Jordan Pemberton
CS 325 Algorithms
Portfolio Project

Sudoku

Brief Description & Game Rules

Sudoku is played on a grid of size $n^2 \times n^2$, that is subdivided into n^2 equal "zones", each of size $n \times n$. A set of n^2 distinct "symbols" are used to fill these cells. Initially, some cells in the grid contain a symbol, and these starting cells cannot be altered. A move is made by inserting one of the symbols into an empty cell. The goal is to fill the entire grid with the symbols such that every row, column, and zone contain each symbol exactly once.

A standard game of Sudoku uses a board of size 9 × 9, that is subdivided into 9 equal zones, each size 3 × 3. In a standard Sudoku game, the 9 distinct symbols used are the numbers 1 through 9, and to solve the game, each row, column, and zone must contain each number 1 through 9 exactly once.

Time Complexity

A given potential solution to a Sudoku puzzle can be verified in polynomial time. Given a filled board, we can determine if it is solved by checking each cell on the board to see if it meets the requirements. For each cell, the entry must satisfy 3 conditions:

1. The entry must not be present anywhere else in its row.
2. The entry must not be present anywhere else in its column.
3. The entry must not be present anywhere else in its zone.

These checks can be done in polynomial time, since each check will take at most the time it takes to search the entire row /column /zone.

Given a board of size *N*, where *N* is the total number of cells on the board (not the board or zone size), we can determine if the board is solved in time *N * 3(sqrt(N)),* because *N* is the total number of cells to check, and *sqrt(N)* is the size of each row, column, and zone. The time complexity to verify a potential solution is therefore *O(N),* which is polynomial.

NP-Completeness

The solvability of the generalized Sudoku problem of size $n^2 \times n^2$ has been shown to be NP-Complete, meaning that a solution to the problem can be verified in polynomial time, but the problem cannot be solved in polynomial time (as far as we know).

A Sudoku puzzle has three sets of constraints that must be satisfied:

1. Each of the n^2 zones must contain each of the n^2 symbols exactly once.
2. Each of the n^2 rows must contain each of the n^2 symbols exactly once.
3. Each of the n^2 columns must contain each of the n^2 symbols exactly once.

The variables of the problem are the $n^2 \times n^2$ (= n^4) total cells, excluding any that have been preassigned fixed-value symbols (the starting clue tiles).

Since the Sudoku problem is a NP-Complete problem, a given potential solution can be verified in polynomial time, but there is not a know way to solve the puzzle in polynomial time. Because it is a NP-Complete problem, Sudoku is equivalent to and can be reduced to any other NP-Complete program. For example, the Sudoku problem is equivalent to and can be reduced to the Hamiltonian cycle problem, or the N-Color problem.

Using backtracking, a Sudoku puzzle of size *N* can be solved in exponential time. In the worst case, if every possible combination of options has to be checked, the runtime would be factorial, O(*N!*). Since the Sudoku problem is comparable /reducible to Hamiltonian Cycle problem, and vice versa, the two problems have similar runtime complexities. Using backtracking to solve the Hamiltonian Cycle problem, the worst case runtime complexity for a graph of size *N* is also O(*N!*).

Proof of NP-Completeness

1.) Prove that the Sudoku problem exists in NP, by showing that a solution can be verified in polynomial time.

A potential solution to the Sudoku problem (a filled board) can be verified in polynomial time by checking every cell on the board to see if it meets the requirements. A Sudoku puzzle of size $n^2 \times n^2$ has three sets of constraints that must be satisfied:

1. Each of the n^2 zones must contain each of the n^2 symbols exactly once.
2. Each of the n^2 rows must contain each of the n^2 symbols exactly once.
3. Each of the n^2 columns must contain each of the n^2 symbols exactly once.

So for each cell, the entry must satisfy 3 conditions:

1. The entry must not be present anywhere else in its row.
2. The entry must not be present anywhere else in its column.
3. The entry must not be present anywhere else in its zone.

These checks for each cell can be done in polynomial time, since each check will take at most the time it takes to search the entire row /column /zone. On a board of size $n^2 \times n^2$, each row, column, and zone will be size n^2, and on a board of total size N, each row, column, and zone will be size *sqrt(N)*.

Therefore, given a filled board of total size N (where N is the total cells on the board, not the size of a zone or side), we can determine if the board is solved in time *N * 3(sqrt(N))*, because N is the total number of cells to be verified, and *sqrt(N)* is the size of each row, column, and zone. The time complexity to verify a potential solution is therefore *O(N)*, which is polynomial.

Therefore, the Sudoku problem can be solved in polynomial time, and does exist in NP.

2.) Show that a know NP-Complete problem is able to reduce to the Sudoku problem, and that this know NP-Complete problem can be transformed into the Sudoku problem by a polynomial algorithm.

a.) Chose a know NP-Complete problem, preferably with a similar structure, to compare to the Sudoku problem.

The N-Color problem is a know NP-Complete problem. In this problem, given a graph of nodes and a set of *n* colors, it must be determined if it is possible to color the nodes in the graph so that no node is directly linked to (is adjacent to) another node of the same color.

b.) Show a polynomial algorithm to transform an arbitrary instance of *x* in the N-Color problem into an instance of *x'* in the Sudoku problem.

The N-Color problem can be modified and transformed into (reduced to) the Sudoku problem in the following manner:

The graph contains *N* total nodes, and these nodes are linked together in the following manner:

1. There are *sqrt(N)* "*R*" sets R_1 .. $R_{sqrt(N)}$, and each of these sets contains *sqrt(N)* total nodes.
2. There are *sqrt(N)* "*C*" sets C_1 .. $C_{sqrt(N)}$, and each of these sets contains *sqrt(N)* total nodes.
3. There are *sqrt(N)* "*Z*" sets Z_1 .. $Z_{sqrt(N)}$, and each of these sets contains *sqrt(N)* total nodes.

4. Each node in the graph is directly linked to (is adjacent to) the *sqrt(N) - 1* other nodes in its "*R*" set.
5. Each node in the graph is directly linked to (is adjacent to) the *sqrt(N) - 1* other nodes in its "*C*" set.
6. Each node in the graph is directly linked to (is adjacent to) the *sqrt(N) - 1* other nodes in its "*Z*" set.

It must be determined if all *N* nodes in the graph can be colored with the *sqrt(N)* distinct colors, such that no nodes are adjacent to other nodes of the same color. This is the same as the Sudoku problem, where it must be determined if all *N* cells on the board can be assigned one of the *sqrt(N)* symbols such that no cells share their assigned symbol with any of their adjacent cells (those that share a row, column, or zone).

The nodes in this graph represent the cells on the Sudoku board, which are the variables of the problem. The edges on the graph represent the adjacency relationships between cells that share rows, columns, or zones, and are the constraints that must be satisfied. The set of *sqrt(N)* distinct colors represents the set of *sqrt(N)* distinct symbols used to fill the Sudoku board.

Therefore, the N-Color problem, a known NP-Complete problem, can be transformed into and reduced to the Sudoku problem.

c.) Prove that the instance of *x* in the N-Color problem can only exist if the instance of *x'* in the Sudoku problem exists. That is, if *x* exists in the N-Color problem, then *x'* exists in the Sudoku problem, and if *x'* exists in the Sudoku problem, then *x* exists in the N-Color problem.

Just as the N-Color problem can be transformed into and reduced to the Sudoku problem, the Sudoku problem can be transformed into and reduced to the N-Color problem in the following manner:

Each cell in the Sudoku board of size $n^2 \times n^2$ is a node in a graph. Each cell (node) is directly connected to (is adjacent to):

- every other cell (node) that is in its row of size n^2,
- every other cell (node) in its column of size n^2, and
- every other cell (node) in its zone of size n^2.

The n^2 available symbols are now n^2 available colors. It must be determined if each node (cell) in the graph can be colored so that no nodes are directly linked to (adjacent to) any other nodes of the same color. If this is possible, then there is a solution to the Sudoku puzzle.

The n^4 nodes in the graph are equivalent to the n^4 cells on the board. The edges are equivalent to the relationships between cells on the Sudoku board, which are the Sudoku row, column, and zone constraints. The n^2 colors used to color the nodes are equivalent to the n^2 symbols used to fill the Sudoku board's cells.

Therefore, just as the N-Color problem can be transformed into and reduced to the Sudoku problem, the Sudoku problem can be transformed into and reduced to the N-Color problem. Any instance of *x'* in the Sudoku problem can be transformed into any instance of *x* in the N-Color problem, and any instance of *x* in the N-Color problem can be transformed into an instance of *x'* in the Sudoku problem.

If *x* exists in the N-Color problem, *x'* exists in the Sudoku problem, and if *x'* exists in the Sudoku problem, *x* exists in the N-Color problem.

Therefore, since both 1 and 2 are true, we have proven that the Sudoku problem is in fact an NP-Complete problem.

Algorithms Used

Verifying Solutions:

To verify a potential solution, my program uses a polynomial-time algorithm that checks each cell on the board in turn, checking if its entry satisfies these constraints:

1. The entry must not be present anywhere else in its row.
2. The entry must not be present anywhere else in its column.
3. The entry must not be present anywhere else in its zone.

If at any point an entry is found to not satisfy any of these constraints, the function returns false. If it reaches the end of the board without finding any discrepancies, the game's constraints have been met, and the game must therefore be solved.

Given a filled board, the program can verify if the board is filled with a valid solution in runtime of *O(N)*, where *N* is the total number of cells on the board, by checking if each cell on the board is valid and meets the required constraints.

Solving Puzzles:

Since the Sudoku program is a NP-Complete problem, to solve a given board, my program uses a backtracking algorithm, which is a brute-force, depth-first, recursive approach. It recursively tries available options /combinations of options until it either returns a solution or exhausts all the possibilities and returns null /false.

Generating Puzzles:

The algorithm I use to generate new Sudoku puzzles relies on the recursive, backtracking algorithm used to solve puzzles. To save some time, however, it starts by filling in two zones on the board in opposite corners. It fills each of these zones by simply shuffling the list of symbols (or "tiles") and inserting them into the cells in each zone. These two zones can be filled first without needing to verify if each entry is valid, because the two zones do not share any rows or columns (or zones). My program starts with the top left and bottom right zones, but other starting zones could be used instead.

After these first two zones are filled, the rest of the board is filled in a recursive manner using the solving algorithm. The program walks through all the remaining empty cells on the board, and at each index, it selects a symbol /tile from the available options and checks if this selection is a valid entry. If the selection is valid, it continues in a depth-first fashion, until it has either completed filling the board, or reached a cell with no remaining valid options. If it reaches a cell with no remaining options, the program backtracks to the last point with alternative options, chooses the next available option at this fork, and then continues filling the board from there.

In this way, the program explores all branch possibilities until either a solution is found (the board is filled), or all possibilities have been exhausted (the board is unsolvable and invalid). If the board is unsolvable, it starts over again with an empty board, and begins again by filling in the two starter zones with randomly shuffled tiles.

Once the entire board is filled with a valid solution, the program must create the actual puzzle, which only contains the starting clue tiles. To do this it removes tiles at random indexes until only the starting number of tiles remains.

The standard Sudoku board is size 9 × 9, and must contain at least 17 starting clue tiles. My program is able to also generate boards of size 4 × 4 and 16 × 16. In theory, it should be able to make larger boards as well, but doing so takes a very long time.

Playing the Game:

The program allows the user to customize the playing board if desired. The default board is size 9 × 9, with 17 starting clue tiles, but boards of size 4 × 4 or 16 × 16 are also available, and the number of starting tiles can be changed, as long as it is more than a set minimum.

The default tile sets use numbers: for a 9 × 9 board, the tiles are the numbers 1 through 9, for a 4 × 4 board, the tiles are numbers 1 through 4, and for a 16 × 16 board, the tiles are numbers 1 through 16. This game also allows players to use letter tiles if they wish.

The game is played in the terminal, and upon each move, the player is shown the board, and directed to enter a row, column. If the cell is empty, the player is directed to enter a tile to insert. If the cell holds a previous entry, the player is allowed to edit their entry if they desire. If the cell holds a starting tile, the player is asked to select a different cell.

Once the board is filled, the program checks if the game has been solved. If it hasn’t been solved, the player can continue playing, and if it has been solved, the player can begin a new game if they wish.