**A report submitted in partial fulfilment**

**of the regulations governing the award of**

**the Degree of BSc. (Honours)**

**Computer Science with Artificial Intelligence**

**at the University of Northumbria at Newcastle**

**Project Report**

**Comparative Analysis of AI-Based Solvers for the NP-Complete Problem Kakuro**

***David William Jackson***

***2019 / 2020***

**General Computing Project**

***Authorship Declaration***

**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 15,097

(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: David William Jackson

# Acknowledgements

The author would first like to thank Martyn Amos for his continued support throughout this project, offering feedback and support when asked. The author would also like to thank my partner, for providing support whenever they could.

# Abstract

Increasing the knowledge we have of how we can benefit from new novel algorithmic techniques is paramount. This report details the building and testing of a virtual simulator and multiple solvers of the Kakuro pencil puzzle game. These puzzle games have been used to safely analyse new algorithms by providing a test-bed that can be adapted to other problems in the machine learning industry, as well as fine tune the techniques of previous attempts at novel algorithms for problem solving. The simulator detailed is accompanied by a random search algorithm and basic Genetic Algorithm that is used to test the simulator. Through testing of these solvers, it was discovered that the random solver, although able to find the solution to multiple instances of Kakuro, was not as efficient as the basic Genetic Algorithm. A further combination of both the randomised solver and the Genetic Algorithm possessed further improvements, showing that combining algorithms can lead to increased benefit.

# Table of Contents

[Acknowledgements 2](#_Toc41523982)

[Abstract 2](#_Toc41523983)

[Table of Contents 3](#_Toc41523984)

[Introduction 5](#_Toc41523985)

[Literature Review 6](#_Toc41523986)

[The rules of Kakuro 6](#_Toc41523987)

[A history of pencil puzzles 7](#_Toc41523988)

[Complexity of the Kakuro Puzzle 8](#_Toc41523989)

[P=NP 8](#_Toc41523990)

[Other Pencil Puzzles 10](#_Toc41523991)

[Nurikabe 10](#_Toc41523992)

[Zen Puzzle Garden 10](#_Toc41523993)

[Sudoku 11](#_Toc41523994)

[Hashiwokakero 12](#_Toc41523995)

[Machine Learning Methods used to solve Pencil Puzzles 12](#_Toc41523996)

[Genetic Algorithms 13](#_Toc41523997)

[Decision Trees 14](#_Toc41523998)

[Ant Colony Optimisation 14](#_Toc41523999)

[Harmony based metaheuristic algorithm 14](#_Toc41524000)

[Backtracking 15](#_Toc41524001)

[Random Search 16](#_Toc41524002)

[Synthesis 17](#_Toc41524003)

[Requirements 17](#_Toc41524004)

[The Simulator 19](#_Toc41524005)

[Designing the Simulator 20](#_Toc41524006)

[Implementing the Code 21](#_Toc41524007)

[The Text Board 22](#_Toc41524008)

[The kCell Class 23](#_Toc41524009)

[The Virtual Board 24](#_Toc41524010)

[Evaluating the fitness of a solution 24](#_Toc41524011)

[Implementing a Randomised Solver 25](#_Toc41524012)

[Implementing a Genetic Algorithm 25](#_Toc41524013)

[Testing the Simulator 27](#_Toc41524014)

[The First Kakuro Board 27](#_Toc41524015)

[Creating Further Boards 27](#_Toc41524016)

[Testing each Solver 28](#_Toc41524017)

[Results 29](#_Toc41524018)

[Random Solver 29](#_Toc41524019)

[Genetic Algorithm Solver 31](#_Toc41524020)

[Combining the solvers 32](#_Toc41524021)

[Comparing the Instance sizes 33](#_Toc41524022)

[Sorting by instance size 33](#_Toc41524023)

[All Instances 34](#_Toc41524024)

[Conclusion 34](#_Toc41524025)

[Evaluation 35](#_Toc41524026)

[Evaluation of the product 35](#_Toc41524027)

[Evaluation of the project process 37](#_Toc41524028)

[Conclusion 39](#_Toc41524029)

[Recommendations 39](#_Toc41524030)

[References 40](#_Toc41524031)

[Appendices 43](#_Toc41524032)

# Introduction

This report will show in detail the process and resolution of a project that aims to create a virtual simulator of the pencil puzzle game Kakuro. The simulator was originally intended to become part of a separate system, but constraints have transformed the aim to that of a standalone simulator. The simulator is able to use two separate algorithms (random-search and Genetic Algorithm), combined with a third aggregation of solvers in order to solve a given instance of the Kakuro puzzle.

The main objective was to ensure that the simulator can read in any valid Kakuro instance after the instance has been converted into a text version, which includes details about certain attributes of the puzzle, and the specific rules of the board in question. The simulator allows other solvers to input possible solutions, the simulator will analyse them and give a score based on how ‘fit’ the solution is, then send that score to the solvers. The solvers will then adjust their approach based on that score. The simulators and solvers were written in C++, which the author does not have much experience with, and compiled with the Ubuntu command line on a Linux operating system.

The randomised solver works by attempting to input as many random strings of the right length possible, using brute force to attempt to solve the problem. This approach resulted in an inefficient algorithm that took a long time to find the correct solution. The next solver, the Genetic Algorithm, attempted to improve a single string over time, finely adjusting the string each time its fitness was measured. This approach tended to work quite well compared to the randomised solver, especially at larger instances. Third, a combination of solvers were used by first employing the randomised solver to run as many random strings as possible to achieve a relatively low score then feeding that into the Genetic Algorithm. The Genetic Algorithm would then improve that string one number at a time, resulting in a better result than the algorithms separately.

This report will first detail an investigation into some of the most popular current Japanese Pencil Puzzles, then research into how some of these puzzles have been used to test novel algorithmic strategies, such as Ant Colony Optimisation, Genetic Algorithms and Simulated Annealing. These approaches, combined with these simple puzzle games, have been used to improve current academic knowledge on novel algorithms and how they can be used to benefit society.

An explanation of the design stage is then given, which details the way that the simulator and the solvers were designed, as well as an explanation of the text versions of the simulators, how they work, and how they are read in by the simulator. This is followed by a detailed description of the synthesis of each aspect of the simulator and solvers. This includes how each section of the simulator was created and how many of the problems were resolved. Furthermore, a full test of the simulators was undertaken and the results were recorded, analysed and explained. As mentioned, the randomised solver performed the worst and the combination of both solvers was by far the most efficient.

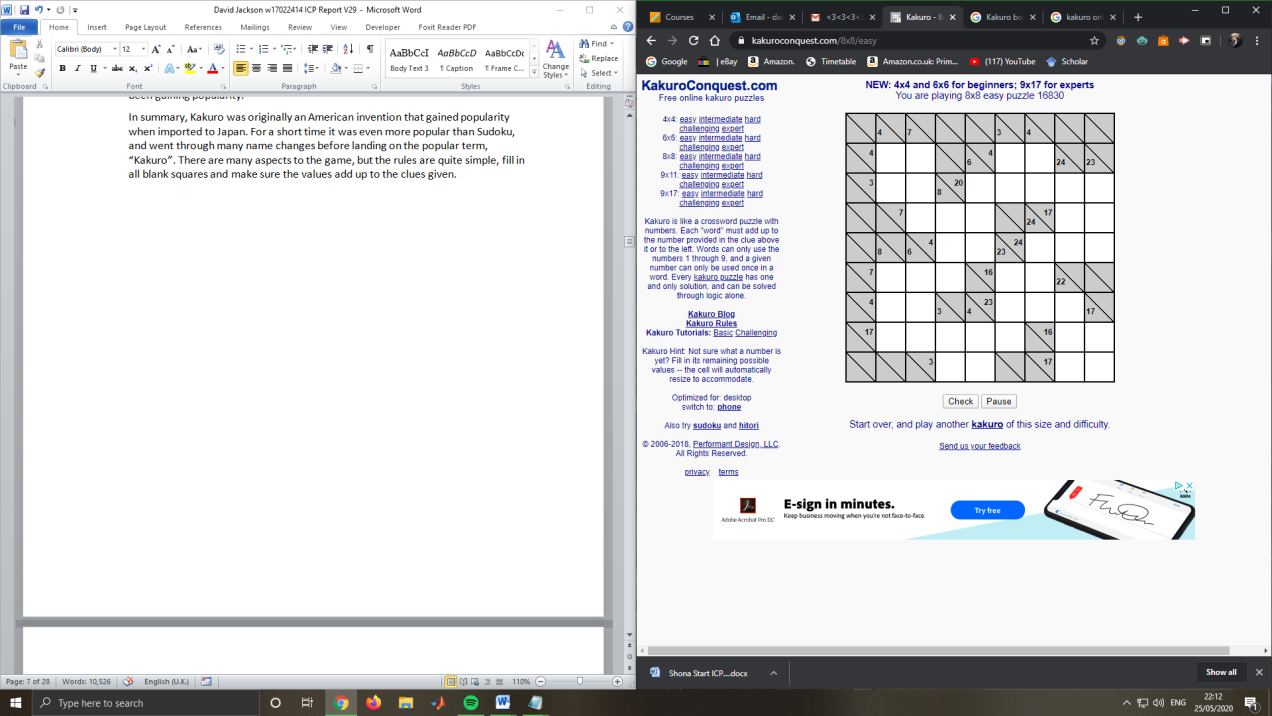
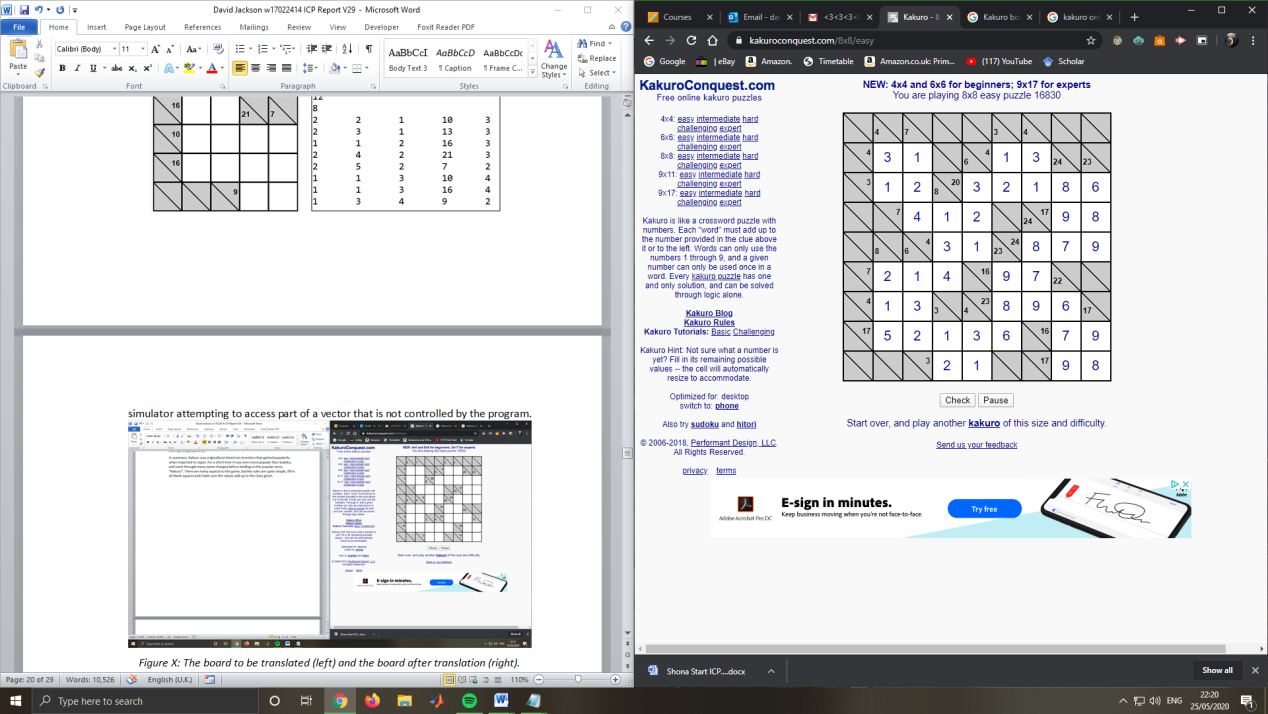
Finally, an evaluation of the end product is given, along with an evaluation of how the process turned out and then followed by a conclusion of the project.

# Literature Review

This chapter will be a discussion around Japanese Pencil Puzzles and how they have been used to improve machine learning techniques by providing a practical test bed for new novel algorithms. Pencil puzzles can be quite unique in the fact they always have a practicable solution, and the solution can be found in many ways using many techniques. The main focus of this paper is around the Japanese Pencil Puzzle Kakuro.

## The rules of Kakuro

There are a few variations on the rules for Kakuro, but the classic rule set comprises of a rectangular or square grid of cells, of which there are three types. The first of which is a blank square, which will be filed by the player. To complete the board, the player needs to choose a number between 1 and 9 for each blank square. Some squares are blocks, and don’t need changing or considering. These are just filler for the blank and clue squares. The third type of square is the most important, known as the clue squares. These squares have either 1 or 2 integers that are used to check the blank squares. Each integer, depending on its position, considers all the squares to its right or bottom, forming a group of blank squares. The group only stretches until the first block square it encounters. Each blank square in the group needs to add up to the integer either above or to the left of the group, depending on whether the group is vertical or horizontal. No number for a blank square can be repeated in the same group, so the difficulty on the board comes with finding the correct numbers that can satisfy all of the clue squares.

There tends to only be a single solution for any given commercial Kakuro Puzzle, although it is very possible for more than one solution to be found. Most commercial puzzles are designed in such a way that they will only ever have one solution, in order to increase the difficulty of the puzzle. While the difficulty of the puzzle is not the main goal of the makers of the board, the number of solutions is a primary factor in the popularity of the puzzle, as the puzzle is deemed too ‘simple’ if there is more than one solution.

*Figure 1: An Incomplete Kakuro grid (left) and a fully complete one (right)*

## A history of pencil puzzles

Kakuro is a puzzle game popularised in Japan that is primarily solved with pencil and paper. Like Sudoku, Kakuro was brought to Japan and gained popularity there, resulting in the widespread fame it has now. The puzzle proved very popular with crossword enthusiasts, and later Sudoku enthusiasts.

It can be traced back to the April/May 1950 issue of “Official Crossword Puzzles” by the New York based Dell Publishing Company. The company published the idea for the puzzle invented by Canadian building constructor Jacob E. Funk, under the name “Cross Sum”. Cross Sum wasn't utilised fully until the mid-1960s, where it appeared in every issue. The puzzle gained some popularity but it wasn’t until 1980, when the president of Nikoli Puzzles imported Cross Sum to Japan, that it gained a large following. Six years later, Maki Kaji, the then-president of Nikoli Puzzles, rebranded Cross Sum as “Kasan Kurosu”, or “Addition Cross” in English, later shortened to its official name, “Kakuro”. The company then developed dedicated Kasan Kurosu puzzle books. From 1986 to 1992, these books were actually higher ranked than dedicated Sudoku Books, also published by Nikoli Puzzles. Although this popularity did not last past 1992, they are still second in the ranking even today, with Sudoku being the most popular. The third most-popular is Slitherlink, which was also developed by Nikoli Puzzles.

Kakuro was not a worldwide success until 2005, when the Sudoku craze was still emerging from the previous year. This was during the time when American and British newspapers started including Sudoku and Kakuro in their puzzle sections, and they were very popular, starting with “The Guardian” (J. McCurry, “The new grid on the block”, The Guardian, September 14th 2005.), when they first started publishing Kakuro in their puzzle section. Kakuro and Sudoku have been some of the most popular puzzle games since then, and have only been gaining popularity.

In summary, Kakuro was originally an American invention that gained popularity when imported to Japan. For a short time it was even more popular than Sudoku, and went through many name changes before landing on the popular term, “Kakuro”. There are many aspects to the game, but the rules are quite simple, fill in all blank squares and make sure the values add up to the clues given.

## Complexity of the Kakuro Puzzle

Kakuro can be quite difficult; especially when the boards include higher integer values of clue squares. Higher integer values mean that there are more possibilities for the values in that section. For example, if a clue square has the number 6 and 2 cells to sum, then there are only two possible combinations of numbers. If the value was 9 instead, then there are 4 combinations. The higher the value of the clue square, then more combinations are possible. This doesn’t work all the way though, as 45 (the maximum value that a clue square can be) only has one combination; which is all of the numbers 1-9.

Grids can be as large as the board-maker wants, and some of the hardest solved Kakuro puzzles can be 50 cells wide or more. The difficulty is not limited to humans solving the puzzle; automatic solvers will also have difficulty as well. In the world of computer science, Kakuro and many others are known as an NP-Complete problem. (Ruepp, O. and Holzer, M., 2010, June).

### P=NP

The term NP-Complete stems from a problem in Computer Science known as the P=NP problem, where N stands for Non-Deterministic and P stands for Polynomial time (Srinivasan, N., 2011, June ). Complexities of computer algorithms can be defined as Polynomial, Exponential, Logarithmic, Linear or Constant. In this case, polynomial is the most important type to consider. Polynomial time is when a problem can be solved in a number of steps equal to O(nk), where ‘n’ is the complexity and size of the input and ‘k’ is a non-negative integer. Compared to Exponential complexity (O(2nk)), Polynomial complexity is noted as very fast, and useful for solving problems. If a problem can be reduced to Polynomial time, then solving that problem will likely be far easier, and possible to do within reason for most computers of this age.

The P=NP problem stems from one of the largest debates in Computer Science; the discussion about whether any problem that can be solved in Non-Deterministic time (NP) can also be solved in Deterministic Polynomial time (P). Deterministic and Non-Deterministic are aspects of what is known as a Turing machine. Named after Alan Turing, these machines are incredibly basic versions of computers that use an infinite amount of tape and a finite number of symbols that can be applied to that tape. Using a set of rules, Turing machines can theoretically use those symbols to compute exactly like any computer built in the past or future, using various rules to solve the problem that is being considered. A Non-Deterministic Turing machine can be presented with two rules at the same time, like a choice it can make to lead to an acceptable state. A Deterministic Turing machine on the other hand will only ever follow one rule at a time.

Continuing from this, a problem that is determined as P can be answered in polynomial time, meaning within a reasonable time for most computers. On the other hand, NP problems can have their answers verified in polynomial time, meaning that the answer given can be stated as either correct or incorrect.

Proving that any NP problem can be converted into a single P problem is a very prevalent question in Computer Science. If proven true, it could change vast sectors of Computer Science. While a very small part of the conversation, Kakuro has been determined and proven to be NP-Complete (Ruepp, O. and Holzer, M., 2010, June). If an NP-Complete problem is solved completely, that solution could theoretically be used to solve all NP problems in the future using the same algorithm in a different application, vastly reducing the time taken for many of these problems to be solved.

There have been many attempts to build an algorithmic automatic solver for puzzles such as Kakuro, with many interesting facets or aspects to consider. These automatic solvers may seem meaningless at first, but learning more about the relationship between machine learning techniques and puzzle games can lead to incredible discoveries, and problem solving techniques that can be transferred into the real world of machine learning.

To conclude, Kakuro has been shown to be NP-Complete, which means that it is considered both NP and NP-Hard. This class denotes problems that are able to be solved with a specific algorithm that could solve any other NP problem. The supposed algorithm will be able to verify that a solution given is suitable or not in polynomial time. The P versus NP debate considers that there might be a way to solve a problem in polynomial time. This is one of the most discussed and controversial debates in computer science industry.

## Other Pencil Puzzles

Kakuro is certainly not the only puzzle to be used to test new machine learning techniques. There are many other puzzles that can be used to test these new strategies. Following are a few examples of those types of puzzles, an explanation of how they work and evaluations of papers that demonstrate novel algorithms that have been tested using those puzzles.

### Nurikabe

Nurikabe was developed in 1991 and published that same year (Dsudoku 2013), once again by Nikoli Puzzles. In its most basic form, Nurikabe is a visualisation of an island archipelago. Each island consists of a single cell at the beginning, and the objective is to extend that cell by a number specified on the cell. The difficulty is in making sure that the island does not directly connect with any other island, also making sure that there are no spaces on the board where there is a two-by-two square or larger of open water. Those are the only constraints put forward by the board, and the challenge resides in making sure that each island is the right length and size, doesn’t touch any other island and making sure there are no two-by-two stretches of sea. This may seem simple, but many times, near the end of the board, the person solving runs into a problem where they need more space for the end of an island, and they can’t make any more space until they reformat many of the other islands.

Nurikabe has also been shown to be NP-Complete (Holzer, Et al. 2014), due to the fact that a solution can be verified quickly but not solved quickly. They came to this conclusion by converting the Nurikabe puzzle into a set of Boolean gates, such as NOT and OR gates. AND gates were not required, as they can be simplified to a combination of OR and NOT gates. Using this system, they were able to define that each example they used was NP-Compete, and could be verified quickly.

### Zen Puzzle Garden

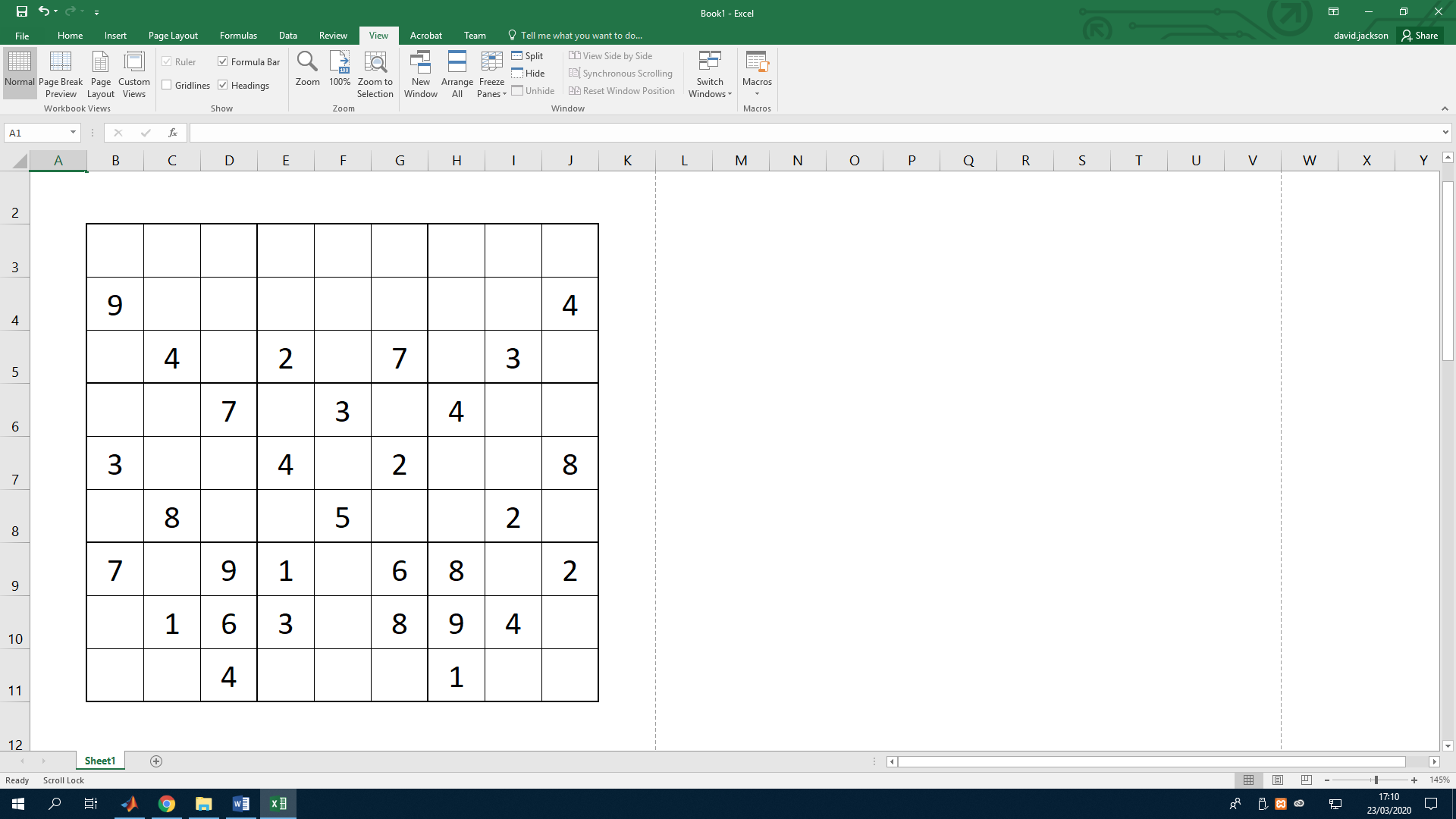
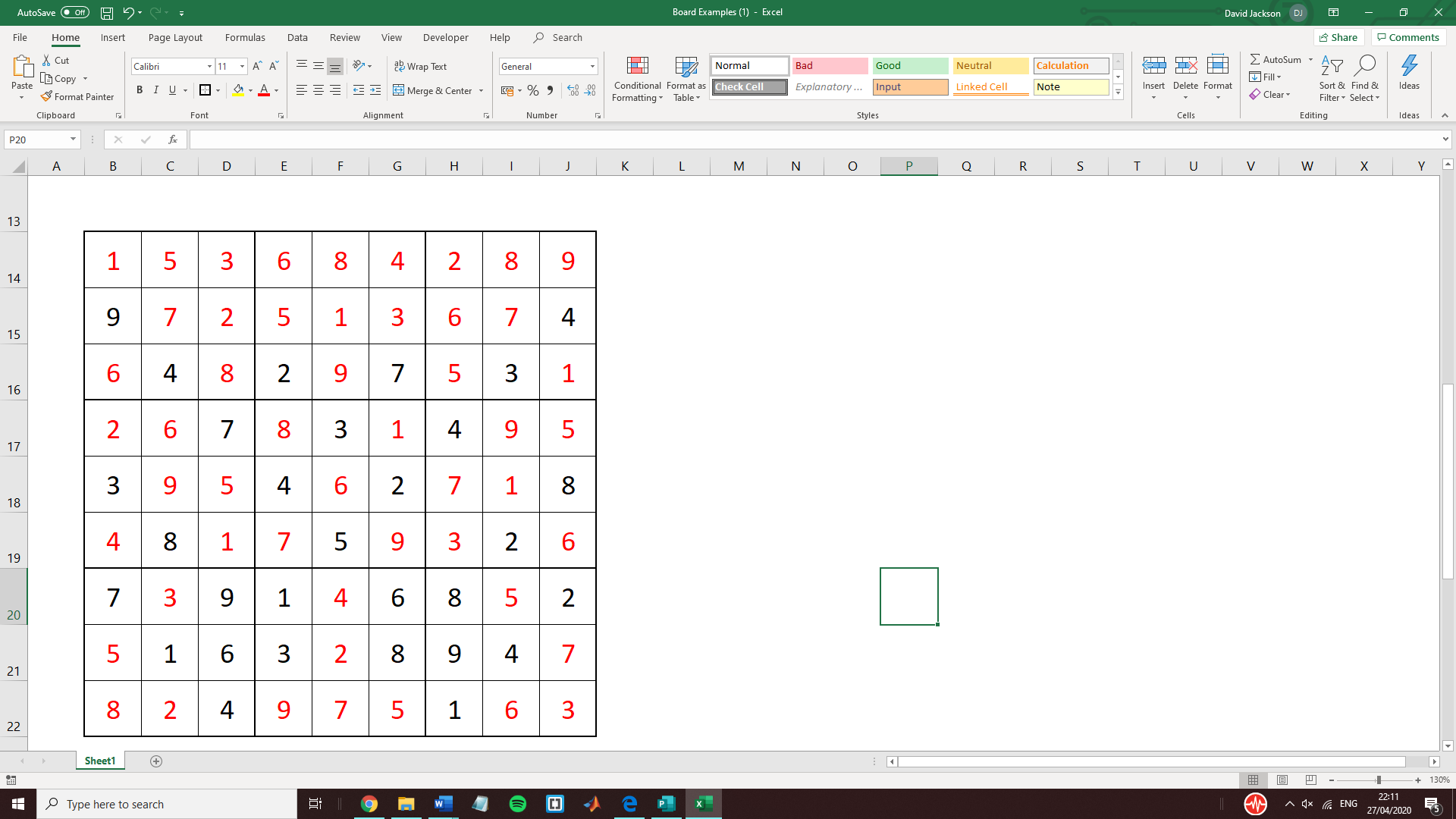
ZPG is quite an old game but can translate quite well to the digital screen. The game is based around a concept known as a “Zen Garden”. These Zen Gardens are actual structures very popular in Japan and many other Asian countries and are primarily sand based flat spaces that have basic objects decorating them, such as rocks, leaves, statues and sometimes even lanterns. The traditional way of maintaining these gardens is to use a wooden rake to till the sand and leave aesthetically pleasing trails in the sand, creating a still art form that is beautiful and calming to curators. The game uses this base structure and has the objective of tilling and raking each square of sand in geometric lines. The difficulty comes with the constraints set in place that force the player to think logically.

Many versions of the game come with objects or obstacles that the player has to either collect, avoid or move. These objects are usually leaves, rocks or statues. The leaves often have to be collected in a certain order, the rocks are impassable and cannot be passed through, and the statues have to be moved to certain places. Further constraints limit the movement of the player. The player can enter the garden from any direction where the path allows and must navigate through the garden and be able to exit onto the path. If the player wants to enter from the path at any given point, they must continue on a straight line until they either exit onto the path again, they hit upon a previous path or they hit an obstacle. When they stop, if they cannot move to their left or right, they lose the game and must start again. Once all collectable objects have been collected, and each sand tile has been raked, the board is solved.

ZPG has been researched thoroughly as to its complexity, and this paper (Houston, R. Et al. 2018) has been used to determine the NP-Completeness of the puzzle. The researchers discovered that ZPG is in fact NP-Complete by converting examples of boards to arbitrary cubic planar graphs. These graphs are originally used in graph theory to demonstrate a 2D plane. The purpose of the paper was to determine if examples of the games contained Hamiltonian Circuits, which are representations of a path around a cubic graph that visits each vertex in turn. If the examples did in fact contain a Hamiltonian Circuit, then they can be considered NP-Complete. They found that as well as the examples all containing Hamiltonian Circuits, the graphs could be computed in polynomial time and checked in a time of O(n log n).

### Sudoku

Sudoku is one of the most popular puzzle games in the world. The puzzle was originally an 18th century Swiss invention, called “Latin Squares”. It appeared in Japan in 1984 and given the name “Sudoku”. The most common form of Sudoku is a board consisting of nine cells by nine cells, split up into nine blocks of three by three cells, shown below. The players’ goal is to fit a single digit (1-9) into each unoccupied cell, with the constraint that each digit cannot be the same as any in its immediate x or y-axis, so any that is in the same row or column. This is not the only constraint however, as each of the blocks of 3-by-3 have to contain no duplicates either, meaning that inserting a digit into a block that has the same digit is not allowed, and will prevent you from completing the puzzle. These simple constraints have led to a very engaging puzzle, and its popularity could be linked to the simplicity of the game.



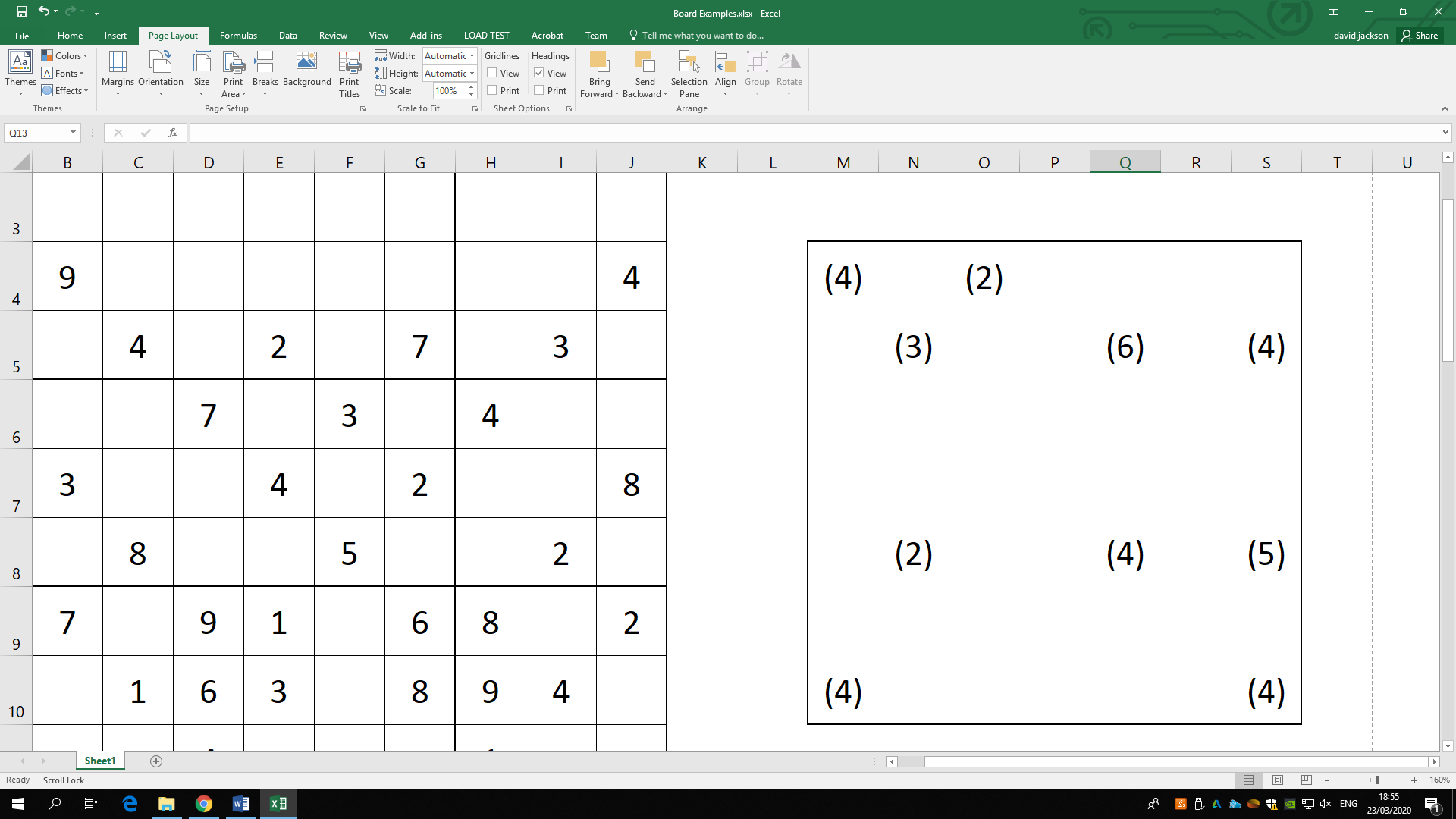
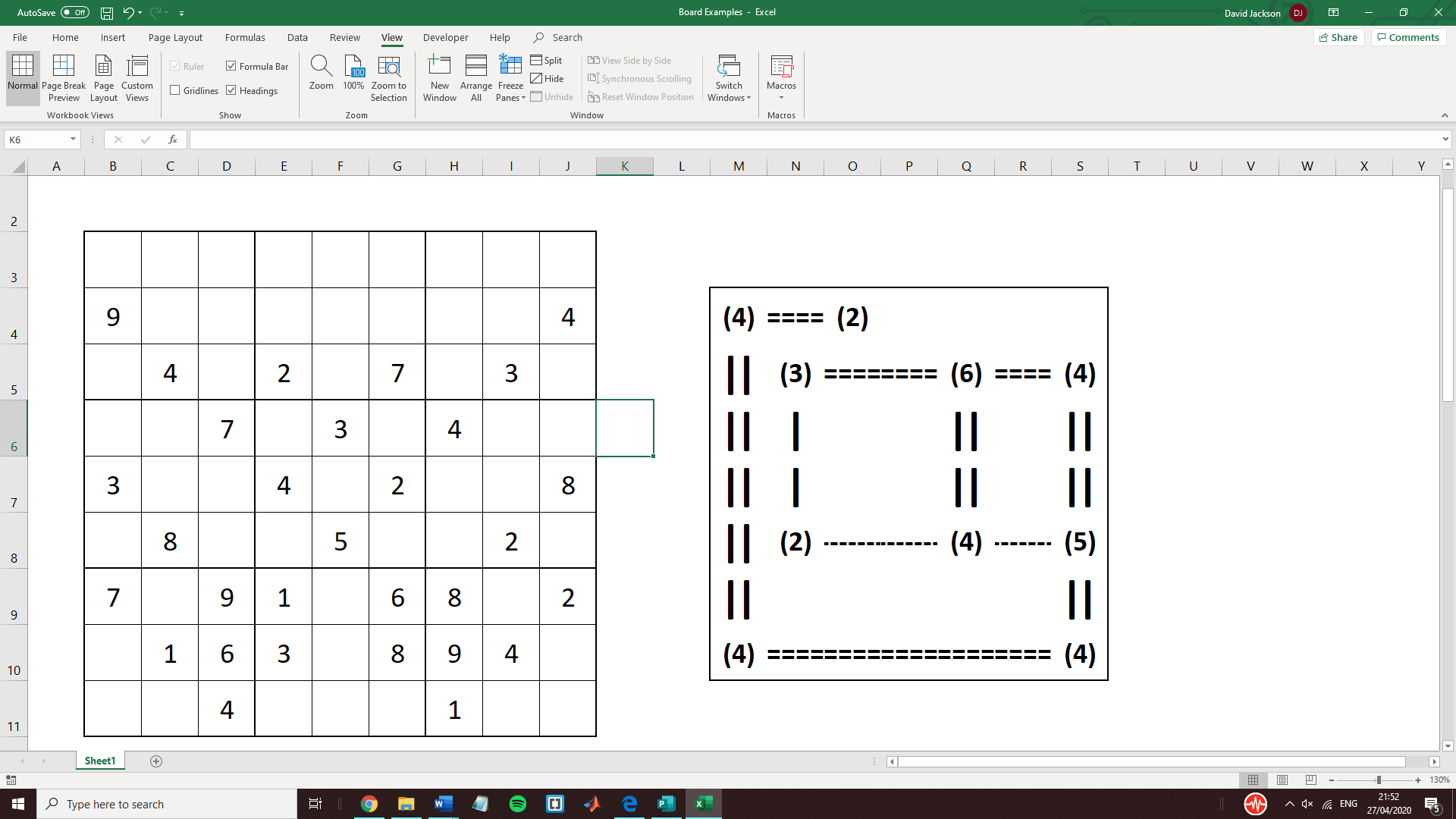
*Figure 2: Example of a base Sudoku board (Left) and the completed form (Right)*

Sudoku has been written about in many articles and publications, likely the most out of any in this report. This may partly be due to the popularity of the game, and partly due to the simplicity of the puzzle and how easy it is to convert to other forms. The main focus of this section of the report will be a paper published in 2019 that makes progress in solving Sudoku with Ant Colony Optimisation (Amos, M and Lloyd. H. 2019). The paper explored a new way to operate the ACO formula by using a new attribute called a “Best Value Operator”. This operator is used to prevent or lower the stagnation of solutions, allowing the algorithm to outperform some of the best algorithms that were in use, especially on the larger instances of Sudoku. Following the work of this paper, the idea was formed to enter this into a general AI-based system that can be used to solve many different types of systems without the need for context-based knowledge of the problem itself.

## Hashiwokakero

Hashiwokakero (or Hashi, as it is often called), is a puzzle based on a small grid with multiple “islands” that can connect to each other via “bridges”. Each island has a specific number of bridges it must have, and when each island has the correct number of bridges, the puzzle is solved. Each island can have up to two bridges connected to each nearby island. An island can only connect to another island if they are above, below, to the left or to the right of it. The puzzle was originally published in “Puzzle Communication Nikoli” in September of 1990, once again by Japanese puzzle book manufacturer Nikoli Puzzles.

The puzzle is not incredibly difficult for the most part, however, the main judge of mastery over the puzzle is the speed at which one can be completed, and some of the larger Hashi boards can take a considerable amount of time to complete. Every bridge built can affect every other connection on the board. If a mistake is made early on, the entire board might need to be redone when that mistake is discovered.

*Figure 3: An example of a base Hashiwokakero board (left), and completed (right)*

## Machine Learning Methods used to solve Pencil Puzzles

There are many different types and species of machine learning models that have been used to solve various puzzles throughout the years. There are two main types of machine learning algorithm, supervised and unsupervised. Supervised models are based on training data that is labelled, or given context. For example, a neural net can be given images that are labelled with what they represent, like “Car” or “Dog”. The neural net will then look for certain features that can be used to identify those objects in the future.

Unsupervised models on the other hand rely on training data that is not labelled. This is very often used in cluster analysis, which is a model that is used to find relationships between clusters of data that are not as obvious when looked at by single observers. Most of the algorithms shown below are either supervised, or semi-supervised. The following algorithms when they are used to solve puzzles tend to be supervised as there is a goal that they are working towards, the completion of a solution to the puzzle.

### Genetic Algorithms

Genetic Algorithms (GA) are based around the theory of ‘survival of the fittest’. One of the main components of the Genetic Algorithm is actually the Fitness Function – a value attributed to the suitability of the generation for solving the problem. Genetic Algorithms work by forming generations of candidates to solve the problem. The most suitable or ‘fit’ candidates share their properties with the next generation. Properties usually entail the values that change within the generation, for example, in a walking simulation (Inada, H and Ishii, K. 2004); the properties would be the amount of force to apply to certain parts of the legs or joints. This generally develops the solution over time, adapting to the training and with these puzzles, eventually finding the correct answer, or an answer as close to perfect as possible. Puzzles can have many solutions, but finding every solution possible is very difficult.

There are multiple phases that are associated with the Genetic Algorithm. They consist of the initial population of the solution, which is generally a random set of variables that will be shaped over time. The population is a number of solutions set by the user that will be configured for each generation. The solutions are then tested for their fitness to solve the problem. Their fitness is then measured, and the solutions that have the best fitness are then selected to provide features for the next generation. Those features are then stored and passed to the next generation. Each member of the next generation is then slightly mutated, so they don’t have all of the same features. The cycle will then continue. For many problems, there is no exact solution, such as walking simulations (Inada, H and Ishii, K. 2004) or image verification/segmentation (Awad, M, Et al. 2007), but when puzzle solving is considered, perfect solutions are required. Some types of puzzles can have multiple solutions, with all of them being considered perfect. However, the puzzles featured in this system usually only have one or two possible solutions, so the solvers do tend to have a goal to move toward. As mentioned in the Rules section of this report, Kakuro puzzles are usually intended to only have one solution, but there are puzzles that exist that have more than one.

### Decision Trees

Decision Trees, or “Tree-Based Search”, is a non-parametric based method that implements the “Divide-and-Conquer” strategy (Alpaydin, E., 2020). Non-parametric based algorithms contain definitive decisions on each input that will generate an output based on certain ranges for that variable. So, each input will generate an output not based on its local area as a variable, but based on a set of decisions made by the algorithm. This is the opposite of parametric algorithms, which contain local regions for identifying input information. Parametric algorithms use graph clusters to classify sets of outputs, and take a ranges of values to determine the output, and then use probabilities to determine the answer.

This type of algorithm uses many nested and hierarchical methods of determining where the input data lies, then forms an output based on all of those statements. This means that the training data will set the hierarchic structure of the algorithm and determine the outputs based on how the inputs are analysed.

### Ant Colony Optimisation

Ant Colony Optimisation (ACO) stems from physical ant colonies. Ants give off and sense specific pheromones that can be used as a form of communication. In real colonies, signals could be used for situations such as food being found, the base being under attack, or rival ant colonies being found. Similar virtual pheromones are used in the Ant Colony Optimisation algorithm. A ‘path’ is designated to each node, and the ‘pheromone’ is used to designate how close a path is to the correct answer. As the algorithm continually develops the solutions, the pheromones will grow stronger or weaker as the feedback dictates. This system has been used to form a solver for other games in the past, such as Nurikabe (Amos, M., Crossley, M. and Lloyd, H., 2019, July.).

As mentioned previously in the Sudoku section, ACO has also been used to solve Sudoku in the past (Amos, M and Lloyd. H. 2019), which has shown that using evolutionary algorithms can be quite useful when solving NP-Complete puzzles and can nearly always be improved upon. The paper used a similar ant-based system to step through a Sudoku puzzle.

### Harmony based metaheuristic algorithm

Based on the way that musicians compose music when improvising, this Harmony based metaheuristic algorithm uses prebuilt small parts of a solution to pull together a full solution for the problem. The particular version of this method is the SAHS, or Self-Adapted Harmony Search algorithm (Panov, S and Koceski, S. 2013)

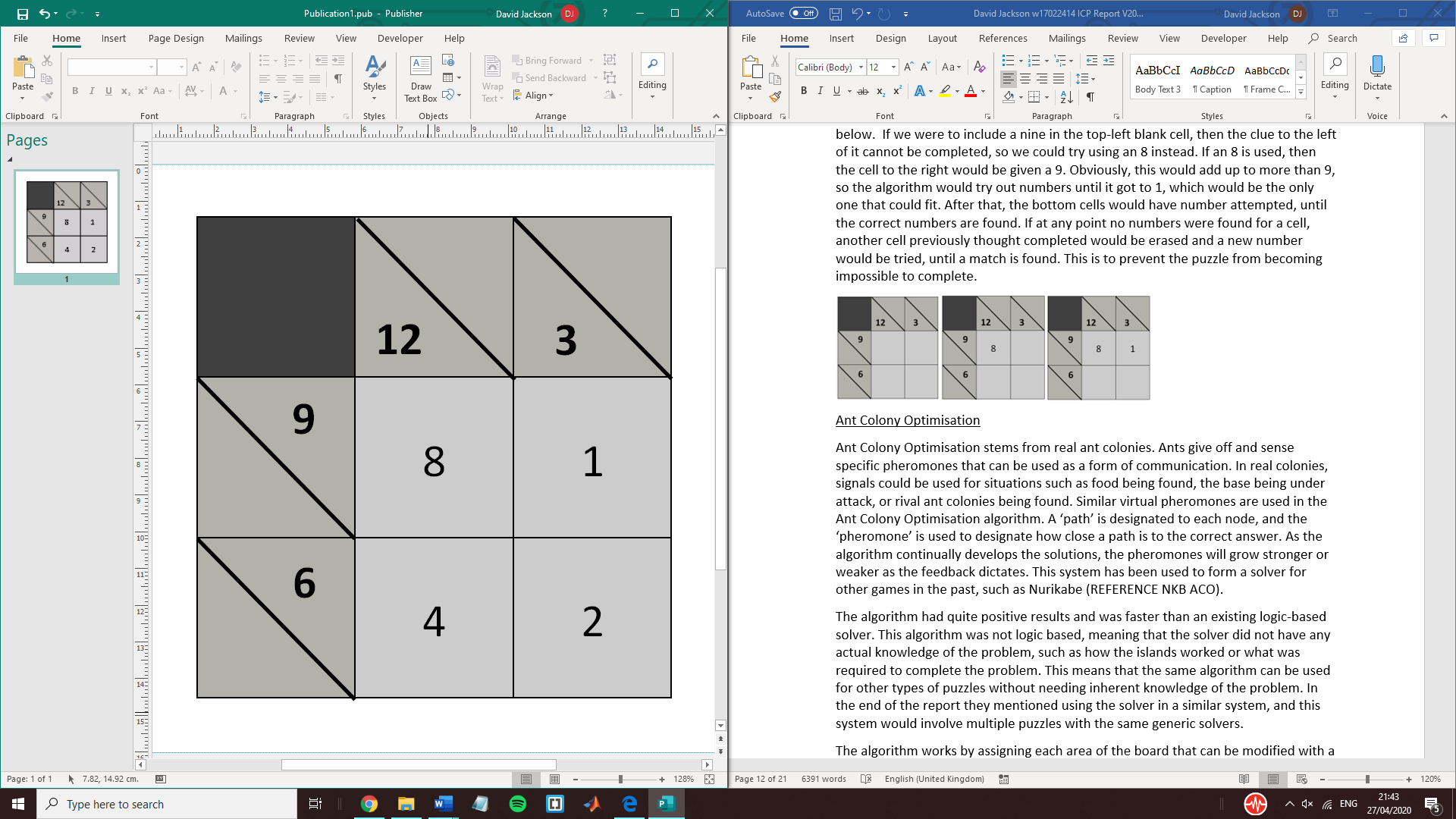
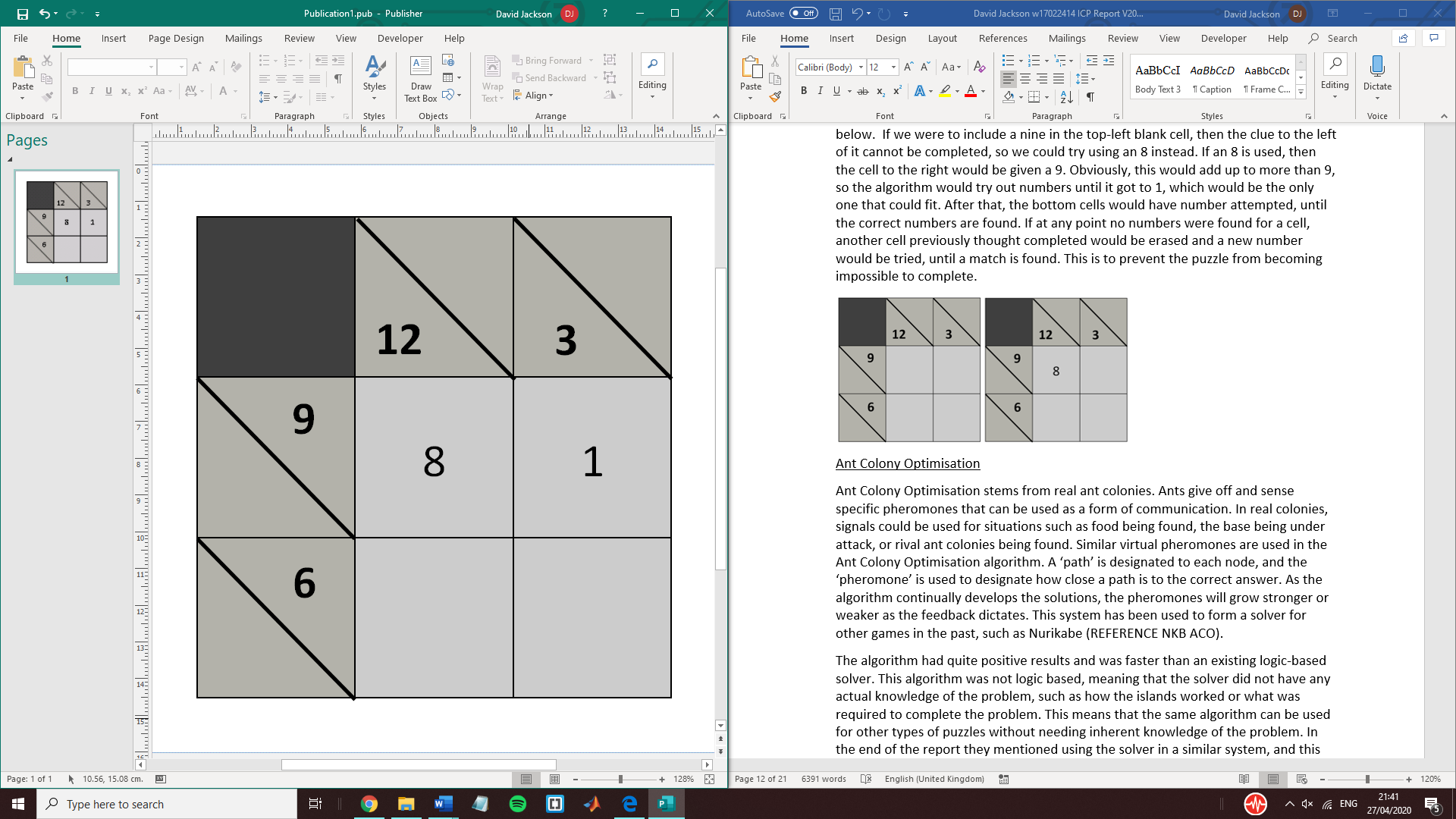
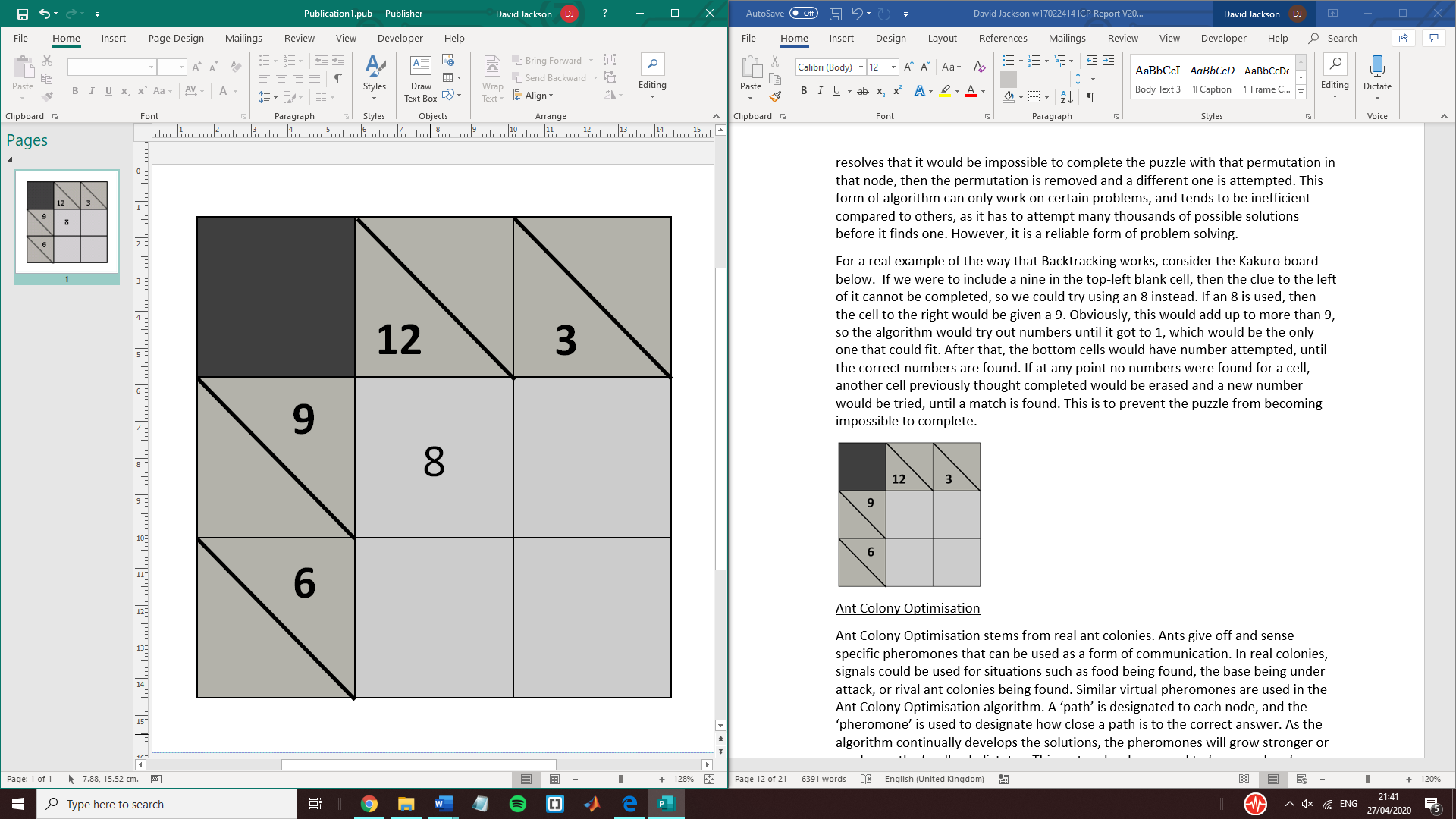
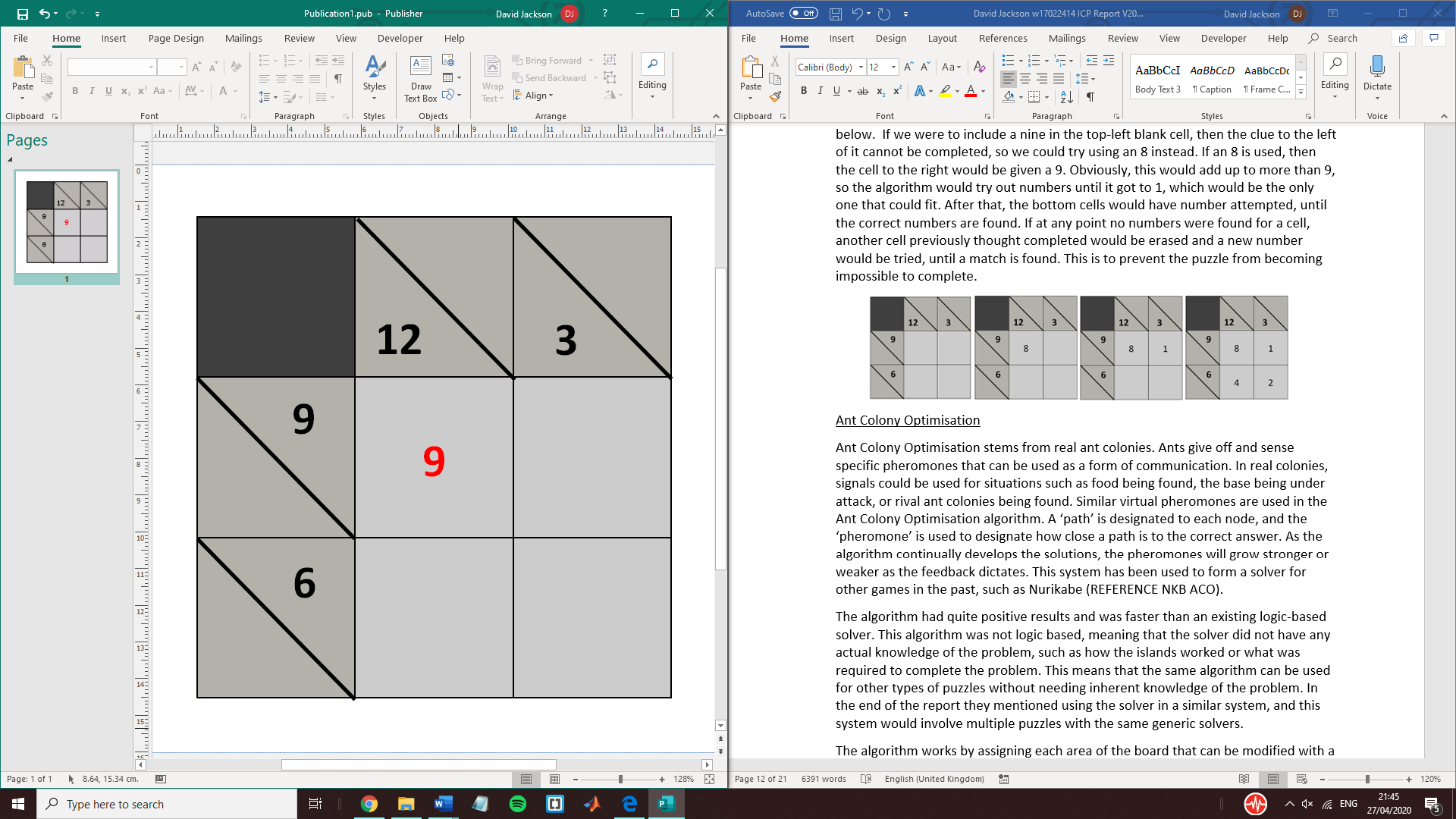
The report uses the SAHS algorithm to parse through the Kakuro board. Harmony search attempts to mimic the process of musicians improvising music pieces on the fly and draws from a bank of previous solutions to get as close to solving the problem as possible, similar to how musicians will use previous harmonies to develop new ones. Self-adapting by nature, they change as the problem is being solved. They are using a metaheuristic approach, which describes the act of allowing multiple separate solutions that are mutated throughout to be used on the same system. A higher level algorithm will be implemented that, depending on the specific problem, will choose a suitable solution and continuously mutate them to eventually solve the problem.

The problem with this paper, and some other types of novel metaheuristic algorithms, is that they are generally only slightly modified versions of pre-existing techniques (Weyland, D. 2015). The report mentioned details a theory that many of the novel methods that are based on real metaphors, are not much more efficient or have higher potentials than many of the more mainstream examples. In particular, the report focuses on using Harmony search to find possible solutions for the Sudoku puzzle. It is stated that Harmony search techniques offer no more potential than contemporary evolutionary strategies that are already being developed, and have not improved much in the time since they were theorised.

### Backtracking

Backtracking is likely the closest form of solving some of these puzzles to the way many people do it, although if people attempted to solve a puzzle with this exact method it would take an incredibly long amount of time. Backtracking works by very slowly going through each node in the board and testing each possible value for that node. It starts by filling in a single node and trying to solve the puzzle from there. If the node resolves that it would be impossible to complete the puzzle with that permutation in that node, then the permutation is removed and a different one is attempted. This form of algorithm can only work on certain problems, and tends to be inefficient compared to others, as it has to attempt many thousands of possible solutions before it finds one. However, it is a reliable form of problem solving.

For a real example of the way that Backtracking works, consider the Kakuro board below (figure 4). If we were to include a nine in the top-left blank cell, then the clue to the left of it cannot be completed, so we could try using an 8 instead. If an 8 is used, then the cell to the right would be given a 9. Obviously, this would add up to more than 9, so the algorithm would try out numbers until it got to 1, which would be the only one that could fit. After that, the bottom cells would have number attempted, until the correct numbers are found. If at any point no numbers were found for a cell, another cell previously thought completed would be erased and a new number would be tried, until a match is found. This is to prevent the puzzle from reaching what is known as a “Dead End”, becoming impossible to complete.



*Figure 4: The process of a backtracking algorithm*

### Random Search

Random search is exactly how it sounds, mostly random. While this is often seen as a bad thing, on small instances random searches can be quite positive. While many of the other algorithms are getting started, they still need to fine tune their approaches to the problem, wasting time in the process. For small instances, brute force can be quite useful, especially when the only constraints are a string of numbers. However, this usefulness falls off heavily as the instance size increases.

Random search algorithms tend to use number generators to provide strings of integers. These strings can be generated incredibly quickly, as they are rarely based on any specific variable and are generated as randomly as possible, meaning possible solutions are generated nigh instantly, and the longest part of the process is checking those strings against the constraints. A more improved method would be to check each string is not a duplicate of a previous attempt. However, this would often cost more time than it would save, therefore duplicates can often be found when running millions of solutions very quickly.

# Synthesis

Creating the product will require multiple phases of development. First of all, the requirements of the product must be specified and explained in order to cement what will be expected from the end result. This will involve what the simulator is expected to do, the external files that need to be created, and the requirements of the solvers to be able to run. After the requirements are specified, a more in-depth design summarisation will be shown. This will go in to detail about how each aspect of the program will need to run, and certain C++ features that will be used within the simulator itself.

## Requirements

In order to create a worthwhile product, the author will need to have designed the system in such a way as to satisfy each of the requirements that will be detailed below. These are the requirements that will define how the system runs and the capabilities of each aspect of the system.

The author will need to source and translate as many instances of Kakuro boards into a text form as they can. The number of instances will directly contribute to establishing the consistency and range of the simulator and solvers, so a large number of instances can only help the project. The text files will be in a form that the simulator will be able to understand and convert into a virtual version. Designing the simulator around these text files will be incredibly important so ensuring that every aspect of the text version is considered is of paramount importance.

The main and most important requirement is to build a simulator for the Kakuro boards that reads in a text version of the board and simulate the board virtually. The text file needs to only be read once and stored until it is not needed. The text board will just be a simple string of number characters that are separated by either a tab or a return. This will ensure that the simulator will remain consistent when reading the text file. Consistency is incredibly important as the entire process will not be able to continue if the board cannot be read in the first place.

Every part of the boards needs to be written in the C++ language. This language is undoubtedly the best fit for a simulator of this type, as the language has been heavily used for this type of system in the past, and has many advantages over other languages. For example, C++ allows the user to import specific libraries to use, therefore allowing more choice and control over each aspect of the program. As well as writing the program in the C++ language, the system needs to be tested and run in the Ubuntu programming environment, using the Linux command line. To do this, on university campus, The Linux Ubuntu operating system will be used on the dedicated Ubuntu computers, and when off campus an Ubuntu Virtual Machine will be used to simulate the operating system. The type of virtual machine software used has a tremendous impact on the efficiency and ease of use of the project, therefore after careful consideration VMware Workstation Player (Player, V., 2010) will be used. This software is known for its versatility and reliability, and the author has prior knowledge using the software and is well versed in the inherent systems.

When writing the code for the simulator, the author will use C++ classes to simplify the code and reduce the amount of systems in a single file. C++ classes are used to allow the instantiation of objects and are very useful when generating multiple objects. Separating these into separate files will allow separate debugging procedures to occur and will also allow errors to be resolved faster.

The simulator itself will have many aspects that will allow it to simulate a virtual version of the Kakuro board. As specified earlier, the simulator should be able to read in each text file in the same way and quickly convert them. C++ variables, vectors and arrays will be used to store information about the board. The instance itself will be instantiated into an object, and then converted into a 2D vector, that will allow iteration through each cell in the board when writing and reading information. Multiple functions are required to allow flexibility when testing the product. Functions also allow iteration through the board with lower possibilities of errors, as reusing code can often result in unforeseen errors. In order to solve each instance, two solvers are proposed: a randomised solver and a basic Genetic Algorithm. The randomised solver will generate a random string, insert them into the board and use a function to calculate the fitness of that solution. This will be iterated thousands of times until either the maximum number of evaluations has been hit or the solution has been found. The randomised solver will also check each instance against the previous best instance, and record the best solution and the best score for that solution. This will allow results to be analysed more easily, and in cases where the solution is not found, the best effort will be recorded amongst a number of attempts.

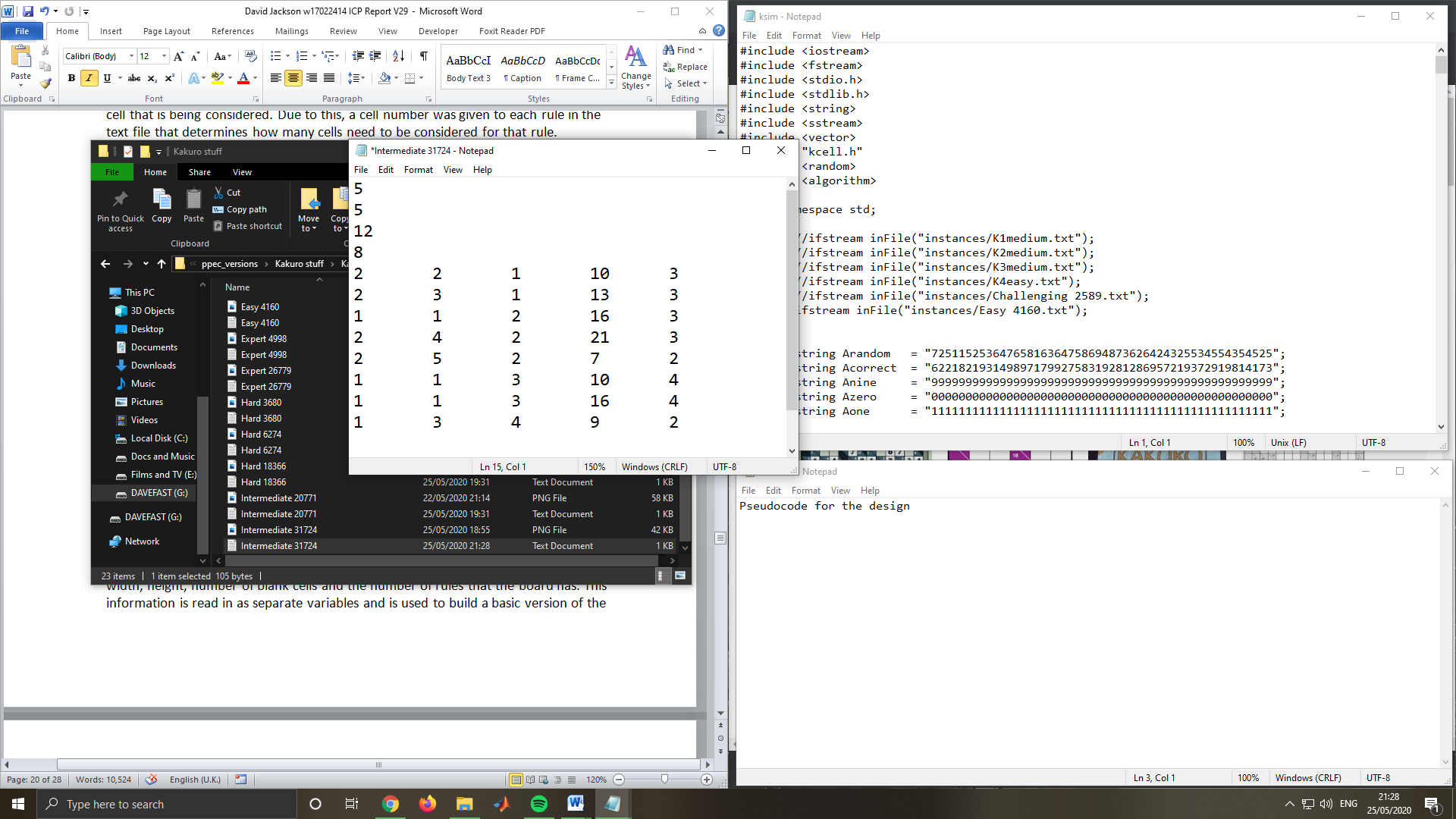
The Genetic Algorithm on the other hand will be a stripped down version of a full-fledged Genetic Algorithm. This version will have the main aspects of a Genetic Algorithm without some of the more advanced features. For example, many Genetic Algorithms run multiple generations at a time when solving a problem, whereas this one will only be running one generation. The main reason for this is due to the performance of the computers that will be used to run the simulations, as running multiple generations at once will be difficult to achieve with the limitation on power. The Genetic Algorithm instead will run a single generation that will iterate through a random string position and test different values, one at a time, and note when improvements are made. The solver will then remember the best solutions and use them on the next generation.

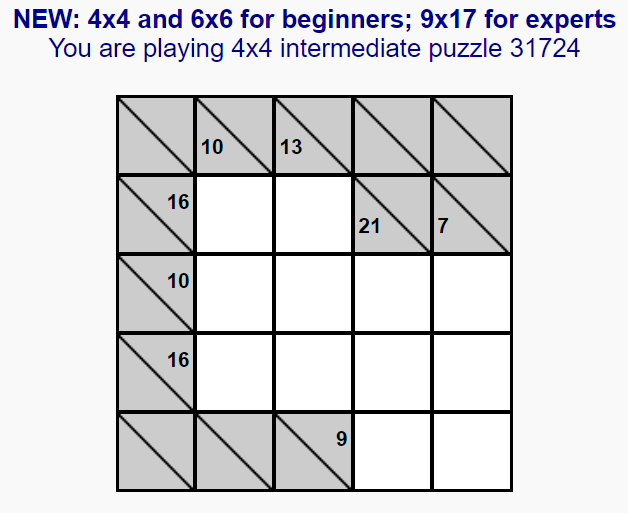
After the solvers are developed, there will need to be extensive testing of each solver and how they handle multiple different types of instances. The instances will vary in size and complexity, allowing an analysis of their efficiency to be undertaken, and an explanatio9n, if possible of the results. This will allow the author to come to conclusions about Kakuro and the relationship between the puzzles and the solvers. Tests will also need to be done that try to break the simulator, then results will be noted and the code will be remedied to fix them.

## The Simulator

In order to solve any Kakuro board instance, a simulator will need to be created that can read in a text file containing information about the board. The best way to do this is to strip down Kakuro into its simplest from, information about the size of the board and a list of the clue cells. This way, the simulator can very quickly read in the board and build a virtual version. If the simulator were to be developed fully for software use, boards could even be stored for use later, or used to test the simulator against others in order to learn more about the relationship between the simulator and the solvers.

As mentioned earlier in the report, Kakuro is a game consisting of a grid of blank cells, rule or ‘clue’ cells, and black ‘void’ cells. The basic rules of importance are as follows: the objective is to insert a digit from 1 to 9 into each blank cell such that the sum of the numbers in each entry matches the clue associated with it and that no digit is duplicated in any entry. The clue cells govern the cells to the right, below or in both directions to the clue. The game tends to be quite easy when the clue values are low, as there are far less possibilities for which numbers can be used in each blank cell. The hardest Kakuro boards are ones in which most of the clue cells have high values. To create the virtual board, a text file including all the information about the board instance is fed to the simulator.

Designing this board is not as straightforward as first thought, as many considerations were made as to what the simulator will need to know. As discussed in a presentation meeting that the author attended and presented in, when calculating the score for a given solution, care must be taken to assure that the calculating function does not include cells that are not governed by the specific clue cell that is being considered. Due to this, a cell number was given to each rule in the text file that determines how many cells need to be considered for that rule. Another aspect that needs to be considered is how to tell the simulator which direction the rule is governing. If the wrong direction is taken, it will very likely create segmentation faults that will stop the program from running, due to the simulator attempting to access part of a vector that is not controlled by the program.



*Figure 5: The board to be translated (left) and the board after translation (right).*

In the Kakuro Simulator, a text file was created to provide information to the simulator. The text file first holds information about the size of the board, its width, height, number of blank cells and the number of rules it holds. After that, each individual rule cell is detailed, and shows where the rule counts each cell and by the number of cells governed. While this may not seem like a board to human eyes reading the text file, the simulator will read that information and build a full Kakuro board using it. The width and height become the size of a 2D vector, the number of blank cells becomes the string length that the simulator sends to the solver, and the rule number becomes the size of an array that holds each rule to be imported into the virtual board. The text-based board can be created from any instance where the numbers on the board are between 1 and 9. There are variations of the Kakuro puzzle where the numbers go higher, sometimes up to 16. After 9, the numbers become hexadecimal, so 10 is A, 13 is D and so on. If this variant were to be included in the simulator, then changes to the way that score is calculated will be required, although it should not be that difficult to implement.

Kakuro is a very fitting puzzle as a simulator, with the only requirement being a string of numbers. With the length of the string preselected at the beginning, the solvers don’t even need to know the context of the board; they only need to know that the string needs to be in the correct order. This means that the puzzle should be easy enough to solve, with the only difficulty being at the end with getting the exact numbers. The Kakuro simulator that has been developed will be fully explained in the Synthesis chapters, later in this report.

## Designing the Simulator

Kakuro, as a grid-based pencil puzzle game, is a perfect fit for 2D arrays. The main objective of the simulator will be to build the supposed instance as a virtual board that can be iterated through and modified easily. This is very useful, as the board will not need to be changed much virtually compared to the paper version.

The simulators main purpose is to be the bridge between the solver and the board. This means that the simulator will automatically create a virtual version of the board that has the infrastructure to be solvable by the solvers. The communication between the solvers and the simulator will need to be developed and implemented as multiple classes. A way to read in any prepared board will need to be developed, and the board must be representable in command line when a solution is found.

As discussed earlier, the board will be in the form of a text file that represents a single instance of the Kakuro game. The text file will be read in using inFile and will be iterated through until the kCell object stores all the information required to build the board. The board, as a text file, will first hold generic information about the board, such as the width, height, number of blank cells and the number of rules that the board has. This information is read in as separate variables and is used to build a basic version of the board. After that, details of each rule in the board is shown, including the direction the rule is facing, its position in the board, the value of the rule and the number of cells governed by the rule. Once each of these rules is read into the simulator, a more advanced version of the board is constructed, and the board is then complete.

Implementing the design in this way means that the board can first be used to hold the proposed solution, and then print out the solution to the command line when it is found. Finding the solution is handled by the solvers. The solvers will use a string of specified number of integers between 1 and 9. When the constraints are given to the solvers, the solver will generate a solution based on those constraints. Each solution is then passed back to the simulator, and the fitness of the proposed solution is measured.

Fitness is measured by individually going through each rule in the board and comparing the amount that each cell is expected to add up to with what the actual cells add up to. This means that if the required value is 10, and the cells have the value 5 and 4, then the score would be 1. However, if the cells are 9 and 8, then the score would actually be 8, because negative numbers are taken as their absolute value, meaning that the -8 becomes 8. The reason for this is because if the absolute value wasn’t taken, then there’s a chance that most solvers would just increase the value in the cells until it reached the total value required, meaning that as long as the value matches the total value, it wouldn’t matter where each number is placed. Rectifying this was paramount, and incredibly important to make sure false positives weren’t reinforced early on in the solving stage.

## Implementing the Code

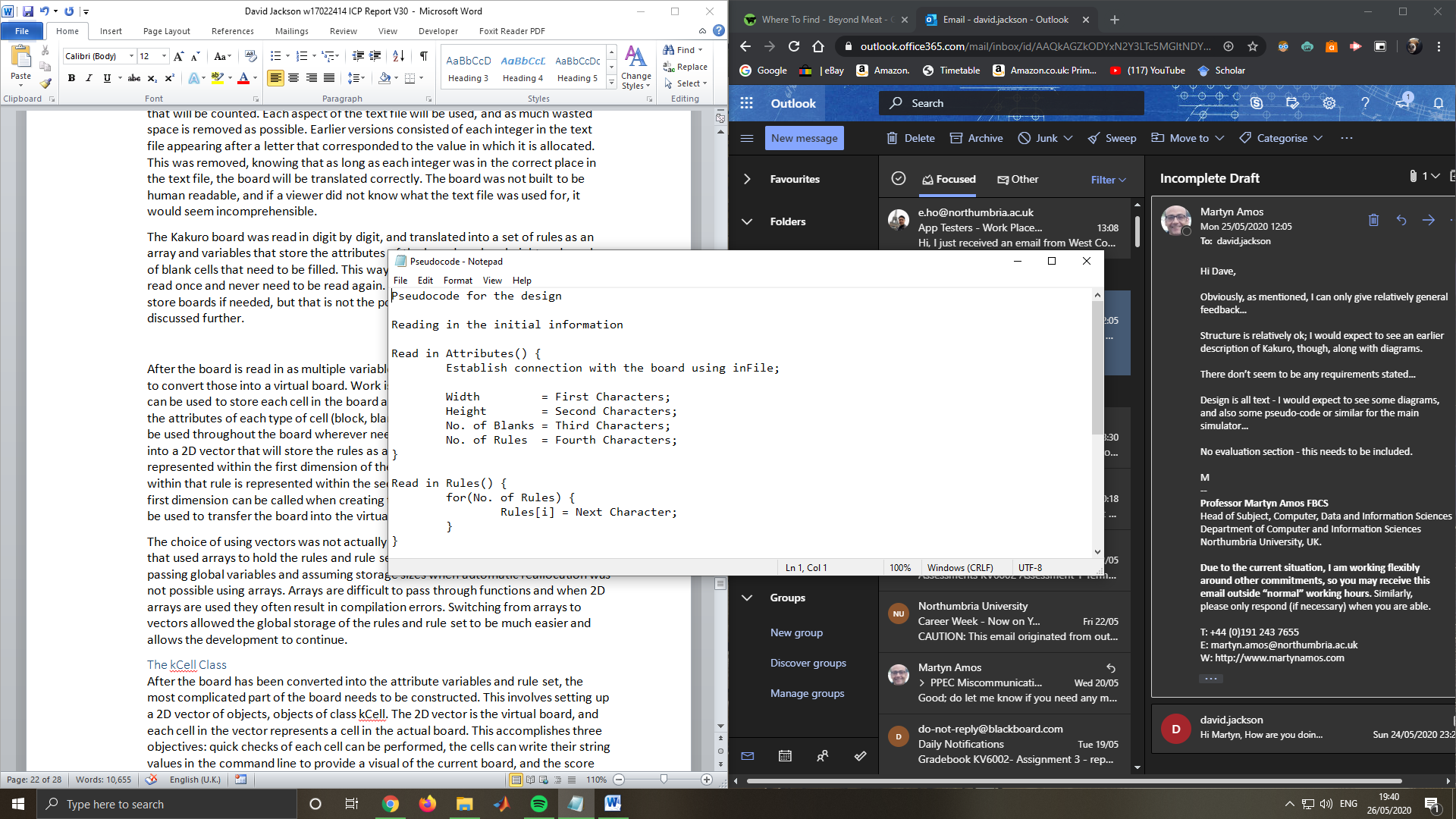
The author does not have any prior experience with using C++ and has mostly worked with Java and similar languages. Their brief experience with C was useful in constructing an idea of how the constraints will translate into the code, but much further study was necessary in order to gain a fuller understanding of the project details. C++ was chosen due to its fast runtime and lack of external supplemental systems that slow down the efficiency of applications. C++ is a versatile and powerful language that has been used extensively for decades.

An Ubuntu Operating System was used in order to develop the C++ code in an environment that would be efficient for running the system. When developing off campus, an Ubuntu Virtual Machine was used to simulate the same environment. Ubuntu has many tools that allow developing C++ code to be as efficient as possible. This includes a command line that can compile and run C++ code without any runtime penalties, and diagnostic systems that facilitate debugging procedures to make sure that testing the progress on the system is as easy as possible.

Firstly, using the notes on constraints a basic C++ file was created that would be the basis for the simulator to come. The first experiments were with code that will read in a text file and store some text as a variable. This was accomplished using the “ifstream” library. This library contains functions that allow text files to be read in and stepped through. After this could be done effectively, the focus moved to storing information from a basic version of the Kakuro board.

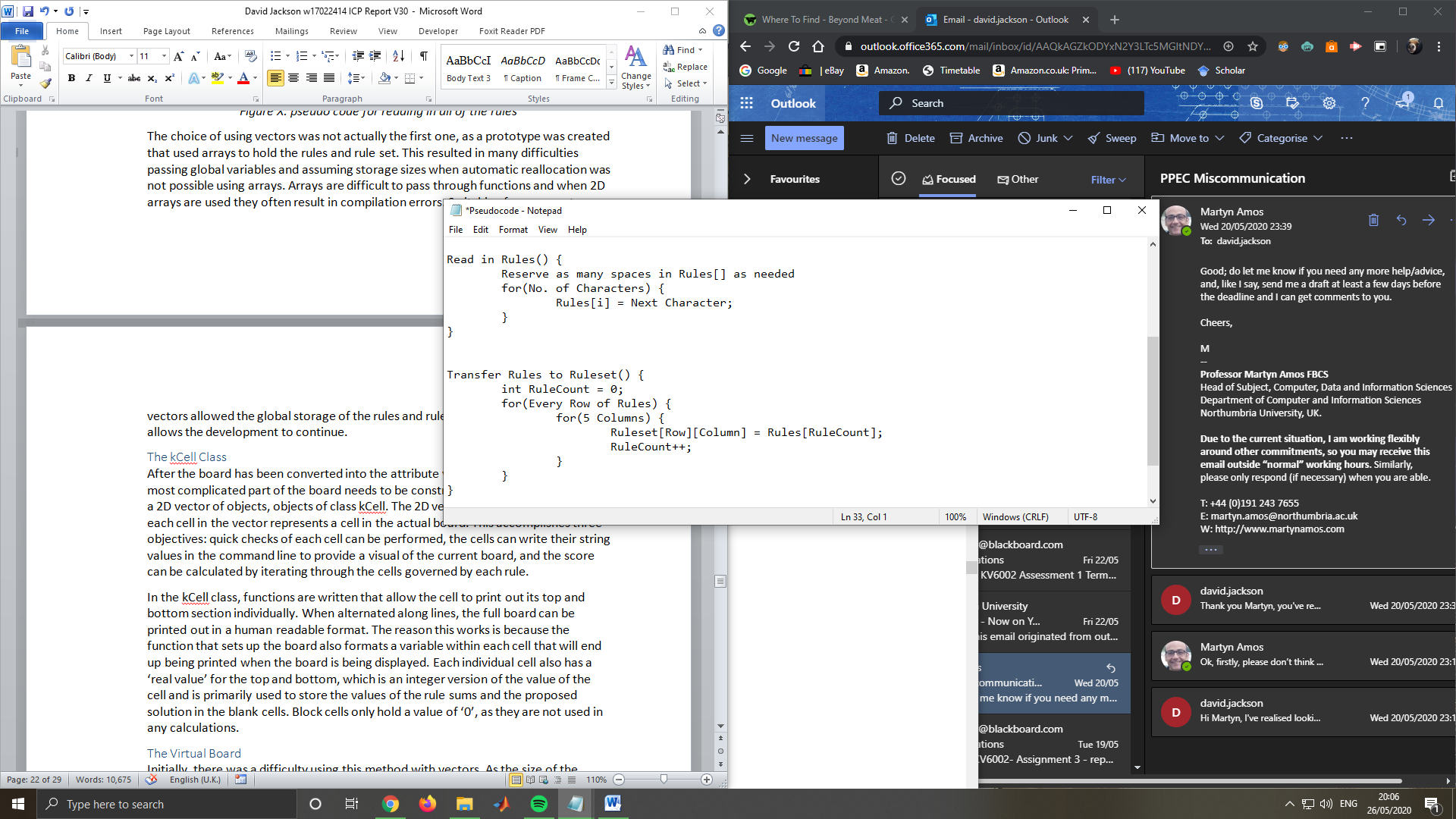
### The Text Board

The basic Kakuro board consisted of each attribute of the board that would be read in, followed by a shorthand version of each rule cell. The rule cells are recorded by their location on the board, the direction that they’re facing, and the number of cells that will be counted. Each aspect of the text file will be used, and as much wasted space is removed as possible. Earlier versions consisted of each integer in the text file appearing after a letter that corresponded to the value in which it is allocated. This was removed, knowing that as long as each integer was in the correct place in the text file, the board will be translated correctly. The board was not built to be human readable, and if a viewer did not know what the text file was used for, it would seem incomprehensible.

The Kakuro board was read in digit by digit, and translated into a set of rules as an array and variables that store the attributes of the board, such as height and number of blank cells that need to be filled. This way, it would be possible for the board to be read once and never need to be read again. This means it would also be possible to store boards if needed, but that is not the point of the simulator, so it will not be discussed further.

*Figure 6: pseudo code for reading in initial attributes*

After the board is read in as multiple variables and an array of rules, the next step is to convert those into a virtual board. Work is done to create a board cell class that can be used to store each cell in the board and the attributes therein. The class has the attributes of each type of cell (block, blank and rule cells), so that each type can be used throughout the board wherever needed. Each rule in the array is converted into a 2D vector that will store the rules as a rule set. Each line of the text file rules is represented within the first dimension of the vector, and each individual integer within that rule is represented within the second dimension. This means that the first dimension can be called when creating the board and each part of that line can be used to transfer the board into the virtual representation.



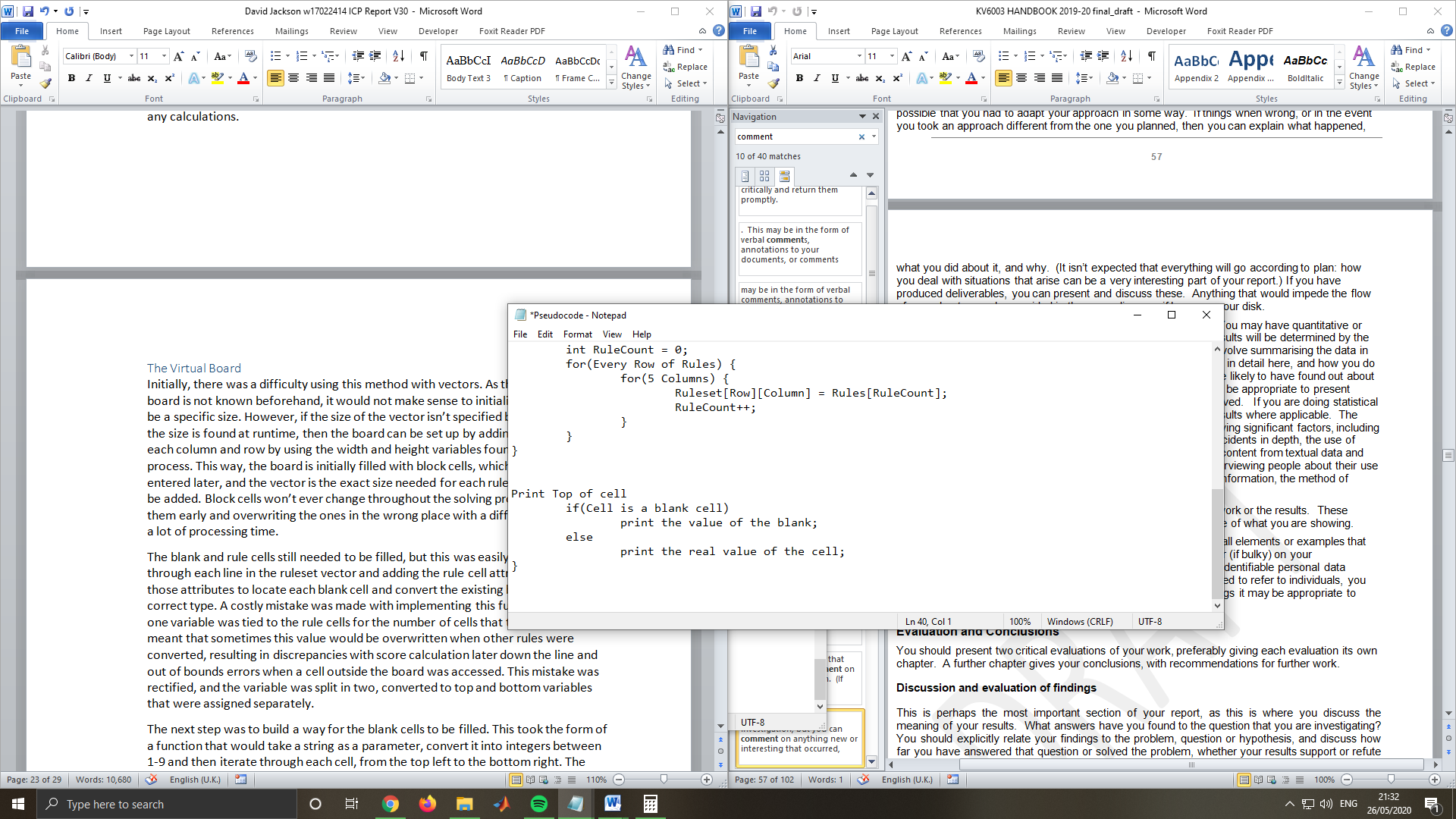
*Figure 7: pseudo code for reading in the rules, then converting them into a rule set*

The choice of using vectors was not actually the first one, as a prototype was created that used arrays to hold the rules and rule set. This resulted in many difficulties passing global variables and assuming storage sizes when automatic reallocation was not possible using arrays. Arrays are difficult to pass through functions and when 2D arrays are used they often result in compilation errors. Switching from arrays to vectors allowed the global storage of the rules and rule set to be much easier and allows the development to continue.

### The kCell Class

After the board has been converted into the attribute variables and rule set, the most complicated part of the board needs to be constructed. This involves setting up a 2D vector of objects, objects of class kCell. The 2D vector is the virtual board, and each cell in the vector represents a cell in the actual board. This accomplishes three objectives: quick checks of each cell can be performed, the cells can write their string values in the command line to provide a visual of the current board, and the score can be calculated by iterating through the cells governed by each rule.

In the kCell class, functions are written that allow the cell to print out its top and bottom section individually. When alternated along lines, the full board can be printed out in a human readable format. The reason this works is because the function that sets up the board also formats a variable within each cell that will end up being printed when the board is being displayed. Each individual cell also has a ‘real value’ for the top and bottom, which is an integer version of the value of the cell and is primarily used to store the values of the rule sums and the proposed solution in the blank cells. Block cells only hold a value of ‘0’, as they are not used in any calculations.

*Figure 8: pseudo code for printing the top part of a cell*

### The Virtual Board

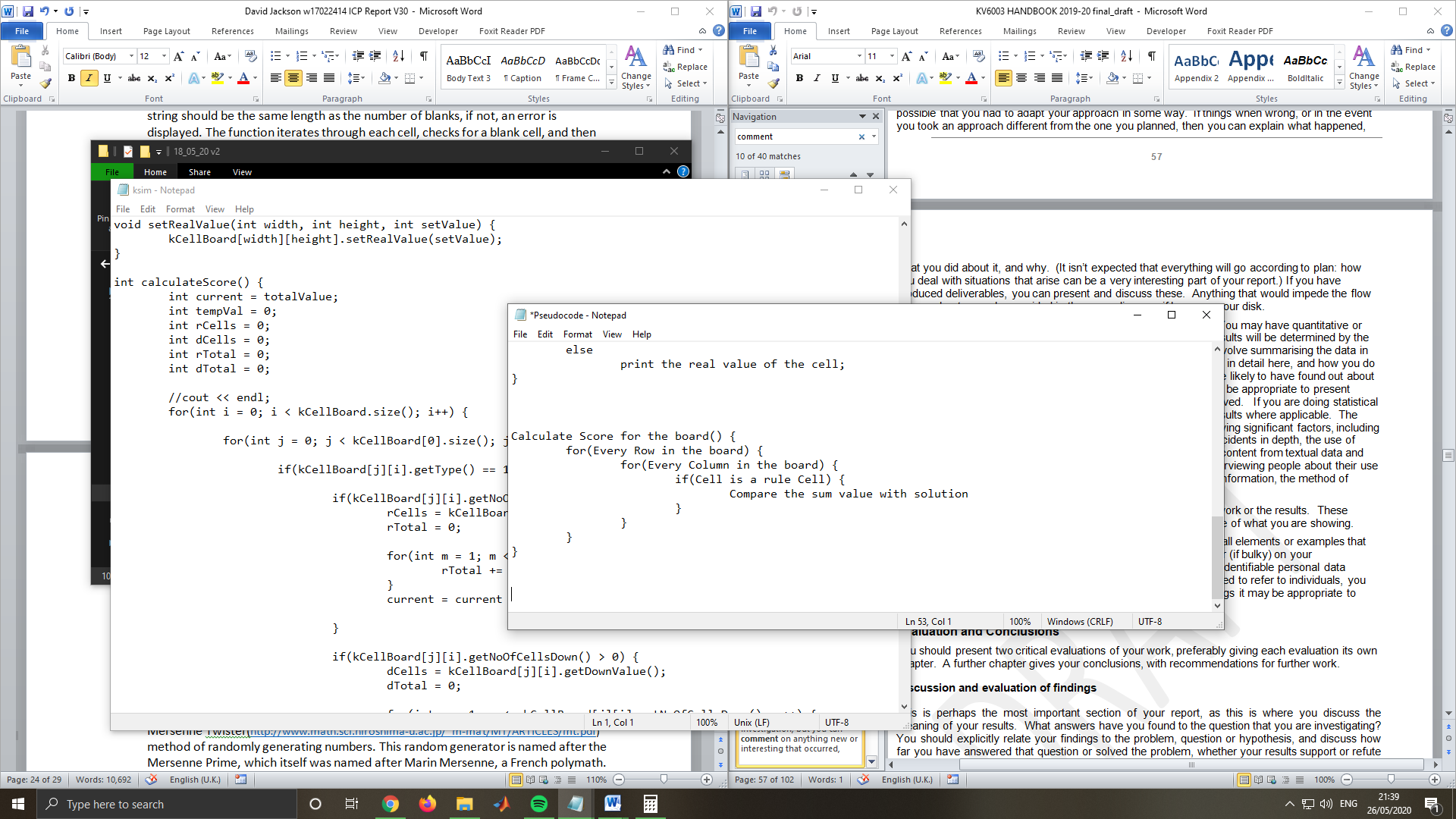
Initially, there was a difficulty using this method with vectors. As the size of the board is not known beforehand, it would not make sense to initialise the vector to be a specific size. However, if the size of the vector isn’t specified beforehand, and the size is found at runtime, then the board can be set up by adding block cells to each column and row by using the width and height variables found early on in the process. This way, the board is initially filled with block cells, which don’t have to be entered later, and the vector is the exact size needed for each rule and blank cell to be added. Block cells won’t ever change throughout the solving process, setting them early and overwriting the ones in the wrong place with a different cell will save a lot of processing time.

The blank and rule cells still needed to be filled, but this was easily done by running through each line in the rule set vector and adding the rule cell attributes, then using those attributes to locate each blank cell and convert the existing block cells into the correct type. A costly mistake was made with implementing this function, as only one variable was tied to the rule cells for the number of cells that they govern. This meant that sometimes this value would be overwritten when other rules were converted, resulting in discrepancies with score calculation later down the line and out of bounds errors when a cell outside the board was accessed. This mistake was rectified, and the variable was split in two, converted to top and bottom variables that were assigned separately.

The next step was to build a way for the blank cells to be filled. This took the form of a function that would take a string as a parameter, convert it into integers 1-9 and then iterate through each cell, from the top left to the bottom right. The string should be the same length as the number of blanks, if not, an error is displayed. The function iterates through each cell, checks for a blank cell, and then inserts the next value in the solution string. This carries on until each blank has been filled, and the next step is to calculate the score (or fitness) of the solution.

### Evaluating the fitness of a solution

Considering that the lower the score, the better, there needed to be a way to check how each rule calculated the blank cell values and assessed how close the solution was to a correct answer. At first the thought was to just sum the total values of the rue cells and compare to a sum of each time each value was checked, then subtracting the result. However, while this initially proved successful, and the function seemed to work perfectly, further tests revealed that merely swapping the places of some integers will still yield a perfect score, which leads to vastly incorrect answers being accepted as a correct answer. This was rectified by individually calculating the difference between the value of each rule cell and the blank values that have been inputted. If the result is negative, then that means that the proposed solution for that rule has integers that are too high, so to fix this, the absolute value of the result is taken. When the absolute value is taken, the negative values become positive, so when the integers provided are larger than the sum required, the score goes higher. After this change was implemented, the correct score was given each time.



*Figure 9: pseudo code for calculating the fitness of the board*

### Implementing a Randomised Solver

The next step was to implement a simple randomised solver, in order to test the values and ascertain a general time for how long it would take on average for the puzzle to be solved. The solver works by ascertaining the length of the answer needed and recursively generating a random string of the same length using the Mersenne Twister (Mastsumoto, M and Nishimura, T. 1998) method of randomly generating numbers. This random generator is named after the Mersenne Prime, which itself was named after Marin Mersenne, a French polymath.

The randomiser tests thousands of solutions against the simulator and updates the best solution for each run. The randomised solver is only intended to be used on smaller instances, however, as many of the larger Kakuro Boards can have over 40 blank cells, which as explained later, can have over 1.47 x 10^38 permutations for the answer string. Testing that many solutions in a realistic time is near impossible, especially as the author does not have access to a powerful computer. For this reason, the randomised solver is primarily used for smaller instances, around 6x6 or below.

### Implementing a Genetic Algorithm

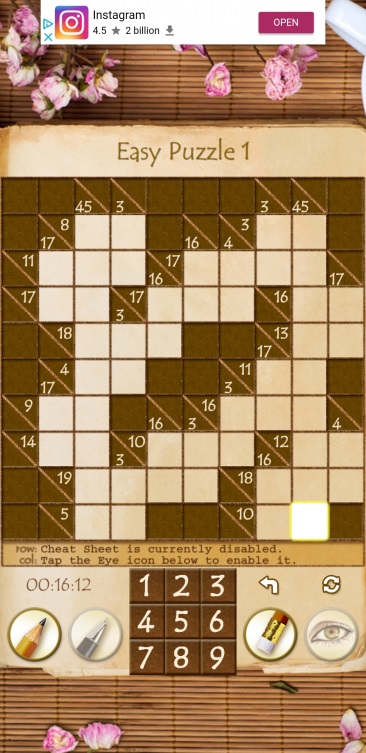
The idea behind the Genetic Algorithm (GA) in the Kakuro simulator is to slowly improve the solution over time, gradually working towards the correct solution. This is a basic version of a GA, that has the elements of most GA’s but does not have all of the advanced features that a dedicated solving algorithm can use. This means that the algorithm is not as fast as a fully developed algorithm, although it does still produce results on small to medium sized boards.

The algorithm works by first randomising a single string. This string then mutates by gradually incrementing the numbers one at a time then checking whether that has improved the score. If it has, then that solution becomes the new general solution, and the next section of the string is incremented. This method is intended to be used to slowly improve the score over time.

As this is a simple GA, it can fall into the problems of other GA’s that have been used to solve similar problems in the past. For example, the answers that are mutating over time might become stuck in a certain position. This is caused by a closed off area of the board having the wrong numbers that add up to the right values. For example, if the board can only be solved when a line is supposed to have the order “45”, but the solver has set the string so that the line reads as “36”, it would still add up to the right amount. This results in the 3 not being incremented, as that would increase the score, making the solution, in the GA’s logic, worse. Following that, it would affect other parts of the board, as other sums would not equal the right values due to the 3 not being incremented and the 6 not being replaced with a 5. These dead ends result in the string not being changed after thousands of generations. When these dead ends occur, there was no way for the simulator to detect these problems; therefore no progress would be made no matter how long the algorithm continued to run.

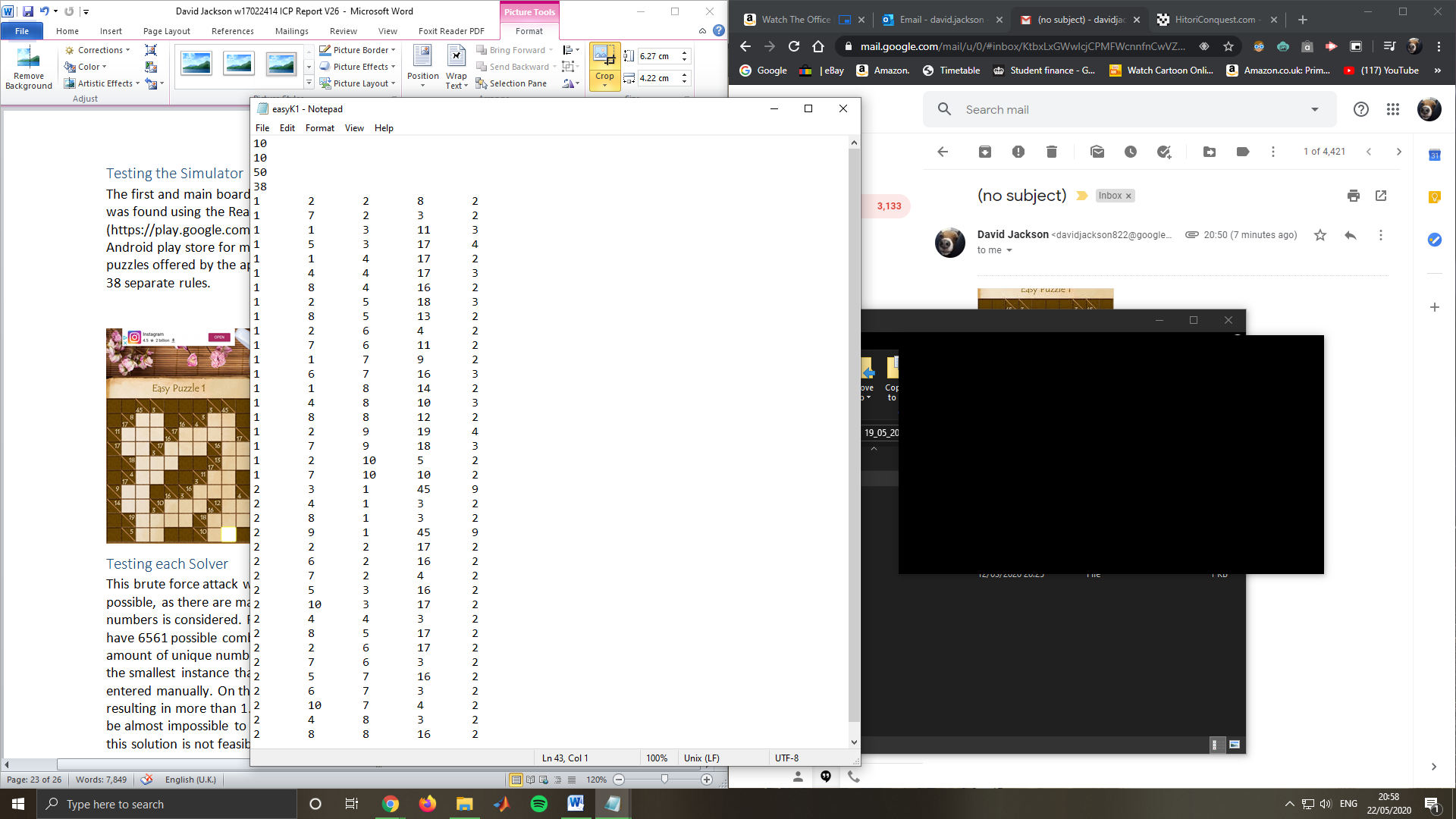
To solve this, changes needed to be made to the GA. While not the perfect solution, the solver will now check if a solution has not been changed in over 10 generations, and if it has not changed, then a small part of the board will randomly be mutated, or given a random set of values. This gives the solution a chance to be improved further, reducing the chances of dead ends happening. While not the most elegant solution, it will allow some of the larger instances to at least be solved as close as possible. If more time was given to complete the simulator, then a fully developed GA would be developed, which would produce far better results. The best improvement would be the ability to run multiple generations at the same time, and have competing solutions tested against each other to ensure genetic diversity.

## Testing the Simulator

The simulator has been tested continuously throughout the development process, and initially there was a single board that was translated into text form. This board has been used to ensure that every addition is working as expected and doesn’t interfere with other parts of the simulator. The Linux command line has been used extensively during testing, as it offers an immediate look at what the simulator is doing when run. The C++ function known as “cout” has been used to print out values of variables or to fully print out the virtual version of the board. Using this, every step of the process can be detailed, and it is far easier to solve problems with the code when it is known when exactly errors occur. Testing often occurred when new variables were added, in order to ensure that the correct value is being translated.

### The First Kakuro Board

The first and main board that was created has been called “K1easy.txt”. This board was found using the Real Kakuro application (Deucher, L., 2020) on the Android play store for mobiles. This was the first puzzle in the “easy” section of the puzzles offered by the app.

The puzzle is 10 cells by 10 cells, has 50 blank cells and 38 separate rules. When creating the text board, as seen below, the first two lines dictate the width and height of the board, respectively. The next two lines are dedicated to the number of blanks squares in the board, which is used when the solvers are attempting to find a solution for the board. The number of blank squares becomes the length of the answer string, so it is very useful to have in the text file. After the number of blank cells has been specified, the number of rules is then specified. A “rule” from now on will define a single sum value that has been specified in one of the rule cells. The value is used to determine the number of times that the function “CopyRulesToRuleset()” is called. This function separates the digits contained in the rules array into individual lines. When testing this section of the simulator, care was taken to ensure that every rule was transferred correctly, therefore a simple recursive nested for-loop was developed that would print out each line of the text file to ensure that each line was read correctly and stores it in the correct place.

### Creating Further Boards

The second Board was also generated from the Real Kakuro application, and was the second board in the “easy” section. This board has a slightly different layout, which is helpful when ensuring that the board can be printed out correctly. Every subsequent board was generated from the website “kakuroconquest.com”, as they had a range of generated boards that were smaller ad a better fit for the application. The naming convention for these boards are as follows; the difficulty of the puzzle, followed by the ID number for that puzzle. The size of the puzzle is contained within the board text itself.

Figure 1: The First Kakuro Board (top) and the text-version (bottom)

### Testing each Solver

When the randomised solver was being developed, it was known that it would be far from the most efficient solver for the larger instances. The randomiser is in its simplest form, a brute force attack to guess a string of numbers, so it is very similar to many hacking algorithms that attempt to crack a string of passwords.

This brute force attack would require many thousands, or millions, of attempts as quickly as possible. There are many permutations that are possible when a simple string of numbers is considered. For example, a tiny Kakuro board that has just four blank squares will have 6561 possible combinations of numbers, using the formula nr, where n is the amount of unique numbers (9) and r is the length of the string. This means that even the smallest instance that can be found has more possibilities than what can be entered manually. On the other hand, most Kakuro boards have over 40 blank cells, resulting in more than 1.47 x 10^38 possible combinations. This also means that it would be almost impossible to test every possible combination on most Kakuro boards, so this solution is not feasible for any but small instances.

# Results

In order to test each solver individually, it was decided by the author to run 5 tests of each solver on each instance, and note the results. It was also decided to test a combination of the solvers afterwards, 3 times for each. This will work by running the random solver first for 100,000 iterations, then using the best solution from those tests as a baseline for the GA. Using this method, brute force can be used to enhance the aspects of the Genetic Algorithm by giving it a stronger start. The following section will detail the results found.

The Y axis of each graph represents the best fitness for each test, and a lower value means a better solution. The graphs are not measuring the overall time taken because many of the instances would take far too long to solve and would skew the results, therefore the value that is being analysed is the efficiency of the algorithms, in terms of how fast they could lower the score before being cut off. The number of iterations for each type of algorithm has been adjusted so they take around the same amount of time, therefore ensuring a fairer test than just taking overall time.

## Random Solver

The first result that will be analysed will be Figure 12, shown below. This figure details the results from a test on every instance, using the randomised solver. The overall trend tends to be a slope upwards, showing positive correlation with instance size. However, there is a more noticeable difference, the fact that after Easy438, the average score increases rapidly, showing that a small change in scope can have a large impact after a certain point. More details on the separate instance sizes can be seen in appendices 2.1 and 2.2. As K2 easy, the only 3x3 instance, was fully solved every time, it will be excluded from many of the examples and analysis, as it was solved very quickly nigh instantly using any combination of solvers.

*Figure 12: Results for all instances with the randomised solver*

If the 3x3 and 5x5 instances are considered, (between K2easy and Easy438, more details in appendices 2.1) a trend is shown. As can be seen, apart from the single 3x3 instance (K2easy), each instance has relatively the same fitness. This can mean that for most 5x5 instances, the solver could find the solution if only given enough time and computing power. For these smaller instances, the results can vary each time the solver is used; however, the difference is only by a small amount, pointing to the fact that the solver is more reliable with smaller instances. K2easy, the only 3x3 that was tested, seems to nigh instantly become solved, as there are only 6561 permutations on a string of length 4. If you compare that to the 5x5, there are 59049 permutations, therefore unless duplicates can be excluded without damage to efficiency, it should take much more time to solve these.

On the other hand, consider the 7x7 and 10x10 respectively (between Hard6274 and K1medium, see appendices 2.2 for a more detailed graph). These instances also can vary, although the main difference is between the 7x7 and 10x10 (Expert26779 onwards). There is a much larger contrast between the 5x5s and the 7x7s compared to the 7x7s and 10x10s. In this increase, the average score seems to nearly quadruple, whereas the later increase doesn’t reach double. This may be due to the very large spike in permutations after the string length reaches past the 5x5 mark. For example, most of the 5x5 examples have a string length of

## Genetic Algorithm Solver

The Genetic Algorithm, although basic, seems to have a much better result than the randomiser. The worst cases for the randomiser were over 100, whereas no scores for the Genetic Algorithm even reached 35. This signals a much more efficient algorithm, and shows promising qualities if it were to be developed further. Further breakdowns of the Genetic Algorithm performance can be seen in appendices 2.3 and 2.4.

*Figure 13: Results for all instances using the Genetic Algorithm*

As can be seen from Figure 13, the same general trend is occurring, where the 5x5s are separated from the larger instances by quite a large margin. However, the margin is not nearly as large in terms of number, where the average difference is around 15 points, whereas the difference previously was around 30-40. This may signal a more efficient algorithm.

Another more noticeable difference is the increase in inconsistency between attempts. The random algorithm would at most have a difference of 5 points between attempts from all instance sizes, whereas the Genetic Algorithm has vast differences in fitness for each attempt. This is likely due to the single random string that is used for the base of the algorithm, and the overall fitness will likely depend on the initial random string. If the Genetic Algorithm were to be improved, the first improvement would have to be the possibility of multiple generations running at once. This would vastly improve the consistency and speed of the algorithm, and allow a much faster time to find the solution. The combination of both random and Genetic solvers is likely to either reduce the inconsistencies or offer a chance at a lower score than the Genetic solver alone.

## Combining the solvers

The main difference between this type of solver and the previous examples is that the random solver was first run under a shorter number of iterations, and then the result of that run was fed into the Genetic Algorithm as the beginning generation. This tended to produce better results that the Genetic Algorithm alone, as will be shown in a later explanation. There is not an incredible difference between the shape of this figure (14) and figure 13, although a few comparisons will be made. The first change is the fact that the combination of both Genetic and random solvers has solved instances in multiple attempts, namely the first two attempts of Hard3680 and the second attempt of Expert 4998. A closer look at these attempts can be seen in appendices 2.5 and 2.6. These solutions were shown to be correct when the author checked their validity against the original board.

There are also multiple instances where one or more of the instances have outlier attempts that generated a much lower score than before. This could be due to the fact that the random solver can create a much lower starting point for the Genetic Algorithm than when it is run alone. Expanding on that in a further project would place more emphasis on randomising in the right places, and enhancing the aspects of the randomiser that are beneficial, such as the speed at which solutions can be generated. If the combination of both solvers were to be improved, running more random permutations at the beginning would likely improve the average score, as it would provide a more solid basis for the Genetic Algorithm.

In order to fully analyse the results of these tests, a better representation of the relationship between the instance sizes is included in the next section.

*Figure 14: Results for all instances of both solvers being used*

## Comparing the Instance sizes

The following graphs represent two things. Figure 15 represents the average results for each instance size. This data was taken by separating the instances into size and then extracting the average by combining all results from that size. The obvious result that can be seen from this is the difference between the random solver and the other two. The random solver nearly always has at least twice the score of the others, meaning that it is far slower most of the time. The biggest difference can be seen in the 10x10 instances, as the random solver has nearly four times the score of the GA and the combination solver. This could be due to the fact that the random solver has far too many permutations to choose from and is far more likely to choose one with a lower score.

### Sorting by instance size

An interesting point to consider is the fact that the GA actually has a slightly better average score than the combination solver for the 7x7 instances. This could be due to an anomaly with the Intermediate20771 and Expert26779 (Figure 16) where they had much higher scores than Hard6274 and Easy1028, even though they are in the same instance size. This could be due to not testing the instances enough times, resulting in an unfair test of the results. If the tests were to be conducted again, many more tests would be done for each instance in order to ensure that such anomalous results were reduced.

*Figure 15: Average results by each solver, sorted by instance size*

### All Instances

The next figure (16), shows the same graph but split into each instance, in order to show the results from the previous figure in more detail. As can be seen, the Genetic Algorithm and combined algorithm can fluctuate when comparing averages. Once again, this could be due to the fact that the Genetic Algorithm only runs one generation at a time, and increasing the number of generations per iteration would vastly improve the consistency and efficiency of the algorithm.

*Figure 16: Average results from each solver, sorted by each instance*

### Conclusion

In summary, the general trend is that the randomised solver tended to perform the worst out of the three, and the Genetic Algorithm worked much better than the random solver. When comparing the Genetic Algorithm with the combined version, the combined version on average performed much better, although it was less consistent on larger instance size. Disregarding a few anomalies, the combined algorithm proved very efficient in the small time it was given to solve the problem, and if further improvements were to be made, then the randomised starter string would be given more time to produce a starting generation. Another improvement would be to run multiple generations at once in the Genetic Algorithm, in order to ensure consistency and allow multiple threads of possible solutions to improve over time.

# Evaluation

The aim of this project initially was to develop a Kakuro simulator that would integrate with a system developed outside of the scope of this project that has similar simulators for other pencil puzzle games. The pre-existing system is currently called the PPEC (Pencil Puzzle Evolutionary Computation) and is a work in progress, and as stated the original aim was to integrate the Kakuro simulator into the set of multiple simulators contained within that system. Those other simulators could interface with multiple other algorithmic solvers that would work similarly to the solvers that the author has proposed; a generic solver that takes a string of numbers and attempts to slowly build towards a specific solution that is tested by the simulators.

The end product of this project is similar in this regard, there is a system that simulates a puzzle game and uses multiple solvers to attempt to pass a string in to the simulator that will evaluate the suitability of the string for solving the instance. However, there was difficulty integrating the built product with the pre-existing system so the objective of the project needed to be changed, in order to accommodate the time constraints of the project. Due to this, the objectives were changed and the new aim was to generate an isolated simulator that would attempt to solve a number of instances by using simple algorithms, then tests would be conducted that would analyse the efficiency of the developed solvers. The rest of this evaluation will keep in mind those objectives.

## Evaluation of the product

While the aim of the project has changed, the author believes that the new aims and objectives have been satisfied. One reason is that the main objective was to develop a simulator that can read in a text version of a Kakuro board and generate a virtual version of the board, fully printable to the command line and adjustable. This objective has been completed. The simulator is able to read in any prepared text file, establish the attributes of the board, then read in each rule for the board and convert that rule into a virtual board, fully satisfying the constraints of the puzzle. The simulator is able to then print out the entire board so that it takes the form of the original instance of the Kakuro puzzle, therefore any viewer can see the entire board in its original form. The secondary aim was to allow the simulator to test the fitness of any solution in the form of a string, and output the result into a variable. While improvements could be made to the speed and efficiency of this calculation, the calculation itself does exactly what it is meant to do; give a score to any string that is passed into it.

The solvers that have been developed were originally supposed to be able to generate the correct solution to any given Kakuro board. This is technically the case, although the results for the much larger boards seem to suggest that the solvers are nowhere near as efficient as they could be, and many improvements could be made to them. One of the main reasons that they are not as efficient as they could be might be due to the time constraints that originated from the change in aims midway through the project. If the author were to begin the project with the aim to create multiple solvers, they might have a better chance of creating more efficient and advanced versions. The changes would likely include, but are not limited to: allowing multiple generations at a time that could mutate and pass their genes onto the next generation, creating more types of solvers to ensure a more thorough analysis of the solvability of the Kakuro puzzle, and lastly to attempt a more cohesive interface with the simulator. The interface currently allows the user, when starting the program using the Ubuntu command line, to specify the file path of the specific instance that they want to solve, and the type of solver to use, “Random”, “GA”, or “Both”. Further improvements could be made to specify a time or number of iterations to perform, running the solver multiple times, or a continuous interface that will allow multiple runs to be made based on user input while the simulator is still running.

There are, however, some downsides to the way that parts of the product are implemented. For example, the Kakuro boards that have been used to test the product have to be converted manually into a text form, and it might have been easier to assist the conversion if secondary software was written that would assist the user in converting a regular board into a text based version, possibly by computer vision analysis or a non-text based interface using JavaScript or some other tool making language. This additional software would allow boards to be converted much faster and more examples could be made in a shorter time. However, these types of systems are usually the focus of entire builds rather than side projects to a separate, larger, project.

Another disadvantage of the simulator is that there is a small chance that some of the boards can generate a score of zero if the board is not in a common form. For example, some of the boards that are essentially a large block of blanks cells without strips of block cells separating them can be configured by the solvers in such a way that allows them to generate a perfect score of zero even if the board has strips the same integer in either the horizontal or vertical axis. This would normally break the rules of Kakuro and result in an invalid solution, whereas the calculations should not be able to break those constraints in the first place. If the author were to attempt this project a second time, they would ensure that the score calculation would check for invalid solutions at the same time as specifying the fitness of the solutions. This was originally implemented in a much earlier version of the score calculation, but after many remodels of the function were made, the verification process was removed as it took a long time to complete. A separate function that checks the validity of solutions would likely be the best way forward, and any solution by the solvers that generated a score of zero would be checked against its validity before sending the score to the solvers.

The testing aspect of the solver could also have been improved, as only performing three tests on each solver and instance seems to have produced anomalous or unfair results, whereas performing many more tests would be beneficial to the analysis of the solvers. If the tests were to be conducted with more time and a much more powerful computer, then the tests could instead be used to identify how long the solvers took to generate a correct answer, rather than constraining the time and performing tests in how the low the score could get within a certain time frame. This change would lead to more useful results, but for current circumstances the most that could be done has been done.

As the product does not seem to concern any form of legal, social, ethical or professional issue, this evaluation should not need to concern those issues. If the original product were developed, that made use of the PPEC system, care would have to be taken not to release the source code for that system and protect the authors of the software. The simulator that the author of this report has developed would also need to be made useable by the authors of the PPEC system, as this has been stated in the ethics form, and full consent is given to allow the simulator to be modified and integrated with PPEC in the future if they so wish.

## Evaluation of the project process

As was explained in the introduction to this section, the project objectives were changed partway through this project. This was due to the fact that the simulator could not be integrated with the pre-existing system, partly due to inexperience with the C++ language itself and partly due to the fact that the author does not have innate knowledge of how the original simulators were integrated. The author does not have experience with integrating code into pre-existing software, and all of their previous projects have been self-isolated and standalone products.

As has been shown before, the author does not have any prior experience with writing in the C++ language, and is more familiar with languages such as Java, JavaScript and other more web-based languages. This inexperience with the source language has generated many problems for the author, such as differences in how the language handles arrays and functions for example. However, the author believes that the research and testing that was conducted has made up for the inexperience with the language, and feels that they did the best job that they could, especially considering the circumstances of the current (as of this report) Covid-19 quarantine, which will be explored later in this evaluation.

A large problem with the project in general could be the fact that the author has difficulties in asking for help in opportune times, and the author is attempting to rectify this and learn from their mistakes. They should have asked for help integrating the system much earlier, or ask for a detailed description of how the PPEC system fully operates and how to better work together to integrate the Kakuro simulator.

Early on in the project, there was a meeting with other contributors to the PPEC system that was managed by Martyn Amos. The point of the meeting was to present each simulator to the group and explain the choices that were made about the design of the product. The author of this report gained a much better understanding of the system during this meeting, and they feel that they should have followed up with more questions about the PPEC simulator. This meeting helped the author to develop a better product and in the end was a positive impact on the project as a whole.

The project overall should have been planned more carefully, and the author should have sent preliminary drafts of code to be examined more often. This resulted in progress being slower than expected and the project plan changing over time. If the project plan was more specific about the number of meetings and more care was taken to ask for feedback, then the project would likely have had a stronger outcome. However, the author does believe that they have performed very well compared to previous projects, and the product was the most ambitious idea that they have previously undertaken. The author also believes that they have vastly improved their knowledge of C++ and the Ubuntu command line, and is now able to make C++ and standalone products much more easily than before.

Currently, as of this report being written, a lockdown is in place by the British government to slow and prevent the spread of Covid-19. While obviously, this is for the benefit of the entire country, some details need to be noted about the effect this has on the author’s ability to work. Northumbria University facilities have been closed, therefore access to useable computer hardware and software is restricted. The author, while developing the product, has been forced to share a computer with another person, therefore reducing the amount of time that can be spent on the project and vastly reducing the computing power available for use. An Ubuntu Virtual Machine has been used to simulate the operating system that the product was originally intended to be run on. This virtual machine is far less powerful than the intended computers, therefore it is harder to test software and fix errors. This is one of the reasons that testing is not based on time taken but rather on progress within a certain timeframe. While this is not the sole reason that the product does not exceed every standard than was expected, the current situation does affect the end product.

## Conclusion

Overall, the product that resulted from this project was software that could simulate the puzzle game Kakuro. The software could then read in a text file that is the bridge between a real Kakuro board and the simulated virtual version. The simulator can then use three different versions of a solving algorithm, with one of those being a combination of the first two, to attempt to solve the Kakuro board being submitted and provide a solution in the form of a text string. While the original goal of the project was just a simulator that could integrate with other simulators and solvers in an external system, the end product is very close to the initial goal, with the extra solvers being developed to test the simulator and the authors knowledge of machine learning techniques.

The project did have difficulties, as the author could have spent more time asking for help and feedback rather than attempting to do everything themselves. However, the system turned out quite well given the circumstances. More testing could have been done if the author had more time and resources, or if they had managed to complete the original objectives of the project. The author is satisfied that they have completed the project to the best of their abilities, especially considering the current Covid-19 lockdown.

Finally, the tests that have been completed have shown that the randomised solvers are nowhere near as efficient as the Genetic Algorithm that has been developed and the combination of them both, aside from some anomalies, have shown to be better than both of them when separated.

## Recommendations

If the author were to expand on the product in any way, the first improvement would be to include more types of algorithmic solvers. This would develop understanding of different techniques, and enhance the author’s knowledge of how these techniques can be used. Another consideration is the slight flaw when calculating the fitness of a problem, where sometimes the solutions that gain a score of 0 can still be invalid with the same integer repeating along an axis.

Further improvements to the GUI would benefit the system greatly, and improving or streamlining the simulator architecture would enhance the overall product.

# References

Alpaydin, E., 2020. Introduction to machine learning. MIT press. pg213, Available at: <https://books.google.co.uk/books?hl=en&lr=&id=tZnSDwAAQBAJ&oi=fnd&pg=PR7&dq=introduction+to+machine+learning+3rd+edition&ots=F2_PbV5oyc&sig=-Jb3VB7fv7QZ_SXiA9rwEY983g8&redir_esc=y#v=onepage&q=introduction%20to%20machine%20learning%203rd%20edition&f=false>

Amos, M., Crossley, M. and Lloyd, H., 2019, July. Solving nurikabe with ant colony optimization. In Proceedings of the Genetic and Evolutionary Computation Conference Companion (pp. 129-130). Available at: “https://dl.acm.org/doi/abs/10.1145/3319619.3338470”

Amos, M. and Don, O., 2007, September. An ant-based algorithm for annular sorting. In 2007 IEEE Congress on Evolutionary Computation (pg. 142-148). IEEE. Available at: “<https://www.researchgate.net/publication/4307916_An_ant-based_algorithm_for_annular_sorting>”

Amos, M and Lloyd, H. (2019). Solving Sudoku with Ant Colony Optimization. [PDF] IEEE, Available at: <https://ieeexplore.ieee.org/abstract/document/8845599>

Asif, M and Baig, R (2009). Solving NP-complete problem using ACO algorithm. [PDF] IEEE, pg1-4, Available at: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5353209>

Awad, M, Et al. (2007). Multicomponent Image Segmentation Using a Genetic Algorithm and Artificial Neural Network. [PDF] IEEE, pg 1-5, Available at: <https://ieeexplore.ieee.org/abstract/document/4317521>

Crook, J.F., 2009. A pencil-and-paper algorithm for solving Sudoku puzzles. Notices of the AMS, 56(4), pp.460-468. Available at: “<https://www.researchgate.net/publication/266964597_A_pencil-and-paper_algorithm_for_solving_Sudoku_puzzles>”

Deucher, L., (2020), Real Kakuro Free – Cross Sums [Mobile Application Software], Retrieved from: “<https://play.google.com/store/apps/details?id=co.rottz.realkakuro&hl=en_GB>”

Dsudoku (2013). Nurikabe Rules and Info. [online] The art of puzzles. Available at: <https://www.gmpuzzles.com/blog/nurikabe-rules-and-info/>.

Felgenhauer, B. and Jarvis, F., 2005. Enumerating possible Sudoku grids. Available at “<http://members.home.nl/jfhm-bours/English/Projecten/Sudoku/sudoku.pdf>”

Hartmanis, J., 1982. Computers and intractability: a guide to the theory of NP-completeness (michael r. garey and david s. johnson). Siam Review, 24(1), p.90. Available at: “<https://www.semanticscholar.org/paper/Computers-and-Intractability%3A-A-Guide-to-the-Theory-Garey-Johnson/bdede1e17c947540b50e6e2db9e8467ddc6e7336>”

Holzer, Et al. (2014). On the NP-completeness of the nurikabe pencil puzzle and variants thereof. [PDF] Germany: Technische Universitat Munchen, pg 10-13, Available at: <https://www.researchgate.net/profile/Markus_Holzer2/publication/228986509_On_the_NP-completeness_of_the_nurikabe_pencil_puzzle_and_variants_thereof/links/00b49524963b29c0a6000000.pdf>

Houston, R. Et al. (2018). Zen Puzzle Garden is NP-complete [PDF], pg 1-4, Available at: <https://arxiv.org/pdf/1106.2104.pdf>

Inada, H and Ishii, K (2004). Bipedal walk using a Central Pattern Generator. [PDF], pg 1, Available at: <https://www.sciencedirect.com/science/article/abs/pii/S0531513104011641>

Mastsumoto, M and Nishimura, T (1998) Mersenne Twister: a 623-dimesionallly equidistributed uniform pseudorandom number generator [PDF] pg3-19 Available at: <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/mt.pdf>

Panov, S and Koceski, S. (2013). Deterministic and metaheuristic approaches to solving Kakuro puzzles. [PDF] IEEE, pg1-4, Available at: <https://ieeexplore.ieee.org/abstract/document/6601364>

Panov, S. and Koceski, S., 2014. Solving kakuro puzzle using self adapting harmony search metaheuristic algorithm. International Journal of Engineering Practical Research, 3(2), pp.34-39. Available at <http://eprints.ugd.edu.mk/12358/1/IJEPR044.pdf>

Player, V., 2010. 7.1. VMware.

Rubin, S.H., Bouabana-Tebibel, T., Hoadjli, Y. and Ghalem, Z., 2016, July. Reusing the NP-Hard Traveling-Salesman Problem to Demonstrate That P~ NP. In 2016 IEEE 17th International Conference on Information Reuse and Integration (IRI) (pp. 574-581). IEEE.Available at “<https://ieeexplore.ieee.org/document/7785793>”

Ruepp, O. and Holzer, M., 2010, June. The computational complexity of the KAKURO puzzle, revisited. In International Conference on Fun with Algorithms (pp. 319-330). Springer, Berlin, Heidelberg. Available at: “<https://link.springer.com/chapter/10.1007/978-3-642-13122-6_31>”

Salcedo-Sanz, S., Portilla-Figueras, J.A., Ortíz-García, E.G., Pérez-Bellido, Á.M. and Yao, X., 2007. Teaching advanced features of evolutionary algorithms using Japanese puzzles. IEEE Transactions on Education, 50(2), pp.151-156. Available at “https://ieeexplore.ieee.org/abstract/document/4200842”.

Seta, T.: The complexities of puzzles, cross sum and their another solution problems (asp). Senior thesis, University of Tokyo, Department of Information Science, Faculty of Science, Hongo 7-3-1, Bunkyo-ku, Tokyo 113, Japan (February 2001)

Simonis, H., 2008. Kakuro as a constraint problem. Proc. seventh Int. Works. on Constraint Modelling and Reformulation. Available at “<http://130.238.12.100/research/group/astra/ModRef08/Simonis.pdf>”

Srinivasan, N., 2011, June. Tutorial: Current state of p vs np problem. In 2011 International Conference on Recent Trends in Information Technology (ICRTIT) (pp. 1-1). IEEE.

Weyland, D. (2015). A critical analysis of the harmony search algorithm—How not to solve sudoku. [PDF] IEEE, pg1,7, Available at: <https://www.sciencedirect.com/science/article/pii/S221471601500010X>

# Appendices

Appendices 1: Terms of Reference

***Comparative Analysis of AI-Based Solvers for the NP-Complete Problem Kakuro***

***Background***

Puzzle games have entertained civilizations as far back as recorded history, alleviating boredom with a simple activity that all ranges of people can participate in. They engage the mind, improve cognitive thinking skills, and can be attributed to an overall increase in problem solving skills when developing. Non computer-based puzzle games can range from simple connect the dots pictures to more complex puzzles like crosswords. These puzzles usually involve either words or numbers, but some of the simplest puzzles involve just moving a single piece around a game board. These types of games seem simple at first, but when the size of the board is increased, the problem starts to become much more complex. The complexity arises from the fact that each node or square affects each other node, meaning that each change has a larger impact on the board when larger boards are used. Automatic solvers also have difficulty with this, because the large boards have far more combinations of solutions.

When computers were invented, many new types of puzzle games were possible, and they allowed many generations to enjoy puzzle games in a new way. In 1961, the computer game Nimbi, inspired by the ancient game Nim (Jorgensen, A.H., 2009), was instrumental in developing computers at the time and a driving force in creating computer programs that could oppose the player. This opposition is similar to modern video game AI systems, and may be one of the first known cases of an AI(Jorgensen, A.h., 2009). While not technically an Artificial Intelligence system as we know it, the game could make moves against an opposing player in order to win, which when stripped down is a very basic form of intelligence.

There are many popular puzzle games that are solely computer based, such as Tetris and Minesweeper. Tetris is a game where the goal is to stay alive as long as possible, by manoeuvring shapes into position and a single mistake can cost the game. This game is an example of a reflex puzzle, which primarily requires thinking multiple moves ahead, and planning each move in a short amount of time. While AI systems can be used on this puzzle, it involves more computer vision and machine learning than the types of puzzles we are interested in.

Minesweeper, on the other hand, is a very slow and methodical game compared to the high speed action of Tetris. Minesweeper involves avoiding bombs that have been planted in a minefield, prompting the player to be slow and thoughtful in their approach. Top players of this game remember patterns and what they mean, which is the type of game that can be solved with pattern recognition algorithms. This doesn’t make the game any less interesting however, as instead the tension is a slow rise while more of the board is being cleared. These games were pivotal in getting people interested in computers and video gaming. Minesweeper on its own was installed on most Windows operating systems at launch.

The difficulty of solving puzzle games like these can vary with the type of game and who is solving it. The most well-known example is Sudoku, which is often a 9x9 grid of squares that the player has to fill with numbers. Solving these puzzles can often be done by noticing patterns and using them to solve the problems faster in the next instance. These inherent traits of patterns being used to solve the puzzles allows them to be a great subject for the automatic solving by Artificial Intelligence applications. These puzzles can be a great way to test various algorithms and the areas in which they should be most effective. For example, one algorithm may be very adept at solving one type of game, whereas another could be far worse, and these differences can be analysed to find out why those algorithms are more or less effective in certain areas.

Many papers have been published using Artificial Intelligence algorithms to solve puzzle games, such as an Atari game solver, (Roibu, A.C., 2019.)( Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013) which prove that different types of games can be solved with the same algorithm. Puzzle games such as Sudoku are heavily focused on when automatic solvers are created. Sudoku is quite a simple game to simulate, as a basic grid is easy for a computer to understand, so the AI systems can integrate with them easily. Another reason is the popularity of Sudoku, and the fact that people without much experience with Artificial Intelligence can understand the result of an automatic solver for Sudoku puzzles.

There are multiple papers concerning puzzle game automatic solvers, such as one by Martyn Amos, Joseph White and Robin Houston that focused on Zen Puzzle Garden (ZPG) (Amos, M. and Coldridge, J., 2012). Their main focus was solving the puzzle with the Genetic Algorithm and the A\* Search Tree algorithm, which ended up being quite successful, proving that for the most part, the Genetic Algorithm was more efficient than general best first-search methods. There were further papers concerning ZPG concluding with proof that ZPG is an NP-Complete computer problem (Houston, R., White, J. and Amos, M., 2011), meaning that it is a very difficult game in which to develop automation. NP-Complete means that there are a very large number of solutions, so finding the most efficient one is difficult.

The project I am undertaking is related heavily to these papers, as they are directly connected to the PPEC (Pencil Puzzle Evolutionary Computation) project helmed by Martyn Amos, Huw Lloyd and Matthew Crossley. The project comprises of a system including five Artificial Intelligence algorithms (Ant Colony Optimisation, Simulated Annealing, Backtracking, Genetic Algorithms and Random Search). Each algorithm has already been implemented into the system, and my task is to integrate with them. There are four Japanese Pencil Puzzle game simulators (Sudoku, Nurikabe, Hashiwokakero and Zen Puzzle Garden) included in the project that can be solved by any of the implemented AI algorithms. Each type is an example of reinforcement learning, a type of algorithm that grows a solution over time with reinforcement from the previous attempt.

The first solver is Ant Colony Optimisation (Amos, M., Crossley, M. and Lloyd, H)( Lloyd, H. and Amos, M., 2019) which projects “Ants” that search through the problem and release “Pheromones” which become stronger or weaker depending on the results of the attempt, eventually finding the correct solution after the ants find the right combination. The second algorithm is Simulated Annealing (Trosset, M.W., 2001), which actually resembles the structure of metal as it is heated and cooled, when the heated metal atoms find the correct position and structure of their original position. Backtracking (Van Beek, P., 2006) is the third algorithm, which builds a solution is small ways one step at a time. The solutions that don’t satisfy the constraints are removed and replaced with better solutions. Next is the Genetic Algorithm (Davis, L., 1991), which simulates evolution in its basic form. Generations of solutions are formed, which are selected by their fitness in solving the problem, and the weaker species are killed off and replaced by the more successful ones until a solution is completed. Finally, there is Random Search (Zabinsky, Z.B., 2010), which operates like it sounds, by randomly searching for the answer at incredible speed.

My part in this project is to develop a simulator for the Japanese Pencil Puzzle Kakuro, which is similar to Sudoku in which a grid needs to be filled in with the numbers 1-9. The Kakuro grid has some reserved squares which dictate certain rows or columns which need to add up to a certain number. There cannot be the same number twice in the same axis, vertically or horizontally. The challenge comes with finding the right placement and value of the input numbers, where no rules are violated and all blank squares have been filled. The simulator I will write will give a selected algorithm a set of options and the algorithm will return its attempt at solving the problem. This simulator should be able to integrate with each type of algorithm and return a solution after a number of attempts.

After the simulator is created and runs well, I will attempt to survey which algorithm is the most efficient with the number of attempts to solve the puzzle, and how that changes with the size of the board. This should give me important information regarding which aspects of the individual solvers are more suited to the puzzle, and if I can understand the data properly, I may be able to make certain proclamations about the algorithms and their usefulness.

***Proposed Work***

The main focus of this project is to compare and contrast the effectiveness of each solver in the PPEC (Pencil Puzzle Evolutionary Computation) system when solving the Kakuro simulator that will be built. The Kakuro simulator itself needs to be able to perform these actions:

* Simulate a game board of Kakuro. The problem instance will be read by the simulator.
* Present the solver(s) with a list of available moves. The solver(s) will need to select a move from that list, which is then used to update the simulator’s internal state (that is, the representation of the current state of the problem instance. This process repeats until either the instance is solved, or the simulator reaches a state where there are no more available moves.
* Show the solution in a form that the user of the program can understand to make sure it is the correct solution. This will also be used in the test plan, in order to ensure that no incorrect solutions are being sent as output.

The primary language used by the system is C++, so the simulator will need to be written in either C++ or a similar language. The Ubuntu command line will be used to form the bridge between the programs and showcase the solution and data that was taken from the solving of the board. The command line will be the central hub that organises each file and manages message transfer between files.

Testing each algorithms’ performance and the rigidity of the Kakuro simulator will need an exhaustive test plan, which should investigate each aspect of the system in order to give context to the suitability of the end product. These tests should answer questions such as:

* Which algorithm has the least iterations on average for solving the Kakuro boards?
* Why is this the most effective algorithm?
* Which algorithm can still keep performing well at larger scopes of Kakuro boards?
* Why do they still perform well at larger scopes?

***Aims***

1. *To obtain a thorough performance analysis of a number of automated solver algorithms on a puzzle game.*
2. *To gain an understanding of the processes involved in managing a large-scale project such as this.*

***Objectives***

1. *Undertake a thorough review of the relevant literature to gain an in-depth understanding of the area.*
2. *Design appropriate representation and constraint propagation schemes for the Kakuro problem.*
3. *Design, implement and test a simulator for the Kakuro problem, suitable for integration into the PPEC platform.*
4. *Undertake experimental studies into the effectiveness of various solvers on the Kakuro problem.*

***Skills***

***Skills I have acquired:***

|  |  |
| --- | --- |
| The skills I have acquired | Where I have acquired these skills |
| The ability to use external software libraries in my research. | The KV5002 module (Computer Networks, Security and Operating Systems) used an external library of software that I have written code to interact with in my 2nd year. |
| The ability to use and acquire new programming knowledge. | Many previous modules, namely KF5012 (Software Engineering Project), KF5002 (Web Programming), KF4006 (Procedural Programming), KF4007 (Object Oriented Programming) and KV5002 (Computer Networks, Security and Operating Systems). |
| Knowledge of the operation and running of the Ubuntu Command Line System. | The KV5002 module (Computer Networks, Security and Operating Systems) necessitated the use of the Ubuntu Command Line system to interact with the external software. |
| Experience with using Virtual Machines to simulate computer Operating Systems. | Self-research and the KV5002 module (Computer Networks, Security and Operating Systems). |
| The ability to thoroughly and fairly test my own software. | The KF5012 module (Software Engineering Practice) and thoroughly testing the architecture was a requirement. |

***Skills I hope to acquire throughout the duration of this project:***

|  |  |
| --- | --- |
| The skills I hope to acquire | How I will acquire these skills |
| Extensive knowledge of C++ architecture. | Using online resources to improve my knowledge (L, S. and Frank, M. 2019), and loaning C++ tutorial literature from the university library. |
| Deeper knowledge of AI systems. | The two modules KF6007 (Artificial Intelligence and Robotics) and KF6052 (Machine Learning and Computer Vision) should give me a more in-depth knowledge and experience using Artificial Intelligence systems. |

***Sources of Information***

***References***

Jorgensen, A.H., 2009. Context and driving forces in the development of the early computer game Nimbi. *IEEE Annals of the History of Computing*, *31*(3), pp.44-53.

Roibu, A.C., 2019. Design of Artificial Intelligence Agents for Games using Deep Reinforcement Learning. *arXiv preprint arXiv:1905.04127*.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Amos, M. and Coldridge, J., 2012. A genetic algorithm for the Zen Puzzle Garden game. *Natural Computing*, *11*(3), pp.353-359.

Houston, R., White, J. and Amos, M., 2011. Zen puzzle garden is NP-complete. *arXiv preprint arXiv:1106.2104*.

Amos, M., Crossley, M. and LLoyd, H., Solving Nurikabe with Ant Colony Optimization (Extended version).

Lloyd, H. and Amos, M., 2019. Solving Sudoku with Ant Colony Optimization. *IEEE Transactions on Games*.

Trosset, M.W., 2001. What is simulated annealing?. Optimization and Engineering, 2(2), pp.201-213.

Van Beek, P., 2006. Backtracking search algorithms. In Foundations of artificial intelligence (Vol. 2, pp. 85-134). *Elsevier.*

Davis, L., 1991. Handbook of genetic algorithms.

***Bibliography***

L, S. and Frank, M. (2019). *Learn C++ | Codecademy*. [online] Codecademy. Available at: https://www.codecademy.com/learn/learn-c-plus-plus [Accessed 18 Nov. 2019].

***Resources***

***Software:***

* VMWare (A system for managing Virtual Machines): This will be used to run the simulator. This is used in order to produce fair and reliable tests. This software is free and available to the public.
* Microsoft Word 2017 and Microsoft Word 2010: Word 2017 is the default for Northumbria University computers, so it will be used frequently for report writing. Word 2010 is the version installed on the personal computer I use to edit reports off-campus.
* Ubuntu Terminal and Command Line v18: The command line will be used to compile and run the system to be created and analysed.
* GNU Compiler v9: This will be used to compile the simulator throughout the creation and testing process.

***Hardware:***

* Northumbria University On-Campus Computers: These are used to write reports and run the VM software.
* Hard-Drive (300GB): A Hard-Drive to store the Program and Virtual Machine Software for writing and testing the simulator. I have already loaned a suitable Hard-Drive from the loans office.
* USB-Drive: Used for storing reports when using Northumbria University Computers.

***Structure and Contents***

***Investigative Project Structure:***

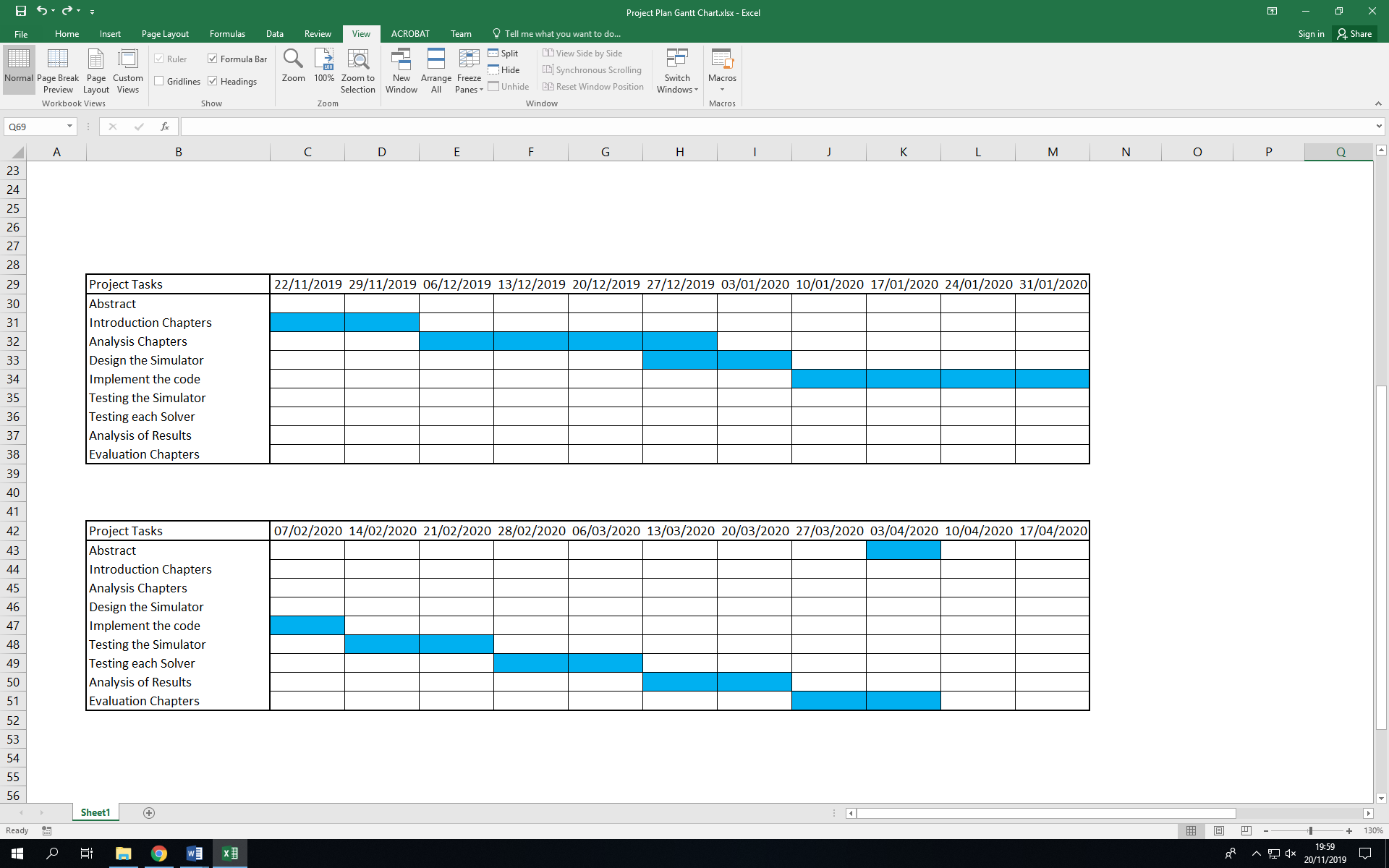
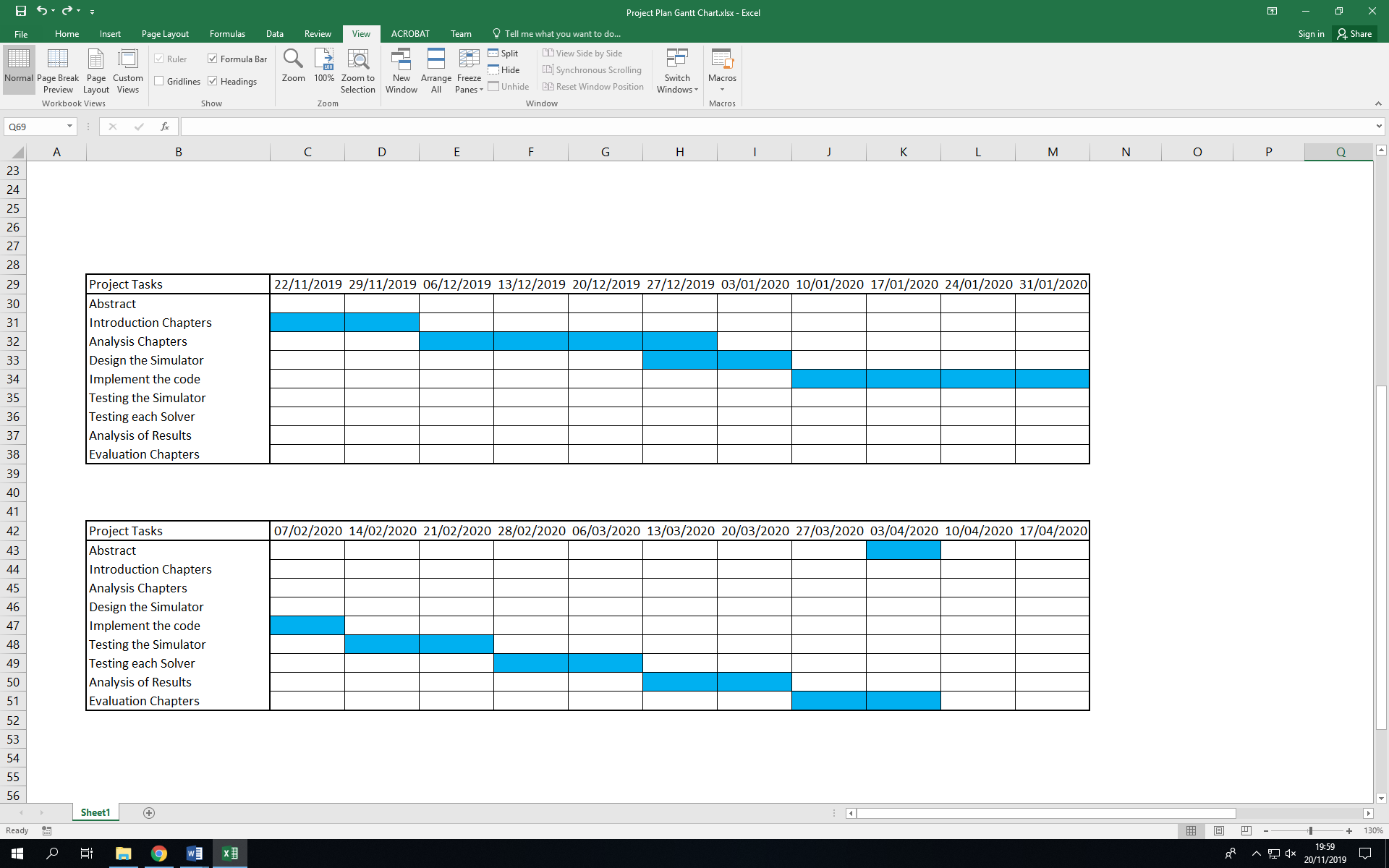
* Introduction
  + Title
  + Abstract
    - Detailing the full contents of the report and the end conclusion.
  + Contents Page
    - Library of each section and where to find each with page numbers.
  + Introduction
    - Brief description of what will happen in the report, and the structure of the contents.
* Analysis
  + Introduction to the Kakuro Problem
    - Details the game and the reason it is a useful project to undertake.
  + Literature review
    - Objective 1
    - Review of the relevant literature surrounding the industry, specifically surrounding each solver and their attributes.
  + The PPEC Software
    - Full breakdown of the history of the PPEC software and its intended use in the future.
  + The Solvers
    - Detailing how each solver works in the PPEC Software.
  + The Simulator
    - Details the simulator I intend to build and what it I mean it to be able to do.
* Synthesis
  + Designing the Simulator
    - Objective 2 & 3
    - This section will cover the design of the simulator, any pseudo code to be presented and the overall structure of the program.
  + Implementing the code
    - Objective 3
    - This section covers how I implement the code, any subjects of interest throughout this process and any difficulties I encounter.
  + Testing the simulator
    - Objective 3
    - Covers using the test plan to conduct thorough tests of the simulator in order to ensure it is as bug free as possible. Should detail any errors discovered in this process.
  + Testing each solver
    - Objective 4
    - Details using the second test plan to conduct tests of the effectiveness of each solver.
  + Analysis of Results
    - Objective 4
    - Detailed analysis of the results of the tests, and any knowledge that can be deduced using this information.
* Evaluation
  + Evaluation
    - Reviews the project as a whole, and any relevant aspects of the project that were either performed well or needed changing.
  + Conclusion
    - Details each important revelation of the project and any final conclusions that were made.
  + Recommendations
    - Explains the ways in which my work can be improved upon and any future related studies I could participate in.

***Marking Scheme***

|  |  |
| --- | --- |
| Report and Practical work | (90% Overall) |
| - Abstract and Introduction | - 5% |
| - Analysis | - 20% |
| - Synthesis: discussion of methods and results | - 20% |
| - Synthesis: quality of practical work | - 30% |
| - Evaluation and Conclusions | - 20% |
| - Presentation | - 5% |
| Viva | (10% Overall) |

***Quality of Practical Work***

* Compliance with any relevant ethical and safety guidelines.
* I will use a Virtual Machine where possible to perform tests upon the simulator in order to preserve a fair testing environment. This will enable me to ensure that the tests are not being affected by external stimuli and can be relied upon.
* I will provide public access to the code that I develop for the simulator, in order to promote clarity in the execution of my test process.
* Appropriate research methodologies will be used throughout this project. These will mainly be quantitative methodologies.

***Project Plan***

Appendices 2.1: Results for instances of 3x3 and 5x5 with randomised solver

Appendices 2.2 Results for instances of 7x7 and 10x10 with randomised solver

Appendices 2.3 Results for instances of 3x3 and 5x5 with Genetic Algorithm

Appendices 2.4 Results for instances of 7x7 and 10x10 with Genetic Algorithm

Appendices 2.5 Results for instances of 3x3 and 5x5 with both solvers

Appendices 2.6 Results for instances of 7x7 and 10x10 with both solvers

Appendices 3: Student Meeting Logbook

|  |
| --- |
| **Student name and ID: David William Jackson** |
| **Academic year: 4th** |
| **Programme: Computer Science with Artificial Intelligence** |
| **Project title: Comparative Analysis of AI-Based Solvers for the NP-Complete Problem Kakuro** |
| **Supervisor: Martyn Amos** |
| **Supervisor email and room number: martyn.amos@northumbria.ac.uk ELB1.22** |
| **Second marker: Neil Eliot** |
| **Second marker email and room number:** |

[Table of Contents (Use Word NavigatION PANE for direct access to each) 1](#_Toc41517514)

[Semester 1 Week 1: 2](#_Toc41517515)

[Semester 1 Week 2: 2](#_Toc41517516)

[Semester 1 Week 3: 3](#_Toc41517517)

[Semester 1 Week 4: 3](#_Toc41517518)

[Semester 1 Week 5: 3](#_Toc41517519)

[Semester 1 Week 6: 4](#_Toc41517520)

[Semester 1 Week 7: 4](#_Toc41517521)

[Semester 1 Week 8: 4](#_Toc41517522)

[Semester 1 Week 9: 5](#_Toc41517523)

[Semester 1 Week 10: 5](#_Toc41517524)

[Semester 1 Week 11: 5](#_Toc41517525)

[Semester 1 Week 12: 6](#_Toc41517526)

[Semester 2 Week 1: 6](#_Toc41517527)

[Semester 2 Week 2: 6](#_Toc41517528)

[Semester 2 Week 3: 6](#_Toc41517529)

[Semester 2 Week 4: 7](#_Toc41517530)

[Semester 2 Week 5: 7](#_Toc41517531)

[Semester 2 Week 6: 7](#_Toc41517532)

[Semester 2 Week 7: 8](#_Toc41517533)

[Semester 2 Week 8: 8](#_Toc41517534)

[Semester 2 Week 9: 8](#_Toc41517535)

[Semester 2 Week 10: 8](#_Toc41517536)

[Semester 2 Week 11: 9](#_Toc41517537)

[Student Checklist 10](#_Toc41517538)

Semester 1 Week 1:

|  |  |  |
| --- | --- | --- |
| **Date and time of meeting: Monday 30th September** | **As scheduled:** Yes | |
| **Brief description of work done since last meeting: First meeting** | | |
| **Number of hours spent on project since last meeting: First Meeting** | |
| **Questions/items to discuss at meeting (agenda): Proposed Project and main goals** | |
| **Agreed tasks for next meeting:** | **Learn more C++ and familiarise myself with the material** |
| **Documents discussed /any other issues:** | **Multiple papers were discussed that included recent findings on the subject.** |
| **Date and time of next meeting: 14th October** | |

Semester 1 Week 2:

|  |  |  |
| --- | --- | --- |
| **Date and time of meeting: No Meeting was scheduled** | **As scheduled: NA** | |
| **Brief description of work done since last meeting: Improved knowledge of C++ and the Algorithms to be used.** | | |
| **Number of hours spent on project since last meeting: 6** | |
| **Questions/items to discuss at meeting (agenda):** | |
| **Agreed tasks for next meeting:** |  |
| **Documents discussed /any other issues:** |  |
| **Date and time of next meeting:** | |

Semester 1 Week 3:

|  |  |  |
| --- | --- | --- |
| **Date and time of session: Monday 14th October 12.30pm** | **As scheduled:** Yes | |
| **Brief description of work done since last meeting: Familiarisation of the C++ language and the material surrounding the project, including papers** | | |
| **Number of hours spent on project since last meeting: 7** | |
| **Questions/items to discuss at meeting (agenda): Use of Software and overall scope of project** | |
| **Agreed tasks for next meeting: Read the full material for the subject across four papers and familiarisation of** |  |
| **Documents discussed /any other issues: PID was completed and signed. TOR was discussed and planned.** |  |
| **Date and time of next session: Monday 21st October 12.30** | |

Semester 1 Week 4:

|  |  |  |
| --- | --- | --- |
| **Date and time of session:24/10/2019 2PM** | **As scheduled:** Yes | |
| **Brief description of work done since last meeting: Research on past papers and familiarity with the software** | | |
| **Number of hours spent on project since last meeting: 14** | |
| **Questions/items to discuss at meeting (agenda): Utilising the software, TOR discussion and project details** | |
| **Agreed tasks for next meeting: Make sure code works, and make a first draft of TOR** |  |
| **Documents discussed /any other issues: TOR background section discussed, and difficulties with code compilation fixed.** |  |
| **Date and time of next session: 31st October 2pm** | |

Semester 1 Week 5:

|  |  |  |
| --- | --- | --- |
| **Date and time of session: 31/10/2019 2PM** | **As scheduled:** Yes | |
| **Brief description of work done since last meeting: Code compilation errors fixed, and most of TOR completed** | | |
| **Number of hours spent on project since last meeting: 10** | |
| **Questions/items to discuss at meeting (agenda): Edits to TOR, specifically the skills section.** | |
| **Agreed tasks for next meeting: Work on TOR and figure out the specifics for the code later.** |  |
| **Documents discussed /any other issues: Background section discussed, beginnings of code established.** |  |
| **Date and time of next session: 07/11/2019 2PM** | |

Semester 1 Week 6:

|  |  |  |
| --- | --- | --- |
| **Date and time of session: 07/11/2019 2PM** | **As scheduled:** Yes | |
| **Brief description of work done since last meeting: Background more established, TOR fleshed out, more plans on future code. Research on similar projects done.** | | |
| **Number of hours spent on project since last meeting: 8** | |
| **Questions/items to discuss at meeting (agenda): Background section (past papers), Names for the project, Ethics forms.** | |
| **Agreed tasks for next meeting: Rewrite Background section** |  |
| **Documents discussed /any other issues: TOR** |  |
| **Date and time of next session:14/11/2019** | |

Semester 1 Week 7:

|  |  |  |
| --- | --- | --- |
| **Date and time of session: 14/11/2019** | **As scheduled:** Yes | |
| **Brief description of work done since last meeting: TOR Background rewritten, Proposed work mostly completed,** | | |
| **Number of hours spent on project since last meeting: 7** | |
| **Questions/items to discuss at meeting (agenda): Proposed work for TOR, Name of project and Ethics forms.** | |
| **Agreed tasks for next meeting: Schedule TOR review and work on analysis chapters** |  |
| **Documents discussed /any other issues: TOR and Ethics Form.** |  |
| **Date and time of next session: 21/11/19 12.30PM** | |

Semester 1 Week 8:

|  |  |  |
| --- | --- | --- |
| **Date and time of session:21/11/19** | **As scheduled:** Yes | |
| **Brief description of work done since last meeting: TOR Completed, Ethics Form Draft completed, and TOR Review Scheduled** | | |
| **Number of hours spent on project since last meeting: 14** | |
| **Questions/items to discuss at meeting (agenda): TOR Review.** | |
| **Agreed tasks for next meeting: Make changes to TOR and prepare for MMU collaboration meeting** |  |
| **Documents discussed /any other issues: Review of TOR was completed.** |  |
| **Date and time of next session: 27/11/2019** | |

Semester 1 Week 9:

|  |  |  |
| --- | --- | --- |
| **Date and time of session: 27/11/2019** | **As scheduled:** Yes | |
| **Brief description of work done since last meeting: TOR changes made and MMU collaboration presentation created** | | |
| **Number of hours spent on project since last meeting: 5** | |
| **Questions/items to discuss at meeting (agenda): MMU Collaboration presentation and program changes to be made.** | |
| **Agreed tasks for next meeting: Work on knowledge of the program and think about the logistics of the coding to be done.** |  |
| **Documents discussed /any other issues: None.** |  |
| **Date and time of next session:19/12/2019** | |

Semester 1 Week 10:

|  |  |  |
| --- | --- | --- |
| **Date and time of session: No session planned.** | **As scheduled: N/A** | |
| **Brief description of work done since last meeting:** | | |
| **Number of hours spent on project since last meeting:** | |
| **Questions/items to discuss at meeting (agenda):** | |
| **Agreed tasks for next meeting:** |  |
| **Documents discussed /any other issues:** |  |
| **Date and time of next session:** | |

Semester 1 Week 11:

|  |  |  |
| --- | --- | --- |
| **Date and time of session: No session planned.** | **As scheduled: N/A** | |
| **Brief description of work done since last meeting:** | | |
| **Number of hours spent on project since last meeting:** | |
| **Questions/items to discuss at meeting (agenda):** | |
| **Agreed tasks for next meeting:** |  |
| **Documents discussed /any other issues:** |  |
| **Date and time of next session:** | |

Semester 1 Week 12:

|  |  |  |
| --- | --- | --- |
| **Date and time of session: 19/12/2019** | **As scheduled:** Yes | |
| **Brief description of work done since last meeting: First Analysis chapters drafted and codebase considered.** | | |
| **Number of hours spent on project since last meeting: 15** | |
| **Questions/items to discuss at meeting (agenda): Coding language choice, draft chapters review and logistics of Literature review.** | |
| **Agreed tasks for next meeting:** |  |
| **Documents discussed /any other issues:** |  |
| **Date and time of next session:** | |

Semester 2 Week 1:

|  |  |  |
| --- | --- | --- |
| **Date and time of session:23/1/2020 1PM** | **As scheduled:** Yes | |
| **Brief description of work done since last meeting: Analysis chapter drafted, codebase started** | | |
| **Number of hours spent on project since last meeting: 15** | |
| **Questions/items to discuss at meeting (agenda): Analysis Chapters, Code Beginnings** | |
| **Agreed tasks for next meeting: Start on coding the simulator** |  |
| **Documents discussed /any other issues: Analysis chapters considered** |  |
| **Date and time of next session: 05/03/2020** | |

Semester 2 Week 2:

|  |  |  |
| --- | --- | --- |
| **Date and time of session: No Meeting Scheduled** | **As scheduled:** N/A | |
| **Brief description of work done since last meeting:** | | |
| **Number of hours spent on project since last meeting:** | |
| **Questions/items to discuss at meeting (agenda):** | |
| **Agreed tasks for next meeting:** |  |
| **Documents discussed /any other issues:** |  |
| **Date and time of next session:** | |

Semester 2 Week 3:

|  |  |  |
| --- | --- | --- |
| **Date and time of session: No meeting Scheduled** | **As scheduled:** N/A | |
| **Brief description of work done since last meeting:** | | |
| **Number of hours spent on project since last meeting:** | |
| **Questions/items to discuss at meeting (agenda):** | |
| **Agreed tasks for next meeting:** |  |
| **Documents discussed /any other issues:** |  |
| **Date and time of next session:** | |

Semester 2 Week 4:

|  |  |  |
| --- | --- | --- |
| **Date and time of session: 05/2/2020 1:30PM** | **As scheduled:** Yes/No | |
| **Brief description of work done since last meeting: Code Solidified, Board read in** | | |
| **Number of hours spent on project since last meeting: 12** | |
| **Questions/items to discuss at meeting (agenda): Code and next steps for it** | |
| **Agreed tasks for next meeting: Work on the Array system to hold the board** |  |
| **Documents discussed /any other issues:** |  |
| **Date and time of next session: 19/2/2020** | |

Semester 2 Week 5:

|  |  |  |
| --- | --- | --- |
| **Date and time of session: 19/2/2020** | **As scheduled:** Yes/No | |
| **Brief description of work done since last meeting: Array can print out board as basic map** | | |
| **Number of hours spent on project since last meeting: 10** | |
| **Questions/items to discuss at meeting (agenda): Vectors, bitmaps, and testing** | |
| **Agreed tasks for next meeting: Convert arrays to vectors and allow printing out of board, then allow answer string to be read and calculated.** |  |
| **Documents discussed /any other issues:** |  |
| **Date and time of next session:18/03/2020** | |

Semester 2 Week 6:

|  |  |  |
| --- | --- | --- |
| **Date and time of session: No Meeting Scheduled** | **As scheduled:** N/A | |
| **Brief description of work done since last meeting:** | | |
| **Number of hours spent on project since last meeting:** | |
| **Questions/items to discuss at meeting (agenda):** | |
| **Agreed tasks for next meeting:** |  |
| **Documents discussed /any other issues:** |  |
| **Date and time of next session:** | |

Semester 2 Week 7:

|  |  |  |
| --- | --- | --- |
| **Date and time of session: No Meeting Scheduled** | **As scheduled:** N/A | |
| **Brief description of work done since last meeting:** | | |
| **Number of hours spent on project since last meeting:** | |
| **Questions/items to discuss at meeting (agenda):** | |
| **Agreed tasks for next meeting:** |  |
| **Documents discussed /any other issues:** |  |
| **Date and time of next session:** | |

Semester 2 Week 8:

|  |  |  |
| --- | --- | --- |
| **Date and time of session: 18/03/2020 Meeting could not be held due to lockdown restrictions** | **As scheduled:** No | |
| **Brief description of work done since last meeting:** | | |
| **Number of hours spent on project since last meeting:** | |
| **Questions/items to discuss at meeting (agenda):** | |
| **Agreed tasks for next meeting:** |  |
| **Documents discussed /any other issues:** |  |
| **Date and time of next session:** | |

Semester 2 Week 9:

|  |  |  |
| --- | --- | --- |
| **Date and time of session: No Meeting Scheduled** | **As scheduled:** Yes/No | |
| **Brief description of work done since last meeting:** | | |
| **Number of hours spent on project since last meeting:** | |
| **Questions/items to discuss at meeting (agenda):** | |
| **Agreed tasks for next meeting:** |  |
| **Documents discussed /any other issues:** |  |
| **Date and time of next session:** | |

Semester 2 Week 10:

|  |  |  |
| --- | --- | --- |
| **Date and time of session: No Meeting Scheduled** | **As scheduled:** Yes/No | |
| **Brief description of work done since last meeting:** | | |
| **Number of hours spent on project since last meeting:** | |
| **Questions/items to discuss at meeting (agenda):** | |
| **Agreed tasks for next meeting:** |  |
| **Documents discussed /any other issues:** |  |
| **Date and time of next session:** | |

Semester 2 Week 11:

|  |  |  |
| --- | --- | --- |
| **Date and time of session: No Meeting Scheduled** | **As scheduled:** Yes/No | |
| **Brief description of work done since last meeting:** | | |
| **Number of hours spent on project since last meeting:** | |
| **Questions/items to discuss at meeting (agenda):** | |
| **Agreed tasks for next meeting:** |  |
| **Documents discussed /any other issues:** |  |
| **Date and time of next session:** | |

Student Checklist

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **Semester 1: Actions** | **Due Date** | **Date Completed** | | PID submitted to Blackboard | Week 3, Monday by 5pm  (OR as soon as possible afterwards if supervisor has been allocated) | 14th October 2019 | | Draft TOR and associated documents (e.g. draft risk assessment) emailed to supervisor and second marker; complete online ethics form | Weeks 4-6 | 20th November 2019 | | TOR Review arranged | Up to Weeks 5-6 | 13th November 2019 | | TOR Review held | Weeks 5-6 (or earlier) | 21st November 2019 | | Final revised TOR submitted to Blackboard | Week 8, Friday by 5pm (at the latest) | 22nd November 2019 | | Final online ethics form and risk assessment form (if required) completed | Week 8, Friday by 5pm (at the latest) | 12th May 2020 | | Draft chapters submitted to Supervisor | Semester 1 Week 10, Friday (or earlier) | 24th May 2020 | |

|  |  |  |
| --- | --- | --- |
| **Semester 2: Actions** | **Due Date** | **Date Completed** |
| Submission of TWO copies of the project report AND the CD/DVD/USB to Student Central, Library Building | Semester 2, Week 11, Thursday 30th April by 4pm | N/A |
| Submission of the project report to the Turnitin submission link in the Assessment area on Blackboard | ---Same as above--- |  |
| Viva arranged | *Arranged around time of submission i.e. during Week 11 Semester 2* | 14th May 2020 |
| Viva held | *During Semester 2, Weeks 12-15. Note; vivas to be completed by Friday 22nd May.* | To be completed : 9th June 2020 |
| Evidence file submitted | *At the viva (Semester 2, Weeks 12-14)* | To be completed:  9th June 2020 |