AIMS Mobile Robotics 2020 - Planning Exercises

1 Overview

In this exercise you will implement the key functions used in Monte-Carlo Tree Search, then, you will complete the implementation of the search and rescue SSP that will be used in the final exercise.

2 The Code

The `aims_planning` folder contains the code for this exercise, including the following files:

- `src/build_search_and_rescue_ssp.py`: A collection of functions to build the search and rescue `SSP` that will be used in the final task.
- `src/condition.py`: Contains multiple `Condition` class definitions, used to build SSPs and representing boolean functions over `StateFactor` objects.
- `src/cost.py`: Defines the `StateActionCost` class, representing a cost function for an SSP, specified using `Condition` objects and action names.
- `src/mcts.py`: Defines the `mcts` function, implementing *monte-carlo tree search*. Both a selection and rollout policy are to be implemented here.
- `src/mcts_chance_node.py`: Code defining the `MCTSChanceNode` class for chance nodes used in `mcts`.
- `src/mcts_decision_node.py`: Code defining the `MCTSDecisionNode` class for decision nodes used in `mcts`.
- `src/solve_search_and_rescue_ssp.py`: Code that runs `mcts` on the search and rescue `SSP`. This can be used to confirm that your `mcts` function is running as you expect it to.
- `src/ssp.py`: Provides an interface for SSPs.
- `src/ssp_impl.py`: Backend implementation of `SSP` class, you shouldn't need to look at this.
- `src/state.py`: Gives the `State` class, a collection of `StateFactor` objects and their corresponding values.

- `src/state_factor.py`: Defines the `StateFactor` class, which can be given a name and specifies a set of allowed values.
- `str/transition.py`: Defines the `ProbTransition` class that is used to represent transitions in SSPs.

We will give a small example of building an SSP using some of these classes later.

3 MCTS Task

Your first task is to complete the implementation of *monte-carlo tree search*. You should first familiarise yourself with the code in the `mcts_chance_node.py`, `mcts_decision_node.py` and `mcts.py` files first. The functions that you need to complete are as follows:

- `src/mcts.py`:
  - `random_rollout_policy:` A policy to be followed during the rollouts of your MCTS.
  - `ucb_selection_policy:` A policy to follow during the selection phase of MCTS, which should select actions using the UCB algorithm.
- `src/mcts_chance_node.py`:
  - `backup:` Implement the '`backup`' function for the chance node. This ammounts to computing a sample average.
- `src/mcts_decision_node.py`:
  - `backup:` Implement the '`backup`' function for the decision node. This ammounts to computing a sample average.

To test your MCTS solution, you can run the following command:
`python solve_deep_sea_treasure.py`. This will immediately print out a visualization of the *Deep Sea Treasure Environment* (Figure 1). After MCTS is run (it will take a little while), it will display the sample average total cost over all trials and most frequently sampled path. The average total cost for a good implementation of MCTS with random roll-outs should achieve a value of at most -800 (the minimum cost is approximately -1000).

4 Main Task Description

It is now helpful to describe the final task to be completed, as it will be relevant for the rest of this exercise sheet. You have been tasked with producing a control policy for a robot to be used in search and rescue tasks.

You may assume the following properties for the specific scenario in which your robot will be deployed. You may assume that you are given a topological map of the one floor building that you need to search. Rubble may appear in any one of the four doorways

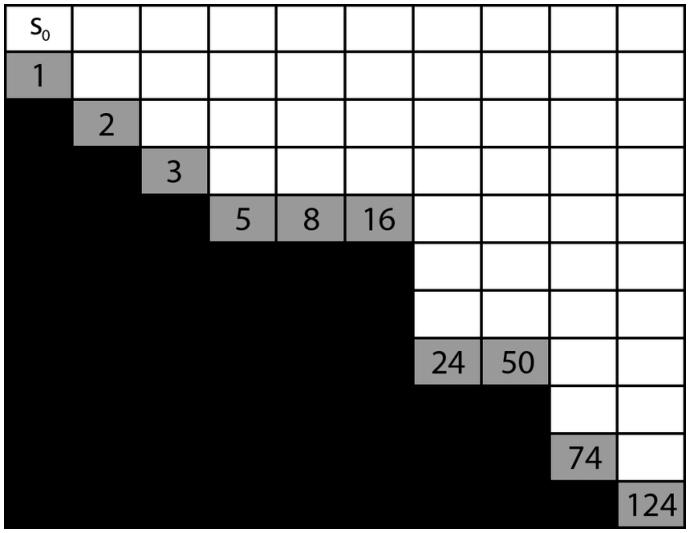

Figure 1: An example of a *Deep Sea Treasure* environment. It is a grid world with a submarine starting in the top left corner. The submarine may move in any direction, but may get caught by a current with small probability. Each timestep has a cost of 1, and when the submarine reaches a treasure it is given a reward (i.e. a negative cost) equal to one of the numbers shown. In our model, the treasure rewards vary from 1 to 1000. The submarine can only carry one piece of treasure so the trial ends once any piece of treasure is reached. Finally, the time horizon is set to 1000 time steps.

which will block the path of your robot. Rubble is approximately classified as small and large, and your robot is capable of clearing these rubble piles which cost 10seconds and 45seconds seconds to clear respectively. You are told in advance how many people may be in the office, and in our scenario that number is two.

Your goal is to identify where the two survivors are in the building and return to the starting location in the minimum time.

5 Modelling Task

In the file `src/build_search_and_rescue_ssp.py` the majority of an SSP for the problem described in Section 4 is provided. You need to complete the implementation by completing the `make_check_for_person_transitions` function. This will

You may need to spend a bit of time familiarising yourself with the other code in `src/build_search_and_rescue_ssp.py`. In particular the `make_search_and_rescue_state_factors` function will be useful as it defines the state factors for the SSP.

After completing this task you should be able to run the command `python solve_search_and_rescue_ssp.py` and after a little bit of time see the average cost in over all trials is below 5000. (If the value is below 5000 then it mans that MCTS has found at least one successful path where it will find both humans).

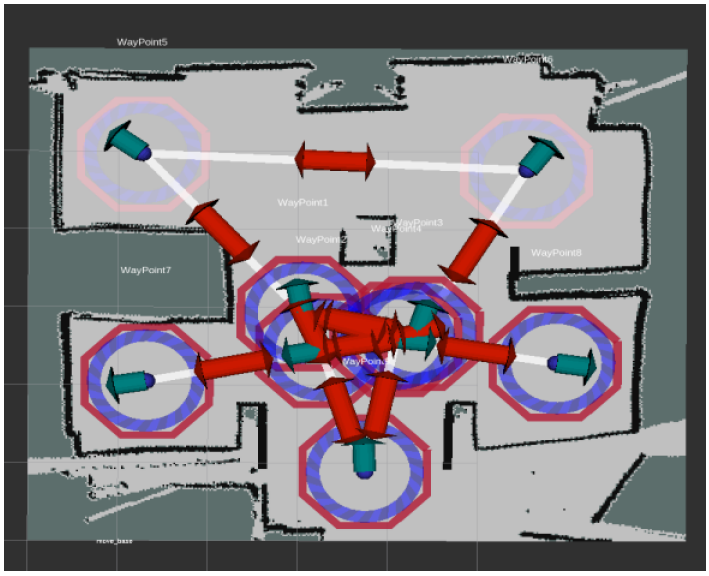

Figure 2: The topological map of the office building. Here red bidirectional arrows indicate edges in the map, and the blue arrows inside each of the circles are the poses associated with "being at" each of the nodes.

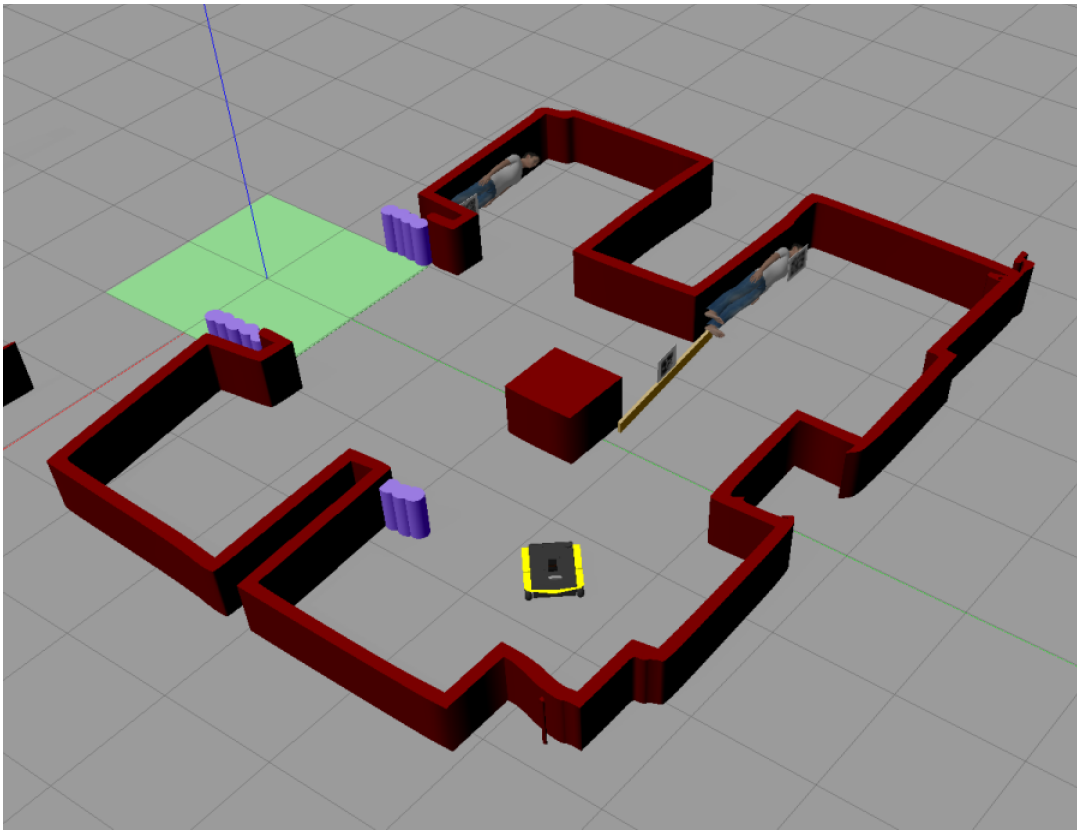

Figure 3: The simulator that your will use to test your overall solution.

6 Improving Your Solution

There are a number of ways that your MCTS implementation can be improved. It is recommended that you at do the first bullet and the second is optional.

- The performance of MCTS depends largely on the quality of the rollout policy used. You can change the policy used for rollouts in MCTS. *Hint: it is useful if your policy is able to find the two humans more frequently than the random baseline policy.*
- The structure that the MCTS code follows is very similar to a more general algorithm *Trial-Based Heuristic Tree Search* (THTS) [1]. You may want to try some of the variants of THTS for the final task.

References

[1] T. Keller and M. Helmert, "Trial-Based Heuristic Tree Search for Finite Horizon MDPs.," in *ICAPS*, 2013.