Division of Engineering Science
UNIVERSITY OF TORONTO

ROB311: Artificial Intelligence

Project #3: Motion Planning and Adversarial Games

Winter 2023

Overview

In this project, you will expand your understanding of two important lecture topics: decision tree learning and the design of game-playing agents. The goals are to:

- build a randomized motion planning solver for a Dubins-type vehicle; and
- build a simple game-playing agent (to be put to the test against your classmates' agents!)

The project has two parts, worth a total of **45 points**. All submissions will be via Autolab; you may submit as many times as you wish until the deadline. To complete the project, you will need to review some material that goes beyond that discussed in the lectures—more details are provided below. The due date for project submission is **Monday, April 03, 2023, by 23:59 EDT**.

As in Project #2, each part already has some basic code in place for you to start with.

Part 1: Motion Planning for a Dubins Vehicle

The rapidly-exploring random trees (RRT) algorithms is a widely-used algorithm for sample-based robot path planning, in part because it is able to take into account *kinodynamic constraints*. These are constraints on velocity and acceleration, for example. RRTs are also able to handle *nonholonomic constrains*, that is, constraints that involve the possible paths taken (parallel parking is such an example: a car cannot move sideways instantaneously). The algorithm has been discussed in lecture and a demonstration can be seen in this video.

A popular class of kinodynamic paths is known as Dubins paths. A Dubins path is the shortest curve that connects two points in the two-dimensional Euclidean plane with a constraint on the curvature of the path, and with prescribed initial and final tangents (vectors) to the path. An assumption is made that the vehicle on the path can only travel forward. This is a very useful model for many robotics applications. You can read more about Dubins path here. An example of such a path is shown in Figure 1. Your task is to:

1. Implement an RRT-based planner for a Dubins-type vehicle in a 2D planar world with circular obstacles. The RRT planning function should accept an RRT problem object (of class `RRT_dubins_problem`) which contains: a map (2D world), a list of circular obstacles, a starting pose, a goal pose and plenty of helper functions for Dubins path generation. The function should produce a list of nodes along a valid path starting from the start node and reaching the goal state. Use the function template in the handout code called `rrt_planning.py`. Precise instructions and conditions have been provided in the comments of this file. The `RRT_dubins_problem` class and a simple test have been implemented in the helper file `rrt_dubins_problem.py`. Helpful Dubins-type path generation functions have been provided in `dubins_path_planning.py`. Keep in mind that there are time limits on the tests, which implies it is essential to reach the goal pose quickly. **HINT:** A purely random exploration strategy is not fast enough in finding the goal. If you already know the goal pose, a "smarter" random exploration strategy can be

employed for a faster solution.

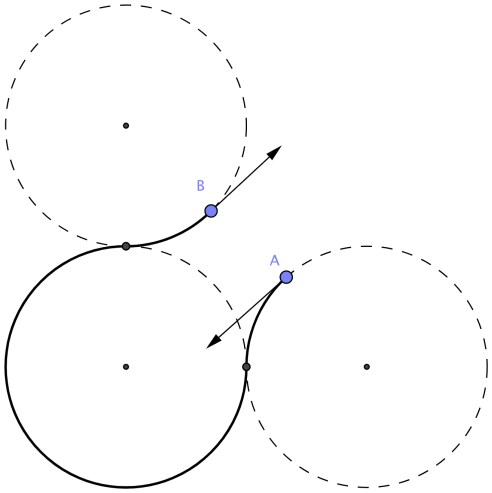

Figure 1: Example Dubins LRL path.

Note that a major chuck of the problem setup and function implementation has already been provided; for example `rrt_dubins_problem.py` already contains a propagation and a collision checking function, but you are encouraged to read these implementations to understand the big picture for a successful implementation. The provided code is well documented for your clarification and understanding.

You will submit and check your implementations of the planning function in `rrt_planning.py` file through Autolab.

Part 2: Adversarial Games

One of the fist applications of AI was to games, and many well known games have now been solved (e.g., you can read the fascinating story about solving checkers here). That is, the optimal set of moves given any starting state is known. However, developing game-playing agents can still be challenging! In this open-ended portion of the project, you will write an agent to play a modified version of rock-paper-scissors with a game (payoff) matrix of the form:

$$\mathbf{M} = \begin{bmatrix} 0 & -a & b \\ a & 0 & -c \\ -b & c & 0 \end{bmatrix}, \tag{1}$$

where a, b, and c are all positive (you will not be given their values). Your task is to:

1. Write a game-playing agent that attempts to win as many games as possible. The class `StudentAgent` in `iterated_single_move_games.py` has three methods that you must implement. As usual, you may only use the import statements present in the file.

Your agent will be pitted against other agents: for each given opponent, your `StudentAgent` class will repeat 1000 rounds against that opponent. See the `play_game` function in `iterated_single_move_games.py` for details. The other agents you are up against are:

- a dumb agent that always chooses the first move (see `FirstMovePlayer`),

- a copycat agent that always copies its opponent's last move (see `CopycatPlayer`),
- an agent that randomly chooses one of the three options with equal probability (see `UniformPlayer`),
- a 'goldfish' agent with very short memory (source code is not available to you),
- an agent that uses a mixed Nash equilibrium strategy (source code is not available to you),
- and an agent that follows a random Markov process that depends on the last round (source code is not available to you).

You do not have to beat every agent head-to-head, but you must win the lion's share of the points (see the Grading section below). The exact strategy that your agent employs is a design decision—however, you must briefly document the technique(s) you used in your code. As the final part of the project, we will hold a round-robin tournament. A portion of your grade on this part of the project will depend on your algorithm's performance in the tournament. Grading details are provided in the section below. Please read all the comments in the starter code for implementation details. You will submit your implementation of `StudentAgent` in `iterated_single_move_games.py` through Autolab.

Grading

We would like to reiterate that **only** functions and methods implemented for the Autolab's tester will affect your marks. As usual, submit your code to Autolab as follows:

```
tar cvf handin.tar *.py
```

Points for each portion of the project will be assigned as follows:

- Motion Planning for a Dubins Vehicle – **20 points** (2 tests with 15 and 5 points respectively.)

  Each test will give you an obstacle map, a start pose and a goal pose. Your function must return a kinematically feasible set of nodes from the start pose that reach the final goal pose with no collisions. The tests will check whether nodes on your paths are correct (we fix the seed for the pseudo-random number generator to ensure results will be the same between runs).

  Time limit: 20 seconds for Test 1, 40 seconds for Test 2.

- Adversarial Games – **25 points** (5 points for tests; 5 points for algorithm description; 15 points for tournament performance)

  Tests: The test will pit your agent in a tournament against the simple agents mentioned above. Note that each agent will play 1000 games against each opponent. You must score over 6500 points to get full marks, or over 3250 points to get half marks.

  Note: the tester will time out if the total run time for all 1000 games exceeds 180 seconds.

  Algorithm Description: In addition to your code, *you must also include a longer comment block* (approximately 10-15 lines, 80 characters per line) that describes your approach and/or the algorithm that you implemented. This comment is to be placed in the docstring right below the header of `StudentAgent`.

  Tournament: After the project submission deadline, we will run all agents head-to-head in a round-robin tournament. The class's algorithms will be ranked and split into tertiles. The top scoring tertile gets 15 points, the middle tertile gets 10 points, and the bottom scoring tertile gets 5 points.

Total: **45 points**

Grading criteria include: correctness and succinctness of the implementation of support functions, proper overall program operation, and code commenting. Please note that we will test your code, and *it must run successfully*. Remember to **submit early and often** to ensure your code runs successfully. Code that is not properly commented or that looks like 'spaghetti' may result in an overall deduction of up to 10%.