Discrete Planners

Manas Gupta

March 28, 2019

1 Introduction

In this project we have extended a library of a discrete motion planner to include a random and an optimal planner for a holonomic robot. We have been given an environment consisting of static obstacles in the form of an 2 dimensional occupancy grid. We have implemented both the planners with a common planning interface which executes a planning method which has the following input:

- *world_state* is a 2D-grid representation of the environment where the value 0 indicates a navigable space and the value 1 indicates an occupied/obstacle space.
- *robot_pose* is a tuple of two indices (x, y) which represent the current pose of the robot in *world_state*.
- *goal_pose* is a tuple of two indices (x, y) which represent the goal in world_state coordinate system.

and returns a path (policy/trajectory) which consists of sequence of waypoints.

- path is a list of tuple (x, y) representing a path from the robot_pose to the *goal_pose* in *world_state* or None if no path has been found.

2 Planners

In this section we are going to discuss planners and how we implemented them.

2.1 Random Planner

A random planner as its name suggests is a non-deterministic planner which induces stochastic behaviour in the planner. Since our agent is constrained to take actions which are orthogonal, we have implemented a Markov chain random walk. In this our agent takes action randomly in any of the four directions and checks the validity of the state. State validity is comprises of checking for the collision with the obstacle as well as checking the state if it has already been visited. If the state has not been visited and is in obstacle free region, agent takes the action and the state is updated.

The planner terminates if either the robot reaches goal or it take actions more than the maximum permitted actions. For the former case it returns the path with the sequence of waypoints and in latter case it returns a failure since feasible path is not found.

Due to non-deterministic in nature often times robot gets stuck in a state from where all the future states are either already visited or in obstacle region. In this scenario we do consider as a special conditions and allows the agent to go to the states which are already visited so that it can come out of the locally stuck state.

Since the planner has low memory we delete the states which were visited before $sqrt(max\_steps\_required)$. This also sometimes helps preventing our agent from getting stuck locally.

2.2 Optimal Planner

An optimal planner always gives the optimal solution in terms of low cost if a solution exist. We have implemented an A* algorithm to compute an optimal policy if exists. For the heuristics, we have used Manhattan distance to compute h value since the permitted actions that our agent can take are orthogonal. A* algorithm is a standard algorithm and can be referred from this paper by Steven Lavalle in depth knowledge- http://msl.cs.illinois.edu/~lavalle/papers/YerLav11.pdf.

3 Implementaion

We have implemented a common interface i.e. a class *Planner* has the following methods:

- `move_up(pose)`: This method is used to find out the state if the agent takes an action to move to a state that is just above the current state.
- `move_down(pose)`: This method is used to find out the state if the agent takes an action to move to a state that is just below the current state.
- `move_left(pose)`: This method is used to find out the state if the agent takes an action to move to a state that is to the left of the current state.
- `move_right(pose)`: This method is used to find out the state if the agent takes an action to move to a state that is to the right of the current state.
- `random_state(world_state, pose)`: This method computes all the possible children (here 4) of a current state and returns a child state at random. We have a functionality to check if the agent is stuck locally (i.e. if all the children are already visited or in obstacle region or out of map boundary), if this is the case we update the flag `no_way_out` to true.
- `collision_check(world_state, pose)`: This method is used to check if the current state is in the obstacle region or outside the map boundary and returns true if any of the condition satisfies.
- `check_visited(visited, pose)`: This method helps to check if the current state has already been visited. It also adds the functionality that if flag `no_way_out` is true i.e. agent is stuck locally it will allow the agent to visit the already visited states again to prevent it from getting stuck permanently.
- `goal_checking(goal_pose, pose)`: This method helps to check if the current state is the goal state.
- `h_value(goal_pose, pose)`: A method to compute h value for A* algorithm using Manhattan distance.
- `trace_path(visited,robot_pose,goal_pose)`: For A* algorithm to compute the set of way points for a policy, we need to backtrack from goal state to initial state and this method facilitates this functionality. Backtracking is done by tracing the parent of the respective nodes starting with the goal state.
- `search_random`: An implementation of the randomized planning algorithm as described in sec.2.1. It returns a feasible path if goal is reached in less than maximum steps required else it terminates and returns a failure as no path was found.
- `search_optimal`: An A* implementation for the optimal policy generation.

4 Performance

Performance of the algorithm is measured in terms of its ability to find the solution of the problem, if exists, in limited amount of time as well as the number of actions required to reach the goal state. An optimal algorithm is expected to return a policy, if exists, with minimum number of actions to be taken and in shortest period of time.

4.1 Random Planner

Being non-deterministic in nature this planner never assures that it will converge to a feasible policy, even if it exists. For our implementation the planners performance depends on the maximum steps required. If this parameter is small then often planner returns a no solution (failure) because it requires more iterations to sample the states to reach goal, as this parameter increases, the probability to find the feasible solution also increases and the planner returns a feasible solution. Though the planner returns a feasible solution, there is no guaranty that the policy is the shortest one, as well as the time taken to compute the policy also increases significantly. The exploration is also significant and requries more memory to store the visited states as compared to optimal planner

This kind of planner is best suited for exploration purposes when the environment and map is not known *a priori*, and with the help of Markov random walk the environment can be explored and mapped. However, if the map is known in the form of occupancy grid, then there is no point of using such algorithm. We have bounded the planner with max_step_required hence **time complexity O(max_step_number)**

4.2 Optimal Planner

Optimal planner guaranties to converge to an optimal policy i.e low cost if there exists one. Its running time is upper bounded by number of vertices in the graph hence **time complexity O(Vertices)**. Since A* is admissible it is well suited for the environments where all the obstacle positions are known. The heuristic helps to explore fewer state and drive the search towards the goal position and hence shortest policy is computed with very less exploration as compared to random planners.

Over all Random planners are well suited for the exploration planning when the environment is not known where as optimal planners are well suited for known environment.

5 Testing

Test cases are written for all the possible scenarios and tested to run successfully signifying correctness of our implementation.