1 INTRODUCTION

This report covers solving the ABC puzzle. Part 1 checks the puzzle's correctness procedurally, more particularly its consistency and whether it is fully filled. Part 2 implements making deductions about the puzzle procedurally. These algorithms are tested for their functionalities and further evaluated according to their performance.

In the third part the program encodes the problem into propositional logic, and then use the LogicNG library to use a SAT solver to solve the puzzles. The SAT solver library can be accessed here: https://github.com/logic-ng/LogicNG.

Running the code instructions

The attached files are already compiled, so there is no need to compile them again.

To run particular part navigate to the src folder in the command window and write *java main <TEST> <test-set>* where action type and specification are arguments.

Arguments: TEST Student or Student3 – is for tests by the student, where Student3 solves ABC1 puzzle declaratively.

Arguments: TEST Staff1 or Staff2 or Staff3 or StaffAll – is for tests by the stuff

Example of a command for calling staff test 1: *java main TEST Staff1*

2 DESIGN AND IMPLEMENTATION

The attached file consists of 6 folders. Folder mains consist of a class main used to run the required algorithms, and Grid and Puzzle are used to specify the current puzzle. There is also one called gridToString, which converts grids to strings. Other folders are part 1-4, which are further discussed below. Folder tests are used for testing.

2.1 Part 1

This part consists of one class called CheckerABC, which has three main functions. Function isFullGrid() has a goal to decide whether the given grid is fully filled with valid symbols or blanks. isConsistent() finds out whether the grid is consistent for the given puzzle that means whether all constraints are obeyed except the constraint that every square should be filled in. Function **isSolution()** is then truth if the both isConsistent() and isFullGrid() are true.

isFullGrid()

Firstly this function checks whether the puzzle is valid by puzzle.checkValid() and also whether the provided grid is valid for that puzzle.

Then it checks every position in the grid, whether it is a valid symbol. It was used validSymbol() method that was added to the Puzzle class. It returns true if the current character is a puzzle's valid symbol. If all characters in the grid meet this condition, then isFullGrid() returns true.

isConsistent()

This function has to check whether the puzzle and grid obey all of the other constraints. It contains five if-statement conditions; if any of them is satisfied, then it returns true. Firstly, as in isFullGrid(), it checks the puzzle's validity and the grid's validity for the puzzle by comparing sizes.

The function sameLetterRows() checks whether there are repeating letters in the row. It takes row by row and converts it to a string. It takes the string and checks each element of that row for letter and blank symbol occurrences. If it exceeds the allowed limit, it returns false. The same is repeated for the columns by sameLetterColumn(), this time, the indexes of chars are switched.

The function clueConstraint() checks when the clue is given whether the first letter in the grid in that direction is the same letter as the clue, except a blank symbol. The algorithm checks each side of the clues separately. It is based on the thought that between the clue and the letter in the grid can be only a particular maximum number of blank symbols. The length of the grid minus the number of valid letters gives it the maximum number of blank symbols. Also, if there is an unfilled character in that range (maxBlankSymbol) or if the clue is unfilled, the constraint is not violated.

Lastly, function isValidSymbol() checks that all characters in the grid are valid relative to the puzzle.

The testing of the class was primarily done on the provided test sets (Staff1). Also, a few more tests were added in the TestStudent class to test when the puzzle with mismatched clues is given or with incorrect dimension. It also tests validity when the clue is a blank symbol.

2.2 Part 2

This part consists of one class, ProceduralABC that implements some of the methods used to solve the ABC puzzle procedurally.

differentCorners()

It checks each corner's clues, and if these two clues are different letters, it assigns a blank symbol to that corner in the grid, unless one of the clues is infilled character.

onlyPlaceForLetterRow()

At first, the whole grid is converted into the arrayList of strings for better manipulation, where one element represents one row. This is done through the function gToString() (from gridToString class), which also takes direction parameter, which decides whether the elements in the final arrayList are rows or columns.

It then loops through the whole arrayList element by element and letter by letter (basically one character from the grid one by one). If it finds an unfilled character in the string and if all the letters from the puzzle were used except one, it assigns the value of the missing letter to the grid instead of the unfilled character.

The same is done for columns in onlyPlaceForLetterCol().

fillInBlanksRow()

It again converts the grid to the string arrayList. The algorithm counts the maxim number of blank characters and loops again through arrayList of strings. If there is less blank symbols in the row than unfilled spots, it then checks whether all the other letters from the puzzle are filled in the row. If that is true algorithm assigns blank symbols to all unfilled spots. The same is done for columns in fillInBlanksCol().

commonClues()

This function calls commonCluesSides() function for each set of clues (e.g., top, left). This function creates three arrayLists where *common* represents clue letter that appeared there more than once, *numOfBlank* represents how many blank spaces is caused by that common clue (e.g., three same letters will cause 2 blank spaces, one pair one blank space). s*ingles* represent letters that appeared in the clue only once.

If all the blank spaces allowed for that row or column (given by sum of all numbers in *numOfBlank*) are assigned by common clues, the rest of the letters from *singles* can be assigned to the appropriate position right next to its clue in the grid.

Additionally, there is a function cornerLetters() that assigns the same letter as a clue to the corner which is surrounded by the same clues. It works similarly as differentCorners().

The testing of the class was primarily done on the provided test sets (Staff2). Also, a few more tests were added to test common clue on blank symbols. One tested that it does not count blank to *singles* and the other that it does not take blank into account in clues. And test for cornerLeters().

2.3 Part 3

This part consists of three classes that implement methods that are used to solve the ABC puzzle declaratively.

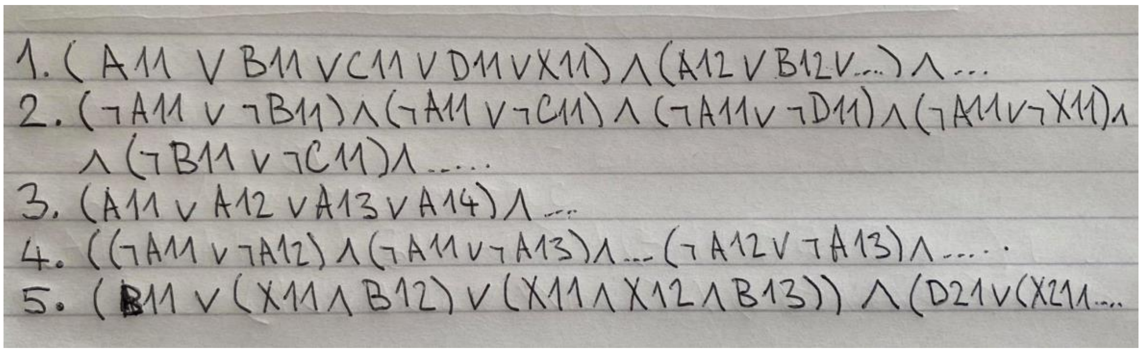
The main class here is DeclarativeABC, where function createClauses() takes the puzzle and translates it to the propositional logic by calling supporting class Constraints. It also translates a grid into propositional logic if one is given. It uses various of for-loops and string appending to get the propositional logic.

There are five constraints to describe the puzzle that is created by Constraints class. These constraints are indicated in figure 1.

Number 1: „Every square contains at least one symbol", number 2 „Every square contains at most one symbol". These two constraints also consider whether there is blankSymbol needed for the puzzle.

Number 3 „Every row/column has every letter at least once", number 4 Every row/column has each letter at most once, these two constraints exclude clauses that would say that there is at least one x or most one x because in some puzzles there might be more than one. This should be fixed by the first two constraints that say each square contains exactly one letter.

Number 5 „Where a clue is given in a row/column, the first symbol in that row/column direction is consistent with the clue".



1. $(A11 \vee B11 \vee C11 \vee D11 \vee X11) \wedge (A12 \vee B12 \vee \ldots) \wedge \ldots$
2. $(\neg A11 \vee \neg B11) \wedge (\neg A11 \vee \neg C11) \wedge (\neg A11 \vee \neg D11) \wedge (\neg A11 \vee \neg X11) \wedge$
   $\wedge (\neg B11 \vee \neg C11) \wedge \ldots$
3. $(A11 \vee A12 \vee A13 \vee A14) \wedge \ldots$
4. $((\neg A11 \vee \neg A12) \wedge (\neg A11 \vee \neg A13) \wedge \ldots (\neg A12 \vee \neg A13) \wedge \ldots$
5. $(B11 \vee (X11 \wedge B12) \vee (X11 \wedge X12 \wedge B13)) \wedge (D21 \vee (X211 \ldots$

Figure 1: Constraints

Finally, class gridToLogic translates the grid when one is given.

Created strings are then fed into the solver in solvePuzzle() that returns true if the puzzle is solvable.

The function modelToGrid() creates the model, which is then converted into the grid.

Testing was done manually on a simple example, and it worked. Then tested on Staff3 test set, and it passed roughly half of the tests.

3 EVALUATION

There are presented two methods to solve the puzzle—the procedural and declarative way using logic.

The procedural solution is much more branched, and there must be considerate every action. When the human look at this, it do not have to be clear what is happening there for the first time. We did not even implement enough rules to fill the grid to complete the solution.

On the other hand, when the human looks at the logic, it is much more comprehensive; those five constraints and then it is clear right away what it is doing. Even though creating functions to convert puzzles to strings is tricky, it is then able to solve the whole puzzle quite easily.