Final Project - Lab Report
AI and Robotics (236609)

Granov, Yotam
yotam.g@campus.technion.ac.il
Mechanical Engineering & Physics
Technion - Israel Institute of Technology

Goldstein, Sharon
sharongold@campus.technion.ac.il
Computer Science & Physics
Technion - Israel Institute of Technology

March 29, 2023

1 Introduction

In this assignment, we were tasked with developing a simple task and motion planning (TAMP) scheme for planning the activities of a TurtleBot3 robot. The robot is located within a closed and fully-mapped room containing 3 workstations, each of which allows for certain actions to be conducted if the robot is in some appropriate configuration. We are provided with an unordered set of tasks that the robot must complete, and each task provides some reward. Our goal is to create plans for the robot (using a decoupled TAMP scheme) to accumulate the highest amount of reward given a certain amount of run time. We also seek to minimize the cost for such plans, which is determined according to the time it takes to execute the plan as well as a cost-map that determines the cost of being in any given position inside the room.

Task and motion planning is a crucial aspect of the field of cognitive robotics, which determines how intelligent robots can execute complex tasks by taking into account both the symbolic (discrete) hierarchies of the required tasks as well as the geometric (continuous) constraints of their environment. TAMP is applicable in almost every application of robotics to our world, from household helping robots that need to plan dish-washing activities to Mars rovers that need to plan their scientific missions efficiently in a fully autonomous manner.

2 Background

We implemented a simple decoupled TAMP scheme for our robot to plan its activities, and thus the theoretical background for our implementation lays mainly in classical planning (for the tasks) and path planning (for the motion). In actuality, though, we chose not to formulate full classical solvers for the sake of choosing task order. Instead, we choose the next task to tackle one at a time (i.e. we don't plan for the entire mission) based on which available task provides the highest reward / lowest cost, and the cost is determined using a standard motion planner (from the ROS `move_base` package). Due to the fact that our decoupled TAMP scheme was so simplistic, we did not need to consult any literature throughout development.

3 Setup

In our setup, we have a simulated Turtlebot3 robot with a LiDAR sensor (for localization and navigation) as well as a sensor for providing odometry (also used for navigation). The map of the robot's environment is provided ahead of time, so the robot does not need to conduct SLAM (simultaneous localization and mapping) but rather just localization to the given map. It conducts the localization using a probabilistic method called the Adaptive Monte-Carlo Localizer (ACML), based on its LiDAR scans - this is implemented for us in the `turtlebot3_navigation` package.

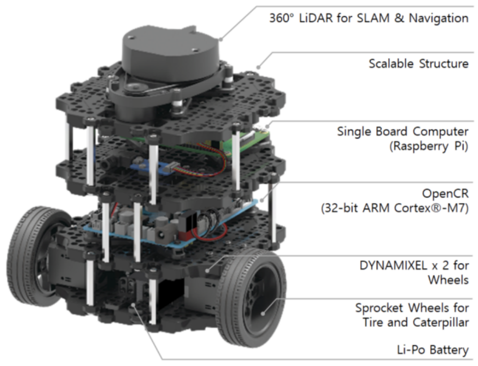
360° LiDAR for SLAM & Navigation
Scalable Structure
Single Board Computer (Raspberry Pi)
OpenCR (32-bit ARM Cortex®-M7)
DYNAMIXEL x 2 for Wheels
Sprocket Wheels for Tire and Caterpillar
Li-Po Battery

Figure 1: The Turtlebot3 robot simulated in this assignment, and its various components

In the environment, we are given a set of 3 workstations (represented as squares) with known locations, as well as the "affordance region" near the workstations which correspond to configurations of the Turtlebot that permit certain actions to be executed. Using the `move_base` navigation package (which provides an action client in ROS), we can use the robot's current location (determined by AMCL) and determine a goal location (using our planning scheme) which are passed to the `move_base` action client in order to receive a potential path between those points. We use this path to determine the cost of conducting a task, and we can execute it if we determine that this path is desirable for our task.

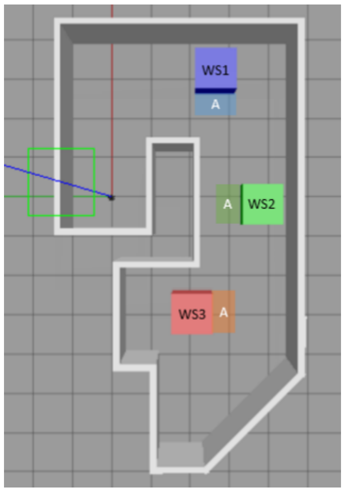

Figure 2: The robot's environment, containing 3 workstations and their affordance regions

3.1 Our Approach

In our approach to this problem, we chose to create a solver which returns the best next task to conduct (in terms of reward minus cost) that can be completed within the remaining time. Each time a task is completed (whether it succeeds or fails) the next task is calculated, and so on until time runs out. The steps for our approach are as follows:

1. Create lookup table that contains all valid activity sequences for all tasks, and for each one records the total cost, reward, and time for that specific sequence. This does not reflect the current location of the robot, only the locations of the workspaces.

2. Update the cost and time required for each activity sequence in the lookup table given the current position of the robot.

3. Choose the next task to attempt based on the activity sequence in the lookup table with the largest $R - C$ value.

4. Execute said task, and check if the task succeeded (i.e. if the accumulated reward has increased). If yes, then remove that task from the lookup table.

5. Repeat steps 2 to 4 until time runs out or all tasks are successfully completed.

The nature of our approach limits the effects of error propagation due to deviations in the actual costs of paths (since we re-plan after each activity sequence is completed), and requires less computation up front. We considered an alternative scheme where we would create a tree containing all possible sequences of tasks (activity sequences) and traverse the tree to find the best sequence of tasks (i.e highest total $R - C$ value) within a given time limit, however we leave this scheme as potential future work given that our method seems to work sufficiently well (and more dynamically responsive to deviations in path cost) anyways.

4 Experiments

To formally test our implementation on the Turtlebot3 ROS simulation, we ran 25 trials where the affordance regions of the workstations were randomly determined each trial. In this set of 25 trials, we conducted 5 subsets of 5 trials each that corresponded to different time limits imposed at execution: 100[sec], 200[sec], 300[sec], 400[sec], and 500[sec]. The available tasks (and their rewards) remained the same across all trials, and are described in the table below.

Task	Activities	Reward
0	ACT1 $\rightarrow$ ACT2	20
1	ACT1 $\rightarrow$ ACT4	10
2	ACT1 $\rightarrow$ ACT3 $\rightarrow$ ACT2	50
3	ACT2 $\rightarrow$ ACT3	30

5 Results

In the following table, we can see the averaged results across 5 trials for each time limit value (for a total of 25 trials).

Time Limit [sec]	Avg. Reward Received	Avg. Total Time [sec]	Avg. Reward / Second
500	110	319.966	0.347
400	110	296.756	0.373
300	100	261.927	0.392
200	68	176.220	0.385
100	34	80.625	0.419

Figure 3: Table containing the results from 25 trials

The data in this table is shown graphically in the following plots.

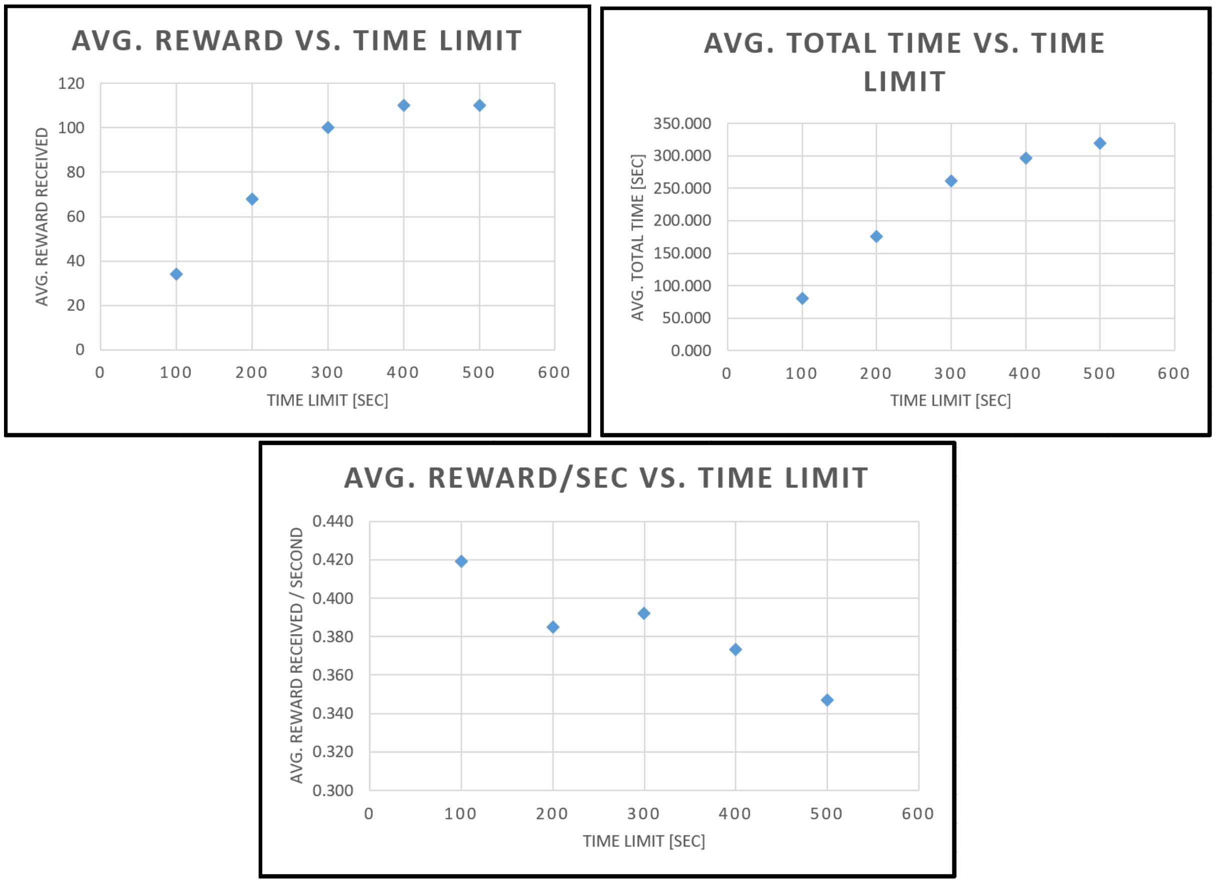

Figure 4: Plots showing the results from 25 trials

As expected, the average amount of reward received will decrease as the time limit decreases. Interestingly, though, the average amount of reward received per second is higher for lower time limits. Above time limits of 300[sec], the robot almost always obtains the maximum possible reward for this setup (in this case, 110). In the following images, we can see the robot attempting to execute Tasks 0 through 3, where the yellow arrows show the order of positions that the robot needs to move between (for Task 0, the robot starts at an initial position of $[0, 0]$ in this image).

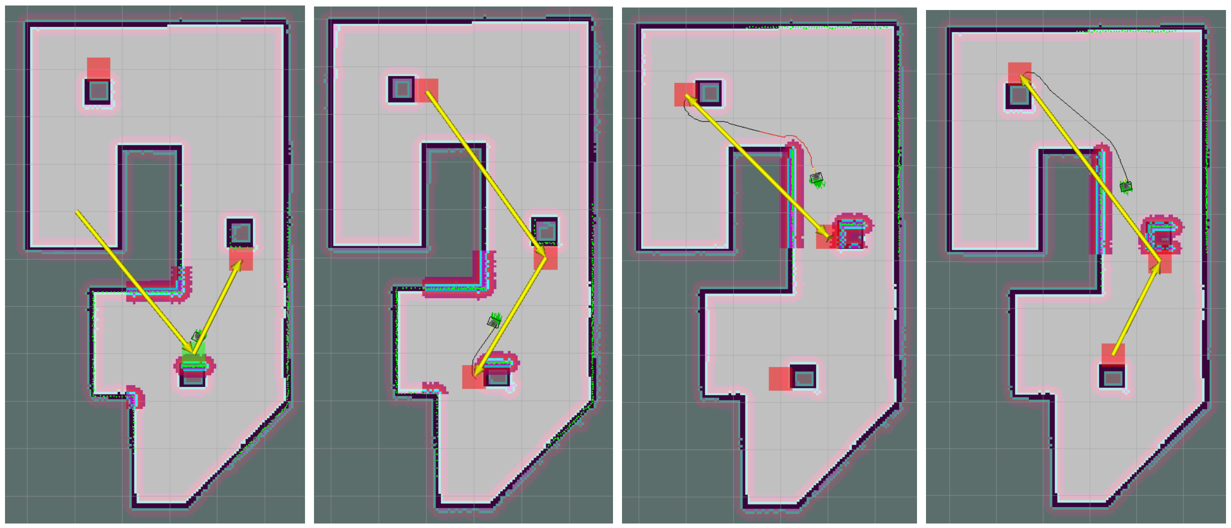

Figure 5: The robot attempting Tasks 0 through 3 (from left to right)

6 Discussion

The results of our experiments appeared promising, in that for any given time limit (of at least 100 seconds) the robot was able to execute a plan that returned a reward of at least 30 and at most the maximum (110). In the current setup, we see that the main improvement we could make would be to plan ahead for the entire time threshold, and just picking the next best task to attempt at a given time. As mentioned before, we could do so using a tree-type structure containing all possible permutations of valid task sequences, however we would need to be careful to prevent too much error propagation through the tree due to estimations of the costs of each activity sequence. Beyond this setup, it might also be interesting to investigate other variants of this problem such as:

- Multi-Agent TAMP: Multiple robots working together (and concurrently) to complete the set of tasks in the same environment
- Dynamic Workstations and/or Obstacles: If there were to be moving workstations or obstacles in the robot's environment, we would need to implement a more effective planning scheme to account for the fact that our initial lookup table is no longer valid for the entire duration of the execution
- Probabilistic Actions: If we would be interested to apply probabilistic planning and reinforcement learning to this problem, we might consider making the actions probabilistic so that executing a given action does not guarantee that it will actually succeed