Division of Engineering Science
UNIVERSITY OF TORONTO

ROB311: Artificial Intelligence

Project #2: Structured Problem Solving and Planning

Winter 2020

Overview

In this project, you will gain further experience with three different topics we have discussed in lecture: formal logic, constraint satisfaction problems, and vehicle motion planning. The goals are to:

- implement an algorithm for logical inference over definite clauses;
- use local search to solve the classic N-queens constraint satisfaction problem (for $N > 100$); and
- build a randomized motion planning solver for a Dubins-type vehicle.

The project has three parts, worth a total of **50 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 **Friday, February 28, 2020, by 11:59 p.m. EST**.

As in Project #1, each part already has a starter script with some basic setup code and a simple problem instances ready to go. Once again, **do not** modify function names, file names, or import statements! Please also remember to test your functions one at a time so that an incomplete function does not cause Autolab to hang and timeout.

Part 1: Inference with Definite Clauses

Propositional logic can be used to answer questions about *entailment*, i.e., whether one fact follows logically from another set of facts. If we restrict ourselves to definite clauses, which are Horn clauses with exactly one positive literal, very efficient inference algorithms exist, such as forward-chaining (see AIMA pg. 258). Your task is to:

1. Implement a simple inference engine to solve problems in propositional logic involving definite clauses only. The engine should accept a knowledge base KB and a query q (which is a propositional symbol) and return *true* if KB entails q (and false otherwise).

You will submit your implementation of `inference_method.py` through Autolab. Please see the docstring of `pl_fc_entails` in `inference_method.py` for a description of the inputs and outputs. The definite clauses will be provided to you in the form of `DefiniteClause` objects from `support.py`.

Part 2: The N-Queens Problem

The N-queens problem is a classic search and constraint satisfaction problem—the goal is to place N queens on an $N \times N$ chessboard such that no queen 'attacks' any other. A queen attacks any piece in the same row or column, or that lies along the same diagonal on the board. An example (partial) partial attempt at a solution

to the 8-queens problem is shown in Figure 1. We will refer to a queen attacking another queen as a 'conflict'.

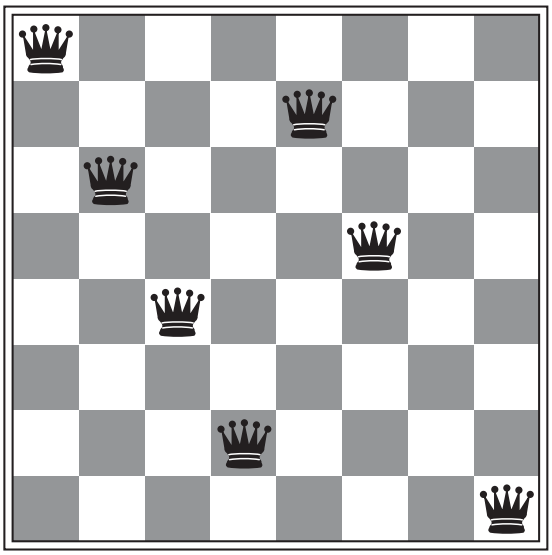

Figure 1: Example partial solution (with a conflict on the main diagonal) to the 8-queens problem (AIMA pg. 72).

Luckily, N-queens is quite easy for local search algorithms to solve, because solutions are distributed fairly densely throughout the state space (see AIMA pg. 221). Your tasks are to:

1. Implement a greedy initialization algorithm for the N-queens problem, that is, an algorithm which starts with one queen on the board (randomly choose the row for the first column) and adds new queens incrementally until all queens have been placed. The greedy strategy should attempt to minimize the number of conflicts for each new queen that is added. Use the function template in the handout code called `initialize_greedy_n_queens.py`. The function should return a $1 \times N$ vector, where the zero-indexed i^{th} entry is the row number of the queen in the i^{th} column. The row numbers should also be in the set $\{0, 1, \ldots, N-1\}$ (i.e., zero-indexed). The docstring of `initialize_greedy_n_queens` provides examples and further details.
2. Build a solver that makes use of the MIN-CONFLICTS heuristic (AIMA pg. 221) to place all queens on an $N \times N$ board without any conflicts. Note that this will be a *randomized* algorithm, and hence different runs may produce different results—in certain instances, you may need to run your function two (or more) times to produce a valid solution. Use the function template in the handout code called `min_conflicts_n_queens.py`, which has extra details in its function's docstring. You will be graded by a procedure that calls your implementation of `initialize_greedy_n_queens` to produce an initialization, then uses your `min_conflicts_n_queens` implementation to search for a conflict-free solution (see the example in the handout code).

As with Project #1, you will submit your implementations of `initialize_greedy_n_queens.py` and `min_conflicts_n_queens.py` through Autolab. For both the initialization and heuristic search functions, be sure to break any ties (in terms of the minimum conflicts heuristic) randomly - this is key for getting the algorithm to perform well.

Part 3: 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 9 and a demonstration can be seen in this video after the 1:00 time-stamp.

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 2. 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 setup that contains: a map (2D size of the world), a list of circular obstacles, a starting pose, a goal and plenty of helper functions for dubins path generation. The function should produce a list of nodes along a valid paths 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 this file. The above mentioned problem-setup as well as a simple test have been implemented in a helper file called `rrt_dubins_problem.py`, while 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 fast.

   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.

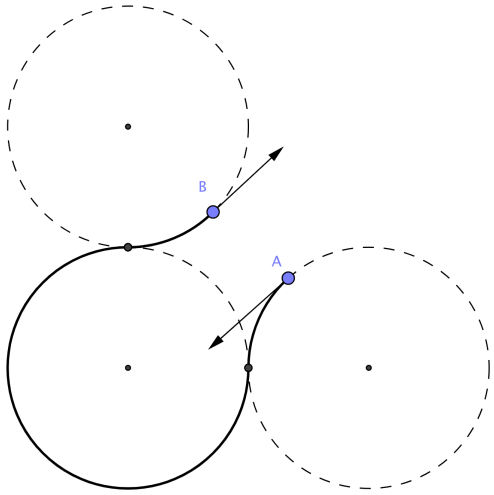

Figure 2: Example Dubins LRL path.

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_dubins_planning.py` file through Autolab.

Grading

We would like to reiterate that **only** functions implemented for the Autolab will affect your marks. There are other questions in the sections above, but only those that ask you to submit a function via Autolab will affect your total. The remainder are useful for your understanding or are there to aid you in creating the functions for Autolab. Points for each portion of the project will be assigned as follows:

- Inference with Definite Clauses – **15 points** (5 tests $\times$ 3 points per test)

  Each test will check whether your function is able to correctly determine if q is inferred by KB.

- N-Queens – **15 points** (3 tests $\times$ 5 points per test)

  Each test will provide a positive integer N and check whether your code outputs an assignment of N queens to the $N \times N$ chessboard such that no queens are attacking one another.

- 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).

Total: **50 points**

To submit your code, put it in a `.tar` file with this command (from a directory containing all the requisite `.py` files:

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

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*. Code that is not properly commented or that looks like 'spaghetti' may result in an overall deduction of up to 10%.