CS531 Programming Assignment 3: SuDoKu

Michael Lam, Xu Hu
EECS, Oregon State University

November 6, 2012

Abstract

In this assignment we design, implement and discuss constraint propagation and backtracking search algorithms in order to solve a specific constraint satisfication problem, SuDoKu.

1 Introduction

SuDoKu is a puzzle and constraint satisfication problem in which every unit (i.e. row, column or box) is an all-diff constraint. Each of the 81 squares can be represented as a variable on a domain of $\{1, 2, 3, ..., 9\}$. SuDoKu may be solved by backtracking search with constraint propagation.

A SuDoKu problem can be classified as easy, medium, hard or evil depending on what rules are required (and also if backtracking search is required) to solve the puzzle. We performed experiments to demonstrate that indeed, harder problems require more rules in order to solve them without backtracking. It turns out that the naked double and triple rules are effective enough to solve most problems without backtracking. However, we also show that backtracking is required to solve certain problems, and that backtracking search solves every puzzle.

Finally, we demonstrate that using the heuristic of picking a variable randomly for assignment during backtracking search rather than the most constrained variable yields a higher number of backtracking. Therefore the heuristic of picking the most constrained variable is effective in reducing the number of backtrackings.

2 Algorithm

The algorithm consists of two major components: constraint propagation and backtracking search. In fact, the constraint propagation step is incorporated into the backtracking search. Some puzzles can be solved with just an application of the constraint propagation. Some puzzles require "sophisticated" rules (e.g. naked triples) to solve with just constraint propagation. Other puzzles require search and backtracking.

2.1 Constraint Propagation

The constraint propagation step applies the following rules to the SuDoKu board in this order to reduce the domain values for variables:

Rule 1 Assign to any cell a value x if it is the only value left in its domain.

Rule 2 Assign to any cell a value x if x is not in the domain of any other cell in that row, column or box.

Rule 3 In any row, column or box, find k squares that each have a domain that contains the same k numbers or a subset of those numbers. Then remove those k numbers from the domains of all other cells of that unit. (Called the "naked double" and "naked triple" rule for $k = 2$ and $k = 3$ respectively.)

Afterwards, these rules are "propagated" across the board, updating the board to be consistent with the new assignments. The process repeats until the board no longer updates (converges) or a variable is found to have an empty domain (contradiction).

In our experiments section, we will show the performance of the constraint propagation step when toggling a subset of these rules on or off.

Algorithm 1 Constraint Propagation

```
function CONSTRAINT-PROPAGATION(puzzle)
while any variable's domain was updated do
  perform rule one
  perform rule two
  perform naked doubles and triple rules
  for square in puzzle do
    if domain is empty then
      return failure
    end if
    if domain is exactly one value then
      for sq in square.neighbors along row, column and box do
        remove square.value from sq's domain
      end for
    end if
  end for
end while
```

2.2 Backtracking Search

Backtracking search is a depth first search algorithm. For each step of the algorithm, it picks a variable to assign a value and then performs constraint propagation as described above. If there is a contradiction after the constraint propagation step, the search discards that assignment and backtracks, making an alternative assignment. If the constraint propagation step was successful, then the search continues making another assignment. The search continues until it finds a complete assignment that satisfies the puzzle, otherwise returns a failure.

There are several heuristics that can be implemented for the backtracking search. For our project, there are two different heuristics for choosing the variable to assign:

1. Pick the most constrained square (i.e. the variable with the least number of domain values other than those already assigned)

2. Pick any random square

In fact, the first heuristic performs better than the second heuristic in general constraint satisfication problems because choosing an assignment from

the most constrained variable will help prune the search tree faster. We will demonstrate the validity of this theory in our experiments.

In the pseudocode, SELECT-UNASSIGNED-VARIABLE is a heuristic function. In our experiment, we implemented the random slot heuristic and most constrained heuristic. Note that in our implementation, we take advantage of creating copies of puzzle states so that upon backtracking, we simply discard the entire state rather than undoing an assignment.

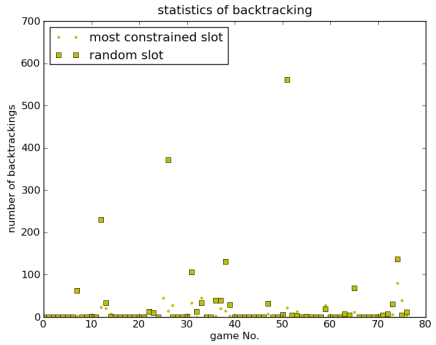
Algorithm 2 Backtracking Search

```
function BACKTRACK(assignment, puzzle)
if assignment is complete then
  return assignment
end if
var = SELECT-UNASSIGNED-VARIABLE(puzzle)
for value in var.domain do
  if value is consistent with assignment then
    add var = value to assignment
    update puzzle to new assignment
    inferences = CONSTRAINT-PROPAGATION(puzzle)
    if inferences is not failure then
      add inferences to assignment
      result = BACKTRACK(assignment, puzzle)
      if result is not failure then
        return result
      end if
    end if
  end if
  remove var = value
  remove inferences from assignment
end for
return failure
```

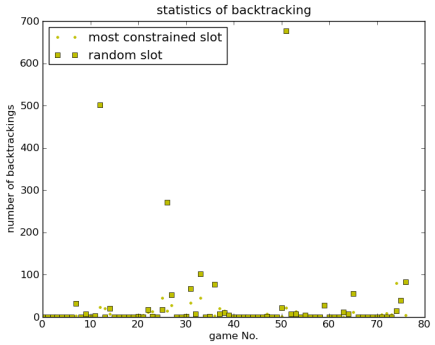
3 Experiments

We solved all 77 problems. Figure 1 shows the number of backtrackings for each problem. The "naked triple (double)" strategy exhibits impressive performance in constraint propagations. Without such kinds of rules, there

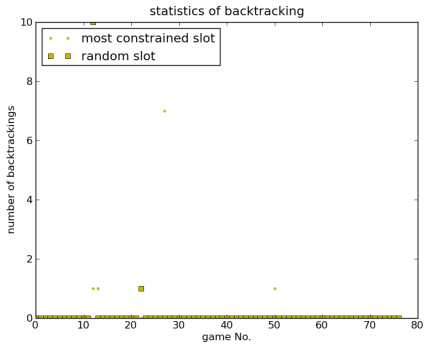
are 1/4 problems that need to use backtracking, no matter if we pick the most constrained slot or pick a slot randomly. Meanwhile, only 2 or 3 problems require backtracking for the "naked triple" case.



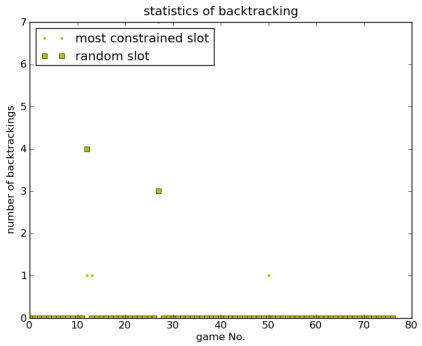
(a) Rule 1



(b) Rule 1,2



(c) Rule 1 + Naked double and triple



(d) Rule 1,2 + Naked double and triple

Figure 1: Number of backtracing with respect to each problem.

Next we show the problem completion in each difficulty level in table 1. As one introduces more rules, more puzzles are solvable. By adding backtracking search, all problems are solvable. This makes sense because backtracking search would have to go through all possible assignments.

For comparing the effectiveness of most constrained slot heuristic and random slot heuristic, we make the same experiment with random slot heuristic, which placed in last 4 rows in table 1. These two heuristics are getting same results, which confirms they are both functioning. However performance-wise, the number of backtrackings for using the constrained slot heuristic is

Complexity	r1	r1,2	r1,2+n2	r1,2+n3	r1,2+n2,3	r1,2+bt	r1,2+n2,3+bt
Easy (23):	21	21	23	23	23	23	23
Medimum (21)	3	3	10	14	19	21	21
Hard (18)	0	0	7	4	15	18	18
Evil (15)	3	3	4	3	8	15	15
Easy rand:	21	21	23	23	23	23	23
Medimum rand	3	3	10	14	19	21	21
Hard rand	0	0	7	4	15	18	18
Evil rand	3	3	4	3	8	15	15

Table 1: Number of complete problems in different combinations of rules. The default heuristic is the most constrained slot. Picking the most constrained slot or picking random slot is the same. Here, r1: rule 1, r1,2: rule1 + rule2, n2: naked double, n3: naked triple, n2,3: n2 + n3.

severely reduced when compared to the random slot heuristic as shown in the previous figure 1. From the graph, in just using rule one and backtracking, the random slot heuristic yields hundreds of backtracking for certain problems whereas the most constrained heuristic solves in less significantly less than one hundred backtrackings. Of course, the random heuristic is random so one does expect the random heuristic to perform well sometimes.

For justifying the difficulty of problems, the average number of filled-in numbers are concerned, as shown in table 2.

	easy	medimum	hard	evil
average number	34.70	29.05	26.28	26.13

Table 2: Average filled-in numbers for each problem level.

The number of rules being used is an important factor. We also use it to estimate the difficulty of a problem and show the average number of different kinds of rules used for each set of problems. The results are shown in table 3. The trend is clear that more hard problems requires more number of rules. Note that for this experiment we use the combination of all rules and backtracking for making sure every problem can be solved.

We further compare with two heuristics in the number of rules being taken. Figure 2 shows all pair of comparisons. By applying these two heuristics, a problem uses more or less the same number of rules. This is consistent with our experiment in the number of complete problems.

	r1	r2	n2	n3	bt
Easy (23):	42.26	0.04	22.00	12.70	0
Medimum (21)	46.95	0.14	31.14	16.81	0
Hard (18)	49.06	0.33	32.11	15.94	0.06
Evil (15)	59.40	0	38.20	22.33	0.60

Table 3: Average number of rules used by problems in different levels. Here, r1: rule 1, r2: rule2, n2: naked double, n3: naked triple, bt: number of backtrackings.

4 Discussion

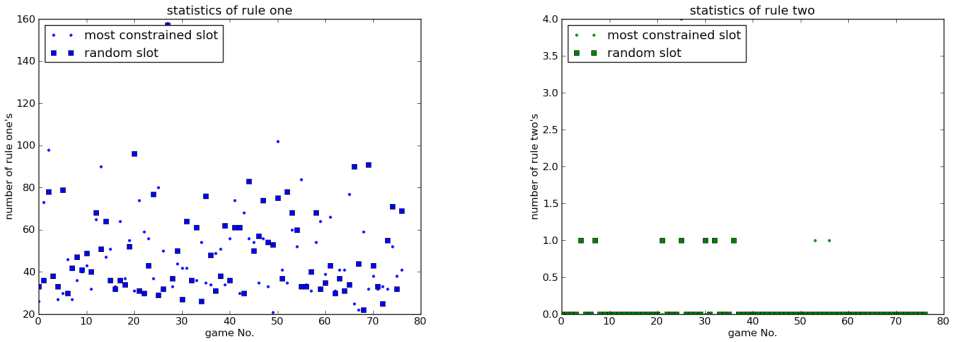
We place some further discussion points in this section.

It appears that rule two was not as effective in reducing the domains of variables. Perhaps this is due to the effectiveness of rule one, since rule one appears a lot and updates have to be propagated around the board anyway. It may also be the order in which we applied our rules since it was applied after rule one.

The naked doubles and triple rules were effective, eliminating backtracking entirely for most problems. This means in real life, the user did not have to make an educated guess for a slot, which is the assignment step of backtracking search. Another possibility is that the user made all successful guesses when he had to; the educated guesses were successful thanks to the most constrained heuristic.

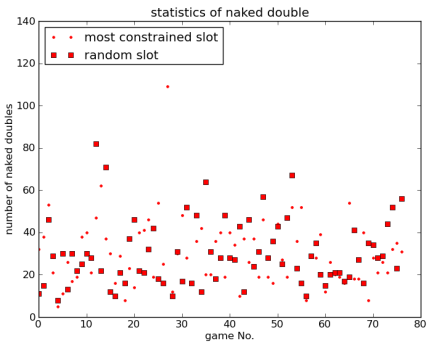
This makes sense because a SuDoKu problem should be solvable with just constraint propagation and a set of rules, no matter how fancy they are, even if the rules are more sophisticated than the naked triples. However, a puzzle creator can easily amuse (or annoy) a solver by forcing him to do backtracking search, which translates to "guessing" squares. Backtracking search forces the user to make guess assignments in real life, and if they make an assignment that contradicts, they have to "erase" their progress and try again by backtracking. We anticipated the use of backtracking search in harder problems, which was correct.

Thefore, we evaluate the following conjecture: "easy problems may be solved by only using the first rule, medium problems may be solved by using the first two rules, and hard and evil problems require the naked triples rule and possibly backtracking." It seems that the conjecture is almost true. Almost all of the easy problems were solvable by the first rule (21 out of 23).

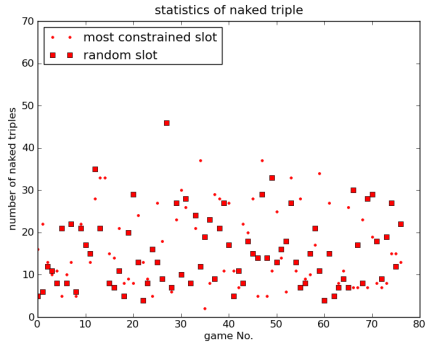


(a) Rule 1

(b) Rule 2



(c) Naked double



(d) Naked triple

Figure 2: Comparisons between most constrained slot heuristic and random slot heuristic in number of rules taken.

Once naked doubles and triples were added, all easy problems were solved, most medium problems were now solved (19 out of 21) and almost all hard and evil problems were solved. It appears that rule two did nothing to help medium problems or any problems at all. However, adding backtracking to any problem solves the problem. These trends are observable in table 1.

Finally, each puzzle was solvable on the order of a second, even with the random slot heuristic. This heavily contrasts the Towers of Corvallis puzzle that used just search, which solved on the order of minutes to hours. While we would need to perform experiments to formalize this heavy contrast, intuitively it shows that constraint propagation takes advantage of the factored representation of each state to outperform general search.