**Vietnam National University – HCM International University**

**School of Computer Science and Engineering**



**ALGORITHMS AND DATA STRUCTURES**

**PROJECT REPORT**

Semester 2, 2023 – 2024

**Instructor: Vi Chi Thanh, Thai Trung Tin**

**Topic: Sudoku Solver**

Name: Nguyen Nhat Khiem

ID: ITDSIU21091

Github repository: [link](https://github.com/Khim3/Sudoku_Solver_DSA_Project)

Contents

[**CHAPTER 1: INTRODUCTION** 3](#_Toc169209694)

[**Overview** 3](#_Toc169209695)

[**Project objectives** 3](#_Toc169209696)

[**CHAPTER 2: PROJECT TIMELINE** 3](#_Toc169209697)

[**CHAPTER 3: METHODOLOGY** 4](#_Toc169209698)

[**Algorithms and Data Structures application** 4](#_Toc169209699)

[**Project Structure** 5](#_Toc169209700)

[**UML** 5](#_Toc169209701)

[Class Descriptions 6](#_Toc169209702)

[Dependency and Association 7](#_Toc169209703)

[**User Interface** 7](#_Toc169209704)

[**App Implementation** 11](#_Toc169209705)

[**CHAPTER 4: CONCLUSION** 12](#_Toc169209706)

[**Accomplishment** 12](#_Toc169209707)

[**Incomplete problems** 12](#_Toc169209708)

[**Future works** 12](#_Toc169209709)

[**References** 13](#_Toc169209710)

# **CHAPTER 1: INTRODUCTION**

## **Overview**

This project is a Sudoku solver developed in Python. It uses logical reasoning to solve any valid Sudoku puzzles. The user-friendly interface, built with Python’s Tkinter GUI package, allows users to input puzzles and visualize the solving process interactively.

## **Project objectives**

The primary objectives of this project are:

1. Implement a Sudoku solver using Backtracking and AC3 algorithms.
2. Compare the performance of these algorithms in solving Sudoku puzzles.
3. Develop an intuitive UI using Tkinter for puzzle input and solution visualization.
4. Enhance user experience with features like solving predefined or random puzzles.
5. Deepen understanding of DSA and OOP concepts through practical application.

# **CHAPTER 2: PROJECT TIMELINE**

|  |  |  |
| --- | --- | --- |
| Stage | Tasks | Week no. |
| Planning | Topic research | 1 |
| Set up timeline |
| Define specififation, requirements |
| Topic confirmation |
| Preparation | Create github repo | 2 |
| Design project structures |
| Get familiar with tkinter and python | 3 - 4 |
| Decide the scopes, aims, and target audients of the projects |
| Implementation | Create the main file with basic UI, add singleton design | 5 |
| Design OOP structure for the project files |
| Modeling the problem as CSP | 6-7 |
| Implement backtracking and AC3 and evaluate with simple puzzles |
| Create input problems with various difficulties (easy, medium, hard) | 8-9 |
| Design the UI appearance |
| Add buttons with functions to the UI | 10 |
| Add custom input feature as well as random problem generator |
| Evaluate and test for potential bugs | 11 |
| Finalize and clean up code | 12 |
| Presentation | Final report and readme | 13-15 |
| Presentation slides |

Table 2. Timeline of the project

# **CHAPTER 3: METHODOLOGY**

## **Algorithms and Data Structures application**

The project employs two primary algorithms for solving Sudoku puzzles: Backtracking and AC3 (Arc Consistency Algorithm 3).

* Algorithms: The project uses Backtracking and AC3 algorithms for solving Sudoku puzzles. Backtracking assigns numbers to cells sequentially, backtracking when no solution is found. AC3 reduces the search space by making each variable arc-consistent with each other.
* Data Structures: The project uses lists for variables, units, and peers, a dictionary for domains, and a queue for the AC3 algorithm.
* Comparison: Backtracking is simple but can be slow for complex puzzles. AC3 is more efficient as it reduces the search space, but it can’t solve puzzles that require guessing and checking.

|  |  |  |
| --- | --- | --- |
|  | Backtracking + MRV | AC3 |
| Time complexity | or |  |
| Space complexity |  |  |
| Time taken | Fast due to improvement | Fast, but time taken can vary |
| Completeness | Always | Not always |
| Optimality | Less optimal | More optimal |
| Data Structue | Stack | Queue |
| Applicability | Any csp | Arc consistency greatly trims the search space |

16of30

Table 3. Algorithm comparison

## **Project Structure**

The project is organized into directories for inputs, outputs, and assets. The input directory contains Sudoku puzzles, and the output directory stores the solutions. Key files include AC3.py and Backtrack.py for the Sudoku-solving algorithms, csp.py for constraint satisfaction problems, sudoku.py for the main logic, and util.py for utility functions. README.md provides instructions and explanations, and requirements.txt lists all the dependencies required to run the project.

## A screenshot of a computer Description automatically generated**UML**

Figure 2. Project's UML

### Class Descriptions

#### SudokuSolver

* Functionality: Manages the Sudoku solver application's UI.
* Attributes:

\_instance: Singleton instance.

root: Tkinter main window.

grid\_frame, cells: GUI elements for the Sudoku grid.

selected\_algorithm, algoLabel, algoMenu: Components for algorithm selection.

selected\_difficulty, difficultyLabel, difficultyMenu: Components for difficulty selection.

solveButton, clearButton, validateButton, generateButton: Control buttons.

timeLabel, time, timeDisplay: Components for displaying solving time.

openButton: Button to open a predefined solution.

initial\_cells: Cells that are initially filled.

* Methods:

\_\_init\_\_(root), \_\_new\_\_(cls), get\_instance(cls, root): Initialization and singleton pattern.

create\_widgets(): Sets up the GUI.

open\_solution(), validate(), solve(), \_solve(), generate\_random(), clear(): Core functionalities.

#### csp

* Functionality: Represents the CSP for Sudoku.
* Attributes: variables, domain, unitlist, units, peers, constraints, values: Core CSP components.
* Methods:

\_\_init\_\_(domain, grid): Initializes the CSP.

getDict(grid): Converts grid string to dictionary.

#### BacktrackingSolver

* Functionality: The BacktrackingSolver class implements a backtracking algorithm specifically tailored for solving Sudoku puzzles. Backtracking is a depth-first search algorithm that incrementally builds candidates to the solutions and abandons a candidate (“backtracks”) as soon as it determines that this candidate cannot possibly lead to a valid solution.
* Attributes:

csp: A CSP instance representing the Sudoku puzzle, including variables (cells), domains (possible numbers), and constraints (Sudoku rules).

assignment: A dictionary that keeps track of the current assignments of values to variables.

* Methods:
  + \_\_init\_\_(csp): Initializes the solver with a CSP instance. It sets up the necessary data structures for solving the given Sudoku puzzle.
  + Backtracking\_Search(csp\_instance): This method sets the current CSP instance to the provided csp\_instance and starts the backtracking search by calling Recursive\_Backtracking().
  + Recursive\_Backtracking(): The core method where backtracking occurs. It checks if the current assignment is complete; if not, it selects an unassigned variable and iterates through its domain values, applying consistency checks and making inferences where possible. If an inconsistency is found, it backtracks by undoing the last assignment.
  + Inference(inferences, var, value): Applies inferences to reduce the search space. If a value is assigned to a variable, it removes this value from all its peers’ domains.
  + Select\_Unassigned\_Variables(): Utilizes the Minimum Remaining Values (MRV) heuristic to select the variable with the fewest legal values left in its domain.
  + isComplete(): Checks if the current assignment covers all variables, indicating that a solution has been found.
  + isConsistent(var, value): Ensures that no constraints are violated by assigning a value to a variable.
  + write(values): Generates a string representation of the solution from the current assignments.
* Algorithm Explanation: The backtracking algorithm tries to assign values to all variables such that no constraints are violated. It selects an unassigned variable and tries all possible values in its domain. If an assignment does not lead to a violation of constraints, it recursively proceeds with this partial solution. If at any point a constraint is violated, it backtracks and tries another value until all variables are assigned or no solutions are possible.
* Complexity: The worst-case time complexity of backtracking algorithms like this one is generally O(n^d), where n is the number of variables (81 in standard Sudoku) and d is the domain size (9 possible values per cell). However, due to pruning from consistency checks and intelligent variable selection (MRV), practical performance can be significantly better than this theoretical upper bound. The complexity of individual methods such as Inference() and Select\_Unassigned\_Variables() can be up to O(n \* d) and O(n) respectively.

#### AC3Solver

* Functionality: The AC3Solver class implements the AC3 (Arc Consistency Algorithm #3) algorithm, which is used to reduce the search space in constraint satisfaction problems (CSP) by achieving arc consistency. A CSP is arc-consistent if, for every variable in the problem, every value in its domain satisfies the variable’s binary constraints.
* Attributes:

csp: An instance of a CSP that contains variables, domains, and constraints that need to be satisfied.

* Methods:
  + \_\_init\_\_(csp): Initializes the AC3Solver with a given CSP instance. This method sets up the solver by storing a reference to the CSP instance which includes variables, domains, and constraints.
  + AC3(): This is the main method that implements the AC3 algorithm. It initializes a queue with all arcs in the CSP and processes each arc to achieve arc consistency. If an arc is found to be inconsistent, the domain of the variable is revised. This process continues until either all arcs are consistent or one of the domains becomes empty, indicating that the CSP cannot be solved.
  + Revise(Xi, Xj): Revises the domain of variable Xi by removing any value that does not satisfy the constraint between Xi and Xj. If a value is removed, it returns True indicating that a revision was made.
  + isConsistent(x, Xi, Xj): Checks if a value x for variable Xi is consistent with every value in the domain of Xj based on the constraints between Xi and Xj.
  + isComplete(): Determines if a solution has been found by checking if every variable has exactly one value left in its domain.
  + write(values): Outputs a string representation of the solution by concatenating the values of all variables in their order.
* Algorithm Explanation: The AC3 algorithm works by establishing arc consistency for all variables in a CSP. An arc (Xi, Xj) is consistent if for every value in the current domain of Xi, there is some allowed value in the domain of Xj. The algorithm iteratively selects an arc from a queue and revises the domain of Xi. If a value is removed from Xi’s domain, then all neighbors of Xi are added back to the queue to check for consistency again.
* Complexity: The worst-case time complexity of AC3() can be O(n^2 \* d^3), where n is the number of variables and d is the maximum domain size. This accounts for every arc being processed multiple times due to revisions in domains. The Revise() function has a complexity of O(d^2), as it checks each value in Xi’s domain against Xj’s domain values. The other methods (isConsistent, isComplete, and write) have complexities ranging from O(d) to O(n), depending on whether they iterate over domains or variables.

#### util

* Functionality: Provides utility functions.
* Methods:

raiseNotDefined(): Error for unimplemented methods.

cross(A, B): Returns cross product of two sets.

### Dependency and Association

* SudokuSolver depends on csp, BacktrackingSolver, and AC3Solver.
* BacktrackingSolver and AC3Solver depend on csp.
* csp uses utilities from util.
* SudokuSolver interacts with Tkinter, threading, and random modules.

## Psuedo code

### Backtring + MRV

Input: csp (constraint satisfaction problem)

Output: assignment (solution to the problem)

1. Function Initialize(csp)

- csp <- csp

- assignment <- {}

2. Function Backtracking\_Search(csp\_instance)

- csp <- csp\_instance

- return Recursive\_Backtracking()

3. Function Recursive\_Backtracking()

- if assignment is complete

- return assignment

- var <- Select\_Unassigned\_Variables()

- domain <- copy of csp.values

- for each value in csp.values[var]

- if value is consistent with assignment

- add var=value to assignment

- inferences <- Inference(inferences, var, value)

- if inferences != "FAILURE"

- result <- Recursive\_Backtracking()

- if result != "FAILURE"

- return result

- remove var from assignment

- update csp.values with domain

- return "FAILURE"

4. Function Inference(inferences, var, value)

- add var=value to inferences

- for each neighbor of var

- if neighbor not in assignment and value in csp.values[neighbor]

- if csp.values[neighbor] has only one value

- return "FAILURE"

- remaining <- csp.values[neighbor] without value

- if remaining has only one value

- flag <- Inference(inferences, neighbor, remaining)

- if flag == "FAILURE"

- return "FAILURE"

- return inferences

5. Function Select\_Unassigned\_Variables()

- unassigned\_variables <- dictionary of (variable, length of csp.values[variable]) for each variable not in assignment

- mrv <- variable with minimum value in unassigned\_variables

- return mrv

6. Function isComplete()

- return if set of assignment keys is equal to set of squares

7. Function isConsistent(var, value)

- for each neighbor of var

- if neighbor in assignment and assignment[neighbor] == value

- return False

- return True

8. Function write(values)

- output <- empty string

- for each variable in squares

- append values[variable] to output

- return output

### AC3

Input: csp (constraint satisfaction problem)

Output: True if assignment is complete, False otherwise

1. Function Initialize(csp)

- csp <- csp

2. Function AC3()

- Initialize queue q

- For each variable Xi in csp.variables

- For each peer Xj of Xi in csp.peers[Xi]

- Add (Xi, Xj) to queue q

- While q is not empty

- (Xi, Xj) <- first element of q

- Remove first element from q

- If Revise(Xi, Xj) is true

- If the length of csp.values[Xi] is 0

- Return False

- For each Xk in (csp.peers[Xi] - set(Xj))

- Add (Xk, Xi) to queue q

- Return True

3. Function Revise(Xi, Xj)

- revised <- False

- values <- set of csp.values[Xi]

- For each x in values

- If isConsistent(x, Xi, Xj) is false

- Remove x from csp.values[Xi]

- revised <- True

- Return revised

4. Function isConsistent(x, Xi, Xj)

- For each y in csp.values[Xj]

- If Xj is a peer of Xi and y != x

- Return True

- Return False

5. Function isComplete()

- For each variable in squares

- If the length of csp.values[variable] is more than 1

- Return False

- Return True

6. Function write(values)

- output <- empty string

- For each variable in squares

- Append values[variable] to output

- Return output

## **User Interface**

* User Interface design

Designing an intuitive and functional user interface was one of the most challenging aspects of this project. To address this, we decided to leverage the simplicity and versatility of the Tkinter library in Python to create our application’s default user interface.

* Main User Interface

Upon launching the application, users are greeted with a main user interface that presents various options, each represented by a button with its unique functionality. By default, pressing the “Solve” button triggers the application to solve a predefined set of 20 basic Sudoku problems using the default algorithm, which is backtracking.

* A screenshot of a computer

  Description automatically generatedSolution display and algorithm selection

Figure 3 . The app's interface

Once the solving process is complete, users can press the “Open Solution” button to display the solution of the recently chosen problems. Users have the flexibility to select the algorithm used for solving, with AC3 being one of the available options.

A screenshot of a computer

Description automatically generated

Figure 4. Option to show the solution for users

* Time Complexity Illustration

To provide users with essential details for algorithm evaluation, we incorporated a time counter that measures the duration of all operations. This feature is particularly useful when illustrating the time complexity of different algorithms, such as the backtracking algorithm when applied to a collection of 15 hard Sudoku puzzles (problem3).

A screenshot of a computer

Description automatically generated

Figure 5. Time taken for solving with Backtracking

* Problem Generation and Customization

For users interested in solving randomly generated problems, Users can choose from Easy, Medium, or Hard levels, and solve them with a click of the “Solve” button. The initial problem numbers are shown in green, and the solution is highlighted in blue.

A screenshot of a computer

Description automatically generated

Figure 6. Solutions for random puzzles

* Custom Problem entry

The “Custom” option in the aplication allows users to input their own Sudoku puzzles, validate them, and choose the solving algorithm. This feature enhances user engagement by offering customization and serves as a learning platform for understanding different Sudoku-solving algorithms.

A screenshot of a computer

Description automatically generated

Figure 7. Vadility check notification for custom puzzle

A screenshot of a computer

Description automatically generated

Figure 8.Filled solution for random puzzle

## **App Implementation**

* OOP: The app uses Object-Oriented Principles for clear data and method encapsulation, making the code maintainable and scalable.
* Design Pattern: The Singleton pattern ensures only one instance exists, avoiding inconsistencies and increased memory usage.
* Algorithms: Backtracking and AC3 algorithms are implemented for solving Sudoku puzzles. Backtracking uses a recursive function, while AC3 reduces the problem’s domain size by enforcing arc consistency.
* User Interface: The Tkinter library is used to build a simple and intuitive interface, allowing users to solve puzzles, view solutions, generate new puzzles, and choose difficulty levels and solving algorithms.
* Multithreading: Multithreading runs the Backtracking and AC3 algorithms in parallel, speeding up the solving process. Threads are managed using the built-in threading module. This robust and efficient Sudoku solver not only solves puzzles but also helps users understand and compare different solving algorithms.

# **CHAPTER 4: CONCLUSION**

## **Accomplishment**

* + Algorithms: The project successfully implemented Backtracking and AC3 algorithms in Python for solving Sudoku puzzles.
  + Performance measure: A detailed comparison was conducted between the two algorithms, providing insights into their efficiency and effectiveness.
  + GUI: An intuitive and engaging graphical interface was developed using the Tkinter library, allowing users to input puzzles and visualize the solving process.
  + Design: The application was designed using Object-Oriented Programming principles and the Singleton design pattern, ensuring modularity, extensibility, and maintainability of the code.

## **Incomplete problems**

* UI Design: Balancing aesthetics and functionality for an engaging user interface was a challenge.
* Algorithm Implementation: While Backtracking and AC3 were used, other advanced techniques like Dancing Links or Stochastic methods could be explored for Sudoku solving.

## **Future works**

* + Additional Algorithm: We plan to add another Sudoku-solving algorithm for improved functionality and performance comparison.
  + UI Customization: Future updates will include options for users to personalize the Sudoku board’s appearance.
  + Computer Vision: We aim to incorporate technology that allows users to input puzzles by taking a picture, which the app will process and convert for solving.
  + User Guide: A user-friendly guide is planned to help users navigate the application and understand its output.

## **References**

1. SÁCH LÊ THÀNH. (n.d.). Retrieved from <https://www.youtube.com/@sachlethanh9853>
2. Welcome to Python.org. (n.d.). Retrieved from <https://www.python.org/>
3. Real Python. (2023). Python GUI Programming With Tkinter. Retrieved from <https://realpython.com/python-gui-tkinter/>
4. Sorting Algorithm Visualizer using Python : Part 1(GUI Tkinter). (2020). Retrieved from <https://www.youtube.com/watch?v=RD1JqhtNgvw&list=PLpp8-k7G_6Y2eVX0WtvsDzkH1PhW2d-Rx>
5. TrungDS10. (n.d.). Introduction-to-Artificial-Intelligent/Sudoku at main · TrungDS10/Introduction-to-Artificial-Intelligent. Retrieved from <https://github.com/TrungDS10/Introduction-to-Artificial-Intelligent/tree/main/Sudoku>