CS531 Programming Assignment 3: SuDoKu

Santosh Kumar, Daniel Magee
(EECS, MIME), Oregon State University

February 19, 2018

Abstract

In this project, we apply a constraint satisfaction problem framework to the popular sudoku puzzle game. In solving the problem, we explore the effect of progressively broader constraint satisfaction strategies on the time cost and necessity of more expensive backtracking search techniques. From our experiments we found that double and triple rules are effective to solve most problems at hard and evil difficulties without backtracking. But, we also found that backtracking was not the most costly operation and that using more complicated inference rules avoided backtracking at the expense of time cost.

1 Introduction

The most common incarnation of the sudoku puzzle is found in newspapers across the world, and consists of a nine by nine grid with several cells initially populated by single digits. Our constraint satisfaction formulation of the sudoku puzzle defines the variables to be the cells in the grid and the domain to be the digits $1 - -9$. A solution to the puzzle is an arrangement of single digit, positive integers in individual cells that satisfy the rules of the game, the constraints. The puzzle's constraints are all *all-diff* type constraints, that is, they require a set of k variables with k domain values to be assigned distinct values. In this case, the $9x9$ grid containing 81 values is partitioned into 9 $3x3$ squares and 9 rows and columns each of which is subject to the *alldiff* constraint. Sudoku problems are generally classified as easy, medium, hard, or evil depending on the number of backtracking moves and the number and complexity of the policies required to arrive at a solution.

2 Algorithm

Our sudoku solver consists of a backtracking search algorithm 1 containing contraint propagation, inference rules, and a next cell choice heuristic. The

backtrack function takes a list of variables, in this case the state of the sudoku grid, a heuristic function, and an inference level as inputs. The state of the sudoku grid is represented as a hash table with the coordinates of the cell in a string as keys, e.g. ‘1a‘ for the top left corner, and the remaining candidate domain values in a string as values. A cell is ‘assigned‘ when there is only one value in it’s domain.

Algorithm 1 Backtracking Search

```
function BACKTRACK(variables, HEURISTIC, inferenceLevel):
if variables is a solution then
  return variables
end if
cell = HEURISTIC(variables)
for value in domain of cell if value is consistent do
  copy variables to variablesCopy
  set cell to value in variablesCopy
  variablesCopy = CONSTRAINT-PROPAGATION(variablesCopy, inferenceLevel)
  if variablesCopy is not failure then
    result = BACKTRACK(variablesCopy, HEURISTIC, inferenceLevel)
    if result is not failure then
      return result
    end if
  end if
end for
increment backtrack count
return Failure
```

Backtracking search is a variant of the depth-first search algorithm. It explores the solution tree like a depth-first search algorithm, but keeps a record of the search path in order to restart the search from the parent node if the branch ends in failure. It begins by checking whether the variables are a solution to the puzzle. Then it elects a cell using the provided heuristic to assign a consistent value. It makes a copy of the variable table and updates that copy with the constraint propagation procedure described in Algorithm 2. When the contraint propagation and inference rules are unable to advance the solution further, the reduced variable table or a failure message is returned. Failure messages occur whenever a variable has an empty domain. If the contraint propagation finds a contradiction and returns failure, we try assigning a different consistent value to the last variable to be heuristically selected. If no values remain in the domain of that variable, we complete a backtrack, increment the counter and return faiulre, meaning that variable cannot produce a solution by assigning a value. If the contraint propagation is successful

and returns an updated sudoku grid with smaller consistent domains for the variables, then BACKTRACK is called with the updated variables. It selects a new cell to assign a consistent value and propagates the contraints.

Algorithm 2 Constraint Propagation

```
function CONSTRAINT-PROPAGATION(variables, inferenceLevel):
while the domain of any variable was changed do
  if any variable's domain is empty then
    return failure
  end if
  for cell in variables do
    if size of cell.domain is 1 then
      for each neighbor of this cell do
        remove cell.value from neighbor.domain
        if size of neighbor.domain is 1 then
          increment inference rule 1 count
        end if
      end for
    end if
  end for
  for i = 1 to inferenceLevel do
    apply inference rules i
  end for
end while
return variables
```

The constraint propagation function shown in Algorithm 2 continues as long as the procedure has made some change in the consistent domain of any variable. The *inferenceLevel* parameter determines how deep the algorithm should search for values to eliminate; we consider three inference levels, single, double, triple, which contain two rules each, naked and hidden. The constraints are propagated by traversing all the cells in the grid, if they are assigned a variable (if there is only one value in their domain), we visit all the cells that are constrained to contain a different variable and remove the value from their domain if it is there.

Since our method of assigning values to variables is simply leaving a single value in the domain of a cell, there is no way to avoid using the first inference rule, naked singles. The naked singles rule assigns a value to a variable if it is the only candidate left in the domain. Propagating the constraints entails checking whether any cell has only one value available to it; checking the length of the domain is effectively the same as assigning the value to the variable if it is the only value left. Since each new assignment detected during constraint

propagation, a neighbor‘s domain is reduced to a size of 1, counts as a naked single inference assignment, we cannot differentiate these two actions and cannot launch the program without any inference abilities at all as requested in the problem statement.

There are two varieties of inference rules, naked and hidden. Naked rules apply to k cells with identical sets of k candidates in the same constraint group (box, column, or row). Finding a naked set allows the values in that set to be removed from all other cells in the constraint group. Hidden rules involve k cells with identical k length subsets within the domains of those cells. Finding a hidden set allows the all the values outside the set in the cells with the hidden subset to be removed. We investigate applying these rules up to $k = 3$. In our results section, we will show the performance of the constraint propagation step when toggling a subset of these rules on or off.

In this program, we created two different heuristics to choose a variable to assign a single value in the BACKTRACKING function.

1. **Fixed selector** - Pick the first square with more than one value in its domain, traversing the grid from top to bottom, left to right.

2. **Most Constrained Variable** - Pick the variable with fewest, but more than one, value in its domain.

While we would expect the most constrained variable heuristic to provide a more efficient path to the solution, it also involves a more expensive operation, traversing and storing the length of the domain for each variable in the grid. This could diminish performance benefits gained from taking a more direct route.

3 Results

We varied the solver parameters: the heuristic and inference rules, and solved all 77 puzzles for all solver configurations. We use four inference levels; at level zero only naked singles are permitted because they are inextricable from the constraint propagation step. At level one we also permit hidden singles, where a cell with several candidates including one that is not found in any other cells in any of its constraint groups, may assign that value to itself. At level two doubles of both types are also permitted, and at level three triples are legal. We were able to find solutions to all puzzles regardless of the heuristic or permitted inference rules because backtracking search is an exhaustive search, it continues until a solution is found if one is available.

Figure 1 shows the average number of backtracks performed by each problem difficulty and solver configuration, and illustrates the effect of the heuristic and high order inference methods on the necessity of backtracking. It

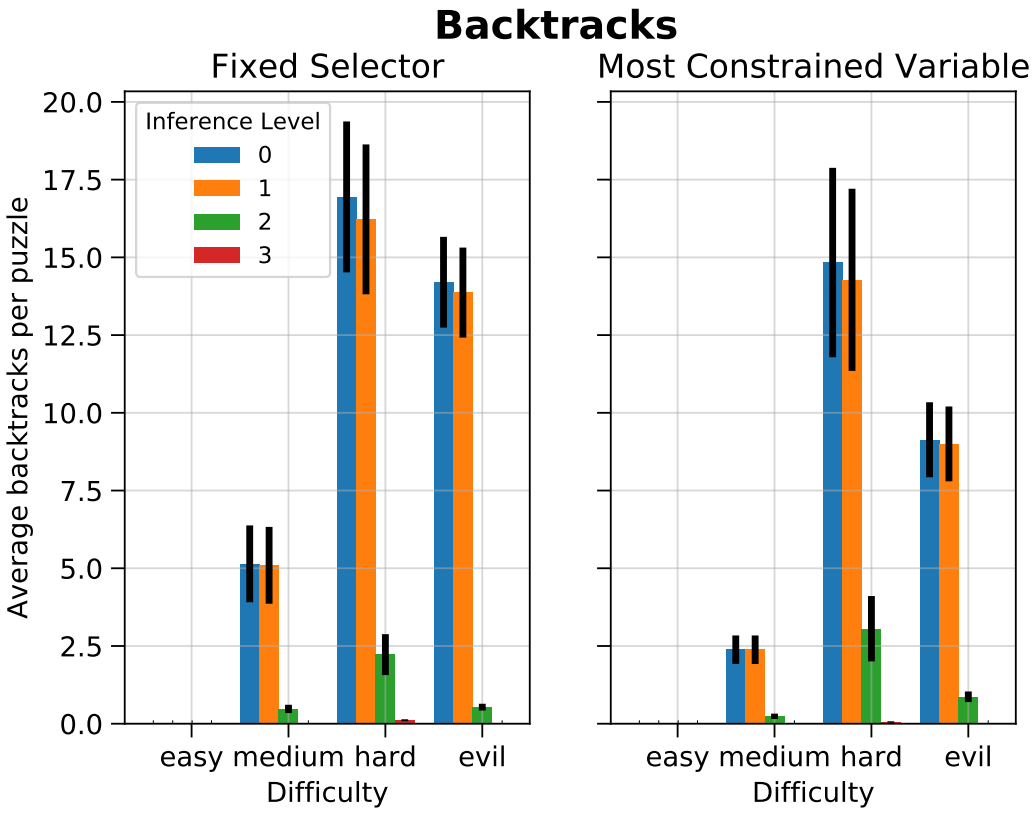

Figure 1: Average number of backtracks per puzzle and solver type.

also shows that easy problems require only naked singles without backtracking to solve the problem. In these problems, propagating constraints always leads to a variable with a single value in its domain. Using doubles dramatically reduces the required backtracking steps for both heuristics on problems harder than easy. Using triple inference rules almost eliminates the need for backtracking entirely. Only two problems, categorized hard require a single backtrack to be solved, and one of these problems requires a backtrack with the most constrained variable heuristic as well.

Reduced backtracking does not lead to increased performance. Figure 2 shows the runtime for each solver; a particularly illustrative comparison is between the average backtracks and runtime result for hard problems using the most constrained variable heuristic allowing double but not triple inferences. Figure 1 shows a roughly 5× reduction in backtracks by adding double inferences, yet the average runtime increases by about a third. This is because inferences are about as costly as backtracks and must occur many times more often.

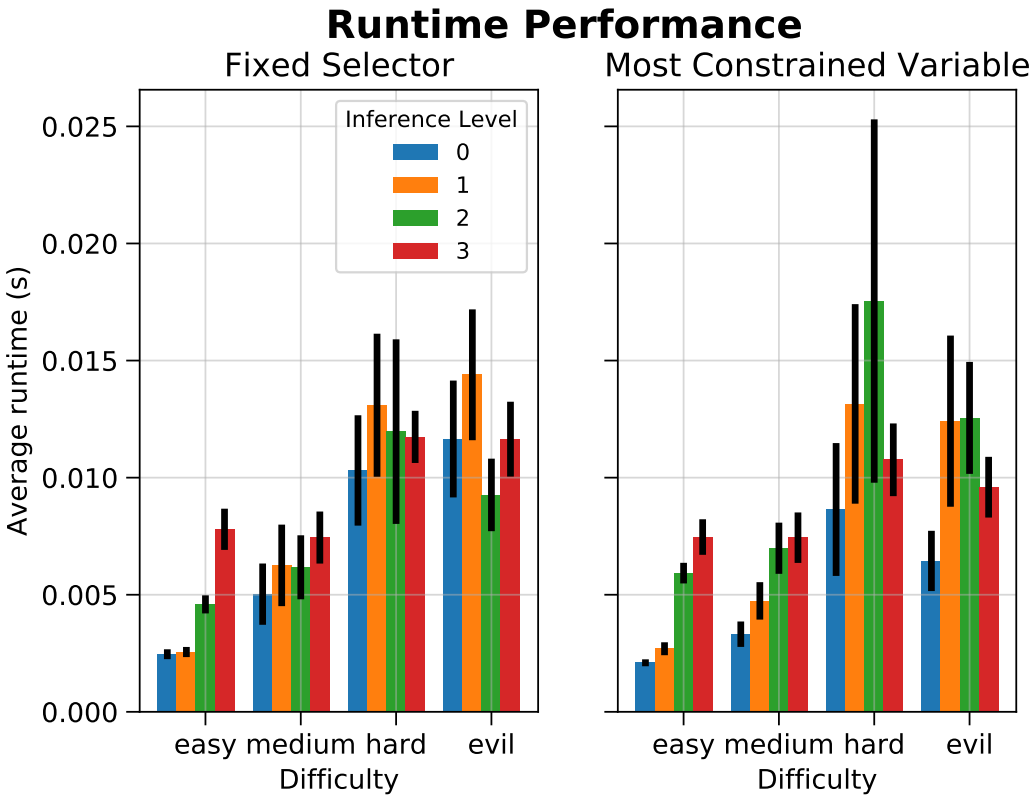

Figure 2: Time cost of solution per puzzle and solver type.

4 Discussion

Using the complexity of inference rules needed to solve a puzzle without backtracking is a good way to gauge the difficulty of a puzzle so the conjecture is roughly correct. Although, we do not observe that backtracking is an overwhelming drag on computational resources, the problems that require the most backtracking do tend to cost the most as seen in Figure 2. That is, increasing inference complexity to reduce backtracking does not effectively reduce runtime cost, but the categories that continue to require backtracking after further inferences are applied tend to take the most time.

Table 1 shows the average number of boxes initially assigned with a single value, which is a proposed measure of difficulty. This would seem to accurately reflect the difficulty of most problems, but miss out on important edge cases of difficulty. Metaphorically, this is like saying that you can judge the time it takes to drive from city to city by the road mileage between them. The 120 mi from Cheyenne to Denver will probably take the predicted 2 hours, but that same distance from Philadelphia to New York will rarely pass so quickly.

The naked doubles and triple rules were effective, eliminating backtracking

Table 1: Average number of initially assigned boxes

	Easy	Medium	Hard	Evil
Avg number	34.7	29.0	26.3	26.1

entirely for most problems. This means that in reality a user would not have to make an educated guess for a slot: the assignment step of backtracking search. This makes sense because a sudoku problem should be solvable with just constraint propagation and a set of rules even if the rules are more sophisticated than the naked triples. However, a puzzle creator can easily amuse (or annoy) a solver by forcing them to perform a backtracking search, which translates to guessing squares and values and saving those guesses to change them later if they fail.

All easy problems were solved without backtracking or inference beyond naked singles. Once double inference rules were added, most medium problems (19 out of 21) could be solved without backtracking. Using hidden singles does seem to mildly diminish backtracking, but also increases time cost.