Long-Term Sensing in Aquatic Environments using Autonomous Yachts

Ben Kirwin
ODCSSS Student
University College Dublin

Abstract

This paper presents a novel algorithm for short-course sailboat navigation through a complex environment. Many of the desired applications for autonomous yachts, such as water quality or sea depth monitoring, require the yacht to operate in shallow coastal waters. Navigation in these areas is made difficult by both constantly changing winds and by physical barriers like islands and shoals. Our algorithm takes into account both sensor and map data and quickly calculates a near-optimal route. This algorithm has been implemented in a Java program and tested successfully on two different robotic sailboats. This paper provides both an explanation of the algorithm as well as details of its implementation.

1 Introduction

Collecting environmental data is more important now than ever before, whether one is tracking changes in climate, pollution levels, or sea life. For this data to be most useful, it must be collected over a wide area and a significant length of time – and ideally at a reasonable price. Unfortunately, however, current sensing platforms do not satisfy these requirements. Manned missions are generally expensive and tedious. Fixed sensors, while individually inexpensive, require large numbers to cover a significant area. For these reasons, unmanned aquatic robots have become a more popular platform for research. Since most of these operate underwater they often suffer from problems with slow speed and power starvation, restricting the area and length of time for which it can collect data.

One solution to these problems is to deploy sensors on autonomous yachts. Since a yacht can move around while expending relatively little energy, it has the potential for a much greater range than a fixed sensor or a motor-powered vehicle; furthermore, since a sailboat has access to both wind and solar power, it can produce energy to power itself and any onboard environmental sensors [1].

The robotic yacht is still a young technology. While a few groups have already demonstrated autonomous sailing for short periods, these projects lack a few of the characteristics required for a long-term sensing platform. One element these boats tend to lack is a robust navigation system. While many of these craft are capable of sailing to a waypoint based on the wind, they do not generally try to avoid obstacles in their path. Conversely, there have been a number of obstacle-avoidance algorithms developed for different fields, but these do not account for wind information in the way a sailboat must. This project, then, is an attempt to take the best of these earlier

results and build on them to create a complete navigational system.

The Mighty Poseidon *encounters a swan.*

2 The *Mighty Poseidon*

The Roboboat project at University College Dublin has developed the hardware for two different autonomous yachts. The first is the *Mighty Poseidon*, a converted remote-control model boat used primarily for testing in relatively sheltered water. The second is a modified Laerling, a boat originally designed for teaching children to sail. While the Laerling is the more likely candidate for a long-term research platform, it was determined to be an inconvenient for development both because the hardware had yet to be finalised and because it is much more time-consuming to deploy. The *Mighty Poseidon* was therefore chosen as the primary testing platform for this project.

2.1 Hardware

The *Mighty Poseidon* is a customized Robbe Estelle, a model yacht produced in Germany. The Robbe Estelle is about 1.1m long, with a 37 cm beam – unusually wide for a model boat of its size. This both increases the stability of the craft and offers more interior space for new hardware. It is possible to control this boat with a remote from shore like a standard RC sailboat. A switch on this remote transfers control to a Handy Board microcontroller onboard the yacht. The Handy Board receives input from a GPS, compass, speed sensor and wind vane; it also controls actuators that adjust the sails and rudder. It is connected to the shore by a serial radio link.

2.2 Software

At the beginning of the project, the software on the *Mighty Poseidon* was already capable of maintaining a constant compass course send to it from ashore. However, it does not have the processing power to run a pathfinding algorithm on a complex map. It was decided the boat would feed the sensor data to the shore, where a notebook would run the pathfinding and send the course information back to the boat.

At first, due to concern about the reliability of the radio link, it was planned to send the course information as a series of waypoints. In this way, if the connection was lost as the *Mighty Poseidon* came to a waypoint it would already have the next destination stored and thus be able to sail on uninterrupted. Not only was this precaution unnecessary – the radio link had much better

signal strength and range than expected – it actually decreased the software's reliability. In variable winds the software needs to regularly recalculate, and each recalculation can involve sending several waypoints over the radio link. This overloaded the link, which caused an unacceptably high level of packet loss. A search for alternatives led to the idea of tracking waypoints exclusively ashore and simply updating the course over the radio link when it needs to be changed. Since this involved sending fewer and shorter packets, reliability was improved immensely.

Other improvements to the *Mighty Poseidon*'s software were made over the course of the project as well. Among the most significant of these were splitting up the program into several processes and reimplementing the methods used to send and receive packets. This sped up the program by ~40%, reduced the code size by ~20%, and increased the reliability of the radio link.

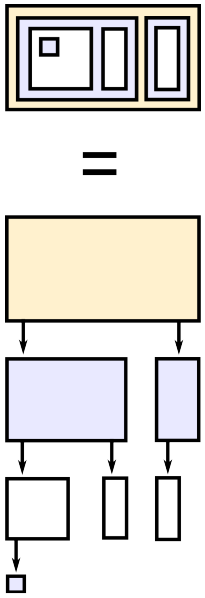
3 Shore Software

The shore-based software includes the pathfinding algorithm, a GUI for displaying program and sensor data and collecting user input, and various mechanisms for communicating with the radio module. This program was developed from scratch over the course of the project.

3.1 Mapping System

Early in the planning stages for this project, it was realized that to test the pathfinder in a meaningful way one would need both a map to test on and a way to display the map and the paths found. These were therefore the first aspects of the program developed.

3.1.1 Map Representation



Map structure.

This software treats a map as a series of polygons, or 'coasts', organized into a tree structure. If one coast is contained within a second, it is considered a child of that second coast (note that since each coastline defines the perimeter of an entire island or body of water, no two will ever overlap). This tree structure has proved useful for various tasks, from drawing the map to checking if a certain point is in a particular body of water.

These maps are stored in human-readable text files. Each text file contains the overall dimensions of the map, followed by a list of coasts (in the order that a pre-order traversal of the tree would produce). For each coast, the file records the number of children, the type ('water' or 'land'), and the list of waypoints making it up.

3.1.2 Map File Creation

In the current implementation, this map must be created by a human user. In the original design, the user had to find waypoints along the coastline manually on a chart and enter them in. However, this was found to be too labourious a process. An auxiliary script was written to take a set of paths from a .kml file exported from Google Earth and format them as a map file. While the user must still create the original .kml file, Google Earth's path-creation process is both quick and intuitive, so maps can now be produced much more quickly and accurately. While it may ultimately be

desirable to completely automate this process, that was decided to be too large a task to fit within the scope of the current project. It should also be noted that using a human-defined map does not reduce the ultimate autonomy of the robot, since in either case the map remains static for the time the boat is on the water.

3.2 Pathfinding Algorithm

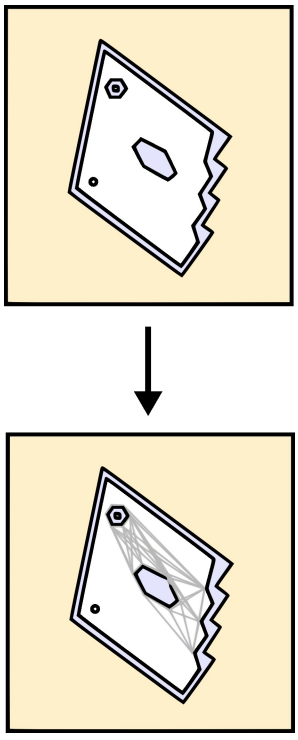
There are three discrete stages to the pathfinding algorithm. The first is a preprocessing stage. When the program is informed a boat is sailing in a given body of water, it creates and stores a pathfinding graph for that body of water. The second stage is called whenever a new leg is added to the route. It adds the start and destination points of the route to the graph, then runs the A* algorithm on the result. This generates a series of waypoints, which is again stored. The third, called whenever the boat reaches a destination or the wind changes, first checks if it is possible to sail directly to the next waypoint. If so, the program instructs the boat to sail directly there; if not, the program finds a new point outside the no-sail zone and directs the boat towards that. Given the knowledge it has at a given time, this algorithm guarantees a shortest route.

Note that the following will use planar geometry as an approximation of the ocean surface. This is generally quite a good approximation, given that most journeys take place over a relatively small portion of the earth's surface. Nevertheless, if more accuracy is desirable, the following can be reformulated in spherical geometry without significantly different results.

Note that while the result of the algorithm is the shortest path, this does not necessarily imply it is the quickest path. A sailboat is a complex system, and it is perhaps likely that in some situations, a different route might result in a quicker journey. However, in the circumstances for which this algorithm was designed (coastal areas with reasonably homogenous currents and winds throughout) straight-line paths have been shown to be faster than alternative steering methods, such as maximising the velocity towards the destination [2].

The proofs of various mathematical or geometrical assertions have been omitted due to space constraints. The omitted proofs can either be found in the references or are trivial enough to be left to the reader.

3.2.1 Generating the Graph



To generate a pathfinding graph, one must first choose which points will become nodes of the graph. Fortunately, it can be mathematically demonstrated that the shortest route between two points (ignoring wind) will consist of straight lines, the endpoints of which must be either the start point, the destination point, or a convex vertex of one of an obstacle (or a concave vertex of the containing polygon). Given the coast containing the sailboat, the program iterates through the vertices marking the coast and then through those of each of its children, collecting the ones that may form part of the shortest route. It then connects every pair of points with a direct line between them. While this

is a slow operation (taking time proportional to the cube of the number of vertices, in the naïve implementation) it needs to be performed only once each time the program is run. The graph could easily be stored in the map file as well, but the preprocessing time is still short enough (a small fraction of a second) that this step seemed unnecessary.

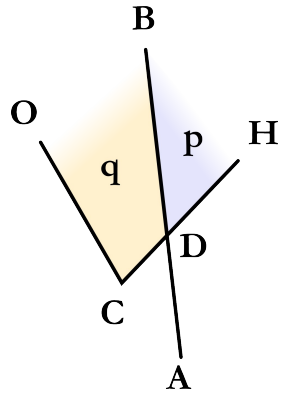
3.2.2 A*

The second stage is responsible for calculating a shortest path through a weighted graph. For this, the program uses the A* algorithm, first described in 1968 [3]. A* is one of the few most popular pathfinding algorithms, and for good reason. Not only does this algorithm have the desirable property of always finding the shortest path between two nodes; but it (provably) checks the fewest number of nodes of any algorithm that uses the same heuristic (method of estimating the distance between each node and the goal) [4]. Luckily, an effective heuristic is available: Euclidean distance, which is easily calculable and already defined to be the shortest possible route between two points. The remaining challenge is to convert A*'s output, which doesn't account for wind, into something that the boat can actually sail.

3.2.3 Tacking Algorithm

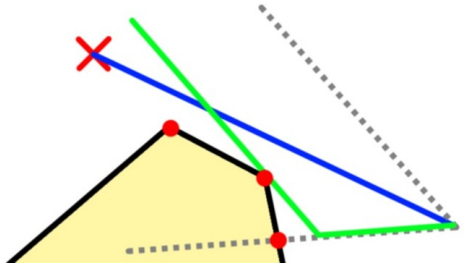
The tacking algorithm first checks if it is possible to sail directly from the current position to the destination. If it is, it instructs the boat to do so.

The rest of the tacking algorithm is based on the following diagram:



In this diagram, A and B are waypoints found in the second step, C is the current position of the boat, H and O are the closest sailable headings on either side of B (usually the port and starboard tacks), D is the intersection, and p and q are regions of the plane.

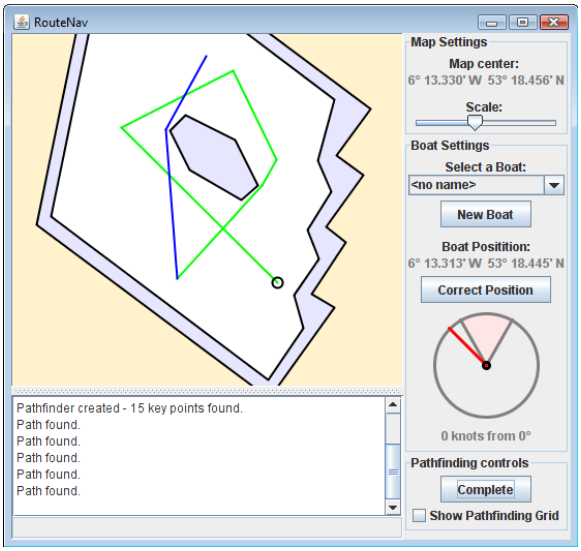
The algorithm makes use of the fact that since A and B are connected nodes in the graph, the line AB must be clear of obstacles. The boat can travel for as far as it likes on one tack (H, for example) – as long as it can return to the line segment AB by travelling on the other tack, the overall route will be an optimal one. For each heading, the algorithm finds the maximum distance it can travel without: a) crossing a land boundary, b) having land block the route back to AB, c) overshooting the goal or d) travelling too far from AB (to avoid attempting to cross an ocean in a single tack, for example). c) and d) are relatively simple math. To perform the more complex a) and b), the algorithm first determines which obstacles are in areas p and q, and then decides how far it can go without colliding with the land. The following diagram illustrates the latter step.



The red points are the relevant points of obstacles in the tacking region (p or q). Note how the boat avoids getting trapped behind the land mass.

Once this step has been completed, the program has to choose which heading to take. There are several methods of arbitration between the two, the simplest of which is simply to alternate between them on successive tacks. However, the current implementation selects the tack that it can sail on for the longest without changing direction. This has proven to give the best results in both simulation and testing.

3.3 GUI



The pathfinder running on a map of UCD lake.

There are three sections to the GUI. The upper left section (an instance of a Map class) displays the current map and any mappable objects, such as the boat. In an early implementation, the Map class contained both drawing methods for every type of drawable object and the logic to determine when they should be drawn. This proved to be cumbersome and prone to bugs. Now, it simply maintains a list of objects it wishes to draw, which are all required to implement the 'Mappable' interface. When the map redraws itself, it simply requests each item in that list to draw itself, in order.

This area is also used for selecting coordinates. Other sections of the program can capture and use this coordinate information for various things, such as adding waypoints. If nothing else captures this information, the click recenters the map. The scroll wheel on the mouse can be used to zoom in and out. This interface is similar to those used by Google Maps or Google Earth, and was found to be intuitive to most users.

The bottom left section is a text input and output area, much in the vein of standard in and out on the command line. This style was chosen partly because of its familiarity to most programmers and engineers and partly to reduce the number of menus displayed on the sidebar and the use of pop-up menus (which slow down the program dramatically on older machines).

The control panel, on the right, offers various map, boat, and pathfinding controls and information. The Map Settings area simply displays the coordinates of the map center and offers a scrollbar to display and change the zoom level to those without a mouse wheel.

The Boat Settings menu first contains a list of currently added boats, as well as a facility for adding new boats to the list. This allows the user to control several boats from the same program, should he or she wish to do

so. This section can be used either to control a physical boat or a simulated one for testing purposes. Below these, there is a graphical display of the current wind direction (including the no-sail zone around it) and the boat's current heading. It also prints the speed and direction of the wind in knots. This control can be used for setting the wind speed and direction while in simulation mode.

The Pathfinding Menu allows the user to add waypoints to the boat's path. It also lets the user display the pathfinding grid, which is useful for both debugging and demonstration purposes.

4 Future Work

This software is only one of the necessary components for a fully autonomous yacht. It has also been implemented in a fashion more suitable for testing than for deployment. While the pathfinding system was written to be fairly agnostic about the surrounding implementation, so that the relevant code could be easily added to a more complex system, this has not yet been done.

One of the simplest and most useful improvements would be to have the pathfinder running onboard the boat. This would remove most of the dependence on the radio, so that the boat could sail for an arbitrarily long time without requiring instructions. While the *Mighty Poseidon* has too simple of a processor to run the pathfinder on a significant map, either the controller could be upgraded or the program could simply be converted to run on the Laerling, which has a full PC already available.

In the long term, it would also be desirable to calculate long-term routes based on tradewinds, weather forecasts, currents and tides. This would enable it to perform longer missions over wider areas, but seriously increases the complexity of the algorithm.

Another obvious addition to the system would be a method of recognising dynamic obstacles on-the-fly (other ships, for example). While the algorithm should be able to be modified relatively easily to take in this new information, gathering the information is not a simple process. Radar systems have high power demands, making them impractical for a small, self-powered craft, and optical recognition is challenging even in well-controlled environments. Nevertheless, if these obstacles were to be overcome, this would be a very valuable addition to any navigation system.

5 Conclusion

Obstacle avoidance is one of the key challenges for autonomous yachts. This project has demonstrated that it is possible, given relatively simple and easily available technology, to build a robot capable of navigating through relatively complex environments in a variety of wind conditions. This brings robotic sailboats one step closer to being a useful – or perhaps even the best – platform for collecting environmental data at sea.

6 Acknowledgements

Sincere thanks are extended Aaron Quigley, Gabriel Muntean and all those involved in organising and running the ODCSSS program – it has been an excellent experience. Thanks also to Simon Dobson, Lorcan Coyle and Olga Murdoch for their supervision and mentorship, and to the Roboboat team, particularly Brian Mulkeen and Caoimhín Ó Briain, for all their help in moving my work forward.

References

1. Cruz, N. A.; Alvez J. C. (2008). “Ocean sampling and surveillance using autonomous sailboats”. *Journal of the Österreichische Gesellshaft für Artificial Intelligence*. **2** (27): 25-31.

2. Stelzer, R.; Pröll, T. (2008) “Autonomous Sailboat Navigation for Short Course Racing”. *Robotics and Autonomous Systems* **4 (**7): 604-614.

3. Hart, P. E.; Nilsson, N. J.; Raphael, B. (1968). “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. *IEEE Transactions on Systems Science and Cybernetics SSC4* **4** (2): 100–107.

4. Dechter, Rina; Judea Pearl (1985). "Generalized best-first search strategies and the optimality of A*". *Journal of the ACM* **32** (3): 505–536.