Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Håvard Melheim

Generating rationally solvable instances of NP-hard logic puzzles

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
June 2022

NTNU
Norwegian University of
Science and Technology

Håvard Melheim

Generating rationally solvable instances of NP-hard logic puzzles

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Task

Title: Generating rationally solvable instances of NP-hard logic puzzles

Student: Håvard Melheim

Supervisor: Sverre Hendseth

Description

On a simple level, Sudoku, as well as other similar logic puzzles provide an entertaining pastime. More interesting is the fact that though these puzzles in their general forms are NP-Complete, instances can be made to be solvable without the need for guesswork. In this project, the candidate will investigate such logic puzzles, and techniques for their creation. A puzzle generator will also be implemented, as well as an app for puzzle solving.

- Choose a logic puzzle to focus on, and explain its rules.
- Investigate and explain computational complexity theory based on the selected puzzle.
- Implement a puzzle generator, capable of generating instances of the selected puzzle that are solvable to a human.
- Implement an app to solve the generated puzzles.
- Discuss the choices made when implementing the app, such as frameworks and languages used.

Abstract

Nurikabe, a pen-and-paper logic puzzle named for a spirit from japanese folklore, is an entertaining pastime that benefits the brain. Like many other similar puzzles it is also NP-Hard in its general form. In this project, the computational complexity of Nurikabe has been investigated, both in its general form, and in its more specific, solvable form, which is how it is commonly presented in newspapers and the like. Furthermore, the possibility of developing a generator algorithm capable of generating solvable instances of Nurikabe has also been investigated and discussed.

To perform these investigations, a set of techniques that can be used to solve Nurikabe boards has been established. Based on these, a prototype Nurikabe generator has been developed, using the Haskell programming language. A web app that simplifies the use of this generator has also been developed, as well as an app that allows users to solve Nurikabe puzzles. These were both developed using the C# programming language.

The developed generator has been proven to be able to generate puzzles of non-trivial difficulty, that are furthermore guaranteed to be uniquely solvable to humans, using the established techniques.

Likewise, the developed app has been proven to allow solution of Nurikabe puzzles of any size, in an intuitive and user-friendly way.

To conclude, implementing a generator capable of generating Nurikabe puzzles solvable to a human solver is most definitely possible. The usability of such a generator is however much dependent on the size of the boards being generated, as the computational complexity of the Nurikabe problem makes keeping the generator performant a challenge.

Sammendrag

Nurikabe, en form for logisk hjerntrim oppkalt etter en ånd fra japansk folklore, er et underholdende og enagsjerende tidsfordriv. I likhet med mange lignende spill er det dessuten NP-hardt i sin generelle form. I denne oppgaven har kompleksiteten til Nurikabe blitt undersøkt, både i dets generelle form og i dets mer spesifikke, løsbare form som ofte er brukt i aviser og lignende. Videre har mulgheten for å utvikle en generasjonsalgoritme, i stand til å generere løsbare Nurikabe-brett, blitt undersøkt og diskutert.

For å utføre disse undersøkelsene ble et sett med teknikker, som kan brukes til å løse Nurikabe-brett, bestemt. Basert på disse har en prototype Nurikabe generator blitt utviklet, ved hjelp av programmeringsspråket Haskell. En web-app som forenkler bruken av denne generatoren har også blitt utviklet, i tillegg til en app som lar brukere løse Nurikabe-brett. Begge disse er utviklet ved hjelp av programmeringsspråket C#.

Den utviklede generatoren har vist seg i stand til å generere brett av ikke-triviell vanskelghetsgrad, som dessuten er garantert å være unikt løsbare for mennesker, ved hjelp av de etablerte teknikkene.

Den utviklede appen har videre vist seg i stand til å la brukere løse Nurikabe-brett av enhver størrelse på en intuitiv og brukervennlig måte.

For å konkludere har altså arbeidet med dette prosjektet vist at generasjon av Nurikabe-brett som er løsbare for mennesker uten tvil er mulig. Brukbarheten til en slik generator avhenger likevel i stor grad av størrelsen til brettet som genereres, ettersom kompleksiteten til Nurikabe-problemet gjør det vanskelig å holde generatoren effektiv.

Contents

Acronyms

ADT Abstract Data Type. 46, 47

API Application Programming Interface. 2, 3, 17, 18, 54, 56–58, 65–67, 74, 76

ASCII American Standard Code for Information Interchange. 7

ASP Active Server Pages. 2, 16–18, 54, 55

AWS Amazon Web Services. 56

CLR Common Language Runtime. 17

DLL Dynamic Link Library. 55, 56, 76, 92

DTO Data Transfer Object. 20, 57, 90, 94

GHC Glasgow Haskell Compiler. 55

HTML HyperText Markup Language. 20

IO Input Output. 27

JSON JavaScript Object Notation. 66

LINQ Language Integrated Query. 18

MVVM Model-View-ViewModel. 19, 20, 56

NP Nondeterministic polynomial time. xi, 1, 6, 9, 10, 13–16, 42, 64, 65, 79, 83, 88

NP-C NP-complete. 1, 6, 9, 10, 83

NP-H NP-hard. 6, 10

OS Operating System. 17, 94

P Polynomial time. 6, 8, 9

UI User Interface. 18–20, 22, 66

URL Uniform Resource Locator. 67

XAML eXtensible Application Markup Language. 18, 19, 81

XML eXtensible Markup Language. 18

Chapter 1

Introduction

Sudoku is perhaps the most famous pen-and-paper logic puzzle in the world, and its rules for filling out a board of (normally) 9×9 cells with the numbers 1-9 will be familiar to many. However, this does not mean that Sudoku is the only pen-and-paper logic puzzle that exists; far from it! Numerous puzzles exist that can be solved using only a pen, a piece of paper, and the brain.

An example of another puzzle of this sort, is Nurikabe, named for a japanese spirit that manifests itself as an invisible wall that impedes travellers in the night. The puzzle Nurikabe, which also goes by the names *Cell Structure* and *Islands in the Stream*, consists of filling out a rectangular board of cells, much like Sudoku. In Nurikabe however, the cells are not marked by the numbers 1-9, but rather by either black or white color, indicating whether the cell is water or an island respectively. See figure 1.1 for an example solution of a simple Nurikabe board. The rules of the game will be explained in more detail in chapter 2 of this report.

Interesting about Sudoku, Nurikabe, and many other puzzles of the same sort, is that their solutions are in general NP-complete (NP-C)[1]. In spite of this, it is perfectly possible to generate instances of these puzzles that are solvable to a human, using nothing but rational inference. It is these instances that have been investigated through this project.

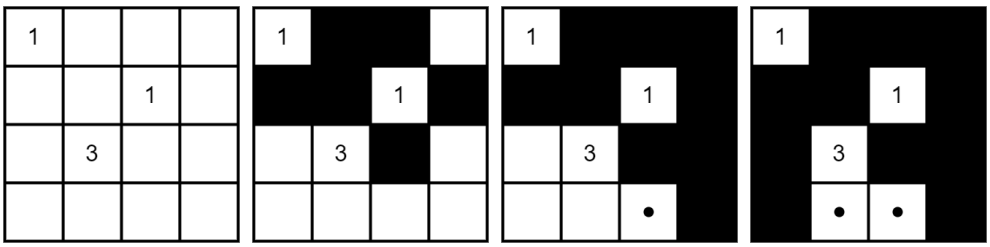


Figure 1.1: An example of a simple Nurikabe board being solved, from left to right, according to the rules and techniques that will be explained below in chapters 2 and 3.

1.1 About the task

The main purpose of this project was for the candidate to develop and hone their skills as a software developer. Being exposed to new programming languages and frameworks, and getting a deep dive into already familiar ones through a larger-scale project, enabled this. It also served to expand the candidate's perspectives about their own role in future projects, as well as their strengths and weaknesses as a developer. As a framework to facilitate this growth, the development of a Nurikabe puzzle, generator capable of procedurally generating puzzles solvable by humans, was chosen. The reasons for this were many; for one, the candidate had an interest in such logic puzzles, and was therefore motivated by the opportunity to work with them. Additionally, the implementation of such a generator, and associated game app, enabled the candidate to utilize multiple skills and disciplines, ranging from algorithm development, to the development of apps and even web APIs.

As this project would result in an app to solve Nurikabe puzzles, a framework for developing such an app had to be chosen and used. Though many mobile app frameworks exist, it was in this project decided to use Microsoft's Xamarin framework to develop the app. This was chosen for several reasons; for one, it is an open-source, completely free framework, meaning there would be no financial obstacles or issues with copyright to deal with. A second reason was that the framework is inherently cross-platform, meaning it could be used to develop apps for both Android and iOS. Taking only these two properties into account, several frameworks were still equally well suited to this project however. In addition to Xamarin, Facebook's React Native, building on Javascript, and Google's Flutter, using the Dart programming language, as well as many others fulfill these criteria. The reason why Xamarin was chosen from among these, was that it would allow for the use of the C# programming language, something the candidate had interest in.

It was also decided to implement the puzzle generator in Haskell, based on suggestions from the supervisor, as well as promising initial results. Pattern recognition, as well as powerful list comprehension and concise logic seemed well suited to the needs of the generator. Also, the functional, recursive nature of the language seemed well aligned with the step-by-step nature of the puzzle solver the generator would be based on. Furthermore, Haskell, as well as functional programming in general, was new to the candidate, and so provided an excellent option to expand their competence and learn a new way of thinking.

This generator was furthermore decided to be contained in a web app, built using the C# programming language, but this time using the ASP.NET Core framework. This would allow the app to access the generator without having to necessarily contain it. Doing this would enable the candidate to delve into the world of web APIs, and also decouple the generator from the solver app in a natural way.

1.2 The structure of this report

This report is split into three main parts. The first of these, found in chapter 2, covers the necessary theory and background information needed to understand the Nurikabe logic puzzle and its complexities. This part also covers the information and theory needed to develop the generator, such as background information about technologies, frameworks and languages used.

The second part, found in chapter 3 details the actual development process; from the development of the generator algorithm, to the development of a Nurikabe app for puzzle solving, and a web API for interfacing between the two. Also included in this chapter is a list of Nurikabe solution techniques, used by the generator to verify the solvability of generated puzzles. This section could arguably also be placed in chapter 2 as a part of the background information about Nurikabe. But as these techniques were formulated by the candidate for this project, and based on their personal experience solving Nurikabe boards, this section was more appropriately placed in chapter 3 as a part of the work performed to develop the generator.

The last part is found in chapters 4 and 5. Chapter 4 shows the results of the project, including both the app and web API, as well as the generator itself. Chapter 5 discusses the more abstract results of the project, such as the candidate's development as a software developer.

Chapter 2

Background and Theory

In this chapter, theory and other background information relevant to this project will be presented and discussed. This includes theory about computational complexity and complexity classes, as well as more practically oriented background information about the technologies used throughout the project. Additionally, the Nurikabe logic puzzle will be more thoroughly examined, by looking at its definitions, rules and common practices, as well as an in-depth analysis of its computational complexity.

2.1 Computational complexity

According to the Encyclopaedia Britannica, *computational complexity* is "a measure of the amount of computing resources (time and space) that a particular algorithm consumes when it runs"[2]. In other words, it is a measure of how much time and memory a given computer would need to solve a given problem. This measure can be used to classify problems into several classes.

In this section, some of these classes will be explained. The classifications presented in this chapter will also form the basis for the investigation into the computational complexity of Nurikabe, as will be performed below, in section 2.2.2. See figure 2.1 for an illustration of how the complexity classes relevant to this report relate to each other.

2.1.1 Problems

Before delving into the different classes of problems, a formal definition of what exactly a *problem* is needs to be established. According to Cormen et al, an *abstract problem Q* can be defined as a binary relation on a set I of *problem instances* and a set S of *problem solutions*[3, p. 1054]. As an example, for a game of Sudoku, a problem instance is a set of given numbers along with their positions in the grid. A problem solution would then be a complete set of numbers and corresponding positions such that all positions in the grid are assigned a number according to the rules of Sudoku. In general, there could be multiple solutions for a single

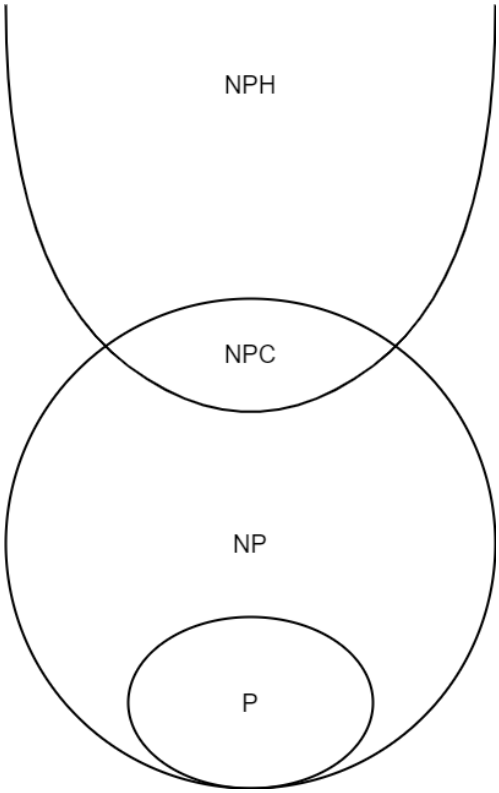


Figure 2.1: Venn diagram illustrating the relationships between the different classes of problems. Note that all problems in P are necessarily also in NP. The same also holds true for problems in NP-C which are the inersection of problems in NP and NP-H.

problem instance, or none at all, however in the case of Sudoku, the rules ensure that there is always exactly one solution to a given problem. The actual problem itself in this example, is then the relation that associates each problem instance with the corresponding solutions.

These abstract problems can be divided into so-called *decision problems* and *optimization problems*, where decision problems are problems having a simple yes/no solution. That means, decision problems are a special case of the abstract problems above that maps the set of instances I to the solution set $S = \{0, 1\}$. For example, in the case of Sudoku, a decision problem could be to determine whether or not a given board has a valid solution. Optimization problems on the other hand, are problems where some value has to be minimized or maximized, i.e. the solution set S is equal to some minimum or maximum value, or the set of parameters producing said minimum or maximum value. An important detail about optimization problems is that they can often be recast as decision problems that are no harder to compute[3, p.1054].

Another special case of the abstract problem defined above, is the so called *concrete problem*. A concrete problem is defined by Cormen et al as a problem whose instance set is the set of binary strings[3, p. 1055]. That is, the instance set has somehow been encoded to the set of binary strings, $\{0, 1\}^*$, where $\{0, 1\}^*$ is the set of all strings composed of symbols from the set $\{0, 1\}$. For example, an abstract problem taking a single natural number as its input can be easily converted to a concrete problem by encoding the natural numbers as binary strings. That is, the instance set is encoded from $\{0, 1, 2, 3, 4, ...\}$ to $\{0, 1, 10, 11, 100, ...\}$. Likewise, the ASCII code encodes alphabetic characters to binary strings and back.

Formal-language theory

To better express the relationship between decision problems and the algorithms that solve them, formal-language theory can be used. According to this theory, an *alphabet* Σ is a finite set of symbols. A *language* L over Σ is then any set of strings made up of symbols from Σ[3, p. 1057]. As an approachable example of this, the Latin alphabet is, as suggested by the name, an alphabet, consisting of the symbols $\Sigma = \{$ *A, B, C, ..., X, Y, Z*$\}$. A language over this alphabet is then English, with L being the set of all words in the English language. Another alphabet, of greater interest to this report is $\Sigma = \{0, 1\}$, where any set of values representable as binary strings can be a language, such as for instance $L = \{0, 1, 10, 11, 100, 101, 110, 111, ...\}$, which is the language of binary representations of natural numbers.

Going back to decision problems, a concrete decision problem Q can now be said to have instance set $S \in \{0, 1\}^*$. That is, any binary string is an instance of Q. Of interest are only the strings for which Q evaluates to *true* however, meaning that Q can be expressed as a language L over $\Sigma = \{0, 1\}$, with $L = \{x \in \Sigma^* : Q(x) = 1\}$.

Using this notation allows the algorithms that solve decision problems to be accurately represented. We say that an algorithm A *accepts* a string $x \in \{0, 1\}^*$

if the algorithm, given x as input, outputs $A(x) = 1$, or in other words that the decision problem instance x evaluates to *yes* or *true*. Furthermore, the language accepted by A is the set of strings that the algorithm accepts, i.e. the language $L = \{x \in \{0,1\}^* : A(x) = 1\}$. Conversely, *A rejects* a string x if $A(x) = 0$. Note that if an algorithm A accepts a language L, this does not necessarily mean that A rejects all strings $x \notin L$; other outcomes are possible, such as for instance that A loops infinitely when given x as input.

This gives rise to the definition of a *decided* language: a language L is said to be decided by an algorithm A if every binary string in L is accepted by A and every binary string not in L is rejected by A. Finally, a language L is *accepted in polynomial time* by an algorithm A if L is accepted by A, and in addition all binary strings $x \in L$ are accepted in polynomial time. A similar definition exists for a language L being *decided in polynomial time*, with the only difference being that x needs to be decided, rather than only accepted, in polynomial time[3, p. 1058]. The concept of polynomial time will be explained in the next section.

Using this formal-language theory, it is also possible to formally define a complexity class: a *complexity class* is a set of languages, membership in which is determined by some *complexity measure* of algorithms that determine whether some string x belongs to a language L in the given complexity class. An example of such a complexity measure could for instance be running time.[3, p. 1059].

2.1.2 The P class of problems

The Polynomial time (P) class of decision problems consists of problems that are solvable in polynomial time. In other words, the P class of problems encompasses all problems that, for some input of size n and some constant k, can be solved in time $O(n^k)$[3, p. 1049]. Many sorting- and searching algorithms are in the P class.

More formally the complexity class P is the set of concrete decision problems that are polynomial-time solvable[3, p. 1055]. A function $f : \{0,1\}^* \to \{0,1\}^*$ is said to be polynomial-time solvable if there exists a polynomial-time algorithm A that, given input $x \in \{0,1\}^*$, produces output $f(x)$[3, p.1056].

A definition also exists using the formal-language theory detailed above, stating that:

$P = \{L \subseteq \{0,1\}^* : \text{there exists an algorithm } A \text{ that decides } L \text{ in polynomial time}\}$.

Problems in the P class are generally considered *tractable*, i.e. simple, or easily solved. Even problems whose best known algorithm takes time $O(n^{100})$ to solve are considered tractable, despite the long running time. The reasoning behind this is of a philosophical nature, rather than mathematical, and has to do with the following points:

Firstly, in general, once the first polynomial-time algorithm for a problem has been found, more efficient algorithms often follow, meaning that the algorithm requiring $O(n^{100})$ might not be the best known algorithm for very long. Secondly, polynomials are closed under addition, multiplication and composition, giving

polynomial-time solvable problems some nice properties. For instance, if the output of one polynomial-time algorithm is fed into the input of another, the resulting, composite algorithm is polynomial[3, p. 1054]. This is beneficial as it means that any problem that can be reduced to a polynomial-time solvable problem in polynomial time, is itself polynomial-time solvable.

2.1.3 The NP class of problems

The Nondeterministic polynomial time (NP) class of decision problems consists of those problems that are "verifiable" in polynomial time[3, p. 1049]. This means that a proposed solution to an NP class problem can be verified to be correct or incorrect in time polynomial in the size of the input to the problem. That is, the proposed solution can be verified in $O(n^k)$ for an input of size n and some constant k. Note that this classification imposes no restrictions on the time needed to solve the problem; only on the time needed to verify a solution once it has been found.

Using formal-language theory, a language L belongs to NP if and only if there exists a two-input, polynomial-time algorithm A and a constant c such that

$$L = \{x \in \{0,1\}^* : \text{there exists a certificate } y \text{ with } |y| = O(|x|^c) \\ \text{such that } A(x,y) = 1\}.$$

If this holds, A is said to *verify* the language L in polynomial time[3, p.1064].

Intuitively, this means that any problem in P is also in NP, as verifying a solution cannot be asymptotically harder than simply solving the problem. In other words, $P \subseteq NP$. It has not as yet been proven that $P \neq NP$, though the common perception is that that P and NP are not the same class.

2.1.4 The NP-C Class of Problems

The NP-complete (NP-C) class of problems can informally be defined as the set of problems in NP that are as "hard" as any problem in NP[3, p.1050]. This means that if any problem in NP-C can be solved in polynomial time, then every problem in NP-C can be solved in polynomial time. To explain this, the concept of reducibilty, which formalizes the notion of one problem being as "hard" as another, first needs to be introduced.

Reducibility

Intuitively, a problem Q is said to be reducible to Q' if any instance of Q can "easily" be converted to an instance of Q', and that the solution to this converted instance provides a solution to the original instance of Q.

A more formal definition can be expressed using formal-language theory: a language L_1 is *polynomial-time reducible* to a language L_2 if there exists a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$,

$x \in L_1$ if and only if $f(x) \in L_2$[3, p. 1067]. Here f is denoted the *reduction function* and a polynomial-time algorithm F that computes f is a *reduction algorithm*. This relationship between two languages L_1 and L_2 is written as $L_1 \leq_p L_2$.

NP-Completeness

Using the above definition of polynomial-time reducibility, along with formal-language theory, the NP-C class of problems can be formally defined. A language $L \subseteq \{0,1\}^*$ is NP-Complete if $L \in NP$ and $L' \leq_p L$ for every $L' \in NP$[3, p. 1069]. In other words, a problem Q is NP-Complete if, and only if, it is in NP and *all* problems in NP are polynomial-time reducible to Q.

As polynomials are closed under addition, multiplication and composition, as explained above, this means that if an algorithm can be found that solves some problem in NP-C in polynomial time, all problems in NP can be solved in polynomial time and hence $P = NP$.

2.1.5 The NP-H Class of Problems

The definition of NP-hard (NP-H) problems is very similar to the definition of NP-C problems, but somewhat less restrictive. Where an NP-complete language is any language $L \in NP$, where $L' \leq_p L$ for every $L' \in NP$, an NP-hard language is *any* language L where $L' \leq_p L$ for every $L' \in NP$[3, p.1069]. In other words, a problem Q is NP-hard if any problem in NP can be reduced to Q in polynomial time.

This means that $NPC \subset NPH$. Interestingly, this also means that as an NP-hard problem does not itself have to be in NP, the NP-H class of problems does not have to be restricted to decision problems. The most common definitions do consider NP-H to only encompass decision problems however, and so this definition will be used for the remainder of the report.

2.1.6 NP-completeness proof

The process for proving that a language L is NP-complete and/or NP-hard is relatively straightforward. To prove that L is NP-hard, it is sufficient to demonstrate that $L' \leq_p L$ for some $L' \in NPH$. Even though the definitions allow any problem in NP-H to be used as L' in this proof, problems in NP-C are typically selected. To also prove that L is NP-complete, it must also be shown that $L \in NP$.[3, p. 1078]

The first of these proofs follows from the fact that polynomials are closed under addition, as well as from transitivity. It is known that all languages $L'' \in NP$ are polynomial-time reducible to any language $L' \in NPH$. If it can be shown that L' is polynomial-time reducible to L, it is implicitly shown that all languages L'' are polynomial-time reducible to L, hence L is NP-hard.

The second proof follows simply from the above definition of NP-complete problems.

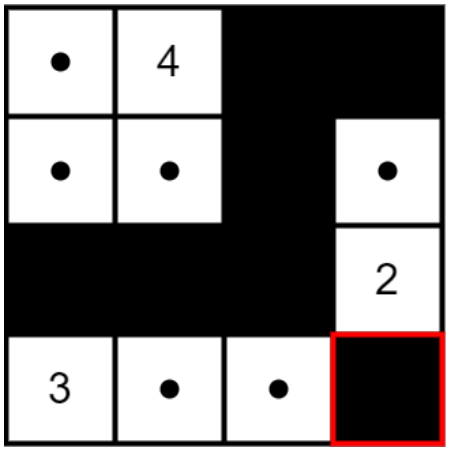
2.2 Nurikabe

As a basis for the app developed in this project, the logic puzzle known as *Nurikabe* will be used. Nurikabe is commonly classified as a *pencil puzzle* or *pencil-and-paper puzzle*, as it is typically presented on a piece of paper, and solved by drawing on the paper with a pencil[4].

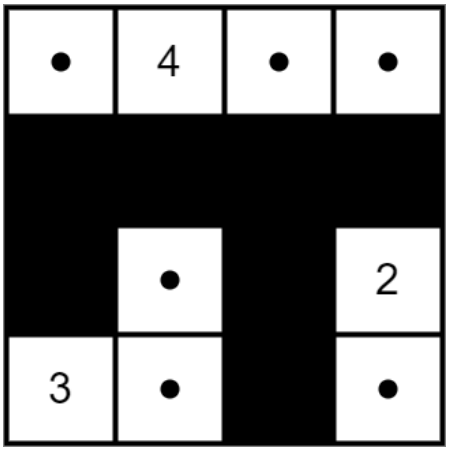
2.2.1 Rules and definitions

A Nurikabe board consists of a grid of cells. This grid is usually square or rectangular, though it could also take other shapes, such as, for instance, the shape of a + sign. Some of the cells in the Nurikabe grid contain a single natural number, and the objective of the game is to fill out all the unnumbered cells as either black or white, according to the following rules[4]:

1. White cells are denoted island cells, and black cells are denoted water cells.
2. All cells containing numbers are island cells, and the number in the cell determines the size (in number of cells) of the island. Islands are groups of orthogonally connected white (island) cells.
3. Islands always contain exactly one numbered cell
4. Different islands are always separated by water cells horizontally and vertically, though not necessarily diagonally.
5. All water cells make up a single area of orthogonally connected cells, i.e. there are no isolated bodies of water. (See figure 2.2 below).
6. Water cells cannot be connected in a way that forms a *pool*, i.e. a 2×2 area of water cells. (See figure 2.3 below).

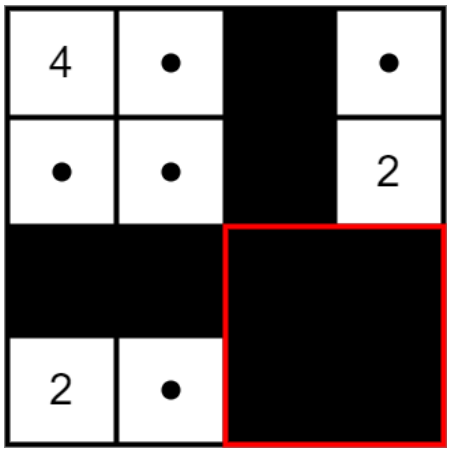


(a) This solution is invalid due to a black cell, marked by a red outline, being isolated. That is, it is not orthogonally connected to the rest of the black cells.

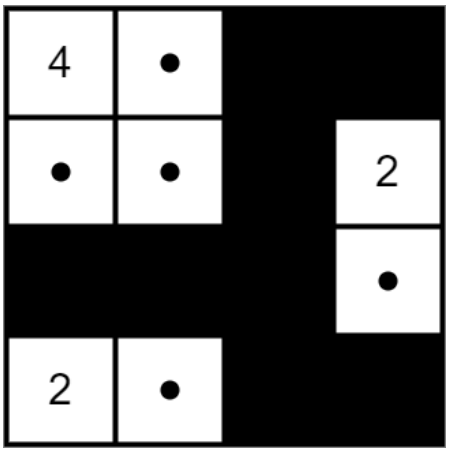


(b) This solution is valid as there are no isolated black cells.

Figure 2.2: A simple Nurikabe board with two potential solutions. One is invalid however, as it contains an isolated black cell.



(a) This solution is invalid as it contains a 2×2 "pool" of black cells, marked by a red outline.



(b) This solution is valid as there is no 2×2 area of all black cells.

Figure 2.3: A simple Nurikabe board with two potential solutions. One is invalid however, as it contains a 2×2 area of all black cells.

In addition to the above rules, some definitions are in order, to accurately describe the game and its components and mechanics:

1. **Known cells:** Cells that by some inference are known to be island cells are denoted known island cells. Likewise, cells that by some inference are known to be water cells are denoted known water cells. Cells that are not known island cells and not known water cells are denoted unknown cells.
2. **Complete island:** A complete island is an island consisting of a number of known island cells equal to the size of the island, as given by a numbered cell. Conversely, an island consisting of a number of known island cells less than the size of the island, as given by a numbered cell, is denoted an incomplete island. Islands that do not contain a numbered cell are always incomplete islands. In a completed Nurikabe board, all islands are complete.
3. **Isolated water region:** An isolated water region, or isolated region, is a region of orthogonally connected water cells that is not orthogonally connected to every other water cell in the board. In a completed Nurikabe puzzle, there are no isolated water regions.
4. **Numbered island:** A numbered island is an island, complete or incomplete, that contains a numbered cell. Conversely, an unnumbered island is an incomplete island that does not contain a numbered cell. In a completed Nurikabe puzzle, there are no unnumbered islands.
5. **Hint:** A hint is the number in a numbered island, along with the numbered cell itself, describing the completed size of an island.
6. **Island growth potential:** The growth potential of an island is the complete size of the island, given by the island's hint, minus the current size of the island, in number of currently marked, connected island cells. In other words, the growth potential of an island is the number of island cells that have to be connected to the island to make it complete. Unnumbered islands have undefined growth potentials.

It is also common practice to mark known island cells with a single dot in the center of the cell. Though not strictly necessary to complete the game, this is done to differentiate between cells that are known to be island cells, and cells that are as yet undetermined. This practice will be used throughout this report, and in the developed app.

Furthermore, for a Nurikabe board to be considered a valid puzzle, it is commonly agreed that it should have only a single, unique solution, and that it should be possible to find this solution using only rational inference, and no trial-and-error, or *bifurcation* as it is sometimes called.

2.2.2 Computational complexity of Nurikabe

Generalized Nurikabe, i.e. Nurikabe where the board can be of any size, has been proven to be NP-complete multiple times, for instance by Holzer et al[4], and by McPhail[5]. As such, a complete proof of this will not be made here, though a summary of the proofs by Mcphail and Holzer et al can be found in appendix A.

As explained above, in section 2.1.4, a problem cannot be NP-complete without also being in NP. This means all NP-complete problems, like all NP problems, have to be decision problems. To fit this definition, the Nurikabe problem has in both the two mentioned proofs been stated approximately as follows:

Problem 1: *Given an $n \times m$ board of either unknown- or numbered cells B, does there exist an $n \times m$ board B' that solves B?*

Here, solving a Nurikabe board is defined as follows:

Definition 1: *A board B' is said to solve a board B if B' is a board of no unknown cells that adheres to the rules of Nurikabe, as stated in section 2.2.1, and all known cells in B are part of B'.*

Clearly, Problem 1 being stated as a decision problem means that neither paper explicitly proves anything about the complexity of *solving* a Nurikabe board. Rather, they only determine that *deciding whether or not a given board has a solution* is NP-complete, and therefore also NP-hard.

A more interesting problem is therefore defined as follows:

Problem 2: *Given an $n \times m$ board of either unknown- or numbered cells B, what, if any, $n \times m$ board B' solves B?*

Note that this problem is not a decision problem, as it does not result in a "yes" or "no" answer, but rather in a solved Nurikabe board (if a solution exists). This problem can therefore not be NP-complete, as it is not in NP. It also cannot be NP-hard, for the same reason. But even if the problem cannot formally belong to any of the complexity classes discussed in this report, some interesting points can still be made about its complexity, using these classes.

Intuitively, the problem of solving a Nurikabe board, as stated in Problem 2 above, can be said to be as "hard" as an NP-hard problem. This is because an algorithm that solves Nurikabe boards can be no faster, at least in the asymptotic sense, than an algorithm determining whether or not a solution exists for the same board; if it was, why not just use the solving algorithm to determine the existence of a solution? Thus, because problem 1 above is NP-complete, problem 2 must be at least as hard, in terms of asymptotic time requirement. This intuitive proof closely mirrors the intuitive proof of $P \subseteq NP$, as stated in section 2.1.3, as the the solution algorithm could easily be repurposed as a verification algorithm.

More formally, if we let L_1 be problem 1 above, and L_2 be problem 2, then the following holds: any instance of L_1 can be transformed to an instance of L_2 in polynomial time. In fact, as both problems take any unsolved Nurikabe board as input, their instance sets are identical, and so no transformation is needed. Furthermore, any solution of L_2 can be transformed into a solution of L_1 by the simple transformation:

$$f : B \to \{0, 1\} \tag{2.1}$$

$$f(b) = \begin{cases} 1, & \text{if } b \text{ is a solved board} \\ 0, & \text{otherwise} \end{cases} \tag{2.2}$$

Here, B is the set of all possible solutions L_2 can produce. That is, it is the, presumably infinite, set of all possible solved Nurikabe boards, as well as a special case for unsolvable boards, for instance an empty board, or something similar.

The complexity of this transformation will depend on the representation of B, and in particular on the representation of the special case of an unsolvable board. Due to the NP-completeness of L_1 however, which guarantees that a solution is verifiable in polynomial time, we can say that the above transformation is, at worst, polynomial itself. This is due to the fact that in the worst case, being that the algorithm solving L_2 outputs a seemingly solved board regardless of the solvability of the input board, the transformation f will consist of deciding whether b is a solution to the input board, which is proven to be polynomial.

As the transformation f is, in the absolute worst case, polynomial, we can say that $L_1 \leq_p L_2$, meaning that L_2 fits definition of an NP-hard problem as far as "difficulty", or complexity, is concerned, even if it cannot formally be classified as one. It can therefore be truthfully stated that solving a Nurikabe board is then at least as complex as any NP-hard problem.

Complexity of solvable boards

Having answered the question of the complexity of generalized Nurikabe boards, the question about the complexity of human-solvable boards naturally follows. For the same technical reasons as with the problem in definition 2 above, solving a human-solvable Nurikabe board cannot, strictly speaking, be neither NP-complete nor NP-hard. It can however still be interesting to investigate the complexity of solving such a board, as compared to the definitions of these complexity classes.

One might assume that the complexity of human-solvable Nurikabe boards is in some way affected by the restrictions imposed on them, being that they should always have a unique solution, and that that solution should be possible to find using only rational inference. However, what actually breaks the complexity of such a puzzle is the fact that it is no longer generalized, and can no longer be of *any* size.

Consider for instance a human-solvable Nurikabe board of size 10×10. As this board is of a fixed, known size, it could theoretically be solved by a program that has every possible instance of a 10×10 Nurikabe board stored in its memory, and mapped to its solution. The board could then be solved in constant time, $O(1)$, by simply looking up the board. This will hold true for any given size of Nurikabe board. In other words, any **given** instance of Nurikabe, even without the added restrictions on human-solvable boards, is solvable in constant time.

That said, a human cannot be expected to memorize every possible instance of Nurikabe boards of size 10×10, or any size for that matter. Furthermore, a human is, in theory, capable of solving a Nurikabe board of any size, and can therefore be considered a generalized Nurikabe solver, unlike the specialized solver imagined above. Likewise, the techniques employed by human solvers, as will be explained in chapter 3, do not rely on assumptions about solvability to work. That is, these techniques will work regardless of whether a board has multiple, one, or no solutions; at least until the point where a board can be deemed unsolvable or a split between multiple solutions is identified. These assumptions are, in other words, only present to increase the enjoyment of the solver; it is after all no fun to solve a puzzle only to find out it is unsolvable.

Looking more closely at the NP-completeness proof for generalized Nurikabe, in the papers by McPhail[5] and Holzer et al[4], it is clear that the Nurikabe board components used in these proofs do in fact comply with the technique restrictions normally only applied to human-solvable boards. That is, the components, when combined to form a complete, logic circuit-analogous, puzzle, are solvable using only the techniques emplyed by human solvers. This means that a human solver would be capable of "solving" any such puzzle, in the sense that they could, using nothing but rational inference and enough time, identify whether the puzzle has zero, one or multiple solutions.

This means that the NP-completeness proofs for Nurikabe, and by extension the above proof of the complexity of solving a board, still hold for strictly human-solvable boards. Any board solvable to a human is at least as "hard" to solve as any NP-hard problem.

This might seem counter-intuitive, as smaller, human-solvable boards can be solved very quickly, both by humans and computers alike. This does however not break the above proof, nor does it prove that $P = NP$. This is because a problem being NP-hard does not in any way mean that a specific instance of the problem must be very hard to solve. Indeed, as explained above, any given instance of Nurikabe is theoretically solvable in constant time to a solver specific to the input size of the problem. Even for generalized solvers, such as humans, small instances of a problem might be very quickly solved. It is only as the size of the problem grows, in this case as the board size increases, that the NP-hardness comes into play, causing a generalized solver to take increasingly long to find a solution.

2.3 Technologies

As was explained in the introduction (chapter 1) of this report, the .NET platform, along with the Xamarin framework, was chosen to develop the app in this project. Furthermore, Haskell was chosen as the language of the generator, with an ASP.NET Core web app serving to simplify calling it. In this section, these technologies will be examined more in depth, with a basis in how they were used in the project

2.3.1 The .NET platform

.NET is an open-source platform, developed and maintained by Microsoft. As such, there are vast amounts of documentation and resources available online, as well as a large developer community, with Microsoft claiming there are as many as 5,000,000 .NET developers world-wide[6]. Additionally, the .NET platform is cross-platform, and multiple implementations of the platform exist, providing compatibility with several different Operating Systems (OS's). These are:

- *.NET*, previously called *.NET Core*, the base .NET implementation compatible with websites, servers, and console apps on Windows, Linux, and macOS.
- *.NET Framework*, supporting websites, services, desktop apps and so on specifically on Windows.
- *Xamarin/Mono*, supporting apps on all major mobile operating systems, such as Android and iOS.

Each of these implementations further support the use of the following programming languages:

- *C#*, a simple, object-oriented and type-safe programming language.
- *F#*, a programming language intended for writing succinct, robust and performant code.
- *Visual Basic* a more approachable language with simple syntax for writing type-safe, object-oriented apps.

Regardless of language and implementation, .NET provides a common base set of Application Programming Interfaces (APIs) called *.NET Standard*, and the platform has its own package manager called NuGet, that allows for installation of additional libraries. OS specific APIs can also be exposed by the corresponding implementation. For instance, the Windows specific implementation .NET Framework includes APIs for accessing the Windows Registry.

Included with every implementation of the .NET platform, is the Common Language Runtime (CLR). This run-time environment provides several services to the programmer, the most important of which being memory management and services that enable communication between different languages; first and foremost the different languages that are a part of .NET, but also other, external languages. Code that is managed by this run-time is commonly referred to as *managed code*, as opposed to *unmanaged code* that is not.

ASP.NET

ASP.NET is an extension of the .NET platform, specifically designed for building web apps. It adds several features to the base .NET framework that simplifies building web apps and web APIs. Among these features are: a framework for processing web requests, a web-page templating syntax called Razor, an authentication system, as well as libraries for common web patterns and editor extensions to provide syntax highlighting and similar features. [7]

ASP.NET Core is the successor to ASP.NET, and has the added benefits of being open-source and cross-platform.

C#

As mentioned above, part of the reason for choosing the .NET platform, was to learn the C# programming language. C# is a high-level, object-oriented and type-safe programming language, used to build applications that run in .NET. Unsurprisingly, it has its roots in the C family of programming languages, and as such shares some basic syntax with C, C++ and other similar languages.

In addition to being an object-oriented language, C# is component-oriented, providing language constructs to enable the creation and use of software components. Furthermore, the language has features to aid in creating robust and durable applications, such as garbage collection to automatically reclaim unused memory, nullable types to guard against unallocated variables, and exception handling.

The language also supports lambda expressions to support more functional programming techniques, Language Integrated Query (LINQ) syntax to simplify data handling independently of source, and asynchronous operations to support distributed systems. [8]

2.3.2 The Xamarin framework

As mentioned in chapter 1, it was for this project decided to use Microsoft's Xamarin framework, building on .NET and C# to develop the app.

Xamarin is an extension of Microsoft's .NET platform, containing tools and libraries for creating apps for Android, iOS, macOS, etc. As Xamarin is an abstraction layer that manages communication of shared code with underlying platform code, the majority of written code can be shared between these platforms, and Microsoft claims that an average of 90% of an application is shared across platforms[9].

Xamarin is further divided into Xamarin.Android, Xamarin.iOS, Xamarin.Essentials and Xamarin.Forms. Xamarin.Android handles compilation of C# applications for Android devices, with Xamarin.iOS providing the same functionality for iOS devices. Xamarin.Essentials provides cross-platform APIs for native features, such as file management. Lastly, Xamarin.Forms is a cross-platform UI framework that allows for the creation of apps and UIs for several platforms using a single code base.

In the following sections, some important concepts of app development in the Xamarin framework will be presented. For the most part, especially for the user interface elements, these concepts were learned by following the offical Microsoft tutorial series for Xamarin.

XAML

The eXtensible Application Markup Language (XAML) is an XML based language created by Microsoft, that allows instantiating and initializing objects, as well as

organizing those objects in parent-child hierarchies, without using code. Though it is adapted to several technologies within the .NET platform, it is mostly used to define the layout of UIs.

In Xamarin.Forms, XAML allows developers to define their app's UI using markup rather than actual code. Though this is not necessary, as UIs can also be created purely through code, using XAML is usually more succinct and readable. XAML is also well adapted to the MVVM pattern commonly found in Xamarin apps, which will be explained below.

As XAML is a markup language and not a full-fledged programming language, it cannot contain any actual code. Therefore, all event handling must take place in the so-called code-behind; a code file associated with the XAML layout. As an example, the XAML UI can define a button's size, positioning and style, but not what happens when the button is pressed. For this, a handler method in the code-behind is required. Typically an app consists of several XAML files; one for each page of the app, as well as some defining common properties and styles, such as the app's primary- and secondary color. [10]

MVVM

The process of creating an app using Xamarin usually involves creating a UI and some code-behind, called the business logic, that handles events and commands to and from the UI. This approach might lead to issues, however, with tight coupling between UI controls and business logic, increasing the difficulty of making UI modifications and of testing.

The Model-View-ViewModel (MVVM) pattern is designed to help alleviate these issues by separating business logic from the UI. It also improves code re-use opportunities, and makes apps easier to develop and maintain.

The MVVM pattern has three main components; the model, the view, and the view model. Figure 2.4 shows how these components are connected, and can intuitively be explained as the view "knowing about" the view model, and the view model "knowing" about the model, but the inverse being hidden. That is, the model is unaware of the view model, and the view model is unaware of the view.

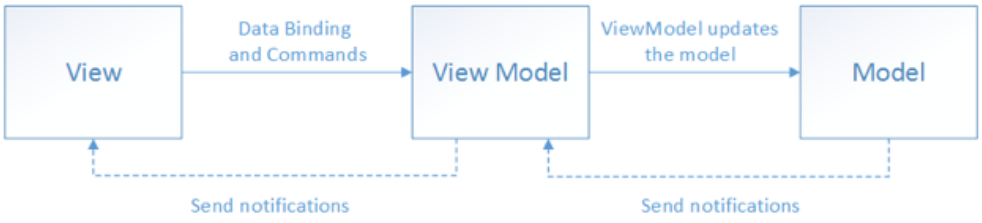


Figure 2.4: Figure illustrating the MVVM pattern[11].

The view in the MVVM pattern is responsible for defining the structure, layout and appearance of the user interface. The view is usually defined in XAML and

has a very limited code-behind that does not contain any business logic. There might be some UI logic in some cases however, to handle visual behaviours such as animations.

The view model in the MVVM pattern implements the properties and commands that the view is bound to. It will also serve to notify the view of state changes by invoking the corresponding events. As an intuitive way of understanding this, what a button does is defined in the view model, but how the button looks to a user is defined in the view.

In addition to this data binding, the view model handles connections between the view and any model classes it requires. This sometimes involves directly exposing these classes to the view, so that the view can bind directly to the data in the model class. Other times, it requires the view model to perform data conversions on the data from the model class, to make it easier for the view to utilize.

Lastly, the models in the MVVM pattern are entirely non-visual classes that contain all the data and functionality of the app. An example of a model could for instance be a Data Transfer Object (DTO) used when communicating with a database, or a more functional part of the app. [11]

User interface

The Xamarin.Forms framework contains many predefined UI components, called controls, that can be used to build an app. These controls are hierarchically divided into four subgroups: pages, layouts, views and cells[12].

Pages take up most, if not all, of the screen when they are displayed. A control page is usually the top-level visual component of any page of an app, and will serve as a container for other controls, structuring them against each other. A *Content page* is the simplest form of page and only contains a single view or layout. A *Flyout page* is another page that, unlike the content page, manages two panes with information; one much resembling the content page, and another representing a menu, or list of actions that can be accessed from the side of the screen. Other examples of pages are *navigation pages*, *carousel pages* and *tabbed pages*[13].

Layouts in Xamarin.Forms are specialized subtypes of views that act as containers for views and other layouts, and is capable of structuring these views and layouts within itself. Some layouts can only have a single child, such as *ContentViews*, and their derivative *Frames* which can display a border around their child. Other layouts can have multiple children, such as *StackLayouts*, which positions its children in a single stack either vertically or horizontally, and *Grids*, that position their children in a grid of rows and columns[14]. Figure 2.5 displays how a stackLayout can structure its children vertically.

In Xamarin.Forms, **View** is a common term for UI objects, that are often called controls or widgets in more graphical programming environments. Some views are designed to present information to a user, such as *Labels* containing blocks of text, *Images* that display images, such as in figure 2.6a, and *WebViews* that can display webpages or HTML content. Other views allow the user to initiate

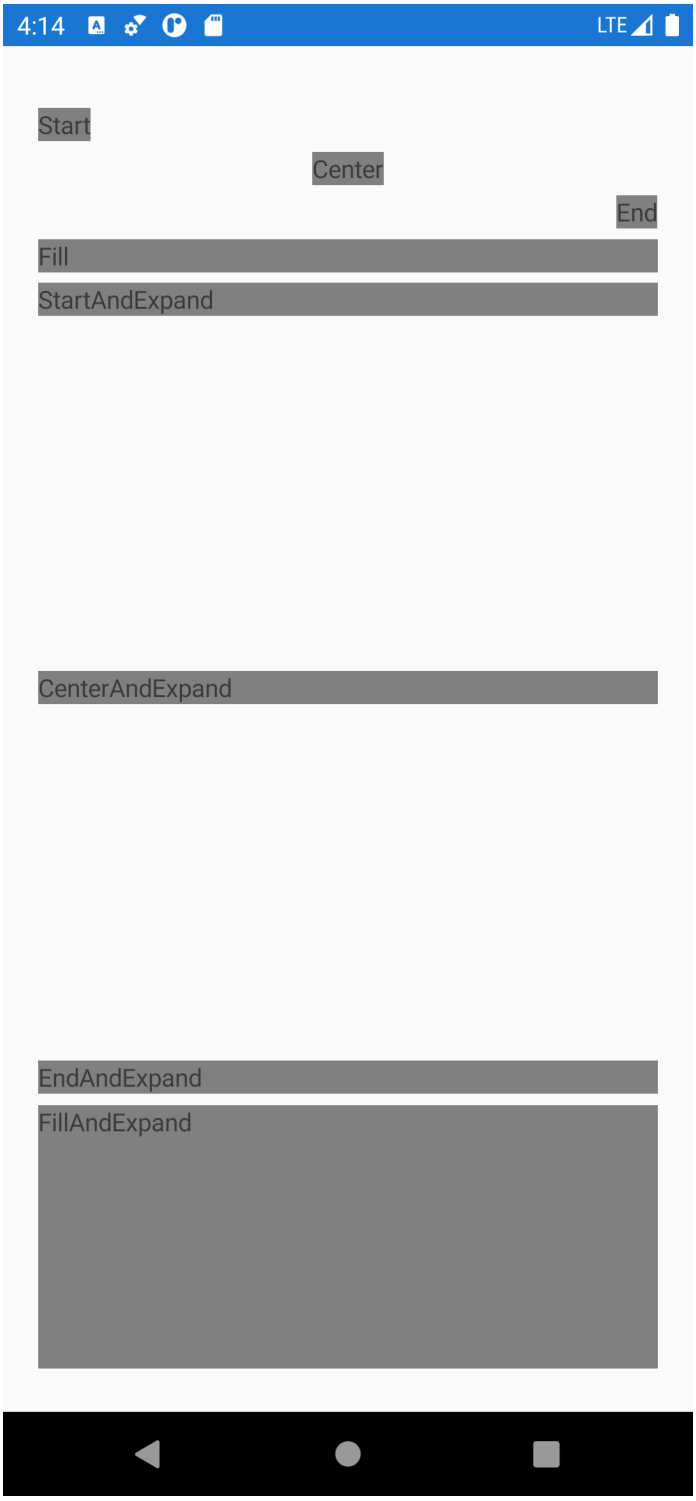
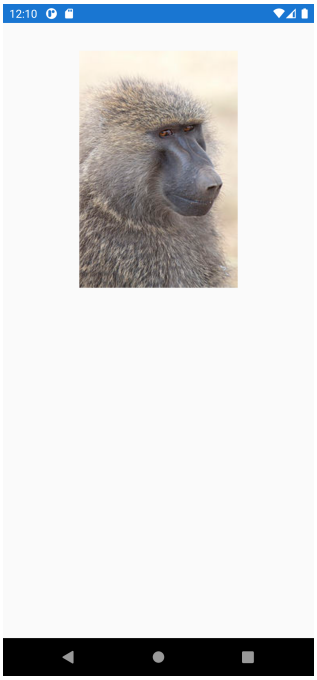


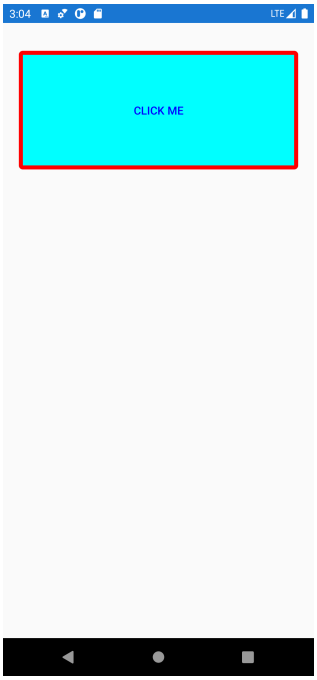
Figure 2.5: Emulator screenshot of simple stackLayout, displaying different alignment and expansion properties of labels. Note that labels have been given a gray background in order to more easily show their alignment and expansion properties.

commands, mainly various forms of *Buttons* like the one in figure 2.6b. Some views are used for setting values of different types, such as *CheckBoxes* for boolean values and *Sliders* for doubles. There are views for editing text, such as *Entry* and *Editor*, as can be seen in figure 2.6c, and others that indicate activity, such as a *ProgressBar*. Lastly there are views that display collections, such as *CollectionView* and *ListView* that both display scrollable lists of items[15]. See figure 2.6d for an example of a collectionView.

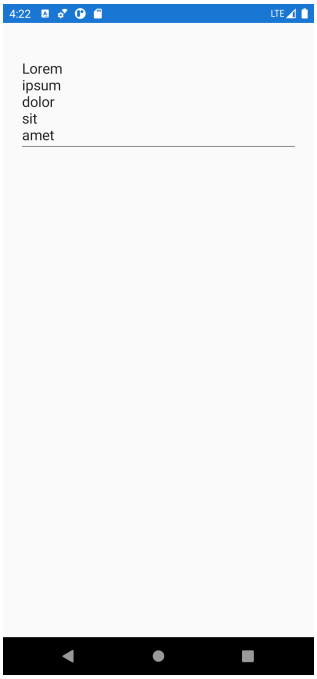
Finally, **Cells** are template UI elements that are used to display items in a table. Cells are used exclusively with the *ListView* and *TableView* views, and are used to customize how entries are displayed, e.g. as *TextCells*, *ImageCells* or *EntryCells*. [16].



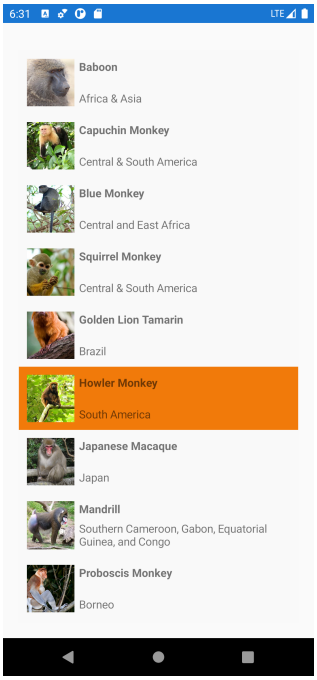
(a) Emulator screenshot of a simple image app, displaying a single image of a Baboon.



(b) Emulator screenshot of a simple button app, displaying a single button with a light blue background and red border.



(c) Emulator screenshot of a simple editor app, displaying an editor used to enter and edit multiline text.



(d) Emulator screenshot of a simple collectionView app, Displaying a collectionView of monkeys, with the Howler Monkey selected.

Figure 2.6: Emulator screenshots of apps displaying various views.

2.3.3 The Haskell programming language

According to the official Haskell wiki, Haskell is a *polymorphically statically typed, lazy, purely functional* programming language[17]. The language is named after Haskell Brooks Curry, whose work in mathematical logic serves as the foundation for Haskell, and other functional programming languages.

Functional programming

As mentioned, Haskell is a purely functional programming language, as opposed to an imperative programming language. A "normal" imperative language, like C, C# or Python, would run a program by executing a series of statements in order, thus altering some global state. This means that variables can be assigned values, only to be reassigned some different value at a later point. More importantly it also means that functions, or methods, can have side effects, such as writing something to a console, or altering some global variable. Haskell on the other hand, like all functional languages, runs a program by evaluating a series of expressions. Where functions in imperative languages define what a computer should *do*, in Haskell they merely state what something *is*. This means that functions in Haskell are restricted to only computing some value and returning it, without any side effects. Because of this restriction, a function called with the same parameters will always return the same result, a concept called *referential transparency*. This concept simplifies the process of proving that a function is correct, and allows for more complex functions to be built by combining simpler ones[18].

Lazy programming language

In addition to being functional, Haskell is also a lazy programming language. This means that unless specifically told otherwise, Haskell won't perform any computations, be they function calls or variable assignments, until the result of the computation is needed elsewhere. By evaluating expressions this way, it is possible to bypass undefined values, such as the results of infinite loops, and it is possible to process formally infinite data structures. As an example of this, Haskell enables a programmer to work on infinite lists, by simply deferring the evaluation of each element until it is actually needed by the program[18]. Another advantage of lazy programming languages, is in terms of efficiency, as values are not calculated unless they are, in fact, needed by the program. In most imperative languages for example, evaluation is *eager*, meaning they would, in most cases, execute all statements written by the programmer, regardless of the statements significance to the final result. Lazy languages like Haskell on the other hand, would completely skip any written expressions that aren't necessary to compute the desired result.

Statical typing

The fact that Haskell is statically typed means that variable types are checked at compile-time, rather than run-time. This allows type errors to be caught before a program actually runs, and saves resources by not having to allocate memory for values of unknown types at run-time. One might assume that this means that variables in Haskell must be given a specific type by the programmer. This is however not the case thanks to type inference, a mechanism that lets Haskell figure out the type of a variable without it being explicitly stated, based on the functions performed with the variable as a parameter or as a returned value. [18]

Polymorphic typing

Haskell implements polymorphism at a very high level. This means that variables are not constrained to being of a single type, but can take on any type allowed by the functions the variables are part of. As an example, the *head* function in Haskell, with type declaration `head :: [a] -> a`, will return the first element of a list, regardless of the type of the list's elements. This is an example of parametric polymorphism, as opposed to ad-hoc polymorphism[19]. With ad-hoc polymorphism, a value can take on several different types, defined by some constraint, but not all. In this case, the variable or function must be given a separate definition for each of the types it can take on. The variable or function can therefore not take on absolutely any type, as is the case with parametric polymorphism. In Haskell this is implemented through the use of *type classes*, which allows for ad-hoc polymorphism where the function is not defined separately for each possible type, but is rather only defined once, and imposes constraints by demanding that variables are of types that instance certain type classes; meaning that they provide implementations for certain standard functions.

Type classes

In Haskell, a type class is "a sort of interface that defines some behaviours"[18]. In other words a type class defines some behaviours that all members of the type class need to implement. An example of this is the **Eq** type class which defines the `(==)` function (note also that in Haskell, `(==)` is simply a function like any other, and not a special *operator* as it would be called in many imperative languages). This means that for a type to be part of the Eq type class, it has to provide an implementation of the `(==)` function. This is very conducive to a high level of polymorphism, as it allows functions to restrict variables' type classes (called class constraints), rather than types, and so enabling functions to be called on any type that is a member of the given type class. Continuing with the Eq example, the *elem* function in Haskell, with type declaration `elem :: (Eq a) => a -> [a] -> Bool`, constrains its input type to be an instance of the Eq type class, allowing it to use the `(==)` function to check whether some value exists in a list.

Note that a single Haskell type can be a member of, or *instance*, multiple different type classes.

Monads

Attempting to answer the question of what a *monad* is, is a difficult endeavor, as evidenced by the sheer multitude of forum posts and articles that attempt to answer the question. This section will therefore make no attempt to explain this, as that would likely require an entire thesis of its own. Instead, this section will focus on how, and for what, monads are used in Haskell.

In Haskell, *Monad* is a type class. Specifically, it is a type class whose members are not concrete types, such as *Int* or *String*, but rather type constructors[18]. Type constructors are special types that take a concrete type as a parameter, and "wraps around it". Examples of type constructors are the list (`[]`) type, that produces a sequence of elements of some single type, and the *Maybe* type that contains either a single element of some type, or *Nothing*.

The Monad type class defines several functions that must be implemented by its member types[18]. Of note are the functions *return* and `(>>=)`. The first of these, with type declaration `return :: a -> m a` function, simply wraps an element of some concrete type a in a monad instance type m. The second function, with type declaration `(>>=) :: m a -> (a -> m b) -> m b`, often referred to as *bind*, takes a value wrapped in a monad, called a monadic value, $m\ a$, and a function that that accepts a value of type a and returns a monad of some other type b, and returns a monad of type b. The Monad type class defines more functions than these, but these are given default implementations, and are rarely used in practice, so won't be detailed here.

The bind function is very useful in practice, as it enables a programmer to chain monadic functions. As an example, consider a function that takes two strings as input and returns a *Maybe String* value. It could for instance be a function that returns a string of all characters that appear in both input strings if any exist, and *Nothing* if not. Using this function to compare three, or more, strings would prove cumbersome, as it would require first comparing two of the strings, before using pattern matching, or a similar construct, to unpack the result. This result would then have to be compared with the next string, before unpacking the result again, and so on. Using bind instead, which is possible because Maybe is an instance of the Monad type class, would greatly simplify this procedure. Using bind would allow the result of the first comparison, i.e. a Maybe String value, to be passed back into the comparison function without having to manually unpack it first. Due to the implementation of the bind function the Maybe monad defines, this also means that if the first two strings return a Nothing when compared, all subsequent calls will return Nothing, without having to do any comparisons. This is obviously correct, as if no characters are common between the first two strings, no characters can be common to all the compared strings.

To further simplify chaining monadic functions, Haskell also introduces the

so-called *do*-notation. Using this notation (through the use of the *do* keyword) allows monadic values to be bound to seemingly non-monadic variables using the <- binding. These variables can then be used as inputs to functions as if they were simple non-monadic values, with Haskell taking care of chaining the bind functions "under the hood".

All the above features allow Haskell to mimic some of the functionality of imperative programming languages, without compromising its functional nature. As mentioned, Haskell allows no side-effects, and functions must be referentially transparent. This would seemingly indicate that many useful operations, such as IO operations are not possible; writing something to the console is, after all, a side-effect, and getting input from the console breaks referential transparency, as a user might choose to input different values at different times. By wrapping such operations in a monad (in this case the *IO* monad), they are allowed, in spite of them "breaking the rules" of normal Haskell. The reason why this is allowed, is that many monads, such as the IO monad, are *one-way monads*, meaning that values may enter the monad, but not return from it. That is, an IO monad can be created from a string, but a string cannot be extracted from an IO monad. This means that failure, which is always a risk when dealing with side-effects and non referentially transparent functions, is completely contained within the monad operations, often indicated by an enclosing *do*, and thereby separated from the rest of the code.

Unlike the IO monad, the Maybe monad is not a one-way monad, as its value can be extracted in many ways, for instance through pattern matching. This means that "illegal" operations, i.e. operations with side-effects etc., are **not** allowed within the Maybe monad.

The Foreign library

The way that Haskell handles memory and stores data differs strongly from how it is done in many other languages. In particular it differs from how it is done the C family of languages, which unsurprisingly includes C#. This means that a Haskell program cannot normally interface with a program written in a C-family language.

Using Haskell's *Foreign* library however, introduces certain new data types, type classes and functions that enable such an interfacing. Of particular interest are the data types in the *Foreign.C.Types* module. These data types mirror the standard data types used in Haskell, and even shares names with them, only prepended by a capital "C". For instance, the module exposes the data type *CInt*, which corresponds to the common Haskell type, *Int*, only occupying memory in a way that resembles C-family languages. Similar types also exist for strings, doubles, chars, and so on.

Another important part of the Foreign library, is the *Storable* module, which introduces the *Storable* type class. This type class allows its instances to be written to memory and referenced, by pointers, in a way that is compatible with C-family

languages. Doing so enables parts of memory to be shared between programming languages in a useful way, but it does come with the caveat that memory used this way has to be manually handled by the programmer; i.e. it has to be manually allocated before being utilized, and manually freed once it is no longer needed.

Chapter 3

Development

In this chapter, the process of developing the Nurikabe puzzle generator will be detailed. This includes the preliminary work done to develop an algorithm capable of generating puzzles, the actual implementation of this algorithm, and the containment of the resulting program in a web app. The development process resulting in the *MasterGame* app is also covered in this chapter.

3.1 Nurikabe solution techniques

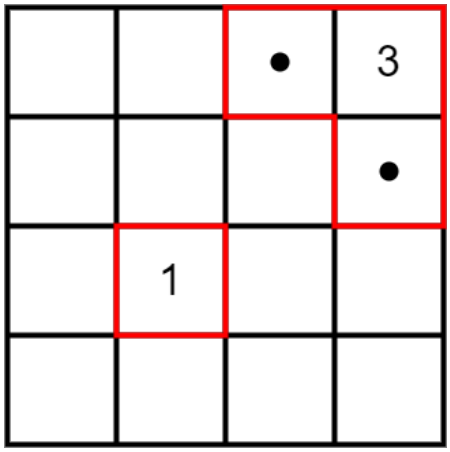
Based on the rules and definitions in section 2.2.1, some techniques for solving Nurikabe puzzles were inferred. These are presented below, grouped into to lists by the perceived difficulty of each technique. These lists of techniques provided the basis for a solver implemented as part of the puzzle generator in this project. Note that these lists are not exhaustive, and other techniques might (and very likely do) exist. For the most part, composite techniques, resulting from the combination of two or more of the listed techniques (or alternatively multiple steps of the same technique), have been omitted for brevity. These composite techniques would likely not require a separate implementation in a computerized solver in any event, meaning they are of little interest in this report.

Note that the figures illustrating the different techniques presented in this section do not necessarily comply with the full set of Nurikabe rules, as they are only meant to be examples of techniques in isolation. Attempts at solving these boards should therefore not be made, as a valid solution might not exist.

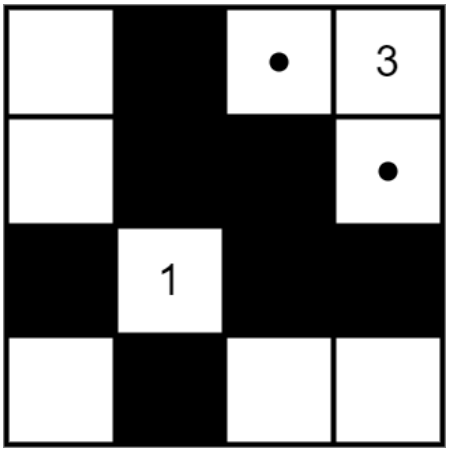
3.1.1 Beginner techniques

These techniques are mostly trivial and will be easily applicable by even completely fresh Nurikabe solvers.

1. **Completed islands:** When an island is complete, all cells orthogonally connected to the island that are not themselves part of the island, are known to be black (water cells). See figure 3.1.
    a. As a special case of this, numbered cells containing the number **1** are always known to be surrounded orthogonally by water cells.



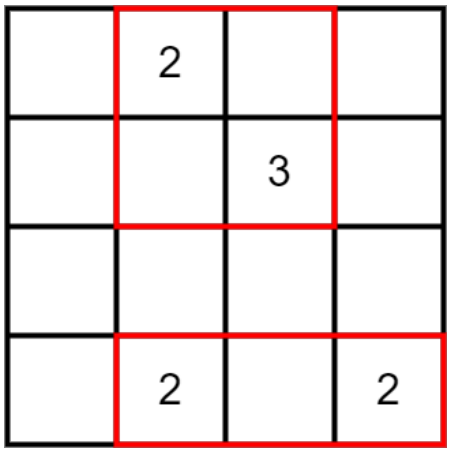
(a) Two completed islands are marked by red outlines.



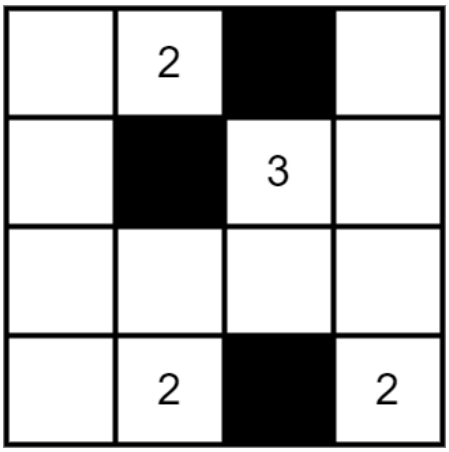
(b) Here, the two completed islands have been surrounded by water cells.

Figure 3.1: A part of a simple Nurikabe board, exemplifying the **completed islands** technique.

2. **Island separation:** In accordance with rule 4 above, cells orthogonally adjacent to two or more different islands must be water cells to ensure the islands are separated (see figure 3.2). Two special cases of this rule exist:
    a. If two numbered cells in the same column or row of the grid are separated by a single cell, that cell must be water.
    b. If two numbered cells are diagonally adjacent to each other, the two cells connecting them orthogonally must be water cells.



(a) Two sets of two numbered cells sharing adjacent cells are marked by red outlines.

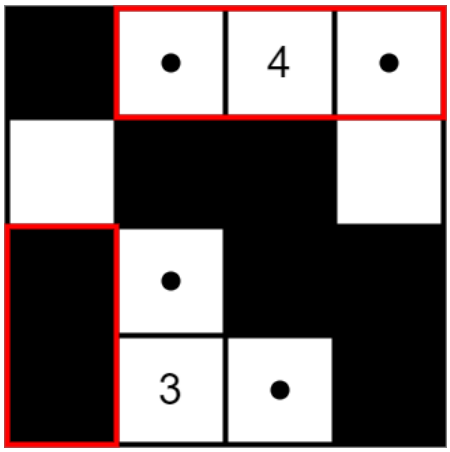


(b) Here, the shared adjacent cells are made water cells in order to ensure island separation.

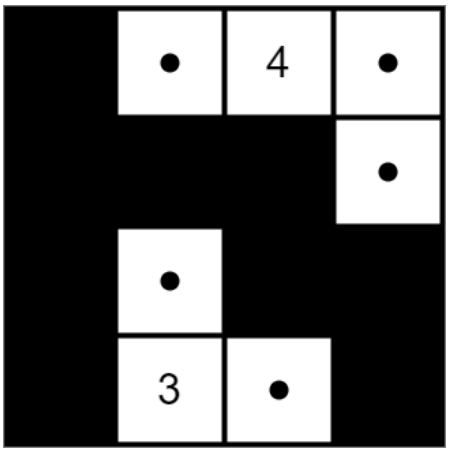
Figure 3.2: A part of a simple Nurikabe board, exemplifying the **island separation** technique.

3. **Island expansion:** If an incomplete island can only expand to a single cell, that cell must be part of the island. See figure 3.3.

4. **Water expansion:** If an isolated water region can only expand to a single cell, that cell must be a water cell. See figure 3.3.



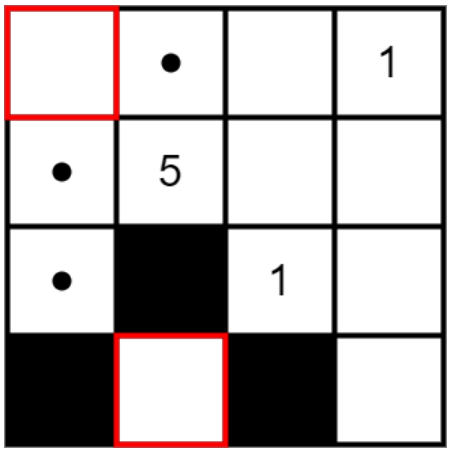
(a) An incomplete island that can only expand to a single cell, and an isolated water region that can only expand to a single cell are marked by red outlines.



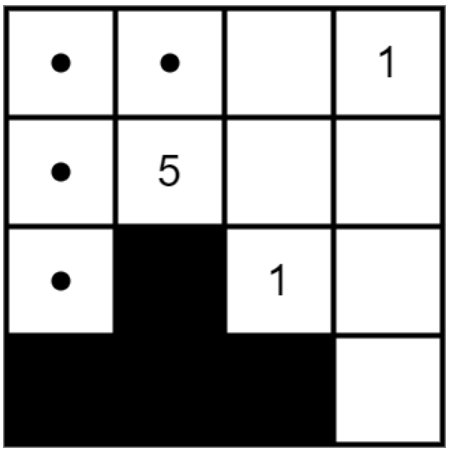
(b) Here, the incomplete island has been expanded to become a complete island, and the isolated water region has been expanded to no longer be isolated.

Figure 3.3: A part of a simple Nurikabe board, exemplifying the **island expansion** and **water expansion** techniques.

5. **Surrounded cell:** If an unknown cell is surrounded by either all island cells or all water cells, the cell must be of the same type as the cells surrounding it. See figure 3.4.



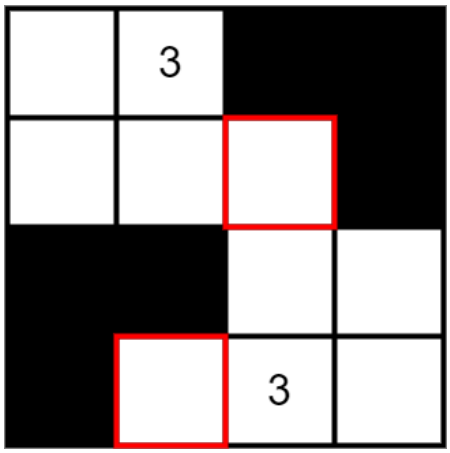
(a) A cell surrounded by water cells and a cell surrounded by island cells are marked by red outlines.



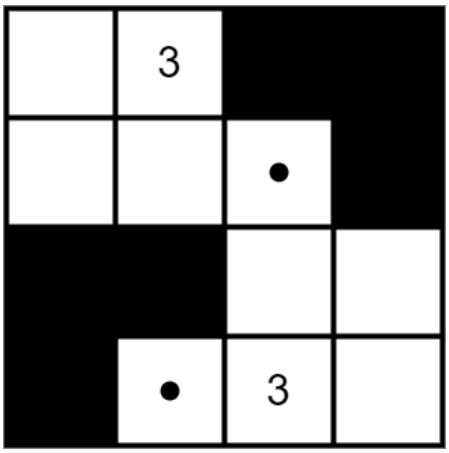
(b) here, the previously marked cells have been filled in to match the type of their surrounding cells.

Figure 3.4: A part of a simple Nurikabe board, exemplifying the **surrounded cell** technique.

6. **Pool avoidance:** If a 2×2 area of cells consists of 3 water cells, the last cell must be an island cell, according to rule 6 above. See figure 3.5.



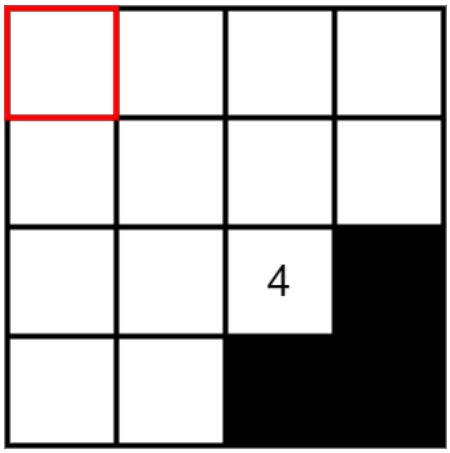
(a) Two unknown cells that are part of 2×2 areas with three water cells are marked by red outlines.



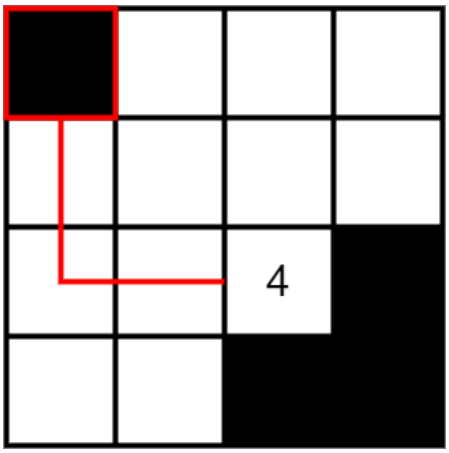
(b) Here, the unknown cells have been made island cells to avoid 2×2 pools of water cells.

Figure 3.5: A part of a simple Nurikabe board, exemplifying the **pool avoidance** technique.

7. **Unreachable Cells:** If an unknown cell is unreachable to all numbered islands in the board, the cell must be a water cell. A cell is unreachable to an island if the traversable distance between the island and the cell is greater than the growth potential of the island. The traversable distance between an island and a cell is here defined as the shortest path from the island to the cell that only traverses unknown cells, or island cells belonging to unnumbered islands. See figure 3.6.



(a) An unknown cell is marked by a red outline.



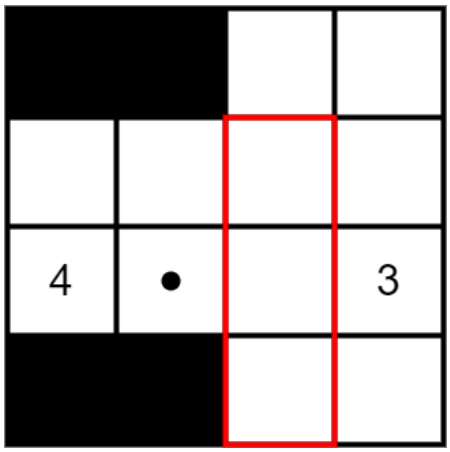
(b) Here, the shortest traversible distance between the island and the unknown cell has been marked by a red line. Note that other paths exist, but none that are shorter. The unknown cell has been marked as a water cell, as the distance between the cell and the island is 4, whereas the island's growth potential is only 3.

Figure 3.6: A part of a simple Nurikabe board, exemplifying the **unreachable cells** technique.

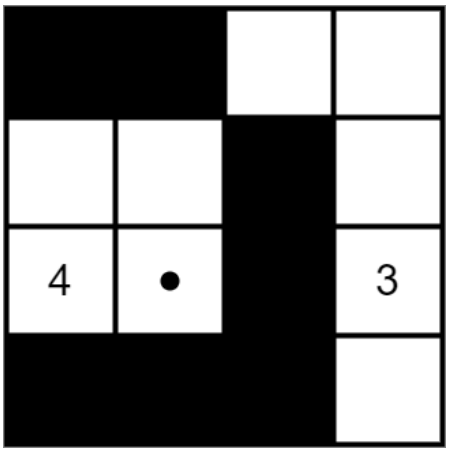
3.1.2 Advanced techniques

These techniques are more advanced than the beginner techniques and as such might take some practice before they can be reliably applied.

1. **Water connectivity:** If an isolated water region can only be connected to the rest of the water cells through a single cell, that cell must be a water cell. This is a generalization of technique 4 above. See figure 3.7 for a visual explanation.



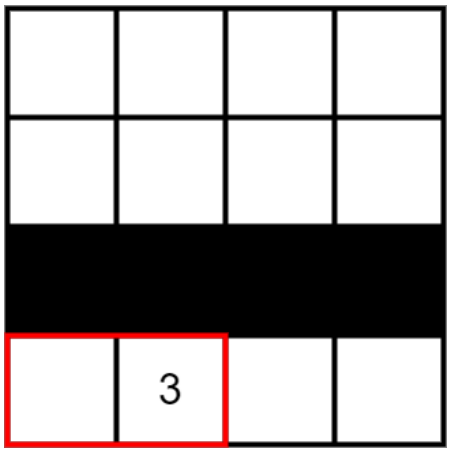
(a) Three cells that are part of the only path that connects the two isolated water regions are marked by a red outline.



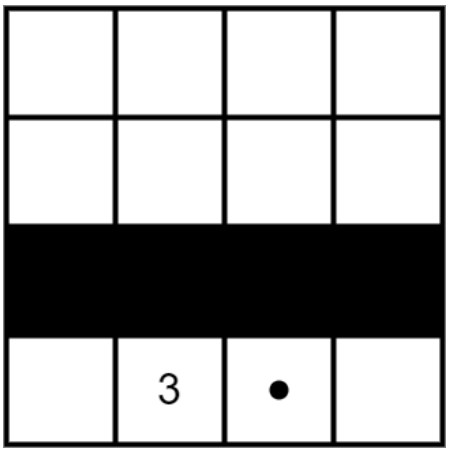
(b) Here, the three cells have been penned as water cells to ensure the two isolated regions can be connected.

Figure 3.7: A part of a simple Nurikabe board, exemplifying the **water connectivity** technique.

2. **Island expansion 2:** If an incomplete numbered island can expand in a number of directions given by x, and for some subset of $x - 1$ of those directions the island can not expand enough to be completed, the Island must expand in the direction not part of the subset.
    a. As a special, explanatory case of this, imagine an incomplete numbered island can expand in two directions. If one of these directions contains fewer connected unknown cells than necessary to complete the island, the island must expand in the other direction. See figure 3.8.



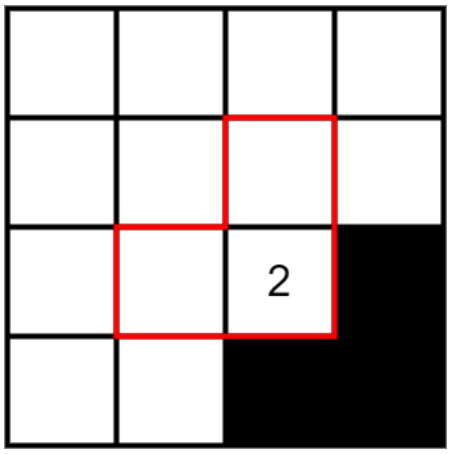
(a) One of two potential expansion cells of the number 3 cell is marked by a red outline. The island cannot expand more than one cell in this direction.



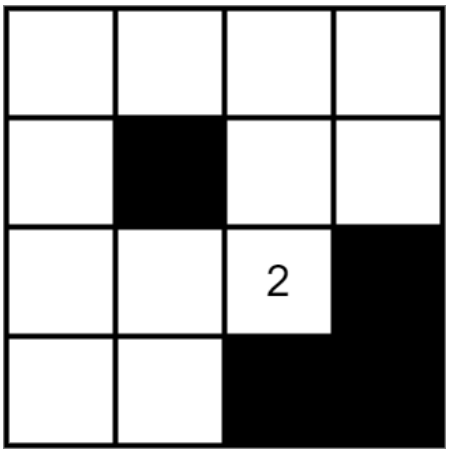
(b) Here the other expansion cell has been marked as an island cell, to ensure that the island can grow to its complete size.

Figure 3.8: A part of a simple Nurikabe board, exemplifying the **island expansion 2** technique.

3. **Island expansion 3:** If an incomplete numbered island only lacks one cell to be complete, and can only expand from a single cell to one of two diagonally adjacent cells, the cell that is orthogonally adjacent to both of these cells, and diagonally adjacent to the island cell, must be a water cell.
    a. As a special, explanatory case of this, imagine an incomplete numbered island consisting of a single, numbered cell, containing the number two. If this cell can only expand to one of two diagonally adjacent cells, either because the island is adjacent to a wall, or because of known water cells, the cell that is diagonally adjacent to the island cell, and orthogonally adjacent to both potential expansion cells must be a water cell. See figure 3.9



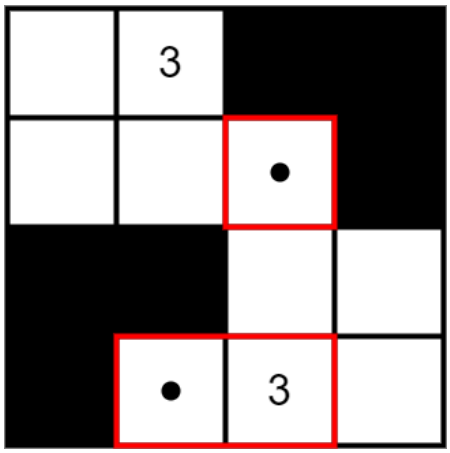
(a) A number two cell and its only two expansion cells have been marked by a red outline.



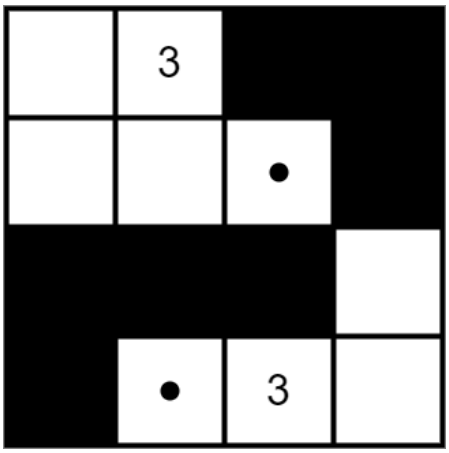
(b) Here, the cell that is diagonally adjacent to the number 2 cell, and orthogonally adjacent to both its expansion cells is marked as a water cell. This is known because the cell cannot be an island cell without breaking the 2-hinted island, regardless of which of the expansion cells it contains.

Figure 3.9: A part of a simple Nurikabe board, exemplifying the **island expansion 3** technique.

4. **Island separation 2:** If a numbered island is separated orthogonally from an unnumbered island by a single cell, and the size of the unnumbered island is greater than or equal to the growth potential of the numbered island, the cell separating the islands must be water (see figure 3.10) . This is more advanced special case of the island separation technique (Beginner technique 2).



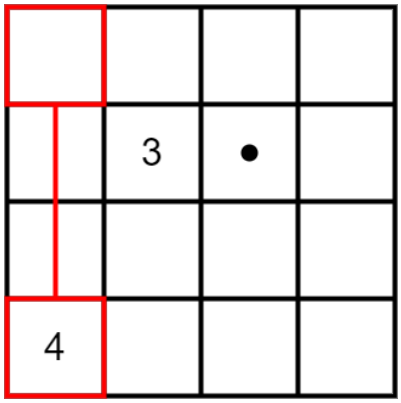
(a) A numbered island of size 2 and with hint 3 has been marked by red outline, as well as an unnumbered island of size 1. Note that the numbered island has growth potential $3-2=1$.



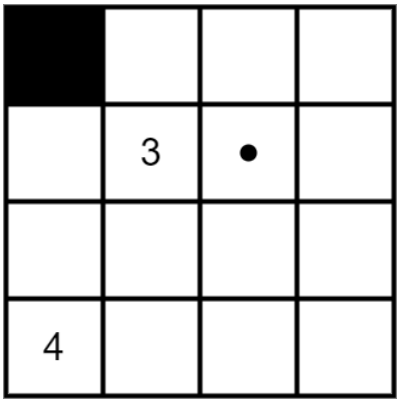
(b) Here the cell separating the two previously marked islands has been marked as a water cell. As the previously marked numbered island had a growth potential of 1, and the unnumbered island a size of 1, the two could not be connected without increasing the numbered island's size to 4, thereby breaking the hint, and consequently the puzzle.

Figure 3.10: A part of a simple Nurikabe board, exemplifying the **island separation 2** technique.

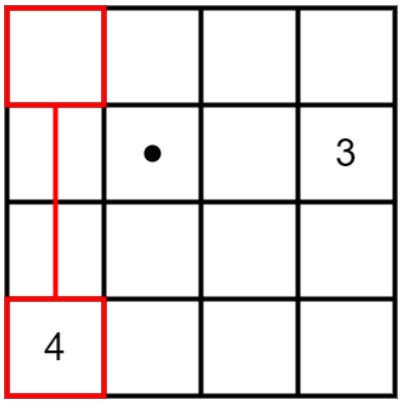
5. **Unreachable cells 2:** If a cell is reachable according to the definition above in beginner technique 7, but the islands that can reach said cell can only do so by connecting to a different numbered island, the cell is deemed unreachable, and therefore marked as a water cell. See figure 3.11.
    a. In a similar case to the above, if a cell is reachable only by islands that would have to connect to some unnumbered island to reach it, and said unnumbered island would render the combined island incapable of reaching the cell, the cell is deemed unreachable and therefore marked as a water cell. The combined island could be rendered incapable of reaching a cell either because its growth potential becomes too small to reach the cell, or because the unnumbered island is greater than the numbered island's growth potential, thereby disallowing connection. See figures 3.11c and 3.11d.



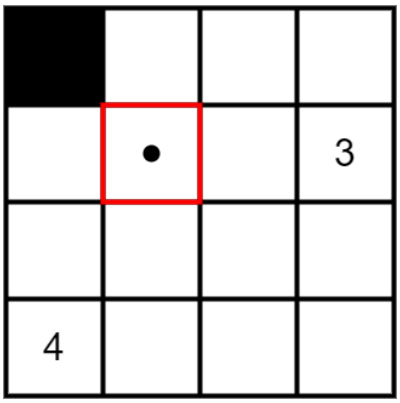
(a) Here, a numbered island with hint 4, and an unknown cell are marked by a red outline. The numbered island has a growth potential of $4 - 1 = 3$ cells, meaning the marked unknown cell is within reach, along the path marked by a red line. Also note that the unknown cell is unreachable to the 3-hinted island.



(b) Here the previously unknown cell has been marked as a water cell, as it could not be reached by the 4-hinted island without connecting it to the 3-hinted island.



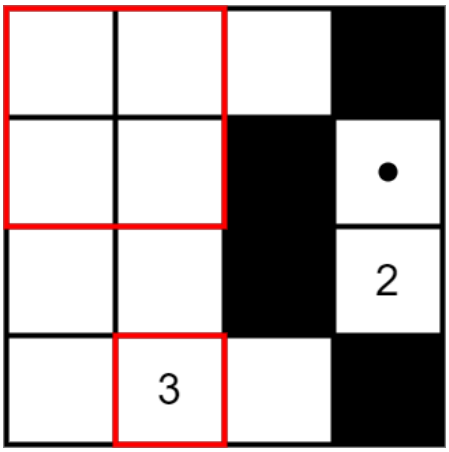
(c) Here, a 4-hinted island, and an unknown cell are marked by red outlines. As before the numbered island has a growth potential of 3, and the marked unknown cell is technically within reach. Note that the marked unknown cell is unreachable to the unmarked 3-hinted island.



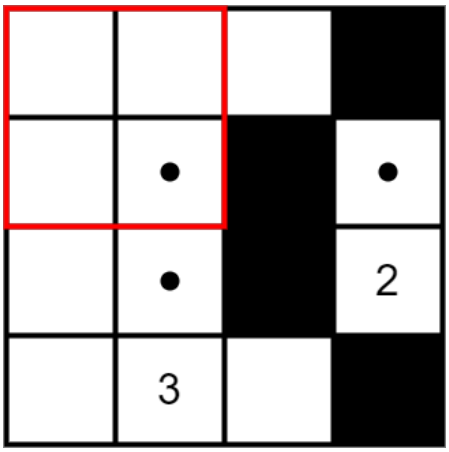
(d) Here, the previously unknown cell has been marked as a water cell, as it could not be reached by the 4-hinted cell without connecting it to a separate unnumbered island, marked by a red outline. This connection would force the 4-hinted island to have a size of 5 to reach the unknown cell, thus breaking the puzzle.

Figure 3.11: A part of a simple Nurikabe board, exemplifying the **unreachable cells 2** technique. Note that figure 3.11b corresponds to a single step in the solution of figure 3.11a, and likewise for 3.11d and 3.11c.

6. **Pool avoidance 2:** If a 2×2 square of water- or unknown cells consists of cells that are reachable to at most a single, common numbered island, said island has to encompass at least one of the unknown cells to prevent a 2×2 pool of water cells. This is a generalization of beginner technique 6
    a. As a special case of this, if only a single possible path exists for the 2×2 square of water- or unknown cells to be reachable, said path is entirely composed of island cells. See figure 3.12 for an example of this.



(a) a 3-hinted island of size 1, as well as a 2×2 square of unknown cells have been marked by a red outline.



(b) Here, the path connecting the 3-hinted island to the 2×2 square of unknown cells has been marked as island cells, as this was the only way for the 3-hinted island to reach the square, and thereby prevent a 2×2 pool of water cells.

Figure 3.12: A part of a simple Nurikabe board, exemplifying the **pool avoidance 2** technique.

3.2 The puzzle generation algorithm

Developing the Nurikabe puzzle generator algorithm, which needed to be capable of generating uniquely solvable puzzles, posed several challenges. Significant among these was the challenge of how to guarantee that a generated puzzle not only had a unique solution, but a unique solution that was possible for a human solver to find. A computer running a brute force algorithm would be capable of finding the solution to any solvable puzzle through simple trial and error if given enough time, even if this would be an NP-hard problem. A human solver on the other hand would not have this ability, and so measures would need to be taken to ensure that any generated puzzle was humanly solvable. These measures took the form of the solution techniques described in section 3.1. If these techniques were somehow encoded in way that a computer could use them, they could be used to verify that a puzzle would be solvable by a human solver.

Another issue was how to control the difficulty of a generated puzzle. A very simple solution to this problem would be to only consider board size when determining difficulty, and simply state that larger boards make for more difficult puzzles. It was however desirable to extend this solution somewhat, to give greater control of the difficulty of a puzzle, but also to limit the size puzzles would have to be to be considered difficult. Once again the answer came in the form of the techniques described in section 3.1, as a puzzle requiring more complex solution techniques intuitively is more difficult to solve.

For the actual generator algorithm, it was decided to take a two-part approach: a completed puzzle would first need to be generated according to the rules described in section 2.2.1. The algorithm would then have to place numbered cells and remove all cell markings, while simultaneously ensuring that the resulting puzzle was uniquely solvable by a human. These two sub-problems will be explored in the following sections.

Note that these sections only cover the general methodology of the generator algorithm. For the actual implementation of this algorithm in Haskell, see section 3.3.

3.2.1 Completed board generation

Two approaches were considered to generate a completed puzzle board. Common to both was that a board size was given, presumably by some difficulty selection in the app.

Starting with an empty board

The first approach involved starting with an empty board, i.e. a board of all unknown cells. A random cell would then be selected as a form of “seed cell”, and marked as a water cell. The water region would then be “grown” randomly from this cell. This would be done by randomly assigning all cells orthogonally adjacent to a water cell as either water or island cells in a recursive manner, until there

were no more cells for the water region to grow into. This assignment would have to be made in such a manner that if the water region at any point could only expand into a single cell, that cell would have to be a water cell. If any unknown cells remained after this recursion, they would all be assigned island cells. This had to be done because any cell still unknown after the recursion would not be orthogonally adjacent to any water cells, and therefore could not be a water cell without creating an isolated water region.

The benefits of this approach would be that the water region would be contiguous by design, as a water cell could only be assigned in a position orthogonally adjacent to another water cell. It would also be relatively easy to ensure that no 2×2 pools of water cells were generated by simply checking, whenever a cell was to be assigned, if that cell would form a 2×2 area of water cells, and if so assigning the cell as an island cell.

On the other hand, this method would present some challenges as well. There would, for instance, be no way of directly controlling the size of islands, nor the number. There would also be a chance that the algorithm would get stuck, in that it might assign island cells in such a way that the water region becomes unable to grow further, as can be seen below in figure 3.13. This would in turn result in unacceptably large islands. These problems could however be alleviated; either by modifying the probability of an assigned cell becoming a water cell, thus affecting both the size and number of islands stochastically, or by checking island sizes after finishing the recursive call, and rerunning the algorithm on any islands deemed to large by some metric, by converting their cells back to unknown cells and restarting the recursive call.

Starting with a board of all water cells

The second approach that could produce a completed board involved starting with a board of all water cells, and then generating islands randomly within that board. This could for example be done by recursively checking for 2×2 pools of water cells and if one was found, making one of its cells an island cell, before growing the island to some predetermined size from there.

The advantages of this approach would be that it would grant a much higher degree of control over the size and shape of islands, which might in turn grant greater control over the difficulty of the final puzzles. This approach would also guarantee that no “pools” exist by design.

There would however also be some significant drawbacks to using this method compared to the first one. Chief among these would be the fact that this approach would not guarantee a single, contiguous water region on its own. This would have to be rectified somehow, perhaps by checking, for each island cell assigned, if the assignment created an isolated water region. If so, the last assignment would have to be undone. Whatever form this rectification would take, it would likely interfere with an island’s ability to grow to its predetermined size, as there might be no legal way to expand an island to its full, predetermined size without creating

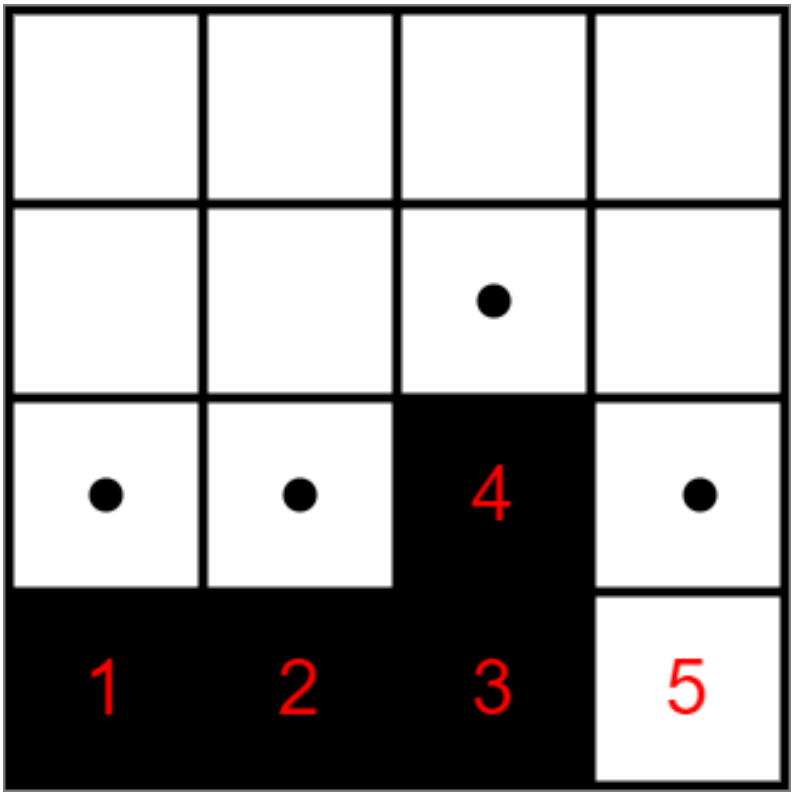


Figure 3.13: The water region in this Nurikabe board has become stuck. The numbers in red describe the order in which the water cells have been assigned. Note that after the assignment of cell 4, the cell above it could still be assigned as an island cell, as the water region could still expand to cell 5. When cell 5 is then assigned however, the region is unable to grow further, and we end up with an unacceptably large island at the top of the board.

isolated water regions. This might not be a significant problem, but it would lessen the degree of control over island size and shape that this approach initially benefit from.

Due to the major drawbacks with the latter of these two approaches, it was decided to move forward with the first approach, letting the water region grow from an empty board.

3.2.2 Placing numbers

As with the completed board generation, two approaches were considered for how to place numbered cells on the board to finalize the puzzle. These approaches are very similar in nature, and both relied on assigning numbers, before removing some information and checking if the resulting board was solvable using the techniques described in section 3.1. Both methods would therefore rely on a solver, using the aforementioned techniques, being implemented.

Removing all markings

The first approach would start with a completed board, produced by the approach described in the previous section (3.2.1). Each island in this board would then have a single numbered cell assigned, with each numbered cell's number equal to the size of the island, for obvious reasons. All other markings would then be removed from the board, and the aforementioned solver would attempt to solve it. If the solver succeeded in solving the board, the numbered cell assignments would be kept, and the puzzle would be finished. If the solver was unable to solve the puzzle on the other hand, that would indicate that the given numbered cell assignments did not produce a human-solvable puzzle. The numbered cells would then be reassigned, and the solver would retry. This would continue until either the solver succeeded in solving the puzzle, thus producing a finished puzzle, or until all combinations of numbered cell placements had been attempted, at which point the puzzle could be concluded to be insolvable by a human regardless of numbered cell assignments, and the generator would start from scratch with a new board.

Intuitively, it seems this approach might allow for finished puzzles with multiple possible solutions, as it at no point checks for them. However, if the solver used strictly adheres to the techniques described in section 3.1, it would be incapable of solving a puzzle with multiple solutions, as it only assigns cells a type when they are absolutely certain to have that type. In other words, if there are any ambiguous cells, as there would have to be for multiple solutions to exist, the solver would be unable to assign them, and so would deem the puzzle unsolvable.

Removing markings one at a time

The second approach to placing numbered cells would start much like the first by randomly assigning numbered cells in islands, once again with each numbered cell's number equal to the size of its island. This approach would then randomly remove a single marking from the board before checking, using the solver, whether the resulting board would be solvable. If so, the generator would remove a single new marking, over and over, until either all markings had been removed, and the finished puzzle had been proven solvable, or until a marking was removed that rendered the board unsolvable. If this occurred, the generator would return to the completed board, and reassign the numbered cells, before retrying. And as with the previous approach, this would go on until either a combination of numbers was found that resulted in a solvable puzzle, or until all combinations had been attempted and a new board would be attempted.

This approach might prove more time-efficient in some cases where a board might very quickly be determined to be unsolvable. In most cases however, this approach would be vastly more inefficient than the first approach, due to the sheer number of calculations needed. An improvement might be made by stating that the algorithm need not check that a board is solvable for each removed marking, but rather only check that the board may be returned to its previous state (I.e. with

the last removed marking still in place). This would however drastically increase the implementation complexity. It would also make it much harder to guarantee that a puzzle deemed unsolvable by the algorithm was actually unsolvable, and not just unsolvable through the specific path checked by the algorithm.

Due to the relative simplicity of the first approach, it was decided to proceed with this, and attempting to solve each instance of a complete board from scratch, rather than iteratively.

3.3 The puzzle generator implementation

As explained in chapter 1, it was decided to implement the Nurikabe generator and solver in Haskell. In this section, the process of implementing the different parts of the generator will be explained, including the more prominent challenges and problems that arose, as well as their solutions.

3.3.1 Data structures

The first step taken in implementing the puzzle generator was defining several data structures to simplify and clarify the operations performed. These data structures were, to varying degrees, based on Abstract Data Types (ADTs) that could be extracted from the rules of Nurikabe as explained above in section 2.2.1. These ADTs were:

- **Cell:** An ADT describing a single cell in a Nurikabe board, in terms of its position, its type (I.e. island, water or unknown), and its number, if it contains one.
- **Board:** An ADT describing an entire board of cells, of varying size.

Along with these ADTs, several other data structures and -types were defined, to clarify the implementation, and to provide consistent and suitable levels of abstraction. Examples of these structures and types are:

- **Coordinate:** A type alias consisting of a tuple of two integers, denoting a position in a Nurikabe board as a coordinate.
- **CellType** A custom data type taking one of three possible values: White, Grey or Black, used to denote the type of a cell. Note that for this data type, a color based typing of cells was used, rather than the more formal typing used in this report, to help with visualizing results. The colors chosen correspond to the colors of a physical Nurikabe board, with Black corresponding to water cells, White to Island cells and Grey to unknowns. Note again the need for a Grey cell type to denote cells that are as yet of undetermined type, even though in an actual board, such cells would in fact be white.
- **Region** a type alias consisting of a list of cells. This was used to denote a connected region of cells of identical type in a board.
- **Island** a type alias consisting of a Region. Note that though this type was, by definition, completely interchangeable with the Region type above, it was

included to differentiate between regions that could be of any type, and regions that were known to be islands.

Having defined these data structures and -types, the ADTs above could easily be implemented; the *Cell* as a custom data type containing a *Coordinate*, a *CellType* and potentially a number. Seeing as not all Cells would have numbers, this was implemented as a *Maybe Int*, to provide clarity and to avoid having to use a specific number to denote unnumbered cells. The actual definition of the Cell data type can be seen below:

```
data Cell = Cell {position :: Coordinate,
    number :: Maybe Int,
    cellType :: CellType} deriving(Eq)
```

The CellType data type was simply declared as `data CellType = White | Grey | Black`.

The *Board* type was implemented as a type alias for a two-dimensional array of Cells, I.e. as a list of lists of Cells. In practice, this was defined simply as `type Board = [[Cell]]`.

3.3.2 The completed-board generator

With all the necessary data structures and types defined, it was time to develop the first part of the generator algorithm, as detailed above in section 3.2.1. For the most part, this implementation matched the procedure explained in said section; the entire water region was grown from a single cell, using a random number generator to decide which Grey cell to expand to, and in some cases to decide what type it should be assigned. Some additions were made however, to improve the quality of the produced board, in the hopes that it would produce a better puzzle. The first of these additions was to check the size of the largest island in the board, after a completed board had been produced. If this island was found to be larger than some number (which was eventually chosen to be equal to the length of the shortest side of the board) the entire island would be converted back to Grey cells, before restarting the generation, thus splitting the island into smaller islands. This was done to alleviate the point made in section 3.2.1, that unacceptably large islands could result from the algorithm becoming stuck with only a single cell left to expand to.

Another addition that was made, was to add another restriction to the assignment of cell types. Initially, the method would decide the type of a cell according to this rule: if the cell was part of a potential pool of water cells, i.e. it was surrounded by an L-shape of black cells, the cell would be assigned White. Otherwise, if the cell was the only cell the Black region could expand to, it would be assigned Black. Lastly, if neither of these conditions were met, the cell was assigned a type randomly based on some probability distribution passed in to the method. This distribution was found to have adequate results at a value of 0.75, meaning cells were assigned as Black three times for every time on was assigned as White. After

implementing this method, it was discovered that the shape of islands, and not only their sizes, were important to the solvability of a board, with longer, narrower islands more often resulting in solvable boards. It was therefore decided to add a third check to the assignment of cell types, before letting the cell be randomly assigned. This check consisted of checking the neighbourhood of the assigned cell for L-shapes of White cells, much like the the first check checked for pools of Black cells. If a cell was inside such a shape, it would be assigned black, thus preventing larger sections of White cells from forming. By adding this check after the two first checks in the cell assignment, the method could favour longer narrower islands, without compromising the more strict rules of the board. The final implementation of the CellType assignment looked as follows:

```
cellType
    | isCellPotentialPool board pos = White
    | length expandableNeighborhood == 1 = Black
    | isCellPotentialDryLand board pos = Black
    | otherwise = randomType
```

3.3.3 Implemented techniques

The next step in developing the generator, was to implement the solution techniques. Unfortunately only the beginner techniques described in section 3.1.1 were implemented, due to a lack of time, meaning that the developed solver would only be capable of solving relatively simple boards. The following sections describe how the different techniques were implemented. The code implementations of the techniques can be found in the attached code. See appendix B.1 for the structure of this code.

Completed islands

The completed islands technique was implemented simply by checking the size of a given island, and, if the size corresponded to a numbered cell in the island, to mark all unknown cells orthogonally adjacent to the island as water cells.

Island separation

The island separation technique was somewhat more complex in its implementation. An elegant solution was however found to be checking the unknown cells bordering every island in the board; any cell appearing in the border of more than one island was marked as water. Care had to be taken to only check the borders of numbered islands however, as known island cells could appear in the board due to other techniques, such as the pool avoidance technique. These unnumbered islands could then not be separated from other islands, as they would have to join up with a numbered island to complete the board.

Island- and water expansion

The island- and water expansion techniques were implemented by grouping all known island- and water cells into regions and checking their borders, i.e. their orthogonally adjacent neighbour cells, for any cells the region could be expanded into. Any region that was incomplete and could only expand to a single cell was expanded to said cell. Some issues with this first implementation quickly presented themselves however. One such issue, resulting from the fact that all regions had to be mapped out before any were expanded, was that when a region was expanded, it might connect itself to another region, thus altering the resulting region's size and shape. The first implementation did not check for this, and so would occasionally end up erroneously expanding a region.

A second issue arose from the fact that the first implementation expanded regions in random order. This, combined with the phenomenon described above wherein a region might not be the same when being expanded as it was when discovered, meant that the water region might be completed before all discovered water regions were expanded, resulting in an already complete water region being expanded into a cell that was in actuality supposed to be an island cell.

These issues were easily fixed by ensuring that any region to be expanded was first mapped out entirely to compensate for any changes to the regions size at its discovery. An additional check of the completeness of the region was then made to guarantee that a complete region could not be expanded. This completeness check for islands merely consisted of checking whether a numbered islands size corresponded to its hint, whereas for water regions, it consisted of checking whether only a single water region existed, and if so checking whether its size was equal to the total size of the board minus the sum of all hints in the board.

Surrounded cell

The surrounded cell technique was simply implemented by checking the orthogonal neighbours of every unknown cell in the board. If all such neighbours of an unknown cell were of the same type (and not unknown), the unknown cell was marked as the same type as all of its neighbours.

Pool avoidance

The pool avoidance technique was likewise implemented by checking the orthogonal and diagonal neighbourhood of every unknown cell for L-patterns, such as the ones seen above in figure 3.5. As this pattern indicated a potential 2×2 pool of water cells, any cell found to have such a pattern in its neighbourhood was marked as an island cell.

Unreachable cell

This technique proved quite a bit more complex in its implementation than the others implemented up to this point. Initially an attempt at implementation was made where every unknown cell had its Manhattan distance to every numbered island in the board calculated. If an unknown cell was found, where, for all numbered islands, the distance between the cell and the island was greater than the growth potential of the island, the unknown cell would be considered unreachable, and marked Black. While this did not result in any incorrect cell markings, the simplifications made in implementation meant the method was of little to no practical value.

A more complex approach was therefore taken, where the area around each unknown cell was expanded through all connected unknown cells incrementally, and at each step comparing the distance traversed to the growth potential of any islands encountered at that step. If any island was found with a growth potential greater than or equal to the distance traversed, the cell was deemed reachable, and the algorithm stopped. If the expansion continued until no more steps could be taken, i.e. the entire area reachable by the cell through unknown cells was discovered, the cell was deemed unreachable, and marked Black.

This implementation proved much more valuable than the last, but it was later discovered that it could in fact wrongly mark cells as unreachable, due to the fact that it disallowed expansion through unnumbered islands. This meant that islands could be deemed unreachable when they were in fact reachable, as can be seen below in figure 3.14. Fixing this issue once it had been identified was a simple matter of allowing the method to expand through island cells belonging to unnumbered islands, resulting in a correct implementation once again.

3.3.4 The Nurikabe solver

Having implemented a sufficient number of techniques to solve simple Nurikabe boards, these techniques were then combined into two different solvers: an “easy” solver, that utilized the *completed islands*, *island separation*, and *island-* and *water expansion* techniques, and a “medium” solver that utilized the same techniques, as well as the *surrounded cell*, *pool avoidance*, and *unreachable cell* techniques.

The solvers were implemented recursively in a step-wise manner. Each of these steps consisted of chaining together the utilized techniques in order of their appearance in section 3.1.1. This chaining was performed in such a way that if a technique produced an altered board, meaning the technique had successfully performed a step in the soltution of the board and marked some cells as either Black or White, the produced board was returned without performing subsequent techniques. This was done because the earlier techniques were, in general, simpler in their implementation, meaning that utilizing these techniques to a larger degree would result in a potentially faster solver.

The board returned from such a step was then checked for completion, by checking the total number of Grey cells in the board; a board with no Grey cells

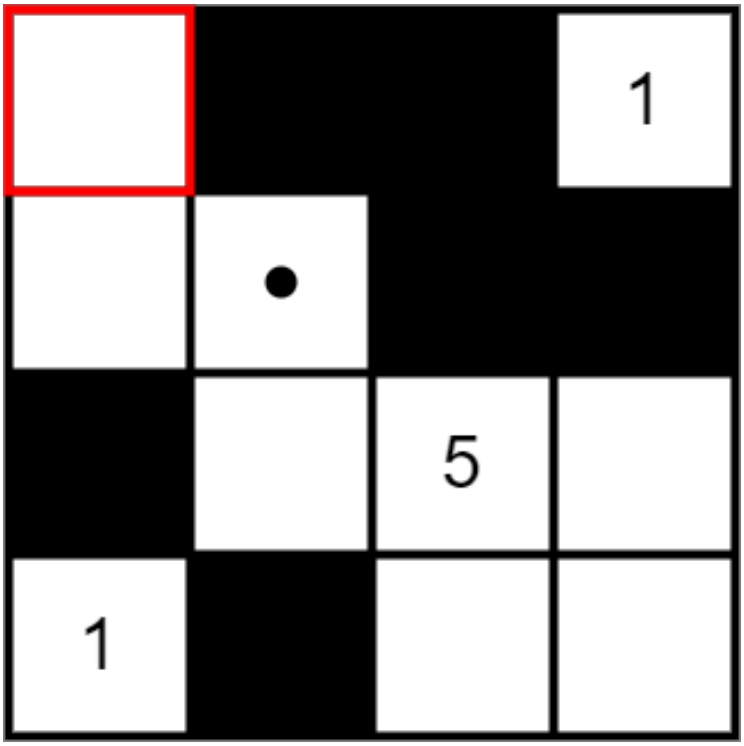


Figure 3.14: An example Nurikabe board with a falsely unreachable cell. Using the second implementation of the **unreachable cell** technique explained above, the unknown cell marked by a red outline would be considered unreachable, due to the unnumbered island positioned between the 5-hint and the unknown cell. In reality, this cell would be very much reachable, by joining the 5-hinted island with the unnumbered island.

was considered solved. If the board was solved, it was returned from the solver in its solved form, and if not, another step through the chain of techniques was performed. If a step at any point returned an unaltered board, indicating that none of the techniques had managed to alter the board, the board was considered unsolvable, and an empty board (i.e. an empty list) was returned from the solver.

To keep track of whether or not a board had been changed by a technique, a data type, called *Monitor* was implemented with two value constructors: *Changed* and *Unchanged*. These value constructors could then be wrapped around some other type; in this case, a *Board*. The reason for implementing this data type was to improve the efficiency of the solvers, by not having to perform multiple comparisons of two boards in each step of the solver to determine whether the board had been changed. Instead, the Monitor constructor of the board could simply be checked.

In addition to implementing the above data type, it was desirable to make it an instance of the *Monad* type class, in order to simplify the chaining of techniques in each solver step. This proved impossible however, due to the fact that it was de-

sirable for any operation on an already changed board to simply return the same changed board with no modifications, to conform with the reasoning explained above. This was incompatible with the Monad type class' declaration of the bind function, `(>>=) :: m a -> (a -> m b) -> m b`, as there was no way of making the desired definition, `Changed a >>= _ = Changed a`, conform with this type declaration; it would return a monad of the same type as was input, rather than of some other type *b*, defined by the input function.

Because the Monad type class was inapplicable to the situation at hand, another type class, the *Modifiable* type class, was implemented, that closely matched the functionality of the Monad type class, but allowing instances to be of a single type. It was defined as follows:

```
class Modifiable m where
(>\=) :: m a -> (a -> m a) -> m a
(>\\=) :: m a -> (a -> m a) -> m a
extract :: m a -> a
```

As can be seen, this type class defined two functions, `(>\=)` and `(>\\=)`, as well as an *extract* function that would simply return the value from the "pseudo monad". Both the `(>\=)` function and the `(>\\=)` function were intended to function much the same as the Monad type class' bind function; the `(>\=)` function would only allow the alteration of Unchanged variables, whereas the `(>\\=)` function would allow the alteration of any Monitor variable, but would keep a Changed constructor regardless of whether or not the applied function modified the value. The Monitor data type instanced the Modifiable type class by implementing the above functions as follows:

```
instance Modifiable Monitor where
(>\=) (Changed x) _ = Changed x
(>\=) (Unchanged x) f = f x

(>\\=) (Changed x) f = Changed $ extract (f x)
(>\\=) (Unchanged x) f = f x

extract (Changed a) = a
extract (Unchanged a) = a
```

Using this new type class, the solvers could elegantly chain techniques in the desired way. As an example of this, a step of the "medium" solver was simply implemented as follows:

```
mediumStep board =
    let
        completedBoard = complete board
        separatedBoard = completedBoard >\= separate
        expandedBoard = separatedBoard >\= expand
        surroundedBoard = expandedBoard >\= surround
        poolAvertedBoard = surroundedBoard >\= avertPools
        unreachableBoard = poolAvertedBoard >\= unreachable
        result = unreachableBoard
    in
        result
```

The Monitor data type and the Modifiable type class, as well as the fully implemented solvers, can be found in the attached code. See appendix B.1 for the structure of this code.

3.3.5 The full generator

Having implemented both the completed board generator, and the needed solvers, it was time to combine them into a generator capable of generating a solvable Nurikabe puzzle from scratch. Initially it was attempted to simply generate a board, using the complete board generator from section 3.3.2, randomly placing hints in the generated islands, and then attempting to solve the resulting board.

More often than not, this resulted in unsolvable boards, as only a single placement of hints was attempted. The generator was therefore enhanced to attempt every possible placement of hints in order, and returning the first one that proved solvable. This somewhat improved the generator, making it result in solvable boards more often, and was so used in a first version of the generator. This would also keep generating new boards until a solvable board was found, in order for the generator to always return a solvable board.

It was later found that this implementation was still too unreliable, and spent unacceptably long attempting hint placements in inherently unsolvable boards. This was especially true for larger boards. A second version of the solver was therefore implemented with an added reliability insurance. This implementation would, as before, generate a completed board, before attempting all possible placements of hints. However, this implementation would not give up if a board was found to be unsolvable. Instead, it would attempt to make the board solvable. To do this, any board found to be unsolvable would have a single island cell, taken from the largest island in the board, reassigned as a water cell, making sure to not violate the rules of the game. This measure was implemented because experience showed that boards with larger islands were in general harder to solve, and therefore more often resulted in unsolvable boards. Splitting the largest island in a board was therefore deemed a fitting way to improve the chances of finding a solvable placement of hints.

If, after reassigning an island cell as water, the board was still unsolvable, the search would continue in a recursive, breadth-first manner. In other words, a new island cell would be reassigned as water, before again attempting all possible hint placements. After all legal island cells had been attempted reassigned, two island cells would be reassigned, and so on.

Though this added complexity did increase the worst-case run time for generating a single board, in practice, the added reliability more than made up for this, by ensuring that generator would run far fewer times. In most cases, the generator would now produce a solvable board for the first generated completed board. The generator was therefore considered complete to be used by the app.

The final generator was split into three different functions in Haskell: *generateEasySeededBoard*, *generateMediumSeededBoard* and *generateHardSeededBoard*. Each of these functions took three inputs: a board size, in the form of a tuple of two *Int*s, a distribution in the form of a *Double*, that described the relative probability of a cell being randomly assigned as a water cell, and a seed, for the random generators, in the form of an *Int*. Each of the three functions returned a tuple of consisting of to values of type *Board*, where one value represented the generated, unsolved board, and the other the solved version of the same board. Note that though the entrypoint for the hard generator was completed, its solver, and therefore generator, was left unfinished due to a lack of time.

3.4 The puzzle generator web app

To interface between the game app, and the generator, it was, as mentioned in chapter 1, decided to implement the generator in a web app, callable by the app. This web app was built using ASP:NET Core, and written in C#. It consisted of three API endpoints: */GenerateSeeded*, */GenerateUnseeded* and */GenerateMultipleBoards*.

All of these endpoints took several input values, through the query. Common for all the endpoints were: a string indicating the desired difficulty of the generated board, i.e. either “Easy”, “Medium” or “Hard”, two integers defining the size of the board to be generated, and a single integer, with default value 10, indicating how long, in seconds, the generator should attempt to generate a board for before giving up. The */GenerateSeededBoard* additionally took an integer seed as input, and the */GenerateMultipleBoards* took an integer number of boards to generate. Note that the difficulty of boards was passed as a string, in spite of being used as an *enumerated type*, or *enum*, in the actual program. This was done to avoid having to use literals (sometimes called “magic numbers”), that in no way signalled their significance, in the API as enums are typically treated as integer values.

3.4.1 Haskell DLLs

In order for the web app to be able to call the generator functions from Haskell, some way of interfacing between the two had to be utilized. This took the form of .NET's Platform Invoke, or *P/Invoke* as it is commonly called. This technology, which is a part of the .NET Standard library, allows managed code, in this case the ASP.NET web app, to call functions from unmanaged code, in this case the Haskell generator functions.

For the Haskell functions to be callable to the C# code through the use of P/Invoke, the Haskell program had to first be compiled into a Dynamic Link Library (DLL). This was done relatively simply by adding a few flags to the Haskell compiler, GHC. Returning boards from the generator to be used in C# code quickly proved challenging however, as the variable size boards couldn't be returned directly. Instead, the boards had to be written to, and kept in, memory in Haskell, and a pointer to the memory area occupied by the board returned to C#. The boards then had to be read from memory in C#, before the memory could finally be freed in Haskell.

As explained in section 2.3.3 however, memory management is quite different in Haskell and C-family languages. This meant that simply writing the data structures to memory was not enough for them to be accessible in C#; first they had to be made to comply with the C way of handling memory. This was no easy feat, as it involved converting the previously defined data types to types consisting of C-compliant base types, and to then make them referenceable by instancing the *Storable* type class.

Creating the C-compliant types was, in itself, no major challenge, as it mostly involved rewriting the previously defined types using the types imported from the *Foreign.C* library. Making these types referenceable proved more of a challenge however, as doing so manually would require intimate knowledge of the workings of Haskell pointers and memory management. As this was not something the candidate possessed, it was instead decided to utilize a tool, in the form of a command, included with GHC: *hsc2hs*.

To use this tool, a *.c* and *.h* file, called *ExportTypes.c* and *ExportTypes.h* respectively, were first created. In these files, structs corresponding to the data types to be exported from Haskell, were created. These files were then imported into a *.hsc* file (*ExportTypes.hsc*), serving as a sort of bridge between Haskell and C. In this file, the new data types, using *Foreign.C* base types, were defined, and made instances of the Storable type class by referencing the structs defined in C and using some syntax specific to *.hsc* files. Calling the *hsc2hs* command with this file as input produced a normal Haskell file, called *ExportTypes.hs* which correctly defined the desired data types, and made them instances of Storable. This file could then be imported by the Haskell program, and its data types used to interface between Haskell and C#.

With these types complete, all that remained was to define the functions that would make up the actual interface. This was done in the *HaskellInterface.hs* file.

Here, the functions to generate boards were declared and implemented, as well as helper functions to allow allocation and freeing of memory for boards, and functions to allow accessing boards and cells from pointers. The entire program was then compiled into a *.dll* file, ready to be called from C#.

Moving on to the C# side of things, P/invoke could now be used to import functions from the Haskell interface. To do this, a Haskell run-time first had to be initialized, using the *hs_init()* function that was exposed in the DLL by default. To ensure this initialization was only executed once, and to hide the implementation details from the rest of the .NET web app, all communication with the Haskell DLL was performed from within a singleton class, called *HaskellRuntimeSingleton*. This class also exposed all the functions needed from the Haskell interface to the rest of the .NET app.

Boards were then generated in the web app, by calling the appropriate generator function, which would return a pointer to a structure of two boards; one solved and one unsolved. This pointer was then used to get pointers to individual cells in each board from Haskell, before these cell pointers were unpacked into a *Cell* struct, and placed into the appropriate board, taking the form of two-dimensional arrays. All these operations were performed from within the *Board* class in C#, which also ensured that the memory occupied by the board in Haskell was freed once the class instance in C# was no longer in use.

3.4.2 Hosting

For the app to be able to generate boards on demand, the web app containing the generator would have to be hosted somewhere, in a way that allowed the app to consume it's API. To do this, several approaches were attempted, ranging from hosting the web app as a server-less API in Amazon Web Services (AWS), to hosting it as a full on server.

The Haskell interface DLL made this difficult however, due to the requirements it imposed on the machine running it. In the end, none of the attempted hosting methods proved capable of calling functions from the DLL, not even when it was compiled as a *.so* file and run on Linux. It was therefore decided to cut losses, and simply manually export some generated boards to the final app, for demonstration purposes.

3.5 The "MasterGame" app

As explained in chapter 1, the Nurikabe game app, eventually simply called "MasterGame", was implemented using the Xamarin framework. Specifically, it was implemented in Xamarin.Forms, the cross-platform implementation of Xamarin. The app was furthermore developed using the MVVM architecture, which meant that it was separated into models, views and view models. The source code for the app can be found in the attached code file, the structure of which is explained in appendix B.1.

3.5.1 Models

The app contained several model classes, mostly concerned with cells, boards and their interactions. To implement cells in the app, three separate classes had to be implemented: a *BoxCell* class, a *LabelCell* class and an *ImageCell* class. These were separated in order to properly utilize the different Xamarin views that would be needed; BoxViews for cells that were uniformly black or white, i.e. water- or unknown cells, Images for dotted cells, i.e. Island cells, and Labels for hint cells. A third cell class, *NurikabeCell* was also implemented to combine the different types of cells, so that a cell could change from one to another. This was done by making an instance of the NurikabeCell class contain either an instance of LabelCell, if it was a numbered cell, or one instance of BoxCell and one of ImageCell if it was not. By detecting long and short presses on the cell (if it was not numbered) the NurikabeCell could then be toggled between displaying the ImageCell, signifying it was an island cell, or a black or white BoxCell, indicating it was a water- or unknown cell respectively.

The board functionality was also split into three parts. A DTO class, called *NurikabeBoardDto* was used to deserialize boards returned by the generator. The *Board* class, that was initially intended to be used to call the web API, was later repurposed to reading boards from local *.json* files. This class also maintained the actual board arrays themselves, and handled operations on them, for instance in connection with solving. That is, when a board was being solved in the app, the Board class would keep track of the state of the board being solved, by modifying the types of the boards cells. The Board class was in turn used by the higher-level *NurikabeBoard* class, that inherited from the Xamarin Grid view. This class handled boards on the app level, by controlling their sizes, and what type of cell should be displayed where in the grid.

Lastly, there was the *GeneratorClient* class, which was intended to be used to consume the web API of the generator. When it was decided to not implement any form of hosting of the generator API however, this class was no longer used.

3.5.2 Views

The views in the final app, of which there were six, were implemented by first defining the layout of the page in the .xaml file, before defining any needed logic in the code-behind file. As view models were also implemented, there was, for the most part, no need for any such logic, with these files containing only the default function calls handling page initialization. The “Nurikabe Menu Page”, serving as the main menu for the app, was the only exception to this, as it needed a few extra code lines in its code-behind to be called every time the page appeared. These code lines would ensure that some values in the page’s view model were correctly updated, to allow a “Continue” button to be displayed only under the correct circumstances.

All views in the app were implemented as ContentPages, and navigation between them was performed in part by the use of a flyout menu, and in part by using a nav-

igation stack. The flyout menu enabled navigation between the app's main menu page, and two information pages that displayed information about Nurikabe and its rules and some information about the context of the app.

Of these, the page containing information about Nurikabe and its rules proved somewhat tricky to implement, due to the amount of text displayed on the page. Initially, this page was implemented by displaying the rules in a numbered list in a ListView, which was nested inside a StackLayout along with the rest of the text, in the form of Labels. However, this meant that while most of the page was stationary, the part that contained the rules could be scrolled, which proved a very unpleasant user experience. The rule list was therefore updated to be a list of specially formatted Labels in a StackLayout, along with the rest of the text. This StackLayout was itself contained within a ScrollLayout, meaning that the entire page could be scrolled, rather than only parts of it, something that proved a much nicer user experience.

Where the flyout menu was used to navigate between the different information pages of the app, the navigation stack enabled navigation between the main menu and the different views used in selecting a board to be played, as well as in actually playing the board. This navigation was performed in the view model corresponding to each page, and was activated through the use of buttons on the page, that were, for the most part, defined in the .xaml file.

3.5.3 Viewmodels

The final app contained a view model for most of the views. Only the "About Nurikabe" and "About this project" pages lacked one, as these pages only contained text, and needed no business logic. For the most part, the view models were relatively simple, both in their functionality and implementation. They handled button commands, which mostly consisted of navigation using the navigation stack, and instantiated model classes to be displayed in the views. Some more complex logic was however also present.

In the view model corresponding to the "Nurikabe Game Page", inputs to the page had to be interpreted in a somewhat complex way. For instance, the view model had to decide whether or not to create a new NurikabeBoard instance, or if it should load a previously created instance. Furthermore, if a new board was to be created, i.e. read from file, the view model had to know what difficulty board to read, and at which index in the file. Functionality was also added that would allow the view model to generate a board by calling a web API, though this was, for reasons explained, not used. The app's "back" button also had to have its functionality overridden for this page, to ensure that the app always returned to the main menu after it being pressed. Lastly, this view model had to be able to read and write boards to and from the app's global context, in order to load and save a board in progress.

Likewise, the view model corresponding to the "Index Selection Page" also needed some more complex functionality, to determine how many levels were

available for a given difficulty, and then to show a button for each of those levels. This was implemented by calling a static function from the Board model class, which read the file corresponding to the difficulty passed to the index selection page, and returned the number of boards in it. A list of buttons, the same length as the number of boards found, was then instantiated, and each was bound to a command that would navigate to the game page, passing along the difficulty and index of the board to be displayed. The list of buttons was then displayed as Xamarin Buttons in a CollectionView in the .xaml file.

The view model corresponding to the Nurikabe menu page was somewhat simpler, but still required logic to determine whether or not to display the "Continue" button, as mentioned above.

3.5.4 Nurikabe implementation

The actual game logic was relatively straight forward to implement. It consisted of the NurikabeCell class detecting long and short presses on cells in the boards, and then changing said cell's state. This state change was performed both in the NurikabeCell class, to adjust how the cell was displayed, but also in the board array in the Board class, so that it would mirror the shown board. Any update to a cell would also trigger a solve check in the Board class, that would check if the unsolved board had placed water cells in exactly the same positions as in the solved board. If so, the board was considered solved, and a pop-up displayed to the user, before bringing them back to the main menu.

For complete instructions on how to navigate the app and play the game, see section 4.3. For instructions on how to install the app on an Android device, see appendix B.2.

Chapter 4

Results

In this chapter, the results of the project will be presented, both in regards to the Nurikabe puzzle generator, and the "MasterGame" app. For more abstract discussion about these results, and for results regarding the candidate's development as a software developer, see chapter 5.

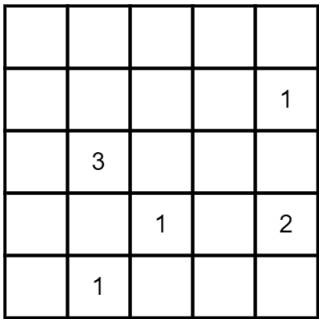
4.1 The puzzle generator

In this project, a Nurikabe generator, capable of generating puzzles solvable by humans, was created. The generator was written in a functional way, using the Haskell programming language. Overall, the generator therefore fulfills the objectives of the project in a satisfactory way. The source code for the generator can be found in the attached *.zip* file; see appendix B.1 for an overview of the structure of the attached code.

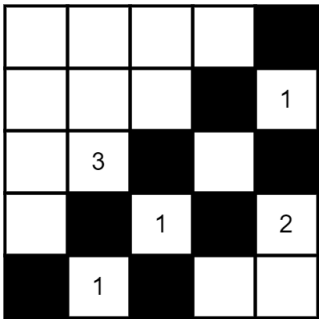
4.1.1 Difficulty of generated boards

The developed generator had fully implemented methods to generate "easy" and "medium" boards. Examples, displaying such generated boards, and demonstrating their solvability, can be found below in figures 4.1 and 4.2. Note that these difficulty levels were defined based on personal experience and convenience, rather than some more concrete difficulty measure, and other classifications might therefore be equally correct.

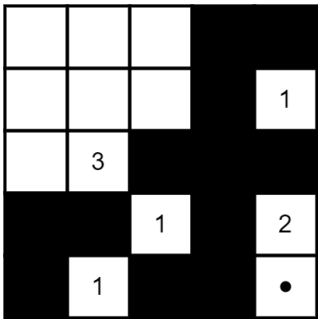
The "easy" board generator utilized techniques 1-4 in section 3.1.1, whereas the "medium" board generator utilized the full list of beginner techniques in the same section. Given more time, a "hard" board generator would also have been implemented using both the full list of beginner techniques in section 3.1.1, as well as the full list of advanced techniques, as described in section 3.1.2. The "medium" board generator would likely also be extended to utilize some of the easier of the advanced techniques. As it stands however, there was not enough time to implement the needed techniques, and the "hard" board generator therefore stands unfinished.



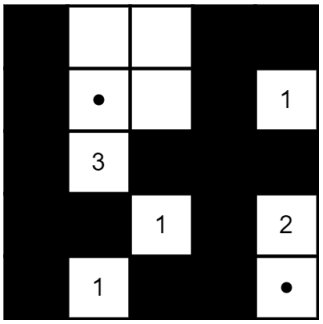
(a) The generated, unsolved board.



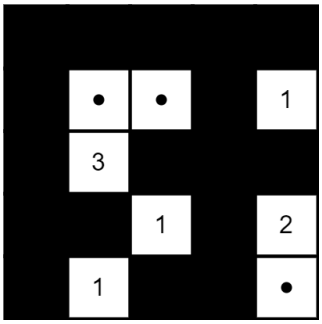
(b) Applying the **completed islands** technique allows all cells orthogonally adjacent to a 1-hinted cell to be marked as water.



(c) Applying the **island-** and **water expansion** techniques, allows the 2-hinted island to be completed, and some more water cells to be marked.

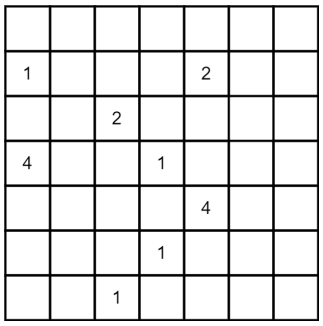


(d) Applying the **expansion** techniques yet again, now on the isolated water region on the left of the board, the 3-hinted island begins to form.

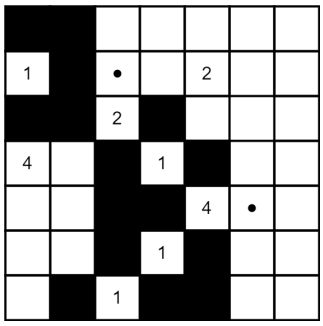


(e) By continuing to use the **expansion** techniques, the 3-hinted island, and with it the board, is completed.

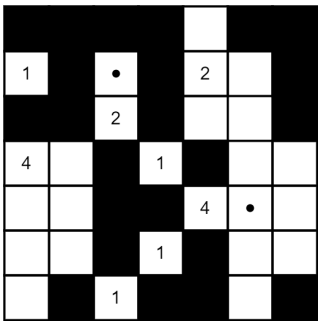
Figure 4.1: A Nurikabe board, of difficulty "easy", and size 5×5, generated by the developed generator, and solved using the techniques described in section 3.1.1



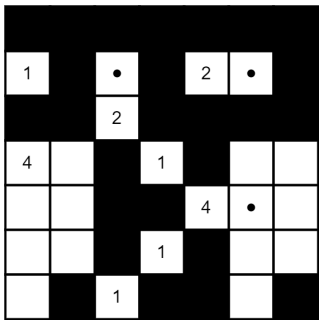
(a) The generated, unsolved board.



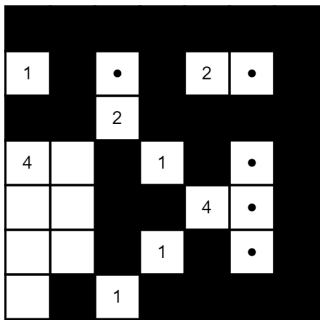
(b) Applying the **completed islands** technique allows all cells orthogonally adjacent to a 1-hinted cell to be marked as water. The **island-** and **water expansion** techniques then allow some more cells to be marked.



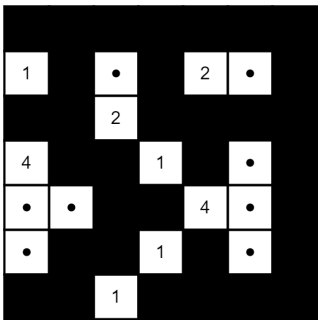
(c) Here, we use the **completed island** technique to mark the first 2-hinted island as complete. We also use the **unreachable cell** technique to mark some more cells as water in the right half of the board.



(d) We use the **pool aversion** technique, followed by the **completed island** technique to complete the second 2-hinted island.



(e) Applying the **pool avoidance** technique just below the second 2-hinted island, followed by the **unreachable cells technique** creates a new potential pool in the bottom right of the board. Avoiding this by using the **pool avoidance** technique completes the first 4-hinted island.



(f) By applying the **expansion** techniques, followed by the **pool aversion** technique, before applying the **expansion** techniques yet again, we complete the last 4-hinted island. With that, the board is complete.

Figure 4.2: A Nurikabe board, of difficulty "medium", and size 7×7, generated by the developed generator, and solved using the techniques described in section 3.1.1

Note that the developed generator in no way guarantees the difficulty of a generated board. In fact, as explained in section 3.3.5, a generated board might artificially be made easier, in order for it to be solvable. Comparing the boards in figures 4.1 and 4.2 however, it is clear to see that the "medium" board is noticeably more difficult to solve than the "wasy" board. This can be justified intuitively by simply looking at the boards. It can however also be justified more concretely; in the example solutions of the two boards, it is clear that the "medium" board takes more steps, i.e. technique applications, to solve, which would translate to a longer solve time for a human solver. Additionally, the "medium" board requires more different techniques to solve. This would also translate to a longer solve time for a human solver, as more time would be needed to decide which technique to apply in a given situation.

This difference in difficulty is, naturally, not unrelated to the size difference between the two boards. One could even go so far as to claim that the size difference is the sole explanation for the increased number of required solution steps in the "medium" board, though this would be hard to verify. Regardless, the difference in size cannot reasonably explain the larger number of different solution techniques required to solve the "medium" board. It can therefore be argued that the generator succeeds in generating boards of varying difficulty, and not only by modifying board size.

4.1.2 The performance of the generator

As far as the performance of the generator is concerned, it can be said to be satisfactory. Generating small to medium size boards is relatively quick; "easy" 5×5 boards are consistently generated in less than one second, and "medium" 7×7 boards rarely take longer than a few seconds to generate.

That said, the generator does not scale well with size, and even 8×8 boards of "medium" difficulty take drastically longer to generate. Also the variation in generation time seems to increase with the board size, meaning larger boards deviate more from their average generation time.

Many factors will contribute to this poor scalability. Presumably, the most significant of these is the fact that the generator relies on applying solution techniques to determine the solvability of a generated board. This means that the generator will necessarily scale very poorly, as solving a board in this manner is, as was shown in section 2.2.2, NP-hard. Few measures, short of proving that $P = NP$ (which is *far* beyond the scope of this project), would improve this scalability. A major redesign of the generator algorithm, removing the need for solving boards, might also serve to improve the scalability, but even at the conclusion of this project it is still uncertain whether such an algorithm could even possibly exist.

Two other factors that contribute significantly to the scalability issue can however be identified, which might be more easily amendable. The first of these is the island size restriction implemented in the generator, as described in section 3.3.2. This restriction allows larger boards to contain larger islands, something that,

from experience, has a two-sided effect: For a given set of solution techniques, it decreases the chance that a generated board is solvable, for any assignment of hints. Conversely, it also means that solving a larger board is more likely to require more advanced techniques. This, in turn, means that the generator will be less likely to deem a board solvable the larger it is, thus increasing the average generation time.

The second factor, which is closely linked with the first, is the reliability measure implemented as described in section 3.3.5. As mentioned in said section, this measure increased the worst-case run-time of the generator, in order to increase it's reliability. That said, the average run-time was observed to decrease for reasonably sized boards, i.e. about 5×5 for "easy" boards and 7×7 for "medium" boards. When increasing the board size however, the run-time quickly worsened with this reliability measure implemented. This is likely because, as mentioned above, the odds of a board being solvable for a given set of techniques decreases as the size of the board increases. Without adding more solution techniques, simply increasing the size of the board will in other words often result in a board that remains unsolvable in spite of the reliability measure's best efforts, thus increasing the run-time. This is evidenced by the fact that generating an "easy" board of size 7×7 takes drastically longer than generating a "medium" board of the same size. Even in situations where the reliability measure was able to make a board solvable, this would likely require more reassignments of cells for larger boards, something that, given the measure's poor worst-case run-time, would worsen generation times.

Based on the factors explained above, some improvements can be suggested, that might improve the scalability of the generator. The most important of these would be to implement the remaining solution techniques described in section 3.1.2, and use these when generating larger boards. As mentioned above, 7×7 boards are drastically faster to generate using the full set of beginner techniques, i.e. "medium" difficulty, than with only a smaller subset, i.e. "easy" difficulty. This means it stands to reason that larger boards will similarly be generated faster with more solution techniques available.

Additionally, redesigning the restriction on maximum island size, either by having a more variable limit, or by implementing some other restriction, perhaps on island shape as well as size, might increase the odds of a generated board being solvable. This would in turn reduce the generation time for boards of a given size.

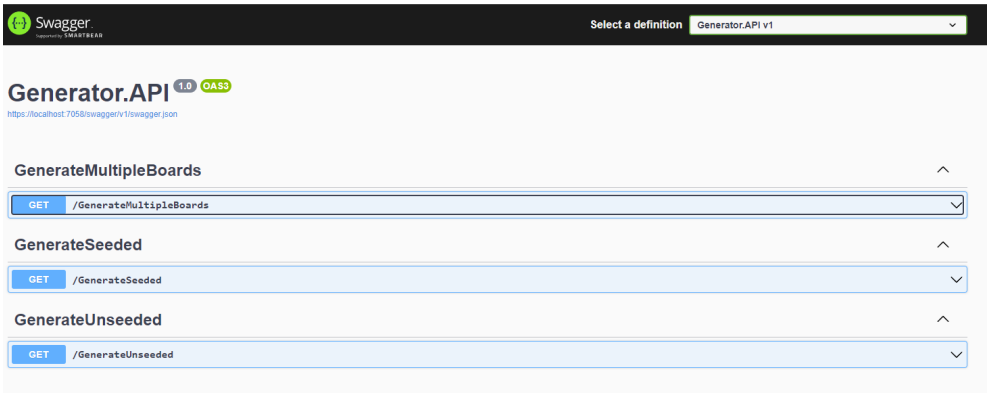
Having suggested these improvements, it should however be restated that, though they might be very successful in decreasing generation-times for relatively small boards, their effect on the overall scalability of the generator will likely be marginal. This is, as explained above, due to the NP-hardness of solving boards, something that could not easily be circumvented.

4.2 The generator web API

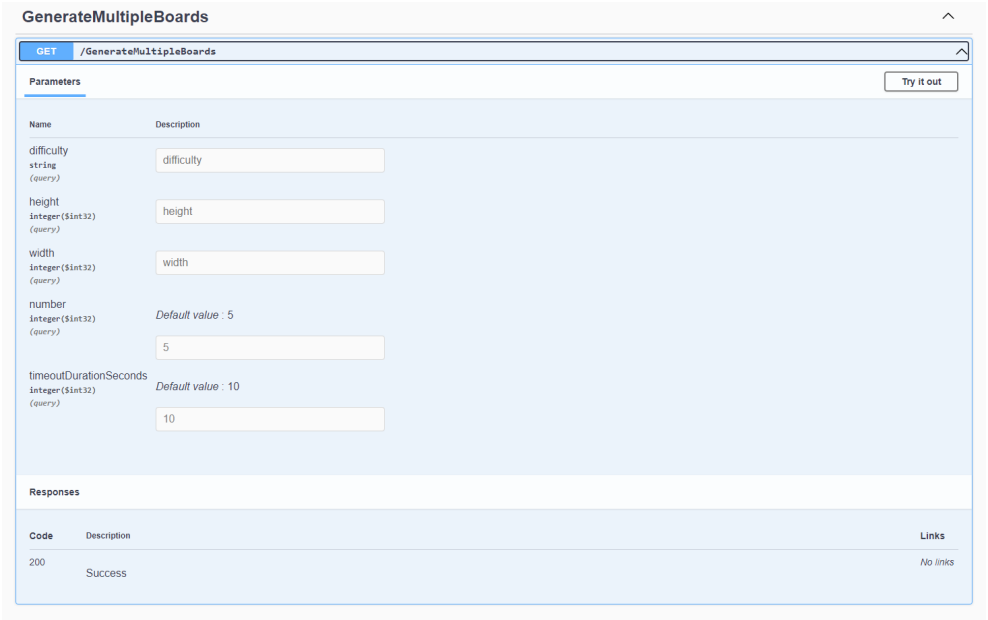
As explained in section 3.4, the finished generator was integrated as part of a web API. This enabled the generator to be used in a simple, intuitive way; espe-

cially when paired with a simple, auto-generated User Interface (UI), made using *Swagger* (See figures 4.3 and 4.4).

When called with correctly formatted parameters, for instance as shown in figure 4.4a, the web API returned a human-solvable board, compliant with the requested board size and difficulty. The returned board was furthermore formatted as a JSON string, meaning it could relatively easily be utilized from various different programs, such as the developed app. Part of such a response can be seen below in figure 4.4b.

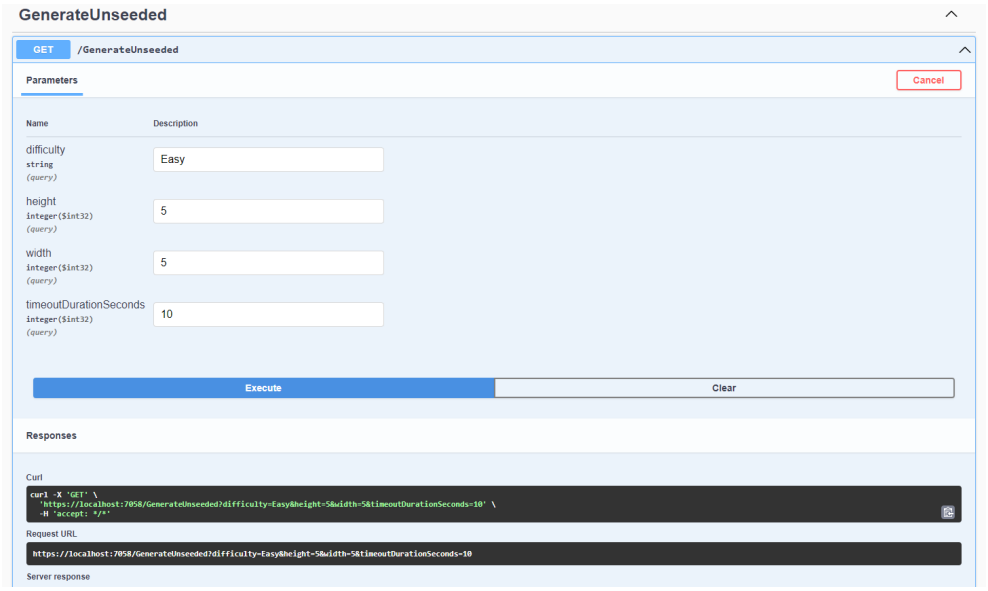


(a) Here, all the available endpoints are listed.

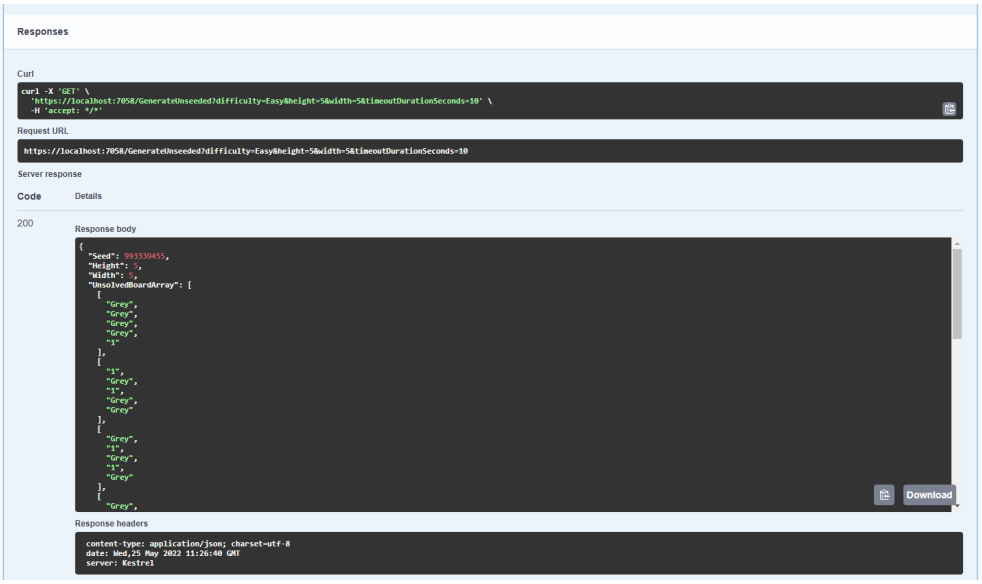


(b) Here, the */GenerateMultipleBoards* endpoint has been expanded, displaying the request parameters. Similar request interfaces existed for both the *GenerateSeeded* and *GenerateUnseeded* endpoints.

Figure 4.3: The auto-generated user interface of the web API.



(a) The request sent to the API. Note the request URL at the bottom, where the parameters have been added.



(b) The response from the API; a human-solvable Nurikabe board, generated based on the parameters sent in the request. Note that the entire response is not shown here; the "UnsolvedBoardArray" continues past the edge of the screen, and is followed by a "SolvedBoardArray", containing the solution to the generated board.

Figure 4.4: An example of a request to the API and the returned response.

4.2.1 Hosting

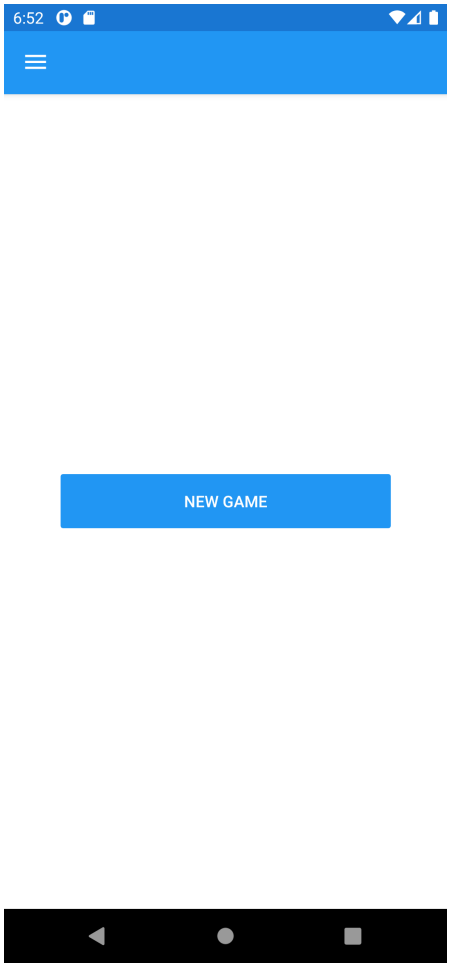
As it stands, the web app is not hosted online in any way. Attempts were made to achieve this, but as explained in section 3.4.2, these did not bear fruit, mostly due to the fact that the web app needed to integrate the Haskell program that constituted the actual generator. Therefore, the only way to use the generator currently, is to run the .NET program locally. This program, in the form of a Visual Studio solution, can be found in the attached code. See appendix B.1 for the structure of the attached code.

4.3 The "MasterGame" app

The developed app, eventually simply called "MasterGame", enabled a user to solve generated Nurikabe puzzles, and so fulfilled the objectives of the project. The app consisted of an intuitive, if minimalistic, user interface, and all the functionality necessary to solve Nurikabe puzzles. Unfortunately, to build the app for iOS would require access to a mac, something that was not available to the candidate during this project. The finished app is therefore only available on Android devices. For instructions on how to install the finished app from the attached .zip file, see appendix B.2.

4.3.1 User interface

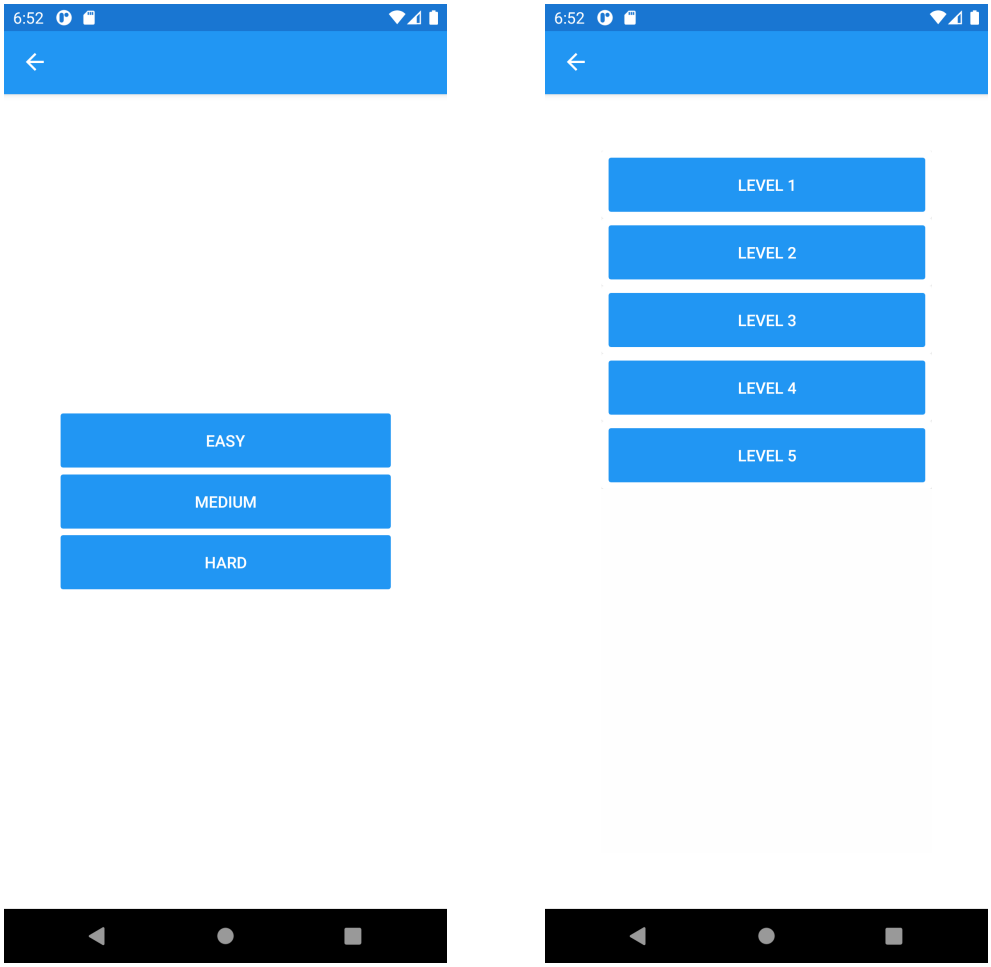
Upon first opening the app, the user will be greeted by a main menu, as shown in figure 4.5a. From here, pressing the "New Game" button will lead to the difficulty selection screen, as can be seen in figure 4.6a. Choosing a difficulty, by pressing one of the buttons, will then lead to a level selection screen, as seen in figure 4.6b. Note that, as previously explained, the "hard" difficulty is not fully implemented, and so attempting to press this button will not have any effect.



(a) The app's main menu if no puzzle is currently in progress.

(b) The app's main menu if a puzzle is currently in progress; in this case, Level 3 of the "medium" puzzles.

Figure 4.5: The app's main menu.

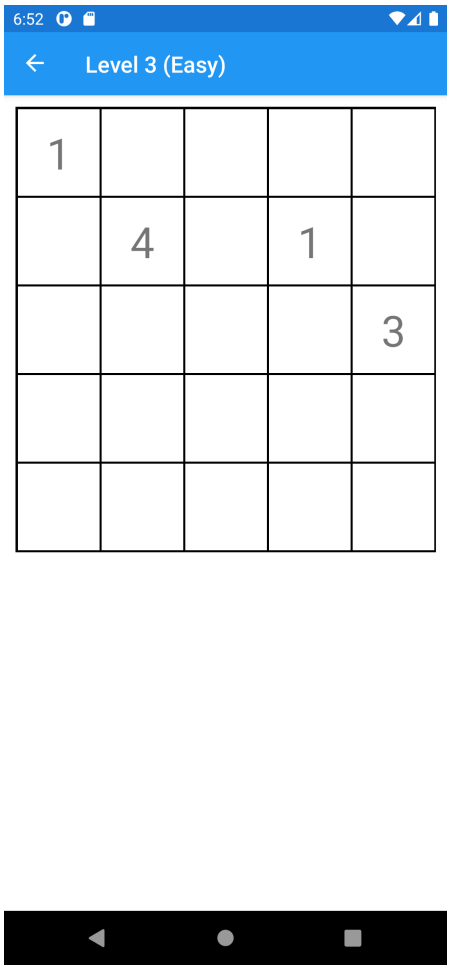


(a) The app's difficulty selection screen. **(b)** The app's level selection screen.

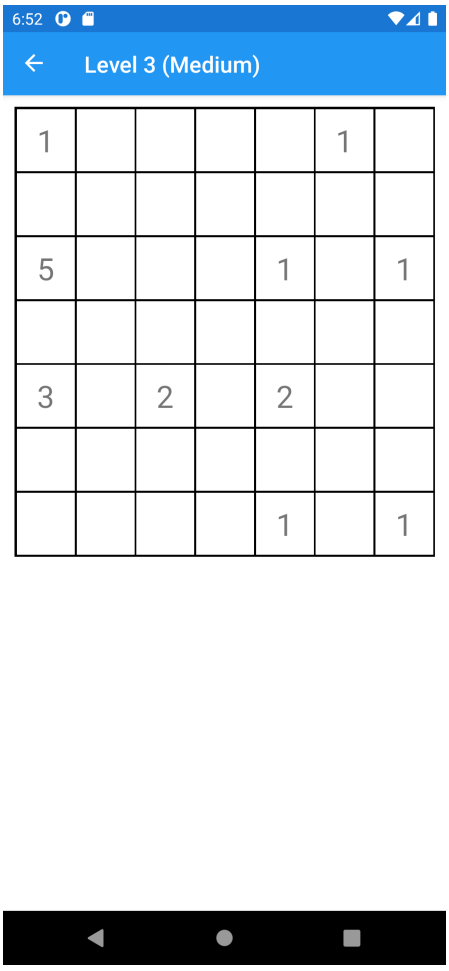
Figure 4.6: The app's puzzle selection screens.

Having chosen a level from either the "easy" or "medium" selections, The game screen is then displayed, as can be seen in figure 4.7 below, and the game can be played. To interact with the game screen, i.e. to play the game, both long and short presses of the screen can be used; short presses will toggle the pressed cell's type assignment between being unknown and water, whereas long presses will toggle it between being unknown and island. How these different assignments look in the app, can be seen in figure 4.8a below.

Pressing the back button, i.e. the small, left-facing arrow in the top left corner of the screen, will cause the app to return to the main menu; only now a "Continue" button will also be shown as can be seen in figure 4.5b. Pressing this button allows the user to return to an unfinished game. Finishing a puzzle will display a pop-up, as seen in figure 4.8b below.

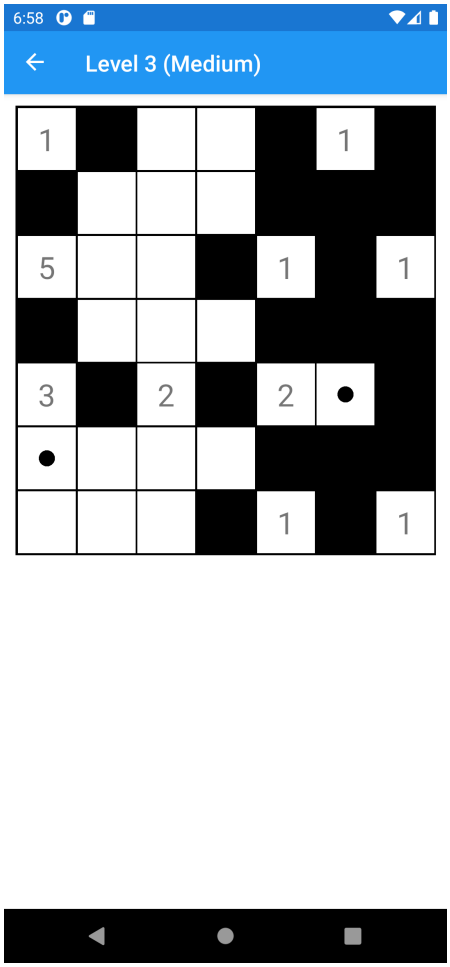


(a) The app's game screen, showing a fresh, "Easy" difficulty board.

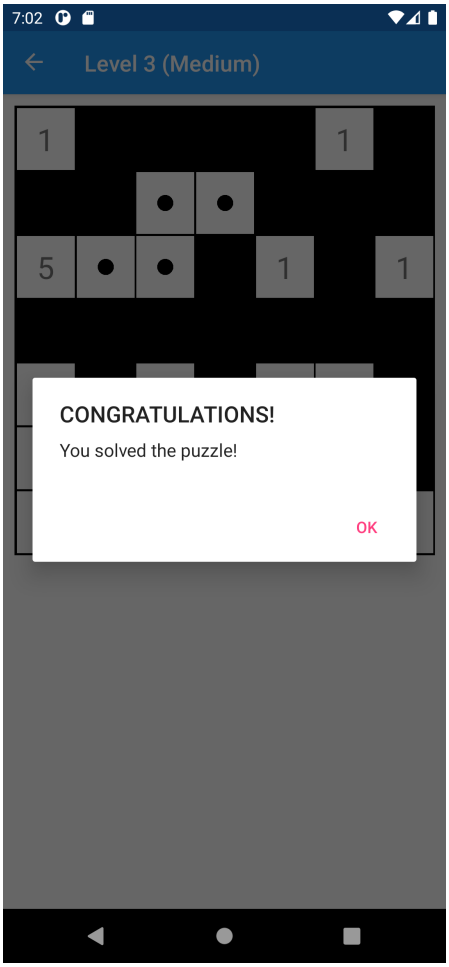


(b) The app's game screen, showing a fresh, "Medium" difficulty board.

Figure 4.7: The app's game screen.



(a) The app's game screen, showing a "Medium" difficulty puzzle in progress. Note how the different cell types are displayed: unknown cells as white boxes, water cells as black boxes, and island cells as white boxes with a central dot.

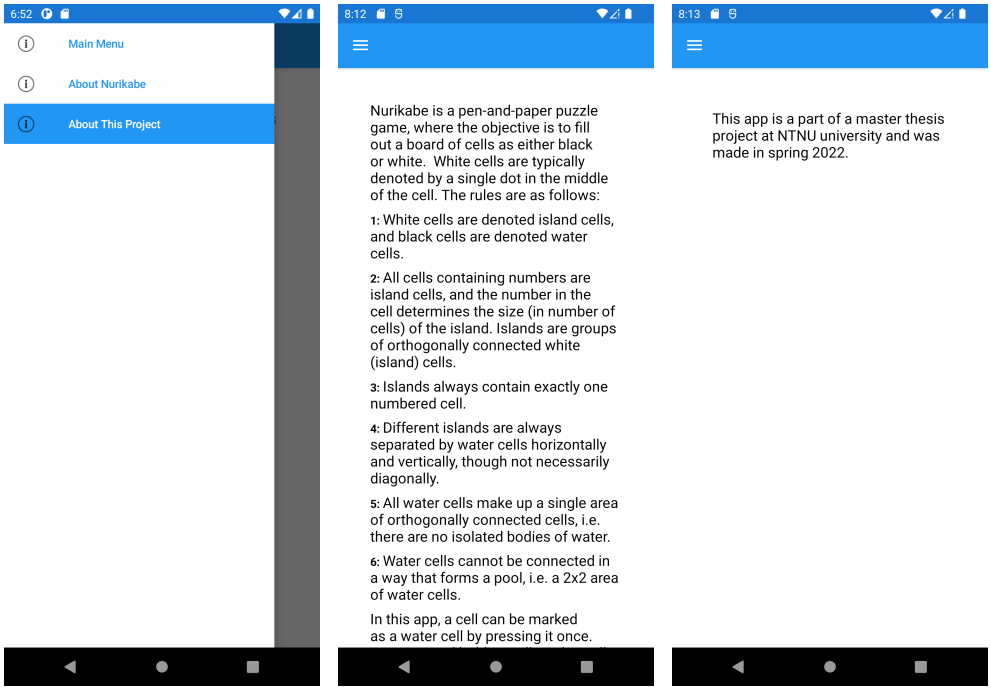


(b) The app's game screen, showing the pop-up alert that appears when a puzzle is solved.

Figure 4.8: The app's game screen.

From the main menu, pressing the top-left button consisting of three vertical lines, often called a *hamburger-menu* due to its shape, will open the app's *flyout* menu, as can be seen in figure 4.9a below. From this menu, it is possible to navigate between the main menu, and the app's two info pages: the "About Nurikabe" page, and the "About This Project" page.

The "About Nurikabe" page, that can be seen in figure 4.9b below, displays the rules of Nurikabe puzzles, as well as the app's interface for solving them. The "About This Project" page displays a short statement about the context of the app, and can be seen in figure 4.9c below.



(a) The app's flyout menu, allowing a user to navigate between the main menu and the info screens.

(b) The "About Nurikabe" info screen, displaying useful information about Nurikabe, and how to use the app.

(c) The app's "About This Project" info screen, displaying a brief statement about the context of the app and this project.

Figure 4.9: The app's flyout menu and info screens.

4.3.2 Board generation

As mentioned, the app was originally intended to consume a web API exposing endpoints that allowed for generation of puzzle boards. Doing so would serve to nicely decouple the app from the generator, something that was desirable so that the generator's at times poor run-time wouldn't affect the performance of the app. Consuming a web API in an asynchronous manner would, after all, be an easily cancellable action if it proved to take too long. Due to the difficulties explained earlier in this chapter however, the web API could not be hosted, and so consuming it from the app was not possible.

Separating the app from the generator would also serve the purpose of simplifying the app's design and development, something that was ideal due to the candidate's lack of experience developing mobile apps. Because of this, and due to a suspicion that the difficulties with hosting the web app would also be present in the app, it was decided not to attempt integrating the generator directly into the app. Once the web API fell short, a few boards were therefore manually imported into the app in the form of *.json* files instead. The "Level selection screen", called "IndexSelectionPage" in the code, was added at this point to enable selecting a puzzle board from a list.

In spite of the finished app having no automated way to generate new boards, the objectives of the project were still considered fulfilled, as an app enabling a user solve Nurikabe puzzles had been developed. Additionally, the root cause of the problems was mainly the incompatibility of two different technologies, resolving which was considered outside the scope of this project.

Chapter 5

Discussion

The goals of this project were to create a generator program capable of generating human-solvable instances of the nurikabe logic puzzle, as well as to create an app that allows solving such puzzles. Both of these goals were in turn intended to serve the overarching purpose of improving the candidate's skills as a software developer. In this chapter, the degree to which these goals have been achieved will be discussed.

5.1 The puzzle generator

As far as the generator is concerned, it can be said to function at an adequate level, as relatively simple boards can be generated at appropriate sizes in decent time. There is however much room for improvement. First and foremost, the implementation of more advanced solution techniques would enable the generator to create more difficult (and, as explained in section 4.1, larger) boards. Additionally, the generator algorithm itself could likely be improved, increasing its reliability and efficiency. Some means that might achieve this were introduced in section 4.1, but as efficiency was not a focus of this project, it was not prioritized heavily in the development.

As mentioned, the final generator lacked the ability to generate boards harder than what was defined as "medium" difficulty. In spite of this, the generator can still be said to have achieved it's purpose. For one, the objective of the project was never to generate the hardest possible Nurikabe puzzles, but rather to investigate if, and how, an program procedurally generating puzzles could be developed. The "easy" and "medium" difficulty boards that have been generated serve as proof of concept of this. Admittedly, implementing a way to generate harder puzzles might have made the resulting app more engaging, but this was again not the purpose of the project, and so would not affect the results or conclusion of it.

5.2 The generator web API

Integrating the generator in a web API, that would enable the solver app to generate boards on demand, proved difficult. Both the compilation of the Haskell code to a DLL usable by the web app, and the hosting of said web app proved troublesome, with the second problem remaining unsolved at the conclusion of this project. These problems both arose from the fact that the generator itself was written in Haskell, a language that is, at its base, incompatible with the C family of programming languages, and thereunder C#. It could therefore be questioned whether choosing to use this language for the generator was an optimal decision. If the main objective of this project was to develop an app to be published, the answer to this question would likely by a definitive no; opting to implement the generator, in its entirety, in C# would likely be more prudent. As the overarching goal was not the publication of an app however, but rather for the candidate to hone their skills as a developer, the benefits of choosing Haskell likely outweighed the costs.

Importantly, the use of the Haskell language served to introduce the candidate to purely functional programming, and with it a skill-set that will undoubtedly be beneficial to a developer. Being aware of the patterns and techniques of functional programming will grant insight into alternative ways of solving problems, something that is always an advantage, even when using imperative languages.

Additionally, by introducing the challenges mentioned above, the use of Haskell allowed the candidate to try their hand at solving development problems that lie outside the realm of programming; namely making different, inherently incompatible, technologies work with each other. Facing, and attempting to solve, such a challenge will surely benefit the candidate in the future. Even if the solution in this case was only a partial success, the experience will serve to improve the candidate's ability both to compare technologies and choose between them. The experience might also help the candidate combine technologies, that may not be designed to work together, if the need should arise.

All told, the created web app can, in spite of its lacking integration, be concluded to have aided in achieving the goals of the project.

5.3 The "MasterGame" app

The developed app is likewise acceptable in its function. It's main purpose has been achieved in a fully satisfactory way, as it, in general, allows users to solve Nurikabe puzzles of any size and difficulty. However, due to challenges discussed above, and in chapter 4, the app was unable to generate puzzles on demand, instead including some pre-generated boards in local files. This meant that the app did not, in practice, enable users to solve puzzles of any size, but only of fixed, predetermined sizes, despite the functionality being fully implemented and functional in the app. Likewise, the app contained code that would allow the consumption of a web API that generated boards of any size, but this remains unfinished

and untested at the conclusion this project, again due to the challenges discussed above.

As far as the user interface of the app is concerned, it is fully functional, though it lacks polish. All buttons work as intended, and allow navigation around the app, but the appearance of these buttons, as well as the rest of the app, is, for lack of a better word, boring. At the conclusion of the project, the app only uses the default colors, designs, and icons of Xamarin components. That said, the focus of the project was not, as previously stated, to develop an app to be published, but rather to allow users to solve Nurikabe puzzles. The outwards appearance of the app was therefore not prioritized in the development.

Likewise, the development of the app served to give the candidate experience with app development, and to familiarize them with new technologies, such as the Xamarin framework. It can therefore be said to have contributed to the overarching goal of developing the candidate's skills as a developer.

Chapter 6

Conclusion

Through the work on this project, the pen-and-paper logic puzzle known as Nurikabe has been investigated. It's computational complexity has been thoroughly examined, and techniques for its solution have been developed. A generator program, capable of generating Nurikabe puzzles solvable to a human, has also been developed, along with an app enabling a user to solve these generated puzzles.

The developed generator serves to prove that it is possible to procedurally generate Nurikabe puzzles that can be solved by humans, even if this generation, like the Nurikabe problem itself, is NP-hard. It also stands to reason that this holds true for the many other logic puzzles that are in their general form NP-hard or NP-complete. Furthermore, the puzzles generated by the generator are non-trivial in their solutions, seemingly indicating that there is potential for a computer program to generate puzzles at a similar level as humans are capable of, even if they might lack the artistic sense of a human generator.

Even though the generator does fulfill its main objective of generating human-solvable puzzles, it is not without flaws, by any means. For the generator to be of any practical use in, for instance in a published app, it would need some significant improvements. For one, it would need to have its efficiency improved, as the current version takes unacceptably long to generate anything larger than a medium sized board. Another area with room for improvement is where the difficulty of the produced puzzles is concerned. The current generator lacks implementation for the harder Nurikabe techniques, and is therefore unable to generate puzzles requiring these techniques to solve. Additionally, the current version sacrifices difficulty for reliability, by simplifying boards that are unsolvable in order to make them solvable. This means there is no way to guarantee that a generated board is as difficult as the generator's settings would imply, with the exception of board size, which only has a small effect on overall difficulty.

The developed app likewise succeeded in fulfilling the objectives of the project. This app is in some ways more completely implemented than the generator in that it enables the solution of Nurikabe boards of any size and difficulty. It is however not perfect, and improvements could be made also here. For instance, the visual elements of the app could use some more work to be more appealing, and easy

on the eyes. Likewise, the interface between app and generator would also need more work, on both ends, to be fully functional.

Just as the concrete, technical objectives of the project were achieved, so too was the overarching goal of improving the candidate's skills as a software developer. Through the work on the project, the candidate was exposed to several new technologies, frameworks and languages, expanding their proverbial toolbox of development skills. They were also exposed to new ways of thinking about, and solving, problems, something that is an advantage in all aspects of engineering, and not just software development. The project can therefore be concluded to have been very successful in achieving the intended goals.

Bibliography

[1] G. Kendall, A. Parkes and K. Spoerer, 'A survey of np-complete puzzles,' 2008.

[2] E. Britannica, 'Computational complexity,' 2021. [Online]. Available: https://www.britannica.com/topic/computational-complexity.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to algorithms*, 3rd ed. MIT Press Ltd, 2009.

[4] M. Holzer, A. Klein and M. Kutrib, 'On the np-completeness of the nurikabe pencil puzzle and variants thereof,' 2008.

[5] B. P. McPhail, 'The complexity of puzzles: Np-completeness results for nurikabe and minesweeper,' The Division of Mathematics and Natural Sciences, Reed College, Dec. 2003.

[6] Microsoft, *Why choose .NET*, https://dotnet.microsoft.com/en-us/platform/why-choose-dotnet, Accessed: 10.05.2022.

[7] Microsoft, *What is ASP.NET*, https://dotnet.microsoft.com/en-us/learn/aspnet/what-is-aspnet, Accessed: 10.05.2022.

[8] Microsoft, *A tour of the c# language*, https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/, Accessed: 21.03.2022.

[9] Microsoft, *What is xamarin?* https://docs.microsoft.com/en-us/xamarin/get-started/what-is-xamarin, Accesssed: 22.03.2022.

[10] Microsoft, *Xamarin.forms XAML basics*, https://docs.microsoft.com/nb-no/xamarin/xamarin-forms/xaml/xaml-basics/, Accessed: 22.03.2022.

[11] Microsoft, *The model-view-viewmodel pattern*, https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm, Accessed: 22.03.2022.

[12] Microsoft, *Controls reference*, https://docs.microsoft.com/en-us/xamarin/xamarin-forms/user-interface/controls/, Accessed: 04.04.2022.

[13] Microsoft, *Xamarin.forms pages*, https://docs.microsoft.com/en-us/xamarin/xamarin-forms/user-interface/controls/pages, Accessed: 04.04.2022.

[14] Microsoft, *Xamarin.forms layouts*, https://docs.microsoft.com/en-us/xamarin/xamarin-forms/user-interface/controls/layouts, Accessed: 04.04.2022.

[15] Microsoft, *Xamarin.forms views*, https://docs.microsoft.com/en-us/xamarin/xamarin-forms/user-interface/controls/views, Accessed: 04.04.2022.

[16] Microsoft, *Xamarin.forms cells*, https://docs.microsoft.com/en-us/xamarin/xamarin-forms/user-interface/controls/cells, Accessed: 04.04.2022.

[17] H. community, *Introduction*, https://wiki.haskell.org/Introduction, Accessed: 12.05.2022.

[18] M. Lipovača, *Learn You a Haskell for Great Good!* Accessed: 12.05.2022.

[19] H. community, *Polymorphism*, https://wiki.haskell.org/Polymorphism, Accessed: 12.05.2022.

Appendix A

Summary of the proof of the Nurikabe problem being NP-complete

In this section, the proof of NP-completeness for Nurikabe will be briefly summarized. For the full proof, see Holzer et al[4] or McPhail[5].

First and foremost, a decision problem needs to be stated, of which to prove the NP-completeness. This is done in section 2.2.2 and so will not be restated here.

A.1 Proof of NP membership

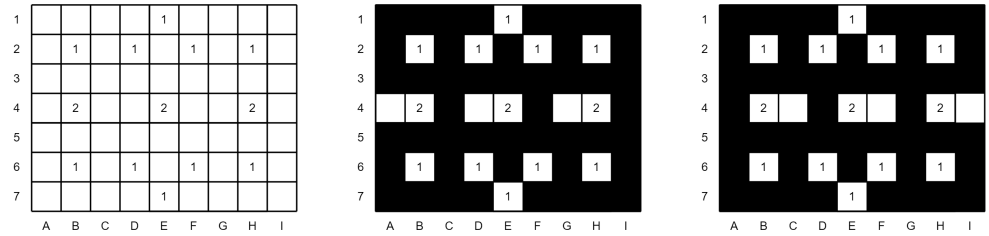
From here, the proof of the NP-completeness of Nurikabe consists of first proving that the problem is NP. McPhail proves this by describing an algorithm that can verify a solution to a Nurikabe board in time $O(N^3)$ where $N = n \times m$. As this proves a solution to the Nurikabe problem can be verified in polynomial time, it is concluded that the Nurikabe problem is in fact NP.

A.2 Proof of NP-hardness

Next, it must be proved that the nurikabe problem is NP-hard. This is done by proving that a known NP-complete problem can be reduced to the Nurikabe problem in polynomial time. If such a reduction is possible, the Nurikabe problem must be NP-C. To perform this proof, McPhail uses the *Circuit-SAT* problem, whereas Holzer et al uses the *planar 3SAT* problem. Both of these reductions involve showing that a Nurikabe board can be used to create a boolean circuit, consisting of signal-bearing wires, signal splitters and several different logic gates. To this end, it is proven that a Nurikabe board can be used to represent any logic gate. Holzer et al[4] proves this for wires, signal-splitters, NOT and OR-gates as follows:

A.2.1 Nurikabe wire design

A wire is constructed from a Nurikabe board, by letting hints with value 2 carry a signal horizontally as shown in figure A.1 below. Note that, as is clear from figure A.1, several of these wires could be chained together to form longer wires. Holzer et al also design further signal components, including input units, as well as a phase shifter, to ensure that wires can be positioned in a way that enables the components to follow. These components have however not been included here for berivty's sake, but can be found in the paper[4]. Included there are also designs of components solely used to construct a complete Nurikabe board from these components; i.e. easily, uniquely solvable rectangular board components that can be fit in between the logic components to create a complete, solvable Nurikabe board.



(a) An unsolved wire board with two possible solutions.

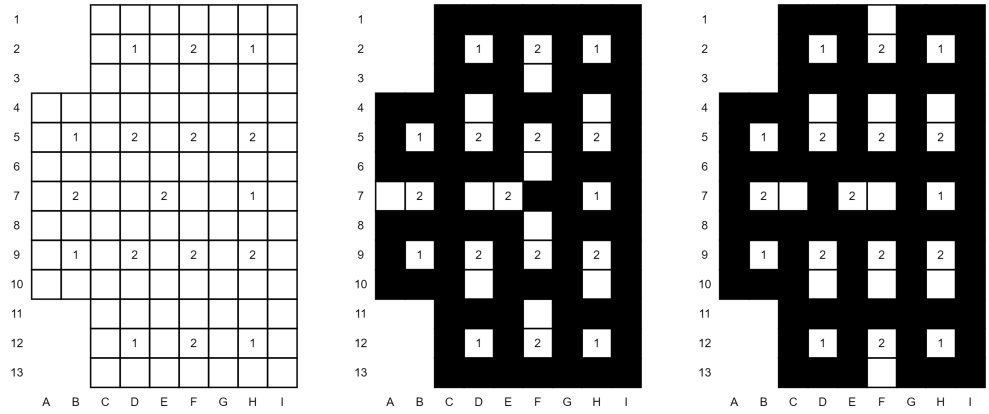
(b) Here, the cell in position **A4** has been marked as an island cell, and all other cells marked according to the only solution that is now possible. Note the leading island cells of both the 2-hinted cells in positions **B4** and **H4**; these indicate a *true* value.

(c) Here, the cell in position **A4** has been marked as a water cell, and all other cells marked according to the only solution that is now possible. Note the trailing island cells of both the 2-hinted cells in positions **B4** and **H4**; these indicate a *false* value.

Figure A.1: Nurikabe boards displaying how a signal-bearing logic wire could be built as a Nurikabe board. Note that the unsolved board in A.1a has two possible solutions, and exactly one is correct based what is input on the leftmost 2 hint, in cell **A4**. The two-cell island that cell **B4** is part of can therefore be considered the input of the wire, and the two-cell island that cell **H4** is part of, can be considered the output, as it's second cell will vary with the input.

A.2.2 Nurikabe signal-splitter design

Next, Holzer et al design a signal-splitter, to enable duplication of a signal. It is constructed as shown below, in figure A.2. Again, it is clear how this component can be made to interface with other components by the use of a two-cell island "interface".



(a) An unsolved signal-splitter board with two possible solutions.

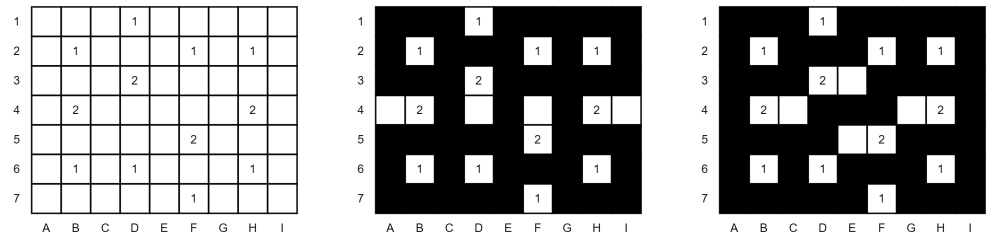
(b) Here, the cell in position **A7** has been marked as an island cell, and all other cells marked according to the only solution that is now possible. Again note the leading island cells of the three 2-hinted cells in positions **B7**, **F2**, and **F12**; these indicate a *true* value.

(c) Here, the cell in position **A7** has been marked as a water cell, and all other cells marked according to the only solution that is now possible. Again note the trailing island cells of the three 2-hinted cells in positions **B7**, **F2**, and **F12**; these indicate a *false* value.

Figure A.2: Nurikabe boards displaying how a logic signal-splitter could be built as a Nurikabe board. Note that the unsolved board in A.2a has two possible solutions, exactly one of which is correct depending what is input on the leftmost 2 hint, in cell **A7**. The island containing cell **B7** can therefore be considered the input, and the islands containing cells **F2** and **F12** can be considered the output.

A.2.3 Nurikabe NOT-gate design

Figure A.3 below shows how Holzer et al designed a Nurikabe board that could be used as a logic NOT-gate.



(a) An unsolved NOT-gate board with two possible solutions.

(b) Here, the cell in position **A4** has been marked as an island cell, and all other cells marked according to the only solution that is now possible. Note that the leading island cell of the 2-hinted island containing cell **B4** leads to a trailing island cell in the island containing cell **H4**; This indicates that a *true* input causes a *false* output.

(c) Here, the cell in position **A4** has been marked as a water cell, and all other cells marked according to the only solution that is now possible. Note that the trailing island cell of the 2-hinted island containing cell **B4** leads to a leading island cell in the island containing cell **H4**; This indicates that a *false* input causes a *true* output.

Figure A.3: Nurikabe boards displaying how a signal-inverting logic gate, or NOT gate, could be built as a Nurikabe board. Note that the unsolved board in A.3a has two possible solutions, exactly one of which is correct based what is input on the leftmost 2 hint, in cell **A4**. The two-cell island that cell **B4** is part of can therefore be considered the input of the wire, and the two-cell island that cell **H4** is part of, can be considered the output, as this island's second cell will vary with the input.

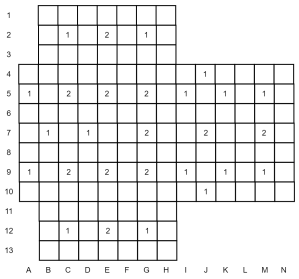
A.2.4 Nurikabe OR-gate design

Figure A.4 below shows how Holzer et al designed a logic OR-gate in the form of a Nurikabe board.

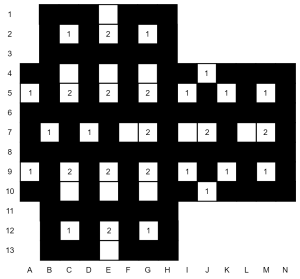
A.2.5 Composite logic gates

Using the logic gates described above, it is now possible to create any desired logic gate. For instance, a NAND gate can be constructed simply by applying a NOT-gate to the inputs of an OR-gate, as evidenced by table A.1 below:

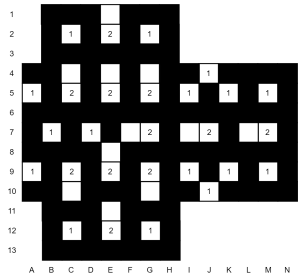
From here, an AND-gate can be constructed similarly easily by negating the output of a NAND-gate, as evidenced by table A.2 below:



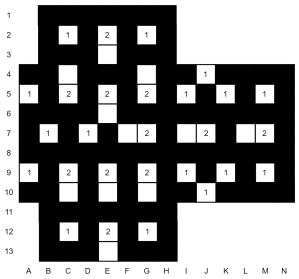
(a) An unsolved OR-gate board, with four total possible solutions.



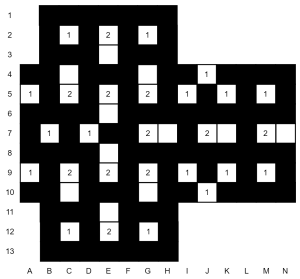
(b) ***true*** **OR** ***true*****:** Here, the cells in positions **E1** and **E13** have both been marked as island cells, making both inputs *true*. All other cells have then been marked according to the only solution that is now possible. The island cell in position **L7** now indicates a *true* output.



(c) ***true*** **OR** ***false*****:** Here, the cell in position **E1** has been marked as an island cell, while the one in position **E13** has been marked as a water cell, making the inputs *true* and *false* respectively. All other cells have then been marked according to the only solution that is now possible. The island cell in position **L7** now indicates a *true* output.



(d) ***false*** **OR** ***true*****:** Here, the cell in position **E1** has been marked as a water cell, while the one in position **E13** has been marked as an island cell, making the inputs *false* and *true* respectively. All other cells have then been marked according to the only solution that is now possible. The island cell position **L7** now indicates a *true* output.



(e) ***false*** **OR** ***false*****:** Here the cells in positions **E1** and **E13** have both been marked as water cells, making both inputs *false*.All other cells have then been marked according to the only solution that is now possible. The island cell in position **N7** now indicates a *false* output.

Figure A.4: Nurikabe boards displaying how a logic OR-gate could be built as a Nurikabe board. Note that the unsolved board in A.4a has four possible solutions, exactly one of which is correct for each combination of inputs on the top and bottom 2 hinted islands, in cell **E1** and **E13**. The two-cell islands that cells **E2** and **E12** are part of can therefore be considered the inputs of the wire, and the two-cell island that cell **M7** is part of, can be considered the output, as it's second cell will vary with the input.

X	Y	$\neg X \vee \neg Y)$	$X \bar{\wedge} Y$
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	0

Table A.1: Truth table for an input-negated OR-gate, as well as a NAND-gate. Note how the negated OR-gate produces the same output as the NAND-gate.

X	Y	$\neg X \bar{\wedge} \neg Y$	$X \wedge Y$
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

Table A.2: Truth table for a negated NAND-gate as well as an AND-gate. Note how both gates produce the same output.

Similar proofs can be made for any desired logic gate. See for instance the appendic of Mcphail[5].

A.2.6 Polynomial time reducibility

From here, both Holzer et al[4] and McPhial[5] proceed to prove how the above transformations, from logic gate to Nurikabe board, can be performed in polynomial time. Likewise it is proved that combining these components into any given boolean circuit can be done in polynomial time, thereby proving that the Nurikabe problem is NP-complete.

Appendix B

Attachments

B.1 Structure of the attached source code

Submitted along with this report, is a .zip file, called “code.zip”, containing the source code for the programs developed in the project: the MasterGame app, the generator and the web app. At the top level, this file contains, within the *code* folder, three items: the *Generator* folder, containing the source code for the generator program and web app, the *MasterGame* folder, containing the source code for the “MasterGame” app, and the “mastergame.MasterGame.apk” file, being the actual developed app. In the following sections, the structure of the source code folders will be explained. Note that not all included files and folders will covered, as some are merely included as part of the Visual Studio solution, and have therefore not been directly used in the project. These files are represented by “...” in the below directory trees.

B.1.1 Generator

The structure of the *Generator* folder is as follows:

- Generator/
  - Controllers/
    - GenerateController.cs
  - Program.cs
  - ...
- Generator.Application/
  - Board/
    - Board.cs
    - HaskellRuntimeSingelton.cs
  - HaskellGenerator/
  - ...
- Generator.Common/
  - Exceptions/
    - BoardTimeoutException.cs
    - UnsovableBoardException.cs
  - ...
- Generator.Dto/
  - BoardDto.cs
  - ...
- Generator.sln

Here, the *Generator* folder acts as the entrypoint of the web-app, specifically through the "Program.cs" file. The "GenerateController.cs" file is the code file containing the actual endpoints of the web app. The *Generator.Application* folder contains the actual generator functionality, including both the .NET components found in the sub-folder *Board*, as well as the generator written in Haskell, which can be found in the *HaskellGenerator* sub-folder. This sub-folder will be explored more in-depth below.

The *Generator.Common* folder contains various functionality used throughout the web-app, main in the form of the exceptions found in the *Exceptions* sub-folder. The *Generator.Dto* folder contains the DTO used to construct the .json string representing a board. Lastly, the "Generator.sln" file serves as the solution-file for the Visual Studio project these files constitute.

HaskellGenerator

The structure of the *HaskellGenerator* is as follows:

- Board/
  - Board.hs
  - Cell.hs
  - Generator.hs
  - Modifier.hs
  - Monitor.hs
- Nurikabe/
  - Easy.hs
  - Generator.hs
  - Hard.hs
  - Medium.hs
  - Nurikabe.hs
- Region/
  - Incomplete.hs
  - Island.hs
  - Region.hs
  - Water.hs
- Techniques/
  - AvertPools.hs
  - Complete.hs
  - Expand.hs
  - Separate.hs
  - Surround.hs
  - Unreachable.hs
- Test/
  - AvertPools.hs
  - Complete.hs
  - Expand.hs
  - Generator.hs
  - Nurikabe.hs
  - Separate.hs
  - Surround.hs
  - Unreachable.hs
- HaskellInterface.hs
- ...

Here, the *Board* folder holds files containing the lowest-level functionality of the generator, such as the definitions of the *Board* and *Cell* data types, as well as

the *Modifiable* type class. These files also handle basic operations on boards, such as generating complete, unnumbered boards (in "Generator.hs") and reassigning cells (in "Modifier.hs").

The *Nurikabe* folder contains the solvers of different difficulty, as well as the generator functionality utilizing these solvers to generate puzzle boards.

The *Region* folder contains code that operates on boards on the region level; i.e. operations that operate on entire islands, water-regions, or unknown regions.

The *Techniques* folder unsurprisingly contains the implementations of the different solution techniques that were implemented.

The *Test* folder contains some of the tests that were written to test the different solution techniques, as well as the more composite solution- and generation functions.

Lastly, the "HaskellInterface.hs" file serves as the entrypoint into the Haskell code for the web app, and exposes the functions and types necessary to use it. also included in *HaskellGenerator* folder, though not included in the above directory tree, are several files that served to aid in the creation of the DLL used in the web app, such as the "ExportTypes" files detailed in section 3.4.1.

B.1.2 MasterGame

The structure of the *MasterGame* folder is as follows:

- MasterGame/
  - MasterGame/
    - Assets/
      - Boards/
      - ...
    - Common/
      - Dtos/NurikabeBoardDto.cs
    - Models/
      - Boards/
      - → Boards.cs
      - → NurikabeBoard.cs
      - Cells/
      - → BoxCell.cs
      - → ImageCell.cs
      - → LabelCell.cs
      - GeneratorClient.hs
    - ViewModels/
      - IndexSelectionViewModel.cs
      - NurikabeDifficultySelectionViewModel.cs
      - NurikabeGameViewModel.cs
      - NurikabeMenuViewModel.cs
    - Views/
      - AboutNurikabePage.xaml(.cs)
      - AboutThisProjectPage.xaml(.cs)
      - IndexSelectionPage.xaml(.cs)
      - NurikabeDifficultySelectionPage.xaml(.cs)
      - NurikabeGamePage.xaml(.cs)
      - NurikabeMenuPage.xaml(.cs)
    - App.xaml(.cs)
    - AppShell.xaml(.cs)
    - ...
  - MasterGame.Android/
    - ...
  - MasterGame.iOS/
    - ...
- MasterGame.sln

As can be seen above, the *MasterGame* sub-folder contains three folders: *MasterGame*, *MasterGame.Android* and *MasterGame.iOS*. Of these, only the *MasterGame* sub-folder is of much interest, as the *MasterGame.Android* and *MasterGame.iOS* sub-folders only contains files needed to build the app for Android and iOS devices, and were therefore rarely used directly in the project. The *MasterGame* sub-folder on the other hand contains the code base for the developed app.

The *Assets* sub-folder contains the garphical assets used in creating the app, i.e. pictures, as well as the imported boards, in the *Boards* sub-folder.

The *Common* sub-folder contains the DTO used to import and deserialize the boards from the .json files. This DTO mirrors the one found in the generator web app.

The *Models* sub-folder contains the so called *business logic* of the app; that is, it contains the core functionality that isn't necessarily directly concerned with visual appearance. Here, the classes containing the functionality for the boards and cells can be found, as well as the classes used to wrap these objects in displayable base classes. An example of this is the *Boards* class (in the "Boards.cs" file) containing the functionality for reading, maintaining and modifying boards, being wrapped in the *NurikabeBoard* class (in "NurikabeBoard.cs") that inherits the Xamarin.forms *Grid* class, thereby making the boards displayable.

The *ViewModels* sub-folder contains the view models of the app, tying the displayed views to the model backend.

Lastly, the *Views* sub-folder contains the views of the app. This includes a .xaml file declaring the visuals of each page, and a .xaml.cs *code-behind* file.

The *MasterGame* folder also contains the "App.xaml", "App.xaml.cs", "AppShell.xaml" and "AppShell".xaml.cs files, serving as the entrypoint of the app, and containing some default Xamarin logic, associated with navigation, styling and so on.

Finally, the top-level *MasterGame* folder also contains the "MasterGame.sln" solution file, used by Visual Studio as the entrypoint for the solution.

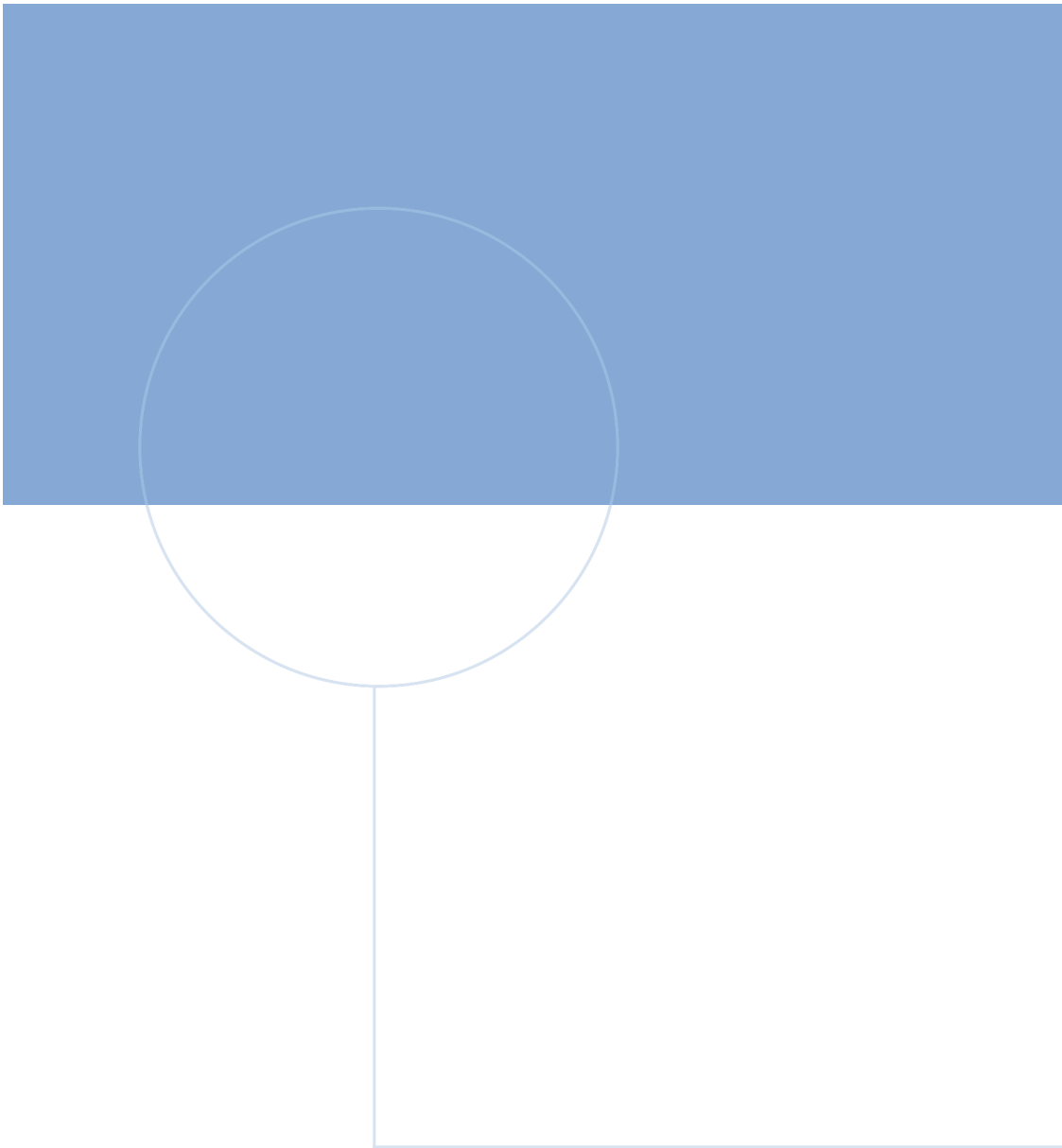
B.2 App and installation

The developed app can be found as the "mastergame.MasterGame.apk" file in the attached "code.zip" file. This is a *.apk* file, meaning it can only be installed on phones, and other devices, using the Android OS. Also note that the app has only been tested on emulators for Android versions 11 and 12, and on a physical device running Android 12. The app might work for other Android versions, but this cannot be guaranteed.

To install the app, simply follow the instructions below:

1. Connect the phone (or other device) the app is to be installed on, to a computer, with the .apk file downloaded and extracted from the .zip file. Ensure that exchanging files with the device is enabled.
2. Copy the *.apk* file to a known location on the device.

3. Locate the .apk file on the device, and tap it once.
4. A prompt will appear, asking if you want to install the app; press “install”.
5. The app will now begin installing. After a moment, a new prompt may appear, stating that *Play Protect* does not recognize the developer of the app. This can safely be ignored. Press “install anyways”.
6. The app will now finish, and can be open by pressing “open” on the prompt that appears, or by locating the app on the device.
7. After the app is installed, a last prompt may appear, asking if the app should be sent to *Play Protect* to be scanned. This is not necessary, and choosing “send” or “don’t send” will have no effect on the app.
8. The app can now be used. Enjoy!



NTNU

Norwegian University of
Science and Technology