CS627 - IP5

Kirsten Reid

6/19/2019

[Background 1](#_Toc11858474)

[Analysis 1](#_Toc11858475)

[State Space 2](#_Toc11858476)

[Traversal Time Complexity 4](#_Toc11858477)

[Heuristic Search 5](#_Toc11858478)

[Implementation 6](#_Toc11858479)

[Pseudocode 6](#_Toc11858480)

[Fundamentals: 6](#_Toc11858481)

[Adding Backtracking (from ): 6](#_Toc11858482)

[Final Design: 7](#_Toc11858483)

[Portfolio 7](#_Toc11858484)

[References 8](#_Toc11858485)

# 

# Background

Algorithms are the foundations of computer science; they operate at the heart of searching, sorting, analyzing, optimizing, computing, and any other processing function that most applications and programs perform. There can be one, many, or no correct way to solve or implement an algorithm, and the ‘best’ choice for an application usually depends on the space and time complexity tradeoffs.

Algorithms can range from simple to astronomically complex (an emerging science is the development of quantum algorithms for quantum physics computers using qubits); however, the concept of improving on known algorithms is in and of itself a cornerstone of algorithmic development. It becomes clear when one approaches problems that increase exponentially in time as the input grows the need to incorporate these improvements for a viable solution.

There are many methods to finding these improvements, but the following sections will show how an ‘artificial’ intelligence can solve a sudoku puzzle, using the concepts of **state-space trees** and **heuristic searching**. At a glance, state-space trees are data structures that house all the possible states at each node, and its variances beneath it. In simplest terms, if you were to find all the combinations of outfits you could wear, each state would be a set of clothing, and each node would branch off based on a state condition (like shirt = yellow, and all the children would be outfit “states” where the shirt is yellow). The leaves of the state tree are complete states that can be returned for a solution, or stopped due to a violation (you can’t wear a yellow shirt with purple pants, so don’t even bother looking at your shoe collection!) Heuristic searching is an optimization technique that incorporates strategic bounds to an algorithm so that it can decide which state space branch to follow (or return to).

# Analysis

The game of Sudoku is a number puzzle where the player must fill in a grid (usually 9x9) with the numbers 1-9, meeting the conditions that:

1. No two numbers can be in the same **row**
2. No two numbers can be in the same **column**
3. No two numbers can be in the same 3x3 square **subgrid**

Even when humans are going through the solving process, a guess and check method is usually employed (unless it’s an “easy” puzzle, with the majority of the spaces filled out and it can be solved using deduction). **Backtracking** is the concept that allows for an algorithm to traverse state space trees, forward and reverse, and discern which branches are feasible to get to a final solution.

In order to develop an ‘intelligent’ process for solving a sudoku puzzle, it is imperative to first understand the states and the heurism that will be utilized in the process.

## State Space

As the name implies, a state space is a state of a given dataset in space. States are defined by the algorithm as potential states that the solution can be in, as it progresses through the input variations. State space trees are a data structure where each node on the state space tree diagram is a potential variation, and the children of each node are variants based on that path. Leaves are either complete solutions or partial solutions that cannot continue (either based on lack of inputs, or conditions to continue).

Said in a completely succinct way, “State space is the set of all paths from root node to other nodes,” (Bhatia, 2013).

A state space diagram of a sudoku puzzle would therefore be shown as (in a very simple 2x2 case):

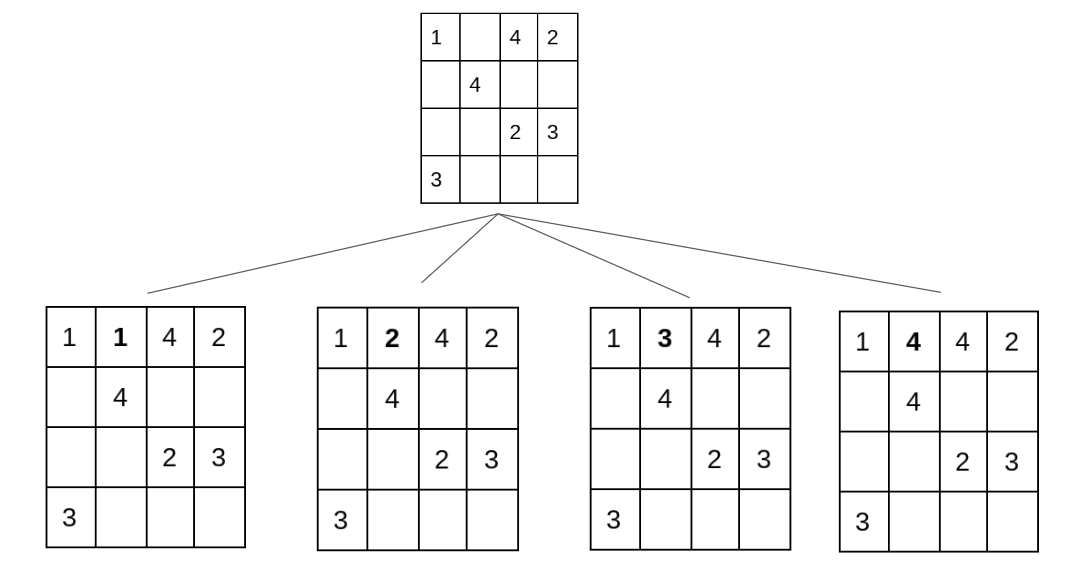
1. Possible values: 1, 2, 3, 4
2. Constraints: all rows, columns, and 2x2 subgrids must be unique (1-4)

Our first state, let’s say, is:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 |  | 4 | 2 |
|  | 4 |  |  |
|  |  | 2 | 3 |
| 3 |  |  |  |

From here, we would need to fill in 1, 2, 3, and 4 into each space (starting at the first **open** space in the first row, moving left to right, top to bottom).

The first state space nodes are:



We can visually see (or rule out with programmed constraints) that the first, second, and fourth violate the conditions, and their state spaces would not be part of a solution.

In terms of backtracking, the state space tree would keep open all available nodes that haven’t yet violated a constraint, so that it can go back and try the next path to find that solution. This saves time when seeing what values will work or not, and allow the program to start with a guess-and-check approach, filling in each empty value and “undoing” choices to backtrack to the next path when it runs into a dead end.

For example, it would only take a few more steps to reach a dead end here (blue choices are ones that have been made in previous states, green is the backtracking node, and red is where the leaf terminated):

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | **3** | 4 | 2 |
| **2** | 4 | **1** | **X** |
|  |  | 2 | 3 |
| 3 |  |  |  |

In this case, you can see that even though the 3rd empty cell could have **1** as a value, without violating any constraints, it forced the next cell to be invalid for **any** input. Therefore, we must backtrack to the state space:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | **3** | 4 | 2 |
| **2** | 4 |  |  |
|  |  | 2 | 3 |
| 3 |  |  |  |

And proceed to the next [valid] node:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | **3** | 4 | 2 |
| **2** | 4 | **3** |  |
|  |  | 2 | 3 |
| 3 |  |  |  |

In summary, a vertex in the state space traversal tree would represent a node, whereas an edge would be a path. Each vertex in the state space tree is a singular state of the sudoku puzzle, in some pattern. The terminology can also apply to graphs, as opposed to trees.

Back-tracking employs a **depth-first** method of searching, meaning it traverses a set of vertices (node states) down each path before moving onto the next vertex along the breadth.

In computer science representation, the state in the puzzle from the last example (with the green **3**) would have the vertex (e for empty)

{1, 3, 4, 2; 2, 4, 3, e; e, e, 2, 3; 3, e, e, e}

## Traversal Time Complexity

The process of exploring each state space vertex for an **n**x**n** Sudoku board with **p** possible input numbers to each square using **brute force** would warrant a time complexity of O(p^(n^2)).

To demonstrate using our previous example, a 4x4 board with 4 possible numbers:

There are 16 spaces, 4 subgrids, 4 rows and 4 columns.

We assume that ‘trying’ a solution, either randomly or via pattern, will be validated per the row, column, subgrid criteria, and if it doesn’t pass, the method will restart completely and try again.

The worst case for brute force would be time complexity of **O(p\*(n^2))**, since we would have to go through EVERY row, and EVERY column, for every value of p.

Thankfully, there are ways of improving the time complexity, using a concept of Heuristic Searching. For reference, the depth-first approach has time complexity **O(V+E)**, where V are vertices and E are edges. (Parnell et all., 2015)

## Heuristic Search

Heuristic searching allows for a simplification of a brute force approach by employing limiting criteria using common sense.

In order to develop these, the types of heuristic approaches that a human would follow could include:

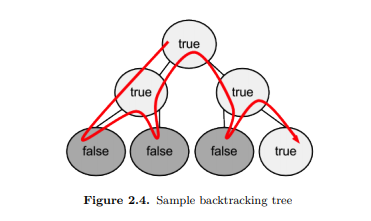
1. Looking at the sub-grid first to ensure that the numbers 1-p are not already used. If 1 were the 9th (bottom right) position in a subgrid, you can eliminate all other state space nodes that use “1” before you traverse all of them, only to end up in error when you do the last check.
2. Another method is to count the number of en-empty (fixed) values in the state. In a 9x9 grid, there will be only nine 1’s in the entire grid. If you count how many are already there, you can eliminate any sub-solution where you would find a difference resulting in more than 9. Or, if there were already four 1’s in the puzzle, eliminate any space state path where you would guess any more than five more 1’s.
3. Finally, it is easiest to start at the sub-grid that has the most constraints than just proceeding left to right, top to bottom. By ordering each subgrid from most to least restrained and solving in that order, a more optimal solving pattern will be discovered. Or, recursion can be used so that as a subgrid is solved, go back to the next-most-complicated grid and repeat the solving (each grid should get sufficiently more restricted as you progress).

# Implementation

To develop a pseudocode, there are some preliminary steps to follow:

1. Start with the brute force algorithm
2. Improve it using depth first state space considerations
3. [Advanced] build in Heuristic search functionality

From B to C require a technique called backtracking, which can be visualized as a series of true-false checks that designate where the pointer will look next. This image from () gives a good visualization:



This figure shows another from Dey, S., 2017: <https://sandipanweb.files.wordpress.com/2017/03/ex_2_btt.jpg?w=676>

## Pseudocode

### Fundamentals:

Given a starting board nxn, with p = n number of choices for a square board

For (0 < k < n) //beginning at the first column

For (0 < j < n) //travelling along the first row

For (1 < i < p) //trying values from 1 to p

### Adding Backtracking (from ):

boolean Solve(choice = some\_previous\_choice):

{

If this choice does not lead to some more other choices

{

If this choice is the desired goal state

return true

else

return false

}

If this choice further has multiple options (choices) to be checked

{

For each further choice

{

check if this choice suceeds by calling Solve(current\_choice)

if Solve(this choice)

return true

}

}

return false

}

### Final Design:

Given a starting board nxn, with p = n number of choices for a square board

If emptyspaces = 0, return true

Else

Go to first empty space in 0<j<n; 0<k<n:

Try:

For i = 1 - p

For j = 1 - n

For k = 1 - n

If [j,k] = valid //theres a value already there

Do nothing

If [j,k] = empty (or 0 or null)

Set [j,k] = i

Function checkvalid =

Test rowviolation; return false

Test column violation; return false

Test gridviolation; return false

ELSE return true

Go to next empty space

## Portfolio

GitHub: <https://github.com/demagoguer/CS627>

Slides: <https://slides.com/demagoguer/cs627-portfolio>

# References

1. Hui, J (2018, Dec 6) *QC - Quantum Algorithm with an example*. Medium. Retrieved from <https://medium.com/@jonathan_hui/qc-quantum-algorithm-with-an-example-cf22c0b1ec31>
2. Spittel, A (2018, Jun 26) *Coming Back to Old Problems: How I Finally Wrote a Sudoku Solving Algorithm*. Retrieved from <https://dev.to/aspittel/how-i-finally-wrote-a-sudoku-solver-177g>
3. Bhatia, S (2013) *Backtracking*. University of Missouri, St Louis. Retrieved from <http://www.cs.umsl.edu/~sanjiv/classes/cs5130/lectures/bt.pdf>
4. BacktobackSWE (2019, Jan 8). *Implement A Sudoku Solver - Sudoku Solving Backtracking Algorithm ("Sudoku Solver" on LeetCode).* Video - YouTube. Retrieved from <https://www.youtube.com/watch?v=JzONv5kaPJM>
5. BacktobackSWE (2018). *SudokuSolver.java*. GitHub. Retrieved from <https://github.com/bephrem1/backtobackswe/blob/master/Dynamic%20Programming%2C%20Recursion%2C%20%26%20Backtracking/sudokuSolver.java>
6. Norvig, P (n.d.) *Solving Every Sudoku Puzzle*. Retrieved from <https://norvig.com/sudoku.html>
7. Sief, G (2017, Dec 26) *Solving Sudoku using a simple search algorithm*. Medium. Retrieved from <https://medium.com/@george.seif94/solving-sudoku-using-a-simple-search-algorithm-3ac44857fee8>
8. Vikstén, H. & Mattson, V. (2013) *Performance and Scalability of Sudoku Solvers.* KTH. Retrieved from <http://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand13/Group1Vahid/report/henrik-viksten.viktor-mattsson-kex.pdf>
9. Parnell, K., Freeze, S., & Smith, P. (2015) *Sudoku*. UNCW. Retrieved from <http://people.uncw.edu/tagliarinig/Courses/380/F2015%20papers%20and%20presentations/F2015%20papers%20and%20presentations/Sudoku/Sudoku.pdf>
10. Dey, S. (2017, Mar 17) *Consistency Checking and then Backtracking with Minimum Remaining Value Heuristic and Forward Checking in Python*. Retrieved from <https://sandipanweb.wordpress.com/2017/03/17/solving-sudoku-as-a-constraint-satisfaction-problem-using-constraint-propagation-with-arc-consistency-checking-and-then-backtracking-with-minimum-remaning-value-heuristic-and-forward-checking/>
11. Lee, Y (2014) *Solving Sudoku by Backtracking*. Retrieved from <https://codemyroad.wordpress.com/2014/05/01/solving-sudoku-by-backtracking/>
12. Munot, B. (2015) *Sudoku Solver Using Backtracking*. Retrieved from <https://www.hackerearth.com/practice/notes/sudoku-solver-using-backtracking-2/>