Universiteit Leiden
The Netherlands

Computer Science

Generating Pipes puzzles using maze-generating algorithms

Meili Hegeman

Supervisors:
Hendrik Jan Hoogeboom & Walter Kosters

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl 24/05/2022

Abstract

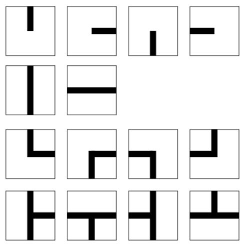
This thesis analyses methods to generate a Pipes puzzle, a logic puzzle also known as FreeNet. A puzzle consists of a field of pipes, where all pipes can be rotated repeatedly by 90°. The goal is to connect all pipes to the power source located at the centre. The fields are generated using different maze-generating algorithms, and are considered valid if it has a unique solution. In this thesis, we convert a field into a Boolean expression and use a SAT solver to verify whether a field is a valid puzzle. We also compare the performance of the different algorithms.

Contents

1 Introduction

Pipes is a puzzle played on a $n \times m$ grid. The field consists of various types of tiles filled with up to three pipes in the directions north, east, south and west. Each tile can be repeatedly rotated by 90°. One of the tiles located at the centre of the field also contains a power source. It does not affect the gameplay, but is merely there to add visuals and context to the game. The puzzle is also known as FreeNet. A field consists of the following tiles:

- Dead-end tile (D)
- Straight tile (S):
- Corner tile (C):
- T-join (T):



We call a side of a tile *open* if a pipe is running from the centre to that side. Two open sides of adjacent tiles can be connected by rotating the tiles until they touch each other. We call a configuration of a field a *safe state* if all tiles are connected with their neighbours in such a way that no open side is left unconnected and no closed side faces an open side. If a safe state does not contain cycles, it is a *solution*.

The puzzle can be played online [1]. A sample start configuration and its corresponding solution can be found in Figure 1.

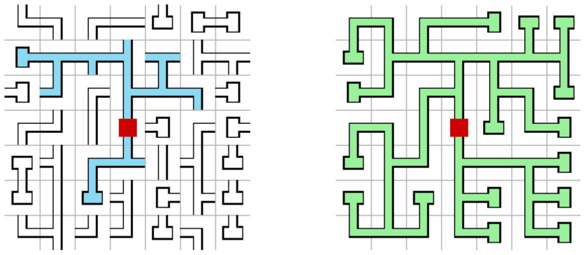


Figure 1: An example of a 7 × 7 puzzle (left) and its solution (right), visualised using software developed for this project

1.1 Complexity

We can easily see that a solution of an instance of the pipes problem can be verified in polynomial time in field size $n \times m$. We can do so by performing a depth-first search, where we need to check that each path ends with a D-tile, and that all tiles are visited exactly once during the search.

A more difficult problem is to determine the computational complexity of the problem. Using a reduction from Hamilton Cycles in graphs with with degree three on grids, one can prove that it is $\mathcal{NP}$-complete [3]. This is done by creating smaller gadgets, consisting of puzzle tiles, which represent parts of a graph. For example, a gadget for a node is shown in Figure 2.

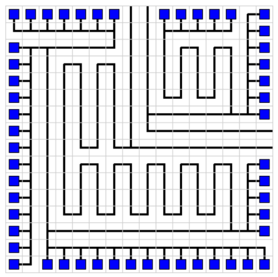


Figure 2: A corner-gadget used for a reduction from Hamilton Cycles in graphs with degree three on grids

1.2 Generating puzzles

A common method to generate puzzles is by brute-force. This method involves generating a field of random tiles and verifying whether it has a unique solution. It can be improved by including some requirements of the puzzle during the generation of fields, and thus create candidate fields with a higher probability of being a valid puzzle[1].

One way to improve the candidate fields is by starting with a solution. The actual puzzle can be created by randomly rotating all the tiles to create a different configuration. A valid solution has the shape of a tree placed on a grid, or rather, a maze. We can use maze-generating algorithms to find solutions. For this thesis, we will analyse the following algorithms: BinaryTree, DepthFirst, Kruskal, Prim and Wilson. The original use of these algorithms is not necessarily to generate mazes, but with small modifications they are easy to implement.

We call a solution unique if it is the only solution for a field. After generating a solution, we have to check that the field cannot be rotated into a different solution. This is done using a SAT solver. A SAT solver is a tool that efficiently calculates whether it is possible to find an assignment for variables which satisfies

[1]`https://twitter.com/PuzzleTeamClub/status/1353454873988255746`

a Boolean expression. As we want to analyse a large amount of puzzles, this is a fast way to do so. We convert the field to a list of clauses which are satisfiable if and only if it is possible to connect all open sides in another way without creating cycles, where at least one tile is in a different orientation than the starting solution. If the expression is satisfiable, the candidate field has multiple solutions and is considered an invalid puzzle.

2 Maze-generating algorithms

To generate puzzles, we will be looking at maze-generating algorithms. A valid solution of the puzzle is the configuration is a safe state, which is a field in which all tiles are connected through pipes, without any cycles. This means that a valid solution has the structure of a spanning tree placed on a grid, also known as a maze. In this section, we will discuss various maze generating algorithms used to create puzzle candidates.

The algorithms used are Binary Tree, Depth-first, Kruskal, Prim and Wilson. While Binary Tree is not commonly used, the other algorithms are usually applied to graphs. Kruskal and Prim compute a minimum spanning tree, where we choose a subset of edges with the minimum possible total weight. Wilson determines a uniform spanning tree, where each edge has the same probability of being chosen.

We view the centres of the tiles as nodes, and connected pipes between them as edges. All algorithms are slightly modified to prevent tiles with four open sides. These modifications are not shown in the pseudocode to keep it clear, but described in each section.

Let T be the set of tiles in the field, and let $D = \{n, e, s, w\}$ be the set of directions. Let u_n, u_e, u_s, u_w denote the northern, eastern, southern and western neighbour of a tile $t \in T$. Let $r(p, q)$ be a random number generator, generating numbers between and including p and q, and let $R(t)$ be a function that randomly selects an unvisited tile.

2.1 Binary Tree

The *Binary Tree algorithm* (Algorithm 1) is a very simple algorithm that generates a maze with the structure of a binary tree. For each tile, the algorithm randomly creates a passage north or west. If a tile is positioned at the top or left edge of the field, it takes the only possible passage [4].

Each node can have at most two children, which makes it impossible to create a tile with four open sides. No additional modifications are needed.

The main advantage of this algorithm is its simplicity. We do not need to keep any additional state or data during the execution, apart from saving the field. All tiles are independent from each other, and can be treated similarly. However, we do see that all generated mazes have a specific texture. As this algorithm only creates passages towards the north and west, half of the orientations of tiles will never

Algorithm 1 Binary Tree

```
 1: for $t \in T$ do
 2:     if $u_n$ does not exist then                        ▷ edge tiles
 3:         connect($t$, $u_w$)
 4:     else if $u_w$ does not exist then
 5:         connect($t$, $u_n$)
 6:     else if $r(0, 1) = 0$ then                          ▷ remaining tiles
 7:         connect($t$, $u_n$)
 8:     else
 9:         connect($t$, $u_e$)
10:     end if
11: end for
```

occur in the maze. The top-left edge piece is exceptional as it has no northern or western neighbours. This C-tile will occur exactly once in the orientation where the eastern and southern side are open. An example of a maze is shown in Figure 3.

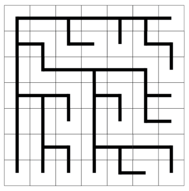


Figure 3: A maze generated by the Binary Tree algorithm

2.2 Depth-first

The *Depth-first algorithm* (Algorithm 2) is commonly used for tree traversal. It starts at the root, and continues along a path as long as possible. Unvisited neighbours will be stored on a stack. Upon reaching a leaf, the algorithm backtracks by popping a node from the stack [7, Chapter 22].

Instead of traversing a tree, we will use the algorithm to build one in our field. We start by adding the centre to the stack. We visit the element at the top and remove it from the stack. Then we connect all unvisited neighbours and add them to the stack in a random order. This prevents a bias towards a certain direction. We continue with a new tile at the top of the stack. If a tile already has three open sides, we will not create a fourth path. The algorithm will visit the tile from another direction.

Algorithm 2 Depth-first

```
1: stack $S$
2: tile $t = t_{\text{middle}}$
3: push $t$ to $S$
4: **while** $S$ is not empty **do**
5:     $t = S$.top
6:     $S$.pop
7:     visit $t$
8:     stack *temp*
9:     **for** $d \in D$ **and** $u_d$ unvisited **do**          ▷ connect neighbours
10:        connect($t$,$u_d$)
11:        push $u_d$ to $S$
12:    **end for**
13: **end while**
```

Although there is no major bias caused by the algorithm, we do find that it tends to give long paths, especially along the boundary. The algorithm wants to traverse the field as far as possible. For boundary tiles, there are fewer directions possible, and thus there is a higher chance of only being able to continue along the boundary. The centre tile is always a T-tile. An example of a maze is shown in Figure 4.

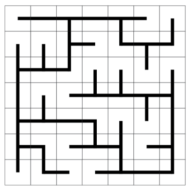


Figure 4: A maze generated by the Depth-first algorithm

2.3 Kruskal

Kruskal's algorithm (Algorithm 3) computes a minimum spanning tree in a weighted graph. It is a greedy algorithm, which means that it will only consider the current state while choosing the best option. It starts by sorting all edges in non-decreasing order, and adds them to the constructed forest in this order provided the edge does not create a cycle. The algorithm uses disjoint sets to keep track of the different components, implemented as a Union-Find data structure. We use the operation FIND to return the component index of a node and UNION to join two components. At the beginning, a unique component index is assigned to each node.

After connecting two nodes, the two components are joined, and all connected tiles get the same index. If two tiles already belong to the same component, it means that the algorithm is about to create a cycle, and the edge is discarded [7, Chapter 23].

In our field, we view the tiles as nodes and the connections between them as edges. If we assign each edge the same weight, we build a uniform spanning tree instead. This means that there is no need to sort the edges. To prevent tiles with four open sides, we also discard an edge if one of the tiles already has three open sides.

Algorithm 3 Kruskal (with Union-Find)

```
 1: vector $V$ with pairs $(t, u)$
 2: int array $set$
 3: int $k = 0$
 4: for $t \in T$ do
 5:     $set[t] = k$                               ▷ unique set for each tile
 6:     $k = k + 1$
 7:     if $u_e$ exists then                       ▷ push all possible connections
 8:         push $(t, u_e)$ to $V$
 9:     end if
10:     if $u_s$ exists then
11:         push $(t, u_s)$ to $V$
12:     end if
13: end for
14: while $V$ is not empty do                      ▷ randomly connect two tiles
15:     $x = r(1, \text{size of } V)$
16:     $t = V[x]_1$, $u = V[x]_2$
17:     if find($t$) is not find($u$) then         ▷ check for cycles
18:         connect($t$, $u$)
19:         union(find($t$), find($u$))
20:     end if
21:     erase $V[x]$ from $V$
22: end while
```

It is necessary to store all possible edges in the field, which requires more memory than most other algorithms for larger puzzles. Fortunately, the complexity is still only $O(n \cdot m)$, which does not make it an issue. An example of a maze is shown in Figure 5.

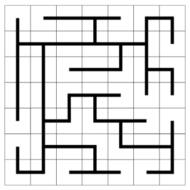


Figure 5: A maze generated by Kruskal's algorithm

2.4 Prim

Prim's algorithm (Algorithm 4) is similar to Kruskal's algorithm. It is another greedy algorithm that computes a minimum spanning tree in a weighted graph. It starts at one node, which will be the root, and stores all reachable edges to unvisited neighbours in a vector. It then continues to add the shortest edge each step [7, Chapter 23].

Once again, we view the tiles as nodes and connections as edges. We give all edges the same weight to generate a random spanning tree and start at the centre of the field.

Algorithm 4 Prim

```
vector $V$ with pairs $(t, u)$
tile $t = t_{\text{middle}}$
for $d \in D$ do                                  ▷ add centre tile
    push $(t, u_d)$ to $V$
end for
while $V$ is not empty do
    visit $t$
    for $d \in D$ do                              ▷ add neighbours to vector
        if $u_d$ unvisited then
            push $(t,\ u_d)$ to $V$
        end if
    end for
    $x = r(1, \text{size of } V)$
    if $V[x]_2$ unvisited then                    ▷ connect random reachable tile
        connect($V[x]_1$,$V[x]_2$)
        $t = V[x]_2$
    end if                                        ▷ else continue with the same tile
    erase $V[x]$ from $V$
end while
```

If a tile already has three open sides, we discard the edge and mark the tile as *unvisited*. Unfortunately, it will not necessarily be reached again from another neighbour. To fix this, we traverse the field afterwards and connect all unvisited tiles to a random neighbour with fewer than three open sides. This is possible if at least one neighbour is not a T-tile. We notice that if all neighbours are T-tiles, there would be a cycle around the unvisited tile, as illustrated in Figure 6.

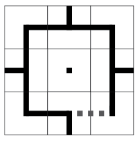


Figure 6: All tiles can be connected with at most three open sides

However, the algorithm creates a tree, which means that no cycles are present. Thus, it follows that we will always be able to reach the tile from at least one side. In this example, the algorithm creates a path between the unvisited tile and its southern neighbour. Even in the unlikely event that a tile is surrounded by unvisited tiles, the structure stays the same, and thus, it will still be possible to connect all of them to the tree without creating cycles. An example of a maze is shown in Figure 7.

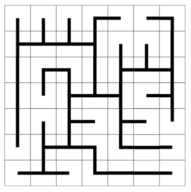
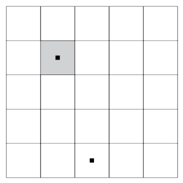


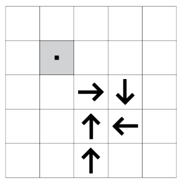
Figure 7: A maze generated by Prim's algorithm

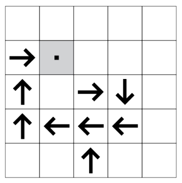
2.5 Wilson

Wilson's algorithm (Algorithm 5) is an algorithm that computes a random spanning tree. It selects a random node to be the root. Next, it chooses an unvisited node and performs a random walk until a visited node is encountered. All nodes along the path will be visited and added to the tree [4].

A *random walk* describes a path that consists of a succession of random steps. In this case, the steps are directions across the field. It is allowed to enter the same tile multiple times. However, the final path has a constraint, as we do not want any cycles. To ensure this, we only save the most recent direction used to exit a tile. Figure 8 illustrates how a random walk is performed and converted to a path in the maze.


(a) Add a random point to the maze and select a random starting point


(b) Start the random walk


(c) Overwrite most recent direction and continue

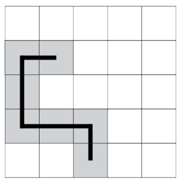

(d) Add all tiles along the path to the maze. All unused directions are discarded

Figure 8: A random walk converted into a path in the maze

After adding the path to the maze, a new random walk is performed. This time, there are more points where the algorithm can end the walk, as new tiles were just added to the maze. We repeat this process until all tiles are included in the maze.

Algorithm 5 Wilson

```
 1: tile $t = R(t)$                              ▷ select a random tile
 2: visit $t$
 3: while $t$ unvisited exists do
 4:     tile $t_{\text{start}} = R(t)$
 5:     $t = t_{\text{start}}$
 6:     int array $dir$
 7:     while $t$ unvisited do                   ▷ perform a random walk over the field
 8:         $r = r(0, 3)$
 9:         $dir[t] = r$                          ▷ remember last direction
10:         $t = t_r$
11:     end while
12:     $t = t_{\text{start}}$
13:     while $t$ unvisited do                   ▷ connect tiles from walk
14:         connect($t$, $t_{dir[t]}$)
15:         visit $t$
16:         $t = t_{dir[t]}$
17:     end while
18: end while
```

If a tile already has three connections, we prevent the algorithm from entering it, to ensure that no tiles with four open sides will occur. An example of a maze is shown in Figure 9.

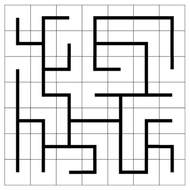


Figure 9: A maze generated by Wilson's algorithm

3 Verifying the puzzles

Recall that a puzzle is valid if there exists a unique solution. We convert our generated field to Boolean expressions and use a SAT solver to check for additional solutions.

3.1 Boolean satisfiability problem

A *Boolean expression* consists of variables and the operators AND ($\wedge$), OR ($\vee$) and NOT ($\neg$). The *Boolean satisfiability problem* (SAT) is the problem of finding whether there exists an assignment for the variables which satisfies the expression. A SAT solver is a tool to determine whether the expression can be satisfied, and gives a possible assignment for the variables. For our experiments, we have used the solver Lingeling[2].

Lingeling requires the input to be in *Conjunctive Normal Form* (CNF). A Boolean expression is in CNF if it is a conjunction ($\wedge$) of clauses consisting of disjunctions ($\vee$). An example would be $(a \vee b) \wedge (b \vee \neg c \vee d) \wedge e$.

3.2 Converting puzzles into a Boolean expression

Let T be the set of tiles in a puzzle and let $D = \{n, e, s, w\}$ be the set of directions. For each tile t, let t_n, t_e, t_s and t_w be Boolean variables that denote the northern, eastern, southern and western directions respectively, which are `true` if and only if that side is open. Furthermore, we define $t_{d'}$ as the opposite side and t_{d+} as the next side in a clockwise orientation of a direction $d \in D$. We can view the field as a set of variables, where each direction has a unique number. An example is shown in Figure 10.

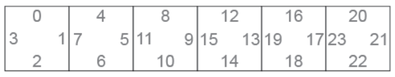


Figure 10: Labelling the directions of the tiles

For the actual puzzle, we add an additional boundary of empty tiles around it. We call this set of tiles T_{boundary}. This allows us to treat each tile in the same way. For a 2×2 board, we would consider the field from Figure 11. We can calculate the value of a tile's northern direction using the formula $4((m + 2)i + j)$, where i, j start from 1 and represent the row and column of the field without boundary

[2]`http://fmv.jku.at/lingeling/`

respectively, and m the width of the puzzle. We can add a value between 0 and 3 depending on which direction we are looking for.

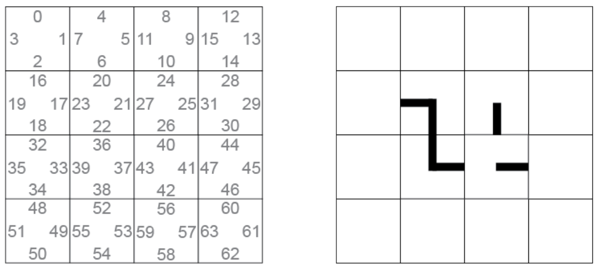


Figure 11: A field for a 2×2 puzzle

Given a maze, we want to generate a Boolean expression that is satisfiable if and only if the puzzle has a second solution. The formula will consist of the following parts:

- the tiles
- the field
- the solution
- a check for connectivity

3.2.1 The tiles

As we start with a candidate puzzle, we know which type of tile can be found at each position. We look at each tile separately, and add a clause to determine the different possible orientations of a tile. Let T_D, T_S, T_C and T_T denote the sets of D-tiles, S-tiles, C-tiles and T-tiles respectively for the given puzzle.

We construct the XOR(x, y) operator as $(x \vee y) \wedge (\neg x \vee \neg y)$ and the EQUAL(x, y) operator as $(x \vee \neg y) \wedge (\neg x \vee y)$, both of which are in CNF. These operators can be used in the expressions for the tiles.

D-tile The dead-end tile has one open side. We can say that for each pair of sides at least one of them must be closed, which means that no two sides can be open. It also holds that at least one side must be open. Combining these requirements makes sure that we have exactly one open side.

$$\bigwedge_{t \in T_D,\, p,q \in D \mid p \neq q} (\neg t_p \vee \neg t_q) \wedge \bigvee_{p \in D} t_p$$

S-tile The straight tile has two open sides opposite of each other. This is the same as saying that for each pair of adjacent sides, exactly one must be open.

$$\bigwedge_{t \in T_S,\, p \in D} \mathrm{XOR}(t_p, t_{p+})$$

C-tile The corner tile has two open sides adjacent to each other. We can simply say that exactly one of each pair of the opposite sides should be open.

$$\bigwedge_{t \in T_C,\, p \in \{n,e\}} \mathrm{XOR}(t_p, t_{p'})$$

T-tile The T-tile has one closed side. This is the opposite of the D-tile, which has one open side. We can construct the expression in the same way, and replace all open sides with closed sides and the other way around.

$$\bigwedge_{t \in T_T,\, p,q \in D \mid p \neq q} (t_p \vee t_q) \wedge \bigvee_{p \in D} \neg t_p$$

3.2.2 The field

The next step is to add information about the field. This means that we set all variables on the edge of the field to `false`, as they do not actually exist in the puzzle, but are merely added so we can treat each tile equally. The SAT solver Lingeling does not allow a variable named x_0, but since it is always positioned at a boundary tile, we can skip it in the actual implementation. We get

$$\bigwedge_{t \in T_{\text{boundary}},\, d \in D} \neg t_d$$

Now we need to treat adjacent variables as the same, since open sides need to touch each other. It is sufficient to say that all eastern and southern neighbours of a tile are equal. Per symmetry, this also gives all northern and western neighbours. Let E be the eastern neighbour of t and S be the southern neighbour of t.

$$\bigwedge_{t \in T} \mathrm{EQUAL}(t_e, E_w) \wedge \mathrm{EQUAL}(t_s \wedge S_n)$$

We can easily calculate which variables should be the same. The eastern neighbour of variable k would be variable $k + 6$, as we add another another tile and a half. The southern neighbour depends on the width m of the field, and has index $k + 4(m + 2) - 2$.

3.2.3 The solution

We started with a solution, and want to verify whether it is the only one. When looking for a possible second solution, we can say that at least one of the sides that is currently open should be closed.

$$\bigvee_{t \in T,\ d \in D \mid t \text{ open at } d \text{ in field}} \neg t_d$$

3.2.4 Connectivity

So far, we have a formula which finds a safe state. The next step is to verify that all tiles are connected without cycles. Recall that a solution has the structure of a tree placed on a grid, which means that the number of connections is one less than the number of tiles. We start from the same field, and thus, the number of connections between tiles is the same for any second solution. Thus, if all tiles are connected, the number of connections is still one less than the number of tiles, and we have found a new tree. We conclude that it is sufficient to check whether all tiles are reachable in the second solution.

We used two different methods to check for connectivity. The first method is to first find an assignment to the formula so far in order to find a safe state. Afterwards, we can verify the assignment by checking for disconnectivity, which corresponds with a simpler formula than connectivity. If we cannot find an assignment, we know that the puzzle must be connected. This method relies on the idea that second solutions are very rare. However, larger puzzles tend to have many different safe states, which are often not all connected. This is why we used a different formula for most experiments, more suited for a SAT solver. The second method is to perform a depth-first tree traversal [5]. We will discuss both in this section.

Disconnectivity For the first method, we need the second assignment and the dimensions of the puzzle to determine whether it is a connected second solution. Each tile will get its own variable. We start by assigning `true` to the first tile t_0. Then we say that for each pair of connected tiles, they must be assigned the same Boolean value. We do this using the EQUAL operator. The field is restricted to a safe state by the clauses above, and thus it holds that all open sides are connected to open sides. Lastly, we check that at least one tile can be assigned with `false`, meaning that it is not connected to the first tile. Note that we are now considering tiles only, and disregarding in which direction the tiles are connected. Once again, we only need to look at the eastern and southern neighbours to consider the whole field. We use the same notation as before and let E be the eastern neighbour of t

and S be the southern neighbour of t.

$$\begin{aligned}
&t_0 \\
&\wedge \bigwedge_{t\in T \mid t_e \text{ is open}} \text{EQUAL}(t_e, E_w) \bigwedge_{t\in T \mid t_s \text{ is open}} \text{EQUAL}(t_s, S_n) \\
&\wedge \bigvee_{t\in T \mid t\neq t_0} \neg t
\end{aligned}$$

We can give each tile a unique integer K from 1 up to and including $n \cdot m$, excluding the boundary. This would be equal to $(i-1)m+j$, but we do not know the values of i and j, because the input consists of an assignment of variables. Let k be a variable which denotes a variable of the field (e.g. a direction of one of the tiles). We use following formula to calculate the value of K without needing i and j:

$$K = (k/4/(m+2)-1)\cdot m + k/4 \bmod (m+2)$$

Depth-first tree traversal In the second method, we include connectivity in the formula itself by performing a depth-first tree traversal. Note that we cannot copy the approach of [5] directly, as it is applied to a given graph. In our field, the graph depends on the orientations of the tiles as chosen by the variable assignment, which makes the edge relation between the neighbouring tiles conditional. We first choose a starting tile, and check whether each tile of the field is reachable in a certain amount of steps, specified by a bound b. Let $v_{K,\ell}$ indicate whether tile K is reachable in at most ℓ steps. We choose a centre tile t_c as starting tile and say that it is reachable in 0 steps. All other tiles need at least one step, or are unreachable.

$$v_{t_c,0} \wedge \bigwedge_{t\in T \, | t\neq t_c} \neg v_{t,0}$$

We construct an implication $x \Rightarrow y$ as $(\neg x \vee y)$ in CNF. A tile is reachable in $\ell+1$ steps if and only if it is reachable in ℓ steps or if one of the neighbours is connected and reachable in ℓ steps. We already have variables t_d and $v_{K,\ell}$ to indicate this respectively. It is not possible to add this conjunction in CNF, as this would be of the form $\neg a \vee b \vee (c \wedge d)$. However, we can introduce a new variable $z_{t_d,\ell}$ which is true if and only if side t_d contains a pipe and that neighbour tile is reachable in ℓ steps. The following expressions ensures that $v_{t,\ell+1}$ is true if and only if the tile is reachable in $\ell+1$ steps:

$$\bigwedge_{t\in T} \bigwedge_{l=1}^{b-1} (\neg v_{t,l+1} \vee v_{t,l} \bigvee_{d\in D} z_{t_d,l})$$

Lastly, we need to add implications to ensure that $z_{t_d,l}$ is true only if neighbour t_d is connected and reachable in l steps. Let u be the neighbour connected to variable

t_d.

$$\bigwedge_{t_d \in T \mid d \in D} \bigwedge_{l=1}^{b-1} (z_{t_d,l} \Rightarrow t_d) \wedge (z_{t_d,l} \Rightarrow v_{u,l-1})$$

Combined, these additional clauses are satisfiable if and only if the new assignment creates a connected solution.

For the SAT solver we need to create a new set of variables for $v_{K,\ell}$. We continue indexing from the highest variable previously used. For readability, we will discard that term in the next explanation. K is the same variable as defined previously, which gives each tile a unique number. We assign the first b variables to the first tile, the next b to the second, etc. We calculate the number of the variable using the formula

$$v_{K,\ell} = (K-1)b + \ell$$

The variables $z_{t_d,l}$ will be numbered after all $v_{K,l}$. The value for the north neighbour can be calculated using

$$z_{t_n,l} = 4b((i-1)m + j - 1)$$

as each tile has four sides, and we need a variable for each possible bound for all tiles. We add 0 to 3 depending on the direction.

We can find the value of $v_{u,l}$ similar to how we found the indices of the neighbours in the first method. Instead of adding 1 or m for horizonal and vertical neighbours respectively, we add or subtract b and $b \cdot m$.

3.3 DIMACS format

Lingeling requires the input to be in DIMACS format. This is a specified format to represent the Boolean expression [2]. There are three different types of lines.

Comment line This is a line to provide additional information that will be ignored by the program. A lowercase `c` denotes the beginning of a comment line.

```
c This is a comment line
```

Problem line This is a line which defines the problem. Each input has exactly one problem line, which must appear before any clauses. A lowercase `p` denotes the beginning of a problem line.

```
p FORMAT VARIABLES CLAUSES
```

The `FORMAT` field refers to the expected format, which in this case will be `cnf`. The `VARIABLES` field describes the number of variables used and the `CLAUSES` field contains the number of clause lines that the input consists of.

Clauses These are the lines which describe the Boolean expression we want to solve. Each line contains a single clause. The variables are represented as numbers, so x_1 will appear as 1. The negation $\neg x_1$ is denoted as -1. Each line will contain a disjunction, where the variables are separated by spaces, and the end of the line is marked with a 0. Different lines are interpreted as conjunctions.

For example, we can take the formula $(x_1 \vee x_2) \wedge \neg x_2 \wedge (x_1 \vee x_3 \vee \neg x_4)$. This would give the following input in DIMACS form:

```
c A small example
p cnf 4 3
1 2 0
-2 0
1 3 -4 0
```

The puzzle from Figure 11 can be described using the following lines:

```
p cnf 139 174      -53 0              -67 66 89 90 0
c The tiles        -54 0              -89 70 0
-1 0               -55 0              -89 21 0
-2 0               -56 0              -90 74 0
-3 0               -57 0              -90 22 0
-4 0               -58 0              67 0
-5 0               -59 0              68 0
-6 0               -60 0              -69 68 98 99 0
-7 0               -61 0              -98 76 0
-8 0               -62 0              -98 26 0
-9 0               -63 0              -99 64 0
-10 0              c The field        -99 27 0
-11 0              1 -7 0             -70 69 102 103 0
-12 0              -1 7 0             -102 77 0
-13 0              2 -16 0            -102 26 0
-14 0              -2 16 0            -103 65 0
-15 0              5 -11 0            -103 27 0
-16 0              -5 11 0            -71 70 106 107 0
-17 0              6 -20 0            -106 78 0
-18 0              -6 20 0            -106 26 0
-19 0              9 -15 0            -107 66 0
```

```
20 22 0
-20 -22 0
21 23 0
-21 -23 0
-24 -25 0
-24 -26 0
-24 -27 0
-25 -26 0
-25 -27 0
-26 -27 0
24 25 26 27 0
-28 0
-29 0
-30 0
-31 0
-32 0
-33 0
-34 0
-35 0
36 38 0
-36 -38 0
37 39 0
-37 -39 0
-40 -41 0
-40 -42 0
-40 -43 0
-41 -42 0
-41 -43 0
-42 -43 0
40 41 42 43 0
-44 0
-45 0
-46 0
-47 0
-48 0
-49 0
-50 0
-51 0
-52 0
```

```
-9 15 0
10 -24 0
-10 24 0
17 -23 0
-17 23 0
18 -32 0
-18 32 0
21 -27 0
-21 27 0
22 -36 0
-22 36 0
25 -31 0
-25 31 0
26 -40 0
-26 40 0
33 -39 0
-33 39 0
34 -48 0
-34 48 0
37 -43 0
-37 43 0
38 -52 0
-38 52 0
41 -47 0
-41 47 0
42 -56 0
-42 56 0
c Connectivy
64 0
-65 64 81 82 0
-81 68 0
-81 21 0
-82 72 0
-82 22 0
-66 65 85 86 0
-85 69 0
-85 21 0
-86 73 0
-86 22 0
```

```
-107 27 0
71 0
-72 0
-73 72 112 113 0
-112 64 0
-112 36 0
-113 76 0
-113 37 0
-74 73 116 117 0
-116 65 0
-116 36 0
-117 77 0
-117 37 0
-75 74 120 121 0
-120 66 0
-120 36 0
-121 78 0
-121 37 0
75 0
76 0
-77 76 128 131 0
-128 68 0
-128 40 0
-131 72 0
-131 43 0
-78 77 132 135 0
-132 69 0
-132 40 0
-135 73 0
-135 43 0
-79 78 136 139 0
-136 70 0
-136 40 0
-139 74 0
-139 43 0
79 0
c Current solution
-20 -21 -24 -36 -37 -40 0
```

4 Code

We wrote a program in C++ to carry out the experiments. It was used to generate mazes and analyse these candidates as puzzles.

The field is implemented as a two-dimensional integer array. Each type of tile has its own number. A simple data structure consisting of four Booleans is associated with each tile number, to indicate whether a side is open, and thus represents the type of tile.

4.1 Functionality

The program is mainly used to generate mazes using different algorithms, and compute its corresponding SAT formula to check for a second solution. It is also possible to read an input file containing tile numbers instead.

The second functionality is to read SAT assignments for a certain field. Based on the dimensions, which are derived from the number of variables, it is possible to print the puzzle created by the assignment on a field, or write a new SAT formula to check for connectivity.

There is also a visual implementation. This allows the user to play the game using the mouse. Clicking a tile will rotate it, and the right mouse button can be used to lock a tile, preventing it from rotating.

4.2 Implementation

The largest part of the code runs in the terminal, and functions to carry out experiments. It is not intended to be user-friendly or playable, but to aid in the experiments. It has the functionality described in the previous section, such as generating algorithms and writing SAT formulas.

The experiments themselves were carried out using simple bash scripts. These used the SAT solver and counted the number of valid puzzles. In case connectivity is checked separately, it keeps track of the different number of configurations found before the puzzle is marked as valid or invalid.

The other part of the program is the visual game, which is written using `glut`. This allows us to create a new window and to draw a field with tiles. The library includes functions to rotate parts, which is useful for the tiles. The flood of water flowing from the water source is visualised. Figure 1 shows the result.

5 Results

We have used our C++ program to generate mazes and a SAT solver to check whether it would be a valid puzzle with a unique solution. We will analyse square puzzles with dimensions 5, 10, 15 and 20. Let N denote the sample size for an experiment.

5.1 Multiple safe states

At first, we thought that second solutions were rare, and expected puzzles to have at most a few possible safe states. Thus, we only looked at the orientations of the tiles to find possible second solutions and carried out a separate check to see if it was connected. We quickly found this expectation to be incorrect, and encountered issues for larger puzzles. Table 1 shows the number of safe states tested before a conclusion was reached, meaning that a second valid solution was found, or that there were no valid safe states left. If the number of safe states was larger than or equal to 200, we used the value 200 to calculate a lower bound for the average. These values are increasing rapidly for larger mazes, with the exception of the Binary Tree algorithm. This makes it an illogical choice to use this method, and thus we chose to use a depth-first tree traversal to verify whether all tiles are connected for the other algorithms.

		Binary Tree	**Depth-first**	**Kruskal**	**Prim**	**Wilson**
5	avg	1	1,56	1,09	1,5	1,12
	max	1	4	3	6	3
	valid	100	67	100	95	95
	≥ 200	0	0	0	0	0
10	avg	1	11,39	2,81	9,37	2,19
	max	1	≥ 200	10	100	15
	valid	100	27	66	41	65
	≥ 200	0	1	0	0	0
15	avg	1	79,79	13,44	62,74	9,61
	max	1	≥ 200	99	≥ 200	≥ 200
	valid	100	3	35	15	36
	≥ 200	0	26	2	17	1
20	avg	1	159,46	54,34	137,12	47,95
	max	1	≥ 200	≥ 200	≥ 200	≥ 200
	valid	100	0	7	3	11
	≥ 200	0	75	16	54	11

Table 1: Number of different safe states to satisfy the tiles ($N = 100$)

We found that these additional ways to connect the pipes are mainly caused by T-tiles and D-tiles. For most tiles, two sides are already determined by the orientation of the neighbours. For T-tiles, this leaves two other orientations. Combined with D-tiles, which are just as flexible, this creates many possibilities. Most of them will not connect all tiles and instead create cycles, which makes them configurations. Figure 12 shows an example of a small puzzle with five possible ways to rotate the pipes.

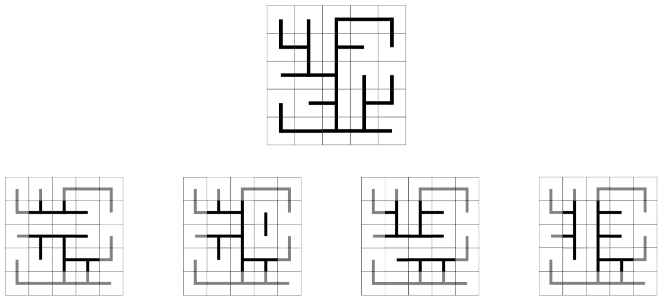
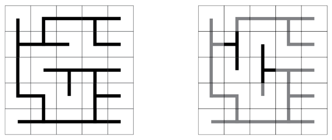
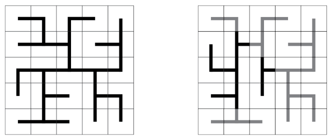


Figure 12: A 5×5 puzzle with five different safe states

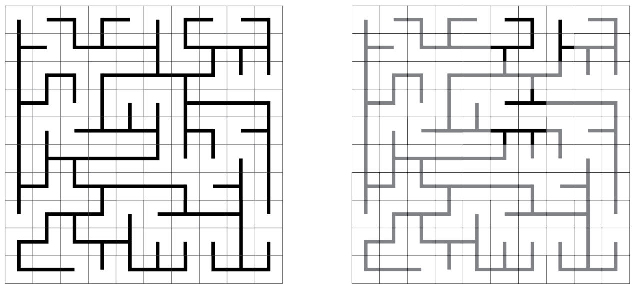
We also notice that second solutions are usually caused locally by a small group of tiles. In some cases, only four tiles are needed. Because we see no large global changes, the structure of the second solution is very similar to the original solution. Figure 13 shows some examples.



(a) Depth-first



(b) Prim



(c) Wilson

Figure 13: A second solution of different mazes obtained by small local changes

5.2 The bound for tree traversal

After finding out that fields can have hundreds of solutions, we decided to check for connectivity using a tree traversal instead. This formula asks for a bound b, which is the maximum distance from the starting tile to any other tile. A safe bound would be $n \cdot m$, as this covers the whole field. Unfortunately, this creates an enourmous number of variables and clauses, especially for larger fields, and takes much time to compute, which is not practical for experiments with many candidate fields. Table 2 shows the number of variables, clauses and computation times with a bound of n^2.

		Binary Tree	**Depth-first**	**Kruskal**	**Prim**	**Wilson**
5	variables	3,316	3,316	3,316	3,316	3,316
	clauses	4,901	4,902	4,891	4,895	4,892
	time	0.02 s	0.01 s	$<$ 0.01s	0.01 s	$<$ 0.01 s
10	variables	50,571	50,571	50,571	50,571	50,571
	clauses	82,728	82,726	82,692	82,727	82,692
	time	0.02 s	0.11 s	0.08 s	0.16 s	0.28 s
15	variables	254,276	254,276	254,276	254,276	254,276
	clauses	429,991	429,995	429,915	430,000	429,921
	time	0.92 s	1.49 s	1.48 s	1.7 s	1.75 s
20	variables	801,931	801,931	801,931	801,931	801,931
	clauses	1,378,195	1,378,203	1,378,068	1,378,220	1,378,064
	time	3.4 s	5.32 s	4.36 s	4.82 s	4.61 s

Table 2: Average number variables, clauses and time per puzzle with a bound of $b = n^2$ ($N = 100$)

Thus, it was necessary to find a feasible bound smaller than n^2. We started by generating all our mazes and calculating what the average and maximum bound was for each size and algorithm. The bounds are listed in Table 3.

In most cases, these are a little over $\frac{1}{2}n^2$. Table 4 shows the variables, clauses and computation times with this bound. We do not know beforehand what a different solution will look like, which means that the bound is not necessarily the same. However, we expect it to have a similar structure, as explained in the previous section. For each puzzle, we add 2, 5, 7 and 10 to the maximum of puzzles of size 5, 10, 15 and 20 respectively and use that as bound. The exception is Prim's algorithm. We add 5 instead of 2 to puzzles of size 5, to have some additional space for unexpected outcomes. For the other puzzles, the maximum is already so close to the theoretical maximum of $n^2 = 25$ that we expect that it is safe to add only

		Depth-first	**Kruskal**	**Prim**	**Wilson**
5	avg	15.64	9.51	5.60	9.91
	max	18	20	11	21
	bound	20	22	16	23
10	avg	47.44	24.20	11.71	26.42
	max	62	54	21	57
	bound	67	59	26	62
15	avg	93.49	40.36	17.13	44.41
	max	123	87	30	100
	bound	130	94	37	107
20	avg	152.55	57.60	22.94	64.39
	max	203	126	35	137
	bound	213	136	45	147

Table 3: Maximum, average and observed bounds of generated mazes ($N = 5000$)

2. Note that we cannot guarantee that all possible second solutions fall within the chosen bound, but we believe this chance to be small enough to be insignificant.

5.3 Number of valid puzzles

Table 5 lists the number of valid puzzles generated by the algorithms, using the bounds from the previous section. The Binary Tree algorithm is tested using the first method for connectivity.

We see that Binary Tree is the algorithm that generates the most valid puzzles. More interesting, it is the only algorithm which generates valid puzzles only. We notice that the performance of the other algorithms decreases heavily as the size of the puzzle increases. This makes sense when we recall that a second solution can originate from few as four tiles. The algorithms of Kruskal and Wilson generate the highest number of valid puzzles after Binary Tree.

5.4 Tiles and texture

Table 6 shows the average number of tiles for each type used in a field.

We notice that the BinaryTree algorithm used a large amount of S-tiles, while for all other algorithms, this tile is used least. Furthermore, for each maze, the number of D-tiles is exactly two more than the number of T-tiles. This is caused by the tree structure.

		Binary Tree	**Depth-first**	**Kruskal**	**Prim**	**Wilson**
5	variables	1,691	1,691	1,691	1,691	1,691
	clauses	2,496	2,497	2,486	2,490	2,487
	time	0.06 s	0.1 s	0.09 s	0.1 s	0.09 s
10	variables	25,571	25,571	25,571	25,571	25,571
	clauses	41,728	41,726	41,692	41,727	41,692
	time	0.31 s	0.3 s	0.24 s	0.21 s	0.16 s
15	variables	25,571	25,571	25,571	25,571	25,571
	clauses	214,726	214,730	214,650	214,735	214,656
	time	0.32 s	0.69 s	0.63 s	0.69 s	0.57 s
20	variables	401,931	401,931	401,931	401,931	401,931
	clauses	690,195	690,203	690,068	690,220	690,064
	time	1.13 s	2.05 s	1.97 s	2.27 s	2.05 s

Table 4: Average variables, clauses and time per puzzle with a bound of $\frac{1}{2}n^2$ ($N = 100$)

	Binary Tree	**Depth-first**	**Kruskal**	**Prim**	**Wilson**
5	5000	3067	4480	3490	4613
10	5000	393	1701	524	2054
15	5000	4	247	34	426
20	5000	0	13	1	48

Table 5: Valid puzzles ($N = 5000$)

We know that the Binary Tree algorithm has a diagonal bias and creates mazes with a very specific texture. Nearly all tiles occur in only two different orientations, opposed to the expected four, and we have paths in only two directions.

The algorithms Depth-first and Prim work from the centre. As a result, the power source is nearly always a T-tile. During the execution of the algorithms, a maze is slowly build. Mazes generated by the Depth-first algorithm tend to have long paths, while Prim's algorithms creates the shortest, as seen in the bounds for tree traversal.

Thus, Kruskal and Wilson seem to be making the most random textures. Contrary to Depth-first and Prim, these algorithms build small parts of the maze until it is complete. It makes sure that adding an edge will not violate the properties of a spanning tree, but during the execution of the algorithm, added edges do not need to create a valid tree.

		Binary Tree	**Depth-first**	**Kruskal**	**Prim**	**Wilson**
5	D-tile	7.25	9.08	7.8	9.76	7.56
	S-tile	6.8	4.86	4.24	2.64	4.58
	C-tile	5.7	3.97	7.16	4.84	7.29
	T-tile	5.25	7.08	5.8	7.76	5.56
10	D-tile	25.98	39.73	28.76	35.56	27.93
	S-tile	28.07	11.04	17.21	15.59	18.5
	C-tile	21.97	11.51	27.27	15.3	27.63
	T-tile	23.98	37.73	26.76	33.56	25.93
15	D-tile	57.28	91.43	63.88	78.87	62.17
	S-tile	61.64	20.18	37.96	36.92	40.5
	C-tile	50.81	23.97	61.28	32.33	62.17
	T-tile	55.28	89.43	61.88	76.87	60.17
20	D-tile	101.04	164.37	113.49	139.78	110.17
	S-tile	108.07	31.73	66.56	66.36	71.12
	C-tile	91.84	41.52	108.47	56.08	110.55
	T-tile	99.04	162.37	111.49	137.78	108.17

Table 6: Average tile distribution ($N = 5000$)

Table 7 contains the average number of tiles of each type found in original puzzles on the website [1]. We did not find a way to efficiently retrieve data about the fields, so for each size, five puzzles have been counted manually.

	D-tile	**S-tile**	**C-tile**	**T-tile**
5	8.2	2.2	8.4	6.2
10	33.2	16.4	19.2	31.2
15	69.4	37.6	50.6	67.4
20	119.4	65.4	97.8	117.4

Table 7: Average tile distribution from original puzzles ($N = 5$)

When we compare those averages to the tile distributions for the mazes generated for this thesis, we find that the Binary Tree algorithm has a high number of S-tiles, and less D-tiles and T-tiles compared to original puzzles. Both Kruskal's and Wilson's algorithm generate puzzles with a high number of C-tiles. The Depth-first and Prim's algorithm creates mazes with more D-tiles and T-tiles, at the expense of C-tiles.

Similar to the texture bias, we find that we have Kruskal and Wilson, who share the same properties, and Depth-first and Prim, who share other properties. The Binary Tree algorithm does not share major similarities with the other algorithms.

6 Conclusion

In this thesis, we analysed five different maze-generating algorithms to create Pipes puzzles: Binary Tree, Depth-first, Kruskal, Prim and Wilson. All candidate fields were verified using a SAT solver to check that no other solutions exist. In general, we found Binary Tree deviates the most from the other algorithms. Furthermore, Depth-first and Prim create similar results, and so do Kruskal and Wilson.

Valid puzzles
The Binary Tree algorithm is the only algorithm that generates valid puzzles only. The performance of all other algorithms decreases heavily as the size of the puzzle increases. At a size of 20 or larger, our current method to generate puzzles is not suitable anymore, and only few valid puzzles are generated. The algorithms of Kruskal and Wilson generate the highest number of valid puzzles after Binary Tree.

Texture bias
The Binary Tree algorithm generated puzzles with a very specific structure. Half the orientations of tiles never occur in the maze, with the exception of the top-left piece. The algorithms Depth-first and Prim both work from the centre. Depth-first generates mazes with long paths. Kruskal and Wilson seem to make the most random textures, where no clear bias can be found.

Tile distribution
We compared the tile distribution of our generated puzzles to original puzzles. The binary tree algorithm has a high number of S-tiles, and less D-tiles and T-tiles than original puzzles. The algorithms of Kruskal and Wilson generate puzzles with a high number of C-tiles. Depth-first and Prim's algorithm create mazes with more D-tiles and T-tiles, at the expense of C-tiles.

Difficulty
A basic strategy for solving this puzzle is to start from the boundary and work towards the centre. For example, an S-tile at the boundary only has one correct orientation. In puzzles with multiple safe states, it can be necessary to guess the orientations of certain tiles in order to finish the puzzle. We expect these puzzles to be more difficult. Binary Tree generates puzzles only a single safe state, and is expected to make the easiest puzzles, whereas Depth-first and Prim generate puzzles with most safe states, which would make them the most difficult to solve.

6.1 Further research

Unfortunately, the number of valid puzzles generated for larger puzzles is extremely low. The research could be expanded in order to find a more effective method to generate puzzles.

The most logical next step would be to improve the maze-generating algorithms. We notice that the performance is really bad for larger puzzles. It will be interesting to either try new algorithms or alter the algorithms to prevent second solutions. It would be helpful if we can find a pattern in the puzzles with multiple solutions, so we can avoid them during the generation.

During the experiments, we were limited by the computation times. We could look for a better SAT solver, or improve the satisfiability formula in order to test even more puzzles or larger dimensions. One way to improve the formula is by reducing the number of variables using a technique called bit-blasting. Rather than having a variable for each possible distance to the starting tile, we can encode the binary representation of that distance instead, and include the binary comparison as clauses into the formula [6]. In our current formula for connectivity, we did not use the fact that a maze is a graph embedded on a grid. Using this structural restriction, it might be possible to find a simpler formula. An idea is to traverse the graph and give each edge a number corresponding with the direction. This way, it might be possible to find cycles more easily.

Lastly, it is possible to place the puzzles in different difficulty classes. This can for instance be done by counting how many times all tiles need to be rotated in order to solve the puzzle using different solving algorithms.

7 References

[1] Pipes - online puzzle game. URL: `https://www.puzzle-pipes.com`.

[2] Satisfiability suggested format.

[3] Marzio De Biasi. The complexity of the puzzle game net: rotating wires can drive you crazy. December 2012. URL: `https://www.nearly42.org/cstheory/the-complexity-of-the-puzzle-game-net-rotating-wires-can-drive-you-crazy/` (version 31-5-2021).

[4] Jamis Buck. *Mazes for Programmers.* The Pragmatic Programmers, 2015.

[5] Yuval Filmus. Sat algorithm for determining if a graph is disjoint. Computer Science Stack Exchange. URL: `https://cs.stackexchange.com/q/111411` (version: 2019-07-02).

[6] D.W. (https://cs.stackexchange.com/users/755/d w). Sat algorithm for determining if a graph is disjoint. Computer Science Stack Exchange. URL: `https://cs.stackexchange.com/q/142450` (version: 2021-07-20).

[7] Ronald L. Rivest Clifford Stein Thomas H. Cormen, Charles E. Leiserson. *Introductions to Algorithms*, chapter 22. MIT Press, 2009.