Homework 1

15-381/681: Artificial Intelligence (Fall 2017)

Out: September 21, 2017
Due: October 3, 2017 at 11:59PM

Homework Policies

- Homework is due on Autolab by the posted deadline. Assignments submitted past the deadline will incur the use of late days.
- You have 6 late days, but cannot use more than 2 late days per homework. No credit will be given for homework submitted more than 2 days after the due date. After your 6 late days have been used you will receive 20% off for each additional day late.
- You can discuss the exercises with your classmates, but you should write up your own solutions. If you find a solution in any source other than the material provided on the course website or the textbook, you must mention the source. All homeworks (programming and theoretical) are always submitted individually.
- Strict honor code with severe punishment for violators. CMUs academic integrity policy can be found here. You may discuss assignments with other students as you work through them, but writeups must be done alone. No downloading / copying of code or other answers is allowed. If you use a string of at least 5 words from some source, you must cite the source.

Submission

For the written portion, please submit your solutions to Gradescope.
For the programming portion, please create a tar archive containing just `magic.py`, and submit it to Autolab. The programming portion will be autograded.

1 Written [40 Points]

1.1 Problem Formulation [8 Points]

Consider the problem of constructing (not solving) crossword puzzles: fitting words into a rectangular grid. The grid, which is given as part of the problem, specifies which squares are blank and which are shaded. Assume that a list of words (i.e., a dictionary) is provided and that the task is to fill in the blank squares by using any subset of the list. Formulate this problem precisely in two ways:

a. As a general search problem. Choose an appropriate search algorithm and specify a heuristic function. Is it better to fill in the blanks one letter at a time or one word at a time?

b. As a constraint satisfaction problem. Should the variables be words or letters?

Which formulation do you think will be better? Why?

1.2 Bidirectional Search [6 Points]

Determine whether each of the following problems can be solved using bidirectional search. Explain your answer briefly.

a. Finding a path from one location to another, given a map of the domain.

b. Coloring a map (with 3 colors).

c. Solving the 8-Puzzle.

d. Solving the N Queen problem.

e. Solving the Rubiks cube.

f. Solving cryptarithmetic problems (i.e. FORTY + TEN + TEN = SIXTY).

1.3 Uniform-Cost Search [5 Points]

Give an example (draw using Latex, attach an image, etc.) of an acyclic directed graph (possibly with negative edge costs) on which uniform-cost search does not find the optimal path. Your graph should have no more than six nodes. Clearly label the start and goal states. Find the cost of the optimal path, and explain the path that uniform-cost search generates.

1.4 A* Tree Search [5 Points]

Give an example of an acyclic directed graph and a heuristic function, with which running A* tree search does not find an optimal path. All the edge costs should be positive. Your graph should have no more than six nodes. Clearly label the start and goal state of the graph, and the heuristic at each node. Find the cost of optimal path and the cost of the path that A* tree search finds.

1.5 A* Graph Search [6 Points]

Give an example of an acyclic directed graph and an **admissible** heuristic function, with which running A* **graph** search does not give an optimal path. All the edge costs should be positive. Your graph should have no more than eight nodes. Clearly label the start and goal state of the graph, and the heuristic at each node. Find the cost of optimal path and the cost of the path that A* graph search finds.

1.6 Sub-optimality Bound [10 Points]

Suppose that the heuristic overestimates the shortest path from any state to the goal by a factor of at most ϵ, where $\epsilon > 1$. Prove that the cost of the path found by A* tree search is at most ϵ times the cost of the optimal path.

2 Programming: Magic Square Variant [60 Points]

2.1 The Problem

Traditionally, a Magic Square is a $n \times n$ square grid filled with distinct values $1, 2, ..., n^2$ such that each row, column, and diagonal sum to the same value. Per the Wikipedia page, there exist many methods for constructing magic squares of a given size.

In this problem, we wish to use CSPs to tackle a variant of the magic square problem. In particular, we remove the constraint that the numbers must be distinct (we can have an arbitrary number of repeats), restrict the domain of numbers to only non-negative digits (0-9), specify the desired sum for each row/column/diagonal (they may be different), and fix the values in certain squares.

More formally: Given a $n \times n$ partially filled square, we wish to fill in the empty cells with integers (between 0 and 9, inclusive) so that each row, each column, and each diagonal sums up to a particular value. The numbers in the cells do not necessarily have to be unique.

Consider the following example, where $n = 3$:

			0	10
	5		2	8
				9
3	12	10	5	9

There exists a solution to this example. One possible solution is the follows:

	5	**5**	0	10
	5	**1**	2	8
	2	**4**	**3**	9
3	12	10	5	9

2.2 Format

Please use Python 2.7 to complete this assignment, in a file named `magic.py`. We have given you some starter code (in the Autolab handout) that shows how to read from the input file and print the output. The specifics of what each line represents is described below. Note that `magic.py` is the only file that you should edit.

2.2.1 Input

The first (and only) command line argument will be the input file.
The format of the input file is as follows:

1. The first line is n, the dimension of the square.
2. The next n lines contains the values of the n rows. If the value is an integer in the range [0,9], then it represents a fixed number that you cannot change. Otherwise, the value is -1 which represents a placeholder for an empty value. (So each of the n lines contains n space-separated values.)
3. The next line is the desired sum for each row, from top to bottom. (So this line contains n space-separated values.)
4. The next line is the desired sum for each column, from left to right. (So this line contains n space-separated values.)
5. The next line is the desired sum for the 2 diagonals. The first value is the desired sum for the top-left to bottom-right diagonal, and the second value is the desired sum for the top-right to bottom-left diagonal. (So this line contains 2 space-separated values.)

For example, the input file for the above example would look like the following:

```
3
-1 -1 0
5 -1 2
-1 -1 -1
10 8 9
12 10 5
9 3
```

2.2.2 Output

Please print your output to stdout.
The format of the output should be the following:

1. The first line should be either `True` if there exists a valid solution, or `False` if there does not exist a valid solution.
2. (a) If the first line is `True`, then the next n lines should contain the values of the n rows. If there are multiple valid solutions, output any one of them. (So each of the n lines should contain n space-separated values.)
   (b) If the first line is `False`, print nothing more.

For example, a sample output for the above example may look like the following:

```
True
5 5 0
5 1 2
2 4 3
```

2.3 Method

Write your code in `magic.py`. You must solve this problem as a CSP. We recommend you use the most-constrained-variable (the variable with the smallest number of remaining values) and least-constraining-value heuristics, and also conflict-directed backjumping (you do not need to implement the most-constraining-variable heuristic, forward checking, or no-good learning). Your code will be tested for speed and correctness on test cases that were designed to perform well on a solution that properly implements the heuristics and conflict-directed backjumping. A good approach may be to start by first implementing basic backtracking, then adding the heuristics, and then adding backjumping. You will likely pass some tests along the way. Note: since Python does not handle recursion well, you will probably need to do most of your work in a non-recursive manner.

2.4 Testing

We have provided a file `check.py` to check a test case against a completed square. The first command line argument should be the path to test case file. The second command line argument should be the path to completed square file. The third argument should be `True` or `False`, depending on whether there is actually a valid solution to the magic square. (For the test cases that we provided, the ones with suffix `_fail` in the file name are the ones without a valid solution.)

In particular, after you've written your code in `magic.py`, you may test against `tests/basic1.txt` by running the following two commands in the `handout` directory:

```
python magic.py tests/sample1.txt > output.txt
python check.py tests/sample1.txt output.txt True
```

If the second command prints `True`, then you have passed the test. Otherwise, the second command prints `False`, in which case you have failed the test.

For all the test cases provided, a completed solution should be able to solve them within a second on the unix.andrew.cmu.edu machines (this is what Autolab uses).

We have provided a `driver.sh` script to time your code against all the sample test cases, with a one second time limit on each. To use this script, use on a unix.andrew.cmu.edu machine, and run the following in the `handout` directory:

```
./driver.sh
```

2.5 Submission

Please submit to Autolab a tar file that contains the file named `magic.py`. One way to do so is to use the following command on the same directory as your `magic.py` file:

```
tar cvzf handin.tar magic.py written.pdf README.txt
```

As of now, there are no restrictions on the submission count. Good luck!