Bachelor thesis
Computing Science

Radboud University

Flood-It as a SAT problem

Author:
Milan van Stiphout
s4596269

First supervisor/assessor:
Dr. C.L.M. Kop
`C.Kop@cs.ru.nl`

Second assessor:
Prof. dr. H. Zantema
`h.zantema@cs.ru.nl`

April 9, 2020

Abstract

Flood-It is a logical puzzle game. It is NP-Complete, meaning that verifying our solution can be done in polynomial time, but we currently do not know of a polynomial time algorithm for solving it. In this thesis, after some similar work on Sudoku, we approach the logical puzzle Flood-It by turning it into a boolean satisfiability problem and solving it using the techniques that are available for those problems. We create an encoding to solve the puzzle regardless of board size or amount of colours used and find bounds to quicker pinpoint the exact minimum size. Additionally, we experiment with an extended version of the encoding of the puzzle, but find no improvement to our encoding by doing this.

Contents

Chapter 1

Introduction

Although the computing power at our disposal is ever-increasing, some problems remain difficult to tackle computationally due to their nature. We do not have a way to solve problems in the NP-Complete class in polynomial time, meaning that as the problem's parameters grow, finding a solution becomes exponentially harder. Any NP-Complete problem can be reduced to any other NP-Complete problem, meaning that being able to solve a problem in the class fast can be useful for other NP-Complete problems.

The boolean satisfiability problem is such an NP-Complete problem. Given a boolean formula, the problem asks whether an assignment exists for its variables such that the formula evaluates to true. Although this problem is NP-Complete, many *inference techniques* exist to solve this problem efficiently. Quite some research is done on this, and it becomes easier and easier to solve many instances of the boolean satisfiability problem.

These promising developments have sparked some research on the application of these solvers on other NP-Complete problems. In one example[8], the NP-Complete puzzle *Sudoku* was rewritten to a boolean formula and solved using inference techniques.

In this thesis, we take another puzzle, called Flood-It, where the goal is to flood a game board in as few moves as possible. Finding the best possible solution to this puzzle is an NP-Complete problem[2]. We show how we convert Flood-It to a boolean satisfiability problem and how we use this along with a modern SAT solver to solve it. Showing how a problem like Flood-It can be solved, may be useful when trying to solve similar problems.

We will start by introducing some preliminary knowledge on different aspects of this thesis (chapter 2). Next, we will formalize Flood-It and present an encoding that specifies how to get a minimal solution for any instance of Flood-It (chapter 3). We will perform some experiments with this encoding (chapter 4), compare our work with existing literature (chapter 5) and conclude the research (chapter 6).

Chapter 2

Preliminary Knowledge

2.1 Flood-It

Flood-It is a single player logical game wherein the player must turn a square board into one colour in as few moves as possible. When there are more than three colours and the board is sufficiently large ($n \geq 3$), Flood-It is NP-Hard.[2]

The game starts off with the field in the top left corner of the board (called the *pivot*) in the flooded state. Any fields in the same colour that can reach the pivot through a path of fields of only that colour are also flooded.

Each move consists of the player choosing a colour. All flooded fields change colour to represent the move's colour. All fields touching the flooded area with the same colour are flooded as well. This is recursive; any field with the flooded colour that can reach the the flooded area through a path of only that colour, is flooded.

The flooded area grows and new neighbours will be eligible for flooding in the next move. This process continues until either the whole playing field is flooded. This must be done in the least possible amount of moves. Figure 2.1 shows a series of moves on a puzzle.

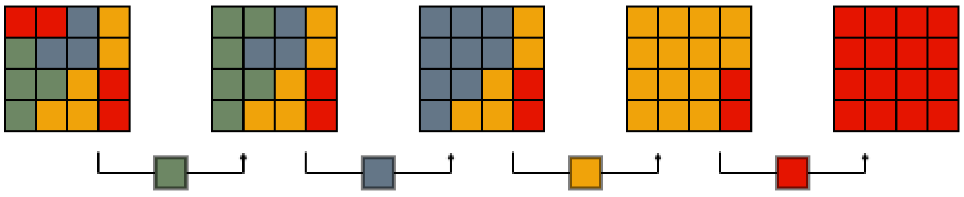
Figure 2.1: A possible (optimal) solution of a Flood-It puzzle.

2.2 Boolean logic and Conjunctive Normal Form

Conjunctive Normal Form (CNF) describes a format for formulas in boolean logic. A CNF formula is a conjunction of clauses, where a clause is a disjunction of literals. A literal is then either a variable or its negation.

An example of a CNF formula would be $\{(a \vee b) \wedge \neg c \wedge (c \vee \neg d \vee e)\}$. We see that disjunctions appear strictly as clauses of the conjunction and that each disjunction contains one or more literals (which may be positive as well as negative). There are also clauses that contain only a single element. These are called unit clauses. In the example above, $\neg c$ is a unit clause.

Any formula in classical boolean logic can be converted to an *equisatisfiable*[1] CNF formula in linear time[3]. This makes the format the go-to for SAT solvers: it is predictable and the conversion can be done quickly.

2.2.1 Transformation rules

In order to define our assertions of Flood in terms of a CNF, we must convert them from classical logic. For that, we will use the following transformations:

1. Double negation elimination
   $\neg\neg P \equiv P$

2. De Morgan's Law
   $\neg(P \wedge Q) \equiv \neg P \vee \neg Q$

3. Material implication
   $P \Rightarrow Q \equiv \neg P \vee Q$

4. Bi-implication rewrite
   $P \Leftrightarrow Q \vee R$
   $\equiv (P \Rightarrow Q \vee R) \wedge (Q \vee R \Rightarrow P)$
   $\equiv (P \Rightarrow Q \vee R) \wedge (\neg Q \wedge \neg R) \vee P)$
   $\equiv (P \Rightarrow Q \vee R) \wedge (\neg Q \vee P) \wedge (\neg R \vee P)$
   $\equiv (P \Rightarrow Q \vee R) \wedge (Q \Rightarrow P) \wedge (R \Rightarrow P)$
   $\equiv (\neg P \vee Q \vee R) \wedge (\neg Q \vee P) \wedge (\neg R \vee P)$

   Note that this rule is extensible; the binary disjunction in the original formula $(Q \vee R)$ can be n-ary. When the original formula looks like $P \Leftrightarrow Q \vee R \vee S$, we add S to the first clause of our final formula and add a new clause $(\neg S \vee P)$. By carefully constructing our rules containing bi-implications to have this format, we can reduce them simply using this rule.

Rules (1), (2) and (3) can be found in [7], or any other introductory logic book. Rule (4) is rewritten using material implication, both De Morgan's Laws, and the distributive law of disjunctions over conjunctions[12].

[1]Two formulae are equisatisfiable if one is satisfied only when the other is.

2.3 The Boolean Satisfiability Problem and SAT Solvers

The boolean satisfiability problem (hereafter called SAT) is concerned with determining whether for a given formula in propositional logic, an assignment for its contained variables exists such that the formula is satisfied. SAT was the first problem to be proven to be NP-Complete.[4]

A SAT solver takes a CNF formula as input, and outputs whether there exists an assignment for the variables such that the formula is satisfied. If the solver finds a satisfying assignment, it also outputs this. It may be the case that there are more satisfying assignments, but the solvers only find one as their primary task is to find out whether the formula is satisfiable.

SAT solvers use a variety of techniques to find solutions to this NP-complete problem as quickly as possible. These techniques are often boolean inference techniques that (conditionally) decrease the amount and size of clauses. Other, more technical techniques also exist: many SAT solvers do random restarts (while maintaining the information they have learned so far) in order to try to prevent being stuck in long branches without learning anything new for a long time.

Boolean formulas have the advantage that various inference techniques can be used on them to simplify them. An example of a boolean inference technique is *unit propagation*:

Example 1. Given a CNF formula $\{a \wedge (\neg a \vee b) \wedge (a \vee \neg c) \wedge (b \vee c) \wedge (\neg c \vee \neg d)\}$, we see unary clause a. For the CNF to be satisfiable, a must be true. This means that (1) any other clauses containing a must also be true, and (2) any occurrences of $\neg a$ can be removed ($\neg a$ will never hold, as this would contradict with clause a). We can thus reduce the CNF formula to
$\{a \wedge b \wedge (b \vee c) \wedge (\neg c \vee \neg d)\}$. The solver has learnt a and will look for a solution containing this, or a contradiction where $\neg a$ *must* hold. In the latter case, the formula is unsatisfiable. The given CNF formula can be solved entirely using unit propagation, but it often takes a combination of inference techniques and guesses to find a possible satisfying assignment.

We will be using MiniSat v2.2.0 (MiniSat was first described in [5]) as our solver of choice. It is a little older, but is still a very strong conflict-driven SAT solver. Just like most modern SAT solvers, it uses the DIMACS[2] input format.

[2]`https://logic.pdmi.ras.ru/~basolver/dimacs.html`

Chapter 3

A SAT encoding for Flood-It

Our strategy to solve Flood-It is to use a SAT solver (as described in section 2.3). In order for this to work, we must encode the puzzle as a boolean satisfaction problem. In this chapter, we will construct an encoding that describes the conditions for Flood-It in terms of a CNF formula. This formula can then be used as input for the SAT solver.

We want our solver to find the shortest sequence of moves that finishes the puzzle. This chapter formulates a strategy, formalizes game concepts and finally proposes a set of predicates and conditions that, when put together in a CNF formula, accurately describe any instance of Flood-It as a SAT problem.

Solving a SAT problem gives us a boolean answer. Thus, we cannot use a single encoding to answer the question “how many moves does it take to solve this Flood-It puzzle?” Rather, we choose a value p and pose the question “can we solve this Flood-It puzzle in p moves?” We then logarithmically approach the lowest possible value of p in order to find the minimal solution for the Flood-It puzzle.

3.1 States and transitions

A state in Flood-It is a configuration of colours of the board’s fields and whether the fields are flooded or not. Transitioning between states happens with moves: a colour is chosen, the flooded area is recoloured to this colour, and any fields (in)directly touching the flooded area that have this colour are marked as flooded. The colours are only important to clarify which fields will get flooded by which move: the goal of the game is to flood all fields, and not necessarily to flood them in a certain colour.

Value p denotes how many transitions (moves) we may make. There are $p + 1$ states: each state has a transition to the next state tied to it, except for the final state. A move at time 0 denotes a transition from the state at time 0 to the state at time 1.

3.2 Constants

For us to be able to specify properties of Flood-It, we must formally define the game and its workings. By naming all aspects of the game and properties of the playing field, we establish a frame of reference to build an encoding from. In this section, we handle all constants: all of the factors of the game that are *independent* of user input.

3.2.1 Game parameters

Besides the parameter p that was introduced at the beginning of this chapter, a game of Flood-It has two more parameters. An overview of all game parameters follows:

- n: the dimension of the board. A board is square, so that it has a size of $n \times n$;
- k: the number of colours available;
- p: the maximum number of moves that may be made to complete the puzzle.

While we are looking for a minimum value of p, the game's difficulty can be adjusted for players by increasing the number of moves they are allowed to make. There is always, however, a strict minimum value for p: a puzzle has a minimum amount of moves required to solve it.

3.2.2 Functions

In order to provide an easy way to refer to some field-related constants, the following functions are defined:

- **Neighbours(f)**: Given a field in a board, produces all orthogonal neighbours of that field.
  Corner fields two neighbours. Edge fields that are not corners will have three neighbours. Non-edge fields will have four neighbours.
- **Cluster(f)**: Given a field in a board, produces all fields connected to it at $t = 0$ with the same colour.
  This is recursive: the function groups all equally coloured fields with a path of the same colour between them.
- **Colour(f)**: Given a field, produces the colour it had at $t = 0$.

For these functions, $f \in \{0, \dots, n^2 - 1\}$. Each field is identified by its own number, starting with 0 for the pivot in the top left corner and then progressing to the right. At the end of the line, we move down one row and start from the left again.

3.3 Predicates

The properties of the game are expressed in the following predicates:

- $Fld(t, f)$
  Given a turn and a field, expresses whether that field is flooded. These predicates together specify the board state.
- $Move(t, c)$
  Given a turn and a colour, expresses whether that colour was chosen at that turn. A move transitions the board into a new state.
- $TchFld(t, f)$
  Given a turn and a field, expresses whether this field is next to a flooded field at that turn. This predicate helps our encoding as it shows which fields are eligible for being flooded next turn.

Here, f is defined as in section 3.2.2. These predicates make up the variables of our formula: we use them to represent the Flood-It puzzle. The assignments of the $Move$ predicates will tell us which moves we need to make in which order to solve a puzzle in a given amount of moves. This is only true, however, if we set proper conditions that all predicates must adhere to. We do this in the following section.

3.4 Flood conditions

The conditions in this chapter are set so that the predicates we established have to follow the rules of Flood-It. Only by doing this, will a possible assignment that the SAT solver finds be relatable to the original puzzle.

3.4.1 Precondition

When the game starts, the game is in state $t = 0$. The fields in the pivot's cluster are flooded, all other fields are not flooded. The precondition is as follows:

1. The cluster containing the origin field is flooded before the game starts.

$$\bigwedge_{f=0}^{n^2-1} \begin{cases} Fld(0, f), & \textit{if } f \in \textbf{Cluster}(0) \\ \neg Fld(0, f), & \textit{otherwise} \end{cases}$$

The case distinction is made to mimic the implementation of condition (1) as closely as possible: the precondition is expressed in unary clauses in the encoding.

3.4.2 Move conditions

All states except the last have a move associated with them. Each move is done with exactly one colour: there must always be a chosen colour for a transition to happen, and a transition may not have more than one colour at once. The following conditions reflect this:

2. At least one move must be made each turn.

$$\bigwedge_{t=0}^{p-1} \bigvee_{c=0}^{k-1} Move(t, c)$$

3. At most one move may be made each turn.

$$\bigwedge_{t=0}^{p-1} \bigwedge_{c=0}^{k-2} \bigwedge_{d=c+1}^{k-1} \neg Move(t, c) \vee \neg Move(t, d)$$

Condition (3) is symmetrical and the limits are set up to avoid duplicate clauses in the encoding. For each combination of moves occurring at a certain time, there is at least one that is not chosen.

3.4.3 Board conditions

During a game, our goal is to flood all fields on the board. Field f will be flooded at time $t+1$ *if and only if* at least one of the following situations occurs:

i f was already flooded at time t;

ii f touched the flood at time t and the move at time t was of colour **Colour**(f);

iii A field in **Cluster**(f) was flooded at time $t+1$ through option (ii).

More formally, we describe this as follows:

$$\begin{aligned} Fld(t+1, f) \Leftrightarrow & Fld(t, f) \vee (Move(t, c_f) \wedge TchFld(t, f)) \\ & \vee Fld(t+1, g_1) \vee \cdots \vee Fld(t+1, g_k) \end{aligned}$$

with $(g_1, \ldots, g_k) =$ **Cluster**(f) *and* $c_f =$ **Colour**(f)

The five conditions in this subsection describe the three situations presented above. Due to how the *TchFld* predicate is defined (in section 3.4.4) and condition (8), any field in **Cluster**(f) that floods will flood f as well. As such, when writing a condition that has $\neg Fld(t+1, f)$ as a consequent, we must make sure that no field in the entire cluster will be flooded.

The opposite is not true: due to condition (8), if any field in the cluster is flooded, they all are (a case of situation (iii)). Any condition with $Fld(t+1, f)$ in the consequent only has to look at an antecedent for which f is flooded, and condition (8) ensures that all fields in $\texttt{Cluster}(f)$ are flooded.

Condition (4) and (5) together form the basis for situation (i); condition (6) and (7) form situation (ii) and condition 8 equals situation (iii).

The five board conditions are:

4. A flooded field will remain flooded for the rest of the game.

$$\bigwedge_{t=0}^{p-1} \bigwedge_{f=0}^{n^2-1} Fld(t, f) \Rightarrow Fld(t+1, f)$$

$$\equiv \bigwedge_{t=0}^{p-1} \bigwedge_{f=0}^{n^2-1} \neg Fld(t, f) \vee Fld(t+1, f)$$

Once a field is flooded, there is no way to un-flood it. For all states except the last, we define that a flooded field is also flooded in the next state.

5. If no member of a field's cluster touches the flooded area and the field is not flooded already, the field will not be flooded.

$$\bigwedge_{t=0}^{p} \bigwedge_{f=0}^{n^2-1} \neg TchFld(t, g_1) \wedge \cdots \wedge \neg TchFld(t, g_k) \wedge \neg Fld(t, f) \Rightarrow$$
$$\neg Fld(t+1, f)$$

$$\equiv \bigwedge_{t=0}^{p} \bigwedge_{f=0}^{n^2-1} TchFld(t, g_1) \vee \cdots \vee TchFld(t, g_k) \vee Fld(t, f) \vee$$
$$\neg Fld(t+1, f)$$

with $(g_1, \ldots, g_k) = \texttt{Cluster}(f)$

6. If a move is chosen with a field's colour and that field touches the flooded area, it is flooded.

$$\bigwedge_{t=0}^{p-1} \bigwedge_{f=0}^{n^2-1} Move(t, c_f) \wedge TchFld(t, f) \Rightarrow Fld(t+1, f)$$

$$\equiv \bigwedge_{t=0}^{p-1} \bigwedge_{f=0}^{n^2-1} \neg Move(t, c_f) \vee \neg TchFld(t, f) \vee Fld(t+1, f)$$

with $c_f = \texttt{Colour}(f)$.

A field that directly touches the flooded area is eligible for flooding in the next turn. If the move colour equals the field's colour, the field is flooded. This combined with the previous rule makes sure that any clusters touching the flooded area eligible for flooding.

7. If a move is chosen that is not f's colour and f was not flooded prior, f is not flooded now.

$$\bigwedge_{t=0}^{p-1} \bigwedge_{f=0}^{n^2-1} \neg Fld(t,f) \wedge \neg Move(t,c_f) \Rightarrow \neg Fld(t+1,f)$$
$$\equiv \bigwedge_{t=0}^{p-1} \bigwedge_{f=0}^{n^2-1} Fld(t,f) \vee Move(t,c_f) \vee \neg Fld(t+1,f)$$

$$\textit{with } c_f = \mathbf{Colour}(f).$$

For an unflooded field, a move that is not of the field's colour will never flood the field. This rule captures that.

8. When a field is flooded, all connected fields with the same colour are also flooded.

$$\bigwedge_{t=0}^{p} \bigwedge_{f=0}^{n^2-1} \quad Fld(t,f) \Leftrightarrow Fld(t,g_1) \vee ... \vee Fld(t,g_k)$$
$$\begin{aligned}\equiv \bigwedge_{t=0}^{p} \bigwedge_{f=0}^{n^2-1} & (\neg Fld(t,f) \vee Fld(t,g_1) \vee \cdots \vee Fld(t,g_k)) \wedge \\ & (\neg Fld(t,g_1) \vee Fld(t,f)) \wedge \\ & \cdots \wedge \\ & (\neg Fld(t,g_k) \vee Fld(t,f))\end{aligned}$$

$$\textit{with } (g_1,\ldots,g_k) = \mathbf{Cluster}(f)$$

In the encoding, all fields are part of clusters. Any field connected to another field with the same colour, will always change together with this other field. The above condition represents this.

3.4.4 Touching condition

Here, we specify what it means when a field is touching the flooded area: a *direct neighbour* is flooded. Neighbours are orthogonal; no diagonals.

9. If and only if at least one of a field's neighbours is flooded, does it touch flood.

$$
\begin{aligned}
& \bigwedge_{t=0}^{p} \bigwedge_{f=0}^{n^2-1} \quad TchFld(t,f) \Leftrightarrow Fld(t,n_1) \vee \cdots \vee Fld(t,n_k) \\
\equiv & \bigwedge_{t=0}^{p} \bigwedge_{f=0}^{n^2-1} (\neg TchFld(t,f) \vee Fld(t,n_1) \vee \cdots \vee Fld(t,n_k)) \wedge \\
& \qquad (\neg Fld(t,n_1) \vee TchFld(t,f)) \wedge \\
& \qquad \cdots \wedge \\
& \qquad (\neg Fld(t,n_k) \vee TchFld(t,f))
\end{aligned}
$$

$$
\textit{with } (n_1, \ldots, n_k) = \texttt{N}\textbf{eighbours}(f)
$$

3.4.5 Postcondition

If the game can be completed in p turns, then at $t = p + 1$ it must hold that all fields are flooded. This does not necessarily mean that p is an optimal solution, just that the optimal solution is at most p moves. The postcondition we get is:

10. After the maximum number of moves has been made, all fields are flooded.

$$
\bigwedge_{f=0}^{n^2-1} Fld(p+1,f)
$$

3.5 Number of clauses

Table 3.1 gives an overview of the clauses each rule adds to the encoding, expressed in terms of the game parameters. This overview shows the structure of our encoding, which is insightful when reasoning about the effectiveness of different rules of inference or SAT solvers.

Condition	**Number of clauses**	**Arity of clauses**
1	n^2	1
2	p	k
3	$p \cdot \frac{k \cdot (k-1)}{2}$	2
4	$p \cdot n^2$	2
5	$(p+1) \cdot n^2$	$\lvert\mathbf{Cluster}(f)\rvert + 2$
6	$p \cdot n^2$	3
7	$p \cdot n^2$	3
8	$(p+1) \cdot n^2 +$ $(p+1) \cdot \lvert\mathbf{Cluster}(f)\rvert$	$\lvert\mathbf{Cluster}(f)\rvert + 1$ 2
9	$(p+1) \cdot n^2 +$ $(p+1) \cdot n^2 \cdot (4n^2 - 4n)$	3 to 5 2
10	n^2	1

Table 3.1: Number and arity of clauses of the conditions presented in chapter 3.4. Arity here refers to the number of literals in a clause.

Chapter 4

Experiments

4.1 A heuristic for approaching a minimal solution

4.1.1 Justification and set-up

Although an efficient encoding is paramount to us quickly solving any Flood-It puzzles, we are still estimating and then logarithmically approaching the minimum solution. This means that better setting our initial estimate means we have to compute fewer (potentially very time-consuming) encodings. Our initial logarithmic approaching algorithm looks as follows:

```
$puzzle \leftarrow$ Flood-It puzzle
$lower\_bound \leftarrow 0$
$upper\_bound \leftarrow$ GreedyMinimalSolution(puzzle)

function Approach($puzzle, lower\_bound, upper\_bound$)
    while $lower\_bound \neq upper\_bound$ do
        $minimal\_solution\_attempt \leftarrow \lfloor \frac{upper\_bound + lower\_bound}{2} \rfloor$
        encoding = Encoding($puzzle, minimal\_solution\_attempt$)
        if SAT-Solver(encoding) then
            $upper\_bound \leftarrow minimal\_solution\_attempt$
        else
            $lower\_bound \leftarrow minimal\_solution\_attempt + 1$
        end if
    end while
    return $lower\_bound$
end function
```

Figure 4.1: Initial algorithm for finding minimal solution.

This is a safe but slow approach: we are guaranteed that *lower_bound* $\leq$ *minimal solution* $\leq$ *upper_bound*, but we also know with high likelihood that the solution is not 0 (the probability for this is $(\frac{1}{k})^{n^2-1}$, which quickly works out to be negligible as n and k grow). Additionally, we know the number of moves required to flood a random board is $\Omega(n)$ with high probability.[2]

A smarter approach is in order. We will attempt to establish a heuristic for the bounds and build a new approaching algorithm that uses these bounds efficiently.

For n in the range $[4, 9]$ and k in the range $[3, 6]$, we generate 100 random puzzles for each combination of n and k and inspect the solutions.

4.1.2 Results and discussion

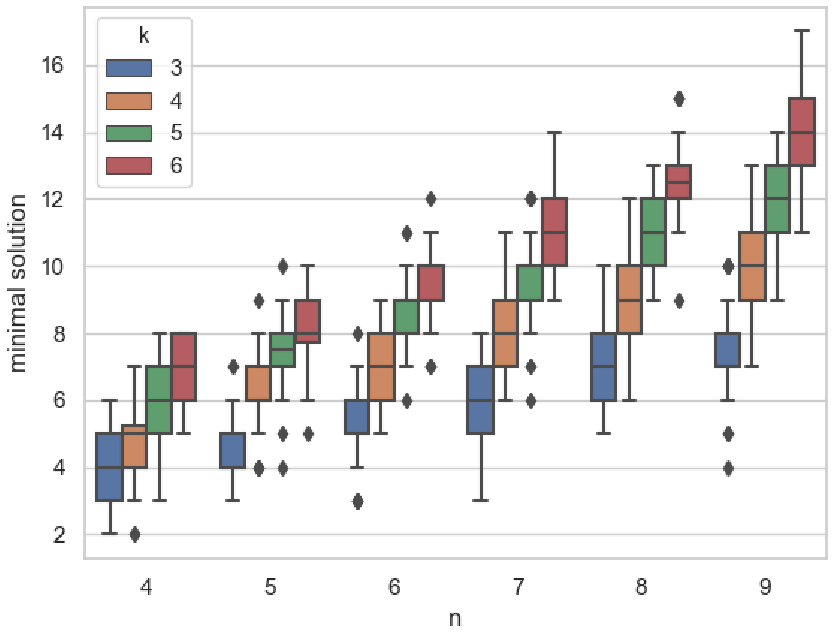

Figure 4.2: The minimum solutions of 100 of each combination of $n \in [4, 9]$ and $k \in [3, 6]$, visualized as box plots.

From the data, we conclude that starting with a lower bound of $n + k - 5$ is below the 25th percentile and $n + k$ is above the 75th percentile for all combinations of n and k.

For our improved algorithm, we set the bounds to these values. When our minimal solution ends up being one of the original bounds, we must verify that this is in fact the minimum solution. This ends up taking longer for these edge cases, but because $n + k - 5 \leq \mathit{minimum\_solution} \leq n + k$ with high likelihood for all tested combinations of n and k, we take this risk.

The new algorithm looks as follows:

```
$puzzle \leftarrow$ Flood-It puzzle
$lower\_bound \leftarrow puzzle.n + puzzle.k - 5$
$upper\_bound \leftarrow puzzle.n + puzzle.k$

$minimum \leftarrow \textsc{Approach}(puzzle, lower\_bound, upper\_bound)$

**if** $minimum = puzzle.n + puzzle.k - 5$ **then**
    **repeat**
        $minimum \leftarrow minimum - 1$
        $encoding \leftarrow \textsc{Encoding}(minimum)$
    **until** $\neg\text{SAT-}\textsc{Solver}(encoding)$
    $minimum \leftarrow minimum + 1$
**else if** $minimum = puzzle.n + puzzle.k$ **then**
    $encoding \leftarrow \textsc{Encoding}(minimum)$
    **while** $\neg\text{SAT-}\textsc{Solver}(\text{encoding})$ **do**
        $minimum \leftarrow minimum + 1$
        $encoding \leftarrow \textsc{Encoding}(minimum)$
    **end while**
**end if**
```

Figure 4.3: Improved algorithm for finding minimal solution. Variable *minimum* holds our final solution.

In our improved algorithm, we first assume that our solution lies between our initial boundaries. If the `Approach`-function gives us an answer that equals one of the original bounds, however, we have to check whether that is truly the minimum solution.

The actual minimum solution may lie below $n + k - 5$, so if `Approach` returns $n+k-5$, we decrement this and see whether the puzzle is solvable for that value. As soon as we try a value for which the puzzle is unsolvable, we know the value above it is the minimum solution. Values closer to $n + k - 5$ have a higher probability of being the minimum solution, which is why we decrement instead of again logarithmically approaching with lower bounds.

If `Approach` returns $n + k$, we must check whether this is actually satisfiable. We test whether it is, and if it is not, we increment our minimum by one and try again. We repeat this until we find the minimum solution. Values closer to $n + k$ have a higher probability of being the minimum solution, which is why we decrement instead of again logarithmically approaching with higher bounds.

4.2 An extended encoding

4.2.1 Justification

The encoding we have presented is minimal: no rule may be removed without breaking it. While this gives us the lowest construction time for encodings, redundant rules may be added that could give the SAT solver extra insights on more quickly handling puzzles. In this experiment, we are adding redundant clauses and seeing how they impact solving time.

4.2.2 Redundant clauses

Two rules are added to the original encoding:

11. If a move is made, the next move will not be of that colour.

$$\begin{aligned}
&\bigwedge_{t=0}^{p-2}\bigwedge_{c=0}^{k-1} Move(t,c) \Rightarrow \neg Move(t+1,c)\\
\equiv &\bigwedge_{t=0}^{p-2}\bigwedge_{c=0}^{k-1} \neg Move(t,c) \vee \neg Move(t+1,c)
\end{aligned}$$

Due to $Move$'s idempotence, performing a second move of the same colour never adds to the flooded area. Therefore, we can remove that option from the list.

12. If a field is flooded, its neighbours touch flood.

$$\begin{aligned}
&\bigwedge_{t=0}^{p}\bigwedge_{f=0}^{n^2-1} Fld(t,f) \Rightarrow TchFld(t,n_1) \wedge \cdots \wedge TchFld(t,n_k)\\
\equiv &\bigwedge_{t=0}^{p}\bigwedge_{f=0}^{n^2-1} (\neg Fld(t,f) \vee TchFld(t,n_1)) \wedge \cdots \wedge\\
&\qquad\qquad (\neg Fld(t,f) \vee TchFld(t,n_k))
\end{aligned}$$

4.2.3 Set up

We attempt to solve 100 randomly generated puzzles twice: once with the minimal encoding as presented in chapter 3, and once with the extended encoding that includes the two rules above. We look at the time the SAT solver takes to solve these puzzles for different n and k values and compare the times of the minimal and extended encoding. Per combination of n and k, we count how many times the minimal encoding was quicker, and how many times the extended encoding was quicker. We take $n \in [4, 11]$ and $k \in [3, 6]$.

4.2.4 Results and discussion

n	**4**		**5**		**6**		**7**	
k	*min.*	*ext.*	*min.*	*ext.*	*min.*	*ext.*	*min.*	*ext.*
3	83	17	83	17	90	10	94	6
4	86	14	94	6	97	3	97	3
5	88	12	95	5	91	9	97	3
6	89	11	96	4	99	1	99	1
n	**8**		**9**		**10**		**11**	
k	*min.*	*ext.*	*min.*	*ext.*	*min.*	*ext.*	*min.*	*ext.*
3	93	7	99	1	99	1	98	2
4	98	2	100	0	100	0	100	0
5	97	3	99	1	100	0	100	0
6	100	0	100	0	99	1	100	0

Table 4.1: For combinations of n and k, shows how many of the 100 puzzles were solved faster by the minimal (*min.*) encoding, and how many were solved faster by the extended (*ext.*) encoding.

As becomes clear from the results in table 4.1, solving the extended encoding takes the SAT solver longer than the minimal encoding in most cases. As n and k grow, the number of puzzles solved faster by the extended encoding decreases. It may be possible that adding more or different redundant clauses *does* reduce solving time (and possible even perform faster than the minimal encoding), but we cannot say this with certainty. Additionally, different SAT solvers may respond differently to these extra clauses.

Chapter 5

Related Work

A lot of variations of Flood-It exist, including a 2-player version and a version where the pivot field is chosen each turn, rather than just being the top left field. For many of these versions and specific boundaries, a proof of complexity has been made.[2, 11] Other than this, we have found no research relating to Flood-It.

SAT was the first problem to be classified as NP-Complete[4] and a lot of work has been done on solving it, much of which can be found in the JSAT journal[1]. The annual competition for SAT solvers[2] showcases new techniques, solvers, and efficiency improvements. SAT remains NP-Complete, but through inference rules, implementation optimizations and technical improvements, the solvers can handle more and more complex problems regardless.[1] SAT solvers have many applications in computing science, prominently including model checking and software verification[13] and cryptography[9, 10].

Earlier research showed specific instances of Sudoku were solved by converting them to a SAT problem and using inference rules on them.[8] These puzzles, however, had only a single solution and could be solved by logic alone. This is not the case for Flood-It, which it is search-based. Additionally, the goal for Sudoku is to find a final configuration that satisfies all constraints, whereas for Flood-It, the shortest path towards the final configuration is the goal. This leads to a structurally different and more complex encoding as states and transitions between states have to be distinguished.

Our research shows an application of SAT solvers and how they can be a part of the process of solving an NP-Complete game. Besides [6] and [8], we could not find other work like this on games.

[1]`http://jsatjournal.org/`

[2]`http://www.satcompetition.org/`

Chapter 6

Conclusions

We have found that Flood-It lends itself quite well to being encoded as a SAT problem. We presented a minimal but complete encoding and an algorithm that uses said encoding to find the minimal solution for a given Flood-It puzzle. The encoding uses ten conditions that are required to hold for the puzzle to be solved in a valid manner. The conditions specify rules regarding possible states of the game, as well as how to transition from one state to the next.

We established heuristic bounds that should make puzzles quicker to solve. We also experimented with adding redundant conditions to our encoding, but this was not successful. This could be due to the form of the encoding not benefiting much from our additional conditions, or due to the choice of solver.

The work done in this thesis can likely be applied very well to similar games, in which finding a path is most important, or the answer has to be approximated logarithmically, or both.

Bibliography

[1] Proceedings of SAT competition 2018. `https://helda.helsinki.fi/bitstream/handle/10138/237063/sc2018_proceedings.pdf`. Accessed: 30-03-2020.

[2] David Arthur, Raphaël Clifford, Markus Jalsenius, Ashley Montanaro, and Benjamin Sach. The complexity of flood filling games. In Luisa Gargano Paolo Boldi, editor, *Fun with Algorithms*, Lecture Notes in Computer Science, page 307–318. Springer, June 2010.

[3] C.e. Blair, R.g. Jeroslow, and J.k. Lowe. Some results and experiments in programming techniques for propositional logic. *Computers & Operations Research*, 13(5):633–645, 1986.

[4] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proc. Symposium on the Theory Of Computing (STOC)*, pages 151–158. ACM, 1971.

[5] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, Lecture Notes in Computer Science, page 502–518, 2004.

[6] Keijo Heljanko, Misa Keinänen, Martin Lange, and Ilkka Niemelä. Solving parity games by a reduction to sat. *Journal of Computer and System Sciences*, 78(2):430–440, 2012.

[7] Patrick J. Hurley. *A concise introduction to logic.* Cengage Learning, 12^{th} edition, 2015.

[8] Inês Lynce and Joël Ouaknine. Sudoku as a SAT problem, 2006.

[9] Ilya Mironov and Lintao Zhang. Applications of sat solvers to cryptanalysis of hash functions. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT*, pages 102–115, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[10] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *Theory and Applications of Sat-*

isfiability Testing - SAT, Lecture Notes in Computer Science, page 244–257, 2009.

[11] Uéverton Dos Santos Souza, Frances Rosamond, Michael R. Fellows, Fábio Protti, and Maise Dantas Da Silva. The Flood-It game parameterized by the vertex cover number. *Electronic Notes in Discrete Mathematics*, 50:35–40, 2015.

[12] Alfred Tarski. *Introduction to Logic.* Oxford University Press, 4th edition, 1994.

[13] Yakir Vizel, Georg Weissenbacher, and Sharad Malik. Boolean satisfiability solvers and their applications in model checking. *Proceedings of the IEEE*, 103(11):2021–2035, 2015.