# **Artificial intelligent coursework 1**

Edinburgh CW1 70

Jugal Kevin Koppalakonda – H00520191

Weihan Yap – H00524171

## **Part 1A**

Q. Define Sudoku formally as a constraint satisfaction problem.

CSP = (X, D, C)

* X = Set of variables – 9x9 grid (81 cells)
* D = Set of domains – possible values for the variables
* C = Set of constraints – Restrict which values can be places on the variable (cell)

This means Sudoku can be used as a constraint satisfaction problem (Yashkochar, 2024, Sep 18)

Q. What are the variables, domains and constraints?

A constraint satisfaction problem is defined by 3 components: Variable, Domains and Constraints.

1. **Variables** - Sudoku consists of a 9x9 guid therefore each of the 81 cells are considered variables
2. **Domain** - The domain is the set of possible values a cell can take. This means in sudoku, each cell can take a number from 1 to 9. The domain is restricted to a single number because a cell can only hold 1 number.
3. **Constraints** - The rules are that each number cannot repeat in any row, column, or block.
   1. Row constraints
   2. Column constraints
   3. Block constraints (3x3 guid)

Q. Discuss the time complexity of brute-force search vs. backtracking in Sudoku

Brute-force search tries all possible solution until the correct one is found. This means all 81 cells will be checked with all possible combination of numbers from 1-9 that satisfy the rules of: each number cannot repeat in any row, column, or block.

The number of completed guid: 9^81

Big-O: O((N^2)^N^4)

* Sub-grid – n x n
* Whole guid – n^2 x n^2
* Each variable can take one of n^2 values
* For each n^4, each with n^2 choices: (n^2)n^4 = (3^2)3^4) = 9^81

Backtracking search tries values step by step and at every step, it checks weather the current run break sudoku’s rules. If a solution is invalid, then it stops and does not explore that path further. This means that it can be faster than brute force because it doesn’t waste time exploring invalid solution.

Big-O (worst-case) is still (O(9^81)). However, average case Backtracking search is better. (GeeksforGeeks, 2012, July 14)

## **Part 1B**

### **Code Breakdown**

def validity(grid, r, c, k):

not\_in\_row = k not in grid[r]

not\_in\_column = k not in [grid[i][c] for i in range(9)]

not\_in\_box = k not in [grid[i][j] for i in range(r // 3 \* 3, r // 3 \* 3 + 3)

for j in range(c // 3 \* 3, c // 3 \* 3 + 3)]

return not\_in\_row and not\_in\_column and not\_in\_box

The validity() function has three main rules -

1. not\_in\_row – a number “k” does not appear in row “r”
2. not\_in\_column – a number “k” does not appear in column “c”
3. not\_in\_box – a number “k” does not appear in the box containing(r,c)

All of these rules must be True in order for the position to be valid.

validity() function returns True if all the conditions are valid, else it returns false.

(GeeksforGeeks. (2012, July 14))

#Find the cell with the fewest possible valid numbers

def find\_unassigned(grid):

best\_cell = None

min\_poss = 10 # higher than 9

for r in range(9):

for c in range(9):

if grid[r][c] == 0:

poss = [k for k in range(1, 10) if validity(grid, r, c, k)]

if len(poss) < min\_poss:

min\_poss = len(poss)

best\_cell = (r, c, poss)

return best\_cell

The function find\_unassigned() implements the minimum required heuristic approach to select the next empty place to fill(the empty cell with the least number of valid options). This function takes the current grid as input and returns a tuple with rows ‘r’ and columns ‘c’ and possibilities for every cell with the least number of possibilities. It will return None if the grid is full.

The function scans all the cells and for each empty cell, calculates a list of possible values using the validity() function. It tracks the cell with the minimum number of possibilities(min\_pos)

#Solve Sudoku using backtracking

def solve(grid):

global steps

steps += 1

cell = find\_unassigned(grid)

if not cell:

return True # returns true when solved

r, c, possibilities = cell

for k in possibilities:

if validity(grid, r, c, k):

grid[r][c] = k

if solve(grid):

return True

grid[r][c] = 0 # backtracks until

return False

The function solve() is a recursive function that solves the Sudoku using MRV and backtracking.

It takes the current grid as input. Initially the function increments the step counter each time called in order to track the search effort. It uses the find\_unassigned() function to find the next cell to fill. It runs recursively until there are no cells left. For every iteration it checks whether a number is fit in that position and fills it in that position. It recursively calls the solve() function on the updated grid. If the recursive call is True, it means the puzzle is solved. If not, it will set the value back to 0 and try next possibility. This is the part where backtracking comes in. If all the possibilities fail, the function returns False.

### **Comparison with A\***

A\* algorithm finds the shortest path to a goal, whereas the implemented solver fills the grid satisfying all the constraints, hence it is called CSP. The key difference between A\* and CSP is how they explore the solution space. **(Kumar, R. 2024, Nov 7)**

CSP(the solver made for sudoku), fills the hardest empty spaces first, reducing the useless exploration, it uses forward checking to prune the impossible paths while backtracking to fill the empty spaces with correct numbers.

A\* treats the entire board as a state while searching, it expands more boards with cells being filled, it prioritises the “most closest solution” as it uses a heuristic-

f(n) = g(n) + h(n)

Although both algorithms can be used to solve Sudoku, CSP is more practical as CSP has a time complexity of O(9^n) and a space complexity of O(n) while A\* has worse time and space complexities compared to CSP. Both algorithms can be used to solve sudoku but CSP is more efficient as it leverages constraints, uses minimal memory and is perfect for pure constraint problems. **(Kynoch, B, 2017, Jun 30).**

For example, there are 40 empty cells, CSP prunes down to a small subset of cells at each step, quickly eliminating dead ends. While A\* might spend huge memory and time expanding all intermediate steps which might not lead to the correct substitution especially if the heuristic is poor.

A\* algorithm is better for pathfinding, puzzles with fewer constraints, while CSP is better for a problem which has a lot of constraints.

(Kumar, R. (2024, November 7))

(Kynoch, B. (2017, June 30))

## **Part 2A & 2B**

### 2.1 Domain structure

#### **Types**

* Rover – used to move between waypoint, collect sample, take images and scan surface. And then transmit or store it in lander.
* Lander – used to deploy rover and store sample and collect transmitted data
* Waypoint – location on the moon where data and sample are located
* Sample – need to be collected by rover
* Image and scan – need to be taken by rover

#### **Predicates**

The predicates describe the relationship between objects.

*Location and deployment-* This shows where rovers and landers are located, and which waypoints are connected

* (rover\_at ?r - rover ?w - waypoint)
* (lander\_at ?l - lander ?w - waypoint)
* (undeployed ?r - rover)
* (connected ?from - waypoint ?to - waypoint)

*Sample/data handling -* This defines what types of data exist at each waypoint

* (image\_at ?w - waypoint)
* (scan\_at ?w - waypoint)
* (sample\_at ?w - waypoint)

*Rover memory/storage -* This hold track of the rover’s memory capacity and what data or sample it currently holds.

* (empty\_memory ?r - rover)
* (holding\_image ?r - rover)
* (holding\_scan ?r - rover)
* (holding\_sample ?r - rover)

*Transmission and storage result-* This indicate that the image/scan/sample has been successfully transmitted or stored.

* (stored\_sample ?l - lander)
* (transmit\_scan ?r - rover ?w - waypoint)
* (transmit\_image ?r - rover ?w - waypoint)

*Landing status -* This shows if a lander has landed before deploying the rover.

* (has\_landed ?l)

#### **Action**

|  |  |  |
| --- | --- | --- |
| Action | Purpose | Effects |
| Land\_lander | Land a lander | Land rover at waypoint |
| Deploy\_rover | Deploy rover from lander | Rover becomes active at waypoint |
| Move | Move rover between connected waypoint | Rover moves to waypoint |
| Collect-image | Capture image | Rover holds image |
| Collect-scan | Scan surface | Rover holds scan |
| Transmit\_image | Transmit captured image | Image transmitted and rover memory is freed |
| Transmits\_scan | Transmit scanned surface | Scan transmitted and rover memory is freed |
| Collect\_sample | Collect a sample | Rover hold sample |
| Store\_sample | Store sample at lander | Sample stored in lander and rover sample space is freed. |

### 2.2 Problem

Problem 1:

Initial state:

* Lander is at wp3
* Lander has landed
* Rover is undeployed
* Rover’s memory is empty
* Waypoint is connected according to Figure 2
* Sample is at waypoint 1
* Image is at waypoint 5
* Scan is at waypoint 3

Goal state:

* Transmit image from waypoint 5
* Transmit scan from waypoint 3
* Store sample at lander

Problem 2:

Initial state:

* Lander 1 is at waypoint 2
* Rover 1 is at waypoint 2
* Rover 1’s memory is empty
* Lander 2 has not landed
* Rover 2 is undeployed
* Rover 2’s memory is empty
* Waypoint is connected according to Figure 3
* One sample at wp1
* One sample at wp5
* One image at wp2
* One image at wp3
* One scan is at wp4
* One scan is at wp6

Goal State:

* Rover 1 will transmit image at wp3
* Rover 1 will transmit scan at wp4
* Rover 2 will transmit image at wp2
* Rover 2 will transmit scan at wp6
* Store sample at lander 1
* Store sample at lander 2

### 2.3 Testing and Results

Both mission 1 and 2 generated a successful plan using Best First Search planners: “BFWS –FF-parser version”. It included rover deployment, movement between waypoint, memory management and image/scan/sample collection.

Best First Width search combines Heuristic search and width-based exploration, which is also called novelty search (Francès et al., n.d.). A heuristic search takes an educated guess to guide the search towards the goal instead of exploring all possibilities (GeeksforGeeks, 2024, May 27). A width-based algorithm explores new actions to avoid wasting time exploring the same kind of situations repeatedly (Lipovetzky, N., n.d.).

BFWS can find solution more efficiently by combining heuristic search with exploring new and different states.

## **Part 2C**

### 2.1.1 Domain Structure

#### Types

* Astronaut – human being used to interact with lander and rover
* Area –
  + Control rooms – used by astronauts to collect transmitted data.
  + Docking bay – used by astronauts to deploy rover and collect samples.

#### Predicates

Astronaut and Area – This keeps track of where astronauts is.

* (astronaut\_at ?a - astronaut ?l - lander ?ar - area)
* (in\_control\_room ?ar - area)
* (in\_docking\_bay ?ar - area)

#### Action

|  |  |  |
| --- | --- | --- |
| Action | Purpose | Effects |
| Move\_Astronaut | Move astronaut between areas within the lander – control room and docking bay. | Astronaut is able to retrieve sample (in docking bay) and collect transmitted data (in control room). |

|  |  |  |
| --- | --- | --- |
| Action | Changes | Effects |
| Deploy\_rover | Added astronaut and area into account | Astronaut will need to be in docking bay to deploy rover. |
| Transmit\_image | Added astronaut and area into account | Astronaut will need to be in control room to receive transmitted image |
| Transmit\_scan | Added astronaut and area into account | Astronaut will need to be in control room to receive transmitted scan |
| Store\_sample | Added astronaut and area into account | Astronaut will need to be in docking bay to receive sample from rover. |

### 2.2.1 Problem

Same as mission 2

### 2.3 Testing and Results

Mission 3 generated a successful plan. It involved astronauts in rover deployment and image/scan transmission and sample collection.

Find the link to the demonstration - <https://1drv.ms/v/c/165782A97D8DE9AB/EeSRnR0VYBdKtEXEwTTZAHkBNTvcVVHU8z-KFNGuSSVzYw>

## Appendix

Output 1

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a black screen

AI-generated content may be incorrect.

Output 2

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

Output 3

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

## **References**

Francès, G., Geffner, H., Lipovetzky, N., & Ramírez, M. (n.d.). *Best-First Width Search in the IPC 2018: Complete, Simulated, and Polynomial Variants*. <https://ai.dmi.unibas.ch/papers/frances-et-al-ipc2018.pdf>

GeeksforGeeks. (2012, July 14). *Sudoku Solver*. GeeksforGeeks. <https://www.geeksforgeeks.org/dsa/sudoku-backtracking-7/#naive-approach-using-backtracking>

GeeksforGeeks. (2017, May 3). *Best First Search (Informed Search)*. GeeksforGeeks. <https://www.geeksforgeeks.org/dsa/best-first-search-informed-search/>

[4] GeeksforGeeks. (2023, June 8). *Constraint Satisfaction Problems (CSP) in Artificial Intelligence*. GeeksforGeeks. <https://www.geeksforgeeks.org/artificial-intelligence/constraint-satisfaction-problems-csp-in-artificial-intelligence/>

GeeksforGeeks. (2024, May 27). *Heuristic Search Techniques in AI*. GeeksforGeeks. <https://www.geeksforgeeks.org/artificial-intelligence/heuristic-search-techniques-in-ai/>

Kumar, R. (2024, November 7). *The A\* Algorithm: A Complete Guide*. Datacamp.com; DataCamp. <https://www.datacamp.com/tutorial/a-star-algorithm>

Kynoch, B. (2017, June 30). *What is the time complexity of A\* search*. Stack Overflow. <https://stackoverflow.com/questions/44849517/what-is-the-time-complexity-of-a-search>

Lipovetzky, N. (n.d.). *Planning for Novelty: Width-Based Algorithms for Common Problems in Control, Planning and Reinforcement Learning*. <https://www.ijcai.org/proceedings/2021/0702.pdf>

Yashkochar. (2024, September 18). *Solving Sudoku as a Constraint Satisfaction Problem (CSP)*. Medium. <https://medium.com/@yashkochar01/solving-sudoku-as-a-constraint-satisfaction-problem-csp-54cb553c3cab>