BACHELOR'S THESIS COMPUTING SCIENCE

"Bridges" as an SMT problem

Solving and generating puzzles using different boolean encodings

RICO TE WECHEL
S4773039

August 22, 2023

First supervisor/assessor:
dr. C.L.M. Kop (Cynthia)

Second assessor:
dr. J.S.L. Junges (Sebastian)

Radboud University

Abstract

A property of NP-complete problems is that we can reduce them to each other. ”Bridges” is an NP-complete logic puzzle. An SMT solver is a tool that can determine the satisfiability of a mathematical formula. Since the Boolean satisfiability problem is a form of a mathematical formula and is proven to be NP-complete, we can reduce the problem of a game of Bridges to an SMT problem, which we can in turn use to efficiently find solutions to puzzles, and even to generate them. In this thesis we will compare two different SMT encodings for the puzzle, of which we find that one is majorly better in terms of efficiency over the other, also when it comes to generating. We will also find that human solvability is a factor to take into account when generating puzzles.

Contents

Chapter 1

Introduction

Throughout the centuries, more and more logical puzzles have been created for people to solve for fun. The best classic example is the game of Sudoku. Sudoku is a game with a small set of easy to understand rules, and a solution to the puzzle is found if every field in the game stays true to the set of rules. It can, however, be very difficult to get to such a point, which it what makes Sudoku a fun puzzle to play.

Sudoku is among one of the puzzles that have been proven to be NP-complete. [9] NP-complete problems have two properties. Property 1: The problem P is in the set of NP problems. This means that the solution to problem P can be *verified* to be an actual solution in polynomial time by a non-deterministic Turing machine, though *finding* an actual solution in polynomial time may not be feasible. Property 2: The problem P is in the set of NP-Hard problems. This means that every problem in NP can be reduced to problem P in polynomial time. These two properties combined we call NP-completeness. [7] If a problem is NP-complete, we get a very interesting property: Every problem that is NP-complete can be reduced to any other problem in the NP-complete class. The first problem that was proven to be NP-complete is the SAT problem. It turns out that SAT came to play a crucial role in solving mathematical problems. For instance, SAT solvers were built that can solve mathematical problems given that the problem has been reduced to SAT. By now, a lot of research has been done on solving mathematical problems using solvers in general. [2]

Nowadays, there are several kinds of mathematical solvers. Amongst those is something called an SMT solver. Throughout this study we will dive in on the background of SMT solvers in general and explore different possibilities and properties of SMT solvers. After that we will look into how we can reduce Bridges to SMT by encoding the game in terms of SMT methods in order to find solutions to the puzzles. This should be possible, as Bridges has also been proven to be NP-complete [1] (though it is sufficient for Bridges to be in NP, because SAT is

NP-complete, and SAT $\subseteq$ SMT, as we will see in Chapter 2 Preliminaries). After that we try to generate puzzles using the same SMT methods and evaluate the human solvability of the generated puzzles. Nearing the end of the paper, we will describe our findings when it comes to efficiency between several ways of solving and generating puzzles using SMT.

Chapter 2

Preliminaries

2.1 Bridges

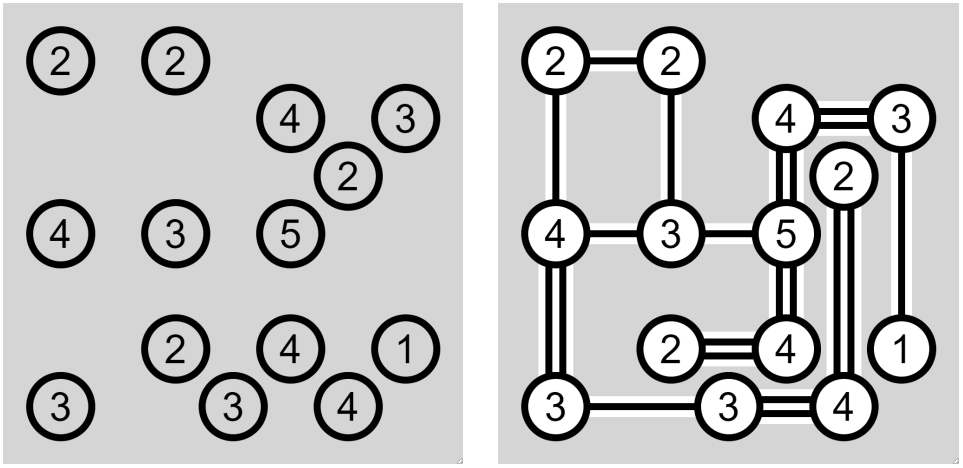
Bridges (Hashiwokakero) is a logic puzzle game invented by a Japanese publisher called Nikoli. [1] In the game, the player has to connect several cells in a grid using bridges such that several conditions are met. The player starts off with a (usually) square grid partially filled with cells. These cells contain numbers that represent the total number of bridges that should be connected to this cell. The goal is to connect every cell in the grid using bridges in such a way that all the numbers in the cells are equal to the number of bridges connected to those cells. In general, the game rules can be summed up as follows:

1. Bridges can only be built between two cells
2. Bridges can be built either horizontally or vertically
3. Bridges can be single (counting as one) or double (counting as two)
4. Bridges may not be obstructed by other bridges or cells
5. For each cell, the number of connected bridges to the cell should match the number inside the cell
6. The network should be connected to be one whole. In other words: Every cell should be reachable from every other cell

Take as example figure 2.1. As you can see, all of the conditions are met in the solution.

We will also make two assumptions about the puzzles in this study. One being that loops *are* allowed in solved configurations. There are also variants of Bridges where loops are not allowed in final solutions, but this will be kept outside the scope of this study. Furthermore, a very important property that we assume is that every puzzle should be uniquely solvable. In other words, for each puzzle there is

exactly one correct solution. This is important when it comes to logical puzzles; if there would be more than one correct solution, there would come a moment in time while solving the puzzle where there is a form of non-determinism; multiple moves can be made that could be considered correct, which defies the point of the game. In general it should hold that it's always logically reasonable that a considered move should bring you a step closer to the solution or not. This may be a very hard logical reasoning, but in any case it should ensure that you're on the right track.



(a) (b)

Figure 2.1: Two configurations of a Bridges puzzle. Figure 2.1a represents an unsolved puzzle, while figure 2.1b represents its solution.

2.2 Automated solvers

Normally, to solve a Bridges puzzle, one would start with solving relatively easy cells, and solving the puzzle step by step. For instance, in the example above, the cell on the left containing the number 6 can be solved instantly. There are only three directions where bridges can go from this cell, as the cell is located on the edge of the game. Because the number is 6 and bridges can at most be double, we know that to 'satisfy' and be done with this cell, we should connect double bridges from the cell containing 6 to the adjacent cells with the numbers 4, 3 and 3. On its turn, the cell in the bottom left with value 3 becomes easy to solve, as to satisfy this cell, there is only one cell left to go to with a single bridge, which is the cell with value 3 on its right side. We can continue reasoning this way to finally solve the puzzle.

In this study, we will look at an automated way to solve these puzzles using Satisfiability Modulo Theory (SMT) solvers. An SMT solver is an automated solver that's capable of solving mathematical formulas. [3] They're able to solve any mathematical theory that contain variables, integers, equalities, inequalities, propo-

sitions, predicates, and even programming data structures like strings and arrays. The way it solves these kind of formulas is by trying to find a valuation for all the variables in the formula such that the formula evaluates to $true$. If such a valuation can be found, we call the formula satisfiable. Otherwise, the formula is called unsatisfiable. When searching for such a valuation, all sorts of different methods are applied that may differ per SMT solver. This includes rewriting the formula to simpler forms and applying lemmas, but they partially also make use of reductions to the Boolean Satisfiability Problem, or SAT in short. [3]

As described in the introduction, SAT was the first problem that was proven to be NP-complete. SMT solvers still make use of SAT solving, as both solvers have a common goal: solving mathematical formulas. While SAT solvers are only scoped to solving Boolean formulas in Conjunctive Normal Form that consist of Boolean variables, SMT solvers add a whole layer over SAT solvers that extend the theories that we can solve. SAT's scope is within the class of NP, while SMT's scope is of even higher complexity than that. [3] Different SMT solvers apply different methods to be able to achieve their goals. While we often do not know what these methods are, we will give some example methods that translate problems normally not in the scope of SAT to something that is within the scope of SAT. For this study, there are two relevant problems that SMT can handle which SAT cannot:

Handling integers and (in)equalities

The main reason for this study to use SMT solvers instead of SAT solvers is to be able to use integers in our Boolean formulas. We want to be able to use them both for variables and for equalities and inequalities. SMT solvers allow integers to exist explicitly, whereas SAT solvers don't, as the latter only work with variables of the type Boolean. One possible way to encode integers using Boolean variables is by using a unary or binary encoding, [4] where a sequence of Boolean values make up a bitstring that resembles the decimal integer. This essentially is equivalent to reducing to SAT. This works for integers up to about 1000. Integers higher than that order of magnitude will make the encoding and solving very tedious and slow. For this study though, such high numbers will not be used and thus a unary or binary encoding should not be considered a problem.

Handling arbitrary, non-CNF Boolean formulas

Another useful property to have is the ability to use any form of Boolean formula as proposition. SAT solvers require formulas to be in CNF. This means that on top-most level, the entire formula should be a conjunction of clauses, where clauses consist of disjunctions of at least one literal. As SMT solvers accept any Boolean formula as input, it can be useful to have a method that translates arbitrary Boolean formulas to Boolean formulas in CNF. An efficient way of doing this is by making use of Tseitin transformations [6]. This transformation can take any Boolean

formula consisting of negations, conjunctions, disjunctions, implications or equivalences and transform it into CNF, which is satisfiable if and only if the original Boolean formula is. An important aspect here is that it can do the transformation in linear time, and the output formula also grows linearly with respect to the input formula, therefore not causing much overhead in the translation from SMT to SAT.

SMT solvers are thus amongst others capable of reducing problems to SAT that are normally not in the scope of SAT, for example using the methods described above. Do note that other than reducing to SAT, SMT solvers often have a lot of other elegant methods that are able to extend their powers over SAT. While in theory encoding Bridges as a SAT problem is possible, we choose SMT over SAT in this study to essentially outsource the work of handling integers, (in)equalities and arbitrary mathematical formulas to the SMT solver. This way, we can focus on formalizing the core rules of the game efficiently.

Chapter 3

Graph Encoding

The idea is that we define variables as well as Boolean formulas that encode a given game of Bridges. The Boolean formulas should be seen as constraints over the variables following the rules of the game as described in section 2.1. We know that if the SMT solver found a valuation that satisfies the Boolean formulas, it found a solution to the puzzle. Because the puzzles we're trying to solve are *uniquely solvable*, we also know that the found solution is in fact *the* solution in that case. In this chapter, we will look into an encoding where we see the puzzle as a graph where the edges resemble bridges and the nodes resemble cells. Other than formalizing the encoding within this section in a generic form, an example of a full encoding for an example puzzle can be found in Appendix A.

3.1 Structure

To build a Boolean formula, we need variables. In this first encoding we define two different kinds of variables. Firstly the main set of variables that, when a valuation for the Boolean formula is found, form the solution to the puzzle directly. These variables directly resemble the bridges in the game. Secondly we have to define helper variables that tell us something about the connectedness of all cells. These variables are needed to describe the connectedness of the entire network of cells as a whole (constraint 6), and will be explained in section 3.7. Finally, before we introduce the variables, it's important to note that throughout this research, we will refer to cells in a game using indexed numbers from top to bottom, left to right. The leftmost cell in the top row will be referenced to by index 0, the second cell in the top row will be reference to by index 1, continuing all the way to the rightmost cell in the bottom row which will be referenced to by index $n - 1$, where n is the total number of cells in the game.

Bridge variables

We define the set of bridge variables B in which each variable $\beta \in B$ can have a natural number value $\in \mathbb{N}$. Each variable takes two natural numbers x and $y \in \mathbb{N}$ as arguments such that variable $\beta_{x,y}$ represents the bridge between cell x and cell y. Important to note is that x and y are reflexive here because bridges are undirected, meaning that $\beta_{x,y} \equiv \beta_{y,x}$. The value of $\beta_{x,y}$ represents the *weight* of the bridge, 0 being non-existent, 1 being a single bridge and 2 being a double bridge. The number of bridge variables is equal to the number of bridges that can potentially exist, without taking into account the presence of other bridges. Each variable represents a different potential bridge. This means that there's a variable for every cell pair where both cells are on the same horizontal or vertical axis and there is no obstructing node in between. As clarification, below a figure can be found displaying all the bridge variables for an example puzzle.

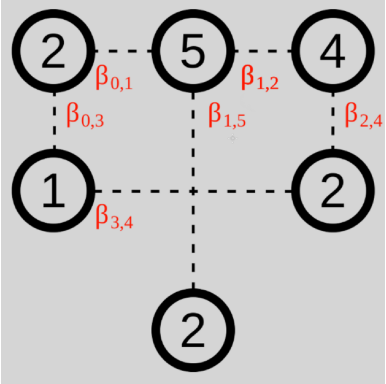


Figure 3.1: An example puzzle with all bridge variables visualized. The dotted lines represent the possible bridges to be drawn, disregarding their weight.

Connectedness variables

We introduce a set of connectedness variables Γ. A connectedness variable $\gamma \in \Gamma$ is a Boolean variable that takes two natural numbers n and $i \in \mathbb{N}$ as arguments such that variable $\gamma_{n,i}$ is $true$ if and only if cell 0 is connected to cell n in **at most** i bridges, regardless of the weight of the bridges. The connectedness variables act solely to express whether cells are connected in a given number of steps in some way or not. For example, if cell 0 is connected to cell 1 with a single bridge, and cell 1 is connected to cell 2 with a double bridge, $\gamma_{2,1}$ should be equivalent to $false$, but $\gamma_{2,2}$ (and all other $\gamma_{n,i}$ where $n = 2$ and $i \geq 2$) should be equivalent to $true$. For an entire network to be connected, it's enough to show that from a given cell in the network, you can reach every other cell. Therefore, it's enough to define connectedness variables in such a way that it only tells something about cell 0's connectedness to every other cell. The shortest path between two cells is a direct bridge, and the longest path can at most be the number of cells minus one. In

other words, the connectedness variables range from $\gamma_{0,1}$ to $\gamma_{n-1,n-1}$, where n is the number of cells in the game. In section 3.7 we describe how we can use these variables to formalize the connectedness of the entire network of cells and bridges.

3.2 Constraint 1: Bridges can only exist between two cells

This constraint is implicitly satisfied because the bridge variables' definition only allows bridges to exist between two cells. It is therefore not needed in this encoding to define an explicit constraint.

3.3 Constraint 2: Bridges are either horizontal or vertical

This constraint is implicitly satisfied because the bridge variables' definition only allows bridges to exist either on a horizontal axis or a vertical axis. It is therefore not needed in this encoding to define an explicit constraint.

3.4 Constraint 3: Bridges are either single or double

It is relatively simple to formalize this constraint. For single and double bridges we can simply allow bridge variables β to only have an integer value of 1 or 2 respectively in the final solution. However, since we defined the bridge variables to represent every *potential* bridge, the set of bridge variables contains variables representing bridges that should not exist in the final solution of the game. Therefore, we should in this encoding not only allow single or double bridges to exist, but also bridges that do not exist. We can formalize this as follows:

$$\bigwedge_{\beta \in B} (0 \leq \beta \leq 2)$$

3.5 Constraint 4: Bridges may not be obstructed

By definition of the bridge variables, this constraint is partially satisfied already. There are no bridge variables that allow bridges between two cells where that same bridge is obstructed by a cell in between. This means that we don't have to explicitly define that bridges may not be obstructed by cells. We do however still have to formalize that bridges may not be obstructed by other bridges. Implicitly, bridges can only be obstructed by bridges with the opposite orientation. For example, if a horizontal bridge would be obstructed by another horizontal bridge, this would mean that there is also a cell obstructing one of the bridges, so we already took care of that problem. All that's therefore left to do is to compare all horizontal bridges to all vertical bridges, and check for crossings. This is formalized below, where B_{hor} is the set of all horizontal bridges, B_{ver} is the set of all vertical bridges, and $bridgesCross(\beta_1, \beta_2)$ returns $true$ if and only if part of a horizontal bridge

overlaps with a vertical part of a different bridge, or vice-versa. This can easily be programmed with help of cell coordinates.

$$\bigwedge_{\beta_1 \in B_{hor}} \bigwedge_{\beta_2 \in B_{ver}} \big(\beta_1 > 0 \wedge \beta_2 > 0 \implies \neg bridgesCross(\beta_1, \beta_2)\big)$$

3.6 Constraint 5: Cell values are satisfied by bridge endpoints

For this constraint we have to formalize that the values in each of the cells are satisfied. Cell satisfaction is reached when the value inside a cell equals the number of bridge endpoints on the coordinates of that cell. To formalize this constraint we define some helper sets and functions. First we define set C as the set of all cells. We also define a function $val(x \in C)$ that takes as argument a cell and returns the value inside the cell. Finally we also define a function $match(x \in C, y \in B)$ that takes as arguments a cell and a bridge variable and returns that same bridge variable if one of the bridge's endpoints matches the coordinates of the cell, and 0 otherwise. A Formalization is then quite trivial:

$$\bigwedge_{c \in C} \big(val(c) = \sum_{\beta \in B} match(c, \beta)\big)$$

3.7 Constraint 6: Every cell is reachable from every other cell

The idea is that if, for every cell n and all path lengths i up to maximum path length we use connectedness variables $\gamma_{n,i}$ to formalize what it means for the involved cell n to be connected to cell 0, we can use those formalizations to ensure that every cell pair is in some way connected. Take the following game of Bridges in graph representation:

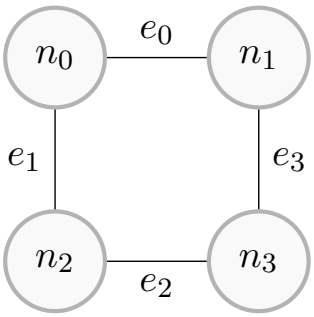


Figure 3.2: A connectedness graph of a game of Bridges. Each edge represents a potential bridge that may exist.

This undirected graph represents a game of Bridges, abstracting away from the values inside the cells and the weights of the bridges. We only care about the possible existence of a bridge between two cells. An edge being present represents that a bridge *may* be built between the relevant two cells, given their coordinates. We call this graph the *potentiality graph* of a game of Bridges. Using logic equivalence operators, we can express connectedness variables in terms of bridge variables and other connectedness variables (by means of adjacent edges and nodes in the potentiality graph) to build one single big constraint that describes the connectedness of the entire game of Bridges.

For example in figure 3.2, if we want to know what it means for node n_0 to be connected with node n_1 directly in one step, we are interested in edge e_0 only, which is represented by bridge variable β_{n_0,n_1}. There is no other way to reach node n_1 from node n_0 in one step other than via e_0, so we can safely say that $\gamma_{n_1,1} \iff \beta_{n_0,n_1} > 0$.

A somewhat more interesting example to investigate would be getting from n_0 to n_1 in at most two steps. We not only need to reason about getting from n_0 to n_1 directly in one step, but also about getting from n_0 to n_1 in two steps. For any connection in 2 steps or more, we can approach the problem recursively, by describing the connection in terms of one of the neighbors of the destination node in one less step, and adding the last step explicitly. In the scenario of going from node n_0 to node n_1, we could either worry about getting from n_0 to n_0 in one step (which is trivial) and take edge e_0 or worry about getting from n_0 to n_3 in one step (which is impossible) and take edge e_3. If we try to formalize this we would get $\gamma_{n_1,2} \iff \gamma_{n_1,1} \vee (\gamma_{n_0,1} \wedge \beta_{n_0,n_1} > 0) \vee (\gamma_{n_3,1} \wedge \beta_{n_3,n_1} > 0)$.

Remark The reason we introduced a second argument i in the connectedness variables (that represents the maximum number of steps it takes to reach one cell from another) instead of simply defining a connectedness variable as the connectedness of two nodes in *any* number of steps is that if we don't, we get a circular dependency between Boolean formulas which results in unwanted behaviour. For example, imagine we want to describe the connectedness between nodes n_0 and n_1 and the connectedness between nodes n_0 and n_3 as displayed in figure 3.2. Using more simple connectedness variables without an argument representing the number of steps, we would get the following two formalizations:

$$\gamma_{n_1} \iff (\gamma_{n_0} \wedge \beta_{n_0,n_1} > 0) \vee (\gamma_{n_3} \wedge \beta_{n_3,n_1} > 0)$$
$$\gamma_{n_3} \iff (\gamma_{n_1} \wedge \beta_{n_1,n_3} > 0) \vee (\gamma_{n_2} \wedge \beta_{n_2,n_3} > 0)$$

If we set all colored variables to $true$ and the uncolored variables to $false$, both equivalences would evaluate to $true$, thus satisfying our formalizations. This valuation would therefore imply however, that nodes n_0 and n_1 are connected and nodes n_0 and n_3 are connected, while the only bridge actually being present in the game is the bridge between nodes n_1 and n_3. This obviously is an impossible scenario, and therefore this method of formalizing the properties does not work. When we add an argument that tells us something about the number of steps the nodes are connected in, this circular dependency is cut out.

If we want to find a solution for which it holds that from every node all other nodes are reachable, we need to abstract away from the example explained earlier, and reason about connectedness of node pairs in graphs in general. We differentiate between four different scenarios:

Node 0 is connected to itself in at most any number of steps

Node 0 is always reachable from itself, namely in 0 steps. Therefore, the connectedness variable representing node 0 with i higher than 0 should be set to $true$, as the variable denotes that node 0 is connected in *at most* i steps. We can trivially formalize this as:

$$\gamma_{0,i}$$

A node is connected to node 0 in at most 1 step

When investigating whether a node can be connected to node 0 in at most 1 step we can encounter two cases: Node n is a direct neighbor of node 0, meaning there exists an edge between the two nodes in the potentiality graph, or node 0 and node n are not direct neighbors, meaning there can never be a direct bridge between the two nodes, and thus there is no edge between the two nodes in the potentiality graph. In the first case, the two cells are actually connected in exactly one step if and only if the corresponding bridge has a weight larger than 0. In the latter case,

the two cells can never be connected in exactly one step. This can be formalized as follows:

$$\begin{cases} \gamma_{n,i} \iff \beta_{0,n} > 0 & \text{(if } e_{0,n} \text{ exists in the potentiality graph)} \\ \neg\gamma_{n,i} & \text{otherwise} \end{cases}$$

(where $n \neq 0$ and $i = 1$)

A node is connected to node 0 in at least 2 steps

If we look at the connectedness of node 0 and node n in 2 steps or more, the basic idea is to express node 0's connectedness with another node that is directly neighboring node n. Either we think about getting from node 0 to the node n's northern neighbor in one less step, and take a final step south to node n, or we think about getting from node 0 to node n's eastern neighbor in one less step, and take a final step west to node n, etc. Intuitively, we also include a recursive term $\gamma_{n,i-1}$ in each definition to indicate that if you can reach node n in at most $i-1$ steps you can also reach node n in at most i steps. The following formalization describes the connectedness of node 0 with node n in at least 2 steps, where N_n resembles the set of directly neighboring nodes of node n in the potentiality graph.

$$\gamma_{n,i} \iff \gamma_{n,i-1} \vee \bigvee_{z \in N_n} (\gamma_{z,i-1} \wedge \beta_{z,n} > 0) \quad \text{(where } n \neq 0 \text{ and } i > 1)$$

A node should be connected to node 0 in at most maximum path length

Finally, now we've formalized what it means for all nodes to be connected to node 0 in some way, we should ensure they *are* actually connected. The choice to define $\gamma_{n,i}$ in such a way that i resembles the maximum number of steps it takes to reach n from 0 allows a very trivial formalization of this property if we take N to be the number of nodes in the game:

$$\gamma_{n,i} \quad \text{(where } n \neq 0 \text{ and } i = N - 1)$$

Now we've formalized every possible case we can encounter when investigating the connectedness of node 0 to every other node for all possible maximum number of steps, we can build the constraint. If we traverse over every connectedness variable $\gamma_{0,1}$ to $\gamma_{n-1,n-1}$, express them in terms of other nodes in the potentiality graph and their adjacent edges following the scenarios above, and combine them into one large conjunction of Boolean formulas, we've built a constraint that ensures connectedness of a an entire Bridges solution.

Chapter 4

Grid Encoding

Now we have a first encoding of the puzzle in SMT, it's interesting to look at other methods of encoding the puzzle as well. While the graph encoding in the previous chapter forms an intuitive way to encode the puzzle, we choose to focus on an at first sight more naive encoding of the puzzle in this chapter. We will look at the puzzle from a perspective where we view the puzzle as 2D grid where every field can be a separate piece: A cell, a bridge piece, or nothing. Just like in the previous chapter, we will again provide an example of a full encoding of a puzzle in Appendix A.

4.1 Structure

In the grid encoding we also use two different kinds of variables. This time, the puzzle is mainly represented by a set of variables that represent a field in the puzzle grid. Again, the variables form a direct solution to the puzzle if a valid valuation for the Boolean formula is found. Instead of using potential bridges as variables, we completely discard the concept of a bridge as something that exists as one entity, but use the field variables to build bridges piece by piece. Secondly, we use the same connectedness variables again as helper variables for the connectedness constraint. They are defined in the exact same way as described in section 3.1, but the way we use them to build the constraints will be slightly different, as described in section 4.5. Even though the potential bridges that can be built are not defined explicitly as variables in the grid encoding, we can still deduce what all potential bridges are since we know what the coordinates of all the cells in the game are. We can therefore still define the set of bridges B which consists of all possible bridges (every cell pair where both cells are on the same horizontal or vertical axis and there is no obstructing node in between). This set turns out to be useful when formalizing the connectedness constraint.

Field variables

We define Φ as the set of field variables. Each variable $\varphi \in \Phi$ can have a value $p \in P$, where the set of pieces $P = \{e, —, ═, \mid, \|, C\}$. Respectively, the pieces available are empty field pieces, single horizontal bridge pieces, double horizontal bridge pieces, single vertical bridge pieces, double vertical bridge pieces, and cells with a value. The value of the field variable represents what piece is placed in the field. The number of field variables equals the number of fields in the grid plus the number of fields needed for the border (see section 4.2). Using all six different field pieces, we can fill up the grid such that it forms a valid puzzle.

4.2 Valid field piece constraint

As in this encoding, the allowed pieces that are placed in the grid have to be defined and are not explicitly part of the game rules, we have to implement a constraint that ensures that only valid pieces can be placed on fields. Most important is that we formalize the initial configuration of an 'empty' game. Only the fields that we know contain a cell may contain a cell. We formalize this explicitly. Aside from that, it's also important to note that it does not hold that every field can contain every piece. Fields on the northern and southern edges of the grid can only be empty, cells or horizontal bridge pieces, fields on the western and eastern edges of the grid can only be empty, cells or vertical bridge pieces, and in the same way corner fields can only be empty or cells. A formalization could become quite complicated, but by defining a border of extra fields around the actual game where we force the fields to be empty, formalization becomes quite trivial. The extra empty fields around the game allow the other constraints to check neighboring fields without having to take into account the possibility of going out of bounds of the actual game.

$$\bigwedge_{\varphi \in \Phi} \begin{cases} \varphi = e \\ \qquad \text{if } \varphi \text{ is on the extra border drawn around the game} \\ \varphi = C \\ \qquad \text{if } \varphi \text{ is supposed to contain a cell} \\ \bigvee_{p \in P \setminus \{C\}} (\varphi = p) \\ \qquad \text{if } \varphi \text{ is not supposed to contain a cell} \end{cases}$$

4.3 Constraints 1, 2, 3 and 4

Recall the first four game rules:

1. Bridges can only be built between two cells

2. Bridges can be built either horizontally or vertically
3. Bridges can be single (counting as one) or double (counting as two)
4. Bridges may not be obstructed by other bridges or cells

A very interesting thing about the grid encoding is that all these game rules can actually be covered by one constraint. When formalizing one of these rules, we quickly find that the most straightforward way to do this is to make implications on what neighboring fields can contain given the piece on the current field. It turns out that these implications on the neighboring fields always result in implications that directly also cover the other three constraints in the list above. For example, if we look at rule 4: ”Bridges may not be obstructed by other bridges or cells”, we can come up with the following constraint: If a given bridge piece is a single vertical bridge piece, the northern and southern neighboring pieces can only be a cell or another single vertical bridge piece. Looking at this formulation, we can see that it directly also covers rule 1, 2 and 3. Of course, this also holds for any combination of single/double and horizontal/vertical bridge pieces, and should be included in the formalization. Figure 4.1 visualizes all the scenarios that we must include in our constraint.

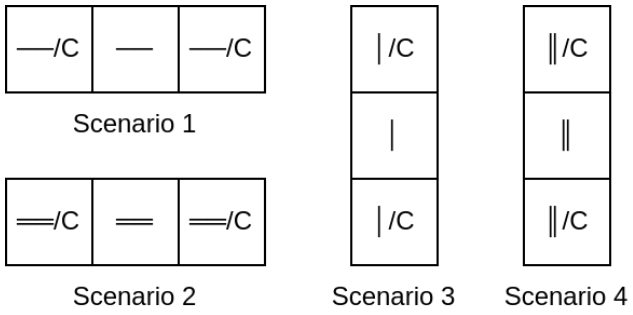


Figure 4.1: A visualization of all the different scenarios that are allowed by the constraint

> ***Note*** The fact that bridge *pieces* can only be single or double and horizontal or vertical, is ensured by the explicit constraint defined in section 4.2. Rule 2 and 3 in this section are about *entire bridges* being single or double and horizontal or vertical.

By applying this reasoning to formalize implications on all neighboring fields *that matter* for every field that contains a bridge piece in the game, we can construct a relatively simple but big constraint to cover all first four game rules. Note that if a field contains a horizontal bridge piece, we only need to set restrictions on their horizontally neighboring fields if we apply the same logic to the perpendicular direction. Naturally, this holds both ways. Figure 4.2 displays why this reasoning works. Also note that if a field is empty or is a cell, its neighbors can be anything.

It is therefore not needed to explicitly define constraints on the neighboring fields for fields that are empty or a cell. The formalization is as follows:

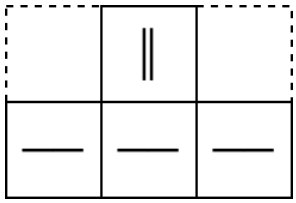


Figure 4.2: An example of a scenario that is prohibited by the constraint. The horizontal bridge pieces meet the requirements of scenario 1 in figure 4.1, but the vertical bridge piece clashes with scenario 4 in figure 4.1

$$\bigwedge_{\varphi_1 \in \Phi_G} \bigwedge_{b \in BP} \begin{cases} \big((\varphi_1 = b) \implies \bigwedge\limits_{\varphi_2 \in NH_{\varphi_1}} (\varphi_2 = b \vee \varphi_2 = C)\big) & \text{if } b \text{ is horizontal} \\ \big((\varphi_1 = b) \implies \bigwedge\limits_{\varphi_2 \in NV_{\varphi_1}} (\varphi_2 = b \vee \varphi_2 = C)\big) & \text{if } b \text{ is vertical} \end{cases}$$

Here $\varphi_1 \in \Phi_G$, where Φ_G is the set of all fields that are actually part of the game, or in other words, are not on the extra border around the game that we introduced. The set of bridge pieces BP ($\subset P$) $= \{—, ═, \mid, \|\}$, the set NH_φ represents the set of fields directly neighboring field φ *horizontally*, and the set NV_φ represents the set of fields directly neighboring field φ *vertically*.

4.4 Constraint 5: Cell values are satisfied by bridge endpoints

This constraint has a similar formalization as it has in the graph encoding. Since we don't work with bridge endpoints on cells in this encoding but with neighboring fields of a cell that can contain a piece, we must map all neighboring field pieces to a natural number value (depending on the direction of the piece with respect to the cell) in order to count. To formalize this constraint we again define set C as the set of all cells and function $val(x \in C)$ that takes as argument a cell and returns the value inside the cell. The set of neighboring fields NF_c represents the set of fields directly neighboring cell c. In the formalization, $ite(p, a, b)$ is a Boolean formula that evaluates to a if proposition p evaluates to $true$, and evaluates to b otherwise. We use a case distinction between vertical and horizontal neighbor pieces to formalize this constraint:

$$\bigwedge_{c \in C} \big(val(c) = \sum_{\varphi \in NF_c} map(c, \varphi)\big)$$

Where

$$map(c, \varphi) = \begin{cases} ite\Big((\varphi = \,|\,), 1, ite\big((\varphi = \,\|\,), 2, 0\big)\Big) & \text{(if } \varphi \text{ is north or south of } c\text{)} \\ ite\Big((\varphi = —), 1, ite\big((\varphi = ═), 2, 0\big)\Big) & \text{(if } \varphi \text{ is east or west of } c\text{)} \end{cases}$$

4.5 Constraint 6: Every cell is reachable from every other cell

For the more complex connectedness constraint in the grid encoding we use the same approach as in the graph encoding, which is described in detail in section 3.7. Earlier we've defined the set of all potential bridges B again. This also means that we are still able to use the concept of the potentiality graph. We can thus use the same case distinction and formalizations as in section 3.7. In order to modify the formalizations to fit the grid encoding though, we need to replace the bridge variables we used in section 3.7 with field variables that neighbor the concerning node. To specify that cell 0 and cell n are connected, it's enough to only specify that a neighboring field of cell 0 contains a bridge piece in the direction of node n, as all constraints defined earlier imply that if a bridge piece is present, it should be part of an entire bridge between two nodes.

Node 0 is connected to itself in at most any number of steps

We use the same formalization as in the graph encoding:

$$\gamma_{0,i}$$

A node is connected to node 0 in at most 1 step

With respect to the formalization of this case in the graph encoding, the only thing that changes is that we not only need to specify the cases where $e_{x,y}$ exists in the potentiality graph, but we also need to differentiate in which direction $e_{x,y}$ exists, as the field pieces to be used are dependent on that. We define $\phi_{x,y}$ as the neighboring field φ of cell x in the direction of node y. We use the ϕ notation instead of the φ notation to differentiate from the field coordinate notation in the running example in Appendix A. The formalization of the constraint is as follows:

$$\begin{cases} \gamma_{n,i} \iff (\phi_{0,n} = —) \vee (\phi_{0,n} = ═) & \text{(if } e_{0,n} \text{ exists in the potentiality graph horizontally)} \\ \gamma_{n,i} \iff (\phi_{0,n} = \,|\,) \vee (\phi_{0,n} = \,\|\,) & \text{(if } e_{0,n} \text{ exists in the potentiality graph vertically)} \\ \neg\gamma_{n,i} & \text{otherwise} \end{cases}$$

(where $n \neq 0$ and $i = 1$)

A node is connected to node 0 in at least 2 steps

Just as in the case where a node n is connected to node 0 in at most 1 step, we don't have to change the formalization of this case radically compared to the formalization in the graph encoding. In fact, we can still apply the same logic, but again we have to differentiate in directions when it comes to deciding what kind of bridge pieces need to be placed. N_x resembles the set of directly neighboring nodes of node x in the potentiality graph, and $\phi_{x,y}$ is the neighboring field φ of cell x in the direction of cell y. We get the following formalization:

$$\gamma_{n,i} \iff \gamma_{n,i-1} \vee \bigvee_{z \in N_n} \begin{cases} \Big(\gamma_{z,i-1} \wedge \big((\phi_{z,n} = \text{—}) \vee (\phi_{z,n} = \text{═})\big)\Big) \\ \qquad \text{(if } e_{z,n} \text{ is horizontal in the pot. graph)} \\ \Big(\gamma_{z,i-1} \wedge \big((\phi_{z,n} = \text{│}) \vee (\phi_{z,n} = \text{║})\big)\Big) \\ \qquad \text{(if } e_{z,n} \text{ is vertical in the pot. graph)} \end{cases}$$

(where $n \neq 0$ and $i > 1$)

A node should be connected to node 0 in at most maximum path length

We use the same formalization as in the graph encoding, with N being the number of nodes in the game:

$$\gamma_{n,i} \quad \text{(where } n \neq 0 \text{ and } i = N - 1\text{)}$$

Finally, by combining every case described above in one big constraint using conjunctions as we traverse over every connectedness variable $\gamma_{0,1}$ to $\gamma_{n-1,n-1}$, we can ensure connectedness of an entire Bridges puzzle.

Chapter 5

Generating

Now we've looked into solving games of Bridges using SMT solvers, it makes sense to look into how we can use SMT solvers in generating puzzles as well. The idea is that if we know how to find solutions to games of Bridges using SMT solvers, we can use that to iteratively add bridges and cells to existing solutions to generate new solvable puzzles.

5.1 Unique solvability

Generating puzzles in general is pretty trivial; we can simply start drawing valueless cells connected by either single or double bridges, keep on expanding, for all cells fill in a value that makes sense with the drawn bridges to finally remove all bridges. That leaves us with a solvable puzzle. However, this gives us no guarantee that the puzzles are *uniquely* solvable. This means that a given solution to a puzzle is the one and only solution. This is a property we are looking for, since having multiple solutions for one puzzle could cause non-determinism while trying to solve the puzzle. That would defy the point of playing the game.

We can use SMT solvers to iteratively build a game of Bridges and by checking if the game is uniquely solvable after we've filled in the cell values and removed the bridges. At the end of each iteration, we feed the puzzle to an SMT solver using one of our encodings, build the constraints described in the previous chapters (depending on the encoding chosen), and determine if the puzzle is uniquely solvable by adding a final constraint that tells the SMT solver that the solution that we've built ourselves should not be a solution.

For the graph encoding, that constraint is formalized as follows:

$$\neg \bigwedge_{\beta \in B} (\beta = solution(\beta))$$

For the grid encoding, the constraint is formalized as follows:

$$\neg \bigwedge_{\varphi \in \Phi} (\varphi = solution(\varphi))$$

In both formalizations, the $solution()$ function takes a bridge variable β or a field variable φ that returns the bridge weight of the corresponding bridge in the solution that we generated or respectively, the field piece of the corresponding field in the solution that we generated.

If the SMT solver's output is that the Boolean formula is unsatisfiable, it means that the solution that we generated is the only solution to that game's configuration of cells and possible bridges, and thus the puzzle is uniquely solvable. If the SMT solver's output is that it found a valuation that satisfies the Boolean formula, it means that it found another solution to the puzzle, and the puzzle configuration is thus not uniquely solvable. We are therefore in search of *unsatisfiability* while generating, in contrast to solving, where we were in search of *satisfiability*.

5.2 Algorithm

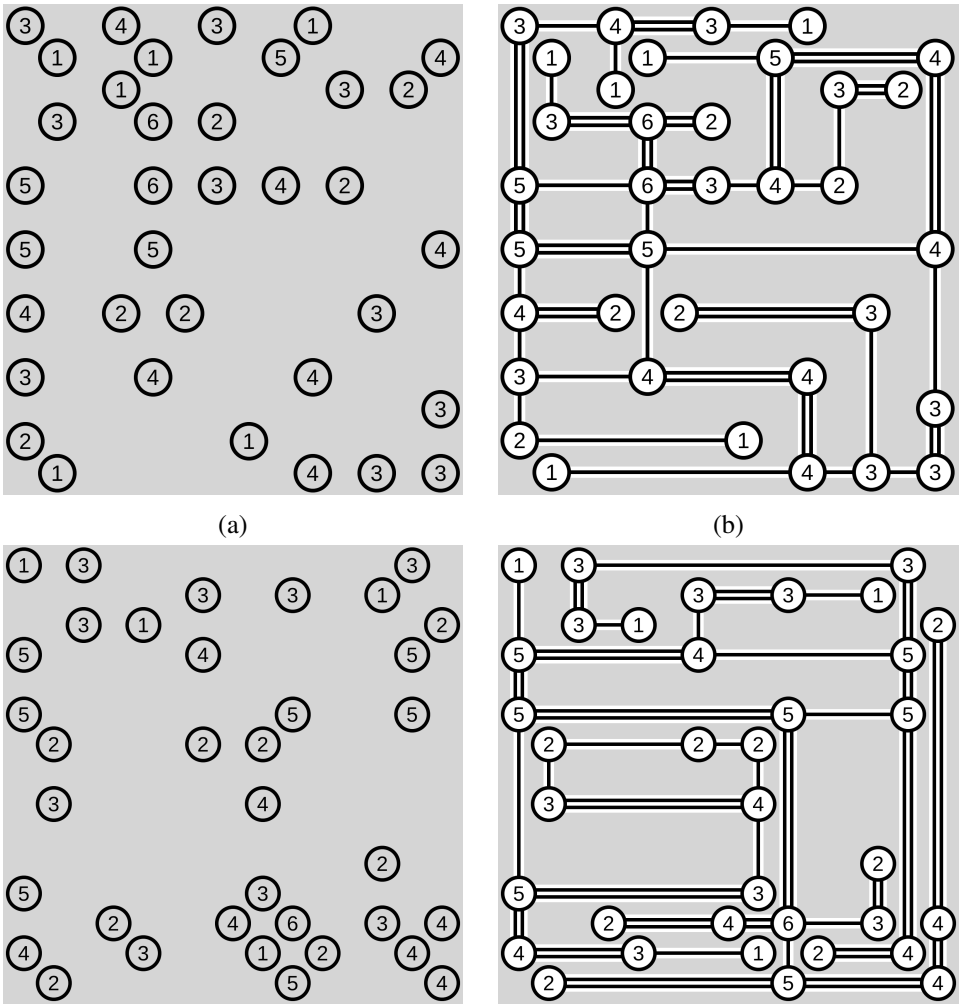
The algorithm that we will use will be as follows: We take as arguments a field size, a number of cells we strive to place on the field, and the encoding we will use to verify unique solvability.
We start off with a single cell placed randomly on the field, without a value. From there on, we enter the main loop that ends when either we've placed the desired number of cells, or we reach a threshold of 1000 tries to create a uniquely solvable puzzle but failed. In the loop, we begin with setting a counter to 0 that keeps track of how often we tried to place a new cell.
Then we enter a new loop that ends when we found a field where we can actually place a new cell. In that second loop, we first check if the number of tries to place a new cell has reached a threshold of 1000. If we have, we can most probably conclude that in the current game configuration no more cells can be added, and we raise an error.
If not, we continue by selecting a random cell from the existing network to work from. From that source cell, we determine all directions in which we can build a bridge, taking into account other existing bridges, cells, and the borders. We select a possible direction at random, and determine the number of available fields we have in that direction to place a new cell, again taking into account all other existing bridges, cells, and the borders. In the range of fields that we have left, we select a random field and place a new cell candidate. We also increment the counter that keeps track of how many times we tried to find a new cell. If it appears that from the selected source cell there are no possible directions to go and place a cell, the inner loop repeats with a different source cell.
After the inner loop, we have found a candidate cell that we add to the network,

connected via a bridge from the chosen source cell. This bridge will either have weight 1 or 2 at random. Now, from this candidate cell, we determine all directions in which an already existing cell in the network can be found. For all these possible directions from the candidate cell, there is a 50% chance to build a bridge (again with weight 1 or 2 at random) from the candidate cell to the already existing cell in the network in the given direction. This step allows loops to exist in the puzzles we generate.
Now that the candidate cell and bridges are added we can check for unique solvability. We count all bridge endpoints per cell, set the cell values accordingly and remove all bridges. We now have a puzzle that we can encode (using the encoding given as argument) as an SMT problem, and feed it to a solver as described in section 5.1. If the puzzle is not uniquely solvable (constraint in section 5.1 is satisfiable), we increment the counter of tries to create a uniquely solvable puzzle, and restore the game state to what it was at the beginning of the main loop, that is, without the new candidate cell and added bridges. If the puzzle is still uniquely solvable (the constraint in section 5.1 is unsatisfiable), we continue with the game state in which the new valueless cell and all new bridges are still added, we reset the counter of tries to create a uniquely solvable puzzle, and increment the counter of the number of cells that we added. At this point we've reached the end of a main loop iteration, and the cycle repeats until either we've placed the desired number of cells, or we reach a threshold of 1000 tries to create a uniquely solvable puzzle but failed.
When we exit the loop, we explicitly check if the threshold of 1000 tries to create a uniquely solvable puzzle has been reached, and raise an error in that case. In case the loop was ended because we've reached the desired amount of cells, we are successful, and we can count all bridge endpoints per cell, set the cell values accordingly, remove all bridges, and return the puzzle.

To summarize; we now have an algorithm that adds a new cell via a bridge to the network every iteration, adds random bridges to existing cells to create loops, and each iteration evaluates whether the puzzle is uniquely solvable or not. The algorithm always produces either a uniquely solvable puzzle of a given field size and a given number of cells, or an error if a uniquely solvable puzzle cannot be generated using the parameters in a certain run. Because the algorithm works with randomness, this result can be different in a next run of the algorithm. On the next page, a concise version of the algorithm is described in pseudocode for improved readability. Also included are two example 15x15 generated puzzles, both presented in solved form as well for clarity.

Algorithm 1: generatePuzzle

Data: $fieldSize \geq 3$, $cellGoal \geq 2$, $solver \in \{Graph, Grid\}$
Result: A puzzle with $cellGoal$ cells of size $fieldSize$ x $fieldSize$
$puzzle \leftarrow$ new puzzle of size $fieldSize$ x $fieldSize$ with 1 cell without value, randomly placed;
$cellCount \leftarrow 1$;
$triesUnique \leftarrow 0$;
while $cellCount < cellGoal$ *and* $triesUnique < 1000$ **do**
    $triesNewCell \leftarrow 0$;
    $srcCell \leftarrow null$;
    $newCell \leftarrow null$;
    while $newCell = null$ **do**
        if $triesNewCell \geq 1000$ **then**
            Error: Not able to generate puzzle with enough cells using given parameters in this run
        end
        $srcCell \leftarrow$ Random existing cell;
        $newCell \leftarrow$ Random new cell neighboring $srcCell$;
        `// null if no new cell/bridge can be placed from srcCell`
        $triesNewCell \leftarrow triesNewCell + 1$;
    end
    $puzzle \leftarrow puzzle$ with $newCell$ and bridge (weight 1 or 2 at random) from $srcCell$ to $newCell$ added;
    for $n \in neighbors(newCell)$ **do**
        50% chance for $puzzle \leftarrow puzzle$ with bridge (weight 1 or 2 at random) from $newCell$ to n;
    end
    if $puzzle$ *is not uniquely solvable (using* $solver$*)* **then**
        $triesUnique \leftarrow triesUnique + 1$;
        $puzzle \leftarrow puzzle$ with $newCell$ and all new bridges removed;
    else
        $triesUnique \leftarrow 0$;
        $cellCount \leftarrow cellCount + 1$;
    end
end
if $triesUnique \geq 1000$ **then**
    Error: Not able to generate puzzle with unique solution using given parameters in this run
else
    Count bridge endpoints per cell and set cell values;
    $puzzle \leftarrow puzzle$ with all bridges removed;
    return $puzzle$;
end



(a) (b)

(c) (d)

Figure 5.1: Two examples of generated puzzles. Figure 5.1a and figure 5.1c represent the generated puzzles, while figure 5.1b and figure 5.1d represent their solutions, respectively

Chapter 6

Experiments

In this chapter we will go over all experiments that were done and results that were found during this study. In general, we measured constraint construction and solving times with both SMT encodings on verified existing uniquely solvable puzzles and on our own generated uniquely solvable puzzles. We also measured the generation times of puzzles using both encodings. Finally, we also had some volunteers solve a number of puzzles, and had them track their solving times.

6.1 Setup

To conduct experiments, we need to make sure we are in possession of a big pool of different puzzles as well as a way to measure the running times of the solver and the generator with as little influence by external factors as possible.

6.1.1 Puzzle tool

Simon Tatham's Portable Puzzle Collection [10] offers a steady platform to play, solve, generate, import and export puzzles of Bridges easily. The website offers several difficulties of puzzles, as well as some tweakable parameters such as the field size, maximum number of bridges per direction and allowing loops or not. S&T's tool can convert any puzzle to an ID and vice versa, which allows us to solve puzzles generated by the tool with our own solver, but also allows the tool to solve puzzles generated by us. Using a python script we were able to extract puzzles in large batches from the tool. We retrieved 100 easy puzzles and 100 hard puzzles for all of the following field sizes: 7x7, 15x15, 30x30 and 50x50. These are the puzzles we will be using in our solving experiments mostly.

6.1.2 Measuring

To measure how effective our solving and generation methods are, we added several different timing measurements throughout our algorithms. For the solvers, we measured the time it takes to build all the actual constraints per puzzle, the time it takes to find a valuation that satisfies all those constraints and the total time to get from puzzle to solution. For the generations, we measured the total time it takes to get from nothing to a uniquely solvable puzzle, and for every main loop iteration of the algorithm we measured the time it takes that iteration to complete and the time it takes that iteration to check whether there is a unique solution. Other than that we also measured for each generated puzzle what the maximum number of consecutive failed tries for finding a new cell and finding a uniquely solvable puzzle is. In the same way we can also measure how often we fail to generate puzzles. We fail to generate a puzzle if we reach the threshold of 1000 tries to either create a uniquely solvable puzzle or to place a new cell in the game, as can be recognized by when the algorithm raises an error.
We run all our experiments on an external Ubuntu (22.10 x64) server with 8gb RAM, a 4 core CPU and SSD main memory. No other active tasks were executed on this external server while the experiments ran for best results. For both solving and generations, we use SMT solver 'SMTInterpol' in this study. [5]

6.2 Solving puzzles

In this set of experiments we let our SMT solver loose on several sets of 100 puzzle files. We have sets of 100 7x7, 100 15x15, 100 30x30 and 100 50x50 easy puzzles from S&T's tool, four sets of 100 hard puzzles of the same sizes from the tool and four sets of 100 puzzles that we've generated ourselves of the same sizes using the graph encoding. We only used the graph encoding for generating since (as we will see in section 6.3) it is faster and the encoding used doesn't change the algorithm and thus what kind of puzzles we will generate. We solve all puzzles once with the graph encoding and once with the grid encoding, and note down all the timing measurings. Timeouts during solving did not happen, fortunately.

The most relevant results are contained in the graphs in this section, while in Appendix A we can find more elaborate results of all the experiments that we ran. In figure 6.1 we can find an easy to observe graph that contains boxplots per puzzle category. The y-axis contains the total time in milliseconds it took the SMT solver to build constraints and satisfy them to get from a puzzle to a solution. The blue boxplots represent puzzles solved with the graph encoding, the orange boxplots represent puzzles solved with the grid encoding. In table A.2, table A.3 and table A.4 we can find extended tables that contain more elaborate data on the actual times measured.

In general, we observe that the grid encoding becomes slower in solving compared to the graph encoding as the game sizes grow. The exact factor depends on the

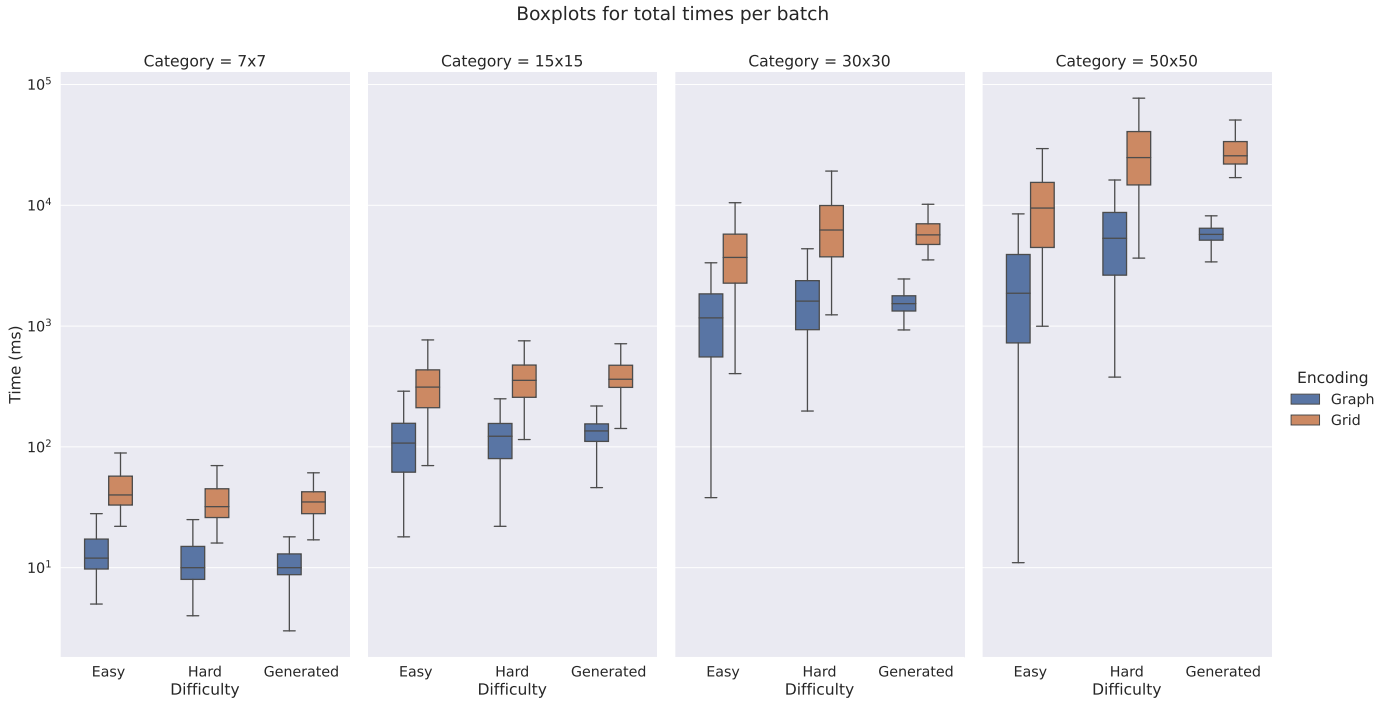


Figure 6.1: Boxplots on the total solution times per puzzle category and encoding

size of the puzzle. For puzzle sizes 7x7 and 15x15, the scaling between solving with the graph encoding vs. solving with the grid encoding always lies around 3, for 30x30 the factor lies around 4, and for 50x50 puzzles the factor quickly grows to 8. We can observe a rough exponential growth in solving times. This is to be expected, as SMT is typically NP-Hard, implying at least exponential growths. The variations in solving times are also often also very high, with outliers downwards for the graph encoding specifically.

It is to be expected that the growth in time is only roughly measurable to be exponential, because there are more factors that have influence here than just the size of the puzzle. For the graph encoding for example, the number of bridge variables depend on the number of bridges that can potentially be built, and this does not have a very direct relationship with the field size, but rather with the placement of all cells. The grid encoding does not rely on bridge variables, but only on variables for every field. Because it creates (implicational) constraints for every field in the game regardless of the fact whether a bridge can potentially be present there or not, it is easier to determine constraint growth factors there. Another factor is the number of cells in the game. While there is a growth visible in number of cells with

respect to the field size, the gaps between the cells can still be large or small. Our sample puzzle sets contain several large size puzzles that contain only a handful of cells, resulting in fewer bridge variables, and thus the graph encoding scoring better. This explains the downward outliers for the graph encoding in figure 6.1.

We can see though, when we look at tables A.2, A.3 and A.4, that the growth in total time it takes to find a solution using the grid encoding is mostly present in finding a satisfiable valuation for the constraints, and not so much in the time it takes to build the constraint. This is most probably because building constraints is a matter of defining variables and placing them in formulas as specified. However, actually solving those formulas involves taking into account every defined formula and finding a valuation of variables that fits for every formula. This takes trial and error, applying tactics (by the SMT solver) to simplify formulas, and calculations to get a result. On top of that, The longer a constraint is and the more variables it contains, the longer it takes for the SMT solver to find a valuation that is satisfying every constraint. Considering that the formulas of the grid encoding's constraints are much longer than those in the graph encoding because of the dependence on field variables, the numbers shown in the tables make sense. The growth is visible in the solving times, and not on the construction times.

Another interesting thing here is that we can see that for smaller game sizes, the time it takes to build the constraints using the graph encoding is relatively short with respect to building them with the grid encoding. In large size puzzles the time it takes the graph encoding to do so nears the time it takes for the grid encoding to do so. This probably is related to the phenomenon described earlier where the number of variables and formulas is dependent on the number of possible bridges in the graph encoding, but not in the grid encoding. For smaller sized puzzles, there is a smaller chance to find potential bridges than in bigger sized puzzles, because in bigger sized puzzles more cells can be placed.

When it comes to the difficulty of the puzzles from S&T's tool, we see that the difficulty itself is not a very influential factor in terms of solving time. There is a difference observable in larger sized puzzles, but not as much as one would expect. This is remarkable, as we would expect that the human solvability factor present in difficulty of a puzzle (see section 6.4) would lead to more potential for SAT and SMT solvers to apply simpler methods like unit propagation and other simplification methods. Although we cannot directly conclude that such methods are not applied as much as we would expect, it is verified in the results that solving puzzles is definitely different for humans than it is for computers. This is most of all visible in the observation that puzzles generated by us (that do not have a human solvability factor) have very consistent solving times with respect to those from S&T's tool.

In general, when we inspect the constraints described in chapter 3 and 4, we can say that when it comes to the graph encoding, the size of constraint 3 grows linearly with the number of potential bridges present in the puzzle, the size of constraint 4

grows quadratically with the number of potential bridges present in the puzzle, the size of constraint 5 grows linearly with the number of cells in game, and the size of constraint 6 grows quadratically with the number of cells in the game. When it comes to the grid encoding, the valid piece constraint grows quadratically with the size of the game, constraints 1, 2, 3 and 4 (which is one constraint in the encoding) grow quadratically with the size of the game, constraint 5 grows quadratically with the number of cells in the game, and constraint 6 grows quadratically with the the number of cells in the game. While we don't have an exact measurement on how the number of cells and the number of potential bridges in the game grows with the size of the game, we can make an approximation that this is also a quadratic factor.

Given everything above, we can conclude that the graph encoding is effectively always faster than the grid encoding because it has fewer quadratically growing formulas than the grid encoding. Moreover, the graph encoding especially works faster on smaller puzzles than the grid encoding because the number of cells and the number of potential bridges (which are relevant for the graph encoding) are more linear with the field size than quadratic, opposed to the field variables in the grid encoding growing quadratically with the field size.

> ***Remark*** In a previous version of the grid encoding, we did not implement the border around the edge of the game. This not only made coding the constraint building of the valid field piece constraint a lot harder than it is with the border, but it also made another element we previously included in the grid encoding redundant to include. What we previously included in the valid field piece constraint was that if a certain field can only be a bridge piece or an empty field, it can *explicitly* not be a cell. Before we added the border around the game grid that forces fields to be empty, this made solving puzzles faster with a factor around 10. Interestingly enough, when we added the border around the game, it turned out that explicitly mentioning that a field cannot be a cell became redundant. With the border around the grid, it was sufficient to only state that a field can only be a bridge piece or an empty field, implicitly stating that the field cannot be a cell. Because with the border around the grid added being a more efficient grid encoding, we decided not to note down the measurements we made on solving puzzles without the border in this paper. This does raise questions though, which we will come back to in Chapter 8: Future work. An example of how the valid field piece constraint in the old grid encoding was formalized:

$$\bigwedge_{\varphi \in \Phi} \begin{cases} \varphi = C \\ \qquad \text{if } \varphi \text{ is supposed to contain a cell} \\ \neg(\varphi = C) \wedge (\varphi = e) \\ \qquad \text{if } \varphi \text{ is not supposed to contain a cell and is on the corner of the grid} \\ \neg(\varphi = C) \wedge \bigvee_{p \in P_{NS}} (\varphi = p) \\ \qquad \text{if } \varphi \text{ is not supposed to contain a cell and is on the north or south edge of the grid} \\ \neg(\varphi = C) \wedge \bigvee_{p \in P_{WE}} (\varphi = p) \\ \qquad \text{if } \varphi \text{ is not supposed to contain a cell and is on the west or east edge of the grid} \\ \neg(\varphi = C) \wedge \bigvee_{p \in P_{mid}} (\varphi = p) \\ \qquad \text{if } \varphi \text{ is not supposed to contain a cell and is not on the edges of the grid} \end{cases}$$

Here, $P_{NS} = \{e, —, ═\}$, $P_{WE} = \{e, \mid, \|\}$ and $P_{mid} = \{e, —, ═, \mid, \|\}$.

6.3 Generating puzzles

In this set of experiments we generate 100 puzzle IDs with the graph encoding and 100 puzzle IDs with the grid encoding for every size (7x7, 15x15, 30x30 and 50x50). For each size, we calculated the average number of cells in a game of that size that's generated by S&T's tool (the difficulty doesn't seem to matter there). For 7x7 this turns out to be 11 cells, for 15x15 it's 36 cells, 30x30 gets 91 cells, and 50x50 gets 151. While, as mentioned before, this ratio varies a lot in puzzles from S&T's tool, we generate puzzles with these number of cells to keep the ratio of size:cells as much in line with with the ratio in puzzles generated via S&T's tool as possible. In tables A.5, A.6 and A.7 in Appendix A you can see the results of the experiments we ran.

First of all, it is important to note that we created quite a successful algorithm. In all generations we made, not a single puzzle failed to generate because unique solvability could not be achieved anymore in a certain iteration. Table A.7 tells us that for any game size it never took more than 5 consecutive iterations of the loop in the algorithm to place a new cell without breaking unique solvability. Moreover, the means also tell us that on average it only takes roughly 1 extra iteration to find a new cell that doesn't break unique solvability with respect to the original try.

Secondly, the algorithm does sometimes fail in achieving the desired number of cells. We did not include the number of times this happens in the tables in Appendix A, but the numbers are as follows: For generation of 7x7 puzzles this hap-

pens in about 33% of all generation attempts, for 15x15 puzzles in about 21% of the generation attempts, and for 30x30 and 50x50 puzzles it never happened. We do however have measurements that hint towards this trend visible in table A.6. The mean, standard deviation and maximum of the number of tries it took to place a new cell in one iteration are much higher in the generation of 15x15 puzzles than in the other puzzles. A possible explanation for this is that for smaller puzzles, there is only limited space to place new cells and bridges. If the algorithm by chance starts by placing a lot of long bridges with respect to the field size, this already removes a lot of the potential to successfully find a new field to place a valid cell in that still meets all the conditions. In larger puzzles this is less of a problem, because it would require lots of consecutive placements of long bridges to lessen the potential of finding a valid placement in the next iteration. Because the algorithm is largely based on chance, this statistically does not happen as often. Another explanation can be that the number of desired cells that we feed our algorithm are based on the ratios from S&T's tool. Because the generation algorithm of S&T's tool presumably does not rely on chance, it could be that these ratios do not work as well on smaller puzzles with the generation method we came up with. In Chapter 8: Future work we will address these issues again and come up with potential fixes.

Now to look at the generation times. In general, again the graph encoding is much faster compared to the grid encoding when it comes to generating. While we don't see the factor of 8 in the 50x50 puzzles like we found while solving them (it is now around 4, according to table A.5), the standard deviation is extremely high, indicating that this factor should probably not be taken as accurate. We can conclude however that the grid encoding is still slower. We roughly see the same trend we found in solving the puzzles, which we can see in the boxplots in figure 6.2. This is to be expected, as this slowness builds upon the slowness that we found in the previous section when it comes to solving one puzzle. As we are essentially solving the same puzzle over and over again but with an extra cell and some bridges added, the time difference becomes extra visible over the iterations.

In figure A.8, figure A.9, figure A.10 and figure 6.3, for both the graph and the grid encodings we plotted the time it takes for the SMT solver to construct and try to satisfy all constraints per loop iteration. This includes the constraint described in section 5.1: Unique solvability, which must result in an unsatisfiable outcome (UNSAT) for a loop iteration to be successful in the algorithm. We call a loop iteration successful if the algorithm does not fail to place a new cell without breaking unique solvability. The time it takes the SMT solver to construct and try to satisfy the constraints in one loop iteration is representable for the total time it takes to successfully complete one loop iteration, because the execution times of the algorithm *without* the part that uses the SMT solver are outweighed by the time it takes to construct and try to satisfy the constraints regardless of the puzzle size. The plot for 50x50 puzzles is included in this section for the best representation. The plots for 7x7, 15x15 and 30x30 puzzles contain the same trend, but can be found

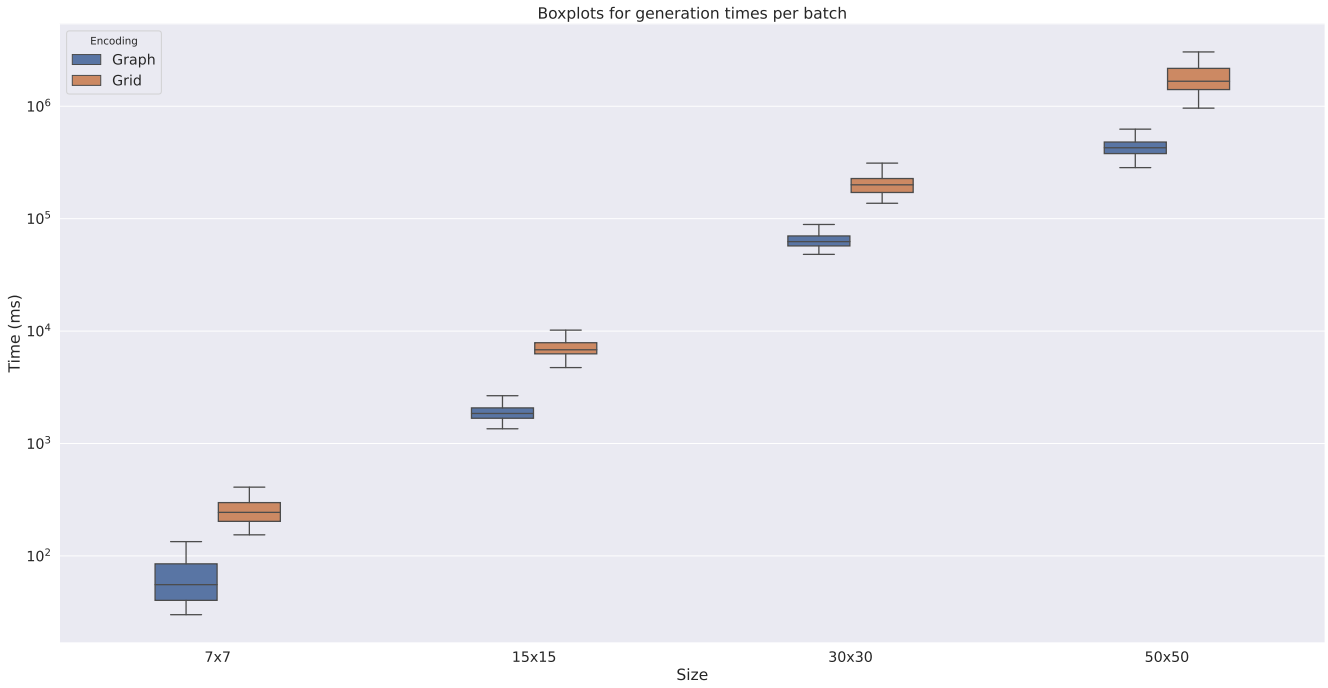


Figure 6.2: Boxplots on generation times per puzzle category and encoding

in Appendix A. It is very easy to see in these plots, especially in the plots for larger game sizes, that the time it takes to verify that a puzzle is still uniquely solvable in the grid encoding consistently grows exponentially, while in the graph encoding the time begins to grow quicker, but later converges to an exponential growth in time as well. This is again to be explained by the phenomenon where the graph encoding performs well especially on smaller sized puzzles because of its dependency on the number of bridge variables, where the grid encoding is constantly dependent on the number of field variables.

There is one final interesting thing to mention when considering the generation times of our algorithm. Note that when solving puzzles we are generally looking to satisfy (SAT) boolean constraints, while in generating uniquely solvable puzzles we are looking for UNSAT based on the constraint described in section 5.1. This raises the question if SAT and UNSAT roughly take the same time to decide, because if not, it is a relevant factor to take into account when comparing solving times to generation times. In table 6.1 we noted down measuring we did on SAT and UNSAT times. We find that indeed UNSAT takes longer to decide than SAT in almost every case. For the graph encoding we see a growth from around 1.25 for

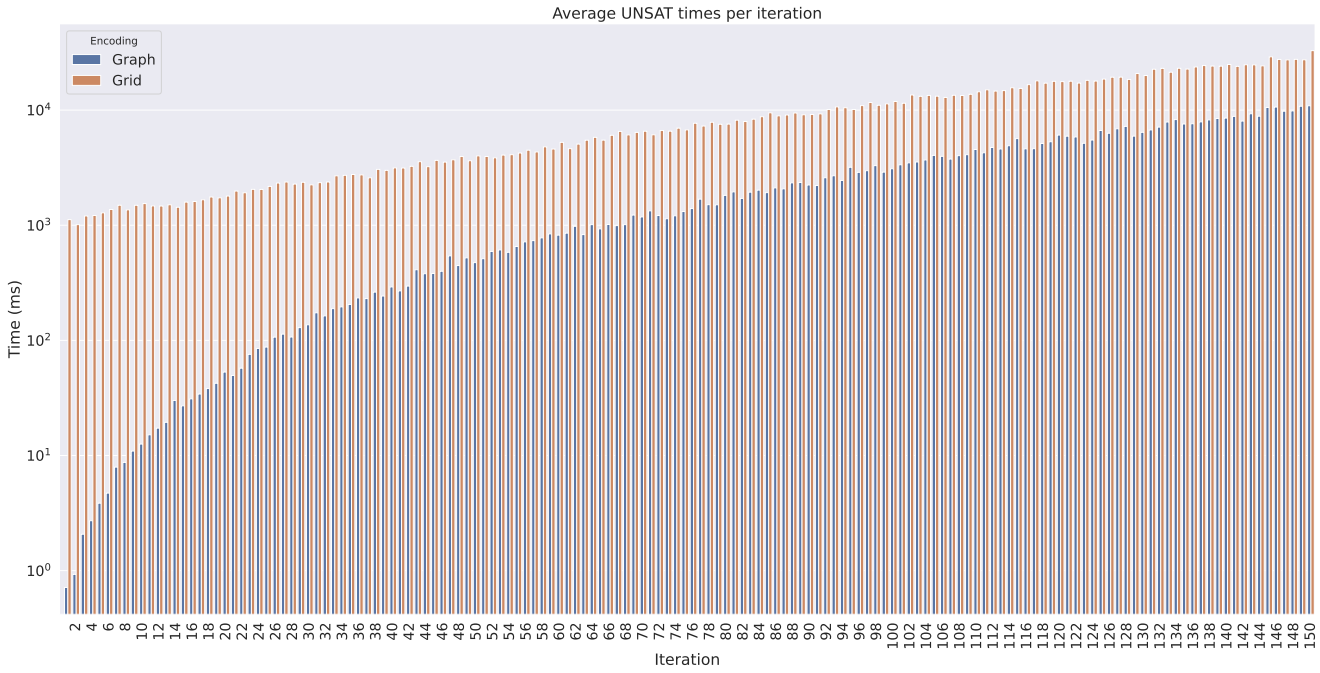


Figure 6.3: Barplot on UNSAT times for verifying unique solvability per iteration while generating 50x50 puzzles

small puzzles to a factor of around 2 for the large puzzles when we compare SAT to UNSAT times, while for the grid encoding the factor stays somewhat constant around 1,25. We can conclude that UNSAT times are indeed a factor one wants to take into account when comparing the generation times to the solving times for both the graph and the grid encoding.

6.4 Human solvability

Where we before had our SMT solver solve both the puzzles from S&T's tool and the puzzles we generated ourselves, in this section we focus on having the puzzles get solved by some human volunteers. Though it was hard to find volunteers that were willing to spend their time on solving puzzles for this study, we can still spot trends and make some conclusions on the difference between puzzles generated from S&T's tool and puzzles generated by our algorithm.

The experiment was set up as follows: We provided five test subjects with the rules of the game and an example game with its solution. We also provided them with

Size	Encoding	Mean SAT (ms)	Std. dev. SAT (ms)	Mean UNSAT (ms)	Std. dev. UNSAT (ms)
7x7	Graph	12.1	6.42	15.54	9.48
7x7	Grid	40.15	19.87	33.96	11.55
15x15	Graph	141.82	45.91	185.18	70.63
15x15	Grid	411.83	176.14	515.47	151.14
30x30	Graph	1605.58	418.42	2806.52	916.47
30x30	Grid	6299.68	2228.00	6800.33	2413.29
50x50	Graph	5877.04	1142.05	11983.91	4430.05
50x50	Grid	29131.89	10510.20	36933.72	14373.90

Table 6.1: SAT vs. UNSAT mean times for puzzles per category

6 warm-up puzzles to get our test subjects familiar with the rules and nature of the game. Two of those games were easy 7x7 games generated by S&T's tool, two games were hard 7x7 games generated by S&T's tool, and the last two warm-up games were 7x7 games generated by our own algorithm. Essentially, we provided the test subjects with 30 15x15 games, of which 10 were easy games from S&T's tool, 10 were hard games S&T's tool, and 10 were generated by our own algorithm. The test subjects did not know what puzzles were generated by what tool or what their difficulty were at the time of solving them. All puzzles were also scrambled up, but for everyone in the same order. We had the test subjects measure their solving times in seconds and note them down on an answer sheet. In table 6.2 we noted down what the average results per category were and in figure 6.4 we made a plot per category in which every individual puzzle solving time per test subject can be seen in a graph.

Difficulty	Mean (s)	Std. dev. (s)	Min (s)	Max (s)
Warmup	59.27	44.57	15	198
Easy	141.12	74.56	31	360
Hard	272.54	207.21	53	1044
Generated	313.40	252.42	87	1200

Table 6.2: Data on time it takes humans to solve generated puzzles

We can conclude a few things from these results. First of all, the puzzles from S&T's tool are confirmed to have a difference in difficulty when it comes to the difficulties the tool provides. We can see that on average the hard puzzles tend to be solved in around double the amount of time compared to the easy puzzles by each test subject. It is also to be expected that the harder the puzzle gets, the

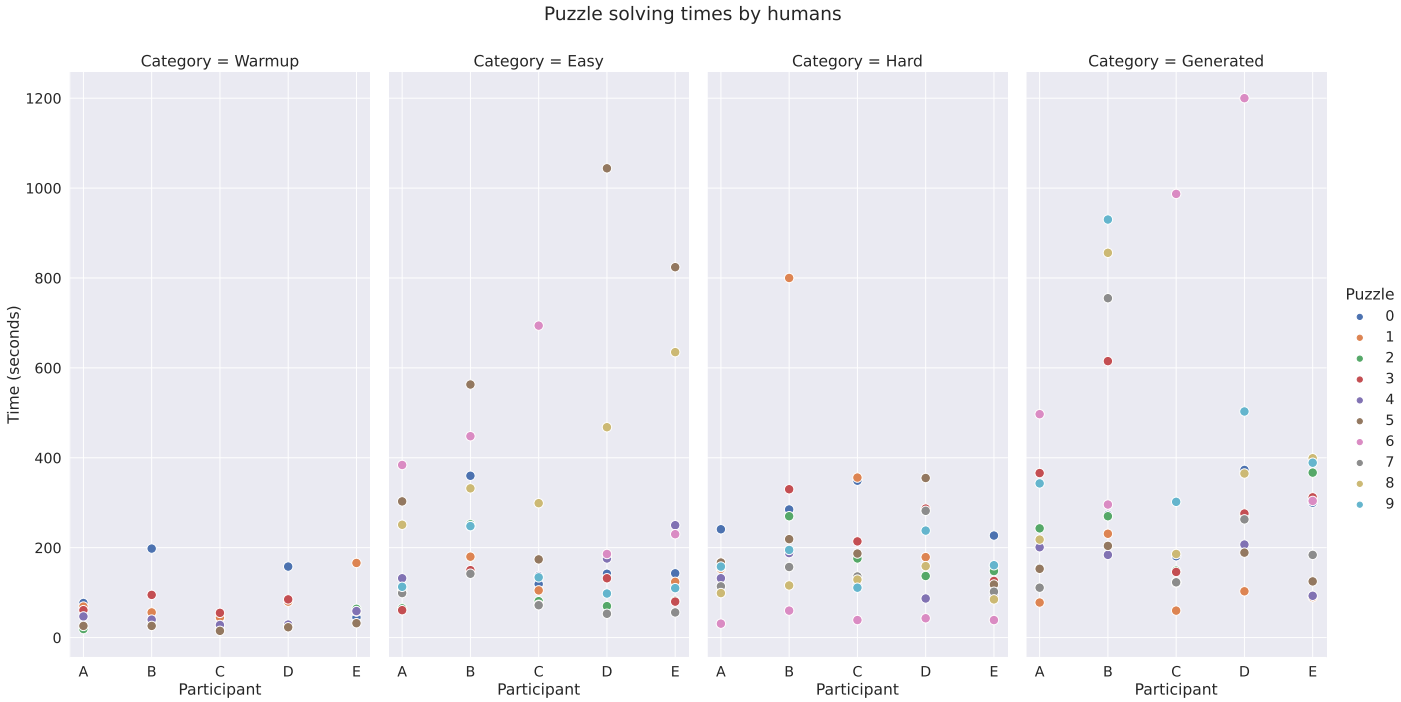


Figure 6.4: Scatterplot on times it takes humans to solve generated 15x15 puzzles

deviation in average solving times also grows, as per puzzle the chance is higher that a test subject might get stuck.

Now, if we look at the solving times of the puzzles that we generated ourselves, we see that on average they are considered the hardest of the categories. For all test subjects the mean, minimum and maximum solving times are higher than those of the other categories, with very high outliers up to 15 to 20 minutes per puzzle. The deviation between solving times of the puzzles generated by us can easily be explained: our generation algorithm did not take into account the human solvability factor. Test subjects reported that it occasionally happened that they got stuck on a generated puzzle, because they felt like they needed to take a gamble on what next bridge to place rather than reasoning about if they can place a bridge with certainty or not. This happened for some hard puzzles from S&T's tool as well, but in much lower frequency. In reality, the human solvability factor here is in the number of steps one has to reason forward in the puzzle. For example, there was one puzzle (puzzle 6 in the category 'Generated' in figure 6.4) we solved ourselves that was verified to be uniquely solvable, but it took an assumption and 20 bridge placements further until we encountered a contradiction, which proved the assumption

to be false. On average, a hard puzzle from S&T's tool would maximally take one assumption and 5 bridge placements to encounter a contradiction. We have the theory that the generation algorithm from S&T's tool counts the number of steps it takes to encounter a contradiction in order to determine the puzzle's difficulty. When we fed the puzzle that took 20 steps to encounter a contradiction to S&T's tool and pressed 'solve', the tool returned an error saying "Game does not have a (non-recursive) solution.". We did however verify that it has a unique solution, so clearly the algorithm behind S&T's tool is built using a reasoning based on human solvability, marking the puzzle as unsolvable. The missing factor of human solvability in some of the generated puzzles is directly visible in the results of the experiments.

In conclusion, the puzzles that we generated ourselves are definitely playable and are mostly comparable to the hard puzzles generated by S&T's tool. There are puzzles however, that are significantly harder because by chance they require much more reasoning steps to be able to make a move that is guaranteed to be correct. In Chapter 8: Future work I will address this issue and make a proposal on how to include the human solvability factor in a generation algorithm.

Chapter 7

Conclusions

In this study, we came up with two different SMT encodings for the logical puzzle game ”Bridges”. One in which the puzzle is encoded as a graph problem with the variables representing the possible bridges to be built, and one where the entire field is encoded as a grid where the variables represent the pieces that are placed on the grid. We used the encodings to both solve puzzles and to generate puzzles.

It turns out that the graph encoding tends to be more efficient when it comes to both solving and generating puzzles compared to the grid encoding. This was to be expected, as the number of formulas and variables to be solved by the SMT solver in the grid encoding is much higher than in the graph encoding.

For solving puzzles, we've seen that the both SMT encodings' solving times grow with at most a quadratic factor with respect to the puzzle sizes. This holds for the puzzles from Simon Tatham's Portable Puzzle Collection as well as the puzzles that we generated on our own. Still, the graph encoding has less of a growth factor than the grid encoding because there are multiple constraints in that encoding that don't rely directly on the puzzle size, but rather on the number of bridge variables. Due to the nature of the grid encoding, the constraints largely do rely directly on the puzzle size.

When it comes to generating uniquely solvable puzzles, we've seen that at least the same growth factor in time as we find in solving puzzles is present, but because the generation algorithm relies on searching for unsatisfiability of a puzzle configuration to ensure there is only one solution to be found, the generation times grow by an even bigger factor. As each iteration in the algorithm searches for unsatisfiability, the total time it takes to generate a puzzle can explode pretty quickly as the size grows. Moreover, In figure 6.3 it's clearly visible that the graph encoding scores better on smaller sized puzzles in generations, as it has a higher growth factor in generation times when it comes to smaller puzzles. We also found that the puzzles we generate do not take into account the factor of human solvability, which leads to

the phenomenon that some puzzles generated can be very hard to solve by applying human logic.

Last mentioned is also reflected in the research we did with test subjects solving the puzzles we generate. Although the pool of test subjects was relatively small, we can still see the trend that the generated puzzles took the test subjects longer to solve than the puzzles from Simon Tatham's Portable Puzzle Collection. Solving times also deviated more because human solvability was not taken into account.

To conclude, Satisfiability Modulo Theory solvers form a very viable toolkit to solve and generate logic puzzles like Bridges. Translating the problem to an SMT problem can be pretty straightforward if one can formalize the rules of the game using the mathematical tactics the SMT solver provides. One must take into account though that when generating puzzles, human solvability is a factor to implement if they search to generate puzzles to publish to an audience.

Chapter 8

Future work

There are a few topic in this research that leave questions or open up possibilities for future work. To start off:

Specifically for this puzzle, it might be interesting to come up with another encoding for the problem that gives more insight on how much influence certain choices within an encoding have. In this study, two very different encodings were used that both contain their own specific constructs in the constraints that it uses. For instance, the grid encoding uses the ite (if-then-else) programming construct as part of a mathematical formula. We suspect the ite construct to be a time-consuming thing for an SMT solver to solve. Coming up with an encoding that differs only little from either the graph encoding or the grid encoding can give more insight on the efficiency of the individual encodings and on the consequences in practical situations of the tactics that SMT solvers use.

Secondly, for this research we only used the SMT solver 'SMT-Interpol' using the Java wrapper library 'JavaSMT'. [8] More research can be done on the effectiveness of this SMT solver or the wrapper to gain insight in their efficiencies. The same encodings can be used to compare timing measurements.

For the generation algorithm there are a number of improvements that can still be made. To name two: Large parts of the algorithm are based upon randomness. Instead of relying on choosing a random source cell in the existing network to work from each time, we could improve the algorithm by keeping track of what existing cells have already been tried to set as source cell, and didn't turn out to work. Another improvement could be made on the way that generating loops in the puzzle works. Currently, after placing a new cell, it scans every direction for cells that the new cell can connect to from its position. Instead, when looking for a field to place a new cell from a source cell, if it turns out no new cell can be placed in a direction, but there is the possibility to build a bridge in that direction, we can try to build the bridge instead of giving up and reiterating to find a new field to place a

new cell. This could improve the way loops are made and perhaps result in puzzles even more similar to those found on Simon Tatham's Portable Puzzle Collection.

Another thing about the generation algorithm in this research is that it doesn't take into account human solvability. In future research, one could look into the possibility of involving that factor in the generation. One way to do this could be to come up with a solving algorithm of the puzzle that uses human-like tactics instead of SMT solving. If a certain threshold of logical thinking steps is reached when evaluating the current puzzle configuration on unique/human solvability, the newly added cell could be removed again. This would lead to generation of puzzles that are more likely to be solvable by humans, and thus more realistic to play.

Lastly, in Chapter 6: Experiments a remark was made that in a previous grid encoding, a small factor of including an explicit statement in a constraint (that should logically be implied by other statements) resulted in huge differences in solving time until another design choice was made for the grid encoding, that at first sight does not seem related. This raises the discussion of how small changes in encodings in general can lead to surprising results in efficiency. More research could be done on if one can come up with a general theory on how these differences can be made insightful to improve future logical encodings. It also raises the question if this is perhaps something specific for SMT-Interpol. It could be interesting to test this with other solvers and investigate how they deal with these small changes between encodings.

Bibliography

[1] Daniel Andersson. Hashiwokakero is np-complete. *Information Processing Letters*, 109(19):1145–1146, 2009.

[2] Tomás Balyo, Marijn Heule, and Matti Jarvisalo. Sat competition 2016: Recent developments. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[3] Clark Barrett and Cesare Tinelli. *Satisfiability modulo theories*. Springer, 2018.

[4] Magnus Björk. Successful sat encoding techniques. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(4):189–201, 2011.

[5] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. Smtinterpol: An interpolating smt solver. In *International SPIN Workshop on Model Checking of Software*, pages 248–254. Springer, 2012.

[6] Thijs de Jong, CLM Kop, and JSL Junges. Mosaic as a sat problem. 2023.

[7] Oded Goldreich. *P, NP, and NP-Completeness: The basics of computational complexity*. Cambridge University Press, 2010.

[8] Egor George Karpenkov, Karlheinz Friedberger, and Dirk Beyer. Javasmt: A unified interface for smt solvers in java. In *Verified Software. Theories, Tools, and Experiments: 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17–18, 2016, Revised Selected Papers 8*, pages 139–148. Springer, 2016.

[9] Inês Lynce and Joël Ouaknine. Sudoku as a sat problem. In *AI&M*, 2006.

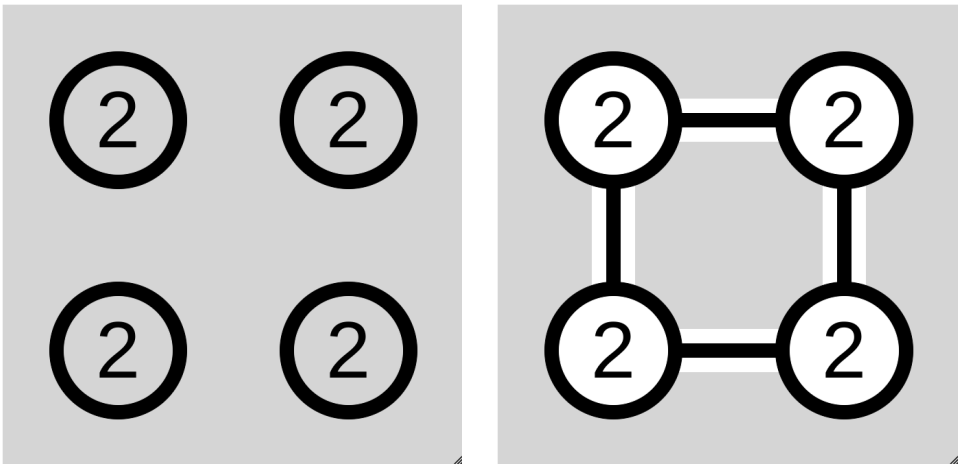
[10] Simon Tatham. Portable puzzle collection. *available from (accessed 2011-07-28)*.

Appendix A

Appendix

A.1 Constraint examples Chapters 3 and 4

These sections contain running examples of the constraints that will be built for the puzzle in figure A.1 following the definitions in Chapters 3 and 4. Though the example puzzle is very elementary and can result in trivial or redundant constraints in some cases, the examples still give insight in the behaviour of the constraints.



(a) (b)

Figure A.1: Two configurations of a very basic Bridges puzzle. Figure A.1a represents an unsolved puzzle, while figure A.1b represents its solution. Recall that we number the cells in the game from top left to bottom right, starting from 0.

A.1.1 Graph encoding

Constraint 3: Bridges are either single or double

$$\begin{aligned} & (0 \leq \beta_{0,1} \leq 2) \\ \wedge \quad & (0 \leq \beta_{0,2} \leq 2) \\ \wedge \quad & (0 \leq \beta_{1,3} \leq 2) \\ \wedge \quad & (0 \leq \beta_{2,3} \leq 2) \end{aligned}$$

Constraint 4: Bridges may not be obstructed

In this example puzzle, $bridgesCross(\beta_{0,1}, \beta_{0,2})$, $bridgesCross(\beta_{0,1}, \beta_{1,3})$, $bridgesCross(\beta_{2,3}, \beta_{0,2})$ and $bridgesCross(\beta_{2,3}, \beta_{1,3})$ all evaluate to $false$. Because of the negation of the function in the constraint we therefore get:

$$\begin{aligned} & (\beta_{0,1} > 0 \wedge \beta_{0,2} > 0 \implies true) \\ \wedge \quad & (\beta_{0,1} > 0 \wedge \beta_{1,3} > 0 \implies true) \\ \wedge \quad & (\beta_{2,3} > 0 \wedge \beta_{0,2} > 0 \implies true) \\ \wedge \quad & (\beta_{2,3} > 0 \wedge \beta_{1,3} > 0 \implies true) \end{aligned}$$

To limit the size of the full constraints to a comprehensible format, we chose to keep the example puzzle as simple as possible. In case there would actually be possible crossing bridges present in the puzzle, they would have the Boolean value $false$ on the right side of the corresponding implication, making the implication (and thus the entire conjunction) evaluate to $false$ in case both involved bridge variables have a value higher than 0.

Constraint 5: Cell values are satisfied by bridge endpoints

$$\begin{aligned} & (2 = \beta_{0,1} + \beta_{0,2} + 0 + 0) \\ \wedge \quad & (2 = \beta_{0,1} + 0 + \beta_{1,3} + 0) \\ \wedge \quad & (2 = 0 + \beta_{0,2} + 0 + \beta_{2,3}) \\ \wedge \quad & (2 = 0 + 0 + \beta_{1,3} + \beta_{2,3}) \end{aligned}$$

Constraint 6: Every cell is reachable from every other cell

$$
\begin{aligned}
& \gamma_{0,1} \\
\wedge \quad & \gamma_{0,2} \\
\wedge \quad & \gamma_{0,3} \\
\wedge \quad & (\gamma_{1,1} \iff \beta_{0,1} > 0) \\
\wedge \quad & \Big(\gamma_{1,2} \iff \gamma_{1,1} \vee \big((\gamma_{0,1} \wedge \beta_{0,1} > 0) \vee (\gamma_{3,1} \wedge \beta_{1,3} > 0)\big)\Big) \\
\wedge \quad & \Big(\gamma_{1,3} \iff \gamma_{1,2} \vee \big((\gamma_{0,2} \wedge \beta_{0,1} > 0) \vee (\gamma_{3,2} \wedge \beta_{1,3} > 0)\big)\Big) \\
\wedge \quad & \gamma_{1,3} \\
\wedge \quad & (\gamma_{2,1} \iff \beta_{0,2} > 0) \\
\wedge \quad & \Big(\gamma_{2,2} \iff \gamma_{2,1} \vee \big((\gamma_{0,1} \wedge \beta_{0,2} > 0) \vee (\gamma_{3,1} \wedge \beta_{2,3} > 0)\big)\Big) \\
\wedge \quad & \Big(\gamma_{2,3} \iff \gamma_{2,2} \vee \big((\gamma_{0,2} \wedge \beta_{0,2} > 0) \vee (\gamma_{3,2} \wedge \beta_{2,3} > 0)\big)\Big) \\
\wedge \quad & \gamma_{2,3} \\
\wedge \quad & \neg\gamma_{3,1} \\
\wedge \quad & \Big(\gamma_{3,2} \iff \gamma_{3,1} \vee \big((\gamma_{1,1} \wedge \beta_{1,3} > 0) \vee (\gamma_{2,1} \wedge \beta_{2,3} > 0)\big)\Big) \\
\wedge \quad & \Big(\gamma_{3,3} \iff \gamma_{3,2} \vee \big((\gamma_{1,2} \wedge \beta_{1,3} > 0) \vee (\gamma_{2,2} \wedge \beta_{2,3} > 0)\big)\Big) \\
\wedge \quad & \gamma_{3,3}
\end{aligned}
$$

A.1.2 Grid encoding

In this example, we use a coordinate system for the field variables: $\varphi_{x,y}$ denotes the field variable belonging to the field on row x, column y in the game *including* the extra border around the game. $\varphi_{0,0}$ denotes the top left field and $\varphi_{4,4}$ denotes the bottom right field.

Valid field piece constraint

$$
\begin{aligned}
& (\varphi_{0,0} = e) \\
\wedge\ & (\varphi_{0,1} = e) \\
\wedge\ & (\varphi_{0,2} = e) \\
\wedge\ & (\varphi_{0,3} = e) \\
\wedge\ & (\varphi_{0,4} = e) \\
\wedge\ & (\varphi_{1,0} = e) \\
\wedge\ & (\varphi_{1,1} = C) \\
\wedge\ & (\varphi_{1,2} = e) \vee (\varphi_{1,2} = \text{—}) \vee (\varphi_{1,2} = \text{=}) \vee (\varphi_{1,2} = \text{|}) \vee (\varphi_{1,2} = \text{‖}) \\
\wedge\ & (\varphi_{1,3} = C) \\
\wedge\ & (\varphi_{1,4} = e) \\
\wedge\ & (\varphi_{2,0} = e) \\
\wedge\ & (\varphi_{2,1} = e) \vee (\varphi_{2,1} = \text{—}) \vee (\varphi_{2,1} = \text{=}) \vee (\varphi_{2,1} = \text{|}) \vee (\varphi_{2,1} = \text{‖}) \\
\wedge\ & (\varphi_{2,2} = e) \vee (\varphi_{2,2} = \text{—}) \vee (\varphi_{2,2} = \text{=}) \vee (\varphi_{2,2} = \text{|}) \vee (\varphi_{2,2} = \text{‖}) \\
\wedge\ & (\varphi_{2,3} = e) \vee (\varphi_{2,3} = \text{—}) \vee (\varphi_{2,3} = \text{=}) \vee (\varphi_{2,3} = \text{|}) \vee (\varphi_{2,3} = \text{‖}) \\
\wedge\ & (\varphi_{2,4} = e) \\
\wedge\ & (\varphi_{3,0} = e) \\
\wedge\ & (\varphi_{3,1} = C) \\
\wedge\ & (\varphi_{3,2} = e) \vee (\varphi_{3,2} = \text{—}) \vee (\varphi_{3,2} = \text{=}) \vee (\varphi_{3,2} = \text{|}) \vee (\varphi_{3,2} = \text{‖}) \\
\wedge\ & (\varphi_{3,3} = C) \\
\wedge\ & (\varphi_{3,4} = e) \\
\wedge\ & (\varphi_{4,0} = e) \\
\wedge\ & (\varphi_{4,1} = e) \\
\wedge\ & (\varphi_{4,2} = e) \\
\wedge\ & (\varphi_{4,3} = e) \\
\wedge\ & (\varphi_{4,4} = e)
\end{aligned}
$$

Constraints 1, 2, 3 and 4

$$
\begin{aligned}
& \Big((\varphi_{1,1} = \text{—}) \implies \big((\varphi_{1,0} = \text{—}) \vee (\varphi_{1,0} = C)\big) \wedge \big((\varphi_{1,2} = \text{—}) \vee (\varphi_{1,2} = C)\big)\Big) \\
\wedge\ & \Big((\varphi_{1,1} = \text{=}) \implies \big((\varphi_{1,0} = \text{=}) \vee (\varphi_{1,0} = C)\big) \wedge \big((\varphi_{1,2} = \text{=}) \vee (\varphi_{1,2} = C)\big)\Big) \\
\wedge\ & \Big((\varphi_{1,1} = \text{|}) \implies \big((\varphi_{0,1} = \text{|}) \vee (\varphi_{0,1} = C)\big) \wedge \big((\varphi_{2,1} = \text{|}) \vee (\varphi_{2,1} = C)\big)\Big) \\
\wedge\ & \Big((\varphi_{1,1} = \text{‖}) \implies \big((\varphi_{0,1} = \text{‖}) \vee (\varphi_{0,1} = C)\big) \wedge \big((\varphi_{2,1} = \text{‖}) \vee (\varphi_{2,1} = C)\big)\Big) \\
\wedge\ & \Big((\varphi_{1,2} = \text{—}) \implies \big((\varphi_{1,1} = \text{—}) \vee (\varphi_{1,1} = C)\big) \wedge \big((\varphi_{1,3} = \text{—}) \vee (\varphi_{1,3} = C)\big)\Big) \\
\wedge\ & \Big((\varphi_{1,2} = \text{=}) \implies \big((\varphi_{1,1} = \text{=}) \vee (\varphi_{1,1} = C)\big) \wedge \big((\varphi_{1,3} = \text{=}) \vee (\varphi_{1,3} = C)\big)\Big) \\
\wedge\ & \Big((\varphi_{1,2} = \text{|}) \implies \big((\varphi_{0,2} = \text{|}) \vee (\varphi_{0,2} = C)\big) \wedge \big((\varphi_{2,2} = \text{|}) \vee (\varphi_{2,2} = C)\big)\Big) \\
\wedge\ & \Big((\varphi_{1,2} = \text{‖}) \implies \big((\varphi_{0,2} = \text{‖}) \vee (\varphi_{0,2} = C)\big) \wedge \big((\varphi_{2,2} = \text{‖}) \vee (\varphi_{2,2} = C)\big)\Big) \\
\wedge\ & \Big((\varphi_{1,3} = \text{—}) \implies \big((\varphi_{1,2} = \text{—}) \vee (\varphi_{1,2} = C)\big) \wedge \big((\varphi_{1,4} = \text{—}) \vee (\varphi_{1,4} = C)\big)\Big) \\
\wedge\ & \Big((\varphi_{1,3} = \text{=}) \implies \big((\varphi_{1,2} = \text{=}) \vee (\varphi_{1,2} = C)\big) \wedge \big((\varphi_{1,4} = \text{=}) \vee (\varphi_{1,4} = C)\big)\Big) \\
\wedge\ & \Big((\varphi_{1,3} = \text{|}) \implies \big((\varphi_{0,3} = \text{|}) \vee (\varphi_{0,3} = C)\big) \wedge \big((\varphi_{2,3} = \text{|}) \vee (\varphi_{2,3} = C)\big)\Big) \\
\wedge\ & \Big((\varphi_{1,3} = \text{‖}) \implies \big((\varphi_{0,3} = \text{‖}) \vee (\varphi_{0,3} = C)\big) \wedge \big((\varphi_{2,3} = \text{‖}) \vee (\varphi_{2,3} = C)\big)\Big) \\
\wedge\ & \Big((\varphi_{2,1} = \text{—}) \implies \big((\varphi_{2,0} = \text{—}) \vee (\varphi_{2,0} = C)\big) \wedge \big((\varphi_{2,2} = \text{—}) \vee (\varphi_{2,2} = C)\big)\Big) \\
\wedge\ & \Big((\varphi_{2,1} = \text{=}) \implies \big((\varphi_{2,0} = \text{=}) \vee (\varphi_{2,0} = C)\big) \wedge \big((\varphi_{2,2} = \text{=}) \vee (\varphi_{2,2} = C)\big)\Big) \\
\wedge\ & \Big((\varphi_{2,1} = \text{|}) \implies \big((\varphi_{1,1} = \text{|}) \vee (\varphi_{1,1} = C)\big) \wedge \big((\varphi_{3,1} = \text{|}) \vee (\varphi_{3,1} = C)\big)\Big) \\
\wedge\ & \Big((\varphi_{2,1} = \text{‖}) \implies \big((\varphi_{1,1} = \text{‖}) \vee (\varphi_{1,1} = C)\big) \wedge \big((\varphi_{3,1} = \text{‖}) \vee (\varphi_{3,1} = C)\big)\Big) \\
\wedge\ & \Big((\varphi_{2,2} = \text{—}) \implies \big((\varphi_{2,1} = \text{—}) \vee (\varphi_{2,1} = C)\big) \wedge \big((\varphi_{2,3} = \text{—}) \vee (\varphi_{2,3} = C)\big)\Big) \\
\wedge\ & \Big((\varphi_{2,2} = \text{=}) \implies \big((\varphi_{2,1} = \text{=}) \vee (\varphi_{2,1} = C)\big) \wedge \big((\varphi_{2,3} = \text{=}) \vee (\varphi_{2,3} = C)\big)\Big) \\
\wedge\ & \Big((\varphi_{2,2} = \text{|}) \implies \big((\varphi_{1,2} = \text{|}) \vee (\varphi_{1,2} = C)\big) \wedge \big((\varphi_{3,2} = \text{|}) \vee (\varphi_{3,2} = C)\big)\Big) \\
\wedge\ & \Big((\varphi_{2,2} = \text{‖}) \implies \big((\varphi_{1,2} = \text{‖}) \vee (\varphi_{1,2} = C)\big) \wedge \big((\varphi_{3,2} = \text{‖}) \vee (\varphi_{3,2} = C)\big)\Big) \\
\wedge\ & \Big((\varphi_{2,3} = \text{—}) \implies \big((\varphi_{2,2} = \text{—}) \vee (\varphi_{2,2} = C)\big) \wedge \big((\varphi_{2,4} = \text{—}) \vee (\varphi_{2,4} = C)\big)\Big) \\
\wedge\ & \Big((\varphi_{2,3} = \text{=}) \implies \big((\varphi_{2,2} = \text{=}) \vee (\varphi_{2,2} = C)\big) \wedge \big((\varphi_{2,4} = \text{=}) \vee (\varphi_{2,4} = C)\big)\Big) \\
\wedge\ & \Big((\varphi_{2,3} = \text{|}) \implies \big((\varphi_{1,3} = \text{|}) \vee (\varphi_{1,3} = C)\big) \wedge \big((\varphi_{3,3} = \text{|}) \vee (\varphi_{3,3} = C)\big)\Big) \\
\wedge\ & \Big((\varphi_{2,3} = \text{‖}) \implies \big((\varphi_{1,3} = \text{‖}) \vee (\varphi_{1,3} = C)\big) \wedge \big((\varphi_{3,3} = \text{‖}) \vee (\varphi_{3,3} = C)\big)\Big)
\end{aligned}
$$

$$
\begin{aligned}
\wedge\quad & \Big((\varphi_{3,1} = —) \implies \big((\varphi_{3,0} = —) \vee (\varphi_{3,0} = C)\big) \wedge \big((\varphi_{3,2} = —) \vee (\varphi_{3,2} = C)\big)\Big) \\
\wedge\quad & \Big((\varphi_{3,1} = ═) \implies \big((\varphi_{3,0} = ═) \vee (\varphi_{3,0} = C)\big) \wedge \big((\varphi_{3,2} = ═) \vee (\varphi_{3,2} = C)\big)\Big) \\
\wedge\quad & \Big((\varphi_{3,1} = |) \implies \big((\varphi_{2,1} = |) \vee (\varphi_{2,1} = C)\big) \wedge \big((\varphi_{4,1} = |) \vee (\varphi_{4,1} = C)\big)\Big) \\
\wedge\quad & \Big((\varphi_{3,1} = \|) \implies \big((\varphi_{2,1} = \|) \vee (\varphi_{2,1} = C)\big) \wedge \big((\varphi_{4,1} = \|) \vee (\varphi_{4,1} = C)\big)\Big) \\
\wedge\quad & \Big((\varphi_{3,2} = —) \implies \big((\varphi_{3,1} = —) \vee (\varphi_{3,1} = C)\big) \wedge \big((\varphi_{3,3} = —) \vee (\varphi_{3,3} = C)\big)\Big) \\
\wedge\quad & \Big((\varphi_{3,2} = ═) \implies \big((\varphi_{3,1} = ═) \vee (\varphi_{3,1} = C)\big) \wedge \big((\varphi_{3,3} = ═) \vee (\varphi_{3,3} = C)\big)\Big) \\
\wedge\quad & \Big((\varphi_{3,2} = |) \implies \big((\varphi_{2,2} = |) \vee (\varphi_{2,2} = C)\big) \wedge \big((\varphi_{4,2} = |) \vee (\varphi_{4,2} = C)\big)\Big) \\
\wedge\quad & \Big((\varphi_{3,2} = \|) \implies \big((\varphi_{2,2} = \|) \vee (\varphi_{2,2} = C)\big) \wedge \big((\varphi_{4,2} = \|) \vee (\varphi_{4,2} = C)\big)\Big) \\
\wedge\quad & \Big((\varphi_{3,3} = —) \implies \big((\varphi_{3,2} = —) \vee (\varphi_{3,2} = C)\big) \wedge \big((\varphi_{3,4} = —) \vee (\varphi_{3,4} = C)\big)\Big) \\
\wedge\quad & \Big((\varphi_{3,3} = ═) \implies \big((\varphi_{3,2} = ═) \vee (\varphi_{3,2} = C)\big) \wedge \big((\varphi_{3,4} = ═) \vee (\varphi_{3,4} = C)\big)\Big) \\
\wedge\quad & \Big((\varphi_{3,3} = |) \implies \big((\varphi_{2,3} = |) \vee (\varphi_{2,3} = C)\big) \wedge \big((\varphi_{4,3} = |) \vee (\varphi_{4,3} = C)\big)\Big) \\
\wedge\quad & \Big((\varphi_{3,3} = \|) \implies \big((\varphi_{2,3} = \|) \vee (\varphi_{2,3} = C)\big) \wedge \big((\varphi_{4,3} = \|) \vee (\varphi_{4,3} = C)\big)\Big)
\end{aligned}
$$

Constraint 5: Cell values are satisfied by bridge endpoints

$$
\begin{aligned}
& 2 = \Big(ite(\varphi_{0,1} = |), 1, ite\big((\varphi_{0,1} = \|), 2, 0\big)\Big) + \Big(ite(\varphi_{1,2} = —), 1, ite\big((\varphi_{1,2} = ═), 2, 0\big)\Big) \\
& \qquad + \Big(ite(\varphi_{2,1} = |), 1, ite\big((\varphi_{2,1} = \|), 2, 0\big)\Big) + \Big(ite(\varphi_{1,0} = —), 1, ite\big((\varphi_{1,0} = ═), 2, 0\big)\Big) \\
\wedge\quad & 2 = \Big(ite(\varphi_{0,3} = |), 1, ite\big((\varphi_{0,3} = \|), 2, 0\big)\Big) + \Big(ite(\varphi_{1,4} = —), 1, ite\big((\varphi_{1,4} = ═), 2, 0\big)\Big) \\
& \qquad + \Big(ite(\varphi_{2,3} = |), 1, ite\big((\varphi_{2,3} = \|), 2, 0\big)\Big) + \Big(ite(\varphi_{1,2} = —), 1, ite\big((\varphi_{1,2} = ═), 2, 0\big)\Big) \\
\wedge\quad & 2 = \Big(ite(\varphi_{2,1} = |), 1, ite\big((\varphi_{2,1} = \|), 2, 0\big)\Big) + \Big(ite(\varphi_{3,2} = —), 1, ite\big((\varphi_{3,2} = ═), 2, 0\big)\Big) \\
& \qquad + \Big(ite(\varphi_{4,1} = |), 1, ite\big((\varphi_{4,1} = \|), 2, 0\big)\Big) + \Big(ite(\varphi_{3,0} = —), 1, ite\big((\varphi_{3,0} = ═), 2, 0\big)\Big) \\
\wedge\quad & 2 = \Big(ite(\varphi_{2,3} = |), 1, ite\big((\varphi_{2,3} = \|), 2, 0\big)\Big) + \Big(ite(\varphi_{3,4} = —), 1, ite\big((\varphi_{3,4} = ═), 2, 0\big)\Big) \\
& \qquad + \Big(ite(\varphi_{4,3} = |), 1, ite\big((\varphi_{4,3} = \|), 2, 0\big)\Big) + \Big(ite(\varphi_{3,2} = —), 1, ite\big((\varphi_{3,2} = ═), 2, 0\big)\Big)
\end{aligned}
$$

Constraint 6: Every cell is reachable from every other cell

$$
\begin{aligned}
& \gamma_{0,1} \\
\wedge\ & \gamma_{0,2} \\
\wedge\ & \gamma_{0,3} \\
\wedge\ & \big(\gamma_{1,1} \iff (\varphi_{1,2} = \text{—}) \vee (\varphi_{1,2} = \text{=})\big) \\
\wedge\ & \Big(\gamma_{1,2} \iff \gamma_{1,1} \vee \Big(\gamma_{0,1} \wedge \big((\varphi_{1,2} = \text{—}) \vee (\varphi_{1,2} = \text{=})\big)\Big) \vee \Big(\gamma_{3,1} \wedge \big((\varphi_{2,3} = \text{|}) \vee (\varphi_{2,3} = \text{‖})\big)\Big)\Big) \\
\wedge\ & \Big(\gamma_{1,3} \iff \gamma_{1,2} \vee \Big(\gamma_{0,2} \wedge \big((\varphi_{1,2} = \text{—}) \vee (\varphi_{1,2} = \text{=})\big)\Big) \vee \Big(\gamma_{3,2} \wedge \big((\varphi_{2,3} = \text{|}) \vee (\varphi_{2,3} = \text{‖})\big)\Big)\Big) \\
\wedge\ & \gamma_{1,3} \\
\wedge\ & \big(\gamma_{2,1} \iff (\varphi_{2,1} = \text{|}) \vee (\varphi_{2,1} = \text{‖})\big) \\
\wedge\ & \Big(\gamma_{2,2} \iff \gamma_{2,1} \vee \Big(\gamma_{0,1} \wedge \big((\varphi_{2,1} = \text{|}) \vee (\varphi_{2,1} = \text{‖})\big)\Big) \vee \Big(\gamma_{3,1} \wedge \big((\varphi_{3,2} = \text{—}) \vee (\varphi_{3,2} = \text{=})\big)\Big)\Big) \\
\wedge\ & \Big(\gamma_{2,3} \iff \gamma_{2,2} \vee \Big(\gamma_{0,2} \wedge \big((\varphi_{2,1} = \text{|}) \vee (\varphi_{2,1} = \text{‖})\big)\Big) \vee \Big(\gamma_{3,2} \wedge \big((\varphi_{3,2} = \text{—}) \vee (\varphi_{3,2} = \text{=})\big)\Big)\Big) \\
\wedge\ & \gamma_{2,3} \\
\wedge\ & \neg\gamma_{3,1} \\
\wedge\ & \Big(\gamma_{3,2} \iff \gamma_{3,1} \vee \Big(\gamma_{1,1} \wedge \big((\varphi_{2,3} = \text{|}) \vee (\varphi_{2,3} = \text{‖})\big)\Big) \vee \Big(\gamma_{2,1} \wedge \big((\varphi_{3,2} = \text{—}) \vee (\varphi_{3,2} = \text{=})\big)\Big)\Big) \\
\wedge\ & \Big(\gamma_{3,3} \iff \gamma_{3,2} \vee \Big(\gamma_{1,2} \wedge \big((\varphi_{2,3} = \text{|}) \vee (\varphi_{2,3} = \text{‖})\big)\Big) \vee \Big(\gamma_{2,2} \wedge \big((\varphi_{3,2} = \text{—}) \vee (\varphi_{3,2} = \text{=})\big)\Big)\Big) \\
\wedge\ & \gamma_{3,3}
\end{aligned}
$$

A.2 Tables and graphs Chapter 6

Size	Difficulty	Encoding	Mean (ms)	Std. dev. (ms)	Min (ms)	Max (ms)
7x7	Easy	Graph	15.69	11.38	5	65
7x7	Easy	Grid	48.25	24.59	22	180
7x7	Hard	Graph	12.9	7.73	4	44
7x7	Hard	Grid	38.0	18.28	16	113
7x7	Generated	Graph	12.1	6.42	3	42
7x7	Generated	Grid	40.15	19.87	17	113
15x15	Easy	Graph	126.41	85.55	18	470
15x15	Easy	Grid	336.66	178.55	70	1044
15x15	Hard	Graph	129.43	69.73	22	474
15x15	Hard	Grid	402.53	218.07	115	1194
15x15	Generated	Graph	141.82	45.95	46	268
15x15	Generated	Grid	411.83	176.14	142	1098
30x30	Easy	Graph	1275.23	846.89	38	3341
30x30	Easy	Grid	4185.28	2594.07	404	12916
30x30	Hard	Graph	1744.05	1104.84	198	7803
30x30	Hard	Grid	7432.49	5213.42	1239	33253
30x30	Generated	Graph	1605.58	418.42	929	3085
30x30	Generated	Grid	6299.68	2229.00	3530	14183
50x50	Easy	Graph	2942.64	3360.11	11	22987
50x50	Easy	Grid	13504.99	13688.43	997	81760
50x50	Hard	Graph	6877.72	5789.78	378	29636
50x50	Hard	Grid	32589.01	30631.11	3652	233734
50x50	Generated	Graph	5877.04	1142.05	3399	9468
50x50	Generated	Grid	29131.89	10510.20	16943	71754

Table A.2: Data on total times to find a solution puzzle category

Size	Difficulty	Encoding	Mean (ms)	Std. dev. (ms)	Min (ms)	Max (ms)
7x7	Easy	Graph	12.15	9.28	3	53
7x7	Easy	Grid	33.58	16.37	17	98
7x7	Hard	Graph	9.61	6.23	3	40
7x7	Hard	Grid	24.0	11.86	11	72
7x7	Generated	Graph	9.14	4.99	3	33
7x7	Generated	Grid	28.59	15.00	13	88
15x15	Easy	Graph	82.29	57.19	12	296
15x15	Easy	Grid	131.32	56.44	50	309
15x15	Hard	Graph	80.89	46.21	17	317
15x15	Hard	Grid	124.34	49.28	52	284
15x15	Generated	Graph	93.92	37.71	32	223
15x15	Generated	Grid	134.28	42.57	62	286
30x30	Easy	Graph	672.98	455.68	28	2245
30x30	Easy	Grid	824.22	386.02	225	1686
30x30	Hard	Graph	849.12	492.14	92	2863
30x30	Hard	Grid	1044.92	695.91	267	6220
30x30	Generated	Graph	848.07	165.73	554	1444
30x30	Generated	Grid	968.59	178.60	640	1580
50x50	Easy	Graph	1394.24	1443.49	8	7349
50x50	Easy	Grid	2030.34	1186.17	622	6383
50x50	Hard	Graph	3178.99	2649.36	214	17009
50x50	Hard	Grid	3623.42	2109.40	937	10622
50x50	Generated	Graph	2742.8	530.88	1847	5049
50x50	Generated	Grid	3258.74	394.85	2379	4422

Table A.3: Data on constraint construction times per puzzle category

Size	Difficulty	Encoding	Mean (ms)	Std. dev. (ms)	Min (ms)	Max (ms)
7x7	Easy	Graph	3.54	2.48	1	15
7x7	Easy	Grid	14.67	10.64	4	82
7x7	Hard	Graph	3.29	2.50	1	16
7x7	Hard	Grid	14.0	9.26	3	62
7x7	Generated	Graph	2.96	1.85	0	9
7x7	Generated	Grid	11.56	8.2	3	53
15x15	Easy	Graph	44.12	33.47	4	174
15x15	Easy	Grid	205.34	144.93	20	830
15x15	Hard	Graph	48.54	30.92	5	189
15x15	Hard	Grid	278.19	191.55	43	1054
15x15	Generated	Graph	47.9	19.38	11	114
15x15	Generated	Grid	277.55	162.13	64	964
30x30	Easy	Graph	602.25	460.43	10	2149
30x30	Easy	Grid	3361.06	2302.62	179	11496
30x30	Hard	Graph	894.93	679.02	63	4940
30x30	Hard	Grid	6387.57	4847.25	940	31512
30x30	Generated	Graph	757.51	348.10	375	1996
30x30	Generated	Grid	5331.09	2190.03	2696	13130
50x50	Easy	Graph	1548.4	2020.43	3	15638
50x50	Easy	Grid	11474.65	12690.42	294	77979
50x50	Hard	Graph	3698.73	3371.85	164	17278
50x50	Hard	Grid	28965.59	29207.20	2643	223112
50x50	Generated	Graph	3134.24	1003.34	909	6862
50x50	Generated	Grid	25873.15	10472.85	13833	68616

Table A.4: Data on constraint solving times per puzzle category

Size:Cells	Encoding	Mean (ms)	Std. dev. (ms)	Min (ms)	Max (ms)
7x7:11	Graph	68.96	40.50	30	231
7x7:11	Grid	261.72	73.81	154	526
15x15:36	Graph	1923.6	359.86	1352	3091
15x15:36	Grid	7212.84	1452.56	4734	13747
30x30:91	Graph	66179.3	13377.07	48123	115451
30x30:91	Grid	210056.18	57940.98	137038	480715
50x50:151	Graph	442830.04	89006.00	284361	762485
50x50:151	Grid	1795369.67	503253.74	961611	3041132

Table A.5: Data on generation times per puzzle

Size:Cells	Encoding	Mean (#)	Std. dev. (#)	Min (#)	Max (#)
7x7:11	Graph	11.29	13.60	2	81
7x7:11	Grid	9.77	10.15	1	50
15x15:36	Graph	24.87	29.10	4	205
15x15:36	Grid	25.90	46.78	5	377
30x30:91	Graph	17.22	13.88	4	95
30x30:91	Grid	17.27	11.99	6	71
50x50:151	Graph	12.19	4.58	5	30
50x50:151	Grid	13.22	5.32	6	33

Table A.6: Data on tries to place new cell per iteration

Size:Cells	Encoding	Mean (#)	Std. dev. (#)	Min (#)	Max (#)
7x7:11	Graph	0.62	0.68	0	2
7x7:11	Grid	0.62	0.68	0	3
15x15:36	Graph	0.96	0.79	0	5
15x15:36	Grid	0.96	0.65	0	2
30x30:91	Graph	1.24	0.57	0	3
30x30:91	Grid	1.27	0.53	0	3
50x50:151	Graph	1.4	0.65	0	5
50x50:151	Grid	1.34	0.54	1	3

Table A.7: Data on on tries to verify unique solvability per iteration

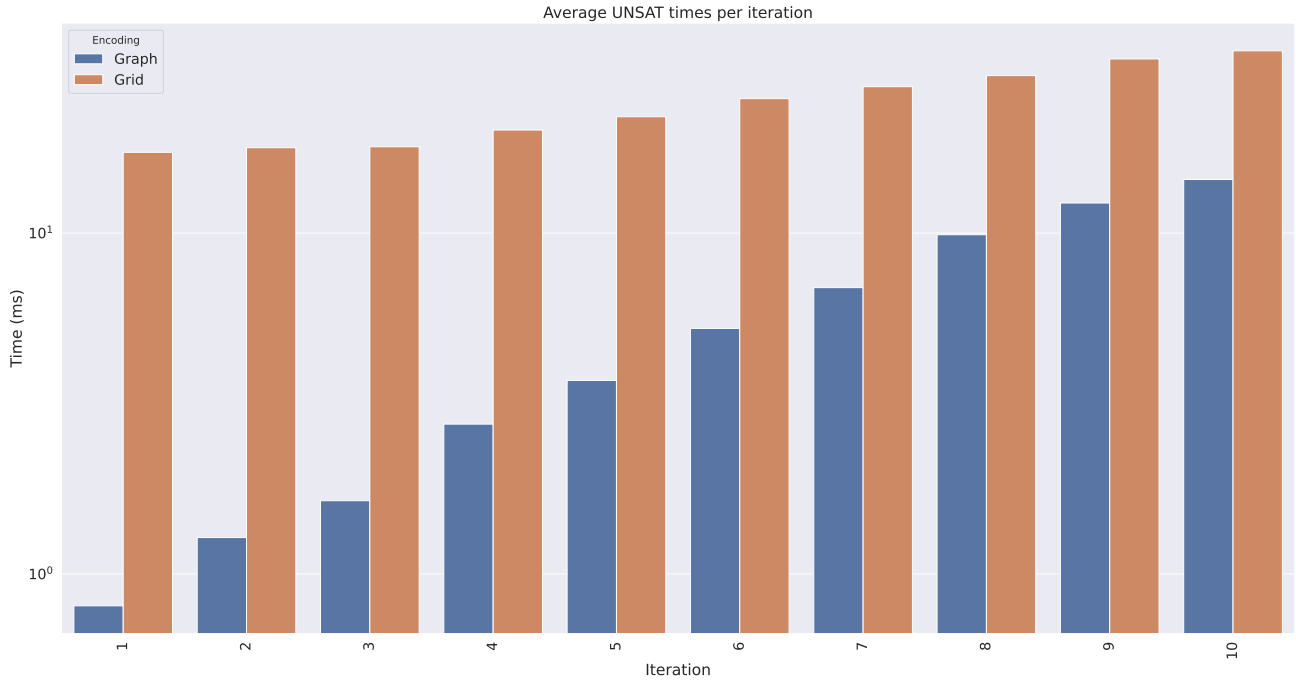


Figure A.8: Barplot on UNSAT times for verifying unique solvability per iteration while generating 7x7 puzzles

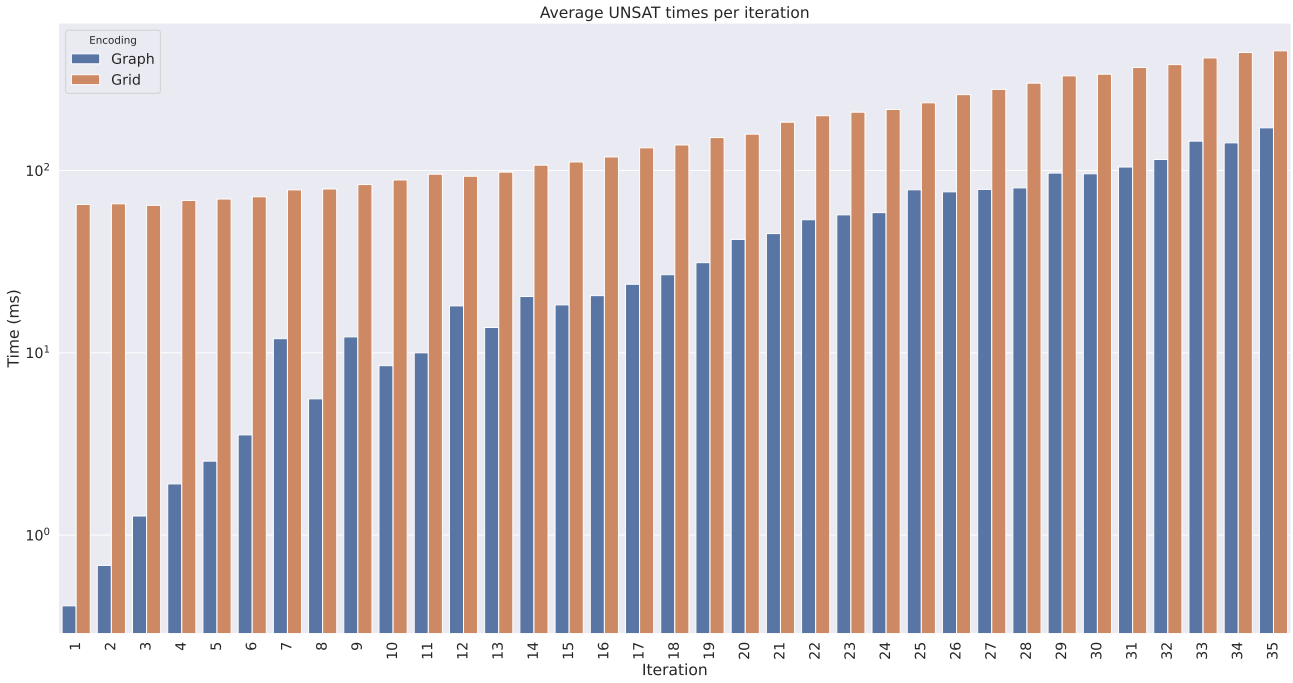


Figure A.9: Barplot on UNSAT times for verifying unique solvability per iteration while generating 15x15 puzzles

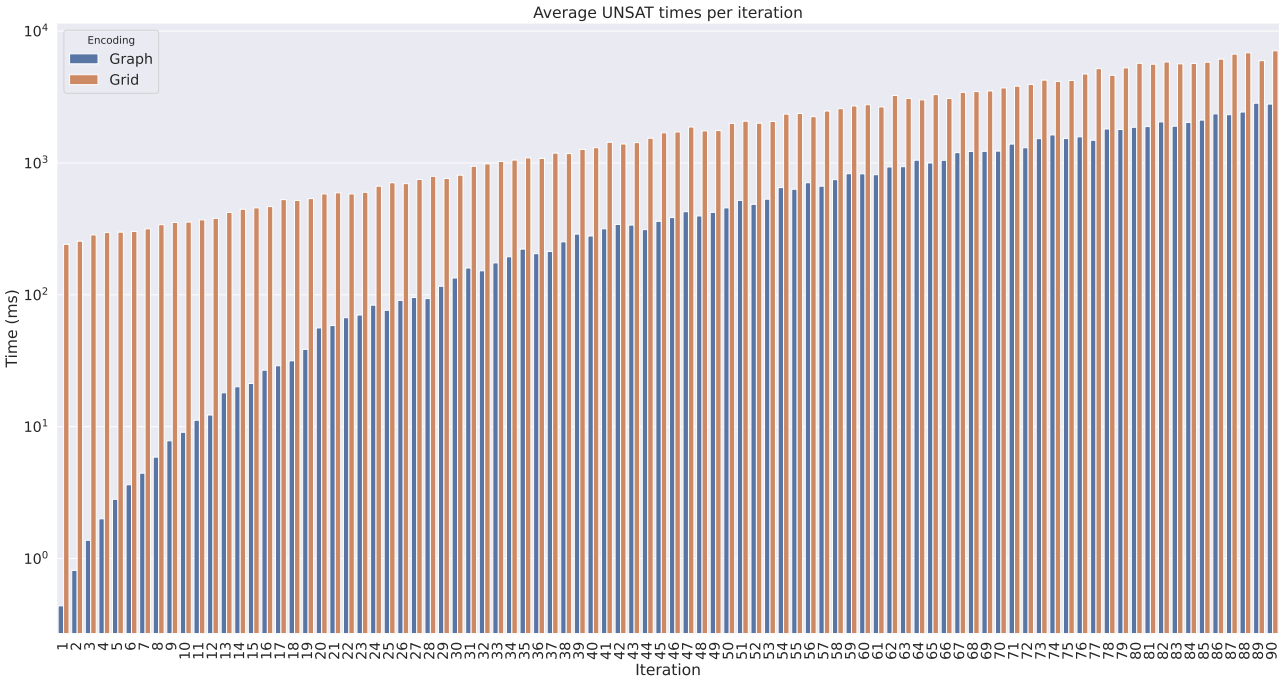


Figure A.10: Barplot on UNSAT times for verifying unique solvability per iteration while generating 30x30 puzzles