

Project 3: Constraint Propagation

# Overview:

A classic Sudoku puzzle is an 81-square grid. Each row of the grid, each column, and each 3x3 sub-grid must contain the numbers 1-9 without repeats. Rows are labeled with alphabetical characters A-I, and columns are labeled with numbers 1-9. The configuration follows:

A1 A2 A3| A4 A5 A6| A7 A8 A9  
 B1 B2 B3| B4 B5 B6| B7 B8 B9  
 C1 C2 C3| C4 C5 C6| C7 C8 C9  
---------+---------+---------  
 D1 D2 D3| D4 D5 D6| D7 D8 D9  
 E1 E2 E3| E4 E5 E6| E7 E8 E9  
 F1 F2 F3| F4 F5 F6| F7 F8 F9  
---------+---------+---------  
 G1 G2 G3| G4 G5 G6| G7 G8 G9  
 H1 H2 H3| H4 H5 H6| H7 H8 H9  
 I1 I2 I3| I4 I5 I6| I7 I8 I9

A square’s “peers” consists of 20 squares—8 in the column, 8 in the row, and 8 in the sub-grid (note that 4 of these overlap). No peer square is allowed to have repeating values. Peers of E5 include:

A1 A2 A3| A4 A5 A6| A7 A8 A9  
 B1 B2 B3| B4 B5 B6| B7 B8 B9  
 C1 C2 C3| C4 C5 C6| C7 C8 C9  
---------+---------+---------  
 D1 D2 D3| D4 D5 D6| D7 D8 D9  
 E1 E2 E3| E4 **E5** E6| E7 E8 E9  
 F1 F2 F3| F4 F5 F6| F7 F8 F9  
---------+---------+---------  
 G1 G2 G3| G4 G5 G6| G7 G8 G9  
 H1 H2 H3| H4 H5 H6| H7 H8 H9  
 I1 I2 I3| I4 I5 I6| I7 I8 I9

A skeleton Sudoku class is provided.

# Part 1: Coding

## Sudoku Class

The provided Sudoku class has several existing methods. A description of these methods follow:

|  |  |
| --- | --- |
| \_\_init\_\_() | Initializes the grid, squares, units, peers, and values structures   * grid – a dictionary that contains the current structure of the puzzle * values – a dictionary that contains possible values for each square * units – a dictionary that contains all units to which a square belongs * peers – a dictionary that contains all peers of each square   Note the keys of each dictionary are “A1”, “B1”, “C1”, … , “I9” |
| cross\_product(A, B) | A method that returns the cross product of two lists. Used to create the grid and 3x3 squares. |
| \_\_str\_\_() | Returns the string representation of the puzzle. |
| load\_file(filename) | Populates the grid from a file. Used for setup. |
| solve() | Calls the search and propagate method in a nested manner. |
| grid\_values(grid) | Converts a list of values into the appropriate grid dictionary. Used for setup. |
| propagate() | Performs constraint propagation  **\*\* You are responsible for the code of this method. \*\*** |
| search() | Performs backtracking search  **\*\* You are responsible for the code of this method. \*\***  **\*\* You might need to create helper functions to do heuristics, un-propagation, etc. \*\*** |

## Part A: Constraint Propagation

### Overview

Constraint propagation, as noted in the textbook and lecture is applying constraints in a forward manner. Some easy Sudoku puzzles are solvable by forward checking alone. Here is an example of a simple puzzle solved entirely by forward checking (puzzle is provided in file **sudoku\_simple.txt**).

|  |  |
| --- | --- |
|  |  |

### Assignment

Complete the code of the propagate method to perform constraint propagation via forward checking. You should be able to solve the provided puzzles **sudoku\_easy1.txt** and **sudoku\_easy2.txt** using this method.

## Part B: Backtracking Search

### Overview

Because forward checking will not solve more-challenging Sudoku puzzles, you need another algorithm to help. Using backtracking search provides another way to solve a Sudoku puzzle relatively easily. (Actually, doing constraint propagation and then backtracking provides an even easier way to do this).

The backtracking search algorithm is in the textbook on page 215, and a better version in the slides and in Lab 4. Rather than expanding nodes in LIFO order, I suggest expanding nodes in the order of **minimum remaining values**.

### Assignment

Complete the code of the search method in the **project3\_sudoku.py** file. To test this method, you may use **sudoku\_hard1.txt** or **sudoku\_hard2.txt**.

# Part 2: Analysis

1. (12 Points) Give a representation of Sudoku as a search problem.

|  |  |
| --- | --- |
| **Initial State** | The puzzle with the given values |
| **Possible Actions** | Update the domain, assign a variable based on a single value in the domain, propagate the constraint based on the value placed in a cell. |
| **Transition Model** | When the model was updating the domain based on variables that were placed in a cell. It was also in transition when it was exploring a path that had to be guessed based on the values that were available in the domain. |
| **Goal Test** | Count of cells that are assigned a variable with an empty domain, and a check to make sure that the variables are unique in their row, column, and region. |
| **Step Cost** | The amount of times that the program had to step through each domain and then had to step back and reassigned the domain if an incorrect variable was assigned to a given cell. |
| **Path Cost** | The number of times the program had to recursively call itself in order to find the proper assignment for each cell and the amount of time that it took to come back from that recursive call. |

1. (15 Points) When working toward a solution of the Sudoku problem, each state can be consistent, complete, or partial. Explain each concept specifically using the Sudoku variables/domains/constraints. Also, explain what happens when after each state (e.g., what happens after a consistent state, after an inconsistent state, a complete state, etc.)

|  |  |
| --- | --- |
| **Consistent State** | If a cell is assigned a variable the cell in the row has a unique variable, each cell in a region has a unique variable, and each cell in the column has a unique variable. The domain that each of the cells can take is between 1-9 inclusively. The puzzle is in a consistent state when all the cells and the domains do not violate the uniqueness clause. It is also consistent when the domains are decremented to represent the values that the cell associated with that domain can take. |
| **Inconsistent State** | The puzzle is inconsistent when there is more than one cell in a row, column, or region that is not unique. The puzzle is also inconsistent when the domains do not represent the values that the cell can take either in terms or having too many values or not enough values that the cell can take. The puzzle was in an inconsistent state when it was updating the domains. There was no time that the cell was assigned a variable that violates the constraint. |
| **Partial Assignment** | The domains represented values that the cells can take before the assignment of a value to that cell. This occurred when the puzzle was updating and placing possible values in the different cells. |
| **Complete Assignment** | When each cell was populated with a variable that did not violate any constraints and the domain did not contain any values. |
| **Solution** | The solution is found when each cell in the column, row, and region is assigned a unique variable and the domains do not contain any values satisfying the given constraint. |

1. (8 Points) This problem asked you to use the MRV (minimum remaining values) heuristic to solve the problem. Explain why this was helpful to solving the problem, and how NOT using a heuristic would have affected the search for the solution.

To deal with the minimum remaining value I created a priority queue that takes in a cell location and the amount of values in the domain. When the time came to guess when forward propagation did not work I was able to find the domain with the fewest possibilities available. This allowed me to make a guess that had a higher probability of being correct. Doing this also lessened the amount of time that the program had to spend in a loop checking and then backtracking. It was important with a memory intensive program that uses recursion to make smart decisions before moving forward. Not using this heuristic would cause the program to backtrack more often and take more time to solve. Finding values quickly was very important because every found value reduced the domain of the other cells significantly. The quicker that I could be sure that the value was correct the quicker the program performs. If the program did not use this method it would have to bruit force search for the values that each cell can take. With 81 cells minus the given values and the domains associated with each of those cells, using bruit force would take a long time and have to search over a lot of different combinations of values.

# Submission and Due Date

Place all files used for this project into a folder named lastname\_project3 (of course, you should use your own last name). ZIP the folder and submit it to the dropbox **at or before Sunday, March 18, 2018 at 11:59pm**. The rubric follows.

# Rubric

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Points Possible** | **Points Gained** | **Comments** |
| Constraint Propagation |  |  |  |
| Propagation works | 15 |  |  |
| Solves Easy Puzzles | 15 |  |  |
| Backtracking Search |  |  |  |
| Backtracking Works | 15 |  |  |
| Solves Hard Puzzles | 15 |  |  |
| Solves Super Hard Puzzles | 5 |  |  |
| Writeup |  |  |  |
| 1. Search Problem | 12 |  |  |
| 1. State Description | 15 |  |  |
| 1. Heuristic Discussion | 8 |  |  |