CS21120: Data Structures and Algorithm Analysis
Assignment 1 – Sudoku

James Edward Euesden – jee22

Abstract:

Creating a Sudoku solving application in Java, testing our abilities to create robust and maintainable data structures and make logical, clean algorithm code.

Contents

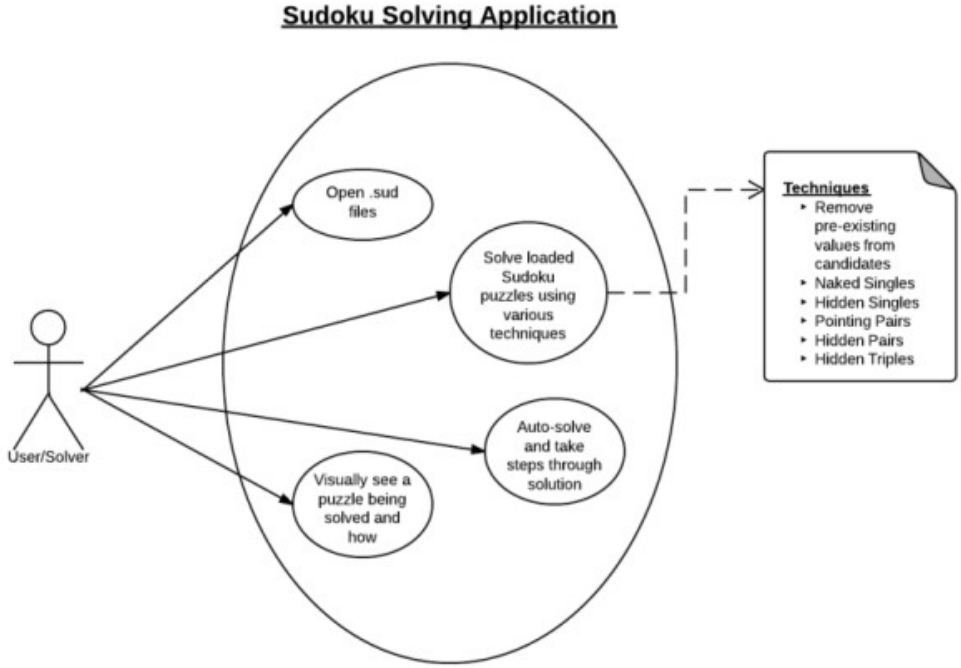
Introduction

The task and problem set was to design a solid data structure and write algorithms for solving sudoku puzzles with a Java application, making good, robust, maintainable and understandable code. There were a number of '.sud' Sudoku puzzle files provided in order for us to test our algorithms and data structure against actual data, though there was no requirement to solve them all.

It was also expected of us to conduct a series of test cases to provide evidence of our application at work, and also write a number of JUnit tests, to unit test each part of the model of the application. While I have documented much of my own implementation here and the reasoning for it, some more in-depth information about the specifics of methods can be found within the JavaDoc for the application.

Analysis and Design

To first get my head around the problem, I drew a Use Case diagram. Although there are not many use cases, it was good to have a base reference for what the resulting application should look like and do, including any techniques that I wanted to use to solve some of the puzzles.



A user must be able to see some sort of an interface, or have input, of what .sud file they would like to load into the solver. Once loaded, this file should be able to be seen in its initial state. The user then has the option of 'taking a step' or 'solving' the puzzle automatically. As the steps are taken, the user should be informed of exactly what steps were taken, where they were taken on the grid (visually and/or by text) and see the solution as the application runs through it's solving algorithms until it reaches the solution.

My Approach

To understand how to tackle these cases, I thought about potential data structures and how it should be built. I decided that at the core of the application would be a class that holds a 2D array of an Object, 'Cell'. Each Cell would hold information on where it is in the grid, what block it belongs to, if it has any value and, if no value, what potential candidates it holds. The candidates are held in an array of size 9, with 'empty' candidates represented by a 0.

This formed the heart of the application, as it is easy to iterate through and perform simple manipulations on the Cell data. It is also relatively easy to get the data in a form that makes it display the same as a normal Sudoku grid (using nested for loops). I have also used 2D arrays in previous projects, so feel comfortable in my understanding of building a data structure around 2D arrays as a base.

However, while 2D arrays are excellent for storing the core data and performing simple operations on the Cells in the order they have been stored, I knew that just this alone would make writing solving algorithms more difficult than necessary. My next step was to think of a way to hold specific collections; Rows, Columns and Blocks, where each cell contained in the collection could not hold the same value as any other cell in that collection. For this task, I chose to use LinkedLists to hold the individual collection data, as they are easy to iterate through, would keep Cells ordered and are relatively good on space/time.

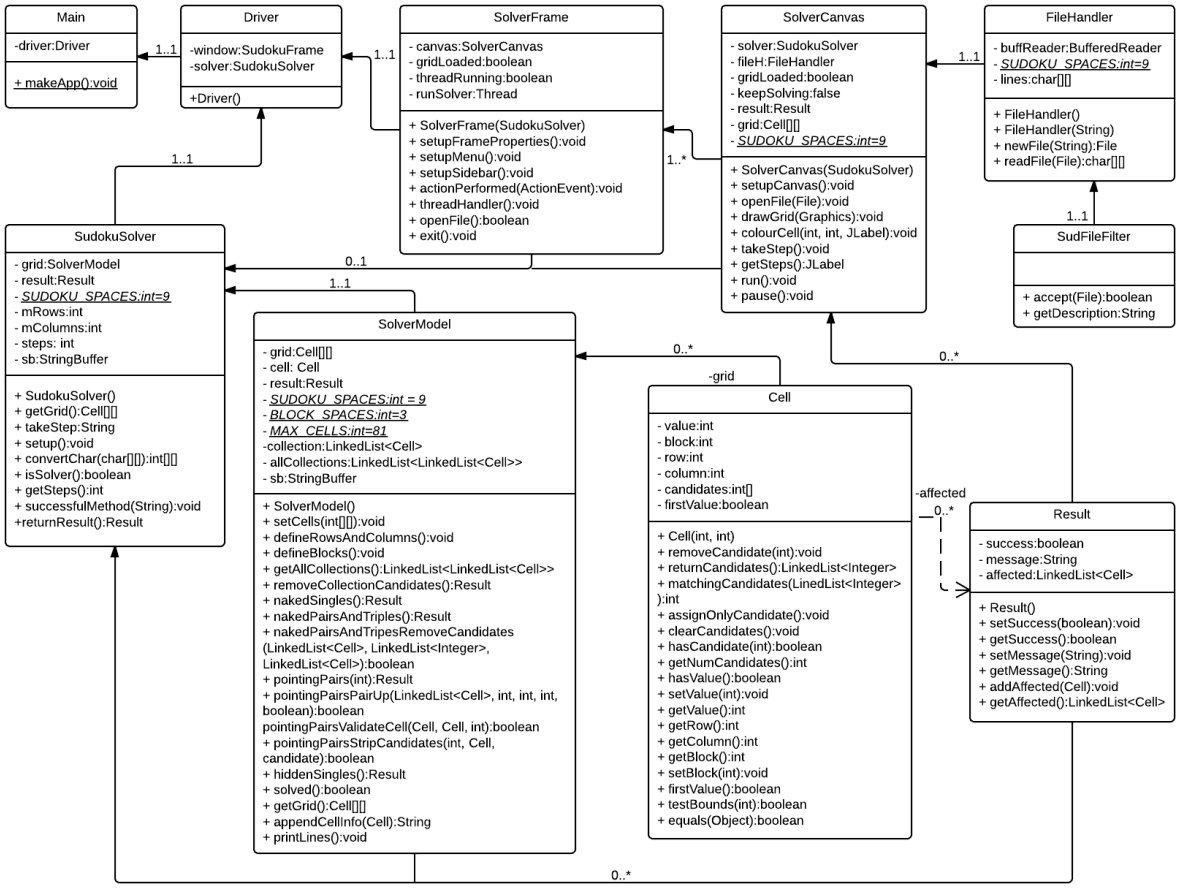
My implementation of this can be seen in my code. When new Cells are created and set into the 'grid' (2D Cell array), a method is later called to go through the grid again and create/store a Row, a Column, a Row, another Column, etc, until all have been stored, and then store all Blocks, all in the same LinkedList, 'allCollections'.

Doing it this way meant that no matter which collection I called, the Cells would all be in order from the start of that 'collection' to the end. To further demonstrate why this is important, think of this: In order to perform the same operation on a Column, a Row and a Block, a different nested for loop must be used for each, as each of them contains legal neighbouring Cells in different adjacent spots depending on the collection type. By using an overall collection list, each collection is ordered as it would normally be, but is stored the same as any other type collection, so can share use in operations regardless. This was one of the most important parts of my data structure, and lead the success of the majority of my algorithms, both in code and in maintainability.

Aside the data structure, I began thinking about what Classes I would need for the application to function, keeping in mind that I would like the end result displayed in a GUI. To best demonstrate this and to give myself a guideline, I drew up a mock UML diagram, implementing an approach to design similar to Model View Controller (MVC). While the final result is not strictly the MVC design, the core concepts of keeping the Model, Viewer and Controller separate have been adhered to, to keep the data mostly separate from the viewer and in their own respective classes. I chose this way as it is reliable for creating well structure code in classes and easy to maintain individual classes.

My final UML diagram is displayed below, and should be understood to be the final draft of my UML class design. (A larger version of this UML diagram has been included with these files for ease of viewing).

UML Diagram for Sudoku Solver



UML/Class description:

As stated, I attempted to keep the data separate from the GUI elements and classes. In regards to the GUI, I have omitted a number of GUI elements from SudokuCanvas and SudokuFrame to save on space.

Main and Driver

There is a Main class, from where the application is started, and creates a new instance of the Driver class. This Driver class creates a new instance of the SudokuSolver, and then of SudokuFrame, passing a reference to the Solver as a parameter.

Model:

SudokuSolver

This class has methods that call to SudokuModel, and determines which order each of the solving algorithms should be executed (takeStep()). Further documentation of these methods can be found in the JavaDoc files. The important thing about this class is that it is the 'go-between' of the Viewer and the Model, receiving commands from the user and executing them by calling to the Model, then returning any results back to the Viewer to be shown to the user.

SudokuModel

The heart of the sorting algorithms and data structures. In here is where all of the solving

algorithms, the individual collections the grid of Cells are contained and can be manipulated. Notable methods are the solving techniques:

- RemoveCollectionCandidates – Removes existing values in collections from Cell candidates.
- NakedSingles – The last remaining candidate in a cell must be it's value.
- HiddenSingles – The only cell with a particular candidate in a collection must be that value.
- NakedPairsAndTriples – Two cells in a collection that share the same candidates or three cells in a collection that share the same candidates, either together or spread between them, must be the only one's available to that collection that can hold those values, so any other cell holding these values as candidates should have those candidates removed.
- PointingPairs – In each block, if there are multiple cells with matching potential candidates, and they are in the same row or column, without those candidates appearing anywhere else in the Block, then only those two cells can be those values, so they 'point' to the other cells in their row/column that should not contain the candidates that the 'PointingPair' share.

I felt these techniques would be enough for my implementation of a solution to the problem presented. As it turns out, these techniques completely solve each provided Sudoku puzzle example. SudokuModel also returns whether all Cells are solved or not, for use in the SudokuSolver and stopping the solving process in the Viewer.

Cell

Holds information pertaining to itself. It's current value, it's location in the grid, what candidates it might have and whether it was an initial cell or a solved cell. It's methods are often getting and setting this information, such as which candidates it has, the full list of candidates, if it has a value and more. There are other methods too, such as 'AssignOnlyCandidate', which assigns the last candidate in the list of candidates, should there be only one left.

Result

Has minimal methods aside setters and getters. Its main function is to be used as a 'result' of performing an algorithm. Initially my thoughts lead to using a simple boolean value on the success of algorithms, but I found that to truly show the user what values have been updated, both visually and in text, it was necessary to write this class to hold that information and be returned in place of a boolean.

It contains a boolean of if the method was successful, any particular message relating to the algorithm used and a list of which cell(s) were affected by the algorithm.

Viewer:

SudokuFrame

The frame is the main container for the GUI, holding a menu bar, buttons for solving/steps, a space for text output of results and a large space for the Sudoku grid. There are issues with this GUI, but since the GUI was my own choice to implement, sticking to the base use cases I decided upon at the start, I feel it is sufficient enough. Should there be more time for this project, I would work on making this UI cleaner and more user friendly.

SudokuCanvas

Holds multiple JLabels, one for each Cell in the 2D array of cells in order to visually display the state of the Sudoku puzzle. Depending what block the cell is in, or whether it was just 'solved' by the SudokuSolver, the background colour of the JLabel changes. Each cells candidates are also

displayed in a full list, as are any initial values (blue) and solved values (red).

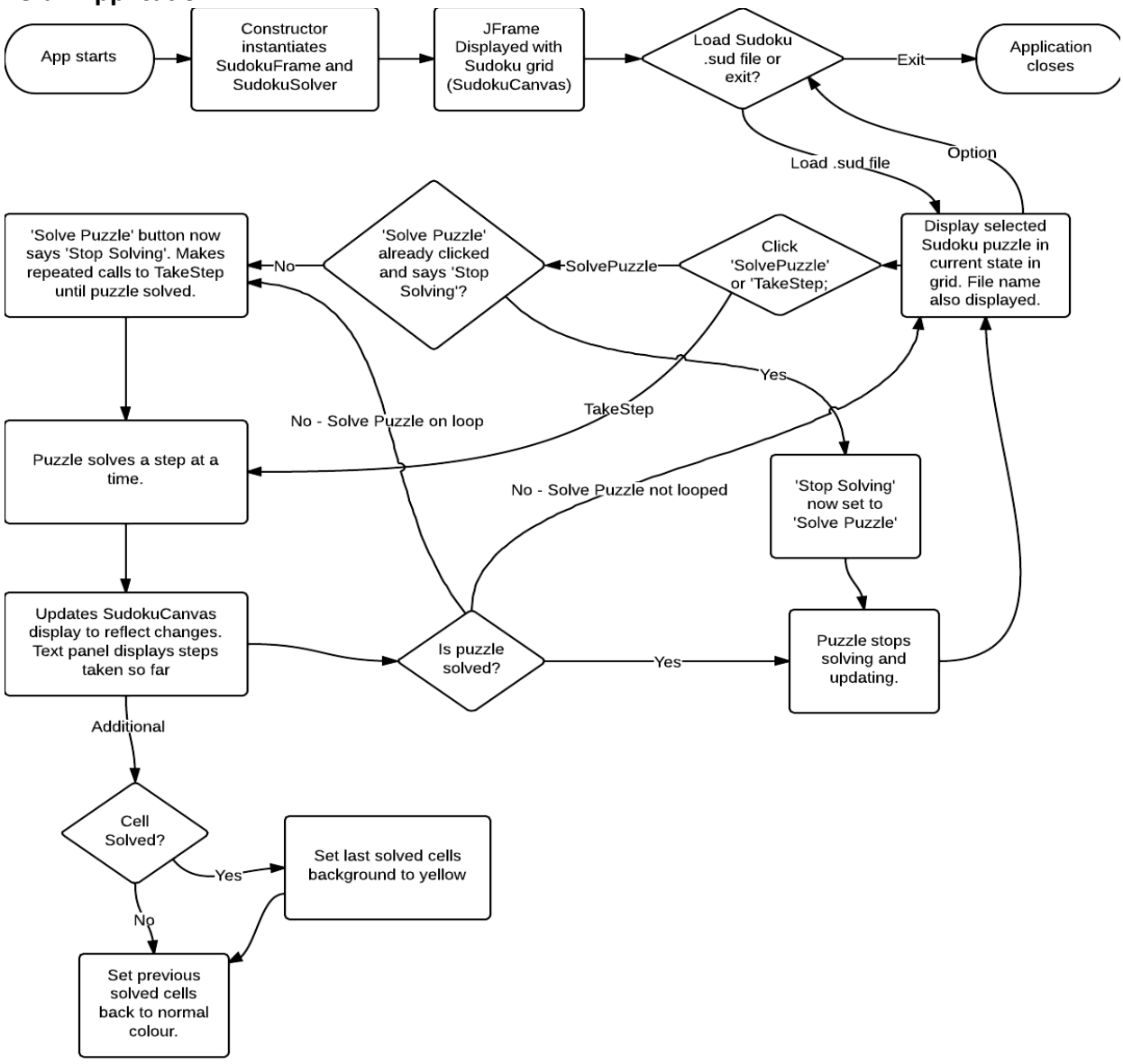
SudFileFilter
A custom file filter. The original version of this code dealt with .txt files, and was used in my first year CS12420s assignment, 'Blockmation'. By just altering the file type suffix to .sud, I was able to reuse this class. Its function is purely to only display directories and files ending in .sud when a user is selecting a file to open from the SudokuFrame. It will not stop them from displaying 'All Files' and seeing all files to load in. Should this project be continued outside of a simple assignment project, I would address this issue.

Implementation

Algorithms and Flowcharts

Once I knew what I thought the application should do and look like, I began thinking of potential designs and pseudo code for each solving technique that would later be implemented. To aid me in this process, I took down notes in pseudo code and then turned those into flowchart diagrams, to show the way the solution process should go. Each of these can be seen below and is explained.

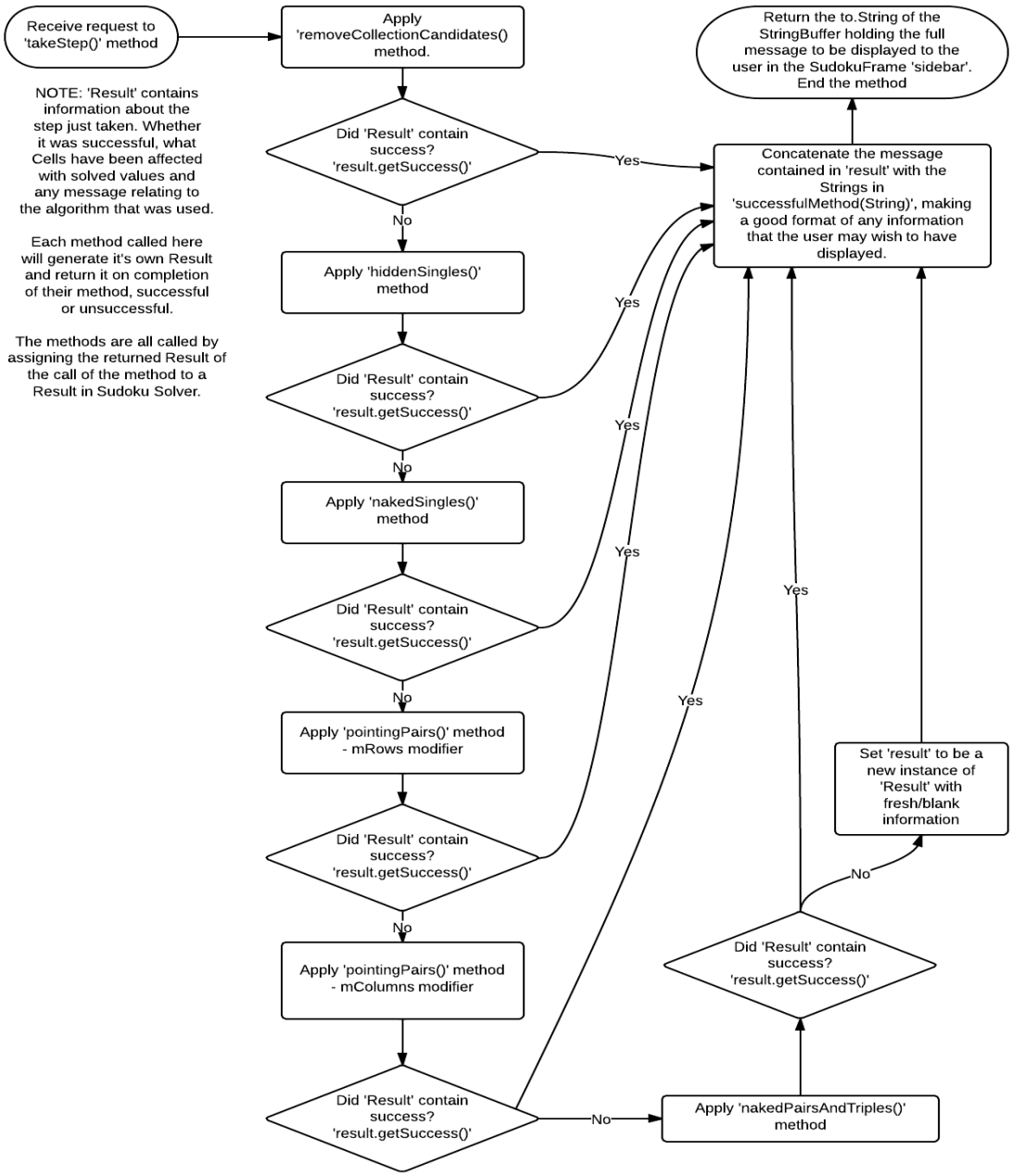
Overall Application:



The application allows the user to load in a .sud file. From there, they may take the solving a step at a time or let the solver repeat all of the necessary solving steps to complete the puzzle automatically. While solving the puzzle, the application updates the display to reflect the most recent changes in the Model to the user.

The option to be able to load up a new file and/or exit the application should be available at all times, and stop a solution from running. With more time for the project, I would look into implementing the ability to take backward steps and also to choose the steps taken by the solver, rather than having it make them for the user based on a priority order, which can be seen below:

Application Solving Techniques - Priority Order Flowchart (SudokuSolver – takeStep()):

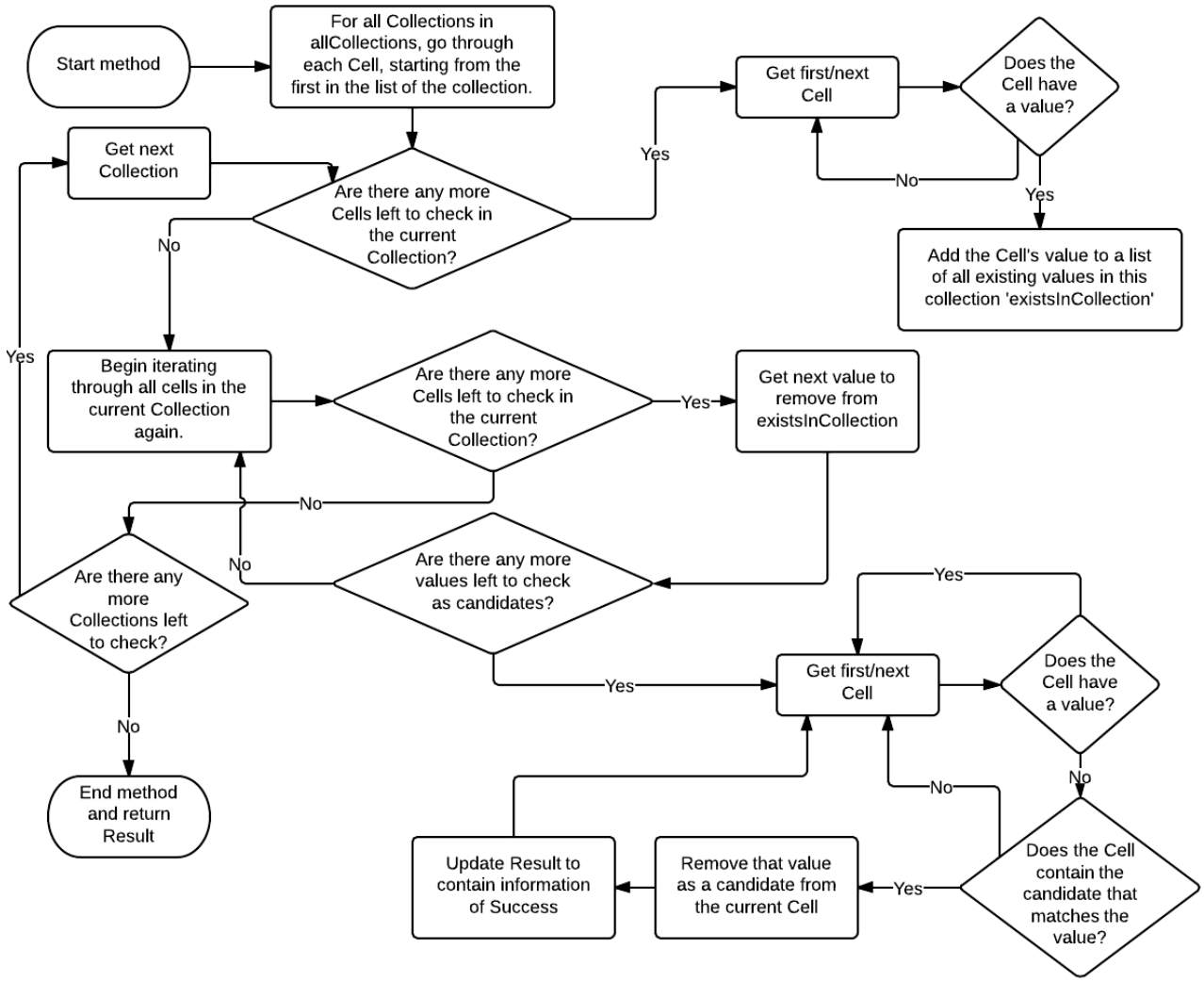


The principle behind the 'priority order' approach of solving puzzle 'step-by-step' is that the most advanced solving techniques are not called to be used unless the simple ones fail. This attempts to ensure that the application does not use every solving function every time it is asked to call a step.

One downside with this is that if the next step to solve happens to be the last solving algorithm (in this case, nakedPairsAndTriples()), the solver will still attempt to apply and execute each previous method, adding to the complexity of the process. I f given more time I would look into improving this. Though it is difficult to try and find a solution to a problem without first attempting to use the solution, so as of right now I am not sure of the correct way to improve this implementation.

After establishing how each step of the solver was to be run, and knowing that the grid was complete when all Cells had obtained values, the next step was to implement the algorithms. I felt the best way to represent these was through the use of Flowcharts and explanations of the method and why it is written as such.

removeCollectionCandidates():

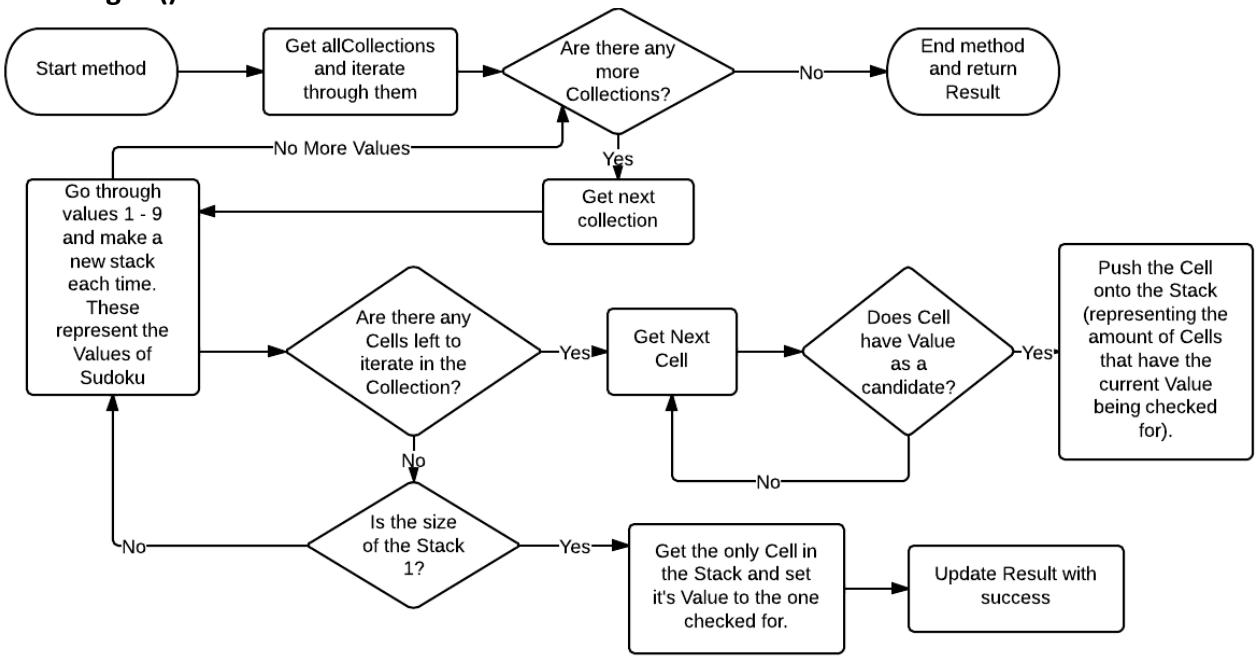


The method is simply as was described above, “If a Cell in a Row, Column or Block has a value, then this Value cannot exist as a candidate in any other Cell in that collection as each Row, Column or Block may only contain one of each value, 1 - 9”. The simplest way to implement this was by looking what Cells had values in any one particular collection, saving any values found to a list and

then later going back through the collection and removing those found to be existing values from unsolved Cells candidate lists.

This is a good example of how my data structure of having each 'Collection' in the same list helped cut down multiple lines of similar code, iterating through each different type of collection with one method for all Cells and types, rather than multiple repeated nested for loops for each type.

nakedSingles():

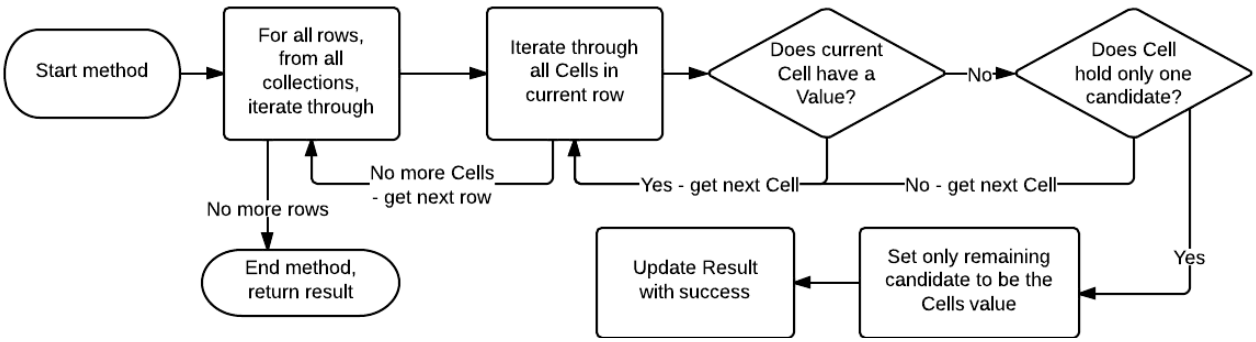


A Naked Single in Sudoku is where only one Cell in a collection has a particular value as a candidate, even if it has many other candidates, it can be assured that this is the only Cell that can be that value if no others in the collection share this Candidate.

My way of implementing this was by going through each collection, and then creating a Stack to represent each value, iterating through the cells in that current collection, 9 times (one for each possible sudoku value), looking in its candidates for the number of times the iteration has been over so far (the amount of times being the 'value' being searched for, starting at 1).

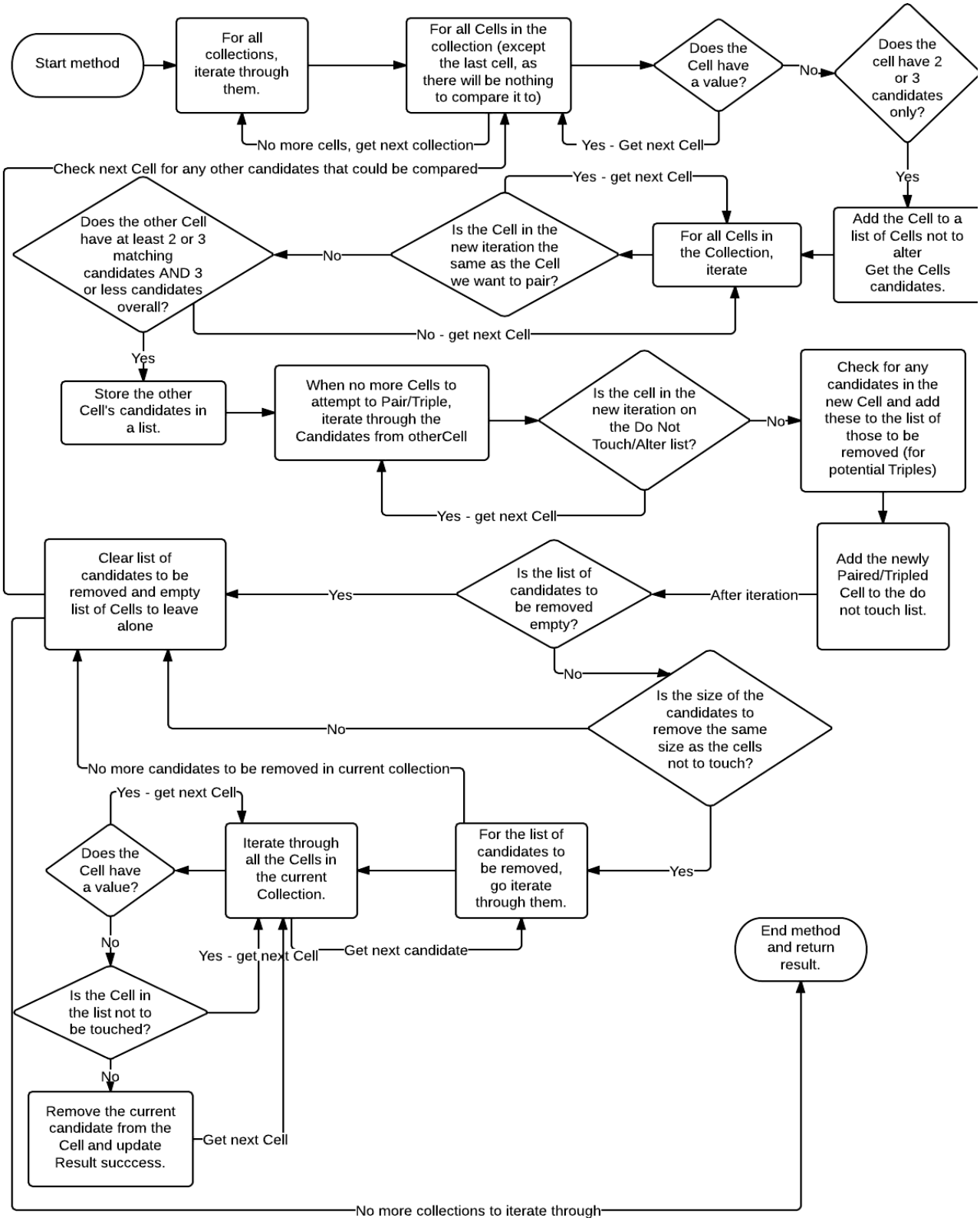
If this value is found in a Cells candidates, that Cell is pushed onto the Stack. With this in mind, we can be sure that if by the end of the collection there is only one item on the stack, this must be the only Cell that can contain the value, and so we may set that Cell's value and move onto the next value.

hiddenSingles()



Hidden Singles scans through each Cell in the grid (in my implementation, I based this off iterating through all rows stored in 'allCollections') and looks to see how many candidates they have. If they only have one candidate, it is clear that this Cell must be that candidates value, as it can be nothing else. This is perhaps the most simplest of all solving algorithms (not including candidate removal techniques).

nakedPairsAndTriples()

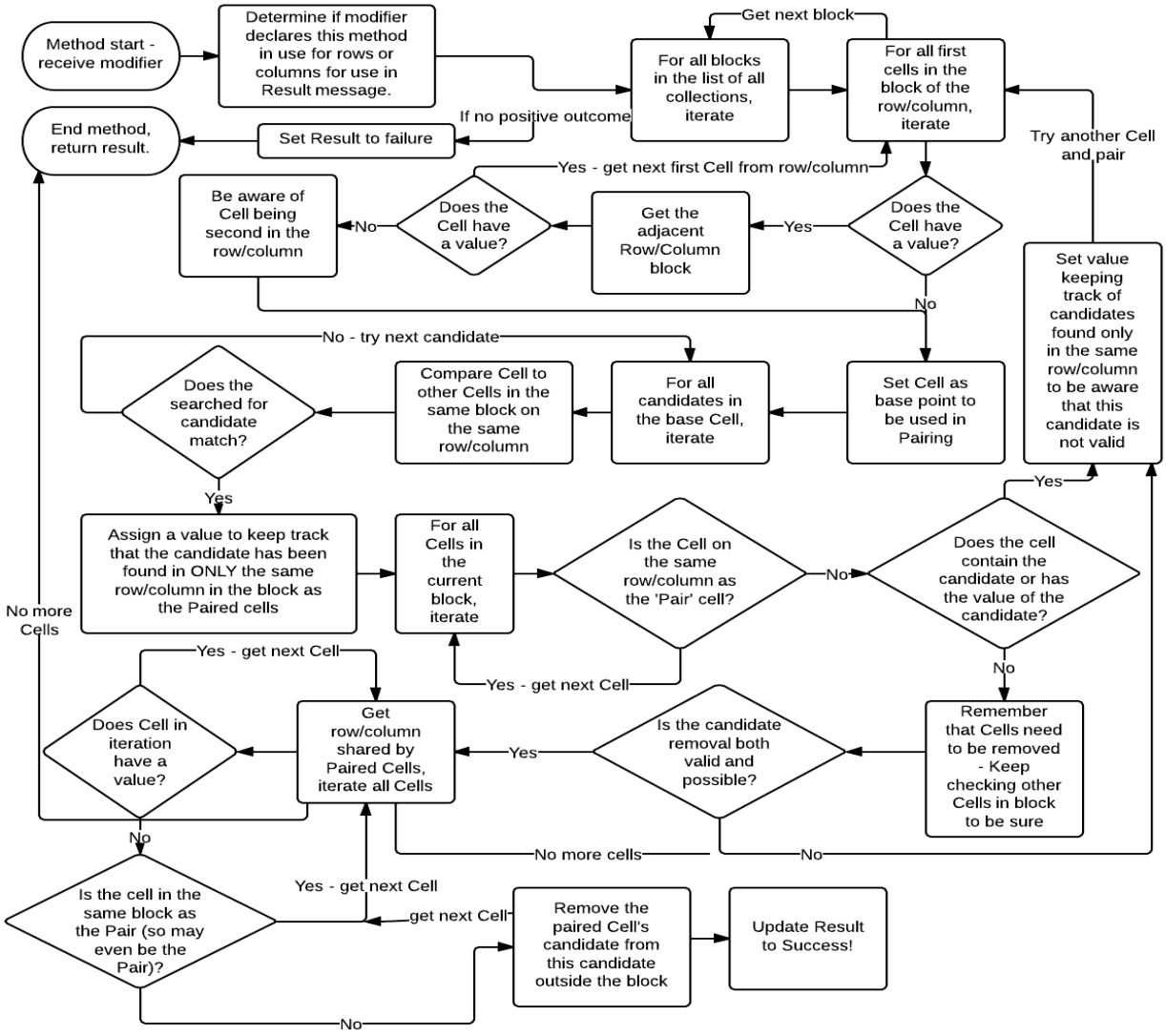


A slightly more advanced technique of Sudoku, yet still relatively simple. However when coding, becomes something quite complex. Naked Pairs and Triples looks for Cells that have only 2 or 3 candidates, and that share these candidates between other Cells who also only have 2 or 3 candidates, that are in the same collection. If they are found, it means no other Cell in that collection can hold these values, as there are only 2 cells holding 2 shared candidates (and no others) each, or three cells that hold a combination of any 3 values, and no others.

With this knowledge in mind, the most notable thing about this method, and one that I found the most tricky to implement even though I understood it and what I wanted to do. It was that no matter what, if we have the same amount of Cells with some matching candidates as we have amount of candidates to be removed from Cells, outside the Pair/Triple, then this must be a legal removal of candidates. Implementing this was a simple check of sizes of lists, but it took me some time to figure out this was the best way of confirming that the Pair/Triple was correct.

If there were say, three candidates and only two Cells paired, then this would not be a legal move, as we do not know if the 'extra' candidate would be a choice for these candidates, or belong to another Cell elsewhere. The technique only works if candidates in the Cells match and are their ONLY choices. The same applies for Triples, although it is possible to have a Cell with three candidates, and two other pair cells with two candidates shared to the Cell with three but not to each other. This means that if they can only be those values, other Cells cannot.

pointingPairs()



Out of all solving algorithms, I found Pointing Pairs to be perhaps the most challenging. Based upon the way in which we iterate through 2D arrays and lists normally, I had to structure the method so that no matter what, the algorithm would go through each block, as stored in a LinkedList, but go through the rows and columns in that block separate to one another (where cells are arranged left-to-right, top-to-bottom in a list). Ensuring the code was maintainable and reusable by both Cells and Rows resulted in me using a 'modifier' to change the way in which the method behaved depending upon whether the search was for Rows or Columns.

The full explanation of this can be found in the JavaDocs and code comments, however, in brief the modifier is either 1 or 3. 1 refers to Rows, 3 to columns. As we know that a normal Sudoku grid is always 9x9, and blocks in these grids are always 3x3, we can determine that modifiers of 1 or 3, when using a for loop to go through a list, are the perfect ways of pulling out an element that may be in a row or column.

To explain, think of a Row: {Cell, Cell Cell}, and how a column is just the same but along the Y axis. However, the first Cell of the first Row, when Cells are stored left-to-right, top-to-bottom by Rows, is also the first Cell of the column, with the second Cell in the Row as the first Cell in the second Column, and visa versa, where the second Row's first Cell is the second Cell in the first Column.

With this in mind, you can think that if the cells locations are values (0, 1, 2), to get to what would be 'the second Cell in the first Column', we would need the Cell at location (3), assuming we have a 3x3 grid. If we are starting from 0, we know that if 3 is a modifier for columns, and the condition in the algorithm for finding the 'next' cell in the collection is '1 * modifier', 1*3 will give us location '3', the next column Cell, while modifier 1 would give us '1', the location of the next Row Cell.

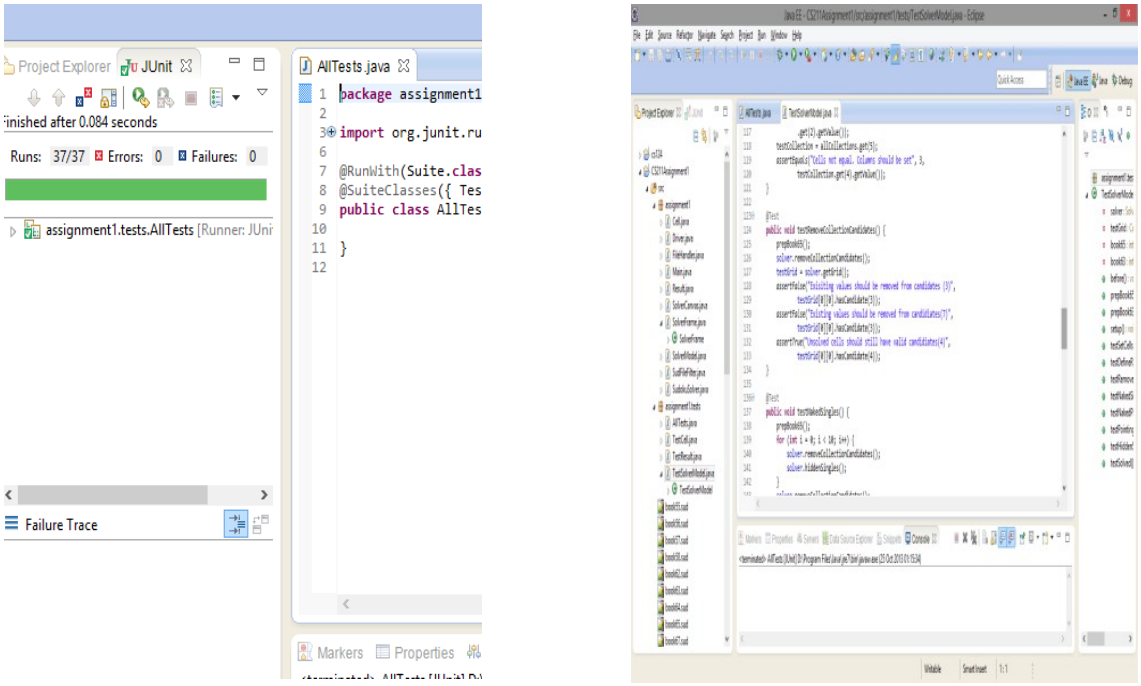
Once I had discovered this through use of pseudo code, my flowchart and hand drawn workings of each iteration through the Cells, the algorithm came together as you see in the code. Before this, the algorithm was similar but was hard coded to go through rows in one method and columns in another method. Wanting to keep repeated code to a minimum and for ease of maintainability, this was my solution.

With this in place, finding Pointing Pairs was as simple as comparing the candidates of unsolved Cells on the same row/column, in the same block. If there was a match, the algorithm then checks to see if this candidate appears anywhere else in the block. If it does, we can be assured these are not pointing pairs. If not though, the cells lead themselves to 'pointing' at the rest of the row/column where we can iterate through every other Cell in the same row/column and remove that candidate.

This is another method in which my data structure was extremely useful. Not only was it pivotal to have the Cells know their own location in regards to their blocks, rows and columns, but having a large list of collections, already separated into the correct cells, made keeping this code much cleaner and more maintainable than otherwise. While I feel my implementation is far from perfect and I'm sure there are other, better ways to do it, I feel I did Pointing Pairs in my application justice for the most part.

Testing

Throughout the time I was writing the code for this application, I was also writing unit tests using JUnit. By doing this, I could be sure that any results I got from the tests would show if the individual methods worked correctly and as they should as I wrote them. It also helped with any changes I made in my Model classes. I knew that when I did make any changes, when I ran my tests they should still give the same correct results, or show me that along the way something had broken.



Junit examples during my Sudoku Solving testing.

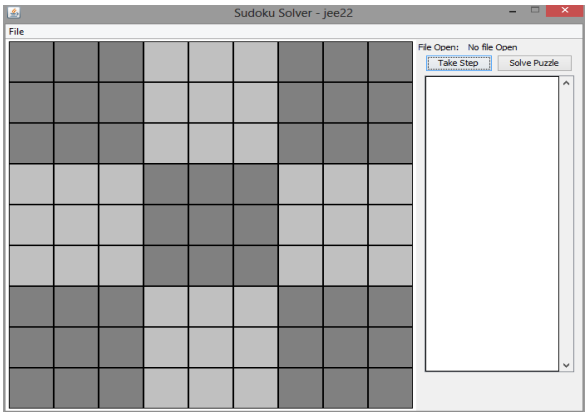
There are Junit testing classes for the SudokuModel, Cell and Results class, as these are the main classes that deal with the handling and manipulations of data. For all other classes, it was generally a point of if it didn't work, I would know very quickly when launching the application, such as classes involved with the GUI. In order to test these, I ran some black-box testing using Test cases and a Test Table, setting targets for specific requirements to be met, based upon my original Use Case diagram, and created a test table, with screen shots as evidence of my results.

Test Table:

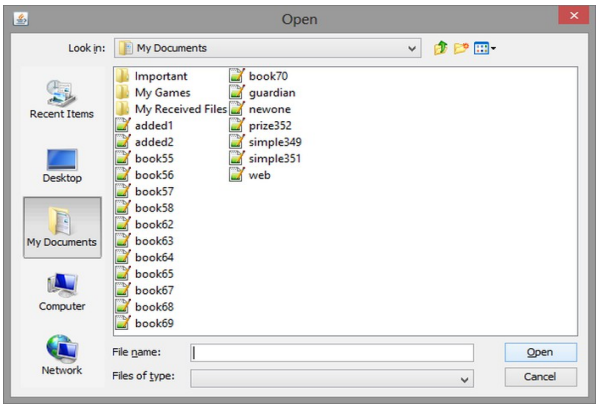
Testing Ref	Description	Action Taken	Expected Outcome	Screenshots
V.01	Load application.	Application started.	Application loads and should display a frame with an empty Sudoku grid and have 'No file loaded'.	SS.001
V.02	Opening .sud files.	Using the file opener to find and open a .sud file.	Opens the correct chosen file, displays a new grid, with the file name and initial state space displayed.	SS.002 SS.003

Testing Ref	Description	Action Taken	Expected Outcome	Screenshots
V.03	Using Take Step on unloaded grid.	User clicks 'Take Step' button while puzzle not loaded.	No changes.	N/A
V.04	Using Take Step on loaded grid.	User clicks 'Take Step' button while puzzle loaded.	The grid updates to display any changes, and some texts informs the user of what was done to make this occur.	SS.004
V.05	Visually seeing solved Cells.	User clicks 'Take Step' until a Cell is solved.	Once a cell is solved, it no longer displays candidates but instead has a number in it's place, and a yellow background after just being solved.	SS.005
V.06	Using Solve Puzzle on unloaded grid.	User clicks 'Solve Puzzle' while puzzle not loaded.	No changes.	N/A
V.07	Using Solve Puzzle on loaded grid.	User clicks 'Solve Puzzle' while puzzle is loaded.	The grid continuously updates as the algorithms work to solve the puzzle, until the solution is found. Text shows all steps taken, the 'Solve Puzzle' button becomes a 'Stop Solving' button to pause the solution.	SS.006
V.08	Loading a puzzle in the middle of another one solving.	Use clicks 'Open' to load another puzzle while an already loaded puzzle is already solving.	The solving should stop, the 'Stop Solving' button should change back to 'Solve Puzzle' and the user should be allowed to load in a new puzzle.	SS.007
V.09	Using 'Solve Puzzle' or 'Take Step' on a solved puzzle	User clicks 'Solve Puzzle' or 'Take Step' on an already completed puzzle.	No changes.	N/A

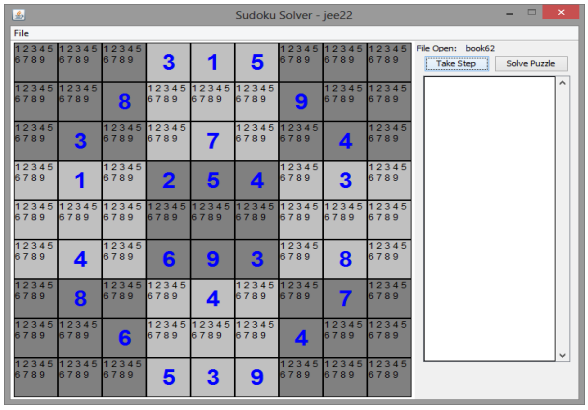
Screenshots – Testing Evidence



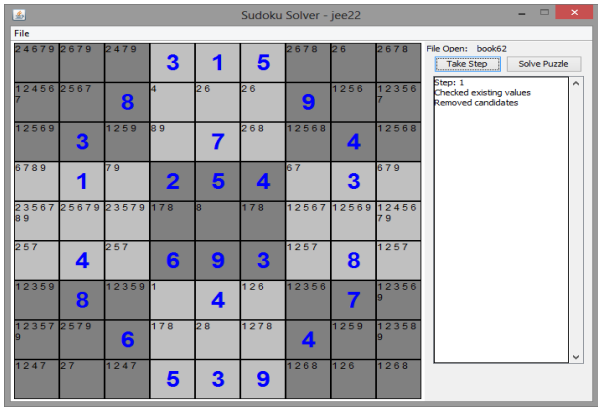
SS.001 – Loaded application



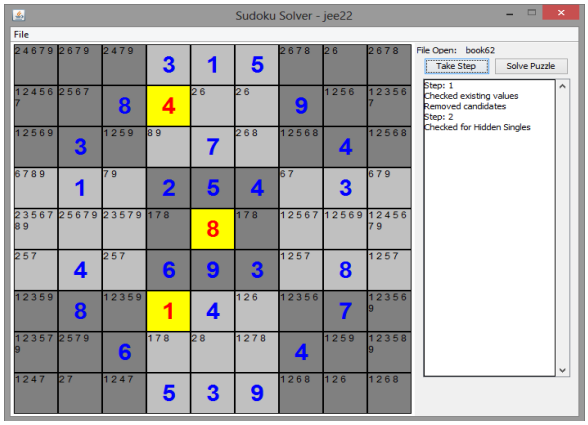
SS.002 – Loading .sud files



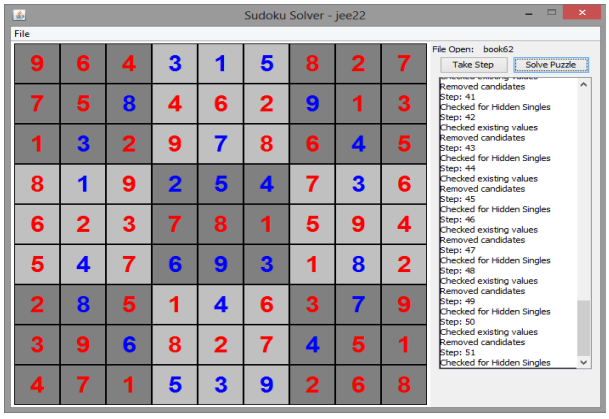
SS.003 – a loaded in .sud (book62)



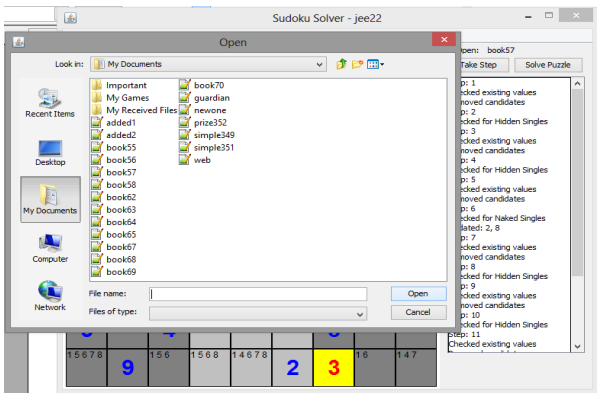
SS.004 – book62.sud after 'Take Step'



SS.005 – Seeing solved cells



SS.006 – book62.sud puzzle solved



SS.007 – Opening file while solving

Including both the Unit tests and my Test tables, my application passed all tests and succeeds in all use cases I originally set out to do for this application. There are still areas with the application that do not work so well however, and I will cover these in my conclusion and closing notes.

Conclusion

With the challenge of designing a good, maintainable data structure and solving algorithms for Sudoku and the short period of time given for this task, I feel that I have both learned a lot and done the best I could in achieving my desired outcome in attempting to meet the assignment requirements. There are a number of things I would change were I given more time or knew better however, but for what was given I feel that my application meets the set targets and the reasons for my data structure set up and algorithm implementations explanations are good.

In particular, I would work on finding more efficient and less complex ways of solving the puzzles. While I feel my current implementations are not terrible, there is a feeling that they can always be improved, and I hope in time I will come to know how.

I would also wish to work on the user interface a lot more. Right now it is the bare basics in what it shows the the user. If given the time, I would add labels to give the cells references, add a 'Back step' button, allow the user to attempt to solve the puzzle themselves, with checking from the program, and have buttons for the user to manually try and take different steps using the various techniques for solving Sudoku. More advanced techniques to solve the puzzles could be added too, and working on a better way of 'taking steps', rather than multiple nested if statements based on order of priority.

I would also work on making the application a little more robust, such as better error checking on the files it reads in (ensuring that they truly are .sud files). Currently, if it reads in a rogue file, it will start a new grid and throw an Exception. Although the application doesn't 'bomb', it would be nice to provide the user with some feedback that they have loaded an incorrect file, or not allow them to even see any files other than .sud, even when using 'Show: All files' in the file chooser.

However, since the GUI was not the core task of this assignment, I feel what I have created is enough to display the algorithms as they solve, what they do and where they do it on the grid.

The text updates and scroll bar could also be improved though. Firstly, the auto-scroll as the updates come through (JtextArea rather than Jlabels), and display the exact co-ordinates of not just Cells that get solved by themselves, but all Cells solved in a single step. Likewise, I would like to give the user more feedback on which candidates have been changed/removed.

In terms of the Data Structure, I would be tempted to revisit the 'allCollections' LinkedList and create my own class to support this, holding additional information about what Values are contained in each collection, whether that full collection was already solved and more. I may even make an Interface of 'CollectionTypes', and then have Rows, Columns and Blocks separately.

However, with this it would be more of a strain on the Space complexity of my application, as I would be holding multiple references to the same Objects, and I would perhaps have to call methods multiple times over in order to use it on each type of collection if I did keep them separate and not in the same single list and so I am happy with the implementation I have right now.

Overall, I am relatively pleased with my resulting application, given the time constraints. I am definitely still learning how to create better, more robust, cleaner, maintainable and easy to

understand code and applications, and this assignment has taught me more during the creation of the Sudoku Solver. My data structure allows simple and more complex solving techniques to be relatively easily written for andperformed, using minimal repeated code, and my algorithms attempt to solve the puzzle in the quickest time possible. I have enjoyed doing this assignment and pushing myself to better understand programming.

Assignment Feedback Form

Year: 2013 - 2014

Module: CS21120

Assignment Number: 1

Assignment Description: Sudoku Solver in Java

Worth 25% of final mark for this module
How many hours (approx.) did you spend on this assignment? 35

Expected Letter Grade: B

And why?
My code solution solves each puzzle, has a GUI to display the solving solutions and uses various techniques and algorithms in a logical priority order to solve all puzzles provided. I have provided testing for the application in the form of Unit testing my Model classes and providing examples of testing the final implementation of the application with test cases and black box testing. However, I feel that many of my algorithms may have too much complexity and could be cut down considerably, I am just not sure how at this point in time. I also feel that there are a number of areas in my application that are not as robust or easily maintainable as they should be. This is something I am working on improving as I am studying this module.
I feel my choices for the data structure for the application were good for the task at hand, although perhaps not ideal. Again, during the course of this module, I will be learning and improving my choices for data types in order to build better applications in the future.

What did you Learn?
That focusing on planning algorithms and data structures ahead of coding is very useful and gives a clear path to solving a problem. I also learned more time management, how to spread out my time when challenged with a difficult problem over a short period of time, and how sometimes it is better to not go for the complete solution, but the better solution (in this case, solving all of the Sudoku puzzles was not a requirement, but maintainable code was).
I certainly learned more about writing good algorithms that allow for no mistakes in their logic and how to build data structures that can be used and manipulated in multiple ways to reach solutions.