CSE 150 Programming Assignment #3

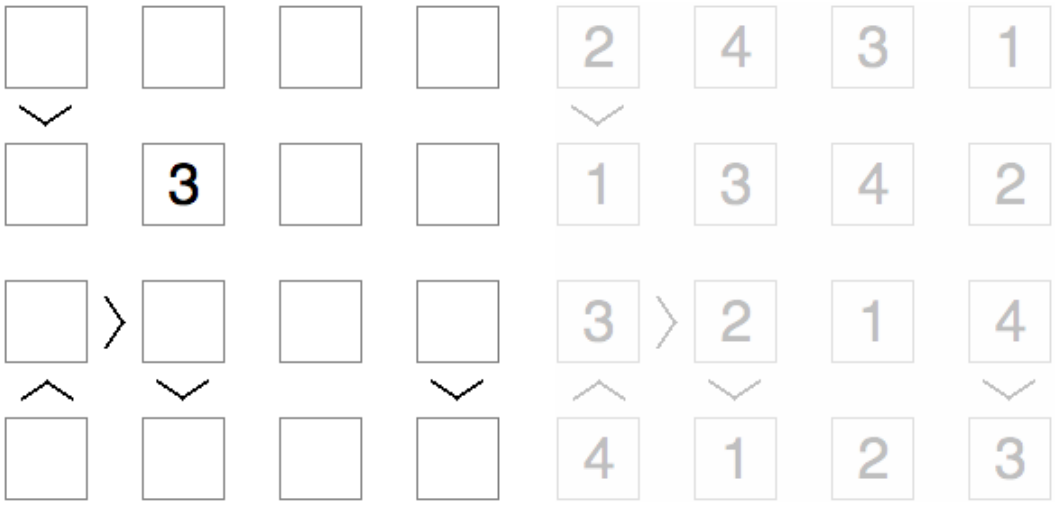
Due: 02/06 at 11:59 PM
Thanks to Gary Cottrell and Tomoki Tsuchida

1 Overview

In this project, you will develop algorithm to solve Futoshiki game. Although there are several different approaches to solving Futoshiki puzzles, you will be developing a generic binary constraint satisfaction problem (CSP) solver to solve the puzzles.

2 The Futoshiki Puzzle

The Futoshiki puzzle is played on a square grid, such as 5 x 5. The objective is to place the numbers 1 to 5 (or whatever the dimensions are) such that each row and column contains each of the digits 1 to 5. Some digits may be given at the start. In addition, inequality constraints are also initially specified between some of the squares, such that one must be higher or lower than its neighbor. These constraints must be honored as the grid is filled out.



(a) An unsolved Futoshiki puzzle. (b) A solved Futoshiki puzzle

Figure 1: An example of an "easy" Futoshiki puzzle (from http://www.futoshiki.org/. Numbers from 1 through 4 are filled such that each row and column contains each of the digits, and the inequalities are followed.

Like Sudoku, the game requires satisfying *AllDiff* constraints across rows and columns of the board. However, there will be additional binary constraints for the inequalities between the cells.

3 Provided Code

We have provided code to deal with the basic mechanics of the game, and some stub code for each problem already. In particular, the `Futoshiki` class provides methods to parse the board and produce binary CSPs. The following is a brief description of each class provided in the `assignment3.py` code:

- The `Variable` class represents a particular variable (such as X_1). Each `Variable` has an associated domain, which in the case of Futoshiki is either the integers 1 through N (the size of the board) or the number already displayed on the board.
- The `Variables` class represents a collection of `Variable` objects with the ability to "begin transaction" and "rollback". The "rollback" method will revert any changes made to the variable domains (and assignments) that occurred since the last "begin transaction" method. You should find this method useful for implementing the backtracking search with the AC3 inference. (See problem 3 for more details.)
- The `Constraint` class represents a binary constraint between two variables `var1` and `var2`. The constraint is satisfied (`is_satisfied(val1, val2) == True`) when the values of `var1` and `var2` (`val1` and `val2`) satisfy the relationship specified by `relation`. In the case of Futoshiki, "not equal" (`operator.ne`), "less-than" (`operator.lt`) and "greater-than" (`operator.gt`) relations are used.
- The `Constraints` is a collection of `Constraint` objects with the ability to look up constraints by the variables. **It also has the ability to return the neighbors of the node in the constraint graph, as well as the arcs involved in the constraints**. For example, to find all constraints involving a variable X_i (neighbors of X_i), you can use

  ```
  for constraint in constraints[x_i]:
          # All constraint.var1 are x_i
          constraint.is_satisfied(....)
  ```

  Please see the class docstring for more detailed usages.
- The `BinaryCSP` class defines a binary CSP problem, and it has the `variables` (a list of CSP variables) and `constraints` (an instance of `Constraints`). The `assignment()` method returns a dictionary of current variable assignments. (Note that this is provided for viewing purposes only, you probably do not need to use this method in your implementation.)
- The `Futoshiki` class has methods to parse and print the board, as well as a method to produce a binary CSP (`to_binary_csp()`) and to use a CSP solver to solve a Futoshiki puzzle `solve_with()`).

The Futoshiki board is represented with 0 for the "empty" cells and other numbers for pre-filled cells. ">", "<", "v" and "^" (hat) characters are used to represent left-right and top-down inequalities. A typical initial Futoshiki board looks like the following:

input1.txt

```
0>0>0 0
      v
0 0 0>0

0 0 0 0
^
0 0<0 1
```

Utilities

You can use the `fetch_puzzle.py` to fetch puzzles of various difficulties from http://www.futoshiki.org/. To do this, you can run the following in the command-line:

```
$ python fetch_puzzle.py [board size] [difficulty] [puzzle id]
```

For instance

```
$ python fetch_puzzle.py 4 1 128
```

To fetch 4x4 "easy" level puzzle (with ID=128).

Once you've implemented a solver, you may also use the `solve_puzzle.py` to solve a puzzle from the command line. By default, it calls the `backtracking_search` method in `p6_solver.py`, but you can optionally specify a different python filename and a method name to use as the solver.

For example:

```
$ python solve_puzzle.py < ../problems/p6/in/input1.txt
```

To solve the above puzzle, or

```
$ python solve_puzzle.py ../solutions/p3_basic_backtracking.py
      < ../problems/p6/in/input1.txt
```

to use the basic backtracking solver.

You can also pipe the output from `fetch_puzzle.py` directly, for e.g.

```
$ python fetch_puzzle.py 4 1 110 | python solve_puzzle.py
```

Automated Test Cases

You can also perform a subset of automated testing by running `test_problems.py` in the `tests` directory:

```
$ cd tests
$ python test_problems.py
```

This executes the test corresponding to problems 1 to 6 by giving some input in the `in` directories and comparing the output against ones in `out` directories. The tests will be reported as a "failure" if the output of your code does not match the text files the `out` directories. A good practice is to run the tests before doing the problems and observe that they fail. Then, once you implement the problems correctly, your tests should pass. There will be more test cases in the actual online submission site, and you are encouraged to add more of your own inputs and outputs in the `problems` directory!

4 Problems

Problem 1

Implement the `is_complete` method that returns `True` when all variables in the CSP has been assigned.

Hint: The list of all variables for the CSP can be obtained by `csp.variables`. Also, if the variable is assigned, `variable.is_assigned()` will be `True`. (Note that this can happen either by explicit assignment using `variable.assign(value)`, or when the domain of the variable has been reduced to a single value.)

Problem 2

Implement the `is_consistent` method that returns `True` when the variable assignment to value is consistent, *i.e.* it does not violate any of the constraints associated with the given variable for the variables that have values assigned.

For example, if the current variable is X and its neighbors are Y and Z (there are constraints (X, Y) and (X, Z) in `csp.constraints`), and the current assignment is $Y = y$, we want to check if the value x we want to assign to X violates the constraint $c(x, y)$. This method would not not check $c(x, Z)$, because Z is not yet assigned.

Problem 3

Implement the basic backtracking algorithm in the `backtrack()` method. It is "basic" in a sense that the variable ordering, value ordering and inference heuristics are not implemented yet. In other words, **you only need to implement the `backtrack()` method in this problem**. (But you should of course make calls to the `select_unassigned_variable()` and other methods.)

Hint: As noted earlier, you may find it necessary / useful to be able to revert any changes that have been made to the variable assignments and domain changes. To do this, you can use the transaction-inspired methods in the `Variables` class:

```
csp.variables
csp.variables.begin_transaction()
# Do whatever you need with the variables (assignment, domain reductions)
csp.variables.rollback()
# This undoes whatever
```

Problem 4

Implement the AC3 algorithm in the `ac3()` method. Depending on the `arc` parameter given, it should also act as the Maintaining Arc Consistency (MAC) algorithm described in p.218 of the textbook. You do not need to worry about preserving domain changes in this method.

Problem 5

Implement the variable and value ordering heuristics in `select_unassigned_variable` and `order_domain_values` methods. For the variable heuristics, implement the minimum-remaining-values (MRV) and the degree heuristic as the tie-breaker. For the value ordering, implements the least-constraining-value (LCV) heuristic.

Problem 6

Complete a faster backtracking search algorithm by augmenting the basic backtracking algorithm with the MAC inference and the variable and value ordering heuristics written so far.

Problem 7

Submit a write-up for this project in PDF. You should include the following:

- Description of the problem and the algorithms used in the problems (culminating in the solution in Problem 6).
- Run the solver you developed on different types of puzzles and measure how the solution time varies with the board size and difficulty ratings. Show your results in two separate graphs (one for the board size and another for the difficulty rating). Summarize your finding in a paragraph.
- **Extra credit**: You may also analyze the effect of each type of heuristics (inference, variable and value ordering). Compared to the basic solver in p3, how much does each type of heuristic speed up the solver for this problem?
- A paragraph from each author stating what their contribution was and what they learned.

Your writeup should be structured as a formal report, and we will grade based on the quality of the writeup, including structure and clarity of explanations.