FIT

Documentation for the semester work called **Sudoku solver**

Created by:
Karolina Zegeryte
2023

Description:

The main goal of this project is to create a solver for the well-known puzzle called Sudoku using different techniques and algorithms to test their effectivity in this task.

Sudoku is a popular puzzle game that consists of a 9x9 grid divided into 9 3x3 blocks. The goal of the game is to complete the grid so that each row, each column, and each block contains all the numbers from 1 to 9, with each number only having to appear once in each row, column, and block.

The program has 3 algorithms for solving Sudoku puzzle:

1. Brute Force using backtracking method;
2. modified CSP using backtracking, forward checking and MRV;
3. Satisfiability (SAT) or conversion to propositional feasibility;

Each of these used algorithms will be described on the pages below.

The program is written in Python and uses the PySAT library, which provides an interface to various SAT solvers. In this case, the Glucose4 solver is used.

Backtracking Sudoku Solver:

The program "Backtracking Sudoku Solver" (the first algorithm) is designed to solve the Sudoku game using the backtracking method.

Which features it includes:

a. **find_empty_location(board_, location):**
This function takes a Sudoku board and a list with two elements as arguments. It runs through the entire Sudoku board and returns the first empty cell (represented by zero), updating the location list with that cell's coordinates. If there are no empty cells, the function returns False.

b. **is_in_row(board_, row_, num),**
is_in_column(board_, col, num),
is_in_box(board_, box_start_row, box_start_col, num):
These functions check if the specified number is used in the corresponding row, column, or 3x3 box. All functions return True if the number is already in use, and False otherwise.

c. **is_safe(board_, row_, col, num):**
This function takes a board, cell coordinates, a number and checks if it is safe to put that number in the specified cell. A number is considered safe if it is not already used in the corresponding row, column, and 3x3 block.

d. **solve_sudoku_brute_force(board_):**
This function implements a reverse lookup method for solving a Sudoku game. It looks for an empty cell on the board and then tries to put the numbers from 1 to 9 into that cell, checking each one for safety. If the number can be safely placed in the cell, the function calls itself recursively for the next empty cell. If at some point it is not possible to find a suitable number for an empty cell, the function returns False, which causes the previous step to be rolled back.

It is important to note that this code implements a reverse lookup method, which may not be efficient for complex Sudoku puzzles due to its exponential nature. This algorithm is used to make the comparison between CST and SAT algorithm more exciting.

CSP Sudoku Solver:

The program "CSP Sudoku Solver" is designed to solve the game of Sudoku using the concept of General Satisfaction Constraints (CSP - Constraint Satisfaction Problem).

The main classes and functions of the program:

a. **class CSP**:
Defines the base class for the satisfaction constraint task.

-__init__(self, variables, domains, constraints):
Initializes an object of the CSP class with the given variables, domains, and constraints.
Input parameters:
variables (list): List of task variables.
domains (dictionary): A dictionary where keys are variables and values are lists of valid values for each variable.
constraints (dictionary): A dictionary where the keys are variables and the values are sets of neighboring variables associated with this variable by constraints.
Includes the following methods:

- is_satisfied(self, var, value, assignment):
Checks if the given value of a variable satisfies the specified constraints.
Input parameters:
var (variable): The variable for which the value is being checked.
value: The value to be checked.
assignment (dictionary): A set of already assigned variables and their values.
Output value:
True if the value satisfies the constraints, *False* otherwise.

- assign(var, value, assignment):
Assigns a value to a variable in an assignment set.
Input parameters:
var (variable): The variable to be assigned a value.
value: The value to be assigned to the variable.
assignment (dictionary): A set of already assigned variables and their values.

- revert_assignment(var, assignment):
Removes an assignment to a variable from an assignment set.
Input parameters:
var (variable): The variable to be removed from assignments.
assignment (dictionary): A set of already assigned variables and their values.

- **n_conflicts(self, var, value, assignment):**

Returns the number of conflicts (constraint mismatches) for the given variable, given the given value.

Input parameters:

var (variable): The variable for which to count the number of conflicts.

value: The value of the variable for which to count the number of conflicts.

assignment (dictionary): A set of already assigned variables and their values.

Output value:

The *number* of conflicts (constraint mismatches) for the given variable at the given value.

Static methods:

b. **backtracking_search(csp):**

Runs a backtracking algorithm to solve the CSP problem.

Input parameters:

csp (object of CSP class): The CSP task to be solved.

Output value:

The solution to the CSP problem as a *set* of assigned variables and their values, or *None* if no solution is found.

c. **backtrack(assignment, csp):**

An auxiliary recursive function for finding a solution to the CSP problem using the backtracking search algorithm.

Input parameters:

assignment (dictionary): The current set of assigned variables and their values.

csp (object of CSP class): The CSP task to be solved.

Output value:

The solution to the CSP problem as a *set* of assigned variables and their values, or *None* if no solution is found.

d. **select_unassigned_variable(assignment, csp):**

Implements the MRV (Minimum Remaining Values) heuristic for selecting the next unassigned variable.

Input parameters:

assignment (dictionary): The current set of assigned variables and their values.

csp (object of class CSP): CSP task.

Output value:

The unassigned *variable* with the fewest possible values.

Functions for Solving Sudoku

e. **convert_sudoku_board_to_csp(sudoku):**

Converts a Sudoku problem to an instance of the CSP class.

Input parameters:

sudoku (list): A Sudoku matrix where empty cells are denoted by zeros.

Output value:

An instance of the *CSP* class representing a Sudoku problem.

h. **solve_sudoku_CSP(sudoku):**

Solves a Sudoku problem using CSP algorithm and MVR technique.

Input parameters:

sudoku (list): A Sudoku matrix where empty cells are denoted by zeros.

Output value:

The solution of a Sudoku problem in the form of a *matrix*, where the empty cells are filled with the corresponding values.

The basic idea behind the "Minimum Remaining Values (MRV)" heuristic is to select the variable with the fewest possible values to assign first. This helps to reduce the size of the search space.

To improve, we can add "Forward Checking" , which allows us to check if all variables associated with the current one have possible values to assign. If not, then we go back a step (backtrack), which allows us not to follow the impossible path.

Sudoku SAT Solver:

The program "Sudoku SAT Solver" is designed to solve the game Sudoku using the technique of Satisfiability (SAT) solvers.

The algorithm converts the game rules and the current state of the Sudoku board into a conjunctive normal form (CNF) formula, which is then solved by the SAT solver.

The algorithm contains a set of functions for converting a Sudoku board into a CNF formula, generating Sudoku rules in the form of CNF clauses, and solving the formula using the SAT solver.

Therefore, this algorithm is an effective tool for solving Sudoku puzzles using the powerful techniques available in the area of constraint satisfaction and SAT solving.

Functions:

a. **variables_to_numbers(i, j, d):**

Returns a unique variable number for a cell with coordinates (i, j) and digit d.
Function parameters:

i: cell row index, from 1 to 9.

j: cell column index, from 1 to 9.

d: the number in the cell, from 1 to 9.

The function returns a unique variable number.

b. **set_sudoku_rules() function:**

Generates basic Sudoku rules that must be followed. The rules include the stipulations that each block, row, and column must contain all the digits from 1 to 9, and each digit can only appear once.
The function returns a list of all Sudoku rules, represented as a *list* of CNF clauses.

c. **initial_board_clauses(board_) function:**

Generates CNF clauses based on the initial configuration of the Sudoku board.
The function parameter is a 2D list representing the initial state of the Sudoku *board*.
The function returns a *list* of CNF clauses corresponding to the board's initial configuration.

d. **board_to_cnf(board_) function:**

Converts the current state of the Sudoku board and Sudoku rules into a conjunctive normal form (CNF) formula.
The input parameter is a 2D *list* representing the state of the Sudoku board.
The function returns a CNF *formula* which can then be used to solve the board with the SAT solver.

e. **solve_sudoku_SAT(board_) function:**

This feature uses a SAT solver to solve a Sudoku board.
The input parameter is a 2D *list* representing the state of the Sudoku board. If a solution exists, the function returns a 2D list with the solution. If no solution exists, the function returns None.

The **main** block of code that runs when this file is executed defines a Sudoku board and uses the solve_sudoku_() functions to attempt to solve the puzzle. If a solution is found, it is printed to the console, otherwise or in case of given an inappropriate grid the error sentence is shown. It takes board from the file and decides which algorithm to use depending on user's input:

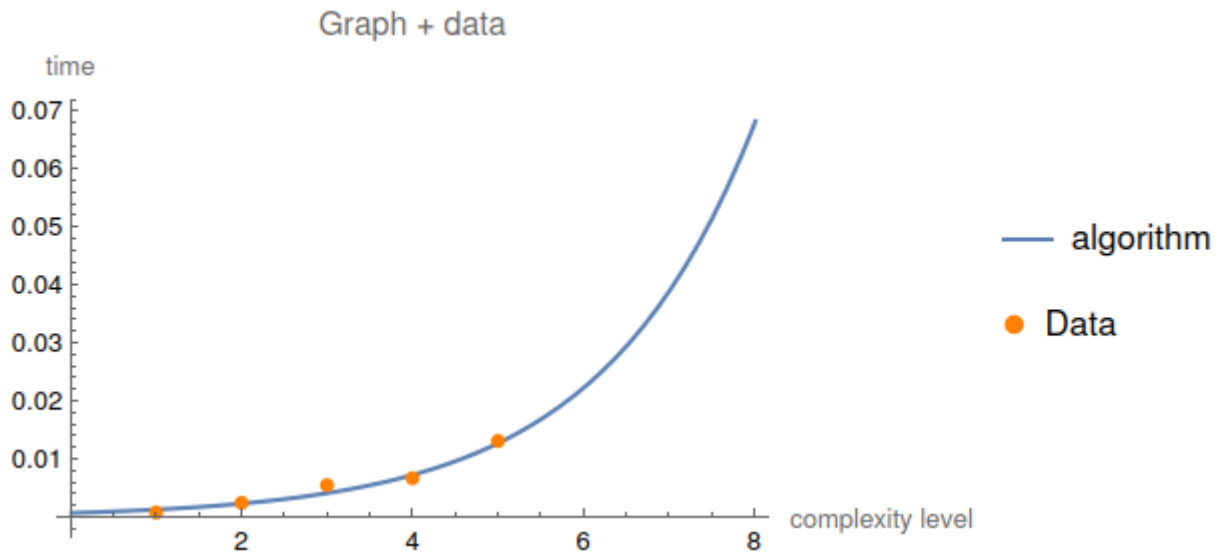
1 – runs brute force algorithm
2 – runs CSP algorithm
any other number – runs SAT algorithm

Experimental and testing section:

In this code three different techniques of creating a solution where it exists were taken and here are measurements of time using library time for each technique at each complexity level of Sudoku puzzle board:

1 – easy level
2 – normal level
3 – hard level
4 – expert level
5 – expert+ level

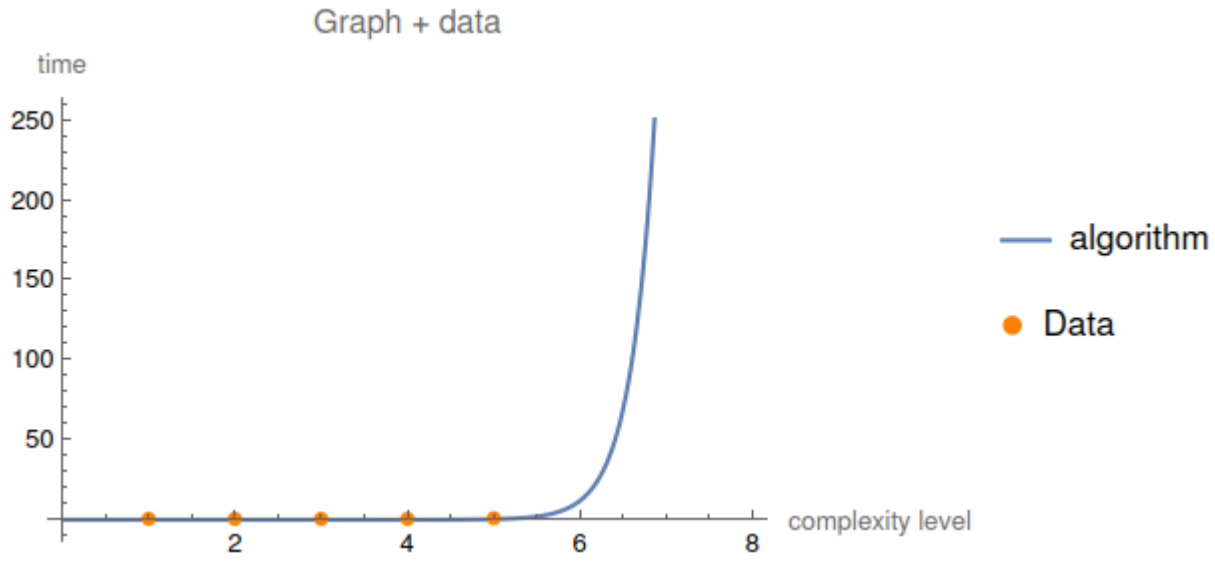
For **Brute force** we have the following graph results:



And the average measurements for each time and level are:

Easy = 0.0007185935974121094
Normal = 0.002210855484008789
Hard = 0.0044095516204833984
Expert = 0.006468057632446289
Expert+ = 0.0074901580810546875

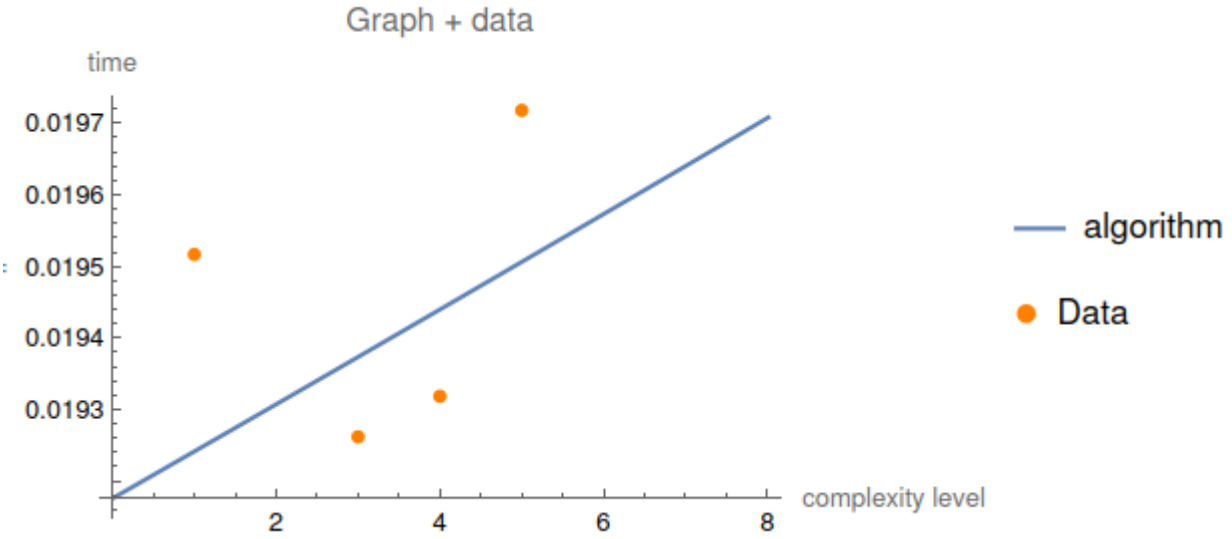
For **CSP**:



And the average measurements for each time and level are:

Easy = 0.0013988018035888672
Normal = 0.0021529605102539062
Hard = 0.004818916320800781
Expert = 0.007306575775146484
Expert+ = 0.37527036666870117

For **SAT**:



And the average measurements for each time and level are:

Easy = 0.019144296646118164
Normal = 0.019157886505126953
Hard = 0.01960468292236328
Expert = 0.019774436950683594
Expert+ = 0.019955158233642578

As the conclusion, I could say that these results surprise, because Brute Force seems to be the fastest algorithm with Sudoku puzzle, despite the fact that Brute Force and CSP we running for almost the same time except the last complexity level results. It was quite predictive that the time of running of each algorithm will raise with the complexity of the given problem, as we see on graphs above. According to my subjective point of view, I could say that SAT seems to be the best algorithm. The time of searching for the solution is almost the same for each complexity which may mean that the main time consuming calculation may take other functions, such as board_to_cnf with set_sudoku_rules and initial_board_clauses. If these functions will be updated and give better performance according the time, the results may be much better than given by Brute Force.