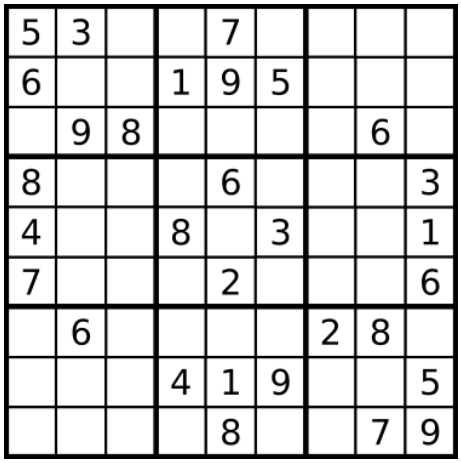
COMP3620/6320 Artificial Intelligence

Assignment 2: CSP - Solving Sudoku

The Australian National University
Semester 1, 2014



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

1 Background

Constraint Satisfaction Problems (CSPs) are a class of problems where, unlike the previous search problems we have encountered, states have a simple, standard representation.

In a CSP we have a set of variables, each of which has a given domain, which is a set of possible values the variable can take. We also have a set of constraints, which constrain the values of one variable based on the values of other variables. A state in this problem is an assignment to some or all of the variables a single value from each of their domains.

CSPs are useful for representing a wide range of problems. In this assignment, we will be using a very simple kind of CSP to solve Sudoku problems. In particular, we will be looking at CSPs where variables have finite domains and where we can have only binary and unary constraints, and special *all-diff* and *all-same* constraints.

Sudoku, is a fun, if sometimes frustrating puzzle that most, if not all, of you should have tried before. The point of the puzzle, an example of which is shown at the top of this page, is to fill in all of the squares with a number from 1 to 9. You must ensure that each number appears exactly once in each row, column and 3 by 3 box.

You can read all about this type of CSP problem and Sudoku in Chapter 6 of the text book (or Chapter 5 of the previous edition, though that lacks info about Sudoku).

2 Preliminary: Sudoku and Backtracking Search

Some easy Sudoku problems can be solved deductively, by simply observing that in some squares there is only one possible number. When this number is inserted, it restricts the possibilities for others squares and we can hopefully proceed. Sometimes we are not so lucky. There may not be enough information to find squares with only a single possibility. In this case we have to guess and then backtrack when we get stuck.

In this assignment you will be working with an algorithm we have written for you which does just that, called backtracking search. We have implemented the basics of the algorithm and made a framework where you will implement *variable selection* and *value ordering heuristics* and *inference* (constraint propagation) methods to hopefully make the search more efficient.

Once you have implemented these heuristics and inference methods, you will run some experiments to determine their performance.

All of the code you will have to write for this assignment is contained in the files:

- `heuristics.py`
- `inference.py`
- `generator.py`

The other file which is important to read is `csp.py` as this contains the data structures you will have to read to implement the required heuristics and inference methods.

The subdirectory `problems` contains a set of 10 sudoku problems which you will need to test your heuristics and inference methods on. These problems have a syntax as described in `problem_syntax.txt`. In short, you can define variables with a given domain. You can define binary and unary constraints on these variables and also special all-diff and all-same constraints, which the system compiles into normal binary constraints.

Unary constraints are removed in preprocessing, by restricting the domains of variables, so the whole search takes places with just binary constraints.

It is important to note that these constraints are specified using the model given in the text book, where a constraint explicitly lists the valid pairs of values. However, within the solver constraints are stored as a *constraint graph*. In a constraint graph nodes are (variable, value) pairs and edges between nodes indicate that they cannot occur together in a solution. Remember this when you are working with the constraints!

To get started try running `python solver.py -h` for help options. You can test a simple problem, running the default heuristics, with no inference methods by typing `python solver.py problems/sudoku01.csp -k`.

3 Q1: Variable Selection Heuristics (10 marks)

In `heuristics.py` there are 5 variable selection heuristics, which when given a CSP and a current *assignment* (of values to variables) will select the next variable for the backtracking search to branch on.

We have implemented the first of these to get you started, the `next_variable_heuristic`, which simply returns the next unassigned variable in the list of variables.

For this question, you will need to implement:

1. `degree_heuristic` – the maximum degree heuristic (4 marks);
2. `mrv_heuristic` – the minimum remaining values heuristic (4 marks);
3. `degree_mrv_heuristic` – the maximum degree heuristic with the minimum remaining values heuristic for tie-breaking (1 marks);
4. `mrv_degree_heuristic` – the minimum remaining values heuristic with the maximum degree heuristic for tie-breaking (1 marks).

See the comments in `heuristics.py` and `csp.py` for hints about what data structures to use.

4 Q2: Value Ordering Heuristics (5 marks)

In `heuristics.py` there are 2 value ordering heuristics, which when given a variable x, a CSP, and the current assignment will return an ordering over the values in the current domain of x.

We have implemented the first of these `default_value_ordering` to get you started. This heuristic simply returns values in the order they appear in the variable definition in the problem definition.

For this question, you need to implement the `least_constrained_value_ordering` function, which returns a list of values ordered preferring those which rule out the fewest choices for neighbouring variables in the constraint graph (5 marks).

Again, see the comments in `heuristics.py` and `csp.py` for hints about what data structures to use.

5 Q3: Forward Checking (10 marks)

In `inference.py` there are a two inference (constraint propagation) functions, which are used to hopefully make solving CSPs easier. For this question you need to implement the forward checking algorithm in the function `forward_checking`.

This algorithm can be run whenever a variable x is assigned a value v. The algorithm reduces the current domains of x by removing any values that are inconsistent with v. It indicates if there is a conflict detected, and otherwise returns a list changes to the problem, including new assignments.

You can find a brief description of this algorithm in the Constraint Satisfaction Problems chapter of the text book. Also see the comments in `inference.py` ideas about how to proceed.

6 Q4: Arc Consistency (10 marks)

For this question you need to implement the AC-3 (arc consistency) algorithm and the MAC (maintaining arc consistency) algorithms. They differ only slightly, in their starting set up and both should be implemented together in the `arc_consistency` function in `inference.py`.

The AC-3 algorithm is used by the solver as a preprocessing step to prune values from the the domains of variables if these values are inconsistent with all values for neighbouring variables in the constraint graph.

The MAC algorithm does arc-consistency starting with the arcs of a single variable, whenever we make an assignment during search.

You can find a description of these algorithms in the Constraint Satisfaction Problems chapter of the text book. Also see the comments in `inference.py` ideas about how to proceed.

10 marks will be awarded for correctly implementing both algorithms. Partial marks will be awarded for a partial or incorrect implementation.

7 Q5: Experiments (10 marks)

For this question you will run some experiments with the heuristics and inference methods that you implemented (and the ones we implemented for you).

You should have noticed that when you run the solver, it returns the number of nodes expanded and the search time. Generate a table which contains the expanded nodes (min, max, average, median), and runtime (min, max, average, median) for each individual heuristic and inference method used within search and as a preprocessing step, for each of the 10 supplied Sudoku puzzles (7 marks).
Also include in your table three interesting combinations of heuristics and inference methods, your choice of which.

Explain why you think the worst performing performs worst and the best performing performs best on the example puzzles (3 marks).

8 Q6: Generating Sudoku Puzzles (5 marks)

Write code in `generator.py` to generate random Sudoku puzzles with a given number of squares filled in. You should output the puzzles in the same CSP format as the given input puzzles.

You should follow the following procedure when generating a puzzle with k squares filled:

- While there are less than k squares filled, randomly choose a square with no value filled in and fill in a random value that does not occur anywhere else in the same row, column or 3 by 3 box;
- If all squares are filled in, stop;
- Otherwise, if it is not possible to choose such a square and value, then there is a *conflict*, so stop.

If you reach a conflict, the the resulting Sudoku puzzle is obviously not solvable. Otherwise, it may have one or more solutions.

Your code should print only the CSP encoding of the puzzle you generate to stdout. If your program is unable to place the required number of values due to a conflict, it should print a single line containing the number of values it was able to place.

9 Q7: Probability of Generating Solvable Puzzles (optional, 5 bonus marks)

Using your generator and the solver, determine the probability of randomly generating a solvable Sudoku problem when filling in different numbers of boxes with initial values. Provide statistics for every number of starting values from 1 to 81, using at least 100 samples at each size. The generator failing to place the required number of numbers will count as an unsolvable instance.

Graph this information in a sensible way. Include this graph along with a brief explanation of your results, in `experiments.pdf`.

Some problems may prove very difficult to solve. If a problem takes more than 1 minute to solve you may discount it even though this may skew your results slightly. If you have this problem, make an additional graph which indicates the percentage of problems which you were actually able to solve with each number of variables.

You can still get full marks without doing this question, but the bonus marks can help you make up for marks you may lose elsewhere.

10 Submission

When you have finished the assignment, make and submit (on Wattle) a `uxxxxxx.tgz` file containing the following files:

- `heuristics.py`
- `inference.py`
- `generator.py`
- `experiments.pdf`

Please do not submit anything other than a PDF to describe your experiments. We will not mark this question if you submit it .doc, .docx, or any other such format. You can produce this in any way you like and then output it to a PDF. All modern office suites will allow you to export (or print) to a PDF. If you want to give us something beautifully typeset, why not try LaTeX?

Auto-grading will be used to assess the correctness of your code, but all submissions will be looked at manually as well. Please comment your code as it helps us to determine your intent and give part marks when things don't work as intended. Also include docstrings on all classes, methods and functions you write (including type strings).