CS311 – Spring 2022
Introduction to Artificial Intelligence
HW Assignment #2

Please see the accompanying video on the course Panopto site.

Due dates:
Friday, March 18th by 11:00am (via the submission link on canvas)

Solving Sudoku with AC3

Rules for group work:

I suggest working in a GROUP on this assignment. You may work in groups of 2 students. **You will submit only ONE assignment (including the written portion) for the group**. You may not "divide and conquer" this assignment. That is, all students must be present (virtually is ok) and contributing while any work on the project is being done. In other words, you must "pair-program" to complete the assignment. Along with the honor code, you must include, in a comment at the top of your assignment the following statement: "All group members were present and contributing during all work on this project." This statement will be treated in the same manner as the honor code, so please make sure it is present and true.

For this assignment, you will be creating an intelligent agent that solves Sudoku. Your program will use backtracking search with the AC3 algorithm.

First, get familiarized with the game:

http://www.sudoku.com/

A skeleton program is provided here:

https://middlebury.instructure.com/courses/10232/files/folder/HWs/HW2

The program allows you to select a difficulty level – easy, medium, hard, or random (all will be solvable boards). The program also allows you to choose whether to play using a GUI or on the console.

Every game of Sudoku is a Constraint Satisfaction Problem with the following properties:

- Variables are the 81 cells on the board.
- Domain of each variable is an integer from [1, 9].
- Constraints are that all the variables in each row, column, and 3x3 sub-board are distinct.

Part 1:

Write a program that solves Sudoku by implementing the pseudocode discussed in class. You may use your own pseudocode that is different from the one we discussed, but your program must use backtracking search with AC3.

Program Requirements (if using our pseudocode):

1. Your program should contain the following methods:
    - `AC3Init()`
        - sets up `globalDomains`
        - calls `allDiff()`to set up and `neighbors` and `globalQueue`
        - makes the initial call to `backtrack()` (already done for you).
    - `allDiff()` – Fills in `neighbors` and `globalQueue` by defining the constraints between sets of variables, specifically that all neighbors (i.e. variables that are in the same arc) must be assigned distinct values (explained on page 213 of the course reference book).
    - `backtrack()` – Takes as parameter a cell number and the current domain values and does the following:
        - calls `AC3()`to check whether the CSP can be satisfied (i.e. there is a valid board) given the domain values so far; returns true or false accordingly
        - if the CSP can be satisfied, then it selects a value to assign to the given cell and recursively calls `backtrack()` on the next cell to check whether this new assignment still satisfies the CSP. Note that this second recursive call is actually the one that checks the validity of assigning the value to the initial cell.
    - `AC3()`– performs the actual AC3 algorithm (pseudocode on page 209).
    - `Revise()`– part of the AC3 algorithm – determines if the domain of a variable can be reduced (pseudocode on page 209).
2. For the GUI to work properly, your `backtrack()` method must return true if a valid board is discovered and false otherwise; the final board must be stored in `int[][] vals`.

You may also find the following data structures useful (they are included in the code) but you are not required to use them:

- `globalDomains` – an array of size 81 of ArrayLists of Integers. Each index contains the domain for the corresponding cell.
- `neighbors` – an array of size 81 of ArrayLists of Integers. Each index contains the neighbors (as defined by the arcs) for the corresponding cell.
- `globalQueue` – a Queue containing the initial set of Arcs.
- `Arc` – a simple class for storing the indices of a pair of cells. This may be useful for storing the arcs of the CSP.

The program outputs the number of total operations (i.e. the number of times `Revise()` and `valid()` are called) and the number of recursive operations.

Part 2:

Your job now is to create a customized solver (using any techniques you wish) that solves Sudoku as quickly as possible. This is the solver you will enter into the class tournament (see below). Here are some techniques you can try:

- Most constrained variable
- Least constraining value
- Forward checking
- Some combination of the above
- Some new idea of your own!

Write the code for your customized solver in the method `customSolver()`. You may, of course, write other methods for your customized solver, but they should all be called from `customSolver()`.

When you run the program without the GUI, it will output the amount of time (in milliseconds) that it takes to solve the puzzle. We will use (averages of) these times during the tournament so your goal should be to reduce this time.

Output:

We would like to grade your programs as efficiently as possible. So please make sure your program does not output anything other than what is asked for above. Points will be deducted for extraneous output/debugging statements.

ReadMe:

As always, please provide a ReadMe that explains the following:

1. The names of the students in your group.
2. Names of files required to run your program.
3. Any known bugs in your program (you will be less penalized for bugs that you identify than for bugs that we identify).

Your ReadMe should also contain a brief discussion of your customized solver. Did it perform better than the AC3 solver? If so, why do you think so? If not, can you think of any ways to improve it?

Submission:

Please include, in a comment at the top of every file you submit, the following:

- Full names of students in the group
- Group honor code: "All group members were present and contributing during all work on this project"
- Middlebury honor code

Zip only the following files and use the HW1 link on canvas to submit exactly **ONE** zipped file per group:

- SudokuPlayer.java
- ReadMe

Name your zip file as follows: HW2_<last names of all group members>. So, for example, if your group consists of the last names Alvarez and Baker, you should name your file HW2_AlvarezBaker.

Grading:

[70] - Correctness of implementation of above methods
- [14] `AC3Init()`
- [14] `allDiff()`
- [14] `backtrack()`
- [14] `AC3()`
- [14] `Revise()`

[20] – Your customized solver
[10] - Comments in code/Readme

Part 3 – Submit one typed (you may hand-write the graph/tree) copy per group with the names of both students via the link on canvas by the due date specified above. Name your file as you did for the Programming section, e.g. WrittenHW2_AlvarezBaker.doc, WrittenHW2_AlvarezBaker.docx, or WrittenHW2_AlvarezBaker.pdf

1. [9] Problems 6.2 Parts (a) – (c) only from the course reference text.

2. [20] You are in charge of scheduling for computer science classes that meet Mondays, Wednesdays, and Fridays. There are 5 classes that meet on these days and 3 professors who will be teaching these classes. You are constrained by the fact that each professor can teach only one class at a time. The classes are:
   • Class 1 - Intro to Programming: meets from 8:00-9:00am
   • Class 2 - Intro to Artificial Intelligence: meets from 8:30-9:30am
   • Class 3 - Natural Language Processing: meets from 9:05-10:00am
   • Class 4 - Computer Vision: meets from 9:05-10:00am
   • Class 5 - Machine Learning: meets from 9:35-10:30am

   The professors are:
   • Professor A, who is available to teach Classes 3 and 4.
   • Professor B, who is available to teach Classes 2, 3, 4, and 5.
   • Professor C, who is available to teach Classes 1, 2, 3, 4, 5.

   a) [5] Formulate this problem as a CSP in which each course is assigned a professor who is available to teach that course.
   b) [3] Draw the constraint graph for this CSP.

c) [7] Assume that courses are assigned in increasing order (so C1 first, then C2…) and professors are assigned in alphabetic order (so A, then B…). Show the search tree that yields from applying backtracking with forward checking and arc-consistency on this CSP.
d) [5] Give all solutions to this problem.