F29AI Coursework 1

Written by Oliver Racey and Godfrey Lam (Group 14)

Part 1: Implementation of a Sudoku Solver using Python

1A:

Sudoku is a well-known logical puzzle, that involves filling a 9x9 grid (subsequently made with 9 3x3 grids) with digits ranging from 1-9.

If we were to describe Sudoku as a CSP (Constraint Satisfaction Problem), then:

CSP is a triple [X,D,C] where X are Variables, D are Domains, and C are constraints. Below I have a readable form, and the proper formal form.

X – Each cell in the 9x9 grid is a value, therefore there are 81 values.

D - For an empty cell, D can be any digit from 1-9, as it is unfilled. For a filled cell, the domain is just the value of the digit.

C – Has 3 rules.

a. No two variables in the same row can have the same value.
b. No two variables in the same column can have the same value.
c. No two variables in the same 3x3 sub grid can have the same value. (Learn-Sudoku, 2008)

In proper formal:

	Formal
Variables (X)	$X = \{X_{i,j} \mid 1 \leq i, j \leq 9\}$
Domains (D)	$D = \{D_{i,j}\}$ For an empty cell $[X_{i,j}]$: $\{D_{i,j}\} = \{1,2,\ldots,8,9\}$ For prefilled cell [v]: $\{D_{i,j}\} = \{v\}$
Constraints (C)	$X_{i,j} \neq X_{p,q}$ For every pair of distinct variables (in Boolean logic), (i = p) v (j = q) v ([(i-1)/3) = [(p-1)/3] ^ [(j-1)/3] = [(q-1)/3] Two cells (i,j) (p,q) are not valid if values are in the same row, column or 3x3 sub grid

Our implementation uses backtracking instead of brute force searching. In brute force, we would try every possible combination of numbers for empty cells. In the worst case, there are 9^{81} possible sudoku combinations. This means the O Complexity is $O(9^M)$ where M is the number of empty cells. This means the complexity is exponential and so is an inefficient approach.

Instead, our approach uses backtracking with constraints/pruning. Backtracking is better for Sudoku because as soon as we break a constraint, we immediately backtrack and try a different digit. Rather than generating and testing every possible grid, it can abandon invalid paths (known as pruning). If we know a digit placement is invalid, then we can cut off that path and reduce the number of explored states. In practice, this can make it exponentially faster. In backtracking, most cells have less options because the constraints narrow down the choices. If we have a valid digit inputted, then we only need to check branches that contain the digit, which effectively means that the branching

factor drops by half. This means that the average runtime of a backtracking search is $O(N^M)$, where N is the branching factor. Both have a worst-case complexity of $O(9^M)$ if we both have empty cells.

Another implementation we could try is an A* search algorithm. A* works so it calculates cost (e.g., the number of assignments made), and A* uses a heuristic estimate which calculates the estimated remaining work, and then it expands nodes that looks the closest to a solution . If the heuristic function is well written, A* can search towards better states and look through fewer nodes, compared to our algorithm which checks every value in a cell. However, A* also takes up more memory because it must keep a list of opened and closed lists of explored states. This means that for a problem like Sudoku, these lists can be huge and use more memory. For sudoku, backtracking with constraint checking is usually faster and more memory efficient. A* is best when you need the shortest path, or you have a problem with a minimum cost (kartik, 2025). As Sudoku only needs a valid board to work, then there is no shorter solution.

1B:

Originally the program was all done procedurally, and the terminal worked as intended. However, due to the optional segment of adding a GUI into the program, we have kept the solver logic procedural in one class, and the GUI is a separate class, that calls the procedural solver. This means that our code is a hybrid of procedural and OOP and is low coupled because the GUI calls the solver, so the procedural solver is logically independent.

Below is a full explanation of our implementation.

The .csv files were formatted as an array of singular integers, separated by commas:

testpuzzle.csv > data

```
8,0,7,1,5,0,0,9,6,0,6,5,3,0,7,1,4,0,3,4,1,0,8,0,7,0,2,5,9,3,4,6,8,2,7,0,4,0,0,0,1,
```

Note that this assumes that the input data is always valid, due to how the data is represented.

We eventually convert these files to be a List of Lists, where each list is a specific row.

```
rt csv
rt tkinter as tk
 tkinter import messagebox, filedialog # for file dialog and message boxes.
rt time
```

Tkinter (Python, 2025) was the GUI library we used. It seemed the most popular and easiest to learn for us.

```
class SudokuFunctions:
    def __init__(self):
        # define board dimensions and initialising the board
        self.row = 9
        self.column = 9
        self.board = [[0 for _ in range(self.column)] for _ in range(self.row)]
        self.backTrackCount = 0 # global variable to count the number of backtracks
```

Above, just showing we define the board, row, column and backtrack count as global variables. This means the GUI can interact with them.

```python
def check_valid(self, num, row, col):
    '''
    function to check if a number can be placed in a specific position
    checks the row, column and 3x3 grid to ensure no duplicates

    '''
    for i in range(9):
        # if a number is also found in the same row or column, return false
        if self.board[row][i] == num and col != i:
            return False
        if self.board[i][col] == num and row != i:
            return False
    # defines the starting row and column of the 3x3 grid
    start_row = (row // 3) * 3
    start_col = (col // 3) * 3
    # for each value in the 3x3 grid, if a number is found, return false
    # this is so we are able to follow the rule of sudoku that states no duplicates in a 3x3 grid
    for i in range(start_row, start_row + 3):
        for j in range(start_col, start_col + 3):
            #  instruction to make sure we don't compare the cell to itself
            if self.board[i][j] == num and (i, j) != (row, col):
                return False
    return True
```

```python
def validate_init_board(self):
    '''
    function to validate the initial board, so that we don't waste time trying to solve an invalid puzzle
    we loop through each value of the board
    if the value is not 0 (not empty), we check if its valid in that position
    if any value is found to be invalid, return false (meaning puzzle is invalid)
    '''
    for i in range(9):
        for j in range(9):
            val = self.board[i][j] # gets the value at the current position
            if val != 0: # if the value is not 0 (not empty)
                self.board[i][j] = 0 # temporarily set it to 0 to avoid self-comparison in check_valid
                if not self.check_valid(val, i, j):  # check if the value is valid in that position
                    self.board[i][j] = val # restore the value
                    return False # if not valid, return false
                self.board[i][j] = val # restore the value
    return True
```

check_valid is a function that a number can be placed in a row, column or 3x3 sub grid. If we find a value that is equal to the number in the row, column, or sub grid, then we would return false. This means we do not revisit this digit again, whether its for the column, row or sub grid

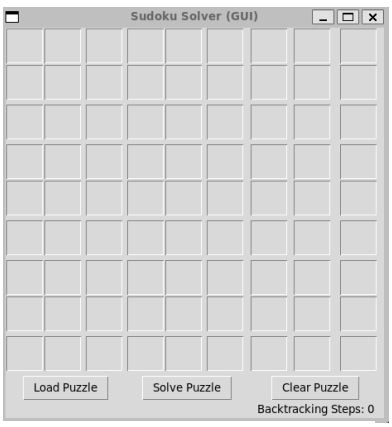
validate_init_board is a function to check that the puzzle we are solving is solvable. This just checks it at the start of the program, not when the csv file is loaded in. We check every cell, check if it's valid, and return true if it hits the end of the loop. If one value is not valid, we return false, therefore the board is not valid.

```python
def find_next_empty_space(self):
    '''
    function to find the next empty space (0) in the board
    loops through each value of the board
    if a 0 is found, return the position (row, column)
    if no empty space is found, return None (means board is full)
    '''
    for i in range(9):
        for j in range(9):
            if self.board[i][j] == 0:
                return (i, j)  # row, column
    return None
```

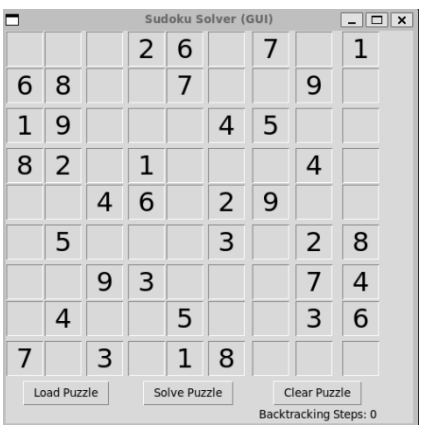
find_next_empty_space is just a helper function to find the next space on the board (or value 0). If we don't find an empty space, then we return None (as the board is full).

```python
def solve_sudoku(self):
        You, 21 hours ago • Uncommitted changes
    find_space = self.find_next_empty_space() # find the next empty space
    if not find_space:
        return True
    else:
        row, col = find_space # get the row and column of the empty space

    for num in range(1, 10): # loop through numbers 1-9
        if self.check_valid(num, row, col): # if the number is valid in that position
            self.board[row][col] = num # place the number there


            if self.solve_sudoku(): # recursively call the function to try and solve the rest of the puzzle
                return True

            self.board[row][col] = 0 # if not, reset the position to 0 (backtrack)
            self.backTrackCount += 1 # increment backtrack counter
    return False # if no number is valid in that position, return false (means puzzle is unsolvable)
```

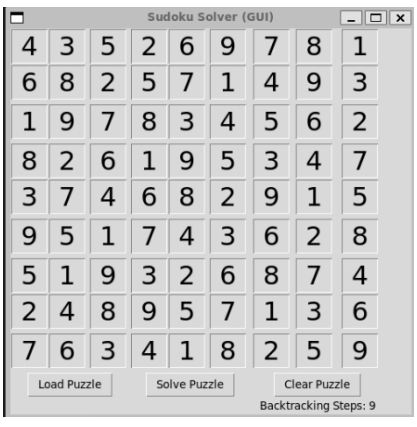
Here is the solve_sudoku function, which defines the logic. We find the next empty space and return the coordinates. If we don't find an empty space, then the puzzle is completed, so return True. Then, we loop through numbers 1-9, checking at each value if the number is valid. If it is valid, then we update the value to the number from the loop. Then we have a recursive call to try and solve the puzzle. If we have solved the puzzle, then we break the call and return True. If we haven't solved the puzzle yet, then we reset the position to 0 (meaning we backtrack) and increment the counter. If we try all numbers and none work for this cell, then we return False. The puzzle is unsolvable as there must be a valid number if the puzzle was correct.



II. GUI on startup.



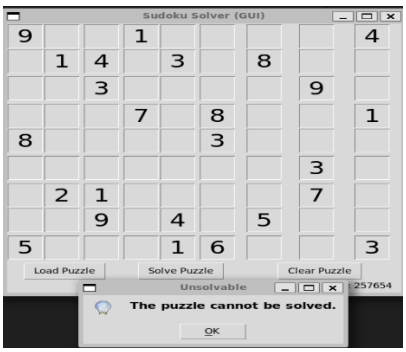
III. GUI w/ puzzle loaded.

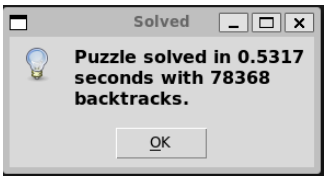


I. GUI w/ finished puzzle.

IV. GUI, different puzzle, initial board invalid

V. GUI, different puzzle, board is unsolvable.



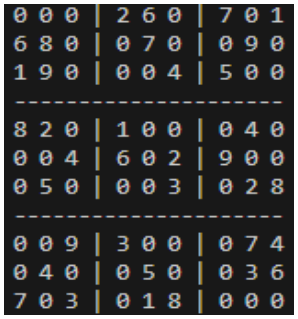


VI. Dialog box showing time completion and number of backtracks.

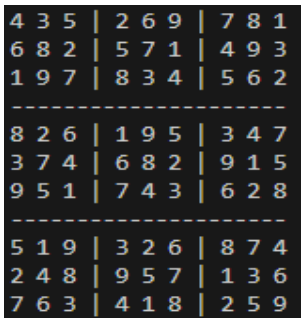
```
Puzzle validated in: 0.000211 seconds
Solve started at 17:22:15.443811
```

VII. Above and below, terminal outputs (time taken)

```
Solution found in: 0.000341 seconds
Solution completed at 17:22:15.444437
```



IX. Terminal outpu (unsolved puzzl



VIII. Terminal output (solved puzzle)

Operation	Time complexity	Explanation
Reading all values into 2D list	$O(n^2)$	All values of each cell must be accessed.
Printing all values	$O(n^2)$	All values of each cell must be accessed.
Checking one value is valid	$O(n)$	Iterations across row and column take O(n) time.
Checking the whole board is valid	$O(n^3)$	O(n) operation on every single value of board.
Trying a value in a cell	$O(n^2)$	O(n) operation to try a value, and on a singular value.

Puzzle File	Given cells	Empty cells	Backtracks	Time	Error Message	Source of puzzle
puzzle1.csv	36	45	9	0.0004s	N/A	(University of Arizona, 2006)
puzzle2.csv	24	57	2193	0.0146s	N/A	(University of Arizona, 2006)
puzzle3.csv	19	62	78368	0.4801s	N/A	(University of Arizona, 2006)
puzzle4.csv	0	81	310	0.0021s	N/A	
puzzle5.csv	81	0	0	0.0001s	N/A	(University of Arizona, 2006)
puzzle6.csv	20	61	980671	6.0750s	N/A	(crackingthecryptic.com, 2021)
puzzle7.csv	29	52	39776	0.2527s	N/A	(grantm, 2020)
notvalid.csv	24	57	N/A	N/A	“Puzzle invalid”	(sudopedia, 2007)
notvalid2.csv	25	56	N/A	N/A	“Puzzle unsolvable”	(sudopedia, 2007)

Our implementation does include a GUI, but all it does is just provide a visual aid. Although our implementation is fully functional, it’s inefficient. Currently, the algorithm selects the first available empty cell and sequentially tries numbers, but that could mean that we explore unnecessary branches. If we implemented a heuristic that prioritises cells with the fewest possible options, then we could make it more efficient. We could also implement forward checking (Russell & Norvig, 2020), which is when we look ahead immediately after assigning a value to a variable. If you place a value into a 5, forward checking will remove 5 from the possible values of all cells in the row, column, and sub grid. This means it could stop the program going through a path that is impossible. Both strategies have the same concepts as an A* search. If we incorporate heuristics, then it would explore far fewer possibilities. For harder puzzles, this would make it more efficient, however for easier Sudoku puzzles, it would prove to be more inefficient. While in principle A* search can be applied to Sudoku, A* is not necessarily well adapted to constraint satisfaction problems in its nature. A* applies heuristic evaluation functions to guide search to one best goal, which is to de-prioritise or even ignore other relevant constraints. Sudoku, however, requires satisfying all the constraints equally, rather than optimising one metric. Backtracking, by always maintaining constraint validity, is thus a better method for solving Sudoku. A* may become feasible by building a heuristic that anticipates constraint violations, but such heuristics are computationally expensive and likely to be inadmissible or non-optimal.

Part 2: Automated Planning

2A – Modelling the domain

To create predicates, actions, and objects, we must figure out what the mission aims to accomplish, in this case, exploring the moon using a lander and rover.

From the specifications, we gather the following statements.

- Rovers must be deployed before any actions can be performed
- Rovers can only hold one sample, and one piece of data at a time, to free up space for more samples or data, these must be dropped off at a lander or transmitted remotely.
- Landers may land at any location, but will remain stationary after landing, and cannot take off and land at a new location.

Due to the open-ended nature of the requirements, assumptions about the world and goal have been made:

- If a rover and lander have not been deployed, or given a location, then the planner will select an optimal location for the lander to land.
- There are no restrictions on how many "items" may occupy a specific way point, for example, 2 landers, a rover, and a sample may exist at one location.
- Each time a rover must drop off or transmit data, it is always done with the lander the rover initially came from, resulting in a 1-1 relationship between landers and rovers
- All data and samples must be transmitted or dropped at a lander to be marked as complete

This allows us to create the following types, actions, and predicates

Type	Description
Parent type – Item Child types – Lander, Rover, Sample	Represents physical objects within the world, such as landers, rovers, and samples
Parent type – Data Child types – Scan, Picture	Represents digital objects within the world, that is collected by the rover, whenever required
Location	Locations which the rover can traverse between

Predicate	Parameters	Description
at	?i - Item ?l1 - location	The item is at a specific location
connected	?l1 - location1 ?l2 - location2	There is a path from location 1 to location2
deployed	?r - rover	The rover is currently deployed
notDeployed	?r - rover	The rover is not currently deployed
hasSample	?r - rover ?s - sample	The rover currently contains a sample
noSample	?r - rover	The rover currently does not contain a sample
dropped	?s - sample	The sample has been delivered to a lander
hasData	?r - rover ?d - data	The rover contains a piece of data
noData	?r - rover	The rover does not contain a piece of data
uploaded	?d - data	The piece of data has been uploaded to a lander
dataAt	?d - data ?l1 - location	The piece of data is at this location
landed	?l - lander?l1 - location	The location a lander has landed at
notLanded	?l - lander	The lander has not yet landed at a location
linked	?l - lander ?r - rover	The link between a rover and its lander

Action	Parameters	Preconditions	Effects	Description
Move (rover)	?r - rover ?l1 - location ?l2 - location	(at ?r ?l1) (connected ?l1 ?l2) (deployed ?r)	(not (at ?r ?l1)) (at ?r ?l2)	The rover moves from its old location to a new location
Pickup (rover)	?r - rover ?s - sample ?l1 - location	(at ?r ?l1) (at ?s ?l1) (deployed ?r) (noSample ?r)	(hasSample ?r ?s) (not (at ?s ?l1)) (not (noSample ?r))	The rover pickups a sample from the location, the sample is removed from map
Drop (rover)	?r - rover ?l - lander ?s - sample ?l1 - location)	(at ?r ?l1) (landed ?l ?l1) (deployed ?r) (hasSample ?r ?s) (linked ?l ?r)	(dropped ?s) (noSample ?r)	The rover drops off a sample at its lander, and is now free to pick up another sample
GetData (rover)	?r - rover ?d - data ?l1 - location	(at ?r ?l1) (dataAt ?d ?l1) (deployed ?r) (noData ?r)	(hasData ?r ?d) (not (dataAt ?d ?l1)) (not (noData ?r))	The rover collects a piece of data from the location, and the data is removed from the map
SendData (rover)	?r - rover ?l - lander ?d - data	(deployed ?r) (hasData ?r ?d)(linked ?l ?r)	(uploaded ?d) (noData ?r)	The data is uploaded to its lander, and the rover is now free to collect more data
land (lander)	?l - lander ?r - rover ?l1 - location	(notLanded ?l) (linked ?l ?r)	(not (notLanded ?l)) (landed ?l ?l1) (at ?l ?l1) (notDeployed ?r)	The lander lands at a location
DeployRover (lander)	?l - lander ?r - rover ?l1 - location	(notDeployed ?r) (landed ?l ?l1) (linked ?l ?r)	(landed ?r ?l1) (deployed ?r) (noData ?r) (noSample ?r) (not (notDeployed ?r))	The lander deploys the rover, with an initially empty memory and cargo hold

2B: Problem Setup

For mission 1-3, the PDDL problems were evaluated with “BWFS –FF-Parser version” in VSCode with the PDDL extension by Jan Dolejsi

Each problem is composed of the following components

- Objects – The items and concepts which exists in the world
- Initial states – facts about the world
- Goal states – tasks that need to be completed

Mission1	Mission2
Objects: - Locations : wp1 - wp5 - Samples: sample1 - Data: scan1, pic1 - Lander - lander1 - Rover – rover1 Initial States: - Waypoints connected as per the specification - Rover linked to lander, and starts undeployed - locations of data points and samples Goals: - All data, and samples are uploaded and collected	Objects: - Locations: wp1 – wp6 - Samples: sample1 and sample2 - Data: scan1-2, pic1-2 - Landers: lander1 and lander2 - Rovers: rover1 and rover2 Initial States: - Waypoints connected as per the specification - Rovers linked to landers - Rover1 and lander1 deployed at wp2 - Rover2 is still waiting to land - Locations of datapoints and samples Goals: - All data and samples are uploaded and collected

2C: Domain extensions

In part 2C, 2 astronauts are added to the mission, which introduces new types, predicates, preconditions and actions, extending the functionality of the domain used in 2A

The following assumptions were made:

- Astronauts remain stationed within the lander at all times
- 1-1 relationship between astronauts and landers (and transitively 1 rover)

New feature or changes	Description
Type - astronaut	A child type of item, as astronauts are physical entities
Types – db, cr	A child type of location, represents the docking bay and control room of landers
Predicate - astronautLink (?a - astronaut ?l - lander)	Links an astronaut to a lander, required to prevent unintended rover-lander interactions
Action - astronautMove	Identical to the “move” action, but takes an astronaut type instead of a rover,
Renamed action – move -> roverMove	Readability change
Actions – deployRover, drop, sendData	Takes 2 additional parameters, ?a - astronaut, and ?l2 - cr/db* within the lander, adds 2 additional preconditions to check the correct link between landers and astronauts (astronautLink ?a ?l), and that the astronaut is in the correct place before action can be completed (at ?a ?l2) *cr for sendData, db for drop and deployRover

Mission 3 – a revised version of mission 2 with changes or new features added
New objects: - Astronauts: Bob and Alice - Locations: internal locations within the lander cr1-2, db1-2 New initial states: - Connections between internal areas of landers - Astronaut links to respective landers Goals remain unchanged

Appendix

.csv files

puzzle1.csv -
0,0,0,2,6,0,7,0,1,6,8,0,0,7,0,0,9,0,1,9,0,0,0,4,5,0,0,8,2,0,1,0,0,0,4,0,0,0,4,6,0,2,9,0,0,0,5,0,0,0,3,0,2,8,0,0,9,3,0,0,0,7,4
,0,4,0,0,5,0,0,3,6,7,0,3,0,1,8,0,0,0

puzzle2.csv -
0,2,0,6,0,8,0,0,0,5,8,0,0,0,9,7,0,0,0,0,0,0,4,0,0,0,0,3,7,0,0,0,0,5,0,0,6,0,0,0,0,0,0,4,0,0,8,0,0,0,0,1,3,0,0,0,0,2,0,0,0,0,0
,0,0,9,8,0,0,0,3,6,0,0,0,3,0,6,0,9,0

puzzle3.csv -
0,2,0,0,0,0,0,0,0,0,0,6,0,0,0,0,3,0,7,4,0,8,0,0,0,0,0,0,0,0,0,3,0,0,2,0,8,0,0,4,0,0,1,0,6,0,0,5,0,0,0,0,0,0,0,0,0,1,0,7,8,0
,5,0,0,0,0,9,0,0,0,0,0,0,0,0,0,0,4,0

puzzle4.csv -
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

puzzle5.csv -
2,7,6,3,1,4,9,5,8,8,5,4,9,6,2,7,1,3,9,1,3,8,7,5,2,6,4,4,6,8,1,2,7,3,9,5,5,9,7,4,3,8,6,2,1,1,3,2,5,9,6,4,8,7,3,2,5,7,8,9,1,4,6
,6,4,1,2,5,3,8,7,9,7,8,9,6,4,1,5,3,2

puzzle6.csv -
0,0,0,1,0,2,0,0,0,0,6,0,0,0,0,0,7,0,0,0,8,0,0,0,9,0,0,4,0,0,0,0,0,0,0,3,0,5,0,0,0,7,0,0,0,2,0,0,0,8,0,0,0,1,0,0,9,0,0,0,8,0,5,0,7,0,0,0,0,0,6,0,0,0,0,3,0,4,0,0,0 (Known as the hardest Sudoku puzzle in the world)

puzzle7.csv -
9,0,0,2,0,0,7,0,1,0,0,1,0,7,0,9,0,0,0,0,0,0,1,9,0,5,0,5,0,0,0,0,0,2,0,0,0,2,6,0,9,0,5,7,0,0,0,4,0,0,0,0,0,8,0,3,0,4,6,0,0,0,0,0,0,7,0,2,0,3,0,0,4,0,9,0,0,3,0,0,7 (code is 01367a7605d1 on website)

notvalid.csv -
6,0,1,5,9,0,0,0,0,0,9,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,4,0,7,0,3,1,4,0,0,6,0,2,4,0,0,0,0,0,5,0,0,3,0,0,0,0,1,0,0,0,6,0,0,0,0,0,3,0,0,0,9,0,2,0,4,0,0,0,0,0,0,1,6,0,0

notvalid2.csv -
9,0,0,1,0,0,0,0,4,0,1,4,0,3,0,8,0,0,0,0,3,0,0,0,0,9,0,0,0,0,7,0,8,0,0,1,8,0,0,0,0,3,0,0,0,0,0,0,0,0,0,0,3,0,0,2,1,0,0,0,0,7,0,0,0,9,0,4,0,5,0,0,5,0,0,0,0,1,6,0,0,3

References

crackingthecryptic.com. (2021, August 15). *"Steering Wheel"*. Retrieved from Sudoku Solver: https://www.youtube.com/watch?v=Ui1hrp7rovw & http://sudokupad.app/7gJb9G8fRt

grantm. (2020). *sudoku-exchange-puzzle-bank.* Retrieved from Github: https://github.com/grantm/sudoku-exchange-puzzle-bank

kartik. (2025, July 23). *A* Search Algorithm*. Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/dsa/a-search-algorithm/

Learn-Sudoku. (2008, December 31). *Sudoku Rules & Object of the game.* Retrieved from learn-sudoku.com: https://www.learn-sudoku.com/sudoku-rules.html

Python. (2025, October 16). *tkinter- Python interface to Tcl/Tk*. Retrieved from docs.python: https://docs.python.org/3/library/tkinter.html

Russell, S. J., & Norvig, P. (2020). *Artificial Intelligence: A Moden Approach (4th Edition).* Prentice Hall.

sudopedia. (2007, October 11). *Invalid Test Cases*. Retrieved from sudopedia.enjoysudoku: http://sudopedia.enjoysudoku.com/Invalid_Test_Cases.html

University of Arizona. (2006, January 20). *Example Puzzles and Solutions*. Retrieved from arizona.edu: https://sandiway.arizona.edu/sudoku/examples.html

Video Link

https://heriotwatt-my.sharepoint.com/personal/or2008_hw_ac_uk/_layouts/15/stream.aspx?id=%2Fpersonal%2For2008%5Fhw%5Fac%5Fuk%2FDocuments%2FG14%20%2D%20F29AI%20CW1%20DEMO%2Emp4&nav=eyJyZWZlcnJhbEluZm8iOnsicmVmZXJyYWxBcHAiOiJPbmVEcml2ZUZvckJ1c2luZXNzIiwicmVmZXJyYWxBcHBQbGF0Zm9ybSI6IldlYiIsInJlZmVycmFsVmlldyI6InZpZXciLCJyZWZlcnJhbFZpZXciOiJNeUZpbGVzTGlua0NvcHkifX0&ga=1&referrer=StreamWebApp%2EWeb&referrerScenario=AddressBarCopied%2Eview%2Ec29a633d%2Dd5ac%2D4f87%2D83d1%2D68d634472b76