Approaching Sudoku Puzzles with Search Techniques from Artificial Intelligence

M.W. Hadley
Lafayette College
111 Quad Drive, Box 8395
Easton, PA 18042
hadleym@lafayette.edu

ABSTRACT

Sudoku puzzles are a popular test of memory and logic for people. The puzzle is a nine-by-nine grid that is further subdivided into nine three-by-three subgrids. Each row, column and board in the grid must have only one of each of the numbers one through nine.

This paper applies constraint satisfaction searches to sudoku. Constraint satisfaction formalizes a problem into a set of variables with domains that are restricted by given constraints. This paper presents five constraint satisfaction search techniques: backtracking search (B), B with variable ordering heuristic (BH), forward checking (F), F with variable ordering heuristic (FH) and constraint propagation (CP). (The same variable ordering technique was applied to B and F to obtain BH and FH.) Each technique was tested against sixteen sudoku puzzles. Each technique was evaluated in terms of three dimensions: first, the number of constraint checks were performed during runtime-a constraint check is when a sudoku puzzle is checked to see if any of the rows have duplicates, columns have duplicates or boards have duplicates; second, the mean path length where a path is defined as the steps from a starting puzzle until an inconsistent state or solution is reached; third, the time required to process a puzzle.

B and BH involve brute force searching through all possible ways of completing the sudoku puzzle. F is FH with the added ability to check that the domains for unassigned variables is not empty. The variable ordering technique is based on the estimating the number of constraints for each variable and for each value in the variable's domain. Variables with the most estimated constraints are selected first, and domain values with the least estimated constraints are selected first. CP uses the logic of the constraints to follow through the implications of assigning a particular value to a particular variable.

Across all comparisons that were made, CP came out with the best performance and F came out with the worst performance. BH was a close second best in performance. Other trends cannot be commented on because of the noise in the sample of sudoku puzzles. Further studies should use much larger sample sizes to sharpen the statistical analysis.

Categories and Subject Descriptors

J.0 [**Computer Applications**]: General—*Problem Solving*; G.0 [**Mathematics of Computing**]: General—*Statistical Analysis*

General Terms

Search Techniques, Constraint Satisfaction

Keywords

Backtracking, Forward Checking, Heuristics, Constraint Satisfaction

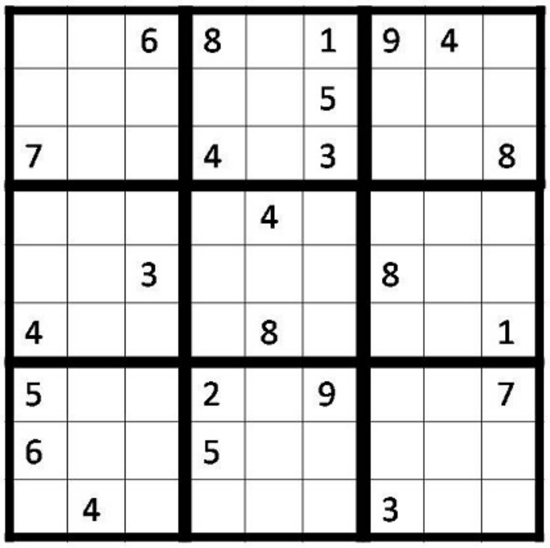
1. INTRODUCTION

Man has a preoccupation with puzzles and challenges: anything ranging from chess to jigsaw puzzles to Tetris. One such puzzle that has become very popular in the recent years is Sudoku [**?**]. Not only is man interested in solving these puzzles, but man is also fascinated with creating intelligent agents to solve these puzzles. One field that has a plethora of approaches to solving puzzles is Artificial Intelligence[1] (AI) [**?**]. In this paper, I will take three standard approaches to problem solving in Artificial Intelligence and apply them to Sudoku.

Sudoku, japanese for "single numbers," is a puzzle that is solved by filling in the blanks of a grid with the numbers one through nine. The grid is nine-by-nine and is further divided into nine three-by-three subgrids (see figure 1). There are three constraints on how you may fill in the blanks of the grid. First, each entry in a row must be different and must be in the range one through nine. Second, each entry in a column must be different and must be in the range one through nine. Finally, every entry in a subgrid must be different and must be in the range one through nine [**?**].

Each puzzle starts with a given number of blanks filled in, the givens. There should be enough givens to ensure that

[1]The definition of the field of Artificial Intelligence is hotly debated. Some argue AI is the study of modeling human intelligence. Others argue that AI is the study of problem solving. No matter what your definition, the usefulness of the theories and algorithms is undeniable [**?**].



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

Figure 1: Sudoku Example

there is only one solution to the puzzle. Using the givens and the rules of the game (the constraints), one can logically reason what numbers must be in which blanks. (Assuming all the constraints are met) when all the blanks are filled in, the sudoku puzzle has been solved [?].

The game of sudoku has a small number of simple rules. This well-defined characteristic makes the game amendable to many problem solving approaches in AI. The approaches in this paper fall under the category of search techniques. Search techniques look through a set of possible states[2] using some logic for the desired solution state [?].

In terms of sudoku, the possible states are all the ways that you can fill in the blanks of the grid. The rules of the game and the given numbers are the logic that determines whether a state is a valid state or not. The solution is achieved by looking through the possible states until you find a filled in board that meets the constraints.

I implemented three standard search approaches from AI: backtracking, forward checking and constraint propagation. I created a fourth algorithm, a heuristic, to be overlaid on top of the standard backtracking and forward checking to reduced the number of constraint checks. Thus, there are five total algorithms: backtracking, backtracking with heuristic, forward checking, forward checking with heuristic and constraint propagation.

I will analyze the performance of these algorithms on a selection of sixteen sudoku puzzles. I will show, through different performance metrics, that constraint propagation is the best algorithm (with backtracking with heuristic as a close second) and forward checking is the worst.

2. FORMAL DESCRIPTION

To be specific, the techniques I implemented come from the area of search called constraint satisfaction. This is a type of search seeks to formalizes the whole problem space into variables with domains that are subject to constraints. The advantage of constraint satisfaction is that the formalization captures the structure of the problem domain allowing for a deeper understanding of the problem at hand than other search techniques [?]. Techniques were drawn from this area of search because sudoku puzzles fit very naturally within the guidelines of constraint satisfaction.

Variables are the objects that are manipulated in the search. They are the blank entries in the Sudoku grid. Each variable has a domain which is the set of the values that variable can take. The domain for the each entry in an empty sudoku board is 1,2,3,4,5,6,7,8,9. As numbers are added to the board, the domains shrink according to the constraints. The constraints are the rules of sudoku (no two of the same number in a row, column of subgrid and only the numbers one through 9 used). The first constraint is so common in problems that it is typically given the name alldiff. The alldiff constraint requires that the specified variables must all hold different values [?]. Thus, the constraints are the set of elements in every row/column/subgrid must be alldiff and every domain must be a subset of the set 1,2,3,4,5,6,7,8,9.

There are numerous algorithms that take the constraint satisfaction approach. I choose three and decided to implement them: backtracking, forward checking and constraint satisfaction. I also implemented a heuristic to improve the backtracking and forward checking algorithms. In all, I have five implemented techniques: backtracking with heuristic and without heuristic, forward checking with and without heuristic and constraint propagation.

The first algorithm, backtracking, is a simple search that uses ordered trial-and-error to move from state to state until a solution is reached. When an inconsistent state is generated, the search "backtracks" to the last consistent state and continues searching [?].

In sudoku, the states represent partially filled puzzles. I start with a puzzle that has given numbers. I fill each blank one at a time (in order from top to bottom and left to right within each row). Each blank is assigned values in the following order: 1,2,3,4,5,6,7,8,9. After a blank is filled with value, the puzzle is checked for constraint violations. If nothing is violated, the next blank is filled in. If there is a violation, a different value is put into that blank. If there are no more values to try, I move back a blank and try a new value in it. This process is repeated until either the puzzle is completed or all states have been tried. See figure 1 for a pseudocode representation of backtracking.

The second algorithm, forward checking, has the same general structure as backtracking (fill in the blanks in the same ordered way and backtrack when an inconsistency is found), except forward checking has an "early detection" for constraint violations. An explicit data structure holding the current domains of the unfilled variables is kept. Each time a variable is filled, the data structure is updated. In addition to checking whether a filled value violates a constraint, forward checking checks to make sure no unassigned variable domains are empty. If a domain for an unassigned variable is empty, then there will be an inconsistency later unless the

[2]Possible states include states with blanks left

```
def backtracking(puzzle):

  if puzzle is complete:
    return the puzzle

  blank = findNextBlank(puzzle)

  for value in [1,2,3,4,5,6,7,8,9]:

    assignValueToBlank(puzzle, value, blank)

    if puzzle is consistent:
      result = backtracking(puzzle)

      if result != "Fail":
        return result

    removeValueFromBlank(puzzle, value, blank)

   return "Fail"
```

Figure 2: The pseudocode for Backtracking

algorithm backtracks. Checking a variable to make sure it is still consistent is called node consistency, and checking all variables for node consistency is called 1-consistency [?]. See figure 2 for the pseudocode.

On top of these first two algorithms, I overlaid the option of an ordering heuristic. A heuristic is a search trick used to try to find the solution faster. An ordering heuristic involves change the order of variables selected and/or the order of the values selected (i.e. not choosing blanks from top to bottom and/or not choosing values in the order 1,2,3,4,5,6,7,8,9) [?]. I implemented one heuristic to order the variables selected and another heuristic to order the values selected.

The variable ordering attempts to list the variables in order of decreasing number of constraints on the variable's domain (i.e. pick variables with the smallest domains first). This is generally called a minimum remaining values heuristic [?]. To estimate this without actually checking how many constraints are on each variable, I simply count the number of filled grids in each variables row, column and subgrid. This gives me a very rough idea of the number of constraints on each domain.

The value ordering attempts to list the values in increasing order of how likely they are to cause inconsistencies. This is generally called the min-conflicts heuristic [?]. In order to rank the values, I counted how many of each value were already placed in the puzzle. The values are then ordered from least occurrences to most occurrences. This is also a rough estimate, but it is cheaper to calculate than performing an analysis of all the relevant domains to find the min-conflicts ranking.

My final algorithm, constraint propagation, enforces strong 2-consistency, which is the same as enforcing both 1-consistency and 2-consistency. 1-consistency was performed by forward checking. 2-consistency is based on the idea of arc consistency (making sure that a pair of variables is consistent). Take the pair variables, V1 and V2, with domains, D1 and D2. V1 is arc consistent with V2 if for every value in D1, there is a value in D2 that does not violate any constraints. 2-consistency requires that every pair of variables be arc consistent [?].

```
def forwardChecking(puzzle, domains):

  if puzzle is complete:
    return the puzzle

  blank = findNextBlank(puzzle)

  for value in [1,2,3,4,5,6,7,8,9]:

    assignValueToBlank(puzzle, value, blank)
    Save a copy of domains
    Update domains based on the new value

    if no domain in domains is empty:
      if puzzle is consistent:
        result = forwardChecking(puzzle,\


          if result != "Fail":
            return result

      removeValueFromBlank(puzzle, value,\

      Revert domains back to the saved copy

   return "Fail"
```

Figure 3: The pseudocode for Forward Checking

This algorithm carries the same backbone as forward checking with the added 2-consistency check (see figure 3 for pseudocode). This algorithm also uses the variable and value ordering heuristics that were applied to backtracking and forward checking. 2-consistency is implemented by running arc consistency on every pair of variables until either a pair is inconsistent or no new changes to the domains are made.

3. SYSTEM PERFORMANCE

Comparing the performance of the algorithms is complicated. I created a dataset of sixteen puzzles[3] and collected data about each algorithm's performance on those puzzles[4] There are two main comparisons I performed: within puzzle and across puzzles. Each comparison was performed along the three dimensions: number of constraint checks, mean path length[5], and processing time.

[3]All puzzles came from http://www.dailysudoku.com/sudoku/. The most recent four puzzles were chosen from each category available: easy, medium, hard, very hard. The website assures that all of its puzzles have one and only one solution.

[4]The python program with the algorithms was run on an Intel T2500 2.00Ghz processor with 2.00 GB of ram.

[5]Mean path is hard to interpret. The optimal algorithm would solve the puzzle in one path. Thus, the optimal mean

```
def constraintProp(puzzle,domains):

  if puzzle is complete:
    return the puzzle

  blank = findNextBlankInOrdering(puzzle)

  for value in findValueOrdering(puzzle}:

    assignValueToBlank(puzzle,value,blank)
    Save a copy of domains
    Update domains based on the new value

    if no domain in domains is empty:
      do until no new changes are made or\
          until a pair is inconsistent:
        for every V1,V2 in domain:
          arcConsistent(V1,V2,domain)
          break if V1 and V2 are inconsistent
      if all pairs are consistent and\
          the puzzle is consistent:

        result = constraintProp(puzzle,domains)

        if result != "Fail":
          return result

     removeValueFromBlank(puzzle,value,blank)
     Revert domains back to the saved copy

   return "Fail"

def arcConsistent(V1,V2,domains):
  for every value, v1, in the domain of V1:
    if there is a value in domain of V2 that\
        does not violate constraints:
      v1 is allowable
    else:
      Remove v1 from the domain of v1

  if the domain of V1 is empty:
    return V1 and V2 are not arc consistent
  else:
    return V1 and V2 are arc consistent
```

Figure 4: Pseudocode for Constraint Propagation

A constraint check is when an algorithm checks to make sure that every row, column and subgrid meet the all different constraint; this is the "‘check if puzzle is consistent"’ line from the pseudocode representations of all the algorithms. A path is defined as the sequence of states from the starting state (the sudoku with only the givens filled in) through an ending state (an inconsistency is reached or a solution is found). Processing time was calculated through Python's time module.

From here on, the algorithm names will be abbreviated. B stands for backtracking. BH stands for backtracking with the variable and value ordering heuristic. F stands for forward checking. FH stands for forward checking with the variable and value ordering heuristic. CP stands for the constraint propagation algorithm.

3.1 Within Puzzle Results

For within puzzle comparisons, I evaluated how each algorithm performed on each puzzle. Each puzzle generates a ranking along one of the three dimensions. An example processing time ranking in descending time order would be: FH, F, CP, BH, B. B would the fastest for solving puzzle x, and FH would the slowest at solving puzzle x. I combined these results into matrices (see tables 1-3). Each row and column are labeled with algorithms. Each entry represents the number of puzzles in which the row algorithm outperformed the column algorithm.

In terms of constraint checks, the trend is very clear. B performs more checks than BH which performs more checks that F which performs more checks than FH which performs more checks than CP. CP robustly performs the least number of constraint checks in all puzzles of the dataset.

The trend in processing time is less clear. B, BH and CP perform about the same (for time performance, lower times are best). In the majority of puzzles, F is outperformed by every other algorithm. FH has a clear time increase over F, but FH still performs worse than the other algorithms in the majority of puzzles.

3.2 Across Puzzle Results

For the across puzzle comparisons, I calculated mean performance and standard deviation of performance for each algorithm across all three dimensions. The results are shown in tables 4 and 5.

The trends across mean constraint checks and mean processing time are less distinct than they seem because of the wide standard deviations. A much larger sampling of sudoku puzzles is needed to verify any trends based solely on the means. With that said, CP required the least time and performed the least number of constraint checks. CP also had the smallest standard deviations with respect to constraint checks and processing time.

path is equal to the deepest path possible (filling in every blank in the sudoku puzzle). A close to optimal algorithm that explores a few shallow paths before finding the solution path would still have a high mean path length. It is not clear to me that mean path adds any useful information above and beyond the time and constraint check metrics, so mean path is generally disregarded in the analysis.

Within Puzzle: Constraint Checks					
-	B	BH	F	FH	CP
B	-	5	0	1	0
BH	11	-	6	0	0
F	16	10	-	6	0
FH	15	16	10	-	0
CP	16	16	16	16	-

Table 1: Within Puzzle: Constraint Checks. Each entry represents how many times the row algorithm outperformed (performed fewer constraint checks) the column algorithm. B is backtracking. BH is backtracking with variable/value ordering heuristic. F is forward checking. FH is forward checking with variable/value ordering heuristic. CP is constraint propagation.

Within Puzzle: Mean Path Length					
-	B	BH	F	FH	CP
B	-	13	3	5	6
BH	3	-	6	2	2
F	13	10	-	7	6
FH	11	14	9	-	6
CP	10	14	10	8	-

Table 2: Within Puzzle: Mean Path Length. Each entry represents how many times the row algorithm outperformed (had a smaller average path length) the column algorithm.

Mean constraint checks create a fuzzy ranking; from lowest number of checks to highest: CP, FH, F, BH, B. Standard deviation of the mean follows the same ranking from lowest standard deviation to highest. B is the least consistent and has the worst constraint check performance over all the puzzles. The consistency and constraint check performance increases as you move towards CP.

The mean time also shows a rather fuzzy trend in terms of time required: F > FH B > BH > CP. The standard deviation of time matches the ranking. F has the highest time requirement and highest standard deviation. CP and BH has the lowest time requirements and also the lowest standard deviations.

4. CONCLUSIONS

I approached sudoku puzzles from a constraint satisfaction approach. I implemented five search algorithms and tested them on a sampling of sixteen sudoku puzzles. I analyzed the results in terms of within and across puzzle comparisons.

Within Puzzle: Time					
-	B	BH	F	FH	CP
B	-	7	16	12	7
BH	9	-	15	15	6
F	0	1	-	5	2
FH	4	1	11	-	5
CP	9	10	14	11	-

Table 3: Within Puzzle: Time. Each entry represents how many times the row algorithm outperformed (required less time) the column algorithm.

Across Puzzles: Means					
-	B	BH	F	FH	CP
CC	45457.59	16133.06	6562.82	4006.53	123.06
MPL	22.92	24.91	20.93	21.23	20.01
Time	5.28	2.04	10.93	6.39	1.04

Table 4: Means Along Three Dimensions: Numbers are rounded to the second decimal place. CC stands for constraint checks. MPL stands for mean path length.

Across Puzzles: Standard Deviation					
-	B	BH	F	FH	CP
CC	69204.63	20782.42	7350.75	5030.80	70.14
MPL	3.34	4.81	2.59	3.77	4.42
Time	7.96	2.41	12.52	7.91	0.84

Table 5: Standard Deviation Along Three Dimensions: Numbers are rounded to the second decimal place. CC stands for constraint checks. MPL stands for mean path length.

The results from within and across puzzle line up across the constraint checks dimension. CP > FH > F > BH > B in terms of constraint check performance. This is a rather artificial way of quantifying performance. It helps to create benchmarks against which to compare the algorithm because it is so well defined, but it does not give a complete picture of how each algorithm performs generally. It says nothing about the processing that occurs that is not directly unrelated to constraint checking.

From a functional standpoint, time is one of the most important metrics of performance. If I am going to use an algorithm, I want to know how long it will take. Unfortunately that is based on my hardware, the software being run, my inputs to the algorithm, etc.

The data's granularity is too coarse to ascertain the trend across all of the algorithms with respect to time, but the big trends pop out. BH and CP outperform the other algorithms and F performs the worst. The within puzzles comparison puts BH and CP at about the same level in terms of time required to process any given puzzle; BH and CP will outperform each other about as many times as they are outperformed. The within puzzles comparison also shows F is clearly outperformed by B, BH, CP and FH. Adding the across puzzle results gives a better sense of magnitude of these differences in performance. CP and BH both have very small time requirements, so on average they are the best choices. CP outperforms BH slightly, but it is possible that this trend results from a sampling bias or processor variance, etc. F is the worst choice on the average puzzle by a substantial margin.

The other most important metric is consistency. If I have a time estimate of how long my program will take, I want that time estimate to be accurate. Standard deviation of time gives an idea of how spread out the data points are from the mean; this gives an estimate of how constant the performance is over different puzzles. If an algorithm's time has a very low standard deviation, its time will vary little over different inputs. If an algorithm's time has a very high standard deviation, its time will vary massively over different inputs. CP and BH both have a tight standard deviation of 0.84 and 2.41 respectively, while F has the massive standard deviation of 12.52. Again CP outperforms BH slightly, and F comes out with the worst performance.

Combining all the results together, CP is the fastest and most consistent search. BH is a close second, but performs many more constraint checks. F performs less constraint checks than BH, but ends up performing the slowest and least consistent of the algorithms. B and FH sit in between the extremes, but the granularity of the statistics prevents definitive conclusions.

The other main result is that FH generally performed better than F, and BH generally performed better than B. The heuristics used were very rough estimates, so it would be interesting for future work to experiment with more accurate measures. In particular, the time required to estimate the constraints on each variable (by counting the filled in blanks in the corresponding row, column and subgrid) may not be that different from actually counting the number of distinct elements in the row, column, subgrid. The estimate will count a five in a row and a five in a column as two distinct constraints, when in reality, they are just one constraint on the variable's domain.

Future work should also look at what level of k-consistency is optimal for solving sudoku puzzles. The only difference between CP and FH is 2-consistency, but that caused large gains in performance. 3-consistency could potentially boost performance even more.

The major limitation of this study is the sample population. The first problem is size; sixteen puzzles is too small of a sample to sift out the majority of the meaningful signal from the data. A better experimental design would be to create a sudoku generator that is hooked up to implementations of the algorithms. Large random samples of puzzles could then be run to attain more stable statistics. The second problem is control; there was no control over the difficulty of the sudoku problems (other than the mysterious ranking system of the website). It would be a much better sample if the difficulty of the puzzles could be controlled. It is possible that backtracking fails miserably on difficult puzzles, but performs just as well as constraint propagation on easy puzzles.

5. REFERENCES

[1] B. Hayes. Unwed Numbers. *American Scientist.*, 94(1):12, January 2006.

[2] S. Russell and P. Norvig. *Artificial Intelligence: a modern approach.* Pearson Education Inc., New Jersey, 2001.