COMS 4701 Artificial Intelligence

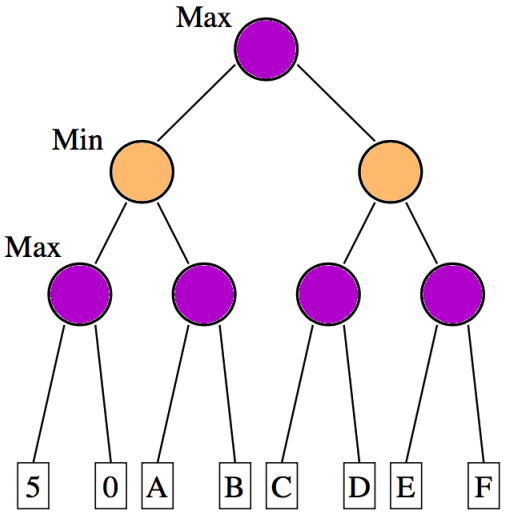
Homework 3 – Due date: Friday March 6th, 2020

Justify all your work to receive full credit

A - Written

1. Alpha-Beta Pruning

   Consider the following game tree. Let A through F be real numbers. We explore the nodes with minimax and α-β pruning.



   (a) Give a domain for A, so B is pruned.

   (b) Let $A = B = 5$. Suggest values for C and D such as the subtree with children E and F is pruned.

2. Iterative Deepening in Adversarial Search

   Provide at least two reasons why Iterative Depth Search (also called Depth First Iterative Deepening DFID) is useful in solving adversarial two-player games like chess.

   Suggested reading: Section 7 of Depth-First Iterative Deepening, Korf 1985:

   `https://courseworks2.columbia.edu/courses/95748/files/folder/READING?preview=6834361`

3. Consider a dataset with 90 negative examples and 10 positive examples.

   (a) Suppose a model built using this data predicts 30 of the examples as positive (only 10 of them are actually positive) and 70 as negative. What are the numbers of True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN).

   (b) What measure derived from these numbers can help detect the poor prediction ability of the model? Consider the measures of accuracy, precision, recall, specificity defined below. Justify your choice.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad Precision = \frac{TP}{TP + FP}$$

$$Recall(sensitivity) = \frac{TP}{TP + FN} \quad Specificity = \frac{TN}{TN + FP}$$

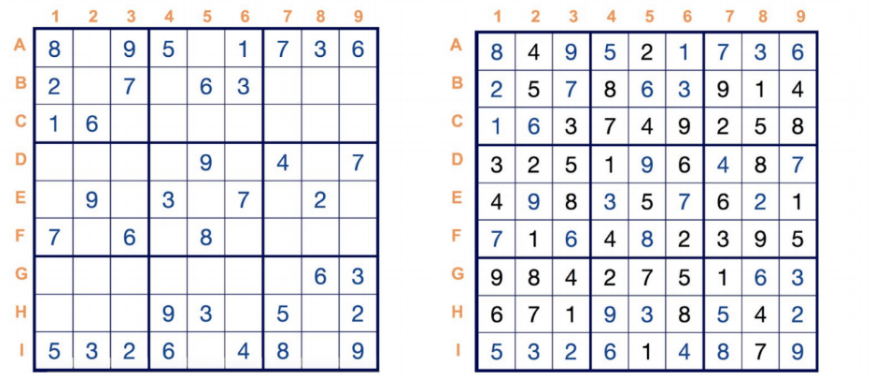
.

B - Programming

Please read all the sections carefully:

I. Introduction
II. Backtracking Algorithm
III. Important Information
IV. What You Need To Submit
V. Before You Submit

I. Introduction



	1	2	3	4	5	6	7	8	9
A	8		9	5		1	7	3	6
B	2		7		6	3			
C	1	6							
D					9		4		7
E		9		3		7		2	
F	7		6		8				
G								6	3
H				9	3		5		2
I	5	3	2	6		4	8		9

	1	2	3	4	5	6	7	8	9
A	8	4	9	5	2	1	7	3	6
B	2	5	7	8	6	3	9	1	4
C	1	6	3	7	4	9	2	5	8
D	3	2	5	1	9	6	4	8	7
E	4	9	8	3	5	7	6	2	1
F	7	1	6	4	8	2	3	9	5
G	9	8	4	2	7	5	1	6	3
H	6	7	1	9	3	8	5	4	2
I	5	3	2	6	1	4	8	7	9

The objective of Sudoku is to fill a 9x9 grid with the numbers 1-9 so that each column, row, and 3x3 sub-grid (or box) contains one of each digit. You may try out the game here: **sudoku.com**. Sudoku has 81 **variables**, i.e. 81 tiles. The variables are named by **row** and **column**, and are **valued** from 1 to 9 subject to the constraints that no two cells in the same row, column, or box may be the same.

Frame your problem in terms of **variables**, **domains**, and **constraints**. We suggest representing a Sudoku board with a Python dictionary, where each key is a variable name based on location, and value of the tile placed there. Using variable names **A1**... **A9**... **I1**... **I9**, the board above has:

- sudoku_dict["B1"] = **2**, and
- sudoku_dict["E2"] = **9**.

We give value **zero** to a tile that has not yet been filled.

Executing your program

Your program will be executed as follows:

```
$ python3 sudoku.py <input_string>
```

In the starter zip, `sudokus_start.txt`, contains hundreds of sample unsolved Sudoku boards, and `sudokus_finish.txt` the corresponding solutions. Each board is represented as a single line of text, starting from the top-left corner of the board, and listed left-to-right, top-to-bottom.

The first board in `sudokus_start.txt` is represented as the string:

003020600900305001001806400008102900700000008006708200002609500800203009005010300

Which is equivalent to:

0 0 3 0 2 0 6 0 0
9 0 0 3 0 5 0 0 1
0 0 1 8 0 6 4 0 0
0 0 8 1 0 2 9 0 0
7 0 0 0 0 0 0 0 8
0 0 6 7 0 8 2 0 0
0 0 2 6 0 9 5 0 0
8 0 0 2 0 3 0 0 9
0 0 5 0 1 0 3 0 0

Your program will generate `output.txt`, containing a single line of text representing the finished Sudoku board. E.g.:

483921657967345821251876493548132976729564138136798245372689514814253769695417382

Test your program using `sudokus_finish.txt`, which contains the solved versions of all of the same puzzles.

III. Backtracking Algorithm

Implement **backtracking** search using the minimum remaining value heuristic. Pick your own order of values to try for each variable, and apply forward checking to reduce variables domains.

- Test your program on `sudokus_start.txt`.
- Report the number of puzzles you can solve and the mean, standard deviation, min, and max of the runtime over all puzzles in `sudokus_start.txt`.

IV. Important Information

1. Test-Run Your Code

Test, test, test. Make sure you produce an output file with the **exact format** of the example given.

2. Grading Submissions

We test your final program on **20 boards**. Each board is worth **5 points** if solved, and zero otherwise. These boards are similar to those in your starter zip, so if you solve all those, you'll get full credit.

3. Time Limit

No brute-force! Your program should solve puzzles in **well under a minute** per board. Programs with much longer running times will be killed.

4. Just for fun

Try your code on the world's hardest Sudokus! There's nothing to submit here, just for fun. For example:

Sudoku:
800000000003600000070090200050007000000045700000100030001000068008500010090000400

Solution:
812753649943682175675491283154237896369845721287169534521974368438526917796318452

IV. What You Need To Submit

1. Your `sudoku.py` file (and any other python code dependency)
2. A `README.txt` with your results, including the:
   - number of boards you could solve from `sudokus_start.txt`,
   - running time statistics: min, max, mean, and standard deviation.

V. Before You Submit

- Ensure that your file is named `sudoku.py`
- Ensure that your file compiles and runs.