Final Project Report

Foundations of Artificial Intelligence (CS 5100)
Multi-Sudoku Constraint Satisfaction Problem

Yun Cheng (001798530)
Dishant Kapadiya (001681610)
Shraddha Satish Thumsi (001681742)

1. Problem description

We are solving four forms of multi-Sudoku, with 2 sub-puzzles, 3 sub-puzzles, 4 sub-puzzles, and 5 sub-puzzles, as shown below:

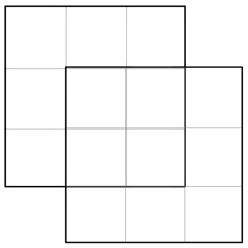
Figure 1: 2x2 Multi-Sudoku

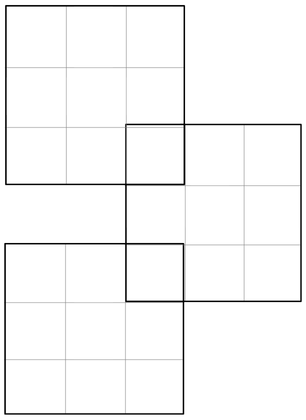
Figure 2: 3x3 Multi-Sudoku

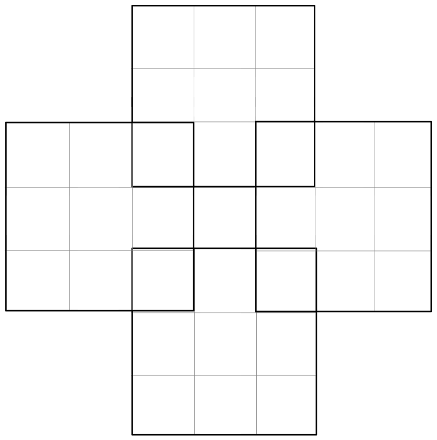
Figure 3: Sohei-Sudoku

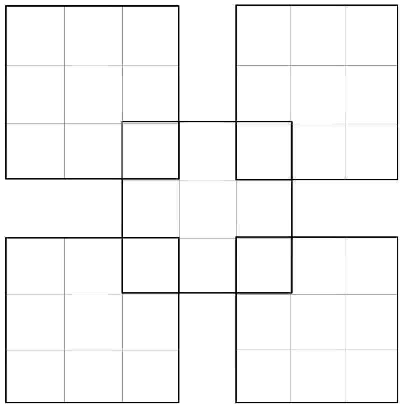
Figure 4: Samurai-Sudoku

Input: string of 81 characters per line, each representing a regular Sudoku sub-puzzle, with empty squares indicated using a '.' character. Multiple lines form a multi-Sudoku that is a combination of those sub-puzzles.

Output: solution to a multi-Sudoku puzzle

2. Algorithms

The algorithms/heuristics used to solve the multi-Sudoku Constraint Satisfaction Problem (CSP) include depth first search (DFS), forward checking, constraint propagation, minimum remaining values heuristic, and maximum degree heuristic.

The multi-Sudoku CSP is defined as follows:
Variables: each square in the multi-Sudoku puzzle
Domains: {1,2, 3, 4, 5, 6, 7, 8, 9}
Constraints: 9-way alldiff for each row, 9-way alldiff for each column, 9-way alldiff for each box of each sub-puzzle.

We define a collection of nine squares (column, row, or box) of a sub-puzzle a *unit* and the squares that share a unit the *peers*. Thus, depending on the form of multi-Sudoku, the different shape of the puzzle will result in different number of peers for a given square because of the overlaps. Once the peers are defined correctly for each of the different multi-Sudoku game types (2x2 Multi-Sudoku grid, 3x3 Multi-Sudoku grid, Sohei-Sudoku grid, and Samurai Sudoku grid), the same search algorithms used to find a solution apply for all game types.

Forward checking

The search for a solution utilizes forward checking, where each time a node is expanded, the domains of the other unassigned variables are updated accordingly. As the definition of the problem states, unfilled squares each start out with the numbers 1 to 9 in their domains. Following the alldiff constraints of each sub-puzzle of a multi-Sudoku puzzle, the algorithm will eliminate the possible numbers from the domains of squares until only a single number remains for each square. In Python, the squares are represented in a key-value pair dictionary, where the keys are the square's location on the grid of each 9 by 9 sub-puzzle and the values are the domains of the squares, represented by a string of numbers.

Constraint propagation

Forward checking can only catch conflicts right before they cause a certain branch to fail. To detect errors earlier, we make use of constraint propagation, where we assign values that only have one possible choice immediately, and then allow the constraints to propagate to peers recursively:

I. If a square has only one possible value, then eliminate that value from the square's peers.
II. If a unit has only one possible place for a value, then put the value there.

With constraint propagation, we fail conflicting branches sooner, so that the number of squares to be searched is typically much fewer than the total number of unfilled squares before we strike upon a valid solution.

Depth first search

To search for a solution, we use depth first search and search one unfilled square at a time. Using the Minimum Remaining Values heuristic, we choose an unfilled square with the minimum size domain first. For that square, we choose a value from that square's domain. Then following statement I, we eliminate that value from the square's peers. If, during this elimination process we find that either statement I or II is true, we continue to propagate these constraints to the relevant squares. If no conflicts arise, then we choose another square using MRV, and repeat. Each square keeps track of a copy of the current puzzle up to that point. At any point where there is a conflict, the recursive algorithm returns false and we go back to a square that has a non-conflicting puzzle. We know we have solved the puzzle once the domain of each square is reduced to length 1, i.e. once we have defined a value for each square without conflicts.

Minimum remaining values heuristic

The Minimum Remaining Values heuristic chooses the square with the smallest domain. This heuristic works because it picks a variable that most likely to cause a failure soon; thus the trees can be pruned early.

Maximum degree heuristic

To break tie situations in the MRV heuristic where two or more squares have the same number of remaining values, we choose the square with the most unassigned peers. This cuts down on the number of legal successor states to it, resulting in a performance improvement of nearly double on average.

Value ordering

Once we have chosen a square using MRV and degree heuristic, the order of the values we choose from that square can also have an effect on performance. Initially, we attempted to choose the least constraining value, leaving maximum flexibility for subsequent variable assignments. This LCV is found by looping through the square's values and peers, and counting how many peers have that value in their domains. Then the value with the smallest number of counts is the LCV. However, after testing against performance without the LCV heuristic, it was determined that the performance decreased. The benefits of LCV did not outweigh the intense computation required to determine the LCV, and hence the values were considered in numeric order instead.

3. Results

The results of this project are measured in the average time taken to solve a grid. In addition to that, we also consider the number of nodes expanded by the DFS search running behind. In order to understand the result, we need to understand what affects the performance. The factors that influence performance are as follows:

I. Number of sub-puzzles
II. Number of squares overlapping
III. Difficulty of puzzle (number of unfilled squares)

Number of sub-puzzles essentially mean the number of normal 9x9 Sudoku grids in a whole puzzle. The greater the number of grids in a puzzle, more difficult it gets to solve. Performance is directly proportionally to the number of grids a puzzle has. In our cases, Samurai Sudoku is the toughest to crack while 2x2 Multi-Sudoku is the easiest one.

Moreover, the performance is also swayed by the number of overlapping squares. The squares that overlap are bound by constraints from all the grids they are associated with. Any change in them would propagate to all the grids they are part of. This increases the chances of backtracking during DFS and hence hampers performance.

Lastly, performance is also affected by the difficulty of the puzzle given. Difficulty here means the number of unfilled squares in the puzzle grid. These squares are indicated in the input using the '.' character. As the number of unknown increases, the level difficulty increases, which increases the number of nodes searched through DFS.

Keeping all these parameters in mind, we tested all the cases with few grids and found following results:

	Average time taken (s)	**Average nodes expanded**
2x2 Multi-Sudoku	34.01	223,431
3x3 Multi-Sudoku	7.82	71,697
Sohei-Sudoku	201.34	1,928,488
Samurai Sudoku	642.12	5,823,981

The grids were a mixture of easy, moderate and hard level grids which enables us to evaluate algorithm irrespective of the question. From the data, we can say that the performance would surely decrease as we raise the complexity of the Sudoku. In addition to that, notice that we have 2 overlaps in 3x3 puzzles as compared to others which have 4 overlaps. It clearly portrays that having more overlaps impacts performance greatly. However, the algorithm portrays completeness. It always finds a solution if it exists. For a solution to exist we need to take care of the puzzle given to solve. It should contain minimum of 17 squares filled in any given sub-puzzle with at least 8 different digits. Also, the overlapping and the consequent constraints, different for each type of puzzle, must be taken care of in the code.