VERIFYING MAZE-LIKE GAME LEVELS WITH MODEL CHECKER SPIN

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS OF
THE MIDDLE EAST TECHNICAL UNIVERSITY
BY

ONUR TEKİK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF INFORMATION SYSTEMS

NOVEMBER 2021

Approval of the thesis:

VERIFYING MAZE-LIKE GAME LEVELS WITH MODEL CHECKER SPIN

Submitted by ONUR TEKİK in partial fulfillment of the requirements for the degree of **Master of Science in Information Systems Department, Middle East Technical University** by,

Prof. Dr. Deniz Zeyrek Bozşahin Dean, **Graduate School of Informatics**	________________
Prof. Dr. Sevgi Özkan Yıldırım Head of Department, **Information Systems**	________________
Assoc. Prof. Dr. Aysu Betin Can Supervisor, **Information Systems, METU**	________________
Assist. Prof. Dr. Elif Sürer Co-Supervisor, **Modeling and Simulation, METU**	________________

Examining Committee Members:

Assoc. Prof. Dr. P. Erhan Eren Information Systems, METU	________________
Assoc. Prof. Dr. Aysu Betin Can Information Systems, METU	________________
Assist. Prof. Dr. Elif Sürer Modeling and Simulation, METU	________________
Assist. Prof. Dr. Burkay Genç Computer Engineering Dept., Hacettepe University	________________
Assoc. Prof. Dr. Banu Günel Kılıç Information Systems, METU	________________

Date: _*November 5th, 2021*

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name: Onur Tekik

Signature :

ABSTRACT

VERIFYING MAZE-LIKE GAME LEVELS WITH MODEL CHECKER SPIN

Tekik, Onur
MSc., Department of Information Systems
Advisor: Assoc. Prof. Dr. Aysu Betin Can
Co-Advisor: Assist. Prof. Dr. Elif Sürer

November 2021, 82 pages

In this thesis, we present a new methodology that includes procedural generation and verification of maze-like game levels. The methodology employs a model checker, called SPIN, to both produce a winning sequence of actions and to formally verify custom game design properties. To verify a game level, we propose automated tailoring on template game models, considering the level-in-test, specifically designed according to the game rules. By leveraging the counterexample generation feature of SPIN, we find one or more solutions to the level-in-test and use PyVGDL to animate the solutions.

To show this methodology's effectiveness, we conducted five different experiments. These experiments include performance comparisons in level solving between the proposed and existing methodologies —A* Search and Monte Carlo Tree Search— and demonstrations of the proposed approach's usage to verify a game level with respect to existing requirements. The work also includes a pipeline for the generation of maze-like puzzle levels that includes two levels of cellular automata.

Keywords: Procedural content generation (PCG), model checking, formal verification, Video Game Description Language (VGDL), Puzzle games

ÖZ

MODEL KONTROLCÜSÜ SPIN İLE LABİRENT BENZERİ OYUN SEVİYELERİNİ DOĞRULAMAK

Tekik, Onur
Yüksek Lisans, Bilişim Sistemleri Bölümü
Tez Danışmanı: Doç. Dr. Aysu Betin Can
Yrd. Tez Danışmanı: Dr. Öğr. Üyesi Elif Sürer

Kasım 2021, 82 sayfa

Bu tezde, labirent benzeri oyun seviyelerinin yöntemsel üretimi ve doğrulamasını içeren yeni bir metodoloji sunmaktayız. Bu metodoloji hem kazanan eylemler dizisini üretmek hem de özel oyun tasarım özellikleri doğrulamak için kullanılan SPIN isimli bir model kontrolcüsünü içermektedir. Bir oyun seviyesini doğrulamak için oyun kurallarına göre özel tasarlanmış oyun modeli şablonlarının teste tabii olan seviyeye göre otomatik düzenlenmesini öneriyoruz. SPIN'in karşı örnek üretme özelliğini kullanarak, teste tabii olan seviyeye bir ya da daha fazla çözüm bulup, PyVGDL'i kullanarak bu çözümleri canlandırmaktayız.

Bu metodolojinin etkenliğini göstermek için beş farklı deney düzenlenmiştir. Bu deneyler önerilen metodolojinin var olan metodolojilerle —A* Arama Algoritması ve Monte Carlo Ağaç Araması— performans kıyaslarını ve önerilen metodolojinin oyun seviyelerinin var olan gereksinimlere göre doğrulanmasında kullanımını gösterimini içermektedir. Tezde ayrıca labirent benzeri bulmaca seviyelerinin üretimi için kullanılabilecek iki seviye hücresel otomat içeren bir hat bulunmaktadır.

Anahtar Sözcükler: Yöntemsel içerik üretimi, model kontrolü, formel doğrulama, Video Oyunu Tanımlama Dili (VGDL), bulmaca oyunları

To My Family

ACKNOWLEDGEMENTS

First of all, I would like to express my gratitude to my advisor and my co-advisor, Dr. Aysu Betin Can, and Dr. Elif Sürer for their patience and the constant guidance in my research. Their guidance and knowledge enlightened my way in my darkest times. They were always open to me, offering constant constructive criticism and suggestions.

Besides my advisors, I would like to thank the rest of my thesis committee for their insightful comments and suggestions.

Also, I would like to thank my parents Abdullah and Havva Tekik, for raising me and supporting me in my journey of life. Thanks to them, I have a steady moral compass and the will to work harder always.

Last but not least, I'm grateful to my fiancé İlknur for bearing with me all the time and raising my spirit. She was always supportive of me, and I cannot dream myself handling this work without her.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CA	Cellular Automata
CentOS	Community Enterprise Operating System
CPU	Central Processor Unit
CTL	Computational Tree Logic
GHz	Giga Hertz
GVG-AI	General Video Game Playing Artificial Intelligence
LTL	Linear Temporal Logic
MCTS	Monte Carlo Tree Search
NP	Non-deterministic Polynomial-time
NPC	Non-player Character
PROMELA	Process (or Protocol) Meta Language
PSPACE	Polynomial-space
PyVGDL	Python implementation of Video Game Description Language
RAM	Random Access Memory
VGDL	Video Game Description Language

CHAPTER 1

INTRODUCTION

Procedural generation of video game levels, which is the automated process of creating game levels algorithmically, has been an essential topic in gaming research for decades [1]. Part of this importance comes from the goal of having unlimited content for games. However, having unlimited content has its downsides. One of them is the resources needed for verifying whether a level fits a specific requirement. Like the generation of the game levels, verifying them also requires computational or labor resources.

A common practice for verifying whether a game level fits specific requirements is called "Playtesting," which may include intensive human labor or computational time since this process is not automated. It can be semi-automated [2], or it is done via memory and CPU-intensive methods like Monte Carlo Tree Search (MCTS) or evolutionary algorithms [3], [4]. This thesis proposes another method for verifying maze-like game levels, which is an automated and model-based approach. Despite the advancements in the research about formal verification, its adoption in game development still comes short. The main reason for the absence of model-checking in game development is the requirement of mathematical background to properly leverage such tools [5].

Another practice used to verify a game is having automated tests that will run on the level-in-test. Since a test suite cannot cover all possible executions, automated tests can only be used to find a few unfit levels or bugs. However, a model-checking approach may prove the absence of any flaws that will result in an unfit level with respect to a specific requirement. Therefore, a game level can be verified not for a specific subset of executions but for all of them.

A puzzle game is a system that has a set of rules that can be rendered into a behavioral model in the player's perspective, and an initial configuration, which in this case is a game level; and game-ending or game-breaking conditions that can be specified in a linear temporal logic (LTL) formula. All those properties make puzzle games possible to be modeled and formally verified with the help of model-checking software. Building from this, as our first and foremost contribution, we are proposing four models for three different maze-like puzzle games. These three games are a regular maze-solving game, a maze-solving game that includes an opponent to be played against, and Sokoban. The models proposed are configurable with respect to the level to be verified. The verification is done by implementing four different template models in PROMELA, the input language of our model checker choice. After configuring these template models with respect to the level-in-test, we use the SPIN model-checker to generate an example solution to the level-in-test. Also, another contribution is a framework to generate and/or verify game levels. The framework can generate maze-like puzzle levels for three games that we have focused on, configuring template game model files in PROMELA language, using the model-checker tool SPIN to generate a sequence of actions that is evidence that the game possesses the property, and demonstrating those actions on screen, using PyVGDL.

We have developed four different models to run for verifying three different games. These four models are used in five experiments to show how the proposed approach may fit in the gaming industry and show how it can be used to verify game levels for two existing requirements from the gaming literature. In the first experiment, a model is run against the A* search path finding algorithm to be compared in a regular maze solving game. In the second experiment, the proposed approach was compared against Monte Carlo Tree Search (MCTS), a common approach used in game testing. The third experiment shows how a user can leverage trade-off decisions when designing her template model to gain an advantage. The last two experiments include two different requirements, one qualitative and one quantitative, and aims to show how a user may use the proposed approach to test their own complex requirements.

The main benefit created with this thesis is a framework presented to verify and create maze-like puzzle games. By using the work presented here, a game designer can verify a level designed by a human being with respect to any specified requirement. Although it is possible to find a bug or design mistake by automated tests, this work takes an extra step to verify the level and prove the absence of a mistake with respect to any complex requirement. The verification process is totally automated and can be placed in a procedural content generation pipeline. Also, the work includes a level generator that creates maze-like puzzle games with no complex algorithms.

This thesis makes the following contributions:

- Presents a methodology to verify game levels with respect to custom requirements formally.
- Presents a pipeline that includes procedural generation and verification of maze-like game levels.
- Presents a set of experiments that includes performance comparisons of the proposed methodology with respect to existing methodologies and use cases.

The rest of the thesis is structured as follows: In Section II, preliminary information on cellular automata, SPIN and PROMELA, and GVG-AI and PyVGDL are given. Section III includes a literature review. Section IV describes our proposed methodology. Section V presents the properties we focused on and gives the LTL formulae we used for these properties. The details about the experiments are in Section VI. Section VII presents and discusses the results of those experiments, and Section VIII concludes the thesis.

CHAPTER 2

PRELIMINARIES AND LITERATURE REVIEW

This chapter includes background information on preliminary material used frequently in this thesis and the status of the literature on the topic.

2.1. Background

This section includes subsections introducing preliminary material on cellular automata, SPIN, PROMELA, and GVG-AI, which are used or referenced in this work.

2.1.1. Cellular Automata

John von Neumann and Stanislaw Ulam originally proposed cellular automata as a mathematical model for self-reproducing organisms [6]. A cellular automaton includes a grid of cells, each in a state from a set of possible states. For each cell, there are some other cells considered as "neighbors." A cell's next state is calculated with a fixed rule, called a "state transition function," which often includes states of the cell itself and its neighbors'. Mostly, the state transition function is applied to all cells simultaneously.

Types of cellular automata used in this work are elementary cellular automata [7] and a life-like cellular automaton that can create maze-like levels from a random seed called "Maze" [8]. Elementary cellular automata are the simplest kind of one-dimensional cellular automata. States of the cells are binary, zero, and one, meaning there are only two states possible for a cell to be in. The cells are put in a line, and the next state of any cell depends on its two neighbor cells' state and its own. Since three binary inputs are affecting a binary output, there are limited variations of elementary cellular automata, $2^{2^\wedge 3}$ = 256 in total. Those rules are grouped into three different categories regarding the output pattern. These categories are simple, oscillating, and chaotic. Simple rules result in a stable end state, oscillating rules result in an ending pattern with few states that repeat stably, and chaotic rules result in a pattern that is partly stable or oscillating but unpredictable at the same time. "Maze" is a life-like automaton that can be used to generate a maze-like output. By maze-like, we mean that its output includes long and connected corridors that can be interpreted as a maze. The automaton works on the Moore neighborhood, which includes four main neighbors, four secondary neighbors (North, northeast, east, southeast, south, southwest, west, and northwest). The state transition rule of "Maze" is abbreviated as (B3/S12345), which can be explained as "If a cell is dead (floor), and exactly three of its Moore neighbors are alive (wall), the cell is born (turns into a wall). If a cell is alive (wall), and its 1-5 Moore neighbors are also alive, the cell survives (stays as a wall). Otherwise, it dies (turns into a floor)".

2.1.2. Model checking, linear temporal logic, SPIN, and PROMELA

Model checking is a computational technique used for automatically verifying concurrent systems that either are finite-state or have finite-state abstractions. It is done via submitting a model of the system-in-test to a model checker. Tools that implement algorithms that perform model checking are called model checkers. The requirement the system is verified against is submitted as a temporal logic formula. While checking the model, the model checker performs an exhaustive search on the state-space of the submitted model until it finds a specific state where the requirement submitted does not hold. If such a state is reached, the model checker returns the state transitions made until that point as a counterexample for the user to examine. If the exhaustive search ends without a single state breaking the requirement, the system gets verified.

Model checking has been used in critical systems for software verification for decades. Microsoft Research's SLAM project, a device driver verifier, has been used to find bugs in Windows Device Drivers. However, model-checking has a disadvantage like any other candidate to perform this job. This disadvantage is called the state-space explosion problem. While the number of state variables and concurrent processes in the system are increasing, the system's state space grows exponentially. This may result in failures due to the verifier system's lack of memory or computational power quickly. The state-space explosion problem has no solution found today. However, there exist numerous optimization techniques for tackling the state-space explosion problem. These techniques include both model-side and model-checker-side techniques. Model-side techniques include creating a more abstract system model to reduce state variables or processes, using compositional reasoning to make decision-making more local to the processes, and cone of influence reduction. Model-checker-side techniques include using binary decision diagrams to represent state transition systems more efficiently and employing partial order reduction to reduce the number of states that need to be enumerated.

Model checkers include ways to express requirements. One popular way of doing this is by using linear temporal logic (LTL). Linear temporal logic formulae can be used to encode facts in time, such as a condition will never or eventually be true or a a condition that can never be true unless some other is. An LTL formula consist of propositional variables, logic operators, and temporal operators. In Table 1, the most common temporal operators are presented.

A Büchi automaton is an automaton that accepts an infinite number of inputs. In order to check if a property holds in a model presented to it, SPIN looks for intersections between two different Büchi automata. One of those automata represents the model of the system, while the other represents the LTL formula the model-checker is given. Vardi and Wolper showed that any LTL formula can be translated into a Büchi automaton [41]. Using this conversion, LTL formula submitted to SPIN is conversed into a Büchi automaton. After, the model presented to SPIN is also converted into a Büchi automaton in a two-level conversion. First, the model is translated into a Kripke structure, which is a quadruple M = (S, R, I, L), where S is the finite set of states of the model, R is the relation of transition between states in S, I is the set of initial states, and L is a state labeling function. Afterwards, the Kripke structure extracted from the model is transformed into a Büchi automaton, that accepts infinite executions the system model can generate. Once two different Büchi automata is extracted from the inputs SPIN is given, SPIN compute the intersection between these two automata, and test the resulting automaton's emptiness. This intersection represents the executions that are possible in the model that are violating the property.

Table 1: Temporal operators used in linear temporal logic.[42]

Operator Name	Representation	Explanation
Next	**X** A	Property A has to hold at the next state.
Finally	<> A, **F** A	Property A has to hold some state until the state transitions end.
Globally	**G** A, **[]** A	Property A has to hold for all states from this point.
Until	A **U** B	A has to hold at least until B holds. B has to hold at a point in time.
Weak Until	A **W** B	A has to hold at least until B holds. B may or may not hold.
Release	A **R** B	B has to hold until and including the point where A holds, but this change may not happen.
Strong Release	A **M** B	B has to hold until and including the point where A holds, which has to happen in a point.

SPIN, a model-checker tool available for free, is a generic verification system that supports the design and verification of systems that include asynchronous processes [9]. The software was awarded the ACM System Software Award in 2002. SPIN employs partial order reduction to tackle the state-space explosion problem. The tool can conduct interactive simulations and generate C code optimized for model checking from a behavioral model specified in PROMELA language.

SPIN accepts correctness properties expressed in linear temporal logic (LTL). SPIN can check input behavioral models with respect to an LTL formula to generate a counterexample to the formula if possible. A counterexample is a sequence of state transitions in the behavioral model of the system that will result in a violation of the LTL formula. An LTL formula consists of propositional variables, logic operators, and temporal operators. SPIN can check input behavioral models with respect to a property stated as an LTL formula to generate a counterexample to the formula if possible.

The behavioral model of the system given to SPIN is represented in PROMELA. PROMELA (Process or Protocol Meta Language) is a language built for modeling processes in parallel systems. It realizes Dijkstra's Guarded Command Language [10]. The language also includes a communication structure between processes. Different processes can communicate through channels with predetermined data structures. A channel may either be a buffer-based or a rendezvous-based channel. A rendezvous-based channel is a channel with zero buffering ability. A message sent to a rendezvous-based channel blocks the execution of a process until the message is received from another process. Conditional control flow structures in PROMELA are non-deterministic, meaning any random flow with its guarding condition true can be chosen with an equally random chance. However, the user can instruct the model-checker to evaluate all non-deterministic choices at once. Also, it is possible to include C code snippets and C program files to PROMELA code for additional logical capabilities if need be. These C codes are callable from the PROMELA side, and they can access any variable or process at the PROMELA side.

SPIN has directives for making some trade-off decisions. Those trade-off decisions either increase speed or reduce memory usage with some loss on other aspects. Some examples of these decisions are used in our work. First is using bitstate hashing, reducing state-space table size, and increasing memory effectiveness while searching. The second is disabling array boundary violation checks, increasing the verification speed of SPIN. The third is for generating more than one unique counterexample if possible. The final one is a runtime option that instructs SPIN to return a counterexample with the smallest number of state changes. This one is used for instructing SPIN to generate shortest path solutions. Figure 1 explains how we use SPIN step-by-step by highlighting artifacts generated along with the usage.

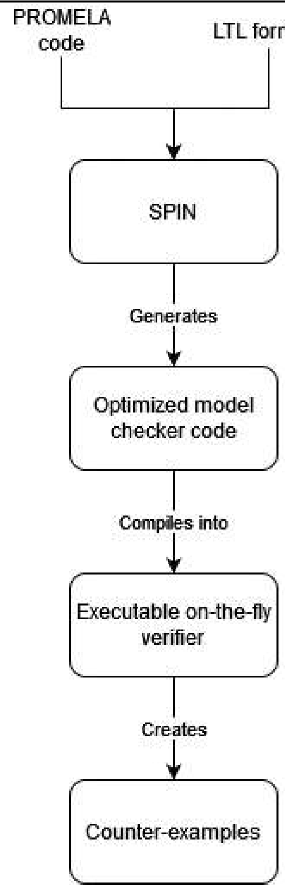


Figure 1: SPIN's workflow

In our work, we have used SPIN to verify various game design properties and to find a solution to the maze-like puzzle games. Since we have used a model-checker instead of an algorithm-based solution, we were able to deliver different properties with the exact same PROMELA model.

2.1.3. GVG-AI and Py-VGDL

GVG-AI is an artificial intelligence competition held in multiple categories, including general video game playing, level, and rule generation. For having a standard notation, a common Video Game Description Language (VGDL) was proposed in 2013 [11], called PyVGDL. Nowadays, the project has moved to PyVGDL 2.0 [12]. Two different files define games in PyVGDL: one is a game rules file, including sprites included in the game, how they are represented in the game file, how they interact with each other, and when the game terminates. The other is a level file, which is basically a 2-D array representing sprites positioning when the game starts.

In our work, we have forked PyVGDL to implement a predictable non-player character (NPC) type that follows the A* algorithm to chase a target and a controller that takes all the moves that it is going to do upfront. There are two reasons for the implementation of that specific NPC. The first is to make it predictable, making the same moves in the game model and engine. By making it predictable, we gain consistency between the actual game and its model. The second reason is to benchmark the model-based solution against a perfect opponent. By making its opponent play optimally in a two-player game, we can benchmark the optimality of the solution that the model-based approach gathers. The implementation of the controller is for the ability to detach the game after submitting the moves provided by SPIN. Rather than submitting the moves one by one and waiting for the evaluation of the moves given, by implementing this controller, we can instruct the game to be played in parallel while the pipeline can work on the next level-in-test. The fork is reachable at [13].

2.2. Literature Review

Model-checking is a technique used for formally verifying reactive systems [14]. The model checking process includes modeling a system's behavior and then fully exploring the system's behavior with respect to temporal logic formulae, where the formulae stand for system requirements. Although solutions using the model-checking approach tend to suffer from the exponential nature of the model-checking problem [15] [16], model checking is used to study games including major video game titles [17] [18], simple single-player [19] [20], or multi-player games [21].

Although verification of software, in general, is a very active area for research, verification of games is only partially investigated. Research in games regarding formal verification focuses on three main areas.

The first of these areas is verification of the game itself by verifying game rules. Verification of a multiplayer mobile game, *Penguin Clash,* is done with a model checker by Rezin et al. [21]. The work is focused on verifying the robustness and correctness of game rules. The game is verified against four different basic formulae that verify the game's requirements using a model-checker. These requirements include basic collision behavior and winnability of the game by both players. Also, the work proposes a pipeline for creating verified games by placing a model-checking phase between game design and game integration. However, the work is built for verifying the game rules instead of content related to the game. Verification of an RPG game specified in a domain-specific language is done by Barroca et al. [22]. The game development at this work is done in a Model-Driven way. For the starting point, the developer creates a model for a role-playing game. Next, the game model is transformed into a model to be verified in an algebraic Petri net. After verification, the model of the developer is transformed into code for a mobile game. Although the proposed model is promising, the necessary steps for constructing a successful game model are not explicitly stated. The game of *Bomberman* is verified using a discrete-event simulation in a model checking environment in another work [19].

Another area is automated game balancing. In a multiplayer game where a player picks a strategy against others, it is expected to have a balance between strategies to have more player excitement. If one way of playing the game is strictly better than others, the other strategies become obsolete, and the game can lose its variety. Kavangh et al. studied automated game balancing with the help of a probabilistic model checker [20]. With the help of a model checker, their solution mimics how a player adapts between different strategies in a chain to see if any strategy is superior to others. Milazzo et al. studied extracting successful strategies and balancing game core mechanics with the help of statistical and probabilistic model-checkers [23]. The work includes three case studies that include answering a game design question via model-checking.

The final area of focus is the verification of a storyline in a story-driven game. Holloway focuses on pointing out logically incorrect parts in a game's storyline before the actual development of the game starts [18]. Verifying educational adventure video games with the help of model checking is achieved by Moreno-Ger et al. [24]. The work focuses on scene-based educational adventure games. After the verification step, if the game fails in the model-checker, an animation of how the game failed is generated from the counterexample trace from the model-checker. However, our work differs from this one in one main aspect. Since this work's focus was on adventure games, which are built on a scenario, there exist no fixed game rules like our target genre. So, in this work every different scenario must be verified with a totally different model. On the contrary, our work is focused on puzzle games, which have a fixed rule set and a level configuration. Thus, our target genre allows us to generate models from template models, unlike this work.

We used cellular automata to generate maze-like game levels in our main pipeline. Usage of cellular automata on procedural generation of game levels is also one of the popular research areas over the last decade [25], [26], [27]. The main reasons for the usage of cellular automata are their simplicity and reliability [25]. A simple cellular automata-based algorithm is evaluated for its performance in generating tunnel-based maps by Johnson et al. [25]. Cellular automata are also used with the help of genetic algorithms to generate mazes [26], [27]. There is also a pure cellular automata solution proposed [8]. One other methodology proposed to generate game levels includes multi-layered cellular automata and Hilbert curves [28]. There is also a different usage of cellular automata called self-referencing cellular automata [29]. The self-referencing cellular automata are developed to compensate for the lack of feedback between the elementary cellular automata rules and its output. The rule depends on the output line-by-line. The output and the rule of cellular automata may oscillate mutually, get fixed in a mutual point, or the output may be oscillating with a fixed rule. However, the output of a self-referencing cellular automaton is not fit to be a puzzle level. Cellular automata are also used in machine learning literature. A cellular automaton type called Classificational Cellular Automaton was used as a classifier in a Multiple Classifier System [30].

We used three games in our study. These games are a single-player maze, a multi-player maze, and Sokoban. Both mazes and Sokoban are well-studied games from various perspectives.

Maze is a game where the player tries to move on to a specific tile to exit a labyrinth-like level. Maze solving is mainly studied as a path-finding problem, and according to Goyal et al., two of the most effective methods are identified as the A* search algorithm and the Dijkstra's algorithm [31]. We used the A* search algorithm as a baseline in our research for comparing our proposed method between these two methods.

Sokoban is a game that consists of the pusher who must push several boxes into a set of designated storage locations without getting himself, or the boxes stuck [32]. Sokoban is a game that is proved to be an NP-hard and PSPACE-complete problem [33], [34]. Usage of pattern databases is salvaged for finding optimal solutions to Sokoban levels by Pereira et al. [35]. Pattern databases store shortest distances from abstract states to abstract goal states, and they are used to find optimal heuristic functions by introducing an intermediate goal state. One other successful method for solving Sokoban levels is the iterative-deepening A*, proposed in [36]. However, to the best of our knowledge, no work in the literature uses a model-based approach to solve any of these games.

To conclude this section, there are various applications of model-checking in video games. However, these are mainly focused on the game rules or story design. These are two steps that are placed before the actual software development for the game. However, our proposed method is useful at the stage of deployment and stages before development. Also, our proposed method is focused on verifying game levels instead of the game itself. These two aspects of our method make it unique in the literature.

CHAPTER 3

METHODOLOGY

We have developed a framework that procedurally generates a maze-like level and verifies whether it fits against the requirements provided. Also, the score for the counterexample provided by SPIN is returned for the possible usage of the score in assessing the level. We did not use this aspect for assessing the level but added this feature for the games requiring a specific score to be won. The rules of the game are stated in a template model specific to that game. For the games we used in our work, four different template methods are proposed. To verify whether a level is solvable, the template game model is configured with respect to the level-in-test, and SPIN is instructed to check the game model against an LTL formula that states the game level can never be won. This instruction results in SPIN finding a counterexample, a sequence of actions that wins in that game level. Thus, the level gets verified, and a winning sequence of actions is acquired. In this step, it is detected whether the level is solvable or not. Then, the acquired counterexample is put in PyVGDL to reanimate the counterexample extracted from the level. Also, the points obtained by the counterexample are collected and can be used as an additional requirement. The pipeline that includes a level generator and a level verifier is at [37].

The games that we used with our pipeline are:

- Maze: A single-player maze game that has only one exit point. No items or interactions except the exit point is included.

- Race: A maze game in which the player is racing to solve the maze against an NPC that uses an A* algorithm to find an exit point. The maze consists of walls and floors only, and there are no items to interact with except the exit point.

- Sokoban: A puzzle video game in which a player pushes around boxes in a warehouse to place them in predetermined positions. The player wins when all boxes are in a predestined position. The game has few variants. The Sokoban variant is used in this work where the player tries to fill holes with boxes around. This is also the variant presented with PyVGDL as an example game.

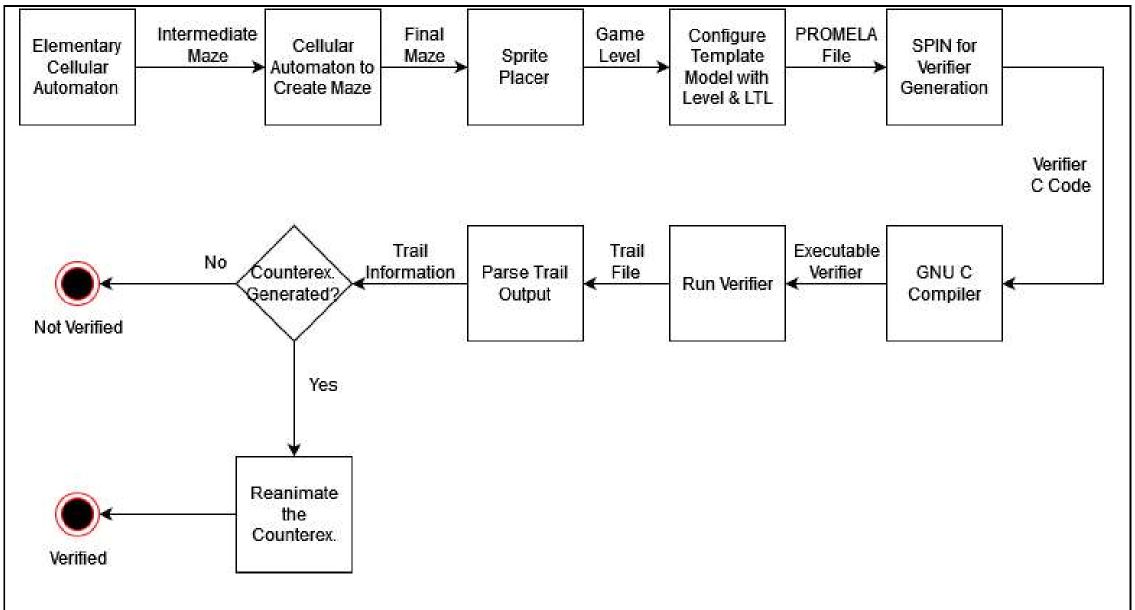


Figure 2: The main pipeline proposed.

All games' rulesets, which include sprite types used in the game, the interactions between the sprites, and ending conditions of the game, are presented in Appendix-A. Also, configurable game models are presented in Appendix B.

The proposed pipeline in Figure 2, which is explained in detail in the following sections, works as follows: First, a maze level is generated with the help of two levels of cellular automata. The width of the level is determined from a random line that the user feeds, and the height of the level is taken equal to its width, making levels square. The line fed to the pipeline is also used as a random seed for the maze level generation. Then key sprites are placed in the maze to create a game level. Next, the model corresponding to the game is configured for solving that level, and the LTL formula to check the requirement is appended. This configuration includes embedding the level map and sprite locations in the PROMELA code's initialization process. Then SPIN is used to generate an executable verifier to collect a verifying sequence of actions. If the level is verified, moves provided at that point are played on the map to reanimate how the requirement is satisfied by the moves suggested by SPIN.

3.1. Level Generator

3.1.1. Maze Generation

Our level generation module uses elementary cellular automaton implementing rule 150, one of the chaotic rules according to Wolfram [7]. The cells can have two states, a floor, or a wall, being 0 and 1, respectively. We used empty cells for padding, and we started the automaton with a randomly generated line of full and empty cells. Every next state of the cells is appended below the starting line until the level becomes a square. The output looks like a maze, but it is too rough. To make it more maze-like, we applied another level of operation.

The next component of the cellular automaton is a life-like cellular automaton specialized to generate maze-like outputs [8]. The rule of it is abbreviated as B3/S12345, meaning if a cell is a floor and has precisely three Moore neighbors marked as a wall, it is born (turned into a wall), if a wall has one to five wall cells in its Moore neighborhood, it survives (stays as a wall). The rule is continuously calculated repeatedly on the maze candidate until the changes that happen between generations are negligible (For a 24 by 24 maze, this value is taken as four or less).

Figure 3 shows the three main steps of maze generation. In Figure 3-a, the random line to start generation is given. In Figure 3-b, the output of the elementary cellular automaton with rule 150 is shown. As seen, the output is quite complex and contains closed rooms with sawtooth-like walls. To generate a more maze-like level, an extra step is needed. Then, the life-like cellular automaton mentioned above is applied. The output after the application is a maze-like level with lots of corridors with straight walls. Also, the output contains no closed rooms. The final output of the maze generation step is in Figure 3-c.

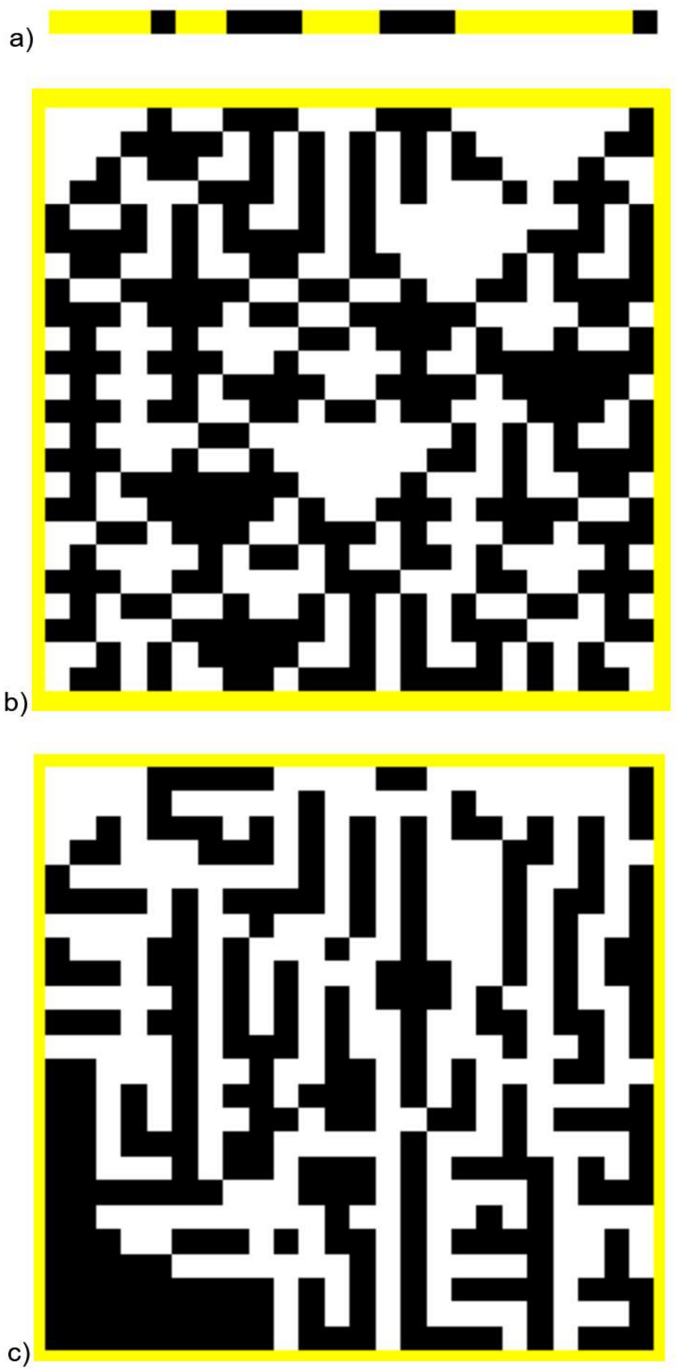


Figure 3: Steps of procedural maze generation.
a: The random line, b: The output of the first level of CA, c: The output of the second level of CA.

3.1.2. Sprite Placement

We made different placement choices for each different game type.

For the game “Maze,” we place the avatar and the exit on the two floor cells that are most distant in Manhattan distance, which are often on opposite ends of one of the diagonals of the level.

For the game “Race”, the avatar and the opponent are placed on the two floor cells that are most distant in Manhattan distance, and the exit portal is placed at the midpoint of them, at equal distance to both of them in a beeline. Since the level is not symmetrical and the opponent moves optimally, using A* search, some of the generated levels end up unwinnable.

For Sokoban, levels are not generated by our procedural content generation pipeline. The reason for that is, the generation pipeline we use results in long and narrow corridors that disallow maneuvering around boxes to direct them into holes. The level set used to test the models developed for Sokoban can be found at [38], and an example level from the set can be found in Figure 4.

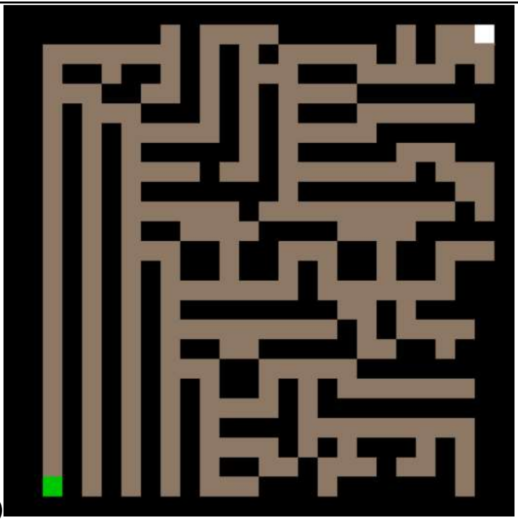
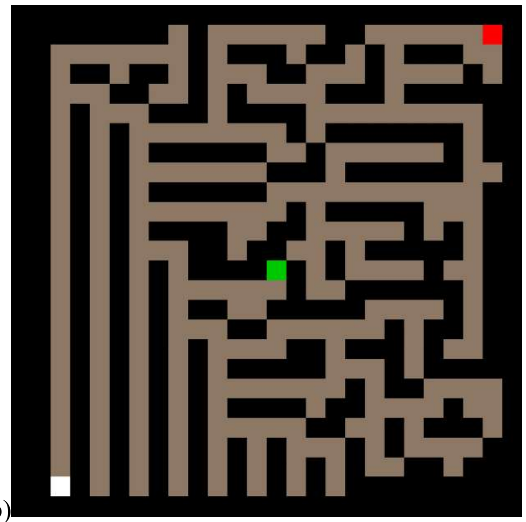
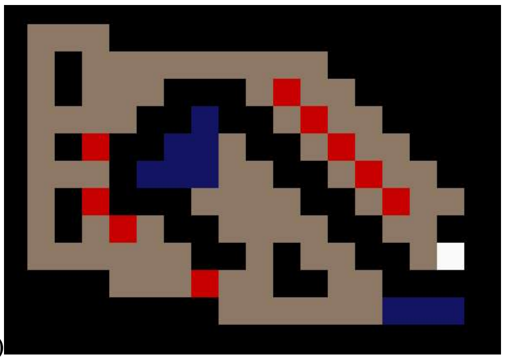
a)

b)

c)


Figure 4: Generated maze-like levels with sprite placement.
a: For Maze, b: For Race, c: An example Sokoban level from the level set.

3.2. Level Verifier

We use SPIN as the model-checker to verify whether the auto-generated with respect to custom requirements. Also, we use SPIN to generate a winning sequence of actions to be able to show the level is winnable. Our framework's level verifier step takes the game level as input and modifies the template behavioral models that we developed in PROMELA. We have developed four different template models for three games that we used in this study. While the games "Maze" and "Race" have one model each, Sokoban has two different models. The template models developed for this purpose are explained in this section. To verify the level, we use the counterexample generation feature of the model-checker SPIN. We give SPIN the model for the level, with an LTL formula that states, "The game with that level does not fit the given requirement" as input and uses SPIN to generate a sequence of actions that will eventually fit the requirement submitted. Since we use model-checking to verify levels, we verify different properties without changing the PROMELA model, but only the LTL formula.

A template model is a behavioral model of the game that includes processes for different players involved in the game. All players make turn-based moves on the same copy of the level, which is represented as a two-dimensional array. This array's sizes and content are left undefined in the template model and automatically initialized with respect to the level-in-test. Processes representing a player's behavior include the main loop that goes on until the game is won or lost, making decisions in each iteration loop. The decision is going to either up, down, left, or right cell for a navigation-based loop if possible. With every move done, the array that represents the game level is updated.

Once model-checker SPIN is given a model, it creates a Kripke structure from it. From that structure, a Büchi automaton of the given system is generated. Once the automatic-representation of the system and the property presented are generated, SPIN looks for a possible intersection between these two automata to find an accepting cycle. If the result of the intersection operation is an empty language, this will point out that there is no accepting cycle found in the model presented with respect to submitted LTL formula. Thus, the algorithm that finds out solutions to games is a part of the model-checker, instead of the models used.

The model that can verify "Maze" levels is navigation-based. There exists only one process for the game, representing the avatar. The avatar process includes only a main loop, and it makes a non-deterministic, navigation-based decision every iteration of the loop until the game has won. This is only a model of the Maze game, but not a maze-solving algorithm. This decision is moving to one of the four main directions, up, down, left, or right. Since there is no rule against the total number of moves, there is no way to lose a valid maze level. However, a maze level can never be won if poorly generated or designed, having no way to reach an exit.

The pseudocode of the avatar process is shown in Figure 5. Also, a solution produced with this model can be found in Figure 6. The path produced as the solution is highlighted with the color white, starting from the white square to the green one. The number of moves that it takes to arrive at every corner on the path is given in the figure.

```
Process Avatar_Maze(GridOfCells[ ][ ], avatar_pos_1, avatar_pos_2)
won = False
While won != True:
In a non-deterministic way:
        if GridOfCells[avatar_pos_1 - 1][avatar_pos_2] is either a floor or a portal:
           then move avatar to (avatar_pos_1 - 1, avatar_pos_2)
           if the avatar has collided with the portal:
              then won = True
        else if GridOfCells[avatar_pos_1 + 1][avatar_pos_2] is either a floor or a portal:
           then move avatar to (avatar_pos_1 + 1, avatar_pos_2)
           if the avatar has collided with the portal:
              then won = True
        else if GridOfCells[avatar_pos_1][avatar_pos_2 - 1] is either a floor or a portal:
           then move avatar to (avatar_pos_1, avatar_pos_2 - 1)
           if the avatar has collided with the portal:
              then won = True
        else    GridOfCells[avatar_pos_1][avatar_pos_2 + 1] is either a floor or a portal:
           move avatar to (avatar_pos_1, avatar_pos_2 + 1)
           if the avatar has collided with the portal:
              then won = True
 END
```

Figure 5: Pseudocode for the avatar process model for the Maze.

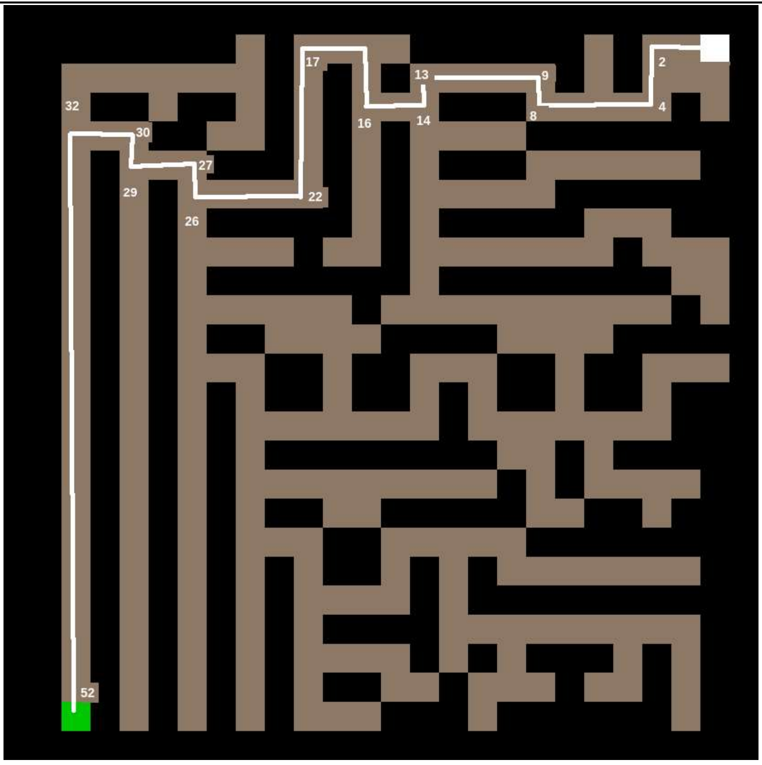


Figure 6: An example solution produced by the Maze model.

The model used for producing a winning sequence of actions for the game “Race” is also navigation-based. The model includes two processes, one representing the avatar and one representing the NPC opponent. These two processes make changes on the very same two-dimensional array, representing the game level. The processes work in a turn-based manner. To achieve a turn-based model, two rendezvous PROMELA channels are used. After making a move, both processes pass their turns to the other by the respective channel and wait for the other to pass their turn at the start of their loops from the other channel. Like the model of the game “Maze,” the avatar process makes non-deterministic, navigation-based decisions in every iteration of its loop. However, the opponent’s decision is deterministic. A* search algorithm is implemented for both the game and the model. The same heuristic, the Manhattan distance, is used for both implementations to produce the same outcome. To have an A* search algorithm in the model, a C code snippet is implemented. The processes pass to each other after making decisions until one of them touches the maze’s exit point. This is only a model of the Race game, but not an algorithm to solve Race games.

Pseudocode for both avatar and opponent processes can be found in Figure 7 and Figure 8. Also, a solution produced by the “Race” model can be found in Figure 9. In the figure, the path taken by the avatar is highlighted with white, and the path taken by the opponent, which is produced by the A* algorithm, is highlighted with red. Also, in the figure, the number of moves that both players need to do in order to arrive at every corner is given.

```
RendezVousChannel1 -> Channel with 0 buffering capacity.

RendezVousChannel2-> Channel with 0 buffering capacity.
Process Avatar_Race (GridOfCells[ ][ ], avatar_pos_1, avatar_pos_2)
won = False
lost = False
While (1):
(RendezVousChannel1.getMessage) //Locks process up until the rendez-vous happens.
        If (won == True or lost == True):
           End
        In a non-deterministic way:
        if GridOfCells[avatar_pos_1 - 1][avatar_pos_2] is either a floor or a portal:
           then move avatar to (avatar_pos_1 - 1, avatar_pos_2)
           if the avatar has collided with the portal:
              then won = True
        else if GridOfCells[avatar_pos_1 + 1][avatar_pos_2] is either a floor or a portal:
           then move avatar to (avatar_pos_1 + 1, avatar_pos_2)
           if the avatar has collided with the portal:
              then won = True
        else if GridOfCells[avatar_pos_1][avatar_pos_2 - 1] is either a floor or a portal:
           then move avatar to (avatar_pos_1, avatar_pos_2 - 1)
           if the avatar has collided with the portal:
              then won = True
        else    GridOfCells[avatar_pos_1][avatar_pos_2 + 1] is either a floor or a portal:
           move avatar to (avatar_pos_1, avatar_pos_2 + 1)
           if the avatar has collided with the portal:
              then won = True
   RendezVousChannel2.sendMessage
 END
```
Figure 7: Pseudocode for the avatar process' model for the Race.

```
RendezVousChannel1 -> Channel with 0 buffering capacity.
RendezVousChannel2-> Channel with 0 buffering capacity.
Process Opponent_Race (GridOfCells[ ][ ], opponent_pos_1, opponent_pos_2)
won = False
lost = False
While (1) :
(RendezVousChannel2.getMessage) // Locks process up until rendez-vous happens.
        if (won == True or lost == True):
           End
        Get the result of A* Search
        if the result is moving up:
           then move opponent to (opponent_pos_1 - 1)(opponent_pos_2)
        else if the result is moving down:
            then move opponent to (opponent_pos_1 + 1)(opponent_pos_2)
        else if the result is moving left:
            then move opponent to (opponent_pos_1)(opponent_pos_2 - 1)
        else:
            then move opponent to (opponent_pos_1)(opponent_pos_2 + 1)
        if the opponent has collided with the portal:
            then lost = True
        RendezVousChannel1.sendMessage
END
```

Figure 8: Pseudocode for the opponent process' model for the Race.

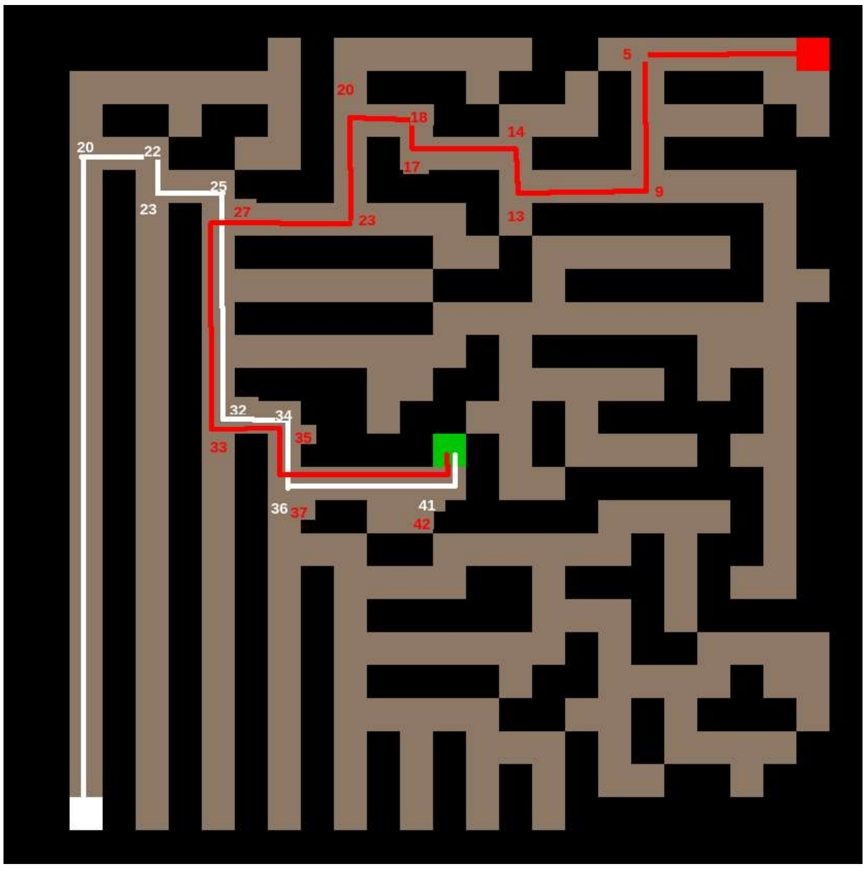


Figure 9: An example solution produced by the Race model.

There are two different models developed for verifying Sokoban levels. One is a vanilla model, which is navigation-based, while the other uses the help of a path finding algorithm, called the push-level model. Like the models before, the Vanilla model is navigation-based, except this time the model checks if moves done can be used for solving a Sokoban map. An avatar process makes navigation-based decisions in a loop and updates the array representing the game level. Navigated cells can result in a basic movement or a push. For this model, more challenging levels become impossible to win since the gigantic search space of a PSPACE-complete game like Sokoban.

Thus, we adopted a more sophisticated approach to this specific game from [32]. This model assumes that in a game like Sokoban, the moves that do not change any sprite's position beside the avatar's are irrelevant. Considering this, we proposed a push-level model for reducing the search space. The push-level model uses the help of a path finding algorithm. Like the other Sokoban model, this one also includes a process that symbolizes the avatar, making decisions until the game ends. However, this time, the decisions are about the pushes that the avatar can make without changing any other sprite's position, instead of the basic navigation moves it can do. Before every loop, the model is informed about pushes that the avatar can perform without changing any other sprite's position beside itself, and the avatar process decides non-deterministically on doing one of these pushes. This is done via a helper C code that checks available pushes on the map before the avatar makes a decision. The avatar process is informed about possible pushes it can make before every decision by an array that is shared between the helper C code and the model checker code. After the model makes a choice, the 2D array that resembles the game level is updated accordingly; all moves done before the push to make the push possible, and the pushing move is recorded, and another push-level search is performed in the level until there are no pushes available. At that point, having zero boxes in the game level makes it a win. The helper C code mentioned can be found in Appendix C.

The expectation from push-level search is reducing the state-space by the abstraction done in it. However, in that model, any push is equally likely to be done at any point. The situation was different in the vanilla model. In the vanilla model, all one-cell movements are equally likely, so the boxes that are far away are less likely to be pushed than near ones. This will result in randomly switching boxes while solving the level, and the avatar is likely to perform lots of transitions between boxes. This will make the resulting movesets to be larger in the push-based model. In the end, the expectation from that model is to have a larger solving rate but with a larger solution size.

The Vanilla model for solving Sokoban levels is presented in Figure 10, and the push-level model is presented in Figure 11. An example solution to an example Sokoban level for the push-based model is shown in Figure 12. The lines drawn on the level show the movement done by the avatar. An example solution to the very same Sokoban level by the vanilla model is shown in Figure 13. Both figures include four sub-figures that should be read in an order of top-left, top-right, bottom-left, and bottom-right.

```
Process Avatar_Sokoban_1 (GridOfCells[ ][ ], avatar_pos_1, avatar_pos_2)
remaining_boxes = Number of boxes
While remaining_boxes > 0:
In a non-deterministic way:
If GridOfCells[avatar_pos_1 - 1][avatar_pos_2] is either a floor or it is a box and GridOfCells[avatar_pos_1 - 2][avatar_pos_2] is not a wall or a box:
        then move avatar to (avatar_pos_1 - 1, avatar_pos_2)
        if the avatar has collided with a box and GridOfCells[avatar_pos_1 - 2][avatar_pos_2] is a hole:
                then remove both the hole and boxes
                remaining_boxes--;
        else if the avatar has collided with a box:
                then move box to (avatar_pos_1 - 2, avatar_pos_2)
else if GridOfCells[avatar_pos_1 + 1][avatar_pos_2] is either a floor or it is a box and GridOfCells[avatar_pos_1 + 2][avatar_pos_2] is not a wall or a box:
        then move avatar to (avatar_pos_1 + 1, avatar_pos_2)
        if the avatar has collided with a box and GridOfCells[avatar_pos_1 + 2][avatar_pos_2] is a hole:
                then remove both the hole and boxes
                remaining_boxes--;
        else if the avatar has collided with a box:
                then move box to (avatar_pos_1 + 2, avatar_pos_2)
else if GridOfCells[avatar_pos_1][avatar_pos_2 - 1] is either a floor or it is a box and GridOfCells[avatar_pos_1][avatar_pos_2 - 2] is not a wall or a box:
        then move avatar to (avatar_pos_1, avatar_pos_2 - 1)
        if the avatar has collided with a box and GridOfCells[avatar_pos_1][avatar_pos_2 - 2] is a hole:
                then remove both the hole and boxes
                remaining_boxes--;
        else if the avatar has collided with a box:
                then move box to (avatar_pos_1, avatar_pos_2 - 2)
else if GridOfCells[avatar_pos_1][avatar_pos_2 + 1] is either a floor or it is a box and GridOfCells[avatar_pos_1][avatar_pos_2 + 2] is not a wall or a box:
        then move avatar to (avatar_pos_1, avatar_pos_2 + 1)
        if the avatar has collided with a box and GridOfCells[avatar_pos_1][avatar_pos_2 + 2] is a hole:
                then remove both the hole and boxes
                remaining_boxes--;
        else if the avatar has collided with a box:
                then move box to (avatar_pos_1, avatar_pos_2 + 2)
END
```

Figure 10:Pseudocode for the first avatar process' model for the game Sokoban.

```
***Process Avatar_Sokoban_2*** (GridOfCells[ ][ ], avatar_pos_1, avatar_pos_2)
remaining_boxes = Number of boxes
choices[remaining_boxes * 4] = 0,0,0…,0 //4 choices per box, for every side.
***While*** remaining_boxes > 0:
        **Run** push-level search() #choices array is filled
        **In a non-deterministic way:**
        **for** choice, choice_index **in** choices:
                **if** choice **is** available:
                        **Run** push(choice_index) #Helper C-code that will update the map and variables accordingly.
```
Figure 11: Pseudocode for the second avatar process' model for the game Sokoban.

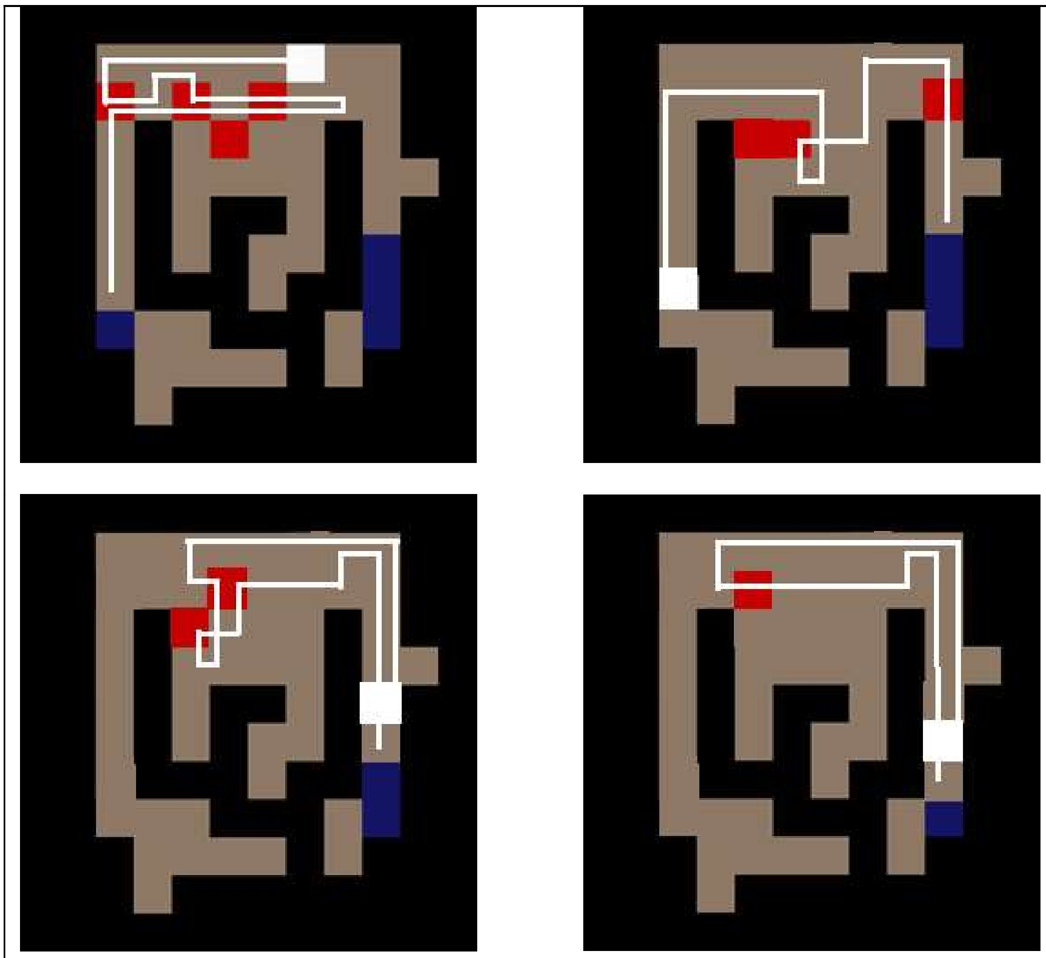


Figure 12: Solution created by the push-based model

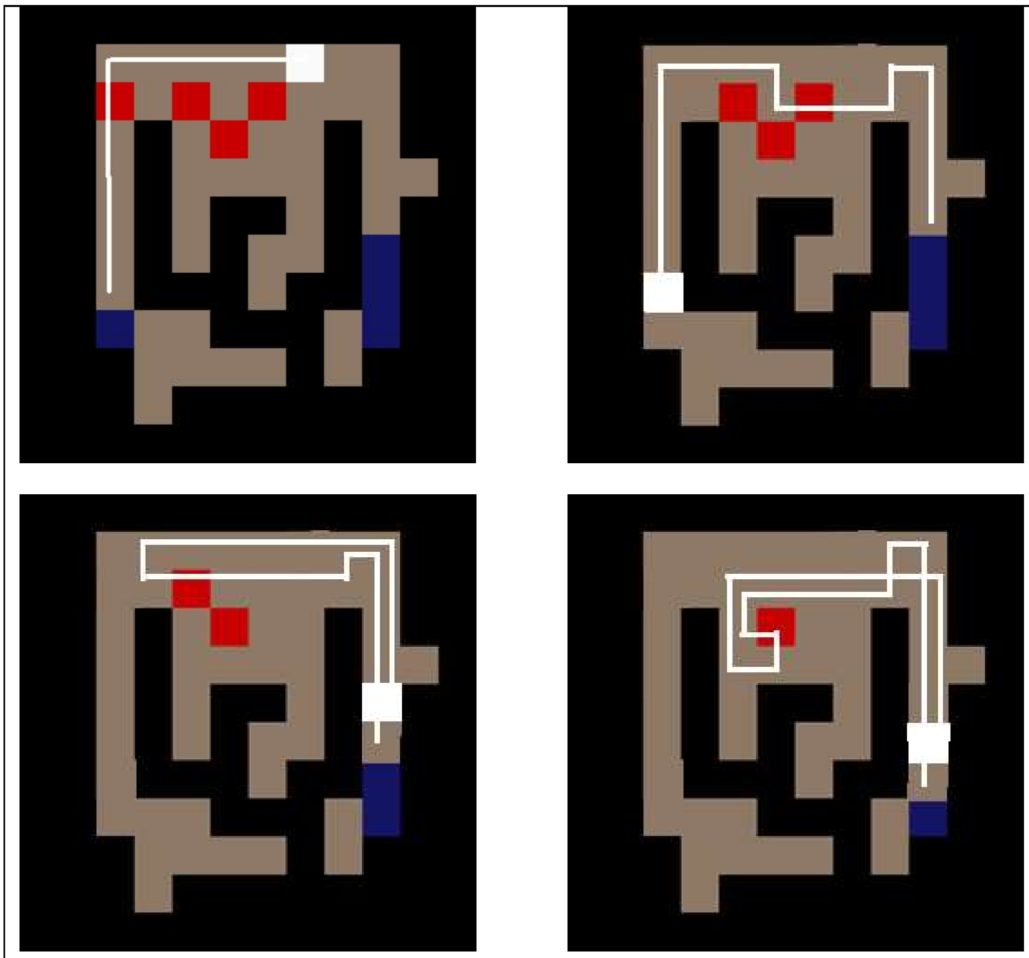


Figure 13: Solution created by the vanilla model.

3.3.Verifying Game Design Properties

Every design is done to satisfy some properties which are decided at the start. Puzzle games are no exception. Some properties can differentiate a good level from a bad one. Designers mostly use a source to look up when it comes to design for guidance.

One of these guides is Jesse Schell's book "The Art of Game Design" [39]. The book is a comprehensive guide in the design and is used as a textbook in different universities. One of the chapters from the book is dedicated to designing puzzle games and includes a list to look up. The list is called "Ten Puzzle Principles," and it includes ten principles that need to be satisfied to have a nice puzzle game. Some of the items can be related to the level design, while others can be related to the design of game rules or how levels are cascaded with each other. Since we picked three different games and checked game levels one by one, some of the rules are not applicable in our research. The reason for that is that the game rules of the chosen games are predetermined and cannot be changed.

Six rules that we cannot verify with our research are:

#1: “Make the Goal Easily Understood,” which is all about how easy and relatable game mechanics and goals are. Since our games’ goals are predetermined, this rule cannot be verified with our work.

#2: “Make It Easy to Get Started,” which is about how easy the player can understand the goal and controls. Since the goals and controls are fixed for the games we use, this rule cannot be verified in our work.

#5: “Increase Difficulty Gradually.” This principle is thought of as a matter of cascading the levels the right way instead of a level’s design. Thus, this requirement is impossible to be verified in the scope of our work.

#7: “Pyramid Structure Extends Interest.” This principle mainly points out how different puzzles can serve the same purpose, and thus it cannot be considered in our work’s scope.

#8: “Hints Extend Interest.” Since we have chosen three games with fixed rules in our work, it is impossible for us to verify games from that perspective.

#10: “Perceptual Shifts are a Double-Edged Sword”: Creating “Aha!” moments with game mechanics are not relatable with our work since the games we use have predetermined and fixed rules.

The four decisions that can be related to the level design are:

#3: “Give a Sense of Progress.” Although this can be interpreted as progress between levels, it also can be interpreted as progress in-level. Thus, it can be related to the level design.

#4: “Give a Sense of Solvability.” This is highly related to principle #3 and can be considered a level design issue because of the same reasons.

#6: “Parallelism Lets the Player Rest.” This can be related to level design since a parallel set of challenges may exist at a level, so reaching the same goal differently is possible.

#9: “Give the Answer!”. This principle is about serving the answer when the player gives up. This is not a level design decision, but serving the solution and being sure of the level’s solvability, can help the level designer.

The listed principles, however, are all qualitative ones. A designer may have quantitative goals in mind. Such goals may include the number of turning moves, straight moves, or their ratio to each other [40].

Our goal by designing this framework was to help the level designers check their desired properties if they submit a model of their game, their level, and the property they want to check. In order to see the effectiveness of our work, we checked both qualitative and quantitative properties to verify various game levels and explained them in the following subsections.

3.3.1. Sense of Continuous Progress and Solvability

Schell states that a good puzzle game should make the player see progress when solving a problem [39]. This property is highly coupled with another one, convincing the player that the game is solvable. For demonstrating these properties, we used Sokoban. The reason we choose Sokoban is, there is an opportunity to display visual progress. Since there are several boxes at the start, and the goal is to clear all of them, we decided to take the number of boxes remaining in the level to keep track of the user's progress. To check whether a continuous and uniform progression exists in the level, we watch the number of moves between the points where the player reduces the number of boxes by one. In order to check whether these properties are satisfied in a given Sokoban level, we used the LTL formula:

$$[]\ (!\ (win)\ ||\ (c\_moves > N))$$

Where win is a boolean variable that becomes true when the player wins a game level, c_moves is an integer variable that states the moves done without visual progress in the game. N is a number submitted by the designer and shows the maximum consecutive moves that are allowed without visual progress in the game level. The formula can be translated into:

"This level of Sokoban can never be won, or the consecutive moves done without visual progress is always greater than N."

This formula, when fed to SPIN, will generate a counterexample such as:

$$!([]\ (!\ (win)\ ||\ (c\_moves > N)))$$

$$\equiv\ <>!(!(win)\ ||\ (c\_moves > N))$$

$$\equiv\ <>((win)\ \&\&\ (c\_moves <= N))$$

This formula can be translated as:

"This level of Sokoban will be eventually won, and the consecutive moves done without visual progress will stay under or be equal N at the end."

Using the LTL formula given above, the level gets verified with respect to the requirement of constant visual progress. It has been proven that there is a solution to the level that has visual progress for every N move.

Our motive by adding this LTL formula to the example formulae is that it includes verification of a level with respect to two of Schell's ten principles of puzzle design. The first rule it includes is rule number 3, which gives the user a sense of progress. This is the desired thing to have in puzzle design since it keeps the user interested on the level. If visual progress takes a lot of time, it is possible for a user to quit a game due to the lack of interest. The second rule it includes is rule number 4, which gives the user a sense of solvability. Again, if visual progress in a puzzle level takes too much time, a user can quit playing since this situation may make the user judge the level as too hard to solve.

By including these two rules by Schell, this LTL formula can be used to verify a game level against a requirement of constantly grabbing the player's attention.

3.3.2. Producing a Solution

It is stated that the answer should be presentable to the player when needed to create an 'Aha!' moment [39]. So, a good framework that can verify a game level better presents a winning sequence of actions to the player or designer when need be. To do this, we use SPIN's playback functionality. Any model run against an LTL formula produces a "trail" file when a counterexample is generated. It is also possible to get information from that trail file with the help of the playback function. We use the LTL formula [](!win) to generate a winning sequence of state transitions recorded in the trail file. Win is a boolean variable that becomes true when the player wins a game level. However, in a puzzle game, to generate that 'Aha!' moment, you need to present an understandable and straightforward way to victory, which is mostly the shortest one. To produce the shortest path to victory, we used a run-time option presented with SPIN, fetching the counterexample with the smallest state transitions. In our case, this means the one with the least moves.

After the counterexample with the shortest path is found and recorded to a trail file, the trail file is played back to re-animate, which moves are done to achieve victory in the level. After getting the moves performed in the winning sequence, with the help of PyVGDL, the counterexample is played visually in a window to show the solution produced by our framework to the audience.

Using the LTL formula above, the level gets verified against the requirement of having a solution. It has been proven that level can be solvable by the player.

Our motive by adding this LTL formula to the example formulae is that it verifies a level for being playable. It includes Schell's 9th principle of puzzle design since it produces a winning output. Also, using this formula, since it is a simple one, the model can have less state variables, reducing the state-space. Reducing state-space will result in faster searches and verifications.

By verifying the levels with minimum resources, this LTL formula can be used to verify a game level in an environment with fewer resources as fast as possible.

3.3.3. Possibility of Different Solutions

Schell also states that there should be more than one way for a player to finish a puzzle game successfully [39]. Having more than one way to finish corresponds to generating more than one winning sequence. If at least two different sequences of actions can generate a counterexample, we accept that the level is solvable with more than one strategy parallel to each other. To display this property, we need SPIN to generate two counterexamples that have different state transitions. The tool we developed can instruct SPIN with run-time options to achieve this. After SPIN generates more than one counterexample in multiple trail files, the tool checks the filesystem for the files and decides whether the level is open to parallel strategies or not.

In order to demonstrate these properties, we use a simple LTL formula of:

[] (!win)

, which states that the level can never be won. This will result in a counter example of:

<> (win)

, which states that the level will be won eventually. We pass the LTL formula with our models to SPIN, use appropriate options, and try to produce multiple possibilities that can produce the counterexample we want. If more than one possibility can produce the same counterexample, it is safe to say that the level can be won with different parallel strategies.

By using the given LTL formula and creating more than one counter-example, the level gets verified against the requirement of having multiple solutions. It has been proven that the level at least has two different solutions.

Our main motive by adding this formula to the example formulae is that it includes verification of a level with respect to Schell's main principles. The principle it includes is the 6th one, which is the principle of parallelism. Having more than one solution, the level lets the player to choose one of the possible solutions to win. This may result in a player solving a level easier, since even if the player cannot discover a solution, there is always another one to find. This may be a desired property at a puzzle level.

By including this rule by Schell, this LTL formula can be used to verify a game level against a requirement of increased solvability since by giving more than one way to the player, the level gets more solvable.

3.3.4. Number of Turning Moves, and Number of Straight Moves

Instead of qualitative design goals, the level designers may have a quantitative requirement to fulfill instead. Kim et al. state that the number of turning moves or number of straight moves done to win a maze can be desired properties while designing a maze [40]. In order to check these properties in a maze model, we present the LTL formula

[] (! (win) || (moves < N))

, where moves are the desired kind of moves done in a playthrough, either straight or turning. The formula can be translated to English as: “The game never can be won, or the desired moves are always under N.” The counterexample produced by the LTL formula is:

!([] (! (win) || (moves < N)))

≡<>!(!(win) || (moves < N))

≡ <>(win && (moves >= N))

, which can be translated as: “The game will eventually be won, and the moves are greater than or equal to the desired number.”

Using the LTL formula above, the level gets verified against the requirement of having a certain amount of turning moves in the winning path.

Maze designers may desire such a property since turning moves increase difficulty. Adding more turning moves has two different contributions to a maze level’s difficulty level. First, it increases the length of the solution. Longer solutions may be perceived as more complex by the player. Second, having more turning moves can make solutions to a maze level harder to detect, while direct corridors to the solution are easier to detect.

By verifying a level with respect to the requirement of having a certain amount of turning moves, a level designer can test the levels in difficulty.

3.3.5. Twistiness of the Path to the Victory

A designer may focus on the ratios of straight and turning moves with respect to each other. The twistiness of the path shows the ratio of turning moves with respect to straight moves. An increased twistiness might require the player to be more agile to win.

Kim et al. state that turning moves’ ratio to the straight moves can be a desired property when designing mazes [40]. To verify a level against such a requirement, we use the LTL formulae:

[] (!(win) || (ratio_of_turns_to_straights < N)) for verifying a level which needs to be twistier than the ratio of N.

[] (!(win) || (ratio_of_turns_to_straights > N)) for verifying a level which needs to be less twistier than the ratio of N.

Using the LTL formula above, the level gets verified against the requirement of having a certain ratio of turning moves against straight moves in the winning path.

This property can be desired by maze designers since it increases the difficulty without depending on the level size. Adding more turning moves have the same two contributions to a maze level's difficulty level with the prior property regardless of the level size.

By verifying a level with respect to the requirement of having a specific ratio of turning moves, a level designer can test the levels in difficulty.

CHAPTER 4

EXPERIMENTS AND RESULTS

In this work, three different games are implemented using the GVG-AI framework. The list of the games is presented in the methodology section, and their rules are in Appendix A. The first two of these games are simple maze games. In the game "Maze," an avatar is expected to solve a maze with no other sprites involved. In the game "Race," the avatar is expected to solve the maze before its opponent does. The last game implemented is Sokoban, a well-known game for its computational difficulty [34].

The experiments are grouped into three main categories. The first group of experiments includes experiments done to determine parameters for the other experiments. Only one experiment in this category is used to determine the level sizes for the other experiments.

In the second group of experiments performance of the proposed model-based approach is measured against existing methods. There are three different experiments in this group. We first compared the proposed model-based approach against the A* search algorithm, a well-known and widely used algorithm, in maze solving. In the second experiment in this group, the proposed approach's performance is evaluated against Monte Carlo Tree Search in the game "Race." The last experiment in this group is a comparison of two different models for solving Sokoban levels. Since Sokoban is a demanding game to solve, this experiment aims to show how trade-off decisions may affect the performance of the proposed solution.

The third group of experiments demonstrates the ability of our approach in the verification of game design properties. There are two different experiments presented in this category. The first one is to verify levels for a qualitative property such as property from subsection C from Section V. The second one is to verify levels for a quantitative property such as subsection V from Section V. First of the two experiments in this category is aiming to verify "Race" levels for the existence of multiple ways to victory. The other experiment aims to verify maze levels for the twistiness of the path to victory, which is defined as turning moves' ratio to the straight moves [40].

All experiments are done on a computer that has an Intel Core i5-3470 CPU at 3.2 GHz clock frequency, 16 GB of RAM, and CentOS 7 as its operating system. All experiments were run in a single core of the CPU. More in-depth explanations of the experiments are given in their respective sub-sections.

4.1. Preliminary Experiments for Determining Parameters

4.1.1 Experiment 1

For the upcoming experiments, we needed to decide on few parameters. One of these parameters is the level sizes to use. To determine which level sizes to use, we have conducted a preliminary experiment.

This experiment aims to determine which level sizes to use in the other experiments. The experiment is designed as follows; three different candidates will be used in the following experiments, A-Star search, the Model-Based approach, and Monte Carlo Tree Search are offered the same 100 maze levels to solve in the same size. This size starts at 8 by 8 and increases 2 by 2 until one of the candidates fails to deliver a solution to any of the levels given. For MCTS, we have established a time limit. The limit here is 100 times more of the time scored by the model-based method. The last level size played will be designated as the maximum level size for the rest of the experiments.

The only metric collected from this experiment is the success rate of the candidates with respect to different sizes.

4.2. Experiments Regarding Performance Comparisons

4.2.1. Experiment 2

Path finding has been a popular topic in the literature for decades, and various algorithms are proposed so far. These algorithms include Dijkstra's Algorithms, A-Star Search, Breadth-first Search, and Depth-first Search. Path finding algorithms also find their usage in the literature for game research, mostly in mazes since a maze is nothing but a path finding problem.

Our second experiment aims to see the scalability of the proposed model-based approach with respect to one of the popular path finding algorithms, the A-star search. To assess the performance of the proposed approach versus the A-star search, we used the navigation-based model for the game "Maze." To generate the shortest path as its output, we used SPIN with the option that returns the counterexample with the least number of state changes. Since the model is navigation-based, the state changes occur in every move done by the avatar. Thus, the model-based approach is instructed to produce the shortest path possible when this option is used. The avatar's opponent is an agent implemented by us in PyVGDL, employing A-star search. As the heuristic of the A-star search, Manhattan distance is used.

We have given both candidates the very same maze levels in nine different sizes (8 by 8, 10 by 10, 12 by 12, 14 by 14, 16 by 16, 18 by 18, 20 by 20, 22 by 22, and 24 by 24), all generated by the pipeline described in Section IV. For each size, we generated 50 different levels for two candidates to solve. There are 450 levels in total.

The metrics we have collected for the experiment are the average solving time and the average solution length with respect to level size.

4.2.2. Experiment 3

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm notable for its employment in games. The advantage of MCTS is its usability without domain knowledge: The state information, possible actions, and the result of these actions is sufficient to use MCTS in any game. Thus, the method is highly used in the literature of game research.

The third experiment aims to compare the performance of the proposed approach with Monte Carlo Tree Search (MCTS) with a twist. The modification we have done is for making MCTS more prone to exploration instead of wasting computational time with back-and-forth moves. We keep a record of how many times a tile is moved on to and give a movement punishment, respectively. Since repetitive movements will result in more punishment, the agent is encouraged to explore further in the maze faster instead of doing repetitive actions.

Two competitors were run in two different games, “Maze” and “Race.” Both solvers are fed with the same levels in three different sizes (8 by 8, 16 by 16, and 24 by 24) generated by the pipeline we proposed. For every level size and game pair, both candidates are given the same 50 levels to solve (300 levels in total). As MCTS’s exploration parameter 1.41 (square root of 2) is chosen, and MCTS agent is given a hundred times more time to complete its search. The reason for that is MCTS having a disadvantage. This disadvantage is employing no domain knowledge, and the proposed model-based approach is built on domain knowledge.

The average length for the solutions found and the rate of solving are collected as metrics.

4.2.3. Experiment 4

The fourth experiment aims to compare the performances of the proposed two different template models for Sokoban in the same pipeline. These two models are the navigation-based model and the push-level model. More detailed information about these template models is in Section IV. Two competing models are run in the same 20 levels from Alberto Garcia 1-1 level pack [38].

Since the problem is memory-intensive, we used each model with the bitstate hashing instead of an exhaustive search in SPIN. This choice switches the state space to the bitstate search space, resulting in a much smaller state-space table, optimizing SPIN in memory usage.

While the navigation-based model works at a more detailed level, resulting in a larger state-space, the push-based model is more abstract and optimized for a smaller state-space with a performance loss on solution optimality.

The main expectation from the experiment is obtaining higher solution rates with the push-level search while losing performance on solution length. The experiment shows that harder or more resource-hungry problems can be solved using the proposed approach, using trade-off decisions. In the experiment, optimality was compromised to solve a memory-hungry problem.

The metrics compared in this experiment are the average time that competitors take in a level, either solved or not, rate of solving, and the solution length.

4.3. Experiments on Verifying Complex Properties

4.3.1. Experiment 5

The game “Race” is played against a bot opponent that uses the A* algorithm for making optimal moves. Since the player has no obvious advantage against its opponent in the sprite placement, some of the levels end up unwinnable. Also, some of the levels that are winnable end up having a one move margin between the avatar and its opponent to win. Schell states that there should be more than one way for a player to finish a puzzle game successfully [39]. Since a level cannot be verified not only if the level is unwinnable but also if this margin is equal to 1, this property is a valuable one to check in the game “Race.”

By conducting this experiment, we aim to show how the proposed method can be used to verify a level using a qualitative requirement. The requirement chosen here is the level having multiple ways to victory. The requirement and the verification method are presented in subsection C of Section V.

For this experiment, 200 “Race” levels are generated with the pipeline presented in the Methodology section, in the size of 24 by 24. The levels were put to the test against two different requirements, winnability and having multiple solutions. The requirements can be found in subsections B and C from Section V. The count of levels that satisfy the requirements are kept as a metric.

4.3.2. Experiment 6

A level designer may have quantitative requirements in mind when designing or generating a level. For a puzzle level, this requirement can include the total number of moves to solve a level or count or proportion of some kind of moves to other kinds of moves. To be more specific, for a maze level, this requirement can be a limit in turning moves’ ratio to straight moves. The proposed model-based approach can also be used to verify these quantitative requirements.

The aim of this experiment is to show how the proposed model-based approach can be used for verifying quantitative requirements. The requirement chosen is the twistiness of the solution path. The requirement and the verification process are presented in subsection V of Section V in detail.

200 “Maze” levels are generated with the pipeline presented in section IV, in the size of 24 by 24. The levels are tested against the requirement of having a minimum twistiness of 20%. Since it is possible for SPIN to generate counterexamples that have more twistiness with repetitive actions, a hard limit is placed on the number of moves done while solving a level. This limit is chosen as 55 moves, being ten more from the average shortest path in a 24 by 24 maze, according to the results found from Experiment #2. The count of levels that satisfy the requirement is kept as a metric to work on.

4.4. Results and Discussion

In this study, we have focused on the following questions:

- How does the model-based approach compare to the A-star search algorithm on scaling in shortest path finding?
- How does the model-based approach perform in puzzle games with respect to Monte Carlo Tree Search?
- Does the model-based approach scale with a game like Sokoban, which is known to be hard to solve in limited memory or limited time?
- How does the model-based approach can be employed to verify qualitative and quantitative requirements?
-

4.4.1. Experiment 1

Experiment #1 was conducted to determine the maximum level size to be used in the experiments. The experiment includes continuously increasing the level size until one of the three candidates fails to solve any level.

The metrics collected here are the solving rate of three candidate level solvers. We have collected this metric to have a comparison between the candidates and to find out a maximum level size for the upcoming experiments. We have continued the experiment until one of the candidates failed to bring up any solution to the given levels in size.

There is a single table, Table 2, showing the result of this experiment. It includes the percentage of levels solved by the candidates with respect to the level size.

Table 2: Results of Experiment #1.

Level size	Model-Based App.	A-Star Search	MCTS
8 X 8	100%	100%	40%
10 X 10	100%	100%	37%
12 X 12	100%	100%	31%
14 X 14	100%	100%	25%
16 X 16	100%	100%	18%
18 X 18	100%	100%	15%
20 X 20	100%	100%	9%
22 X 22	100%	100%	4%
24 X 24	100%	100%	0%

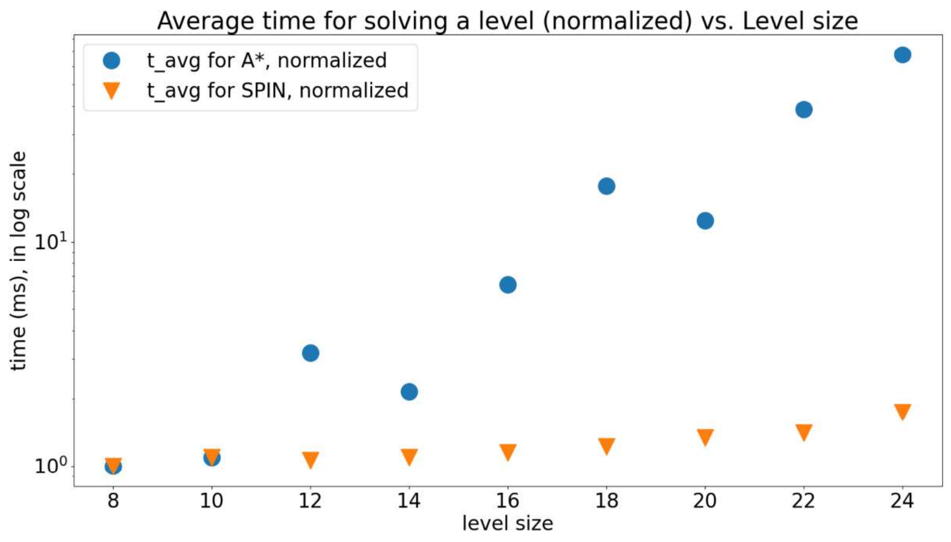
The results indicate that levels larger than 24 by 24 are not practical to work on. The larger levels are impractical for distinguishing the model-based approach from the MCTS since the rate of solving for the MCTS will not change for the larger levels. Also, larger levels do not make sense for distinguishing the model-based approach from the A-Star search since they are both 100% successful anyways.

4.4.2. Experiment 2

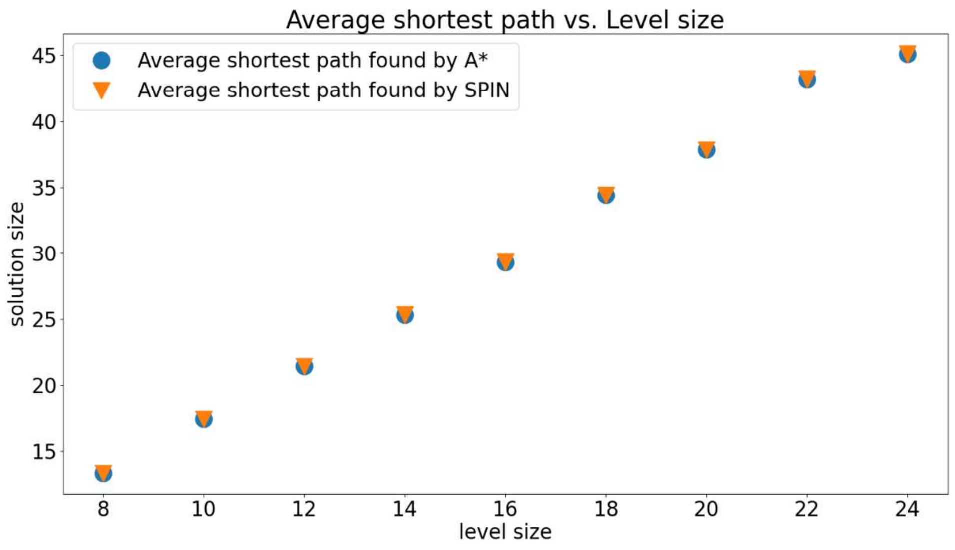
Experiment #2 seeks an answer to the first question. For this, we used our template model proposed for the game "Maze" against the classical A-star search. More information about the setup can be found in Section V.

The metrics collected here are the average time for solving a level and the average size of the solution brought up with respect to both competitors' level size. The main reason for the first one is to see the performance of the model-based solution with respect to the algorithmic solutions in pathfinding problems. However, a direct comparison would not be comparable since the environment of the two is different. The model-based solution has multiple file-operations, a source compilation, execution of this compilation's output, and a Python code moderating all of these; while the A* Search algorithm was implemented on Python. So, we have normalized these datasets to see how the solving time increased while the level size increased. As a result, the information extracted from this metric is not a solid performance comparison but a scalability comparison. The second metric is included to show the ability of the model-based solution in shortest path finding.

There are two figures, Figures 14-a, and 14-b, about the comparison. Figure 14-a shows the average time measured for the candidates to solve levels with respect to the levels' size. However, since the question is on scalability, we normalized both data with respect to their first data point. Thus, the data represents how much the average time for solving a level increased while the level size increased. Also, this normalization neutralizes the differences between the two candidates. Figure 14-b shows the average number of moves in the shortest path found in the experiment.



a)



b)

Figure 14: Results from experiment 2
a: Average time to solve wrt level size, normalized. b: Average shortest path wrt level size

Figure 14-b shows that the model-based approach has the same outcome length as the A* path-finding algorithm. This means that the proposed approach can find the shortest path to the solution in each maze. However, the candidates differ in scalability. With the increasing level size, the A* search seems to lose more performance relatively than the proposed approach. When the total space is multiplied by 9, the proposed methodology's average solution-finding time only increased 75%, while its competitor's increased nearly 70 times. To address our first research question, even if the A* search is faster for smaller levels than the proposed method, the trend in data shows that it is only fair to assume that this performance gap will be closed while the levels get larger.

The results of this experiment indicate that the model-based approach can be used as an alternative to a purely algorithmic approach in path finding problems. Since the proposed approach includes multiple file operations, it would be better to use the algorithmic approach in smaller levels. However, in relatively larger levels, using the proposed method is advised.

4.4.3. Experiment 3

Experiment #3 aims to find an answer to the second question. To assess the performance of the model-based approach with respect to MCTS, two different games are played in three different sizes by both agents.

The metrics collected here are the rate of success for both candidates and the average length of the solution they brought up in two different games and three different sizes. We collected these metrics to have a comparison between the proposed approach with respect to Monte Carlo Tree Search in solving maze-like puzzle games. The first metric is about finding a solution, while the second metric is collected to compare the quality of the solution brought up by both candidates.

Table 3, Table 4, Table 5, and Table 6 reflect the metrics collected in the experiment. Table 3 shows the success rates of both candidates in three different sizes for the game "Race." Table 4 shows the same metric for a maze game. Table 5 shows the average solution length of both candidates in three different sizes for the game "Race." Table 6 does the same for the game Maze.

Table 3: Success rates for the game Race.

Success Rate, Game = Race	Level Size: 8 x 8	Level Size: 16 x 16	Level Size: 24 x 24
MCTS	88%	21%	0%
Model-based	100%	100%	100%

Table 4: Success rates for the game Maze.

Success Rate, Game = Maze	Level Size: 8 x 8	Level Size: 16 x 16	Level Size: 24 x 24
MCTS	34%	14%	4%
Model-based	100%	100%	100%

Table 5: Average solution length for the game Race.

Avg. Sol. Len., Game = Race	Level Size: 8 x 8	Level Size: 16 x 16	Level Size: 24 x 24
MCTS	6.75	20.33	No solutions exist
Model-based	6.11	16.14	27.06

Table 6: Average solution length for the game Maze.

Avg. Sol. Len., Game = Maze	Level Size: 8 x 8	Level Size: 16 x 16	Level Size: 24 x 24
MCTS	42.12	127.14	353
Model-based	13.54	27.72	44.32

These results show that for maze solving, the success rate of the MCTS drops drastically while the level size grows. For the game "Race," since the game is not much fault-tolerant, referring to the sprite placement section, the drop in the solve-rate of the MCTS is even sharper. Although their performances are close for the smallest levels, in all six cases, the model-based solution is more performant with respect to its competitor, raising the difference as the level size increases.

To answer our second research question, the results indicate that the proposed approach has a certain advantage over Monte Carlo Tree Search in maze-like puzzle games since it relies more on computational power and memory, while the input size increases, the gap between two competitors` performances grows even more.

While doing these experiments, our expectations were parallel to the results we got from the experiment. We were aware of the advantage the model-based approach has over the MCTS due to the MCTS being a non-informed search, while the model-based approach is built on domain information. Nevertheless, even if the two are not exact alternatives to each other, the results point out that there is a practical advantage of using a model-based approach over MCTS when possible.

4.4.4. Experiment 4

Experiment #4 aims to answer the third question. To achieve this, two different models are used to solve levels of Sokoban, a memory-hungry game when tried to solve. One of the candidate models is a lower-level implementation, which may result in a much deeper search space with respect to the other.

The metrics collected from the Experiment 4 are the average time spent on a level, either solved or not, rate of solving, and average solution length for both models in the same Sokoban levels. By collecting these, we aimed to see if our more abstract model has improved the situation. The main expectation was to have faster and more solutions to the levels, but with less quality. The reason for that expectation is, by creating a more abstract level, the state-space that we are searching for is reduced. So, more levels should have become solvable with the abstract model, since some of the failures for the first model was because of the system running out of memory. By reducing the memory costs, we were expecting higher solution rates with the second model. Also, since the state-space is reduced by switching the model, it should take less time to search that space. So, the expectation was to have significantly shorter times for solving a level. However, since the second model is more abstract than the other one, we were expecting more inefficient solutions on average.

Figure 15 shows the result of this experiment. The figure uses bar plots to compare these two models in performance. The performance metrics collected here are solving rate of level set, time spent on a level, whether solved or not and the average length of returned moves in SPIN's counterexample.

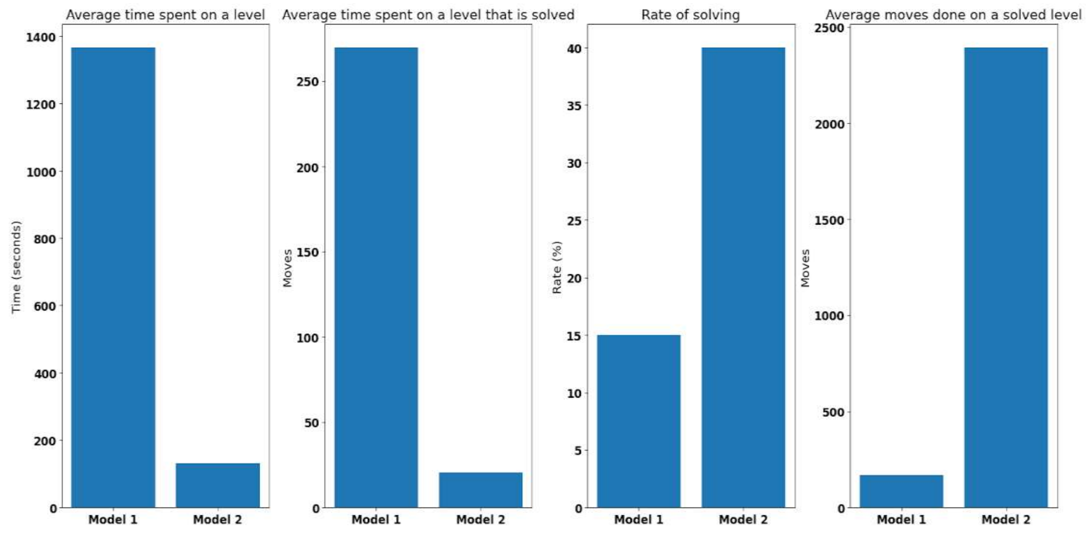


Figure 15: Metrics collected from Experiment #4. 1: Navigational, 2: Push-based

Figure 15 shows that switching from a navigation-based model to a push-based model increased the rate of solving by 166% and decreased solving time by nearly 90%.

However, the average length of solutions has increased by 15 times. We believe that further optimizations on the push-based model can reduce the resulting move set, such as tunnel macros or goal macros [32]. To address our third research question, this experiment shows that template models can be subject to trade-off decisions in the model design to get a better result if needed. Also, it shows that SPIN's different options that create a trade-off can be used to improve performance.

The results indicate that the model-based approach may react quite differently when certain trade-off decisions are made. In the experiment, that decision was shrinking the search space by using the push-level search, which trades optimality of the solution with the performance and the speed.

4.4.5. Experiment 5

Experiment #5 aims to find an answer to the fourth research question with a qualitative requirement. The game "Race" was chosen to conduct the experiment. The experiment was done to verify procedurally generated levels for the game "Race" with respect to the requirement of having multiple ways to finish the game.

The data collected here are the count of winnable levels and the count of levels that have an alternate way of winning the level. The main purpose of this experiment is to compare the output of the experiment with the expectations in order to show that the proposed methodology is usable in verifying qualitative requirements.

There are three factors that affect the result of this experiment. First, the avatar or its opponent has no obvious advantage over each other in the level generation phase. Either one can be advantageous, or none of them may have. Second, the player has the advantage of the first move. Both in the model and in the game, the player moves first, before its opponent. This gives the player an advantage in levels that are neutral in the level generation phase, where the goal is equally distant to the player and its opponent. The last one is that having an alternate way to victory can be ensured by either having more than one equally long but different path or having an advantage over your opponent that can afford to make a sub-optimal move. In the light of these factors, our expectation from the experiment was as follows: We were expecting that more than half of the levels generated would be winnable with the advantage of the first move. Also, we were expecting that the levels generated which satisfy the requirement will be just under but near 50%.

Table 7 shows the result of this experiment. The table has the total number of levels generated, the number that satisfies the winnability requirement, and the number of levels with more than one way to victory.

Table 7: The results of Experiment #5.

Total	Winnable	Has an alternate path
200	113	95

The result that we obtain from the experiment looks parallel with the expectations. Since the avatar has the advantage of the first move, 56.5% of the levels generated end up being winnable. This result is fitting to the expectation of having more than half of the levels being winnable. 47.5% of the levels generated have an alternate path of victory for the avatar, which is aligned with the expectation of having less than half of the levels having an alternate path.

This result indicates that the proposed model-based approach can be used to verify qualitative requirements to answer our fourth question.

4.4.6. Experiment 6

Experiment #6 aims to find an answer to the fourth research question with a quantitative requirement. The game of choice to verify a quantitative requirement is the game of maze. The requirement chosen to test levels is having at least 20% twistiness. By twistiness, we mean the ratio of turning moves done by the avatar with respect to all moves. Since twistiness can be increased by making a turning move repeatedly, we established a limit to the total moves that can be done. This limit is chosen with the help of the result from Experiment #2. Since the average shortest path in a 24 by 24 maze is nearly 45, the upper limit for the total moves that can be done is chosen as 55. The reason for this limit's existence is that the requirement can be satisfied with continuously performing repetitive movements. We have determined this value because we want to prevent SPIN from abusing the game rules to verify the level.

The data collected here is the count of levels that satisfies the requirement. The main purpose of this experiment is to compare the output of the experiment with the expectations in order to show that the proposed methodology is usable in verifying qualitative requirements.

We were expecting nearly all of the levels to be verified in this experiment. There are two main reasons for this expectation. First is, the target percentage is quite low. For a target placed at the cross corner of the level, 20% twistiness seems achievable. One regular turn makes four straight moves acceptable, and one U-turn makes eight straight moves acceptable. The other reason is that the upper limit for the total moves that can be done. For an average level, the avatar has ten extra moves that can be used to boost the twistiness of the winning path. Because of these two reasons, we were expecting a tiny percentage of levels not satisfying this requirement.

The result of this experiment can be found in Table 8. The table includes the total number of levels and the number of levels that satisfy the requirement.

Table 8: The results of experiment #6.

Total	Satisfies the requirement
200	196

The result obtained from the experiment is in the same direction as the expectations. With the help of low expectations and extra moves, 98% of the levels generated end up satisfying the requirement. This result fits our expectation, which is nearly all the levels fitting the requirement.

This result indicates that the proposed model-based approach can be used to verify quantitative requirements to answer our fourth question.

4.4.7. Further discussions

There exists further discussion out of this work's scope under this chapter. Two main discussions worth mentioning are applying this methodology to multiplayer games and salvaging cloud computing for performance gain.

At this stage, the work only includes one player and an NPC. Implementing a multiplayer game is also possible; however, this is not achievable with a personal computer since two deciding processes will expand the state-space exponentially, and the simulation will crash due to insufficient resources. Also, some modifications should be made to see the re-animation of the game on PyVGDL.

We have mentioned how much model-checking problems can be demanding before. Due to the state-space explosion problem, more complex games and bigger levels may fail because of memory or computational time problems. In order to achieve our goal without the limitation of physical computers, one can try to salvage cloud computing. Belletini et al. proposed a framework for model checking Computational Tree Logic (CTL) formulae using distributed systems and cloud computing facilities [43].

CHAPTER 5

CONCLUSION

The main focus of this thesis is creating a model-based level verifier framework for maze-like puzzle games. For this, we proposed automatic configuration of pre-developed model templates of these games, configured according to the level on the test, to be run with the model-checker software SPIN. The proposed method was run against the A* path finding algorithm in a maze solving game and a slightly modified Monte Carlo Tree Search algorithm in a game that has a low tolerance to sub-optimal actions. Also, two different models of the game Sokoban were run against each other to investigate the trade-offs that can be taken to scale a model for a PSPACE-complete, memory-intensive game.

Our results show that the mean performance of the proposed method converges to one of the best path finding algorithms in a simple maze game and considerably outperforms Monte Carlo Tree Search in a different game that includes an opponent. Also, our results show that the proposed method can be modified or optimized by making some trade-off decisions to scale for harder-to-solve games. Besides, our work includes a two-level cellular automata solution for generating maze-like levels.

Although this work's scope is on maze-like puzzle games, it can be applied to any game that can be modeled without specifying a game level.

To the best of our knowledge, this is the first study to propose a verification method for maze-like puzzle game levels using model checking. We showed that this method could successfully be used for verifying game levels, either automatically or manually generated.

In the future, we aim to use more complex linear temporal logic formulae, probabilistic model checkers, and salvage cloud computing to improve our work. Also, we aim to enlarge our pool of models by implementing models for different games.

REFERENCES

[1] O. Drageset, M. H. M. Winands, R. D. Gaina, and D. Perez-Liebana, "Optimising Level Generators for General Video Game AI," presented at the IEEE Conferenece on Games, London, UK, 2019.

[2] S. Iftikhar, M. Z. Iqbal, M. U. Khan, and W. Mahmood, "An automated model based testing approach for platform games," presented at the International Conference on Model Driven Engineering Languages and Systems, Ottawa, ON, Canada, 2015.

[3] L. Mugrai, F. Silva, C. Holmgard, and J. Togelius, "Automated Playtesting of Matching Tile Games," presented at the Conference on Games, London, UK, 2019.

[4] P. Garcia-Sanchez, A. Tonda, A. M. Mora, G. Squillero, and J. J. Merelo, "Automated playtesting in collectible card games using evolutionary algorithms: A case study in hearthstone," *Knowledge Based Systems,* vol. 153, p. 13, 2018.

[5] M. Khazeev, V. Rivera, M. Mazzara, and L. Johard. "Initial steps towards assessing the usability of a verification tool." (accessed 2021).

[6] P. Sarkar, "A brief history of cellular automata," *ACM Computing Surveys,* vol. 32, no. 1, pp. 80-107, 2000.

[7] S. Wolfram, *A New Kind of Science*. Wolfram Media Inc., 2002.

[8] D. Eppstein, "Growth and Decay in Life-Like Cellular Automata," *Game of Life Cellular Automata,* 2009.

[9] G. J. Holzmann, "The Model Checker SPIN," *Transactions on Software Engineering,* vol. 23, no. 5, pp. 279-295, 1997.

[10] G. J. Holzmann, "Basic Spin Manual," ed. Murray Hill, NJ, United States: AT&T Bell Laboratories.

[11] T. Schaul, "A Video Game Description Language for Model-based or Interactive Learning," presented at the Conference on Computational Intelligence in Games, Niagara Falls, Canada, 2013.

[12] R. Vereecken. "PyVGDL." https://github.com/rubenvereecken/py-vgdl (accessed 2021).

[13] O. Tekik. "PyVGDL, forked." https://github.com/iamonur/py-vgdl (accessed 2021).

[14] C. Baier and J.-P. Katoen, *Principles of Model Checking*. Cambridge, Massachusetts, United States: The MIT Presss, 2008.

[15] D. Buchs, S. Hostettler, A. Marechal, and R. M., "Alpina: An Algebraic Petri Net Analyzer," vol. 6015, ed. Lecture Notes in Computer Science: Springer, 2010, pp. 349-352.

[16] A. Khemiri and J. Pinaton, "Limiting state space explosion of model checking using discrete event simulation: combining DEVS and PROMELA," *Proceedings of the 2019 Summer Simulation Conference,* pp. 1-12, 2019.

[17] S. Radomski and T. Neubacher, "Formal verification of selected game-logic specifications," presented at the Workshop on Engineering and Interactive Computer Systems with SCXML, 2015.

[18] L. T. Holloway, "Modeling and Formal Verification of Gaming Storylines," Phd, University of Texas at Austin, 2016.

[19] A. Yacoub, M. E. A. Hamri, and C. Frydman, "DEv-PROMELA: modeling, verification, and validation of a video game by combining model-checking and simulation," *Simulation,* vol. 96, pp. 881-910, 2020.

[20] W. J. Kavanagh, A. Miller, G. Norman, and O. Andrei, "Balancing turn-based games with chained strategy generation," *IEEE Transactions on Games,* 2019.

[21] R. Rezin, I. Afanasyev, M. Mazzara, and V. Rivera, "Model Checking in Multiplayer Games Development," presented at the International Conference on Advanced Information Networking and Applications (AINA), 2018.

[22] B. Barroca, E. Marques, V. Balegas, A. Barisic, and V. Amaral, *The RPG DSL: a case study of language engineering using MDD for Generating RPG Games for Mobile Phones*. 2012.

[23] P. Milazzo, G. Pardini, D. Sestini, and P. Bove, "Case studies of application of probabilistic and statistical model checking in game design," *IEEE/ACM 4th International Workshop on Games and Software Engineering,* pp. 29-35, 2015.

[24] P. Moreno-Ger, R. Fuentes-Fernandez, J.-L. Sierra-Rodriguez, and B. Fernandez-Manjon, "Model-checking for adventure videogames," *Information and Software Technology,* vol. 51, pp. 564-580, 2009.

[25] L. Johnson, G. N. Yannakakis, and J. Togelius, "Cellular automata for real-time generation of infinite cave levels," *Proceedings of the 2010 Workshop on Procedural Content Generation in Games,* pp. 1-4, 2010.

[26] C. Adams and S. Louis, "Procedural maze level generation with evolutionary cellular automata," presented at the *IEEE Symposium Series on Computational Intelligence (SSCI)*, 2017.

[27] A. Pech, P. Hingston, M. Masek, and C. P. Lam, "Evolving cellular automata for maze generation," presented at the *Australasian conference on artificial life and computational intelligence*, 2015.

[28] Y. Macedo and L. Chaimowicz, "Improving procedural 2D map Generation based on multi-layered cellular automata and Hilbert curves," presented at the SBGames, 2017.

[29] T. Pavlic, A. Adams, P. Davies, and S. I. Walker, "Self-referencing cellular automata: A model of the evolution of information control in biological systems," ed, 2014.

[30] P. Povalej, P. Kokol, T. Welzer, and B. Stiglic, "Machine-Learning with Cellular Automata," presented at the Advances in Intelligent Data Analysis VI, 6th International Symposium on Intelligent Data Analysis, Madrid, Spain, 2005.

[31] A. Goyal, P. Mogha, R. Luthra, and N. Sangwan, "Path finding: A* or Dijkstra's?," *International Journal in IT and Engineering,* vol. 2, 2014.

[32] F. Marocchi and M. Crippa, "Monte Carlo Tree Search for Sokoban," MSc, Politecnico di Milano, 2017.

[33] D. Dor and U. Zwick, "Sokoban and other motion planning problems," *Computational Geometry,* vol. 13, no. 4, pp. 215-228, 1999.

[34] J. C. Culberson, "Sokoban is PSPACE-complete," The University of Alberta, 1997.

[35] A. G. Pereira, M. R. P. Ritt, and L. S. Buriol, "Finding Optimal Solutions to Sokoban Using Instance Dependent Pattern Databases," presented at the *Sixth Annual Symposium on Combinatorial Search*, 2013.

[36] A. Junghanns and J. Schaeffer, "Domain-dependent single-agent search enhancements," presented at the IJCAI, 1999.

[37] O. Tekik. "Main work for this thesis." https://github.com/iamonur/the_legendary_pipeline (accessed 2021).

[38] A. Garcia. "Alberto Garcia 1-1 Sokoban Level Set." https://www.sokobanonline.com/play/web-archive/alberto-garcia/1-1 (accessed 2021).

[39] J. Schell, *The Art of Game Design: A Book of Lenses*.

[40] P. H. Kim, J. Grove, S. Wurster, and R. Crawfis, "Design-centric maze generation," presented at the *International Conference on the Foundations of Digital Games*, 2019.

[41] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," in *Protocol Specification Testing and Verification*, 1995, pp. 3-18.

[42] Wikipedia. "Temporal Logic." Wikipedia. (accessed 03/10/2021, 2021).

[43] M. Camilli, C. Bellettini, L. Capro, and M. Mattia, "CTL Model Checking in the Cloud Using MapReduce," presented at the International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, 2014.

APPENDICES

APPENDIX A

VGDL Game Rules

Sokoban:

```
BasicGame
    SpriteSet
        floor > Immovable color=BROWN
        hole > Immovable color=DARKBLUE
        box > Passive color=RED
        wall > Immovable color=BLACK
        avatar > MovingAvatar color=WHITE
    TerminationSet
        SpriteCounter stype=box limit=0 win=True
        Timeout limit=1000000 win=False
    InteractionSet
        avatar wall > stepBack scoreChange=-100
        avatar EOS > stepBack
        avatar hole > stepBack scoreChange=-100
        avatar floor > nullEffect scoreChange=-1
        box EOS box > undoAll
        box avatar > bounceForward
        box wall box > undoAll
        box hole > killBoth scoreChange=-100
    LevelMapping
        1 > wall
        0 > floor
        B > box floor
        H > hole floor
        A > avatar floor
```

Maze:

```
BasicGame
    SpriteSet
        goalportal > Immovable color=GREEN
        wall > Immovable color=BLACK
        floor > Immovable color=BROWN
        players > MovingAvatar
            avatar > alternate_keys=True color=WHITE
    TerminationSet
        SpriteCounter stype=goalportal limit=0 win=True
    InteractionSet
        avatar EOS > stepBack
        avatar wall > stepBack scoreChange=-100
        floor avatar > NullEffect scoreChange=-1
        goalportal avatar > killSprite scoreChange=1000
    LevelMapping
        1 > wall
        G > goalportal
        A > avatar floor
        0 > floor
```

Race:

```
BasicGame
    SpriteSet
        goalportal > Immovable color=GREEN
        wall > Immovable color=BLACK
        floor > Immovable color=BROWN
        opponent > LookupChasear stype=goalportal color=RED
        players > MovingAvatar
            avatar > alternate_keys=True color=WHITE
    TerminationSet
        SpriteCounter stype=goalportal limit=0 win=True
        SpriteCounter stype=opponent limit=0 win=False
    InteractionSet
        avatar EOS > stepBack
        goalportal opponent > killSprite scoreChange=-1000
        avatar wall > stepBack scoreChange=-100
        floor avatar > NullEffect scoreChange=-1
        opponent wall > stepBack
        goalportal avatar > killsprite scoreChange=1000
        opponent avatar > stepBack
        avatar opponent > stepBack
```

```
LevelMapping
    1 > wall
    E > opponent floor
    G > goalportal
    A > avatar floor
    0 > floor
```

APPENDIX B

Template Game Models

All editable areas are marked with a '#'.

1. Sokoban, Navigational model

```
typedef row {
  byte a[#];
}
byte remaining_goals = #;
bit win = 0;
bit lose = 0;
row map[#];
proctype avatar_sokoban(int x; int y){
  map[x].a[y] = 2;
  byte w, a, s, d;
  do
  ::(win == 0) ->
    w = map[x].a[y-1];
    a = map[x-1].a[y];
    s = map[x].a[y+1];
    d = map[x+1].a[y];
```

```
if
:: (w != 1 && w != 4) -> //Cannot move into a wall or hole.
 atomic {
  printf("Avatar - W\\n");
  if
  :: w == 0 -> //Empty cell
   map[x].a[y] = 0;
   map[x].a[y - 1] = 2;
   y = y - 1
  :: w == 3 -> //A box
   if
   :: map[x].a[y - 2] == 1 -> //This is a wall, box cannot be pushed.
    skip
   :: map[x].a[y - 2] == 4 -> //This is a hole, so a hole is filled.
    map[x].a[y] = 0;
    map[x].a[y - 1] = 2;
    map[x].a[y - 2] = 0;
    y = y - 1;
    remaining_goals = remaining_goals - 1;
    if
    :: remaining_goals == 0 -> win = 1
    :: else -> skip
    fi
   :: map[x].a[y - 2] == 0 -> //This is a floor, a valid push.
```

```
      map[x].a[y] = 0;
      map[x].a[y - 1] = 2;
      map[x].a[y - 2] = 3;
      y = y - 1
     fi
    fi
   }
 :: (a != 1 && a != 4) -> //Cannot move into a wall or hole.
   atomic {
    printf("Avatar - A\\n");
    if
    :: a == 0 -> //Empty cell
     map[x].a[y] = 0;
     map[x-1].a[y] = 2;
     x = x - 1
    :: a == 3 -> //A box
     if
     :: map[x - 2].a[y] == 1 ->
      skip
     :: map[x - 2].a[y] == 4 ->
      map[x].a[y] = 0;
      map[x - 1].a[y] = 2;
      map[x - 2].a[y] = 0;
      remaining_goals = remaining_goals - 1;
```

```
        x = x - 1;
        if
        :: remaining_goals == 0 -> win = 1
        :: else -> skip
        fi
      :: map[x - 2].a[y] == 0 ->
        map[x].a[y] = 0;
        map[x - 1].a[y] = 2;
        map[x - 2].a[y] = 3;
        x = x - 1
      fi
    fi
  }
:: (s != 1 && s != 4) -> //Cannot move into a wall or hole.
  atomic {
    printf("Avatar - S\\n");
    if
    :: s == 0 -> //Empty cell
      map[x].a[y] = 0;
      map[x].a[y+1] = 2;
      y = y + 1
    :: s == 3 ->
      if
      :: map[x].a[y + 2] == 1 ->
```

```
        skip
      :: map[x].a[y + 2] == 4 ->
        map[x].a[y] = 0;
        map[x].a[y + 1] = 2;
        map[x].a[y + 2] = 0;
        remaining_goals = remaining_goals - 1;
        if
        :: remaining_goals == 0 -> win = 1
        :: else -> skip
        fi;
        y = y + 1
      :: map[x].a[y + 2] == 0 ->
        map[x].a[y] = 0;
        map[x].a[y + 1] = 2;
        map[x].a[y + 2] = 3;
        y = y + 1
      fi
    fi
  }
:: (d != 1 && d != 4) -> //Cannot move into a wall or hole.
  atomic {
    printf("Avatar - D\\n");
    if
    :: d == 0 -> //Empty cell
  map[x].a[y] = 0;
  map[x + 1].a[y] = 2;
  x = x + 1
:: d == 3 ->
  if
  :: map[x + 2].a[y] == 1 ->
    skip
  :: map[x + 2].a[y] == 4 ->
    map[x].a[y] = 0;
    map[x + 1].a[y] = 2;
    map[x + 2].a[y] = 0;
    remaining_goals = remaining_goals - 1;
    if
    :: remaining_goals == 0 -> win = 1
    :: else -> skip
    fi;
    x = x + 1
  :: map[x + 2].a[y] == 0 ->
    map[x].a[y] = 0;
    map[x + 1].a[y] = 2;
    map[x + 2].a[y] = 3;
    x = x + 1
  fi
fi
```

```
      }
    fi
  :: else -> break
  od;
  printf("Avatar - Win \\n")
}

init {
  # //Walls configured here
  # //Boxes configured here
  # //Holes configured here
  run avatar_sokoban(#,#);
}}

ltl  { [] !win };
```

2. Sokoban, Push-based model

All editable areas are marked with a ‘#’.

```
c_code{#include "../spin/sokoban2.c"};
typedef row {
  byte a[#];
```

```
}
byte remaining_goals = #;
bit win = 0;
bit map_inited = 0;
int choices[#];
row map[#];

proctype map_init() {
  int i;
  int ii;
  for (i : 0 .. #) {
    map[i].a[0] = 1;
    map[i].a[#] = 1;
  }
  for (i : 0 ..{width}) {
    map[0].a[i] = 1;
    map[#].a[i] = 1;
  }
  for (i : 1 .. #) {
    for (ii : 1 .. #) {
      map[i].a[ii] = 0;
    }
  }
  #//Walls are added here
```

```
 #//Boxes are added here
 #//Holes are added here
 map_inited = 1;
}
proctype avatar_sokoban(int x; int y) {
 map[x].a[y] = 2;
 int last_move = -2;
 c_code{sokoban_init();};
 do
  ::(win != 1) ->
   if
   #//All choices are added here.
   fi;
   if
   :: (remaining_goals == 0) -> win = 1; break
   :: else -> skip
   fi
  :: else -> break
 od;
 printf("Won\\n");
}
init {
 atomic {run map_init();};
 run avatar_sokoban(#, #);
```

```
}
ltl { [] !(win) };
```

3. Maze model

Editable areas are marked with a '#'

```
typedef row{
  byte a[#]
  }
  bit win = 0;
  row map[#];
proctype avatar_mazesolver(int x; int y){
    map[x].a[y] = 2;
    byte w, a, s, d;
    bit foo;
    do
    ::(win == 0) ->
        w = map[x].a[y-1];
        a = map[x-1].a[y];
        s = map[x].a[y+1];
        d = map[x+1].a[y];
        if
        :: w != 1 -> printf("Avatar - W\\n");
            if
  :: w == 0 ->
     map[x].a[y] = 0;
     map[x].a[y-1] = 2;
     y = y - 1
  :: w == 3 ->
     win = 1
  fi;
:: a != 1 -> printf("Avatar - A\\n");
  if
  :: a == 0 ->
     map[x].a[y] = 0;
     map[x-1].a[y] = 2;
     x = x - 1
  :: a == 3 ->
     win = 1
  fi;
:: s != 1 -> printf("Avatar - S\\n");
  if
  :: s == 0 ->
     map[x].a[y] = 0;
     map[x].a[y+1]=2;
     y = y + 1
  :: s == 3 ->
     win = 1
```

```
        fi;
      :: d != 1 -> printf("Avatar - D\\n");
        if
        :: d == 0 ->
          map[x].a[y] = 0;
          map[x+1].a[y] = 2;
          x = x + 1
        :: d == 3 ->
          win = 1
        fi
      fi
    :: else -> break
    od;
    printf("Avatar - Win\\n")
}

init{
    int i, ii;
    for (i : 0 .. #) {
      map[i].a[0] = 1;
      map[i].a[#] = 1;
    }
    for (i : 0 .. #) {
      map[0].a[i] = 1;
```

```
        map[#].a[i] = 1;
    }
    for (i : 1 .. #) {
        for (ii : 1 .. #) {
            map[i].a[ii] = 0;
        }
    }
    # //Generic placement of the walls.
    map[#].a[#] = 3; // Portal placement
    run avatar_mazesolver(#, #);
}
ltl  { [] !win };
```

4. Race model

```
typedef row{ //This is for creating 2-D arrays, since they are not supported.
    byte a[#];
}
c_code{#include "../spin/bfs.c"};
bit win  = 0;
bit dead = 0;
chan avatar_turn = [0] of {bit}; //rendez-vous at (start of avatar, end of opponent).
chan opponent_turn = [0] of {bit}; // The opposite rendezvous
row map[#]; //This is your 2-D array.
```

```
int next_x;
int next_y;
proctype avatar_same_goal(int x; int y){
  map[x].a[y] = 2;
  byte w, a, s, d;
  bit foo;
  do
  ::((win == 0) && (dead == 0)) ->
          avatar_turn ? foo;
          if
          :: dead == 1 -> break
          :: else -> skip
          fi;
          w = map[x].a[y-1];
          a = map[x-1].a[y];
          s = map[x].a[y+1];
          d = map[x+1].a[y];
          if
          :: ((w != 1) && (w != 4)) -> printf("Avatar - W\\n");
                  if
                  :: w == 0 ->
                          map[x].a[y] = 0;
                          map[x].a[y-1] = 2;
                          y = y - 1
```

```
        :: w == 3 ->
                win = 1
        fi
:: ((a != 1) && (a != 4)) -> printf("Avatar - A\\n");
        if
                :: a == 0 ->
                        map[x].a[y] = 0;
                        map[x-1].a[y] = 2;
                        x = x - 1
                :: a == 3 ->
                        win = 1
                fi
        :: ((s != 1) && (s != 4)) -> printf("Avatar - S\\n");
                if
                :: s == 0 ->
                        map[x].a[y] = 0;
                        map[x].a[y+1] = 2;
                        y = y + 1
                :: s == 3 ->
                        win = 1
                fi
        :: ((d != 1) && (d != 4)) -> printf("Avatar - D\\n");
                if
                :: d == 0 ->
```

```
                                        map[x].a[y] = 0;
                                        map[x+1].a[y] = 2;
                                        x = x + 1
                                  :: d == 3 ->
                                        win = 1
                      fi
            fi;
            opponent_turn ! foo;
:: else -> break
od;
if
:: (win == 1) -> printf("Avatar - Win\\n"); opponent_turn ! foo;
:: (dead == 1) -> printf("Avatar - Dead\\n"); opponent_turn ! foo;
fi;
}

proctype opponent_same_goal(int x; int y; int xx; int yy){
  map[x].a[y] = 4;
  bit foo;
  do
   :: (win == 0 && dead == 0) ->
      opponent_turn ? foo;
      if
      :: win == 1 -> break
```

```
    :: else -> skip
    fi;

    c_code{calculate_next_move_to_portal_avatar_blocks(Popponent_same_goal->x,
Popponent_same_goal->y, &(now.next_x), &(now.next_y));};
    map[x].a[y] = 0;
    foo = map[next_x].a[next_y];
    map[next_x].a[next_y] = 4;
    x = next_x;
    y = next_y;
    if
    :: foo == 3 -> dead = 1; break
    :: else -> skip
    fi;
    avatar_turn ! 0
  :: else -> break
  od;
  if
  :: win == 1 -> c_code{printf("Opponent - Win\\n");}; avatar_turn ! 0
  :: dead == 1 -> c_code{printf("Opponent - Dead\\n");}; avatar_turn ! 1
  fi;
}

init{
  int i, ii;
  for (i : 0 .. #) {
   map[i].a[0] = 1;
   map[i].a[#] = 1;
  }
  for (i : 0 .. #) {
     map[0].a[i] = 1;
     map[#].a[i] = 1;
  }
  for (i : 1 .. #) {
     for (ii : 1 .. #) {
         map[i].a[ii] = 0;
     }
  }
  # //Generic placement of walls
  map[#].a[#] = 3; //Place portal
  run avatar_same_goal(#, #);
  run opponent_same_goal(#, #, #, #);
}
```

APPENDIX C

Sokoban Push-Based Model : Helper C Code

```
#ifndef NUM_OF_BOXES
#define NUM_OF_BOXES 1
#endif

#ifndef MAX_LEN
#define MAX_LEN 26
#endif

#define NUM_OF_CHOICES NUM_OF_BOXES*4

char sokoban_map[MAX_LEN][MAX_LEN];
char sokoban_map2[MAX_LEN][MAX_LEN];

unsigned int sokoban_avatar_1, sokoban_avatar_2;

//Used for debugging purposes
void soko_map_print(){
    for(int i = 0; i < MAX_LEN; i++)
    {
        for(int j = 0; j < MAX_LEN; j++){
            fprintf(stderr, "%c",sokoban_map[i][j]);
        }
        fprintf(stderr, "\n");
    }
}

//Copies the map from the model checker.
void sokoban_putMapToMem() {
  unsigned int i, j;
  for(i = 0; i < MAX_LEN; i++) {
   for(j = 0; j < MAX_LEN; j++) {
     switch(now.map[i].a[j]){
       case 0:
         sokoban_map[i][j] = ' ';
       break;
       case 1:
         sokoban_map[i][j] = 'w';
       break;
       case 2:
         sokoban_map[i][j] = 'a';
         sokoban_avatar_1 = i;
         sokoban_avatar_2 = j;
       break;
       case 3:
         sokoban_map[i][j] = 'b';
       break;
       case 4:
         sokoban_map[i][j] = 'h';
       break;
       default:
         exit(0);
     }
   }
  }

}

//Mark the places avatar can go without pushing.
void sokoban_recurseAvatarizedMap(int avatar_f, int avatar_s) {
  sokoban_map2[avatar_f][avatar_s] = 'a';
  if( sokoban_map2[avatar_f - 1][avatar_s] == ' ')
    sokoban_recurseAvatarizedMap(avatar_f - 1, avatar_s);
  if( sokoban_map2[avatar_f + 1][avatar_s] == ' ')
    sokoban_recurseAvatarizedMap(avatar_f + 1, avatar_s);
  if( sokoban_map2[avatar_f][avatar_s - 1] == ' ')
    sokoban_recurseAvatarizedMap(avatar_f, avatar_s - 1);
  if( sokoban_map2[avatar_f][avatar_s + 1] == ' ')
    sokoban_recurseAvatarizedMap(avatar_f, avatar_s + 1);
  return;
}

//Calls the function above.
void sokoban_getAvatarizedMap() {
  memcpy(sokoban_map2, sokoban_map, MAX_LEN * MAX_LEN);
  sokoban_recurseAvatarizedMap(sokoban_avatar_1, sokoban_avatar_2);
}

//Updates the choices matrix of the model checker
void sokoban_updateTarget(int coor_1, int coor_2, int box_num) {
  if(sokoban_map2[coor_1 - 1][coor_2] == 'a')
    switch(sokoban_map2[coor_1 + 1][coor_2]){
      case 'a':
      case ' ':
      case 'h':
        now.choices[box_num*4 + 0] = 1;
      break;
      default:
        now.choices[box_num*4 + 0] = 0;
    }
  else now.choices[box_num*4 + 0] = 0;

  if(sokoban_map2[coor_1][coor_2 - 1] == 'a')
    switch(sokoban_map2[coor_1][coor_2 + 1]){
      case 'a':
      case ' ':
      case 'h':
        now.choices[box_num*4 + 1] = 1;
      break;
      default:
        now.choices[box_num*4 + 1] = 0;
    }
  else now.choices[box_num*4 + 1] = 0;

  if(sokoban_map2[coor_1 + 1][coor_2] == 'a')
    switch(sokoban_map2[coor_1 - 1][coor_2]){
      case 'a':
      case ' ':
      case 'h':
        now.choices[box_num*4 + 2] = 1;
      break;
      default:
        now.choices[box_num*4 + 2] = 0;
    }
  else now.choices[box_num*4 + 2] = 0;

  if(sokoban_map2[coor_1][coor_2 + 1] == 'a')
    switch(sokoban_map2[coor_1][coor_2 - 1]){
      case 'a':
      case ' ':
      case 'h':
        now.choices[box_num*4 + 3] = 1;
      break;
      default:
        now.choices[box_num*4 + 3] = 0;
    }
  else now.choices[box_num*4 + 3] = 0;
  return;
}

//Updates all targets.
void sokoban_updateAllTargets(){

  for(unsigned int a = 0; a < NUM_OF_BOXES*4; a++) {
    now.choices[a] = 0;
  }

  sokoban_getAvatarizedMap();

  unsigned char box_number = 0;
  for(unsigned int i = 0; i < MAX_LEN; i++) {
    for(unsigned int j = 0; j < MAX_LEN; j++) {
      if(sokoban_map2[i][j] == 'b') {
        sokoban_updateTarget(i,j,box_number);
        box_number++;
      }
    }
  }
  now.remaining_goals = box_number;
}
//Pushing a box, called from the model checker.
void push_a_box(unsigned int ind1, unsigned int ind2, unsigned int pos) {
  now.map[ind1].a[ind2] = 2;
  now.map[sokoban_avatar_1].a[sokoban_avatar_2] = 0;
  printf("Push@ %d %d ; Side: %d\n", ind1-1, ind2-1, pos);
  switch(pos) {
    case 0:
      switch(sokoban_map[ind1 + 1][ind2]) {
        case 'a':
        case ' ':
          now.map[ind1 + 1].a[ind2] = 3;
        break;
    case 'h':
      now.map[ind1 + 1].a[ind2] = 0;
      now.remaining_goals--;
    break;
    default:
      exit(0);
  }
break;
case 1:
  switch(sokoban_map[ind1][ind2 + 1]){
    case 'a':
    case ' ':
      now.map[ind1].a[ind2 + 1] = 3;
    break;
    case 'h':
      now.map[ind1].a[ind2 + 1] = 0;
      now.remaining_goals--;
    break;
    default:
      exit(0);
  }
break;
case 2:
  switch(sokoban_map[ind1 - 1][ind2]){
    case 'a':
    case ' ':
      now.map[ind1 - 1].a[ind2] = 3;
    break;
    case 'h':
      now.map[ind1 - 1].a[ind2] = 0;
      now.remaining_goals--;
    break;
    default:
      exit(0);
  }
break;
case 3:
  switch(sokoban_map[ind1][ind2 - 1]) {
    case 'a':
    case ' ':
      now.map[ind1].a[ind2 - 1] = 3;
    break;
    case 'h':
      now.map[ind1].a[ind2 - 1] = 0;
```

```
          now.remaining_goals--;
        break;
        default:
          exit(0);
      }
    break;
    default:
      exit(0);
  }
}
//Pushing a box, called from the model checker.
void push(int choice){
  unsigned int box = choice/4;
  unsigned int side = choice%4;
  for(unsigned int i = 0; i < MAX_LEN; i++){
    for(unsigned int j = 0; j < MAX_LEN; j++){
      if(sokoban_map[i][j] == 'b') {
        if(box == 0){
          push_a_box(i, j, side);
          sokoban_putMapToMem();
          return;
        }
        else box--;
      }
    }
  }
}
//First, call this
void sokoban_init(){
  sokoban_putMapToMem();

  sokoban_updateAllTargets();
}

//The entry point of the call from the model checker
void sokoban_push(unsigned int choice){
  sokoban_putMapToMem();
  push(choice);
  sokoban_updateAllTargets();
}
```

TEZ İZİN FORMU / THESIS PERMISSION FORM

ENSTİTÜ / INSTITUTE

Fen Bilimleri Enstitüsü / Graduate School of Natural and Applied Sciences	
Sosyal Bilimler Enstitüsü / Graduate School of Social Sciences	
Uygulamalı Matematik Enstitüsü / Graduate School of Applied Mathematics	
Enformatik Enstitüsü / Graduate School of Informatics	X
Deniz Bilimleri Enstitüsü / Graduate School of Marine Sciences	

YAZARIN / AUTHOR

Soyadı / Surname : Tekik
Adı / Name : Onur
Bölümü / Department : Bilişim Sistemleri / Information Systems

TEZİN ADI / TITLE OF THE THESIS (**İngilizce** / English) : VERIFYING MAZE-LIKE GAME LEVELS WITH MODEL CHECKER SPIN

TEZİN TÜRÜ / DEGREE: **Yüksek Lisans** / Master [X] **Doktora** / PhD [ ]

1. **Tezin tamamı dünya çapında erişime açılacaktır. /** Release the entire work immediately for access worldwide. [ ]
2. **Tez iki yıl süreyle erişime kapalı olacaktır.** / Secure the entire work for patent and/or proprietary purposes for a period of **two year**. * [ ]
3. **Tez altı ay süreyle erişime kapalı olacaktır.** / Secure the entire work for period of **six months**. * [X]

***** *Enstitü Yönetim Kurulu Kararının basılı kopyası tezle birlikte kütüphaneye teslim edilecektir.*
A copy of the Decision of the Institute Administrative Committee will be delivered to the library together with the printed thesis.

Yazarın imzası / Signature Onur Tekik **Tarih** / Date 08/11/2021