master thesis in computer science

by

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements for the degree of Master of Science

supervisor: assoc. Prof. Dr. Georg Moser, Institute of Computer Science

Innsbruck, 28 December 2015

Master Thesis

Classic Nintendo Games are Completely Hard

Josef Lindsberger
josef.lindsberger@student.uibk.ac.at

28 December 2015

Supervisor: assoc. Prof. Dr. Georg Moser

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Die vorliegende Arbeit wurde bisher in gleicher oder ähnlicher Form noch nicht als Magister-/Master-/Diplomarbeit/Dissertation eingereicht.

Datum	Unterschrift

Abstract

The complexity of (classic Nintendo) games like Super Mario Bros., Donkey Kong Country and Metroid has been of interest to several researchers, which led to multiple articles introducing methods to prove complexity of such games. In a sequence of articles [2,3,4] by Aloupis et al., results for several such games are given. They introduced frameworks to show hardness of games for NP and PSPACE. Those frameworks combine the functionality of several components, so called gadgets. If these gadgets can be realized using a given games' mechanics, that game is NP- resp. PSPACE-hard. The frameworks, reducing from SAT resp. QSAT, specify game worlds given logic formulas. Playing the game represents the search for a satisfying assignment for the formula. If the player manages to reach the finish location, the formula is satisfiable. There have been realizations of gadgets that turned out to be imperfect due to a too loose or even missing formal definition of the introduced gadgets, which allowed flaws to occur that led to incorrect results. The aim of this thesis is to refine the results in the article of Aloupis et al. by providing precise and detailed definitions. This is necessary to avoid flaws in the actual implementations of the frameworks for specific games. Also, the gadgets of the PSPACE-framework have been realized using the mechanics of Super Mario World, which proves this game to be PSPACE-hard and, performing a further step, PSPACE-complete. As the complementary practical part of this thesis, both NP- and PSPACE-frameworks have been implemented in one actual game, based on the gadgets for Super Mario World. So the player runs through a game world that represents a (quantified) propositional formula and acts as solver to deliver results for SAT and QSAT problems. Further, playing the game might (possibly) wake the player's interest in logic.

Acknowledgments

The author is grateful for the various forms of support he got while working on this thesis. He wants to thank his family for constantly pushing him as it got stressful at some times and he seemed to lose his interest in completing his work. He also sends huge thank towards his grandmother, who was a great financial support. There also go special thanks to his passed away dog, whose walks gave him the atmosphere and time to think about some of the problems that had to be solved. He wants to thank Bertram Felgenhauer as well, who gave some support, hints and ideas that helped to write this thesis. There also go thanks to The Spriters Resource[1] for providing the sprites that are used to create the images of this thesis and used in the implementation. Least but not last, the author wants to thank himself for choosing the right supervisor. And last but most valuable, the supervisor Georg Moser shall receive massive thanks for giving the author the opportunity to work on this very interesting topic and for his patience as the author proved to be hard of understanding when the discussion about that special part of the thesis came up. You know what I mean!

[1] `http://www.spriters-resource.com/`

Contents

1 Introduction

Video games have established themselves over the last decades to be a huge part of society. Not only as a funny way to spend spare time but also as interesting and challenging programming tasks and, in the sense of this thesis, to be an interesting field of complexity theory. Over the years, the complexity of many games has been analyzed and several methods to prove those games' complexities have been introduced by multiple researchers around the world in their articles. One of those articles [4] has been published by Aloupis et al. They are focussing on classic Nintendo games like Super Mario Bros.[1], the Donkey Kong Country[2] series, several Legend of Zelda[3] games, Metroid[4] and Pokemon[5]. They are introducing two frameworks that consist of multiple components (also known as gadgets). These frameworks are reducing from problems known to be complete for different complexity classes. Therefore hardness of the frameworks for that complexity classes follows. In particular, one framework (logspace-) reduces from SAT, which is NP-complete, while the other framework reduces from QSAT, which is PSPACE-complete. The complexity class NP describes the set of problems for which a solution can be guessed non-deterministically and verified within polynomial time by a deterministic Turing machine. The complexity class PSPACE describes the set of problems that are decidable within polynomial space on a deterministic Turing machine. If the game mechanics of a given game allow to implement the functionality of each gadget of a framework, (a generalized version of) that game is a member of the "framework problem", which means that the game is NP- or PSPACE-hard, depending on the chosen framework. As the frameworks behave equivalently to the problems they are reducing from, the NP-framework represents an instance of SAT, in particular a formula in CNF, while the PSPACE-framework represents an instance of QSAT, in particular a formula in prenex normal form.

Both frameworks represent their formulas as a graph from which actual game worlds can be created, based on the implementations for specific games. While playing such game worlds, the player has to make decisions which one of two literal paths to follow and perform actions in order to reach the finish location. These decisions made by the player are representing a variable assignment. If the finish is reachable, the assignment satisfies the formula. So the player acts as SAT-/QSAT-solver.

[1] http://en.wikipedia.org/w/index.php?title=Super_Mario_World&oldid=656286985
[2] https://en.wikipedia.org/w/index.php?title=Donkey_Kong_Country_(series)&oldid=668297528
[3] https://en.wikipedia.org/w/index.php?title=The_Legend_of_Zelda&oldid=673426701
[4] http://en.wikipedia.org/w/index.php?title=Metroid&oldid=660166538
[5] https://en.wikipedia.org/w/index.php?title=Pok%C3%A9mon&oldid=668579014

More precisely, Aloupis et. at. introduced implementations of the NP-framework for several Super Mario games, Donkey Kong Country 1, several old Legend of Zelda games, Metroid and several Pokemon games. They further gave implementations of the PSPACE-framework for Donkey Kong Country 2 and 3 and several newer Legend of Zelda games. Unfortunately, two gadgets (one for Super Mario Bros. in [2, 3, 4] and one for Metroid in [2]) turned out to be imperfect due to a too loose or even missing formal definition, which allowed flaws to occur that led to incorrect behavior of the framework. The aim of this thesis is to refine the contents of the article of Aloupis et al. by providing precise and detailed definitions. This is necessary to avoid flaws in the actual realizations of the frameworks for specific games. In particular, without a precise definition, leakage possibilities could arise that enable the player to unintendedly change from one path to another, which would break the game.

Also, the gadgets of the PSPACE-framework have been realized using the mechanics of Super Mario World, which proves this game to be PSPACE-hard and, by further showing membership in PSPACE, PSPACE-complete. These results are shown in a spirit similar to the proofs for the metatheorems of Viglietta in [11]. Some of Viglietta's metatheorems built the basis for parts of the work of Aloupis et al. In particular, the special gadgets of the PSPACE-framework have already been introduced in Viglietta's article. He further shows the mechanisms of location traversal and opening doors by using pressure plates and discusses the associated impacts on complexity of different application possibilities of those mechanisms. For example, Super Mario World is shown to be PSPACE-complete by that mechanisms.

As the complementary practical part of this thesis, both NP- and PSPACE-frameworks have been implemented in one actual game, based on the gadgets for Super Mario World. So the player runs through a game world that represents a (quantified) propositional formula and acts as solver to deliver results for SAT and QSAT problems. Further, playing that game might (possibly) wake the player's interest in logic, programming and in general in computer science.

1.1 Example Run

The following series of figures shall demonstrate the use of the implemented PSPACE-framework by a given example formula. Each figure gives some information about the displayed situation or state of the game. Some figures provide explanations of what happened since the previous figure that has not been documented by a separate figure. The example formula consists of only four variables and three clauses, still the resulting game world turns out to be huge. Therefore, not everything that happened is captured by a screenshot.

Figure 1.1 As the program starts, the formula-enter frame shows up. An example formula in prenex normal form is entered: $\forall s \exists p \exists e : \overline{s} \vee e \wedge p \rightarrow p \wedge l$. The player clicks on the parse button to check the syntactic correctness of the input and transform the (matrix part of) the formula to CNF.

Figure 1.2 As the input is syntactically correct, the resulting formula in prenex normal form is shown below the input. If the input would not be syntactically correct, information about that would be displayed instead. The parse button disappeared and the start button is now visible. Via "Mode" in the menu bar, the game mode can be changed between normal and hardcore mode. In hardcore mode, the original formula would be displayed in the game instead of the transformed matrix part in CNF, which is more challenging in making variable assignments as the literals to satisfy cannot be read off the displayed formula directly.

Figure 1.3 The game starts in normal mode and Mario stands on the start location. The major part of the screen is used to render the game. On the top, information about the elapsed time and reached score is shown. On the bottom, the formula in its current state is displayed. On the right, next to the start gadget, the quantifier gadget for the free variable l begins. Free variables have to be treated as existentially quantified and visited first such that player makes one (permanent) assignment on them. Free variables are not displayed in the prefix part of the formula.

Figure 1.4 Mario follows the literal path for l and reaches the first crossover gadget. He has to wait until the rotating firebars let him quickly pass horizontally. Leakage to the vertical path is not possible due to the layout of the firebars.

Figure 1.5 Mario grabs the movable trampoline on the bottom (which was hidden behind the other/fixed trampoline) and carries it up the open path of the currently visited door gadget. Note that due to the game mechanics, Mario cannot leave the open path by climbing the vine, while carrying the trampoline.

Figure 1.6 Mario drops the trampoline down the narrow section where it lands on the rotating block. Now the door gadget is traversable and the corresponding literal l in the second clause is satisfied. Therefore it is highlighted green in the formula.

Figure 1.7 After Mario has visited all clauses in which literal l occurred, the literal path leads him back to the existential quantifier gadget.

Figure 1.8 As the literal $\neg l$ does not occur in any clause, its closing literal path directly connects to its back path and Mario can continue to the next quantifier gadget.

Figure 1.9 Mario reaches the universal quantifier gadget for s and is forced to follow literal path s.

Figure 1.10 All doors corresponding to literal s have been opened. Mario reaches the existentially quantified variable p and decides to follow the literal path for $\neg p$.

Figure 1.11 Before literal $\neg p$ is satisfied by opening its door, all doors for literal p have to be closed by jumping against the rotating block from below using the fixed trampoline in the close path. Therefore p is not satisfied and highlighted red in the formula.

Figure 1.12 After Mario opened the door for $\neg p$, he visited the quantifier gadget for e and followed the positive literal path to close the door for $\neg e$. Mario now traverses the check path. As literal s satisfies the first clause, Mario can use the placed trampoline to jump up and continue traversing the check path.

Figure 1.13 Mario traversed the check path successfully as all clauses have been satisfied. He followed the check path back through the internal paths of the quantifier gadgets until he reached the universal quantifier gadget for s. He now is forced to follow the literal path for $\neg s$. This includes closing all doors corresponding to literal s.

Figure 1.14 Mario closed all doors representing literal s (no doors for $\neg s$ could be opened). In order to satisfy the first clause, he had to follow literal path p which forced to close the door of $\neg p$ in the last clause. As the second and third clause kept being satisfied by l, the assignment of variable e did not matter. Mario followed literal path e again. As all clauses were satisfied, Mario could successfully traverse the check path through all clause gadgets. He now follows the check path back to the quantifier gadgets.

Figure 1.15 Mario traversed the check path through the internal paths of the quantifier gadgets. As both literal paths of the only universally quantified variable s have already been processed, the check path leads Mario to the finish gadget, where he touches the finish flag and wins the game.

Figure 1.16 The game-over panel shows up, offering options to restart the game or end it and return to the formula-enter frame.

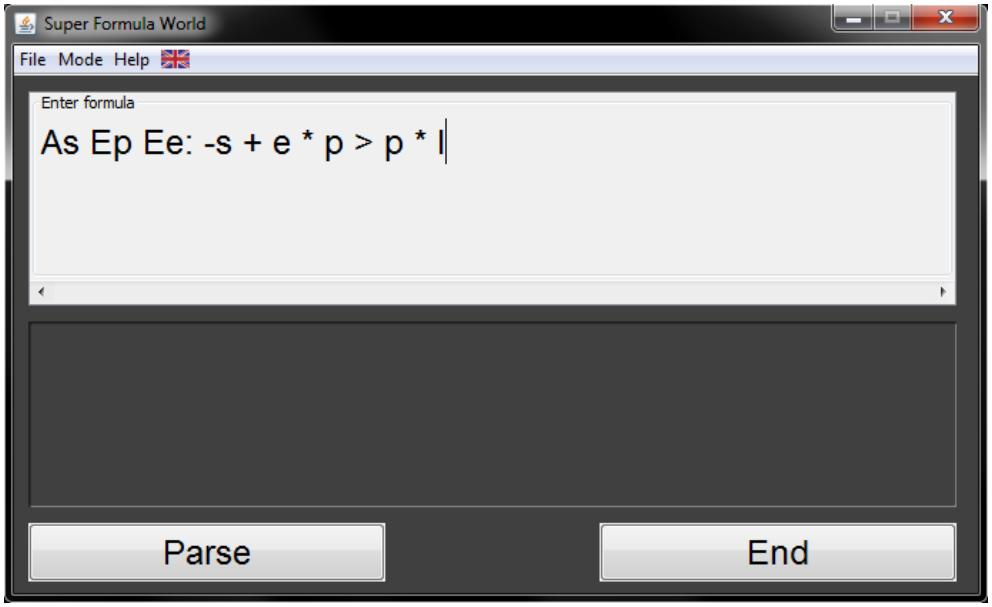

Figure 1.1: Example run

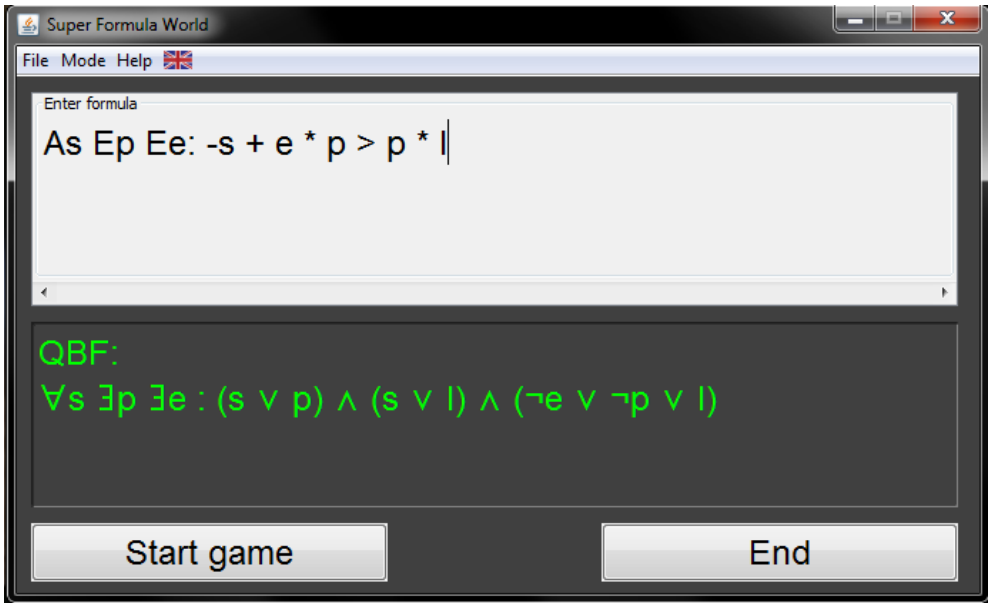

Figure 1.2: Example run

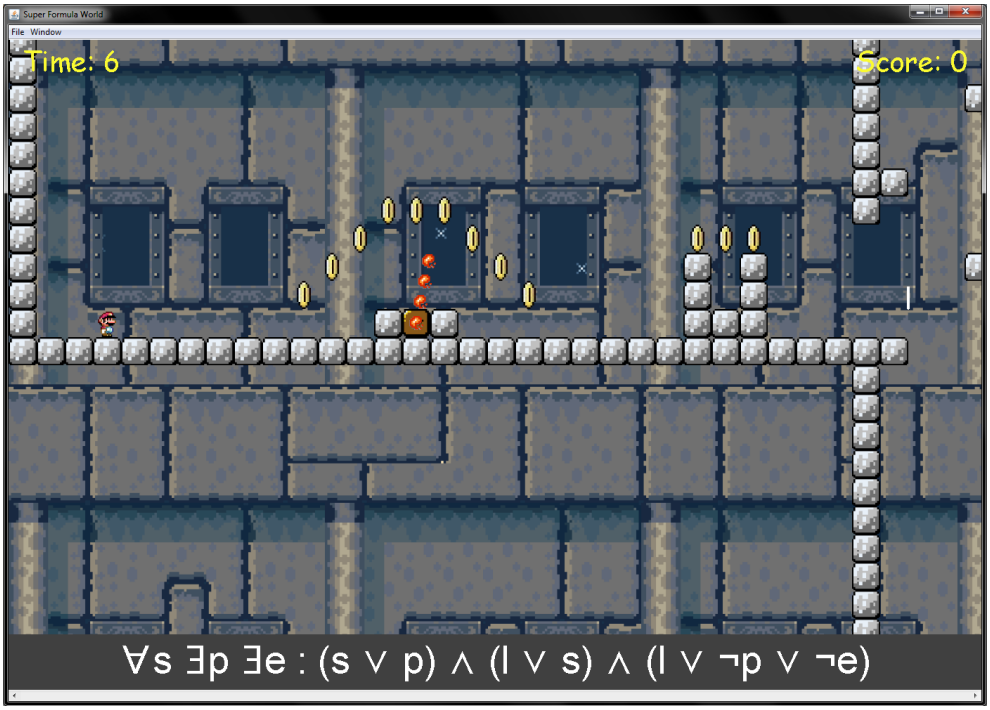

Figure 1.3: Example run

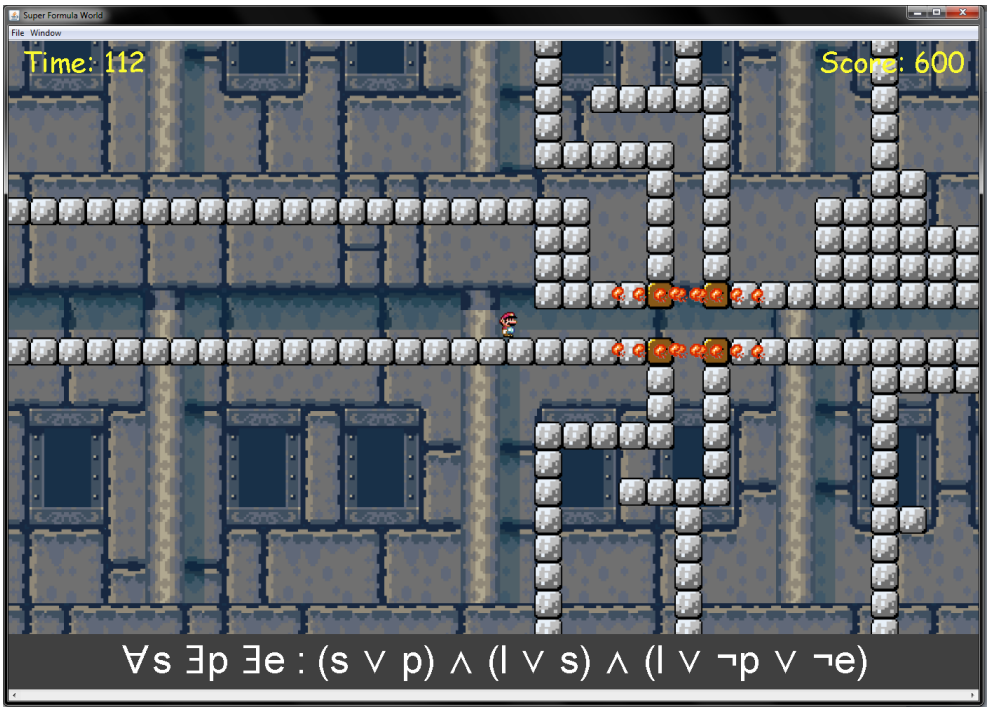

Figure 1.4: Example run

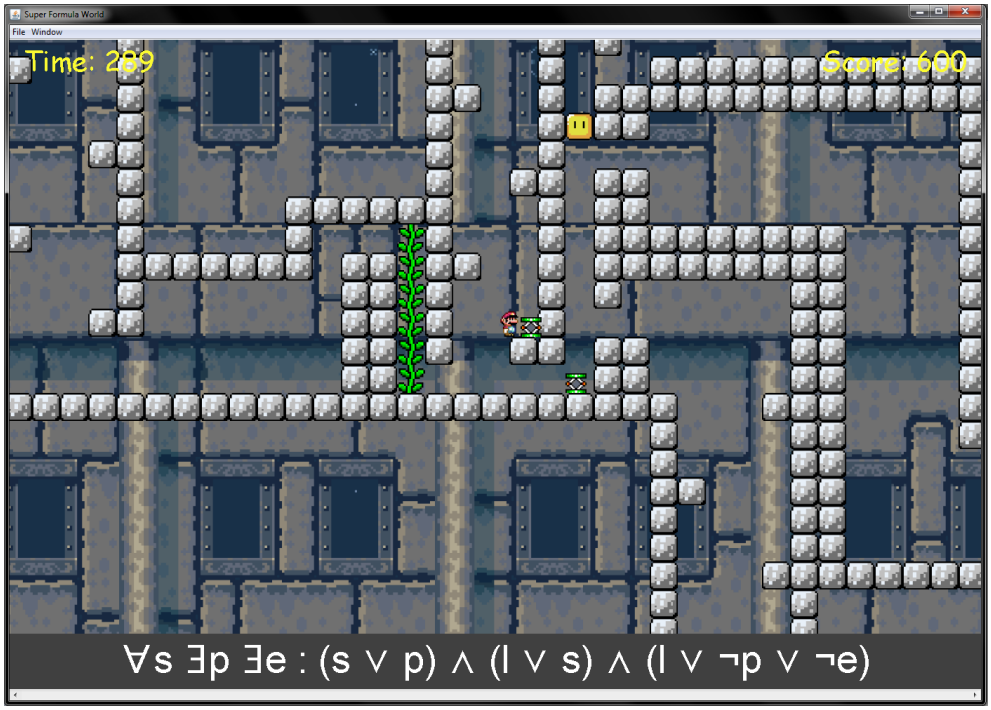

Figure 1.5: Example run

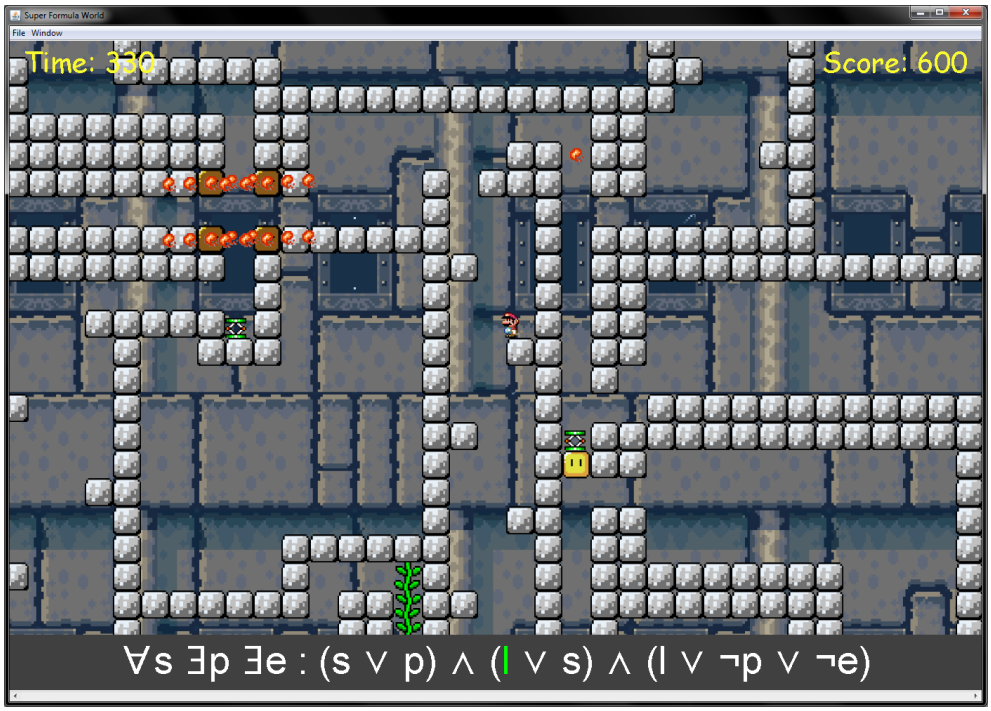

Figure 1.6: Example run

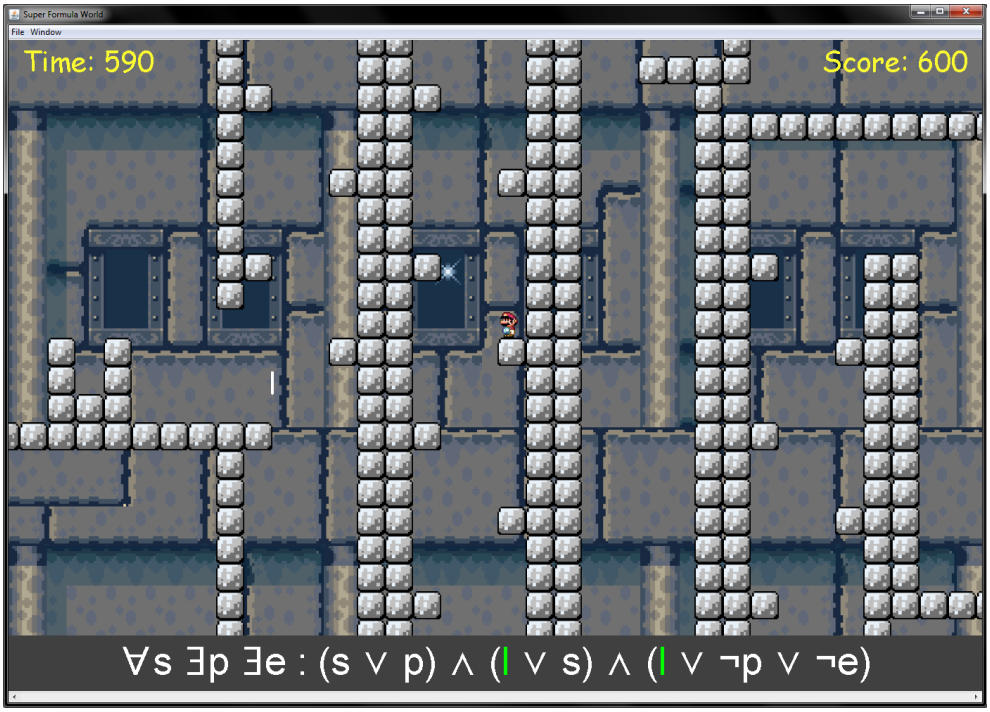

Figure 1.7: Example run

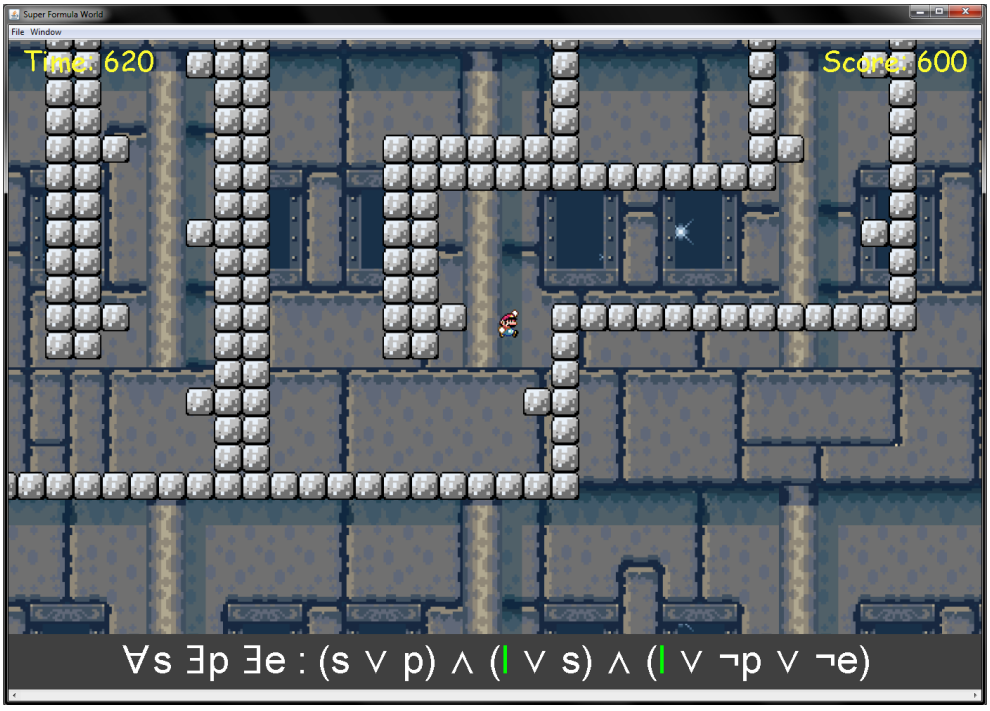

Figure 1.8: Example run

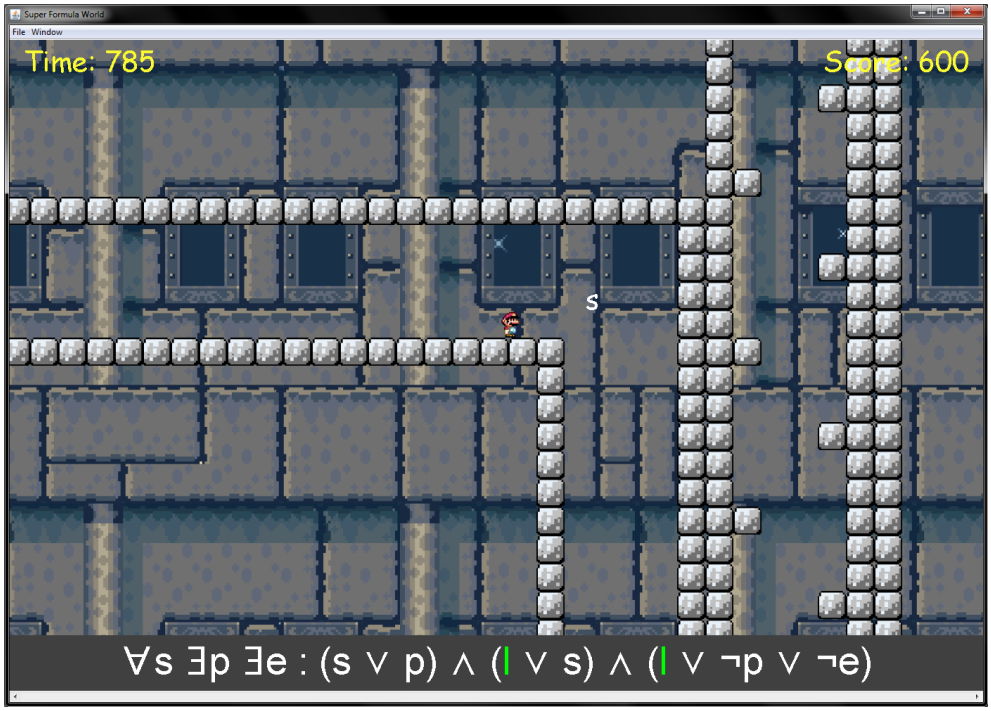

Figure 1.9: Example run

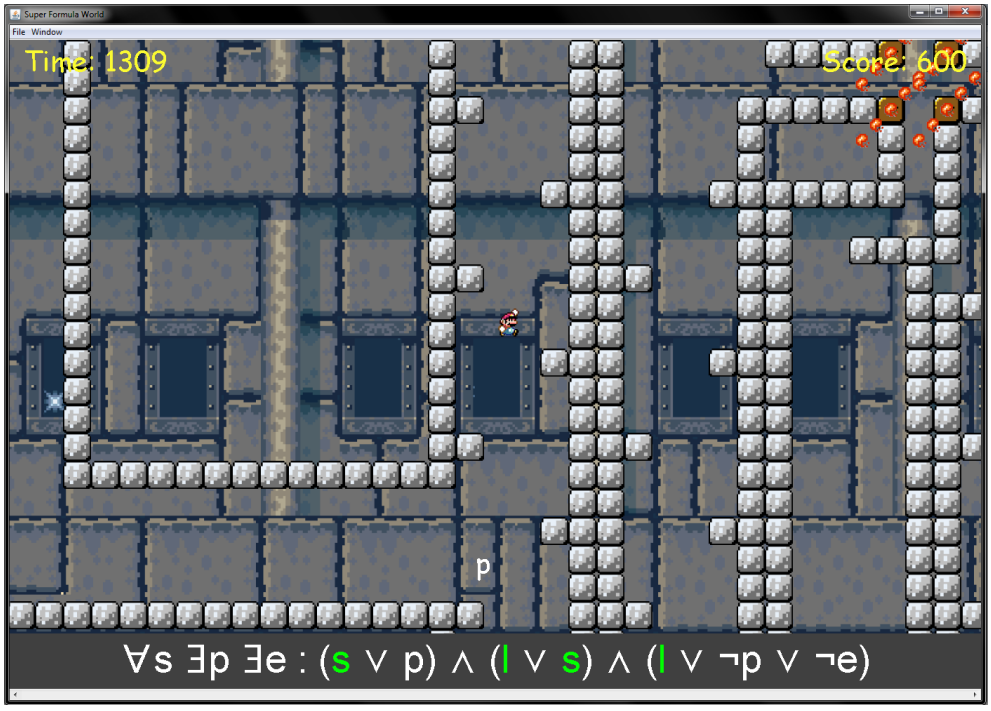

Figure 1.10: Example run

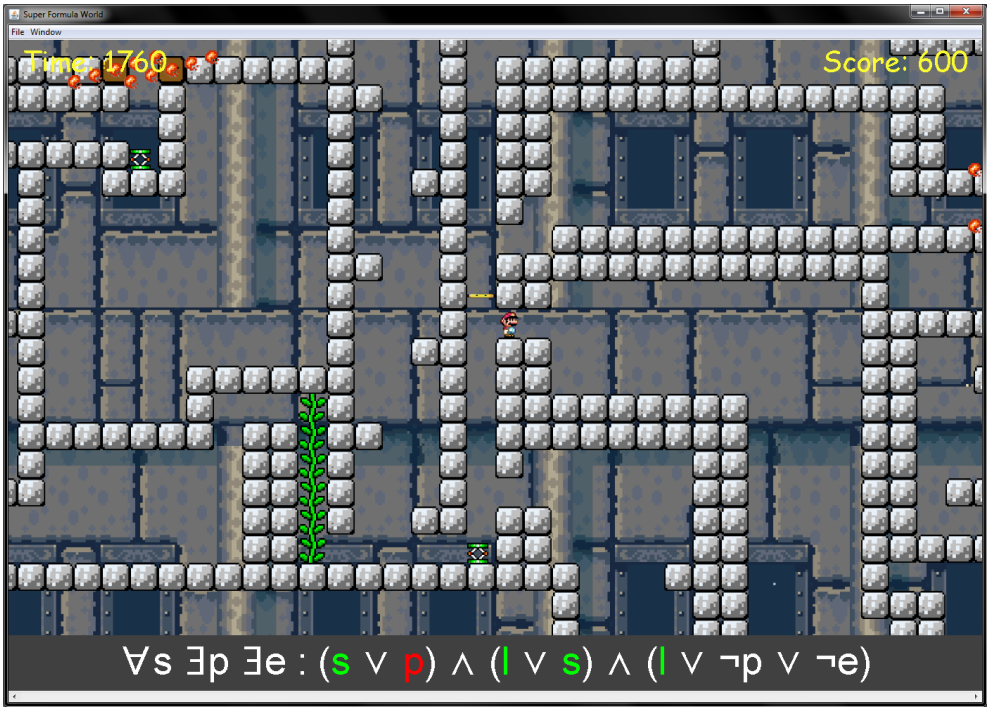

Figure 1.11: Example run

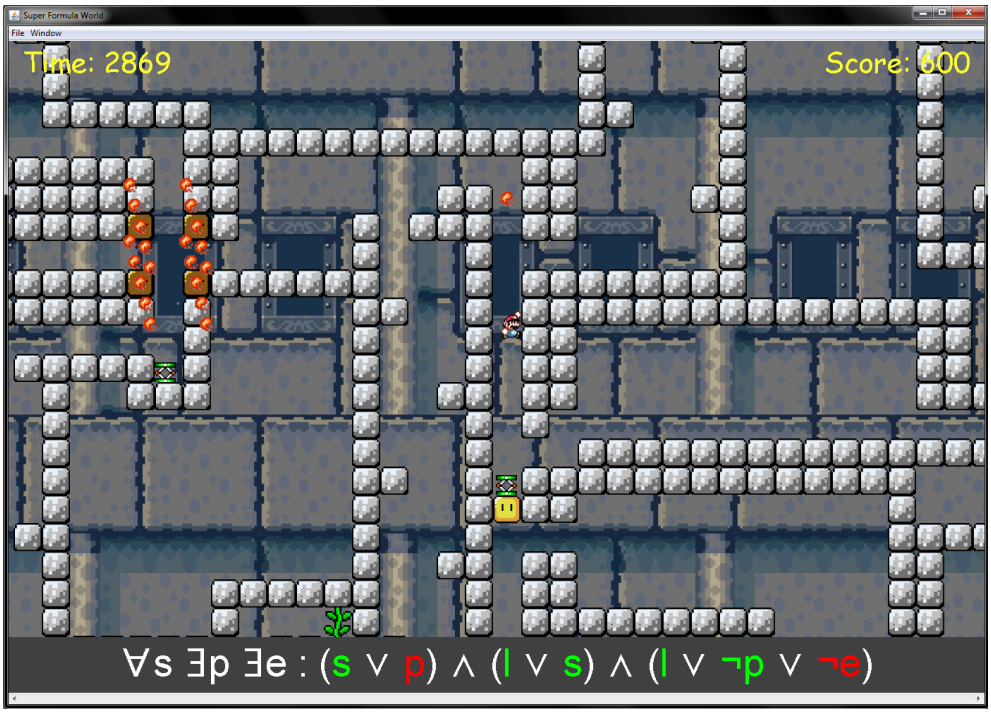

Figure 1.12: Example run

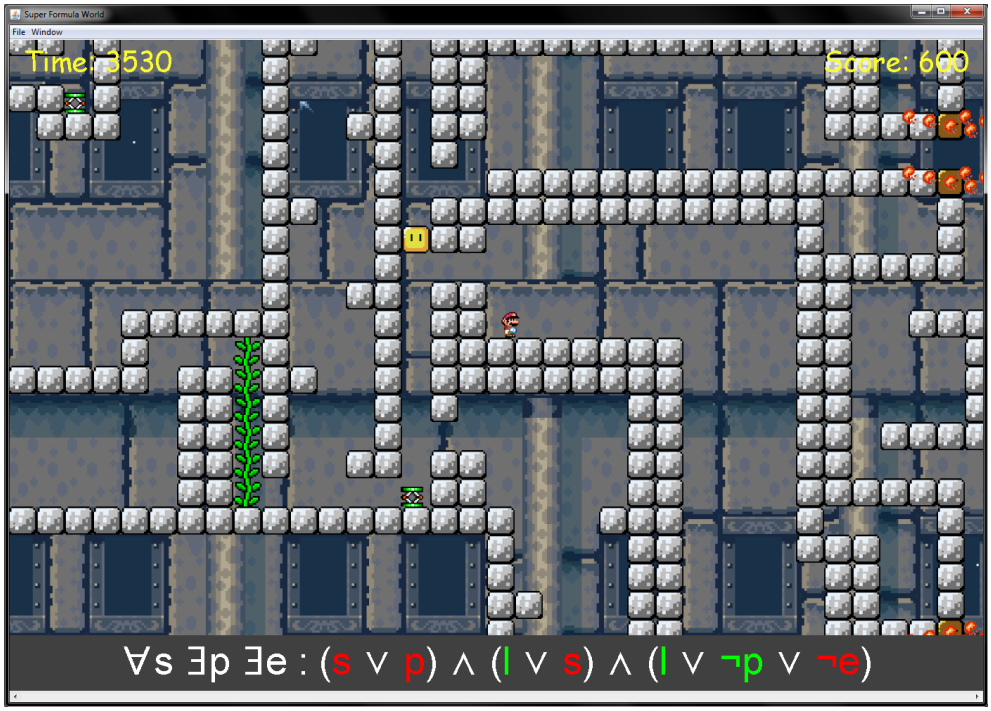

Figure 1.13: Example run

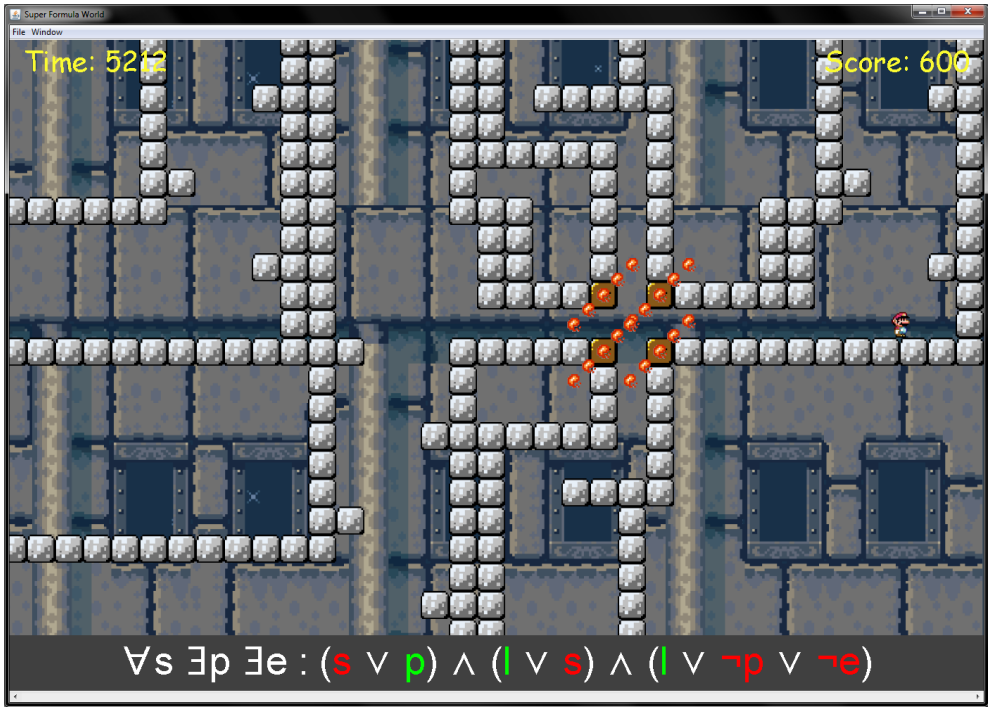

Figure 1.14: Example run

Figure 1.15: Example run

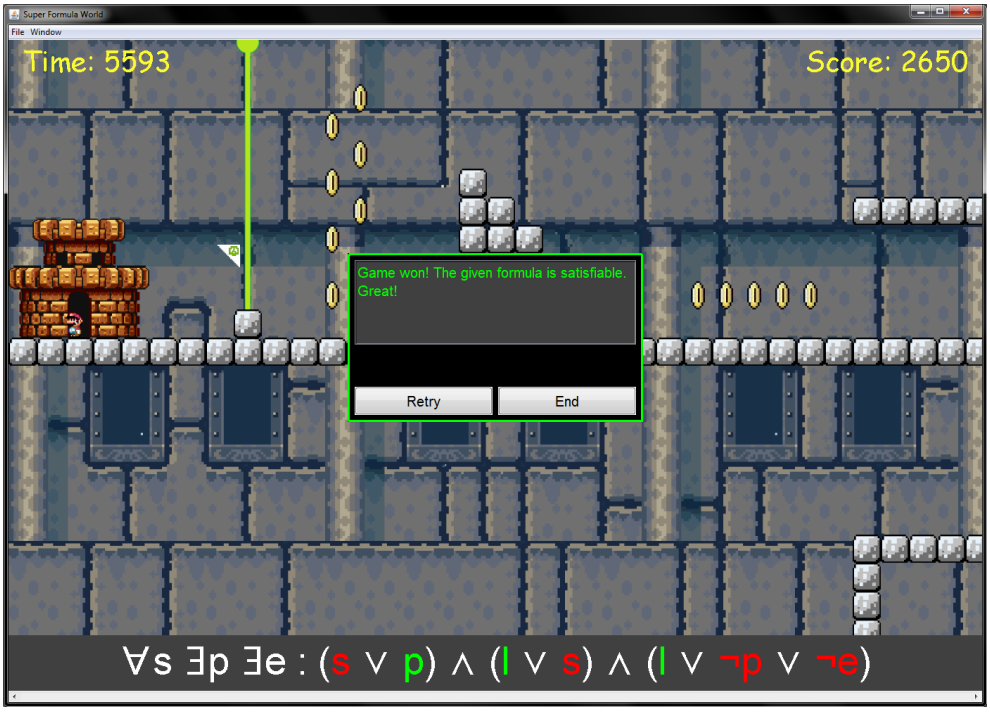

Figure 1.16: Example run

1.1.1 Structure

The structure of this thesis looks as follows: First, in Chapter 2, the most important terms are defined and a basic introduction in Super Mario games is given. Chapter 3 explains the NP-framework. This consists of its multiple construction steps and a look at the flaws that existed in the (first) realizations of the crossover gadgets for two games. In Chapter 4, the PSPACE-framework is explained by elaborating each construction step. Further, an instantiation of this framework for Super Mario World is introduced. The reductions of both the NP- and PSPACE-framework are given and formally proven in their corresponding chapters as well. Chapter 5 explains how both frameworks are implemented as a Java program. This includes a high-level perspective on the physics engine and on how game worlds are created. Chapter 6 shows how the program is used by explaining the GUI, controls and how to install / run the program. In Chapter 7, some articles regarding complexity results of several games are mentioned, while Chapter 8 concludes this thesis.

2 Preliminaries

In this chapter, the most important terms used in this thesis will be defined. Also, an introduction to Super Mario will be given.

2.1 Definitions

Definition 2.1 (Turing machine [10])**.** A Turing machine (TM) is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states
2. Σ is the input alphabet not containing the blank symbol $\sqcup$
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function
5. $q_0 \in Q$ is the start state
6. $q_{accept} \in Q$ is the accept state
7. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$

Definition 2.2 (Non-deterministic Turing machine [10])**.** A non-deterministic TM is a TM whose transition function offers multiple ways to proceed computation, instead of one unique (deterministic) way: $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$

Definition 2.3 (Turing machine with I/O)**.** A TM with I/O is a three-tape TM with two additional tapes besides the work tape, one tape for its input and one for its output. The transition function is $\delta : Q \times \Gamma^3 \to Q \times \Gamma^3 \times \{L, R, S\}^3$, where S denotes, that the head of a tape may stay on its position/is not forced to move.

In the following, the classes TIME($t(n)$) and NTIME($t(n)$) are defined, upon which NP is defined.

Definition 2.4 (TIME($t(n)$) [10])**.** Let $t : \mathbb{N} \to \mathbb{R}^+$ be a function. Define a time complexity class, TIME($t(n)$), to be the collection of all languages that are decidable by an $O(t(n))$ time deterministic Turing machine.

Definition 2.5 (NTIME($t(n)$) [10])**.**

$$\text{NTIME}(t(n)) = \{L | L \text{ is a language decided by a } O(t(n)) \text{ time nondeterministic Turing machine}\}$$

Corollary 2.6 (NP [10])**.** $NP = \bigcup_k NTIME(n^k)$.

Analogously, space complexity classes can be defined.

Definition 2.7 (SPACE($t(n)$) [10])**.** Let $t : \mathbb{N} \to \mathbb{R}^+$ be a function. Define a space complexity class, SPACE($t(n)$), to be the collection of all languages that are decidable by an $O(t(n))$ space deterministic Turing machine.

Definition 2.8 (NSPACE($t(n)$) [10])**.**

$$\text{NSPACE}(t(n)) = \{L | L \text{ is a language decided by a } O(t(n)) \text{ space nondeterministic Turing machine}\}$$

Definition 2.9 (PSPACE [10])**.** PSPACE is the class of languages that are decidable in polynomial space on a deterministic Turing machine. In other words, PSPACE = $\bigcup_k \text{SPACE}(n^k)$.

Definition 2.10 (L / logspace [10])**.** The complexity class L describes the set of problems that are decidable within a logarithmic amount of space in terms of the input size on a deterministic Turing machine: L = SPACE($\log n$) This class is also referred to as logpsace.

Theorem 2.11 (Savitch's Theorem [10])**.** *For any function $f : \mathbb{N} \to \mathbb{R}^+$, where $f(n) \geqslant n$, $NSPACE(f(n)) \subseteq SPACE(f^2(n))$.*

From Savitch's Theorem follows that deterministic Turing machines can do the same as non-deterministic Turing machines with only a quadratic overhead in terms of needed space. From this follows the equivalence NPSPACE = PSPACE, as a squared polynomial stays polynomial.

Definition 2.12 (Literals)**.** Given a set of variables V, the set of literals L is defined as follows: $L = \{v, \overline{v} \mid v \in V\}$. Literals are Boolean variables or their negations.

For the sake of readability at a later point in the thesis, the $\overline{v}$ notation is preferred over the standard $\neg v$ notation for negative literals.

Definition 2.13 (CNF)**.** A formula is in conjunctive normal form (CNF) if it is a conjunction of clauses, where each clause is a disjunction of literals. A CNF formula F has the following form: $F = \bigwedge_{1 \leqslant i \leqslant |\mathcal{C}|} C_i$, with $C_i = \bigvee_{1 \leqslant j \leqslant |C_i|} l_j$.

Definition 2.14 (Prenex normal form (QBF))**.** The prenex normal form adds a quantifier section at the beginning of a CNF formula (called prefix). This states, which variables are existentially ($\exists$) or universally ($\forall$) quantified in the following formula (called matrix). Example 2.15 shows how this looks in practice.

Example 2.15 (Prenex normal form (QBF))**.** The following shows an example formula in prenex normal form:

$$\exists x_1 \forall x_2 ((x_1 \vee x_2) \wedge (x_1 \vee \neg x_2))$$

Definition 2.16 (logspace reduction)**.** Let L_1 and L_2 be languages. A function R from L_1 to L_2 computed on a Turing machine M with $\forall x : x \in L_1 \Leftrightarrow R(x) \in L_2$, is called a reduction. If R can be computed on M using a logarithmically size-bound worktape, it is a logspace reduction. Then $L_1 \leq_{log} L_2$ denotes this reduction.

Definition 2.17 (hardness / completeness)**.** A problem X is said to be hard for a complexity class C if every problem in C can be reduced to X within polynomial time. This means X is at least as hard as any problem in C. A problem X is said to be complete for C if it in addition to C-hardness is a member of C.

Definition 2.18 (SAT)**.** Given a Boolean formula, the goal is to decide whether a variable assignment ϕ can be found to make the formula evaluate to true.

If this is the case, the formula is satisfiable by that assignment which leads to the term satisfiability problem (SAT). SAT is the most common NP-complete problem.

Definition 2.19 (Assignment for QBF)**.** In contrast to the SAT problem, a single/primitive assignment ϕ does not suffice to describe the satisfiability of QBF. A QBF assignment Φ describes the set of primitive (satisfying) assignments $\phi \in \Phi$ where ϕ comes from the function $f : \{T, F\}^m \rightarrow \{T, F\}^{n-m}, x \mapsto \phi$, with n denoting the number of (all) variables, m denoting the number of universally quantified variables and $x = x_1, \ldots, x_m$ denoting a combination of universally quantified variables on which ϕ is based.

In other words, there is a primitive assignment for each truth-value combination of the universally quantified variables. A primitive assignment has to satisfy each clause of the formula by setting the existentially quantified variables with respect to a given combination of universally quantified variables x. If for all 2^m combinations of x a satisfying primitive assignment can be found, the given formula is in QBF with its satisfying assignment Φ.

Definition 2.20 (QBF / QSAT)**.** A fully quantified Boolean formula (QBF) is a Boolean formula in which each variable is quantified, either existential or universal. The formula has to be in prenex normal form. The goal is to decide whether there is an assignment Φ for a QBF that evaluates to true. This leads to an alternative term for this problem, namely quantified SAT problem (QSAT). QSAT is the most common PSPACE-complete problem.

Definition 2.21 (directed acyclic graph (DAG))**.** A directed acyclic graph is a graph consisting of nodes and edges where each edge may only be traversable in one direction (directed graph) and in addition no path through the graph may be closed. In other words, there may not be any cycles (acyclic).

2.2 Super Mario

This section gives some insight in the world of Super Mario. There will be a brief view on Mario's history, other actors appearing in his games and the general game mechanics.

2.2.1 History

There is a series of jump-and-run platform games playing in the universe of Super Mario. Mario's girlfriend Peach is constantly being kidnapped by sinister creatures. It all began in 1981 when Mario had do defeat Donkey Kong, who was the first to kidnap Peach. In 1983, Mario's brother Luigi appeared for the first time. Between 1985 and 1988, the Super Mario Bros. series was released, in which the evil Bowser kidnapped Peach with the exception of Super Mario Bros. 2 in which Peach was a playable character among others. In 1990 and 1995, Super Mario World and its sequel were released. Those games introduced Yoshi, a friendly dinosaur. Super Mario 64, released in 1996, was the first 3D Super Mario game. After that, the modern era of Super Mario games began with Super Mario Sunshine (2002), the Super Mario Galaxy series (2007, 2010), the New Super Mario Bros. series (2009, 2012) and Super Mario 3D World (2013).

2.2.2 Actors

This subsection describes some of the most important actors of the Super Mario games.

Mario Mario is an Italian plumber, living in Brooklyn, New York, that constantly has to defend his home and friends from sinister creatures. He is rather short and wearing a red cap and a mustache. He is the main actor of (most of) the games named after him.

Luigi Luigi is Mario's younger brother. He is wearing a green cap and a mustache. Luigi repeatedly appears to help Mario fulfilling his quests.

Peach Peach is Mario's girlfriend. She is the princess to the mushroom kingdom and wearing a pink dress, blonde hair and a golden crown. She is vulnerable to being kidnapped by evil invaders.

Yoshi Yoshi is a friendly, green dinosaur that helps Mario on some of his quests. He can eat enemies and excrete them inside eggs or keep them in his mouth to disgorge later.

Donkey Kong Donkey Kong is a gorilla that kidnapped Peach once long time ago. This is not the same Donkey Kong as the one in the Donkey Kong Country games, still both Donkey Kongs are related.

Koopa Koopa is a tortoise that walks around in Mario's world and bites him if he touches it. Mario can jump on a Koopa, which knocks it out. Mario can then use Koopa's shell by kicking or carrying it.

Bowser Bowser is an evil, giant Koopa that constantly kidnaps Peach. He has a spiky shell and can spit fire. Sometimes he can throw hammers as well. He lives in a castle full of traps that should keep Peach's rescuers away.

Goomba Goomba is a small, cute-looking animal, consisting only of head and two feet. Goomba bites Mario if he approaches from the side. Jumping on Goomba will kill it.

Mushrooms There are three kinds of mushrooms: A red one that gives strength and turns Mario into Super Mario, a green one that grants Mario an extra life and a violet one, that is poisonous and hurts or kills Mario.

2.2.3 Game mechanics of Super Mario World

Super Mario World is a side-scrolling platform game. The game offers a side-view and actors are influenced by gravity. The goal of the game is to walk through the finish gate in order to end a level. The player has to navigate Mario through a game world and overcome obstacles, while avoiding falling off platforms. The player will encounter enemies and has to defeat or circumvent them. Different enemies have different abilities, so the player has to act accordingly. Mario can jump, crouch, carry specific objects and, after consuming certain items, turn into Super Mario, shoot fire balls or fly using a cloak. Further, there are stars that make Mario invincible for a short amount of time. At some points the player has to overcome height. If his jump height does not suffice, Mario can carry and use trampolines to jump higher. If Mario collides with fire, he gets hurt or dies, depending on whether he is Super or normal Mario. Mario can swim. While in water, gravity is reduced and Mario can jump out of the water if he is on the surface.

For the implementation of the introduced frameworks just a few of the mentioned game elements are needed. Therefore the implementation will offer reduced content, mighty enough to provide the needed functionality.

3 NP-Framework

3.1 Introduction

In [2,3,4], Aloupis et al. introduced a framework to show NP-hardness for (Nintendo) platform games by creating game worlds based on given CNF formulas. This framework reduces from the NP-complete problem SAT: $\forall\phi : \phi \in \text{SAT} \Leftrightarrow R(\phi) \in \text{NP-FRAMEWORK}$. Instantiations for games like Super Mario Bros., the Donkey Kong Country series, several Legend of Zelda games, Metroid and Pokemon have been shown. As the original articles of Aloupis et al. do not give much insight in how that framework has to be constructed, this thesis will give more detail. Unfortunately, the presentation of the framework in [2, 3, 4] is too imprecise and also faulty. This shall be treated here as well.

The framework defines a graph that represents the assignment and evaluation process of a CNF formula. The nodes of this graph are given by so called gadgets. There are different types of gadgets, each providing its own functionality.

Start gadget This gadget represents the start node of the graph. If the game mechanics require a special state (like the Super mushroom for Mario) this state has to be set here.

Variable gadget This represents the assignment of a variable. The player has to decide, whether to follow one or the other outgoing path. It is important to prevent the player from backtracking to a variable gadget and follow the other path.

Clause gadget To satisfy a clause, the contained literals have to be set appropriately. A clause gadget provides entries for those literals, such that the player can satisfy the clause by performing an action. Therefore, the clause gadget has to be initially locked in such a way, that the game mechanics provide means to unlock the gadget. In the case of Super Mario this is done by kicking the shell of a Koopa down the gadget that will smash the blocking bricks. Later in the game, after all variables have been assigned, the player needs to check, whether the assignment satisfies each clause of the formula. Therefore, a clause gadgets needs to provide a check path that can only be traversed if the clause has been unlocked.

Crossover gadget Most likely, some of the paths/edges that connect different gadgets will cross each other. This is caused by the two-dimensionality of those platform games. If there was a third dimension, paths could easily pass each other without collision. As the game worlds are bound to a two-dimensional

grid, crossing paths need to be resolved by the use of crossover gadgets. It is important to prevent the player from leaking from one crossing path to the other. The layout of the crossover gadget strongly depends on the game mechanics.

Finish gadget This gadget represents the finish node of the graph. If the game mechanics required a special state to be set in the start gadget, a final action needs to be performed based on that state in order to reach the actual finish. (Super Mario needs to be big in order to smash bricks.)

Jump-down gadget This is an additional gadget (not part of [4]) that is needed to join two paths such that only one path continues from here in the graph. It is important to prevent the player from leaking from one to the other ingoing path.

Figure 3.1 shows how the original framework introduced in [4] (without jump-down gadget) looks like. Figure 3.2 shows how the rearranged, new version of the framework (including the jump-down gadget), which is going to be introduced in this thesis, looks like.

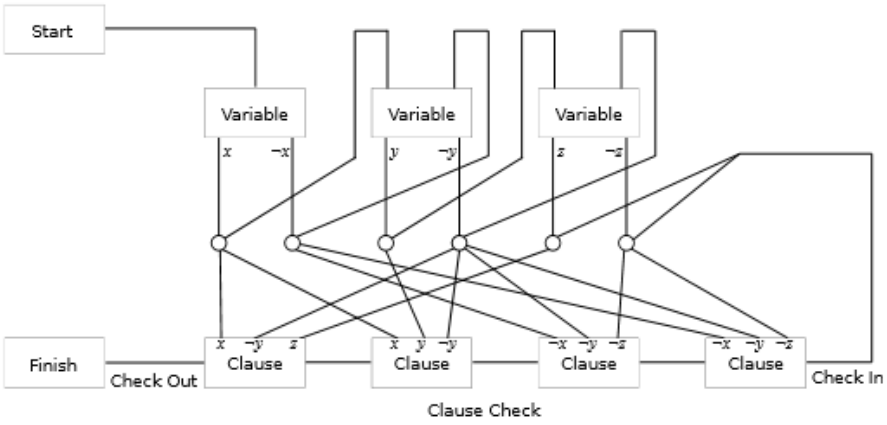

Figure 3.1: Original framework [4]

Explanation of Figure 3.1 The player follows the path from the start gadget to the first variable gadget. Let the player choose literal x and follow its path to the distributor, denoted as circle. The player satisfies all clauses containing literal x one after another by traversing the path to a clause gadget, performing an action to unlock the check path at the bottom of the gadget and going back to the distributor via the path that led to the clause gadget. This procedure repeats until each clause containing x has been unlocked. Then the player follows the path from the distributor to the next variable gadget, where the whole assignment procedure is repeated. After the unlocking process for the last variable gadget has been finished, the player continues to the check path marked as *Checkin* in the figure. Note that the paths from the distributors towards the check path for both literals need to be joined before actually reaching the

check path. In the modified version of the gadget (Figure 3.2), this is achieved by the jump-down gadget. The player then follows the check path through the lower part of the clause gadgets towards the finish gadget. This is only possible if all clauses have been unlocked while traversing the literal paths.

Section 3.2 shows the construction of the framework. This construction is split into multiple steps, each described in its own subsection. Section 3.4 shows flaws that have been observed for Super Mario Bros. and Metroid, introduced by imperfect crossover gadgets. The problem with Metroid existed up to version 2 ([2]) of the original article. In their later versions ([3] and [4]), the flawed gadget for Metroid has been removed.

3.2 Construction of the Framework

In this section the steps involved in the construction of the framework (introduced in [2]) are described in detail. This section concludes with a complexity analysis of the construction. The construction steps used here will lead to a framework that differs from the original one, still providing the same functionality. The changes allow an easier implementation.

Definition 3.1 ($\prec$-order)**.** The $\prec$-order defines an order over the set of literals. Given the set of variables V and the number of variables n, the set of literals L is ordered as follows: $i < j \Rightarrow v_i \prec \overline{v_i} \prec v_j \prec \overline{v_j}$, with $1 \leqslant i, j \leqslant n$ and $v_i, \overline{v_i}, v_j, \overline{v_j} \in L$. As it is needed at some points in the construction, this order can be extended by the start and finish points, where $start \prec v_1$ and $\overline{v_n} \prec finish$, leading to the overall order: $start \prec v_1 \prec \overline{v_1} \prec \cdots \prec v_n \prec \overline{v_n} \prec finish$.

Definition 3.2 (Primary/secondary path)**.** If two (literal) paths cross each other, those will be called the primary and secondary path: The $\prec$-smaller path is primary, while the $\prec$-larger path is secondary.

Definition 3.3 (Two-dimensional grid)**.** All elements used in the construction of the framework and the mentioned games need to be assigned a location. These locations correspond to coordinates on a coordinate system/grid, which is kept two-dimensional. This means, a location (x, y) defines the x- and y-components within that grid.

3.2.1 Structure of the gadgets

Each gadget provides some kind of functionality. The player has to enter the gadget, do something and leave it again. In terms of graph theory the player traverses the gadget through one or multiple nodes. Therefore, all gadgets will be represented by a set of (entry and exit) nodes in the construction of the framework. When needed at a later point in the construction, a node may be referenced via the gadget that contains the node, given the name of the (literal or check) path which traverses the node. In particular: When the paths are going to be created, the locations of the nodes to be connected with each other have to be known. As each node has a fixed location within (the local coordinate system of) its gadget, the location of a node can be calculated by the location

of the containing gadget plus the node's relative offset to the gadget location. The following describes the structure of each gadget, where *begin* denotes the node where a subpath begins and *end* denotes the end of a subpath. So *begin* denotes outgoing edges, while *end* denotes ingoing edges.

Definition 3.4 (Gadget structure)**.**

Start gadget As this is one of the endpoints of the framework, it consists of only one node, labeled $(start)_{begin}$.

Variable gadget This contains two entry and two exit nodes. Considering variable v_i, the entries are labeled based on the $\prec$-order predecessor of v_i, let it be u. The first entry is labeled $(u)_{end}$, the second is labeled $(\overline{u})_{end}$. If $i = 1$ then $u = start$, otherwise $u = v_{i-1}$. The first exit is labeled $(v_i)_{begin}$, while the second is labeled $(\overline{v_i})_{begin}$.

Clause gadget Clauses can be traversed via both literal and check paths. The nodes of the literal paths are labeled with the name of the corresponding literal. There are two nodes for literal l, labeled $(l)_{end}$ and $(l)_{begin}$. The check path consists of one entry and one exit node, labeled $(check)_{end}$ and $(check)_{begin}$. There is an internal edge between these nodes.

Crossover gadget Each crossover gadget consists of two entry and two exit nodes. Given two crossing literal paths s and t, with order $s \prec t$. The primary entry of the gadget is labeled based on literal $(s)_{end}$, while the secondary is labeled $(t)_{end}$. The corresponding exits are labeled with $(s)_{begin}$ and $(t)_{begin}$. Internally, $(s)_{end}$ and $(s)_{begin}$ as well as $(t)_{end}$ and $(t)_{begin}$ are directly connected.

Jump-down gadget The jump-down gadget consists of two entries and one exit. The entry nodes are labeled based on the last variable: $(v_n)_{end}$ and $(\overline{v_n})_{end}$. The check path starts in this gadget, so the exit is labeled $(check)_{begin}$.

Finish gadget This is the second endpoint of the framework with only one node. Its predecessor belongs to the check path, so the node is labeled $(check)_{end}$.

Subsection 3.2.4 shows why the nodes are labeled this way.

3.2.2 Arranging the gadgets

As already mentioned, gadgets are positioned on a two-dimensional grid. This defines a coordinate system that grows in top-right direction. If not stated different, each gadget is considered to be of width and height 1. The sizes of the gadgets in this construction are idealized. To get the gadgets implemented in a game, larger and non-uniform sizes are required. This will just stretch the graph without changing the structure or introducing any problems. The gadgets are then arranged in the following way:

Definition 3.5 (Gadget location)**.**

Start gadget This is placed in the top left, above the first variable gadget. So the start gadget is placed on position $(0, 2 \cdot n + 4)$ where $n = |V|$. The value of the vertical position is explained in the following variable gadget definition.

Variable gadgets Given n variables, those are indexed from v_0 to v_{n-1}. All variables are placed on the top but below the start gadget. Each variable needs space for two literal paths. So the position of the i^{th} variable $(0 \leqslant i < n)$ is $(2 \cdot i, 2 \cdot n + 2)$. The vertical position comes from the needed space for the literal paths plus an extra two units of space needed to keep distance at the top and bottom of the literal paths.

Clause gadgets Given m clauses, those are indexed from C_0 to C_{m-1}. All clauses are placed on the bottom row right of the last variable, starting at horizontal position $2 \cdot n$. Each clause has to provide space for each variable to enter and leave the clause, which leads to a width of $4 \cdot n$ for each clause. The vertical position of the clauses is 0. So the position of the j^{th} clause $(0 \leqslant j < m)$ is $(2 \cdot n + j \cdot 4 \cdot n, 0)$.

Crossover gadgets The positions of the crossover gadgets will be assigned during their creation in the next step of the construction (see Subsection 3.2.3).

Jump-down gadget This is placed far right in the bottom row next to the last clause gadget. So its position is $(2 \cdot n + m \cdot 4 \cdot n, 0)$. The x-component results from the width of the variable gadgets plus the width of the clause gadgets.

Finish gadget This is placed in the bottom left next to the first clause gadget, so the position is $(0, 0)$.

Example 3.6. Figure 3.2 shows how the framework looks like with this new layout for the example formula $(x \vee \overline{y} \vee z) \wedge (x \vee y \vee z \vee \overline{z})$. To save space, the width of the clauses has been reduced in this figure. Also, crossover gadgets are not explicitly shown, although indicated by crossing edges.

Arranging the gadgets in the described way allows crossovers to only appear between variables, clauses and the jump-down gadget. As the edge from the start to the first variable gadget lies in a region in which by construction no other edge can cross that edge, there is no need for additional crossover gadgets. The same applies to the edges connecting the check paths of the clause gadgets and the finish: In the bottom region no other edge can cross the check path. This reduces the number of crossover gadgets in relation to the original framework and simplifies the implementation as an actual game as the locations of the crossover gadgets are well defined. Further, this simplifies the process of setting up the paths between the gadgets.

3.2.3 Defining crossover gadgets

In order to construct a fully defined graph, additional nodes representing the crossover gadgets are needed.

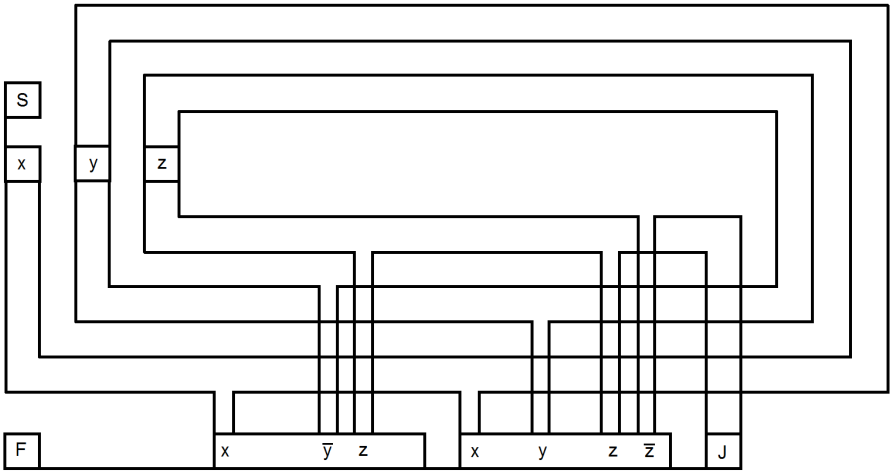

Figure 3.2: New layout of the framework

Definition 3.7 (Crossover gadget)**.** A crossover gadget has to fulfill some requirements:

1. There are two internal paths c_1, c_2 that lead through the gadget and resolve the crossing point. They represent the primary and secondary path. The game mechanics has to guarantee that each crossover gadget is traversed at most twice: Both the primary and secondary path each at most once. The secondary path has to provide one-way functionality at its exit. This avoids revisiting the crossing section at a later point.

2. There is at most one path $c \in \{c_1, c_2\}$, such that traversing c manipulates the gadget in such a way, that leakage from $d \in \{c_1, c_2\}, d \neq c$ to c gets possible. Then d has to represent the primary path, while c does for the secondary path. This means, d as the primary path has to correspond to the $\prec$-smaller path that traverses the crossover, while the secondary path corresponds to the $\prec$-larger path.

3. There is a traversal order: The secondary path may not be traversed before the primary path.

Although the definition of the crossovers has to be precise, it turns out that it does not have to be too restrictive. Note that Requirement 2 allows a special case of leakage. With this, the framework can be applied to different game mechanics and therefore to more games. The secondary paths of crossover gadgets implement one-way functionality. This means that the secondary path can only be traversed in one predefined direction (by Requirement 1). As the crossover gadget may not get manipulated when traversing the primary path, backtracking and revisiting the crossing section via the primary path cannot cause any problems. So the primary path does not have to provide one-way functionality. The essence of this crossover definition has been stated by Remark 2.1 and Remark 2.2 in [3] and [4]. There is no need to re-traverse the gadget

backwards which allows to exploit gravity in many games. In fact, the gadget must disallow the player to revisit the actual crossing section via the secondary path. Therefore the one-way functionality needs to be implemented at the exit of the crossover gadget. Figure 3.15 shows an example of this for Super Mario: The secondary (vertical) path provides one-way functionality, such that Mario has to fall down a passage and cannot revisit the crossing section after traversing the secondary path. In contrast, Figure 3.13 (taken from [3]) does not provide this and introduces problems.

Not having this one-way functionality, leakage could get possible. As a special case of leakage is allowed (by Requirement 2), the term leakage has to be classified:

Definition 3.8 (Leakage classification)**.** The leakage direction defines, whether leakage allows the player to jump back or forward in the graph. This leads to different situations, producing wrong results in terms of a game state that does not represent the chosen variable assignment.

Forward leakage This allows the player to take a shortcut and skip parts of the graph. This could cause tautology formulas to be unsatisfied. Example 3.9 shows a case where forward leakage leads to an unsolvable game although the assignment would satisfy the formula.

Backward leakage This allows the player to repeat parts of the graph and make different decisions for the same variable. This could cause contradictions to be satisfied via illegal variable assignments. Example 3.10 demonstrates this.

The leakage moment defines at which state of a crossover traversal leakage can occur the first time. In particular, this implies how often a crossover gadget can be traversed before leakage gets possible.

First traversal The player could leak from the primary to the secondary path (forward leakage) and from the secondary to the primary path (backward leakage). If leakage is possible during the first traversal, the design of the gadget would have failed completely as nothing is restricted.

Second traversal The player traverses the first time safely. The gadget gets manipulated during the first or second traversal, such that leakage is possible during the second traversal. As the traversal order is determined by the $\prec$-order over the literals, the first traversal will always be via the primary path. So leakage is only possible from the secondary to the primary path which by definition leads to backward leakage.

Third traversal The player traverses the primary and secondary paths safely. After the second traversal the gadget gets manipulated, such that leakage would be possible during a third traversal. As the graph is directed and acyclic (with Requirement 1) the player can reach a crossover gadget no more than the intended two times. Therefore, third-traversal leakage does not introduce any problems and may be allowed.

Example 3.9. Consider the tautology $(x \vee \overline{x}) \wedge (y \vee \overline{y})$. As shown in Figure 3.3, the player can follow the literal path for $\overline{x}$ to the highlighted crossover and leak to the path for literal $\overline{y}$ towards the jump-down gadget. The player enters the check path and gets stuck as the second clause has not been satisfied. The paths in this example are arranged such that forward leakage occurs. The path creation in this figure does not follow the method that is going to be introduced, which allows this kind of leakage to occur. This could be easily avoided by a different arrangement, though this is the purpose of the example.

Example 3.10. Consider the contradiction $(x \vee y) \wedge (x \vee \overline{y}) \wedge (\overline{x} \vee y) \wedge (\overline{x} \vee \overline{y})$. As shown in Figure 3.4, the player can follow the blue literal path for x, satisfy the third and fourth clause and continue to variable y. The player follows the green literal path for y to satisfy the first clause and continues to the highlighted crossover with the blue path from where the player can leak backwards in the graph to variable y, from where the literal path $\overline{y}$ is chosen to satisfy the second clause as well and successfully traverse the check path, satisfying a contradiction with an illegal assignment. Note that the paths in this figure are not constructed in the way that is going to be explained in this thesis. This has been chosen for simplicity reasons since the only purpose of this figure is to demonstrate leakage.

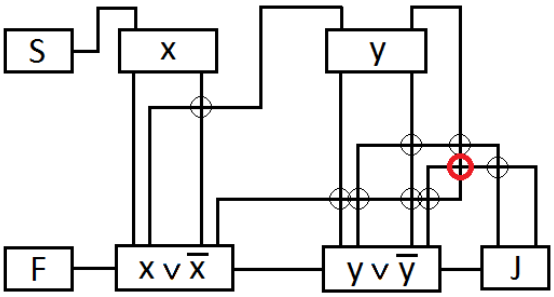

Figure 3.3: Forward leakage

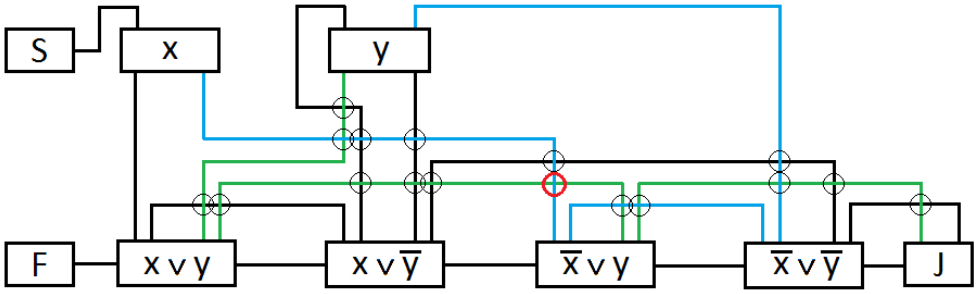

Figure 3.4: Backward leakage

From this classification the following can be deduced: If only one path c through a gadget leads to leakage after it has been traversed, then c has to be assigned the secondary path while the safe path d gets the primary path. This leads to third-traversal leakage, which is fine. If there were leakage from both paths c and d to each other, this would lead to second-traversal leakage. Requirement 3 allows to cross both literal paths of one variable with each other. In this case only one of both paths of the crossover will be traversed. If only the primary path is visited, which would be the default case, there will not be any leakage possibilities by definition. If in the general case the secondary path is traversed first, this means that the secondary path is visited before the primary path, which would introduce leakage possibilities when traversing the primary path as the secondary path got manipulated before. Requirement 3 implies that if both literal paths of the same variable cross each other, the secondary path may be the only traversed path, as only one of both literal paths can be traversed. So the manipulated gadget will not be visited again and leakage is not possible.

By providing each gadget a two-dimensional index that represents its physical location, gadgets can be arranged such that the number of crossing paths is reduced. Therefore, all possible crossover gadgets appear between variable and clause gadgets as well as the jump-down gadget. Those crossover gadgets are defined in the following way:

Definition 3.11 (Set of crossover gadgets)**.** Given the set of variables V, the set of clauses $\mathcal{C}$ and the set of literals L with the order over the literals $v_1 \prec \overline{v_1} \prec \cdots \prec v_n \prec \overline{v_n}$, the set of crossover gadgets X can be defined as follows: For each literal $l_i \in L$ with $1 \leqslant i < |L| = 2 \cdot n$, $\mathcal{C}_{l_i}$ defines the set of clauses that contain l_i. The literal locations within these clauses define the horizontal locations of crossover gadgets, needed to resolve crossing paths as literal path l_i crosses all previous literal paths l_j with $0 \leqslant j < i$, when turning into its vertical section to visit a clause. For each clause $C \in \mathcal{C}_{l_i}$, the horizontal location (relative to the origin of the coordinate system) of the node labeled l_i within C is stored in x_{C,l_i}. For each literal l_j preceding l_i with $0 \leqslant j < i$, two new crossover gadgets are created, located at (x_{C,l_i}, y_j) and $(x_{C,l_i} + 1, y_j)$, where y_j denotes the vertical height on which literal path j runs.

Explanation of Definition 3.11 The literal paths i that will introduce crossovers start at index 1, ignoring the first literal l_0 as the first literal path can be created without crossing any other path. The order of clauses in $\mathcal{C}$ may be arbitrary, but fixed. This assigns each clause C an index/location k in the order of $\mathcal{C}$ with $0 \leqslant k < |\mathcal{C}|$. So the horizontal location of a crossover $x_{C,l_i} = 2 \cdot n + k \cdot 4 \cdot n + 2 \cdot i$ is composed as follows: $2 \cdot n$ refers to the width of n variable gadgets (defined by the size of two literal paths), $k \cdot 4 \cdot n$ comes from the number of clauses to be skipped in order to reach clause C, $2 \cdot i$ skips the ingoing and outgoing vertical parts of the i previous literal paths. The vertical location on which the literal path for l_j will run is defined by $y_j = j + 1$. This value calculates from j literals to skip plus 1 to make the first literal path run above the gadgets at the bot-

tom. So for each literal path that gets crossed by the current literal path, two crossover gadgets need to be placed above the literal l_i within a clause gadget: One for the path coming down to the gadget, one for the path going back up.

The particular implementation of a crossover gadget has to define which way through it to be the primary and the secondary path. It also needs to define the entrance and exit points of both ways. This leads to a fixed layout of the crossover gadgets. Still, as all gadgets may be mirrored and the paths may be wound around them, any intended traversal order can be achieved. See Figure 3.5 for an example.

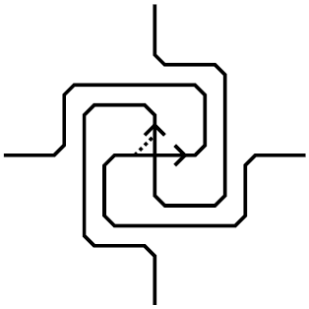

Figure 3.5: Rotation for Crossover gadgets

3.2.4 Setting up the paths

As the gadgets have been located and each gadget offers a set of nodes, the graph can be completed by setting up paths to connect those nodes. The crossover gadgets can be referenced via their location. The paths are constructed based on the $\prec$-order of the literals including start and finish: $start \prec x_1 \prec \overline{x_1} \prec \cdots \prec x_n \prec \overline{x_n} \prec finish$. Let V be the set of variable gadgets, $\mathcal{C}$ the set of clause gadgets, X the set of crossover gadgets and S the start gadget. The construction needs a list of gadgets from which literal paths can start: $subpathGadgets = \{v_1, v_1, v_2, v_2, \ldots, v_n, v_n\}$. From each variable gadget v_i, two literal paths labeled v_i and $\overline{v_i}$ are starting. Therefore, each variable appears twice in the list. Next, the list of node labels is needed that describes which nodes of the gadgets have to be connected in the current subpath, given a specific label: $subpathLabels = \{v_1, \overline{v_1}, v_2, \overline{v_2}, \ldots, v_n, \overline{v_n}\}$. The elements in both lists correspond to each other: The first label in $subpathLabels$ describes the literal path that starts from the first gadget in $subpathGadgets$. Further, the start path goes from the start to the first variable gadget and the check path starts at the jump-down gadget and goes through all clauses towards the finish gadget. These two paths are treated separately. The paths are then set up as described by the following algorithm:

First, the start path is created. This is done by simply adding an edge from the $(start)_{begin}$ node of the start gadget to the $(start)_{end}$ node of the first variable gadget. Then the literal paths are set up: To do this, the index of the currently treated literal is denoted by i, which is initialized with 0.

```
function createPaths()
    for i ← 0 to subpathGadgets.size() do
        node ← subpathGadgets.get(i)
        l ← subpathLabels.get(i)
        xLocs ← horizontalLocationsOfLiterals(C ∪ {jumpDown}, l_end)
        sort(xLocs)
        y ← i + 1
        curX ← i
        curY ← 2 · n + 2
        while not xLocs.isEmpty() do
            x ← xLocs.pop()
            connectCrossovers((curX, curY), (curX, y), (x, y), (x, 0), l)
            curX ← x + 1
            curY ← 0
        end while
        //not currently at jump-down gadget; create closing back path
        if i ⩽ subpathGadgets.size() − 2 then
            maxX ← 2 · n + m · 4 · n + 2 · n − i − 2 + 1
            maxY ← 4 · n − i
            connectCrossovers((curX, curY), (curX, y), (maxX, y),
              (maxX, maxY), (i + 2, maxY), (i + 2, 2 · n + 2), l))
        end if
    end for
end function
function connectCrossovers(list(locs), label)
    curLoc ← locs[0]
    for i ← 1 . . . locs.size − 1 do
        destLoc ← locs[i]
        for all x ∈ X do
            //crossover x lies between start and end locations therefore
            //by construction needs to have a node called label_end
            if inInterval(x.posX, curLoc.posX, destLoc.posX) and
              inInterval(x.posY, curLoc.posY, destLoc.posY) then
                Edge(curLoc, x.getLocation(label_end))
                curLoc ← x.getLocation(label_begin)
            end if
        end for
        Edge(curLoc, destLoc)
        curLoc ← destLoc
    end for
end function
```

Figure 3.6: Path creation algorithm (NP-framework)

Recall that the value of n denotes the number of variables, while m denotes the number of clauses. The current location is accessible via $curX$ and $curY$. The i^{th} elements are taken from the lists, namely $node \in subpathGadgets$ and $l \in subpathLabels$. The list of horizontal locations on which the label l_{end} appears in gadgets in $\mathcal{C} \cup \{jumpDown\}$ is denoted by the list $xLocs$, in which the elements are sorted in ascending order. Each literal path starts with a vertical edge from the node labeled l_{end} in the variable gadget $node$ (location $(i, 2 \cdot n + 2)$) down to (i, y) with $y = i + 1$. Then, the first element from $xLocs$ namely x is taken (and removed) from the list and a horizontal path goes to (x, y). That path connects all crossover gadgets that fulfill the location constraint $([curX, x], y)$. This means, that the horizontal location has to lie in the interval $[curX, x]$ and the vertical location is y. Then, a vertical path goes down to $(x, 0)$, connecting all crossovers that fulfill the constraint $(x, [y, 0])$. If the jump-down gadget is currently visited, one of the last two literal paths has been finished. If it was the last literal, all literal paths have been set up. If it was the second last literal, the whole procedure starts anew for the last literal path with $node$ and l. As the last literal path has been finished (the lists are empty), the creation of the check path can conclude the construction of the framework. In the case, that the currently visited gadget is a clause, the procedure continues in the following way. As the clause gadget provides an internal edge to $(x + 1, 0)$, the path creation continues from this location. A path up to $(x + 1, y)$ is created, again considering all crossovers in that region. Then, the procedure starts again with the next element in $xLocs$ until this list is empty. As no further clause contains the label l_{end}, the path has to lead to the next variable gadget. To do so, the path first has to go right to $(maxX, y)$ with $maxX = 2 \cdot n + m \cdot 4 \cdot n + 2 \cdot n - i - 2 + 1$, considering all crossovers fulfilling the location constraint $([curX, maxX], y)$. Then the path goes up to $(maxX, 4 \cdot n - i)$, continuing left to $(i + 2, 4 \cdot n - i)$ and finally down to the next variable gadget $(i + 2, 2 \cdot n + 2)$. For the last three edges no crossover gadgets have to be considered. Now, the current literal path is completed and the overall procedure can start again for the next literal.

Finally, the check path has to be set up. It starts from the $(check)_{begin}$ node of the jump-down gadget and connects to the $(check)_{end}$ node of the last clause c_m. The clause internally provides an edge to its $(check)_{begin}$ node. The check path continues to the previous clause gadgets until the first clause c_1 is reached. From here, the final edge leads to the $(check)_{end}$ node of the finish gadget, which completes the graph.

All crossovers that are encountered on a horizontal path are traversed via their primary path, while crossovers connected on a vertical path are traversed via the secondary path. The nodes within a gadget and their locations on the coordinate system can be used equivalently. This means, that given a gadget and the name of a contained node, this node's location on the coordinate system can be deduced. During the previous construction the more convenient alternative has been used.

3.2.5 Reduction

This subsection gives insight in the reduction from SAT on which the whole construction is based. This contains correctness and complexity proofs.

Lemma 3.12. *Let the described construction be called NP-FRAMEWORK problem. There is a function R with the following property:*

$$\forall \phi : \phi \in SAT \Leftrightarrow R(\phi) \in \text{NP-FRAMEWORK}$$

Therefore R is a reduction from SAT to NP-FRAMEWORK.

Proof. It is important that the framework behaves correctly at any time, in particular, it may not allow the player to exploit leakage to break the game. So the player may only traverse those paths that are intended to be traversed at any point throughout the game. Note: A solvable game g (finish gadget can be reached) states that $g \in$ NP-FRAMEWORK. The proof is split in two parts:

($\Rightarrow$): If the formula is in SAT, then the framework generates a game that can be solved: $\forall \phi : \phi \in \text{SAT} \Rightarrow R(\phi) \in \text{NP-FRAMEWORK}$

Membership in SAT means that there is a variable assignment ϕ that satisfies the formula. When the player reaches a variable gadget v, a literal path is chosen based on the variable assignment of v in ϕ. On these literal paths, the player visits clause gadgets (via entrances that represent literals in those clause gadgets) and performs actions to unlock the check path within the gadgets. After the last variable gadget has been traversed, the assignment is done and needs to be verified, by traversing the check path. The check path visits each clause gadget and can only pass if each clause gadget has been visited and unlocked before. As the player followed literal paths based on ϕ which satisfies all clauses, each clause gadget can be passed and the player reaches the finish gadget.

($\Leftarrow$): If the framework generates a solvable game, then the represented formula is in SAT: $\forall \phi : \phi \in \text{SAT} \Leftarrow R(\phi) \in \text{NP-FRAMEWORK}$

A solvable game means that the player could reach the finish gadget after (successfully) traversing the check path. Based on the state of the game at this point, the chosen assignment can be read off. In terms of the graph, representing the game, assignment means that the player makes decisions which of two paths to follow and unlocking the check path of clause gadgets lying on that path. The player has to choose the (positive or negative literal) paths within the variable gadgets such that all clause gadgets can be passed on the check path. As the game is solvable, there is such a variable assignment. The state of each literal-path-entrance in the clause gadgets represents one of its literals. If that state has been changed (by performing an action while visiting it), the literal is true, if the state remains unchanged, it is false. In the case of Super Mario, kicking Koopa's shell changes the state. Based on the states of those literal-path-entrances, the variable assignment can be deduced. Note that the unchanged entrances are not of relevance here. As the finish gadget is reached, the assignment ϕ is read off. This assignment might not include all variables (as the unchanged entrances are ignored). In that case the missing variables are generically set to false and added to ϕ in order to complete the assignment.

Finally, ϕ is verified by feeding it to the formula. As each clause gets satisfied, the formula is in SAT. □

Complexity This paragraph shows the complexity of the described construction of the framework.

Lemma 3.13. *The function R denoting the reduction SAT $\leq_{log}$ NP-FRAMEWORK is logspace computable.*

Proof. The NP-FRAMEWORK problem consists of multiple parts. In this proof we focus only on the interesting parts of the computation. The generation of the variable and clause gadgets is trivial and ignored here. Also the generation of the edges between the gadgets is considered straight-forward. As the construction of the crossover gadgets is not trivial, a complexity analysis is needed. It needs to be checked, whether the computation can be performed within logarithmic space. Given a Turing machine with I/O with an encoding of all clauses and literals on its input tape, where the literals need to be in $\prec$-order, the work tape looks like that:

$$i\ j\ k\ x\ y,$$

with its elements stating the following:

- i: This is a binary encoding of the currently checked literal path l_i.
- j: This is a binary encoding of the second literal path l_j that will cross l_i with $j < i$.
- k: This is a binary encoding of the currently checked clause C_k, whether it contains literal l_i.
- x: This is a binary encoding of the horizontal location of l_i, contained in clause C_k.
- y: This is a binary encoding of the vertical location on which the literal path for l_j is running.

With $2 \cdot n$ literals and m clauses the work tape needs $2 \cdot \log(2 \cdot n) + \log(m) + \log((m+1) \cdot 2 \cdot n + 1) + \log(2 \cdot n + 2)$ bits of space, which is obviously logarithmic in terms of the input size.

Based on that structure of the work tape, the calculation looks like this (all indices starting from 0): For each literal l_i with $1 \leqslant i < |L| = 2 \cdot n$ and $n = |V|$, each clause C_k with $0 \leqslant k < |\mathcal{C}| = m$ is checked, whether it contains l_i. If so, the horizontal location of the node labeled l_i within C_k is calculated and stored in x. The calculation is done as $x = 2 \cdot n + k \cdot 2 \cdot n + 2 \cdot i$. For each literal l_j with $0 \leqslant j < i$ two new crossover gadgets are created with locations (x, y) and $(x + 1, y)$ with $y = j + 1$. □

3.3 Application to Super Mario World

In this section, the described gadgets applied to the game mechanics of Super Mario World will be shown. Most of the gadgets are based on the introduced gadgets of Aloupis et al. in [4]. Figure 3.7 shows the start gadget. Mario jumps down and consumes the mushroom that is hidden behind the left question-mark block and leaves the gadget via one of the exits to the bottom (which does not matter). Figure 3.8 shows a variable gadget with two entrances at the top and two exits at the bottom. Mario can only leave the gadget via one of the exits. The left one leads to the positive literal path, the right one to the negative literal path. Leakage via the other entrance is not possible. Figure 3.9 shows an instance of the crossover gadget. Mario traverses the primary path after getting hit by the Koopa on the left which turns him back into small Mario. At the right, Mario transforms back into Super Mario, using the mushroom block. Super Mario can smash the brick on the right and leave the gadget. The secondary path is traversed from bottom to top, smashing the bricks in the crossing section, jumping down a vertical tunnel (which provides one-way functionality) and leaving via the top exit. Leakage from the primary to the secondary path is not possible as the secondary path has not yet been traversed, which would have smashed the bricks, and small Mario is not able to smash bricks. Leakage from the secondary to the primary path is not possible as Mario is big which prevents him from passing the narrow parts on the left and right of the crossing section. Mario cannot backtrack from another gadget via the secondary path from top to the crossing section as the gadget provides one-way functionality towards the top exits. In Figure 3.10, a clause gadget is shown. Mario enters via one of the literal entrances at the top and kicks Koopa's shell down which will smash the bricks at the bottom and unlock/satisfy the clause. Leakage to the bottom is not possible, as Mario needs to be big at a later point in the game. Leakage to another literal entrance is not possible as well. In Figure 3.11, the jump-down gadget is displayed. It joins both literal paths coming from the last variable gadget. The positive literal path enters at the top left, the negative literal path enters at the top right. The single bottom exit connects to the check path. Leakage to the other entrance is not possible. Figure 3.12 shows the finish gadget. Mario needs to be big in order to smash the bricks and reach the finish location, represented by the flag. If small Mario had leaked to the check path in a clause gadget, he would not be able to reach the flag.

Figure 3.7: Start gadget

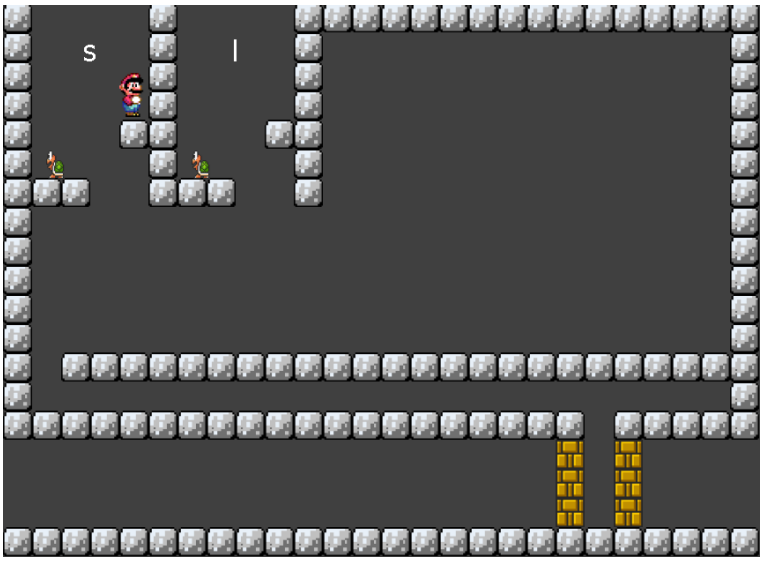

Figure 3.10: Clause gadget

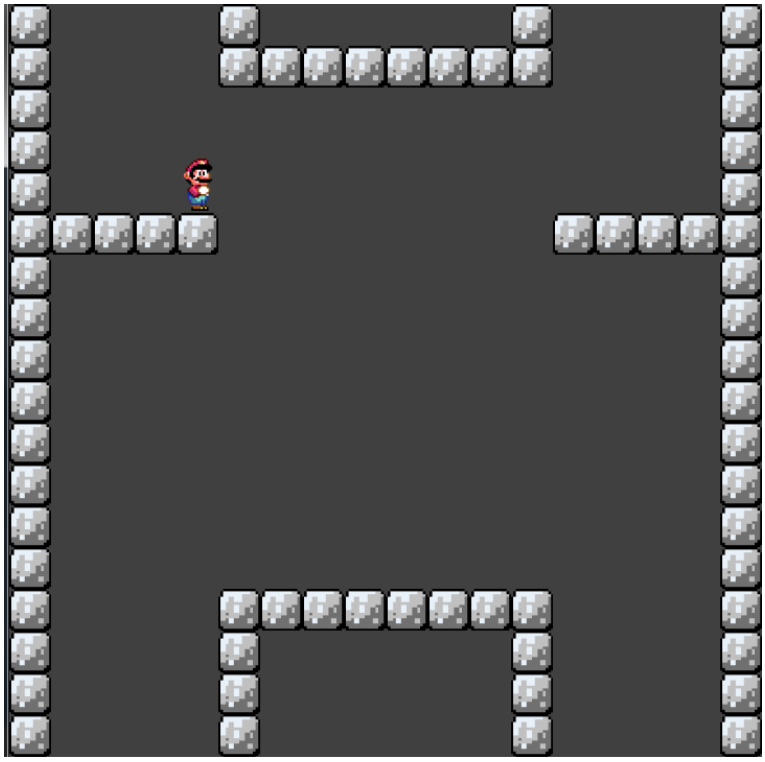

Figure 3.8: Variable gadget

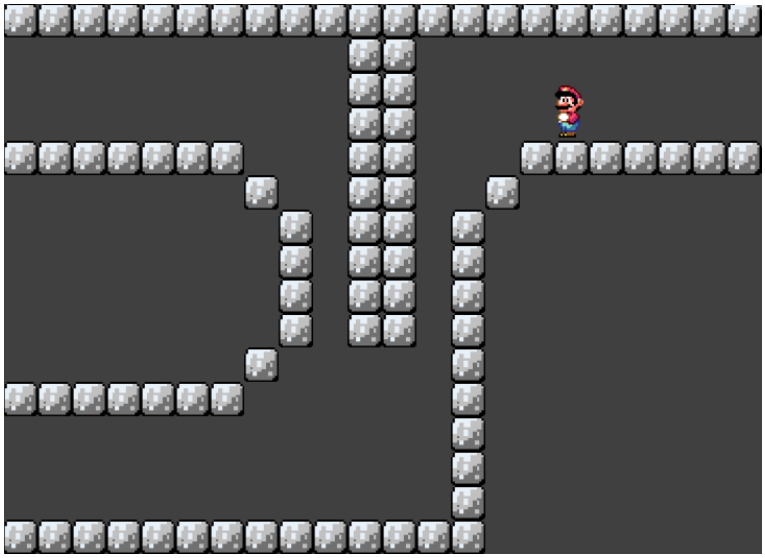

Figure 3.11: Jump-down gadget

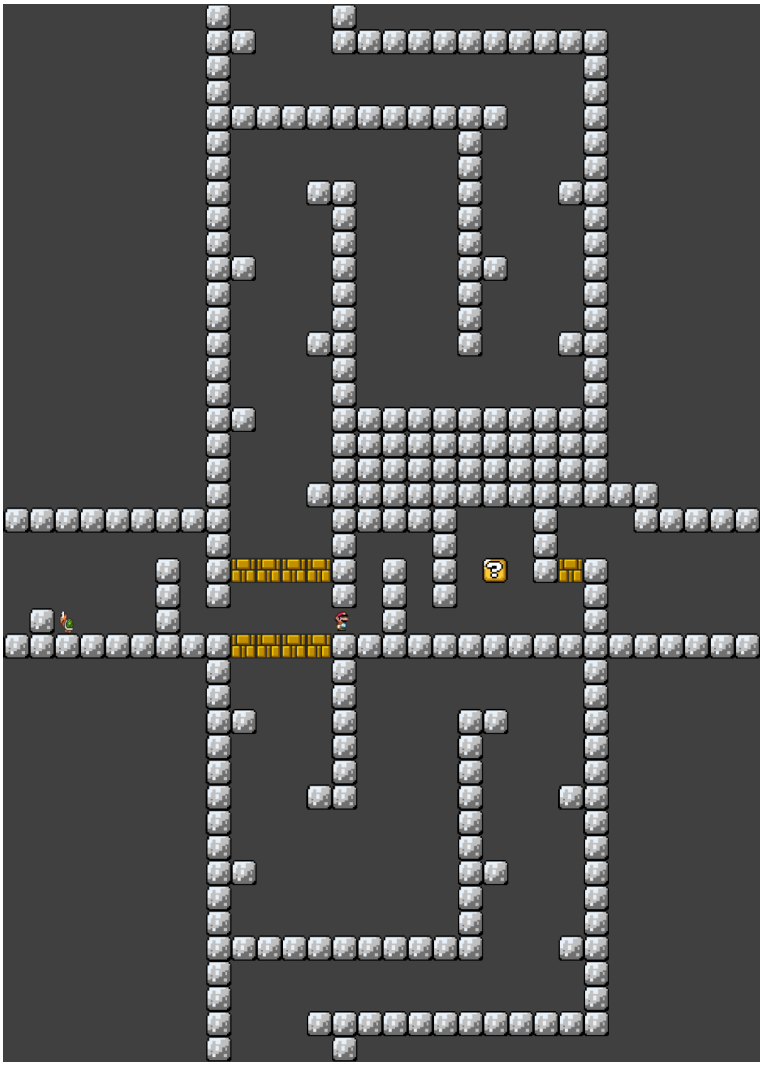

Figure 3.9: Crossover gadget

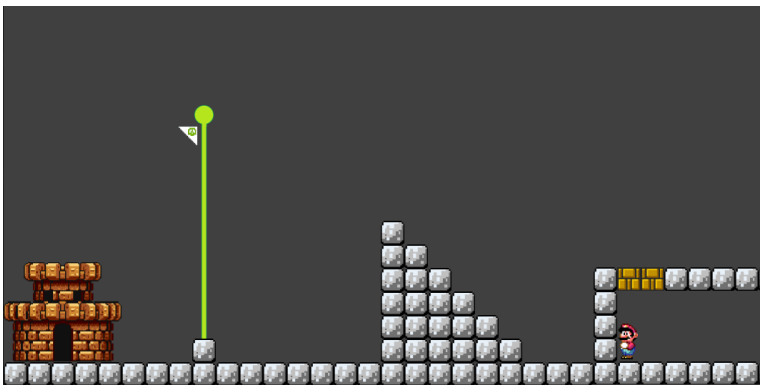

Figure 3.12: Finish gadget

3.4 Flaws in Particular Games

In [2] the described framework has been applied to several games. Unfortunately, the implementation of the crossover gadget for some of that games allows backward leakage as those implementations do not conform Definition 3.7. Problems and possible solutions shall be discussed in this section.

3.4.1 Super Mario

Considering the layout of the crossover gadget (see Figure 3.13) for Super Mario games in [2,3,4], a subtle flaw can be observed: The introduced crossover gadget does not provide one-way functionality at the exit of the secondary path which allows the following: Super Mario first traverses the primary and secondary path regularly. As the secondary path has been left via the top exit, Super Mario directly reaches another crossover gadget in its primary path, such that Mario gets hurt by the Goomba and turns into small Mario to go back and visit the previous gadget via its secondary path from the top. Small Mario can then leak through the primary path. At a later point Mario needs to turn into Super Mario again in order to smash the brick in the finish gadget. Therefore, another crossover gadget is needed that is traversed through its primary path after this leakage had occured. Figure 3.14 shows an instantiation of this (with simplified construction). As Mario follows the green path and reaches the upper highlighted crossover, he gets hurt and goes back to the lower highlighted crossover as small Mario. From here he leaks back to the variable gadget for y via the blue path and follows the literal path for $\overline{y}$. The first crossover on this path is traversed via the primary path and Mario can turn into Super Mario again.

This problem can easily be solved by changing the layout of the crossover gadget for Super Mario. Figure 3.15 shows a safe version of the crossover gadget.

3.4.2 Metroid

Considering the layout of the crossover gadget (see Figure 3.16) for the Metoid games in [2], a subtle flaw can be observed: As Samus traverses from below, she shoots both regenerating blocks in vertical direction and jumps up on a solid block on the side. There she waits until the destroyed blocks have been regenerated and opens the upper block again. Then she can jump down on the lower block, turn into Morph-ball and leak towards the exit of the horizontal path. This is second-traversal/backward leakage, which does not conform with Definition 3.7. Figure 3.17 shows a safe version of the crossover gadget without leakage possibilities. As Samus needs to traverse horizontally, she jumps in the hole on the side and shoots the regenerating blocks in the horizontal way. Note that the game mechanics do not allow Samus to aim that exaclty to shoot one of the blocks that belong to the vertical way. In particular, Samus can only shoot while standing and shooting is only possible in a few fixed directions. Standing Samus is two blocks tall and her weapon is located at the upper block. When

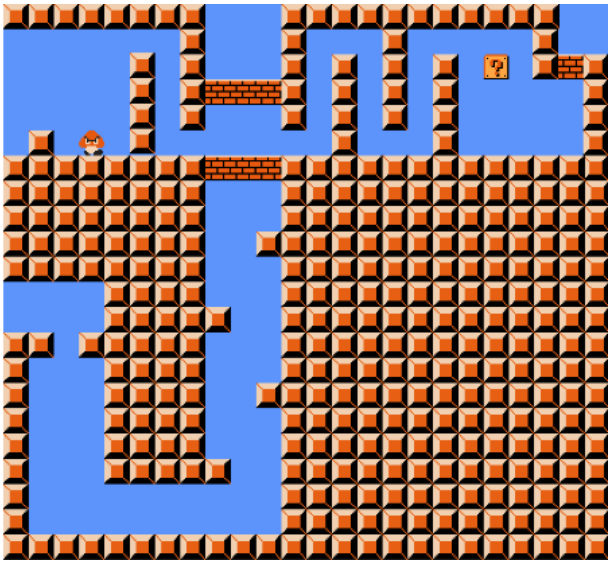

Figure 3.13: Problematic crossover gadget for Super Mario [4]

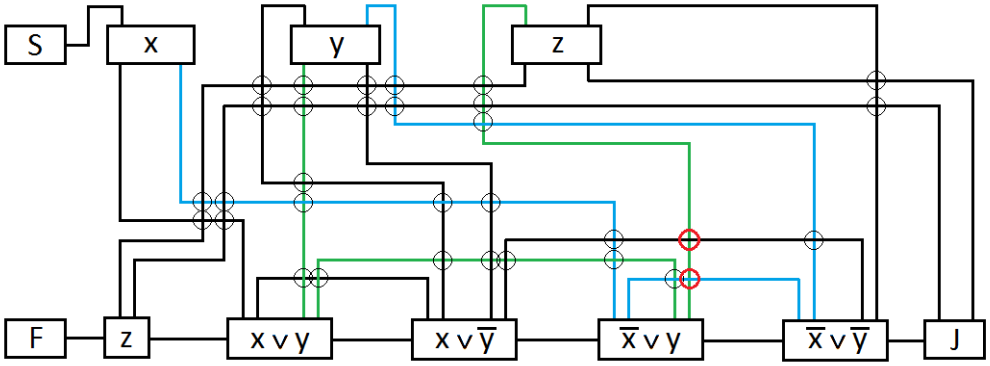

Figure 3.14: Problematic instance for Super Mario crossover

Samus is in Morph-ball mode, she only is one block tall but cannot shoot. If precise aiming was possible, new leakage would have been introduced as she could shoot the blocks belonging to the vertical path while traversing the horizontal path and leave the gadget through the bottom exit. When Samus traverses vertically, she approaches from the top, shoots both blocks in vertical direction and falls down through the gadget. If she only shoots the upper block, she cannot escape horizontally, as there are blocks that cannot be shot from her location. Note that if the gadget would have been designed to traverse from bottom to top, similar to the original version of the gadget, Samus could shoot the lower block, wait some time, then shoot the upper block and shoot the horizontal block as well while jumping, then wait again to jump in exactly that moment such that the lower block regenerates while Samus has jumped above it. Then she could turn into Morph-ball and exit horizontally before the block

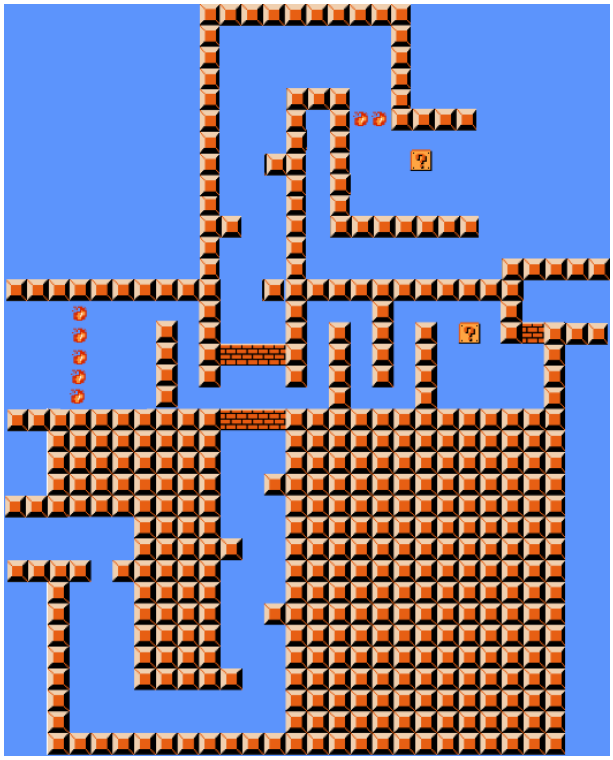

Figure 3.15: Safe crossover gadget for Super Mario

on the horizontal way would have been regenerated.

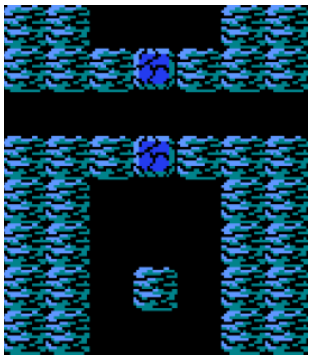

Figure 3.16: Crossover gadget for Metroid [2]

In their later versions [3, 4], Aloupis et al. introduced a safe crossover gadget for Metroid (see Figure 3.18). Due to the layout of this gadget, the enemies cannot be shot, so Samus has to avoid collision with them. The enemies are moving clockwise on the top left and counter-clockwise on the top right section. Samus has to wait for the gap between the enemies and use it to follow the path towards the middle of the gadget from where she can fall down and leave the gadget via the exit that is opposite to the entry. The enemies on the bottom are timed such that Samus can leave through the intended exit but cannot leak

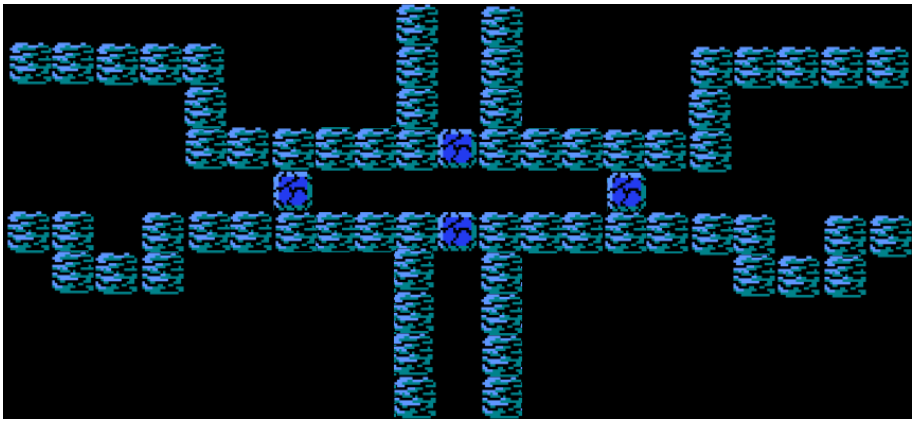

Figure 3.17: Safe crossover gadget for Metroid

through the other exit.

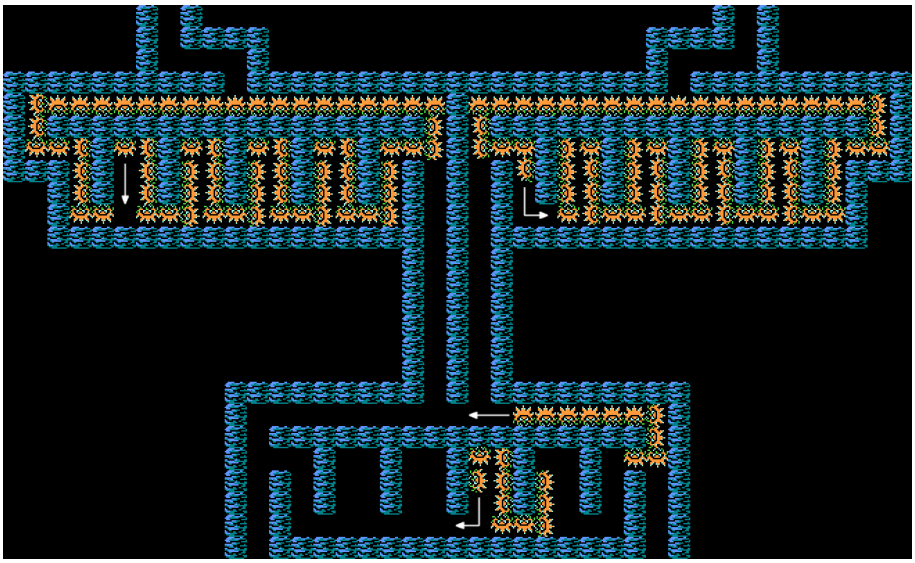

Figure 3.18: Safe Crossover gadget for Metroid from [3]

3.4.3 NP-completeness

In the very first version of their article [1], Super Mario Bros. and Metroid have been claimed to be NP-complete. In the later versions these claims have been removed as they are not true. A certificate describing the process of solving a game world has to be polynomially-bounded in length in order to be verified in polynomial time. As there are enemies in these games that move and change their positions, the player has to be able to react on that, which lets the certificate grow exponentially. Thus, those games cannot be in NP.

4 PSPACE-Framework

4.1 Introduction

Similarly to the NP-framework of Chapter 3, Aloupis et al. introduced a framework for PSPACE-hardness in [2,3,4], which reduces from the PSPACE-complete problem QSAT. In [4], instantiations of this framework for Donkey Kong Country and The Legend of Zelda have been shown. This thesis adds an instantiation for Super Mario World, which is therefore shown to be PSPACE-hard. Further, membership in PSPACE is shown, thus Super Mario World is PSPACE-complete. Again, these proofs are similar to the proofs for the metatheorems of Viglietta in [11].

The gadgets used in the PSPACE-framework have to fulfill one (additional) requirement: They have to be non-exhaustive.

Definition 4.1 (Non-exhaustiveness)**.** A gadget is non-exhaustive if it does not contain exhaustive game elements. To be an exhaustive game element means, it can be used only a restricted number of times (like smashing a brick in the Super Mario games). On the other hand, a non-exhaustive game element can be used arbitrarily often without changing its behavior. Non-exhaustive gadgets are needed to handle the repeating/recursive behavior of the framework.

The PSPACE-framework consists of four basic types of gadgets. The start, crossover and finish gadgets, which are the same as already introduced in Chapter 3 for the NP-framework, and the door gadget as the only new basic type. Note that given the non-exhaustiveness constraint, the definition of the crossover gadget in Subsection 3.2.3 gets replaced by non-exhaustiveness: Since the gadget is non-exhaustive, it may not be manipulated when traversed. This makes the requirements for primary/secondary paths to allow third-traversal leakage obsolete. As the gadget may not be manipulated, leakage cannot get possible during gameplay. With this changed definition, leakage has to be prevented in the first place by creating robust crossover gadgets. Note further that this might not be possible with specific game mechanics, which reduces the number of games on which the PSPACE-framework is applicable. (The start and finish gadgets, as they are the end points of the graph, will only be visited once and are therefore allowed to not fulfill the non-exhaustiveness constraint.)

Door gadget A door gadget represents a part of the graph that can only be traversed if a certain condition is fulfilled. Initially, the door is closed. In order to allow traversal, the door needs to be opened. At some point after a door has been traversed, it needs to be closed again. Therefore, two separate paths to and from the door gadget (one to open and one to close) are needed. Leakage from one path to another must be prevented. The player must be able to

change the state of the door gadget, when traversing the open or close path. It is important to force the player to reset the (traverse) path, when visiting the close path. On the other hand, when visiting the open path, [4] allows the player to decide whether or not to perform the action to make the traverse path traversable. This allows an easier construction of the door gadget for some games (see Figure 4.1). Further, it is important to allow multiple traversals of a door gadget. So door gadgets can only be constructed for games that provide non-exhaustive elements (like the tire in Donkey Kong Country or the trampoline in Super Mario World). Also, the actions needed to change the state of the door may only have local effect. This means, everything that can be done in the open or close path is not allowed to alter the state of the game somewhere else. For instance, the trampoline in the Super Mario World instantiation in Section 4.3 may not be carried out of the door gadget as it could be used to break the framework by opening other doors.

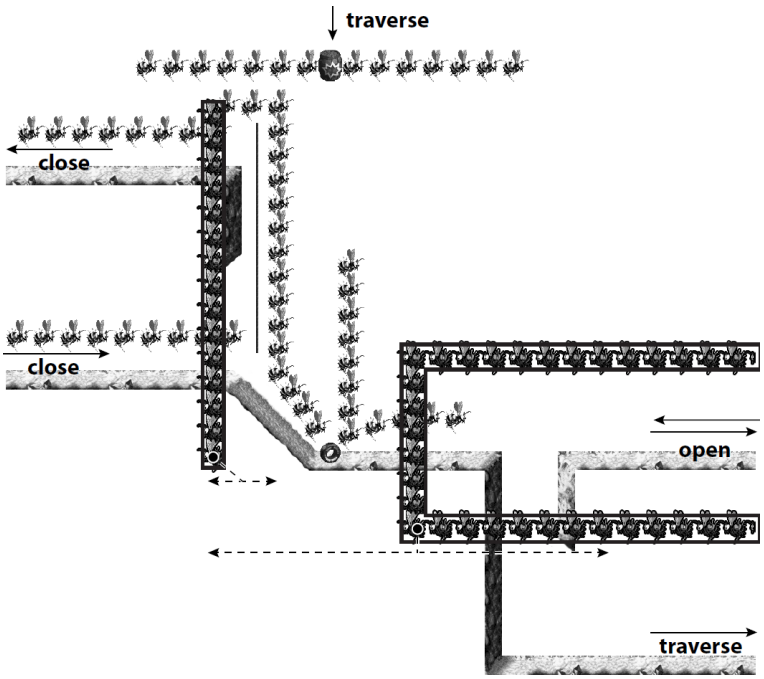

Figure 4.1: Door gadget for Donkey Kong Country [3]: The player enters and leaves on the same path which allows to do nothing and keep the door "closed".

Given the crossover and door gadgets, the following additional gadgets can be constructed:

Generalized door gadget As the construction of the door gadget strongly depends on each game's mechanics, with entrances and exits of the traverse, open and close paths on (most likely) arbitrary sides of the gadget, a generalized version of the door gadget shall be introduced to describe the framework abstractly. Figure 4.2 shows the generalized door gadget. The entrances and exits

of both open and close paths are placed on the top of the gadget, with the open path on the left and the close path on the right. The entrances of both paths are left of the corresponding exits. This is because the literal paths connecting these entrances/exits are coming from the left. If the entrance and exit locations were switched, additional crossover gadgets would be necessary. The entrance and exit of the traverse path are on the right side of the gadget, with the exit placed above the entrance. In order to construct this layout given a normal door gadget, the paths to and from the entrances and exits (most likely) have to be rerouted, potentially introducing crossing paths and therefore crossover gadgets. See Figure 4.14 for the internal structure of the generalized door gadget built on the normal door gadget for Super Mario World. (See Figure 4.13 for the schematic structure of that normal door gadget.) This generalized layout allows a convenient and consistent construction of the framework.

Clause gadget A clause gadget represents a clause of a CNF-formula. For the PSPACE-framework, the clause gadget consists of multiple door gadgets, each representing a literal of the clause. If and only if a literal satisfies the clause, at least one door in the clause gadget has to be opened, allowing traversal through the clause gadget. Therefore, the door gadgets have to be placed in parallel within the clause gadget. See Figure 4.3 for the internal structure of the clause gadget.

Existential quantifier gadget This gadget represents an existentially quantified variable. The assignment process is done by choosing one of two paths within the gadget, visiting either all positive or all negative occurrences of the variable. Figure 4.4 shows the structure of the existential quantifier gadget. This is based on a figure introduced in [4], meaning that the figure shown here specifies the concepts from [4] more precisely. For the positive path, this means the player traverses the open path of each door gadget that represents the positive literal of the current variable. In addition, the player has to traverse the close path of each door gadget, representing the negative literal. Analogously the same for the negative path. Both paths lead back to the existential quantifier gadget after the last door gadget of the negated literal has been closed. As the paths get joined, leakage from one to the other path must be prevented. This is done by the use of two additional door gadgets, mutually closing each other. This control part of the existential quantifier gadget is shown in Figure 4.5. The check path of the framework, leading towards the finish gadget, traverses the existential quantifier gadget on its top in the opposite direction without leakage possibility. This path is only reachable, if the player could traverse all clause gadgets, which means the current variable assignment satisfied the formula. Basically, this does the same as the variable gadget of the NP-framework in Chapter 3. One might notice that when traversing an existential quantifier gadget, the player might open the doors belonging to one literal path and then going back inside the quantifier and follow the other literal path opening doors as well. This does not yield any problems, as the previously opened doors are getting closed by the currently traversed literal path.

Universal quantifier gadget This gadget represents a universally quantified variable. In contrast to the existential quantifier gadget, the player does not choose one of two literal paths but has to traverse both. First, the path for the positive literal has to be traversed. As the player reaches the universal quantifier gadget via the check path, the path for the negative literal has to be traversed. As the player reaches the gadget again after traversing the negative literal path, the check path has to be continued without possibility of taking the negative literal path again. Therefore, several door gadgets are needed to avoid undesired leakage. Figures 4.6 and 4.7 show the structure and the (separated) control part of the universal quantifier gadget. The player enters from the bottom left and first traverses the open path of each positive literal, before traversing the close path of each negative literal. Then door b is opened, before door a is opened, traversed and closed. Doors get always closed right after they have been traversed to avoid leakage. The player can only leave the gadget to the bottom right at this point as door c is closed. When the player enters the gadget via the check path on the top right, the only possibility is traversing and closing door b, as door d at the top is closed. Now, the player traverses the open path of each negative literal, before traversing the close path of each positive literal. Then, door d is opened, before door c is opened, traversed and closed. The player can only leave the gadget on the bottom right and enter it again on the top right via the check path. This time the player can only traverse door d, close it and leave the gadget on the top left. Placing the open paths for each door within the gadget this way avoids any leakage.

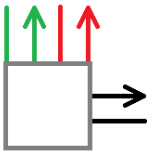

Figure 4.2: Generalized door gadget

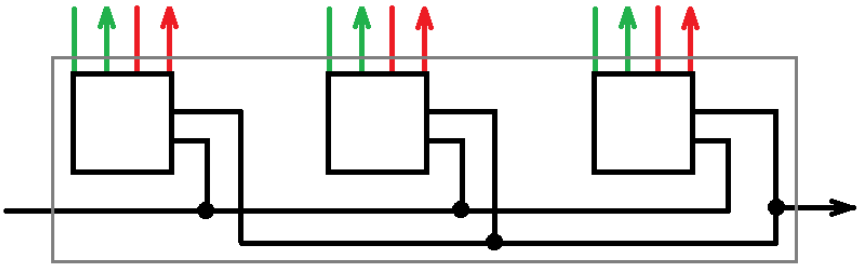

Figure 4.3: Clause gadget (based on [2])

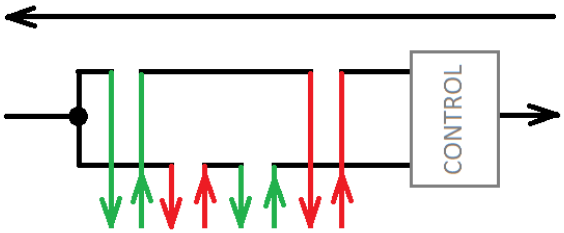

Figure 4.4: Existential quantifier gadget (based on [2])

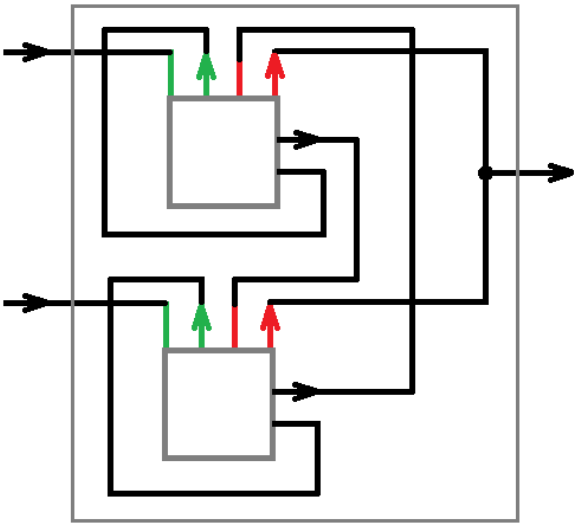

Figure 4.5: Existential quantifier gadget (control part)

Section 4.2 shows the single steps of the constuction of the framework as well as a view on its complexity. In Section 4.3, an instantiation of the framework and its needed gadgets is shown.

4.2 Construction of the Framework

In [4], a high-level construction of the framework is sketched, showing its functionality. Based on that and the new layout of the NP-framework (see Figure 3.2), a fully detailed version of the PSPACE-framework has been constructed that reduces the number of needed crossover gadgets. This section shall show the single steps of the framework's construction, concluding with a complexity analysis and a correctness proof.

4.2.1 Structure of the gadgets

Each gadget has to provide entrances and exits in order to allow the player to traverse it. As the framework describes a graph and the gadgets represent sets of nodes (with internal edges connecting them), those nodes need to be

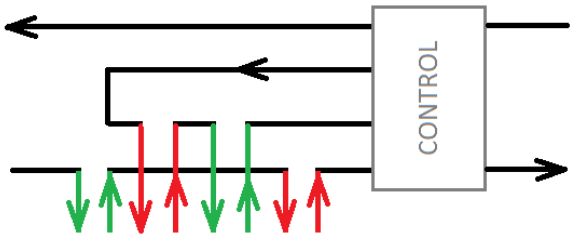

Figure 4.6: Universal quantifier gadget (based on [2])

uniquely identifiable and accessible via the gadget. This is needed in a later construction step when the literal and check paths are connecting the nodes. The following describes the structure of the gadgets not already mentioned in the same form in Chapter 3. In the following, opening and closing paths for doors receive a special notation:

Definition 4.2 (+ / − notation)**.** Paths and nodes associated with the open path of a door gadget are defined by a leading +, while a leading − does for the close path.

Definition 4.3 (Gadget structure)**.**

Generalized door gadget The generalized door gadget (as well as the normal door gadget) consists of three entries and three exits, one for each open, close and traverse path. Each generalized door gadget is assigned a label l. The entry and exit nodes of the open path are labeled $(+l)_{end}$ and $(+l)_{begin}$. Similarly, the nodes of the close path are labeled $(-l)_{end}$ and $(-l)_{begin}$. All those nodes are placed on the top of the gadget in the following order from left to right: $(+l)_{end}, (+l)_{begin}, (-l)_{end}, (-l)_{begin}$. The entry and exit nodes of the traverse path are labeled $(l)_{end}$ and $(l)_{begin}$. They are placed on the right side of the gadget with the exit above the entry.

Clause gadget A clause C containing n literals is represented as a clause gadget consisting of n generalized door gadgets, where each door represents a literal $l_i \in C$ with $1 \leqslant i \leqslant n$. Each generalized door gadget provides its entries and exits of the open and close paths on the top of the clause gadget. The entry and exit nodes of the check path on the left and right of the clause gadget are labeled $(check)_{end}$ and $(check)_{begin}$.

Existential quantifier gadget On the bottom left, there is an entry node labeled $(assign)_{end}$. The exit node on the bottom right is labeled $(assign)_{begin}$. The check path enters on the top right, labeled $(check)_{end}$ and exits on the top left node, labeled $(check)_{begin}$. The gadget provides exit and entry nodes for both literals of the quantified variable v. The overall order of nodes of both literal paths (from left to right) is $(+v)_{begin}, (+v)_{end}, (-v)_{begin}, (-v)_{end}, (+\overline{v})_{begin}, (+\overline{v})_{end}, (-\overline{v})_{begin}, (-\overline{v})_{end}$.

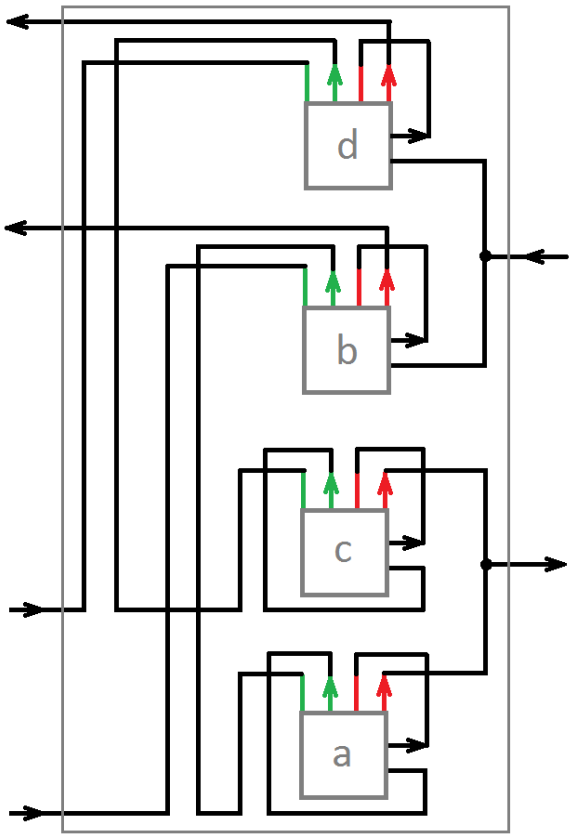

Figure 4.7: Universal quantifier gadget (control part)

Universal quantifier gadget There is the assign path with nodes $(assign)_{end}$ and $(assign)_{begin}$ at the bottom left and right as well as the check path on the top with nodes labeled $(check)_{end}$ on the right and $(check)_{begin}$ on the left. At the bottom, the gadget provides exit and entry nodes in the same order as the existential quantifier gadget (from left to right): $(+v)_{begin}, (+v)_{end}, (-v)_{begin}, (-v)_{end}, (+\overline{v})_{begin}, (+\overline{v})_{end}, (-\overline{v})_{begin}, (-\overline{v})_{end}$.

The labels within generalized door gadgets can refer either to a literal or an arbitrary value, when used as an internal door of another gadget (like a, b, c, d within the universal quantifier gadget). Internal doors do not provide entries or exits that are visible outside of the containing gadget. In contrast to this, external doors (in particular the doors of the clause gadget) need unique labels such that they can be identified when the paths are set up. The check path within the clause gadget connects the traverse paths of the doors in the following way: It enters from the left and visits the entry nodes of the traverse paths of each door. The check path leaves the clause gadget on the right, connecting the exit nodes of each traverse path. To traverse the check path, at least one of the doors needs to be opened. See Figure 4.8 for the labeling of the clause gadget. The assign path within the existential quantifier gadget splits into an upper and lower path, representing the positive and negative literal paths. The exit and

entry nodes for the open path of the positive literal are connected to the upper path. Further, the upper path contains the exit and entry nodes for the close path of the negative literal. The nodes in the positive literal path are labeled in the following order from left to right: $(+v)_{begin}, (+v)_{end}, (-\overline{v})_{begin}, (-\overline{v})_{end}$. On the negative (lower) literal path, labels appear in the following order: $(-v)_{begin}, (-v)_{end}, (+\overline{v})_{begin}, (+\overline{v})_{end}$. Note that the nodes of the negative literal path lie in between the nodes of the positive path. From an outside view, the universal quantifier gadget provides the same structure as the existential quantifier gadget. The nodes starting directly from the assign path are $(+v)_{begin}, (+v)_{end}, (-\overline{v})_{begin}, (-\overline{v})_{end}$, while the nodes connected to the path between the internal doors b and c are $(-v)_{begin}, (-v)_{end}, (+\overline{v})_{begin}, (+\overline{v})_{end}$. Figure 4.9 shows the labeling of both quantifier gadgets.

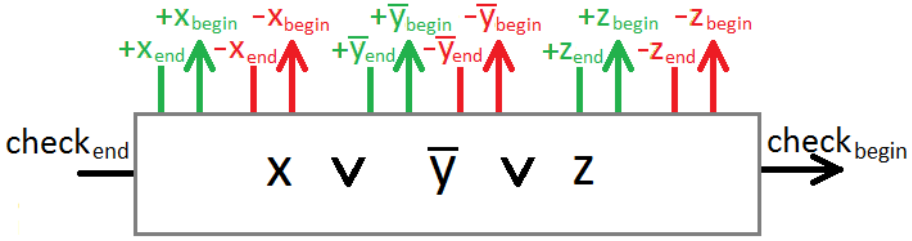

Figure 4.8: Clause gadget labeling example: $x \vee \overline{y} \vee z$

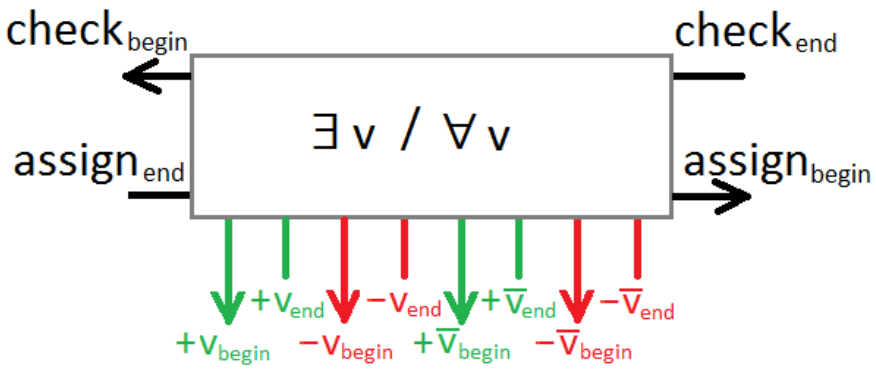

Figure 4.9: Labeling for both existential and universal quantifier gadgets

By providing gadgets with nodes labeled like this, the assign, literal and check paths can be easily set up in a later construction step.

4.2.2 Arranging the gadgets

Given a two-dimensional grid, defining a coordinate system growing in top right direction, gadgets are arranged by assigning them to a location on that grid. The size of a gadget depends on the number of paths that are connected to it. If not stated different, each gadget is considered to be of width and height 1. The sizes are idealized to allow a more convenient construction. This means

that internal control structures (like the doors a, b, c, d in the universal quantifier gadget) are ignored in this step of the construction. The implementation requires gadgets of larger size. This will not introduce any problems, as the graph just gets stretched by the larger gadgets. The gadgets are placed in the following way:

Definition 4.4 (Gadget location)**.**

Start gadget This is placed in the top left, on position $(-2, 8 \cdot n + 1)$.

Existential / Universal quantifier gadgets Let the formula contain n quantified variables. Both existential and universal quantifier gadgets have a size of $(8, 2)$. For each i with $1 \leqslant i \leqslant n$, a quantifier gadget is created on location $((i-1) \cdot 8, 8 \cdot n + 3)$. If $i \bmod 2 = 1$, an existential quantifier gadget is placed on this location, if $i \bmod 2 = 0$, a universal quantifier gadget is used. This leads to a row of alternating quantifier gadgets on the top, starting with an existential quantifier gadget. As quantifier alternation is not needed in practice though, this can be dropped such that the quantifier order only depends on the given formula.

Clause gadgets Given n quantified variables and m clauses. These gadgets are placed on the bottom in a row, right to the quantifier gadgets, leading to a location offset of $(8 \cdot n, 0)$. The clause gadgets share a constant size of $(4 \cdot 2 \cdot n, 1)$, ignoring the actual number of literals per clause but providing space for every possible literal. This uniform size allows easy construction while applying for all real clause sizes. Therefore, clause gadget j with $1 \leqslant j \leqslant m$ is placed on location $(8 \cdot n + (j-1) \cdot 4 \cdot 2 \cdot n, 0)$.

Crossover gadgets The positions of the crossover gadgets will be assigned during their creation in the next step of the construction (see Subsection 4.2.3).

Finish gadget This is placed in the top left, above the start gadget on position $(-2, 8 \cdot n + 3)$.

Figure 4.10 shows how the PSPACE-framework abstractly looks like for a given example. Crossover gadgets are not displayed in this figure. Note that the finish gadget is placed above the start gadget. The PSPACE-framework differs from the NP-framework here. This comes from the fact that a check path leading back to the quantifier gadgets does not cross any literal paths if it passes the quantifier gadgets on their top. Therefore, the check path within each quantifier gadget is placed above the assign path, hence the finish gadget has to be above the start gadget. Recall that the first two (outgoing and incoming) edges of a quantifier gadget belong to the open path of the positive literal, the next two edges belong to the close path of the negative literal. The last four edges belong to the open path of the negative literal and the close path of the positive literal.

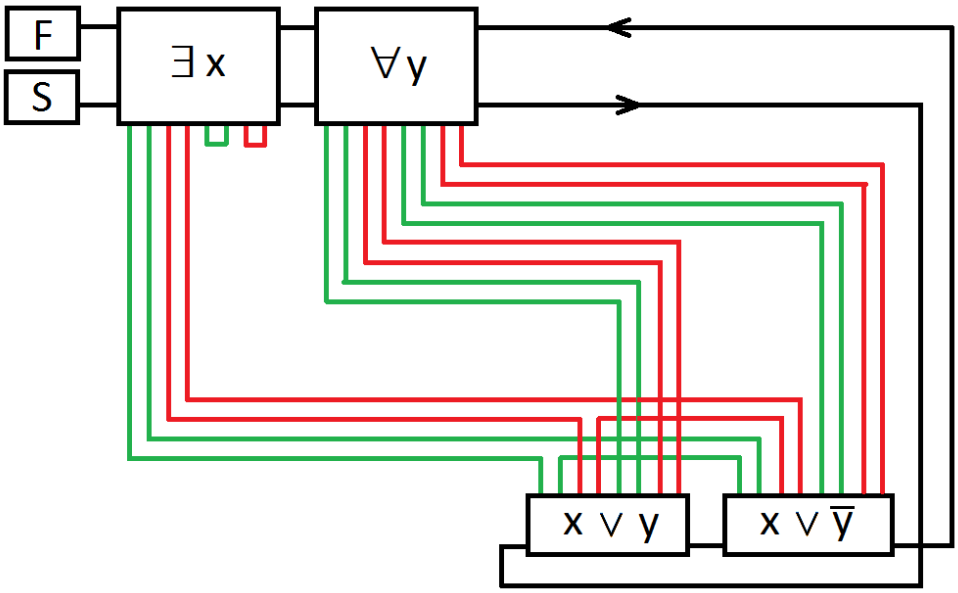

Figure 4.10: PSPACE-framework example: $\exists x \forall y((x \vee y) \wedge (x \vee \overline{y}))$

4.2.3 Creating crossover gadgets

When paths cross each other, crossover gadgets are needed to resolve the crossing section, avoiding any leakage from one path to another. The (basic) definition and requirements of the crossover gadgets have already been given in Chapter 3. Recall that the crossover gadget of the PSPACE-framework additionally needs to be non-exhaustive. This subsection deals with locating the external crossover gadgets used in the PSPACE-framework. Given the set of variables V and the set of clauses $\mathcal{C}$, an extended set of literals L, including open and close information of the corresponding doors in the clause gadgets, can be defined with its elements following the order $+v_1 \prec -v_1 \prec +\overline{v_1} \prec -\overline{v_1} \prec \cdots \prec +v_n \prec -v_n \prec +\overline{v_n} \prec -\overline{v_n}$ where $n = |V|$. The set of crossover gadgets is defined as follows.

Definition 4.5 (Set of crossover gadgets)**.** Using L, the locations of the crossover gadgets can be determined: For each $l_i \in L$ with $1 \leqslant i < |L| = 4 \cdot n$, $\mathcal{C}_{l_i}$ defines the set of clauses that contain l_i. Note, that i starts at 1, ignoring literal path l_0 as the first path can be created without crossing another path. For each clause $C_k \in \mathcal{C}_{l_i}$ with $1 \leqslant k \leqslant m$, where $m = |\mathcal{C}|$, the horizontal location (based on the origin of the coordinate system) of the node labeled l_i within C_k is calculated as $x_{C_k,l_i} = 8 \cdot n + (k-1) \cdot 4 \cdot 2 \cdot n + 2 \cdot i$. Note that the index k of a clause still refers to the index of that clause within the full set of clauses $\mathcal{C}$. This is needed to be able to access the coordinates of the clause C_k. For each l_j with $0 \leqslant j < i$, it needs to be checked whether the literal path for l_i crosses the below-lying literal path for l_j. The paths will cross, if l_j appears in a clause C_h with $h > k$, meaning C_h lies right of the current clause C_k in the graph. Due to the literal order within the clause gadgets, the path for l_i cannot cross the

path for l_j in the region of the current clause C_k, if C_k is the last clause containing both l_i and l_j. Therefore, only clauses on the right of C_k (with $h > k$) are relevant. If this condition is fulfilled, four crossover gadgets are created at locations (x_{C_k,l_i}, y_j), $(x_{C_k,l_i} + 1, y_j)$, $(x_{C_k,l_i}, y_j + 1)$ and $(x_{C_k,l_i} + 1, y_j + 1)$ with $y_j = 2 \cdot j + 1$. In addition, there is another crossover gadget on the bottom right of the framework. As the clause gadgets should be traversed by the check path from left to right (which saves one internal crossover gadget per clause), one additional crossover gadget is needed to resolve the crossing section of the check path with itself, leading to and from the clause gadgets. This crossover is located at $(8 \cdot n + m \cdot 8 \cdot n + 1, -1)$.

4.2.4 Setting up the paths

As all gadgets have been located, each providing a set of nodes, the literal and check paths can be set up to finish the graph. Given the set of quantified variables V, the set of clauses $\mathcal{C}$ and the set of crossover gadgets X. The quantifier gadgets shall be represented by their quantified variables. In the following, gadget v_i refers to the i^{th} quantifier gadget. The construction needs a list of gadgets, serving as start nodes of literal paths, called $subpathGadgets = \{v_1, v_1, v_1, v_1, \ldots, v_n, v_n, v_n, v_n\}$. From each quantifier gadget v_i four subpaths $(+v_i, -v_i, +\overline{v_i}, -\overline{v_i})$ are starting. Therefore, each variable appears four times in the list. Further, the list of labels is needed to describe the nodes that need to be traversed by the current path: $subpathLabels = \{+v_1, -v_1, +\overline{v_1}, -\overline{v_1}, \ldots, +v_n, -v_n, +\overline{v_n}, -\overline{v_n}\}$. The elements in both lists correspond to each other. The i^{th} label in $subpathLabels$ describes the literal path, starting from the i^{th} gadget in $subpathGadgets$.

Given this information, the literal paths can be set up as follows: Let i describe the index of the currently created literal path, initially $i = 0$. Let $n = |V|$ describe the number of quantified variables and $m = |\mathcal{C}|$ denote the number of clauses. The current location of the path is accesible via $curX$ and $curY$. The i^{th} elements are taken from both lists, namely $node \in subpathGadgets$ and $l \in subpathLabels$. The horizontal locations on which label $(l)_{end}$ appears in clause gadgets are stored in the list $xLocs$ with its elements sorted in ascending order. If the label $(l)_{end}$ does not appear in any clause, its path simply connects the nodes labeled $(l)_{begin}$ and $(l)_{end}$ within the quantifier gadget $node$ and continues with the next literal path. Otherwise, if there are clauses containing $(l)_{end}$, the literal path is set up, creating triples of subpaths. Each triple begins with a vertical part, continues horizontal and ends with a vertical part. Literal path i starts from a quantifier gadget at the location $(i, 8 \cdot n + 1)$. So initially $curX = i$ and $curY = 8 \cdot n + 1$. Each subpath-triple visits the following four locations: $((curX, curY), (curX, y), (x, y), (x, 0))$ with $y = 2 \cdot i + 1$ denoting the height on which the horizontal parts of the path are lying and x being the first element of $xLocs$ (where x is removed from $xLocs$ at the same time). Those subpaths might be single edges but probably will have to visit crossover gadgets that are located in the regions of the literal paths. A crossover gadget has to be connected by a subpath from location a to b if it fulfills the location constraint $([x_1, x_2], [y_1, y_2])$ with $x_1 = \min(a_x, b_x)$, $x_2 = \max(a_x, b_x)$, $y_1 = \min(a_y, b_y)$

```
function createPaths()
    for i ← 0 to subpathGadgets.size() do
        node ← subpathGadgets.get(i)
        l ← subpathLabels.get(i)
        xLocs ← horizontalLocationsOfLiterals(C, l_end)
        sort(xLocs)
        y ← 2 · i + 1
        curX ← i
        curY ← 8 · n + 1
        //no clauses to visit; directly connect to ingoing node of gadget
        if xLocs.isEmpty() then
            connectCrossovers((curX, curY), (curX + 1, curY), l)
            continue
        end if
        //visit clauses
        while not xLocs.isEmpty() do
            x ← xLocs.pop()
            connectCrossovers((curX, curY), (curX, y), (x, y), (x, 0), l)
            curX ← x + 1
            curY ← 0
        end while
        //create back path
        connectCrossovers((curX, curY), (curX, y + 1), (i + 1, y + 1),
            (i + 1, 8 · n + 1), l)
    end for
end function
function connectCrossovers(list(locs), label)
    curLoc ← locs[0]
    for i ← 1 . . . locs.size − 1 do
        destLoc ← locs[i]
        for all x ∈ X do
            //crossover x lies between start and end locations therefore
            //by construction needs to have a node called label_end
            if inInterval(x.posX, curLoc.posX, destLoc.posX) and
             inInterval(x.posY, curLoc.posY, destLoc.posY) then
                Edge(curLoc, x.getLocation(label_end))
                curLoc ← x.getLocation(label_begin)
            end if
        end for
        Edge(curLoc, destLoc)
        curLoc ← destLoc
    end for
end function
```

Figure 4.11: Path creation algorithm (PSPACE-framework)

and $y_2 = \max(a_y, b_y)$. Again, this means that the horizontal location has to lie between x_1 and x_2 and the vertical location has to lie between y_1 and y_2. For each location-pair being connected in this step of the construction, this check for crossover gadgets has to be performed. As a subpath-triple has been created, it needs to be checked whether there are further clauses containing the current label $(l)_{end}$ that need to be connected or whether the literal path can be finished by heading back to the entry node labeled $(l)_{end}$ in the quantifier gadget. In either case, the current location of the path has to be updated after a subpath-triple has been finished. So $curX = x + 1$ as each subpath has to start from a node labeled $(l)_{begin}$ next to the entry node, while $curY = 0$. If *xLocs* is not empty, the next subpath-triple leads to another clause gadget. Otherwise, the path leads back to the quantifier gadget, connecting the locations $((curX, curY), (curX, y+1), (i+1, y+1), (i+1, 8 \cdot n+1))$. The height of $y+1$ indicates that the back-path runs above the actual literal path.

The paths between the start, finish and quantifier gadgets as well as the check path are created separately: The start path is created by a single edge from the $(start)_{begin}$ node in the start gadget to the $(assign)_{end}$ node in the first quantifier gadget. The finish path connects the $(check)_{begin}$ node of the first quantifier gadget with the $(check)_{end}$ node of the finish gadget. For each i with $1 \leqslant i < n$, the $(assign)_{begin}$ node of the i^{th} quantifier gadget is connected with the $(assign)_{end}$ node of the $(i+1)^{th}$ gadget. In addition, the $(check)_{begin}$ node of the $(i+1)^{th}$ quantifier gadget connects to the $(check)_{end}$ node of the i^{th} gadget. For each j with $1 \leqslant j < m$, the $(check)_{begin}$ node of the j^{th} clause gadget is connected with the $(check)_{end}$ node of the $(j+1)^{th}$ clause gadget. No crossover gadgets will appear in those regions, so single edges can be used. The (remaining) check path consists of two subpaths, one leading to and one coming from the clause gadgets. These subpaths, described as location-tuples, look as follows: $((8 \cdot n, 8 \cdot n + 2), (8 \cdot n + m \cdot 8 \cdot n + 1, 8 \cdot n + 2), (8 \cdot n + m \cdot 8 \cdot n + 1, -2), (8 \cdot n - 1, -2), (8 \cdot n - 1, -1), (8 \cdot n, -1))$ leading to the first clause gadget and $((8 \cdot n + m \cdot 8 \cdot n, -1), (8 \cdot n + m \cdot 8 \cdot n + 2, -1), (8 \cdot n + m \cdot 8 \cdot n + 2, 8 \cdot n + 3), (8 \cdot n, 8 \cdot n + 3))$. For these two subpaths the crossover location constraint has to be applied.

The nodes within a gadget and their locations on the coordinate system can be used equivalently. During this construction the more convenient alternative has been used.

4.2.5 Reduction

This subsection gives insight in the reduction from QSAT on which the whole construction of the PSPACE-framework is based. This contains correctness and complexity proofs.

Lemma 4.6. *Let the described construction be called PSPACE-FRAMEWORK problem. There is a function R with the following property:*

$$\forall \Phi : \Phi \in QSAT \Leftrightarrow R(\Phi) \in PSPACE\text{-}FRAMEWORK$$

Therefore R is a reduction from QSAT to PSPACE-FRAMEWORK.

Proof. Again, it is important that the framework behaves correctly at any time, in particular, it may not allow the player to exploit leakage to break the game. So the player may only traverse those paths that are intended to be traversed at any point throughout the game. Note: A solvable game g (finish gadget can be reached) states that $g \in$ PSPACE-FRAMEWORK. The proof is split in two parts:

($\Rightarrow$): If the formula is in QSAT, then the framework generates a game that can be solved: $\forall \Phi : \Phi \in \text{QSAT} \Rightarrow R(\Phi) \in \text{PSPACE-FRAMEWORK}$

Membership in QSAT means that for each combination of the universally quantified variables the existentially quantified variables can be assigned in such a way that each clause gets satisifed. This leads to a set of assignments $\Phi = \{\phi_1, \ldots, \phi_{2^m}\}$ where m denotes the number of universally quantified variables. Note that there might be more than one satisfying assignment for each combination of universally quantified variables. The set Φ only asks for one though. The player traverses the quantifier gadgets by choosing literal paths based on an assignment $\phi_i \in \Phi$. On these literal paths, the player visits and opens doors (that represent literals) in clause gadgets. As soon as all quantifier gadgets have been traversed, the assignment is done and needs to be verified by traversing the check path. The check path visits each clause gadget and can only pass if at least one of its doors is open (a literal satisfies the clause). As the player followed literal paths based on ϕ_i which satisfies all clauses, each clause gadget can be passed and the player returns to the quantifier gadgets where the procedure starts over with another (not yet chosen) assignment $\phi_j \in \Phi$. As all assignments in Φ have been checked (each satisfying the formula), the player can finish the game.

($\Leftarrow$): If the framework generates a solvable game, then the represented formula is in QSAT: $\forall \Phi : \Phi \in \text{QSAT} \Leftarrow R(\Phi) \in \text{PSPACE-FRAMEWORK}$

A solvable game means that the player could reach the finish gadget after (successfully) traversing the check path, given an assignment for each combination of universally quantified variables. In terms of the graph, representing the game, assignment means that the player makes decisions which of two literal paths for each variable to follow and open doors lying on that path. The order in which those assignments are chosen is defined by the framework: Each time the player reaches the quantifier gadgets, the values of the universally quantified variables change in a bitwise scheme where the i^{th} bit represents the i^{th} universally quantified variable with $1 \leqslant i \leqslant m$. Interpreted as a binary number it counts from $2^m - 1$ down to 0 with a 1 bit denoting the variable set to true while a 0 bit denotes the variable set to false. The framework reroutes the player when visiting a universal quantifier gadget via the check path according to that scheme. This way, each combination of universally quantified variables is chosen exactly once. The player has to choose the (positive or negative literal) paths for the existentially quantified variables such that all clause gadgets can be passed on the check path. As the game is solvable, there are such assignments for each combination of universally quantified variables. Each door in the clause gadgets represents one of its literals. If the door is open, the literal is true, if the door is closed, it is false. Based on that, the variable assignment can be read off the state of the open doors. Note that the closed doors are

not of relevance here. After each traversal of the check path the assignment ϕ_i with $1 \leqslant i \leqslant 2^m$ is read off and added to the set of assignments Φ. The read off assignments might be of different length as possibly not all variables need to be used to create a satisfying assignment. In that case the missing variables are generically set to false and added to ϕ_i in order to complete the assignment. If the player finally reaches the finish gadget, Φ is verified by feeding its assignments to the formula. As Φ contains assignments for all combinations of universally quantified variables and each of this assignments satisfies each clause, the formula is in QSAT. □

Complexity This paragraph shows the complexity of the described construction of the framework.

Lemma 4.7. *The function R denoting the reduction QSAT $\leq_{log}$ PSPACE-FRAMEWORK is logspace computable.*

Proof. The construction of the PSPACE-FRAMEWORK problem consists of multiple parts. This proof focusses only on the interesting parts of the computation. The generation of the start, finish, quantifier and clause gadgets (directly coming from a given formula) is trivial. Also the generation of the paths connecting two gadgets is considered straight-forward. What remains interesting is the generation of the crossover gadgets, in particular, calculating where two paths cross each other. It needs to be checked, whether the computation can be performed within logarithmic space. Given a three-tape Turing machine with an encoding of all literals and clauses on its input tape, where the literals follow the extended order $+v_1 \prec -v_1 \prec +\overline{v_1} \prec -\overline{v_1} \prec \cdots \prec +v_n \prec -v_n \prec +\overline{v_n} \prec -\overline{v_n}$. The work tape then looks like this:

$$i\ j\ k\ h\ x\ y,$$

with its elements denoting the following:

- i: This is a binary encoding of the currently checked literal path l_i.
- j: This is a binary encoding of the second literal path l_j that potentially gets crossed by the vertical parts of literal path l_i with $j < i$.
- k: This is a binary encoding of the currently checked clause C_k, whether it contains literal l_i.
- h: This is a binary encoding of the second clause C_h that contains literal l_j with $h > k$.
- x: This is a binary encoding of the absolute horizontal location of literal l_i if it is contained in clause C_k. If $l_i \notin C_k$, this value does not matter.
- y: This is a binary encoding of the absolute vertical location on which the horizontal parts of the literal path l_j are running.

With n quantified variables, $8 \cdot n$ literals and m clauses the work tape needs $2 \cdot \log(8 \cdot n) + 2 \cdot \log(m) + \log((m+1) \cdot 8 \cdot n) + \log(8 \cdot n)$ bits of space, which obviously is logarithmic in terms of the input size.

With this work tape, the calculation works as follows, with all indices starting from 0: For each literal l_i with $1 \leqslant i < |L| = 8 \cdot n$ and $n = |V|$, each literal clause C_k with $0 \leqslant k < |\mathcal{C}| = m$ is checked, whether it contains l_i. If so, for each literal l_j with $0 \leqslant j < i$ it needs to be checked if l_j appears in a clause C_h with $k < h < m$, meaning C_h lies right of the current clause C_k in the graph. If so, l_i crosses l_j and the horizontal location of the node labeled l_i within C_k is calculated as $x = 8 \cdot n + k \cdot 8 \cdot n + 2 \cdot i$. Four new crossover gadgets are placed at locations (x, y), $(x+1, y)$, $(x, y+1)$ and $(x+1, y+1)$ with $y = 2 \cdot j + 1$. □

4.3 Instantiation for Super Mario World

In [2, 3, 4], the PSPACE-framework has been applied to Donkey Kong Country and Legend of Zelda, making those games PSPACE-hard and further showing membership in PSPACE, thus categorizing them PSPACE-complete. The crucial step in the application of the PSPACE-framework is to create door and crossover gadgets that follow the given game mechanics, using only non-exhaustive game elements without allowing any leakage. This thesis shall introduce the missing door gadget and a new version of the crossover gadget for Super Mario World and show its membership in PSPACE, therefore adding it to the list of PSPACE-complete games.

As the door and crossover gadgets need to be traversable multiple times, non-exhausting game elements are needed. Super Mario World offers trampolines, that can be picked up (when approaching horizontally) and dropped. Trampolines can be used arbitrarily often and allow Mario to bounce off and jump higher. Another non-exhausting element of Super Mario World is the rotating block. If Mario jumps from below, it begins to rotate and gets passable for a short amount of time. This means Mario will not collide with that block, which allows him to be bounced past it using a trampoline. If (big) Super Mario performs a spinning jump on top of the rotating block, it gets destroyed. Therefore, Mario has to be in small state and super mushrooms are neither allowed nor needed in the implementation of the PSPACE-framework as the mushroom blocks needed to spawn super mushrooms would also be exhaustive. Mario collides with trampolines when touching them from the left, right and top. There is no collision if Mario touches a trampoline from the bottom, in particular trampolines can fall through Mario without collision. Further, it is assumed that trampolines can be either fixed or movable and do not collide with each other, which allows them to be "stacked within" each other. Given this information, the door gadget and a non-exhaustive and less complex crossover gadget for Super Mario World (in comparison to the NP-framework) can be created.

4.3.1 Door gadget

As the door gadget consists of three paths, its functionality is split according to that paths. One-way functionality does not need to be implemented within that gadget, as this is sufficiently achieved by the surrounding graph. So if Mario leaves the gadget via its entrance, no harmful steps can be performed.

Open path Mario enters from the top left and jumps down the left vertical tunnel. At the bottom, Mario heads right where a fixed trampoline blocks his way. If the door is in its closed state, there is another (movable) trampoline inside the fixed one. Mario carries the movable trampoline via the left vertical tunnel to the top and throws it down the right vertical tunnel. The trampoline falls until it reaches the rotating block. Mario cannot leak to the traverse path via the right vertical tunnel as the fire on the top would kill him. (Note, that there are no super mushrooms in the PSPACE-framework.) Mario again jumps down the left tunnel and leaves via the bottom left. If the door is already in its opened state, Mario immediately leaves the gadget via bottom left. As demanded by the framework, Mario can choose to leave the door in closed state without opening it. Leakage to the close path is not possible as the fixed trampoline cannot be passed. It is important to prevent the player from carrying the trampoline outside the gadget as a closed door could be opened via its traverse path by placing the carried trampoline. Therefore, a vine at the exit of the open path forces Mario to climb which is not possible while carrying a trampoline.

Traverse path Mario enters from the upper middle path on the right and needs to jump up and leave via the top right. To overcome this height, the door has to be opened before by throwing the movable trampoline down the right vertical tunnel. Using the trampoline, Mario can reach the upper path and leave. Leakage to the open path is not possible as there is fire placed on the top. Leakage down to the close path is not possible, as even if there is no trampoline, allowing Mario to reach the rotating block, Mario is small and cannot destroy it with a spinning jump. The traverse path represents the actual door: If it is traversable, the door is open, otherwise the door is closed.

Close path Mario enters from the bottom right and uses the fixed trampoline to jump up to the next horizontal tunnel and leave the gadget. The trampoline will bounce Mario against the rotating block before he can leave horizontally, such that the movable trampoline (if previously placed on the rotating block) falls to the ground. It is important for the fixed trampoline to bounce fast and high enough, not allowing Mario to leave without hitting the rotating block, as the close path always has to make the traverse path not traversable.

Figure 4.12 shows the door gadget for Super Mario World while 4.13 is reduced to its schematic structure. The green arrows denote the open path, black arrows denote the traverse path and red arrows denote the close path.

4.3.2 Non-exhaustive crossover gadget

Given trampolines and spiky, rotating stone balls, a slim crossover gadget can be created that is non-exhaustive and keeps the resulting game worlds smaller than the version in the NP-framework. Figure 4.15 shows the new crossover gadget for Super Mario World. The primary path leads horizontally from left to right through the gadget. All stone balls are rotating clockwise with the same speed. There are four locations on which those balls are placed. Each ball has a twin that rotates with an angle offset of 180 degrees. This rotating behavior leads to short phases in which Mario can quickly run horizontally or pass throuth vertically without getting hit by a stone ball. Leakage is not possible, since if one path is free, the other is blocked. Waiting in the crossing region for the other path to get traversable will not work as there always is a ball that will pass Mario's location before he gets a chance to leak to the other path. The primary path leads from left to right. The secondary path leads from the bottom to the top. Mario uses the trampoline to launch him with very high speed such that he can reach the horizontal way on the top. Figure 4.16 shows an alternative version in which Mario has to fall down in the vertical path at a well-timed moment, instead of using a trampoline to launch him up. If there was only one of these versions, winding paths around it (as mentioned in Section 3.2.3, Figure 3.5) would be necessary as only one of both bottom-up or top-down traversals would be possible without that technique. Given both versions of the non-exhaustive crossover gadget, horizontal mirroring and no leakage possibilities (which removes any primary and secondary path declarations as there is no predefined traversal order), no path winding is required which saves a lot of space in an actual game world.

4.3.3 Start and finish gadgets

As super mushrooms are neither needed nor allowed in the PSPACE-framework, new versions of the start and finish gadgets are needed as well. This is achieved by simply removing the super mushroom block and barrier from the start and finish gadgets introduced in [4].

4.3.4 PSPACE-completeness

As the PSPACE-framework can be applied to the generalized version of Super Mario World, this game is PSPACE-hard. Now it shall be shown to be PSPACE-complete by proving its membership in PSPACE.

Theorem 4.8 (Membership in PSPACE)**.** *Generalized Super Mario World is a member of PSPACE.*

Proof. To describe a state of the game, the following information is needed: Mario's location, the location of the trampolines and the states of the rotating blocks (traversable or not). In addition, the information whether Mario is carrying a trampoline is needed. This describes everything that can change while playing and is a relevant part of the game. Given a game world of size $n = width \times height$ with t trampolines and r rotating blocks, there are $\log(n)$

bits for Mario's location, $t \cdot \log(n)$ bits for the locations of the trampolines, r bits for the states of the rotating blocks and 1 more bit to describe whether a trampoline is currently carried. This leads to a total of $(t+1) \cdot \log(n) + r + 1$ bits needed to describe a state of the game, which is linearithmic in the size of the input and therefore within polynomial space.

The goal is to move Mario to a defined finish location in order to win the game. The algorithm non-deterministically guesses an assignment which allows the player to reach this finish location. By Savitch's theorem NPSPACE = PSPACE, so Super Mario World is a member of PSPACE. □

Corollary 4.9 (PSPACE-completeness)**.** *Generalized Super Mario World is PSPACE-complete.*

Proof. From the applicability of the PSPACE-hardness framework and Theorem 4.8 follows that generalized Super Mario World is PSPACE-complete. □

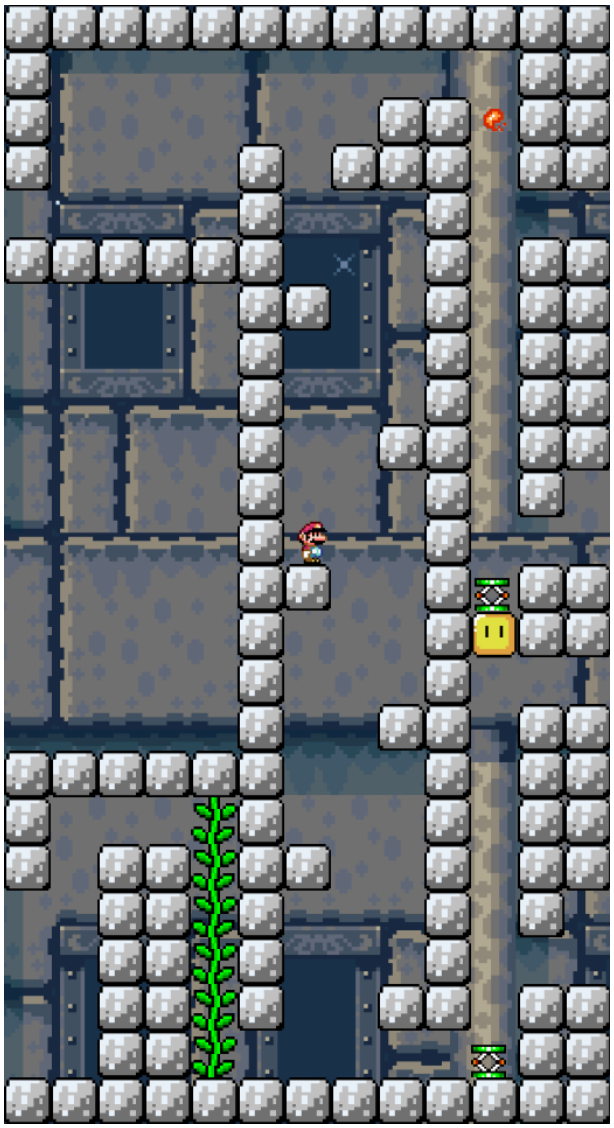

Figure 4.12: Door gadget for Super Mario World (open state)

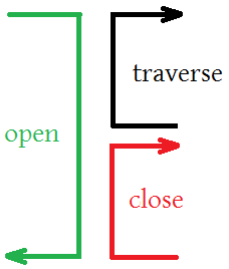

Figure 4.13: Door gadget for Super Mario World (schematic)

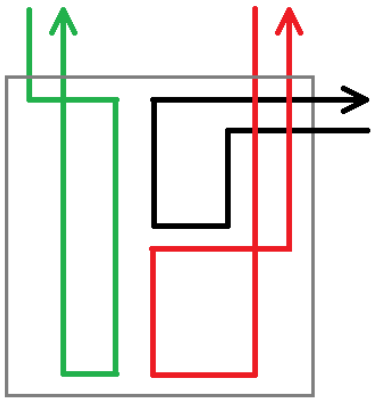

Figure 4.14: Generalized door gadget, Super Mario World (internal structure)

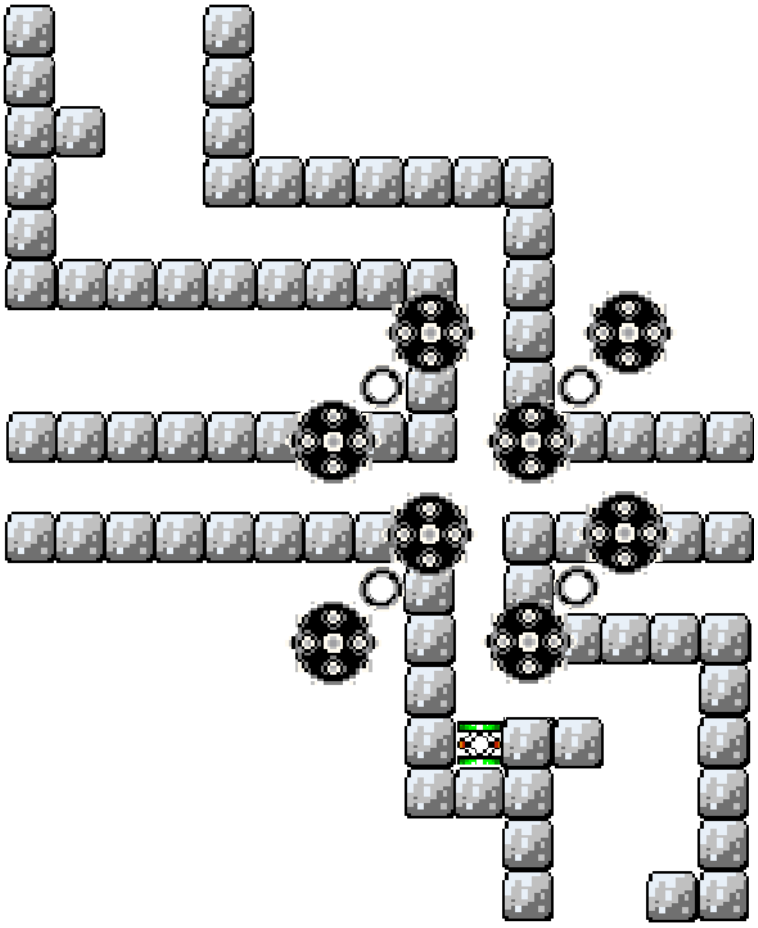

Figure 4.15: Non-exhaustive crossover gadget for Super Mario World (v1)

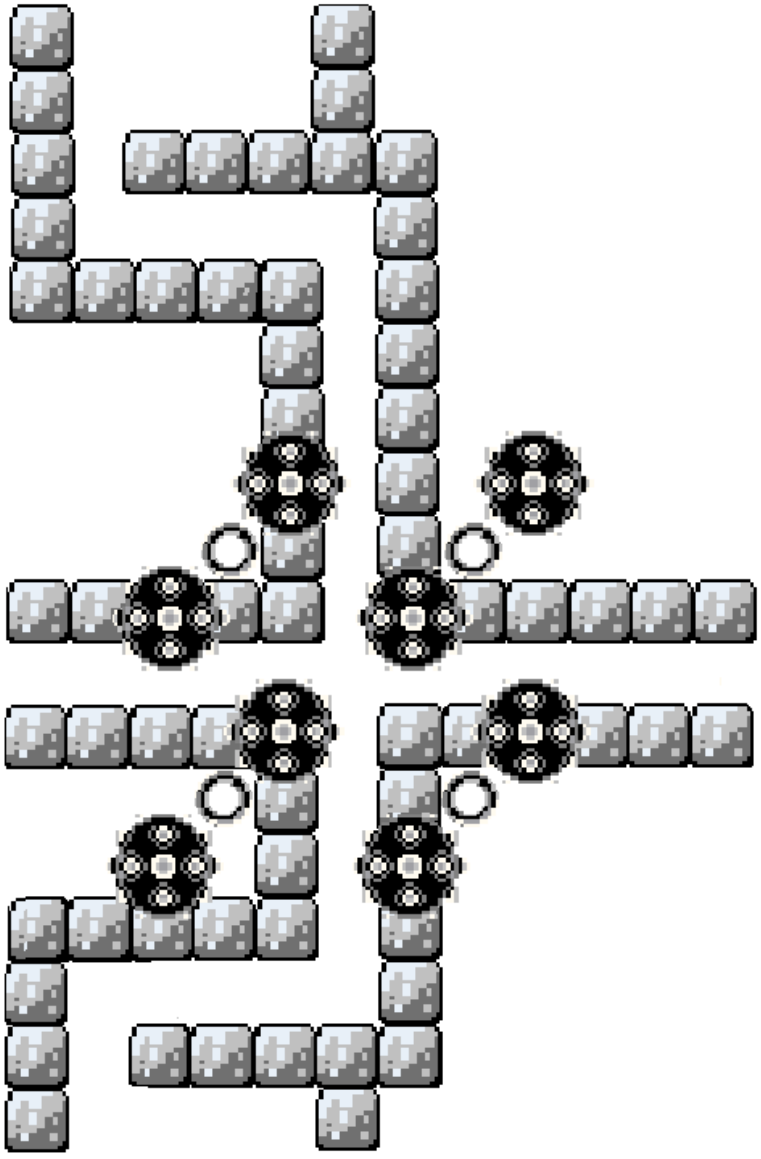

Figure 4.16: Non-exhaustive crossover gadget for Super Mario World (v2)

5 Game Engine

5.1 Introduction

To complement the previous theoretical parts, both introduced frameworks have been implemented in Java as generalized versions of Super Mario World.

This part shall briefly explain the implemented game engine. This is roughly split into two sections: First the physics engine is introduced, giving some details about movement and collision handling. The next section explains how the game world is created and rendered based on a given (CNF or QSAT) formula.

Where referred to the implementation, class names are written in italic and capitalized (like *Block*).

5.2 Physics

This section explains the (primitive) physics engine used in the game which consists of movement and collision handling. All coordinate systems grow positively to the right and bottom.

5.2.1 Kinematics

To move an object from point A to point B, some parameters like location, velocity and acceleration need to be known. All these parameters are vectors, in terms of Super Mario World they are 2-dimensional. The engine separates horizontal and vertical movement. There is gravity, that permanently accelerates the *Movables* (Mario, Koopas and mushrooms) downwards until a specified maximum speed is reached or a collision occurs. The movement process works as follows: Each timer tick (with a timer interval $t_t = 16ms$), the current velocity v gets manipulated by an acceleration value a_t which is adjusted to t_t. As well, the current location s gets manipulated by a velocity value v_t which also is adjusted to t_t. The calculations are given in the next definition.

Definition 5.1 (Velocity and location)**.**

$$v = \begin{cases} sign(v) \cdot v_{max} & \text{if } |v| \geqslant v_{max} \\ v + a \cdot \frac{t_t}{1000} & \text{else} \end{cases}$$

$$s = s + v \cdot \frac{t_t}{1000}$$

If Mario jumps (upwards), his vertical velocity gets subtracted by a specified jump speed value v_j: $v_y = v_y - v_j$.

5.2.2 Collision

Each *Collidable* object (Mario, Koopas, mushrooms, different types of blocks) defines a collision shape. This shapes are consistently kept simple as rectangles. To check if two objects are colliding, these rectangular shapes and the locations of the objects are used. Simply checking, whether the rectangles are overlapping would not suffice here, as the collision direction matters and needs to be determined as well. Usually, a collision means restoring the previous locations of the colliding objects. Without any refinement this would not allow Mario to walk as he permanently collides with the ground. Therefore a slightly more complex collision detection strategy is needed: Given the old and current location vectors of a *Movable* object m, namely s_{old} and s_{cur}, the process is split in horizontal and vertical collision detection. Both steps check, whether the rectangle of m overlaps with the rectangle of another object o. What changes is the collision location of m. For the horizontal check, the collision location is $s_{col_x} = \{s_{cur}.x, s_{old}.y\}$. This means, only the x-component of the current location is used to determine a collision in horizontal direction. For the vertical check, analogously, the collision location is $s_{col_y} = \{s_{old}.x, s_{cur}.y\}$. Note that the old location will always refer to the location that has been visited before a collision occured. Therefore, components of that old location can be used to determine collision in a given direction as only the x- or y-component of the current location can be responsible for a collision. The function collides$(m, o, direction)$ takes two object locations and the check direction ($HORIZONTAL$ or $VERTICAL$) as arguments and returns the collision direction ($NONE, LEFT, RIGHT, UP, DOWN$). To determine $LEFT$ or $RIGHT$ during the $HORIZONTAL$ collision check (if the rectangles are overlapping) simply compare the horizontal locations of both objects. If m is located left of o, the collision direction of m with o is $RIGHT$. The other collision directions are determined analogously. The physics engine then performs horizontal and vertical collision checks, calculating collides$(s_{col_x}, s_o, HORIZONTAL)$ and collides$(s_{col_y}, s_o, VERTICAL)$ separately. If the horizontal respectively vertical check returns something different from $NONE$, the old x- respectively y-component of m gets restored.

With this method, walking on ground or sliding down a wall is possible, as collision will occur in only one direction, such that the *Movable* object can continue moving in the other direction.

5.3 Game Creation

In this section, the high-level steps to create a game world will be explained. It starts with calculating and providing space large enough to build a game world upon. As this worlds tend to turn out large, a data structure has been implemented to split and keep them small and manageable. Having that, gadgets (based on templates) can be inserted at calculated locations. As the gadgets have been placed they need to be connected by paths. In fact, the paths are created as continuous lines before the gadgets get inserted upon them. This removes the effort of creating multiple small paths, connecting the gadgets with

each other.

5.3.1 Providing Space

Given the structure of an (NP- or PSPACE-) framework, a formula (with its number of variables and clauses) and the real sizes of each gadget, the size of a game world $size_{game}$ can be calculated. All sizes are 2-dimensional and measured in *Blocks*. The game engine does not use one large array of *Blocks* but a hash-map of *MapParts*. Each *MapPart* has a constant size and is assigned an index that describes its location on a higher-level grid. A *MapPart* can be accessed via the hash-map given its index. The number of *MapParts* is calculated as follows:

$$num_{MapParts} = \left(\left\lceil \frac{size_{game}.width}{size_{MapPart}.width} \right\rceil, \left\lceil \frac{size_{game}.height}{size_{MapPart}.height} \right\rceil \right)$$

The advantage of splitting the game world into *MapParts* is the reduced (to constant) effort on collision handling and rendering, independent of the input size. The region of interest is reduced to the *MapPart*, currently visited by an actor (Mario or Koopa) and its eight neighboring *MapParts*. If the actor reaches the border of a *MapPart*, the next one in the according direction has to be checked. To keep it simple, all neighbors are taken into account. The size of the *MapParts* is determined by the screen and block sizes (those sizes in pixels), such that in each case rendering nine *MapParts* (one in the center and eight neighbors) suffices to fill the whole screen without introducing unrendered borders when moving and scrolling the game world.

5.3.2 Creating Paths

Based on the structure of the framework (NP or PSPACE) and the variables and clauses used in the formula, the locations of the variable / quantifier and clause gadgets can be determined. Instead of placing those gadgets, this creation step uses that locations to set up the paths before the gadgets. As already mentioned, placing gadgets upon existing, continuous paths is easier than creating multiple short paths in between those gadget. Note that the game world consists of *BlockHolders* that contain *Blocks* of different *BlockTypes*. This allows to simply replace an existing *Block* with another one. In fact, this only matters when dealing with crossover gadgets. As the locations of that crossover gadgets can be calculated in advance, the paths only need to know these vertical and horizontal locations to be created accordingly.

The path creation is done on a separate grid that represents a high-level view of the game world: As the path-tiles have a specific size different from 1×1 (6×6 *Blocks* in the current implementation) which complicates the path creation, the size of a path-tile is reduced to 1×1 in that high-level grid (called *map* in the following source code). This grid is filled with markers, denoting where the path goes. This is done in such a way that the number of direction changes is minimized. So each path connecting two gadgets takes at most three straight lines (with two changes). Note that crossover gadgets are

not considered during path creation. When two gadgets are connected in the high-level grid, the real path has to be created. As there are different path-tiles needed (bottom-left corner, bottom-right corner, horizontal, vertical-fall, vertical-climb,...), the information of which tile to use has to be read off of the high-level grid. This is done by checking the neighboring indices for markers. Further, to determine if a vertical path goes up or down (climb or fall), these tiles are specially marked in the grid. A path heads upwards if the next point to be connected by the path lies above the current point. Based on the grid, a neighbor sum is calculated as follows: If a neighboring tile is not empty, a value is added to the sum ($UP = 1, RIGHT = 2, DOWN = 4, LEFT = 8$). Further, if the current tile belongs to an upwards path, 16 is added. With this sum the correct path-tile can be chosen and created in the actual game world.

To make continuous paths with path-tile-sizes of $n \times n$ possible, the size of each gadget $size_{gadget}$ needs to be a multiple of n, so $size_{gadget}.width \mod n = 0 \wedge size_{gadget}.height \mod n = 0$.

Source code The following piece of Java source code (Figures 5.1 and 5.2) is demonstrating the path creation in the actual program. It is cut down to only show the important regions of code. The method *createPaths*() works on a specified list of lists of points to be connected with each other. In some cases, multiple paths (each described by a list of points) have to join each other, so there may not be any walls at the joining locations. This only works when all those subpaths are treated as parts of the currently created path. Therefore, a list of lists of points is needed. This information is provided by the algorithm in Figure 4.11. The method *connectPoints*() fills *map* (the high-level grid) with information about which parts of the game world the currently created subpath (described by the list of points) will occupy. The method *calcNeighborhood*() calculates, which neighboring tiles of the game world need to be accessible from the current tile (described by x and y), using the information provided by *map*. The returned neighborhood value is used to choose the proper, predefined path tile which gets inserted at the according location in the game world.

5.3.3 Placing Map Templates

Given the locations of gadgets and path-tiles, *MapTemplates* are used to fill the *BlockHolders* of the *MapParts* with *Blocks*, according to a desired pattern. As each *Block* is defined by a *BlockType*, *MapTemplates* can be kept simple as arrays of *BlockTypes*. When creating a *Block*, its absolute location is calculated as location of the gadget plus index inside the *MapTemplate*. From this location (and the *size* of a *MapPart*) the *MapPart* containing the according *BlockHolder* and the index of that *BlockHolder* can be extracted as

Definition 5.2 (Calculating block holder to be used)**.**

$$mapPart_{index} = \left(\left\lfloor \frac{loc.x}{size.width} \right\rfloor, \left\lfloor \frac{loc.y}{size.height} \right\rfloor \right)$$
$$blockHolder_{index} = (loc.x \bmod size.width, loc.y \bmod size.height).$$

```
private void createPaths(List<List<Point>> locationList){
    Dimension path = Constants.PATH_SIZE;
    Dimension size = new Dimension(gameSize.width/path.width +
        1, gameSize.height/path.height + 1);
    int map[][] = new int [size.width][size.height];
    for(List<Point> locations : locationList)
        connectPoints(map, locations);

    //insert map templates according to neighborhood (which
       sides need to connect to each other)
    for(int i = 0; i < size.width; i++)
        for(int j = 0; j < size.height; j++)
            if(map[i][j] != 0)
                insertMapTemplate(i * path.width, j *
                    path.height,
                    MapLoader.mapMapping.get("path" +
                    calcNeighborhood(map, size, i, j)));
}

private void connectPoints(int map[][], List<Point> points){
    if(points == null || points.isEmpty())
        return;

    Point pos1 = points.get(0), pos2;
    int path = Constants.PATH_SIZE.width;

    for(int l = 1; l < points.size(); l++){
        pos2 = points.get(l);

        for(int i = Math.min(pos1.x, pos2.x) / path; i <=
            Math.max(pos1.x, pos2.x) / path; i++)
            for(int j = Math.min(pos1.y, pos2.y) / path; j <=
                Math.max(pos1.y, pos2.y) / path; j++){
                if(map[i][j] == 0){
                    map[i][j] = (pos1.y > pos2.y ? 2 : 1);
                        //upward paths marked 2
                }else{
                    map[i][j] = Math.max(map[i][j], (pos1.y >
                        pos2.y ? 2 : 1));
                }
            }

        pos1 = pos2;
    }
}
```

Figure 5.1: Source code of path creation

```
private int calcNeighborhood(int map[][], Dimension size, int
    x, int y){
     /* up: 1
      * right: 2
      * down: 4
      * left: 8
      * if current block is marked as upward path: 16
      */
     int sum = 0;
     if(y > 0 && map[x][y-1] != 0)
         sum += 1;
     if(x < size.width-1 && map[x+1][y] != 0)
         sum += 2;
     if(y < size.height-1 && map[x][y+1] != 0)
         sum += 4;
     if(x > 0 && map[x-1][y] != 0)
         sum += 8;

     if(map[x][y] == 2)
         sum += 16;

     return sum;
}
```

Figure 5.2: Source code of path creation

Note that a *MapTemplate* may cover multiple *MapParts* which does not introduce problems as each location is calculated separately.

The *MapTemplates* are read from *.map*-files as the program is started. Those *.map*-files look as follows. Figure 5.3 shows an example for the upwards path-tile. Comment lines begin with the # sign. The line starting with : stores template information for width, height and name. In the following lines, the *w*s are denoting *BlockType* 'wall'.

5.3.4 Rendering

On every timer tick, the current state of the game is rendered on the screen. The *Blocks* in the *BlockHolders* contained in the *MapPart* currently visited by Mario and its eight neighboring *MapParts* are taken into account. The *BlockType* of each *Block* determines which *Slideshow* to be chosen. This *Slideshow* is accessed via a hash-map given the *BlockType*. A *Slideshow* consists of a list of images, the animation length and the current time value (with respect to the animation length). On each timer tick, the current time of each *Slideshow* is increased by the timer interval. As the current time reaches the animation length, the time is kept within bounds by $time = time \bmod animLength$. This produces a continuous flow of images. Given the number of images of a *Slideshow* and its $progress = \frac{time}{animLength}$, the current image to be rendered is determined. Rendering is done *MapPart*- and *BlockHolder*-

```
# defining line begins with ':'
# and ends with information
# about 'width height name'
#
# up: 1
# right: 2
# down: 4
# left: 8
# marked as upwards path: 16
#

: 6 6 path21

w . . . . w
w . . . . w
w . . . w w
w . . . . w
w . . . . w
w w . . . w
```

Figure 5.3: Example path-tile

wise from top left to bottom right. This way, the illusion of depth can be created as higher elements on the front will cover (parts of) elements on the back. Games with a top-down (isometric) view like Legend of Zelda can use that effect, though Super Mario World (with its side view) does not benefit from that.

6 Usage and Installation

This chapter shows how to use the program. Section 6.1 explains the GUI with the structure of its frames. In Section 6.2 the controls show how to interact with Mario and the game. Section 6.3 concludes this chapter with the installation requirements of the program.

6.1 GUI

The major part of this section has been copied from the author's bachelor thesis [8], which already explained the GUI part as it was dealing with the implementation of the NP-framework. A few things had to be added or changed in order to implement the PSPACE-framework. The GUI mainly consists of three frames: $FormulaEnterFrame$, $HelpFrame$ and $GameFrame$.

FormulaEnterFrame

When the program starts, this frame is shown. It serves as the starting point to use the progam's functionality. The user can specify an arbitrary (quantified) propositional logic formula in the text area. The following operators can be used:

- Quantifiers:
  - $\forall$: A
  - $\exists$: E
- Binary operators:
  - $\oplus$: ^
  - $\rightarrow$: $>$
  - $\vee$: $+, |$
  - $\wedge$: $*, \&$
- Unary operators:
  - $\neg$: $-, !$

Before the game can be generated, the formula has to be parsed, by pressing the $Parse$ button. If no problems occur, the $Parse$ button disappears and the $Startgame$ button gets visible. By pressing that button, the $FormulaEnterFrame$ disappears, the $GameFrame$ is shown and the game starts. If the parser detects syntactically incorrect input or tautologies, some information about that is shown in the GUI. The user can define SAT or QSAT formulas. If the input

starts with a prefix, denoting the quantifiers of the variables used in the following matrix part, it defines a QSAT formula and the PSPACE-framework gets applied. On the top of the frame, a menu bar provides the following options:

- *File*
  - *Open*: Opens a file chooser which allows to load DIMACS *.cnf* (SAT and QSAT) and BoolTool [9] *.frm* files.
  - *End*: The program closes.
- *Mode*
  - *Normal*: During the game, the formula (with its matrix part) transformed to CNF will be shown. This means the player can directly see a nicely structured formula from which the game world has been created. This gives the player an overview of which clauses have already been satisfied and which not. If the NP-framework is used to represent the given formula, the player gets the opportunity to reset to previously visited variable gadgets in the game.
  - *Hardcore*: During the game the original formula will be shown and the player cannot reset to earlier variable gadgets. Resetting the game to the start gadget is still possible though. In addition, to avoid the possibility to reconstruct the CNF formula by visiting clause gadgets, the literal names have been removed from the clause gadget entrances. Solving a formula without seeing the CNF transformation can be a real challenge if exclusive OR ($\oplus$) operators are used.
- *Help*
  - *Show help*: The $HelpFrame$ is shown. The $FormulaEnterFrame$ will not disappear.
- <language icon>
  - *DE*: Select German language.
  - *EN*: Select English language.

Via the GUI the user can only enter single formulas. In order to let the user specify (multiple) formulas and reuse them arbitrarily, the use of DIMACS *.cnf* and BoolTool *.frm* files is supported. DIMACS *.cnf* files are widely spread and can be considered as standard format for CNF formulas. By adding a quantifier section to the beginning of the file, QBF formulas can be declared as well. The type of those files changes to QDIMACS, although the file extension stays *.cnf*. Within BoolTool *.frm* files multiple CNF formulas can be defined per line, starting with the keyword *frm*. The files may also contain comment lines, defined by the keyword *c*. The program allows to choose and open multiple files at once. All formulas in all opened files are parsed and a series of game worlds is created, one playable after another. If a file could not be parsed, the file is ignored and the user gets notified about that. The selection of multiple files is not restricted to one type: *.cnf* and *.frm* files may be mixed. Files are read in the order they are displayed in the file chooser window, not in the order of

selection. Later formulas are accessible only if all previous formulas could be solved. This allows a user to provide a series of formulas of increasing difficulty.

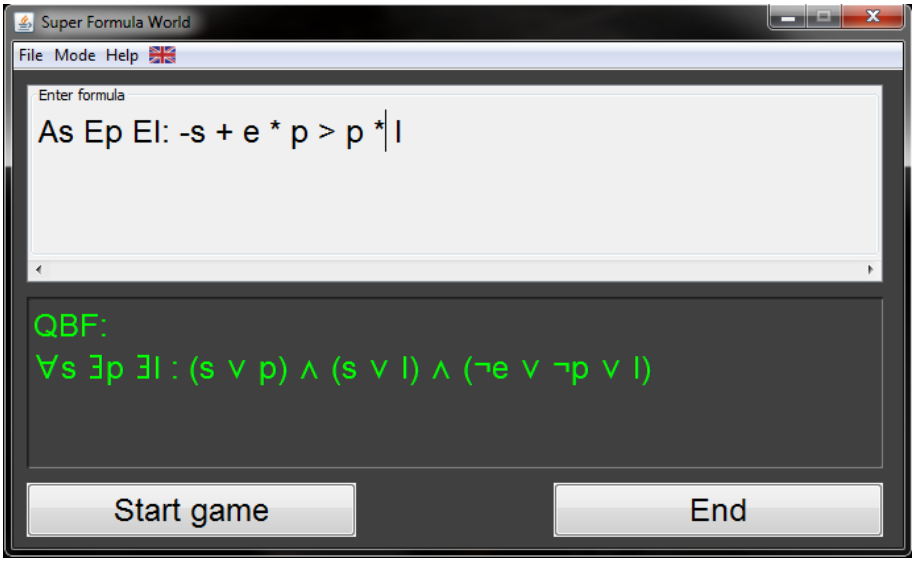

Figure 6.1: FormulaEnterFrame

HelpFrame

On the left side buttons provide help on different topics:

- *General*: General information about the program and its purpose is displayed.
- *Mechanics*: The mechanics of the game are described. What can the player do? How can that be achieved?
- *Formula*: The required format of formulas, as well as a description of how the parser works will be displayed.

The help description to those categories is displayed in the label right to the buttons. On the bottom left of the frame, the *Back* button closes the help frame.

GameFrame

In this frame, the game will be rendered and the current state of the formula and game information is displayed. The major part of the *GameFrame* is used to render the game. At the top, the currently passed time and score is displayed, while at the bottom the formula in its chosen representation (CNF or original input) is shown. The state of the formula is highlighted using colors.

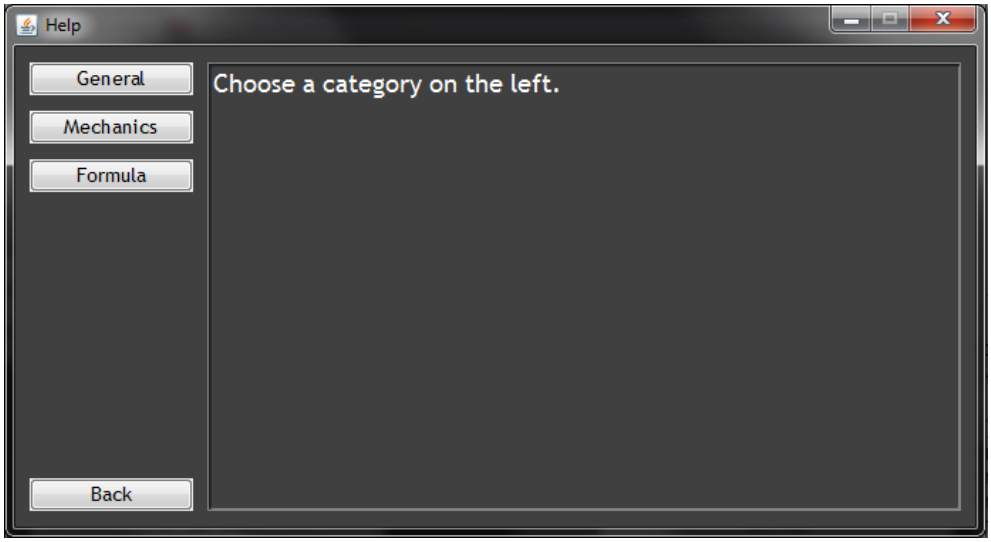

Figure 6.2: Help frame

NP-framework CNF formulas are displayed clause-wise: If a literal satisfies the clause, the whole clause is replaced by a green T. If on the other hand a variable decision makes a literal of the clause unsatisfiable, this is signalized by a red F. If every literal of the clause is unsatisfiable, the whole clause gets unsatisfiable and is replaced by a single red F. If a game starts in hardcore mode, the original formula is displayed. This formula probably does not consist of CNF clauses and therefore has to be treated in a different way. As a variable gets assigned, every occurrence of that variable gets replaced by a green T or red F in the formula, depending on its assignment. In hardcore mode it is the player's task to find out whether the chosen assignments satisfy the formula.

PSPACE-framework As the state of a literal changes by opening or closing its door, that literal gets colored green or red, depending on its state. This does not affect the state of the same literal in other clauses. A clause itself is not marked as T or F. This is due to the recursive behavior of the PSPACE-framework: If the clauses got replaced by their states, the player would not see the formula during the next assignment process in order to make decisions. So the player would have to memorize the formula before actually playing the game. As this is not intended, the PSPACE-framework differs in its representation of the state of the formula from the NP-framework.

GameOverPanel If Mario finishes a game world or dies, the *GameOverPanel* is shown within the *GameFrame*. The player gets the following options:

- *Next Level*: The game world representing the next formula in the series is created. This option is only available if a series of formulas has been read from files, the current formula is not the last one and the finish flag of the current level has been reached.

- *Retry*: The current level resets.
- *End*: The *GameFrame* disappears and the *FormulaEnterFrame* is shown again. Formulas and game worlds get deleted.

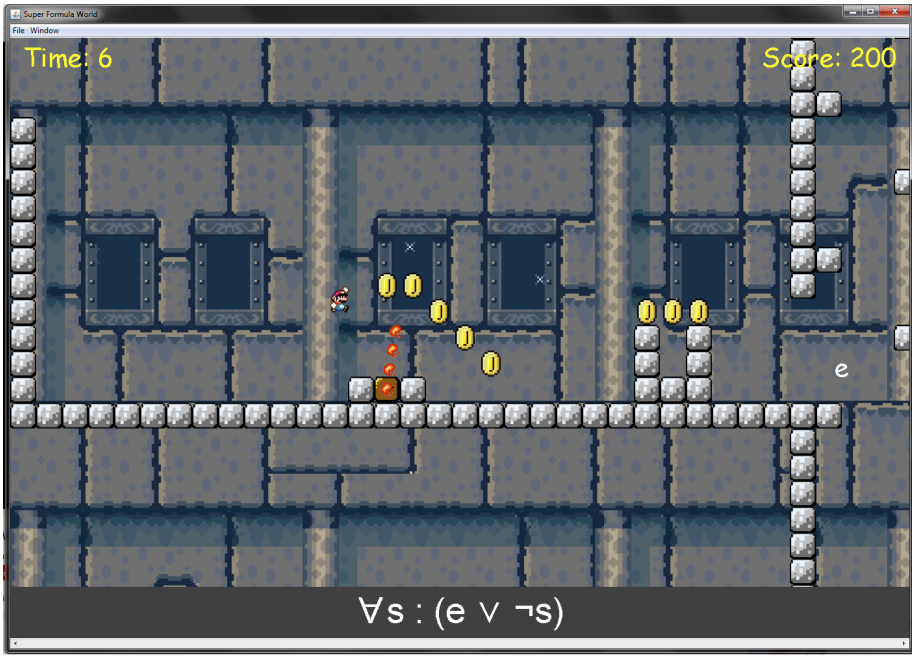

Figure 6.3: Game frame

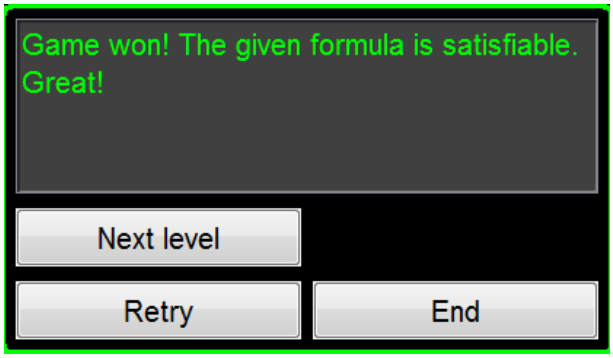

Figure 6.4: Game over options

On the very top of the frame a menu bar provides the following options:

- *File*
  - *Restart*: The current game is reset. More specifically, all recognized actions of the player are undone.
  - *End*: The *GameFrame* disappears and the *FormulaEnterFrame* is shown again. Formulas and game worlds get deleted.

- *Undo*: The player gets the opportunity to reset the game to an earlier state. For every entered variable gadget, a menu item is added. Note that entering quantifier gadget of the PSPACE-framework will not affect the menu.
  - *Undo until: start*: Mario resets to the start gadget. This equals *File/Restart*.
  - *Undo until: x*: Mario is reset to variable gadget x. All recorded actions performed after that gadget was entered will be undone.
- *Window*
  - *Fullscreen*: If this option is checked, the game turns into full-screen mode. If it is unchecked, the old size before full-screen mode gets restored.

The effect of the operating system maximize button has been disabled. The size of the frame can be changed by the player, providing a minimum size of 800 x 600 pixels.

6.2 Controls

As the game has started, the player can control Mario and the game in the following ways:

- A: Move left
- D: Move right
- W: Climb up vine
- S: Crouch / climb down vine
- F: Grab / drop trampoline
- *Space*: Jump
- Esc: Quit game and return to $FormulaEnterFrame$; if in $FormulaEnterFrame$, close program
- M: Show / hide minimap
- $\leftarrow, \rightarrow, \uparrow, \downarrow$: Navigate in minimap
- ,: Decrease minimap transparency
- .: Increase minimap transparency
- $1 - 9$: Set slow / fast motion factor to predefined values
- $+/-$: Increase / decrease slowmotion factor by a small value
- 0: Reset slowmotion factor to 1

In addition, for testing reasons and lazy or clumsy players, the game features some cheats:

- $F6$: Turn god-mode on / off. This prevents Mario from getting hurt or killed.
- $F7$: Turn multi-jump on / off. This allows Mario to repeatedly jump while in the air.
- $F8$: Turn collision on / off. This allows Mario to pass walls and enemies without being harmed.

Players that reached finish locations using cheats should be able to judge for themselves, whether the given formula is satisfiable or not!

6.3 Installation

The program has been written in Java 1.6, so to run it the Java Runtime Environment (JRE) 1.6 or higher has to be installed on the computer. The program can be started by simply double-clicking the Java Runnable (.jar) file or via the console using "*java -jar Mario-v2.1run.jar*".

7 Related work

Over the years, researchers came up with several articles that deal with NP- or PSPACE- complexity proofs for different games. Some of them are following completely new approaches, while some of them are similar to or based on others. This chapter will exemplarily mention some of these articles.

In [6] (first version released in 1996), Dor and Zwick have shown PSPACE-completeness for an alternative version of Sokoban[1] and NP-hardness for the original version. Their alternative version of Sokoban differs from the original as blocks may be of size 1×2. Further, the player can push two blocks at once and pull one block backwards. Using this "mightier" version, they were able to construct several gadgets which they used to model Turing machine programs to further simulate a linear bounded automaton. The decision problem of reaching a defined accepting state on the automaton, given an initial configuration is proven to be PSPACE-complete. What remained an open question was to prove whether the original version of Sokoban is PSPACE-complete. This has been done in 1997 by Culberson in [5]. He introduced a few more and complex gadgets wich were able to handle the original Sokoban game with 1×1-sized blocks and a player that can only push one block and cannot pull blocks. Given these gadgets, more complex Turing machine cells have been constructed. Again, the decision problem of reaching a defined final dell on the automaton, given an initial configuration is proven to be PSPACE-complete.

In [7], Minesweeper[2] is shown to be NP-complete by Kaye. He is simulating Boolean circuits by modeling and joining the logical gates of the circuit. These gates are built as Minesweeper configurations which allow to logically conclude the rest of the gate's mines after making the decision, whether the represented Boolean value is $true$ or $false$. The decisions propagate through the game world towards the designated terminated wire. If and only if the Boolean value delivered to this terminated wire is $true$, the configuration is called consistent. Given an instance of SAT, this framework can represent and evaluate the formula (as decisions have been made). So SAT can be reduced to the introduced Minesweeper framework. It follows from its contruction that the framework itself is in NP. Combining both results, Minesweeper is NP-complete.

A more recent article by Walsh [12] proves NP-hardness for the popular game Candy Crush[3]. He is following an approach of a changing game world, as multiple neighboring blocks are deleted as they have formed a row of at least three blocks. The blocks above the deleted ones are falling down to fill the gaps. This changing behavior is used to construct game worlds in advance that

[1] http://en.wikipedia.org/w/index.php?title=Sokoban&oldid=657401744

[2] https://en.wikipedia.org/w/index.php?title=Minesweeper_%28video_game%29&oldid=676277080

[3] http://en.wikipedia.org/w/index.php?title=Candy_Crush_Saga&oldid=665762542

will change upon performing a move such that the concluding configuration allows further moves or leads to recursive behavior where new rows of blocks get formed are directly deleted. The introduced framework is reducing from 3-SAT, which is NP-complete. Therefore, the framework is NP-hard and as membership in NP is shown as well, the framework is NP-complete. Given a game world, the number of deleted rows is counted and if a specified value is reached, the instance of 3-SAT is satisfiable.

In [11], Viglietta shows complexity results for several games (Pac Man[4], Prince of Persia[5] and Starcraft[6] to be mentioned). These results are based on several metatheorems that had to be elaborated in advance. Some of these metatheorems built the basis for parts of the work of Aloupis et al. In particular, the special gadgets of the PSPACE-framework have already been introduced in Viglietta's article. He further shows the mechanisms of location traversal and opening doors by using pressure plates and discusses the associated impacts on complexity of different application possibilities of those mechanisms: Using these, games like Prince of Persia can be shown to be PSPACE-complete.

Finally, the article [4] on which this thesis is based on is going to be mentioned once more. Aloupis et al. have introduced frameworks to prove hardness of video games. They are focussing on classic platform Nintendo games and giving complexity results for Super Mario, Donkey Kong, Legend of Zelda, Metroid and Pokemon games. The frameworks are able to categorize specific games to be NP- resp. PSPACE-hard. Both frameworks are consisting of multiple components, so called gadgets. If one game's mechanics do allow to implement the functionality of each of those gadgets, then that game is hard for NP or PSPACE respectively. Unfortunately, two of the introduced gadget implementations have not been flawless in earlier versions of the article. One has been fixed by an elegant new version of the gadget, the other one still contains a subtle flaw (that could easily be fixed by making a minor change). The frameworks are generating game worlds based on logic formulas. If and only if the game can be beaten, the formula is satisfiable. It has to be noted, that even though the introduced PSPACE-framework is based on a framework of Viglietta in [11], they still differ as Viglietta assumes remote-controlled doors while Aloupis et al. have to actually visit doors to operate them locally, which makes the whole concept applicable to a wider range of games. This difference also causes the PSPACE-hardness result for Super Mario World (shown in this thesis) to not follow from the metatheorems in [11].

During the work on this thesis, while implementing the game, a problem with the door gadget arose: At this point, the requirement that actions to change the state of doors have to be local has not been postulated yet. A test player figured out that the trampoline of a door gadget could be carried outside the door gadget and used later to make the traverse path of another door traversable, which broke the framework. Therefore the requirement was introduced and the problem was solved. As the framework relates to the work

[4] http://en.wikipedia.org/w/index.php?title=Pac-Man&oldid=665867450
[5] http://en.wikipedia.org/w/index.php?title=Prince_of_Persia&oldid=663518042
[6] http://en.wikipedia.org/w/index.php?title=StarCraft&oldid=666330438

of Viglietta in [11], it was of interest which impact this problem has on the original metatheorems. As it turned out that the framework of Viglietta differs in exactly that aspect (door operation) from the framework of Aloupis et al, Viglietta does not need to deal with that problem as it cannot occur with his remote-controlled doors: He is not required to explicitly visit doors and perform actions to change their states (which introduced the problem).

8 Conclusion

This thesis is based on the work of Aloupis et al. [2, 3, 4] whose purpose is to show complexity results for classic Nintendo platform games. In order to do so, Aloupis et al. are introducing hardness frameworks for both NP and PSPACE. Those frameworks have to provide some kind of functionality which is implemented by several types of gadgets. If the mechanics of a given game allow to realize the functionality of each gadget, that game is hard for NP or PSPACE respectively. If further membership of the game for that complexity class can be shown, the game is complete for NP or PSPACE respectively.

This thesis aims at a detailed and formal definition of the frameworks. This is necessary in order to avoid unintended behavior of some gadgets. In particular, there have been flaws in the original crossover gadgets for Super Mario and Metroid games. Those flaws came from a too shallow formal definition of the crossover gadget. By formalizing crossovers and introducing leakage constraints (for the NP-framework), this flaws get avoided. The NP- resp. PSPACE-framework can be reduced to the SAT resp. QSAT problem. These reductions are shown to be log-space computable. Further, both frameworks are shown to behave correctly, in particular, a game can be finished if and only if the corresponding SAT or QSAT formula is satisfiable.

Aloupis et al. are providing implementations of their frameworks and therefore complexity results for several games in [2,3,4]. To complement these results, this thesis shows generalized Super Mario World to be PSPACE-complete.

Both frameworks are implemented as a Java program. As generalized Super Mario World is applicable for both NP- and PSPACE-frameworks, this game can be used to implement both frameworks. This allows to reuse the game mechanics and some gadgets. This thesis gives a high-level description of the game engine. Further the GUI is explained and an example run shows how to use the program from entering a formula to finishing a game world.

Besides the possibility of using this program to show satisfiability of given SAT or QSAT formulas, the main purpose of it is to serve as a learning software. It should wake the user's interest in satisfiability and logic in general in a funny way by playing a game. It could be used in computer science class in school. The author would be happy if a single child finds its way to computer science because of this game.

Bibliography

[1] Greg Aloupis, Erik D. Demaine, and Alan Guo. Classic Nintendo Games are (NP-)Hard. *CoRR*, abs/1203.1895, 2012.

[2] Greg Aloupis, Erik D. Demaine, and Alan Guo. Classic Nintendo Games are (Computationally) Hard. *CoRR*, abs/1203.1895, 2014.

[3] Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic Nintendo Games are (Computationally) Hard. In *Fun with Algorithms - 7th International Conference, FUN 2014, Lipari Island, Sicily, Italy, July 1-3, 2014. Proceedings*, pages 40–51, 2014.

[4] Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic Nintendo Games are (Computationally) Hard. *Theor. Comput. Sci.*, 586:135–160, 2015.

[5] Joseph C. Culberson. Sokoban is PSPACE-complete, 1997.

[6] Dorit Dor and Uri Zwick. SOKOBAN and other motion planning problems. *Comput. Geom.*, 13(4):215–228, 1999.

[7] Richard Kaye. Minesweeper is NP-complete. *The Mathematical Intelligencer 22*, nr. 2:9–15, 2000.

[8] Josef Lindsberger. Classic Nintendo Games are Hard. Technical report, University of Innsbruck, 2013.

[9] Markus Plattner. BoolTool goes OCaml. Technical report, University of Innsbruck, 2007.

[10] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.

[11] Giovanni Viglietta. Gaming is a hard job, but someone has to do it! *CoRR*, abs/1201.4995, 2012.

[12] Toby Walsh. Candy Crush is NP-hard. *CoRR*, abs/1403.1911, 2014.