**Homework 1 - Programming**

-Prasanth S Iyer

piyer@usc.edu

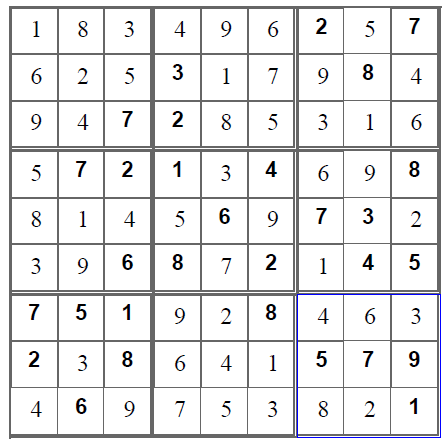
USC ID: 1958-5902-92

1. **Solving Sudoku with Backtracking in CSPs**

**Question 1 (5 pts)**

Solve the given Sudoku puzzle by hand.

**Solution:**

****

**Question 2 (10 pts)**

Formulate a Sudoku puzzle as a constraint satisfaction problem.

**Solution:**

Any n x n Sudoku puzzle can be formulated as a constraint satisfaction problem in the following way:-

**Variable Set:**There are n2 different variables in a Sudoku puzzle corresponding to each of the n2 cells in the Sudoku.

**∴** X = {X11, X12, ... Xnn}

**Domain:**

Each variable can take any value from 1 to n (n = 9 in a 9x9 Sudoku.) However, values of some variables are initially provided as part of the puzzle.

**∴**Dij = initial value(Xij) if value of variable Xij is initially specified.

= {1, 2, 3,..., n} otherwise.

**Constraints:**

There are 3 types of constraints in a normal Sudoku puzzle:-

1. **Row constraints:**

Each row must have distinct elements.

**∴**For any row i,

Xi1 ≠ Xi2 ≠ Xi3 ≠...≠ Xin

1. **Column constraints:**

Each column must have distinct elements.

**∴**For any column j,

X1j ≠ X2j ≠ X3j ≠...≠ Xnj

1. **Block constraints:**

Each √n x √n square must have distinct elements.

**∴**For any square with left-top corner: {p√n + 1, q√n + 1}

and right-bottom corner: {p√n + √n, q√n + √n} where 0 ≤ p ≤ √n - 1 and 0 ≤ q ≤ √n - 1

X (p√n + 1) (q√n + 1) ≠ X(p√n + 1) (q√n + 2) ≠...≠ X(p√n + 1) (q√n + √n) ≠ X(p√n + 2) (q√n + 1) ≠...≠ X(p√n + √n) (q√n + √n)

**Goal:** Assign a value to each variable such that all constraints are satisfied.

**Question 3 (40 pts)**

Write a Java or C++ program to solve a Sudoku puzzle using backtracking search for CSPs.

**Solution:**

I have implemented a Java program - SudokuSolver.java to solve the Sudoku puzzle.

**Compilation Syntax:** javac SudokuSolver.java

**Command-line Usage:** java SudokuSolver <input\_file> <output\_file>

* **Assumptions:**

I made the following assumptions while implementing the Sudoku Solver:-

1. I assume that the input file will contain a valid n x n Sudoku where n is a perfect square.
2. I further assume that all lines in the input file will be of size n where n is the size of the Sudoku.
3. I also assume that the Sudoku has a single solution. If it has multiple solutions, the program will output any one of the possible solutions.

* **Implementation Details:**

1. *Variable Set:* 
   1. Each variable is implemented as a cell consisting of the row and column indices, the value currently set for that cell and a boolean variable that signals if the value of that cell has already been determined.
   2. The Sudoku Grid is implemented as a n x n matrix of cells.
2. *Constraints:*

The constraints are implemented using 3 bit vectors:-

* 1. **Row vector:** A vector of n bits for storing the list of available values for each row.
  2. **Column vector:** A vector of n bits for storing the list of available values for each column.
  3. **Block vector:** A vector of n bits for storing the list of available values for each

√n x √n block.

The ith bit in each vector determines if the value i has already been used in that row/column/block or if it is still available for use. In order to check the values that can be assigned to a particular cell, one simply needs to check the common bits among the corresponding row, column and block vectors for that particular cell.

1. *Domain:*

The domain of each variable is checked with the constraints.

1. *Goal Test:*

The goal is reached whenever values are assigned to each cell without violating any of the constraints.

1. *Backtracking:*

Backtracking was implemented using a simple recursive algorithm:-

solveSudoku(Cell currentCell)

1. For each set of possible values from 1...n
   1. Assign a value i to currentCell
   2. If setting currentCell to i violates a constraint, continue the loop with next possible value. Otherwise, remove the value i from the corresponding row, column and block vector.
   3. nextCell = getNextCell()
   4. If solveSudoku(nextCell) is successful, return success.
   5. Add back the value i to the corresponding row, column and block vector and continue with next iteration of the loop.
2. If none of the values work for the current cell, return failure.
3. *Get next cell to be assigned a value:*

Since we do not use any heuristic to get the next cell to be assigned a value, values are assigned in the order:

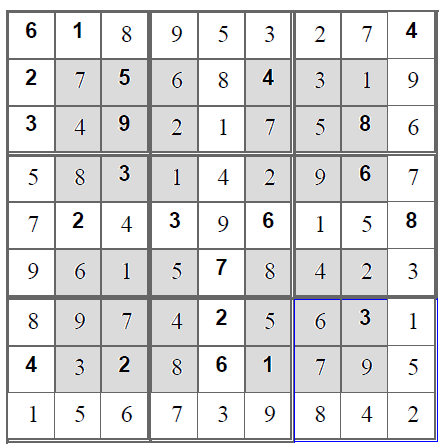
Cells[0][0], Cells[0][1], ... , Cells[0][n-1],...Cells[n-1][n-1].

While choosing the next cell, we ignore values that have been initially set up from the input file.

1. **Solving Sudoku with Backtracking in CSPs**

**Question 4 (5 pts)**

Solve the given Hyper Sudoku puzzle by hand.



**Question 5 (5 pts)**

Formulate a Hyper Sudoku puzzle as a constraint satisfaction problem

Any n x n Hyper Sudoku puzzle can be formulated as a constraint satisfaction problem in the following way:-

**Variable Set:**There are n2 different variables in a Hyper Sudoku puzzle corresponding to each of the n2 cells in the Hyper Sudoku.

**∴** X = {X11, X12, ... Xnn}

**Domain:**

Each variable can take any value from 1 to n (n = 9 in a 9x9 Hyper Sudoku.) However, values of some variables are initially provided as part of the puzzle.

**∴**Dij = initial value(Xij) if value of variable Xij is initially specified.

= {1, 2, 3,..., n} otherwise.

**Constraints:**

There are 4 types of constraints in a Hyper Sudoku puzzle:-

1. **Row constraints:**

Each row must have distinct elements.

**∴**For any row i,

Xi1 ≠ Xi2 ≠ Xi3 ≠...≠ Xin

1. **Column constraints:**

Each column must have distinct elements.

**∴**For any column j,

X1j ≠ X2j ≠ X3j ≠...≠ Xnj

1. **Block constraints:**

Each √n x √n square must have distinct elements.

**∴**For any square with left-top corner: {p√n + 1, q√n + 1}

and right-bottom corner: {p√n + √n, q√n + √n} where 0 ≤ p ≤ √n - 1 and 0 ≤ q ≤ √n - 1

X (p√n + 1) (q√n + 1) ≠ X(p√n + 1) (q√n + 2) ≠...≠ X(p√n + 1) (q√n + √n) ≠ X(p√n + 2) (q√n + 1) ≠...≠ X(p√n + √n) (q√n + √n)

1. **Hyper Block constraints:**

Each √n x √n hyper square must have distinct elements. Now, based on my analysis of Hyper Sudokus, it can be mathematically proven that an n x n Sudoku will have at most (√n - 1)2 hyper squares.

**∴**For any hyper square with left-top corner: {p√n + r, q√n + c}

and right-bottom corner: {p√n + √n + r - 1, q√n + √n + c - 1}

where 0 ≤ p ≤ √n - 1, 0 ≤ q ≤ √n - 1

and r = (p + 2)%(√n + 1), c = (q + 2)%(√n + 1)

X (p√n + r) (q√n + c) ≠ X(p√n + r) (q√n + c + 1) ≠...≠ X(p√n + r) (q√n + √n + c - 1) ≠ X(p√n + r + 1) (q√n + c) ≠...≠

X(p√n + √n + r - 1) (q√n + √n + c - 1)

**Goal:** Assign a value to each variable such that all constraints are satisfied.

**Question 6 (20 pts)**

Write a Java or C++ program to solve a Hyper Sudoku puzzle using backtracking search for CSPs.

**Solution:**

I have implemented a Java program - HyperSudokuSolver.java to solve the Hyper Sudoku puzzle.

**Compilation Syntax:** javac HyperSudokuSolver.java

**Command-line Usage:** java HyperSudokuSolver <input\_file> <output\_file>

* **Assumptions:**

I made the following assumptions while implementing the Sudoku Solver:-

1. I assume that the input file will contain a valid n x n Hyper Sudoku where n is a perfect square.
2. I further assume that all lines in the input file will be of size n where n is the size of the Hyper Sudoku.
3. I also assume that the Hyper Sudoku has a single solution. If it has multiple solutions, the program will output any one of the possible solutions.

* **Implementation Details:**

1. The implementation is exactly the same as the Sudoku solver with the only difference being an additional bit vector added to represent the hyper block constraint.
   1. **Hyper Block vector:** A vector of n bits for storing the list of available values for each √n x √n hyper block. Note that we need only (√n - 1)2 such vectors as the Hyper Sudoku can have at most (√n - 1)2 hyper squares.

The checking of the hyper block vector only needs to be done for cells that belong to a particular hyper block. It can easily be seen that any Cell[row][col] for which either:

* + 1. row%(√n + 1) = 0 or
    2. col%(√n + 1) = 0

will not belong to a hyper block. All other cells will belong to exactly one hyper block.

As before, the ith bit in each vector determines if the value i has already been used in that row/column/block/hyper block or if it is still available for use. In order to check the values that can be assigned to a particular cell, one simply needs to check the common bits among the corresponding row, column, block and hyper block vectors for that particular cell.

1. *Get next cell to be assigned a value:*

Once again, since we do not use any heuristic to get the next cell to be assigned a value, values are assigned in the order:

Cells[0][0], Cells[0][1], ... , Cells[0][n-1],...Cells[n-1][n-1].

While choosing the next cell, we ignore values that have been initially set up from the input file. I highlight this for the Hyper Sudoku Solver as this is the only difference between the Hyper Sudoku Solver and the Hyper Sudoku Solver with the Minimum Remaining Value Heuristic.

**Question 7 (15 pts)**

Discuss how the following heuristics can improve the performance of your backtracking search in

solving a Hyper Sudoku puzzle:

* **Minimum remaining values (MRV) heuristic**

The main idea of the MRV heuristic is to choose the variable with the fewest legal values that satisfy all constraints since this is most likely to cause a failure and thereby prune the search tree.

In the Hyper Sudoku puzzle, the MRV heuristic can be used in choosing the next cell to be assigned a value. Instead of picking a random cell, we simply pick the cell that has the fewest possible values remaining.

In my implementation, we can simply pick the cell which has the fewest common bits in its row, column, block and hyper block vector. The possible performance improvements that this heuristic offers are:-

1. When the algorithm is expanding an incorrect node, as soon as the constraints are propagated such that there are no remaining values for a particular cell, that is the cell that will be the next selected for assigning values. The algorithm would immediately find no possible values for this cell and consequently, no further values are assigned to the remaining cells. The algorithm simply prunes the remaining subtree and backtracks to the previous legal position.
2. Since at each point, the algorithm chooses the cell with the fewest legal values possible, it is much more likely to cause a failure by expanding fewer nodes i.e. earlier than random selection.
3. Since at each point, the algorithm chooses the cell with the fewest legal values possible, it is much more likely to arrive at a correct outcome by expanding fewer nodes than random selection.

* **Degree heuristic**

The main idea of the degree heuristic is to choose the variable that is involved in the most number of constraints with other unassigned variables as this would reduce the branching factor on future choices.

In the Hyper Sudoku puzzle, the degree heuristic can be used along with the MRV heuristic in choosing the next cell to be assigned a value as a tie-breaker for the MRV heuristic. Instead of picking a random cell from among a set of cells with the fewest possible values remaining, we simply pick the cell involved in the most number of constraints with other unassigned cells.

Note that the degree heuristic by itself is not that efficient as the MRV heuristic, but, it can definitely make the MRV heuristic better if used as a tie-breaker.

In my implementation, we can find the number of cells involved in constraints with a particular cell by simply finding the number of unassigned cells in the row, column, block and hyper block of that cell. We can then pick the set of cells which have the fewest common bits in their row, column, block and hyper block vectors (MRV) and then use the number of unassigned cells in their respective rows, columns, blocks and hyper blocks as a tie-breaker amongst them. The possible performance improvement that this heuristic offers is:-

When the algorithm is expanding an incorrect node, since the current cell constrains the most number of unassigned cells, the number of possible values for the unassigned cells is minimum. Consequently, when used with MRV, it is more likely to fail after expanding fewer number of nodes than a simple MRV algorithm as it leaves fewer choices for the remaining unassigned variables.

* **Least constraining value heuristic**

The main idea of the least constraining value heuristic is after choosing a variable, to choose the value that would rule out the least number of choices for the remaining unassigned variables as this would provide the most chance of finding a successful solution.

In the Hyper Sudoku puzzle, every assigned value always rules out at most one (or none if it is already ruled out) of the choices for the cells in its row, column, block and hyper block. So, we can choose a value that is already ruled out for the most number of cells in its row, column, block and hyper block as this would still leave the maximum number of choices for these cells.

In my implementation, we can use the MRV heuristic along with the degree heuristic as a tie-breaker to choose the next cell to assign a value to. Once this is done, we pick a legal value for this cell such that it appears in the domain of the least number of cells in its row, column, block and hyper block. *(Remember that the domain of a cell could be found in my implementation by finding the common bits in its row, column, block and hyper block vectors.)*

The possible performance improvement that this heuristic offers is:-

Given that MRV and degree heuristics work well in pruning a large part of the Hyper Sudoku's incorrect subtrees, the least constraining value heuristic will, given a variable with minimum remaining values, provide the best chance of finding a probable solution by choosing a value that rules out the least number of values for cells in its row, column, block and hyper block. Consequently, it is more likely to find a solution by expanding fewer nodes than an algorithm that only uses a combination of MRV and degree heuristics.

**Extra Credit (10 pts)**

Choose at least one heuristic listed in Question 4 and implement it.

**Solution:**

I have implemented a Java program - HyperSudokuSolverHeuristic.java to solve the Hyper Sudoku puzzle with the minimum remaining value (MRV) heuristic.

**Compilation Syntax:** javac HyperSudokuSolverHeuristic.java

**Command-line Usage:** java HyperSudokuSolverHeuristic <input\_file> <output\_file>

* **Implementation Details:**

1. The implementation is exactly the same as the Hyper Sudoku solver with the only difference being in how the next cell to be assigned a value is chosen.
2. *Get next cell to be assigned a value:*

I used the MRV heuristic to get the next cell to be assigned a value. I do this by checking the domain of all unassigned cells and returning the cell with smallest available domain. I implemented this in the following way:-

1. I found the size of the domain of each of the n2 cells by checking the number of common bits in its row, column, block and hyper block vectors.
2. I then checked if a cell has been assigned a value by maintaining the boolean variable stored for each cell.
3. Finding the cell with the smallest available domain was then done by a simple linear scan of the domain of all unassigned cells.

Note that this would be an overhead of O(n2) each time we choose a cell to assign a variable to. However, we do end up expanding far fewer nodes in return as my experiments suggested.

Show the comparison between the heuristic and non-heuristic in your report. Report the number of nodes expanded before the solution is reached. Run an experiment and report some quantitative results.

I tested the Hyper Sudoku solver with and without heuristic using multiple Hyper Sudokus of varying difficulties from the online site: http://www.sudoku-space.com/hyper-sudoku and the results are as follows. The difficulty here is based on the difficulty provided on the site. Further all tests were performed under similar conditions on the same machine.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Size** | **Number of given values** | **Difficulty** | **Total time taken to solve (in ms)** | | **Number of nodes expanded** | |
|  |  |  | **With MRV heuristic** | **Without heuristic** | **With MRV heuristic** | **Without heuristic** |
| 9 x 9 | 22 | Easy | 33 | 35 | 102 | 3002 |
| 9 x 9 | 23 | Easy | 37 | 22 | 143 | 2261 |
| 9 x 9 | 22 | Easy | 178 | 182 | 1961 | 49064 |
| 9 x 9 | 23 | Medium | 27 | 14 | 58 | 1596 |
| 9 x 9 | 23 | Medium | 41 | 22 | 154 | 2421 |
| 9 x 9 | 23 | Medium | 30 | 22 | 90 | 2277 |
| 9 x 9 | 22 | Hard | 38 | 171 | 137 | 26231 |
| 9 x 9 | 20 | Hard | 32 | 17 | 74 | 1851 |
| 9 x 9 | 24 | Hard | 35 | 156 | 115 | 25851 |

One of the interesting things that one can clearly notice is the fact that although the MRV heuristic expands much fewer nodes in all cases, it does take time comparable to the approach without the heuristic. One main reason for this is the fact that finding the cell with the minimum remaining value takes O(n2) time for an n x n Sudoku with the linear scan approach that I have implemented and this is done for each expanded node. One could use smarter data structures like heaps to ensure that this time is reduced.

However, in terms of number of nodes expanded, it is clear that the MRV approach provides a huge improvement as was expected.