271 Final Project Report
Sudoku Solvers

Hao Ni 90562952
Rui Liu 10283095
Sijie Yu 40484388
Yue Ding 20809476

Problem Definition

Sudoku is one of the most famous constrained satisfaction problems. In a Sudoku game, the player is given a grid containing 9 × 9 cells, or nine 3 × 3 sub-grids. Some of the cells are pre-filled with specified numbers, and the rest are empty. The goal is to fill out all the empty cells of the given grid under the constraints that each row, column and sub-grid contains an exact permutation of 1 to 9.

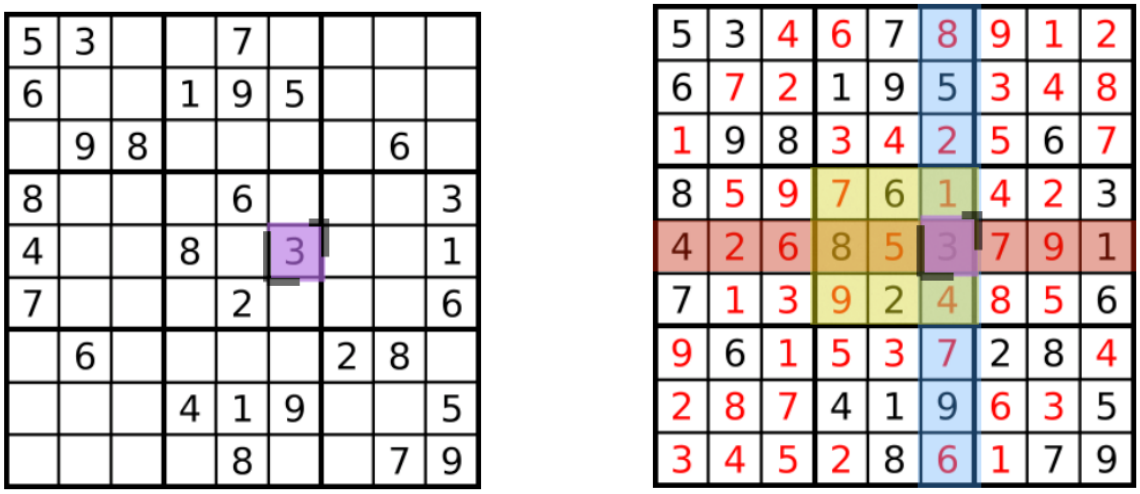

Fig 1. Illustration of Sudoku game

Variables of the game are the eighty-one cells, and the domain includes integers from 1 to 9. In other words, each variable have the choices of being filled with a number from one to nine. Constraints apply on choosing which number to fill out a certain empty cell. For each row, each column and each sub-grid, we can only have numbers from 1 to 9 without duplicates. In other words, for each row, by filling out empty cells, we will get a row of numbers ranging from 1 to 9, without duplicates. Similar constraints apply to each column and sub-grid.

Existing Sudoku solving algorithm

Various methods have been investigated in the literature, most of which are based on backtracking. So the basic backtracking method is discussed first, followed by improvements based on it.

Backtracking
A simple backtracking algorithm to solve Sudoku is as follows: we pick in a particular variable and value order, assign a particular value to a variable, if a conflict is found during this assignment, we will backtrack to the previous assignment and try a different value.

Randomness can be introduced in both variable and the value order, which avoids the extreme situation that ends up exploring all possible assignments. But overall the backtracking is very time-consuming.

Improvements are based on methods that can help prune the unnecessary branches as soon as possible and reduce the search space:

1. Forward Checking. At each assignment, we check if the variable affected are still left with values to assign. If not, the current assignment must be invalid, and we can backtrack immediately.
2. Arc-Consistency. This is a much stronger dead-end cutter than forward checking but very inefficient. The complete arc-consistency means whenever a variable is assigned, check the arc-consistency of the variables that are affected by this assignment. Keep adding the affected nodes to the checklist and check arc-consistency for those as well. If any conflict is found, backtrack occurs.
3. Most constrained variable. Pick the variable that has the most number of constraints. In Sudoku it is the one with the minimum size of the domain.
4. Most constraining value. Pick the variable that will constrain the least number of nodes. In Sudoku it is the value that will cause the smallest number of nodes to shrink their domain size.
5. Stochastic local search. By local search, we basically define a heuristic function and set an initial randomized state. The heuristic function is used to calculate the distance from the current state to the goal and evaluate the values for the next possible neighbor state and progress to the next favorable state. For example, a simple heuristic function is to calculate the number of conflicts existing in the grid.The efficiency of this algorithm depends on the how accurately the heuristic function predicts its distance to the final goal. The algorithm may also lead to a dead loop where the algorithm stuck in the local minimum. A randomized state transfer can be introduced to make it occasionally jumps out of the loop and explore some other paths.

These methods helps reduce search space, resulting in a smarter way of picking variables and values. The overhead, however can dominate in some cases, leads to worse performance against basic naive backtrack, which will be seen in later sections.

Project implementation

In this project, two methods are implemented for comparisons. The first one is the naive backtrack, in which the variable and value ordering is the natural order. The other is using forward checking with minimum remaining values heuristic.

Before the details of two methods are discussed, the implementation for game itself is shown in the class design diagrams below.

As the sudoku problem is based on a grid, which is a 9 × 9 matrix of cells, we need the classes to represent the grid and each cell.

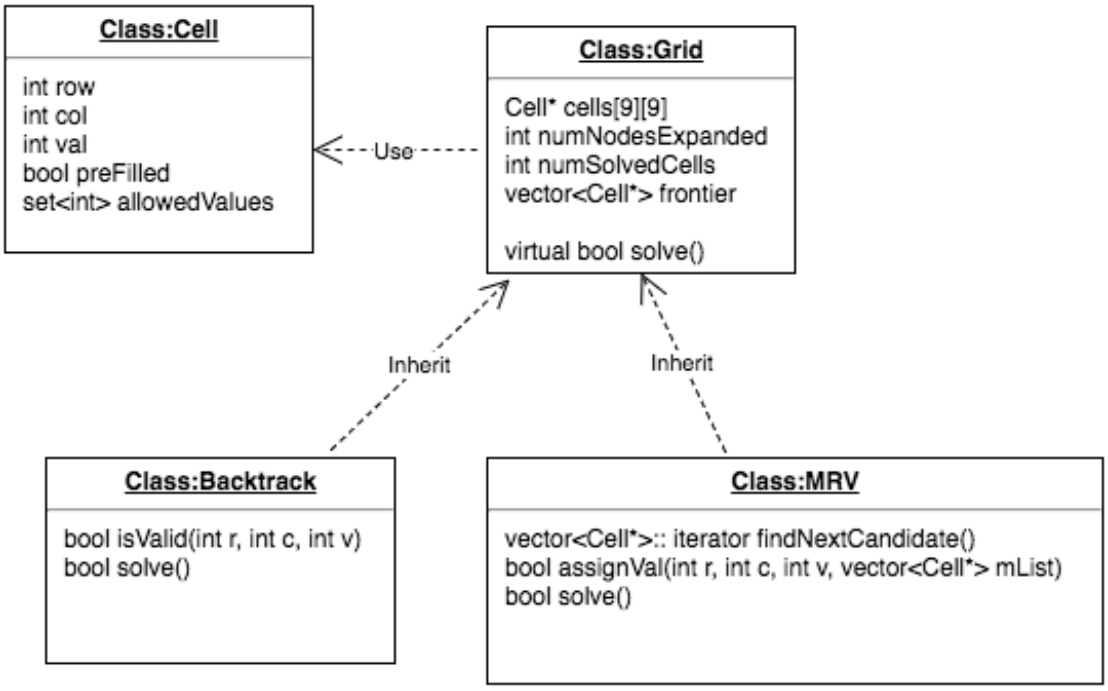

Fig 2. Class design

Class Cell:
row, col: defines the location of the cell
val: the value assigned to this cell: 0~9, 0 means it's empty
prefilled: whether this cell is pre-filled
allowedValues: legal values that can be assigned to this cell

Class Grid:
cells: matrix of cells
frontier: collection of variables to be explored
numNodeExpanded/ numSolvedCells: self explained
solve: virtual function, to be implemented by its inherited classes

Classes of two methods:

1. Naive backtrack

The code snippet is shown below. Equivalently all the cells are inside the frontier and examined one by one. For each cell, each value from 1 to 9 is checked with isValid() function, which will check the same row, same column and same sub-grid to see if there is any conflict.

```
bool solveFrom(int row, int col) {
   if (numSolvedCell == TOTAL_CELL) return true;
   if (cells[row][col]->preFilled)
      return solveFrom(col == GRID_DIM - 1 ? row + 1 :
         row, col == GRID_DIM - 1 ? 0 : col + 1);

  for (int val = 1; val <= 9; val++)
      if (isValid(row, col, val)) {
         cells[row][col]->value = val;
         if(solveFrom(col == GRID_DIM - 1 ? row + 1 :
            row, col == GRID_DIM - 1 ? 0 : col + 1))
            return true;
      cells[row][col]->value = Cell::EMPTY_VALUE;
         numSolvedCell--;
      }
   return false;
}
```

Fig 3. code snippet for naive backtrack

2. Forward checking with minimum remaining values heuristic

First of all, we don't need to check the illegal values, so the domain of each cell need to be adjusted after a cell is filled with a value. As for variable ordering, the intuition is that we should accommodate the variables with fewest legal values.

An example is shown below. The cell with fewest legal value is assigned with 9. All conflicting cells have to remove 9 from their domains. These modified cells are also stored in a list for backtrack purpose.

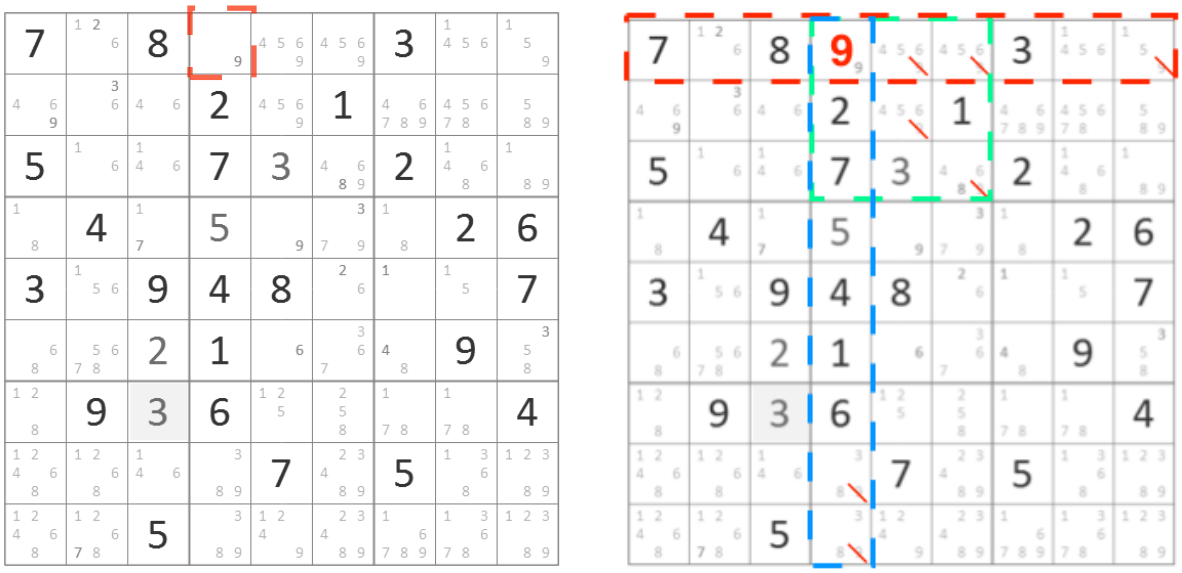

Fig 4. illustration of minimum remaining values heuristic

The code snippet of the forward checking with MRV heuristic is shown below.

```
bool solve() {
   if (frontier.size() == 0)
      return true;
   std::vector<Cell*>::iterator curCell_itr = findNextCandidate();
   Cell* curCell = * curCell_itr;
   frontier.erase(curCell_itr);
   std::set<int> possibleValues = curCell->allowedValues;
   for (std::set<int>::iterator it = possibleValues.begin();
         it != possibleValues.end(); it++) {
      int curVal = *it;
      std::vector<Cell*> modifiedCells;
      if(!assignCell(curCell, curVal, modifiedCells))
         continue;
      if (solve())
         return true;
      for (int i = 0; i < modifiedCells.size(); i++)
         modifiedCells[i]->allowedValues.insert(curVal);
   }
   curCell->value = 0;
   frontier.push_back(curCell);
   return false;
}
```

Fig 5. code of forward checking with MRV

Testing results

The testing is carried out with a Macbook Pro with following specs:
CPU: 2.2 GHz Intel Core i7
Memory: 16 GB 1600 MHz DDR3

To test our method over different difficulties of puzzles, 50 easy and 95 hard puzzles are obtained from http://norvig.com/sudoku.html. The execution time of both methods to solve easy and hard puzzles are shown below. The red series is the time taken by naive backtrack while the blue series is FCMRV.

As expected, in general the naive backtrack takes much longer than the FCMRV method. The MRV heuristic helps reduce number of unnecessary nodes significantly and save tremendous amount of time. However, the overhead of forward checking, namely reducing conflicting cell's domain and reverting them during backtrack can dominate the execution time when the input is such a puzzle that leads to relatively low number of backtrack.

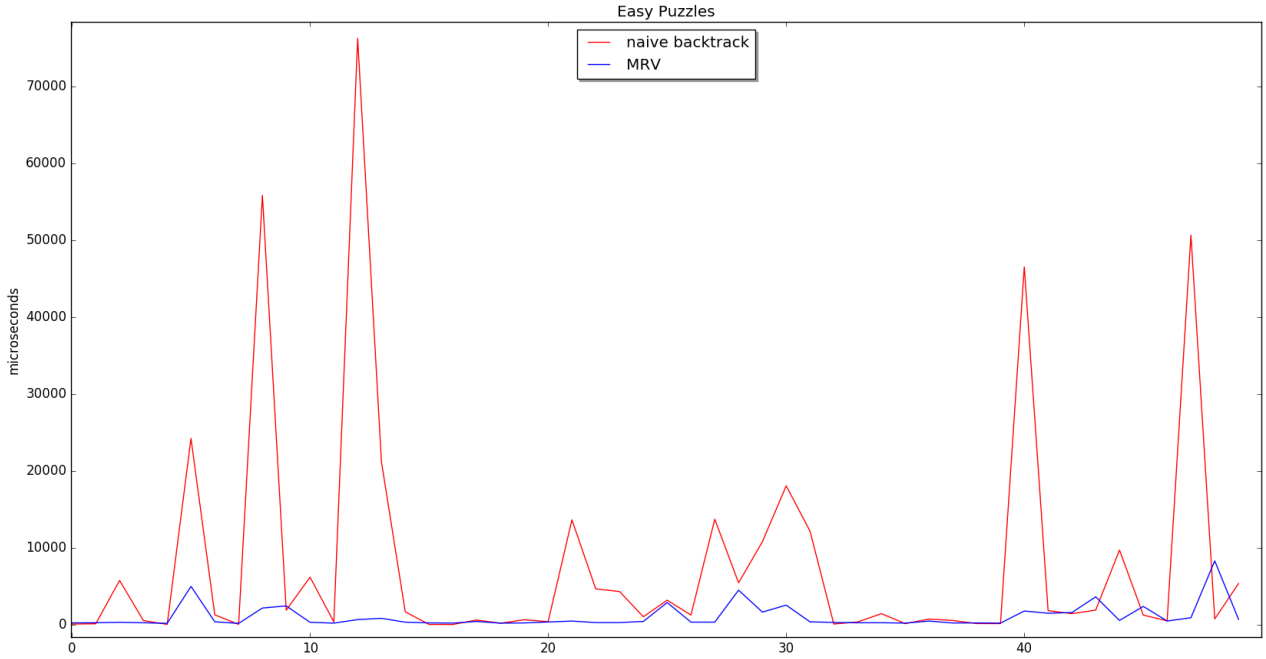

Fig 6. execution time of easy puzzles naive backtrack vs FCMRV

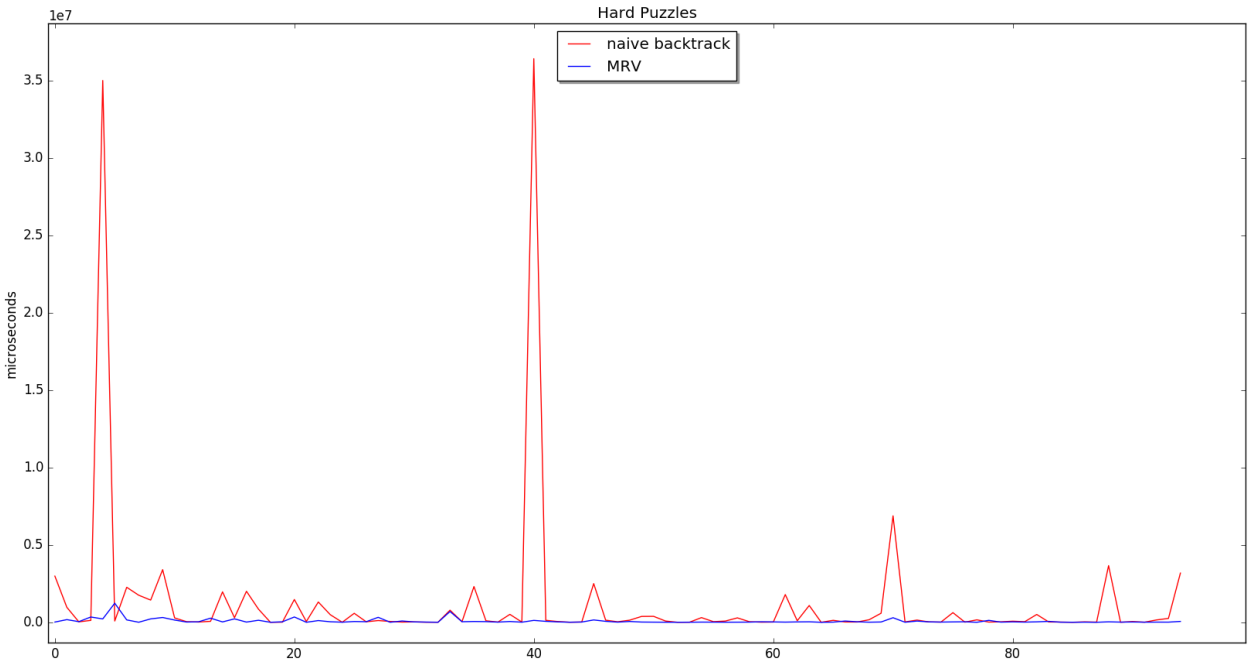

Fig 7. execution time of hard puzzles naive backtrack vs FCMRV

Conclusion

Two sudoku solvers are implemented using naive backtrack and forward checking with minimum remaining value heuristic in C++. Both solvers can solve the problem of various difficulty successfully, yet with different efficiency. Generally speaking, for the easy puzzles, the naive backtrack can outperform FCMRV, because the overhead of reducing domain and reverting cells dominate in FCMRV. However, as the puzzle gets more complicated, the number of nodes expanded and backtrack increases drastically in naive backtrack, which leads to poor performance compared to FCMRV.

Future improvements can include other heuristics. In addition, a min-heap can be used as the frontier to find the cell with minimum number of legal values to reduce time complexity from O(n) to O(1). This overhead reduction could lead to the result that FCMRV can out perform naive backtrack in all cases.

Appendix

The algorithm statistics for given benchmarks is attached:

Naive backtrack

	#node	**#backtrack**	**execution time(us)**
easy	324	278	126
medium1	1973	1924	553
medium2	839	789	277
hard1	918	864	376
hard2	1680	1626	523
evil1	9793	9737	3163
evil2	76341	76285	26462
evil3	1367	1311	578

FCMRV

	#node	#backtrack	execution time(us)
easy	65	19	317
medium1	60	11	337
medium2	61	11	387
hard1	54	0	262
hard2	57	3	292
evil1	1042	986	6635
evil2	439	383	2665
evil3	56	0	267