Universiteit Leiden
Opleiding Informatica

Attacking the n-puzzle

using SAT solvers

Name: Luc Edixhoven

Date: July 15, 2016

1st supervisor: dr. W.A. Kosters
2nd supervisor: dr. H.J. Hoogeboom

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

In this paper we describe a SAT-based approach to solving the n-puzzle. After a brief introduction of the n-puzzle and a common approach to finding optimal solutions to it, we study a translation of the puzzle game Sokoban to SAT. We use this to propose a translation of the n-puzzle to SAT, which can be combined with state-of-the-art SAT solvers to find optimal solutions to the n-puzzle. Upon evaluation, we conclude that our approach does find optimal solutions but cannot as of yet compete with established algorithms such as A* in terms of computing time.

Contents

Abstract **i**

1 Introduction **1**

2 The n-puzzle **3**

2.1 Rules and definitions . . . 3

2.2 Solvability . . . 4

2.3 Complexity . . . 4

2.4 Wilson's puzzle graph . . . 5

3 A* search **7**

3.1 Dijkstra's algorithm . . . 7

3.2 Informed search . . . 8

3.3 Heuristic functions . . . 9

3.4 Variants of A* search . . . 10

4 SAT solvers **11**

4.1 The Boolean Satisfiability Problem . . . 11

4.2 SAT solvers . . . 12

5 From Sokoban to SAT **13**

5.1 Sokoban . . . 13

5.2 Translation . . . 14

5.2.1 Variables . . . 14

5.2.2 Constraints . . . 15

5.3 Example . . . 17

6 Translating the n-puzzle to SAT **19**

6.1 Variables . . . 19

6.2 Essential constraints 20
6.3 Additional constraints 21

7 Evaluation **23**
7.1 Programs and machines 23
7.2 The 8-puzzle 24
7.3 The 15-puzzle 25

8 Conclusions and further research **27**

Bibliography **27**

Chapter 1

Introduction

Be it due to one's parents owning an extensive collection of puzzles or because of treasure hunts in online games, most people will at some point in their lives have had an encounter with the 15-puzzle, where one has to slide tiles to reach a specific target configuration. Over the course of the years, various methods have been invented to solve these puzzles. For instance, many people choose to first solve the first, topmost row, then the second, then the third and so forth, until only two unsolved rows remain. They then proceed to solve the leftmost unsolved column until the puzzle is solved. This method is easy to apply, but rarely leads to solutions that are even close to being optimal. More complex methods exist and usually lead to better, faster solutions, but these also commonly require computers to execute them within reasonable time.

In the light of the recent improvements in the area of SAT solvers and due to them having been succesfully applied to solving similar, dynamic puzzles, we decided to research a new approach to solving the n-puzzle which uses these SAT solvers, by translating an instance of the n-puzzle to an instance of SAT and subsequently solving that instance of SAT using state-of-the-art SAT solvers.

In Chapter 2, we introduce the n-puzzle, its rules and its definitions, and elaborate on its solvability and its complexity. In Chapter 3, we describe a usual approach to solving the n-puzzle, the A* search algorithm, along with its predecessor, Dijkstra's shortest path algorithm, and several heuristic functions for the n-puzzle. In Chapter 4, we introduce the SAT problem and SAT solvers. In Chapter 5, we analyse the SAT-based approach of Naoyuki Tamura to solving the dynamic puzzle game Sokoban. We document his translation of Sokoban to SAT, and provide an example case of this translation. In Chapter 6, we propose our own translation of the n-puzzle to SAT, providing the essential constraints and two additional constraints which aim to restrict the search space. In Chapter 7, we compare our SAT-based approach to solving the n-puzzle to implementations of A* search and Dijkstra's algorithm. Finally, in Chapter 8, we summarise our findings and conclusions, and suggest several areas which may be interesting for further research on the topic.

This research was done as a Bachelor thesis at the Leiden Institute of Advanced Computer Science (LIACS), Leiden University, under the supervision of Walter Kosters and Hendrik Jan Hoogeboom.

Chapter 2

The n-puzzle

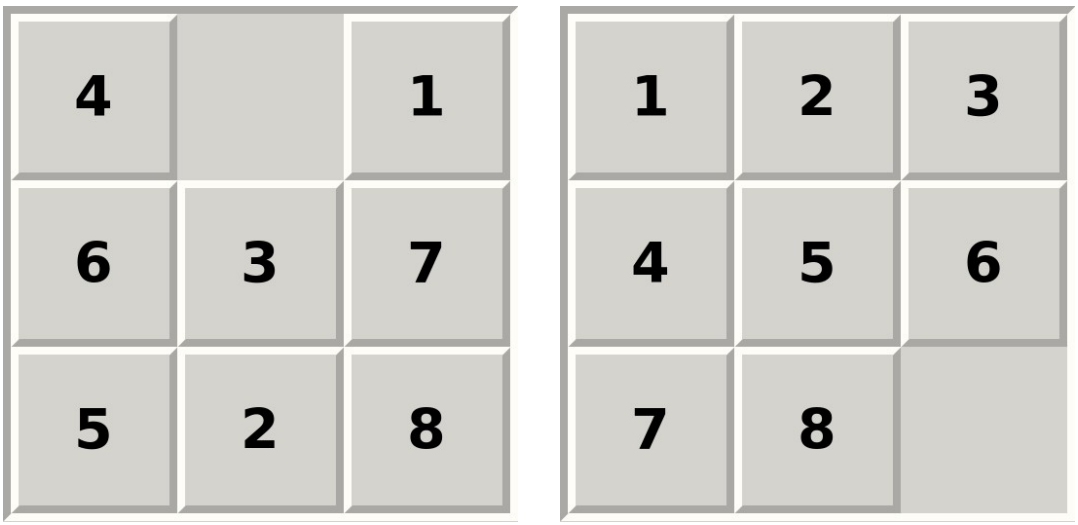
The n-puzzle is a well-known sliding-block puzzle, first introduced in the late 19th century for $n = 15$. The puzzle became a craze in the US, Canada and Europe in 1880 and continues to be popular even today. For over a century, the invention and popularisation of the puzzle were accredited to the famous American puzzle-maker Sam Loyd, although it was shown in 2006 by Jerry Slocum and Dic Sonneveld that he was not involved with either. The actual inventor of the puzzle was Noyes Chapman, a postmaster in Canastota, New York [1].

2.1 Rules and definitions

An instance of the the n-puzzle consists of a usually square board with tiles numbered from 1 to n and a single empty tile[1]. The *state* of the puzzle consists of the location of each of the n tiles and the empty tile. One can move tiles by sliding them into the empty tile, hence swapping the locations of the empty tile and one of its neighbours. We will refer to moves by the direction of the movement of the empty tile, being either *Left*, *Up*, *Right* or *Down*. The puzzle is said to be solved when a specified target state is reached. This target state is typically a configuration where the tiles are ordered from 1 to n, with the empty tile being either in the upper left or lower right corner. The *distance* between two states is the minimum amount of moves required to obtain one from the other.

An instance of the 8-puzzle, which is played on a 3×3 board with tiles numbered from 1 to 8, is shown in Figure 2.1.

[1] As the tiles plus the empty tile have to completely cover the board, it must hold that $n + 1 = w \times h$, w being the width and h the height of the board.



(a) An initial state (b) A target state

Figure 2.1: An instance of the 8-puzzle, consisting of an initial state and a target state. Pictures taken from [2].

2.2 Solvability

Not long after the 15-puzzle, which is played on a 4×4 board, attained popularity in the 1880's, numerous wealthy Americans offered prizes of up to $1000 for a solution of the so-called 14-15 puzzle, in which all the tiles are in the correct location but the tiles numbered 14 and 15 have been swapped. Just as when one removes a corner or edge cube from a Rubik's cube, rotates it and replaces it, this renders the puzzle unsolvable. In fact, exactly half of all possible states of the n-puzzle can be reached from a given initial state, as shown with a parity argument by Johnson and Story in 1879 [3]. Using this approach, it is trivial to determine for any given instance of the n-puzzle whether it is solvable or not.

2.3 Complexity

While determining whether an instance of the n-puzzle has a solution is trivial, determining whether it has a solution within a given amount of moves is far more complex. This problem was proved in 1986 to belong to the class of NP-complete problems by Ratner and Warmuth [4]. The amount of reachable states grows exponentially as one raises the dimensions of the puzzle, which easily results in a far too large search space. The 8-puzzle has $9!/2 = 181\,440$ reachable states, whereas the 15-puzzle, which is played on a 4×4 board, has $16!/2 \approx 10^{13}$ reachable states and the 24-puzzle (5×5) has approximately 10^{25} reachable states. While solving the 8-puzzle optimally is trivial with the use of modern computers[2] and random 15-puzzles can be solved optimally within milliseconds using the most advanced search algorithms, random 24-puzzles still take several hours to solve optimally even with the best hardware and software available [5]. One can imagine it

[2]The entire search space can actually be stored in memory, allowing one to simply look up the solution instead of having to compute it.

would be practically impossible for a computer to optimally solve an instance of, for instance, a 10×10 board, where the amount of reachable states has 158 digits.

Due to its complexity, the n-puzzle is a popular benchmark for new search algorithms in the field of artificial intelligence.

2.4 Wilson's puzzle graph

In 1974, Wilson published his research on graph puzzles, which are a generalisation of the n-puzzle when one considers it as a labeled graph with exactly one 'empty' label, in which one is allowed to swap the empty label with one of its neighbours. In his research, he proved that an arbitrary finite, simple, biconnected, non-polygonal, bipartite graph has two disjoint sets of connected states [6]. The single exception is the θ_0-graph or the "Tricky Six Puzzle", as seen in Figure 2.2, which has six such sets. The n-puzzle, for $n \geq 3$, is such a graph, thus having two disjoint sets of connected states. Two configurations of the 15-puzzle and the corresponding graph puzzles are shown in Figure 2.3. For further reading on Wilson's work on graph puzzles, we recommend [7].

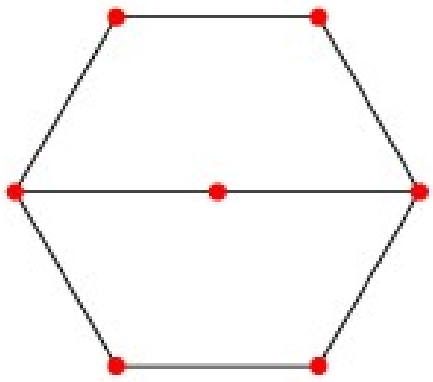


Figure 2.2: The θ_0-graph or the "Tricky Six Puzzle". Picture taken from [8].

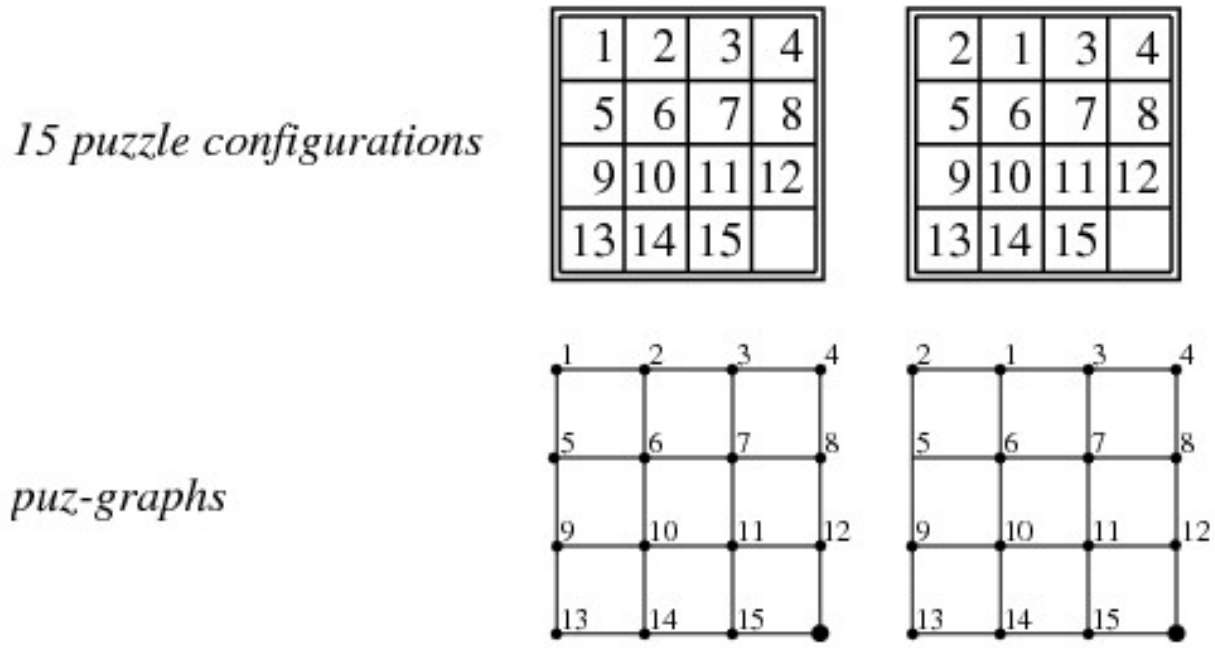


Figure 2.3: Two configurations of the 15-puzzle and their corresponding graph puzzles. Picture taken from [7].

Chapter 3

A* search

Perhaps *the* classical approach to solving the n-puzzle optimally is to use an algorithm called A* search, or A* for short, on the graph consisting of the different states of the puzzle. The A* algorithm and other variants of it, such as IDA*, are amongst the most well-known search algorithms in the field of artificial intelligence, and keep being used to address a wide variety of problems.

3.1 Dijkstra's algorithm

Problem 1. Given two nodes P and Q in a weighted graph, what is the path with the minimum total length that connects the two?

When handed Problem 1, most computer science graduates will probably first think of Dijkstra's 1959 shortest path algorithm [9], which was the most efficient known algorithm for solving the problem before A* search. As Dijkstra's algorithm and A* search are conceptually quite similar, we will first briefly explain the general idea of Dijkstra's algorithm before expanding it to A* search.

During its execution, Dijkstra's algorithm keeps track of which nodes it has visited, the *visited nodes*, and of which unvisited nodes are connected with a single branch to the set of visited nodes, the *candidate nodes*. For the candidate nodes, it also keeps track of the length of the shortest path from the start node to the candidate node, which we will call its *path cost*. The algorithm then proceeds as follows:

1. Visit the start node P.

2. Investigate the unvisited nodes connected to the last visited node. If a node was already a candidate node, update its path cost if a shorter path is made possible through the last visited node. If the node

was not yet a candidate node, add it to the set of candidate nodes and initialise its path cost as the path cost to the last visited node plus the weight of the branch connecting the two nodes.

3. Visit the candidate node with the minimal path cost, then return to step 2. Repeat this process until the target node Q has been visited, at which point the shortest path from P to Q has been found.

3.2 Informed search

Dijkstra's algorithm finds an optimal solution[1]. However, it does not have any sense of direction and will visit *every* node closer to the start node than the goal node before it reaches the goal. If one were to apply Dijkstra's algorithm to finding the shortest route by train from Leiden to Porto, it does not do any good to even consider visiting Copenhagen[2] because no sensible route will ever visit it — however, because it is closer to Leiden than Leiden is to Porto, Dijkstra's algorithm will visit it, along with numerous other similar cases, before returning the shortest path.

The A* search algorithm, designed in 1968 by Hart, Nilsson and Raphael, makes up for this shortcoming by using knowledge specific to the problem, beyond the definition of the problem, to steer the search in a good direction. Instead of selecting the candidate node with the minimal path cost, A* also considers the estimated distance from the candidate node to the goal, which we will call its *heuristic value*. It can thus be called an *informed search* algorithm. A* search still finds an optimal solution, on the condition that the *heuristic function*, or *heuristic* for short, which computes the heuristic value of a node, is both *admissible* and *consistent*. A heuristic is admissible if it never overestimates the distance to the target node. It is consistent if the estimated total distance, including the path cost, does not decrease while traversing the state space: for arbitrary nodes n and n', connected by branch a, it must hold that $h(n) \leq h(n') + c(n, a, n')$, $h(x)$ being the heuristic value of node x and $c(x, y, x')$ being the cost of traversing from node x to node x' using branch y. It should be noted that a consistent heuristic is always admissible. Thus, consistency is a stronger constraint than admissibility[3].

Note that the search graph Dijkstra's algorithm and A* search are operating on is defined implicitly. Unless a node is visited or is a candidate node, the algorithm has no knowledge of it and has no need to store it in memory. As such, the size of the search space depends on the efficiency on the algorithm.

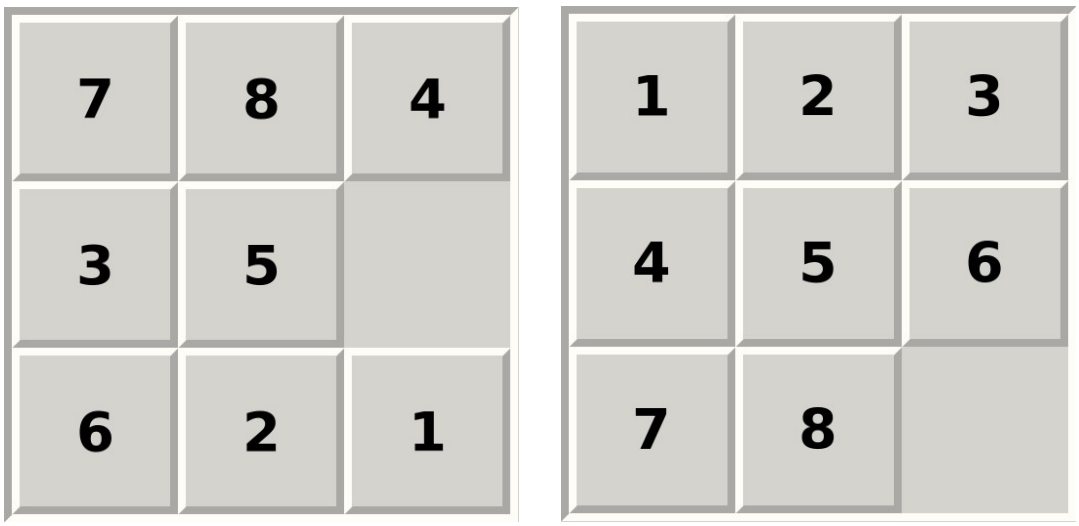
[1] That is to say, Dijkstra's algorithm finds an optimal solution as long as the branch weights are nonnegative. In fact, most search algorithms start behaving in a weird fashion when one considers negative distances.

[2] Naturally, we do not want to discourage anyone from visiting Copenhagen altogether — it is a nice place. We merely suggest that it may not be a good idea to plan this visit when one is in a rush to reach Portugal, which lies in the exact opposite direction.

[3] In practice, it is hard to find a heuristic that is admissible but not consistent unless one is explicitly trying to do so.

3.3 Heuristic functions

In this section, we will describe several consistent heuristic functions using the example in Figure 3.1. The distance between the initial state and the target state in the figure is 27.



(a) The initial state (b) The target state

Figure 3.1: A possible initial state of the 8-puzzle with its target state. Pictures taken from [2].

Heuristic 1. Misplaced tiles: The number of tiles that are not in their target position, excluding the empty tile. In Figure 3.1a, only one of the tiles is in its target position. Thus, its heuristic value is 7. Note that the algorithm is now also discouraged from moving the tile numbered 5, although this will have to be done at some point in time if one is to solve the puzzle.

Heuristic 2. Manhattan distance[4]: The sum of the horizontal and vertical distances of the tiles to their target positions, again excluding the empty tile. This is more accurate than Misplaced tiles, which does not consider the distance between a tile and its target position. In Figure 3.1, it can be easily seen that the tile numbered 1 will have to move at least four times — twice horizontally and twice vertically — but Misplaced tiles only adds a single move. The Manhattan distance heuristic gives a total heuristic value of 19 for tiles 1 to 8.

Heuristic 3. X-Y: The minimal number of horizontal moves to move all tiles to their target column, added to the minimal number of vertical moves to move all tiles to their target row. This is more accurate than the Manhattan distance. In Figure 3.1a, the tiles numbered 2, 5 and 8 are already in their target column. However, the tiles numbered 1, 3, 4 and 6 will have to cross this column, for which either the 2, 5 or 8 will have to be moved to another column — and later returned. The Manhattan distance does not take this into account, but the X-Y heuristic does. The X-Y heuristic has the added benefit that, because it does not have to differentiate between numbers that belong in the same column or row, depending on whether it is computing the horizontal or vertical distance, the number of possible states is drastically reduced and can be precomputed and stored in memory for smaller puzzles. Using dynamic programming, it can be shown that the X-Y heuristic only has to

[4]The Manhattan distance is also commonly called the *city block distance* or *taxicab distance*.

store 105 different states for the 8-puzzle, and 24 964 for the 15-puzzle. For the state in the figure, the heuristic value using the X-Y heuristic is 23.

The heuristic functions described above produce different heuristic values. The closer the heuristic value is to the actual distance, the fewer suboptimal paths will be considered by A* before finding a solution. When deciding which heuristic function to use, one should consider the accuracy and complexity of a heuristic function. Although a heuristic function might be more accurate, if it is also very complex and takes a long time to compute the heuristic value, one might be better off using a less accurate but faster heuristic function. For our research, we mainly used the X-Y heuristic function.

A more complex heuristic function is to use pattern databases. The idea behind pattern databases is to solve simpler subproblems, of which the solutions can be stored in memory, and combining those solutions to solve the original problem. Heuristics of this kind are much more accurate than the ones mentioned above. For further reading on pattern databases, we recommend [10].

3.4 Variants of A* search

The main drawback of A* search is its memory usage. As it keeps all of the visited and candidate nodes in memory, A* search will usually run out of memory before finding a solution when presented with a large-scale problem. Numerous solutions have been suggested to counter this issue.

Adding the concept of iterative deepening to A* search leads us to **iterative-deepening A***, or IDA* for short. IDA* performs a depth-first search up to a certain limit on the sum of the path cost and the heuristic value. When a solution cannot be found within the current bounds, the limit is increased. This is repeated until a target node has been found. During execution, IDA* only needs to store the nodes along the path it is currently exploring, resulting in a huge gain in memory usage. However, as it must regenerate the same nodes every cycle, it also takes more time to find a solution.

Somewhere between remembering everything and remembering practically nothing is **simplified memory-bounded A***, or SMA* for short, which remembers as much as it can. It behaves exactly as A* until memory is full. At this point, it forgets the worst leaf node it has found so far and backs up its value to its parent. In this way, we still know how worthwhile it is to regenerate this subtree and might get back to it later, when all other paths look worse than the forgotten subtree.

For further reading on A* and related algorithms, we recommend [5].

Chapter 4

SAT solvers

In this chapter, we will take a closer look at the Boolean Satisfiability Problem, which is a problem originating from the field of propositional logic, and algorithms dedicated to solve this problem.

4.1 The Boolean Satisfiability Problem

In propositional logic, a Boolean formula consists of Boolean variables, which can have the values *True* or *False*, and the operators AND (conjunction, $\wedge$), OR (disjunction, $\vee$) and NOT (negation, $\neg$)[1]. For the sake of conciseness, other derived operators are also commonly used, such as the implication operator ($a \rightarrow b = \neg a \vee b$). An example of a Boolean formula is given in Example 1.

Example 1. $((a \wedge b) \vee c) \rightarrow (d \wedge \neg b)$

The Boolean Satisfiability Problem, also known as SATISFIABILITY for short, is a problem from the field of propositional logic. The SATISFIABILITY problem can be formulated as in Problem 2.

Problem 2. Given a Boolean formula ϕ, does there exist a valuation of the Boolean variables occurring in the formula such that ϕ evaluates to *True*?

For the Boolean formula in Example 1 this may not be very difficult — assigning *False* to every variable satisfies the entire formula — but in general this is not an easy task. In 1971, Cook proved SATISFIABILITY to be NP-complete by showing that an instance of an arbitrary problem in NP can be reduced to an instance of the Boolean Satisfiability Problem [11].

There are many variations of SATISFIABILITY with more restrictions. For instance, CNF-SAT, or simply SAT, is the problem of determining the satisfiability of a Boolean formula in conjunctive normal form, CNF for

[1] Note that one can obtain each of AND and OR from the other combined with NOT. For instance, $a \wedge b = \neg(\neg a \vee \neg b)$.

short. A Boolean formula in CNF is a conjunction of clauses, where each clause is a disjunction of literals, a literal being either a Boolean variable or its negation. By adding the restriction that each clause must have exactly three literals, we obtain 3CNF-SAT, or 3-SAT for short. Despite the added restrictions, however, both SAT and 3-SAT remain NP-complete problems.

Also closely related to SAT are the Constraint Satisfaction Problems, or CSPs for short. While these also consist of variables and constraints, they also contain a set of domains of values of the variables, which may very well differ from the Boolean domain $\{0, 1\}$, and the constraints may also consist of any subset of k variables and a any k-ary relation on their corresponding domains.

4.2 SAT solvers

A SAT solver is an algorithm designed for solving instances of SAT. We will also use the term "SAT solver" to denote a computer program which implements such an algorithm.

One of the most well-known algorithms for solving the SAT problem is the backtracking-based Davis-Putnam-Logemann-Loveland algorithm, or DPLL for short. The algorithm first chooses a literal, assigns a truth value to it, simplifies the formula by propagating the value of the chosen literal, and recursively checks if the resulting simplified formula is satisfiable. If this is not the case, it attempts the same check again with the opposing value for the chosen literal. Additionally, the algorithm uses *unit propagation*, which checks for clauses containing a single unassigned literal and assigns to the variable the value required to make the literal true, and *pure literal elimination*, which checks for variables that only occur with one polarity in the formula and assigns to the variable the value required to make this polarity true, thus making all the clauses containing them true.

Although the DPLL algorithm is currently over 50 years old, it is still used as the basis for many modern SAT solvers, which improve upon it with the use of, amongst others, *Conflict-Driven Clause Learning*, or CDCL for short. When a conflict occurs during the backtracking process — when a sequence of truth value assignments causes the formula to not be satisfiable — a SAT solver with CDCL will determine which combination of variables is responsible for the conflict and adds an additional constraint to the SAT instance to prevent a future occurrence of the same conflict. This is also true for the SAT solver we chose for our research. We decided to use Plingeling by Armin Biere, of the Johannes Kepler University Linz [12]. We chose for Plingeling, the parallel version of Lingeling, because of its good performance in recent competitions such as the SAT-Race 2015 [13]. Furthermore, we chose Plingeling over the sequential Lingeling to fully utilise the hardware of the machine we used to run our experiments.

For further reading on SAT and SAT solvers, we recommend [14, 15].

Chapter 5

From Sokoban to SAT

Naoyuki Tamura [16] of Kobe University, Japan, wrote a program [17] in the constraint programming language Copris [18] that reads an instance of the puzzle game Sokoban, which is somewhat similar to the n-puzzle, as input and translates this into a Constraint Satisfaction Problem, or CSP for short, which was mentioned in Chapter 4. This CSP is in turn translated to an instance of SAT. In this chapter, we will only document the first translation.

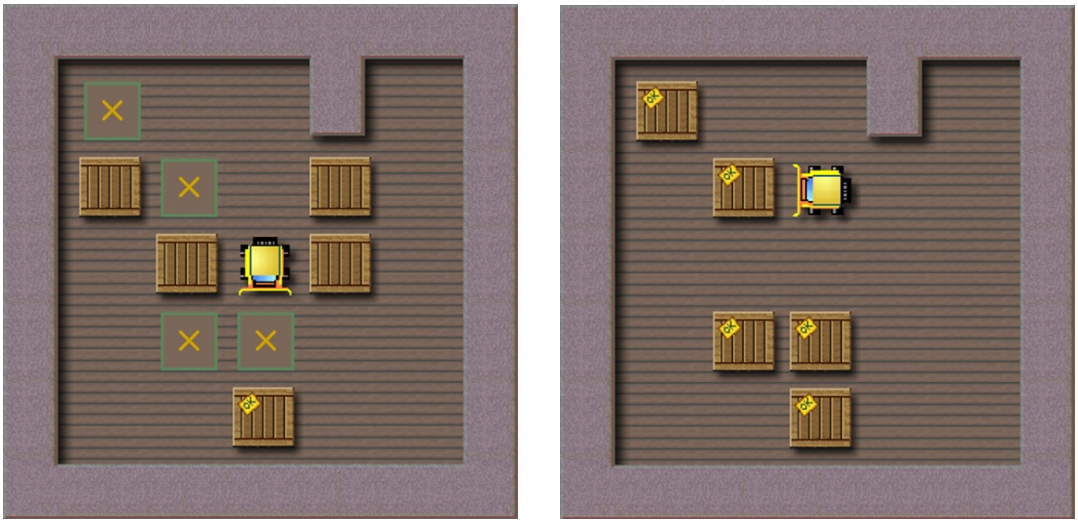
5.1 Sokoban

Sokoban [19] is a puzzle game that was developed in 1981 by Hiroyuki Imabayashi. In Sokoban, the *player*[1] controls a character, Sokoban[2]. The player moves around in a warehouse, which is a two-dimensional array of tiles, and interacts with the *boxes* that are scattered across the warehouse. The objective of the game is to push the boxes onto a set of special tiles, the *goals* or *goal tiles*. An instance of Sokoban and its objective state are shown in Figure 5.1.

The player may move orthogonally and may not move through *walls* or boxes. When the player moves onto a tile that contains a box, he attempts to push the box a single tile in the direction the player is moving. This attempt succeeds if the tile where the box would end up in is not a wall tile and does not already contain a box. The puzzle is said to be solved when every goal tile contains a box.

While the solvability of the n-puzzle, as mentioned in Section 2.2, is trivial, determining the solvability of an instance of Sokoban is significantly more complex.

[1] From here on, "the player" is used to refer to the character that is controlled by the physical player, not the physical player himself.
[2] "Sokoban" can be roughly translated as "warehouse keeper".



(a) An initial state (b) The corresponding objective state

Figure 5.1: An instance of Sokoban and its objective state. The player is depicted by a bulldozer. Pictures taken from [20].

5.2 Translation

Because it is unknown beforehand how many pushes it will at least take to solve a puzzle[3], Tamura's program builds a CSP for a fixed amount of pushes. If this CSP does not have a solution, it builds a CSP with a doubled push limit. This step is repeated until a solution is found. If there is no solution, the program will keep on running until it is externally halted.

5.2.1 Variables

The following variables are used in the translation of a puzzle with a height of m tiles and a width of n tiles, where the program is currently looking for a solution with a minimum of min and a maximum of max pushes[4]:

1. An integer $steps \in [min, max]$ denoting the amount of pushes that has been used to solve the puzzle. This effectively caps the amount of pushes that may be used during this attempt at solving the puzzle.
2. For all steps $s \in [0, max]$, an integer $pi(s)$ denoting the vertical coordinate of the player at step s.
3. For all steps $s \in [0, max]$, an integer $pj(s)$ denoting the horizontal coordinate of the player at step s.
4. For all steps $s \in [0, max]$ and for all tiles (i, j), a Boolean[5] $p(s, i, j)$ denoting whether the player's location is tile (i, j) at step s.
5. For all steps $s \in [0, max]$ and for all non-wall tiles (i, j), a Boolean $b(s, i, j)$ denoting whether the tile (i, j) contains a box at step s.

[3]Note that we only count the amount of times the player actually pushes a box. Moves that do not affect boxes are disregarded.
[4]From here on, "step n" denotes the situation after having pushed n boxes.
[5]Note that Booleans are used as integers throughout the constraints, having value 1 when true and value 0 when false.

6. For all steps $s \in [0, max]$, a Boolean $g(s)$ denoting whether the puzzle has been solved at step s or at an earlier step.

7. For all steps $s \in [0, max-1]$, integers $di(s)$ and $dj(s) \in \{-1, 0, 1\}$, respectively denoting the vertical and horizontal pushing direction of the player during the transition from step s to step $s+1$. As the player may only move orthogonally, at least one of the variables must be 0.

8. For all steps $s \in [0, max-1]$ and for all non-wall tiles (i, j), a Boolean $r(s, i, j)$ denoting whether the tile (i, j) is reachable[6] for the player at step s.

9. For all steps $s \in [0, max-1]$ and for all non-wall tiles (i, j), an integer $x(s, i, j)$ used in defining the reachability variable r.

10. For all steps $s \in [0, max-1]$ and for all non-wall tiles (i, j) and their neighbouring non-wall tiles (k, ℓ), a Boolean $e(s, i, j, k, \ell)$ used in defining the reachability variable r. It denotes whether the player can move[7] from tile (i, j) to tile (k, ℓ) at step s.

11. For all steps $s \in [0, max-1]$ and for all non-wall tiles (i, j), an integer $in(s, i, j)$ used in defining the reachability variable r. It denotes whether tile (i, j) is reachable for the player at step s, but is not the tile the player is currently standing on.

5.2.2 Constraints

The variables declared in Section 5.2.1 are combined as follows to implement the rules of Sokoban:

1. Step declaration

   (a) For all steps $s \in [0, max]$ and for all tiles (i, j): $(p(s, i, j) = 1) \Leftrightarrow (pi(s) = i \wedge pj(s) = j)$. This synchronizes the two ways of tracking the player position and enforces that the player is standing on a single tile at every step.

   (b) For all steps $s \in [0, max]$ and for all wall tiles (i, j): $p(s, i, j) = 0$. This enforces that the player can never stand on a wall tile.

   (c) For all steps $s \in [0, max]$: $(g(s) = 1) \Leftrightarrow And(gs)$[8], where gs is the set of formulas $b(s, i, j) = 1$ for all goal tiles (i, j). This defines the goal of the puzzle.

   (d) For all steps $s \in [0, max]$: $(steps \leq s) \rightarrow (g(s) = 1)$. This defines the number of pushes needed to solve the puzzle.

[6] A tile is reachable if the player can move onto it without moving through walls or boxes, and without pushing boxes.
[7] Note that to move from tile a to tile b, both tiles must be reachable for the player.
[8] The function And denotes the logical $\wedge$ for multiple variables, having value 1 if and only if all of its clauses have value 1, and having value 0 in all other cases.

2. Initial configuration

   (a) For the tile (i, j) the player is standing on in the initial configuration: $p(0, i, j) = 1$. For all other non-wall tiles (i, j): $p(0, i, j) = 0$. This defines the initial position of the player.

   (b) For all non-wall tiles (i, j) that contain a box in the initial configuration: $b(0, i, j) = 1$. For all other non-wall tiles (i, j): $b(0, i, j) = 0$. This defines the initial position of the boxes.

3. Reachable tiles

   (a) For all steps $s \in [0, max - 1]$ and for all non-wall tiles (i, j): $in(s, i, j) = Add(nb)$[9], where nb is the set of variables $e(s, k, \ell, i, j)$ for all neighbouring non-wall tiles (k, ℓ). This enforces that, if a tile is reachable, there is only one way of reaching it — $in(s, i, j) \in \{0, 1\}$, hence there can not be more than one variable $e(s, *, *, i, j)$ with value 1.

   (b) For all steps $s \in [0, max - 1]$ and for all non-wall tiles (i, j) and their neighbouring non-wall tiles (k, ℓ): $(e(s, k, \ell, i, j) = 1) \rightarrow (x(s, k, \ell) > x(s, i, j))$. This enforces that the player can only move forward, as $x(s, k, \ell)$ can not be both greater and less than $x(s, i, j)$.

   (c) For all steps $s \in [0, max - 1]$ and for all non-wall tiles (i, j): $(p(s, i, j) = 1) \rightarrow (in(s, i, j) = 0)$. This enforces that the player can not move to his current position.

   (d) For all steps $s \in [0, max - 1]$ and for all non-wall tiles (i, j): $(p(s, i, j) = 0) \rightarrow ((in(s, i, j) = 1) \vee (x(s, i, j) = 0))$. This enforces that for any tile where the player is not currently at, either the tile can be reached ($in(s, i, j,) = 1$), or the tile can not be moved from – as $(x(s, i, j) = 0) \rightarrow (e(s, i, j, *, *) \neq 1)$ follows from Constraint 3b.

   (e) For all steps $s \in [0, max - 1]$ and for all non-wall tiles (i, j): $(b(s, i, j) = 1) \rightarrow (in(s, i, j) = 0)$. This enforces that the player cannot move through boxes.

   (f) For all steps $s \in [0, max - 1]$ and for all non-wall tiles (i, j): $(r(s, i, j) = 1) \Leftrightarrow ((p(s, i, j) = 1) \vee (in(s, i, j) > 0))$. This effectively combines the previous statements to define the reachable tiles for the player.

4. Transition between two steps

   (a) For all steps $s \in [0, max - 1]$: $g(s) = 1 \Leftrightarrow (di(s) = 0 \wedge dj(s) = 0)$. This makes the player stand still after the goal has been reached in order to minimize the computing time.

   (b) For all steps $s_0 \in [0, max - 1]$ and $s_1 = s_0 + 1$: $(di(s_0) = 0 \wedge dj(s_0) = 0) \Leftrightarrow (pi(s_0) = pi(s_1) \wedge pj(s_0) = pj(s_1))$. This enforces that the player's coordinates do not change if he has not moved.

[9] The function *Add* denotes the arithmetic $\sum$ function. Its value is the sum of the value of all its clauses.

(c) For all steps $s_0 \in [0, max-1]$ and $s_1 = s_0 + 1$, and for all non-wall tiles (i, j): $(p(s_1, i, j) = 1) \rightarrow ((p(s_0, i, j) = 1 \wedge g(s_0) = 1) \vee (b(s_0, i, j) = 1 \wedge Or(push))$[10], where *push* is the set of formulas $(di(s_0) = di) \wedge (dj(s_0) = dj) \wedge (r(s_0, i - di, j - dj) = 1) \wedge (b(s_0, i + di, j + dj) = 0)$, with $(di, dj) \in [(1, 0), (-1, 0), (0, 1), (0, -1)]$, where both $(i - di, j - dj)$ and $(i + di, j + dj)$ are non-wall tiles. This enforces that when the player moves somewhere, either he was already standing there and the puzzle is already solved, or the tile contained a box which is pushed by the player.

(d) For all steps $s_0 \in [0, max-1]$ and $s_1 = s_0 + 1$, and for all non-wall tiles (i, j): $(b(s_1, i, j = 1) \Leftrightarrow (((b(s_0, i, j) = 1) \wedge (p(s_1, i, j) = 0)) \vee Or(push))$, where *push* is the set of formulas $(di(s_0) = di) \wedge (dj(s_0) = dj) \wedge (b(s_0, i - di, j - dj) = 1) \wedge (p(s_1, i - di, j - dj) = 1)$, with $(di, dj) \in \{(1, 0), (-1, 0), (0, 1), (0, -1)\}$, where both $(i - di, j - dj)$ and $(i + di, j + dj)$ are non-wall tiles. This enforces that the only tiles containing a box at the new step are those that already contained a box and haven't been interacted with, and the tile a box was pushed into by the player.

In his translation, Tamura also defines *dead blocks* and *bad blocks*, which are used to provide more constraints to enable the SAT solver to solve the problem more efficiently. We left these out in our description of the translation as they are not essential for solving the puzzle.

5.3 Example

To illustrate how the algorithm works, we analyse a trivial example. This example is shown in Figure 5.2. Its solution is to push once to the right.

```
   0  1  2  3  4
0  #  #  #  #  #
1  #  @  $  .  #
2  #  #  #  #  #
```

Figure 5.2: Trivial example of Sokoban; `#` represents a wall tile, `@` represents the player, `$` represents a box and `.` represents a goal tile.

The translation of the example in Figure 5.2 results in the CSP in Table 5.1. This problem can be solved by assigning values to the variables as shown in Table 5.2.

[10]The function *Or* denotes the logical $\vee$ function for multiple variables, having value 1 if and only if at least one of its clauses has value 1, and having value 0 in the opposite case.

	Variables				
1	$steps \in [0,1]$	2	$\forall s \in [0,1] : pi(s) \in [0,2]$	4	$\forall s \in [0,1], \forall i \in [0,2], \forall j \in [0,4] : p(s,i,j) \in \{0,1\}$
6	$\forall s \in [0,1] : g(s) \in \{0,1\}$	3	$\forall s \in [0,1] : pj(s) \in [0,4]$	5	$\forall s \in [0,1], \forall j \in [1,3] : b(s,1,j) \in \{0,1\}$
7	$di(0) \in [-1,1]$	8	$\forall j \in [1,3] : r(0,1,j) \in \{0,1\}$	9	$\forall j \in [1,3] : x(0,1,j) \in [0,3]$
7	$dj(0) \in [-1,1]$	11	$\forall j \in [1,3] : in(0,1,j) \in \{0,1\}$	10	$e(0,1,2,1,1), e(0,1,1,1,2), e(0,1,3,1,2), e(0,1,2,1,3) \in \{0,1\}$
	Constraints				
2a	$p(0,1,1) = 1$	2a	$\forall i \in \{1,3\} : p(0,1,i) = 0$	1b	$\forall s \in [0,1], \forall i \in [0,2] \forall j \in [0,4] \mid (i \in \{0,2\} \vee j \in \{0,4\}) : p(s,i,j) = 0$
2b	$b(0,1,2) = 1$	3a	$in(0,1,1) = e(0,1,2,1,1)$	3a	$in(0,1,2) = e(0,1,1,1,2) + e(0,1,3,1,2)$
2b	$\forall i \in \{1,3\} : b(0,1,i) = 0$	3e	$\forall i \in [1,3] : b(0,1,i) = 1 \rightarrow in(0,1,i) = 0$	1a	$\forall s \in [0,1], \forall i \in [0,2], \forall j \in [0,4] : (p(s,i,j) \Leftrightarrow pi(s) = i \wedge pj(s) = j)$
3f	$\forall i \in [1,3] : r(0,1,i) \Leftrightarrow p(0,1,i) = 1 \vee in(0,1,i) > 0$	3b	$e(0,1,1,1,2) = 1 \rightarrow x(0,1,1) > x(0,1,2)$	4b	$di(0) = 0 \wedge dj(0) = 0 \rightarrow pi(0) = pi(1) \wedge pj(0) = pj(1)$
3b	$e(0,1,2,1,1) = 1 \rightarrow x(0,1,2) > x(0,1,1)$	3b	$e(0,1,2,1,3) = 1 \rightarrow x(0,1,2) > x(0,1,3)$	3b	$e(0,1,3,1,2) = 1 \rightarrow x(0,1,3) > x(0,1,2)$
3c	$\forall i \in [1,3] : p(0,1,i) \rightarrow in(0,1,i) = 0$	1d	$\forall s \in [0,1] : steps \leq s \rightarrow g(s) = 1$	3d	$\forall i \in [1,3] : p(0,1,i) = 0 \rightarrow in(0,1,i) = 1 \vee x(0,1,i) = 0$
1c	$\forall s \in [0,1] : g(s) = 1 \Leftrightarrow b(s,1,3) = 1$	4d	$b(1,1,1) = 1 \Leftrightarrow ((b(0,0,0) = 1 \wedge p(1,1,1) = 0) \vee (di(0) = 0 \wedge dj(0) = -1 \wedge b(0,1,2) = 1 \wedge p(1,1,2) = 1))$		
4a	$g(0) = 1 \Leftrightarrow di(0) = 0 \wedge dj(0) = 0$	4d	$b(1,1,3) = 1 \Leftrightarrow ((b(0,1,3) = 1 \wedge p(1,1,3) = 0) \vee (di(0) = 0 \wedge dj(0) = 1 \wedge b(0,1,2) = 1 \wedge p(1,1,2) = 1))$		
4d	$b(1,1,2) = 1 \Leftrightarrow ((b(0,1,2) = 1 \wedge p(1,1,2) = 0) \vee (di(0) = 0 \wedge dj(0) = -1 \wedge b(0,1,3) = 1 \wedge p(1,1,3) = 1) \vee (di(0) = 0 \wedge dj(0) = 1 \wedge b(0,1,1) = 1 \wedge p(1,1,1) = 1))$				
4c	$p(1,1,2) = 1 \rightarrow (p(0,1,2) = 1 \wedge g(0) = 1) \vee (b(0,1,2) = 1 \wedge ((di(0) = 0 \wedge dj(0) = -1 \wedge r(0,1,3) = 1 \wedge b(0,1,1) = 0) \vee (di(0) = 0 \wedge dj(0) = 1 \wedge r(0,1,1) = 1 \wedge b(0,1,3) = 0)))$				
4c	$\forall i \in \{1,3\} : p(1,1,1) = 1 \rightarrow (p(0,1,1) = 1 \wedge g(0) = 1) \vee (b(0,1,1) = 1)$				

Table 5.1: The resulting CSP after applying the translation in Section 5.2 to the Sokoban instance in Figure 5.2. For the sake of reference, the number of the corresponding variable or constraints in the translation described in Section 5.2 has been added to the left of each variable and constraint. In total, the CSP consists of 58 variables and 91 constraints.

$steps$	1	$pi(0)$	1	$pi(1)$	1	$pj(0)$	1	$pj(1)$	2	$p(0,1,1)$	1	$p(1,1,2)$	1
$di(0)$	0	$b(0,1,1)$	0	$b(0,1,2)$	1	$b(0,1,3)$	0	$b(1,1,1)$	0	$b(1,1,2)$	0	$b(1,1,3)$	1
$dj(0)$	1	$r(0,1,1)$	1	$r(0,1,2)$	0	$r(0,1,3)$	0	$x(0,1,1)$	3	$x(0,1,2)$	0	$x(0,1,3)$	0
$in(0,1,1)$	0	$in(0,1,2)$	0	$in(0,1,3)$	0	$e(0,1,1,1,2)$	0	$e(0,1,2,1,1)$	0	$e(0,1,2,1,3)$	0	$e(0,1,3,1,2)$	0
$g(0)$	0	$g(1)$	1										

Table 5.2: The solution of the CSP in Table 5.1. Note that only the variables $p(s,i,j)$ have been included that have value 1. The 28 variables not appearing in the table have been assigned value 0.

Chapter 6

Translating the n-puzzle to SAT

After studying Tamura's translation of Sokoban to an instance of the Constraint Satisfiability Problem, or CSP for short, as described in Chapter 5, we propose a similar translation of the n-puzzle to an instance of CSP. Following Tamura's example, we use the programming language Copris [18], which takes care of the translation from a CSP to an instance of SAT. Therefore, we will not elaborate on this second part of the translation. Likewise, our implementation of this translation builds a CSP for a fixed amount of moves and keeps doubling the move limit until a solution is found or it is manually halted.

6.1 Variables

The following variables are used in the translation of a puzzle with a height of h tiles and a width of w tiles, where the algorithm is currently looking for a solution with a minimum of min and a maximum of max moves:

1. An integer $steps \in [min, max]$ denoting the amount of moves that has been used to solve the puzzle.
2. For all steps $s \in [0, max]$, integers $empty_i(s) \in [0, h-1]$ and $empty_j(s) \in [0, w-1]$, respectively denoting the vertical and horizontal coordinates of the empty tile at step s. These do not change anymore once the puzzle has been solved — that is to say, for $s > steps$.
3. For all steps $s \in [0, max]$ and for all tiles (i, j), an integer $num(s, i, j) \in [0, h \times w - 1]$ denoting which number or label the tile contains at step s. The number 0 represents the empty tile.
4. For all steps $s \in [0, max]$, a Boolean $goal(s)$ denoting whether the puzzle has been solved at step s or at an earlier step.
5. For all steps $s \in [0, max - 1]$, integers $d_i(s), d_j(s) \in \{-1, 0, 1\}$, respectively denoting the vertical and horizontal movement of the empty tile during the transition from step s to step $s + 1$.

For a formula for a solution consisting of s steps on a board with n tiles, this results in $1 + (s+1) * (n+3) + 2s$ variables. For instance, the formula for the 8-puzzle with 32 steps has 461 variables, and the formula for the 15-puzzle with 64 steps has 1364 variables.

6.2 Essential constraints

The variables declared in Section 6.1 are combined as follows to implement the rules of the n-puzzle, where h still represents the height and w the width of the puzzle:

1. Initial configuration:

   For all tiles (i, j): $num(0, i, j)$ is the number of the tile (i, j) in the initial configuration. The empty tile is numbered as 0.

2. Step declaration:

   (a) For all steps $s \in [0, max]$ and for all tiles (i, j): $(num(s, i, j) = 0) \Leftrightarrow (empty_i(s) = i \wedge empty_j(s) = j)$. This synchronizes the two ways of tracking the location of the empty tile.

   (b) For all steps $s \in [0, max]$: $Alldifferent(numbers)$, where $numbers$ is the set of variables $num(s, i, j)$ for all tiles (i, j). The function $Alldifferent(S)$ evaluates to true if and only if all of the elements in the set S have different values. This enforces that the tiles all contain different numbers.

   (c) For all steps $s \in [0, max]$: $(goal(s) = 1) \Leftrightarrow (And(gs))$, where gs is the set of formulas $num(s, i, j) = 1 + j + w \times i$ for all tiles except the bottom right tile, which should contain the 0 — this automatically follows when combining this constraint and constraint 2b.

   (d) For all steps $s \in [0, max]$: $(steps \leq s) \rightarrow (goal(s) = 1)$. This defines the number of moves needed to solve the puzzle.

3. Movement — transition from step s to step $s + 1$:

   (a) For all steps $s \in [0, max - 1]$: $d_i(s) = 0 \vee d_j(s) = 0$. This enforces that the player only makes orthogonal moves.

   (b) For all steps $s \in [0, max - 1]$: $goal(s) = 1 \Leftrightarrow (d_i(s) = 0 \wedge d_j(s) = 0)$. This enforces that no more moves are made once the puzzle has been solved and that moves have to be made as long as the puzzle has not been solved.

   (c) For all steps $s \in [0, max - 1]$: $(d_i(s) = 0 \wedge d_j(s) = 0) \rightarrow (empty_i(s) = empty_i(s + 1) \wedge empty_j(s) = empty_j(s + 1))$. This enforces that, if one has not moved, the empty tile should not have moved.

   (d) For all steps $s \in [0, max - 1]$ and for the moves $(i, j) \in \{(1, 0), (-1, 0), (0, 1), (0, -1)\}$: $(d_i(s) = i \wedge d_j(s) = j) \rightarrow (empty_i(s) + i \geq 0 \wedge empty_i(s) + i \leq h \wedge empty_j(s) + j \geq 0 \wedge empty_j(s) + j \leq$

$w \wedge empty_i(s) + i = empty_i(s+1) \wedge empty_j(s) + j = empty_j(s+1)$). This enforces that the player only makes valid moves and that the empty tile actually moves.

(e) For all steps $s \in [0, max-1]$ and for all tiles (i, j): $\neg((empty_i(s) = i \wedge empty_j(s) = j) \vee (empty_i(s+1) = i \wedge empty_j(s+1) = j)) \rightarrow (num(s, i, j) = num(s+1, i, j))$. This enforces that, if the tile was not the empty tile during either one of the two steps, its contents should be unchanged. Combined with constraint 2b and constraint 3d, this ensures that the empty tile actually swaps places with the tile next to it — after all, there is only one number left to place, only two tiles are unchanged and one of them already contains the empty tile.

For a formula for a solution consisting of s steps on a board with n tiles, this results in $n + (s+1) * (n+3) + s * (n+7)$ constraints. For instance, the formula for the 8-puzzle with 32 steps has 917 essential constraints, and the formula for the 15-puzzle with 64 steps has 2723 essential constraints.

6.3 Additional constraints

On top of the essential constraints described in Section 6.2, we have experimented with several additional constraints in order to further restrict the possibilities and thus speed up the process:

- Not allowing to undo the previous move, as it does not make any sense to do so. If there exists a solution consisting of n moves, where move i is undone by move $i+1$, the state after move $i+1$ is the same as the state after move $i-1$. By removing moves i and $i+1$, one would obtain a solution consisting of $n-2$ moves. Furthermore, this restricts the number of possible moves at every step, thus reducing the size of the search space.

  This can be done by adding, for all steps $s \in [1, max-1]$: $d_i(s-1) = 1 \rightarrow d_i(s) \neq -1$, $d_i(s-1) = -1 \rightarrow d_i(s) \neq 1$, $d_j(s-1) = 1 \rightarrow d_j(s) \neq -1$, $d_j(s-1) = -1 \rightarrow d_j(s) \neq 1$.

- Not allowing moves which alter "completed" blocks — consecutive rows or columns where all the tiles are in their target position, rows starting from the top and columns starting from the left. This makes the algorithm focus on the remainder of the puzzle, which is a smaller, simpler instance of the n-puzzle.

  This can be done for rows by adding, for all steps $s \in [0, max-1]$ and for all heights $cap \in [1, h-2]$, where h is the height of the puzzle: $And(complete(s, cap)) \rightarrow And(complete(s+1, cap))$, where $complete(s, cap)$ is the set of formula $num(s, i, j) = 1 + j + w \times i$ for all $i \in [0, cap]$ and all $j \in [0, w]$, w being the width of the puzzle. The corresponding constraints for columns can be constructed analogously. Note that this constraint does change the optimality of the solutions found.

For a formula for a solution consisting of s steps on a square board with n tiles, this results in $4(s-1) + 2s(\sqrt{n}-2)$ additional constraints. For instance, the formula for the 8-puzzle with 32 steps has 188 additional constraints, and the formula for the 15-puzzle with 64 steps has 508 additional constraints.

Chapter 7

Evaluation

In this chapter, we compare the execution times of implementations of the algorithms described in the previous chapters. We first describe the programs and machines we used for our evaluation and then compare their performance on puzzles of different degrees of difficulty.

7.1 Programs and machines

The programs we used for our evaluation are as follows:

1. Our own implementation in C++ of the A* search algorithm using the X-Y heuristic function, as described in Chapter 3. This program finds optimal solutions.
2. Our own implementation in C++ of Dijkstra's shortest path algorithm, as described in Chapter 3. This program finds optimal solutions.
3. Our implementation in Copris of the SAT-based algorithm described in Chapter 6, adapted from Tamura's implementation for solving Sokoban [17], which is described in Chapter 5, using the parallel SAT solver Plingeling [12]. This program has a flag *opt*, which, when set to *True*, instructs the program to find an optimal solution. When set to *False*, the program finishes when it finds any solution which satisfies the current move limit, which in many cases is more than necessary.
4. Program 3 using the sequential SAT solver MiniSat [21] instead of Plingeling.

The machines we used for our evaluation are as follows:

1. A machine with 128 CPU threads, of which 64 have been used simultaneously while running Plingeling.
2. A machine with 8 CPU threads, of which 4 have been used simultaneously while running Plingeling.
3. A machine with 4 CPU threads, of which 2 have been used simultaneously while running Plingeling.

7.2 The 8-puzzle

To evaluate the performance of the programs on the 8-puzzle, we ran the aforementioned programs on the same set of 100 random solvable instances of the 8-puzzle. Programs 3 and 4 were run for both values of the *opt* flag. For all the programs, we noted the total execution time, the average execution time and the standard deviation in the execution time. These can be found in Table 7.1.

		Execution time		
	opt flag	Total	Average	Standard deviation
Program 1 (machine 3)		0.3767	0.0037	0.0013
Program 2 (machine 3)		27.4419	0.2744	0.2076
Program 3 (machine 1)	*False*	766.5890	7.6658	1.6215
Program 3 (machine 1)	*True*	2091.0082	20.9100	6.5046
Program 3 (machine 3)	*False*	339.0015	3.3900	0.8953
Program 3 (machine 3)	*True*	771.8428	7.7184	3.0342
Program 4 (machine 3)	*False*	217.5099	2.1751	0.5060
Program 4 (machine 3)	*True*	496.0598	4.9605	2.0594

Table 7.1: Total execution time, average execution time and standard deviation for 100 random instances of the 8-puzzle. All values are in seconds.

We make two major observations from Table 7.1. The first is the comparison of A* search and Dijkstra's algorithm with the SAT-based programs. Program 1, our implementation of A* search with the X-Y heuristic, significantly outperforms all of the other programs. Where program 2, our implementation of Dijkstra's shortest path algorithm, needs about a quarter of a second on average, and our SAT-based programs need at least several seconds, our implementation of A* search only needs a few milliseconds to find an optimal solution on average. Although it does not rival our implementation of A* search, program 2, our implementation of Dijkstra's shortest path algorithm, also outperforms our SAT-based programs by at least an order of magnitude on average execution time. However, it should be noted that our implementations of A* search and Dijkstra's algorithm run natively on our machines, while our SAT-based programs are interpreted. The SAT formulas generated by the program are fed to the used SAT solvers, which in our case run natively. The SAT solver then determines the satisfiability of the formulas and returns its findings. Depending on how much time is spent on generating the formulas and how much time is used on actually determining their satisfiability, some gains in execution time might be made by implementing the translating step in a natively run program.

Our second observation is that program 3, our implementation of our SAT-based algorithm using the SAT solver Plingeling, runs faster on machine 3, which uses only two threads for running Plingeling, than on machine 1, which uses 64 threads to perform the same task. Furthermore, program 4, which uses the sequential SAT solver MiniSat, runs faster on machine 3 than the program using the parallel SAT solver Plingeling did on either machine. This leads us to believe that, at least for the relatively small 8-puzzle, the speedup caused by using parallelism does not outweigh the overhead it creates due to the need of sharing information among many threads.

7.3 The 15-puzzle

After running all of the programs on the 8-puzzle, we also ran a selection of them on a few instances of the larger 15-puzzle. For this, we chose two relatively easy instances, having minimal solutions of 47 and 50 moves.

For the first puzzle, which has a minimal solution of 50 moves, program 1, our implementation of A* search, running on machine 3, needed less than 100 milliseconds to find an optimal solution. Program 4, our implementation of our SAT-based algorithm using MiniSat, running on machine 2 with the *opt* flag set to *False*, needed close to 37 hours to find a solution at all. We ran program 3, our implementation of our SAT-based algorithm using Plingeling, three times on machine 1, using 64 threads, on this same puzzle with the *opt* flag also set to *False*. It found solutions in respectively close to 30 minutes, close to 75 minutes and close to three hours. We then ran the algorithm using Plingeling again on machine 1 on the second puzzle, which has a minimal solution of 47 moves, with the *opt* flag set to *True*. It then found an optimal solution in a little under six hours. Our implementation of A* search again needed under 100 milliseconds to find an optimal solution. Note that we did not run program 2, our implementation of Dijkstra's shortest path algorithm, on any instance of the 15-puzzle, as it would be far too costly memorywise.

These few runs show us a few important things. First and foremost, our implementation of our SAT-based algorithm is capable of finding optimal solutions for the 15-puzzle, which was earlier said to have approximately 10^{13} reachable states.

When comparing the execution time of our SAT-based approach to that of our implementation of A* search, however, these runs show the difference in execution time to be even greater than they were for the 8-puzzle. Our implementation of A* search found an optimal solution approximately 250 000 times faster than our implementation of our SAT-based algorithm, which was running on 64 threads. Therefore, we consider it plausible that the already considerable differences in execution time shown in Table 7.1 for the 8-puzzle will only become greater for larger instances of the n-puzzle — especially when one does not have 64 CPU threads at their disposal.

Another observation that can be made based on these results is that at first sight, for larger instances of the n-puzzle, parallelism seems to offer significant improvements. Where, on the 8-puzzle, the program using the parallel SAT solver Plingeling performed worse than one using the sequential SAT solver MiniSat, the parallel program performs significantly better on the 15-puzzle than the sequential one, given enough CPU threads to fully utilise the benefits of parallelism.

When run on a more difficult instance of the 15-puzzle, having a minimal solution of 61 moves, our implementation of A* search needed close to a minute to find an optimal solution on machine 3. Assuming the relative difference in execution time will not diminish when compared to the earlier puzzle with a minimal solution of 47 moves, this would mean our SAT-based approach would need at least six months to find an optimal solution, and quite possibly a lot more. We have not attempted to verify this.

Chapter 8

Conclusions and further research

Summarising the previous chapters, we conclude that an approach based on SAT solvers can be used to find optimal solutions for the n-puzzle, as shown in Chapter 7. However, we also conclude that the approach we propose in Chapter 6, based on Naoyuki Tamura's approach to a similar problem, described in Chapter 5, can not as of yet compete with established algorithms such as A* search, which was presented in Chapter 3, and needs a lot of improvement if it is to ever do so. The results presented in Chapter 7 clearly show this. Furthermore, we believe that such improvement is possible, although we dare not speculate if enough improvements are possible to bring this approach on par with, for instance, A* search.

We believe further improvement is possible on the additional constraints addressed in Section 6.3. Adding more constraints on top of the ones we have proposed would likely further restrict the possibilities and further speed up the process. One such constraint might be to require, at each step in the translation, the remaining number of moves to be greater or equal to the Manhattan distance of the state at the current step. The Manhattan distance is described in Section 3.3.

Another possibility is to analyse SAT solvers and attempt to optimise them for solving this specific type of problems. This is an area we did not explore in this paper.

Bibliography

[1] Jerry Slocum and Dic Sonneveld. *The 15 Puzzle Book: How it Drove the World Crazy*. Slocum Puzzle Foundation, 2006.

[2] Simon Tatham's Portable Puzzle Collection. http://www.chiark.greenend.org.uk/~sgtatham/puzzles/. Accessed: 2016-06-24.

[3] Wm. Woolsey Johnson and William E. Story. Notes on the "15" puzzle. *American Journal of Mathematics*, 2:397–404, 1879.

[4] Daniel Ratner and Manfred Warmuth. Finding a shortest solution for the $n \times n$ extension of the 15-puzzle is intractable. In *Proceedings AAAI-86*, pages 168–172, 1986.

[5] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education Limited, third edition, 2010.

[6] Richard M. Wilson. Graph puzzles, homotopy, and the alternating group. *Journal of Combinatorial Theory*, 16:86–96, 1974.

[7] Puz-Graph — from Wolfram Mathworld. http://mathworld.wolfram.com/Puz-Graph.html. Accessed: 2016-06-19.

[8] theta_0 Graph — from Wolfram Mathworld. http://mathworld.wolfram.com/Theta-0Graph.html. Accessed: 2016-07-14.

[9] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[10] Joseph C. Culberson and Jonathan Schaeffer. Searching with pattern databases. In *Conference of the Canadian Society for Computational Studies of Intelligence*, pages 402–416. Springer, 1996.

[11] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[12] Armin Biere. Lingeling, Plingeling and Treengeling. http://fmv.jku.at/lingeling/. Accessed: 2016-06-19.

[13] SAT-Race 2015. http://baldur.iti.kit.edu/sat-race-2015/. Accessed: 2016-06-19.

[14] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley Professional, 2015.

[15] Armin Biere, Marijn Heule, Hans van Maaren and Toby Walsh. *Handbook of Satisfiability*. IOS Press, 2009.

[16] Homepage Naoyuki Tamura. http://bach.istc.kobe-u.ac.jp/tamura.html. Accessed: 2016-04-14.

[17] Sokoban Solver in Copris, Naoyuki Tamura. http://bach.istc.kobe-u.ac.jp/copris/puzzles/sokoban/. Accessed: 2016-04-14.

[18] Copris: Constraint Programming in Scala, Naoyuki Tamura. http://bach.istc.kobe-u.ac.jp/copris/. Accessed: 2016-04-14.

[19] Sokoban Wiki. http://www.sokobano.de/wiki/index.php?title=Main_Page. Accessed: 2016-04-14.

[20] JSoko on SourceForge. https://sourceforge.net/projects/jsokoapplet/. Accessed: 2016-02-19.

[21] MiniSat. http://minisat.se/. Accessed: 2016-07-11.