Task and Motion Planning For Autonomous Underwater Vehicles (AUVs)

Xavier O'Keefe XAVIER.OKEEFE@COLORADO.EDU
Department of Aerospace Engineering Sciences
University of Colorado Boulder

Abstract

In this project I present the design and evaluation of a task and motion planning algorithm for a submarine, as part of work with the Colorado RoboSub club. This year the club is converting a previously remotely-operated submarine into a fully autonomous one, and task and motion planning is an essential part of this process. We intend to compete in the RoboNation RoboSub competition, which attracts some of the most prestigious universities from all around the world. This competition typically involves 3-4 tasks, all of which must be completed autonomously without human interference. The planner was implemented and shown to produce paths that were of reasonable distance, consistency, and validity 100% of the time, while obeying the specifications of the tasks.

1 Introduction

Underwater Unmanned Vehicles (UUVs) provide access to deep underwater environments that humans can not explore on their own. Similarly to space exploration, this allows for a multitude of science missions that allow access to a vast range of knowledge about our oceans and lakes. UUVs have been used to map the ocean floor, maintain underwater infrastructure, and monitor marine life populations. This field is poised to grow greatly over the next few years, as advances in sensing technologies, robotics, and autonomy will allow for a much wider range and complexity of missions.

There are two main classes of UUVs: Remotely Operated Vehicles (ROVs) and Autonomous Underwater Vehicles (AUVs). ROVs are tethered to the surface, and require that an operator controls the robot at all times. ROVs are currently used when missions are more complex or require a human to make decisions, however this is becoming less popular. AUVs are taking over the field, as companies and scientists are eager to automate their sub-sea missions. AUVs are untethered and instead run off of their own power. This results in an easier platform to use that requires smaller ships to launch. AUV operators are able to simply place their robot in the water, have it run the mission on its own, and then retrieve the data upon robot return. This allows for missions to be run much more often, and for the robot to execute the same tasks repeatedly at the same level.

The RoboNation RoboSub competition is an annual competition held in southern California that draws talent from elite institutions all around the world. Each year, tasks are created that test a vehicle's ability to perform tasks and make decisions autonomously. These tasks are representative of typical AUV applications in the real world, and therefore provide students with opportunities to gain valuable technical skills.

This paper applies task and motion planning to AUVs in order to implement a high-level task planning framework that allows the submarine to traverse the RoboSub environment and visit each task. The Rapidly Exploring Random Tree (RRT) algorithm was used to plan paths between waypoints, and an LTL formula was used to ensure that the robot completes all tasks correctly.

2 Problem Description

The first step is to create and define a planning environment, along with some declared objectives. The robot will be modeled as a point robot for simplicity, and will operate in a three-dimensional workspace. The robot must visit all three stations, s_1, s_2, and s_3, and attempt each stations task. If any of the tasks fail, the robot must re-visit the task until it is complete. Once the robot has completed all three tasks, it must go to the designated goal state. First, this process was solved with a simple planner that chose its next destination by finding the nearest station, and traversing all stations repeatedly until every task was finished. After this was accomplished, sharks were introduced into the pool, requiring that the robot visits the hospital before visiting station 2, if attacked by a shark. An LTL formula was developed to express these constraints, and a planner was built to enforce them. The environmental characteristics are given below.

Workspace Specifications

Workspace (WS)	$[-10, 10] \times [-10, 10] \times [-10, 10]$

Obstacles

$\mathbf{O}_1$	$(0.0, 0.0, 0.0)$, $r = 0.25$
$\mathbf{O}_2$	$(0.5, 0.5, 0.5)$, $r = 0.15$
$\mathbf{O}_3$	$(1.0, 1.0, 0.0)$, $r = 5.0$
$\mathbf{O}_4$	$(5.0, 5.0, 6.0)$, $r = 3.0$
$\mathbf{O}_5$	$(-5.0, -6.0, -4.0)$, $r = 3.0$

Path and Locations

Initial State	$(3.0, -5.0, 0.0)$
Goal State	$(0.0, -9.0, 0.0)$
Station 1, s_1	$(9.0, 9.0, 9.0)$
Station 2, s_2	$(-9.0, -9.0, -9.0)$
Station 3, s_3	$(-9.0, 9.0, -9.0)$
Hospital	$(-9.0, 0.0, 0.0)$

Here are the major steps taken to complete this project.

- **Motion Planning:** Implement a motion planning algorithm to plan through the workspace. The function should take in two waypoints, a start and a goal, plan a path, and output this path to a log file for plotting.
- **Plan for consecutive tasks:** The planner needs to visit all stations and attempt all tasks. Task completion will be simulated, and the robot must re-visit failed tasks. Once all tasks have ben completed, go to the final state.

- **Develop and integrate LTL with task planner:** Each task in the RoboSub competition is quite separate - the tasks do not have any dependencies on other tasks or items in the environment. In order to force the sub to handle more complex environments, I will introduce sharks into the pool, requiring the robot to heal itself before continuing on its path. Develop an LTL formula and automata for this. Track each proposition as the robot moves through the tasks. Use these elements to determine the robots current state in the automata, and then determine if the robot is in an acceptance state. Terminate when this happens.

3 Methodology

3.1 Motion Planning

The motion planning layer of this project needed to quickly find paths between two abstract points in the workspace. The workspace was sparsely populated with obstacles and quite large compared to the size of the submarine. The motion planner also had complete knowledge of the environment and obstacles. Path quality is not of great concern as the submarine is not under extreme time pressure. The submarine needed to plan paths often, so a fast planner was preferred. These constraints led me to choose RRT for its quick, sampling-based approach. The motion planning layer was designed to easily plan between points in the environment, and as such it was implemented to take in two points and return a path between the two.

An important decision I made in the development of this project was to use a purely geometric motion planner with no respect to any dynamics equations. This is because the equations of motion for the submarine are unknown - and any attempt to replicate them would be complete guesswork. I could have used a simple integrator propagator function, like we did in homework 9, but this was also deemed unnecessary because the submarine is able to move in any direction at some velocity, so it is able to follow any generated path. Using this path on the physical robot would require that the submarine keeps its nose tangent with the path, and simply follows the x, y, and z elements of the path, however this will not be difficult to implement in software.

The RRT planner was implemented using the OMPL library with custom collision checking. The planner was modified to output path data to a log file. Planning was done in the state space, with the robot using an SO3 space. The orientation quaternion of start and goal states was held constant for simplicity. The state space is as follows, with position $\vec{x}$ and orientation $\vec{\omega}$:

$$\vec{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \qquad \vec{\omega} = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} \tag{1}$$

3.2 Consecutive task planning

The first method of task planning took quite a simple approach, but one that would work in the constraints of the robosub competition. Many of the tasks are not dependent on one another, so there is no need for any sort of temporal logic. RoboSub tasks are also quite challenging for an autonomous robot, so there is a possibility that a task may take too long, and that the best choice for the robot is to move on to another task. Creating conditions for timing and completeness of tasks quickly made LTL formulation and automata following unwieldy, and a simple queue was determined to provide a simpler and equally effective task plan.

The implementation for this system is as follows. The robot is aware of its environment, and chooses the closest station as its first task. The robot computes a motion plan with the motion planner, and attempts to complete the task. The task was assigned a value of complete, with probability p, or failure. Success is representative of completing the task in competition, and failure is representative of failing the task or running out of time on the given task. The robot then proceeds to the next closest stations, attempting each task. Stations with failed tasks were stored in a separate list, and re-tried until all tasks succeed.

Algorithm 1 Consecutive Task Plan

```
 1: function planConsecutive
 2:     $time \leftarrow$ GetCurrentTime()
 3:     $firstIndex \leftarrow time \,\%\, nAgents$
 4:     GetMotionPlan(Current State, Stations[firstIndex])
 5:     if $taskSuccess$ then
 6:         $completeTasks \leftarrow Stations[firstIndex]$
 7:     end if
 8:     while completeTasks.size != Stations.size do
 9:         $time \leftarrow$ GetCurrentTime()
10:         $index \leftarrow$ ClosestStationIndex()
11:         GetMotionPlan(Current State, Stations[index])
12:         $taskSuccess \leftarrow time \,\%\, p$
13:         if $taskSuccess$ then
14:             $completeTasks \leftarrow Stations[firstIndex]$
15:         end if
16:     end while
17:     GetMotionPlan(Current State, FinalDestination)
18:     return Path
19: end function
```

3.3 LTL Planning

In order to make the submarine's task planner more robust an LTL formula was developed, and the resulting automaton was used to ensure the robot met all conditions and reached an acceptance state. The RoboSub tasks are independent of one another, as mentioned earlier, so a more complex scenario was developed. The robot must visit 3 stations, s1, s2 and s3. Sharks have been added to the pool this year, and the robot must handle them gracefully. Station 2 requires that the robot is sterile and healthy. If a shark attacks the robot, the robot must go to the hospital to be sterilized before visiting station 2. In this problem task completion was simulated again, with tasks having probability 33% of failure. This problem has been formulated with the following LTL formula and variables:

s_1**:** True if s_1 has been reached, false otherwise.
s_2**:** True if s_2 has been reached, false otherwise.
s_3**:** True if s_3 has been reached, false otherwise.
*attack***:** True if the sub is has been attacked, false otherwise.
*healthy***:** True if the sub is healthy, false if sharks attack.

$$F(s_1) \wedge F(s_2) \wedge F(s_3) \wedge G(\text{attack} \implies (\neg s_2 \, U \, \text{healthy}))$$

As the code was developed, it became clear that the attack and healthy variables were always inversely correlated, so they could be modeled as one single proposition. The actual automata implemented in the code is below:

$$F(s_1) \wedge F(s_2) \wedge F(s_3) \wedge G(\neg s_2 \, U \, \text{healthy})$$

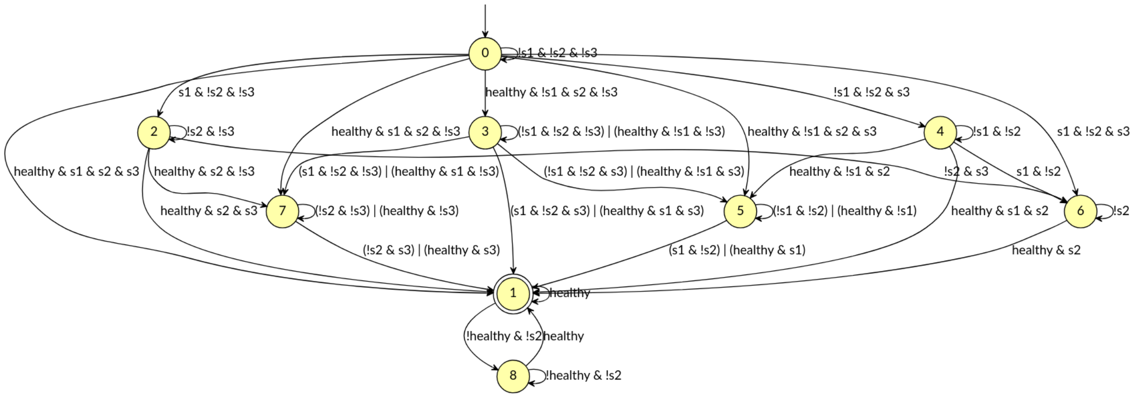


Figure 1: Visualization of the automata, produced by Spot (Duret-Lutz et al., 2022).

The implementation of this formula was somewhat similar to the consecutive task planner. Code was written to enable easy modifications, so long as the new automaton uses the same variables. The user need only to modify the *GetNextState* function with their automaton's HOA, and add/remove states form their respective enums and functions, as neccessary. This requires modification in only 4 places. The first action of the robot is chosen, with equal probability, from options: proceed to state 1, proceed to state 2, proceed to state 3, or shark attack. The code then enters a loop, which introduces another shark attack every 10 iterations. The maximum number of shark attacks is 2. The code chooses one action per loop, and can either go to healing station, station 3, station 2, or station 1. This was written such that the sub visits the healing station if it is not healthy, and prioritizes station 2 to avoid the health constraint. The sub exits the loop when it has reached an acceptance state.

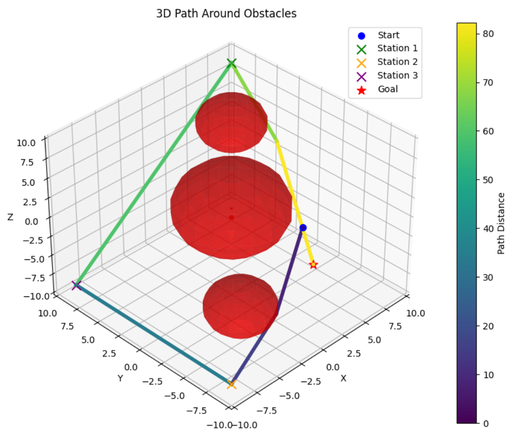
Algorithm 2 LTL Task Plan

```
function PLAN
    $time \leftarrow$ GETCURRENTTIME()
    $firstIndex \leftarrow time \,\%\, nAgents$
    if time % 4 == 0 then
        GETMOTIONPLAN(Current State, Station 1)
    else if time %4 == 1 then
        GETMOTIONPLAN(Current State, Station 2)
    else if time %4 == 2 then
        GETMOTIONPLAN(Current State, Station 3)
    else
        SHARKATTACK()
    end if
    while !$AcceptanceState$ do
        if time % 10 == 1 then
            SHARKATTACK()
        end if
        $automatonState \leftarrow$ GETSTATEFROMAUTOMATA($state$)
        if !$Healthy$ then
            GETMOTIONPLAN(Current State, Health Station)
            continue
        end if
        if !$s_1$ && !$s_2$ && !$s_3$ then
            $randomStation \leftarrow$ PICKRANDOM($s1, s2, s3$)
            GETMOTIONPLAN(Current State, $randomStation$)
            continue
        end if
        if !$s_3$ || !$s_2$ || !$s_1$ then
            $station \leftarrow$ PICKSTATION($s1, s2, s3$)
            GETMOTIONPLAN(Current State, $station$)
            continue
        end if
    end while
    GETMOTIONPLAN(Current State, FinalDestination)
    return Path
end function
```

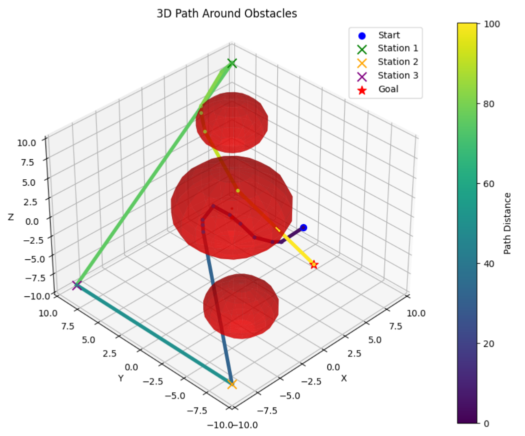
4 Results

4.1 Consecutive task planning

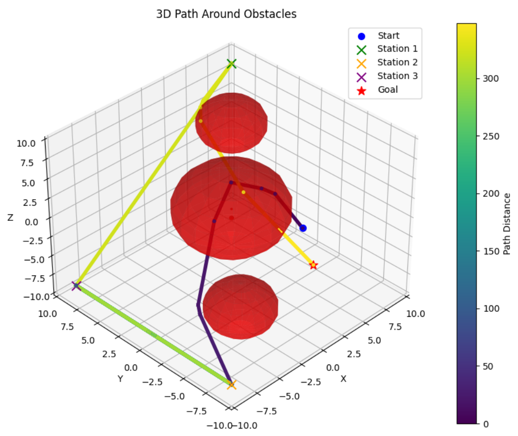
The consecutive task planner was effectively able to accomplish its goals. The motion planning layer consistently produced high-quality paths through the environment, and the task planner effectively ensured that all tasks were visited and completed. The planner generated valid plans in 100% of trials, and ran to completion in under a second. Time was not chosen as a metric because both planners ran in such similar times. Some example paths are shown below. Note that in figure 2d, we see a path length of over 2000, indicating a very high number of failed task attempts.



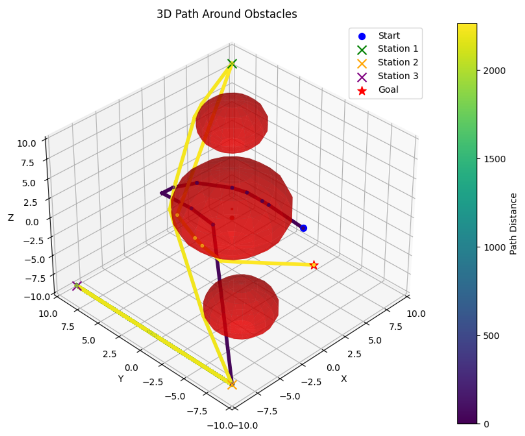
(a) Very high path quality



(b) Good path quality



(c) Small number of task re-trys



(d) Very large number of re-trys

Figure 2: Sample paths for consecutive planner

Average path length was also computed over 1000 iterations. The average length of path produced by this planner was 138.3, with median 94.54, and standard deviation of 306.09. This number included some large outliers, with values going as high as 2000. These values are results of poor randomization, and would not be feasible with the real system. In order to produce a more realistic data set, values outside of 1.5x the Inter Quartile Range (IQR) were excluded. With the cleaned dataset, a mean of 96.06, median of 93.99, and standard deviation of 8.63 were found.

Q_1**:** 25th percentile of the data.
Q_3**:** 75th percentile of the data.
IQR**:** $Q_3 - Q_1$.
x**:** Original data set.
x'**:** Data set with outliers excluded.

$$x'_i = \{x_i \mid Q_1 - 1.5 \times \text{IQR} \leq x_i \leq Q_3 + 1.5 \times \text{IQR}\}$$

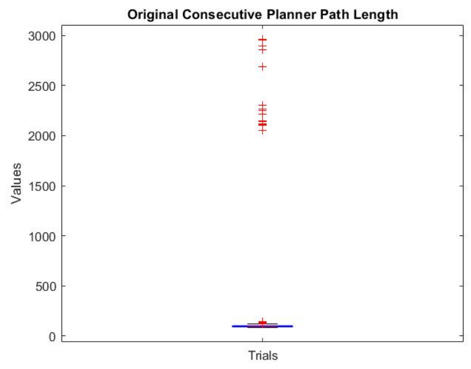


Figure 3: Original Consecutive Planner Data

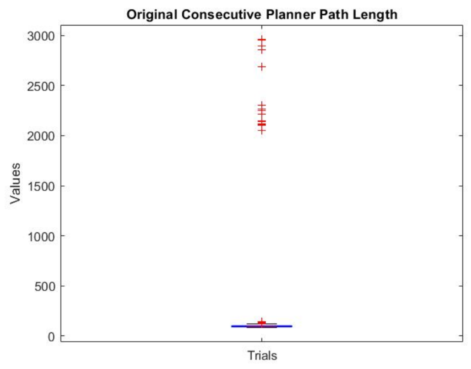
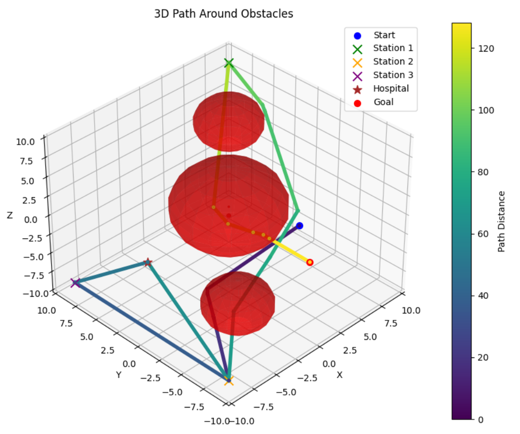


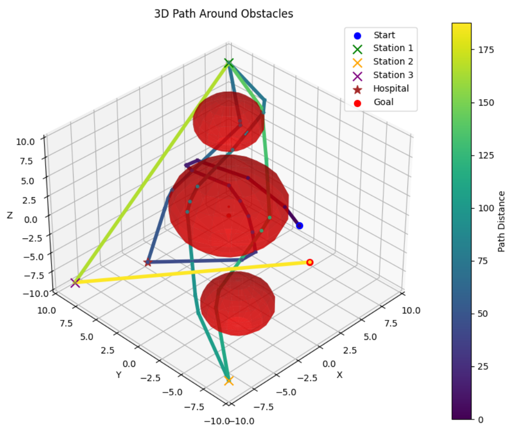
Figure 4: Filtered Consecutive Planner Data (Outliers Removed)

4.2 LTL Planning

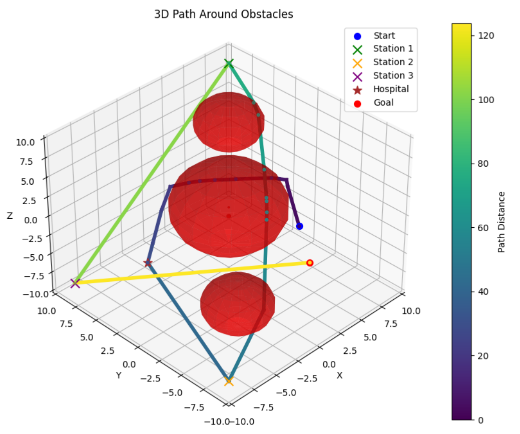
The LTL planner was also able to accomplish its goals. The paths between waypoints are similarly high-quality, and the robot visited the hospital whenever needed. It also revisited tasks as needed, ensuring that all tasks were completed before terminating at the goal. This planner also generated complete paths in 100% of trials, and ran to completion in under a second. Several different trials are illustrated below. Note that the hospital was not always visited (figure 5d), and that the planner was able to choose different stations to visit first.



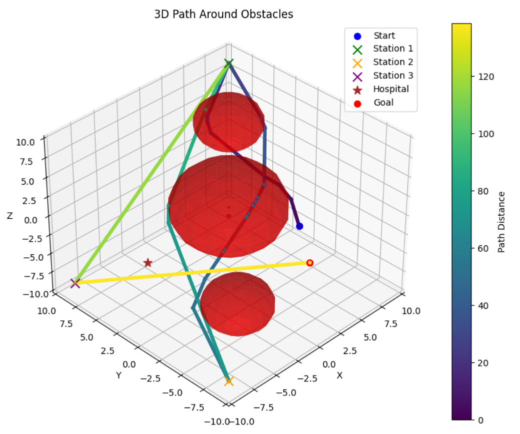
(a) Visit station 2 first and hospital



(b) Exceptionally long path



(c) Visit hospital first



(d) Visit station 1 first, and no hospital

Figure 5: Sample paths for LTL planner

Average path length was also of interest. Running the simulation 1000 times yielded an average path length of 111.64, median of 114.99, and a standard deviation of 22.87. This data set actually had no outliers, so there was no cleaning required.

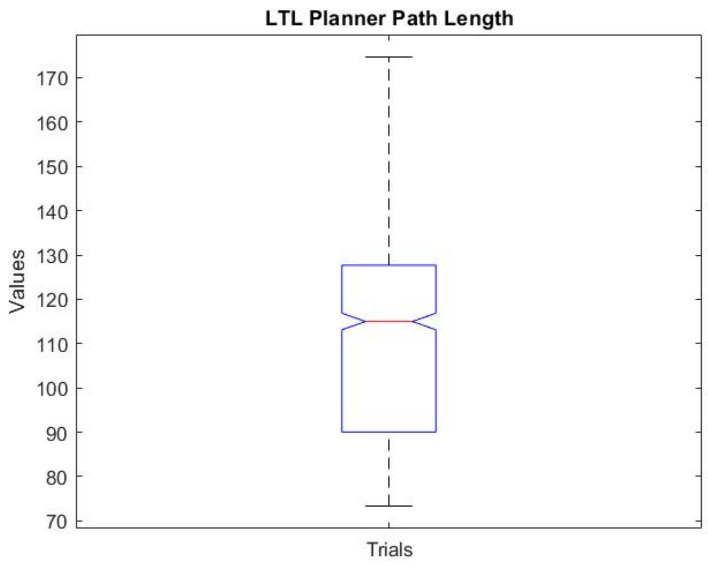


Figure 6: Original LTL Planner Data

5 Discussion

Both proposed planners were able to produce valid plans for 100% of cases, and both were able to run quite quickly. Both planners were able to visit all tasks, and revisit tasks as needed. Both planners provided substantially different task plans, indicting that they were reacting to simulated environment conditions. The motion planning algorithm had no issue finding paths between points, which was expected for a sparsely populated environment.

While both planners accomplish the same thing, they are not comparable because the LTL planner may visit a fourth station, the hospital. The implementation of the two planners also makes them incomparable because the consecutive planner selects the nearest incomplete node to travel to, while the LTL planner has a hard-coded series of logic that enforces a strict order of task completion. This means that comparing path length between planners would be comparing apples to oranges. However, it does make sense that adding a new possible destination would increase the average path length, which we see in these data sets.

The consecutive planner had so many large outliers in its results because the of the dependency on the std::now() function in C++ and the modulo operator. While effort was made to vary to moduli, there were still start times that resulted in tasks not getting marked as complete for many cycles. This is seen in box plot of figure 3, as well as the path generated in figure 2d. In a real application, the implementation of the lower level tasks would have much higher probability of task completion, and would not be dependent on the start time at all. While a time or distance constraint was not enforced in this project, this is another real world constraint that the robot would have to abide by. This would require implementing more sophisticated logic into the planner.

Another interesting result is the standard deviation of the LTL planner. Both the cleaned and original data had standard deviations in the 20s, which is likely a result of the shark attacks. One shark attack added a substantial distance to the path, and shark attack frequency was somewhat randomly distributed between 0, 1 and 2 attacks. The average direct distance between stations and the hospital was 15.8. RRT does not plan the direct path, rather it must go around the obstacles in the middle of the workspace. Unfortunately data was not collected on this, but an additional path length on the order of 20 is a reasonable expectation, and the data supports this. This is likely the cause of the high standard deviation.

One of the most exciting outcomes from this project is the ability of the code to be quickly changed to enforce different automata. Currently, the user must update the initial and acceptance states, as well as the GetNextState() function, which is responsible for following the automata. Stations can be easily added as well. In future work, this code could be modified to take in an automata as a text string, and follow it without the user having to do any additional work. This would streamline the development process, and make this code much more usable in a wider variety of settings.

6 Conclusion

Motion planning is an incredibly useful field, but without task planning the applications to mobile robotics are somewhat limited. Applying task planning layered on top of motion planning expands the capability of robots to perform a series of tasks on their own. Completing a sequence of tasks in a row is fairly simple, but elegantly handling conditions requires another level of complexity. This project successfully implemented both, providing a solid start for making Colorado RoboSub's submarine fully autonomous.

Acknowledgments

Thank you to Professor Morteza Lahijanian for a semester of wonderful lectures and help with problems, and to Yusif Razzaq for helping me to debug and implement homeworks.

Appendix 1.

Source code can be found at this public repository. The README file contains instructions on installation and use. src/planConsecutive.cpp contains the source file for the consecutive planner, and src/planLTL.cpp contains the source file for the LTL planner.

`https://github.com/xavier2933/AMP-final.git`

Bibliography

Alexandre Duret-Lutz, Etienne Renault, Maximilien Colange, Florian Renkin, Alexandre Gbaguidi Aisse, Philipp Schlehuber-Caissier, Thomas Medioni, Antoine Martin, Jérôme Dubois, Clément Gillard, and Henrich Lauko. From Spot 2.0 to Spot 2.10: What's new? In *Proceedings of the 34th International Conference on Computer Aided Verification (CAV'22)*, volume 13372 of *Lecture Notes in Computer Science*, pages 174–187. Springer, August 2022. doi: 10.1007/978-3-031-13188-2_9.