COMPSCI 683
Spring 2019
Due: February 28, 2019 at 11:59 PM
Submit via GradeScope

Assignment 2

Problem 1 (20%)

Show that any CSP that has a finite domain for each variable can be transformed into a CSP with only binary constraints. Follow these steps:

1. Show how a single ternary constraint such as "A+B=C" can be turned into three binary constraints by using an auxiliary variable. (*Hint:* consider a new variable AB whose domain is *pairs* of numbers.)
2. Next, show how constraints with more than three variables can be treated similarly.
3. Finally, show how unary constraints can be eliminated by altering the domains of variables.

Problem 2 (60%)

Sudoku is a fairly old puzzle that is now a worldwide phenomenon. You can type "sudoku" into Google, or read the Wikipedia article to get more information than you could possibly imagine. Key facts about standard Sudoku puzzles:

- Every puzzle has a unique solution.
- Every puzzle is solvable without trial-and-error, given suitable inference methods.
- The designated difficulty of a puzzle is usually determined by the difficulty of the inference methods required to solve it.

In addition to the rules, many web sites offer extensive discussion of methods that humans can use to solve Sudoku without using trial-and-error search.

You need to write a program that can solve Sudoku problem instances. To test your program, you should apply it to the 16 puzzles in the included .zip. These puzzles are from "Sudoku: Easy to Hard", by Will Shortz. The original collection includes 100 puzzles (from 1-25 are 'light and easy', 26-50 are 'moderate', 51-75 are 'demanding', and 76-100 are 'beware! very challenging').

Each puzzle is stored in a file that looks as follows:

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

1. (5) Explain how Sudoku can be represented as a CSP (how many variables are needed? what are their domains? what are the constraints?).
2. (15) Write a program that can read a Sudoku puzzle from a file and solve it using the backtracking-search algorithm that was discussed in class (slide 15). Keep in mind that instead of representing the constraints explicitly in your program, it might be easier to write a function that checks if a given variable assignment results in a conflict with the values already placed in its row/column/square.

   Try to solve a few of the Sudoku instances using this algorithm and make sure that it works. Why can plain backtracking search solve Sudoku puzzles quickly despite the size of the problem?

   In case you find it helpful, we have created a method to read in and write a Sudoku puzzle to a file. The code is included with the puzzle files.

3. (20) Now instrument `backtracking-search` so that it counts the total number of *guesses* made using the following method. Whenever the backtracking algorithm prepares to loop through a list of *k*>0 possible values, we will say that *k*-1 guesses are made. (This way, the algorithm is still charged for guessing even if it is lucky

and its first value choice succeeds.) Show the numbers of guesses made for each of the 16 instances in the above collection. Try both plain backtracking and backtracking with the MRV (minimum remaining values) heuristic.

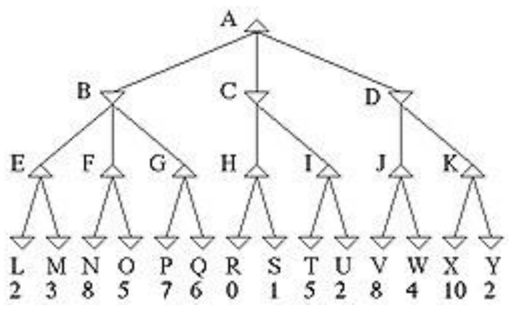
4. (20) We would like our solver never to resort to guessing. For this to happen, we will need better inference methods. In this context, an *inference method* is a function that examines the current state of the CSP and deduces additional facts about the remaining unassigned variables -- either ruling out one or more values for a variable or asserting a particular value. The method can then modify the current state accordingly. Notice that every time a method modifies the current state, other methods (or indeed the method itself) may become applicable again.
   A *waterfall* is a set of methods that is applied repeatedly, until all the methods fail to do anything useful.

   Modify `backtracking-search` (similar to Figure 6.5 in the textbook, or slide 41) so that it applies a waterfall to modify the current state before making a recursive call. (Be sure to have the program keep track of what the waterfall does so that it can be undone!) Initially, your waterfall should contain no inference methods, so nothing will change. Then add as your first inference method the arc consistency algorithm AC-3 (shown in Figure 6.3 in the textbook, or slide 28). Show the number of guesses made for each of the 16 instances when you use AC-3 as the only inference method.

   Now, find some Sudoku inference methods on the web (ones written in English, of course, not programs!). For each, describe how it works with an example; explain whether it is already covered by AC-3 and using MRV; and, if it is not covered, implement it and add it to your waterfall. After each addition, check the number of guesses for each of the 16 puzzles. Ideally, you will be able to get the numbers to zero for all puzzles. Hint: It may be possible to define your own inference methods by examining the current state at each point where a guess is made, to see what a smart human would do at that point (if one happens to be available).

Problem 3 (20%)

Consider the following game tree in which the static heuristic scores are given for all the tip nodes:



1. What move should MAX choose?
2. What nodes would not need to be examined using the alpha-beta algorithm -- assuming that nodes are examined in left-to-right order?

Problem 4 (optional)

Suppose we play a variant of Tic-Tac-Toe in which each player sees only his or her own moves. If the player makes a move on a square occupied by an opponent, the board "beeps" and the player gets another try. Would the backgammon model (the EXPECTIMAX-EXPECTIMIN algorithm described in Section 5.5) suffice for this game, or would we need something more sophisticated? Why?