Northumbria University NEWCASTLE

A report submitted in partial fulfilment of the regulations governing the award of the Degree of BSc. (Honours) Computer Science at the University of Northumbria at Newcastle.

Project Report

Generation of Large Futoshiki Puzzles with Unique Solutions.

Software Engineering Project

Connor Campbell

W18003255

Word Count: 17371

2021/2022

1 Declarations

I declare the following:

(1) that the material contained in this dissertation is the end result of my own work and that due acknowledgement has been given in the bibliography and references to ALL sources be they printed, electronic or personal.

(2) the Word Count of this Dissertation is 17371

(3) that unless this dissertation has been confirmed as confidential, I agree to an entire electronic copy or sections of the dissertation to being placed on the eLearning Portal (Blackboard), if deemed appropriate, to allow future students the opportunity to see examples of past dissertations. I understand that if displayed on eLearning Portal it would be made available for no longer than five years and that students would be able to print off copies or download.

(4) I agree to my dissertation being submitted to a plagiarism detection service, where it will be stored in a database and compared against work submitted from this or any other School or from other institutions using the service.

In the event of the service detecting a high degree of similarity between content within the service this will be reported back to my supervisor and second marker, who may decide to undertake further investigation that may ultimately lead to disciplinary actions, should instances of plagiarism be detected.

(5) I have read the Northumbria University/Engineering and Environment Policy Statement on Ethics in Research and Consultancy and I confirm that ethical issues have been considered, evaluated and appropriately addressed in this research.

SIGNED: C.Campbell

[1]

[1]The word count of this paper when including figures is 19,936 due to formatting, 2,565 words have been deducted to negate this.

2 Acknowledgements

I would like to show my gratitude to my supervisor Mark Sinclair for his guidance throughout this project during development of the project, and the structure of the report.

I would also like to extend this to my parents for their support with not just this project but all of my academic studies.

3 Abstract

Currently, the generation of Futoshiki puzzles is limited in a few ways. Not many different generators exist and those that do are limited to making 9×9 puzzles. Along with this, research into solvers is minimal and where solving methods exist, solvers designed to work with Futoshiki are few; most Latin square puzzle solvers are designed to solve Sudoku.

The focus of this project is to tackle all of these issues by creating a generator that can make Futoshiki puzzles larger than 9×9. The project was built on a piece of supplied code by the project supervisor (Sinclair, M.C, October 2021) that included the structure of Futoshiki and a simple solver. This was done using a generator to the create puzzles and a backtracking solver to verify them, checking if the puzzle is solvable and that there is only one solution. This implementation was done in Java.

The success of this implementation is complex as the exhaustive approach taken proves too slow for the comprehension of more complex and larger puzzles. However, this is evaluated throughout the report and the potential reasoning for the shortcomings of the solver are commented on in the evaluation.

Overall the project provides a solid foundation for the generation of larger puzzles and a critical evaluation of its implementation.

Contents

4 Introduction

Japanese pencil puzzles such as Futoshiki (*Futoshiki* 2022) and Sudoku (*Sudoku* 2022), are a form of Latin square puzzle (Chlond 2013) in which a constraint-based puzzle is created. One of the largest players in publishing these puzzles is Nikoli (*Nikoli* 2022), developing the Sudoku which is now distributed in over 120 countries. The reason for the popularity increase can be put down to Wayne Gould, who developed a program that could quickly generate puzzles making publishing them far easier as opposed to manually designing them (Shortz 2006).

Past this, puzzles were developed to see if they could be made more difficult. A common way of doing this was to increase them in size, this can be seen on *Janko Futoshiki* (Angela and Janko 2020) when looking at the more difficult Futoshiki they are always of size 8 or 9, larger than the easier variants that can be seen at size 4 or 5, however no larger. Similarly, large Sudoku exist some containing alpha-numeric characters (*Puzzle Madness* 2022)

A key feature of Futoshiki puzzles is that they have a unique solution, because of this constraints must be used to eliminate other solutions. To do this, generators must be able to solve puzzles alongside creating them to verify that the puzzle is both solvable and that the solution found is unique.

4.1 Aims and Objectives

4.1.1 Aims

To develop a backtracking solver with the ability to construct a tree of solutions to a given Futoshiki puzzle.

The backtracking solver is required to find solutions that may be missed by the solver within the supplied code.

To study how the manipulation of constraints affects the completion of Futoshiki puzzles.

Understanding how constraints affect the solvability and feasibility of puzzles will be key to properly assigning constraints to the puzzle.

To generate large Futoshiki puzzles with unique solutions.

With both of the initial two aims combined, this will be the overall goal of the project.

4.1.2 Objectives

Investigate available literature pertaining to Futoshiki puzzles.

Investigate literature for solutions within other Latin puzzles that may have transferable ideas.

Examining key literature will give a vast understanding of how the program will be built and how it can be sculpted to best meet the aims. This means expanding research beyond Futoshiki and into other Latin square puzzles.

Design a pseudocode solution.

Design a UML class diagram.

The creation of planning and documentation artifacts allows for the program to have a more structured design, and where it is necessary, it can be used to assist in understanding the code that has been developed.

Develop decision trees for the completion of puzzles.

In solving the puzzle, a decision tree must be made to mark out the solving order that the solver takes to ensure duplicate routes are not explored.

Develop a system to add inequalities based on the instance's decision tree.

Develop a system to adjust starting values based on the instance's decision tree.

To create a Futoshiki, constraints need to be added to the Latin square in order to populate it. Adding them based on the decision tree will speed up generation overall.

The program does not break memory requirements.

The program has been optimised to ensure it has minimal runtime.

Ensure the decision tree is updated based on the new constraints and not fully remade.

All of these objectives relate to the efficiency of the program and ensuring high-level build quality to improve performance. Following these will result in fewer crashes and faster runtime.

The program can export generated instances in a readable format.

To ensure that the puzzles are usable by the user post-generation, exporting in a readable format to the user will be needed as the program will not handle them in an easily interpreted manner.

An evaluation of the success of the program.

To properly assess the quality of the program, an evaluation of its success against the aims and specific goals will be needed. This will allow for the quality of the program to be assessed fully.

The program has been appropriately tested.

To ensure that the program functions properly, appropriate testing will be needed. This also allows for an analysis of program performance, making the evaluation of its success more focused.

Documentation of discoveries through development. Completion of the final report.

Denoting how the program was developed and discoveries met is key to assessing and reflecting on how development was undergone.

4.2 Product Overview

The product is designed to be a Futoshiki generator, that can generate puzzles with unique solutions larger than size 9. This will allow for greater diversity in the puzzles and adds a new level of complexity not before seen in Futoshiki puzzles.

As discussed earlier, creating the puzzles will require two components, a generator to create the puzzles, followed by a solver to check if the puzzle only contains a single solution to verify its uniqueness. The generator approach taken is random, flooding the Latin square with a fixed number of constraints and then handing this puzzle to the solver for verification. The generator will adjust the number of constraints to be added based on the size of the puzzle, adding less to smaller puzzles, and more to larger ones as they have both more availability for assigns and inequalities.

The method of solving used is backtracking as it offers a complete scan for solutions as it will try every possible solution, whereas using solving methods alone allows for gaps. The solver has solving methods, and it applies these at every stage that the solver attempts to add values in order to find solutions. This ensures that if solutions are available, the backtracker will find them through a brute force style approach.

4.3 Background and Motivation

Puzzle-solving and generation is a driving force in algorithms and particularly Futoshiki solving algorithms have been used in real-world applications such as optimisation of photovoltaic arrays (Sahu, Nayak, and Mishra 2016). The limitation of Futoshiki occurring at

9×9 is seemingly unnecessary and there is no conclusive research that disproves that puzzles larger could exist similar to how they do with Sudoku. Even academics who have conducted several larger studies into the workings of Futoshiki such as Kazuya Haraguchi (Haraguchi 2013; Haraguchi and Ono 2014), have not approached nor disproved the idea that larger puzzles could exist.

The primary motivation is the gap for a program that is able to surpass this size 9 limit, and to push Futoshiki generation to its limit.

4.4 Approach and Tools

The project was developed entirely in Java using the Eclipse IDE, built on top of code supplied by the project supervisor (Sinclair, M.C, October 2021).

5 Literature Review

Throughout, the notation of $\mathrm{R_k}$ and $\mathrm{C_k}$ will be used to refer to row k or column k respectively. Accordingly, $\mathrm{R_kC_k} < \mathrm{R_{k+1}C_k}$ would show the inequality between two cells.

5.1 General Latin Square Puzzles

5.1.1 Features

Latin squares are a grid of $n \times n$ filled with values, where each column and row respectively has no duplicate values (Burn 1975). For the use as a puzzle, a filled Latin square is mostly useless as filling the square is traditionally the goal of the puzzle (Palomo 2016). The starting form of a puzzle typically falls into one of the following forms (Palomo 2014). Examples of the following can be seen in Figure 1.

Partial Latin Square (PLS), a grid containing values in some cells in which there are no repeated values in an individual. See Latin Square 1.

Completable Partial Latin Square (CPLS), a PLS, in which every cell can be filled without breaking any of the rules for a Latin square. See Latin Square 2.

Uniquely Completable Partial Latin Square, a CPLS with a single solution. See Latin Square 3.

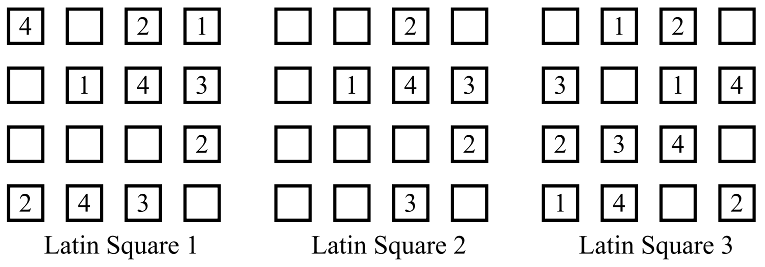


Figure 1: *Latin Square Forms*

5.1.2 Sudoku

A common form of CPLS is Sudoku. A Sudoku is a CPLS of order $n \times n$, though traditional Sudoku size is 9×9, this is made up of smaller grids of order $\sqrt{n} \times \sqrt{n}$. Where the larger Sudoku follows CPLS rules, the smaller grids individually may not contain duplicate values and can only use values from 1 to n (Ercsey-Ravasz and Toroczkai 2012). In the following section some solving methods can be found (*Sudoku techniques* n.d.).

5.1.2.1 Scanning In solving, the constraints are used to evaluate the candidate numbers in each cell, and from here cells can be narrowed down to one or a few values to progress solving. The new values added act as new constraints. Figure 2 demonstrates solving of Sudoku, starting from Sudoku 1, R_8 has two candidate values of 2 and 9 that can go in either C_3 or C_9. As C_9 already contains a 2, C_9 must be 9 forcing C_3 to contain 2. This can be seen in Sudoku 2. The values entered in Sudoku 2 can be used to fully solve the bottom right grid. The solving of this grid can be seen in Sudoku 3 and the basis of the solving in Table 1.

Constraint	Causes
$7 \in R_7$	$R_9C_7 = 7$
$3 \in R_8$	$R_7C_8 = 2$
$3 \in R_8$	$R_7C_7 = 3$

Table 1: *Sudoku Solving Table*

5.1.2.2 Naked Pairs When looking at the candidate numbers for a cell, a position can be encountered in which two cells in a square have only the same two candidate numbers, creating a naked pair. In solving, this is useful because while the cells remain unsolved another cell may be put in the position to be solved due to the ability to treat the naked pair as solved temporarily. An example of naked pairs in use on a 3x3 cell can be found in Figure 3.

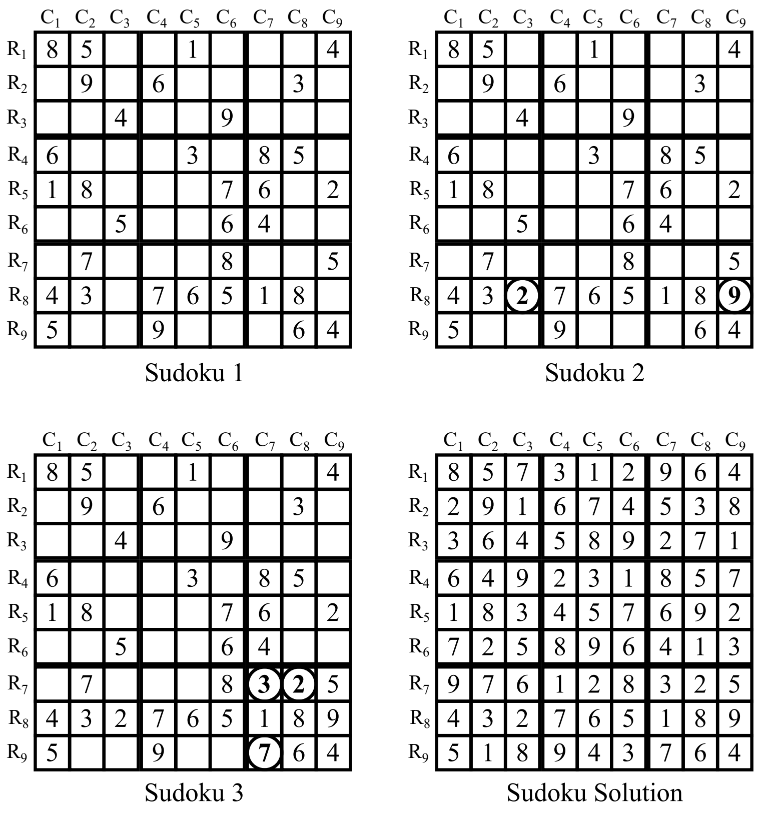


Figure 2: *Sudoku Solving*

5.2 Futoshiki

5.2.1 Tutorial

Solving Futoshiki is similar to solving Sudoku, each cell has a set of candidate numbers and from this one is chosen to input a value that will add new constraints and change the other candidate numbers. In Futoshiki candidate numbers are found using both the column and row rules from Latin squares but also the inequalities that exist between the cells (Lloyd et al. 2021).

The solving order of a Futoshiki can be seen in Figure 4. The base puzzle is seen in Futoshiki 1. The first row solved is R_4, of the constraints on cells R_4C_2, R_4C_3 and R_4C_4, R_4C_2 has the candidate values 1 and 2. Due to $1 \in C_1$, $R_4C_1 = 2$. This causes $R_4C_2 = 1$, $R_4C_3 = 3$ and $R_4C_4 = 4$. This progression can be seen in Futoshiki 2. The final solution can be seen in Futoshiki Solution, with the full solving order in Table 2.

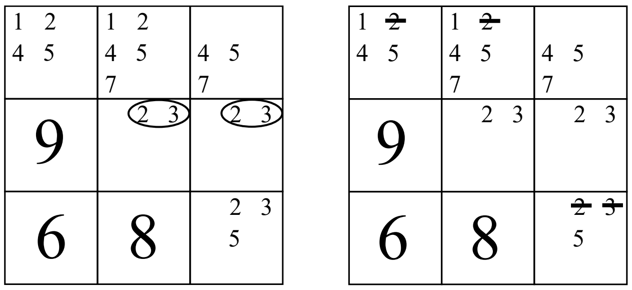


Figure 3: *Naked Pairs*

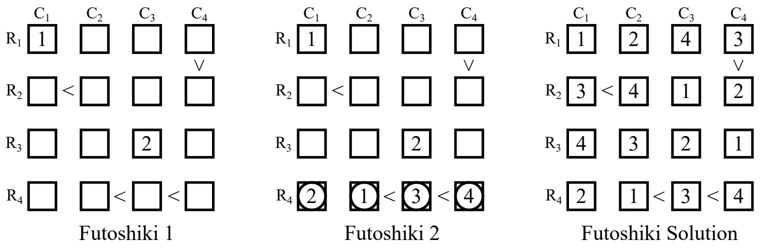


Figure 4: *Futoshiki Solving*

5.2.2 Features

One of the commonly found features of Futoshiki caused by the constraints is the relation of respectively adjacent cells through inequalities, coined as the ascending chain rule (ACR) (Berthier 2013). In an $n \times n$ Futoshiki, a chain of length k would be identified by ($1 < \mathrm{k} < \mathrm{n}$), containing a sequence of k+1 cells each related by the '$<$' inequality. This allows the cells to not be restricted to specific rows and columns thus allowing for cells $\mathrm{R_kC_k}$ and $\mathrm{R_{k+1}C_{k+2}}$ to relate where they typically could not. The chain in this example could be something like the following: $\mathrm{R_kC_k} < \mathrm{R_{k+1}C_k} < \mathrm{R_{k+1}C_{k+1}} < \mathrm{R_{k+1}C_{k+2}}$. This can be extracted from the Futoshiki to form a feature of its own and be analysed independently if necessary.

Let $\mathrm{R_kC_k} < \mathrm{R_{k+1}C_k} < \mathrm{R_{k+1}C_{k+1}} < \mathrm{R_{k+1}C_{k+2}} < \mathrm{R_{k+2}C_{k+2}} < \mathrm{R_{k+2}C_{k+3}}$, be ($C_0$, C_1, ..., C_4). If the Futoshiki in use is 7x7, ACR would dictate that the range of possible values for C4 would be 5, 6 and 7 eliminating over half of the available values allowed by traditional LS rules. Figure 5 shows how this rule can be applied to reduce solutions in a full chain.

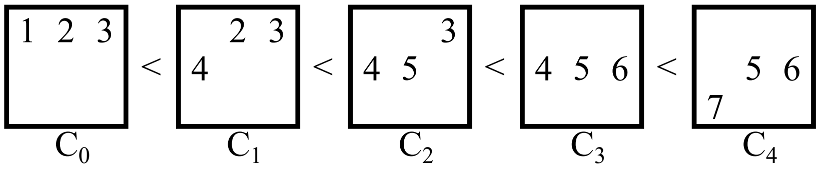


Figure 5: *Ascending Chain Rule*

Another set of features to be identified are hills and valleys. A hill can be defined as a pair of ascending chains of potentially different lengths but in the same row or column. The chains of length k, are defined as (C_0, C_1, ..., C_k) and (C'_0, C_1, ..., $C'_{k'}$) with the length of these

Focus	Reason	Changes
C_3	$1 \in R_1$	$R_2C_3 = 1$ && $R_1C_3 = 4$
R_2	$R_2C_1 < R_2C_2$ && (1,2) $\in C_1$	$R_2C_2 = 4$
C_1	$4 \notin C_1$	$R_3C_1 = 4$
R_2	$2 \notin R_2$	$R_2C_4 = 2$
R_1C_4	$R_2C_4 < R_1C_4$ && $4 \in R_1$	$R_1C_4 = 3$
R_1	$2 \notin R_1$	$R_1C_2 = 2$
C_2	$3 \notin C_2$	$R_3C_2 = 3$
R_3	$1 \notin R_3$	$R_3C_4 = 1$

Table 2: *Futoshiki Solving Table*

chains equating to 1 = k – k' (Berthier 2013).

This creates two rules for eliminating results from a hill or valley. Applied to an $n \times n$ Futoshiki, in a hill, m being the smallest value available in C_0 and C'_0, from possible values for C_k between 1 and m+k+k'-1 can be eliminated. The valley rule is similar, however, consider M to be the largest value in both C_k and $C'_{k'}$, from C_0 the candidate numbers n-M-(k+k')+1 and n can be removed.

In 6 application of the hill rule can be seen. The first row is the structure prior to the application of ACR and hill rule. In the second row, ACR is applied removing some values prior to the application of hill rule. C_k being the seventh cell and $C'_{k'}$ being the second cell this creates the following conditions: k = 2, k' = 3, k + k' = 5, n = 9.

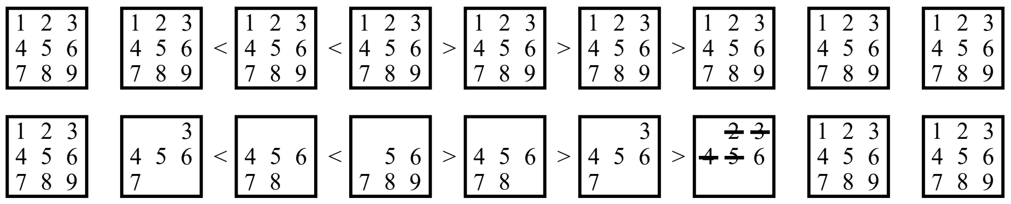
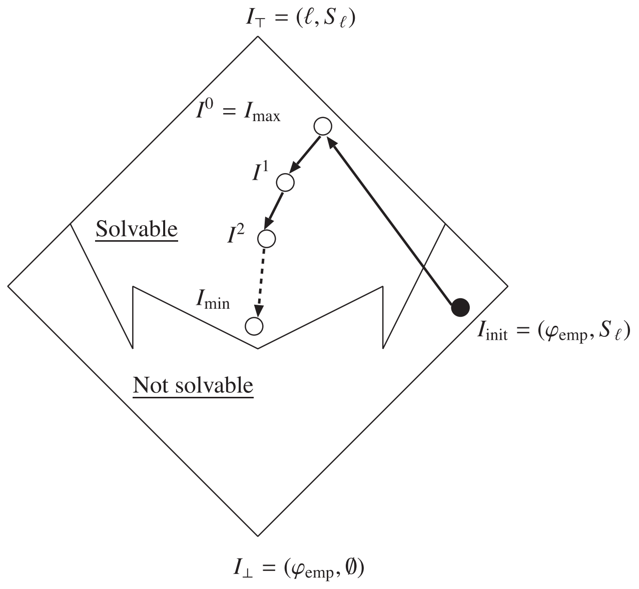


Figure 6: *Hill Rule*

5.2.3 Difficulty

Deciding the optimal number of inequalities contained within a puzzle is dependent on the difficulty. Placing too many inequalities may make the puzzle too easy whereas having too few can push the puzzle to being unsolvable. Figure 7 contains a Hasse diagram (Hansen 2004), from "The number of inequality signs in the design of Futoshiki puzzle" (Haraguchi 2013), detailing the impact of the number of inequalities on the solvability of a Futoshiki. Where I_{init} is the completely blank puzzle with no values input, I_{max} is a puzzle with the maximum possible number of inequalities, I_{min} is the minimum amount while the puzzle is still solvable, I_1 and I_2 exist in between I_{min} and I_{max} on a spectrum of I_{max}, I_1, I_2 to I_{min}.



$I_\top = (\ell, S_\ell)$
$I^0 = I_{\max}$
I^1
I^2
Solvable
$I_{\min}$
$I_{\text{init}} = (\varphi_{\text{emp}}, S_\ell)$
Not solvable
$I_\perp = (\varphi_{\text{emp}}, \emptyset)$

Figure 7: *Haraguchi's Hasse Diagram (Haraguchi 2013) on the solvability of Futoshiki by number of inequalities.*

5.3 Futoshiki Solving Algorithms

There are several different algorithmic approaches to solving Futoshiki. Haraguchi discusses two different methods in a *Greedy Algorithm* (GA), *Local Search* (LS) and a *Matching Based Algorithm* (MBA) (Haraguchi and Ono 2014). The JPOP framework is designed for combinational black-box optimisation (Lloyd et al. 2021), however as it uses Japanese pencil puzzles, such as Futoshiki, as a method for testing efficiency and effectiveness, the algorithm used is comparable to GA and MBA.

The premise in which the GA, LS, MBA and JPOP operates is as follows:

GA A random cell is selected from the puzzle, from the set of candidate numbers a random one is selected. This is repeated until a solution is created, if the solution is invalid the algorithm will restart from the initial puzzle prior to values being input and start the algorithm again. The algorithm will never use the same set of candidate numbers twice. This is repeated until a complete solution is made (Haraguchi and Ono 2014).

MBA Cells in the puzzle are assigned the value k in the order (1, 2, . . . , n). Which cells get assigned a value, is decided by using the maximum matching values with the ones made using the graph ($\mathrm{R} \cup \mathrm{C}$, $\mathrm{E_k}$) where $\mathrm{E_k}$ is an edge set evaluated by comparing the maximum available values of the current solution against the values of a blank puzzle (Haraguchi and Ono 2014).

LS Using a traditional local search algorithm (Ishibuchi and Murata 1996), similar to those found in AI (artificial intelligence) algorithms, sets used within the puzzle are replaced repeatedly until a solution is found (Haraguchi and Ono 2014).

JPOP From all of the candidate numbers available the solver will select one to use. After this a new set of candidate values are presented to the solver and selected from this set again. This process is repeated until the complete solution is found (Lloyd et al. 2021).

Where MBA and LS offer higher yields in terms of efficiency and runtime, the complexity of their design may exceed the scope of the project. GA and JPOP are very similar in terms of taking a more brute force solving approach and where this may not have the same benefits of shorter runtime and lower memory requirements, this can be made up for in other ways, see Section 5.6.

Futoshiki can be made more difficult by increasing the amount of candidate numbers for a cell, this can be done by reducing the number of starting values or inequalities (Haraguchi 2013). Because of this, defining the difficulty of a puzzle can be done through the size of the decision tree it has, the more child nodes each node has, the more potential options there are for values and the longer the branches the more candidate numbers created by the new node chosen.

5.4 Generating Puzzles

5.4.1 Constraint-Based Generation

When generating puzzles, adhering to constraints is essential to ensuring both puzzle feasibility and quality, "Step-wise Explanations of Constraint Satisfaction Problems" (Bogaerts et al. 2020) discusses the development of a crossword generation algorithm that does not use constraints. This works by starting with a list of words to use, randomly selecting a word to be the starting word for the puzzle the iterating through the list searching for ones that could be used in intersections. This is noted as being a greedy approach as constantly iterating through the list is sub-optimal.

Constraints can be used to assist with this issue. In any form of puzzle generation a set of potential assigns (PA) are available, constraints can be used to narrow this set down massively. PA are the values that are valid inserts into specific cells, for example in an empty Latin square of size n = 4, values 1 - 4 are PA for any cell, however if a 3 was to be inserted into R_1C_3, any PA that indicate inserting a 3 into a C_3 or R_1 cell are removed, this can be seen in Figure 8, Puzzle 1 showing prior to adding a value and Puzzle 2 showing 3 being added to R_1C_3. Solving methods can be applied to reduce the number of PA (Sun et al. 2008) such as those seen in 5.3.

An advancement of this can be seen when applying genetic algorithms. Genetic algorithms are computer-based optimisation methods used to progress the quality of a product (Tsai and Chou 2011). GA attempt to employ aspects of evolution through natural selection similar to Darwinism, this involves mutation and and crossovers of genes from a predefined data set

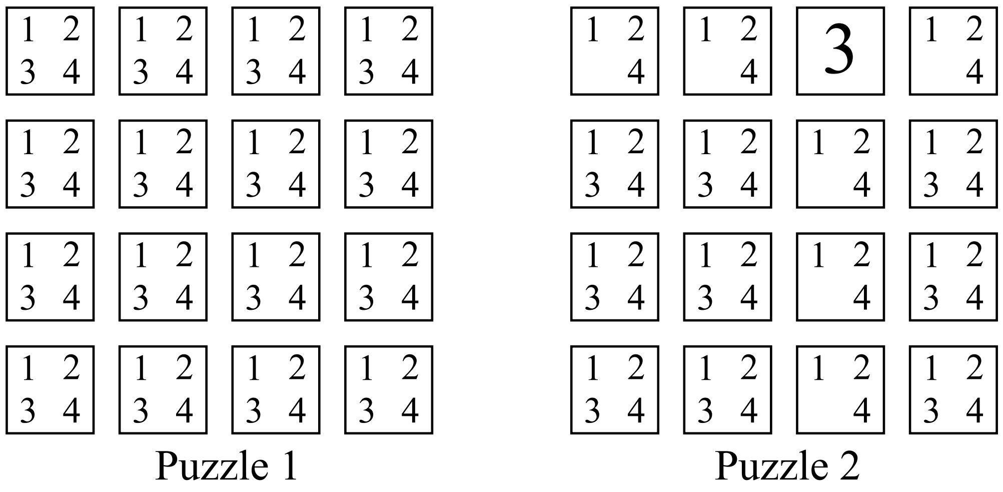


Figure 8: *Potential Assigns.*

(Yang 2021). This allows for solving of complex non-convex problems that cannot be solved through linear means such as a more traditional algorithm or mathematical function (Chau et al. 2017). In the case of puzzle generation, genes will be identified in the form of features of the puzzle, and then from this mutations can be made to make new puzzles.

"Solving, rating and generating Sudoku puzzles with GA" (Mantere and Koljonen 2007) discussed the generation of Sudoku puzzles using genetic algorithms, this approach took a solved puzzle, removed numbers and then attempted to solve it. Though it was noted in the study that their approach in places made it difficult to verify the uniqueness of the puzzle, the greater focus of the study was to rate the quality of puzzles. This was done by after the puzzle was generated, the puzzle is solved and the number of stages the solve was completed in is taken, this is repeated to take an average number of stages to represent the difficulty rating of the puzzle.

5.4.2 Procedural Generation

The form of generation being used here to create puzzles is a form of procedural generation akin to that which can be seen in modelling. Procedural generation is a method of creating data using an algorithm, rather than manually creating it (Emmanuel et al. 2019). The reason this is used so often in modelling is that it can help to simulate the randomness that can be seen in real-world environments. "Algorithms for procedural generation and display of trees" (Nuić and Mihajlović 2019) discuss the importance of this when modelling trees in video game environments. Algorithms can be applied to ensure that when generating the trees, a different tree is constructed every time, without the artist having the remodel it. The focus of this is the algorithm's ability to operate within a set of bounds to output a different result while still abiding by the criteria initially set.

The most common usage of procedural generation can be seen in video games (Viana and

Santos 2019). In games that involve randomly generated levels often procedural generation is used to allow for the creation of authentic feeling levels that can be created on-demand, an example of this can be in the level design in The Binding of Isaac (BorisTheBrave 2022). In traditional puzzles research is limited however, a collection of puzzles called Simon Tatham's Portable Puzzle Collection (Tatham 2022) exists which is a collection of 40 puzzles that all use procedural generation to create new variations of the puzzle on demand.

While the algorithms discussed in "Algorithms for procedural generation and display of trees" (Nuić and Mihajlović 2019) do not directly apply to the case of Futoshiki generation, the fundamental function does. Without the randomness that procedural generation applies.

5.4.3 Generation Tree

The structure of generation can be broken down into a tree, with each branch being a different path that the generator could go down and the nodes representing a state that the puzzle can arrive at, eventually reaching a point where a puzzle is created. How far down the tree the generator should go before deeming the puzzle to be a solvable or adequate difficulty is far more abstract (Hunt, Pong, and Tucker 2007) as going too far down the tree will likely lead to an unsolvable puzzle, however too early in the tree multiple solutions are likely to occur, but the concepts to be applied in deciding this are discussed in Section 5.2.3. An example of a generational tree for a Futoshiki can be seen in Figure 9, note this would not be a full tree as this would be far larger, but is designed to give an idea of how stage by stage a puzzle would be built.

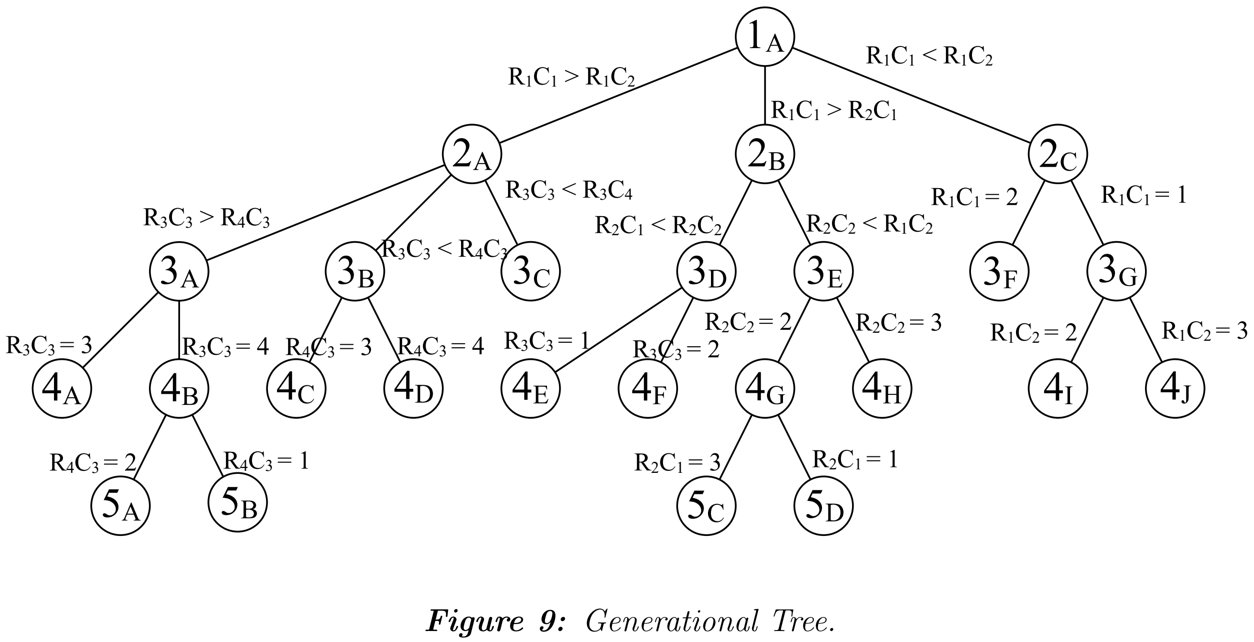


Figure 9: *Generational Tree.*

5.5 Backtracking Solvers

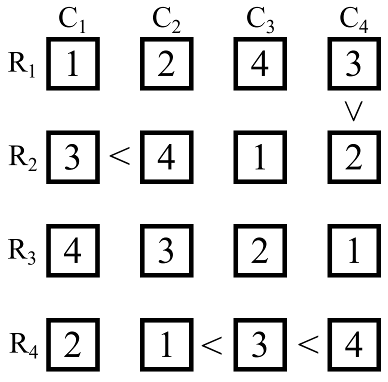
To verify the quality of the puzzle, a backtracking solver must be implemented. This is the process of solving through a tree structure (Esteve et al. 2021). Similar to the methods discussed for generation in Section 5.4.2, however in this case rather than stopping at a point in the tree, the whole tree will be generated.

Backtracking takes a brute-force approach to solving by taking every PA that could be inserted, selecting one, updating the potential, and inserting a new one (Schottlender 2014). Break points can occur in three situations,

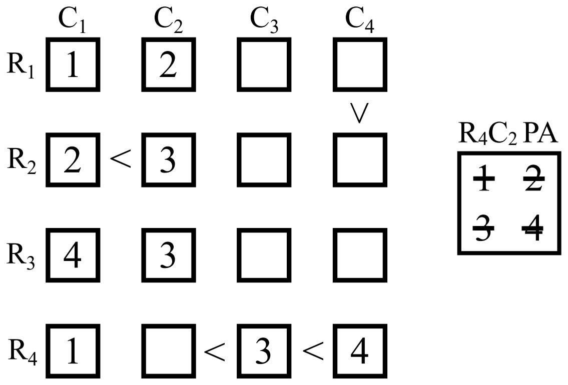
Solved The puzzle has been solved, each cell contains a value. An example of this can be seen in Figure 10a.

No PA The puzzle is not solved and there are no more potential assigns at the current depth.

Infeasible At least one of the cells has no potential assigns, the puzzle cannot be solved. An example of this can be seen in Figure 10b.



(a) Solved Futoshiki Puzzle.



(b) Infeasible Futoshiki Puzzle.

Figure 10: *Futoshiki Backtracking Break Points.*

In the case of both of the papers discussed, "The effect of guess choices on the efficiency of a backtracking algorithm in a Sudoku solver" (Schottlender 2014) and "Heuristic and Backtracking Algorithms for Improving the Performance of Efficiency Analysis Trees" (Esteve et al. 2021), once a solution had been found backtracking would cease. Generation differs as there should only be one solution to the puzzle, meaning that the break point in generating would be found at solved, but backtracking not does cease similar to how it would in solving, the depth of the backtracker will decrease and backtracking will proceed to search for other solutions.

Research in this area is limited as the majority of studies have either gone down the route of using ML (machine learning), similar to that discussed in "Solving, rating and generating Sudoku puzzles with GA" (Mantere and Koljonen 2007), or AI will be used to assist in generation, such as those discussed in “Experimental comparison of uninformed and heuristic AI algorithms for N Puzzle and 8 Queen puzzle solution” (Mathew and Tabassum 2014). Some of the features applied by ML and AI algorithms can be applied such as search trees and depth based solving, however many of them exceed the scope of the project due to this, the application would exceed development means and time.

5.6 Methods for Limiting Memory Use

Due to the size of the trees being generated, memory usage will be of concern. Memory usage is controlled by both the memory management within the programming language and the methods used within the algorithm to reduce memory usage. Programming can fall into a few different paradigms, however only OOP (object-oriented programming) and functional are the two chosen to be discussed.

5.6.1 Programming Paradigms

5.6.1.1 Object-Oriented Programming According to “Object-oriented programming: Themes and variations” (Stefik and Bobrow 1985), OOP revolves around the transfer of data between objects. In the case of this, objects are devices of storing data and procedures, they will restrict what data is available to the other objects in order to improve runtime and efficiency, this is where the idea of object inheritance comes from.

OOP takes a more logic-based approach to development making it often the more desirable approach to learning how to program as it has the lowest barrier to entry (Wegner 1990), some of the most popular pure OOP languages include Objective C (Rawlings 1989) and Smalltalk (Goldberg and Robson 1983), however most of the popular OOP languages such as Java and Python include functional features, so while not as pure as the prior mentioned, in practice they are the preferred OOP languages.

5.6.1.2 Functional Programming Functional programming takes a more mathematical approach to development, functions are not sub-routines as they would be in OOP. “The Promises of Functional Programming” (Hinsen 2009) discusses the most noticeable features of functional programming; functional will always return the same result from a function, akin to how they would in mathematics, Similarly variables are not used, if a piece of data is to be stored it is done so in a specific memory location.

Though it is typically more difficult to program in, the greater control over aspects such as memory often results in vastly better runtime and more efficient memory usage (Alic, Omanovic, and Giedrimas 2016).

Functional programming’s popularity has been decreasing over time due to its higher entry level than OOP and the introduction of features that were associated with functional being added too OOP such as multi-threading, but a few of the more used ones include LISP (Steele 1990), Scheme (Dybvig 2009) and Haskell (Hudak and Fasel 1992).

5.6.1.3 Multi-Paradigm Since different programming paradigms have advantages, some languages exist as a middle ground between paradigms employing some of their best features (Sargeant 1993). Typically, languages lean more into one paradigm than the other, for example, Java is more OOP however it does have some features of functional programming such as lambda expressions (Tsantalis, Mazinanian, and Rostami 2017). Similarly, C++ has some of the abstraction features that would be typical of OOP (Weir 1993), while still supporting features of procedural programming.

Based on this, the best approach to use for puzzle generation would be to use a multi-paradigm language and to take advantage of the features offered that can lean into the functional side of development, the code supplied as a foundation for the generator is in Java meaning that leaning into the functional aspects of Java will be crucial.

5.6.2 Memory Usage Reduction Within Java

Java memory usage is automated, using a method called garbage collection (Pizlo and Vitek 2008). This works by the JVM detecting when memory is low and will select objects from the heap to delete, however, there is no guarantee that the correct objects will be deleted due to this process being automated. C++ alternatively, allows for direct memory allocation which allows for the programmer to control how memory is managed. Java does however have design patterns programmers have developed to help reduce memory usage. Two of these are discussed in this section, those include Flyweight in-depth and briefly Proxy.

5.6.2.1 Flyweight The objective of Flyweight is to reduce the number of objects but increase the volume at which they can be used (Baca and Vranic 2011). This involves creating a shared object called a flyweight that can be used in multiple different contexts at the same time but act independently. This involved two parts, an extrinsic state and an intrinsic state (Vonthron, Koch, and König 2018). The extrinsic component acts as the content for the flyweight, holding the information to be used. The intrinsic component represents a separate part of the object that will be shared and when combined they will create a complete object. Modern web browsers display this behaviour with images, they will load the image once, and from then on will simply refer to the image already loaded to draw it again.

Design patterns : elements of reusable object-oriented software (Gamma and Helm 1995) discusses that the most prevalent use of this can be seen in most text editors. Each character will be stored as a Flyweight but will have no context as to where it is positioned in the text, this will be controlled at the parent level, thus for storing a word the editor would simply store pointers to the set of flyweights that would represent each character. For scale, this means that every time a character in a sentence is repeated, storage space is being saved. Particularly with smaller objects this can be useful, as rather than requiring to deal with large memory addresses, much smaller indexes can be used to refer to data resulting in massive reductions in memory usage.

The advantages in efficiency are noted by "Understanding the impact of object oriented programming and design patterns on energy efficiency" (Maleki et al. 2017) showing a 50.3% decrease in energy consumption when applied to a system running a high complexity program. This can have drawbacks on runtime (Gamma and Helm 1995), however in most scenarios, it is going to be faster to simply create a new object rather than searching for one that fits the criteria being put forward. The trade-off in this however is that saving memory is going to cause the program to run slower, so for the situation, the question must be asked if the improved memory efficiency is worth the extra runtime.

5.6.2.2 Proxy Proxy functions by creating a proxy of an object and whenever the original object is to be accessed, it will be done so through the proxy instead (Li and Q. Wang 2009). Having a controller object will allow for faster runtime as accessing the object directly, particularly in larger data structures, can be slow.

Since Proxy's primary focus is on security as it prevents objects from accessing original copies of other objects, researching into runtime improvements is small however noted by (Sheng and Y. Wang 2018) in cases it can see noticeable increases.

5.7 Testing for Correctness of Research Code

Testing exists to find possible errors within systems. While in a typical system you may be able to test simply by observing how it behaves and interacts, in software development it is far more common to create a testing environment in which the program is automatically tested. This is because many of the fundamental aspects of a program cannot always be seen at the user level such as function returns or variables.

The testing strategies discussed in this section are, Black Box and White Box, and the testing methods are focused on Unit, Integration and System testing (Nidhra and Dondeti 2012; Singh 2011). Other testing methods exist in a similar area to the ones focused on in this analysis such as Compatibility, Usability and Security, however due to the nature of the study these are not applicable to development.

5.7.1 Black Box

Black box testing has three main components, the input, an expected output and the actual output. This is important at all stages of development as they can act as constant informants that aspects of the software are working correctly, this restricts Black Box to focusing on testing the functionality of code (Khan et al. 2011).

By creating scenarios that assume the worst theoretical situation software can encounter, testing that the program can successfully handle these is key making robust software. Because in cases programs have a large number of potential inputs, testing all of the inputs is infeasible, thus choosing which input values to test is crucial. Two methods of this are Boundary Value Analysis (BVA) and Equivalence Class Testing (ECT) (Reid 1997; Singh 2011).

BVA By taking values in the upper and lower bound of the set of potential inputs, which according to *Software Testing* (Singh 2011), the likelihood of encountering errors is increased, and testing these, edge cases in which the software may not have been equipped to deal with can be tested for.

ECT Separating the set of potential inputs into groups that are functionally similar should return the same result, theoretically only one from each of these groups should require testing.

For example, a program can only compute values between X_1 and Y_1. ECT would declare three groups, below X_1 as invalid, between X_1 and Y_1 that are valid, and the last group of values above Y_1 that also return invalid. From this, it is known that BVA would be applied to the second group, slightly above X_1 another boundary is declared called X_2, and a second extra is added slightly below Y_1. The testing values will now be defined as values between X_1 and X_2, then another set of tests on the values between Y_1 and 2_2. Figure 11 shows how this may look.

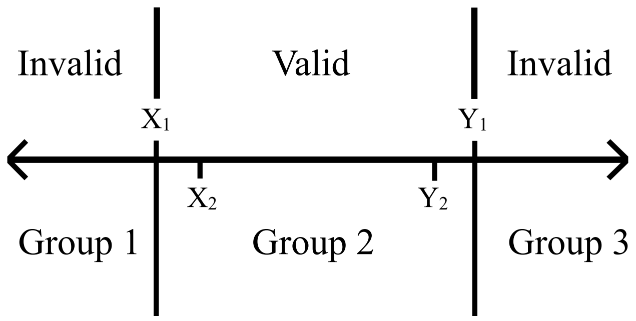


Figure 11: *BVA with ECT Diagram.*

5.7.2 White Box

Where Black Box is concerned with the functionality of the software, White Box is focused on the structure of the software. To separate, Black Box looks at if the correct software is built focusing more on the requirements, White Box checks if the software is built correctly, focusing more on implementation (Nidhra and Dondeti 2012). The main difference between the two is that black box testing cannot see the code, but White Box can. The goal of this is to be able to have a more detailed breakdown of the issues within the program in order to potentially have a better idea of how to repair the issue, it is often essential in debugging efficiently. Two of the methods of this include Desk Checking (DC) and Control Flow Testing (CFT) (Hayardeny, Fienblit, and Farchi 2004; Singh 2011).

Desk Check This involves denoting how an algorithm functions, manually calculating what results it should output, and then testing that the program outputs the predicted results. In order to be effective throughout development, these are planned as early in the development cycle as possible.

CFT By creating a model of how the program should run, or a control flow graph, test cases can be created at each stage of note in the program and these can be independently tested. This aims to ensure that in testing all areas of the code are accounted for.

CFT in the case of puzzle generation can be difficult as depending on the paths that the generator may go down, different stages may be arrived at, because of this either selecting the paths most likely to fail or taking more care in selecting points to test is important.

In combining these, using CFT, a plan can be created showing where Desk Checks should be completed. In the case of a puzzle generation, the tree that should be mapped out by the generator can be manually planned using the algorithm that is to be used, then when development arrives at the stage where the program can be ran, testing that it follows the same path as the algorithm using Desk Checks that have been plotted based on the control flow graph.

Fitting similarly into this category is design-by-contract (Meyer 2002). This involves giving assertions to various classes and functions to ensure that while running they run correctly, for example, if a function should only run using a value between 30 and 50, if the value given to the function is not within these bounds an error will be thrown. This makes it easier to pinpoint exactly where the code went wrong, as rather than the function attempting to run with the improper value resulting in a failure at an arbitrary location later in the code, an error will be returned stating that the value was improper in the specific function.

5.7.3 Unit Test

Unit testing is a form of White Box testing that involves testing specific elements of the program such as function returns (Singh 2011). This involves setting up a series of tests with an assertion in the result (Hamill 2004), in doing this, specific break-points and failures in the code can be pinpointed. The main issues with unit tests are that writing a test for every function in the program can take a very long time, and as the program develops, these tests may need to be updated.

For example, if a function is designed to return the numerical value of alphabetical letters ($A = 1$, $Z = 26$), an assertion could be set so that if D is input, 4 is always returned. In a situation where a program returns the wrong values but still runs seemingly correctly, the function causing the error is clear.

5.7.4 Integration Test

Integration testing is focused on the transfer of data between areas of the program (Singh 2011). This causes Integration Tests to fall into both Black Box and White Box testing. Creating interfaces between areas of the programs creates coupling a side effect of this, particularly in programs with high coupling, opportunity for error in the transfer of data increases as the number of dependencies begins to increase. To avoid this, high coupling is generally avoided and only the minimum amount of data necessary is transferred between units.

Encapsulation from OOP can help with this (Snyder 1986). Encapsulation refers to the ability to hide elements in classes that may only be required within the class and not be required by a class that may interface with it. For example in Java, setting a function to private only allows it to be used within the class and public allows it to be used by anything interfacing with the class. This can help to ensure that only the correct values are being interacted with during actions on objects.

If a program is using an API request for example, potentially the request could return no data. To react to this, the Integration Test would return an error if no data is returned, rather than the program attempting to proceed without the data from the API.

5.7.5 System Test

System testing involves testing all components of the system together (Singh 2011). Because of this, it is often one of the last stages of testing. This involves more of a manual approach and while it can be automated using various testing frameworks or environments for the purpose of this study, only manual will be discussed. Manual often involves more than one individual monitoring components of the system as they run and documenting their performance for review.

“A UML-Based Approach to System Testing” (Briand and Labiche 2002) discusses how this can be paired with UML artifacts to derive system test requirements. For example, using use cases and the sequence diagram to create analysis points in the System Test and to ensure that the system is functioning correctly. This approach is effective as it offers a clear plan as to how the System Test can be conducted using documents that are relevant throughout all of development and likely will not change.

6 Commentary on Requirements Specification

Requirements for this project are defined using MoSCoW. This means evaluating requirements by importance; in order of priority, these are, "must have", "should have", "could have" and "won’t currently have" (Ahmad et al. 2017). Using this it is easy to break down all of the possible components and prioritise them in order to focus development on higher priority tasks. Using this also helps to drive development as when one requirement is met, deciding what to move on to next is easier. The "won’t currently have" requirements also give an idea of where the project could be taken in future to anyone who may be overseeing development. These requirements can be found in Figure 12. Throughout these requirements will be referred to as RQ-x where x is the number of the requirement.

6.1 Requirements Capture

This project will have no user input and will only rely on the user receiving the puzzle in the most optimal form because of this, most of the requirements are functional. The non-functional requirements in this case relate mostly to how expandable the software is, its optimisation and how the puzzle is output.

Due to this project being proposed by the supervisor (Sinclair, M.C, October 2021), most of the requirements were laid out prior to the project. This denoted that the generator would need to be broken down into two main parts, generation and backtracking. Generating would generate a puzzle with unverified uniqueness or quality, then the backtracker would scan the puzzle by attempting to solve it by inputting all PA at varying depths to decide if the puzzle was unique or not. If the puzzle was unique it would be returned to the user, and if not the puzzle would be regenerated.

6.2 Backtracker Requirements

Requirements: **2, 3, 4, 16, 18**.

No.	Requirement	Priority	Type
1	A puzzle generator that is able to generate a puzzle larger than 9 x 9 with a single solution.	Must have.	Functional
2	A backtracking solver that is able to count the number of solutions to a Futoshiki puzzle.	Must have.	Functional
3	The backtracker can create a tree of the search order when searching for solutions.	Must have.	Functional
4	The backtracker takes advantage of the supplied solver code.	Must have.	Functional
5	The generator can add constraints optimally.	Must have.	Functional
6	The system is built within Java.	Must have.	Functional
7	The generator can be run within a single executable.	Must have.	Functional
8	Generated puzzles are returned to the user in an easily readable format.	Must have.	Non-Functional
9	Optimised memory usage.	Must have.	Non-Functional
10	Be able to output the puzzle.	Should have.	Functional
11	Can be adjusted to be able to work with several different puzzle sizes.	Should have.	Functional
12	Well commented/documented code.	Should have.	Non-Functional
13	Provide fast puzzle generation.	Should have.	Non-Functional
14	Integration with an external framework (ie. J-POP) for final puzzle testing.	Could have.	Functional
15	Preventions in place to reduce likelihood of duplicate puzzles being created.	Could have.	Functional
16	The ability to rate puzzles based on the solution search tree.	Could have.	Functional
17	The output is formatted in a standard format (ie. Janko)	Could Have	Non-Functional
18	The solution search tree will not be dynamically updated based on updates from in the puzzle.	Won't Have.	Functional
19	A GUI for easier user interaction	Won't Have.	Functional

Figure 12: *Requirements Specification*

These requirements outline what the backtracker should output and briefly how it can go about this. As mentioned in RQ-*4*, code has been supplied by the supervisor (Sinclair, M.C, October 2021). This will be discussed in further detail in Section 8.2, but this code is a Java version of how J-POP represents puzzles. Because the supplied solver only contains a few solving methods, the backtracker will attempt to brute force finding other solutions through inserting every possible value. Since the backtracker needs to be able to solve the puzzle at each level, this solver can be used to do this however modifications to this will be needed to be made. As discussed in Section 5.5, a full solving tree will need to be constructed rather than just up until a solution is found in order to count the solutions to verify uniqueness.

As stated in Section 5.4.1, rating puzzles can be a useful component of generation as it can be used to critique difficulty. While creating a varied set of puzzles is not strictly the goal of this project, were the framework to be built on top of this may be useful.

It was originally intended to update the solution tree as constraints were added to the puzzle, however research in the literature review has shown that due to the waterfall effect of changing aspects of the puzzle in the development environment being used this likely will not be possible. Due to this RQ-*18* has been labelled as "Won't Have".

6.3 Generator Requirements

Requirements: **1, 5, 8, 15**.

The main decision in generation is how many constraints to add in order to keep the solution count at one. This was discussed in the literature review and the studies done in “The number of inequality signs in the design of Futoshiki puzzle” (Haraguchi 2013) be used to shape this. To fulfil RQ-*5*, the generator must carefully select where constraints are to be added rather than sporadically. This could be done through adding constraints based on reducing highest the number of PA or using features commonly seen in Futoshiki such as those discussed in 5.2.2 to retroactively add values.

Because the puzzle needs to be returned to the user, returning it in a human-readable format is needed. This could be done a few different ways, whether through displaying the locations of all the starting constraints to the user so that they may build it themself or through simply displaying the puzzle to the user. The former is certainly the easier method as on occasions finding an optimal way to display the puzzle in a similar manner to an image could be difficult however, this method of showing the user the full puzzle is the more optimal method as it is able to be immediately interpreted.

The generator without guidance will go down the same path of generation resulting in the same puzzle being made. In adding an aspect of randomness, the likelihood of the program only outputting the same puzzle can be reduced. Introducing this aspect of randomness is done by ensuring that the generator does not go down the same path each time, for example if the decision tree is mapped, ensuring that the solver does not go down the same branches and forcing it to attempt to run different paths on each run will improve the variety of puzzles generated.

6.4 Build Requirements

Requirements: **6, 7, 11, 12, 13**.

As stated earlier, the supervisor supplied code (Sinclair, M.C, October 2021) is written in Java and while it was found in the literature review that a more functional programming approach would yield better results, having a foundation to build the generator on top of will outweigh the performance drawbacks that may have. It was found in the literature review that Java does however still have methods of operating in a functional format (Tsantalis, Mazinanian, and Rostami 2017). Should the program require this level of optimisation, use of these features is available.

Due to the potential size of the data being dealt with, high build quality will be key. Poor build quality will likely result in failure to generate due to memory limits being exceeded. Methods such as Flyweight will be used to avoid this and overall improve performance.

Relating back to being able to build on top of the code, commenting and high quality structure are important to ensure that the code is easily interpreted. To do this, all functions and classes will be commented on stating what they do and how they should be used, and structure will be made clear through the use of supporting UML artifacts.

6.5 Other Requirements

Requirements: **10, 14, 17, 19**.

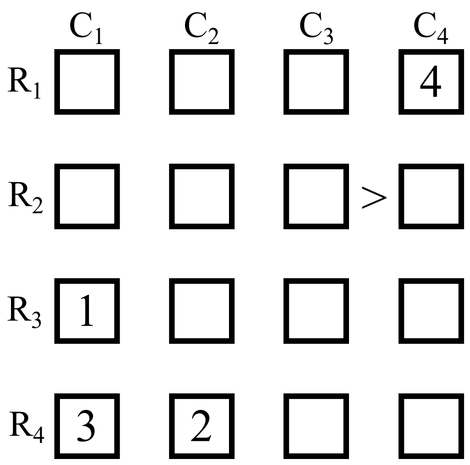
Several puzzle solving frameworks exist outside of the supervisor (Sinclair, M.C, October 2021) provided one. In porting the puzzle into an external framework as a final layer of testing, this can further ensure the validity of the puzzle and ensures that the puzzle is not only solvable using the algorithm used through generation. Though this is not fully necessary as the likelihood of a the generator creating a puzzle that can only be solved by a single solver is low, and would likely come down to a bug in the solver.

The ability to output the puzzle alongside displaying as discussed in RQ-*10* and RQ-*17* is not essential however will likely be done. This can be done by exporting the puzzle in a text file in varying formats, these are discussed near the end of Section. In doing this, the project becomes far more modular as it opens up the software to be used alongside other frameworks. For this project, a set of puzzles were scraped from the Futoshiki section of Janko (Angela and Janko 2020). These were returned in a machine readable format that could be used by a solver. Figure 13 shows in two parts how this is done, Figure 13a being the scrape of the puzzle with line indicators added, and Figure 13b being the built puzzle. Line 1 defines the width and height of the puzzle, the size of the next part will be decided by the size of the puzzle as it simply prints the Latin Square for the puzzle, with *-1* being unfilled squares. Line 6 is the number of inequalities and following on the remaining lines this are the inequalities, with the cells being numbered from 0 to n^2 from left to right then downwards proceeding.

A GUI, while aesthetically pleasing, will not be necessary for the project. The goal of this project is simply to generate the puzzle and because of this, the program will be ran and display the puzzle to the user from a terminal, and potentially output it for later use as discussed in earlier in this section. Though it is not do be ruled out that in later renditions of the program that a GUI could be added.

```
1 |  4
2 |  -1 -1 -1 4
3 |  -1 -1 -1 -1
4 |  1 -1 -1 -1
5 |  3 2 -1 -1
6 |  1
7 |  6>7
```

(a) *Janko Puzzle Scrape.*



(b) *Janko Puzzle.*

Figure 13: *Janko Futoshiki Puzzle 44.*

7 Development Tools

Because the program is being developed in Java, the IDE Eclipse will be used. Eclipse is an open source IDE largely built up from community made plugins to assist in development (*Eclipse* 2022), it is currently estimated to be the second most popular IDE according to *Top IDE Index* (*Top IDE Index* 2022) which is based off of Google Trends. In general for development either a text editor or an IDE is used, text editors such as Visual Studio Code (*Visual Studio Code* 2022) offer basic functionality of writing and editing text sometimes offering extra features such as auto completion of words or variables. An IDE on the other hand is a complete development environment, offering support for compiling and testing code. Often IDE's are much heavier than text editors due to the functionality they offer, however when working with larger projects this trade-off is necessary.

For this project Eclipse has been deemed the best choice as the volume of plugins available that make improve development and familiarity with the IDE through past projects allowing for better usage of its capabilities. For example one of the plugins that will be used is UCDetector (Spieler 2022), this is used to detect code that may not be written in a correct standard or code that may be redundant. UCDetector does not automatically correct code for the user and will only prompt that code may need to be changed. Code linting (Hansson 2014) is also a common feature introduced by plugins, allowing for code to be automatically be checked for both logic and stylistic errors. Use of various plugins through out overall will increase quality of code as they save on development time and will often catch flaws that the developer might not.

For development of UML artifacts, both Umbrello and UMLet will be used (*Umbrello* 2022; *UMLet* 2022). Umbrello will be used for class diagrams as it supports the best functionality for building large classes, it does support functionality for developing sequence diagrams, however it can be restrictive and does not perform well. Because of this, UMLet will be used to develop sequence diagrams instead as its ability to develop them is of a higher standard.

8 Design

8.1 Functional Design Analysis

As stated during discussion of the requirements, development will be broken down into two components, generation and the backtracker. The purpose of the generator is simply to create a puzzle with unknown uniqueness, then verify the uniqueness using the backtracker.

8.1.1 Backtracker

The backtracker is there as a last resort to find solutions to a puzzle that the solver may not have accounted for. It does this by attempting to brute force every possible assign into every cell. While this is not the cleanest implementation, it is infeasible to teach the solver every possible solving method and despite the drawbacks in performance that come with it, backtracking is the easiest way to check for solutions.

The backtracker has two components in which it uses to trace puzzles, *Level* and *State*. *Level* is the current stage that solver is at this stores the PA (potential assigns, see Section 5.4.1) for the current stage of the puzzle, and *State* stores all of the information on the puzzle such as all of the assigns it has received so far. Each *Level* represents a new value being added to the puzzle, this can represent the depth of the search as in the backtracker there is a stack of the *Level* class. How *Level* and *State* will function can be seen in Figure 14.

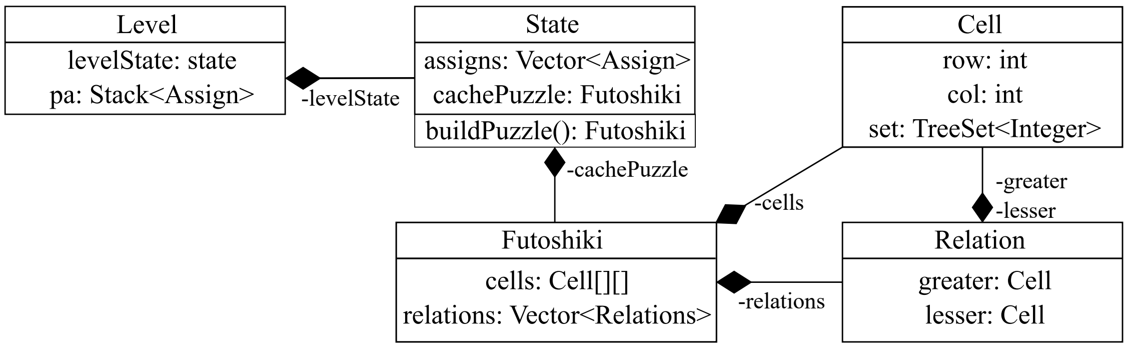


Figure 14: *Medium Level Level and State Class Diagram.*

Algorithm 1 shows how the backtracking will be done.

To break down how this will work, a puzzle is made by the generator with unverified uniqueness and added to the *levelStack*, this creates depth 1. Immediately the number of solutions already found is checked as if this has exceeded one, backtracking can stop as the puzzle

Algorithm 1 Backtracking Algorithm

```
1: $levelStack \leftarrow startingPuzzleLevel$
2: while $depth > 0$ do
3:     if $solutionCount > 1$ then
           break                                          ▷ Multiple Solutions Found
4:     end if
5:     if $traceLevel = true$ then
           $newLevel = currentLevel \leftarrow currentLevel.randomPA$
           $levelStack \leftarrow newLevel$
6:     else $levelStack = levelStack - currentLevel$
7:     end if
8: end while
9: if $solutionCount = 1$ then
       return puzzle                  ▷ Puzzle was traced and one solution found
10: end if
```

does not have a unique solution. If the solution count is one or lower, the puzzle is tested to see if the depth can increase. This is a feasibility test to check if a break point has been arrived at (See Section 5.5), this has four potential outcomes each returning true or false, Table 3 outlines these scenarios.

Scenario	Cause	Output
1	The puzzle has been solved.	False.
2	The puzzle is infeasible.	False
3	No more PA.	False
4	None of the above are true.	True

Table 3: *Backtracker Scenarios*

If scenario 1 is encountered, it is checked if this solution has already been found or if it is new, if the solution is new it is added to the solution list. All of the first three scenarios return false as they declare that the current point is a leaf node and the parent node must be returned to, this is done by removing the current solution from the stack so that the loop will resume at the previous level thus reducing the depth. However, if scenario 4 is encountered then the depth can be increased and a new level will be added to the top of the stack. Assuming the clause of no more than one solution is maintained, the two outcomes are met, if a solution was found, it is returned to the user and the program can stop. If no solution was found, a new puzzle will be generated and sent to the backtracker for testing.

8.1.2 Generator

Puzzle are sent to the backtracker on request. The generation process populates the puzzle with valid constraints with no concern as to the solubility of the puzzle as this will be dictated within the backtracker. The number of constraints that will be added will be adjusted during testing to fit runtime requirements as harder puzzles typically have less constraints and have a larger generation tree creating a larger runtime. Generation can be broken down into two stages, relation generation and number generation.

Relation generation is done first, a random cell is chosen then a set of numbers between 1 and 8 is generated, this represents the direction and location of the inequality. This is because from a cell there are four positions (north, east, south, west) they can be placed and two different locations and two directions (outwards, and inwards) that it can face. Not every relation however is valid for example, cell R_1C_1 cannot have a relation north or east. An example of how an invalid assign might attempt to be added can be seen in Figure 15. The solver code has a function that can test if a relation is valid this will be used to test the relation. The reason that the set is not predefined with the valid relations is because, the only cells that have invalid relations are boundary cells meaning that if the puzzle is 7x7 or larger there are more interior than exterior cells meaning it would be faster to assume that the cell being assigned to is going to be an interior cell.

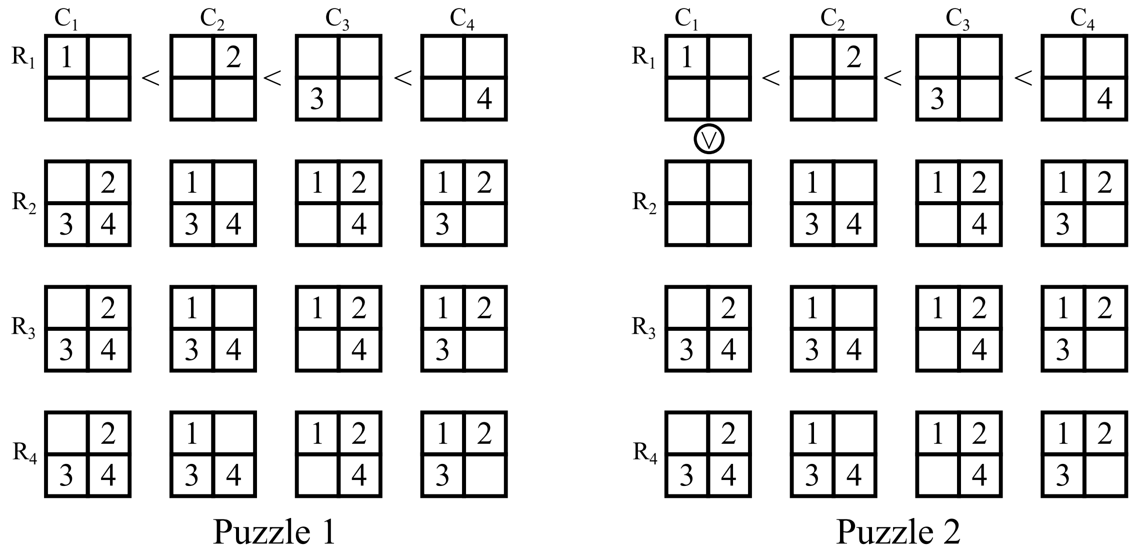


Figure 15: *Generator Invalid Assign.*

To create a relation, a random number from the set is chosen and its representative inequality is tested to see if it is valid to add to the puzzle. If it is not valid it will be removed from the set, and a new number will be selected and so forth. If it is valid it will be added to the puzzle. This process is repeated until the predefined number of inequalities are added.

After this numbers are added, this is far more simple as each cell has a set of numbers which are valid assigns. Because of this, a random cell can be selected and then from its set of valid assigns, one can be selected and added. This process is repeated until the predefined number of values are added.

8.2 Supplied Code

Prior to this projects inception, the study was provided with a packages called *futoshikisolver*, a high level class diagram of this code can be seen in Figure 16. This code was provided by the project supervisor (Sinclair, M.C, October 2021) and is a Java version of the J-POP Futoshiki solver, with this a few aspects of the project are simplified. Because to find solutions the generator must be able to solve puzzles, this package will cover that. The package offers the ability build Futoshiki puzzles through manual input and will dynamically solve them as values are added. It offers two solving methods, scanning (Section 5.1.2.1) and naked pairs (Section 5.1.2.2), if either of these methods conditions are met at a stage in the current stage of the puzzle it will update the puzzle based on that. A bonus of this functionality is that the puzzle must keep note of the PA of each cell as it essential for both of these solving methods. The generator will be able to take advantage of this in testing adding every PA before progressing levels.

However not all of the PA are valid as the solver assumes that the value that the user inputs will not invalidate the puzzle at the next level, this is because the solver is designed to receive an unsolved puzzle that is already verified as valid therefore will return an error if it receives a value that causes any of the cells to become invalid, even if the assign was claimed to be PA. This creates a scenario where the puzzle is only being tested at the current level. To fix this, the list of PA must either be filter to cause them not to invalidate the next stage of the puzzle.

8.3 Structural Design Analysis

For the design of the program several different UML artifacts were made. Unified modeling language (UML), is used in software development to act as a visual aid in how a program should be structured and or act "A systematic review of UML model consistency management" (Lucas, Molina, and Toval 2009). Several different UML artifacts exist for different purposes, the ones used in this study were a few class diagrams of varying depth and a sequence diagram. Class diagrams are used for displaying all of the classes and how they interface with each other, a good use of this would be that coupling becomes very clear through its use.

Sequence diagrams are used to describe an action of a system and how the different components of the system would interact with each other in order to complete the action and the order they do this in. This is useful as it offers a clear order of how a system would complete an operation. For this project only a single sequence diagram was designed as it was useful for the understanding of how the backtracker would scan puzzles and adjust levels. The sequence diagram offered a more clear understanding of how these loops and alternative paths would function.

8.3.1 Supplied Code Class Diagram

Appendix 4 displays a full class diagram of the supplied code, and Figure 16 showing a high level version of this. This offers functionality for making Futoshiki; each class supporting a different aspect of the puzzle. A puzzle is built up from a set of Cell objects the number of these is determined by *Futoshiki.SETSIZE*. Each cell has, a row, a column, a set of PA, relations, and constraints.

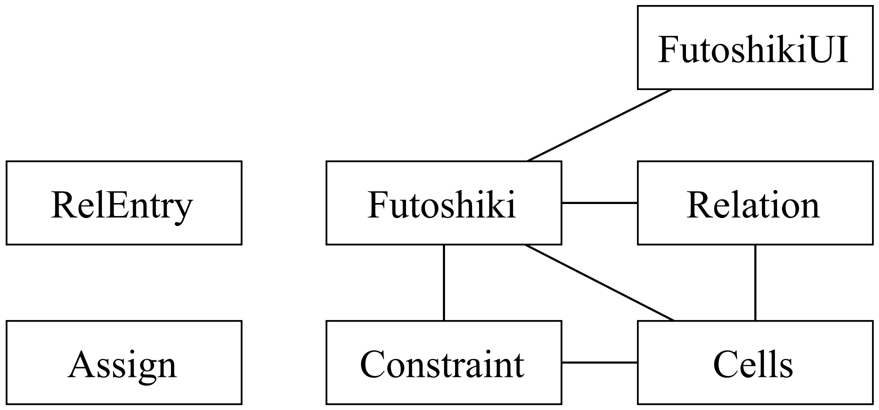


Figure 16: *High Level Supplied Code Class Diagram*

The row and column store the location on the puzzle of the cell. Relations are the inequalities between cells, in storing just the greater and lesser cell all of the relevant information regarding constraints is gathered as this shows both location and orientation.

The set of PA are the PA available for the cell controlled by the constraints. Constraints are sets of cells built in rows and columns to reduce PA, these are decided by solving methods such as *checkUnique* and *checkPairs*. For example every time a value is added to a cell, all of the cells that could be impacted must be updated, this is done by constraints due to the set of cells within the constraint dictating which cells are impacted by certain changes. The same rules are applied by constraints regarding relations, cells that are of greater or lesser than another are ensured to remain this way.

The futoshikisolver package has a text-based UI component for users to enter puzzles using, this is not of use to the project. The classes involved in this are *FutoshikiUI* and *UserEntry*. *RelEntry* is separate from this as it represents the passive component of relations and will be of use for exporting puzzle information.

8.3.2 Full Class Diagram

Appendix 3 shows a full class diagram for the generator, with the yellow classes representing classes made for the purpose of this study and the blue ones being from the supplied code. Figure 17 shows a high level version of this. To take advantage of flyweight, *State* represents the build order for a puzzle rather than just storing the puzzle. On occasions however the puzzle will be needed, from this *State* can build up the puzzle and store it so that if the puzzle is requested again, it will not be rebuild the puzzle and will call it from *puzzleCache*. *Level* stores *State* along with the PA at the current stage. PA is a stack of *Assign* as the level is generated and is reduced as the level is explored, each time a PA is tested, it is removed from the PA stack so that it is not tested twice. This is controller within the backtracker.

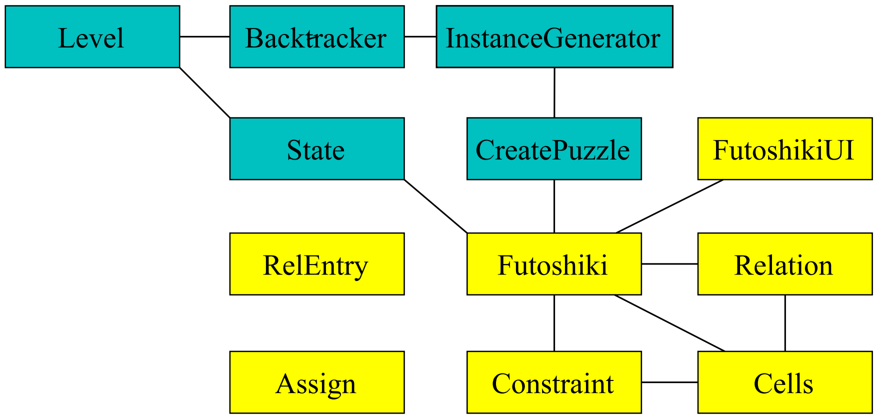


Figure 17: *High Level Full Program Class Diagram*

The backtracker contains a stack of *Level* each with one assign that the last does not. This stack is either incremented or reduced depending on the feasibility of the puzzle, this idea is developed upon in Section 8.1.1.

8.3.3 Sequence Diagram

The sequence diagram shown in Appendix 5 briefly outlines how the generator will create puzzles. *CreatePuzzle* requests a puzzle from *InstanceGenerator*, this is returned then sent to the backtracker. The backtracker will attempt to find solutions in order to verify the uniqueness of the puzzle. The backtracker requests the list of potential assigns from *Level* at the top of the *Level* stack. To get these the *State* builds the puzzle, because the puzzle is represented in flyweight form, the *State* uses the build order of the puzzle to create it from new. This means obtaining the build instructions for the starting puzzle from *InstanceGenerator* then applying the list of assigns for the current *State*.

At this point, the puzzle has been built up to the current level. This was done to create a deep clone of the current puzzle so that assigns can be tested on it, this is necessary because the solver code that is being used to generate a list of PA, as mentioned earlier are not all valid assigns. As a solution to this, each assign is tested by adding it to a copy of the puzzle at the current level. If an error is returned, the assign that was used is removed from the PA as it is not valid. Once all of the PA are tested, a true list of PA is returned to *Level*.

The level is then traced using the feasibility test to decide if the depth is to be increased or decreased. To increase the depth, the top assign is taken from the PA stack and added to the puzzle, and used to create a new level. To reduce the depth, the current level is removed from the level stack. If the level stack is not empty, the process of tracing the level will repeat. If the level stack is empty and a solution is found, the *tracePuzzle* will return true and the puzzle will be returned to the user.

This process will be repeated until a puzzle is returned to the user.

9 Implementation

This section offers a breakdown of how all of the classes have been developed and the functionality they support.

9.1 Instance Generation

As discussed earlier the puzzle requires two components to be added to it, starting values and relations. The *InstanceGenerator* would do this by adding relations first then values, this was done because through development the generator showed to have an easier time adding relations before values. This is likely because the function for testing relations to ensure they would not invalidate the puzzle was a more complex function than simply adding a number from the chosen cells set of PA.

The functionality for adding relations changed largely during implementation. The initial design for using the valid assign test within *futoshikisolver* upon implementation did not work. This is likely due to the same error as with potential assigns in which the current level is being tested, but not the next. Another issue with the initial design is that it relied on trying the assign on the base puzzle in a try catch block and if the catch was triggered the relation could not be added, as *futoshikisolver* would return an error if an invalid assign was added. The issue with this is that even if the relation was invalid and an error was thrown, the assign would still be added but the try catch would prevent the program from crashing.

The fix for both of these problems was to instead add all orientations of the relations to a vector then test all of them on a clone of the base puzzle, then add the ones that were successful to a new vector. From this new vector, select one relation at random and add it to the puzzle. This also showed that through future development, tests would have to be performed on clones due to errors persisting through try catch blocks.

The assign function functions almost identical how it was mentioned in design, however due to the error with the supplied code in which puzzles are only tested at the current level, assigns had to be tested prior to being added.

As a final catch for errors within the puzzle, when an instance is generated a build order is saved. This is done for a few reasons, if the base puzzle is required for testing it can be built using the build order, when the puzzle is exported the build order is required and for final testing of the puzzle having the build order is useful. In the last stage before testing the puzzle for solutions, the puzzle that is going to be tested is attempted to be built using the build order, if this returns any errors then the puzzle is not valid. This helps to catch any assigns or relations that may have been added that should not have.

9.2 Cloning

Because test instances of classes are used throughout for testing of adding assigns and constraints a clone method was essential. Because the default clone method for an object in Java creates shallow copies, these will act more as a reference to the original object rather than acting as a new object like a deep copy will (Khoshafian and Copeland 1986). To solve this, a clone method was created for *Level*, *State* and the puzzle that was currently being traced that was able to create deep copies.

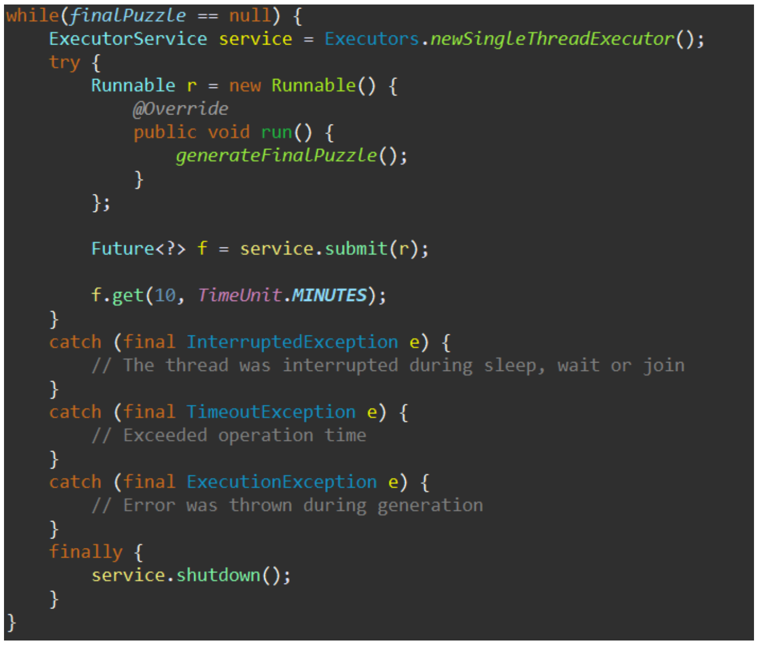
Cloning the *Futoshiki* would be difficult as the puzzle contains a multidimensional array of the class *Cell*, for example in a puzzle of size 9, 81 *Cell* are being checked for values. As a work around to this, *InstanceGenerator* would save a vector of both the relations and assigns in the puzzle currently being tested. From this upon request a puzzle identical to the puzzle being tested can be built using the build order for testing. The effectively acted as a clone method. For *Level* and *State*, the variables within these could be easily cloned using Java's traditional cloning methods.

9.3 Final Puzzle Generation

To generate the unique puzzle, the class *CreatePuzzle* is used as an interface to combine both the backtracker and generator to create a puzzle. Initially the generator will create a puzzle, *CreatePuzzle* then checks if it is a valid puzzle. It does this through the clause mentioned earlier where the puzzle is attempted to be built. If the puzzle fails to build, then a new puzzle is requested and this process repeats until a functioning puzzle is provided. Once a functioning puzzle is provided then next stage is progressed to.

This involves searching for solutions, because *futoshikisolver* dynamically solves puzzles as constraints and assigns are added, the puzzle is immediately tested to check if it has already been solved and if it has, the puzzle is returned. If the puzzle is not already solved it will be passed to the backtracking solver in order to check for deeper solutions.

Due to how long backtracking could potentially last, the duration this went on for was capped. The backtracker was estimated to be able to explore 3,000 levels per minute, in some cases with larger puzzles backtracking would last over an hour, meaning an estimated over 180,0000 assigns were tested. A puzzle of this complexity is both infeasible for both the user to solve and impractical to wait this length for the puzzle. As a work around to this, if backtracking lasted over 10 minutes, the puzzle would be scrapped and a new one would be moved onto, estimating that puzzles with a complexity of over 30,000 PA are abandoned by the generator. Because the timer would need to run at the same time as the generator, the use of threads would be needed. This was done using the ExecutorService JDK API (baeldung 2021), which takes advantage of Java's threading options. By simplifying threading options, ExecutorService creates a pool of threads in which tasks can be assigned to and when provided with a task will assign to one of them until the timer is completed or the block of code has finished running, after which the thread is closed. The implementation of this can be seen in Figure 18.



```
while(finalPuzzle == null) {
    ExecutorService service = Executors.newSingleThreadExecutor();
    try {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                generateFinalPuzzle();
            }
        };

        Future<?> f = service.submit(r);

        f.get(10, TimeUnit.MINUTES);
    }
    catch (final InterruptedException e) {
        // The thread was interrupted during sleep, wait or join
    }
    catch (final TimeoutException e) {
        // Exceeded operation time
    }
    catch (final ExecutionException e) {
        // Error was thrown during generation
    }
    finally {
        service.shutdown();
    }
}
```

Figure 18: *Timed Generation*

9.4 Puzzle Export

Along with showing the completed puzzle to the user, the puzzle can also be exported in the format that was depicted earlier in Figure 13a allowing for direct import into frameworks such as J-POP. This is done by taking advantage of the Java class *FileWriter* which allowed for writing to a .txt file. Not depicted in planning, this was created inside of a new class called *WriteToFile*, which would take strings and write them to a file named "Generation *timestamp*".

This meant iterating through the build order to insert values as necessary. The size of the puzzle was initially printed on the first line. For each cell of the puzzle, the build order was checked for an assign, if one was found it was inserted at the current location being checked, and if none were found a -1 was inserted. This was then printed in the file row by row. Relations were far easier as the size of the vector of relations could simply be printed followed by the relations one by one on a new row.

9.5 Supplied Code Modifications

The supplied code was left mostly untouched as most of it was in a whole state. The main functionality added was adding an equals method to *Futoshiki*, which in turn meant adding an equals method to *Relation* and *Cell.* Checking the equality of two puzzles was necessary for the scenario in backtracking in which two solutions are found, because two solutions can be identical just they were found down different routes of the solver. To test the equality of the two puzzles tests are were ran to see on both puzzles the relations were the same and the cells were the same.

The puzzles cells are stored in a 2 dimensional array. This means that if an equality method is added, the whole array can be tested using *deepEquals().* To compare if two instances of *Cell* are the same the row and column need to be compared to ensure the location of the cell is the same, and then the set of PA can be compared as if the cells have the same PA they are likely under the influence of the same constraints. Relations required more than just an equals method in the relations class. The initial equals was done just through checking that the greater cell and lesser cell in values were the same across the two relations. However because vectors have order, the default equals method for comparison takes order into account when, in this case, the order of the relations makes no difference. To fix this, if the vectors are the same size, for each of the values of one vector they are checked if they are contained within the other. Both of these competitors are applied to test if two *Futoshiki* instances are the same.

Another functionality that was added was to clone puzzles, however this required searching through the puzzle for relations and numbers and thus its use was avoided as it had significant implications on runtime; saving the build order for the puzzle and rebuilding was faster in every scenario.

10 Testing

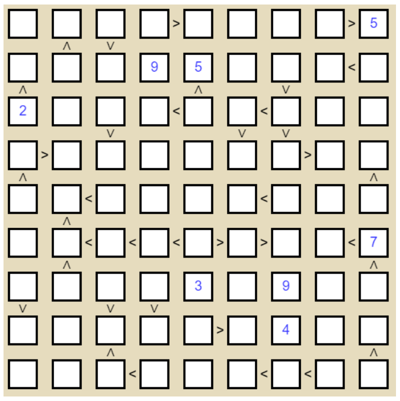
Testing of this project can be broken down into two main sections, functional testing and performance testing. Functional testing revolves mostly around unit testing, ensuring that functions within the program are running correctly. Alongside unit testing, manual outputs were used for user checking throughout runtime. Performance testing focuses on how the program runs, for example how long it takes to run.

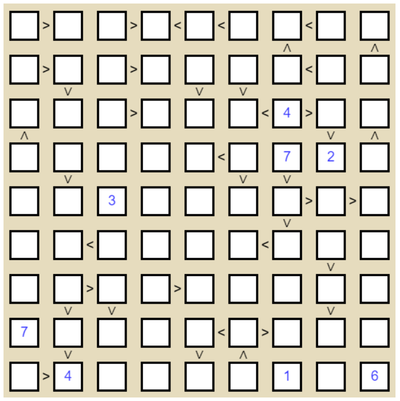
10.1 Test Plans

The functional tests do not have the same levels of documentation as performance tests as they are in place to detect errors in the running of the program and because of this they are used to guide development, assisting in finding errors. Performance testing allows for testing of individual components and system testing, evaluating on if the component performs its task successfully and how well it does this.

Unit tests will be performed using the Java framework JUnit (*JUnit5* 2022). This allows for easy writing of unit test within Java, as tests can be written in a class as a function, and the function has an assertion that if met causes the unit test to return false. The class can then be ran and Java will compile all of the tests and return the results of them showing which tests returned false and the trace if so. This will allow for the testing of all critical functions. Alongside this a boolean variable called *testMode* has been included in most functions, when set to true, several lines of code that allow for console output to help debug are ran. For example, in the backtracker when *testMode* is true, puzzles are displayed and the depth of the solver are out to the console during running. This was mostly used for debugging throughout development, however it is also useful as a form of integration testing to ensure that the correct variables are being passed through the system during running.

For performance testing will be done in a number of different ways because a few different components must be tested. For the backtracker, this will be tested using three edge cases of 10×10 puzzles classified as difficult by Janko (Angela and Janko 2020). These puzzles can be found in Figure 19, the backtracker will be tested to see if it can verify the uniqueness of these puzzles and how long it takes to do so.


(a) *Janko Futoshiki 200.*


(b) *Janko Futoshiki 210.*

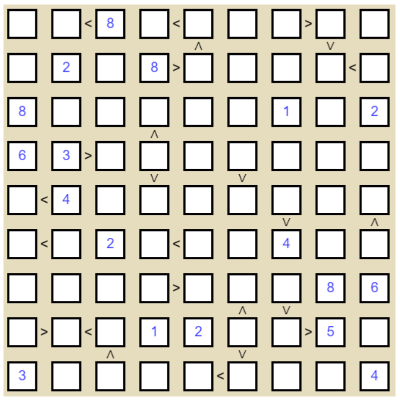

(c) *Janko Futoshiki 220.*

Figure 19: *Puzzles for Backtracker Testing (Angela and Janko 2020).*

For generation this will be tested by seeing how long it takes for the generator to construct a puzzle with unverified uniqueness. This was done five times for puzzles scaling in size, starting at 4×4 and going up to 10×10, recording how long each took to generate.

And finally for system testing, similarly to how generation was tested, puzzle creation was done scaling in size starting at 4×4 up to 10×10 recording how long it took to create the puzzle, and how many instances were tested for uniqueness. This was done ten times for each size so that measurements can be taken to evaluate the success and efficiency of the solver. If the runtime of any of these tests exceeds 24 hours, the tests will be classified as a failure. Through development, in the larger puzzles upwards of 1000 instances were tested before a unique one was found. Because of this the timeout (the period of time before the puzzle is abandoned, see Section 9.3) was adjusted to 15 seconds. This means that the solution for the puzzle will likely be a one that can either solved through the methods within *futoshikisolver* or will be a one found at a high level in the backtracker. This change was made to suit this test and was reverted afterwards.

For these tests the conditions showed in Table 4. These conditions were selected using a combination of observations from pre-generated puzzles, along with throughout development when informally testing generation of puzzles values were adjusted accordingly to find the best fit for these.

Puzzle size ($n \times n$)	Assigns	Relations
4	2	12
5	3	15
6	4	20
7	5	30
8	6	45
9	7	65
10	8	90

Table 4: *Futoshiki Conditions.*

10.1.1 Functional Testing

As stated earlier, JUnit (*JUnit5* 2022) was the testing framework used to setup unit tests. 16 tests were done to ensure that functions were working as they should, the two to be discussed are *testBasePuzzle*, used to build the puzzle currently being generated to ensure that it can successfully be built without error, and *nextLevel*, this s responsible for finding the next level to be added to *levelStack*. To do this, two puzzles would be needed, because testing should assume a worst case scenario where possible, the worst case scenario for testing building a puzzle from the build order would be for the build order to be invalid, so for this an invalid puzzle would be needed. In testing *nextLevel*, a puzzle that has multiple potential assigns remaining will be needed. Both of these puzzles used for the tests can be seen in Figure 20.

To test if the base puzzle is build-able, the build order in *InstanceGenerator* is set to that which would build an invalid Futoshiki. Once this is done the function *testBasePuzzle* is ran,

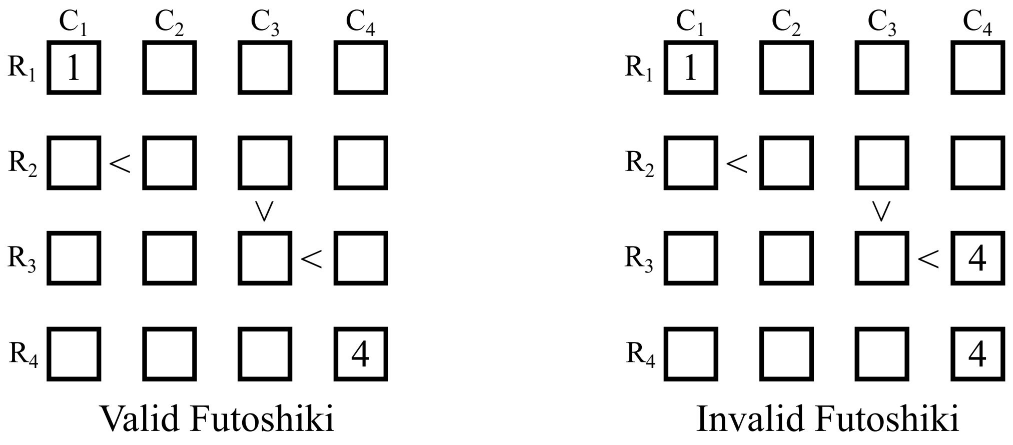
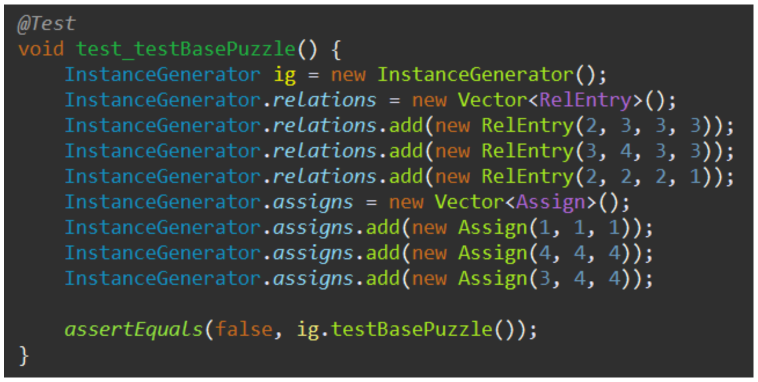


Figure 20: *Valid and Invalid Puzzles Used in Testing*

this attempts to build the puzzle and if an error is encountered it returns false. Since the puzzle being given to the test is invalid the test is a success if *testBasePuzzle* returns false. The code for this can be seen in Figure 21.

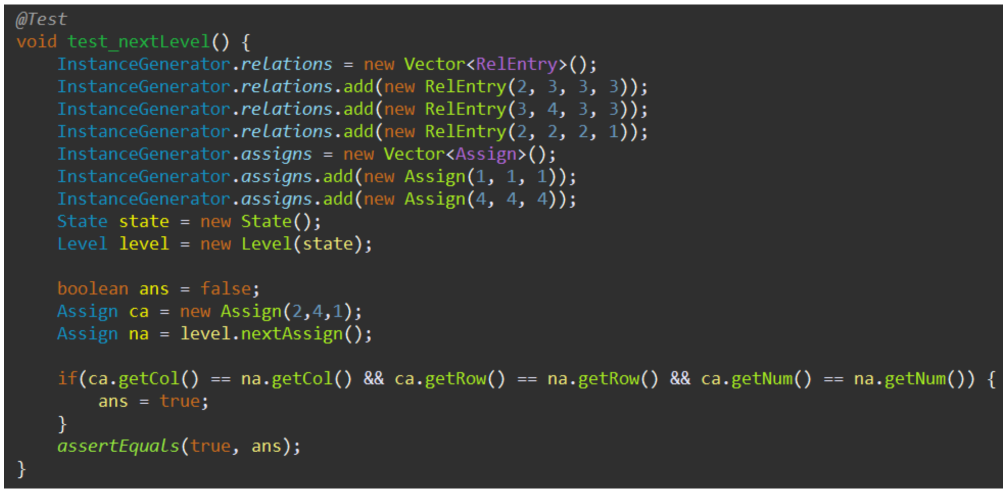


```
@Test
void test_testBasePuzzle() {
    InstanceGenerator ig = new InstanceGenerator();
    InstanceGenerator.relations = new Vector<RelEntry>();
    InstanceGenerator.relations.add(new RelEntry(2, 3, 3, 3));
    InstanceGenerator.relations.add(new RelEntry(3, 4, 3, 3));
    InstanceGenerator.relations.add(new RelEntry(2, 2, 2, 1));
    InstanceGenerator.assigns = new Vector<Assign>();
    InstanceGenerator.assigns.add(new Assign(1, 1, 1));
    InstanceGenerator.assigns.add(new Assign(4, 4, 4));
    InstanceGenerator.assigns.add(new Assign(3, 4, 4));

    assertEquals(false, ig.testBasePuzzle());
}
```

Figure 21: *testBasePuzzle Test Code.*

Testing what the next level to be traced should be an only be done if the next level is known, therefore it can be tested against the return of *nextLevel* to see if they are the same. To do this, the predefined valid puzzle shown in Figure 20 was fed into the test and its the next assign was declared using *nextAssign*; another assign is then declared which is what the next assign to create the next level should be. Each of the variables within the two assign are then compared for equality this includes, the row, the column and the value. If all of these values are equal between the actual next assign and the correct next assign, the test is passed. The code for this can be seen in Figure 22.



```java
@Test
void test_nextLevel() {
    InstanceGenerator.relations = new Vector<RelEntry>();
    InstanceGenerator.relations.add(new RelEntry(2, 3, 3, 3));
    InstanceGenerator.relations.add(new RelEntry(3, 4, 3, 3));
    InstanceGenerator.relations.add(new RelEntry(2, 2, 2, 1));
    InstanceGenerator.assigns = new Vector<Assign>();
    InstanceGenerator.assigns.add(new Assign(1, 1, 1));
    InstanceGenerator.assigns.add(new Assign(4, 4, 4));
    State state = new State();
    Level level = new Level(state);

    boolean ans = false;
    Assign ca = new Assign(2,4,1);
    Assign na = level.nextAssign();

    if(ca.getCol() == na.getCol() && ca.getRow() == na.getRow() && ca.getNum() == na.getNum()) {
        ans = true;
    }
    assertEquals(true, ans);
}
```

Figure 22: *nextLevel Test Code.*

Alongside these two tests the 14 other tests were ran throughout development as a means of finding errors, however post development these tests were re-ran three times each to determine the if the code was still behaving appropriately. All of the tests were passed in the three testing sessions and the results of this along with the functions tested can be seen in Table 5.

Test	1	2	3
test_setNum()	Success	Success	Success
test_getSetAsString()	Success	Success	Success
test_getSet()	Success	Success	Success
test_isValidRelation()	Success	Success	Success
test_ValidAssign_value()	Success	Success	Success
test_ValidAssign_relation()	Success	Success	Success
test_puzzleEqual_true()	Success	Success	Success
test_puzzleEqual_false()	Success	Success	Success
test_cellEqual_true()	Success	Success	Success
test_cellEqual_false()	Success	Success	Success
test_relationsEqual_false()	Success	Success	Success
test_relationsEqual_true()	Success	Success	Success
test_clone_true()	Success	Success	Success
test_cloneBasePuzzle()	Success	Success	Success
test_testBasePuzzle()	Success	Success	Success
test_nextLevel()	Success	Success	Success

Table 5: *Unit Test Results*

10.1.2 Performance Testing

For performance testing, the results of the complete generations of puzzles can be seen across Tables 6 and 7. Table 6 showing how long it took for the generator to create a build order for the puzzle and export it in the appropriate format, and since the generator functions by repeat generating puzzles until a unique one is made, Table 7 shows how many puzzles were generated before a unique one was made.

Size	Test Number (Hours:Minutes:Seconds)									
	1	2	3	4	5	6	7	8	9	10
4	<0:00:01	<0:00:01	<0:00:01	<0:00:01	<0:00:01	<0:00:01	<0:00:01	<0:00:01	<0:00:01	<0:00:01
5	<0:00:01	<0:00:01	<0:00:01	<0:00:01	<0:00:01	<0:00:01	<0:00:01	0:00:15	<0:00:01	0:00:16
6	0:01:45	0:02:29	0:01:15	0:01:15	0:01:03	00:00:15	0:04:01	0:05:18	0:02:31	0:02:46
7	0:02:10	0:01:40	0:06:53	0:02:30	0:22:05	0:21:24	0:26:17	0:03:08	0:07:36	0:32:07
8	0:30:07	0:56:40	02:59:26	0:41:36	0:40:39	0:40:11	8:11:35	8:32:26	1:14:03	2:57:04
9	Failure	Failure	Failure	Failure	Failure	Failure	Failure	Failure	Failure	Failure
10	Failure	Failure	Failure	Failure	Failure	Failure	Failure	Failure	Failure	Failure

Table 6: *Unique Puzzle Generation Runtime Tests.*

Size	Test Number									
	1	2	3	4	5	6	7	8	9	10
4	1	1	1	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	3	1	4
6	11	15	7	9	8	2	22	38	15	17
7	8	7	27	9	80	75	93	12	26	146
8	121	226	714	167	164	159	1963	2044	299	709
9	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed
10	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed

Table 7: *Unique Puzzle Generation Count Tests.*

In testing backtracking, on the initial attempt at solving any of the three puzzles (Janko 200, 210 and 220, see Figure 19) the solving time exceeded 24 hours, however this does not mean that the backtracker was working incorrectly. Because of this, the test was adjusted to backtrack for one hour, but log the solving order to see that if in the hour any duplicate paths were searched. Through this test, across all three puzzles a total of 606,231 nodes were explored with none of the backtracker sessions showing any duplicate paths explored.

Puzzle	Nodes Explored	Duplicate Paths Explored
Janko 200	189,703	0
Janko 210	132,056	0
Janko 220	284,472	0

Table 8: *Janko Backtracker Test Results.*

The instance generation test results can be seen below in Table 9. These show overall that puzzles can be created relatively quickly, all instances below size 9 showing less than a second in generation time. Even on the largest puzzle of size 10 generation remains quick excluding Test 3 which appears to be an outlier as it almost doubles the next largest result.

Puzzle	Test (Seconds (4 d.p))									
	1	2	3	4	5	6	7	8	9	10
4	0.004	0.0007	0.0011	0.0015	0.0015	0.0008	0.0011	0.0013	0.0008	0.0008
5	0.004	0.0032	0.0046	0.0008	0.0032	0.0068	0.0062	0.0012	0.0049	0.0054
6	0.0098	0.0042	0.0018	0.0029	0.0038	0.0027	0.0042	0.0036	0.0018	0.002
7	0.0086	0.0249	0.0101	0.008	0.0075	0.0049	0.0083	0.0052	0.0068	0.0065
8	0.0453	0.0478	0.0133	0.0402	0.0155	0.0298	0.0286	0.0427	0.0145	0.0257
9	0.0922	0.0588	0.0728	0.0445	2.0119	0.077	0.0637	0.0773	0.0768	2.0171
10	1.0497	5.0761	18.0861	9.045	2.0657	6.0006	0.063	8.0673	1.0093	5.0569

Table 9: *Generator Test Results.*

10.1.3 Testing Conclusions

Due to the functional testing being mostly used as a method of debugging during development, few conclusions can be drawn from this however, seeing that all of the unit tests return true on multiple sessions it can be said that the likelihood that the program is running correctly is high. This can also be seen when manually viewing how the program is running by enabling the test flags, when viewing the program working no visible errors were able to be seen.

One of the largest issues immediately noticeable is how quickly the runtime and generations increase as the puzzle gets larger at an exponential rate, this makes it clear to see why on sizes 8 and 9 the time limit was exceeded. The data shows incredibly high variance too, with at size 8 some puzzles taking upwards of eight hours, this cannot be left down to an outlier as it occurred twice, and contrasting this some has a runtime of less than an hour. However based on this, we can conclude that once the puzzle exceeds size 5, the likelihood of the initial generation being a unique puzzle is very low meaning that multiple puzzles need to be tested vastly increasing runtime. The average time to check an instance at size 6 was just ten seconds then past this it was always averaging 15, meaning that it can be assumed that the backtracker was needing to be used on each instance as 15 seconds was the timeout duration for the backtracker. It can be drawn from this that potentially increasing the backtracker timeout duration may have found a unique puzzle faster as one of the abandoned puzzle may have been a solution.

With regard to the backtracker, tests showed that it explores an average of 3,367 nodes per minute, this varying depending on the complexity of the puzzle. For example assuming two puzzles have the same number of nodes however one has more leaf nodes, the one with more leaf nodes will likely be explored faster as it is easier to evaluate if a node has no branches than it is to compute the number of branches the node has. Because of this, it is likely that Janko 220 (Figure 19c) has far more leaf nodes than the other two tested as it was explored at around 4761.2 (1 d.p) nodes per minute where Janko 200 (Figure 19a) was explored at 3161.7 (1 d.p) and Janko 210 (Figure 19b) at just 2201 (1 d.p) nodes per minute.

Testing generation times alone can only yield so many answers as to where the program is working at its slowest as even if they generator is churning out puzzles quickly, unless they are unique they will be rejected by the solver. All this allows for is the solver to be supplied with puzzles of varying quality quickly which functions however this just shifts where the bottleneck of the program lies. Overall however the generator is able to supply the solver with puzzles at a good rate allowing for minimal wait time between testing puzzles.

Overall, the test results can only be so accurate as the sample size created by the tests is still small, for more accurate results more tests would need to be done. However the number of tests completed are enough to gain a baseline of the quality of how the program operates.

11 Evaluation

The following section will evaluate not just the program, but the development process that supported it.

11.1 Product Evaluation

11.1.1 Build Quality

The build quality of the program is adequate for the purpose however is lacking in areas that could push it further.

As to be expected with a brute force approach the process of solving is messy, the both the backtracker and generator act aimlessly. The operate within the bounds of what is possible however their is no aim with the approach they take, it is to simply exhaust every possible outcome until a solution is found. The current approach to generation can be described as a needle-in-haystack approach (Andricioaei and Straub 1996) and while this works, guiding both of these components more would improve their performance massively. For example, if the generator had a method of knowing where to put relations rather than putting them simply where they are valid, the quality of puzzles sent to the solver would increase massively. Akin to this, if the backtracker had an idea of which branches were likely to lead to solutions it could focus on these first to potentially find duplicate solutions faster resulting in instances being abandoned faster if they are confirmed non-unique.

However the application of flyweight in the program has helped it massively. It was observed at some points that the backtracker was operating at a depth of 30, meaning that 30 *State* classes were having to be stored. Rather than having to store and use 30 *Futoshiki*, the backtracker could simply store *State* at much lower cost and if necessary, could build a *Futoshiki* from this. A similar approach helped the generator as when generating inequalities, rather than creating a vector of *Relation* for each inequality, each *Relation* was able be represented as an integer array that simply contained each of the four pieces of information key to a relation, the greater cell row and column, and the lesser cell row and column.

The implementation of ExecutorService was also a good choice as it allowed for a timer to be set on puzzles but unlike a typical timer, it allowed it to be created in an alternative thread to protect runtime and ensure that if the function that was running inside it got stuck, the timer would not as it was running asynchronously. Potentially the idea of multi-threading could have been pushed further however allowing for different components of generation to run on different threads, such as the next puzzle being generated while the current is being checked for uniqueness.

It was initially deemed that during development an entire solution tree was to be mapped by the backtracking solver. While this still reigns true, the entire tree does not need to be stored. Only two components need to be stored within the solver, any solutions found, and the route to the current node of interest. In doing this memory requirements are lowered massively as no longer is it required that past routes are saved.

Along with this, all of the code is commented in a consistent format to ensure that all functions of the code are broken down so that through development and post-development, the program can be easily interpreted. The program has also been cleared of unused functions or variables to improve clarity of code.

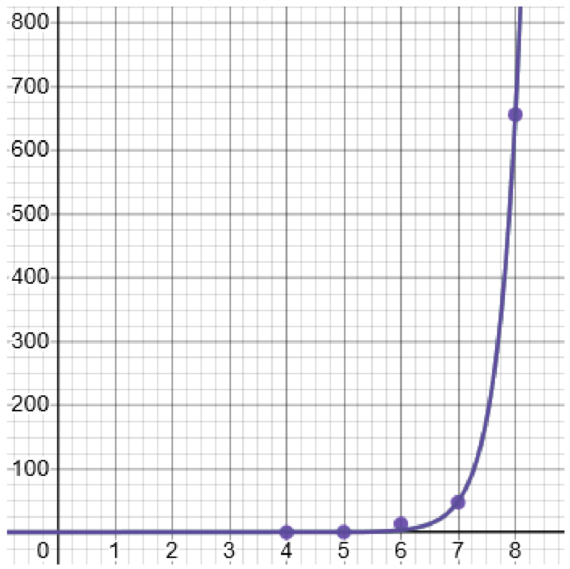
11.1.2 Fitness for Purpose

The overall goal of the project was to create a program that could generate Futoshiki puzzles larger than $9 \times 9$and while during tests it did not display that it could do this, all of the other tests point towards that it potentially could. The limit for testing generating a unique puzzle was 24 hours and while for sizes 9 and 10 it failed this it did not fail due to error, it failed due to timeout. Due to the nature of the generator, in theory it will eventually find a puzzle that is unique as it simply populates an empty Latin square with the features of a Futoshiki, past this it is on the solver and backtracker to verify this.

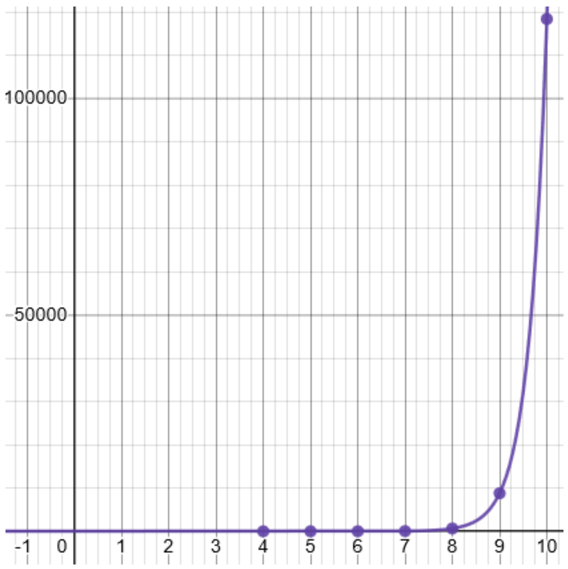
However the backtracker also showed to be functioning correctly during testing. The reason that the 24 hour time limit was exceeded is likely due to the amount of nodes that the larger puzzles contain. For example, Janko 200 (Figure 19a) has 404 branches from the from a random node at depth 1, 385 branches from a random node at depth 2 and 372 from from a random node at depth 3. The number of branches continues to decrease at a similar rate to this until around depth 28 is reached. Assuming that each of the nodes at these depths contain the same number of branches, it can be estimated that the number of nodes contained in the Janko 200 solution tree is $1.175592752 \times 10^{56}$ (10 s.f). Assuming that no solution is found during backtracking and the solver searched every node at the rate displayed during testing (3,161 nodes per second), it would take $7.074257828 \times 10^{44}$ (10 s.f) years to trace the puzzle. It can be concluded based on testing that it is likely that the program could generate 10×10 puzzles, however just not in any feasible amount of time.

Using the data set from Table 7, and taking the average generation count for each puzzle size then plotting this on a graph, it can be estimated how long and how many generations puzzles larger than 8×8 would have taken. This can be done using exponential regression (Matthys and Beirlant 2003), this involves taking data points that grow at an exponential rate and creating a curve of best fit to estimate where values that were not already known might be. This was done using the graphing software software Desmos (*Desmos* 2022). Using exponential regression Figure 1 was made and from this values can be projected. The curve produced by this can be seen in Figure 23, with Figure 23a showing only the known points, and Figure 23b including the projected points at size 9 and 10.

$$y = \left(6.1393 \times 10^{-7}\right) \times 13.445^{x} + 1 \quad (1)$$



(a) *Without Projected Points.*



(b) *With Projected Points*

Figure 23: *Curve of Best Fit*

From this it can be seen that a size 9 and 10 puzzle would take an estimated 8,814.8 (1 d.p) and 118,503.3 (1 d.p) respectively. Because past size 7, the testing samples showed an average of 15 seconds, it can be estimated that if the tests were persisted past the 24 hour limit, a size 9 puzzle would have taken 36.7 hours (1 d.p) to generate and a puzzle of size 10 would have taken 493.8 hours (1 d.p) to generate. However both of these estimations do assume that the solver runs flawlessly which cannot be properly evaluated for large puzzles, it also assumes that the number of inequalities and values being initially added allow the puzzle to be solved either by just the solver or the solution tree only contains around 750 nodes as this is all the solver would have time to check before the puzzle was abandoned.

11.1.2.1 Evaluating Against Requirements To further evaluate the fitness for purpose, the program can be compared to the requirements specification (Shown in Figure 6).

As many of these requirements were met, only the ones that were not met or require further explanation will be discussed.

A puzzle generator that is able to generate a puzzle larger than 9×9 ***with a single solution.***

As discussed above, it is likely that the program could do this, however there is no way to verify this due to the amount of time required.

A backtracking solver that is able to count the number of solutions to a Futoshiki puzzle.

The backtracker supports this, however instance is abandoned after finding two solutions so this functionality is disabled. If in further development of the program this was required it could be enabled.

The backtracker can create a tree of the search order when searching for solutions.

A form of search tree is explored, however only the route from the root node to the current node is stored as the previous branches do not require to be stored because if they contain any features that are required to be stored such as solutions, this can be done without storing the entire branch.

The generator can be run within a single executable.

This was not done due to the program being in a state where its ability to complete its intended purpose is still debatable, meaning exporting it into a run-able format is of little use. Leaving it in its current format would be better for further development that could occur.

The generator can add constraints optimally.

This is not done as the generator adds constraints randomly. The reason behind this is because of the exhaustive approach taken to designing the whole program not supporting the generator understanding how constraints should be added causing them to be added randomly based on where possible.

Provide fast puzzle generation.

As discussed earlier, this varies on the size of the puzzle however for the size of the puzzle that are designed to be generated the program does not support fast generation. This comes down to mostly how unguided most of the components of the program are.

Integration with an external framework (ie. J-POP) for final puzzle testing.

The lack of this mostly comes down to J-POP being written in C++, and while this would still be possible the complexity of integration outweighs the necessity for its implementation.

Prevention's in place to reduce likelihood of duplicate puzzles being created.

In a manor this is in place, because the values and inequalities being input into the puzzle are randomly generated, the chances of receiving duplicate puzzles is lowered. However due to puzzles of the same size containing the same number of constraints, the likelihood of receiving a duplicate puzzle is increased slightly, reducing this further could be done through adding variance to the number of constraints added.

The ability to rate puzzles based on the solution search tree.

This could have been done by counting the number of nodes found by the solver, however the need for implementation was low and may have taken away from developing more important concepts. For larger puzzles such as Janko 200 (Figure 19a), the number of nodes would be incalculable within Java and would have thrown an error.

11.1.2.2 Evaluating Against Terms of Reference The project often shifted form during development, and while this may make it seem as though development was unfocused this allowed for the project to be shaped by the environment it was being developed in rather than a set of boundaries defined before development was started. And while the aims stayed the same, objectives changed as the freedom for the project to take shape while still pertaining to the initial aims meant that ideas that may have appeared possible before development, such as adjusting the solution tree as constraints were added, did not need too much time to be committed to as focus was able to be shifted to other or new objectives. An example of this can be seen in suggesting that the puzzle be exported in readable format, while this would be useful, through development it was decided that the format that J-POP used to import puzzles would be more effective as it could lead to integration down the line.

Not all objectives were adjusted however, for example the program does not break memory requirements, and while not met ensuring minimal runtime was still an objective of the project. Similar to this a vast research was done into literature regarding Latin square puzzles and Futoshiki creating a strong foundation for the project to be built on top of. Alongside this lots of documentation was created, this including a sequence diagram and pseudocode algorithm for the more complex parts of the program, and various levels of class diagrams for the more structurally complex areas of development.

Overall objectives were followed as they were seen fit, if the project no longer deemed an objective fit, it was abandoned. This meant adjusting objectives to ensure the project could meet its aims.

11.2 Process Evaluation

11.2.1 Tools and Methods

All of the tools stated in Section 7 were used in development and while all of these tools fulfilled their purpose sufficiently, retrospectively a few changes would have been made. Instead of using Eclipse, IntelliJ (*IntelliJ* 2022) may have been a better alternative. IntelliJ is a JetBrains based IDE (*JetBrains* 2022); where Eclpise offers a more open-source plugin oriented approach, IntelliJ aims to offer a more complete out-the-box experience without the need for searching for plugins that may pick up on where the IDE is lacking. IntelliJ also offers plugin support so if in a case where it was missing a feature that Eclipse had, there is likely a plugin to fix it. The reason that it was initially not selected was due to lack of experience with it, however this project may have offered a good starting point to learn how to use it.

The UML development tools also fall into a similar category of which they were able to successfully create accurate UML artifacts, however UML Lab (*umllab* 2022) may have offered a more whole package. Rather than having to rely on two programs in UMLet and Umbrello to create artifacts, UML Lab offered support for both to a high level, and may have also allowed for opportunity to create artifacts that the aforementioned two did not enabling greater planning and documentation of the project.

Another tool was added early in development, GitHub. GitHub is a code hosting platform enabling version control (*GitHub* 2022), this allowed for storing cloud backups of code and reverting to older versions of the code. This meant that should a problem emerge in the form of loss of code, a backup could be restored from, or if code was overwritten it could be restored from a previous version. It also allowed for easy sharing of code with the supervisor (Sinclair, M.C, October 2021). While alternatives to GitHub such as BitBucket (*BitBucket* 2022) exist, in the scope of the project GitHub offered a sufficient service.

Development overall loosely followed the Gantt chart proposed in the terms of reference, time allocation remained flexible to allow for time to flow over should other areas require more time. For example the backtracker took far more time than the Gantt chart suggested due to its complexity, but generating non-unique instances took far less. Ordering of these was also switched early in development, backtracking being completed first, this was due to the complexity of backtracking and the none reliance on the generator as the backtracker could be provided with puzzles that had predefined outcomes for testing. Overall the Gantt chart was good for giving the project structure, however it felt restrictive and because of this, as stated prior, the timeline was flexed in places.

11.2.2 Personal Development

With this being a Java based project, my Java skills were built upon. As mentioned earlier, multi-threading was a topic I had not used within Java, only within C, however using ExecutorService I have since learnt how this can be used and will apply it future projects for improved efficiency. Along with this, my understanding of Flyweight design methods and how these can be applied to reduce memory usage. My ability to develop large programs while also building off of an existing project was also tested and developed significantly. Developing my own tests and working with new testing frameworks such as JUnit has also changed my approach to development and how tests can help to shape development and assist with debugging.

My set theory skills have also improved. Many of the papers studied in the literature review were used heavy set theory to explain constraints and puzzle solving algorithms, through reading many of these my understanding of set theory is now at confident level.

However my biggest development would be in learning how to use LaTeX. LaTeX is the word processing software used to write this report, offering an easier method for referencing and designing figures. This allowed for creating a more fluid report and a more professional look. While it has a steep learning curve, I will use it for future reports as it offers a good package for creating academic literature.

11.3 Conclusions and Recommendations

11.3.1 Aims and Objectives

Initially, in the *Aims of Project* section of the terms of reference, three aims were outlined. Two of those were met, thee program contains a backtracking solver with the capability to construct trees of solutions and throughout development, constraints were constantly being unofficially tested and changed to see which would give the best results. If the generator was taking too long, too many constraints were added, and if the backtracker was deeming puzzles invalid too fast, there were not enough constraints. The final objective was to be able to generate large Futoshiki puzzles with unique solutions, and while if the program can do this or not is still unverified, it can be said that it does not do this in a sufficient amount of time.

Majority of the goals set in the *Objectives* section were met, even if some were adjusted slightly more to fit the scope of development as it was progressed through. A few of the objectives had to be dropped after the research phase made it clear that these would not be achievable using the approach this study took, some examples of these being, the ability to dynamically add constraints and adjusting the solution tree rather than regenerating. Dynamically adding constraints would mean that the generator would need a method of knowing where constraints should be added based on where the puzzle went wrong, to do this it would need a form of analysis system to know which features are good and which are bad. Adjusting the solution tree became difficult due to the waterfall effect that adding values creates, for example if a value is changed, any constraint set with this value is also changed, changing all of the cells in that constraint set, this process is then repeated for those cells in the constraints set. Because of this the tree would be changing too much to make most of it unusable and by the time the the whole tree was checked for changes, it was likely going to be faster to just remake the solution tree with the new constraints added.

11.3.2 Conclusion

While overall the projects overarching goal of generating large Futoshiki puzzles was not met, understandings of the concepts that would allow for this to happen were met and with further development it is likely that this project could generated large unique puzzles. And while the largest challenges in the backtracker and generator were not completed to a perfect standard, the implementation used leads to a functioning model that offers a solid foundation for both a puzzle generator and backtracking solver.

If this project were to be restarted, development could have been done using languages that offer more control over how memory is used such as C or C++. While this would have meant fully rebuilding the supplied code in those languages, it may have in the long run yielded better results due to the increased amount of control.

As a program designed to push the limits in what is possible in Futoshiki generation, the software offers a solid foundation that can hopefully lead into a functioning large Futoshiki generator.

11.3.3 Recommendations

A few approaches could be taken to improving how the functions. Adding more solving methods would reduce the number of paths that the solver would go down, as it would reduce the number of potential assigns at each level. With more development time methods such as the ones discussed in Section 5.2.2 could be implemented, however the complexity of these exceeded the limited development time of this project. Further methods for solving outside of this report can also be found such as Triples and Quadruples or Highs and Lows (*Futoshiki Strategies* 2022).

Discussed in Section 11.1.1, was the idea of guiding the generator and backtracker. As stated in that section this would require a form of data set to provide backing, this could be done through genetic algorithms. This would offer features that the generator could look to replicate and form its puzzles off of, potentially reducing the number of instances that would be need to sent to the backtracker before a unique solution is found. Similarly they could be implemented into the backtracker to allow it to recognise features and allow for easier solving of puzzles, J-POP offers this form of solving (Lloyd et al. 2021) and they have been used in order to solve other optimisation problems (Deep et al. 2009). However as discussed in Section 5.4.1 these exceed the means of the project.

Genetic algorithms are not the only solving methods that could be implemented. Some of the other effective solving methods as shown by J-POP could also be implemented such as Simulated Annealing (Huang and Hsieh 2011) and Ant Colony Optimisation (Ping et al. 2014).

12 Bibliography

References

Ahmad, Khadija Sania et al. (2017). “Fuzzy_MoSCoW: A fuzzy based MoSCoW method for the prioritization of software requirements”. In: *2017 International Conference on Intelligent Computing, Instrumentation and Control Technologies (ICICICT)*, pp. 433–437. DOI: 10.1109/ICICICT1.2017.8342602.

Alic, Dino, Samir Omanovic, and Vaidas Giedrimas (2016). “Comparative analysis of functional and object-oriented programming”. In: *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 667–672. DOI: 10.1109/MIPRO.2016.7522224.

Andricioaei, Ioan and John E Straub (1996). “Finding the needle in the haystack: Algorithms for conformational optimization”. In: *Computers in Physics* 10.5, pp. 449–455.

Angela and Otto Janko (2020). *Janko Futoshiki.* URL: https://www.janko.at/Raetsel/Futoshiki/index.htm (visited on 04/10/2022).

Baca, Pavol and Valentino Vranic (2011). “Replacing Object-Oriented Design Patterns with Intrinsic Aspect-Oriented Design Patterns”. In: *2011 Second Eastern European Regional Conference on the Engineering of Computer Based Systems*, pp. 19–26. DOI: 10.1109/ECBS-EERC.2011.13.

baeldung (2021). *A Guide to the Java ExecutorService.* URL: https://www.baeldung.com/java-executor-service-tutorial (visited on 04/23/2022).

Berthier, Denis (2013). “Pattern-based constraint satisfaction and logic puzzles”. In: *arXiv preprint arXiv:1304.1628.*

BitBucket (2022). URL: https://bitbucket.org/product (visited on 05/01/2022).

Bogaerts, Bart et al. (Aug. 2020). “Step-wise Explanations of Constraint Satisfaction Problems”. English. In: *ECAI 2020 - 24th European Conference on Artificial Intelligence, including 10th Conference on Prestigious Applications of Artificial Intelligence, PAIS 2020*

- *Proceedings*. Ed. by Giuseppe De Giacomo et al. Vol. 325. Frontiers in Artificial Intelligence and Applications. Netherlands: IOS Press, pp. 640–647. ISBN: 978-1-64368-100-9. DOI: `10.3233/FAIA200149`.

BorisTheBrave (2022). *Dungeon Generation in Binding of Isaac*. URL: `https://www.boristhebrave.com/2020/09/12/dungeon-generation-in-binding-of-isaac/` (visited on 04/23/2022).

Briand, Lionel and Yvan Labiche (Sept. 2002). “A UML-Based Approach to System Testing”. In: *Software and Systems Modeling* 1.1, pp. 10–42. DOI: `10.1007/s10270-002-0004-8`. URL: `https://doi.org/10.1007/s10270-002-0004-8`.

Burn, R. P. (1975). In: *The Mathematical Gazette* 59.408, pp. 116–117. ISSN: 00255572. URL: `http://www.jstor.org/stable/3616653`.

Chau, Thomas et al. (2017). “Chapter Two - Advances in Dataflow Systems”. In: ed. by Ali R. Hurson and Veljko MilutinoviÄ‡. Vol. 106. Advances in Computers. Elsevier, pp. 21–62. DOI: `https://doi.org/10.1016/bs.adcom.2017.04.002`. URL: `https://www.sciencedirect.com/science/article/pii/S0065245817300128`.

Chlond, Martin J (2013). “Puzzle—Latin Square Puzzles”. In: *INFORMS Transactions on Education* 13.2, pp. 126–128.

Deep, Kusum et al. (2009). “A real coded genetic algorithm for solving integer and mixed integer optimization problems”. In: *Applied Mathematics and Computation* 212.2, pp. 505–518. ISSN: 0096-3003. DOI: `https://doi.org/10.1016/j.amc.2009.02.044`. URL: `https://www.sciencedirect.com/science/article/pii/S0096300309001830`.

Desmos (2022). URL: `https://www.desmos.com` (visited on 05/01/2022).

Dybvig, R Kent (2009). *The Scheme programming language*. Mit Press.

Eclipse (2022). URL: `https://www.eclipse.org/eclipseide/` (visited on 04/23/2022).

Emmanuel, Kati Steven et al. (2019). “A Beginners Guide to Procedural Terrain Modelling Techniques”. In: *2019 2nd International Conference on Signal Processing and Communication (ICSPC)*, pp. 212–217. DOI: `10.1109/ICSPC46172.2019.8976682`.

Ercsey-Ravasz, Mária and Zoltán Toroczkai (Oct. 2012). “The Chaos Within Sudoku”. In: *Scientific Reports* 2.1, p. 725. ISSN: 2045-2322. DOI: `10.1038/srep00725`. URL: `https://doi.org/10.1038/srep00725`.

Esteve, Miriam et al. (2021). “Heuristic and Backtracking Algorithms for Improving the Performance of Efficiency Analysis Trees”. In: *IEEE Access* 9, pp. 17421–17428. DOI: `10.1109/ACCESS.2021.3054006`.

Futoshiki (2022). URL: `https://www.futoshiki.org/` (visited on 05/02/2022).

Futoshiki Strategies (2022). URL: `https://www.atksolutions.com/articles/futoshikistrategies.html` (visited on 05/01/2022).

Gamma, Erich and Richard Helm (1995). *Design patterns : elements of reusable object-oriented software*, pp. 375–381.

GitHub (2022). URL: `https://github.com/` (visited on 05/01/2022).

Goldberg, Adele and David Robson (1983). *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc.

Hamill, Paul (2004). *Unit test frameworks: tools for high-quality software development.* " O'Reilly Media, Inc.".

Hansen, Stuart (2004). "The Game of set®: An Ideal example for introducing polymorphism and design patterns". In: *ACM SIGCSE Bulletin* 36.1, pp. 110–114.

Hansson, Daniel (2014). "Continuous Linting with Automatic Debug". In: *2014 15th International Microprocessor Test and Verification Workshop*, pp. 70–72. DOI: `10.1109/MTV.2014.25`.

Haraguchi, Kazuya (2013). "The number of inequality signs in the design of Futoshiki puzzle". In: *Journal of information processing* 21.1, pp. 26–32.

Haraguchi, Kazuya and Hirotaka Ono (2014). "Approximability of latin square completion-type puzzles". In: *International Conference on Fun with Algorithms.* Springer, pp. 218–229.

Hayardeny, A., S. Fienblit, and E. Farchi (2004). "Concurrent and distributed desk checking". In: *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* Pp. 265–. DOI: `10.1109/IPDPS.2004.1303337`.

Hinsen, Konrad (2009). "The Promises of Functional Programming". In: *Computing in Science Engineering* 11.4, pp. 86–90. DOI: `10.1109/MCSE.2009.129`.

Huang, Kou-Yuan and Yueh-Hsun Hsieh (2011). "Very fast simulated annealing for pattern detection and seismic applications". In: *2011 IEEE International Geoscience and Remote Sensing Symposium*, pp. 499–502. DOI: `10.1109/IGARSS.2011.6049174`.

Hudak, Paul and Joseph H Fasel (1992). "A gentle introduction to Haskell". In: *ACM Sigplan Notices* 27.5, pp. 1–52.

Hunt, Martin, Christopher Pong, and George Tucker (2007). "Difficulty-driven sudoku puzzle generation". In: *The UMAP Journal* 29.3, pp. 343–362.

IntelliJ (2022). URL: `https://www.jetbrains.com/idea/` (visited on 05/01/2022).

Ishibuchi, H. and T. Murata (1996). “Multi-objective genetic local search algorithm”. In: *Proceedings of IEEE International Conference on Evolutionary Computation*, pp. 119–124. DOI: 10.1109/ICEC.1996.542345.

JetBrains (2022). URL: https://www.jetbrains.com/ (visited on 05/01/2022).

JUnit5 (2022). URL: https://junit.org/junit5/ (visited on 04/23/2022).

Khan, Mohd et al. (2011). “Different approaches to black box testing technique for finding errors”. In: *International Journal of Software Engineering & Applications (IJSEA)* 2.4.

Khoshafian, Setrag N. and George P. Copeland (June 1986). “Object Identity”. In: *SIGPLAN Not.* 21.11, pp. 406–416. ISSN: 0362-1340. DOI: 10.1145/960112.28739. URL: https://doi.org/10.1145/960112.28739.

Li, Huawen and Qingjie Wang (2009). “Proxy Pattern Informatization Research Based On SaaS”. In: *2009 IEEE International Conference on e-Business Engineering*, pp. 518–521. DOI: 10.1109/ICEBE.2009.99.

Lloyd, Huw et al. (2021). “J-POP: Japanese Puzzles as Optimization Problems”. In: *IEEE Transactions on Games*, pp. 1–1. DOI: 10.1109/TG.2021.3081817.

Lucas, Francisco J., Fernando Molina, and Ambrosio Toval (2009). “A systematic review of UML model consistency management”. In: *Information and Software Technology* 51.12. Quality of UML Models, pp. 1631–1645. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2009.04.009. URL: https://www.sciencedirect.com/science/article/pii/S0950584909000433.

Maleki, Sepideh et al. (2017). “Understanding the impact of object oriented programming and design patterns on energy efficiency”. In: *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*, pp. 1–6. DOI: 10.1109/IGCC.2017.8323605.

Mantere, Timo and Janne Koljonen (2007). “Solving, rating and generating Sudoku puzzles with GA”. In: *2007 IEEE Congress on Evolutionary Computation*, pp. 1382–1389. DOI: 10.1109/CEC.2007.4424632.

Mathew, Kuruvilla and Mujahid Tabassum (2014). “Experimental comparison of uninformed and heuristic AI algorithms for N Puzzle and 8 Queen puzzle solution”. In: *Int. J. Dig. Inf. Wireless Commun.(IJDIWC)* 4.1, pp. 143–154.

Matthys, Gunther and Jan Beirlant (2003). “Estimating the extreme value index and high quantiles with exponential regression models”. In: *Statistica Sinica*, pp. 853–880.

Meyer, Bertrand (2002). *Design by contract.* Prentice Hall Upper Saddle River.

Nidhra, Srinivas and Jagruthi Dondeti (2012). “Black box and white box testing techniques-a literature review”. In: *International Journal of Embedded Systems and Applications (IJESA)* 2.2, pp. 29–50.

Nikoli (2022). URL: `https://www.nikoli.co.jp/en/` (visited on 05/02/2022).

Nuić, H. and Ž. Mihajlović (2019). "Algorithms for procedural generation and display of trees". In: *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 230–235. DOI: `10.23919/MIPRO.2019.8757140`.

Palomo, Miguel G. (Feb. 2014). "Latin Polytopes". In: *arXiv e-prints*, arXiv:1402.0772, arXiv:1402.0772. arXiv: `1402.0772 [math.CO]`.

— (2016). *Latin Puzzles.* arXiv: `1602.06946 [math.HO]`.

Ping, Gu et al. (2014). "Adaptive ant colony optimization algorithm". In: *2014 International Conference on Mechatronics and Control (ICMC)*, pp. 95–98. DOI: `10.1109/ICMC.2014.7231524`.

Pizlo, Filip and Jan Vitek (2008). "Memory Management for Real-Time Java: State of the Art". In: *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pp. 248–254. DOI: `10.1109/ISORC.2008.40`.

Puzzle Madness (2022). URL: `https://puzzlemadness.co.uk/16by16giantsudoku/medium` (visited on 05/02/2022).

Rawlings, R. (1989). "Objective-C: an object-oriented language for pragmatists". In: *IEE Colloquium on Applications of Object-Oriented Programming*, pp. 2/1–2/3.

Reid, S.C. (1997). "An empirical analysis of equivalence partitioning, boundary value analysis and random testing". In: *Proceedings Fourth International Software Metrics Symposium*, pp. 64–73. DOI: `10.1109/METRIC.1997.637166`.

Sahu, Himanshu Sekhar, Sisir Kumar Nayak, and Sukumar Mishra (2016). "Maximizing the Power Generation of a Partially Shaded PV Array". In: *IEEE Journal of Emerging and Selected Topics in Power Electronics* 4.2, pp. 626–637. DOI: `10.1109/JESTPE.2015.2498282`.

Sargeant, John (1993). "Uniting functional and object-oriented programming". In: *International Symposium on Object Technologies for Advanced Software*. Springer, pp. 1–26.

Schottlender, Moriel (2014). "The effect of guess choices on the efficiency of a backtracking algorithm in a Sudoku solver". In: *IEEE Long Island Systems, Applications and Technology (LISAT) Conference 2014*, pp. 1–6. DOI: `10.1109/LISAT.2014.6845190`.

Sheng, Nan and Yahai Wang (2018). "The Software Design of Modular Instrument Based on Proxy Pattern". In: *2018 Chinese Automation Congress (CAC)*, pp. 976–979. DOI: `10.1109/CAC.2018.8623470`.

Shortz, Will (2006). *The 2006 TIME 100.* URL: `http://content.time.com/time/specials/packages/article/0,28804,1975813_1975838_1976198,00.html` (visited on 05/02/2022).

Singh, Yogesh (2011). *Software Testing.* USA: Cambridge University Press. ISBN: 1107012961.

Snyder, Alan (1986). "Encapsulation and inheritance in object-oriented programming languages". In: *Conference proceedings on Object-oriented programming systems, languages and applications*, pp. 38–45.

Spieler, Jörg (2022). *UCDetector.* URL: `http://www.ucdetector.org/` (visited on 04/23/2022).

Steele, Guy (1990). *Common LISP: the language.* Elsevier.

Stefik, Mark and Daniel G Bobrow (1985). "Object-oriented programming: Themes and variations". In: *AI magazine* 6.4, pp. 40–40.

Sudoku (2022). URL: `https://sudoku.com/` (visited on 05/02/2022).

Sudoku techniques (n.d.). URL: `https://www.conceptispuzzles.com/index.aspx?uri=puzzle%5C%2Fsudoku%5C%2Ftechniques`.

Sun, Baochen et al. (2008). "A New Algorithm for Generating Unique-Solution Sudoku". In: *2008 Fourth International Conference on Natural Computation.* Vol. 7, pp. 215–217. DOI: `10.1109/ICNC.2008.788`.

Tatham, Simon (2022). *Simon Tatham's Portable Puzzle Collection.* URL: `https://www.chiark.greenend.org.uk/~sgtatham/puzzles/` (visited on 04/23/2022).

Top IDE Index (2022). URL: `https://pypl.github.io/IDE.html` (visited on 04/23/2022).

Tsai, Jinn-Tsong and Ping-Yi Chou (2011). "Solving Japanese puzzles by genetic algorithms". In: *2011 International Conference on Machine Learning and Cybernetics*. Vol. 2, pp. 785–788. DOI: `10.1109/ICMLC.2011.6016787`.

Tsantalis, Nikolaos, Davood Mazinanian, and Shahriar Rostami (2017). "Clone Refactoring with Lambda Expressions". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 60–70. DOI: `10.1109/ICSE.2017.14`.

Umbrello (2022). URL: `https://umbrello.kde.org/` (visited on 04/23/2022).

UMLet (2022). URL: `https://www.umlet.com/` (visited on 04/23/2022).

umllab (2022). URL: `https://www.uml-lab.com/en/uml-lab/` (visited on 05/01/2022).

Viana, Breno M. F. and Selan R. dos Santos (2019). "A Survey of Procedural Dungeon Generation". In: *2019 18th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, pp. 29–38. DOI: `10.1109/SBGames.2019.00015`.

Visual Studio Code (2022). URL: `https://code.visualstudio.com/` (visited on 04/28/2022).

Vonthron, Andre, Christian Koch, and Markus König (Feb. 2018). "Removing duplicated geometries in IFC models using rigid body transformation estimation and flyweight design pattern”. In: *Visualization in Engineering* 6.1, p. 2. ISSN: 2213-7459. DOI: 10.1186/s40327-018-0061-x. URL: https://doi.org/10.1186/s40327-018-0061-x.

Wegner, Peter (Aug. 1990). “Concepts and Paradigms of Object-Oriented Programming”. In: *SIGPLAN OOPS Mess.* 1.1, pp. 7–87. ISSN: 1055-6400. DOI: 10.1145/382192.383004. URL: https://doi.org/10.1145/382192.383004.

Weir, C.A.F. (1993). “Case study: implementing an object-oriented design in C++”. In: *IEE Colloquium on Recent Progress in Object Technology*, pp. 8/1–8/2.

Yang, Xin-She (2021). “Chapter 6 - Genetic Algorithms”. In: *Nature-Inspired Optimization Algorithms (Second Edition)*. Ed. by Xin-She Yang. Second Edition. Academic Press, pp. 91–100. ISBN: 978-0-12-821986-7. DOI: https://doi.org/10.1016/B978-0-12-821986-7.00013-5. URL: https://www.sciencedirect.com/science/article/pii/B9780128219867000135.

13 Appendices

13.1 Appendix 1 - Terms of Reference

KV6003: Individual Computing Project
Project Terms of Reference

Connor Campbell
W18003255
Computer Science Pure

Generation of large Futoshiki puzzles with unique solutions.

Supervisor – Mark Sinclair
Second Marker - Kezhi Wang

Software Engineering Project

Contents

1. Background of Project

A. Background and Introduction

Puzzle solving currently exists in several forms, from physical puzzles such as the Rubik’s Cube (Rubik's Cube - Wikipedia, 2021), or the Soma cube (Soma cube - Wikipedia, 2021), but many of the more popular formats exist in non-physical form such as mathematical puzzles (Mathematical puzzle - Wikipedia, 2021). These are broken down mainly into two categories, theory puzzles such as crosswords (Daily Crossword, 2021) or Lexica (Lexica Puzzles, 2021), or logic puzzles such as sudoku (Sudoku Online Puzzles, 2021) or futoshiki (Futoshiki Online Puzzles, 2021). The drive behind these puzzles is for the challenge in finding the solution and the satisfaction gained through solving. Because of this, various forms of puzzles have likely been a common pastime for centuries (D. Demaine, L. Demaine and Rodgers, 2008).

As technology has progressed, the creation of non-physical puzzles has become far easier, algorithms can be used to both solve and create puzzles of various difficulties. Often, creating a puzzle is more difficult than solving it due to the nature of the constraints presented by the puzzle (Colton, 2002), for example any puzzle built off of the format of a Latin square (Chlond, 2013) may only have one solution, accounting for other potential solutions is often out of the question when the size of the puzzle exceeds rational size. Algorithms, however, are able to normalise the massive number of potential solutions and ensure that they are reduced down to a single solution. Proof of this is seen in the largest recorded sudoku puzzle ever made (recorded by Guinness World Records) containing 280 grids (Guinness World Records, 2021).

Because the creation of puzzles also involves the ability to solve them, it is a driving force behind artificial intelligence (AI) and machine learning (ML) algorithms as the heart of both puzzle solving and AI is being given a scenario and being able to find the optimal solution. This places puzzle technology at the forefront of most modern-day AI industries (Rao and Mitra, 2008).

B. Solving Futoshiki Puzzles

Futoshiki puzzles are single solution Latin square puzzles that use inequalities as the constraint to their solving. This means there are three main rules in solving a Futoshiki, inequalities must be followed, each column and row cannot have duplicates numbers, and the value of the numbers used cannot exceed the size of the columns or rows (Lloyd, Crossley, Sinclair and Amos, 2021). Figure 1 shows the basic order of solving a Futoshiki.

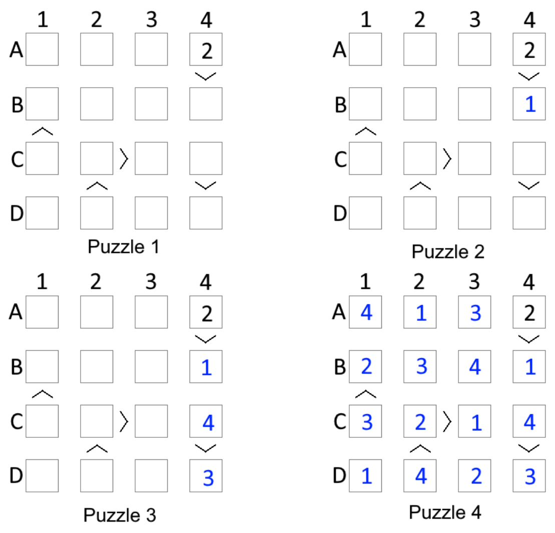


Figure 1 - Futoshiki Solving

Going off of Figure 1 the grid is 4x4 making the max value 4, Puzzle 1 shows cell B4 must contain a number smaller than 2 making it only possible value 1 creating Puzzle 2. Because no duplicate values are allowed in the column's cells C4 and D4 can only contain 3 or 4 and because C4 has to be larger than D4, C4 must contain 4 and D4 must contain 3, making Puzzle 3. Following these solving rules eventually the square is complete and will look like Puzzle 4.

The largest Futoshiki generator available currently is limited to 9x9, however this limitation is seemingly unfounded (Sinclair, M.C, Personal Communication, October 2021). Other Latin square puzzles such as Sudoku are available in sizes larger than 9x9. In the creation of the puzzles to ensure that only one possible solution is available, both inequalities and starting values are used. Defining these two factors is the primary focus of this project.

C. Literature Review

Kazuya Haraguchi has several papers on Latin square puzzle solutions and construction, in specific one on the approximation of Latin square puzzles to ensure that they have a proper solution (Haraguchi and Ono, 2015), and another on how many inequalities need to be used in the Futoshiki order to make the puzzle both solvable and its solution unique (Haraguchi, 2013).

The former discusses a few different algorithms for solving, the *Greedy Algorithm*, a random cell is selected from an empty solution, and a random value from the available set is input, this is repeated until the solution is finished, if the solution is correct the algorithm stops, if in correct the algorithm will repeat again from the empty solution starting with a different value. The *Matching Based Algorithm* counts up the same number of empty cells as the size of the grid, if the grid is 4x4, four empty cells will be classified, and then given values from 1 to 4 in the order they were selected, repeating this until the puzzle is complete. *Local Search* involves applying a typical local search algorithm similar to the ones used in AI solutions, essentially constantly replacing the sets used within the puzzle until an optimal solution is found (Haraguchi and Ono, 2015).

In the construction of the backtracing solver, the J-POP framework offers a good starting point (Lloyd, Crossley, Sinclair and Amos, 2021). The solver is presented with all of the possible choices for values to input that satisfies the constraints, the solver will choose one of these values to input. Once a value is input the solver will repeat the previous step getting a new set of possible choices and selecting one from this set again. This format is repeated until a solution is found.

Once the program is able to solve puzzles the final step would be to allow the program to modify the puzzle with new constraints to eliminate alternative solutions or remove constraints to make the puzzle solvable. In the second Haraguchi paper mentioned, futoshiki designs are tested using varying numbers of inequalities to dictate the quality and difficult of the puzzle. Reducing the number of inequalities almost always makes the puzzle more difficult as it offers far more pathways for the solver to take where the designs with more inequalities offer a more structured pattern for the solver to follow. Where the research details how the quality of the puzzle deteriorates at the number of inequalities strays away from 100, a computer is always going to prefer easy to solve puzzles where a human may prefer a challenging puzzle. The research also was unable to determine the optimal location for them placing them randomly.

D. Purpose

The J-POP study was co-piloted by my supervisor and my work is largely an extension of this in a way as it will be heavily based off of the work already done. As already mentioned futoshiki puzzles larger than 9x9 do not currently exist and no research into the development of them has seemingly been explored and filling this gap would be the overall goal of my project.

Solutions to these larger futoshiki puzzles by the J-POP framework would also give further insight into the futoshiki problem space.

2. Proposed Work

The program will be designed to be able to generate futoshiki puzzles of a size specified by the user that are able to exceed 9x9. In generating the puzzles, the most difficult part is ensuring that the proposed puzzle only has one solution.

To ensure the solution is unique the program must also be able to solve the puzzle, to do this a backtracing solver will be developed based on the J-POP framework. This will create a tree of all possible solving orders to the puzzle (Crawford, Aranda, Castro and Monfroy, 2008) with the leaf nodes representing two different results, complete or incomplete. For the Futoshiki puzzle to be a perfect puzzle, the tree must only have one complete leaf node and all of the rest of the leaf nodes incomplete.

An example of this in action can be applied to the solving of the J-POP framework. The initial puzzle acts as the root of the tree, and the first value input acts as the first node K, from this another number is attempted to be input creating another node from K, K_1. If K_1 is deemed invalid, K_1 is deemed an incomplete leaf node, the program then returns to K to create K_2. This process will repeat until a full tree is made.

As soon as the solver finds a second complete leaf node in the tree, the tree is deemed imperfect. If the tree is imperfect, extra constraints will be added to the puzzle to remove potential solutions and the original tree will be reconsidered to account for the new constraints. The program will continue to add constraints until the tree is deemed perfect and only has one complete tree node.

Positioning of the constrains is the main difficulty and area of research required for the project as this is the only way to control the number of solutions available, this will dictate the success of the project.

3. Aims of Project

The aims of this project are to:

- To develop a backtracing solver with the ability to construct a tree of solutions to a given futoshiki puzzle.
- To study how the manipulation of constraints affect the completion of futoshiki puzzles.
- To generate large futoshiki puzzles with unique solutions.

4. Objectives

The objectives to judge the success of my project are:

- Investigate available literature pertaining to futoshiki puzzles.
- Investigate literature for solutions within other Latin puzzles that may have transferable ideas.
- Design a pseudocode solution.
- Design a UML class diagram.
- Develop decision trees for the completion of puzzles.
- Develop a system to add inequalities based on the instance’s decision tree.
- Develop a system to adjust starting values based on the instance’s decision tree.
- Ensure decision tree is updated based on the new constraints and not fully remade.
- The program can export generated instances in a readable format.
- The program does not break memory requirements.
- The program has been optimised to ensure it has minimal runtime.
- The program has been appropriately tested.
- An evaluation of the success of the program.
- Documentation of discoveries through development.
- Completion the final report.

5. Skills

A. Existing Skills

For the most part, my project is almost entirely based on my ability to program in Java. This is an area in which I am confident in as I have had to use it several times throughout my academic career with much success in modules such as Program Design and Development and Software Engineering Practice.

I will be building off of an existing java prototype and this is also something I am familiar with, during my Program Design and Development module, we were given an existing program to optimise and adjust to fit the given criteria. The experience in adjusting software to fit will help me to make the supplied code fit my purpose and the experience in optimisation will help because of the potentially heavy memory costs of the program when unoptimised.

B. New Skills

Many of the research papers I have studied have heavy use of set theory, while I have used this in past, my experience and knowledge of it is very minimal. Developing my knowledge of set theory will help me to better understand the research around my project and will help me to implement the ideas being presented in these papers.

6. Bibliography

Chlond, M., 2013. Puzzle—Latin Square Puzzles. INFORMS Transactions on Education, 13(2), pp.126-128.

Colton, S., 2002. Automated puzzle generation. Proceedings of the AISB'02 Symposium on AI and Creativity in the Arts and Science,.

Crawford, B., Aranda, M., Castro, C. and Monfroy, E., 2008. Using Constraint Programming to solve Sudoku Puzzles. 2008 Third International Conference on Convergence and Hybrid Information Technology,.

D. Demaine, E., L. Demaine, M. and Rodgers, T., 2008. A Lifetime of Puzzles. 1st ed. Boca Raton: Taylor & Francis Group, p.59.

Haraguchi, K. and Ono, H., 2015. How Simple Algorithms Can Solve Latin Square Completion-Type Puzzles Approximately. Journal of Information Processing, 23(3), pp.276-283.

Haraguchi, K., 2013. The Number of Inequality Signs in the Design of Futoshiki Puzzle. Journal of Information Processing, 21(1), pp.26-32.

Lloyd, H., Crossley, M., Sinclair, M. and Amos, M., 2021. J-POP: Japanese Puzzles as Optimization Problems. IEEE Transactions on Games, pp.1-1.

Rao, T. and Mitra, S., 2008. Synergizing AI and OOSE: Enhancing Interest in Computer Science through Game-Playing and Puzzle-Solving. AAAI Spring Symposium: Using AI to Motivate Greater Participation in Computer Science. 2008.

Dictionary.com. 2021. Daily Crossword. [online] Available at: <https://www.dictionary.com/e/crossword/>

En.wikipedia.org. 2021. Mathematical puzzle - Wikipedia. [online] Available at: <https://en.wikipedia.org/wiki/Mathematical_puzzle>

En.wikipedia.org. 2021. Rubik's Cube - Wikipedia. [online] Available at: <https://en.wikipedia.org/wiki/Rubik%27s_Cube>

En.wikipedia.org. 2021. Soma cube - Wikipedia. [online] Available at: <https://en.wikipedia.org/wiki/Soma_cube>

Futoshiki.org. 2021. Futoshiki Online Puzzles. [online] Available at: <https://www.futoshiki.org/>

Guinness World Records. 2021. Largest multi-sudoku puzzle. [online] Available at: <https://www.guinnessworldrecords.com/world-records/386582-largest-multi-sudoku#:~:text=The%20largest%20multmi%2Dsudoku%20puzzle,called%20MATH%20POWER%20since%202016.>

Sudoku.com. 2021. Sudoku Online Puzzles. [online] Available at: <https://sudoku.com/>.

Vexuspuzzle.com. 2021. Lexica Puzzles. [online] Available at: <http://www.vexuspuzzle.com/lexica.html>

7. Resources

My project is very light on resources, it will only require, my personal computer, Eclipse a free open-source IDE and code supplied by my supervisor.

8. Structure and Contents of Project Report

A. Report Structure

My report will be structured with the sections that follow:

- Title Page
- Authorship Declaration
- Acknowledgements
- Abstract
- List of Contents
- Introduction
- Research and Planning
- Design, Implementation and Testing
- Evaluation and Conclusions
- References
- Appendices
- Terms of Reference
- Other appendices.

B. List of Appendices

The following documents will be included in my project:

- Source Coding
- Test Plans
- Results

9. Assessment Criteria

My project will be assessed based on fitness for purpose (FFP) and build quality (BQ) the areas to be assessed are.

- Unique futoshiki generator.
    - FFP – Is able to generate larger than 9x9 unique futoshiki puzzles.
    - FFP – Produces instances in a standard format (ie the same format as J-POP).
    - BQ – Does not have excessive runtime.

    - BQ – Does not break memory limits.
- Testing plans.
    - FFP – Plans are followed throughout, and results are used to improve the product.
    - BQ – Plans are made in an easy-to-read format.
- Design plans.
    - FFP – UML format is used to preserve standard development procedures.
    - BQ – UML is used to a high standard.

10. Project Plan

A Gantt chart of the project plan schedule can be found below:

	Week 1	Week 2	Week 3	Week 4	Week 5
	29/10/2021	06/11/2021	14/11/2021	22/11/2021	30/11/2021
Literature study	🟥 36 Hours	🟥	🟥		
Program design planning			🟪 12 Hours		
Generating a generic none-unique instance.				🟨 24 Hours	🟨
Producing solution trees from instances.					🟧 12 Hours

	Week 6	Week 7	Week 8	Week 9	Week 10
	08/12/2021	16/12/2021	24/12/2021	01/01/2022	09/01/2022
Producing solution trees from instances.	🟧 12 Hours				
Development of backtracing solver.	🟩 45 Hours	🟩	🟩		
Apply backtracing solver to generic instance to add constraints.				🟩 15 Hours	

	Week 11	Week 12	Week 13	Week 14	Week 15
	17/01/2022	25/01/2022	02/02/2022	10/02/2022	18/02/2022
Apply backtracing solver to generic instance to add constraints.		🟩 45 Hours	🟩		
Generate unique puzzle.			🟦 51 Hours	🟦	🟦
Prototype testing.					🟥 6 Hours

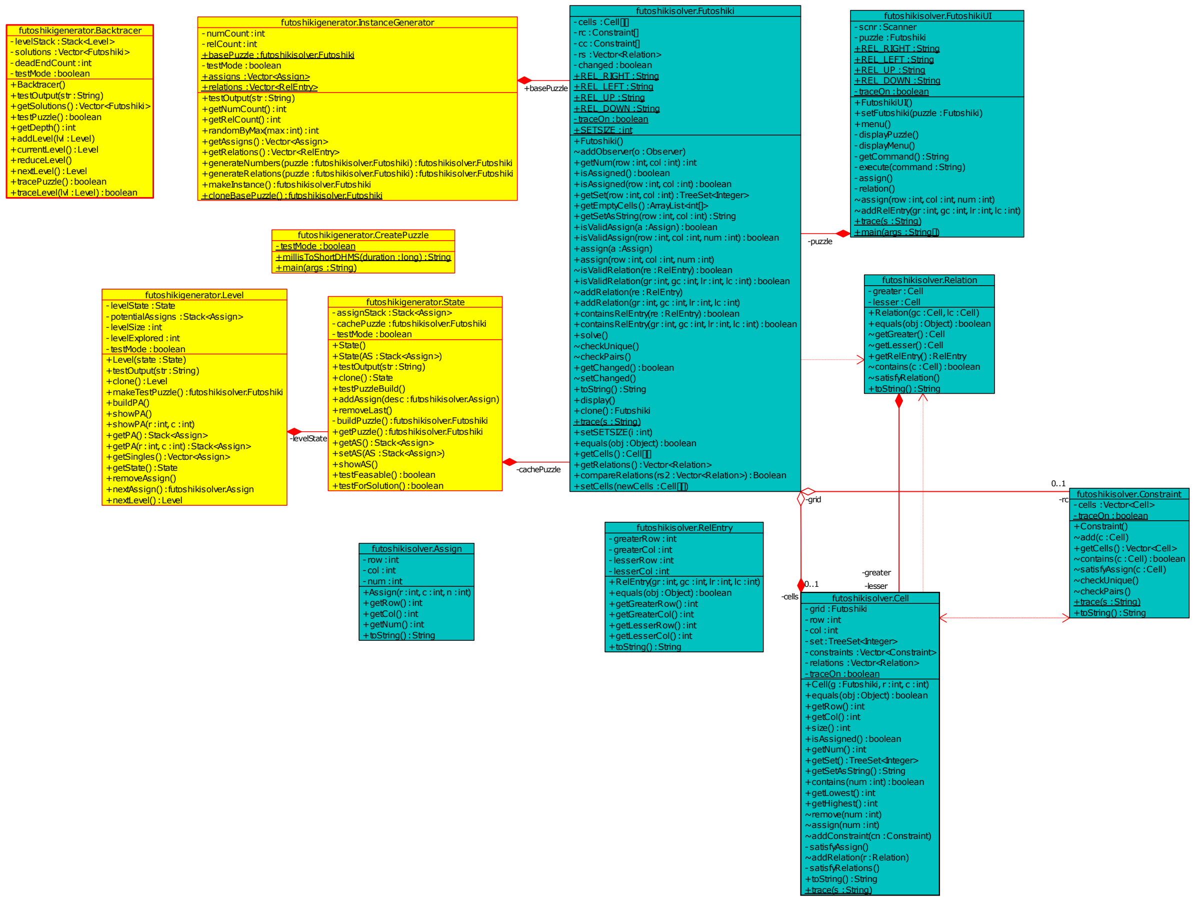
	Week 16	Week 17	Week 18	Week 19	Week 20
	26/02/2022	06/03/2022	14/03/2022	22/03/2022	30/03/2022
Improve efficiency of prototype.	18 Hours				
Further testing for final build.			6 Hours		
Writeup of final results.				45 Hours	

	Week 21	Week 22	Week 23
	07/04/2022	15/04/2022	23/04/2022
Final edits of project prior to submission.	12 Hours		

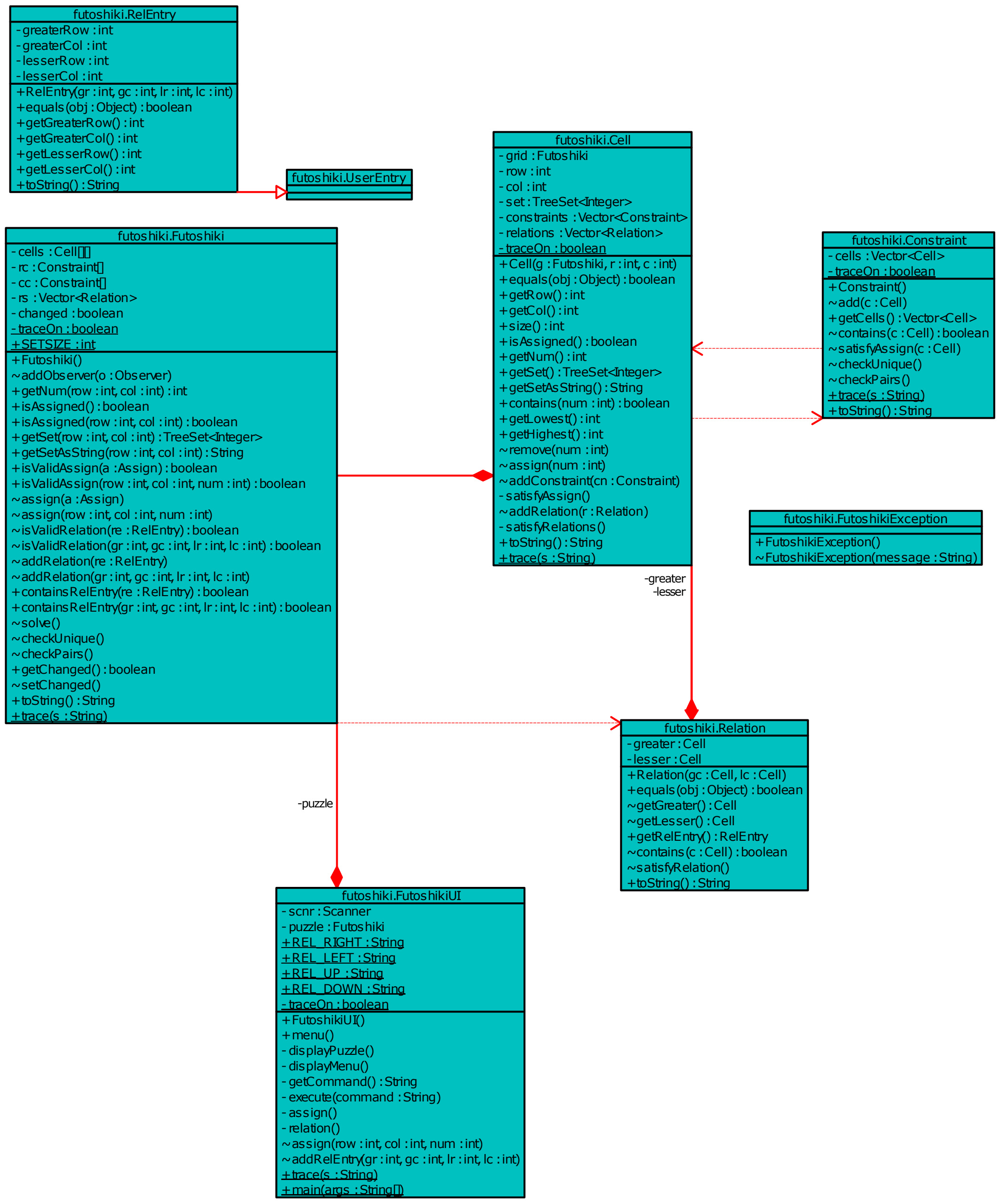
13.2 Appendix 2 - MoSCoW Requirements List

No.	Requirement	Priority	Type
1	A puzzle generator that is able to generate a puzzle larger than 9 x 9 with a single solution.	Must have.	Functional
2	A backtracking solver that is able to count the number of solutions to a Futoshiki puzzle.	Must have.	Functional
3	The backtracker can create a tree of the search order when searching for solutions.	Must have.	Functional
4	The backtracker takes advantage of the supplied solver code.	Must have.	Functional
5	The generator can add constraints optimally.	Must have.	Functional
6	The system is built within Java.	Must have.	Functional
7	The generator can be run within a single executable.	Must have.	Functional
8	Generated puzzles are returned to the user in an easily readable format.	Must have.	Non-Functional
9	Optimised memory usage.	Must have.	Non-Functional
10	Be able to output the puzzle.	Should have.	Functional
11	Can be adjusted to be able to work with several different puzzle sizes.	Should have.	Functional
12	Well commented/documented code.	Should have.	Non-Functional
13	Provide fast puzzle generation.	Should have.	Non-Functional
14	Integration with an external framework (ie. J-POP) for final puzzle testing.	Could have.	Functional
15	Preventions in place to reduce likelihood of duplicate puzzles being created.	Could have.	Functional
16	The ability to rate puzzles based on the solution search tree.	Could have.	Functional
17	The output is formatted in a standard format (ie. Janko)	Could Have	Non-Functional
18	The solution search tree will not be dynamically updated based on updates from in the puzzle.	Won't Have.	Functional
19	A GUI for easier user interaction	Won't Have.	Functional

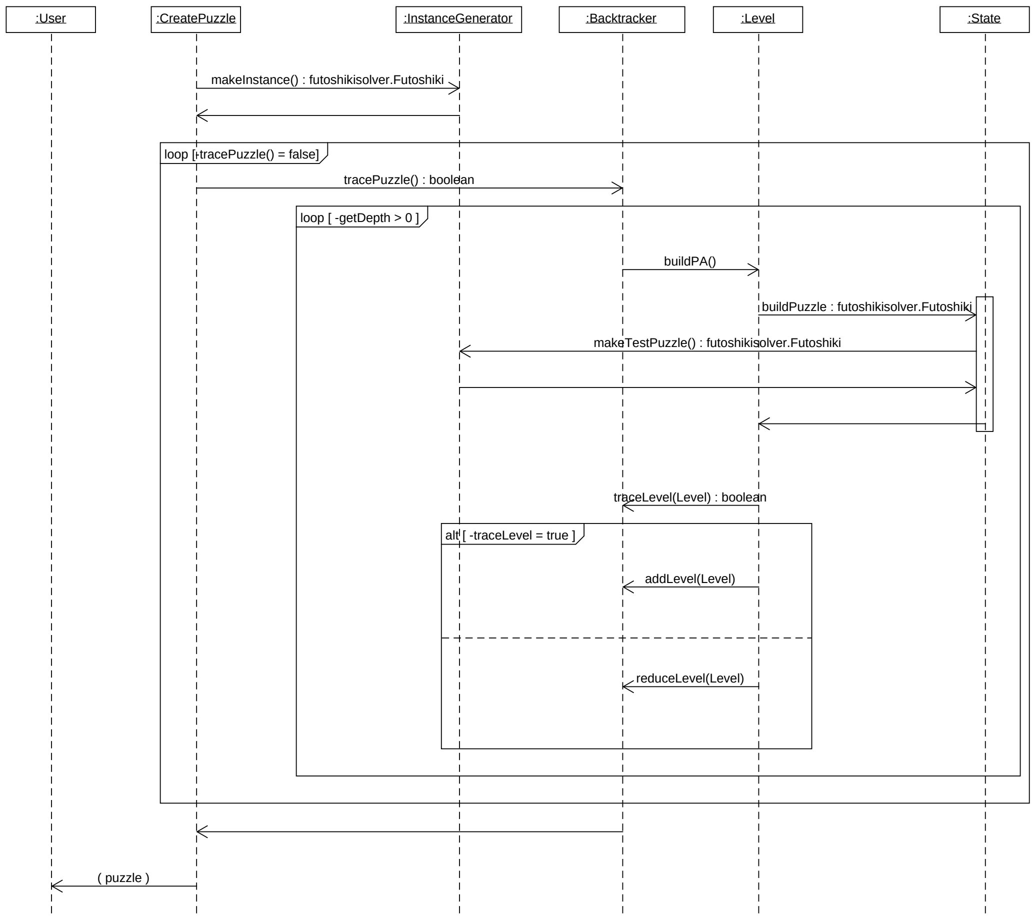
13.3 Appendix 3 - Full Class Diagram


futoshikigenerator.Backtracer
- levelStack : Stack<Level>
- solutions : Vector<Futoshiki>
- deadEndCount : int
- testMode : boolean
+Backtracer()
+testOutput(str : String)
+getSolutions() : Vector<Futoshiki>
+testPuzzle() : boolean
+getDepth() : int
+addLevel(lvl : Level)
+currentLevel() : Level
+reduceLevel()
+nextLevel() : Level
+tracePuzzle() : boolean
+traceLevel(lvl : Level) : boolean
futoshikigenerator.InstanceGenerator
- numCount : int
- relCount : int
+basePuzzle : futoshikisolver.Futoshiki
- testMode : boolean
+assigns : Vector<Assign>
+relations : Vector<RelEntry>
+testOutput(str : String)
+getNumCount() : int
+getRelCount() : int
+randomByMax(max : int) : int
+getAssigns() : Vector<Assign>
+getRelations() : Vector<RelEntry>
+generateNumbers(puzzle : futoshikisolver.Futoshiki) : futoshikisolver.Futoshiki
+generateRelations(puzzle : futoshikisolver.Futoshiki) : futoshikisolver.Futoshiki
+makeInstance() : futoshikisolver.Futoshiki
+cloneBasePuzzle() : futoshikisolver.Futoshiki
+basePuzzle
futoshikisolver.Futoshiki
- cells : Cell[][]
- rc : Constraint[]
- cc : Constraint[]
- rs : Vector<Relation>
- changed : boolean
+REL_RIGHT : String
+REL_LEFT : String
+REL_UP : String
+REL_DOWN : String
- traceOn : boolean
+SETSIZE : int
+Futoshiki()
~addObserver(o : Observer)
+getNum(row : int, col : int) : int
+isAssigned() : boolean
+isAssigned(row : int, col : int) : boolean
+getSet(row : int, col : int) : TreeSet<Integer>
+getEmptyCells() : ArrayList<int[]>
+getSetAsString(row : int, col : int) : String
+isValidAssign(a : Assign) : boolean
+isValidAssign(row : int, col : int, num : int) : boolean
+assign(a : Assign)
+assign(row : int, col : int, num : int)
~isValidRelation(re : RelEntry) : boolean
+isValidRelation(gr : int, gc : int, lr : int, lc : int) : boolean
~addRelation(re : RelEntry)
+addRelation(gr : int, gc : int, lr : int, lc : int)
+containsRelEntry(re : RelEntry) : boolean
+containsRelEntry(gr : int, gc : int, lr : int, lc : int) : boolean
+solve()
~checkUnique()
~checkPairs()
+getChanged() : boolean
~setChanged()
+toString() : String
+display()
+clone() : Futoshiki
+trace(s : String)
+setSETSIZE(i : int)
+equals(obj : Object) : boolean
+getCells() : Cell[][]
+getRelations() : Vector<Relation>
+compareRelations(rs2 : Vector<Relation>) : Boolean
+setCells(newCells : Cell[][])
futoshikisolver.FutoshikiUI
- scnr : Scanner
- puzzle : Futoshiki
+REL_RIGHT : String
+REL_LEFT : String
+REL_UP : String
+REL_DOWN : String
- traceOn : boolean
+FutoshikiUI()
+setFutoshiki(puzzle : Futoshiki)
+menu()
- displayPuzzle()
- displayMenu()
- getCommand() : String
- execute(command : String)
- assign()
- relation()
~assign(row : int, col : int, num : int)
~addRelEntry(gr : int, gc : int, lr : int, lc : int)
+trace(s : String)
+main(args : String[])
-puzzle
futoshikigenerator.CreatePuzzle
- testMode : boolean
+millisToShortDHMS(duration : long) : String
+main(args : String)
futoshikisolver.Relation
- greater : Cell
- lesser : Cell
+Relation(gc : Cell, lc : Cell)
+equals(obj : Object) : boolean
~getGreater() : Cell
~getLesser() : Cell
+getRelEntry() : RelEntry
~contains(c : Cell) : boolean
~satisfyRelation()
+toString() : String
futoshikigenerator.Level
- levelState : State
- potentialAssigns : Stack<Assign>
- levelSize : int
- levelExplored : int
- testMode : boolean
+Level(state : State)
+testOutput(str : String)
+clone() : Level
+makeTestPuzzle() : futoshikisolver.Futoshiki
+buildPA()
+showPA()
+showPA(r : int, c : int)
+getPA() : Stack<Assign>
+getPA(r : int, c : int) : Stack<Assign>
+getSingles() : Vector<Assign>
+getState() : State
+removeAssign()
+nextAssign() : futoshikisolver.Assign
+nextLevel() : Level
-levelState
futoshikigenerator.State
- assignStack : Stack<Assign>
- cachePuzzle : futoshikisolver.Futoshiki
- testMode : boolean
+State()
+State(AS : Stack<Assign>)
+testOutput(str : String)
+clone() : State
+testPuzzleBuild()
+addAssign(desc : futoshikisolver.Assign)
+removeLast()
- buildPuzzle() : futoshikisolver.Futoshiki
+getPuzzle() : futoshikisolver.Futoshiki
+getAS() : Stack<Assign>
+setAS(AS : Stack<Assign>)
+showAS()
+testFeasable() : boolean
+testForSolution() : boolean
-cachePuzzle
0..1
-grid
-rc
futoshikisolver.Constraint
- cells : Vector<Cell>
- traceOn : boolean
+Constraint()
~add(c : Cell)
+getCells() : Vector<Cell>
~contains(c : Cell) : boolean
~satisfyAssign(c : Cell)
~checkUnique()
~checkPairs()
+trace(s : String)
+toString() : String
futoshikisolver.Assign
- row : int
- col : int
- num : int
+Assign(r : int, c : int, n : int)
+getRow() : int
+getCol() : int
+getNum() : int
+toString() : String
futoshikisolver.RelEntry
- greaterRow : int
- greaterCol : int
- lesserRow : int
- lesserCol : int
+RelEntry(gr : int, gc : int, lr : int, lc : int)
+equals(obj : Object) : boolean
+getGreaterRow() : int
+getGreaterCol() : int
+getLesserRow() : int
+getLesserCol() : int
+toString() : String
-greater
-lesser
0..1
-cells
futoshikisolver.Cell
- grid : Futoshiki
- row : int
- col : int
- set : TreeSet<Integer>
- constraints : Vector<Constraint>
- relations : Vector<Relation>
- traceOn : boolean
+Cell(g : Futoshiki, r : int, c : int)
+equals(obj : Object) : boolean
+getRow() : int
+getCol() : int
+size() : int
+isAssigned() : boolean
+getNum() : int
+getSet() : TreeSet<Integer>
+getSetAsString() : String
+contains(num : int) : boolean
+getLowest() : int
+getHighest() : int
~remove(num : int)
~assign(num : int)
~addConstraint(cn : Constraint)
- satisfyAssign()
~addRelation(r : Relation)
- satisfyRelations()
+toString() : String
+trace(s : String)

13.4 Appendix 4 - Supplied Code Class Diagram



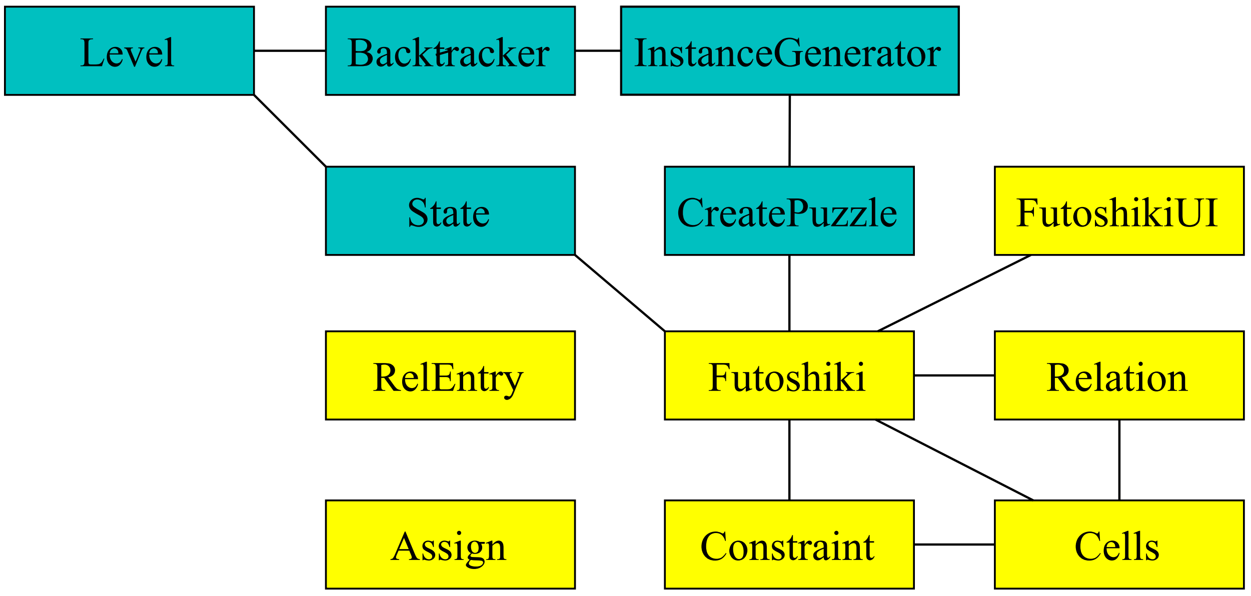
13.5 Appendix 5 - Sequence Diagram


:User
:CreatePuzzle
:InstanceGenerator
:Backtracker
:Level
:State
makeInstance() : futoshikisolver.Futoshiki
loop [-tracePuzzle() = false]
tracePuzzle() : boolean
loop [-getDepth > 0]
buildPA()
buildPuzzle : futoshikisolver.Futoshiki
makeTestPuzzle() : futoshikisolver.Futoshiki
traceLevel(Level) : boolean
alt [-traceLevel = true]
addLevel(Level)
reduceLevel(Level)
(puzzle)

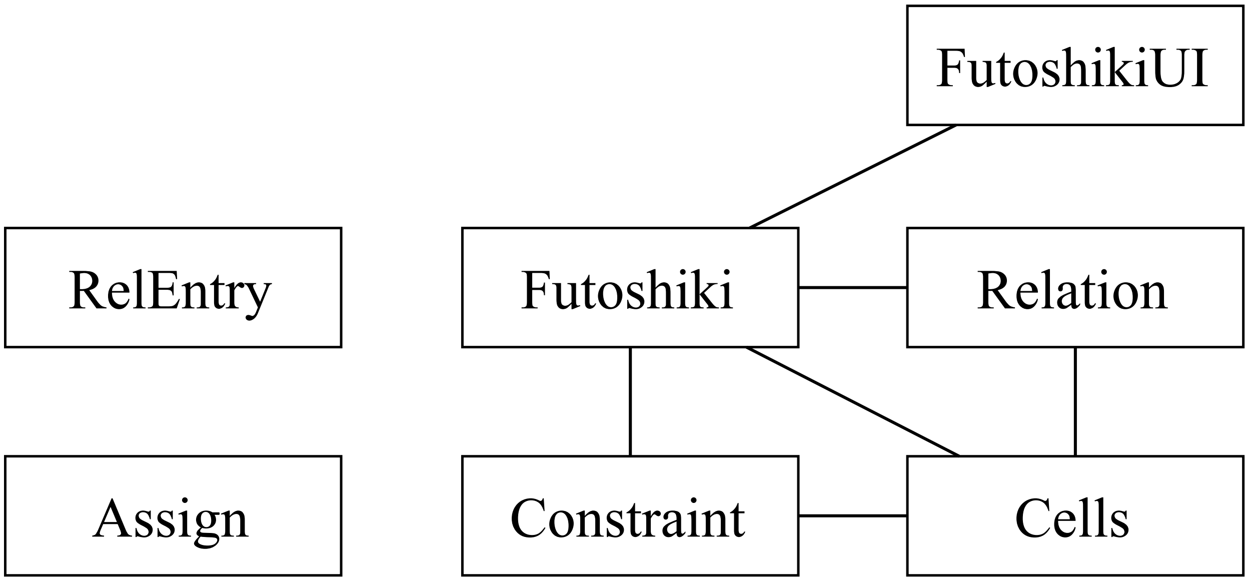
13.6 Appendix 6 - Unique Puzzle Generation Test

Unique Futoshiki Generation Time.											
Futoshiki Size (n*n)	Criteria	Test Number									
		1	2	3	4	5	6	7	8	9	10
4 (rel 12)	Runtime	<00:00:01	<00:00:01	<00:00:01	<00:00:01	<00:00:01	<00:00:01	<00:00:01	<00:00:01	<00:00:01	<00:00:01
	Generations	1	1	1	1	1	1	1	1	1	1
5 (rel 15)	Runtime	<00:00:01	<00:00:01	<00:00:01	<00:00:01	<00:00:01	<00:00:01	<00:00:01	00:00:15	<00:00:01	00:00:16
	Generations	1	1	1	1	1	1	1	3	1	4
6 (rel 20)	Runtime	00:01:45	00:02:29	00:01:15	00:01:15	00:01:03	00:00:15	00:04:01	00:05:18	00:02:31	00:02:46
	Generations	11	15	7	9	8	2	22	38	15	17
7 (rel 30)	Runtime	00:02:10	00:01:40	00:06:53	00:02:30	00:22:05	00:21:24	00:26:17	00:03:08	00:07:36	00:32:07
	Generations	8	7	27	9	80	75	93	12	26	146
8 (rel 45)	Runtime	00:30:07	00:56:40	02:59:26	00:41:36	00:40:39	00:40:11	08:11:35	08:32:26	01:14:03	02:57:04
	Generations	121	226	714	167	164	159	1963	2044	299	709
9 (rel 65)	Runtime	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed
	Generations										
10 (rel 90)	Runtime	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed	Failed
	Generations										

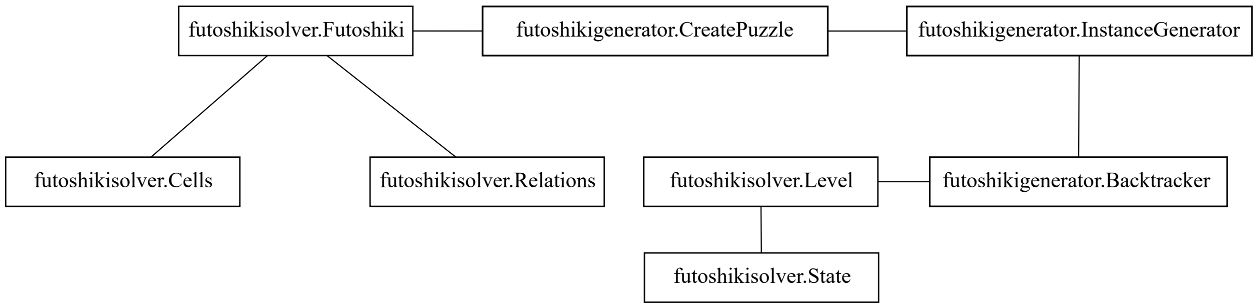
13.7 Appendix 7 - Full Code High Level Class Diagram



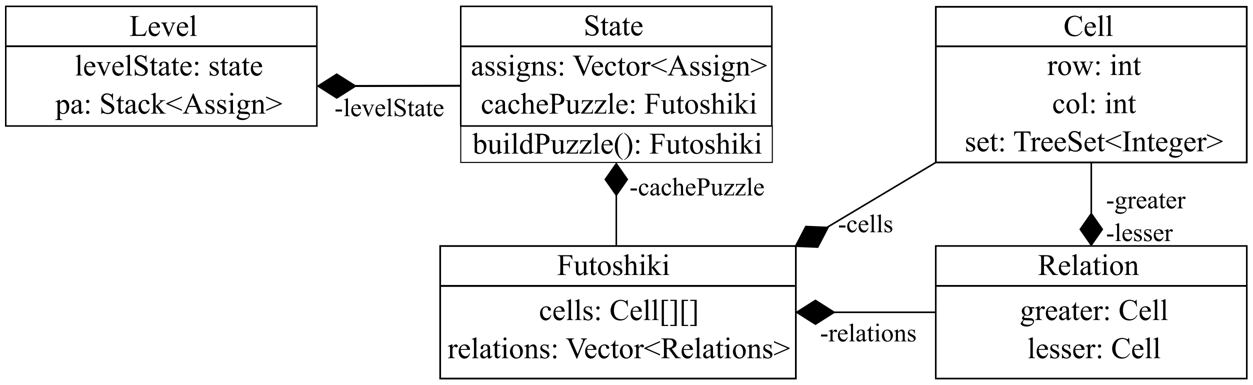
13.8 Appendix 8 - Supplied Code High Level Class Diagram



13.9 Appendix 9 - High Level Class Diagram


futoshikisolver.Futoshiki
futoshikigenerator.CreatePuzzle
futoshikigenerator.InstanceGenerator
futoshikisolver.Cells
futoshikisolver.Relations
futoshikisolver.Level
futoshikigenerator.Backtracker
futoshikisolver.State

13.10 Appendix 9 - Medium Level State and Level Class Diagram



13.11 Appendix 10 - Pseudocode Solution

Pseudocode Solution

```
while(finalPuzzle = empty)
    puzzle = new Futoshiki
    for(int i = 0; i < relationCount; i++)
        validRelations = All Valid Relations
        relation = validRelations.random
        puzzle.addRelation(relation)
    end for
    for(int i = 0; i < valueCount; i++)
        validValues = All Valid Values
        relation = validValues.random
        puzzle.addValue(values)
    end for
    startingPuzzleLevel = createLevel(puzzle)
    LevelStack ← startingPuzzleLevel
    while depth > 0 do
        if solutionCount > 1
            break
        end if
        if traceLevel = true then
            newLevel = currentLevel ← currentLevel.randomPA
            levelStack ← newLevel
        else
            levelStack = levelStack − currentLevel
        end if
    end while
    if solutionCount = 1
        puzzle = finalPuzzle
    end if
end while
return finalPuzzle
```

13.12 Appendix 11 - Unit Test Results

Function Name	Test Number		
	1	2	3
test_setNum()	✓	✓	✓
test_getSetAsString()	✓	✓	✓
test_getSet()	✓	✓	✓
test_isValidRelation()	✓	✓	✓
test_ValidAssign_value()	✓	✓	✓
test_ValidAssign_relation()	✓	✓	✓
test_puzzleEqual_true()	✓	✓	✓
test_puzzleEqual_false()	✓	✓	✓
test_cellEqual_true()	✓	✓	✓
test_cellEqual_false()	✓	✓	✓
test_relationsEqual_false()	✓	✓	✓
test_relationsEqual_true()	✓	✓	✓
test_clone_true()	✓	✓	✓
test_cloneBasePuzzle()	✓	✓	✓
test_testBasePuzzle()	✓	✓	✓
test_nextLevel()	✓	✓	✓

13.13 Appendix 12 - Generator Speed Test Results

Puzzle Size	Test (Seconds (4 d.p))									
	1	2	3	4	5	6	7	8	9	10
4	0.004	0.0007	0.0011	0.0015	0.0015	0.0008	0.0011	0.0013	0.0008	0.0008
5	0.004	0.0032	0.0046	0.0008	0.0032	0.0068	0.0062	0.0012	0.0049	0.0054
6	0.0098	0.0042	0.0018	0.0029	0.0038	0.0027	0.0042	0.0036	0.0018	0.002
7	0.0086	0.0249	0.0101	0.008	0.0075	0.0049	0.0083	0.0052	0.0068	0.0065
8	0.0453	0.0478	0.0133	0.0402	0.0155	0.0298	0.0286	0.0427	0.0145	0.0257
9	0.0922	0.0588	0.0728	0.0445	2.0119	0.077	0.0637	0.0773	0.0768	2.0171
10	1.0497	5.0761	18.0861	9.045	2.0657	6.0006	0.063	8.0673	1.0093	5.0569

13.14 Appendix 13 - Ethical Approval

Full Name	Connor Matthew Campbell
Northumbria Email Address	connor.m.campbell@northumbria.ac.uk
Faculty	Engineering and Environment
Department	CIS
Submitting as	Undergraduate student
Has this project received ethical approval from an external organisation?	No
Module Code (UGT/PGT only)	KV6003
Module Tutor UGT/PGT only)	Xiaomin Chen
Research Supervisor	Mark Sinclair
Named Submission Coordinator	mark.sinclair@northumbria.ac.uk
Ethical Risk Level	Low

Title of your research project	Generation of large Futoshiki puzzles with unique solutions
Outline General Aims and Research Objectives	Futoshiki is a Japanese pencil puzzle similar to Sudoku. However, rather than placing numbers uniquely in rows, columns and boxes, as in Sudoku, Futoshiki puzzles have unique numbers only in rows and columns, but also have 'relations'. The numbers in certain cells are marked as either less than or greater than those in an adjacent cell. While solving large Futoshiki puzzles is challenging in its own right, this project will seek to generate new large (i.e. more than 9x9) Futoshiki puzzles that have demonstrably unique solutions.
Please give a detailed description of your research activities	I will be developing an instance generator for puzzles larger than 9x9. This would involve: • Researching current information around my topic (puzzle generators/solvers, latin square puzzles). • Planning a design for my generator. • Creating a backtracing solver to evaluate instances. • Generating unique instances of puzzles. • Testing the final generator. • A writeup of the final results. Because of this none of the data being collected is sensitive.
Anonymising Data (mandatory)	No research data would be collected as the project is purely software based. Any futoshiki instances downloaded will be under public domain.
Storage Details (mandatory)	No research data would be collected as the project is purely software based. The

	software developed will be stored locally or in a GitHub repository. Adequate backups will be made and stored.
Retention and Disposal (mandatory)	I confirm that I will comply with the University’s data retention schedule and guidance.
Proposed Start Date	18/11/2021
Proposed End Date	31/07/2022
I confirm that I have read and understood the University's Health and Safety Policy.	✓
I confirm that I have read and understood the University’s requirements for the mandatory completion of risk assessments in advance of any activity involving potential physical risk.	✓
I can confirm that there are no physical risks associated with this project and so no risk assessments are required.	✓
I confirm my supervisor has reviewed the contents of this document	✓
I confirm I have assessed the ethical risk level of my work correctly and answered the above sections as fully and accurately as possible.	✓