CMPSCI-683 Homework Assignment #2: Constraint Satisfaction and Adversarial Search

Patrick Pegus II

March 1, 2016

Problem 1. 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.

1. Let $D_{AB} = \{a+b | a \in D_a, b \in D_b\}$ be the domain of the variable AB. Then $A+B = C$ can be replaced by $AB = C$, $A = a$, $B = b$.
2. Iteratively remove variables from non-binary constraints, such as

   $$V_1 * \cdots * V_i * V_{i+1} * \cdots * V_m = V_n$$

   by replacing it with

   $$V_1 * \cdots * V_i V_{i+1} * \cdots * V_m = V_n,\ V_i = v_i,\ V_{i+1} = v_{i+1}$$

   until only binary constraints exist. Where $D_{V_i V_{i+1}} = \{v_i * v_{i+1} | v_i \in D_{v_i}, v_{i+1} \in D_{v_{i+1}}\}$.
3. Any unary constraint U that imposes a condition on variable V, such as $U(V) = True$, can be eliminated after reducing the domain of V to only values v where $U(v) = True$.

Problem 2. 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 following 16 puzzles: See `/data/sudoku_puzzles/`.
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. Explain how Sudoku can be represented as a CSP (how many variables are needed? what are their domains? what are the constraints?).

   Sudoku can be represented as a CSP with a matrix of variables $M = V_{ij}$ for the entries in the puzzle. i and j are the row and column index, respectively. Since i and j are integers in $[1, 9]$, there are 81 variables. The domains of each V_{ij} are the integers in $[1, 9]$. The constraints are:

   (a) $V_{ij} \neq V_{ik}$, if $j \neq k$

   (b) $V_{ij} \neq V_{kj}$, if $i \neq k$

   (c) When M is partitioned into 9 3x3 matrices $m = V_{ab}$, $V_{ab} \neq V_{cd}$, if $a \neq c$ or $b \neq d$.

2. 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 16). 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. Explain why can plain backtracking search solve Sudoku puzzles quickly despite the size of the problem?

   Please see the `simpleBacktrackSearch` function in csp.py for the implementation of the backtracking algorithm. Plain backtracking search can solve easier Sudoku puzzles quickly because constraints of the puzzle often render variable assignments inconsistent quickly. Suppose no variable assignment could be determined to be inconsistent until all variables were assigned, then plain backtracing would have to enumerate all possible combinations of variable values in the worst case.

3. Now instrument backtracking-search so that it counts the total number of guesses made. 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 backtacking with the MRV (minimum remaining values) heuristic.

	Guess count	
Puzzle	Plain	MRV
1	130	171
2	832	200
10	11567	1268
15	43417	3198
25	**367431**	33045
26	**497103**	38149
48	**176487**	99282
51	**514666**	26076
62	**423581**	1319
76	**367354**	126492
81	**300203**	72855
82	**393530**	**428756**
90	**457224**	320336
95	**431714**	**619242**
99	**343688**	30412
100	**378558**	78429

As shown the MRV heuristic drastically reduced guessing, allowing most of sudoku puzzles to be completed without inference within the 5 minute timeout. Guess counts occurring during a timeout are bold face. One iteration of AC-3 was run on the puzzle domain so that MRV makes a difference.

4. Please see the inferences method of the Sudoku class in sudoku2.py for the implementation of the waterfall. Please see the ac3 method of the Sudoku class in sudoku2.py for the implementation of AC-3.

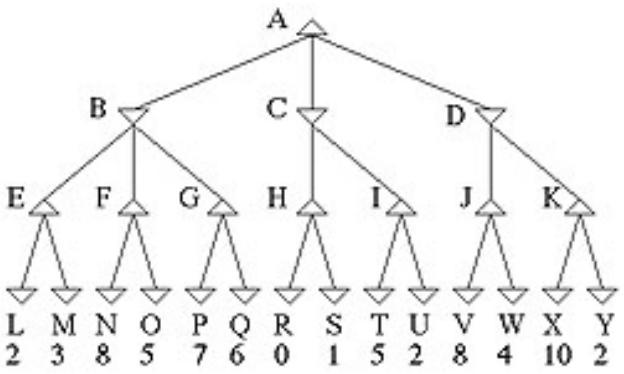
	Guess count	
Puzzle	AC-3	AC-3 + Neighbors Can't
1	0	0
2	0	0
10	0	0
15	0	0
25	0	0
26	2	2
48	19	**18**
51	7	7
62	4	4
76	10	10
81	4	4
82	35	35
90	44	**42**
95	20	**19**
99	12	12
100	10	10

The Neighbors Can't inference method assigns a value to a variable if the variable's neighbors assignments or domain do not include that value. Arc consistency limits the domain of a variable assigning a value to a variable if the domain size becomes 1. The Neighbors Can't method assigns a value to a variable if no other neighbor could possibly take on that value. As shown in the table above it made little improvement. It would have been more effective had it checked neighbors in a row, then column, then submatrix, instead of all neighbors at once. Please see the neighborsCant method of the Sudoku class in sudoku2.py for the implementation of this method.

5. Propose a method to determine the difficulty of a given Sudoku puzzle automatically using your solver. You don't have to implement your technique.

   The difficulty of a puzzle can be estimated by the number of iterations through each inference method.

Problem 3. Consider the following game tree in which the static heuristic scores are given for all the tip nodes.



1. What move should MAX choose?

   MAX should move to node D.

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

   Nodes O, Q, I, T, U, and Y would not need to be examined.

Problem 4. 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?

Something more sophisticated is needed. This variant of Tic-Tac-Toe is partially observable. Therefore, a deterministic terminal test and utility function would not suffice. It would also be difficult to use chance nodes to create a probability distribution over available actions, since the probability of available actions are not only influenced by the number of moves so far, but also the strategy of the opponent.

Problem 5. Consider the following procedure for choosing moves in games with chance nodes:

- Generate a suitable (say, 1000) dice-roll sequences down to a suitable depth (say, 8).
- With known dice rolls, the game tree becomes deterministic. For each dice-roll sequence, solve the resulting deterministic game tree using alpha-beta.
- Use the results to estimate the value of each move and choose the best one.

Will this procedure work well? Why (not)?

This procedure will work well. The large number of sequences causes the action returned most often by alpha-beta over the sequences to converge to the action resulting in the state with highest expected utility, as Expectiminimax does. When compared to Expectiminimax, it reduces time complexity by $\frac{1000}{d^8}$ where d is the number of distinct dice rolls. Additionally, the probability of each dice roll can be unknown.