Maksym Turkot
11/27/22

Constraint Satisfaction Problem Solver for Sudoku Puzzles

Introduction

CSPs are a very important subset of search problems in AI because their versatility allows many applications and the amount of research has made the solvers very efficient. This project implements one of such solvers designed specifically for solving Sudoku puzzles. I will run the program on the same set of puzzles three times using plain depth-first search algorithm (DFS), AC-3 preprocessing with DFS, and again utilizing forward-checking algorithm. I will then compare the performance results of each algorithm.

This report includes the following sections:

Program Design

Project was developed using Java programming language. The following pages provide a detailed description of each class that builds up the program. This section may be glanced over for familiarity with the program structure. Class diagram is provided in Figure 1.

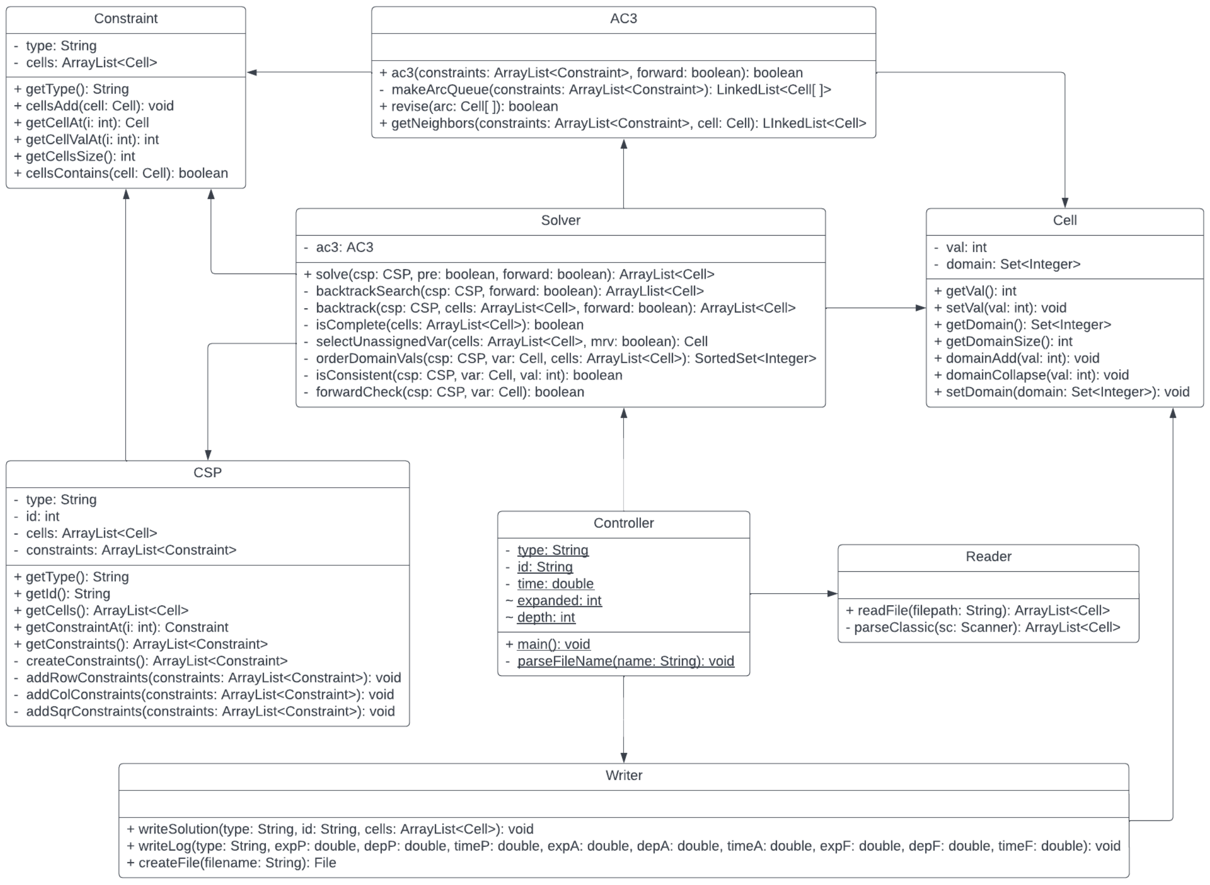

Constraint
- type: String
- cells: ArrayList<Cell>
+ getType(): String
+ cellsAdd(cell: Cell): void
+ getCellAt(i: int): Cell
+ getCellValAt(i: int): int
+ getCellsSize(): int
+ cellsContains(cell: Cell): boolean
AC3
+ ac3(constraints: ArrayList<Constraint>, forward: boolean): boolean
- makeArcQueue(constraints: ArrayList<Constraint>): LinkedList<Cell[ ]>
+ revise(arc: Cell[ ]): boolean
+ getNeighbors(constraints: ArrayList<Constraint>, cell: Cell): LInkedList<Cell>
Solver
- ac3: AC3
+ solve(csp: CSP, pre: boolean, forward: boolean): ArrayList<Cell>
- backtrackSearch(csp: CSP, forward: boolean): ArrayList<Cell>
- backtrack(csp: CSP, cells: ArrayList<Cell>, forward: boolean): ArrayList<Cell>
- isComplete(cells: ArrayList<Cell>): boolean
- selectUnassignedVar(cells: ArrayList<Cell>, mrv: boolean): Cell
- orderDomainVals(csp: CSP, var: Cell, cells: ArrayList<Cell>): SortedSet<Integer>
- isConsistent(csp: CSP, var: Cell, val: int): boolean
- forwardCheck(csp: CSP, var: Cell): boolean
Cell
- val: int
- domain: Set<Integer>
+ getVal(): int
+ setVal(val: int): void
+ getDomain(): Set<Integer>
+ getDomainSize(): int
+ domainAdd(val: int): void
+ domainCollapse(val: int): void
+ setDomain(domain: Set<Integer>): void
CSP
- type: String
- id: int
- cells: ArrayList<Cell>
- constraints: ArrayList<Constraint>
+ getType(): String
+ getId(): String
+ getCells(): ArrayList<Cell>
+ getConstraintAt(i: int): Constraint
+ getConstraints(): ArrayList<Constraint>
- createConstraints(): ArrayList<Constraint>
- addRowConstraints(constraints: ArrayList<Constraint>): void
- addColConstraints(constraints: ArrayList<Constraint>): void
- addSqrConstraints(constraints: ArrayList<Constraint>): void
Controller
- type: String
- id: String
- time: double
~ expanded: int
~ depth: int
+ main(): void
- parseFileName(name: String): void
Reader
+ readFile(filepath: String): ArrayList<Cell>
- parseClassic(sc: Scanner): ArrayList<Cell>
Writer
+ writeSolution(type: String, id: String, cells: ArrayList<Cell>): void
+ writeLog(type: String, expP: double, depP: double, timeP: double, expA: double, depA: double, timeA: double, expF: double, depF: double, timeF: double): void
+ createFile(filename: String): File

Figure 1. Class Diagram

Controller

Contains the **main()** method that starts the program execution, as well as a helper method **parseFileName()** that gets information about the puzzle from the file name.

main()

Creates the reader and writer objects and runs the program execution. First it iterates through the puzzle files in the puzzles folder and constructs the CSP problems for each puzzle. It then runs the solver in three configurations: plain DFS, with AC-3 preprocessing, and with forward checking. Each solution is written to a solution file, and execution data is written to a log file.

parseFileName()

Uses regex to split a filename and retrieve the type of the puzzle and its id.

Cell

Constructs the Cell datatype that represents the variable of the CSP. It has a value field, as well as domain set of possible values. Methods are simple getters an dsetters that allow for retrieval and manipulation of data in the domain.

Constraint

Constructs the constraint data type for the Sudoku puzzle. Constraints can be of two types: “alldiff” for most sudoku constraints, and “sum” for killer sudoku. Constraint contains a list of cells that participate in this constraint. Methods consist of getters and setters for data manipulation.

CSP

Constructs a data structure for a CSP problem, specifically, sudoku puzzle. It holds a list of cells (variables) and a list of constraints for the puzzle. Most methods are getters and setters for data manipulation, as well as methods for constraint construction.

createConstraints()
Runs the constraint creation process.

addRowConstraints()
Iterates through cells in a list and retrieves cells that are in the same sudoku row. Creates an “alldiff” constraint for each row in the puzzle.

addColConstraints()
Iterates through cells in a list and retrieves cells that are in the same sudoku column. Creates an “alldiff” constraint for each column in the puzzle.

addSqrConstraints()
Iterates through cells in a list and retrieves cells that are in the same sudoku square of nine cells. Creates an “alldiff” constraint for each relevant square in the puzzle.

Solver

Runs the generic CSP solution algorithm. At the core is the backtracking algorithm and helper methods for it. A separate forward checking method is used for the respective solver configuration.

solve()
Takes the CSP as a parameter, as well as “preprocess” and “forward” switches. If both switches are false, solver runs the plain BFS algorithm to solve the problem. If

"preprocess" is true, solver calls the AC-3 algorithm first. If AC-3 returns false, or the assignment is complete afterwards, solver returns immediately. Otherwise, it calls the backtracker. If "forward" flag is true, solver calls the backtracker with "forward" flag as well which runs the searcher in a forward checking mode.

backtrackSearch()
A wrapper method for the backtrack search algorithm.

backtrack()
The core searcher algorithm. Checks if assignment is complete. If not, picks a variable and tries all values in its domain. For each value, consistency with the constraints is checked, and the backtracker is called with the new assignment. At this stage, if solver is run in a forward checking mode, a forward check is called for the selected variable value, and if true, the backtracker is called. If backtracker returns cell assignment, this assignment is returned. Otherwise, different value for the variable is tried, and the process repeats.

isComplete()
Checks if there are unassigned cells in the puzzle.

selectUnassignedVar()
Runs the logic of selecting a variable for assignment. Default functionality selects the first empty value found. If MRV logic were used, this method would select a cell with the smallest domain.

orderDomainVals()
Orders the values of the domain for selection. By default, orders the elements in the natural ordering fashion. If LRV logic were used, it would place values that cause least constraint violation higher in the ordered set.

isConsistent()
Checks if the assignment is consistent with the picked value. Checks if any of the "alldiff" constraints already contains the value.

forwardCheck()
Runs the AC-3 preprocessing algorithm on the collection of constraints that include the current variable. Domains of each cell are saved and restored after the AC-3 processing. If a violation is found, method signals the BFS to backtrack immediately.

AC3

Implements the AC-3 arc validation algorithm.

ac3()
Creates a queue of binary constraints (arcs) from the “alldiff” constraints that were passed as a parameter. Each pair in the queue is popped and revised. If changes were made to a cell’s domain, new arcs containing this new cell and its neighbors are added to the queue. If at any point domain size of some cell falls to zero, algorithm signals inconsistency.

makeArcQueue()
Loops through the “alldiff” constraints and for each pair of cells generates two arcs (connecting two cells in both directions).

revise()
Iterates through the domains of both cells in an arc. If the domain of the second cell contains the value in the domain of the first cell, and the second cell has a single value in its domain (its value, consequently), this value is deleted from the domain of the first cell.

getNeighbors()
Finds constraints that contain a given cell, and returns all cells in these constraints, except for the cell itself. These are the neighbors of the cell in the constraint graph.

Reader

This class implements the reading features of the project.

readFile()
Reads the specified file by calling a necessary helper function depending on the puzzle type to store the information into the system.

parseClassic()
Parses the contents of a file with a classic Sudoku puzzle.

Writer

This class implements the reading functionality of the program.

writeSolution()
Writes the solution to a Sudoku puzzle to a file.

writeLogFile()
Writes program execution data to the log file, appending to the existing file.

createFile()
Creates a new file with a specified path. If the file already exists, it is overwritten.

User Manual

The entire program is in the turkotm folder. Source code is located in the src/main folder, tests are in the src/test folder, and data files are in the data folder.

Configuring the program

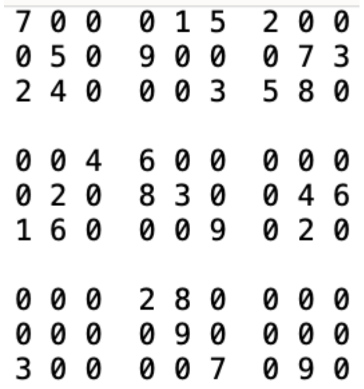
Program automatically reads puzzle files and figures out puzzle type based on the filename. It takes no input from the terminal.

Puzzles are located in the data/puzzles folder. The filename of the puzzle should have the following form:

<puzzleType>_<id>.txt
classic_3.txt

To have the program parse files correctly, puzzleType must spell "classic", "triple", or "killer". Only classic puzzles can be parsed at the moment.

The classic puzzle file must contain cell numbers written in order, separated by a whitespace. Order of numbers must be as they appear in the puzzle: left to right, top to bottom. Empty cells should be represented by a "0". Additional whitespace can be added for readability. Example:



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

0 0 4  6 0 0  0 0 0
0 2 0  8 3 0  0 4 6
1 6 0  0 0 9  0 2 0

0 0 0  2 8 0  0 0 0
0 0 0  0 9 0  0 0 0
3 0 0  0 0 7  0 9 0
```

Figure 2. Puzzle File

Running the program

Program was developed using Java.

To run the program, please make sure you are located in the turkotm folder. The Makefile is provided, so you can run the program using the following commands:

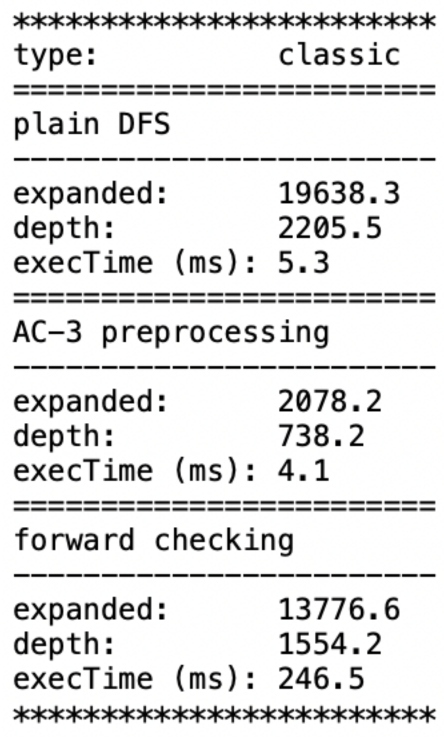
- **make** or **make compile** - compiles the program
- **make run** - runs the program
- **make clean** - removes compiled .class files.

When running, program will read puzzles from the folder, solve them, and write corresponding solutions to the "solutions" folder. Program execution data will be appended to the "log.txt" file.

Program Output

Solution file is very similar to the puzzle file, except that 0s are replaced with correct numbers, and filename includes a "-sol" suffix. Solution files are written to "data/solutions" folder.

Upon completion, program will record results in the "log.txt" file, located in the data folder. Each entry in the log file includes the aggregated runtime data for the solutions of one type of puzzles. The structure of the log file entry is as follows:



```
************************
type:          classic
========================
plain DFS
------------------------
expanded:      19638.3
depth:         2205.5
execTime (ms): 5.3
========================
AC-3 preprocessing
------------------------
expanded:      2078.2
depth:         738.2
execTime (ms): 4.1
========================
forward checking
------------------------
expanded:      13776.6
depth:         1554.2
execTime (ms): 246.5
************************
```

Figure 3. Log File

The file provides the following performance data for the puzzle solver:

- **expanded** - average number of variables that were considered during the search.
- **depth** - average number of backtracks during the search.
- **execTime (ms)** - average execution time per puzzle, in milliseconds.

Puzzle Encoding Design

Puzzle encoding

Each puzzle is encoded into a data structure called CSP. It contains two key fields that completely describe the puzzle:
- Cells (variables)
- Constraints.

Cell holds the value of the variable, as well as the domain of possible values. If value of a cell is non-zero, then its domain must contain only the value of the cell.

Constraints can be "alldiff" and "sum" (latter used for Killer Sudoku (not implemented)). Constraint holds the list of cells that are bound by this constraint. For example, a row constraint will have all cells in that row.

General Solver

The puzzle encoding allows the solver to be completely general. It doesn't care about what the puzzle looks like or how many cells or constraints it has. It only cares about a collection of cells and a collection of constraints that bound these cells. It then manipulates these two lists of data structures to produce a solution.

For example, the classic sudoku has 81 cells and 27 constraints, 9 cells each (cells overlap in some constraints). The solver takes these two lists and produces the solution.

In the same way, we can encode the Triple Doku as 171 cells and 73 constraints that contain subsets of these cells. The CSP solver will take these two lists as it did for the classic Sudoku and produce the result in the exact same manner.

For the Killer Sudoku, we would have the same 81 cells and 27 constraints as for the classic Sudoku, plus the "sum" constraints that contain the desired sum and a collection of cells that are bound by those sum constraints. Again, the CSP solver will take the list of cells and a list of constraints and produce the solutions it did for the previous two cases. The only modification to the solver is the addition of logic to check the consistency of the sum constraint. Everything else stays the same.

Therefore, the Solve class and its methods are completely general and solve puzzles independently from the puzzle style.

Performance Results

Analysis was performed by solving 10 puzzles of each puzzle type three times: with a plain depth-first search (DFS), with AC-3 preprocessing, and with forward-checking. The following metrics were gathered:

Search Cost (Values considered)
Shows how many values the solver tried for all variables. This gives an idea of how efficient the solver was when picking values.

Depth (Backtracks)
Shows how many times solver ran into an erroneous solution. This gives an idea of how effective the solver was in its analysis and how many times it made the wrong choice.

Execution Time
Time it took for an agent to solve the puzzle in milliseconds. Time is the average time to solve a single puzzle across all instances.

Classic Sudoku

The following bar graphs summarize the three key performance averages for each configuration of the solver for a classic Sudoku puzzle.

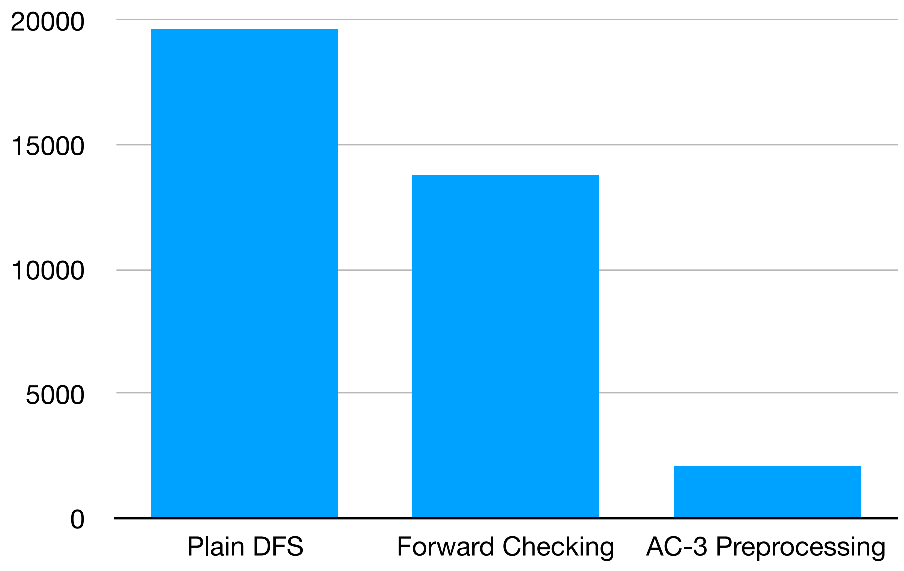


Figure 4. Classic, Nodes Expanded

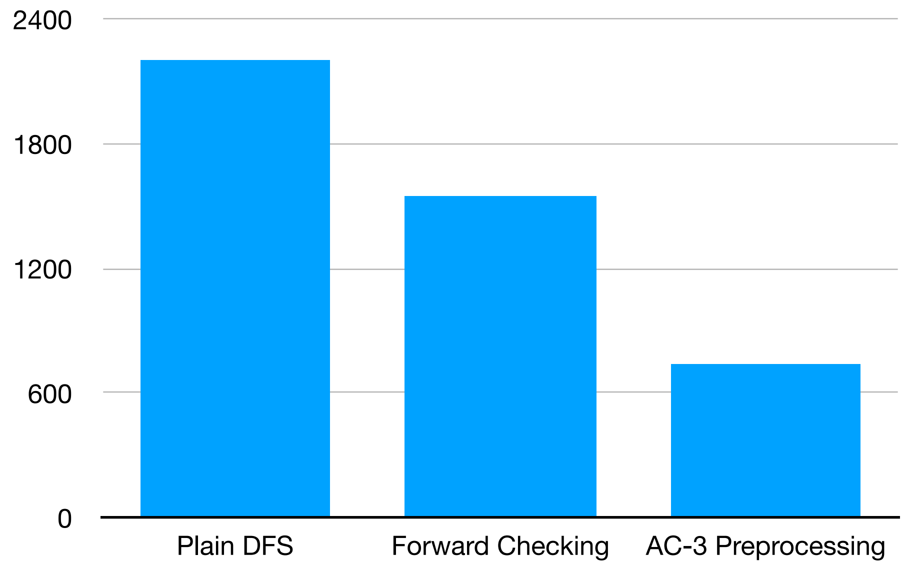


Figure 5. Classic, Depth

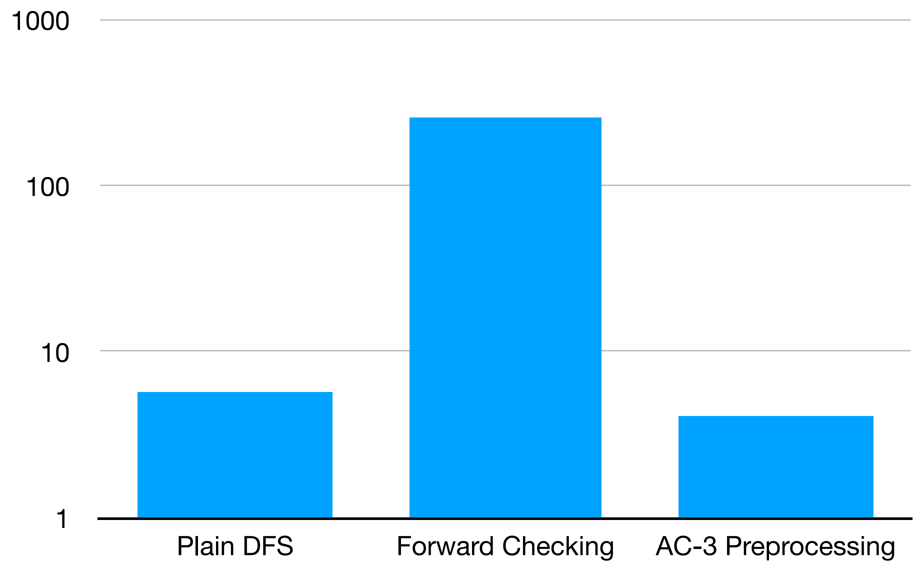


Figure 6. Classic, Execution Time (ms)

We see that AC-3 Preprocessing shows better performance than both Plain DFS and Forward Checking. The latter shows better performance than the former, except for the execution time, which is orders of magnitude greater (notice logarithmic scale in the third graph).

Results Summary

From the data we see that AC-3 Preprocessing demonstrates the best results in all three metrics. Plain DFS performs the worse in the Expanded and Depth metrics, while Forward Checking sits in the middle, except for the execution time, where it performs 40 times slower.

The benefit of AC-3 Preprocessing comes from reducing the domains of all the variables to as few values as possible. This has a direct impact on the Expanded metric, as now the program has less values to consider. The depth is reduces as well, since now the program is not wasting time analyzing values that clearly resulted in failure. Because the puzzles were of medium complexity, AC-3 Preprocessing was able to simplify the solution by a considerable margin.

Forward checking is more interesting. It also adds the benefit of AC-3, except now instead of checking all constraints in the puzzle beforehand, it checks constraints that contain the variable currently under consideration. This process does not eliminate values from the domains of the variables, because it's unknown whether the value chosen is correct. It only lets us detect an inconsistency earlier in the process. This results in a reduced Expanded and Depth metrics, but not as much as in the Preprocessing case. Because we are running AC-3 for each value we consider, this adds a very large overhead to the program, which causes a much slower execution time. It is possible that combining Forward Checking with MRV would improve the performance.

Overall, AC-3 Preprocessing has demonstrated a much better performance over Plain DFS and Forward Checking.