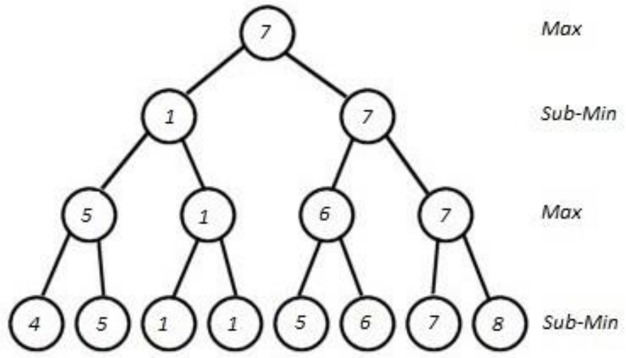
CS 486 – Assignment #3
Justin Franchetto
20375706

Section I: Adversarial Search

1) An optimal MIN will make the best move, which leaves MAX with some amount of utility, u. Suppose we have a sub-optimal MIN: if he happens to choose the best move, then the utility given to MAX will still be u. However, if he chooses a non-optimal move, MAX gets utility q > u. This means that in the very best case, our sub-optimal MIN can give MAX a utility of u, and otherwise it gives utility greater than u. Therefore, the utility, q, a sub-optimal MIN gives will be ≥ u, the utility given by the optimal MIN.

2) We consider the game tree below:



At this stage, the optimal move for MAX would be to choose (5, 1, 6, 8), which would lead an optimal MIN to choose (1, 6) at the following stage. However, if MAX recognizes that his opponent is SUB-MIN, it could make a sub-optimal move and choose (5, 1, 6, 7) at this stage. The reason for doing so is that while a SUB-MIN may be able to distinguish 6 from 8 and choose the 6 correctly, it may not be able to tell which one of 6 or 7 is better because they are more similar to each other. In this situation, SUB-MIN could mistakenly choose (1, 7), allowing MAX to have a higher utility at the final stage than it would have had against an optimal MIN. The downside to this approach is that if the SUB-MIN sees our trick, we could end up losing utility, but if they do not then we receive a higher reward.

Section II: Constraint Satisfaction (Cryptarithmetic Puzzle)

1) The following table shows the steps taken throughout the solving of this puzzle. While the puzzle is solved using backtracking and forward checking, along with the MRV and LCV heuristics, the propagation of constraints is also critical because it eliminates several possibilities based on the structure of the puzzle. We notice two things immediately:

   i. S + S = Y + X1*10 implies that if X1 is 1, then S must be ≥ 5
   ii. Z + X5 = B implies that B = Z + 1. Since B ≠ Z, then there must be a carry over, so we also have that X5 = 1

At this point, we can begin solving the puzzle. An in-depth discussion is not needed for every step, but there are a couple of steps worth noting:

i. <u>Step 1:</u> Assigning 9 to the variable E, we get that E + X4 = I + 10. This implies that X4 = 1 and also that I = 0, R = 8 and later that X2 =1 after propagating these results.
ii. <u>Step 4:</u> We assign Y = 2, which at this point leaves us with S = 1 or 6 (this turns out to be incorrect later). From this, we get that X1 = 0, which holds true until the puzzle is completed, so therefore S < 5.
iii. <u>Step 6:</u> We notice that assigning Y=2 and S=1 leaves no values remaining for A such that other constraints can be met. We backtrack at this point, assigning Y = 4 and eventually S =2.
iv. <u>Step 9:</u> Choosing Z = 6 immediately constraints B = 7 (since B = Z + 1) and A = 5.
v. <u>Step 13:</u> Finally, we assign N =1, which allows us to assign X3 = 1, since O = 3 and A = 5.

	Variable	Z	E	R	O	S	N	B	I	A	Y	X1	X2	X3	X4	X5
Step																
0		(0-9)	(0-9)	(0-9)	(0-9)	(0-9)	(0-9)	(0-9)	(0-9)	(0-9)	(0-9)	0/1	0/1	0/1	0/1	1
1		(1-7)	**Select 9**	8	(1-8)	(1-4)	(1-6)	(2-8)	0	2, 4, 5, 7	2, 4, 6, 8	0/1	0/1	0/1	1	1
2		(1-7)	**9**	**Select 8**	(2-7)	(1-4)	(1-6)	(2-7)	0	2, 4, 5, 7	2, 4, 6	0/1	0/1	0/1	1	1
3		(1-7)	**9**	**8**	(2-7)	(1-4)	(1-6)	(2-7)	**Select 0**	2, 4, 5, 7	2, 4, 6	0/1	0/1	0/1	1	1
4		1, (3-6)	**9**	**8**	(3-7)	1	1,3,4,5,6	(4-7)	**0**	4,5,7	**Select 2**	0	0/1	0/1	1	1
5		(3-6)	**9**	**8**	(3-7)	**Select 1**	3,4,5,6	(4-7)	**0**	4,5,7	**2**	0	0/1	0/1	1	1
6	**Backtrack**									*BAD*		0	1	0/1	1	1
7		5,6	**9**	**8**	3, 6,7	2	1,3,5,6	6, 7	**0**	2,5,7	**Select 4**	0	1	0/1	1	1
8		5,6	**9**	**8**	3, 6,7	**Select 2**	1,3,5,6	6, 7	**0**	5,7	**4**	0	1	0/1	1	1
9		**Select 6**	**9**	**8**	3, 6,7	**2**	1,3,6	7	**0**	5	**4**	0	1	0/1	1	1
10		**6**	**9**	**8**	3,7	**2**	1,3	**Select 7**	**0**	5	**4**	0	1	0/1	1	1
11		**6**	**9**	**8**	3	**2**	1,3	**7**	**0**	**Select 5**	**4**	0	1	0/1	1	1
12		**6**	**9**	**8**	**Select 3**	**2**	1	**7**	**0**	**5**	**4**	0	1	0/1	1	1
13		**6**	**9**	**8**	**3**	**2**	**Select 1**	**7**	**0**	**5**	**4**	0	1	0/1	1	1
14		**6**	**9**	**8**	**3**	**2**	**1**	**7**	**0**	**5**	**4**	0	1	0	1	1

After assigning a value to every variable, we verify to ensure that the constraints are met and the solution is correct:

ZEROES	598392
<u>+ ONES</u>	<u>+ 3192</u>
BINARY	**701584**

We can see that the final result is consistent with the values assigned to the variables. The puzzle is complete and all constraints are met.

Section III: Constraint Satisfaction (Arc Consistency)

1) The arc from N to W is consistent because for every element in N, there is at least one element in W such that the conditions are satisfied. (1 -> Added, 3 -> Green, 5 -> Stare).

2) The arc from W to N is inconsistent because of the element, Fever. While all of the other elements in W have an element in N that satisfies the condition, there is no N such that the constraint is satisfied for Fever. By removing it from the domain of W then the arc from W to N would become consistent.

Section IV: Programming Question (Sudoku Solver)

1) We let X_{ij} be the value of the square at (I, j) in our puzzle, with $1 \leq X_{ij} \leq 9$. Then, there are three sets of constraints that we should consider:

   i) Row Constraints: Each row in the game board must contain the number 1 to 9, and each number can only appear once throughout the row. There are a total of 9 row constraints. Formally:

   $R_k : (X_{k1}, X_{k2}, X_{k3}, X_{k4}, X_{k5}, X_{k6}, X_{k7}, X_{k8}, X_{k9})$ for k = (1, 2, … 9) where AllDiff(X_{kj})

   ii) Column Constraints: Each column in the game board must contain the number 1 to 9, and each number can only appear once throughout the column. There are a total of 9 column constraints. Formally:

   $C_k : (X_{1i}, X_{2i}, X_{3i}, X_{4i}, X_{5i}, X_{6i}, X_{7i}, X_{8i}, X_{9i})$ for k = (1, 2, … 9) where AllDiff(X_{ik})

   iii) Box Constraints: Each box in the game board must contain the number 1 to 9, and each number can only appear once throughout the box. There are a total of 9 box constraints. Formally:

   $B_{(k, l)} : (X_{(k, l)}, X_{(k, l+1)}, X_{(k, l+2)}, X_{(k+1, l)}, X_{(k+1, l+1)}, X_{(k+1, l+1)}, X_{(k+2, l)}, X_{(k+2, l+1)}, X_{(k+2, l+2)})$
   for (k,l) = { (1,1), (1,4), (1,7), (4,1), (4,4), (4,7), (7,1), (7,4), (7,7)} where AllDiff(X_{kl})

   For a total of 27 constraints for the Sudoku puzzle to be solved.

2) The Sudoku solver works by implementing the following methods:

   i) **Most Restricted Variable (MRV):** When choosing which square to operate on first, we choose the one that has the fewer remaining values. We break ties using the Most Constraining Variable.

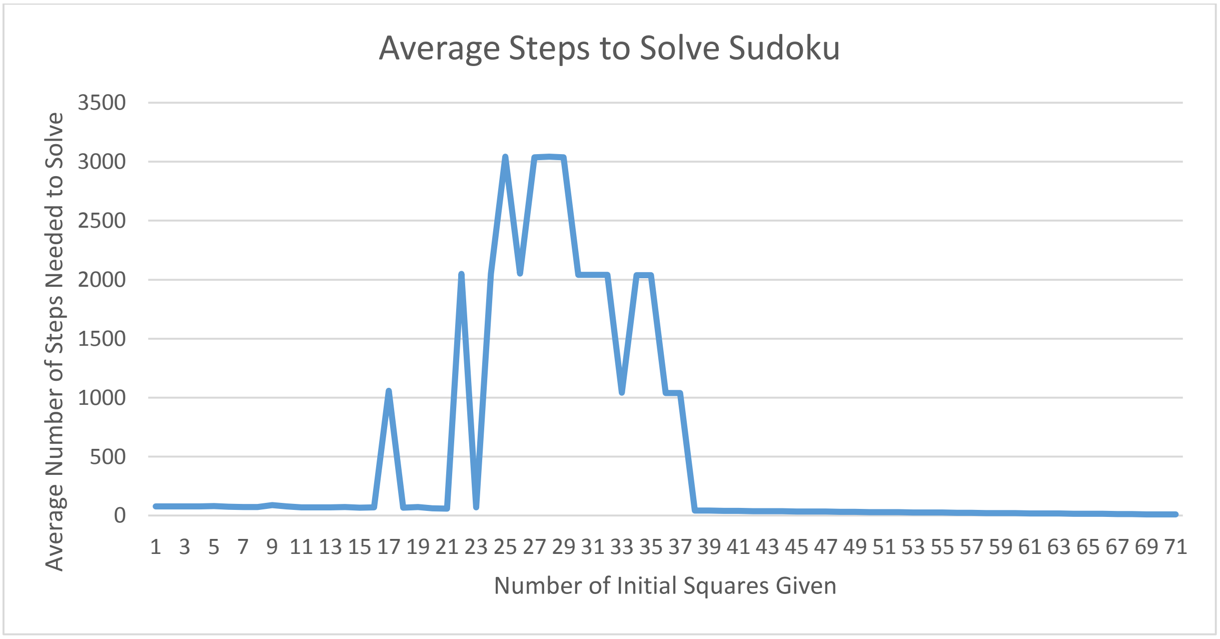
   ii) **Most Constraining Variable Tiebreaker**: When choosing the Most Restricted Variable, we always choose the one that is most constraining on the other squares if there is a tie. That is, we choose the one who has the greatest number of adjacent (row, column, box) squares who have not been assigned a value.

   iii) **Least Constraining Value (LCV):** After choosing which variable to assign to, we choose a value from its options such that it constrains the fewest number of adjacent squares. At this stage, we utilize forward checking to see if selecting such a move would be invalid and cause us to backtrack later.

   iv) **Forward Checking:** When choosing a value to assign to a square, we want to make sure that doing so will not cause an error in the puzzle. If we notice that this move would cause an error, then we discard it as a possibility at this time. It may be restored later via backtracking, if the values in adjacent squares change.

   v) **Backtracking:** It is inevitable that our solver will make mistakes without the use of some special heuristic. Both our MRV and LCV methods will return an error if backtracking is necessary, and it is at this point that our solver will backtrack. Throughout the puzzle, the solver keeps track of what moves it has made so far; if it needs to backtrack, it removes the last move made from the list and restores the square back to its previous state, removing the bad value from its possible moves. In the event that this does not open up an option for the square returned by MRV, then a full reset of the bad square is performed: its possible options are reset to be all values not used by adjacent squares at this time. It is also possible for the solver to backtrack so far that it backtracks all the way to beginning of the puzzle, but still notices that the square returned by MRV has no possible moves. At this point, the puzzle becomes stuck and gives up, assuming a number of steps greater than 10,000.

By running the solver on each of the problem instances and collecting data regarding the average number of steps needed to solve the puzzle against the number of initial values, we obtain the following plot:



As the plot indicates, the solver performs well for puzzles with a number of starting values outside the range of 21 to 39; in fact, the solver uses the minimum number of moves (meaning it assigns the correct value on the first attempt for all squares in the board) for almost all puzzles outside this range. This is because puzzles with too many or too few initial values have either very loose or very tight constraints on what can go in the remaining squares. In either case, the solver is able to easily fill the remaining squares, because either there are so many possibilities that it chooses wrong with low probability (because there are not many wrong moves), or there are so few possible values left for it to choose from, respectively. With little to no backtracking necessary, the solver can solve the puzzle in something very close to the minimum moves.

However, the solver performs much worse on puzzles that have a number of starting values between 21 and 39, and in some cases it exceeds the 10,000 step limit. The puzzles in this range have the a number of starting numbers such that it is difficult for the solver to decide which squares should be assigned to first and what should be assigned to them. Even though a tiebreaker is used for the MRV, there is not one used when choosing the LCV, meaning the solver has to guess at which value is best to use. Forward checking helps to eliminate values that would cause an error, but it doesn't help when trying to choose value is best. For puzzles in this range, there are a greater number of ties at this stage than for puzzles outside of this range. This means that the solver is making more guesses at the LCV, and the potential for error is greater, eventually leading to backtracking and a greater number of steps to solve the puzzle, or giving up.