Solving Sudoku Puzzles using Backtracking Algorithms

Jonathan Christopher / 13515001
Program Studi Teknik Informatika
Institut Teknologi Bandung
Bandung, Indonesia
13515001@students.itb.ac.id

Abstract—Sudoku is a popular puzzle consisting of a 9x9 grid of squares which must be filled using the digits 1-9 according to several constraints. This paper investigates two backtracking approaches for finding solutions for Sudoku puzzles: naïve backtracking and backtracking with constraint propagation; as well as other techniques that may be used to optimize solving algorithms.

Keywords—Sudoku; backtracking; constraint satisfaction; constraint propagation

I. INTRODUCTION

A Sudoku puzzle is a type of puzzle in the form of square grid. The grid consists of 81 square cells, arranged in 9 rows and 9 columns. The grid itself is further divided into nine non-overlapping 3x3 subgrids. Initially, only some of the cells in the grid would be filled. To solve the puzzle, one must fill in the remaining cells with the digits 1 to 9 such that the digits must be unique in each row, column or subgrid.

Sudoku puzzles were likely invented in the 1970s by Howard Garns. It was popularized in Japan by a magazine in 1984. It was during this time it got its name, which in Japanese roughly means ‘single numbers’. Sudoku started to become popular in other countries when a man named Wayne Gould created a computer program that could generate Sudoku puzzles, and proposed for it to be published by daily newspapers [1].

Solving a Sudoku puzzle is a constraint satisfaction problem, as one seeks a solution which digits fulfills the column, row and subgrid uniqueness constraints. Many types of algorithms can be used to solve this kind of problem, ranging from simply brute-forcing the digits up to non-deterministic algorithms like genetic programming. However, each solver algorithm differs in their time and space complexities.

The choice of algorithm and optimizations used is important in creating a Sudoku solver. Although a 9x9 grid might not seem big, the permutation of digits required to fill it increases exponentially. A simplistic brute-force solver that generates all possible solutions would not be able to solve Sudoku puzzles in a reasonable length of time, as there are approximately 6,670,903,752,021,072,936,960 possible valid Sudoku grids [1]. Smarter algorithms that needs less time and memory space are needed.

Moreover, solving Sudoku puzzles has been shown to be an NP-complete problem by Takayuki Yato and Takahiro Seta from University of Tokyo [1]. This means that currently, there are no known polynomial-time deterministic algorithm for solving Sudoku puzzles. It also means that if a polynomial time solution for Sudoku is discovered, the polynomial time algorithm can also be applied to solve other NP-complete problems, many which have more important usage in the real world.

This paper focuses on using backtracking algorithms to solve Sudoku puzzles. Backtracking is chosen because with the right optimizations, it can produce solutions in reasonable time and memory space. Two approaches are be considered: naïve backtracking, which fills in each cell one by one, and backtracking with constraint propagation, which executes non-branching operations in between node visits to fill in cells which values are already determined by other cells.

Fig. 1. A Sudoku puzzle (hard difficulty sample puzzle used in the author’s experiment, from the New York Times 18 May 2017 daily puzzle: https://www.nytimes.com/crosswords/game/sudoku/hard).
by R.J. Walker, Golomb, and Baumert. In backtracking, a
solution is constructed in a depth-first pattern from a sequence
of decisions. If a decision leads to a dead end, from which no
valid solution can be constructed, the current sequence of
decisions is partially unwound, up to a point from which a
different decision can be tried [2].

The algorithm starts from a root node, which is typically an
empty solution. This node is then expanded to reveal other nodes
(partial solutions) which are only a step/decision away from the
current node. A newly expanded node is then selected as the next
current node. The new current node is then checked to determine
whether the partial solution it represents can be part of a valid
full solution. If not, the current node is pruned or cut off.
Otherwise, it is then recursively expanded, just like the previous
current node.

Using backtracking for search is an improvement over using
brute-force techniques, such as exhaustive search. Brute-force
approaches need to consider all solutions. The number of
possible solutions is usually very large, as most solutions are
permutations or combinations of some form, which increases
exponentially. In backtracking, a large fraction of these possible
solutions can be pruned at every step, when it became known
that a partial solution won’t be part of a valid full solution. Pruning helps to quickly reduce the number of solutions which
must be generated and checked.

There are several general properties of backtracking
approaches [2]:

1. Problem solution

A problem solution is represented by a n-tuple of decisions
for each step \( X = (x_1, x_2, ..., x_k) \), \( x_i \in S_i \). Each \( x_i \) represents a
decision taken when constructing the solution, from all possible
different decisions \( S_i \). The possible decisions might be different
for each step taken, but it could also be the same.

2. Generating function

A generating function \( T(k) \) generates all possible decisions
that can be taken at step \( k \). One of the values generated is then
assigned to \( x_k \), which is a decision made to construct the current
partial solution.

3. Bounding function

The bounding function \( B(x_1, x_2, ..., x_k) \) checks whether a
partial solution leads to a valid full solution. If yes, it returns true
and the algorithm proceeds to generate more decisions that can
be made from this step; otherwise, this partial solution is pruned
and discarded.

The set of all complete solutions for a problem is called the
solution space. It is represented as a n-tuple of decisions which
has exactly the number of decisions which is required to lead to
a full solution.

It can also be organized as a state space tree. A tree is a graph
which does not contain cycles. Every vertex/node of the tree
represents a state or step in constructing the solution to a
problem, while each edge represents a decision made at every
step/state which transforms that state into a different state in the
next step. A path starting from the root of the tree down to a leaf
vertex represents a full solution; the set of all such paths is the
solution space.

A valid solution to the problem is searched by finding a path
through the state space tree, starting from the root and ending at
a valid leaf vertex. A solution is constructed by traversing the
tree in a depth-first order. In traversing the tree, the vertices that
are currently being visited are called live nodes. Vertices that are
expanded from the live node are called expand-nodes. Every
time a vertex is expanded through an edge, the path from the root
vertex to itself becomes longer. A vertex is only expanded if
when checked by the bounding function, it is still found to lead
to a valid full solution. Otherwise, it is ‘killed’ or pruned at
becomes a dead node, which will not be expanded again

When a vertex has been fully expanded and can’t be
expanded any more, but a full solution has not been found, the
search will backtrack and return to the parent of the vertex. The
search will end when either all vertices have been visited, or a
valid full solution has been found. A valid full solution is called
a goal node in the state space tree [2].

A backtracking algorithm is commonly implemented
recursively, as tree traversal is inherently recursive in nature. Its
basis is when the search reaches a leaf node, on which it checks
whether the full solution that has been constructed is a valid
solution or note. Its recurrence is to enumerate all decisions (tree
edges) that can be taken from the current state (tree vertex/node),
then to visit them if the bounding function does not eliminate
any them.

As backtracking is usually recursive in nature, then its worst
case time and space complexity can be calculated using
complexity theorems related to recursive functions. Its time
complexity is exponential or factorial and depends on the time
taken to compute each vertex and the average number of edges

Fig. 2. Depth-first traversal/search of a state space tree.
(source: http://www.w3.org/2011/Talks/01-steven-phenotype)
leading away from each vertex. Its worst-case time is thus not very different from those of brute-force algorithms. However, a good bounding function will greatly reduce its average-case time and space complexity by increasing the amount of vertices pruned.

Backtracking algorithms are commonly used to solve NP-hard problems. NP-hard problems currently have no known polynomial-time solution. Backtracking algorithms are not polynomial-time solutions since their worst-case complexity are exponential; but in most cases, with a good choice of bounding function, backtracking will provide an exact solution to most cases of those problems in reasonable time.

III. SOLVING SUDOKU PUZZLES USING NAIVE BACKTRACKING

A Sudoku puzzle is an ideal candidate for backtracking because of several factors. First, it is too hard to simply brute force, as the size of its solution space is prohibitively large. Second, it solution can be constructed from a sequence of decisions: which digits to pick for each remaining empty cell. To solve Sudoku puzzles using naive backtracking, the solution structure, generating function, and bounding function must first be defined.

The solution structure for solving Sudoku puzzles is a n-tuple \( X = (x_{11}, x_{12}, ..., x_{99}) \), \( x_{ij} \in \{0, 1, 2, ..., 9\} \). Each cell value \( x_{ij} \) represents a digit in a cell on the \( i^{th} \) row and \( j^{th} \) column of the grid. At each state, each cell can contain the digits 1 to 9 or the value 0, which represents an empty cell. Initially, all cells except are marked empty, except the cells which digits are known from the problem. Partial solutions are constructed as the algorithm runs. A partial solution still contains empty cells. A full or completed solution has no more empty cells remaining.

The generating function \( T(X) \) for a Sudoku puzzle receives a partially filled Sudoku grid and outputs all possible digits that can be filled for each empty cell. It is implemented by first creating a n-tuple \( A = (a_{11}, a_{12}, ..., a_{99}) \), \( 1 \leq i \leq 9, 1 \leq j \leq 9 \), where each \( a_{ij} \) is a set initially containing the digits 1 to 9 for empty cells, or an empty set for non-empty cells. Each set \( a_{ij} \) represents the digits which can be filled to the corresponding cell if it is not yet filled. Then for each non-empty cell \( x_{ij} \) in \( X \), the digit contained in the current cell is deleted from all \( a \) in the row \( i \), column \( j \), and the same subgrid as the current cell. After this process ends, only digits that can be filled legally according to Sudoku constraints will remain in the set for each non-empty cell.

For solving Sudoku puzzles, the bounding function used is simply a function checking the constraints that must apply for a valid Sudoku grid. It receives a partially filled Sudoku grid and returns true or false depending on whether the given input grid fulfills all of the Sudoku constraints. These constraints alone reduces a lot of invalid solutions, without having to resort to other heuristics. It is implemented by simply looking through each non-empty cell and checking whether its value actually occurs in any of the other cells in the same row, column or subgrid. If any duplicate digit is found in the same row, column, or subgrid, the function halts and returns false. Otherwise, the function continues to check other cells. If it successfully checked all non-empty cells, it then returns true.

A recursive implementation of a naïve backtracking solver typically consists of the implementations of the generating and bounding functions, and a recursive solve(grid) function. The solver function is called with the problem grid as a parameter, given as a two-dimensional array. It first checks whether the given Sudoku grid is valid, using the bounding function. If it is, it checks whether the grid has no more non-empty cells, indicating that the Sudoku puzzle is solved.

If the Sudoku grid is not completely solved yet, the solver function proceeds to generate the sets of digits that are available to pick for every non-empty cell in the grid, using the generating function. It then picks a non-empty cell, then expands the state of the current grid by copying the current grid, then filling in the corresponding cell in the copied grid using a digit which is in the set of available digits for that cell. The copied grid is then passed recursively to the solver function.
As the partially filled grids are passed deeper in the recursion tree, it must reach a state in which the grid is invalid or fully filled in, as there are only 81 cells in the grid, and at each recursion step a cell is filled. If a valid solution is found, the solver function will return the fully solved Sudoku grid, which will in turn cascade up through the recursion tree. If an invalid grid or partial solution is found, the solver function will return false, thus prompting the solver to backtrack as the current instance of the function is popped from the top of the call stack. The solver will attempt to try another digit from the available digits set. The solver will also backtrack when the available digits set is exhausted, as every non-empty cell must have a digit. If there are no solution for the given Sudoku grid input, the solver will continuously backtrack until the topmost instance of the solver function (which is called by the main program) also returns false.

IV. SOLVING SUDOKU PUZZLES USING BACKTRACKING WITH CONSTRAINT PROPAGATION

Using the naïve backtracking approach is often sufficient for solving most Sudoku puzzles. However, there are several harder problem instances which may require more time and memory space when solved using backtracking. To be able to solve even more puzzles in reasonable time, one can use a better approach to backtracking.

The approach of backtracking with constraint propagation is inspired by the way humans try to solve intermediate to hard Sudoku puzzles. A human doing a Sudoku puzzle will not attempt to naively ‘guess’ digits like the naïve backtracking solver algorithm. Humans have very limited short-term memory relative to modern computers, and thus they would try to avoid having to backtrack as much as possible since backtracking is cumbersome to do on paper.

Typically, humans would first try to fill in ‘fixed’ or non-ambiguous cells first. ‘Fixed’ cells are empty cells which value can be directly inferred from the values of non-empty cells in nearby rows, columns, or subgrids. They have exactly one valid digit in the available digits set, so that filling them in does not require branching or backtracking. As filling in a cell can provide more clues to the values of neighboring cells, filling in one cell can lead to a cascade of cells being filled.

The action of filling ‘fixed’ cells without branching is called constraint propagation. Constraint propagation causes cell values which are directly dependent on the current values of neighboring cells to be updated all at once. Constraint propagation can be integrated with backtracking before generating expansions from the current state. This ensures that ‘fixed’ cells in the current state have been filled, reducing the number of empty cells and increasing the number of cells filled at each step of the recursion.

Unlike the expansion done when constructing solutions, constraint propagation does not need much memory space. Constraint propagation also reduces the number of nodes/states visited by the backtracking solver algorithm, since every step of the algorithm can now fill in more than one cell in the grid. It has the effect of reducing average memory space used, and also the time taken to find a valid solution.

Constraint propagation is implemented in a procedure named reduce(grid). This procedure takes a Sudoku grid and applies constraint propagation on it. First, it finds the set of available digits for each non-empty cell, similar to the available digits sets calculated by the generating function. Next, it checks whether any of the cells has only one possible digit to fill in. If a cell has only one possible digit, the cell is filled in using that digit. After that, the set of available digits is recalculated for all non-empty cells. This process is repeated until there are no more empty cells with only one possible digit, as at this point, there are no more ‘fixed’ cells – meaning that to fill in more empty cells, we must guess between at least two values and branch.

V. COMPARING AND OPTIMIZING BACKTRACKING APPROACHES

To compare between both the naïve and constraint-propagating backtracking approaches, the author ran a small experiment with three sample Sudoku puzzles with varying difficulties, taken from the New York Times’ 16 May 2017 daily puzzles\(^1\). Each puzzle is run using both approaches. For each execution, the execution time and the nodes visited by the backtracking algorithms are recorded.

<table>
<thead>
<tr>
<th>Sudoku Puzzle / Approach</th>
<th>Execution Time (s)</th>
<th>Nodes visited</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easy (naïve backtracking)</td>
<td>0.0310001373291 s</td>
<td>45</td>
</tr>
<tr>
<td>Easy (with constant propagation)</td>
<td>0.00099997270275 s</td>
<td>1</td>
</tr>
<tr>
<td>Medium (naïve backtracking)</td>
<td>0.150000095367 s</td>
<td>299</td>
</tr>
<tr>
<td>Medium (with constant propagation)</td>
<td>0.029000043869 s</td>
<td>59</td>
</tr>
<tr>
<td>Hard (naïve backtracking)</td>
<td>0.878999948502 s</td>
<td>1702</td>
</tr>
<tr>
<td>Hard (with constant propagation)</td>
<td>0.229000091553 s</td>
<td>430</td>
</tr>
</tbody>
</table>

Fig. 4. Execution times and visited nodes count of the author’s Sudoku solver implementation

The experiment results shows that backtracking with constant propagation consistently executes faster than naïve backtracking, especially for easy puzzles. Even for hard puzzles, constraint propagation improves the execution times, making it around 4 times as fast as the naïve backtracking algorithm’s execution times for the corresponding puzzles.

Besides a reduction in execution times, the number of nodes visited during the search is also reduced when using constraint propagation. This is as predicted, as constraint propagation reduces the number of empty cells which must be searched. As recursive search and backtracking imposes a greater overhead than a simple loop, reducing the amount of recursion branching in the search algorithm can significantly improve the performance of the algorithm.

<table>
<thead>
<tr>
<th>Sudoku Puzzle / Approach</th>
<th>Execution Time (s)</th>
<th>Nodes visited</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easy (naïve backtracking)</td>
<td>0.0299999713898</td>
<td>44</td>
</tr>
<tr>
<td>Easy (with constant propagation)</td>
<td>0.000999927520752</td>
<td>1</td>
</tr>
<tr>
<td>Medium (naïve backtracking)</td>
<td>0.204999923706</td>
<td>412</td>
</tr>
<tr>
<td>Medium (with constant propagation)</td>
<td>0.0639998912811</td>
<td>126</td>
</tr>
<tr>
<td>Hard (naïve backtracking)</td>
<td>0.411000013351</td>
<td>772</td>
</tr>
<tr>
<td>Hard (with constant propagation)</td>
<td>0.108000040054</td>
<td>200</td>
</tr>
</tbody>
</table>

Fig. 5. Execution times and visited nodes count of the author’s Sudoku solver implementation, with minimum available digits-first variable ordering enabled.

Another heuristic investigated by the author is minimum remaining values variable ordering. Using this heuristic, when selecting the empty cell to fill after generating the available digits, a cell with the smallest number of possible digits is selected [2]. The purpose of this heuristic is to maximize the probability of picking the correct digit for the cell when pruning branches. For example, when picking a cell with five possible digits to fill in, each digit has a probability of only 0.2 of being the correct digit for the cell; but when picking a cell with only two possible digits, each digit has a probability of 0.5 of being the correct digit.

The minimum remaining value variable ordering heuristic does sometimes help in reducing the number of nodes visited and thus the execution time. However, in some cases, such as the medium-difficulty sample puzzle, it has the opposite effect of slightly increasing the number of nodes visited and the execution time. More experiments are needed on a representative set of sample problems before the usefulness of this particular heuristic can be concluded.

VI. CONCLUSION

Backtracking is a suitable approach for solving Sudoku puzzles, due to the relatively large size of its state space. Backtracking can solve most Sudoku puzzles in reasonable time by pruning large parts of the state space tree that does not contain the solution as it searches it.

From the author’s experiment, it can be concluded that constraint propagation improves the performance of backtracking algorithms for solving Sudoku puzzles when compared to naïve backtracking. The minimum remaining values variable ordering heuristic can be helpful for speeding up execution in some cases; however, a more conclusive experiment is needed to ensure that the heuristic is really applicable for the case of solving Sudoku puzzles.
ACKNOWLEDGMENT

The author thanks Dr. Masayu Leyla Khodra, S.T, M.T. and Dr. Ir. Rinaldi Munir, M.T. as the lecturer of the author’s Algorithm Strategy class, and also the author’s classmates, for their guidance in preparing this paper.

REFERENCES