Universiteit Leiden
The Netherlands

Opleiding Informatica

Generating and Solving
Sigmar's Garden Puzzles

Max Ruigrok

Supervisors:
Hendrik Jan Hoogeboom & Mark van den Bergh

BACHELOR THESIS

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

17/07/2025

Abstract

This research analyses the solitaire puzzle game Sigmar's Garden. First, we prove that Sigmar's Garden is $\mathcal{NP}$-complete. We then propose two methods of solving puzzle instances, namely by using a standard backtracking approach and by using a SAT-solver. Furthermore, we propose a novel way of generating puzzle instances in linear time for Sigmar's Garden and similar tile-matching puzzle games such as Mahjong Solitaire.

Contents

1 Introduction

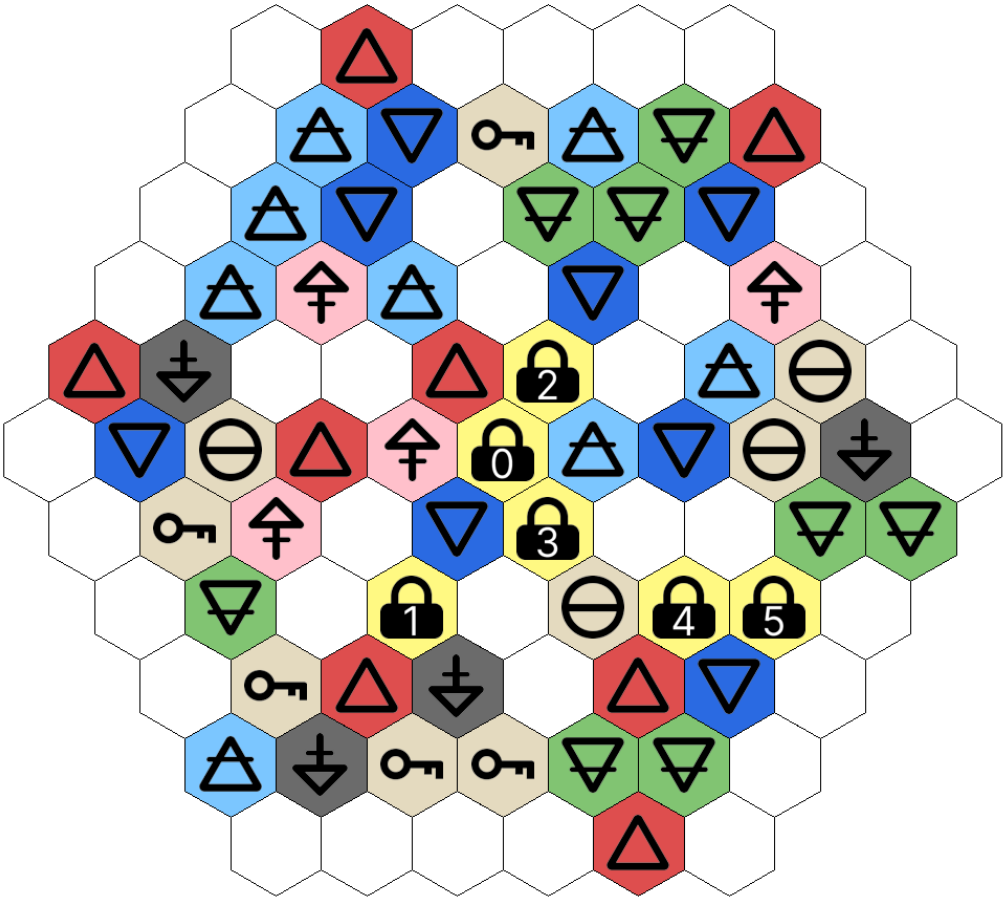


Figure 1: Instance of a Sigmar's Garden puzzle, in a "pinwheel"-formation.

Solving puzzles is a common pastime. Many games with simple rules can give rise to surprisingly deep strategic and computational complexity. This thesis investigates such complexity in the context of Sigmar's Garden, a solitaire puzzle derived from the game Opus Magnum by Zachtronics [Zac].

The main premise of the game is to remove tiles from the board in pairs. While the rules are relatively easy to grasp, the vast number of valid moves at any given state of a puzzle causes an extremely large combinatorial space, making the solving and generation of these puzzles computationally demanding.

The goal of this thesis is to answer the following research question, composed of three parts:

1. Is Sigmar's Garden $\mathcal{NP}$-complete?
2. How can Sigmar's Garden puzzles be solved efficiently?
3. How can Sigmar's Garden puzzles be generated with a guaranteed solution?

The first part of the research question is answered by reducing Bounded NCL to Sigmar's Garden, a proven $\mathcal{NP}$-complete problem. By performing this reduction, Sigmar's Garden is proven to be $\mathcal{NP}$-complete.

The second part of the research question is answered by designing two approaches aiming to solve a puzzle instance: using an algorithmic approach (backtracking in combination with dynamic programming) and a logical approach (converting the puzzle to CNF and solving it with a SAT-solver.

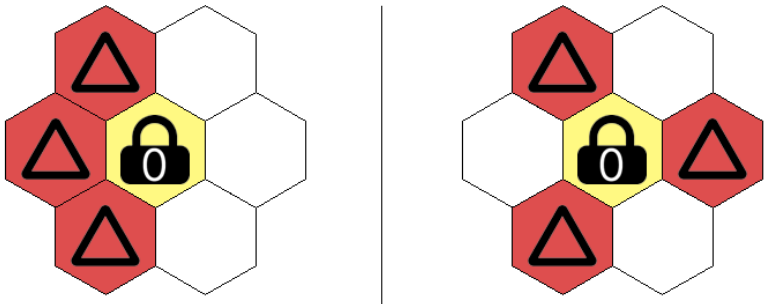
The third and final part of the research question is answered by first researching existing implementations of the game, while also providing a novel algorithm for generating not only Sigmar's Garden puzzles, but tile-matching puzzles in general.

This bachelor thesis was conducted at LIACS under the supervision of Hendrik Jan Hoogeboom and Mark van den Bergh.

1.1 Thesis overview

Section 2 contains an explanation of Sigmar's Garden and terms that reoccur throughout the thesis. Section 3 covers some papers related to this thesis. Section 4 proves that Sigmar's Garden is $\mathcal{NP}$-Complete. Section 5 briefly goes over the process of creating a program to work with Sigmar's Garden puzzles. Section 6 and 7 describe methods of solving and generating Sigmar's Garden puzzles respectively. Section 8 presents a quantitative analysis on the methods introduced in Section 6 and 7. Finally, Section 9 gives a summary of the thesis and mentioning possible follow-up research.

2 Sigmar's Garden



(a) Center tile removable. (b) Center tile not removable.

Figure 2: Example of tile availability. Tiles may only be removed if it has three contiguous free neighbors.

Sigmar's Garden is a solitaire puzzle game that originated from Opus Magnum, an alchemy-inspired factory automation game developed by Zachtronics [Zac]. Sigmar's Garden features a honeycomb-shaped board with 11 cells on the diagonals. Each cell on the board is either empty or contains a tile. An instance of the puzzle can be seen in Figure 1.

The puzzle is considered solved if all tiles are removed from the board. This is done by repeatedly removing tiles from the board in pairs, which in turn frees up new tiles. A tile is considered removable if it has three contiguous neighboring cells that are empty. Figure 2a shows a position where the center tile is removable in its current state, while Figure 2b shows a position where the center tile is not available, since it does not have three contiguous free neighbors. This requires one of the three surrounding tiles to be removed before the center tile becomes removable.

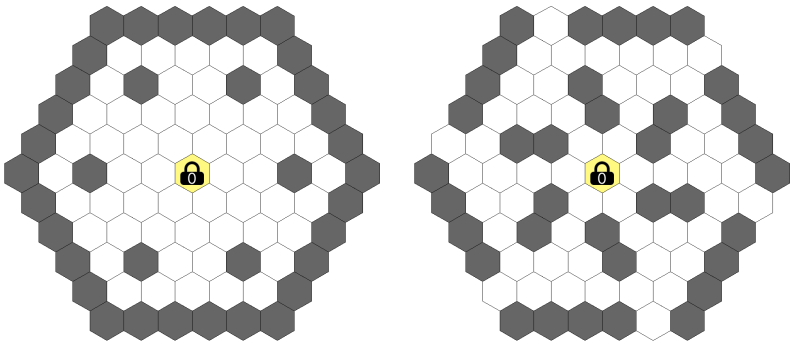
The types of tiles and the number of times they occur is as follows:

- **Cardinal elements**: we define the number of cardinal elements as N_C. For most of the thesis, N_C is fixed to four cardinal elements, which we assign names to. With $N_C = 4$, the number of tiles per cardinal element is n_1 tiles of air, n_2 tiles of earth, n_3 tiles of fire and n_4 tiles of water (represented by cyan, green, red and blue tiles respectively), where each n_i is even. These tiles form a pair with tiles of the same element. For a standard puzzle, for example the puzzle shown in Figure 1, $n_1 = n_2 = n_3 = n_4 = 8$.
- **Salt**: n_s tiles of salt (represented by Θ), where n_s is even. These form a pair with other salt tiles, or with one of the cardinal elements. Note that combining a salt with a cardinal element disrupts the parity; if one tile of an element is paired with a salt, an odd number of tiles of that element are left behind, so another cardinal element of the same type of tile must be matched with a salt in order to restore the parity. For a standard puzzle, $n_s = 4$.
- **Vitae and mors**: n_v tiles of vitae and n_v tiles of mors (represented by up and down arrows respectively). A vitae tile can only form a pair with a mors tile. For a standard puzzle, $n_v = 4$.

- **Metals**: n_ℓ tiles of metals, denoted by a lock with its index. A metal tile can only form a pair with a key. A metal with index k can only be removed if all metals with $index > k$ are removed. The last metal with an index of 0 does not form a pair with a key; it can be removed on its own if all other metals have been removed. From this point onward, metals will be referred to as locks, where lock-n indicates a lock with an index of n. For a standard puzzle, $n_\ell = 6$, including lock-0.
- **Quicksilver**: $n_\ell - 1$ tiles of quicksilver, denoted with a key. A key can only form a pair with a lock. Keys do not have an index; a key can form a pair with any lock. From this point onward, quicksilver will be referred to as a key.

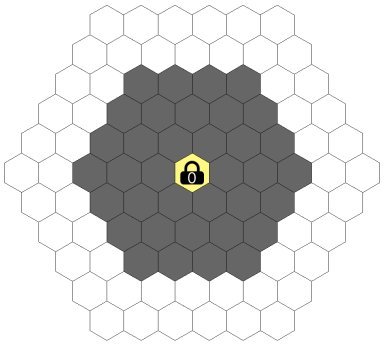
The remainder of this thesis contains keywords that have a relation to Sigmar's Garden in the context of this paper. The following list aims to disambiguate such keywords.

- **Move**: the act of removing a pair of tiles from the board. This in turn frees up neighboring tiles. Removing the final lock, which does not form a pair with another tile, is also considered a move.
- **Frontier**: all tiles that are removable as of the current board state, according to the availability rules displayed in Figure 2. In the context of generating puzzles, the frontier consists of all cell positions where a tile may be placed. In both interpretations, the frontier also contains tiles that are not yet removable or placable due to other constraints, such as locks with a higher index still being present on the board.
- **Cell**: a position on the board that either contains a tile or is empty.
- **Template**: a grid with identical dimensions to the board that specifies for each cell whether a tile may be placed there. The position of lock-0 is also denoted in the template. A list of all templates mentioned in this thesis can be found in Figure 3.
- **Free**: a cell that contains no tile is called free. This means surrounding tiles are not obstructed by this cell.
- **Removable**: a tile is considered removable if the tile can be taken off the board in its current state, as described in Figure 2.

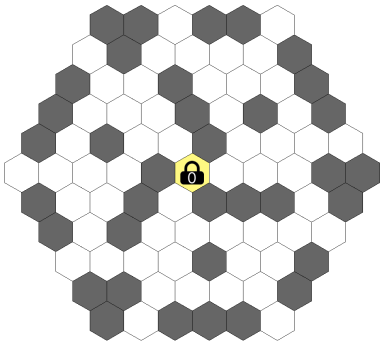


(a) Dense

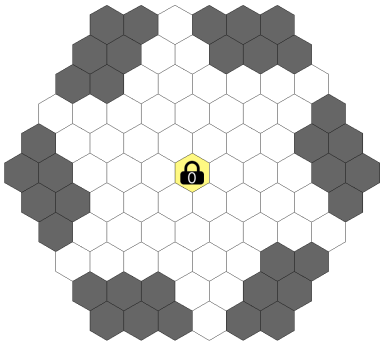
(b) Pinwheel



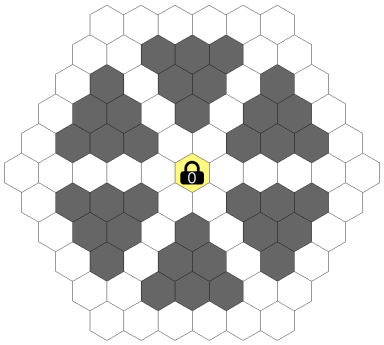
(c) Rings



(d) Sparse



(e) Star



(f) Wheel

Figure 3: Subset of the templates featured in Opus Magnum's Sigmar's Garden. Templates may appear as a mirrored version of itself in the game.

3 Related Work

While no prior research has been performed on Sigmar's Garden, similar puzzles have been of interest in academic research. Most notably, Mahjong Solitaire has been featured in research regarding complexity and solvability analysis.

Van Rijn conducted research on various puzzles including Mahjong Solitaire, and proved it is $\mathcal{NP}$-Complete by reducing it from Bounded NCL (Nondeterministic Constraint Logic). [vR12] The thesis was based on previous research that showed that if a puzzle can encode a set of predefined gadgets, it can be reduced from Bounded NCL and is therefore $\mathcal{NP}$-Complete [HD09]. This was done by defining Mahjong Solitaire as containing disjoint tile sets, where each tile set contains an even number of tiles. Furthermore, it is assumed to contain a finite but arbitrarily large number of tiles. These assumptions allow Mahjong Solitaire to encode the gadgets and is therefore shown to be $\mathcal{NP}$-Complete.
Section 4 proves Sigmar's Garden is $\mathcal{NP}$-Complete using a similar construction as the one described by Van Rijn.

In another thesis, Michiel de Bondt provided a general solving algorithm for Mahjong Solitaire, that improved upon previous solvers with several orders of magnitude [dB]. This was achieved through two main additions:

- **Pruning the search tree**: by performing an initial pruning scan; impossible positions are detected before any recursive backtracking is required.
- **Detecting critical groups**: instead of solving a position by removing tiles in pairs, this improvement solves positions in groups. It detects critical groups by selecting tiles and performing the aforementioned pruning scan. If successful, it is marked as being a critical group. This process is repeated until a solution is found. If all pruning scans fail, the board is deemed unsolvable.

The paper also touches upon alternative methods such as random solving. The amount of attempts needed to solve a puzzle could serve as a difficulty measure for that puzzle instance.

Finally, De Bondt analyses different Mahjong puzzle templates provided by the program used in the research. Interestingly, this research found that for many templates, only a small margin of puzzles generated using that template were solvable. This indicates that generating a Mahjong puzzle that is guaranteed to be solvable is non-trivial.

Sigmar's Garden has been of interest to programmers, prompting them to make open source implementations of the game. These implementations also provide generators for the puzzle. These are of particular interest in Section 7, where a closer look is taken at two open source implementations: [Dot] and [Hru].

4 Complexity

(a) CHOICE vertex. (b) AND vertex. (c) FANOUT vertex. (d) OR vertex.

Figure 4: Types of vertices in constraint graphs. Each vertex requires the sum of incoming edge weights to be greater or equal to 2.

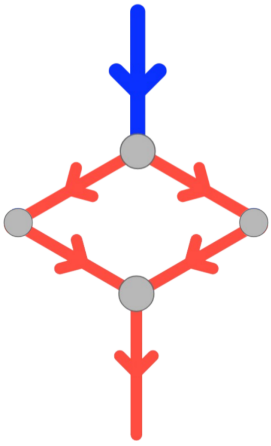
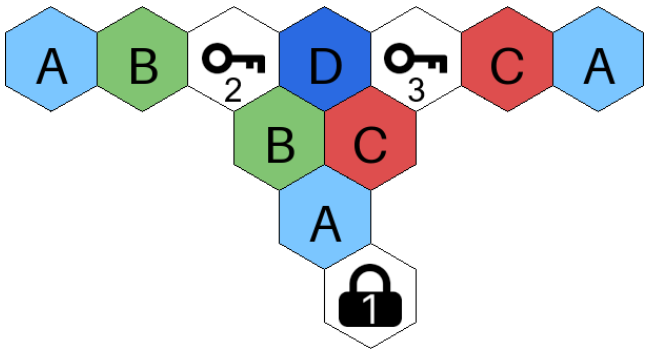


Figure 5: Example constraint graph that is non-reversible.

In "Games, Puzzles and Computation" [HD09], Hearn and Demaine prove that bounded nondeterministic constraint logic is $\mathcal{NP}$-complete by reducing it from 3-SAT. A problem instance consists of a connected graph with directed edges. A boolean formula in 3-SAT is satisfiable if and only if the formula reduced to Bounded NCL is reversible. A constraint graph is considered reversible if the so-called target edge is reversed. The reduction algorithm encodes the clauses in four types of vertices, each with different properties. These vertices are shown in Figure 4. A constraint graph consists of edges with weights of either 1 or 2, colored red and blue respectively. Each vertex requires the combined weights of the incoming edges to be greater or equal to 2. The bottom edge in the CHOICE vertex as seen in Figure 4a can be turned into an incoming edge, allowing one of the top edges to be converted into an outgoing edge while keeping the inflow equal to 2, effectively making a choice between the two outgoing edges. The AND vertex in Figure 4b requires both bottom edges to be turned into incoming edges, resulting in the top edge becoming an outgoing edge. The FANOUT vertex showing in Figure 4c converts the top edges into outgoing edges when the bottom edge is reversed. Finally, the OR vertex shown in Figure 4d needs only one outgoing edge to be reversed in order to turn the top edge into an outgoing edge.

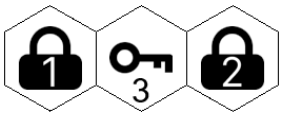
Figure 5 contains an example of a constraint graph. It consists of a CHOICE vertex, with both edges connecting to an AND gadget. The top blue edge is considered the target edge. This constraint



(a) CHOICE gadget.

(b) AND gadget.

(c) FANOUT gadget.



(d) OR gadget.

Figure 6: Bounded NCL vertices encoded as Sigmar's Garden gadgets. All tiles apart from the lock and key tiles are local to that gadget.

graph is non-reversible, as the CHOICE gadget only allows one of the incoming edges to be reversed, while the AND gadget requires both outgoing edges to be reversed.

The authors argue that, if a game or puzzle is able to represent these four vertices, any instance of Bounded NCL can be reduced to that puzzle, therefore making it $\mathcal{NP}$-complete [HD09]. They show this by reducing Bounded NCL to a variety of 1- or 2-player games, such as sliding-block puzzles, Rush Hour and Amazons. Van Rijn expanded upon this research in their thesis, by showing that Bounded NCL can be reduced to Klondike, Mahjong, Nonograms and a variety of other games, which are thereby proven to be $\mathcal{NP}$-complete [vR12].

In this section we prove that Sigmar's Garden is $\mathcal{NP}$-complete, using a similar proof structure as the Mahjong Solitaire proof in [vR12]. Van Rijn encoded the four vertices into Mahjong gadgets, where each gadget was locked by one or more locks. Depending on the gadget, unlocking these locks allows keys to be freed, consequently allowing other gadgets to be unlocked.

To make the proof work, we generalize the standard Sigmar's Garden game as having an arbitrarily large but finite set of disjoint tile sets $\mathcal{C}$, with $|\mathcal{C}| = n_C$. This represents the cardinal element tiles, which is typically set to four disjoint sets. Each disjoint set contains an even number of tiles. All tiles in a tile set can match with any other tile in that set. All other rules are unchanged relative to the definition in Section 2; tiles are removable if they have three or more contiguous free neighbors; a legal move consists of the removal of two tiles of the same tile set; and the puzzle is considered solved if all tiles have been removed from the board.

We show that Sigmar's Garden is $\mathcal{NP}$-complete by reducing Bounded NCL to Sigmar's Garden. Each vertex in a Bounded NCL graph can be replaced by the gadgets shown in Figure 6. In these gadgets, all tiles with the same color and letter are contained in the same disjoint tile set, and can therefore be paired together. It is important to note that the lock and key symbols in the gadgets

do not function in the same way as described in Section 2; they do not have an ordering in which they can be removed, they only symbolize a connection between gadgets.

Lemma 4.1 *The construction in Figure 6a satisfies the same constraints as a Bounded NCL CHOICE vertex, with the lock-1 corresponding to the input edge and the keys corresponding to the output edges.*

Proof. In order to get one of the keys available, the lock tile needs to be removed, otherwise nothing can be done with this gadget. Only the two removable A tiles can be paired, but after this, nothing can be done. When the lock tile is removed, the newly freed A tile can be paired with one of the outer A tiles. After this, depending on which A tile was removed, either the two B tiles or the two C tiles can be removed. This frees up one of the keys. After this, the other key cannot be removed as it is locked between the D tile and either a B or a C tile, depending on which pair of tiles was removed. □

Lemma 4.2 *The construction in Figure 6b satisfies the same constraints as a Bounded NCL AND vertex, with the locks corresponding to the input edges and the key corresponding to the output edge.*

Proof. In order to get the key tile available, both lock tiles need to be removed. If none of the locks are removed, nothing can be done with this gadget. If only one of the locks is removed, the A tile is freed up but it cannot be paired with the other A tile as it is locked behind the other lock tile. If both lock tiles are removed, both A tiles can be matched together, freeing up the key tile. □

Lemma 4.3 *The construction in Figure 6c satisfies the same constraints as a Bounded NCL FANOUT vertex, with the lock corresponding to the input edge and the two keys corresponding to the output edges.*

Proof. In order to get the key tiles available, the lock tile needs to be removed. As long as this lock is present, nothing can be done with this gadget. When the lock is removed, it frees up one of the A tiles, allowing it to be paired with the other A tile. This consequently frees up both key tiles. □

Lemma 4.4 *The construction in Figure 6d satisfies the same constraints as a Bounded NCL OR vertex, with the locks corresponding to the input edges and the key corresponding to the outgoing edge.*

Proof. In order to get the key tile available, either one of the lock tiles need to be removed. If none of the locks are removed, nothing can be done with this gadget. If only one of the locks is removed, the key becomes available. If both locks get removed, the key also becomes available. □

Theorem 4.5 *Sigmar's Garden is $\mathcal{NP}$-complete.*

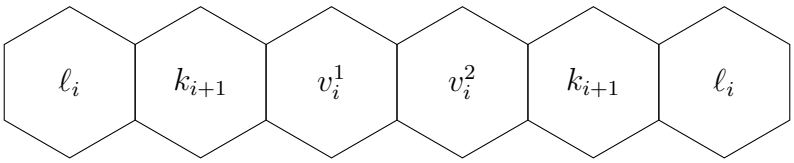


Figure 7: Sigmar's Garden victory gadget.

Proof. Reduction from Bounded NCL. Given a constraint graph made of CHOICE, AND, FANOUT and OR vertices, we construct a corresponding Sigmar's Garden arrangement using the gadgets shown in Figure 6. We define a set of victory tiles $\mathcal{V}$ that contains a lock for each key in the gadgets and a key for each lock in the gadgets. These victory tiles are distributed over victory gadgets, defined later. These gadgets are unlocked by reversing the victory vertex in the constraint graph. This allows all gadgets that were not (completely) removed, to be taken off the board anyway. Additionally, for every CHOICE gadget used, $\mathcal{V}$ contains an A tile and a D tile, restoring the parity of these tiles.

Each victory gadget is of the form shown in Figure 7, guarded by locks ℓ_i. In between are two k_{i+1}, which are keys that form a pair with the left and right locks of the next victory gadget. This chain of locks and keys continues throughout all victory gadget, except the last victory gadget which does not have r_{j+1} tiles, as there is no next victory gadget. The final key representing the victory edge in the constraint graph unlocks a FANOUT gadget, so that the two keys made available open up the first victory gadget. Each victory gadget contains two tiles in between the k_{i+1} tiles, namely the lock and key tiles corresponding to that victory gadget's vertex. A separate victory gadget exists for each CHOICE gadget, that contains two tiles, namely the A and D tiles.

The number of tiles and tile sets is linearly bound by the number of vertices in the corresponding Bounded NCL graph. Any instance of Bounded NCL can be converted into an instance of Sigmar's Garden. Therefore, Sigmar's garden is $\mathcal{NP}$-hard.

Moreover, the order of removing the tiles serves as a solution for the puzzle instance. The solution can be verified in polynomial time by following this removal sequence. Therefore, Sigmar's Garden is $\mathcal{NP}$-Complete. □

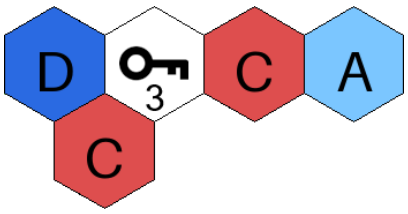


Figure 8: Remainder of CHOICE gadget when the left key is removed.

To illustrate, the remainder of this section will feature an example of how a CHOICE gadget would be encoded in the victory gadgets. Figure 8 shows the CHOICE gadget from figure 6a where the key with index 2 has been removed, leaving the other key unobtainable. We label their respective locks with the same index, although these are not explicitly shown in the figure.

Figure 9 shows the victory gadgets included to allow the entire gadget to be removed. Both locks with index 4 are unlocked by keys from the previous victory gadget. Both keys 5 unlock the following victory gadget, that are not included in this figure. The first victory gadget contains the lock and

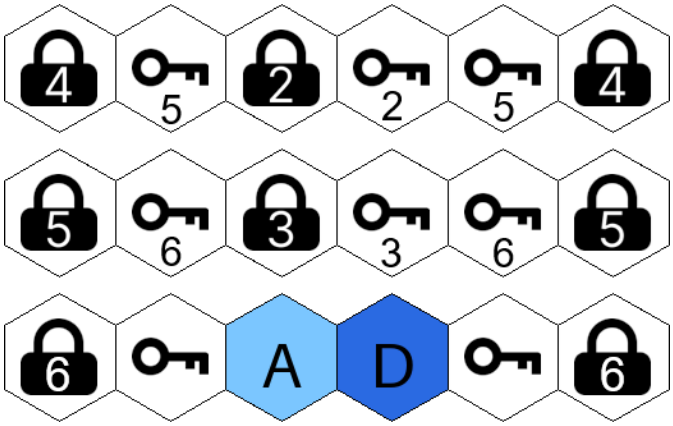


Figure 9: Victory gadgets for CHOICE gadget.

key with index 2. This pair is already removed in Figure 8, so they can be matched against each other. The following gadget contains the lock and key with index 3. As the second key is still locked behind two other tiles, the next victory gadget must be opened to remove the A and D tiles. Next both C tiles can be removed. The lock from the second victory gadget can be matched with the remaining key, and the key can be used to unlock the gadget connect to this CHOICE gadget. Now that all tiles have been removed, the board is fully cleared, meaning the corresponding constraint graph is reversible.

5 Implementation

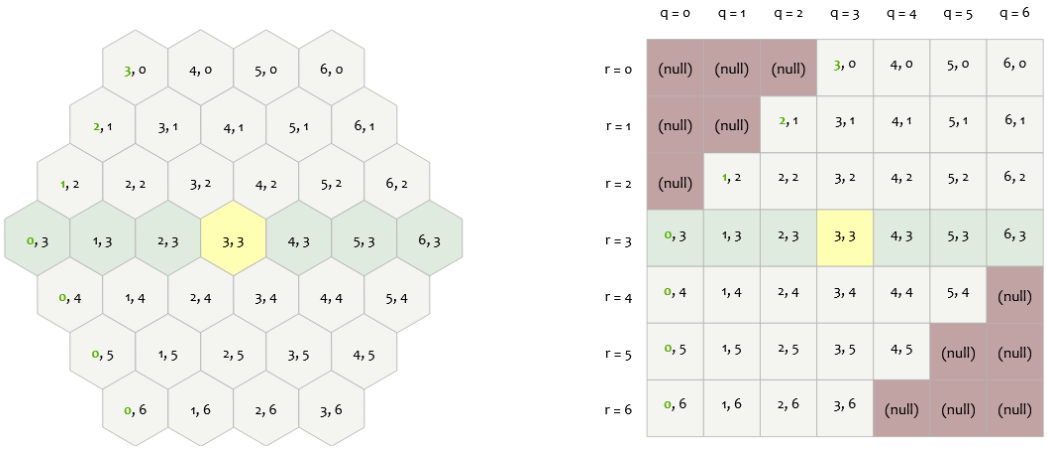


Figure 10: Conversion from array to hexagonal grid coordinates [red].

In order to be able to conduct research on Sigmar's Garden puzzles, a program was created using C++. This program allows the user to create, display and solve puzzles — either manually or by using a solver. The user is also able to create their own templates and generates puzzles using these templates. With a focus on modularity, the user is able to easily swap out or create new features.

One of the most notable details of Sigmar's Garden is its honeycomb-shaped board. This makes the implementation less trivial than an ordinary square grid. While coordinate systems such as cubic coordinate systems exist with elegant mathematical properties, this implementation opted for a simpler offset system, visualized in Figure 10. This method allows each cell to be indexed using only two coordinates. While it might not have the mathematical properties of the cubic system, the board indexing is hidden behind an abstraction layer, meaning the underlying implementation is not relevant to the user.

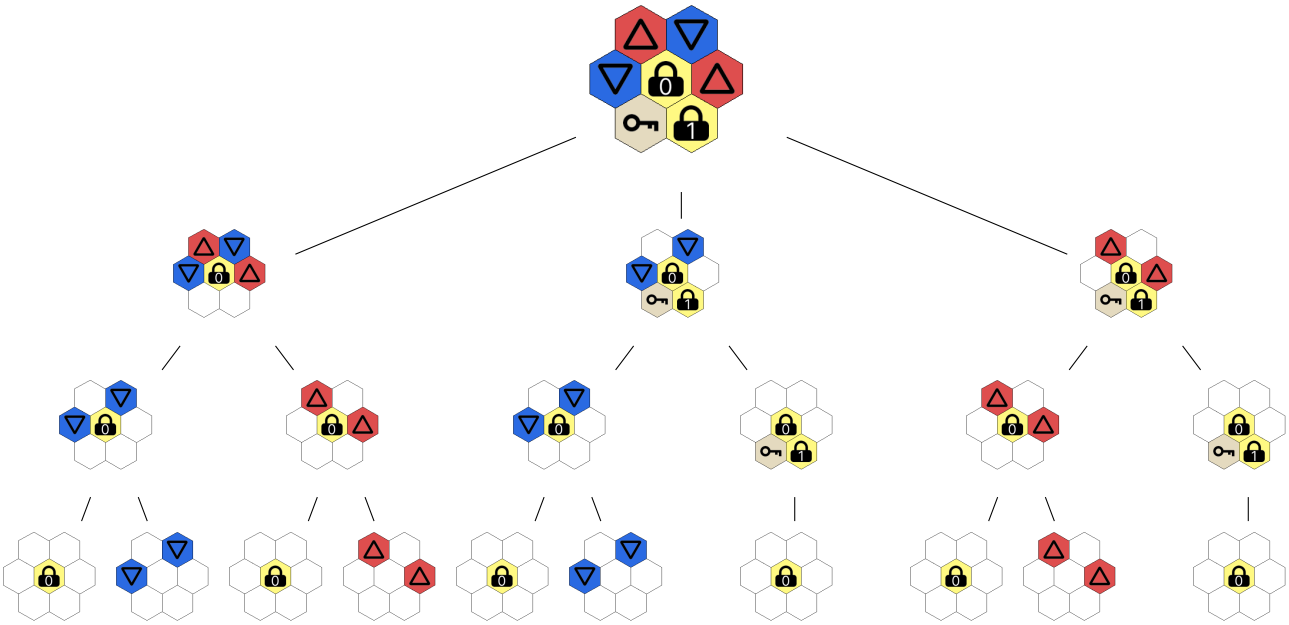


Figure 11: State-space tree of a board with seven tiles. The end states where the board is completely cleared are omitted.

6 Solver

As proven in Section 4, there exists no algorithm that is able to solve a Sigmar's Garden puzzle in polynomial time. Despite this, the following section attempts to create algorithms that efficiently solves these puzzles. Section 6.1 shows a rudimentary backtracking algorithm which gets improved in Section 6.2. Section 6.3 shows a method of encoding a Sigmar's Garden instance into a boolean formula, allowing a SAT-solver to find a solution to the puzzle.

6.1 Initial solver

Algorithm 1 Rudimentary backtracking algorithm for Sigmar's Garden

```
function FindSolutions(board)
    if board is clear then                                   ▷ Solution found
        return 1
    end if
    moves ← FindMoves(board)
    if moves is empty then                                   ▷ Board position not solvable
        return 0
    end if
    for move in moves do
        board ← DoMove(board, move)          ▷ Perform move and update board accordingly
        if FindSolutions(board) then
            return True
        end if
        board ← UndoMove(board, move)
    end for
    return False                                             ▷ No solution in this sub tree
end function
```

The initial solver is a rudimentary backtracking algorithm that traverses the entire state-space tree of the puzzle instance until it reaches a leaf node. Figure 11 shows the state-space tree for an arbitrary puzzle with seven tiles. The figure shows that such an algorithm would visit a total of five positions, including the end state. However, in a worst case, such an algorithm would have to traverse the entire state-space tree.
Algorithm 1 explains the high-level functioning of the solver. We assume that *board* is some object representing the puzzle. The functionalities of the functions called in Algorithm 1 is as follows:

- FindMoves(): finds all legal moves that can be formed with the tiles present in the frontier given the current state of the board. This function should omit moves that are not yet possible, such as lock-and-key pairs when a lock with a higher index is still present on the board.
- DoMove(): remove the tiles from the board as described in *move*.
- UndoMove(): place the previously removed tiles back on the board as described in *move*.

Lemma 6.1 *Let k denote the number of tiles on a board and d the average number of legal moves at any given time-step. Then, the number of states in the state-space tree is at least $d^{\lceil \frac{k}{2} \rceil}$.*

Proof. There are k tiles, removed over $\lceil \frac{k}{2} \rceil$[1] moves. At each time-step, the algorithm considers an average of d moves. Therefore, the states in the state-space tree is at least $d^{\lceil \frac{k}{2} \rceil}$. □

[1]The number of moves is rounded up due to the potential presence of the lock-0, which is removed on its own.

This function works in theory, but could perform poorly in practice. This is mostly caused by the vast state space of the puzzle. Lemma 6.1 shows that the lower bound of the number of states in a state-space tree of a puzzle is $d^{\lceil \frac{k}{2} \rceil}$. This is equal to the number of positions a naive backtracking algorithm has to visit in a worst-case scenario. Given 55 tiles on a standard Sigmar's Garden board and a fairly conservative $d = 4$ — which is reasonable as the number of valid pairs at a given state often approach 30 —, we get $d^{\lceil \frac{k}{2} \rceil} = 4^{\lceil \frac{55}{2} \rceil} \approx 7.21 \cdot 10^{16}$ positions. If the algorithm managed to visit 10^9 positions each second, it would still take over two years to traverse the entire state-space tree. It should be noted that such a puzzle is unlikely to exist, as there are many solutions to any given puzzle. However, if the puzzle is unsolvable — for instance, a deadlock such as in Figure 15 occurring in the puzzle — would require the solver to visit nearly every state in the state-space tree.

6.2 Sophisticated solver

In order to combat the inefficiency of the naive backtracking algorithm, dynamic programming is applied. This improves the efficiency of the algorithm by eliminating the need to visit positions more than once. This is implemented by keeping a map $DP\colon x \mapsto s$ where x is an encoding of a state of a puzzle board, and s is the number of unique solutions given board state x. While the algorithm halts after finding the first solution, it is possible to let the algorithm navigate the entire state-space tree and use s to count the number of solutions of a given puzzle. x is a bit string where the i-th bit stores whether the i-th tile in the puzzle template is occupied by a tile or if it is removed. To use space efficiently, cells that have never contained a tile are omitted from this bit string. For example, x for the state in Figure 11 with all tiles present is equal to 1111111, while x for the state with only the last lock left is equal to 0001000. Using this method, a standard Sigmar's Garden puzzle state can be encoded using 55 bits, which fits in an unsigned 64-bit integer type.

For each step in the algorithm, it checks if the board has already been visited by checking its presence in DP. If so, the value can be retrieved and returned. Otherwise, the number of solutions is calculated and inserted into DP.

This addition significantly cuts down on the amount of positions needed to visit, as many tiles could be done in any order. For instance, Figure 11 shows that the water tiles and fire tiles can be removed in either order, but the remaining board is the same. With dynamic programming, this position is not further expanded the second time.

In its current state, the positions are visited in the order determined by FindMoves(), which might not be optimal, as it returns the pairs in canonical order (i.e., ascending order by tile index). To combat this, the moves are ordered using a priority queue, that favors moves with a high heuristic value. Five different heuristics were tested:

- **None**: baseline heuristic that maintains the order provided by FindMoves().
- **Lock distance**: the minimum of the distances between the tiles in the move and the lock with the highest index. This encourages the algorithm to prioritize freeing the locks, as they are essential to freeing up tiles otherwise locked behind a lock.
- **Most neighbors freed**: how many neighboring tiles are made available by removing the

tiles in this move. This prioritizes moves that unlock more neighbors, which increases the frontier size and in turn the number of possible moves.

- **Most neighbors**: the number of neighbors the tiles in the move have. This is similar to the previous heuristic, except for the fact that it is easier to compute.
- **Distance to center**: the inverse distance to the center of the board. This prioritizes moves that have tiles closer to the center of the board, which unlocks tiles from the inside out.

The aim of these heuristics is to have the solver prioritize moves that lead to a solution, in a fashion that attempts to mimic a human strategy. For instance, the heuristic that prioritizes moves closer to a lock is a strategy that closely resembles the way a human might attempt to solve a puzzle. The performance of the heuristics are shown in Section 8.

6.3 SAT-solver

Instead of verifying the solvability and finding a possible solution to a puzzle instance using an intuitive algorithmic method, another approach is to encode the puzzle instance to a boolean formula and have it be solved by a SAT-solver. The goal of a SAT-solver is to solve the boolean satisfaction problem (SAT) by finding an interpretation of a boolean formula that satisfies it, meaning the boolean formula evaluates to true. While SAT is proven to $\mathcal{NP}$-Complete, SAT-solvers are highly optimized and are often able to solve complex problems in seconds.

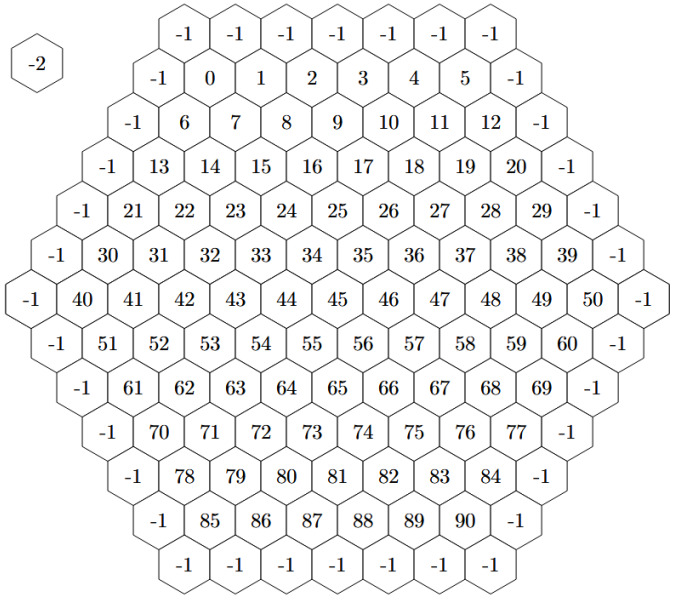


Figure 12: Tile numbering used for encoding a standard format puzzle instance to a boolean formula.

In this section, a procedure for encoding a Sigmar's Garden puzzle instance into a boolean formula in Conjunctive Normal Form (CNF) is given. This representation was chosen because of its common use in SAT-solvers, including Glucose [AS09], the SAT-solver used in this thesis. It reads from a file in DIMACS format and optionally returns an interpretation that led to the boolean formula being satisfied. The general form of a boolean formula in CNF is as follows:

$$\phi = C_1 \wedge C_2 \wedge ... \wedge C_m,$$

where each clause C_i is of the form

$$C_i = (l_{i1} \vee l_{i2} \vee ... \vee l_{in}).$$

Each ℓ_{ij} is a boolean literal, meaning $\ell_{ij} \in \{x_i, \neg x_i\}$.

For example, consider the following formula:

$$\psi = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2).$$

This boolean formula is in CNF, as it is a conjunction of disjunctions. This formula is only satisfiable if and only if $x_1 \leftrightarrow x_2$. On the contrary, the following formula is unsatisfiable, as it requires x_1 to

be both true and false at the same time, which is impossible:

$$\psi = x_1 \wedge \neg x_1.$$

The procedure to convert a puzzle instance to a boolean formula described below works for any board that is "legal", in the sense that for each each lock (apart from lock-0), the board should have a key, and if lock-k exists, then all locks with a lower index down to lock-0 are expected to exist. Furthermore, lock-0 is expected to always be present, although its position which is generally found in the center of the board, is not enforced. Although missing tiles (for instance, one less vitae tile compared to the number of mors tiles) is considered an illegal board, a SAT-solver is able to correctly label it as unsatisfiable.

Figure 12 shows the board numbering that is used throughout the procedure. Each tile on the board is labelled with a positive integer starting at 0. The board has one layer of padding, indicated by the cells with -1. Finally, lock-0 not having a standard tile to match against, makes the encoding process more complex. To avoid the complexity that arises from the absence of a standard tile for lock-0 to form a pair with, a "hidden key", denoted with index -2, is added that matches only with lock-0 and is removable at all times and is not present on the board. This allows the gold pair to be treated more similarly to the other pairs.

First, we define some constants that will be used throughout the encoding procedure. We define the set of all cell indices of cells that contain a tile (i.e., not free) at t_0 as T. The number of tiles will be represented by $n_t = |T|$. For clarity, $-2 \notin T$. The set of all tiles including the hidden key will be referred to as $\hat{T} = T \cup \{-2\}$. The number of pairs is derivable from the number of tiles, but because of its common use, we define it as $n_p = \lceil \frac{n_t}{2} \rceil$. As the hidden key of lock-0 is not counted as a standard tile, n_p is rounded up to account for this. We define the number of cells on a board as $n_c = 3 \cdot dim \cdot (dim - 1) + 1$, which is derived from the sequence of centered hexagonal numbers [OEI]. In this formula, dim represents the number of cells on one side of the hexagonal grid. As an example, the dimension of the board shown in Figure 1 and all standard Sigmar's Garden boards in general is equal to 6.

As two tiles are removed in the same time step, we define two different types of time steps. We define time steps for which point in the sequence a pair of tile is removed. The number of pairs removed is equal to n_p, so there are n_p time steps in the context of pairs. When working with individual tiles, we define time steps where one tile is removed per time step. This means that for a move $Pair_{i,j,t}$ (defined in Formula 5), the first tile is removed at tile timestep $2t - 1$ and the second tile is removed at time step $2t$.

The first group of clauses are constants that define all cells i that are free at the first time step, t_0. These are all cells that do not contain a tile at the beginning of the game, including the padding cells, denoted by the variable $Free_{i,0}$. We can define this as follows:

$$\bigwedge_{i=-1}^{n_c} \ell_i, \quad \text{where } l_i = \begin{cases} Free_{i,0}, & i \notin T, \\ \neg Free_{i,0}, & i \in T. \end{cases} \tag{1}$$

We also define constants that enforce that no tiles were removed at t_0, with variable $Removed_{i,0}$.

We do this as follows:

$$\bigwedge_{i \in \hat{T}} \neg Removed_{i,0}. \tag{2}$$

Note that the hidden key is not included in either conjunctions, as they are not referenced in any of the following clauses.

We recursively define the availability of a tile by verifying whether it was either free during the previous time step, or if it was removed during that time step. We do this as follows:

$$\bigwedge_{i \in T} \bigwedge_{t=1}^{n_t+1} (\neg Free_{i,t} \vee Removed_{i,t-1} \vee Free_{i,t-1}). \tag{3}$$

Note that this rule adds clauses for each time step including $n_t + 1$, as Formula 4 requires these variables to be set. Additionally, note that the hidden key is not included in these clauses, as this tile does not exist on the same plane as the rest of the board, so no tile's availability is dependent on that of the hidden key.

Here, we also define the "winning" condition. We consider a board to be solved if and only if all tiles have been removed from the board. We manage this by having constants that require each tile to be free after all moves. In this conjunction, the hidden key is included.

$$\bigwedge_{i \in \hat{T}} Free_{i,n_t+1} \tag{4}$$

Next, we define all valid pairs on the board. An initial intuition might be encoding the type of each tile and having the SAT-solver check for each pair of tiles whether they form a pair. However, this vastly increases the number of clauses and the complexity of the boolean formula. Instead, all valid pairs are defined before encoding the puzzle, meaning only clauses representing a valid pair exist.

We define a set $J_i = \{j \in \hat{T} \mid j > i \text{ and } i, j \text{ form a valid pair}\}$. J_i contains all tile indices j that form a valid pair with tile i, where $j > i$. We define the formula to encode the tile pairs as follows:

$$\bigwedge_{i \in \hat{T}} \bigwedge_{j \in J_i} \bigwedge_{t=1}^{n_p} (\neg Pair_{i,j,t} \vee Removed_{i,2t-1}) \wedge (\neg Pair_{i,j,t} \vee Removed_{j,2t}). \tag{5}$$

This adds a clause for each unique and valid pair of tiles. It enforces that when $Pair_{i,j,k}$ is true, meaning this move is performed, that tiles i and j must be removed at this time step.

Similar to Formula 5, we enforce that when $Pair_{i,j,k}$ is performed, tiles i and j must be removable at that same move time step. This is enforced as follows:

$$\bigwedge_{i \in \hat{T}} \bigwedge_{j \in J_i} \bigwedge_{t=1}^{n_p} (\neg Pair_{i,j,t} \vee Removable_{i,2t-1}) \wedge (\neg Pair_{i,j,t} \vee Removable_{j,2t-1}). \tag{6}$$

Note that, even though the tiles are removed at different time steps as shown in Formula 5, both clauses check if the tile was available at $2t-1$. This is because the tiles are only removed at different

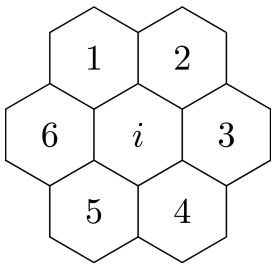


Figure 13: Neighbor indexing of $N_i(k)$.

time steps for individual tiles, but the first tile is not actually removed from the board until the second tile is removed simultaneously.

Next, we define clauses that ensure no two tiles are removed at the same time step.

$$\bigwedge_{t=1}^{n_t} \bigwedge_{i \in \hat{T}} \bigwedge_{\substack{j \in \hat{T} \\ j > i}} (\neg Removed_{i,t} \vee \neg Removed_{j,t}) \tag{7}$$

We also define clauses that ensure a single tile cannot be removed across multiple time steps.

$$\bigwedge_{i \in \hat{T}} \bigwedge_{t_1=1}^{n_t - 1} \bigwedge_{t_2=t_1+1}^{n_t} (\neg Removed_{i,t_1} \vee \neg Removed_{i,t_2}) \tag{8}$$

Together, these clauses ensure any tile can be removed at most once, and only one tile can be removed at any time step. However, in its current form, the SAT-solver permits a solution where no tiles are removed at all, as each $Removed_{i,t}$ can be interpreted as false, satisfying each clause. Therefore, we add disjunctions that require at least one $Pair_{i,j}$ to be interpreted as true for each time step. This is done as follows:

$$\bigwedge_{t=1}^{n_p} \left(\bigvee_{i \in \hat{T}} \bigvee_{j \in J_i} Pair_{i,j,k} \right). \tag{9}$$

Next, the rules of removability must be encoded into the boolean formula. This is enforced by only allowing $Removable_{i,t}$ to be interpreted as true if and only if tile i has three or more contiguous neighbors that are free at time step t. As the $Removable_{i,t}$ variables are only accessed in Formula 6, we only need to define $Removable_{i,t}$ for when t is odd. First, we define a function $N_i(k)$ that returns the index of the k-th neighbor of tile i, according to the neighbor indexing shown in Figure 13. As an example, the fourth neighbor of tile 44 as seen in Figure 12 is tile 55. We encode the removability rules as follows:

$$\begin{aligned} \bigwedge_{i \in T} \bigwedge_{t=1}^{n_p} \bigwedge_{k=1}^{6} (\neg Seq_{i,2t-1,k} \vee Free_{N_i(k),2t-1}) \wedge (\neg Seq_{i,2t-1,k} \vee Free_{N_i((k+1) \bmod 6),2t-1}) \\ \wedge (\neg Seq_{i,2t-1,k} \vee Free_{N_i((k+2) \bmod 6),2t-1}). \end{aligned} \tag{10}$$

This adds a variable $Seq_{i,t,k}$ that represents the k-th sequence of contiguous neighbors that is true only if these neighbors are free at time step t. Formula 10 adds k sequences for each tile in T, but the tile is only removable if at least one of these sequences can be interpreted as true. Therefore, we need to add a disjunction containing all k sequences that enforces at least one of them to be true in order for $Removable_{i,t}$ to be interpretable as true. This is done by adding the following conjunction:

$$\bigwedge_{i \in T} \bigwedge_{t=1}^{n_p} \left(\neg Removable_{i,2t-1} \vee \bigvee_{k=1}^{6} Seq_{i,2t-1,k} \right). \tag{11}$$

Finally, the lock ordering must be enforced. Currently, locks and keys can be removed in any order, as defined in Formula 6. However, the rules of the game state that in order to remove lock-m, all locks with index $k > m$ must have already been removed. Let T_k be the set of all keys. We also define a sequence $L_0, L_1, \ldots, L_{m-1}$ of m locks, where L_i represents lock-i. Now we add the following clauses to enforce the lock ordering:

$$\bigwedge_{i=0}^{m-2} \bigwedge_{j \in T_k} \bigwedge_{t=1}^{n_p} (\neg Pair_{min(L_i,j),max(L_i,j),t} \vee Free_{L_{i+1},2t-1}). \tag{12}$$

This goes over all lock-key pairs and ensures that the previous lock has been cleared. Because of its recursive property, this ensures all previous locks have been removed. Note that $Pair_{i,j,t}$ takes the minimum of the lock and the key for the value of i and the maximum for j. This is because Formula 5 and 6 only define pairs for when $i < j$, so $Pair_{j,i,t}$ with $i > j$ would not exist.

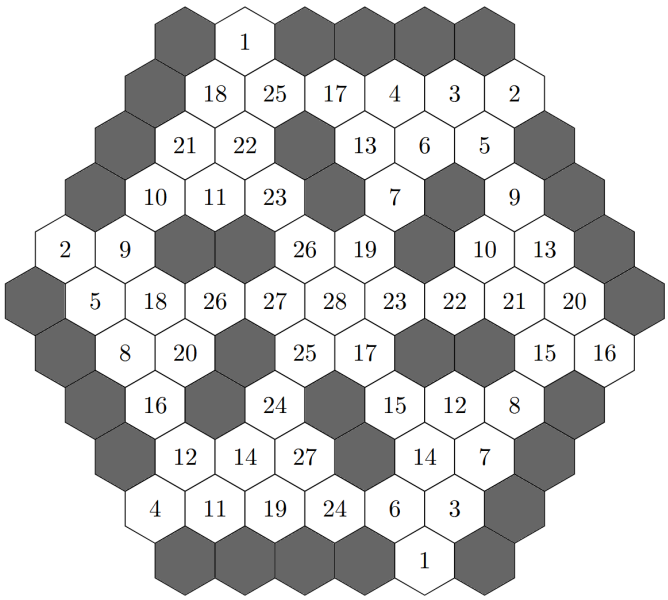


Figure 14: Order of tile removal as generated by SAT-solver, where two tiles with the same index are removed at the same timestep.

Given the output interpretation of the SAT-solver, a possible solution to the puzzle can be extracted. By extracting all $Pair_{i,j,t}$ and aggregating the pairs that are interpreted as true for each k, the tiles to be removed in each move can be found. For instance, when encoding the puzzle shown in Figure 1 into CNF and solving it using a SAT-solver, we get the solution illustrated in figure 14.

7 Generator

The final part of the research consists of creating an algorithm to generate Sigmar's Garden puzzles, that are guaranteed to have a solution. As mentioned in Section 3, the generation of Mahjong Solitaire puzzles with a guaranteed solution is non-trivial, and given the similarities between Mahjong Solitaire and Sigmar's Garden, it is reasonable to suspect the same can be said about generating instances of Sigmar's Garden. This is also backed by numerous open-source implementations that rely on some form of backtracking or verifying the generated puzzle with a solver in order to guarantee its solvability.

The first analyzed implementation by [Dot] generates a board at random and verifies it using a solver. It first procedurally generates a template, so the puzzles are not constrained by a predefined set of templates. It then iteratively decides a position to place a tile at random, until all tiles are placed. Afterwards, the puzzle is either accepted or rejected, depending on the outcome of the solver. The solver uses a backtracking algorithm similar to Algorithm 1, with the addition of memoization. The solving algorithm halts and rejects the puzzle after visiting 500 positions and not having found a solution, which significantly speeds up the generation process, according to the author.

The second implementation by [Hru] uses constraint logic in combination with a backtracker to guarantee a solvable puzzle. The algorithm first generates a stack of pairs that determines the order in which the tiles are placed. This approach enforces that tiles from the same pair are removable at the same time. This works in combination with the principle of "deadlocks", i.e., situations where the second tile of a pair blocks the first tile. In such a case, another combination of positions is used. This move stack approach also allows some additional constraints to be enforced, for example, vitae and mors tiles may not be placed on the outer ring, adding some additional difficulty. If none of the positions yield a solvable sub-solution, the algorithm goes to a backtracking approach. The algorithm utilizes heuristics to first place tiles in cells with a high neighbor count over remote cells. Additionally, if the program attempts to backtrack more than 1000 times, the generation halts and a different template is attempted. The author notes that some templates are more difficult to generate puzzles for when compared to other templates.

A first intuition would be to create a generation algorithm that places tiles from the center outwards. Because of the constraints of the game, progression usually goes from the edges of the board to the center, as the frontier starts with tiles on the outside of the board and as more locks become available, the center of the board gets freed. Because of this natural progression, a bottom-up algorithm seems fitting. The base of the algorithm would consist of a fixed starting point (lock-0). According to the template, the surrounding cells are added to the frontier. Each space in the frontier is a cell where a tile may be placed. This process is repeated until the board is filled. While this would intuitively create puzzles that can be solved — as the order of the placement can be inverted and presented as a solution — there are some constraints to the game that prevent the algorithm in its base form from succeeding. For instance, this method would need the final pair placed to be removable when the puzzle is in its unsolved state, otherwise the move sequence cannot be inverted. This is inherently difficult to prevent in some puzzle templates, for example, the wheel as shown in Figure 3f. The final tiles placed will be on the edges of the outer ring, which cannot be removed at the first time-step. Only the tiles in the corners are available, but as these are not the final tiles placed, it is difficult to guarantee (without backtracking) that any moves are possible in this state.

Algorithm 2 Function to determine whether the placement of a tile conflicts with the principle assumption

```
function IsPlaceable(board, template, pos)
    if board[pos] is not empty or pos not in template then      ▷ Can never contain another tile
        return false
    end if
    neighbors ← GetNeighbors(pos)          ▷ Gets a 1-indexed list of neighboring positions
    i ← 1
    for i to neighbors.size do
        nb ← neighbors[i]
        if nb is empty and nb not in template then    ▷ Check if cell does not contain a tile but will in the future
            continue
        end if
        nb ← neighbors[(i + 1) mod neighbors.size]
        if nb is empty and nb not in template then
            i ← i + 1
            continue
        end if
        nb ← neighbors[(i + 2) mod neighbors.size]
        if nb is empty and nb not in template then
            i ← i + 2
            continue
        end if
        return true          ▷ If three contiguous neighbors are available, the tile can be placed
    end for
    return false          ▷ No sequence of three available Neighbors exist; tile cannot be placed
end function
```

Because of these limitations, each puzzle instance would have to be checked with a solver or generated using additional constraint logic in order to guarantee its solvability. Because of this, a novel method was designed that is able to generate puzzles that are guaranteed to be solvable in linear time. The algorithm works by placing the tiles from the frontier containing all removable cells from the first time-step, instead of generating it backwards or placing the tiles at random. It uses the following assumption:

> If a tile is present on the board, it is assumed an order of moves exist that leads to the removal of that tile, and this sequence cannot be invalidated.

This assumption is enforced by only allowing a tile to be placed if there are three or more contiguous neighboring cells that are each either already occupied by a tile, or not present in the template and will therefore never contain a tile. The method of determining this is explained in Algorithm 2. The implemented function effectively checks the opposite of the principal assumption; namely whether a cell is empty but will contain a tile in the future. When this function returns true, we assume there is a sequence of moves (i.e. the order in which all previous tiles were placed) that leads to this tile having three or more contiguous free neighbors at that specific time-step.

Algorithm 3 Base algorithm for generating tile-matching puzzles

```
function GeneratePuzzle(template)
    board ← {}      ▷ Initialize empty board      ▷ Assumed to be of the same shape as template
    pairs ← GeneratePairStack() ▷ Size assumed to be equal to number of cells in template
    frontier ← FindFrontier(template)              ▷ Find all cells removable at time-step 0
    while pairs not empty do
        pos1 ← frontier[ RandInt(1, frontier.size)]
        repeat
            pos2 ← frontier[ RandInt(1, frontier.size)]
        until pos1 ≠ pos2
        board[pos1] ← pairs.back.first
        board[pos2] ← pairs.back.second
        pop top element of pairs
        remove pos1 from frontier
        remove pos2 from frontier
        neighbors ← GetNeighbors(pos1)
        neighbors ← neighbors ∪ GetNeighbors(pos2)
        for i ← 1 to neighbors.size do
            if IsPlaceable(board, template, neighbors[i]) then
                frontier ← frontier ∪ neighbors[i]
            end if
        end for
    end while
end function
```

Similar to the second open-source implementation analyzed above, the algorithm starts by generating a sequence of tile pairs to be placed. The placement occurs in pairs, so the final placement sequence consists of a list of pairs that will be placed in that order. First, there is a 50% chance for each pair of salts to cross over with a random pair of cardinal elements, meaning one cardinal element and one salt to be swapped. This results in two element-salt pairs, which allows for the generation of puzzles where an element-salt move is required. After that, all other tiles are added to the sequence, and the sequence is shuffled. When the final ordering of pair placements has been determined, the lock tiles can be given an index. The lock tile closest to the top gets index 5, the lock closest to the bottom gets index 1. Note that lock-0 is omitted, as it is treated differently from the other pairs during the placement process.

A base form of this algorithm can be found in Algorithm 3. This algorithm does not contain code specifically for Sigmar's Garden, but is instead applicable to a more general form of tile-based matching games, such as Mahjong Solitaire, under the assumption that the board, tiles and availability algorithm functions work in a similar fashion to the methods described above. Instead, the additions needed to be able to apply the base algorithm directly to Sigmar's Garden are listed below.

The base algorithm is quite similar to the algorithm suitable for Sigmar's Garden in particular, apart for the lock-0 placement. The placements of the tiles are undecided until they are actually placed down, except for lock-0, of which the position is predetermined. One way to take this into account is to keep the dedicated lock-0 cell unoccupied until all other tiles have been placed, and only then place the lock-0 tile. However, this prevents the generation of all puzzles where lock-0 is not the final tile placed. The second method is more elaborate, but is able to generate puzzles where the lock-0 tile is not the final tile to be placed.

Instead of waiting to place lock-0 until the end of the generation process, add a small chance (i.e. $\frac{1}{frontier.size+1}$) of placing the lock-0 tile, if and only if all previous locks have been placed, and the lock-0 position is present in the frontier. This allows puzzles where the final lock is not necessarily the last tile to be removed.

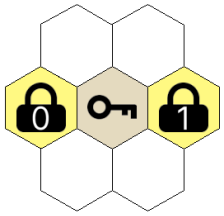


Figure 15: Example of a deadlock that may occur near the end of a game.

8 Results

In the previous section, various heuristics were introduced that attempt to speed up the solving process. For each template-heuristic pair, five puzzle instances were solved on puzzles generated using the game itself and 100 puzzles were solved using the C++ program, the results of which can be found in Table 1 and 2 respectively. Additionally, the average time to compute the puzzles using the C++ program and the SAT-solver for the same puzzles can be found in Table 3. The same puzzles were used for each heuristic per template. The mean for each run is shown, in combination with the standard deviation. It is important to note that only five puzzles were considered for each template from the game itself, so any comparison between the game and the C++ program may not be conclusive.

The number of positions visited per puzzle varies greatly, causing the high standard deviation. While the median number of positions is often lower than the mean, outliers with millions of visited positions are quite common, causing the high mean.

The tables do not show a heuristic that performs vastly better than the other heuristics. While the heuristics often performed better than no heuristic at all, heuristics that were expected to perform well on certain templates seem to under perform. The opposite also appears to be true; the distance to center metric was expected to perform poorly on template 3c since the heuristic could only give three unique outputs for this template. However, on the puzzles generated using the C++ program, this heuristic performs notably well. This indicates that the strategy of removing tiles closer to the center first, may be a better approach to solve puzzles with this template than initially expected.

As previously mentioned, the low number of puzzles taken into account in Table 1 may take away some of the integrity of the results. However, it is still interesting to note that the heuristic that opts for the move that frees up the most neighboring tiles, performs better on puzzles generated by Opus Magnum than all other heuristics on the dense and rings template, seen in Figure 3a and 3c respectively. On the contrary; this heuristic performs the worst on puzzles on the same template, when generated using the C++ program.

For most puzzle instances, the solver is able to find a solution within 10 milliseconds. However, Template 3c with tiles only on the outer two rings and lock-0 in the center, often had the number of positions visited in the millions, taking around 10 seconds to find a single solution. The problem arises when a puzzle is unsolvable. On puzzles where the solver encounters a deadlock near the beginning, the solver would halt quite fast. In a worst-case scenario — puzzles that contain a deadlock near the end — the solver took upwards of 45 minutes before halting. For instance,

Heuristic	Puzzle template				
	Dense	**Pinwheel**	**Rings**	**Sparse**	**Star**
None	1,415 ± 2,619	3,961 ± 5,421	1,420,310 ± 2,835,010	10,215 ± 12,667.2	138 ± 93
Lock distance	270 ± 160	615 ± 752	1,110,260 ± 2,218,950	6,098 ± 9,363	117 ± 39
Most neighbors freed	74 ± 30	709 ± 713	19,354 ± 28,320	870 ± 1,018	331 ± 382
Most neighbors	1,418 ± 2561	655 ± 921	1,420,250 ± 2,835,050	173,009 ± 340,788	180 ± 104
Distance to center	320 ± 215	1,464 ± 2,419	1,421,080 ± 2,834,630	774 ± 1,046	109 ± 76

Table 1: Mean ± standard deviation number of positions visited using heuristics on 5 per template, generated in Opus Magnum.

Heuristic	Puzzle template				
	Dense	**Pinwheel**	**Rings**	**Sparse**	**Star**
None	1,550 ± 6,529	4,649 ± 25,286	545,373 ± 2,472,940	196,597 ± 1,042,840	405 ± 929
Lock distance	1,179 ± 3,504	1,821 ± 8,741	577,630 ± 2,208,530	432,739 ± 1,930,470	722 ± 1,647
Most neighbors freed	1,943 ± 5,221	6,742 ± 25,775	788,053 ± 2,310,010	986,958 ± 3,722,110	709 ± 2,035
Most neighbors	1,831 ± 7,003	8,880 ± 33,213	436,211 ± 2,433,920	335,964 ± 1,768,820	632 ± 2,030
Distance to center	988 ± 2,596	3,793 ± 15,682	420,123 ± 2,420,240	647,589 ± 2,859,290	1,125± 3,392

Table 2: Mean ± standard deviation of number of positions visited using heuristics on 100 instances per template, generated using C++ program.

Heuristic	Puzzle template				
	Dense	**Pinwheel**	**Rings**	**Sparse**	**Star**
None	0.0059 ± 0.0374	0.0142 ± 0.0822	4.3906 ± 24.1876	1.2014 ± 7.2385	0.0009 ± 0.0024
Lock distance	0.0038 ± 0.0146	0.0057 ± 0.0289	3.9474 ± 16.2280	2.8288 ± 13.6514	0.0018 ± 0.0046
Most neighbors freed	0.0099 ± 0.0282	0.0309 ± 0.1240	8.4922 ± 26.2547	9.5988 ± 38.3993	0.0032 ± 0.0109
Most neighbors	0.0077 ± 0.0421	0.0296 ± 0.1193	4.2019 ± 27.8464	2.7429 ± 17.7640	0.0017 ± 0.0062
Distance to center	0.0027 ± 0.0075	0.0114 ± 0.0490	3.8767 ± 25.8291	5.1326 ± 27.1634	0.0034 ± 0.0112
SAT-solver	1.1576 ± 0.7488	1.2128 ± 1.2648	1.2426 ± 0.8044	1.1201 ± 0.7261	1.1569 ± 0.6587

Table 3: Mean ± standard deviation of time to solve puzzle using heuristics and SAT-solver on 100 instances per template, generated using C++ program.

Figure 15 shows a deadlock that may occur near the end of the game. Such a puzzle is unsolvable, regardless of the order of removal.

A traditional sequential solver, such as those implemented in Sections 6.1 and 6.2, does not have explicit methods of determining whether the board is solvable, without having to potentially visit every valid board state. A SAT-solver, however, is not bound to the sequential nature of these previously mentioned algorithmic approaches. For this reason, a SAT-based approach is often able to detect such a deadlock without having to try every possible combination of moves. This turns out to be true in practice; the SAT-solver, single-threaded Glucose [AS09] to be specific, is able to determine whether a given puzzle instance is solvable in around 10 seconds. In the case of a solvable puzzle, this number goes down to about half a second. This, however, does not include the overhead caused by the time taken to convert the puzzle to a boolean formula. The encoding procedure described in Section 6.3 is able to encore the puzzle shown in Figure 1 in 245,000 clauses, containing 28801 unique variables. It takes 1.3 seconds to run in total, of which 0.7 seconds was taken up by the SAT-solver. Compare this to the C++ program, which takes a total of 0.013 seconds, of which 0.0004 seconds was taken up by the solver. When a puzzle is known to be solvable, it is most efficient and practical to use a sophisticated dynamic programming solver as introduced in Section 6.2. The scenario's where the SAT-based approach might be better compared to the previously mentioned approach, is when it is not guaranteed that the puzzle is solvable, or when a puzzle is known to be difficult to solve, for example, template 3c. Table 3 shows that, for both the rings and sparse templates, it is more efficient to encode the puzzle into CNF and solve it using a SAT-solver. The results also show that the time to find a solution is consistently around 1.2 seconds, with no significant difference between templates.

The novel generator introduced in Section 7 manages to generate a guaranteed solvable puzzle in less than 0.0005 seconds, which is to be expected from an algorithm working in linear time. Even though this novel generator works significantly faster than the algorithm described by the second open source implementation mentioned in Section 7, in practice, this difference is minimal. However, this difference becomes vastly more apparent as the size and underlying complexity of the puzzle increase. Relying on a backtracking algorithm for puzzles as little as three times as large quickly becomes infeasible. It is important to note, however, that the mentioned implementation relies on randomly placing the tiles, which allows additional constraints to be enforced, such as vitae and mors tiles not appearing on the outer rings. This does not apply to the novel generation algorithm in its current state, as it relies on a predetermined placement ordering. While some constraints can be suggested, such as vitae and mors tiles not appearing in the top half of the placement ordering, there exists some arbitrary placement process on the board that places these tiles near the outside of the board.

8.1 Discussion

Section 8 showed how the different heuristics compare against each other on various board templates. Unfortunately, these results were not very conclusive. An issue encountered during the process was that the original generation algorithm created for Opus Magnum is not open source, so all puzzles in Table 1 had to be entered into the C++ program manually. This in combination with the fact that the game features many more templates, made it practically impossible to add more puzzles

and perform research with less variance.

While the results on the heuristics were not very conclusive, we can confidently say that the puzzles generated by the C++ program were less difficult than the puzzles provided by the original game. This suggests that the game features some additional constraints to the generation logic that prevents puzzles from becoming too easy, similar to the second open source implementation mentioned in Section 7.

9 Conclusion

This thesis focused on the solitaire puzzle game Sigmar's Garden. Bearing resemblance to Mahjong Solitaire, it requires the player to remove tiles from the board in pairs, with some additional rules and constraints.

It introduces several methods of solving these puzzles, most notably by using a top-down dynamic programming approach with heuristics, and by encoding the puzzle into a boolean formula and solving it using a SAT-solver. The former of which is able to solve almost all puzzle instances in less than 1/100th of a second, apart from some specific puzzle templates that are inherently more difficult to solve. The latter SAT-based approach often requires less than a second to find a solution to a puzzle. For instances where there is no valid solutions to the puzzle, the SAT-based approach takes on average 10 seconds, compared to 45 minutes with the sequential approach.

The thesis also introduces a novel way of generating Sigmar's Garden puzzles and tile-based matching puzzles in general. While most open-source implementations rely on some form of backtracking, this novel algorithm relies on a principle assumption that allows it to generate a puzzle in linear time, that is guaranteed to be solvable.

9.1 Future work

This thesis provided a novel way of generating tile-based matching puzzles, most notably Sigmar's Garden and Mahjong Solitaire. However, future work could research if this algorithm is applicable to other games, such as Shisen-Sho, which is similar to Mahjong Solitaire but has additional connectivity constraints. Alternatively, further research could look into applying a more abstract version of the algorithm to puzzles that do not directly include a matching feature, such as Patience.

Additionally, the novel generation algorithm can be further developed to allow the use of additional restrictions that dynamically apply during the process of placing the tiles on the board. As mentioned in Section 8, some open-source implementation rely on the random placement of tiles which conveniently allows for the injection of additional constraints. This is not so straightforward for the novel algorithm, but could potentially be altered to allow such additions.

The results on the heuristics showed in Section 8 were relatively inconclusive. Future research could be performed on these heuristics, such as combining heuristics, dynamically selecting or adapting heuristics, or developing new heuristics. Ideally, this would allow the solver to halt quicker on notoriously difficult templates, or potentially even unsolvable puzzles. Similarly, further research on the SAT-based approach could show that the addition of clauses can further speed up the process of generating a solution, or adding clauses to more efficiently find deadlocks in the puzzle.

Future research could also show that Sigmar's Garden with a fixed number of cardinal elements N_C is also $\mathcal{NP}$-complete.

References

[AS09] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. *Twenty-first International Joint Conference on Artificial Intelligence*, 2009.

[dB] Michiel de Bondt. Solving mahjong solitaire boards with peeking. Unpublished.

[Dot] Dotleon. https://github.com/dotleon/hexmatch-solver-generator. Accessed: 2025-7-8.

[HD09] Robert Hearn and Erik Demaine. *Games, Puzzles and Computation*. A K Peters, 2009.

[Hru] Ondřej Hruška. https://git.ondrovo.com/MightyPork. Accessed: 2025-7-8.

[OEI] OEIS. https://oeis.org/A003215. Accessed: 2025-6-29.

[red] redblobgames. Map storage of hexagonal grids in axial coordinates. https://www.redblobgames.com/grids/hexagons/#map-storage. Accessed: 2025-29-6.

[vR12] Jan N. van Rijn. Playing games: The complexity of klondike, mahjong, nonograms and animal chess. Master's thesis, Leiden University, 2012.

[Zac] Zachtronics. Opus magnum. https://www.zachtronics.com/opus-magnum/. Accessed: 2025/05/22.