Universiteit Leiden
The Netherlands

Opleiding Informatica

How to solve and generate Numberlink puzzles using SAT

Joshua J. Lelipaly

Supervisors:
Mark van den Bergh & Rudy van Vliet

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

22/01/2025

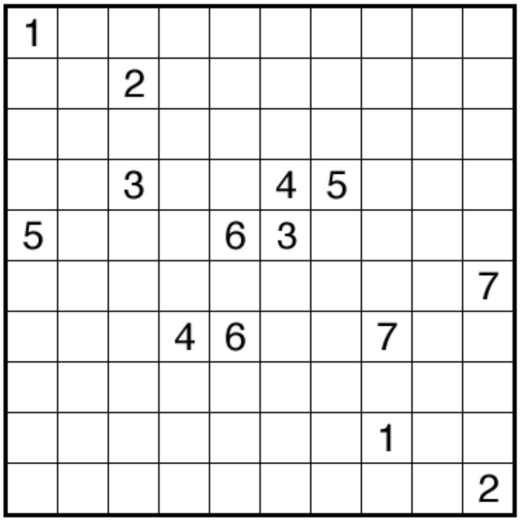
Abstract

This thesis covers the use of a SAT solver to solve any Numberlink puzzle. These puzzles consist of an $m \times n$ grid and p pairs of numbers. The goal is to connect these pairs and to fill every cell of the grid. This SAT implementation is compared to a backtracking implementation. Additionally, we discuss hexagonal cells and explain how this changes the SAT encoding. We also discuss how to generate this puzzle with the SAT implementation and how we can generate these puzzles more quickly using different heuristics.

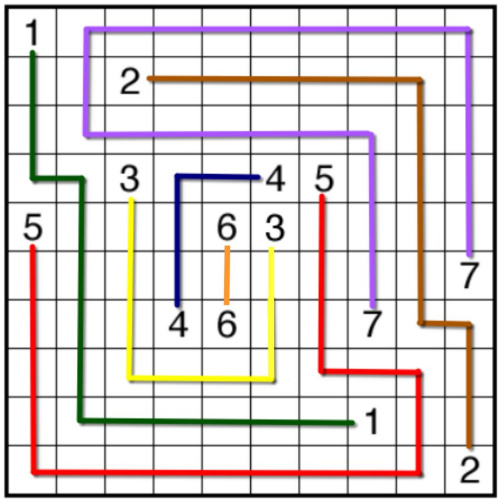
Contents

1 Introduction

Numberlink puzzles are a type of puzzle consisting of an $m \times n$ grid and $p \geq 1$ pairs of numbers $(1,1),(2,2),\ldots,(p,p)$. The pairs of numbers are scattered throughout the grid as seen in Figure 1a. The goal is to connect the numbers with their copy by paths. Each path is a sequence of cells, where every two successive cells share an edge. For the puzzle to be considered solved, every pair needs to be connected, and every cell needs to contain an edge of a path. Additionally, paths cannot cross other paths and cannot cross other numbers. An example of a solution can be seen in Figure 1b.



(a) An example of a Numberlink puzzle

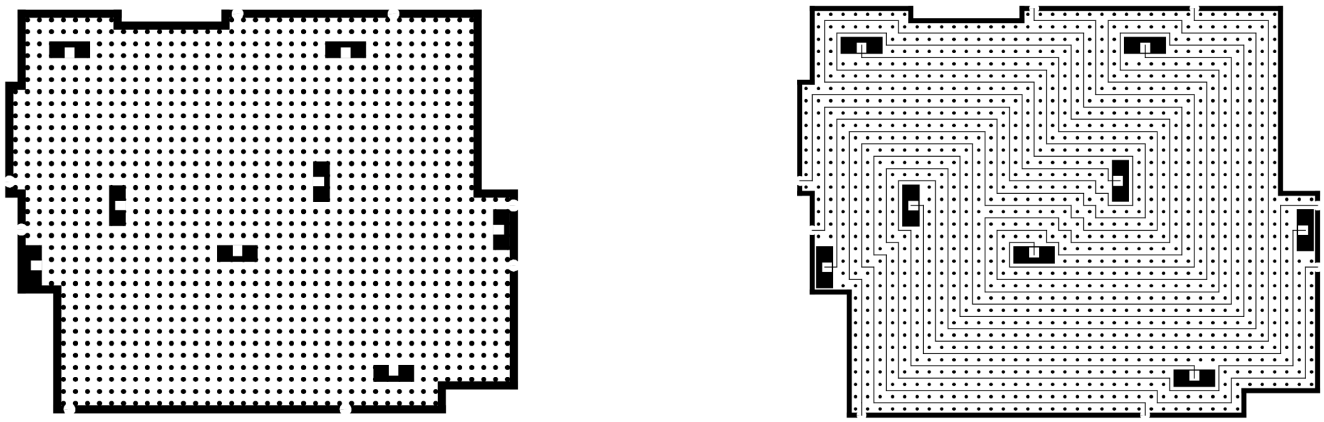


(b) The solution to the Numberlink puzzle

Figure 1: A configuration of a Numberlink puzzle

1.1 Origin

The first version of a Numberlink puzzle was published in 1897 by Sam Floyd in the Brooklyn Daily Eagle [Peg07]. This puzzle has 8 houses in Puzzleland Park and 8 private exits. This park is covered in trees an each household has to move around these trees to get to their dedicated exit. Additionally, these households do not want to see others, so paths cannot cross. An example of this puzzle can be seen in Figure 2a and its solution in Figure 2b.

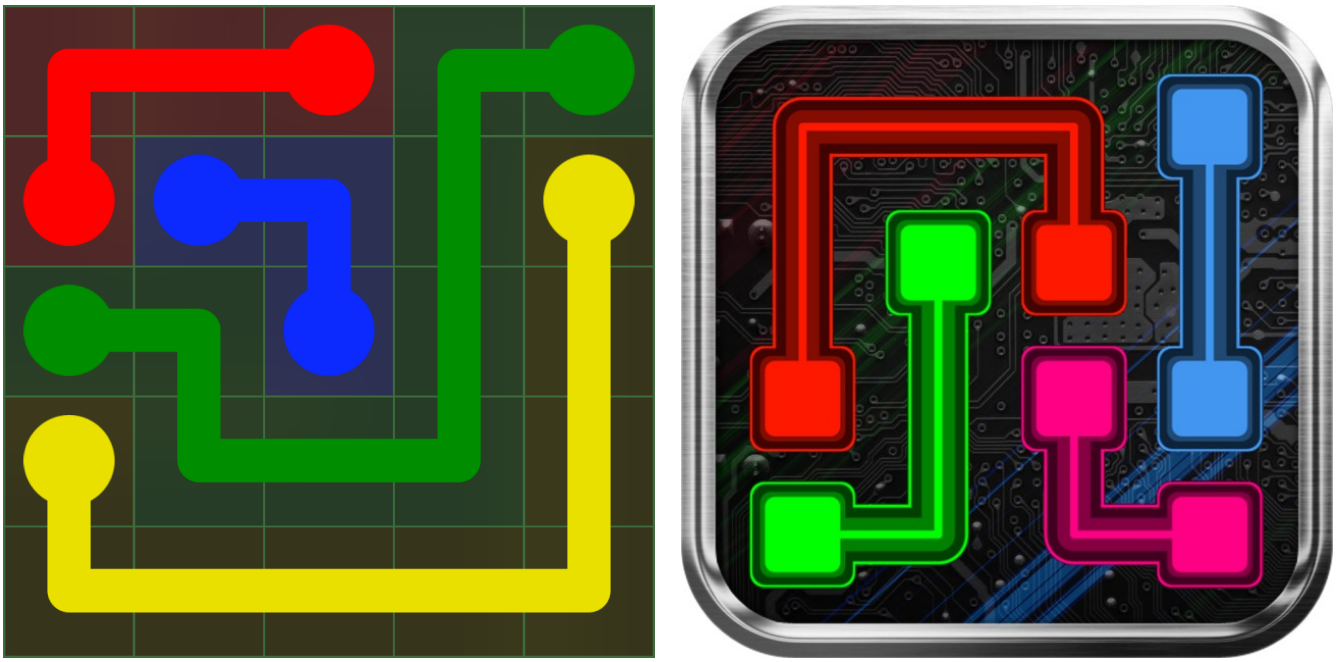


(a) The first version of a Numberlink puzzle

(b) The solved first puzzle

Figure 2: The first version of a Numberlink puzzle and its solution

This type of puzzle was later popularized by Nikoli [Nik]. The difference between this variation and the first version is as follows: the Nikoli variant is a filled grid, with a clear overview of which cell needs to be connected, while the first version resembled an asymmetric park with houses occupying multiple cells.



(a) An example of a Flow Free puzzle

(b) An example of a Wire Storm puzzle

Figure 3: Modern variations of Numberlink

Modern variations include Flow Free and Wire Storm, which are smartphone apps available in the App Store and Play Store [LLC15] [Gre13]. The only difference these variants have in comparison to Numberlink is the fact that these variants use colors instead of numbers to indicate and connect the pairs. An example of these puzzles can be seen in Figure 3.

1.2 Related Work

Numberlink falls under the logic-based combinatorial number placement puzzle category. Students from LIACS have researched many puzzles under this category. Among them, there are Kuroshuto puzzles researched by Hanna Straathof [Str25] in 2025, Fobidoshi puzzles researched by Niels Heslenfeld [Hes24] in 2024, and the comparison between different encodings for the continuity rule of the logic puzzle by Jente de Waart [dW25]. Research outside of LIACS includes the proof that Numberlink is NP-complete by Aaron Adcock et al. [ADD+15], and a physical zero-knowledge proof [RI20] by Suthee Ruangwises and Toshiya Itoh.

1.3 Overview of Thesis

In Section 2, we discuss a simple and straightforward method to solve a given Numberlink puzzle. In Section 3, we explain how to encode a given Numberlink puzzle into a propositional formula so a SAT solver can determine the solution. In Section 4, we discuss how to generate a random Numberlink puzzle and we explain the use of different heuristics to make the generation of puzzles faster. In Section 5, we discuss the changes made if the cells of the puzzle were hexagonal. We perform experiments in Section6 to determine the performance of our solving methods and heuristics. Finally, we give a summary and conclusions in Section 7 and we provide ideas for future work in Section 8.

This thesis was written as part of the bachelor project for Computer Science under the supervision of Mark van den Bergh and Rudy van Vliet, associated with the Leiden Institute of Advanced Computer Science (LIACS). The core research question is "what SAT encoding can be used to solve and generate Numberlink puzzles".

2 Backtracking

A simple approach to solving a Numberlink puzzle would be to use backtracking with depth-first-search. For this, we would need to turn the search space into a tree. We do this as follows. We begin with a cell from the first pair. From there, we go to each of the following directions, beginning with the direction UP, followed by RIGHT, then DOWN, and lastly LEFT. For each of those cells, we go through each of the directions again, except the one we came from. We keep doing this until our corresponding number copy has been found. When a direction is blocked, we move on to the next direction, and when the directions have been exhausted, we go back to the cell we came from. When our corresponding number copy has been found, we repeat the steps above, but with the second pair. We repeat the process until all pairs have been connected and all the cells are filled, or when all possibilities have been exhausted, which means that the puzzle is unsolvable. This algorithm can be seen in Algorithm 1.

3 SAT

The boolean satisfiability problem, which can be abbreviated to SAT, is a problem that decides if there is a valuation of variables of a propositional logic formula so that the formula evaluates

Algorithm 1 Our backtracking algorithm

Require: *pairs* is an array of pairs of cells, t is the total number of pairs

```
1: function BACKTRACKING(currentCell,correspondingCell,p)
2:     previousCell ← currentCell
3:     previousCorrCell ← correspondingCell
4:     previousp ← p
5:     for each direction do
6:         if this direction is blocked then
7:             continue
8:         currentCell ← currentCell.direction
9:         connect currentCell with current path
10:        if puzzle is solved then
11:            return true
12:        if currentCell = correspondingCell then
13:            if p + 1 < t then
14:                p ← p + 1
15:                currentCell ← first cell of pairs[p]
16:                correspondingCell ← second cell of pairs[p]
17:                start a new path
18:            else
19:                disconnect currentCell from current path
20:                currentCell ← previousCell
21:                continue
22:        if BACKTRACKING(currentCell,correspondingCell,P) then
23:            return true
24:        currentCell ← previousCell
25:        correspondingCell ← previousCorrCell
26:        p ← previousp
27:    return false
28: p ← 0
29: currentCell ← the first cell in pairs[0]
30: correspondingCell ← the second cell in pairs[0]
31: if BACKTRACKING(currentCell, correspondingCell, p) then
32:    Solution found
33: else
34:    Puzzle is unsolvable
```

to `true` [For09]. This formula needs to be in conjunctive normal form, CNF for short. This form consists of literals (variables or their negation), which can form a disjunction with other literals forming a clause. Finally, clauses form a conjunction with other clauses to form the expression. The variables can only be set to `true` or `false`. The formula $(A \vee B) \wedge (\neg B \vee C \vee \neg D) \wedge \neg A$ is an example of a propositional formula in CNF. This formula is satisfiable, because when variables A and D are `false` and when the variables B and C are `true` then the formula evaluates to `true`. Furthermore, when there is a valuation of the variables for which the formula evaluates to `true`, this means that the formula is satisfiable, and when there does not exist such a valuation, the formula is unsatisfiable.

3.1 SAT solver

A SAT solver aims to determine whether a given boolean satisfiability problem can be satisfied. We are going to use Minisat [Sö05]. This solver takes a configuration file, of which the first line is formatted as follows: "p cnf [number of literals] [number of clauses]". This line initializes the solver. The lines afterwards are the clauses with a zero to mark the end of a clause. The literals are numbered, and a negative number represents the term's negation. Consider the following example:

```
p cnf 4 3
1 0
-3 -4 0
-1 2 3 0
```

This configuration file states that there are 3 literals in this formula and 4 clauses. The formula encoded is: $X_1 \wedge (\neg X_3 \vee \neg X_4) \wedge (\neg X_1 \vee X_2 \vee X_3)$.

If the formula is satisfiable, the solver gives as output "`sat`" in the first line. The next line of the output, the solver gives a valuation of variables that satisfies the formula. In the case of the example, the output may become:

```
sat
1 2 -3 4
```

If the formula is unsatisfiable, the program just outputs "`unsat`".

3.2 Assigning Literals

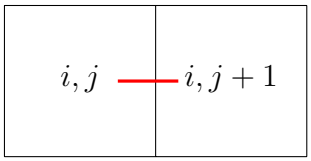
The board has a size of $m \times n$ and has p number pairs. We assign coordinates in the form (i, j) starting with $0, 0$ at the top left corner. j is increased every column to the right while i is increased every row downwards. An example can be seen in Figure 4.

(i,j)		$j=$			
		0	1	2	3
$i=$	0	$0,0$	$0,1$	$0,2$	$0,3$
	1	$1,0$	$1,1$	$1,2$	$1,3$
	2	$2,0$	$2,1$	$2,2$	$2,3$
	3	$3,0$	$3,1$	$3,2$	$3,3$

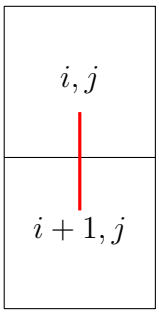
Figure 4: An example grid showing the coordinate system

We define three sets of literals :

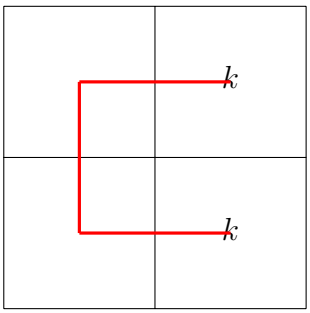
- The set H with literals $h_{i,j}$: This literal is true if the cell at position (i,j) is connected with the cell at position $(i,j+1)$ as seen in Figure 5a.
- The set V with literals $v_{i,j}$: This literal is true if the cell at position (i,j) is connected with the cell at position $(i+1,j)$ as seen in Figure 5b.
- The set N with literals $n_{i,j,k}$: This literal is true if the cell at position (i,j) is covered by a path belonging to pair k as seen in Figure 5c.



(a) $h_{i,j}$ is true when the cell at (i,j) is connected with cell $(i,j+1)$



(b) $v_{i,j}$ is true when the cell at (i,j) is connected with cell $(i+1,j)$



(c) $n_{i,j,k}$ is true if the cell at (i,j) is covered by the path belonging to number k.

Figure 5: A visual representation of the sets of literals

3.3 Encoding the Line Connection Constraints

An arbitrary cell has between two and four neighbors. We define the connection between this cell and its neighbors as $D_0(i,j) = \{h_{i,j}, h_{i,j-1}, v_{i,j}, v_{i-1,j}\}$ where $0 \leq i < m$ and $0 \leq j < n$. If a cell at (i,j)has less than four neighbors, the corresponding connection literal between the missing cell and

the cell at (i, j) is removed in $D_0(i, j)$.

For every cell, the number of lines connected to it has to be exactly 2 if the cell does not contain a number. If it does, then there can only be exactly 1 line connected. This "exactly-one" constraint is seen below. It is composed of 2 constraints: "at-least-one", which is the left clause, and "at-most-one", which is the right part of the formula.

$$\left(\bigvee_{X \in D_0(i,j)} X\right) \wedge \left(\bigwedge_{\{Y,Z\} \subseteq D_0(i,j), Y \neq Z} (\neg Y \vee \neg Z)\right) \tag{1}$$

If the cell needs exactly two lines connected, then we have three possibilities. The cell has either two, three, or four available lines. If there are two available lines, then both lines need to be true.

$$X \wedge Y \tag{2}$$

where X and Y are the 2 available orthogonal lines in $D_0(i, j)$.

If there are only 3 available lines, the formula becomes:

$$\left(\bigvee_{X \in D_0(i,j)} \neg X\right) \wedge \left(\bigwedge_{\{Y,Z\} \subseteq D_0(i,j), Y \neq Z} (Y \vee Z)\right) \tag{3}$$

This is similar to Formula (1) since this is an "exactly-one-false" constraint which is semantically equal to an "exactly-two" constraint when there are only 3 literals.

If there are 4 available lines, the formula becomes:

$$\left(\bigwedge_{\{X,Y,Z\} \subseteq D_0(i,j), X \neq Y \neq Z \neq X} (X \vee Y \vee Z)\right) \wedge \left(\bigwedge_{\{X,Y,Z\} \subseteq D_0(i,j), X \neq Y \neq Z \neq X} (\neg X \vee \neg Y \vee \neg Z)\right) \tag{4}$$

The left and right parts of the formula are similar to the "at-most-one" constraint. The only modification is that we add another literal, making the formula an "at-most-two" constraint. Since we have 4 available lines and 2 of which need to be true, then the other 2 lines need to be false. This is semantically the same as an "at-most-two-false" constraint (left side Formula (4)) and an "at-most-two-true" constraint (right side of Formula (4)).

3.4 Encoding the Number Constraints

There can only be 1 line present at each cell. So, we again make use of the "exactly-one" constraint.

$$\left(\bigvee_{a\leq p} n_{i,j,a}\right) \wedge \left(\bigwedge_{b\leq p, c\leq p, b\neq c} (\neg n_{i,j,b} \vee \neg n_{i,j,c})\right) \tag{5}$$

where p is the number of pairs of the puzzle starting at 1.

Moreover, a connected line implies a shared number with that cell. For $i = 0, 1, \ldots, m-1$, $j = 0, 1, \ldots n-1$, $k = 1, 2, \ldots, p$ we consider the following formulas below. For the horizontal connections, the formula is:

$$h_{i,j} \rightarrow \left(\bigvee_{k=1}^{p} n_{i,j,k} \wedge n_{i,j+1,k}\right) \tag{6}$$

For the vertical connections, the formula is:

$$v_{i,j} \rightarrow \left(\bigvee_{k=1}^{p} n_{i,j,k} \wedge n_{i+1,j,k}\right) \tag{7}$$

These formulas, however, are not in conjunctive normal form. To convert them we need to turn "$h_{i,j} \rightarrow$" into "$\neg h_{i,j} \vee$" and "$v_{i,j} \rightarrow$" into "$\neg v_{i,j} \vee$". Additionally, we need to distribute the formula. The formulas become:

$$\bigwedge_{u_k \in \{j, j+1\}} (\neg h_{i,j} \vee n_{i,u_1,1} \vee n_{i,u_2,2} \vee \cdots \vee n_{i,u_p,p}) \tag{8}$$

$$\bigwedge_{w_k \in \{i, i+1\}} (\neg v_{i,j} \vee n_{w_1,j,1} \vee n_{w_2,j,2} \vee \cdots \vee n_{w_k,j,p}) \tag{9}$$

Finally, the literals $n_{i,j,k}$ corresponding to the cells (i, j) that contain a number k need to be true.

3.5 Preventing Cycles

When the size of the board gets increased or when the number of pairs present is decreased, the probability of cycles appearing gets larger, see Figure 6. To prevent this, we add another set of literals and more constraints. The Vertex Tree approach from de Waart solves this issue [dW25]. The Vertex Tree is a tree on the grid where each corner of a cell is defined as a vertex that needs to point to an adjacent vertex neighbor in one of the following directions: up, right, down, or left. The vertices marked as root do not have to obey this rule. The whole outer layer of the grid is marked as root for the vertex graphs to form spanning trees rooted from the outer layer.

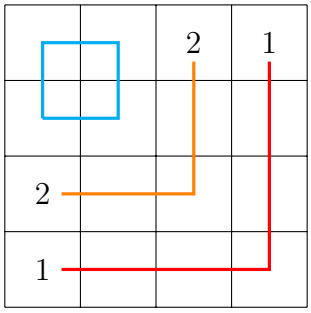


Figure 6: A cycle in the puzzle is not allowed

We add another set of literals:

- The set R with literals $r_{i,j,d}$ where $d \in D = \{\text{UP}, \text{RIGHT}, \text{DOWN}, \text{LEFT}\}$: This literal is true if the upper left vertex of cell (i, j) points to direction d, see Figure 7. To also include the most bottom and most right vertices, we add imaginary cells at $i = m$ and $j = n$ that are not present in the puzzle itself.

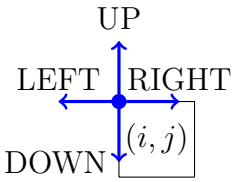


Figure 7: $r_{i,j,d}$ is true if the vertex at (i, j) points in direction d

There can only be one arrow pointing out for each vertex except the outermost layer since they are root. So, for every vertex where $i \neq 0, i \neq m, j \neq 0, j \neq n$, we again use the "exactly-one" constraint from Formula (1).

$$\left(\bigvee_{d \in D} r_{i,j,d}\right) \wedge \left(\bigwedge_{a,b \in D, a \neq b} \neg r_{i,j,a} \vee \neg r_{i,j,b}\right) \tag{10}$$

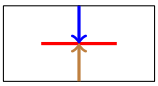


Figure 8: Vertices cannot form local cycles, and they cannot intersect with connected lines

Lastly, arrows from vertices cannot point to each other and cannot cross lines connecting cells, see Figure 8. We define $C_0(i,j) = \{r_{i,j,\text{UP}}, r_{i-1,j,\text{DOWN}}, h_{i-1,j-1}\}$ and $C_1(i,j) = \{r_{i,j,\text{LEFT}}, r_{i,j-1,\text{RIGHT}}, v_{i-1,j-1}\}$. At most one variable must be true per set, and the formulas become:

$$\bigwedge_{\{Y,Z\} \in C_0(i,j), X \neq Y} (\neg X \vee \neg Y) \tag{11}$$

$$\bigwedge_{\{Y,Z\} \in C_1(i,j), X \neq Y} (\neg X \vee \neg Y) \tag{12}$$

An example of the vertex tree can be viewed in Figure 9.

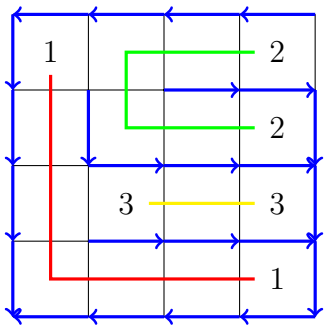


Figure 9: An example solved puzzle with visualization of the spanning vertex trees. The outermost layer of arrows of vertices is redundant since we consider this whole layer as root.

4 Generation and Heuristics

Our method of generating a random puzzle is the following. We first need to decide the size of the board and the amount of pairs in it. Then, for both numbers of each pair, we randomly choose a cell to insert the number. Minisat then makes an attempt to solve the generated puzzle using our encoding, see Section 3. If our solver evaluates the puzzle as unsatisfiable, we choose randomly again until our SAT solver evaluates the puzzle as satisfiable. A visual representation can be seen in Figure 10.

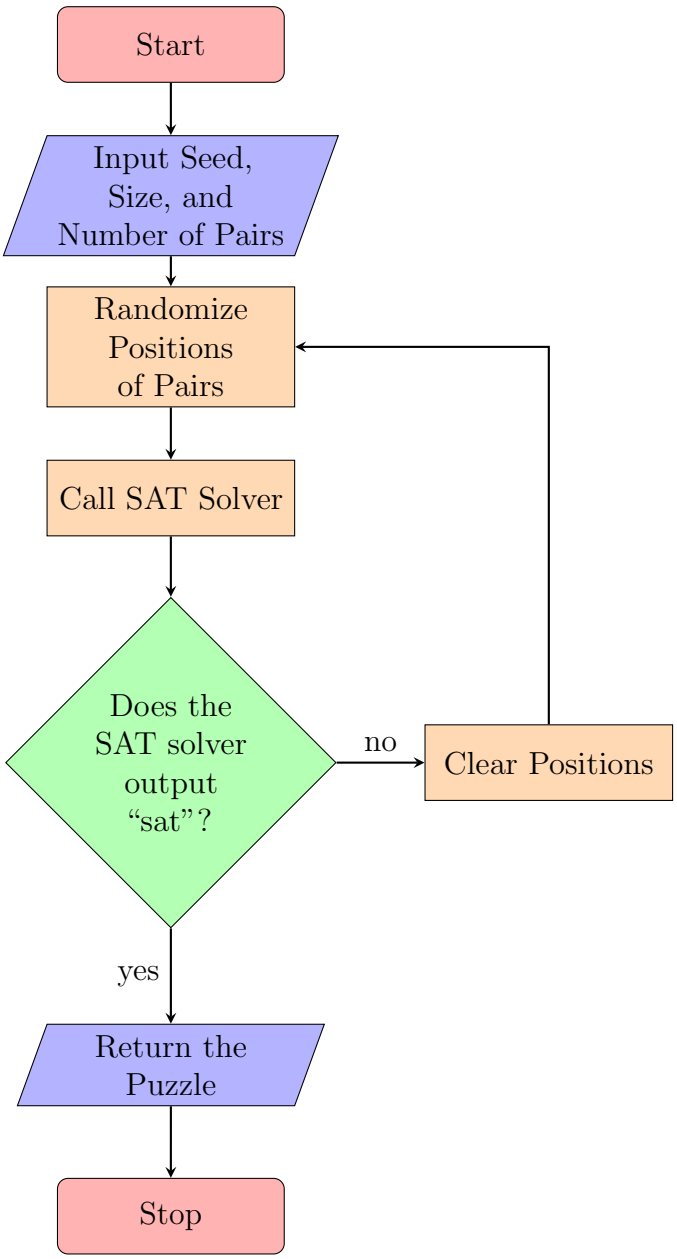


Figure 10: A flow chart of our method of generating a solvable puzzle

Randomly choosing cells and hoping the puzzle is solvable is an inefficient method since we are calling our SAT solver every time an attempt is made to generate a puzzle, and each time our solver is called, we need to wait before "unsat" is concluded. This is a slow process, which is why we want to develop heuristics that aid us in deciding the generated puzzle's solvability. These heuristics are used before the SAT solver is called, and when the heuristics determine that the puzzle candidate is unsolvable, we make another attempt at generating a puzzle, see Figure 11. This is used as an attempt to generate a puzzle faster.

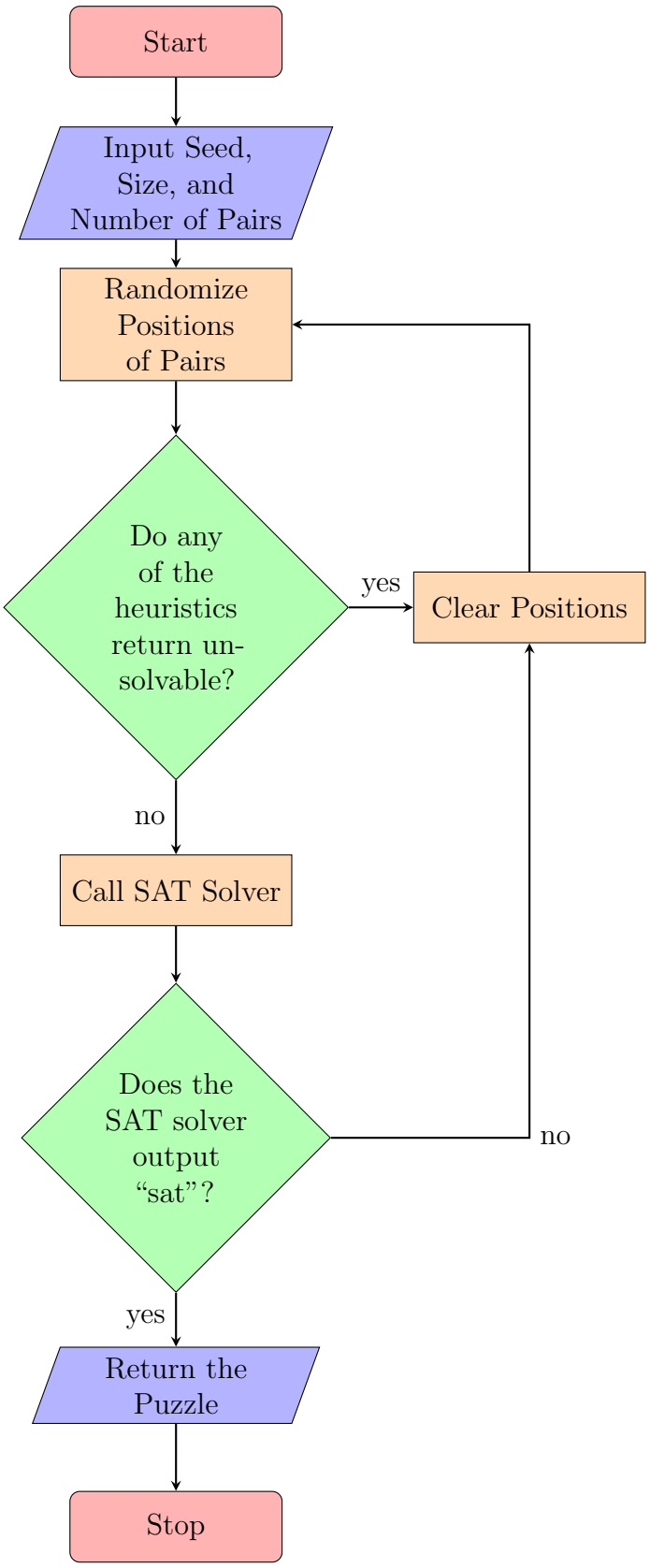
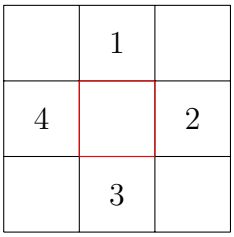


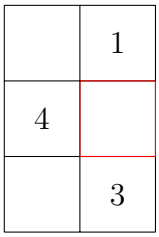
Figure 11: A flow chart of our edited generation method involving using heuristics to make the generation method faster

4.1 Isolated Cells

When a cell is encased by numbers, there exists no path that can connect a pair with this cell. In Figure 12a, we can see that the red cell is unable to connect with any other cell, which makes the puzzle unsolvable. A cell can also be partially blocked when its position is at the edge of the grid, see Figure 12b. In this case, fewer cells are needed to fully isolate this cell.



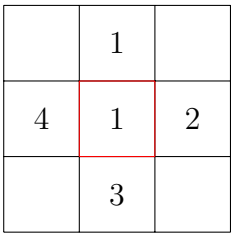
(a) An isolated cell which is encased only by numbers



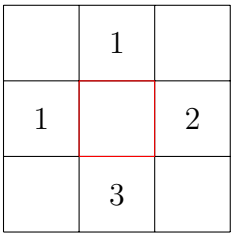
(b) An isolated cell with blocked paths by numbers and the edge of the grid

Figure 12: Examples of isolated cells

This statement has 2 exceptions. The first exception is when cell itself is numbered and its number is the same as an adjacent cell's number, see Figure 13a. The second exception is when the number of an adjacent cell is the same as the number of another adjacent cell, see Figure 13b.



(a) Exception 1: The red cell contains the same number as one of the adjacent cells



(b) Exception 2: A red cell's adjacent cell contains the same number as one of the other adjacent cells

Figure 13: Exceptions where the red cell can be connected

4.2 Parity

We propose a formula that evaluates to `false` if a given puzzle cannot be solved. This is done by evaluating all positions of pairs, specifically the distances between the cells and their copy. The following lemma states that the shortest distance mod 2 is equal to any distance mod 2.

Lemma. *For each path q going from coordinate (i_1, j_1) to coordinate (i_2, j_2), the following holds:* $c_q \bmod 2 = (|i_1 - i_2| + |j_1 - j_2| - 1) \bmod 2$ *where c_q is the number of cells path q covers.*

This lemma works because every step the path takes away from the shortest path, the path has to take a step towards the shortest path eventually, see the following proof.

Proof. We define c_q, which is the number of cells that path q covers. The parity of c_q going from coordinate (i_1, j_1) to coordinate (i_2, j_2) is $c_q \bmod 2 = (|i_1 - i_2| + |j_1 - j_2| - 1) \bmod 2$. This is because the parity of the shortest distance d from (i_1, j_1) to (i_2, j_2) is $(|i_1 - j_2| + |i_1 - j_2| - 1) \bmod 2$. This is due to the following. The shortest distance between row i_1 and row i_2 is $|i_1 - i_2|$ and the shortest distance between column j_1 and column j_2 is $|j_1 - j_2|$. In addition, for every additional vertical step path q takes away from the shortest path, q needs to take another step backwards to finish at the same row. The same applies horizontally to the columns. Finally, the number of cells covered by the distance c_p is $d - 1$ because the distance is calculated from the centers of the cells, while c_p excludes the coordinate cells.

□

When we apply this lemma for all paths belonging to a pair, we obtain a formula which needs to be true if the puzzle is solvable.

Corollary. *With* $di_l = |i_{l1} - i_{l2}|$ *and* $dj_l = |y_{l1} - y_{l2}|$*, we can deduce the following:*

$$\sum_{l=1}^{p} c_l \mod 2 \equiv ((\sum_{l=1}^{p} di_l + dj_l) - p) \mod 2$$

. Which is equivalent to the following formula:

$$(m \times n) \mod 2 \equiv ((\sum_{l=1}^{p} di_l + dj_l) - p) \mod 2$$

.

When we apply this formula to a given puzzle in a boolean expression, it outputs false if the parity for this puzzle is violated, meaning we can already conclude that this given puzzle is unsolvable.

5 The Hexagonal Variant

So far, we have discussed puzzles containing cells with the shape of squares. In a hexagonal-shaped environment, the coordinate system is different compared to the square environment, see Figure 14a. Additionally, in this environment, the paths that connect the pairs are formed by going through one of the six sides of the hexagon instead of the 4 sides of a square, see Figure 14b.

(a) The coordinate system

(b) An example of a connected path

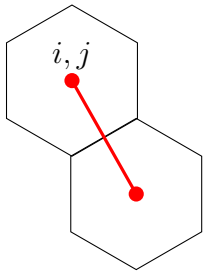
Figure 14: The hexagonal environment

Our DFS method works for this environment as well without any changes because the only difference is that the algorithm now has five other directions it can go to instead of three.

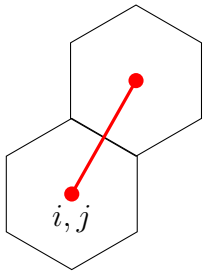
5.1 Redefinition SAT Literals

For our SAT solver to work in a hexagonal environment, we need to redefine our literals:

- The set L with literals $l_{i,j}$: This literal is true if the cell at the bottom right corner of this cell is connected with this cell as seen in Figure 15a.
- The set P with literals $p_{i,j}$: This literal is true if the cell at the top right corner of this cell is connected with this cell as seen in Figure 15b.



(a) $l_{i,j}$ is true when the cell at (i, j) is connected with the cell at its bottom right corner



(b) $p_{i,j}$ is true when the cell at (i, j) is connected with the cell at its top right corner

Figure 15: The hexagonal line connection literals

We keep the definitions of the set H and the set N, see Section 3. Furthermore, we re-encode the line connection constraints as follows: an arbitrary cell has between 2 and 6 neighbors. We define the connection between this cell and its neighbors as $D_0(i, j) = \{h_{i,j}, h_{i,j-1}, l_{i,j}, l_{i-1,j}, p_{i,j}, p_{i-1,}\}$ where $0 \leq i < m$ and $0 \leq j < n$.

5.2 Modification of the line connection constraint

For every cell, the number of lines connected has to be exactly two if the cell does not contain a number. If it does, then there can only be exactly one line connected. We already defined this encoding for this constraint, see Formula (1) in Section 3. If the cell needs exactly two lines connected, then we have five possibilities. The cell has either two, three, four, five, or six available lines. We already discussed the scenarios when the cell has two, three, or four available lines, see Formulas (2), (3), and (4) in Section 3. If the cell has five available lines, the formula becomes:

$$\left(\bigwedge_{\{X,Y,Z\} \subseteq D_0(i,j), X \neq Y \neq Z \neq X} (\neg X \vee \neg Y \vee \neg Z) \right) \wedge \left(\bigwedge_{T \subseteq D_0(i,j), |T|=4} \bigvee_{t \in T} t \right) \tag{13}$$

The left side of the formula is the same as the right side of Formula (4). This is because we want to keep the "at-most-two-true" constraint. The right side of the formula is a modification of the left side of Formula (4) to be an "at-most-three-false" constraint by adding one more literal. The "at-most-two-true-constraint" together with the "at-most-three-false constraint" make an "exactly-two" constraint when there are five variables present. If the cell has six available cells, the formula becomes:

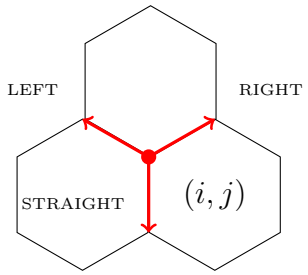
$$\left(\bigwedge_{\{X,Y,Z\}\subseteq D_0(i,j),X\neq Y\neq Z\neq X}(\neg X\vee\neg Y\vee\neg Z)\right)\wedge\left(\bigwedge_{T\subseteq D_0(i,j),|T|=5}\bigvee_{t\in T}t\right) \tag{14}$$

We want to keep the "at-most-two-true" constraint so the right side of the formula is the same as the right side of Formula (4) and Formula (13), We added one more literal on the right side of the formula compared to the right side of Formula (13) to create an "at-most-four-false" constraint. The "at-most-two-true-constraint" together with the "at-most-four-false-constraint" make an "exactly-two" constraint when there are six variables present. For the number constraints, we keep the formulas defined in Section 3 except that we do not use $v_{i,j}$ in this environment and instead use $p_{i,j}$ and $l_{i,j}$.

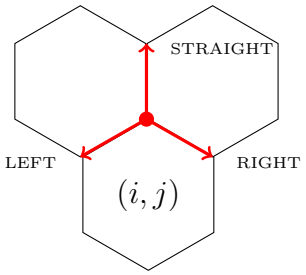
5.3 The Vertex Tree

For the Vertex Tree to be added, we need to add an additional set of literals.

- The set R with literals with literals $r_{i,j,k,d}$ where $k \in K = \{\text{CORNER}, \text{TOP}\}$ and $d \in D = \{\text{LEFT}, \text{STRAIGHT}, \text{RIGHT}\}$: This literal is true if the vertex at k of cell (i, j) points to direction d, see Figure 16. As opposed to the square environment, every cell has six vertices. Since every vertex is shared with at most three cells, we attribute $6/3 = 2$ vertices to every cell. These are the corner vertex and the top vertex.



(a) The literal $r_{i,j,\text{CORNER},d}$ is true if the vertex at the corner of cell (i, j) points to another vertex in direction d



(b) The literal $r_{i,j,\text{TOP},d}$ is true if the vertex at the top of cell (i, j) points to another vertex in direction d

Figure 16: Visualization of literal $r_{i,j,k,d}$

We modify Formula (10) so that it satisfies the hexagonal environment instead of the square environment, see Formula (15).

$$\bigwedge_{k \in K} \left(\left(\bigvee_{d \in D} r_{i,j,k,d} \right) \wedge \left(\bigwedge_{a,b \in D, a \neq b} (\neg r_{i,j,k,a} \vee \neg r_{i,j,k,b}) \right) \right) \tag{15}$$

Lastly, to prevent vertices from pointing to each other, we keep Formulas (11) and (12) from Section 3. However, we need to redefine the sets $C_0(i,j)$ and $C_1(i,j)$ so they match the hexagonal environment. $C_0(i,j) = \{r_{i,j,\text{CORNER,RIGHT}}, r_{i,j,\text{TOP,LEFT}}, l_{i-1,j}\}$ and $C_1(i,j) = \{r_{i,j,\text{TOP,RIGHT}}, r_{i,j+1,\text{CORNER,LEFT}}, p_{i,j}\}$. Furthermore, we also need to add a formula for the vertical vertex connections. We define $C_2(i,j) = \{r_{i,j,\text{TOP,STRAIGHT}}, r_{i-1,j+1,\text{CORNER,STRAIGHT}}, h_{i-1,j}\}$. The added formula is:

$$\bigwedge_{\{Y,Z\} \in C_2(i,j), X \neq Y} (\neg X \vee \neg Y) \tag{16}$$

6 Experiments

We performed some experiments to test the effectiveness and performance of our DFS and SAT implementations, our isolated cells heuristic, and our parity heuristic. In these experiments, the size of the board and the number of pairs are varied. Additionally, the average execution time of 100 different puzzles is considered.

6.1 Comparison Solving Methods

We compare the performance of our backtracking implementation with the performance of our SAT implementation. We have generated 100 puzzles for various sizes and for various numbers of pairs. We then executed our backtracking and our SAT implementation on the same set of puzzles and compared their average solving time. The results can be seen in Table 1. For clarification, we did not perform the experiments on a larger number of pairs with relatively small sizes. This is because the density of the number of pairs over the number of cells increases, making it harder for our generation methods to generate a puzzle.

	5×5	6×6	7×7	8×8	9×9	10×10
2	0.00002601	0.00565	0.932	> 10	> 10	> 10
3	0.00001799	0.00140	0.432	> 10	> 10	> 10
4	0.00000865	0.000467	0.0863	> 10	> 10	> 10
5	0.00000549	0.000151	0.0230	> 10	> 10	> 10
6	0.00000323	0.00007255	0.009921	4.76	> 10	> 10
7	N/A	0.00003266	0.003692	1.23	> 10	> 10
8	N/A	N/A	0.0009434	0.468	> 10	> 10
9	N/A	N/A	N/A	0.0860	> 10	> 10

(a) Average execution times of backtracking in seconds

	5×5	6×6	7×7	8×8	9×9	10×10
2	0.00242	0.002708	0.003063	0.003561	0.00453	0.00499
3	0.002691	0.003147	0.003690	0.004544	0.006426	0.007111
4	0.002868	0.003593	0.004175	0.005485	0.008027	0.0106
5	0.003566	0.004822	0.005811	0.007646	0.0107	0.0190
6	0.005398	0.007322	0.0106	0.01268	0.0171	0.0298
7	N/A	$0.0134s$	0.0206	0.02427	0.0366	0.0472
8	N/A	N/A	0.0410	0.1183	0.0758	0.109
9	N/A	N/A	N/A	0.15	0.193	0.268

(b) Average execution times of SAT in seconds

Table 1: Average execution times of backtracking compared to SAT

As we can see in Table 1, both SAT and backtracking take longer to solve as the size of the board increases. However, our SAT solver is significantly faster than the backtracking method for larger sizes. Interestingly, the execution time of SAT increases as the number of pairs increases, while the execution time of our backtracking method decreases as the number of pairs increases.

6.2 Comparison heuristics

To test the effectiveness of our Isolated Cells heuristic and our Parity heuristic in the generation of puzzles, we measure the average generation times of 100 puzzles with various sizes and pairs twice. The first set of measurements is our baseline SAT implementation, without the heuristic enabled. The second set of measurements is with the Isolated Cells heuristic enabled and the Parity heuristic disabled, and the third set of measurements is with the Isolated Cells heuristic disabled and the Parity heuristic enabled. The results can be seen in Table 2.

	5×5	6×6	7×7	8×8	9×9	10×10
2	0.008586	0.00915	0.0180	0.0157	0.0852	0.310
3	0.0193	0.0134	0.0203	0.0251	0.0769	0.214
4	0.070	0.0502	0.0352	0.0408	0.0974	0.292
5	0.575	0.281	0.171	0.116	0.150	0.369
6	7.16	2.69	1.68	0.665	0.511	0.594
7	> 10	> 10	> 10	8.05	4.65	1.87
8	> 10	> 10	> 10	> 10	> 10	> 10
9	> 10	> 10	> 10	> 10	> 10	> 10

(a) average generation times of the baseline SAT implementation in seconds

	5×5	6×6	7×7	8×8	9×9	10×10
2	0.0107	0.006564	0.0156	0.0170	0.102	0.291
3	0.0207	0.0130	0.0203	0.0224	0.0949	0.527
4	0.0824	0.0427	0.0375	0.0402	0.0824	0.318
5	0.445	0.297	0.197	0.120	0.170	0.291
6	7.41	3.14	1.52	0.709	0.513	0.653
7	> 10	> 10	> 10	7.52	4.55	2.23
8	> 10	> 10	> 10	> 10	> 10	> 10
9	> 10	> 10	> 10	> 10	> 10	> 10

(b) average generation times of SAT with the isolated cells heuristic enabled in seconds

	5×5	6×6	7×7	8×8	9×9	10×10
2	0.004689	0.003653	0.007151	0.005443	0.192	0.482
3	0.008883	0.008388	0.009172	0.009129	0.105	0.471
4	0.0402	0.0249	0.0191	0.0180	0.0806	0.277
5	0.264	0.148	0.0747	0.0621	0.130	0.219
6	2.90	1.37	0.906	0.323	0.483	0.351
7	> 10	> 10	8.05	3.38	2.76	1.58
8	> 10	> 10	> 10	> 10	> 10	> 10
9	> 10	> 10	> 10	> 10	> 10	> 10

(c) Average generation times of SAT with the parity heuristic enabled in seconds

Table 2: average generation time of our baseline SAT implementation compared to the generation times of our Isolated Cells heuristic enabled and the generation times with our Parity heuristic enabled.

These results show that the Isolated Cells heuristic, unfortunately, is not effective. While some cases are faster with the heuristic enabled, there are also some cases where the Isolated Cells heuristic is slower compared to when the heuristic is disabled. Furthermore, most cases do not show a noteworthy difference. However, the results also show that generating with the Parity heuristic is considerably faster in almost all cases. This speedup is especially apparent when the size of the puzzle is small and decreases as the size increases. In a 10×10 puzzle, the speedup becomes approximately zero, making the parity heuristic ineffective. In theory, this speedup should be two, since this heuristic eliminates half of the puzzles our generation methods provide. The results in 2c

reflect this theoretical speedup, because in most cases, the speedup is approximately two.

7 Summary and Conclusion

A Numberlink puzzle consists of an $m \times n$ grid with pairs of numbers, where each number is scattered throughout the board, and the goal is to connect these numbers with their copy via paths. We explained how our backtracking implementation works. This method aims solve this puzzle using depth-first-search by going recursively through the cells of the puzzle for each pair, trying to match with their copy. We also developed a SAT encoding so a SAT solver can solve the puzzle. We defined the puzzle's necessary variables and showed how we encoded the line connection constraint and the number constraint. We also showed how to prevent cycles using the Vertex Tree solution. We explained how to generate random puzzles by randomly scattering numbers throughout the grid, call our SAT solver to see if this configuration is solvable, and try again if it is not. To try to make the generation process faster, we developed the Isolated Cells heuristic and the Parity heuristic. The Isolated Cells heuristic determines if there exists a cell that is trapped by numbers, and the Parity heuristic determines if the parity of the puzzle is correct. Moving on, we discussed the hexagonal variant of Numberlink and how it affects the SAT encoding.

To compare our implementations, we performed experiments to determine which of the two solving methods is faster and the effectiveness and performance of our two developed heuristics. These experiments show that SAT is generally faster in solving Numberlink puzzles compared to DFS, but in smaller sizes and higher pairs, DFS excels. Our experiments also show that our Isolated Cells heuristic is not as effective as we had hoped, but our Parity heuristic is generally faster.

8 Further Research

Some topics for further research include the following. Adding more heuristics to make the generation process even faster. We have only considered blockades and parity of the puzzle, and therefore, there could be more heuristics we have not discussed in this thesis. Moreover, generating valid puzzles from solutions in comparison to randomly shuffling the board or generating unique solutions may be worthwhile to invest time into. Additionally, it might also be interesting to investigate other variants of Numberlink, for example, triangular Numberlink, where the cells of the grid are shaped like a triangle instead of a square or a hexagon. It could also be interesting to experiment with other shapes, or a variant where instead of pairs that need to connect, the goal is connect triplets or maybe quadruplets or more.

References

[ADD+15] Aaron Adcock, Erik D. Demaine, Martin L. Demaine, Michael P. O'Brien, Felix Reidl, Fernando Sánchez Villaamil, and Blair D. Sullivan. Zig-Zag Numberlink is NP-Complete. *Journal of Information Processing*, 23(3):239–245, 2015.

[dW25] Jente de Waart. Comparing different encodings for the continuity rule of the logic puzzle Context. Bachelor's thesis, Leiden University, 2025.

[For09] Lance Fortnow. The Status of the P versus NP Problem. *Communications of the ACM*, 52(9):78–86, 2009.

[Gre13] Aake Gregertsen. Wire Storm — Fun and Addicting Logic Flow Puzzle Game. App Store, 2013.

[Hes24] Niels Heslenfeld. Solving and Generating Fobidoshi puzzles. Bachelor's thesis, Leiden University, 2024.

[LLC15] Big Duck Games LLC. Flow Free. App Store, 2015.

[Nik] Nikoli. Nikoli Puzzles. https://www.nikoli.co.jp/en/puzzles/.

[Peg07] Pegg, E, Jr. Numberlink: Beyond Sudoku. *Mathematica Journal*, 10(3), 2007.

[RI20] Suthee Ruangwises and Toshiya Itoh. Physical Zero-Knowledge Proof for Numberlink. In Martin Farach-Colton, Giuseppe Prencipe, and Ryuhei Uehara, editors, *10th International Conference on Fun with Algorithms (FUN 2021)*, volume 157 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:11, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Str25] Hanna Straathof. Solving and Generating Kuroshuto puzzles. Bachelor's thesis, Leiden University, 2025.

[Sö05] Niklas Sörensson. Minisat. 2005.