Bachelor thesis
Computer Science

Radboud University

An exponential construction for Sokoban

Author:
Jan Potma
s4476360

First supervisor/assessor:
Prof. dr. Hans Zantema
`h.zantema@TUE.nl`

Second assessor:
Dr. Jurriaan Rot
`jrot@cs.ru.nl`

March 27, 2018

Abstract

We show a construction for the puzzle game Sokoban, where any solution to the puzzle is at least exponential in length relative to the size of the puzzle. This is a variant of a construction by John Hoffman, which lacked formal proof. To prove its correctness, we use a combination of macro moves, induction, and model checking using NuSMV. We establish the behaviour of a number of devices in Sokoban, going into greater detail than earlier work. We highlight the properties of the pass-reset device, and use model checking to verify our claims about its functioning. We abstract player movement as a set of elementary transitions (macro moves) through these devices, and apply induction on this system to prove exponentiality.

Contents

1 Introduction **2**
1.1 The game Sokoban 2
1.2 What makes for an interesting puzzle? 3
1.3 Our work 3

2 The construction **4**
2.1 The room 4
2.2 Chaining multiple rooms 7
2.3 Comparison to Hoffman's method 8

3 Proof of exponentiality **9**
3.1 Building blocks 9
3.1.1 The pass-reset device 13
3.2 Abstracting rooms 14
3.2.1 List of transitions 15
3.2.2 Induction 16
3.2.3 Completing the construction 18
3.2.4 Example case for N = 3 19

4 Model checking **20**
4.1 An introduction to NuSMV 20
4.2 Modelling Sokoban 22
4.3 Proving the working hypothesis 25
4.3.1 Transitions through the pass-reset 26
4.3.2 Transitions through the room 27

5 Related Work **28**

6 Conclusions **30**

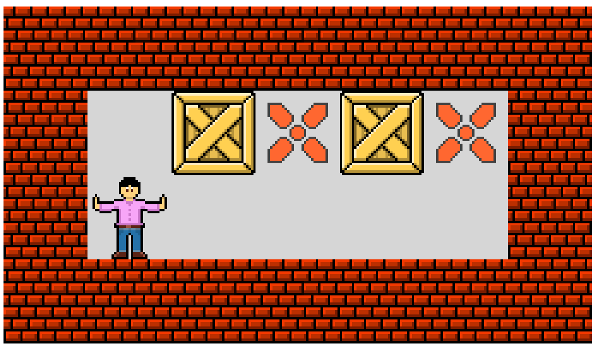
Chapter 1

Introduction

1.1 The game Sokoban

Sokoban is a puzzle game where the player must navigate through a warehouse to push boxes into the correct position. The player moves around using the arrow keys, and can push a box by moving into it, but only if the space behind the box is free. Every puzzle has a starting configuration of all the boxes and the player, and has a goal configuration for the boxes.

Below is a simple puzzle. The two orange crosses indicate the goal positions. Call the box on the left 'box 1', and call the box on the right 'box 2'. (Keep in mind that this numbering is purely for ease of reference: boxes are interchangeable in Sokoban, and can go on any goal space.) If we start by pushing box 1 to the right, it will be impossible to push box 2 into the correct position, since box 1 is in the way. To solve this puzzle, we must first push box 2 to the right, walk back, and then push box 1 to the right.



1.2 What makes for an interesting puzzle?

Sokoban has a large online community with thousands of puzzles. Like our example, many of these puzzles have a solution that is much more difficult than it seems at first glance. These puzzles typically take up a small space, but need many steps to solve. In this paper, we will show a method to make this kind of puzzle. Specifically, we will be aiming for an exponential solution, answering the question:

How do we create puzzles that require an exponential amount of steps to solve, in regard to the number of spaces?

Sokoban has already been proven to be PSPACE-complete [2, 5], so it must be possible. If we can find a method, this could be useful as a basis for future work in complexity theory, by reducing to this method or using our findings as inspiration for other exponential constructions.

1.3 Our work

A construction that is claimed to be exponential already exists, and is credited to John Hoffman [4]. However, proof for this construction is lacking. We use this as the starting point for our work. In this paper, we will:

- Explain our modified version of this construction [Chapter 2]
- Prove this construction is exponential, by using induction and a working hypothesis [Chapter 3]
- Prove the working hypothesis using model checking [Chapter 4]

Chapter 2

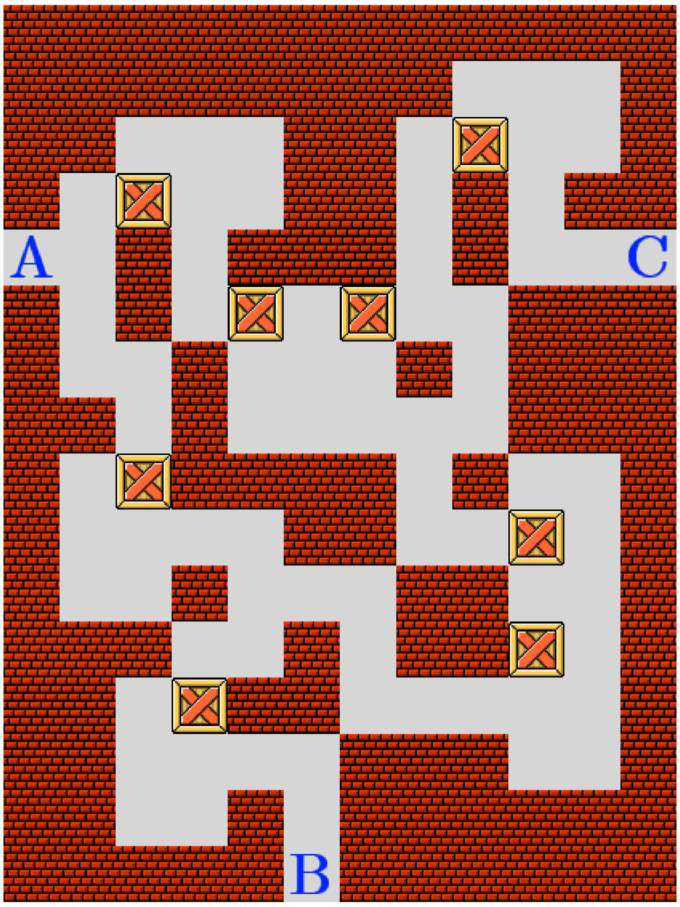
The construction

As an answer our research question, we present a simple method to construct exponential puzzles. This construction is based on Hoffman's original method [4]. We compare our version to Hoffman's construction in section 2.3.

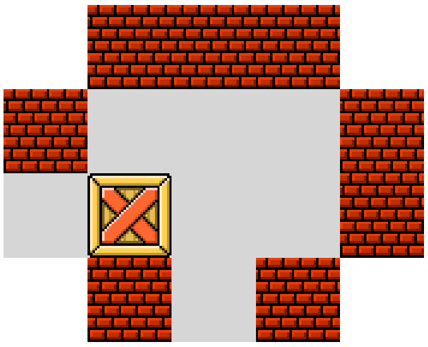
2.1 The room

At its core, the construction uses a single piece that is repeated many times. Hoffman calls this specific piece a *room*. Each room is a partial Sokoban puzzle, with open ends to connect it to the rest of the puzzle. The room we use, in its initial configuration, is depicted on the next page.

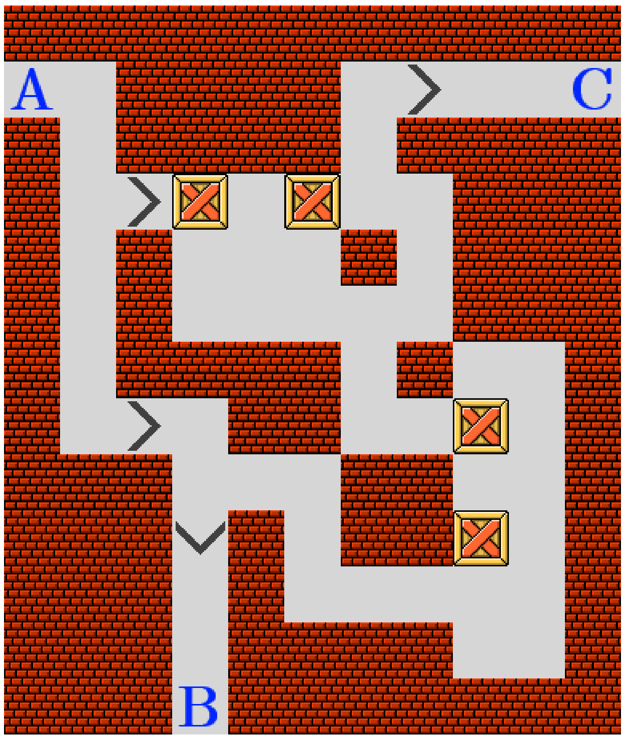
This room might look complicated at first, but it can be broken down into simple parts. The room has three open ends (*exits*), which we designate A, B, C. We indicate these in blue. (These indicators not part of Sokoban itself.) Initially, all boxes in the room are on a corresponding goal position. In principle, these boxes would not need to be moved to solve the puzzle. However, we need to travel across this room to reach other parts of the puzzle. To do so, we will have to move some boxes away from their goal positions, which we must rectify later. We enter the room at exit A. The goal is to get to exit C, but we can also go to exit B instead.



One pattern that repeats itself in the room is the *one-way device*, pictured below. Introduced by Sabey [2], this device only allows the player to go through it in one direction, never the opposite way. In this image, the player can go from the left side to the bottom side, by pushing the box to the right and then left again. The player cannot enter at the bottom and exit on the left, since the player will not be able to push the box back onto the goal position.

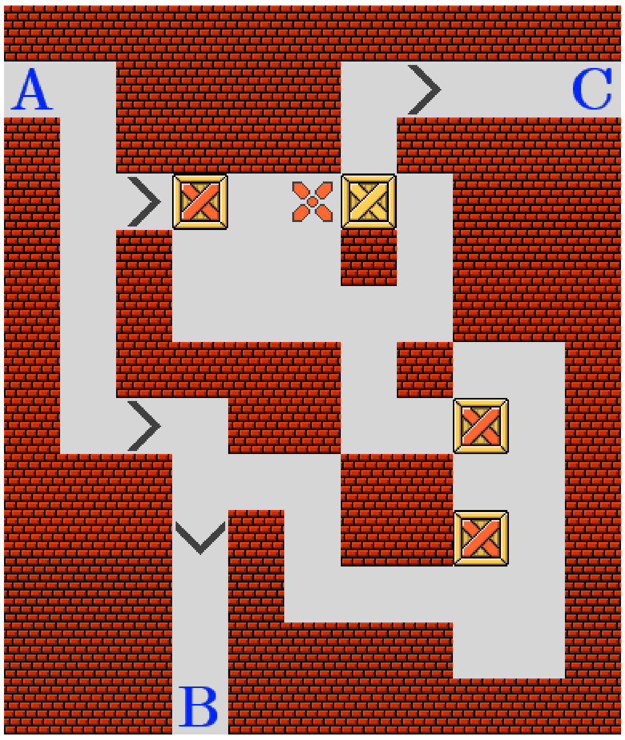


We can use this insight to greatly simplify the room. The image below represents the same room as before, but with the one-way devices replaced with arrows. The player can not move against the direction of the arrow, and boxes cannot pass through either side of the arrow.



Now there are only two sections with boxes remaining, a top part and a right part. The right part is called a *reverser* [2]. A reverser works much like a one-way device: the player can enter on one side and exit at the other, but cannot go in the opposite direction. The twist is that the direction switches each time the player passes through the reverser. Currently, the player can only enter at the bottom and exit at the top. After passing through once, the player can only enter at the top and exit at the bottom. The reverser proves useful when we want to allow the player to go through it to reach another area, but force them to eventually retrace their steps.

With this knowledge, it becomes easier to understand the core functionality of the room. To reiterate, our goal as the player is to go from exit A to exit C. If we go right at the first junction, it will push the boxes there together, rendering the puzzle unsolvable. We must go down to the second junction. Here we can go right and up, through the reverser. But if we leave the room at exit C now, the reverser is in the wrong state. We must eventually go back through the reverser, and exit at B. Before we do so, we can change the configuration of the room by moving the top-right box. After we leave the room at B, it now looks like the image on the next page.



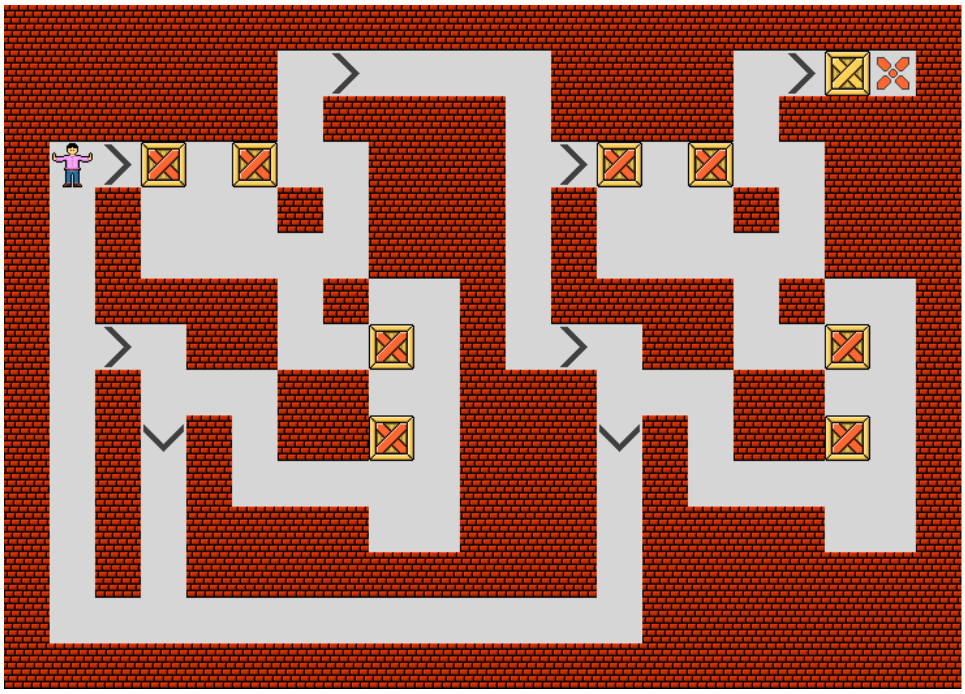
The next time we enter at A we can move right at the first junction, as room behind the box has opened up. To be able to exit at C, we must push the boxes back into their original position before we leave the room. We have now reached our goal of exiting at C, and the room has returned to its initial configuration. Thus, from the initial configuration, we need to traverse the room twice to end up at C, which resets the room.

2.2 Chaining multiple rooms

By placing multiple copies of this room in sequence, we can repeatedly double the required movement to solve the puzzle. We use this in our construction, which we detail here: Repeat the room N times, connecting exits C and A of adjacent rooms. The player begins at exit A of the left-most room, which we will call the start. The start connects to exit B of every room through a corridor. At the remaining open end, exit C of the right-most room, we place a box and a goal space. In the initial configuration of the puzzle, this is the only box not on a goal space. Since puzzle is solved by placing all boxes on a goal space, the goal is to reach exit C of the right-most room.

Exit C of a given room can only be reached if the room is prepared by moving the top-right box beforehand. To do so, the player must have previously travelled from exit A to B, and have come back to exit A.

This means the player has had to visit exit A twice. But this process repeats in the next room: exit at B, come back to A, and exit at C. This doubles the actions required to reach the room, which is doubled in the next, and so on. As a result, we must come back to the start, exit A of the first room, at least 2^N times. Thus, our construction is exponential. The image below shows an example for $N = 2$, which has been slightly horizontally compressed to fit on the page.



2.3 Comparison to Hoffman's method

Hoffman used a similar room in his construction, though with a few differences. The basic concept was the same: require the player to go through the room twice to reach the end. Our version has added several winding tunnels and two one-way devices: one to the right of the first junction, one at exit C. These have greatly aided in proving the properties of the room. One of the winding tunnels solves an issue with Hoffman's room, which lacks a winding tunnel in front of the one-way device between the first and second junction. This unintentionally allowed the player to solve it in 2^{n-1} steps instead of 2^n, by pushing the box past the first junction. This would have complicated the proof by separating it into two distinct processes. Finally, it should be mentioned we have mirrored the room to make the direction of travel left to right, though this is of course a trivial change.

Chapter 3

Proof of exponentiality

The previous chapter described the basic mechanism of the room. However, we mostly merely outlined the steps that the intended solution takes, as well as making some broad claims about the properties of parts of the room. The point of this paper is to scrutinise and solidify these claims through more thorough analysis. In this chapter, we formalise our construction. We will prove the behaviour of the individual parts of the room in section 3.1, and use induction to prove exponentiality in section 3.2, where we show that no shorter solution to the puzzle exists. In the process, one major claim about the room will be assumed here, and proved later in section 4.3 using model checking.

3.1 Building blocks

A Sokoban puzzle of the size of our construction quickly becomes too complex to reason about in a mathematical fashion. We will instead break down the construction into smaller parts. We can claim and then prove statements about these parts, allowing us to abstract them into elementary components. We can then recombine these components that have known traits into larger systems, and apply formal reasoning and logic on these traits to prove the properties of the whole system. As this abstract representation is equivalent to the original puzzle, it proves our construction correct.

We will be making several assumptions and claims in order to convert individual segments of the Sokoban puzzle into mathematical concepts. We will specify these clearly, to mark the basis on which we operate. Some of these are trivial, but nonetheless we strive to justify all assumptions.

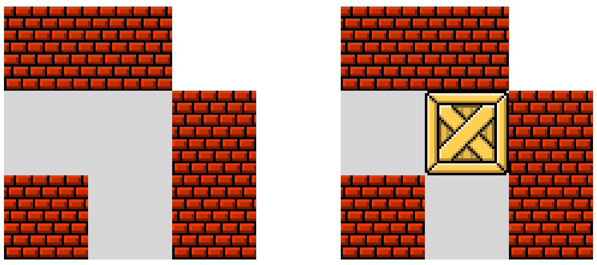
Before we can begin, we need to establish some terminology. We call a configuration of boxes *forbidden* if it is impossible to for the player to solve the puzzle from this configuration, because not all boxes can be pushed onto a goal position. A configuration that is not forbidden is called *legal.* Similarly, moves are called forbidden if they lead to a forbidden configuration.

It is assumed the player only performs legal moves.

Sets of configurations can be abstracted as *states*. A state consists of configurations that are equivalent, i.e. have identical properties. States exist on different levels of abstraction: a part of the room can have multiple states, but a room containing these parts also has states, which are all possible combinations of the states of its parts. A state is called the initial state if it contains the initial configuration of the puzzle.

We can divide the puzzle into isolated components, as it is comprised of disjunct areas connected by distinct paths. These areas are walled-off clearings at least 2 spaces wide and long. Each area fulfils a specific role, and is called a *device* [2]. We have already used the one-way device and the reverser in chapter 2. The paths between devices are called *tunnels* [6]. There are also *junctions*, single spaces that connect to three tunnels.

A tunnel is a collection of pairwise adjacent spaces, with each space orthogonally adjacent to exactly two walls and two spaces. There are two types of tunnels: *straight* tunnels and *winding* tunnels. In a straight tunnel all spaces are in a single line, whereas a winding tunnels contains one or more *bends*. A bend is a tunnel space where the two adjacent spaces are not in line, like the left side of the image below.

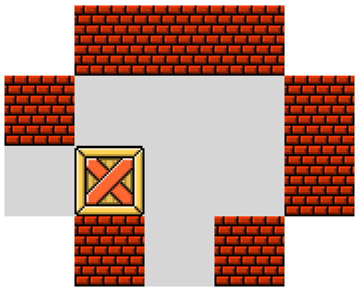


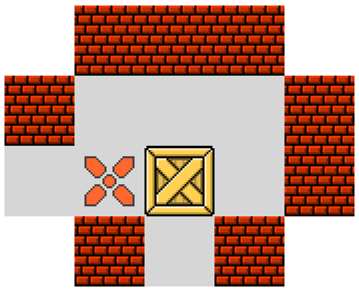
Claim: A box cannot be pushed from one side of a bend to the other, and by extension, a box cannot be pushed through a winding tunnel.

Justification: To push a box through a bend, it would need to be pushed into the bend space, as on the right side of the image above. The box is stuck in this position, it is impossible to move the box to any other position. Therefore, the other side of the bend cannot be reached. To push a box through a winding tunnel, it would need to be pushed through a bend, which is impossible.

This property of winding tunnels is very useful. If a device only connects to winding tunnels, the boxes in the device cannot be pushed out of the device, nor can other boxes be pushed into the device. We will exclusively use winding tunnels in our construction.

We have already introduced the *one-way device*, which connects to two winding tunnels. We know that the box cannot leave the device, and no other box can enter the device, as the tunnels are winding. Within the device, the box only has two legal positions, as represented in the following image. The box may not be pushed into the top row or the right column, as this makes it impossible for the box to return to the goal position. The box may also be in the beginning of either tunnel, as long as it is not pushed into the first bend.





Claim: The one-way device only has two states while the player is not in the device: Either the box is in the bottom tunnel (state 1), or it is in any other legal space (state 0). It can only be traversed left-to-bottom.

Justification: We must show that the box positions within each state have identical properties, but the two states do not. There are two properties a device with two tunnels can have: which direction of travel between tunnels is permitted, and what transitions between states are possible.

State 1: Neither direction of travel is possible:

- Entering from the left tunnel, the box cannot be passed, since it can only be pushed further into the tunnel, until the first bend.
- Entering from the bottom tunnel, the box can only be passed by pushing it into the top row, which is forbidden.

The only way to change state is entering from the bottom tunnel, pushing the box into the right legal position in the device, and returning.

State 0: The player can travel from the left tunnel to the bottom tunnel, but not the other way.

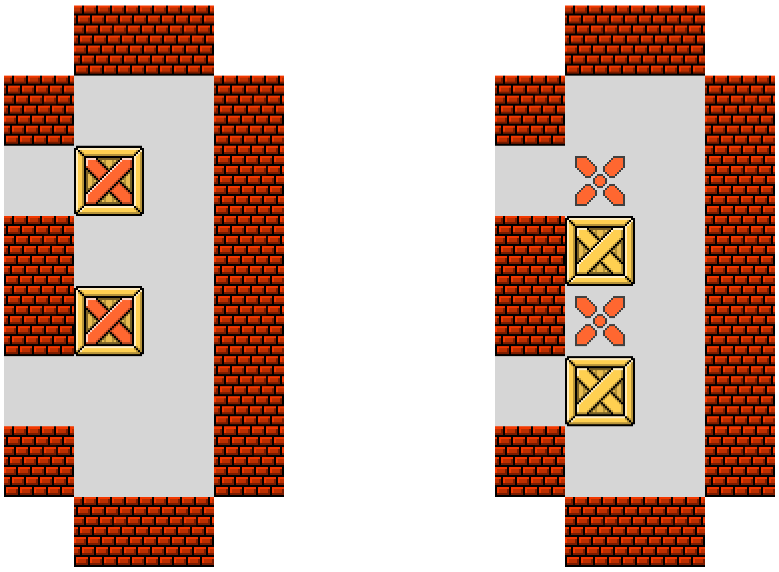
- Entering from the left tunnel, the player must push the box all the way to the right to pass it. Since pushing the box into the bottom tunnel prevents further travel, the box must be pushed left, onto the goal position or into the left tunnel. The player can then go through the bottom tunnel.
- Entering from the bottom tunnel, if the box is all the way to the right, it can only be pushed into the top row, which is forbidden. Otherwise, it can only be pushed into the left tunnel, but it cannot be passed, since it can only be pushed until the first bend.

The only way to change state is entering from the left tunnel, pushing the box into the bottom tunnel, and returning.

Claim: We can require that the player always leaves the box in the goal position upon exiting the one-way device, without affecting the proof. Consequently, the one-way device is reduced to a single state.

Justification: In doing so, we would exclude state 1 from our system. The device starts in state 0, and is always left in state 0. However, since state 1 only reduces the possibilities for travel, there are no solutions that require the device to be in state 1, and the proof is unaffected. Travel through the device allows us to choose where to leave the box, and we can always choose to leave it in the goal position.

As a result of this, we no longer have to worry about the position of the boxes in the one-way device: if all the boxes outside of the one-way devices are on a goal position, the puzzle is solved.



Another device we have already used is the reverser, which connects to two winding tunnels. The right column, as well as the top and bottom row, may not contain a box. This leaves four spaces, as well as both tunnels, for the boxes to be in. The two boxes may not be in the same tunnel: this renders the puzzle unsolvable as they cannot be pushed out.

Claim: The reverser has only two states while the player is not in the device. It can only be traversed in one direction at a time, which reverses every time it is travelled across.

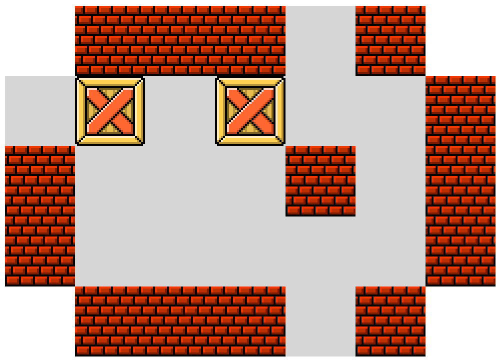
Justification: There are five possible states for the device. State 0 and 1 consist of a single configuration, and are shown in the image. State 2 has a single box in the top tunnel, with the other box anywhere in the device. State 3 has a single box in the bottom tunnel, with the other box anywhere in the device. State 4 has a box in the top-most and a box in the bottom-most legal row/tunnel. However, this state cannot possibly occur: it is not the initial state, and the player would not be able to leave the device after setting it up. We can eliminate states 2 and 3 like we did with state 1 in the one-way device. This results in two states:

State 0:
- Entering at the top, we can only push the box to the right column, which is forbidden.
- Entering at the bottom, we can leave at the top, but must change to state 1 (or state 3).

State 1:
- Entering at the top, we can leave at the bottom, but must change to state 0 (or state 2).
- Entering at the bottom, we can only push the box to the right column, which is forbidden.

3.1.1 The pass-reset device

The core of our room is the *pass-reset device* [2]. This is a device composed of two subdevices: a reverser, and a device we will refer to as the *partial pass-reset device*. We have already established the properties of the reverser. The partial pass-reset device is below. The boxes in the partial pass-reset can only be in the top row/tunnel, minus the right-most space. Only the left box can legally be in the tunnel.



Claim: The partial pass-reset has three states while the player is not in the device.

Justification: There are five possible states for the device:

State 0: Shown in the image.

State 1: Left box on the left goal position, right box on the right-most legal space.

State 2: Left box between the goal positions, right box on the right-most legal space.

State 3: Left box in the tunnel, right box on the right-most legal space.

State 4: Left box in the tunnel, right box on the right goal position or between the goal positions.

Again we eliminate states 3 and 4. All three remaining states have only one configuration. These all have different traversal options: state 0 only prevents travel to and from the left, state 1 prevents travel from the top and travel to the left, state 2 prevents only travel from the top.

Introducing a working hypothesis

These three states, when combined with the two states of the reverser, result in 6 possible states for the pass-reset as a whole. Since all the one-way devices in the room are effectively stateless, this results in 6 corresponding possible states for the entire room.

We will not be examining the states of the room here. Instead, we will refer the reader to section 4.3. There, we use model checking to show that this can be reduced to merely two states, state 0 and state 1. Additionally, we show that this results in the possible traversal options listed in section 3.2.1. For now, we will view this as a working hypothesis which we assume holds true, in order to continue our proof.

3.2 Abstracting rooms

Using the room we now have characterised, we can create the core of our construction. We make a partial Sokoban puzzle, based on a parameter N. This puzzle consists of N rooms that are pairwise adjacent. Adjacent rooms are connected by a winding tunnel that links exit C of the left room to exit A of the right room. The puzzle also has a corridor outside of the rooms, that connects to exit B_i of every room i and connects to exit A_1 of the first room.

We abstract from the concrete Sokoban puzzle, by treating the rooms as black boxes. Every room has a state, 0 or 1, as per our working hypothesis. We combine the individual moves the player makes into *macro moves* [6], a series of moves that are executed as a whole. At the start and end of each macro move, the player must be at the exit of a room, and every room in either state 0 or state 1. We call these macro moves *transitions.*

This new, abstract view of the puzzle also has a state. This state has two parts: the position of the player, and the state of each room. Our notation of a puzzle state is an ordered pair, $(p, \vec{s})$. The first element is the position of the player, p. The second is the combination of all room states, $\vec{s}$.

The first element, p, can be any exit A_i, B_i, C_i to any room $i \leq N$. Thus we have $p \in P = \bigcup_{i=1}^{N} \{A_i, B_i, C_i\}$.

The second element, $\vec{s}$, is a vector of dimension N. We have $\vec{s} \in \{0,1\}^N$. This vector can also be written as $\vec{s} = (s_1, s_2, \ldots, s_N)$, with $s_i \in \{0,1\}$. Every dimension of $\vec{s}$ represents the state of a single room.

Now we can specify transitions, which are how the puzzle state changes. We notate a transition from one puzzle state to another as follows:

$(p, \vec{s}) \mapsto (p', \vec{s}\,')$

Here, $(p, \vec{s})$ is the *antecedent state* of our transition, and $(p', \vec{s}\,')$ is the *subsequent state.*

We can express the new vector $\vec{s}\,'$ in terms of the old vector $\vec{s}$ using assignment. For instance, say we have only changed the state of the first room in this transition, setting it to 0. We would then have $\vec{s}\,' = \vec{s}[s_1 := 0]$.

3.2.1 List of transitions

In this new system, there are only a few elementary transitions we can perform. A complete list is below.

$$
\begin{aligned}
(B_i, \vec{s}) &\mapsto (A_1, \vec{s}), && 1 \leq i \leq N && (3.1)\\
(A_1, \vec{s}) &\mapsto (B_j, \vec{s}), && 1 \leq j \leq N && (3.2)\\
(B_i, \vec{s}) &\mapsto (B_j, \vec{s}), && 1 \leq i \leq N, 1 \leq j \leq N, i \neq j && (3.3)
\end{aligned}
$$

$$
(C_i, \vec{s}) \mapsto (A_{i+1}, \vec{s}), \qquad 1 \leq i < N \qquad (3.4)
$$

$$
\begin{aligned}
(A_i, \vec{s}) &\mapsto (C_i, \vec{s}[s_i := 0]), && 1 \leq i \leq N, s_i = 1 && (3.5)\\
(A_i, \vec{s}) &\mapsto (B_i, \vec{s}[s_i := 1]), && 1 \leq i \leq N, s_i \in \{0,1\} && (3.6)\\
(A_i, \vec{s}) &\mapsto (B_i, \vec{s}[s_i := 0]), && 1 \leq i \leq N, s_i \in \{0,1\} && (3.7)
\end{aligned}
$$

Transitions (3.1), (3.2), and (3.3) are though the previously mentioned external corridor, and transition (3.4) goes through the winding tunnel connecting the rooms. The rest are within a single room. These are part of the working hypothesis, and we will show that these are correct and exhaustive in section 4.3.

We can combine these elementary transition into compound transitions. A compound transition can consist of any amount of elementary transitions. Compounds are indicated with an asterisk, and only note the antecedent state of the first transition, and the subsequent state of the last transition: $(p, \vec{s}) \mapsto^{*} (p', \vec{s}\,')$. Thus, a compound transition can be separated into different sequences of elementary transitions, as long as they have the same result.

3.2.2 Induction

With these transitions, we can finally show that our current system is exponential. We wish to prove that for any N, the number of steps taken to solve the partial puzzle is at least 2^N. We consider the puzzle solved when all boxes are on the goal positions *and* the player is at exit C_N. Thus, the goal state is $(C_N, \vec{0})$.

Since we have abstracted player movement, we can no longer count the number of steps taken by the player. We have instead chosen to count the transitions that have $p = A_1$ in their antecedent state, because each of these transitions requires that the player takes at least one step. It is obvious that many more steps are taken, but if we can prove that the number transitions used is exponential, then this is proof enough.

We introduce a variable n, which is bound from above by N. We apply induction on n to prove that these properties hold for all n, and therefore hold for N.

Induction hypothesis

Consider the transition $(A_1, \vec{s}) \mapsto^* (C_n, \vec{s}\,')$, where $\vec{s}$ has $s_i = 0 \;\; \forall i \leq n$. This transition has some corresponding sequence $((p_0, \vec{s}_0), (p_1, \vec{s}_1), \ldots, (p_k, \vec{s}_k))$ for some k, where $(p_0, \vec{s}_0) = (A_1, \vec{s})$ and $(p_k, \vec{s}_k) = (C_n, \vec{s}\,')$. For such a sequence, the transitions $(p_i, \vec{s}_i) \mapsto (p_{i+1}, \vec{s}_{i+1}) \;\; \forall i < k$ must be valid elementary transitions, as listed in section 3.2.1.

We use as our induction hypothesis:

1. Such a sequence exists.
2. For any sequence, $\#\{i \mid p_i = A_1\} \geq 2^n$
3. For any sequence, $\vec{s}_k$ has $s_i = 0 \;\; \forall i \leq n$

Induction basis

We prove for $n = 1$.

1. We show by example, including the elementary transitions used:

$$(A_1, \vec{s}) \;\overset{3.6}{\longmapsto}\; (B_1, \vec{s}[s_1 := 1]) \;\overset{3.1}{\longmapsto}\; (A_1, \vec{s}[s_1 := 1]) \;\overset{3.5}{\longmapsto}\; (C_1, \vec{s})$$

2. We must prove that any sequence has $\#\{i \mid p_i = A_1\} \geq 2^1 = 2$. The last elementary transition used can only be (3.5), as it is the only elementary transition with $p = C_1$ in the subsequent state. This transition requires that $s_1 = 1$. But because $s_i = 0 \;\; \forall i \leq n$ in the initial state, we must first set $s_1 := 1$, which we can only do with (3.6). Both of these transitions have $p = A_1$ in the antecedent state, and both must be used by any sequence. Thus we have $\#\{i \mid p_i = A_1\} \geq 2$

3. The last transition used by any sequence must be (3.5), which assigns $s_1 := 0$. As a result, $\vec{s}_k$ has $s_i = 0 \ \forall i \leq 1$.

Inductive step

We know that the induction hypothesis holds for $n < N$. We prove for $n+1$

1. We again show by example. Here, IH represents the transition from the induction hypothesis, which by property (1) must exist. By property (3), we know $\vec{s}\,'$ has $s_i = 0 \ \forall i \leq n$, so we may use IH here.

$$\begin{array}{llll}
(A_1, \vec{s}) & \xmapsto{IH}{}^* & (C_n, \vec{s}\,') & \xmapsto{3.4} \\
(A_{n+1}, \vec{s}\,') & \xmapsto{3.6} & (B_{n+1}, \vec{s}\,') & \xmapsto{3.1} \\
(A_1, \vec{s}\,'[s_{n+1} := 1]) & \xmapsto{IH}{}^* & (C_n, \vec{s}\,''[s_{n+1} := 1]) & \xmapsto{3.4} \\
(A_{n+1}, \vec{s}\,''[s_{n+1} := 1]) & \xmapsto{3.5} & (C_{n+1}, \vec{s}\,'') &
\end{array}$$

2. By property (3), we know that any sequence for $(A_1, \vec{s}^{\,\dagger}) \mapsto^* (C_n, \vec{s}^{\,\ddagger})$ has $\#\{j \mid p_j = A_1\} \geq 2^n$, where j is bound by this sequence.

   We must prove that any sequence for $(A_1, \vec{s}) \mapsto^* (C_{n+1}, \vec{s}\,'')$ has $\#\{i \mid p_i = A_1\} \geq 2^{n+1}$. We have $i < k$, with k the length of this sequence.

   The last elementary transition used in any sequence must be (3.5), since this is the only elementary transition with $p = C_{n+1}$ in the subsequent state. The antecedent state must have $p = A_{n+1}$, and have $s_{n+1} = 1$.

   The only way to accomplish the latter is through (3.6), therefore it must also be present in any sequence. It also has $p = A_{n+1}$ in its antecedent state. Its subsequent state has $p = B_{n+1}$.

   Now we know any sequence must be of the following form, where α and β represent unknown transitions.

$$\begin{array}{llll}
(A_1, \vec{s}) & \xmapsto{\alpha}{}^* & (A_{n+1}, \vec{s}\,') & \xmapsto{3.6} \\
(B_{n+1}, \vec{s}\,'[s_{n+1} := 1]) & \xmapsto{\beta}{}^* & (A_{n+1}, \vec{s}\,'') & \xmapsto{3.5} \\
(C_{n+1}, \vec{s}\,'') & & &
\end{array}$$

   Both α and β have $p = A_{n+1}$ in their subsequent state. The only transition where this is the subsequent state (with $n \geq 1$) is (3.4), which in turn has $p = C_n$ in the antecedent state.

Our expanded sequence looks like this:

$$\begin{array}{llll}
(A_1, \vec{s}) & \overset{\alpha'}{\longmapsto}{}^{*} & (C_n, \vec{s}') \overset{3.4}{\longmapsto} (A_{n+1}, \vec{s}') & \overset{3.6}{\longmapsto} \\
(B_{n+1}, \vec{s}'[s_{n+1} := 1]) & \overset{\beta'}{\longmapsto}{}^{*} & (C_n, \vec{s}''[s_{n+1} := 1]) & \overset{3.4}{\longmapsto} \\
(A_{n+1}, \vec{s}''[s_{n+1} := 1]) & \overset{3.5}{\longmapsto} & (C_{n+1}, \vec{s}''[s_{n+1} := 0]) &
\end{array}$$

We can now see that α' is the transition of the induction hypothesis. As for β', we will show it also contains our induction hypothesis, but this requires a few more steps. The only transitions leading out of $p = B_i$ are (3.1) and (3.3). However, the latter puts us back at an exit B_j, from which one of the same two transitions must be taken. This cannot loop forever, as β' eventually leads to $p = C_n$. Thus, β' must start with (3.3) repeated x times, inevitably followed by (3.1). The remaining transition then matches our induction hypothesis, which we can use thanks to properties (1) and (3). This all results in the sequence:

$$\begin{array}{llll}
(A_1, \vec{s}) & \overset{IH}{\longmapsto}{}^{*} & (C_n, \vec{s}') \overset{3.4}{\longmapsto} (A_{n+1}, \vec{s}') & \overset{3.6}{\longmapsto} \\
(B_{n+1}, \vec{s}'[s_{n+1} := 1]) & \overset{3.3}{\longmapsto}{}^{*} & (B_i, \vec{s}'[s_{n+1} := 1]) & \overset{3.6}{\longmapsto} \\
(A_1, \vec{s}'[s_{n+1} := 1]) & \overset{IH}{\longmapsto}{}^{*} & (C_n, \vec{s}''[s_{n+1} := 1]) & \overset{3.4}{\longmapsto} \\
(A_{n+1}, \vec{s}''[s_{n+1} := 1]) & \overset{3.5}{\longmapsto} & (C_{n+1}, \vec{s}''[s_{n+1} := 0]) &
\end{array}$$

By property (2), the sequences that correspond with IH have $\#\{j \mid p_j = A_1\} \geq 2^n$. Since IH is included twice in any sequence, the total sequence has $\#\{i \mid p_i = A_1\} \geq 2 \cdot \#\{j \mid p_i = A_1\} \geq 2 * 2^n = 2^{n+1}$.

3. By property (3), we know that $\vec{s}'$ and $\vec{s}''$ have $s_i = 0 \ \forall i \leq n$

3.2.3 Completing the construction

We have shown that the induction hypothesis holds for any $n \leq N$, and therefore it holds for N. We can conclude that reaching state $(C_N, \vec{0})$ requires an exponential number of steps. We finish our construction by adding dead end, with a box and a goal position behind it, to the remaining open end of our system, C_N. In the initial configuration of the puzzle this is the only box not on a goal position.

This construction functions as a regular Sokoban puzzle; the player only needs to push the boxes onto the goal positions, a position for the player is not required. To reach the last box, the player must have taken at least 2^N steps, by property (2). After the last box has been pushed onto the goal position, the puzzle is solved, since all rooms are in state 0, by property (3). We know that this solution exists thanks to property (1).

3.2.4 Example case for N = 3

This example shows the optimal solution for a construction of size $N = 3$. It shows how quickly the loops add up to a long process, and shows the same transitions as we used in the inductive step. The transition to the next state is noted next to the current state. We work towards reaching the state $(A_1, (1, 1, 1))$: from this point we can go to $(C_3, (0, 0, 0))$, resetting the rooms we move through. In total, the player is at A_1 $8 = 2^3$ times, as expected.

$(A_1, (0, 0, 0))$	(apply 3.6)	$(A_1, (0, 0, 1))$	(apply 3.6)
$(B_1, (1, 0, 0))$	(apply 3.1)	$(B_1, (1, 0, 1))$	(apply 3.1)
$(A_1, (1, 0, 0))$	(apply 3.5)	$(A_1, (1, 0, 1))$	(apply 3.5)
$(C_1, (0, 0, 0))$	(apply 3.4)	$(C_1, (0, 0, 1))$	(apply 3.4)
$(A_2, (0, 0, 0))$	(apply 3.6)	$(A_2, (0, 0, 1))$	(apply 3.6)
$(B_2, (0, 1, 0))$	(apply 3.1)	$(B_2, (0, 1, 1))$	(apply 3.1)
$(A_1, (0, 1, 0))$	(apply 3.6)	$(A_1, (0, 1, 1))$	(apply 3.6)
$(B_1, (1, 1, 0))$	(apply 3.1)	$(B_1, (1, 1, 1))$	(apply 3.1)
$(A_1, (1, 1, 0))$	(apply 3.5)	$(A_1, (1, 1, 1))$	(apply 3.5)
$(C_1, (0, 1, 0))$	(apply 3.4)	$(C_1, (0, 1, 1))$	(apply 3.4)
$(A_2, (0, 1, 0))$	(apply 3.5)	$(A_2, (0, 1, 1))$	(apply 3.5)
$(C_2, (0, 0, 0))$	(apply 3.4)	$(C_2, (0, 0, 1))$	(apply 3.4)
$(A_3, (0, 0, 0))$	(apply 3.6)	$(A_3, (0, 0, 1))$	(apply 3.5)
$(B_3, (0, 0, 1))$	(apply 3.1)	$(C_3, (0, 0, 0))$	(finish)

Chapter 4

Model checking

4.1 An introduction to NuSMV

We will be using a model checker called NuSMV [3] to prove our working hypothesis. This program is given a set of all possible states a given model can be in, and how they are connected. This is called the state space. The program is then given a claim about this state space, in order to verify it. Claims usually assert that a certain state or set of states can or cannot be reached. By exploring the state space, NuSMV will either provide a counterexample, or conclude that no counterexample exists, proving the claim.

Our input for NuSMV typically consists of four sections: VAR, INIT, TRANS, and LTLSPEC.

- VAR lists the variables we use, including a limited range of possible values for each one. This immediately defines all possible states: every combination of values is considered a state.
- INIT limits the set of states that we can start in - we will limit ourselves to only one initial state.
- TRANS is the largest section, and limits how we can move between states. By default, any transition between states is allowed. By requiring certain conditions to be met, we disallow any transition other the elementary transitions for the model.
- LTLSPEC contains our claim. This will be the opposite of what we intend to prove: if we intend to show a state can be reached, we will claim it can never be reached. If the state can be reached, NuSMV will provide a trace showing the steps needed.

NuSMV uses Boolean functions for INIT and TRANS, and extends this to Linear Temporal Logic for LTLSPEC, which is a language to express formulae on infinite paths in state spaces.

An example will help clarify all this. Imagine a simple game using marbles. We start with one marble. Every round, we can either add one marble, or double our marble count. Using only these two operations, can we reach exactly 100 marbles?

```
MODULE main
VAR
    m : 1..100;
INIT
    m = 1
TRANS
     case m < 100  : next(m) = m + 1; TRUE : next(m) = m; esac |
     case m <= 50  : next(m) = 2 * m; TRUE : next(m) = m; esac
LTLSPEC
    G !(m = 100)
```

We can represent this game using the code above. Using this as input, NuSMV can then evaluate it for us. We use the four sections to precisely delimit our model.

- At VAR, we list our variables. We only have one: our marble count, `m`. Since variables must have a bound, we set a maximum of 100 marbles.
- At INIT, we tell NuSMV that we only ever start with `1` marble.
- At TRANS, we express what kind of transition is allowed. We only want NuSMV to either double our marbles or add one. This means our next marble count `m` must be one higher or twice our current count:

  ```
  next(m) = m + 1 | next(m) = 2 * m}
  ```

  However, since NuSMV does not allow `m` to have infinite range, we capped it at 100. If we were to use the line above, and added 1 while we were already at 100, `m` would overflow. So we only allow this if our count is less than 100, using a case distinction (`case...esac`): If our count is less than 100, continue as usual. Otherwise, keep `m` the same. NuSMV requires that each case distinction always has at least one condition that is met, so we set a default case using `TRUE`.

  Similarly, we can only double if the result would be less or equal to 100, so we can only double if we have 50 or fewer marbles.
- At LTLSPEC, we claim that we never have exactly 100 marbles. The G stands for global, stating that the condition after it always holds, and the exclamation mark mathematically negates the condition after it. Of course, if something *always* does *not* happen, it *never* happens. Thus we have `G !(`Φ`)`, with Φ the state we aim to prove is reachable. In this example, Φ is `m = 100`.

This is the NuSMV output we get using our input, which presents a counterexample:

```
-- specification G !(m = 100) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    m = 1
  -> State: 1.2 <-
    m = 2
  -> State: 1.3 <-
    m = 3
  -> State: 1.4 <-
    m = 6
  -> State: 1.5 <-
    m = 12
  -> State: 1.6 <-
    m = 24
  -> State: 1.7 <-
    m = 25
  -> State: 1.8 <-
    m = 50
  -- Loop starts here
  -> State: 1.9 <-
    m = 100
```

4.2 Modelling Sokoban

Now we can start modelling Sokoban instead of marbles. Our model will be (part of) a Sokoban puzzle. The state space then is every possible configuration of the puzzle, with the starting configuration as the initial state. Our claim will be that the puzzle cannot be solved. Usually, we expect this to be untrue, and expect a solution to exist. If so, NuSMV will find out, and give us a shortest solution.

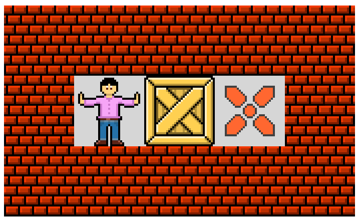
This approach works well for small puzzles. However, it doesn't scale well to the size of puzzle we are working with, quickly taking too long to reasonably process. On top of that, by itself we cannot use it to prove exponentiality. Instead, we use model checking in conjunction with our induction. We used a working hypothesis in order to abstract the room and reduce player movement to a few transitions. We will be using NuSMV to support this hypothesis.

Before we can use NuSMV on our construction, we must first transcribe Sokoban to NuSMV's desired input format, which we automated using the programming language Python. We represent a puzzle by assigning a unique variable to each space. Each space can either be empty, occupied by a box, or occupied by the player. Therefore, we have 3 allowed values (numbered 1 to 3). This means we ignore walls, as they are not relevant to the state space. We only use a single initial state, which is the starting configuration of the puzzle.

We then list every possible transition (which is a very long list) by finding all legal moves. A player move can be found by listing all pairs of adjacent spaces: if one space is occupied by the player and the other is empty, the player can move to the empty space. A box push needs 3 adjacent spaces in a line, with values 3 2 1: the player at one end, a box in the middle, and an empty space behind the box. This transition results in values 1 3 2: the box has been pushed one space and the player has taken its place. Of course, these transitions can be both horizontal and vertical, in both directions.

The transition constraint becomes a disjunction of all allowed transitions. A transition consists of a case distinction with three cases: the legal move its based on, the legal move in the opposite direction, and a default case where no move is made. (Recall that a case distinction must always have one applicable case.) Additionally, the transition requires that all unaffected spaces do not change.

For our claim Φ, we require that all goal space variables have the value 2, i.e. a box is present.



Above is a tiny Sokoban puzzle. If we convert this into input for NuSMV, it looks a little like this (formatted for readability):

```
MODULE main
VAR
x1y1 : 1..3;
x2y1 : 1..3;
x3y1 : 1..3;
INIT
x1y1 = 3 & x2y1 = 2 & x3y1 = 1
TRANS
(
  case
    x1y1 = 3 & x2y1 = 1 :
      next(x1y1) = 1 & next(x2y1) = 3 ;
    x1y1 = 1 & x2y1 = 3 :
      next(x1y1) = 3 & next(x2y1) = 1 ;
    TRUE :
      next(x1y1) = x1y1 & next(x2y1) = x2y1 ;
  esac
  &
  next(x3y1) = x3y1
) | (
  case
    x2y1 = 3 & x3y1 = 1 :
      next(x2y1) = 1 & next(x3y1) = 3 ;
    x2y1 = 1 & x3y1 = 3 :
      next(x2y1) = 3 & next(x3y1) = 1 ;
    TRUE :
      next(x2y1) = x2y1 & next(x3y1) = x3y1 ;
  esac
  &
  next(x1y1) = x1y1
) | (
  case
    x1y1 = 3 & x2y1 = 2 & x3y1 = 1 :
      next(x1y1) = 1 & next(x2y1) = 3 & next(x3y1) = 2 ;
    x1y1 = 1 & x2y1 = 2 & x3y1 = 3 :
      next(x1y1) = 2 & next(x2y1) = 3 & next(x3y1) = 1 ;
    TRUE :
      next(x1y1) = x1y1 & next(x2y1) = x2y1
      &
      next(x3y1) = x3y1 ;
  esac
)
LTLSPEC G !(x3y1 = 2)
```

We can see that the three variables correspond with the three spaces, initialised as in the picture. We then list three main transitions, which are the three large bodies within the parentheses. The first one concerns the two left-most spaces: if one of them contains the player and the other is empty, the player can walk to the other side. The second transition is the same for the two right-most spaces. The third body concerns moving a box: If the middle space contains a box, with the player on one side, and an empty space on the other, the player can move into the space of the box, and the box into the empty space. Finally, we claim that the right-most space can never contain a box. Of course, we can reach this in one simple step:

```
-- specification  G !(x3y1 = 2)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    x1y1 = 3
    x2y1 = 2
    x3y1 = 1
  -> State: 1.2 <-
    x1y1 = 1
    x2y1 = 3
    x3y1 = 2
  -- Loop starts here
```

4.3 Proving the working hypothesis

In section 3.1.1 we showed that the pass-reset only has 6 possible states. We then claimed that only two of these states could occur while the player was outside of the room the pass-reset was in. In this section, we will show why this is, by exploring the pass-reset using model checking.

In the previous section we showed how our program transforms a puzzle into valid input for NuSMV. However, this puzzle has a clear initial configuration and goal. This translates to a single initial state and a single goal state. The room we defined has 6 states, and any of these room states can be in the antecedent or subsequent state of a transition. On top of this, the antecedent and subsequent states of a transition also include the position of the player. Not only do we have to account for several possible initial states, dependent on the room state and the position of player, but we have the added requirement that the player must be at a specific exit in the goal state.

We now want to prove that only two room states can occur in any given room. Clearly, it this will require some work.

4.3.1 Transitions through the pass-reset

We start off with the pass-reset device, by listing every conceivable transition. These transitions use an ordered pair similar to the one we used in section 3.2. The possible player positions in this case are the three tunnels that lead into the pass-reset: the left tunnel of the partial pass-reset device, the bottom tunnel of the reverser, and the top tunnel of the partial pass-reset device. We name these A', B', C'. There are six states the pass-reset device can be in. State 0, state 1, and state 2 match those of the partial pass-reset device, and have the reverser in its initial state. We add three to this state number if the reverser is in the flipped state.

Every transition becomes a puzzle on its own. Instead of using the original goal spaces, we create a new puzzle where the new goal spaces match the positions of the boxes in the subsequent state. The boxes start in the configuration of the antecedent state. The player starts in the exit of the antecedent state, and we add to our claim Φ that the player must be at the exit of the subsequent state. Of course, we limit player movement to within the pass-reset.

Each one of these transitions is run through NuSMV separately. We have used Python to automate this. NuSMV will determine if a solution for the given puzzle exists: if it does, the corresponding transition is possible. Below is a list of all possible transitions found through this process.

$(A',1) \mapsto (A',2)$	$(A',1) \mapsto (C',0)$	$(B',2) \mapsto (A',5)$
$(A',4) \mapsto (A',5)$	$(A',2) \mapsto (C',0)$	$(B',0) \mapsto (B',1)$
$(A',4) \mapsto (B',0)$	$(A',4) \mapsto (C',3)$	$(B',1) \mapsto (B',0)$
$(A',4) \mapsto (B',1)$	$(A',5) \mapsto (C',3)$	$(B',2) \mapsto (B',0)$
$(A',4) \mapsto (B',2)$	$(C',3) \mapsto (B',0)$	$(B',2) \mapsto (B',1)$
$(A',5) \mapsto (B',0)$	$(C',3) \mapsto (B',1)$	$(B',0) \mapsto (C',3)$
$(A',5) \mapsto (B',1)$		$(B',1) \mapsto (C',3)$
$(A',5) \mapsto (B',2)$		$(B',2) \mapsto (C',3)$

While this is a large list, we can drastically reduce the number of transitions by pruning those that cannot occur in practice. Observe that no transition has 4 in its subsequent state. Since 4 cannot be reached, we can eliminate all transitions with 4 in the antecedent state. We can also remove all transitions starting in C', since the one-way device between C and C' prevents us from ever entering the pass-reset from there. This removes the only transitions starting with 3, so transitions resulting in 3 can be scrapped too. Similarly, the one-way device at A' prevents us from leaving there, so we can remove the transitions leading to A'. Since this removes the only transition resulting in 5, we can remove all transitions starting in 5 as well. In the end, our new list is much shorter:

$$\begin{array}{c|c|c} (A',1) \mapsto (C',0) & (B',0) \mapsto (B',1) & (B',2) \mapsto (B',0) \\ (A',2) \mapsto (C',0) & (B',1) \mapsto (B',0) & (B',2) \mapsto (B',1) \end{array}$$

We have now reduced the number of existing states from 6 to 3, completely eliminating the cases where the reverser is flipped.

4.3.2 Transitions through the room

We can now expand our scope to start at the exits of the room rather than those of the pass-reset. Here the one-way devices we used will prove helpful. We know the player cannot enter the room at B or C because of these. The only difference between this system and the system we used for the pass-reset is the added transition between A and B, which keeps the state of the room unchanged: this is the path through the one-way device between the two junctions. After adding this, we start with these transitions:

$$\begin{array}{c|c|c} (A,1) \mapsto (C,0) & (A,0) \mapsto (B,1) & (A,2) \mapsto (B,0) \\ (A,2) \mapsto (C,0) & (A,1) \mapsto (B,0) & (A,2) \mapsto (B,1) \\ (A,0) \mapsto (B,0) & (A,1) \mapsto (B,1) & (A,2) \mapsto (B,2) \end{array}$$

Notice how no transition leads to state 2, except a transition starting in state 2. Since the initial state of the room is 0, we know state 2 can never be reached. Consequently, there are only two states that can occur. If we list the remaining transitions, we can see that they match the transitions in section 3.2.1, concluding our proof.

$$\begin{array}{c|c|c} (A,1) \mapsto (C,0) & (A,0) \mapsto (B,0) & (A,1) \mapsto (B,0) \\ & (A,0) \mapsto (B,1) & (A,1) \mapsto (B,1) \end{array}$$

Chapter 5

Related Work

Sokoban was first proven to be PSPACE-complete by Culberson. In [2], he uses a Sokoban puzzle to emulate a finite tape Turing Machine. This puzzle only has a solution if and only if the corresponding Turing Machine accepts and halts. In doing so, he proves that Sokoban is PSPACE-complete. He also introduces the devices used in many other constructions, including our own. These are the reverser, pass-reset, and one-way device. He credits the one-way device to the unpublished work 'On the complexity of Sokoban', by Stephen Sabey. Much of the proof for these devices is implicit: we have expanded on this proof, and have then abstracted the devices.

Sokoban was again proven to be PSPACE-complete by Hearn and Demaine [5]. They define the Nondeterministic Constraint Logic (NCL) model, which uses a weighted, directed graph with vertices that require a minimum in-flow. They also define a subset of NCL graphs consisting only of AND- and OR-verteces. They create several simple games using these graphs, where the player can switch the flow of edges while maintaining the minimum in-flow. By reduction to Quantum Boolean Formulas, they show these games are PSPACE-complete. Then they reduce several existing games to these new games, including Sokoban. Their proof holds even when there are no walls present, although these typically border off a traditional Sokoban puzzle.

One of the earliest automated solvers for Sokoban is ROLLING STONE, by Junghanns and Schaeffer. [6] Their work manages to solve 12 of 90 puzzles in the standard test set XSokoban [7]. This accomplishment is a benchmark for other solvers, but also is a testimony to the sheer difficulty of Sokoban. ROLLING STONE is an augmented version of IDA*, and deals with the complexity of Sokoban through a variety of optimisations. Two major components are the use of a look-up table to recognise several forbidden configurations, and the use of macro moves. Their use of macro moves is somewhat different from ours, as it is not used for abstraction but for optimisation. We have used macro moves to represent any series of moves that leads from one state to another. Finding the optimal series is not our concern, as long as we can still prove exponentiality. ROLLING STONE uses macro moves for optimisation: it looks for series of moves where once the first move in the series has been performed, the others are inevitable. By immediately completing the entire series, this significantly reduces the steps that the algorithm needs to consider.

Botea, Müller, and Schaeffer provide an alternative to Rolling Stone, called POWER PLAN. [1] They take a planning approach, for which they create Abstract Sokoban. They separate puzzles into tunnels and 'rooms' (we referred to these as areas/devices, since we later combine these into larger systems), and treat rooms as black boxes. In doing so, they abstract individual player movement into macro moves consisting of several steps. They then analyse individual rooms as well as the combination of the black boxes. Their method formed the basis of our approach, though the implementation was done independently and differs in some aspects. Of course, we also had a different goal in mind.

Another solver for Sokoban, was presented by Wesselink and Zantema [8]. The other solvers mentioned here are based on heuristics, but this solver performs an exhaustive search. They model Sokoban as a reachability problem. They use Binary Decision Diagrams (BDDs) to represent the configurations of a puzzle. They overcome the memory problems this approach has by using Reduced Order BDDS (ROBDDs). They also reduce the state space by looking for dead spots: empty spaces from where a box cannot be pushed to a goal space. As a result of these optimisations, they are able to find optimal solutions. Our approach is more similar to theirs than to other solvers, as it is also based on reachability. However, optimisation was not a major goal for us. We chose not to build our own tool but use NuSMV instead, though NuSMV is also based on BDDs.

Chapter 6

Conclusions

We have shown that any solution for our construction requires an exponential amount of steps. To do so, we have abstracted Sokoban using macro moves, resulting in a transition system with two states per room. We have done this through induction, showing that each additional room doubles the required transitions starting in A_1 needed to complete the puzzle. We have closely examined the properties of the devices used in the room, and have used NuSMV to prove the properties of the pass-reset device.

I would like to thank my supervisor Hans Zantema for his input, comments, and everlasting patience.

Bibliography

[1] Adi Botea, Martin Müller, and Jonathan Schaeffer. Using abstraction for planning in sokoban. In Jonathan Schaeffer, Martin Müller, and Yngvi Björnsson, editors, *Computers and Games*, pages 360–375. Springer Berlin Heidelberg, 2003.

[2] Joseph Culberson. Sokoban is pspace-complete. Technical Report TR 97-02, Department of Computing Science, University of Alberta, 1997.

[3] NuSMV development team. NuSMV. `http://nusmv.fbk.eu/`.

[4] Erich Friedman. Problem of the month, March 2000. `https://www2.stetson.edu/~efriedma/mathmagic/0300.html` [retrieved 5 March 2018].

[5] Robert Hearn and Erik Demaine. Pspace-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *CoRR*, cs.CC/0205005, 2002. `http://arxiv.org/abs/cs.CC/0205005`.

[6] Andreas Junghanns and Jonathan Schaeffer. Sokoban: A challenging single-agent search problem. In *Proceedings of the IJCAI Workshop on Using Games as an Experimental Testbed for AI Research*, pages 27–36. Department of Computing Science, University of Alberta, 1997.

[7] Andrew Myers. Xsokoban. `http://www.cs.cornell.edu/andru/xsokoban.html`.

[8] Wieger Wesselink and Hans Zantema. Shortest solutions for sokoban. In Tom Heskes, Peter Lucas, Louis Vuurpijl, and Wim Wiegerinck, editors, *Proceedings of the 15th Belgian-Dutch Artificial Intelligence Conference (BNAIC' 03)*, pages 323–330, 2003.