2A :: Intelligence artificielle

Dashboard / My courses / IA / Labs / Lab 2: Constraint Satisfaction Problem with Sudoku

Lab 2: Constraint Satisfaction Problem with Sudoku

ENSICAEN - Computer Science Department

CS 2I1AE1: Artificial Intelligence

Lab 2: Constraint Satisfaction Problem with Sudoku

Duration: 1 session

In this lab, you will find solutions for problems that are hard to solve without a convenient representation as a *Constraint Satisfaction Problem (CSP)*. You will program several flavors of constraint propagation algorithms.

Table of Contents

Introduction

Sudoku (meaning "*only one single number can fit*" in Japanese) is a puzzle in a 9x9 grid so that each column, each row, and each of the nine 3x3 boxes that compose the grid contains all the digits from 1 to 9 (e.g, Figure 1). The puzzle setter provides a partially completed grid, which typically has a unique solution.

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

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

Figure 1: Initial (left) and solved (right) Sudoku puzzles.

General Instructions

Download the following archive: lab2.zip

This project provides the necessary classes and some puzzle samples. There are 7 puzzles in the module samples.py. The puzzle 0 is a small 4x4 puzzle shown in Figure 2 that can be used for testing purpose. In contrast, the puzzle 6 is claimed by The Telegraph to be the hardest Sudoku Puzzle in the world.

		2	
2			4
		3	
	1		

1	4	2	3
2	3	1	4
4	2	3	1
3	1	4	2

Figure 2: A smaller version of the Sudoku puzzle.

Note that the option '`--help`' displays the list of all possible arguments :

```
./sudoku.py --help
```

Also note that the file README.md lists all the commands that appear in this lab.

A first agent `BS` is coded in the API. It performs a Backtracking Search algorithm (a variant of the uninformed Depth-First Search algorithm), namely generate a complete solution and test if it is acceptable. If not, backtrack, change one cell and test the solution again until a solution is found or all combinations have been tested.

Check our Backtracking Search agent against the puzzle 0:

```
./sudoku.py -a BS -p puzzle0
(Number of explored states: 31)
```

Caveat: Don't use this agent against the other puzzles, some of them take several years to complete.

Exercise 1: Forward Checking Algorithm

Write a new solver that performs the Forward Checking algorithm. Implement the method `solve()` in the class `FC` stub of the module agents.py.

```
class FC( Agent ):
   def solve( self, grid, heuristic_function = default_heuristic ):
```

The method `solve()` should return a dictionary with a unique value for each cell of the puzzle: `{0:'1', 1:'4', 2:'2',..., 15:'2'}`

You can base your work on the code below and the code of the class `BS`:

```
function FC-SEARCH(domains) returns solution/failure
  return RECURSIVE-FC-SEARCH({ }, domains)
function RECURSIVE-FC-SEARCH(assignment, domains) returns solution
  IF assignment is complete THEN return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(assignment, domains)
  FOREACH value in ORDER-DOMAIN-VALUES(var, assignment, domains) DO
      add {var = value} to assignment
      domains1 ← FORWARD-CHECKING(var, value, copy(domains))
           IF domains1 != failure THEN
                result ← RECURSIVE-FC-SEARCH(assignment, domains1)
                IF result != failure THEN return assignment
      remove {var = value} from assignment
  return failure
function FORWARD-CHECKING(var,value,domains) returns domains/failure
  FOREACH xi in domains whose values are constrained by var
    IF xi=v is inconsistent with var=value THEN
                remove v from the domain of xi in domains
                IF the domain of xi is empty THEN return failure
  return domains
```

- The function `SELECT-UNASSIGNED-VARIABLE(domains, assigment)` is the heuristic function. It returns the next cell to consider among the domains. Note that the cell is removed from the domains.
- The function `ORDER-DOMAIN-VALUES()` is meant to implement the heuristic that orders the values of the choosen variable. For now, you can just return the values list as such without sorting it.

- Your method `FORWARD-CHECKING()` should return the new variable domains or None if the variable domains become inconsistent. The method `grid.get_related_cells(cell)` returns the list of cells that cannot have the same value as `cell`. You only have to remove this value from the domain of these cells. Return `None` if one cell has no more value in its domain otherwise return the new `domains`.
- Add the instruction `self.increment_count()` to count the number of recursive function calls.

Some useful methods of the class `Grid`:

class Grid	
`get_domain_values()`	Returns the list of the value domain for all **unset** cells. It is represented as a dictionary. For instance, it returns the following list for the first line of the Sudoku in Figure 2: `domains = {0:('1','2','3','4'), 1:('1','2','3','4'), 2: ('2'), 3: ('1','2','3','4')...}`
`display(assignment)`	Displays the grid with the specified assignment. The parameter *assignment* is a dictionary where `{(1,3): 0,(3,0): 1],…}` means that the cell `(x=1,y=3)` is set to 0.
`get_related_cells(cell)`	Returns all the cells (ie, cell coordinates `(x,y)`) that are in conflict with the specified `cell` regarding the current domains and assignment (ie, the cells in `domains` that cannot have the same value in column or in row or the immediate neighbour than the `cell`). The result is a list of `(x,y)` coordinates.

Some documentation on useful Python functions:

`dictionary[e].remove(v)`	Removes a value *v* in an element *e* of a *dictionary*.
`dictionary[x]=v`	Sets the value *v* for the key *x*.
`if x in dictionary:`	Tests whether the key *x*is in the dictionary.
`del dictionary[x]`	Removes the key *x* and its value.
`new_list=copy.deepcopy(list)`	Creates a separate deep copy of a list.

Test your solution against all the puzzles:

```
python sudoku.py -a FC -p puzzle0
(Number of explored states: 18)
```

```
python sudoku.py -a FC -p puzzle1
(Number of explored states: 5942)
```

```
python sudoku.py -a FC -p puzzle4
(Number of explored states: 1887)
```

```
python sudoku.py -a FC -p puzzle5
(Number of explored states: 10074)
```

```
python sudoku.py -a FC -p puzzle6
(Number of explored states: 2597)
```

Exercise 2: Heuristic

Write a better heuristic function that selects the future cell to consider. The default heuristic `default_heuristic(domains)` chooses the next cell to examine randomly among the list of pending cells. Implement the Most Constrained Variable one.

Fill in the function stub `my_heuristic(domains, assignment, grid )`, where `domains` is the dictionary of the current domain for each cell (eg., `{0: ['1','2,'], 1:['2'], 3: ['4', '5'], …}`) and `assignment` is the current assignment (eg., `{0:'1', 11:'2', 13: '4', …}`).

Some useful documentation on Python `dictionary`:

`for key, value in dictionary.items():`	The `items()` method returns a list of tuple pairs (key, value).

Then, try this heuristic with the harder sudoku puzzles 4, 5 and 6 and compare the number of explored states without and with your heuristic.

```
python sudoku.py -a FC -f my_heuristic -p puzzle4
(Number of explored states: 81)
```

```
python sudoku.py -a FC -f my_heuristic -p puzzle5
(Number of explored states: 232)
```

```
python sudoku.py -a FC -f my_heuristic -p puzzle6
(Number of explored states: 157)
```

Exercise 3: Arc Consistency Checking

In the class `AC3`, write a new agent which only performs preprocessing with the arc consistency technique. It should return the current domains which is a partial solution where all the inconsistent values have been removed. Base your implementation on the algorithm given in of the lecture slides.

Note: In Python, use `dictionary[element].remove(value)` to remove a value of an element in a dictionary.

Try your implementation with the puzzle 2:

```
python sudoku.py -a AC3 -p puzzle2 -c
```

Note: the option `-c` in the previous command checks your solution againts the expected solution.

The solution for this puzzle2 is:

```
{ 1 } { 3 6 8 9 } { 5 }       | { 2 8 } { 7 } { 2 8 9 }    | { 4 } { 3 6 9 } { 2 9 }
{ 7 8 9 } { 3 6 7 8 9 } { 4 } | { 1 2 8 } { 1 8 } { 5 }    | { 3 7 9 } { 3 6 7 9 } { 2 9 }
{ 2 } { 7 9 } { 7 9 }         | { 3 } { 6 } { 4 }          | { 5 7 9 } { 8 } { 1 }
{7 8 9} { 2 7 8 9 } { 7 8 9 } | { 1 4 5 8 } { 3 } { 1 8 }  | { 1 5 7 9 } { 1 4 5 7 9 } { 6 }
{ 4 } { 5 } { 1 }             | { 6 } { 9 } { 7 }          | { 8 } { 2 } { 3 }
{3} { 6 7 8 9 } { 6 7 8 9 }   | { 1 4 5 8 } { 2 } { 1 8 }  | { 1 5 7 9 } { 1 4 5 7 9 } { 4 9 }
{ 6 } { 4 } { 3 8 9 }         |  { 7 } { 1 8 } { 1 2 3 8 } |  { 1 3 9 } { 1 3 9 } { 5 }
{ 5 7 8 } { 1 3 7 8 } { 3 7 8 } | { 9 } { 1 5 8 } { 1 3 6 8 } | { 2 } { 1 3 4 } { 4 8 }
{ 5 8 9 } { 1 3 8 9 } { 2 }   | { 1 5 8 } { 4 } { 1 3 8 }  | { 6 } { 1 3 9 } { 7 }
```

Exercise 4: Arc Consistency combined with Forward Checking

Write a new agent `AC_FC` that is a pure copy of the Forward Checking agent (`FC`) but where arc consistency is used as a preprocessing step to prune the variable domains before caling the recursive function `self.__recursive_fc_search(grid, domains, {})`.

Try your implementation against the puzzles 4, 5 and 6 and compare the number of expeored states without and with the heuristic:

Results without heuristic:

```
python sudoku.py -a AC_FC -p puzzle4
(Number of explored states: 474)
```

```
python sudoku.py -a AC_FC -p puzzle5
(Number of explored states: 5543)
```

```
python sudoku.py -a AC_FC -p puzzle6
(Number of explored states: 832)
```

Results with heuristic:

```
python sudoku.py -a AC_FC -f my_heuristic -p puzzle4
(Number of explored states: 81)
```

```
python sudoku.py -a AC_FC -f my_heuristic -p puzzle5
(Number of explored states: 232)
```

```
python sudoku.py -a AC_FC -f my_heuristic -p puzzle6
(Number of explored states: 157)
```

Exercise 5: Maintaining Arc Consistency

Finaly, implement the Maintaining Arc Consistency algorithm that uses the arc consistency technique as the preprocessing and propagation steps to write the last agent `MAC`.

Try your implementation against the puzzles 4, 5 and 6 and compare the number of expeored states without and with the heuristic:

Results without heuristic:

```
python sudoku.py -a MAC -p puzzle4
(Number of explored states: 81)
```

```
python sudoku.py -a MAC -p puzzle5
(Number of explored states: 459)
```

```
python sudoku.py -a MAC -p puzzle6
(Number of explored states: 161)
```

Resukts with heuristic:

```
python sudoku.py -a MAC -f my_heuristic -p puzzle4
(Number of explored states: 81)
```

```
python sudoku.py -a MAC -f my_heuristic -p puzzle5
(Number of explored states: 95)
```

```
python sudoku.py -a MAC -f my_heuristic -p puzzle6
(Number of explored states: 96)
```

Last modified: Sunday, 17 November 2024, 9:25 AM

◄ Démo Pacman avec Python et TkInter

Jump to...

Lab 3: Adversarial Search Problem with Pac-Man ►