Division of Engineering Science
UNIVERSITY OF TORONTO

ROB311: Artificial Intelligence

Project #2: Structured Problem Solving and Planning

Winter 2023

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
- implement and understand basic decision tree learning, using the greedy information gain and gain ratio splitting criteria described in the lectures

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 **February 26th 2023, 23:59 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'.

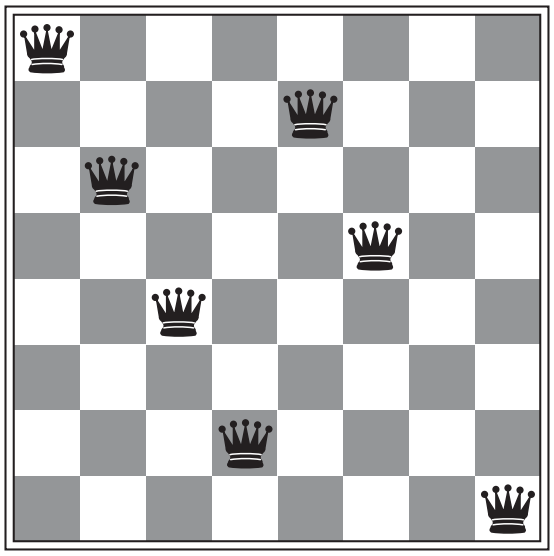

Figure 1: Example partial solution (with a conflict on the main diagonal) to the 8-queens problem (AIMA pg. 72).

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.

HINT: In order to satisfy the time constraints, avoid using loops (especially nested loops) whenever possible. Insetad, make good use of the vectorized NumPy methods in both `initialize_greedy_n_queens.py` and `min_conflicts_n_queens.py`.

Part 3: Decision Trees

Decision trees are useful for a wide range of machine learning tasks, and have the advantage of being readily understandable. We will be concerned with building trees using training data involving discrete-valued attributes. You will implement functions to determine how to split the data (and build the tree) based on the information gain and gain ratio criteria. Your tasks are to:

1. Implement five basic support functions for decision tree learning that compute (in order): discrete entropy, conditional entropy, intrinsic information, information gain, and information gain ratio. Use the function templates `decision_tree.py` in the handout code to get started.
2. Implement a Python version of the Decision Tree Learning algorithm given on pg. 702 of the AIMA text, in the file `decision_tree.py`. Your learning implementation will make use of the functions provided above (and enable splitting using both information gain and gain ratio criteria).

Some utility code has been provided for you in the form of a `TreeNode` class and its methods, and a function to compute the 'plurality value' of a set of examples — do not modify these. Please read all comments in the starter code for implementation details. You will submit your implementations of all 6 template functions in `decision_tree.py` through Autolab.

Submission and Grading

We would like to reiterate that **only** functions implemented for the Autolab's tester 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 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.

  Time limit: 5 seconds per test.

- 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.

  Time limit: 5 seconds for $N \approx 100$, 10 seconds for $N \approx 500$, 20 seconds for $N \approx 1000$.

- Decision Tree Learning – **25 points** (5 tests $\times$ 3 points per test; 2 tests $\times$ 5 points per test)

  The first five tests are of the utility functions for decision tree learning (defined in Part 3); each test is designed to ensure that your support function is operating correctly. The final two tests will involve learning a tree using a hold-out (hidden) dataset.

  Time limit: 5 seconds per test.

  Please note the time limits are are included to catch bugs and infinite loops in submitted algorithm implementations. Correct solutions will run in far less time than the time limits.

Total: **55 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%.