constant headland, brute force swath gen, variable pathing and shortest route planning

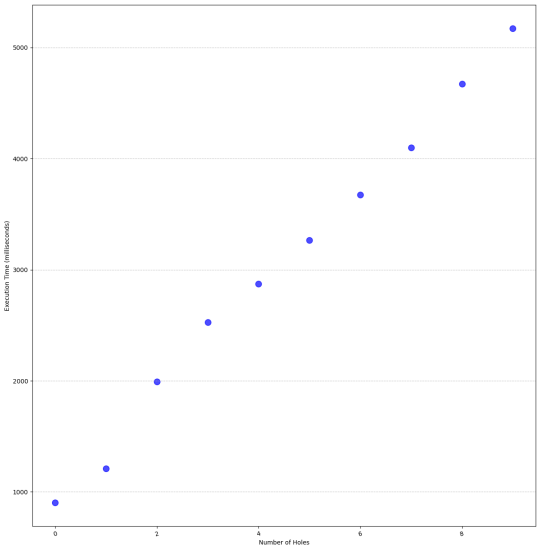

Figure 19: Runtime of path planning system with $10^3 m^2$ area, 6 corners, accurate robot size, constant headland, brute force swath gen, Reeds-Shepp pathing, shortest route planning and variable holes

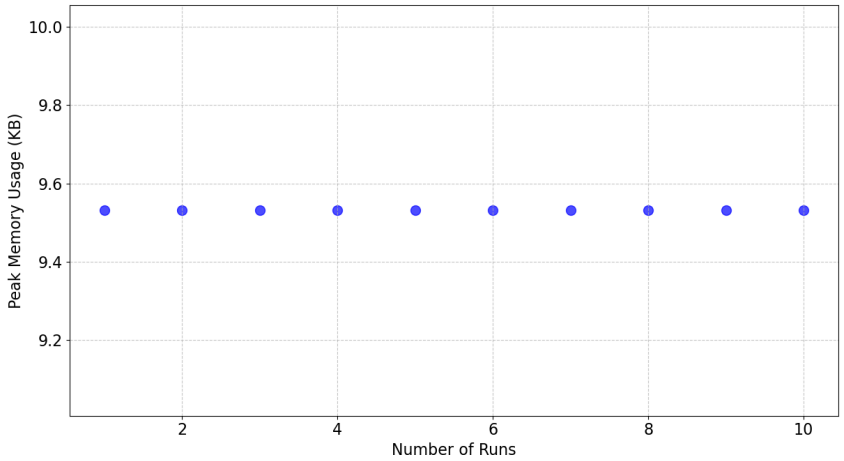

Figure 20: Memory Usage of path planning system with $10^3 m^2$ area, 6 corners, accurate robot size, constant headland, brute force swath gen, Reeds-Shepp pathing and shortest route planning

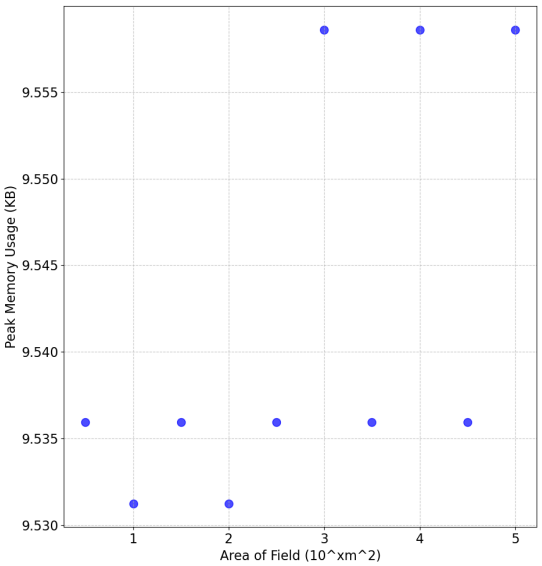

Figure 21: Memory Usage of path planning system with variable area, 6 corners, accurate robot size, constant headland, brute force swath gen, Reeds-Shepp pathing and shortest route planning

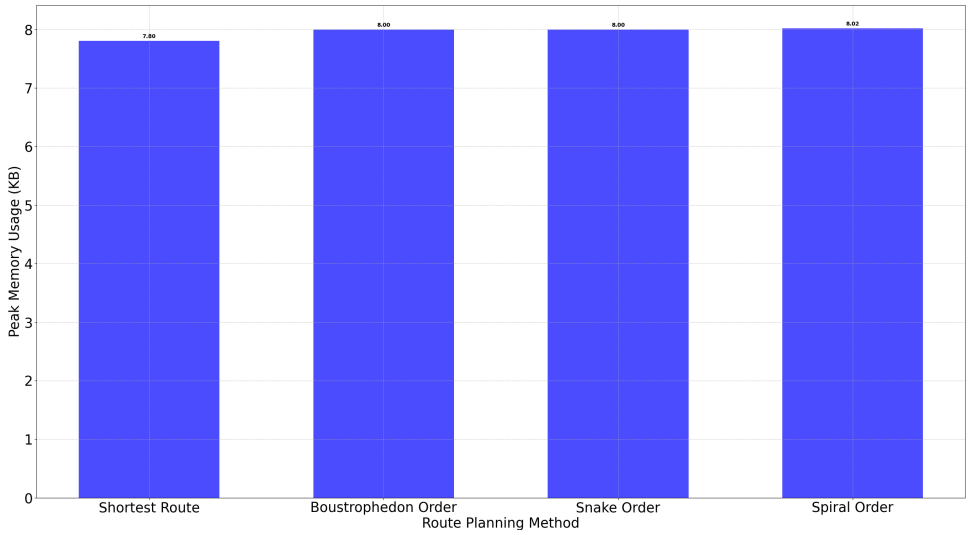

Figure 22: Memory Usage of path planning system with $10^3 m^2$ area, 6 corners, accurate robot size, constant headland, brute force swath gen, variable pathing and shortest route planning

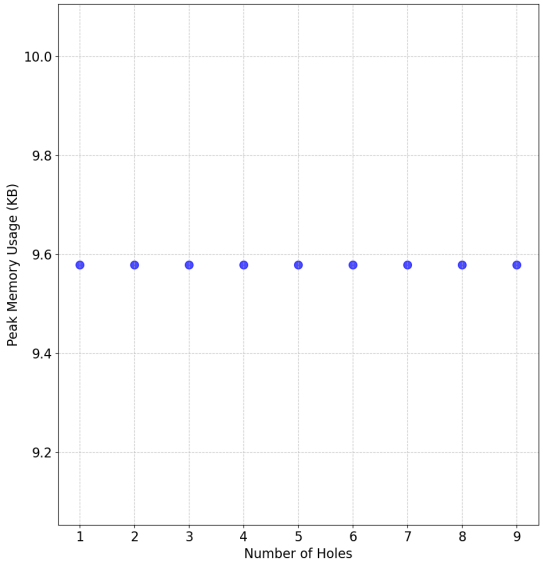

Figure 23: Memory Usage of path planning system with $10^3 m^2$ area, 6 corners, accurate robot size, constant headland, brute force swath gen, Reeds-Shepp pathing, shortest route planning and variable number of holes