



UNIVERSIDAD DE TALCA
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA EN DESARROLLO DE
VIDEOJUEGOS Y REALIDAD VIRTUAL

**Comparison of General Frameworks for Puzzle
Games**

ÁLVARO NICOLÁS SOTO MISSENE

Profesor Guía: NICOLÁS ARTURO BARRIGA RICHARDS

Memoria para optar al título de
Ingeniero en Desarrollo de Videojuegos y Realidad Virtual

Talca – Chile
January, 2026



UNIVERSIDAD DE TALCA
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA EN DESARROLLO DE
VIDEOJUEGOS Y REALIDAD VIRTUAL

**Comparison of General Frameworks for Puzzle
Games**

ÁLVARO NICOLÁS SOTO MISSENE

Profesor Guía: NICOLÁS ARTURO BARRIGA RICHARDS

Memoria para optar al título de
Ingeniero en Desarrollo de Videojuegos y Realidad Virtual

El presente documento fue calificado con nota: _____

Talca – Chile

January, 2026

TABLE OF CONTENTS

	Page
Table of Contents	i
List of Figures	iii
List of Tables	iv
Resumen	v
Abstract	vi
1 Introduction	1
1.1 Context of the Problem	1
1.2 Context of the Solution	2
1.3 Research Question and Objectives	2
1.3.1 General Objectives	3
1.3.2 Specific Objectives	3
1.4 Scope	3
2 Theoretical Framework	4
2.1 Puzzle games, generation and solvers	4
2.1.1 Sliding puzzles	4
2.1.2 Sokoban-type puzzles	6
2.2 General Game Playing	9
2.2.1 Metagame	9
2.2.2 General Game Description Language	10
2.2.3 Digital Ludeme Project	11
2.3 State of the Art	11
2.3.1 Ludii	12
2.3.2 Regular BoardGames	12
2.3.3 PyVGDL	13
2.3.4 PuzzleScript	14

3	Experiment	16
3.1	Solution	16
3.2	Frameworks used and Selected puzzles	17
3.3	Expected Results	17
4	Implementation	19
4.1	Implementation of the puzzle games on the selected frameworks . . .	19
4.1.1	PuzzleScript	20
4.1.2	Ludii	22
4.1.3	PyVGDL	24
4.2	Level generators on the selected frameworks	26
4.2.1	Puzzlescript	27
4.2.2	Ludii	28
4.2.3	PyVGDL	29
4.3	Solvers used for the selected puzzles	29
4.3.1	Puzzlescript	29
4.3.2	Ludii	30
4.3.3	PyVGDL	30
5	Results	31
5.1	Comparison Between the frameworks	31
5.2	Comparison between all generators on the 3 frameworks	33
5.3	Comparison between the puzzle solvers	35
6	Conclusions and final thoughts	38
	Bibliography	39
A:	Code Excerpts from the Implementations	41
B:	Levels used for testing solvers	48

LIST OF FIGURES

	Page
1.1 Map of puzzle game categories by variability.	2
2.1 Computation time comparison.	6
2.2 Separating a 15-puzzle into subsets using a Pattern.	7
2.3 Generation of the map for a Sokoban puzzle.	8
2.4 Representation of the Monty Hall game by using keywords in GDL. .	11
4.1 Example level of Sokoban in Puzzlescript.	21
4.2 Example level of Rush Hour in Puzzlescript.	21
4.3 Example level of Sokoban in Ludii.	23
4.4 Example level of Rush Hour in Ludii.	24
4.5 Example level of Sokoban on PyVGDL.	25
4.6 Example level of Rush Hour in PyVGDL.	25
5.1 9 of the 100 Sokoban levels first generated.	33
B.1 The five Sokoban levels used to test the solvers.	49
B.2 The three Rush Hour levels used to test the solvers.	49

LIST OF TABLES

	Page
5.1 Comparison table between the game implementations	32
5.2 Timing comparison for generating levels	34
5.3 Comparison table for Sokoban level solving on PuzzleScript Next . . .	35
5.4 Comparison table for Sokoban level solving on Ludii	36
5.5 Comparison table for Sokoban level solving on PyVGDL	36
5.6 Comparison table for Rush Hour level solving on PuzzleScript Next .	36
5.7 Comparison table for Rush Hour level solving on Ludii	37
5.8 Comparison table for Rush Hour level solving on PyVGDL	37

RESUMEN

La Generación de Contenido Procedural (PCG en Inglés) se refiere a la creación de contenido a través de algoritmos en vez de crearlos de manera manual. Los beneficios más comunes que trae en el ámbito de los videojuegos es la rejugabilidad que ofrece gracias a que los niveles generados no sean siempre los mismos y la reducción de costo al no tener que crearlos manualmente. Generalmente, juegos con generación de contenido procedural tienen su propio software diseñado específicamente con el juego en que son implementados en mente. Una excepción a esto son los programas que usan Generación General de Videojuegos. Aunque este tema ha sido abordado e investigado en múltiples proyectos, ha sido con otros frameworks o enfocado en aspectos específicos.

En esta memoria se va a comparar y analizar algunos de los frameworks utilizados para Generación General de Videojuegos utilizando un par de juegos de puzle. Usando la metodología experimental, se implementará un generador de niveles simple y un solver por cada juego de puzle seleccionado en cada uno de los frameworks elegidos con el propósito de comparar su desempeño. Las implementaciones serán juzgadas por su rendimiento al generar niveles nuevos, el rendimiento al resolver un nivel de los juegos implementados y la dificultad de implementar todo lo anterior con el propósito de encontrar cuál de los frameworks escogidos es mejor para el propósito de generación y resolución general de juegos de puzles. Esperamos que con los resultados de esta investigación se pueda llegar a un framework eficiente que pueda ser utilizado para mayor y mejor implementación de una gran variedad de juegos de puzle. Esta solución entregaría beneficios para probar nuevas ideas de juegos de puzle, crear una gran variedad de desafíos para juegos de puzle existentes y para generar niveles para variaciones nuevas de juegos de puzle existentes, por nombrar algunos ejemplos.

ABSTRACT

Procedural Content Generation (PCG) refers to creating content through algorithms rather than fully manually. The most common benefits of PCG in regards to video games are replayability thanks to the generated levels not being the same and cost reduction thanks to not having to manually create them. Generally, games with procedural content generation have their own custom-built software specific to the game to generate content properly. One of the few exceptions to this is the work that uses General Video Game Generation. While this subject and others like it have been researched and worked on in multiple projects, it's been with different frameworks or focused on specific aspects of it.

In this thesis we will analyze and compare some of the frameworks used for General Video Game Generation using a couple of chosen puzzle games. Using an experimental methodology we will implement a simple level generator and a puzzle solver for each of the puzzle games chosen on each framework to be reviewed and compared. All the implementations will be judged by their performance when they generate a new instance of a puzzle game, their performance when solving a level of a puzzle game, and the difficulty in implementing the generator and solver with the purpose of finding which framework would work best for the purpose of General Puzzle Game Generation and Solving. We hope that with the results of this investigation we can arrive at an efficient puzzle game framework that can be used for better and easier implementation of a great variety of puzzle games, old or otherwise. This solution would bring benefits for testing out new puzzle game ideas, making a great variety of challenges for an existing puzzle game, making levels for a new variation on an existing puzzle game, to mention a few examples.

1. Introduction

Procedural content generation (PCG) has been the subject of research and development for decades, and it has been used in video games since the 80s. PCG, being able to generate the desired content through pre-defined algorithms and a bit of randomness can generate something different everytime it's used, giving a new experience to the player everytime he plays. Aside from just content generation, AI research and algorithms have also been used to make software simulate playing games with the goal of beating them, which works well for both making Video game AI, and with the purpose of trying to make smarter AI compete on the same level as humans on different games.

1.1 Context of the Problem

While research into procedurally generated or algorithmic content for video games has been done extensively, these endeavors usually focus on one specific side of it (Game Level Generation [1], Game Rules Generation [2], Game Playing [3]) or tend to be software programmed for one specific game only. This makes creating new starting instances of different games a chore, as it requires different software tuned specifically for each game, rather than having one take care of them all. Since constructing a framework that would be able to work with every kind of game would be a daunting task, we will focus on puzzle games. To be more specific, we want a framework that will be able to create simple initial instances of puzzle games and then be able to solve these. Using the map of simple categories shown in fig.1.1 defined by De Kegel and Haahr [4], we will pick puzzle games from categories with low variability, such as sliding puzzles and Sokoban-type puzzles.

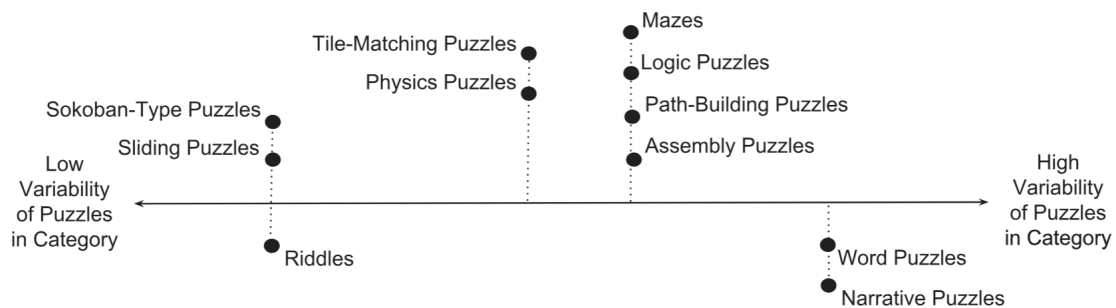


Figure 1.1: Map of puzzle game categories as defined by De Kegel and Haahr [4].

1.2 Context of the Solution

To find the best option for the problem at hand, we will investigate several frameworks that are used for general game generation or playing (like GVG-AI [1], PyVGDL [5] or Ludii [6] for example) and frameworks used to focus on specific puzzle games to choose a handful that could work as candidates to compare. Aside from choosing frameworks, we will pick some 2D top-down puzzle games to use in the comparison.

On each of the chosen frameworks we implement a very simple level generator for the selected puzzle games. In this step it's important to note which framework makes the implementation of all the puzzle games tougher, and which framework is the easiest one to implement different games. This will be evaluated with a questionnaire in regards to how complex the process is. Then, the performance of each framework will be reviewed and compared at the moment of generating a starting instance of a game, making sure to compare every game with the other frameworks attempt at making levels of that same game.

For each level generator implemented on the different frameworks, a puzzle solver must be implemented too. The last comparisons consist on reviewing the performance of every framework and comparing them to find the most efficient solver in several of the games.

1.3 Research Question and Objectives

For this investigation, the research question is “Which is the most efficient framework for generating and solving multiple different puzzle games?”. In the following sections we list our objectives to be able to approach an answer for this question.

1.3.1 General Objectives

The main objective is finding among the selected frameworks for this study, the one framework that is most efficient for both handling level generation and level solving different puzzle games while keeping low implementation costs and effort. This will be done using an experimental methodology, selecting only a few frameworks and 2 puzzle games to be tested and compared.

1.3.2 Specific Objectives

1. Survey existing frameworks and select 2 to 4 candidates to be compared.
2. Choose 2 very simple puzzle games to be implemented. The stated simplicity of these games is based on the categories defined in [4], where puzzles from simpler puzzle games have less variability and tend to be more similar to each other than puzzles from more complex puzzle games.
3. On the chosen frameworks, implement basic generators and solvers for 2 different puzzle games on each of them and then compare the difficulty of implementation on each of them.
4. Compare the performance between the execution of the puzzle solvers for each game on each framework.
5. Compare the difficulty to generate new levels in each of the frameworks.

1.4 Scope

This investigation plans to use 2 to 4 frameworks with very simple generators, possibly random. Each framework will have 2 puzzle games implemented, so that it can solve instances generated of these games. The games will be puzzles of low variability like Sokoban or sliding puzzles.

The implementation of games such as tile-matching puzzles or mazes will be left for possible future work.

2. Theoretical Framework

For the theoretical framework, we start by looking at puzzle games chosen by their simplicity in terms of how varied the method for solving them is, looking at generators and solvers developed for these. Then we talk about General game playing and how it's progressed throughout the years, before arriving at the frameworks that are used nowadays for GGP.

2.1 Puzzle games, generation and solvers

On this section we take a look at the types of puzzle games chosen, along with how initial states for these are generated and algorithms used to solve them.

2.1.1 Sliding puzzles

Sliding puzzles involve moving items or tiles toward goal positions in a constrained grid-based space. Items on the grid can be moved in any free direction and usually all moves are reversible which encourages exploration of all the alternatives in the space [4]. Well-known examples of these kind of puzzles are the 15-Puzzle and Rush Hour. Both had their origins in a physical medium before being digital.

Block-sliding puzzles could be generated by starting from the end configuration and working backwards, though the number of moves used to play backwards is not always as small as the shortest path so it's not a good metric for difficulty [4].

For the sake of Rush Hour solvers, we could consider the problem a node graph, where each node is a possible combination or rather a different state the puzzle can be in. Each two nodes are related if we can get from one state to the other by moving one car one block. The research done in [7] tries a Breadth-First search and notes

the inefficiency of it as a solver, as it explores irrelevant positions and takes a while. This is then changed to a Greedy Best First Search algorithm using a heuristic to determine how close a grid is to being solved: shown in Algorithm 2.1 is an algorithm to determine the heuristic score in this scenario, the less blocked the red car is in the parking lot the better the score will be (lower number, with 0 being a solved grid).

Algorithm 2.1 Heuristic score used

```

1: BlockingCars = [ ]
2: for each Car between RedCar and exit do
3:   BlockingCars.add(Car)
4: end for
5: while BlockingCars changes do
6:   for Car in BlockingCars do
7:     for OtherCar in Board do
8:       if OtherCar blocks Car and OtherCar not in BlockingCars then
9:         BlockingCars.add(OtherCar)
10:      end if
11:    end for
12:  end for
13: end while
14: Score = length(BlockingCars)
15: return Score

```

In figure 2.1 we see 2 graphs comparing the computation time using a breadth-first search and the greedy best first search.

The N-puzzle (commonly known as 15-puzzle but could be 8-puzzle or another number depending on the size chosen for it) consists of a board with numbered tiles, for example 15 numbered tiles on a 4x4 board for the 15-puzzle where the objective is to get each tile to it's goal position which is usually ascending order from left to right, top to bottom. Having 15 tiles in this 4x4 space, the player progresses by moving tiles adjacent to the empty square until they can move all tiles to their goal.

Research on these puzzles was usually done using breadth-first search and deriving heuristic methods for informed search methods such as the A* algorithm. Since a 15-puzzle space will have a 10^{12} node space a good heuristic is needed to solve these efficiently [8]. While A* and IDA* (Iterative Deepening A*) are always capable of solving the N-puzzle problem, the time required for it is increased exponentially when the N value raises.

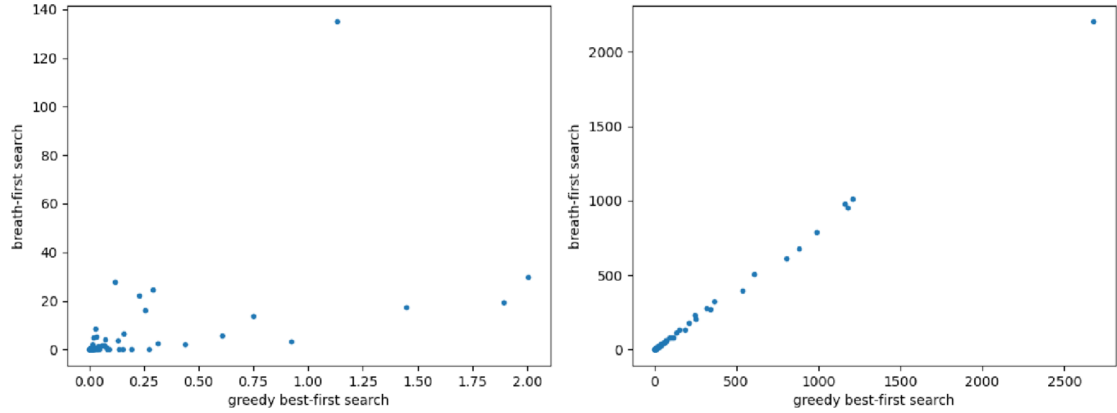


Figure 2.1: Computation time comparison. Solvable levels on the left, unsolvable levels on the right [7].

The ABC algorithm, which can be seen in Algorithm 2.2, simulates bees grouped into three types (employed bees, onlooker bees and scout bees) is a meta-heuristic optimization algorithm that is used for different kinds of optimization problems [8]. In this algorithm the position of a food source corresponds to a candidate solution, and the nectar amount represents the quality of the related solution. The employed bees are responsible for searching the food source around the hive and evaluate its quality, then share the information with the onlooker bees which choose good food sources based on this information, then if the nectar in a food source is depleted the bee of that source becomes a scout bee and looks for new food randomly.

Another method is a pattern, which is the partial specification of an N-puzzle and is based on dividing the N-puzzle into subsets and solving each subset separately [9] as shown in Figure 2.2.

2.1.2 Sokoban-type puzzles

Sokoban-type puzzles are based on the 1982 Japanese game Sokoban, where the objective is to make the player character push all the objects to the designated goal areas in the map. A Sokoban board is made from a grid where the cells can be either “walls” or “passways”, the pathway cells can either be occupied by an object, by the player or be empty. The player has two operations available to him: MOVE (move the player character one step toward the four directions available) and PUSH (move

Algorithm 2.2 Fundamental steps of the modified ABC algorithm from [8].

Initialize the population of solutions x_{ij} , $i = 1, 2, \dots, SN$, $j = 1, 2, \dots, D$

Evaluate the fitness fit_i of the population

set cycle to 1

Repeat

 For each employed bee {

 Produce new solution v_i by (4) and calculate fit_i

 Apply the greedy selection between x_i and v_i

 }

 For each onlooker bee {

 Calculate the probability values p_i for the solution x_i by (5)

 Select a solution x_i depending on p_i

 Produce new solution v_i and calculate fit_i

 Apply the greedy selection process between x_i and v_i

 }

 If there is an abandoned solution for the scout

 Then replace it with a new produced solution by (1)

 Memorize the best solution obtained so far

 cycle = cycle + 1

Until cycle = MCN

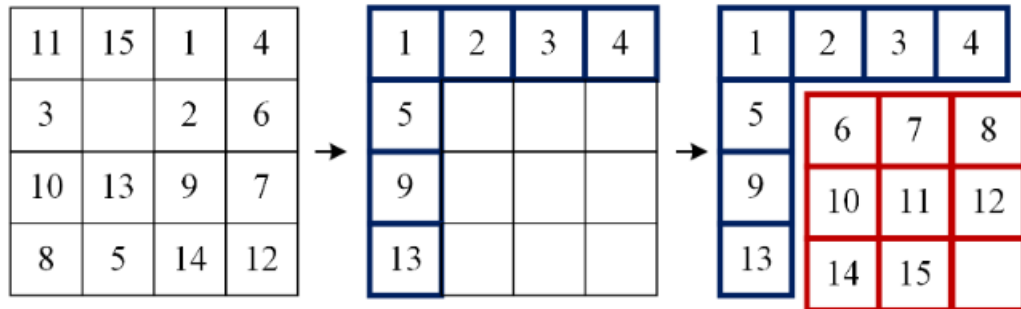


Figure 2.2: Separating a 15-puzzle into subsets using a Pattern [8].

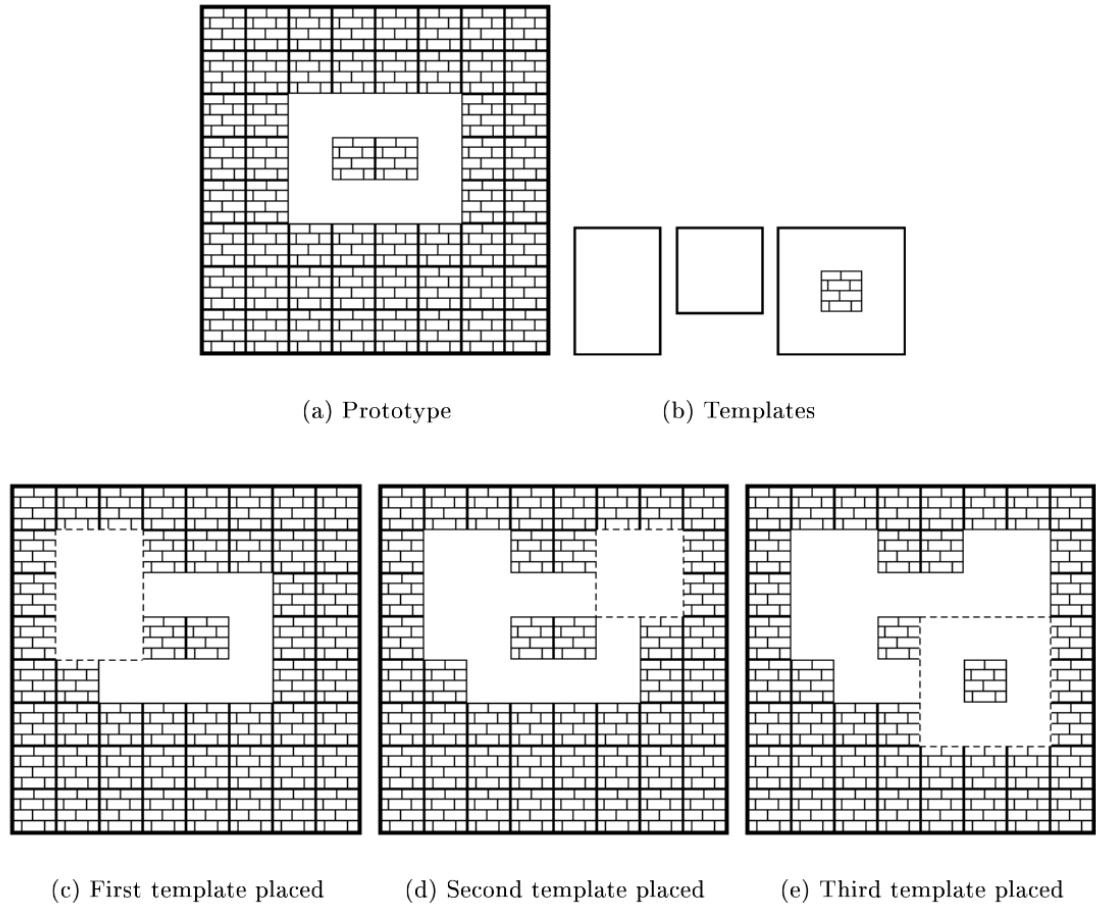


Figure 2.3: Generation of the map for a Sokoban puzzle [10].

an object one step forward by making the character push it from behind) [10].

A generator for this kind of puzzle is expected to generate original problems, generate problems with a legal solution, and make interesting puzzles. The program used in [10] uses a few templates made out of blocks of map made of wall and passway of size 2x2 or bigger and overlaps them with a base map as shown in figure 2.3.

Next, goal areas are chosen on passway cells, cells where the player cannot push his box into are marked as goal-avoid and cannot be chosen as goal areas. Then the objects are placed into the passway area, making sure not to create any “dead end area” which is an area from which objects can’t be moved out of. Goal areas that are in the corner are marked as such and then a goal-range is calculated (areas where the player can push the box from into the goal area). Then the object placement begins: the first object is placed in the goal-range, dead end areas are calculated and

avoided, then the second object is placed in the goal-range too, avoiding being the second needing to pass through a corner goal, this process iterates until all objects are placed.

2.2 General Game Playing

Games of strategy allow us to exercise and improve our intellectual skills by playing and challenging ourselves. This characteristic should make them a good tool to train artificial intelligence to become smarter as it wins more games. Unfortunately the value of building programs to play a specific game is pretty limited. These programs are not smart in the sense that they would be able to pick up other games they weren't specifically designed for and most of the heavy-lifting of analyzing and figuring out how to play properly is done in advance by the programmers.

General game playing (GGP) tackles both of these problems as general game players are systems able to accept descriptions of games at run time and play those games effectively without intervention.

2.2.1 Metagame

In 1992, the concept of Meta-Game Playing is devised by Barney Pell, motivated by the idea of controlling and eliminating human bias when designing an Artificial Intelligence to play an existing game [11]. Rather than being designed to play a game known in advance, the program only knows the class of game beforehand, and takes the rules of the game as input before playing.

In the work done for general chess-like games [11], Metagame is given the components necessary for playing this kind of game. In the paper, symmetric chess-like games are defined informally as “a two-player game of perfect information in which the two players move pieces along specified directions, across rectangular boards. Different pieces have different powers of movement, capture, promotion, and interact with other pieces based on ownership and piece type.” To play these, Metagame uses a search function and an evaluation function to help it choose the most favorable move. To use these functions properly though, it is aided by features called “Advisors” which are categorized in three groups:

- Advisors concerned with indicators of mobility.

- Advisors that deal with capturing interactions, threats and conditions that enable threats.
- Advisors concerned with goals

2.2.2 General Game Description Language

The field of general game playing gained a much broader interest in 2005, thanks to the inaugurated annual AAAI GGP competition, and a few years later became a well established area of research in AI [12]. One of the reasons for the success of it was the institution of the general Game Description Language (GDL) which became the foundation for the development and research of GGP.

GDL follows the tradition of AI Planning. However instead of describing the problem following the perspective of a single agent, GDL takes into account other agents that also have their own actions and goals. Reasoning about the intentions of other players is the basis for Opponent Modelling which is another aspect in which GGP goes beyond AI Planning. GDL, similarly to other planning languages has compact specifications and games like Chess can be fully specified with a few kilobytes of code [13].

To be able to allow concise, high-level specifications for rules of arbitrary finite games, GDL uses a logic programming-like syntax and uses the following keywords:

- `role(r)`, where `r` is a player.
- `init(f)`, `f` holds the initial position.
- `true(f)`, `f` holds in the current position.
- `legal(r,m)`, `r` can do move `m`.
- `does(r,m)`, player `r` does move `m`.
- `next (f)`, `f` holds in the next position.
- `terminal`, the position is terminal.
- `goal(r, v)` `r` gets payoff `v`.
- `sees(r, p)`, `r` perceives `p` in the next position

```

1 role(candidate).
2 role(random).
3
4 init(closed(1)).
5 init(closed(2)).
6 init(closed(3)).
7 init(step(1)).
8
9 legal(random,hide_car(?d)) <= true(step(1)),
10                                true(closed(?d)).
11 legal(random,open_door(?d)) <= true(step(2)),
12                                true(closed(?d)),
13                                not true(car(?d)),
14                                not true(chosen(?d)).
15 legal(random,noop) <= true(step(3)).
16
17 legal(candidate,choose(?d)) <= true(step(1)),
18                                true(closed(?d)).
19 legal(candidate,noop) <= true(step(2)).
20 legal(candidate,noop) <= true(step(3)).
21 legal(candidate,switch) <= true(step(3)).
22 sees(candidate,?d) <= does(random,open_door(?d)).
23
24 next(car(?d)) <= does(random,hide_car(?d)).
25 next(car(?d)) <= true(car(?d)).
26 next(closed(?d)) <= true(closed(?d)),
27                                not does(random,open_door(?d)).
28 next(chosen(?d)) <= does(candidate,choose(?d)).
29 next(chosen(?d)) <= true(chosen(?d)),
30                                not does(candidate,switch).
31 next(chosen(?d)) <= does(candidate,switch),
32                                true(closed(?d)),
33                                not true(chosen(?d)).
34
35 next(step(2)) <= true(step(1)).
36 next(step(3)) <= true(step(2)).
37 next(step(4)) <= true(step(3)).
38
39 terminal <= true(step(4)).
40
41 goal(candidate,100) <= true(chosen(?d)), true(car(?d)).
42 goal(candidate, 0) <= true(chosen(?d)), not true(car(?d)).

```

Figure 2.4: Representation of the Monty Hall game by using keywords in GDL [13].

- random, the random player.

Figure 2.4 shows the classic Monty Hall game, where a car is behind one of three doors and the contestant chooses which one, defined using these keywords.

2.2.3 Digital Ludeme Project

The Digital Ludeme Project is an initiative that kickstarted the research field of Digital Archæoludology, which aims to use Artificial Intelligence techniques along other computational techniques to generate hypotheses and reconstructions based on the data we have of various games throughout history, including board, card and dice games [14]. An example of the approach used in this project to reconstruct games with the incomplete information about rulesets or equipment obtained through archaeological evidence can be done through potential reconstructions and then self-play by a game-playing agent to then judge quantitative indicators of the game quality. For this purpose, the general game playing framework Ludii facilitates this approach as a tool used for easily game modeling and playing [14].

2.3 State of the Art

After having talked about General Game Playing, attempts at solving it and the Digital Ludeme Project, in this section we mention some modern frameworks with the capabilities to do GGP

2.3.1 Ludii

While GDL has led to some important research contributions in the field of GGP, there are key structural aspects of some games that have to be defined from scratch for each game, like a deck of cards, the board used and arithmetic operators.

Ludii instead uses a ludemic model where games are decomposed into Ludemes, conceptual units of game-related units, which allow us to distinguish between a game's form (rules and components) and function (behavior that emerges through play [15]). The ludemic model allows for automatic description of rulesets, comparisons between games and the potential for a single AI tool to be able to model, play and analyze almost any traditional game of strategy.

A game in Ludii is given by a 3-tuple of ludemes $G = \langle \text{Players}, \text{Equipment}, \text{Rules} \rangle$. Players is a finite set of $k+1$ players where k is equal or higher than 1, with the game's control being defined as alternating, simultaneous or real time. Equipment defines a list of containers (boards, player's hands, etc) and a list of components (pieces, cards, tiles, dice, etc). Rules define how the game operates, including Start (a list of starting actions to model the initial state), Play (the set of all possible lists of legal actions that can be realized) and End (which describes the conditions under which the state of the game is considered to be over).

2.3.2 Regular BoardGames

Regular Boardgames (RBG) was introduced with the goal of allowing effective computation of complex games, while still being universal and keeping easy to process game descriptions. Its base is the use of regular languages to describe legal sequences of actions that players can apply to the game state. While it keeps descriptions intuitive, it requires formal basis to be well defined [16].

An RBG description is a 7-tuple, composed of (Players, Board, Pieces, Variables, Bounds, InitialState, Rules).

- Players, finite non-empty set of players.
- Board, representation of the board using Vertices, Edges and a representation of the angles of the edges.
- Pieces, finite non-empty set of pieces, representing the elements that are placed on the board.

- Variables, any variables that could be used and that could vary even if the game state looked the same.
- Bounds, a function specifying the maximum values for every variable.
- Game State, a semi-state taking into account the current player, the pieces on the vertices and the variables used.
- Initial state, the arbitrary semi-state from which the game begins.
- Actions, basic option that can be applied to a game state. The possible actions are: Shift, On, Off, Assignment, Comparison, Switch.
- Rules, defines what are legal actions in the game
- Patterns, denoted by M , where M is valid if there exists a legal sequence of actions under M for the state.

There exists a simple version of the RBG language (low-level RBG) which directly represents an abstract RBG description which is simple and easy to process. There also exists an extension called high-level RBG which allows for more clear and human-readable descriptions.

RBG is flexible enough to be able to describe every finite deterministic game with perfect information [16]. It also has the advantage of very effective reasoning, having similar performance to game-specific reasoners. The natural reasoning algorithm for RBG is a generalization of methods commonly used in game-specific reasoners.

2.3.3 PyVGDL

PyVGDL first emerges from the idea to make a video game description language to be able to specify complete videogames and compile them with a special VGDL compiler. This implementation would make it possible for users to modify existing games easily or have a library of existing implementations that can be customized, this would also allow automatic game creation to be explored further through machine learning algorithms [17].

The way this language is designed has all described objects located in a 2D space with associated coordinates, these can move actively (through player input or internal specified behaviors) or passively (through physics) and interact with other

objects through collisions. All interactions are subject to rules that depend only on the two colliding objects [5].

PyVGDL gets its name from the fact it is implemented in Python and builds directly on the pygame package used for game development. The syntax is heavily based on Python’s syntax but the game descriptions conform to a more strict treelike structure that resembles XML.

The game descriptions in PyVGDL are defined by a level description, which indicates the positions of all objects and the layout of the game in a 2D space, and the game description which describes the dynamics and interactions of the objects in the game. This game description is composed of four different sets of instructions:

- LevelMapping, describes how to translate the characters in the level description into objects and generate the initial game state.
- SpriteSet, defines the classes of objects used. These are derived from an abstract VGDL-Sprite class and are organized in a tree where child classes inherit the properties of its parent class.
- InteractionSet, which defines the events that happen when two objects collide.
- TerminationSet, which defines the criteria by which the game can end.

2.3.4 PuzzleScript

PuzzlesScript bears similarities to PyVGDL in syntax, structure and the games that can be implemented in it. PuzzleScript isn’t declared as a general purpose puzzle game making tool, but it is an easy to use engine to make puzzle games in a 2D space. A PuzzleScript file is divided into 8 sections [18]:

- Prelude, where details about the project can be given.
- Objects, where all in-game objects are declared.
- Legend, here single-character names are given to things to be typed later in levels, or properties are defined for objects so they can be referred to in rules
- Sounds, to add sounds to in-game actions

- CollisionLayers, serves the purpose of separating objects in different layers, as 2 objects cannot coexist in the same space.
- Rules, rules have a left and right side, the engine looks for occurrences that match the left side and replaces them with the contents of the right side.
- WinConditions, where the win condition can be set, these can be multiple required for the game to end too.
- Levels, levels is where the layout for the level is typed out, including the objects in their initial positions.

3. Experiment

In the following section we'll go over the details of the project that will be realized, detailing the experiment and the expected results from it.

While AI research throughout the years has tackled the problem of playing and solving different kind of games, it usually falls into the pitfall of having to overspecialize to do so. Game playing software is created with the specific purpose of playing one game only, and therefore focuses on generating algorithms to play it in advance, leaving the problem solving to the programmer instead of to the software.

One way to attempt to solve this is using frameworks created for general game playing, where general base rules for various games are implemented and the framework is only familiarized with the game in the instant before playing it.

General game playing is a large subject that covers a wide array of games so we are focusing on the genre of puzzle games specifically. While AI has been researched and worked on for several puzzle games individually, what we are looking for is a framework that can effectively work with more than one puzzle game. This will involve both generating simple instances of levels for them and also being able to solve these correctly.

3.1 Solution

Our solution to find the framework we are looking for will be the following experiment: Choosing 3 frameworks and implementing quick and simple generators for 2 puzzle games in each of these frameworks, then implementing a puzzle solver in each of them.

The first comparison is done with the implementation of the generators and solvers, before the actual levels are generated and it's a comparison of the scalability

of the solutions and how flexible it is. This takes into account how much work has to be done to implement both puzzle types instead of just one, and how easily in the future this could be taken to implement other different puzzle games into it.

Next there's another comparison when generating the levels, we compare the performance of each framework, how fast and how efficiently it can generate complete levels for the games. This'd be divided by game and compared by levels per unit of time. Then we have to take into account how many of the total levels are actually solvable to see the efficiency of it.

For the final comparison we will test the solvers on each framework, making them solve the generated levels on each of them and compare them similarly to the comparisons done in [19], using an average of playouts per second on each type of game for each framework, while also watching out for any problems that could come up in the execution of the solver, such as possible trouble solving puzzles that are not unsolvable.

3.2 Frameworks used and Selected puzzles

For this experiment after looking at current general game playing frameworks, Ludii, Regular BoardGames and PuzzleScript were chosen as the three frameworks where the generators and solvers will be implemented. All three of these are made with 2D games in mind, so there's no clash with the selected types of puzzles and puzzlescript itself is made with puzzle games in mind.

The puzzle game types chosen for this experiment will be Sokoban-type puzzles and sliding puzzles similar to Rush Hour, these were chosen for their low variability in terms of elements and possible moves, which makes the generated instances of puzzles similar enough to not be unnecessarily complex.

3.3 Expected Results

On the subject of expected results, first of all we expect three different sets of results, one for the implementation itself of the level generators and the puzzle solvers on each framework, one for the level generators on each framework, also separated for each puzzle and lastly a comparison table to see which framework is more efficient at solving the puzzles it generated.

The first set of results are derived from the flexibility of each solution on each different framework. This will be determined by the amount of shared code a framework has between both puzzle implementations, and how much extra code and complexity has to be added when implementing a game, and if there are any limitations to the framework that cause the game to be implemented differently than expected. This comparison will give us a good idea of which of these would be better at implementing more puzzle games in the future with ease.

The second set of results will consist of two pieces of information gathered from the test, a comparison of the time it takes to generate a large amount of levels generated for each framework and one that compares the amount of solvable levels generated out of the total amount generated. Ideally we will see one of the frameworks get a better result in both games and in both charts, but if that is not the case we will have to take note of what frameworks are the best in several areas.

And the final set of results will be taken from the performance and reasoning demonstrated by the implementations of the puzzle solvers in each of the frameworks when actually solving the previously generated puzzles. For this we will compare the efficiency of each solver by comparing their performance and speed in solving levels by using the same group of levels adapted for each one of the frameworks.

4. Implementation

We will split the description of the implementation in 3 parts. Implementation of the game, Implementation of the random level generator and testing the generators, and implementation of the solver if necessary and testing how efficiently the level solvers work. Each of these sections will describe how it was done and what difficulties the process had on a specific framework and in comparison to the others. Then they will be compared to see which one would be more flexible and efficient on the next chapter.

The implementation of both games on all 3 frameworks can be found on the following repository:

<https://github.com/ansotomUtalca/FrameworkComparisonForPuzzleGames>

4.1 Implementation of the puzzle games on the selected frameworks

For the experiment, the 2 games selected were Sokoban and Rush Hour while the selected frameworks are Ludii, PyVGDL and PuzzleScript Next.

To be able to make the comparisons between the games on each of the frameworks in a fair manner, we must first establish ground rules to make them as similar as possible.

The implementations of Sokoban will be played in a 8x8 square board, with a player avatar that can move in 4 directions, 3 crates to push around, 3 goals where the crates must be positioned on and 5 immovable obstacles placed inside the 8x8 grid. To win all 3 crates must be placed on top of the goals.

The implementations of Rush Hour will be played in a 7x7 square board with a 2x1 piece representing the main piece to get out, 6 vertical 1x2 pieces, 7 2x1 pieces, and the position of the main piece and the escape tile are the same on every level.

4.1.1 PuzzleScript

Implementation of the selected games in PuzzleScript was started in the online editor available on www.puzzlescript.net, but was quickly switched out for PuzzleScript Next, an open-source engine based on combining PuzzleScript, Puzzlescript Next and Pattern:Script to take advantage of features that are not in the original PuzzleScript, mainly use of the mouse to click and drag objects and the Solver tool.

The Sokoban implementation was taken from the example game that is available when making a game with PuzzleScript on the main page with some changes. The dimensions of the puzzle were changed to the previously described size, a “NewLevel” and “Spawn” object were added to be able to implement a randomized level generation that would be executed at the start of the level, and the “Target” object was separated into multiple numbered targets, to be able to keep track of the number of targets and crates being the same when generating a level.

For the implementation of the Rush Hour game, the main mechanics of how the blocks move on the board with the mouse are based on the example “Gridblocked” by Aureolix available on the PuzzleScript Next main page. The size of the puzzle area was switched to 7x7 and both horizontal and vertical blocks were limited to small 2x1 blocks instead of also having 3x1 ones. This change was made due to the fact that in all 3 frameworks used for this experiment all objects occupy only 1 square, and rectangular pieces will instead be composed by 2 objects moving as one. While the implementation for pieces made up of 3 square objects is already in the Gridblocked example, for the sake of making the interactions between objects simpler on the other frameworks pieces were kept to 1 size only.

Implementing both games is a simple task in PuzzleScript Next, though there is almost no code shared among the instructions that were written for both games, as Sokoban works with very few lines and all its crates and goals are generic, while Rush Hour has mouse implementation to move any selected piece easily and every piece is a different type of object. Figure 4.1 and 4.2 show how the games look in this PuzzleScript implementation and examples of the code can be found in Appendix A.

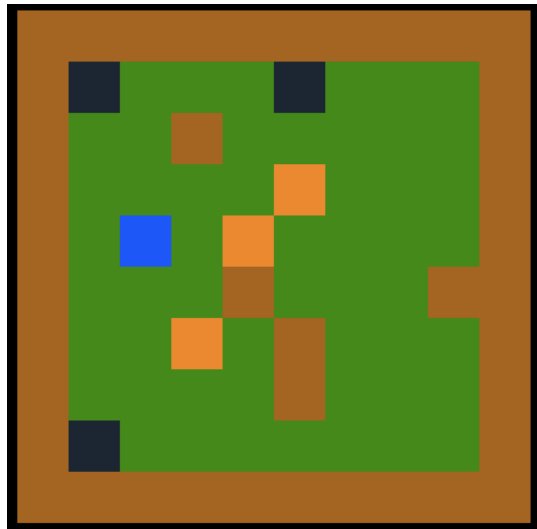


Figure 4.1: Example level of Sokoban in Puzzlescript. The blue square represents the player object, yellow squares are the crates and the black squares are the targets.

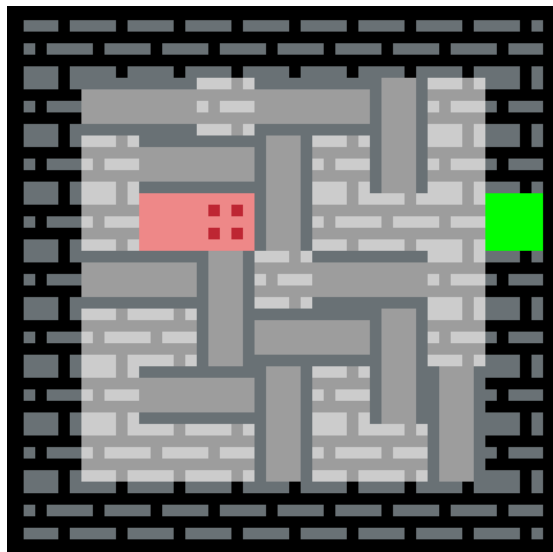


Figure 4.2: Example level of Rush Hour in Puzzlescript, where the goal is to move the grey pieces out of the way and get the light red piece to the green square representing the goal.

Making games with higher variability according to De Kegel and Haahr’s scale such as Mazes or Tile-matching puzzles should be extremely simple. As mazes in their simplest state are similar enough to Sokoban and PuzzleScript Next does have the ability to check for specific objects being in a relative position to others, making tile-matching simple.

4.1.2 Ludii

Implementation of both games in Ludii had to be done from scratch, as neither game or games that could be similar to them are found in the Ludii library, at best using examples of Ludemes from other games as base for certain mechanics.

For the Sokoban implementation, 6 total pieces were defined: “Box”, “Target”, “PlayerPiece”, “Wall”, a piece for the overlap of the player and target, and a piece for the overlap of a box together with one of the targets. While the original idea was to have the targets as Regions in the engine, regions have to be defined before the game starts and cannot be placed at random on the grid, therefore they were defined as pieces.

The PlayerPiece has 4 main behaviors defined for movement in each of the 4 directions. While movement for all directions can be implemented in 1 ludeme, it was separated in 4 to avoid relying on “(to)”, “(last To)” and “(ahead (to))” which overwrites “(to)” anytime any of those 3 variables are called and end with more confusing code to debug and less reliable movements on game execution. The separation of functions in 4 directions also allows for the use of ludemes of fixed directions “N”, “S”, “W” and “E” to check surroundings while pushing crates more accurately.

Aside from the main set of 4 ludemes defined for movement, several other sets of 4 ludemes had to be made for every other interaction that could be triggered while interacting with the box and target pieces by moving the player piece, and a separate set of these had to be added too for interactions while moving the piece that represents a player piece overlapping a target piece (though this one is switched for a normal player piece and a target piece is set where it stood whenever a movement that would take it out of a target is made).

The win condition in this implementation is set as making sure there are no crates left, as the only way for crates to disappear is when they overlap with targets and

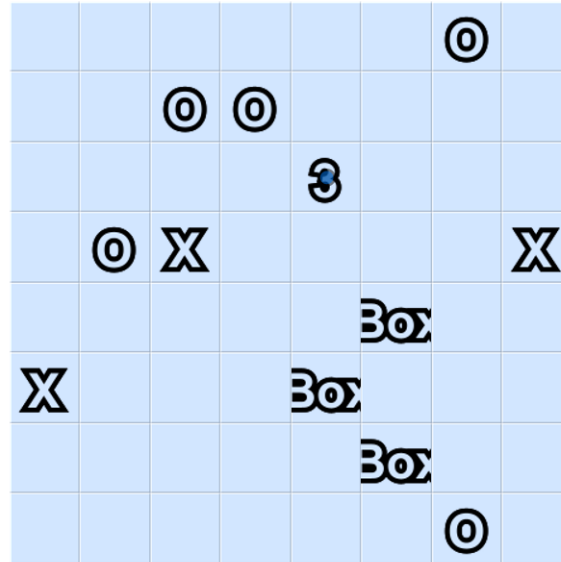


Figure 4.3: Example level of Sokoban in Ludii. “PlayerPiece” represented as 3, “Wall” as O, “Box” as Box and the “Goal” pieces are represented by an X.

become a different piece. When there are no crates left in the game it means all crates have been replaced because they are overlapping the target pieces.

For the Rush Hour implementation, we use 7 different game piece ludemes and several defined regions. The pieces are “Car1” and “Car2” representing the left and right halves of the main game piece, “CircleA1” and “CircleA2” representing the left and right halves of a horizontal piece, “SquareA1” and “SquareA2” to represent the top and bottom halves of a vertical piece and finally a “Wall” piece to block off all edges of the puzzle except for the exit. The defined regions are the “Goal” region, the “Walls” region, and several “Spawn” regions used for the level generator.

The movements allowed were implemented using the way crates were pushed with the directional Move ludemes in our Sokoban implementation, but each piece only has one direction in which it can move. Left-half pieces only move right, right-halves move left, top-halves move down and bottom-halves move up. This is so that they will push their other half of the piece and stay together anytime they move.

The win condition for this game is the “Car2” piece being in the “Goal” region.

While the games can be implemented without much issue, it is adapting them to be able to generate levels later on that makes the implementation more complicated. The Sokoban implementation suffers specifically because sites cannot be random-

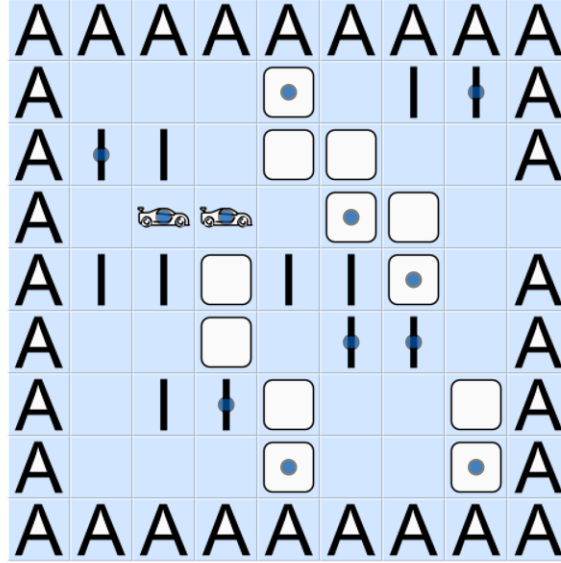


Figure 4.4: Example level of Rush Hour in Ludii. “Car1” and “Car2” shown as cars, the vertical pieces represented by squares, the horizontal pieces represented by lines and the blocks that make up the wall by the letter A.

ized after starting, and implementing the goals as pieces instead of sites resulted in additional code to handle all interactions between the pieces.

Additional puzzle games like mazes and tile-matching games could be implemented in Ludii, the logistics of a tile-matching game could prove very complicated, as checking for specific positions of other pieces in relation to a piece could become very convoluted very easily. Figure 4.3 and 4.4 show how the games look in this specific implementation, and excerpts of the ludemes used can be found in Appendix A.

4.1.3 PyVGDL

PyVGDL comes with an example implementation of Sokoban, this implementation was used as a base and changed to fit the rules and details used in our experiment. The size of the board was reduced to 8x8 and the blocks do not disappear after making contact with the goal, to avoid the puzzle being solved by pushing all blocks into the same goal. The win condition of “no blocks remain” was kept but the blocks were modified so they would change into a different object when on top of one of the goals and changing back into regular blocks when off them.

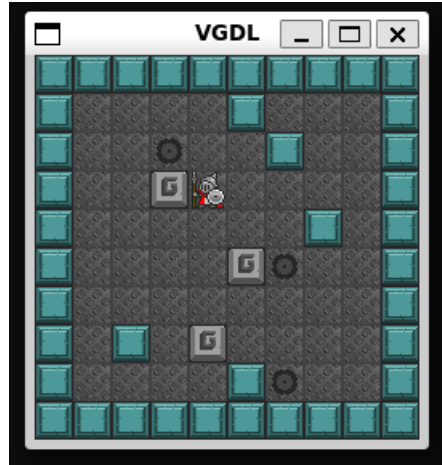


Figure 4.5: Example level of Sokoban on PyVGDL. The player character is represented by a knight by default and the grey blocks must be pushed onto the black circles to win.

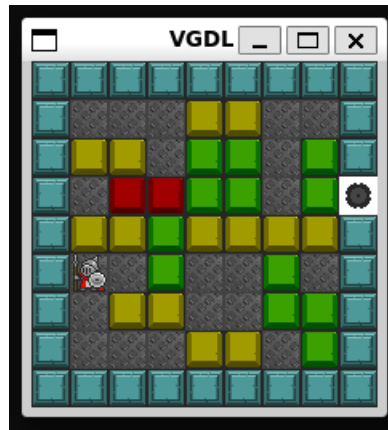


Figure 4.6: Example level of Rush Hour in PyVGDL. The player is once again represented by the knight, the piece that has to be pushed out is colored red, while vertical and horizontal pieces are green and yellow respectively.

For the Rush Hour implementation, much like in the other games all pieces are composed by 2 pieces each, divided into `hori1` and `hori2` for horizontal pieces, `ver1` and `ver2` for vertical pieces, `car1` and `car2` for the main piece and an avatar character that can move in all 4 directions to push the other pieces around.

New functions had to be defined and added to the `effects.py` file, where the effects of interactions between objects are defined. A function for a piece to be pushed by the player, making sure it checks which piece it is and pushing it in the correct direction, since each piece can only move one way. A function that pushes a piece when it collided with a piece of the same direction, mostly used so that horizontal and vertical pieces would move as a whole when pushing a block, but also allows for one horizontal piece to push a different one. To make this work properly a global variable had to be used, rather than keeping it to interactions purely between objects as intended. A new type of avatar was created, based on the movable avatar who can move in all 4 directions but adding an action button that flips a Boolean variable to push blocks, this allows the avatar to walk over other blocks without pushing puzzle pieces by accident.

While the Sokoban implementation is easy to do, it only gets slightly more complicated while adapting it so it can generate levels later on. It is the Rush Hour implementation which caused more problems, as rather than simply using the functions that are already defined in `effects.py`. While it is true that PyVGDL can be modified so the engine can better adapt more games, the fact code had to be modified directly in the engine makes it less flexible than the other two in this aspect. Figure 4.5 and Figure 4.6 illustrate how Sokoban and Rush Hour look in this implementation. Just like for the other implementations, excerpts of the code used can be found in Appendix A.

4.2 Level generators on the selected frameworks

The following section describes how the level generators that have been implemented create levels for both puzzle games, before going into the details for how it works on each of the frameworks and the notable differences on each one.

For the game of Sokoban: Several square wall tiles [precise number not definitive yet] are placed inside the 8x8 grid of the puzzle as obstacles, then the player character and 3 crates are placed on random tiles in the grid too. The placement of every object

is completely random, so the possibility of impossible puzzles where the crates end in corners or the player is trapped are possible.

For the game of Rush Hour: The main piece and the goal always start on the third row from top to bottom on every level, with the main piece one tile away from the left wall, and the goal making an exit on the right wall. Horizontal pieces can never be set in between the main piece and the goal, as that would instantly make a game impossible to beat.

Due to the fact that all the game pieces are composed by 2 individual pieces placed next to each other, randomizing where the pieces are set means making it completely random risks pieces spawning in tiles that are already occupied with other pieces or by walls. While reducing the area where pieces can spawn can make it easy to avoid overlap with walls, it still leaves a risk of overlapping other pieces or making the exit too easily accessible.

To place the pieces in the board, 20 spawn points (9 for vertical pieces and 11 for horizontal pieces) were designated in fixed positions on the 7x7 grid and when a level is generated 6 vertical pieces and 7 horizontal pieces are placed on corresponding spawn points chosen at random.

4.2.1 Puzzlescript

For the level generation in Sokoban, the level starts with the outer walls of the level and a “NewLevel” invisible object inside, every tile of the level has a “Spawn” object overlapping it. In the defined rules of the game, if there’s a “NewLevel” object in the level it will set 5 Walls in random available Spawns, then it’ll attempt to set 3 different targets in random available Spawns. Since any target that is placed on top of a wall would be deleted as it’d create an impossible level, spawns are replaced with the new object to avoid any other objects being able to be placed in the same position. After the targets are placed, it proceeds to spawn a crate for every individual target remaining in a random available spawn. To finish the level generation it places a player object and then replaces the NewLevel object with a Wall so the level generation won’t trigger again.

In the puzzlescript implementation of Rush Hour, to be able to generate levels while following the details specified earlier, 4 “WaveSpawn” objects were created, and were placed accordingly (4 “FirstWaveSpawn” objects on each of the 4 places

where a piece can spawn on the first wave and so on). Next to each WaveSpawn, a SpawnComplement was placed, these were made in two varieties: Horizontal and Vertical.

In order, random spots occupied by the first WaveSpawns will be chosen to place a vertical piece, then the game checks if there's a vertical piece next to a vertical SpawnComplement and if there is it completes the piece, this is done for all vertical and horizontal pieces in the correct order until the all 13 pieces are placed and the level is fully generated.

4.2.2 Ludii

The level generation for the Sokoban levels on Ludii is extremely simple, as all the pieces (walls, crates, targets and player) are set at the same time at the “start” ludeme. Pieces can be set to appear in completely random tiles on the board, or they can be assigned to be randomly placed on a specified region. Like the generation in PuzzleScript, it can easily generate unsolvable levels.

For the level generation in Rush Hour the main piece is set at the start in its designated tile and wall pieces are set in all wall regions, surrounding the entire puzzle area except for the tile where the goal region is. Then the 20 fixed spawn sites are divided in 4 waves of spawning pieces. The first spawn region is for vertical pieces and is between the main piece and the goal, the next wave places horizontal pieces and surrounds the first, then the third wave places vertical ones and the final wave places horizontal pieces again. These alternate and get farther from the goal as they go so they can trap the pieces that stand in between the player piece and the goal.

After a spawn wave places its corresponding pieces, another ludeme looks for all pieces of the type that was placed (Vertical or Horizontal) and places the corresponding piece to turn it into a 2x1 sized piece. Placing the pieces is separated into multiple waves so that when putting the second half of the pieces after their locations were randomly selected there won't be overlap with other pieces. Ludii won't allow for pieces to be placed where another piece already is at the start of the game, so this avoids overlap while also allowing for placement of the pieces to be as close to each other as possible to try to make the levels somewhat challenging when the size of the pieces might be too small.

4.2.3 PyVGDL

For the Sokoban level generation in PyVGDL, two new functions were created in the `effects.py` file. One used to transport objects to a random coordinate in the board when they interacted with a “spawn” object only found at the start of the level while getting rid of the “spawn” object. The other function teleports an object if it ended on the same tile as a different one. This guarantees that blocks, goals and obstacles will be the specified number we want and that they will all be on different spots.

For Rush Hour level generation, 4 new objects were created to act as the 4 waves of spawning pieces, which all move to a random tile out of a pre-selected list, which matches with the pre-selected spawn points we used in the other two frameworks for Rush Hour Levels. To facilitate this a new function was made which keeps all coordinates as 2 long lists of integers and gets a specific one when the function is called.

To avoid objects picking the same position as a previous object and keep all waves going in order, a global variable had to be introduced which keeps track of how many pieces have been placed. When a piece is placed in an empty spot, it’ll spawn the second half of the piece and increase the count, but if a piece attempts to be placed in a spot that isn’t empty it’ll call the function again until it finds a position it can use. pieces that would be placed in the second wave won’t be teleported until all first wave objects are placed to avoid objects overlapping each other.

In comparison to the other two frameworks, it feels convoluted to place pieces in random locations in PyVGDL, as the way to call objects is simply when they interact with other objects instead of being able to be called directly with other functions.

4.3 Solvers used for the selected puzzles

For the question of what kind of solver to use on each framework, it was prioritized any solvers that the framework had by default, as it is a point in favor of it being easier to implement and test new games.

4.3.1 Puzzlescript

Puzzlescript Next comes with an already implemented solver which uses a modified BFS algorithm to look for a solution, while using a heuristic that pushes it towards

nodes that are closer to its win condition. Since it's not compatible with mouse controls, the puzzlescript implementation for Rush Hour had to be changed to also allow a small player character controlled with the arrow keys to push the pieces.

4.3.2 Ludii

Out of the several AI agents Ludii comes equipped with, we selected the UCT (Upper Confidence bounds applied to Trees) Ludii AI, as it manages to resolve both games without resorting to purely random movements.

4.3.3 PyVGDL

For PyVGDL, a BFS Agent was implemented from scratch as a new type of avatar that'd return the correct path after finding a string of moves that would reach the desired win condition. Two versions of this agent were implemented: One for Sokoban that would use all 4 arrows to move around, and another that would use all 4 arrow keys and the space bar for the Rush Hour implementation.

5. Results

5.1 Comparison Between the frameworks

The first results from the experiment are found in comparing the implementation of both games on each framework. Table 5.1 compares the implementations according to how flexible and efficient they are by asking different questions about it.

	PuzzleScript Next	Ludii	PyVGDL
How much shared code is there between the games	Almost no shared code at all, mostly thanks to Sokoban having very few instructions.	Some shared code for movement and pushing blocks, but Sokoban ends with more movement functions due to having interactions with the goal as its own piece.	Very different code due to having to implement a new avatar and having to write new functions for interactions between the pieces.
Would it be easy to implement more puzzle games	Yes.	Yes but there could be limitations for some.	Yes, even though it's possible changes would have to be made in order to implement them properly.
Was it necessary to change part of the framework's code	Though the change to PuzzleScript Next was initially made to move the pieces with a mouse, it would have been possible to adapt it similarly to the PyVGDL implementation.	No, only ludemes were made which is to be expected.	Yes, a new avatar had to be implemented and functions to define interactions accurately.
Does it have a limitation that changes how it was implemented	Unable to have long pieces for Rush Hour, and to imitate longer pieces each piece is a different type of piece instead of a generic one.	Goals not being able to be randomized as sites changes implementation severely.	Biggest limitation is the way the interactions was processed by the game's ticks, as if one interaction caused a similar interaction to happen in a chain, this would be pushed to next tick instead of dealt with properly.

Table 5.1: Comparison table between the game implementations

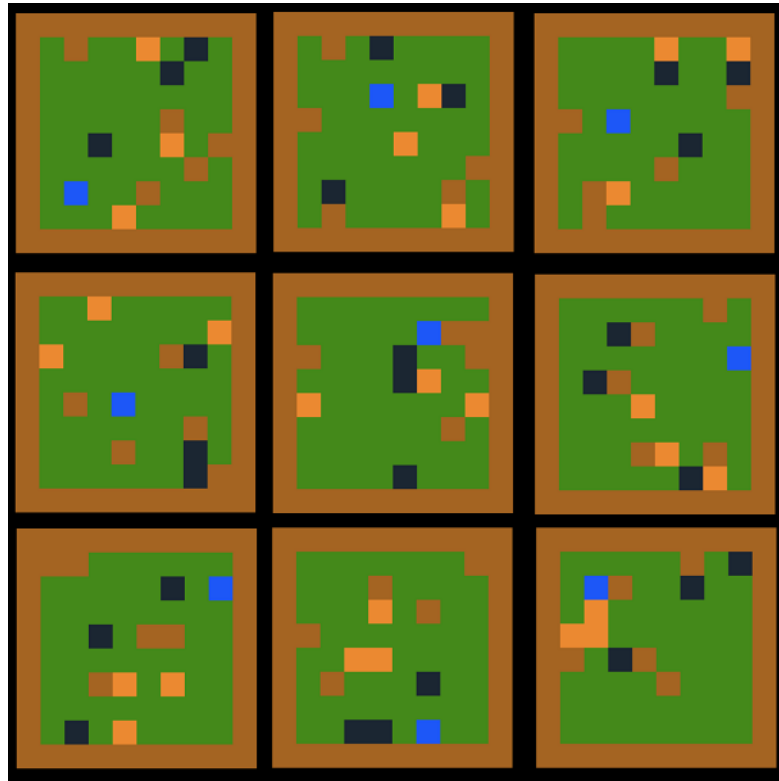


Figure 5.1: 9 of the 100 Sokoban levels first generated. Note how easily puzzles are unsolvable when a block starts next to a wall.

5.2 Comparison between all generators on the 3 frameworks

As a starting point, 100 levels of Sokoban with the generation logic we established above were created in puzzlescript and evaluated on if they're able to be beaten or not. Out of these, only 27 of the generated levels were able to be beaten while the other 73 were not.

From the levels that were not able to be beaten, 1 level had an unreachable goal, 1 level had a block that wasn't able to be moved without trapping it, 1 level started with a block that couldn't be moved, 39 levels started with a block next to a wall that was not able to get to any goals, and the other 31 levels started with blocks that were both next to a wall and unable to be moved. Figure 5.1 shows a few of the generated levels, where unsolvable levels can be spotted at a glance due to the way boxes cannot be pushed off the edges.

From this initial test it is easy to assume that the amount of playable levels would increase drastically if we reduce the range in which blocks spawn

After changing the level generation so the crates would not be placed on the edges. A test was run on each framework to see how fast it would generate 100 levels of Sokoban and how many of these levels can be solved.

	Time to generate 100 levels	Notes
PuzzleScript Next	8 seconds 139 milliseconds	Levels are generated as soon as the level itself is restarted.
Ludii	1 minute 53 seconds 508 milliseconds	Each time Ludii was restarted it took almost a second for the new level to appear and pressing restart faster than that would simply skip showing a level, so a sleep of 0.6s (fastest that could be implemented with consistent results) was called in between each restart
PyVGDL	1 minute 10 seconds 197 milliseconds	It loses some time since it has to be closed and opened again each time the level was generated instead of simply restarting like the other two.

Table 5.2: Timing comparison for generating levels

Out of the 100 levels generated on each framework, the amount of solvable levels were 92 in Puzzlescript Next, 90 in Ludii and 85 levels in PyVGDL for an average of 89 percent between all of them.

5.3 Comparison between the puzzle solvers

To compare how efficient each framework is, several levels have been created and have been implemented into each framework so that all the comparisons can be done using the same cases. At first the levels were taken from some of the levels that were randomly generated in the previous testing, but seeing as Ludii's UCT had trouble solving Sokoban levels with big open spaces, the levels have been reworked to scale up in difficulty so that we might see more results. The five Sokoban levels and three Rush Hour levels that were used for testing can be seen in Appendix B.

In the following results tables, times that took less than a second or way over 10 minutes are not written down with the specific time, since the trouble of getting a more accurate reading does not give any benefit when a comparison against one of those times already makes clear which result is better without much margin for error. For the puzzlescript results for Sokoban, the time it took to solve each level and the amount of nodes visited are shown in Table 5.3.

	Lvl1 Next	Lvl2	Lvl3	Lvl4	Lvl5
Time	< 1s	< 1s	4.429s	< 1s	< 1s
Nodes visited	40	109	13,135	659	498

Table 5.3: Comparison table for Sokoban level solving on PuzzleScript Next

For Ludii the timing is more complicated, as the AI agent takes its turn each second even if it has not found a path, and while the time it takes to move can be adjusted if the time is set as a shorter interval it will visit less nodes before moving, making the movements more random. In table 5.4 we show how many moves it took to solve each puzzle and the thinking time it had between moves.

	Lvl1 Next	Lvl2	Lvl3	Lvl4	Lvl5
Number of moves	10	79	58	120	unsolved
Timer	0.5s	0.5s	1.5s	1.0s	–
Notes	No notes	No notes	Took 3 attempts.	No notes	Several failed attempts and over 400 moves in some

Table 5.4: Comparison table for Sokoban level solving on Ludii

Table 5.5 has the results for the solver implemented in PyVGDL, though every level will eventually be solved due to the nature of a BFS search, all levels that took over 10 minutes were marked as unsolved as it simply can't compare with the other results.

	Lvl1 Next	Lvl2	Lvl3	Lvl4	Lvl5
Time	46s	unsolved	unsolved	unsolved	unsolved

Table 5.5: Comparison table for Sokoban level solving on PyVGDL

For the results in solving the chosen Rush Hour levels, table 5.6 shows the time it takes to solve each one and the amount of nodes visited.

	Lvl1 Next	Lvl2	Lvl3
Time	< 1s	1.3s	1.1s
Nodes visited	264	1,797	1,695

Table 5.6: Comparison table for Rush Hour level solving on PuzzleScript Next

The results for Ludii's UCT agent on the three Rush Hour levels are shown in Table 5.7.

	Lvl1 Next	Lvl2	Lvl3
Number of moves	5	7	50
Timer	1.0s	1.0s	1.0s

Table 5.7: Comparison table for Rush Hour level solving on Ludii

Lastly, table 5.8 shows the results for the Rush Hour level solving in PyVGDL

	Lvl1 Next	Lvl2	Lvl3
Time	31s	4m15s	12m54s

Table 5.8: Comparison table for Rush Hour level solving on PyVGDL

Looking at the results, PuzzleScript Next has the better results on all levels for faster and more efficient solving. In this specific case it is fair to assume that it is thanks to the heuristics its search uses, in comparison to UCT and a simple BFS. Even so, for the sake of this comparison it is useful to acknowledge that it is better with the tools it already has, while keeping in mind that the others could reach better times if a dedicated AI agent was created for them to improve efficiency.

6. Conclusions and final thoughts

Looking at the results in the previous chapter, it is easy to see that PuzzleScript Next is the most convenient out of the three frameworks for the case studied here. It is flexible and well aimed at the creation of 2D puzzle games on top of running smoothly and having a very good level solver already implemented. Of course, that is not to say the other two frameworks are not flexible, as those were created with more than only puzzle games in mind.

If we keep our scope to only 2D puzzle games, it would be convenient to expand and test more games on PuzzleScript Next. The language used is simpler and easier to learn compared to using Ludemes, and it has more flexible functionalities already implemented than PyVGDL. Therefore both the process of implementing games and testing them out should be smoother.

On the other hand, it is important to keep other frameworks in mind for general game playing, specially if we divert from puzzle games or if we decide to test different search and game playing algorithms for AI agents, as both Ludii and PyVGDL do encourage the user to implement their own AI behavior for the games implemented there.

Bibliography

- [1] Ahmed Khalifa, Diego Perez-Liebana, Simon M Lucas, and Julian Togelius. General video game level generation. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 253–259, 2016.
- [2] Ahmed Khalifa, Michael Cerny Green, Diego Perez-Liebana, and Julian Togelius. General video game rule generation. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 170–177. IEEE, 2017.
- [3] Michael Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the aaai competition. *AI magazine*, 26(2):62–62, 2005.
- [4] Barbara De Kegel and Mads Haahr. Procedural puzzle generation: A survey. *IEEE Transactions on Games*, 12(1):21–40, 2019.
- [5] Tom Schaul. An extensible description language for video games. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):325–331, 2014.
- [6] Dennis JNJ Soemers, Éric Piette, Matthew Stephenson, and Cameron Browne. The ludii game description language is universal. *arXiv e-prints*, pages arXiv–2205, 2022.
- [7] Gaspard de Batz de Trenquellleon, Ahmed Choukarah, Milo Roucairol, Maël Addoum, and Tristan Cazenave. Procedural generation of rush hour levels. In *International Conference on Computers and Games*, pages 181–190. Springer, 2022.
- [8] Adem Tuncer. 15-puzzle problem solving with the artificial bee colony algorithm based on pattern database. *JUCS: Journal of Universal Computer Science*, 27(6), 2021.

- [9] Joseph C Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.
- [10] Yoshio Murase, Hitoshi Matsubara, and Yuzuru Hiraga. Automatic making of sokoban problems. In *PRICAI’96: Topics in Artificial Intelligence: 4th Pacific Rim International Conference on Artificial Intelligence Cairns, Australia, August 26–30, 1996 Proceedings 4*, pages 592–600. Springer, 1996.
- [11] Barney Pell. A strategic metagame player for general chess-like games. *Computational intelligence*, 12(1):177–198, 1996.
- [12] Jakub Kowalski and Marek Szykuła. Experimental studies in general game playing: An experience report. *arXiv e-prints*, pages arXiv–2003, 2020.
- [13] Michael Thielscher. The general game playing description language is universal. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1107, 2011.
- [14] Dennis JNJ Soemers, Walter Crist, and Cameron Browne. Report on the digital ludeme project. *ICGA Journal*, 41(3):138–142, 2019.
- [15] Eric Piette, Dennis JNJ Soemers, Matthew Stephenson, Chiara F Sironi, Mark HM Winands, and Cameron Browne. Ludii—the ludemic general game system. In *ECAI 2020*, pages 411–418. IOS Press, 2020.
- [16] Jakub Kowalski, Maksymilian Mika, Jakub Sutowicz, and Marek Szykuła. Regular boardgames. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1699–1706, 2019.
- [17] Marc Ebner, John Levine, Simon M Lucas, Tom Schaul, Tommy Thompson, and Julian Togelius. Towards a video game description language. *Artificial and Computational Intelligence in Games*, page 85, 2013.
- [18] PuzzleScript Documentation. <https://www.puzzlescript.net/Documentation/documentation.html>. Accessed: 2024-07-05.
- [19] Eric Piette, Matthew Stephenson, Dennis JNJ Soemers, and Cameron Browne. An empirical evaluation of two general game systems: Ludii and rbg. In *2019 IEEE Conference on Games (CoG)*, pages 1–4. IEEE, 2019.

A. Code Excerpts from the Implementations

Algorithm A.1 Rule used to handle player and crate interaction for Sokoban in PuzzleScript

(Player movement)

[> Player | Crate] -> [> Player | > Crate]

Algorithm A.2 Rules used to handle being able to drag the blocks with the mouse for Rush Hour in PuzzleScript Next.

(Input handling)

[Trail] -> []

[Click] [Cursor] -> [Click] []

[Click] [PrevDrag] -> [Click] []

[Click] -> [Cursor PrevDrag] nosave

[Cursor] [PrevDrag | Drag] -> [> Cursor] [| PrevDrag]

[horizontal Cursor VBlock] -> [Cursor VBlock]

[vertical Cursor HBlock] -> [Cursor HBlock]

[vertical Cursor VBlock] -> [vertical Cursor vertical VBlock]

[horizontal Cursor HBlock] -> [horizontal Cursor horizontal HBlock]

[Click] -> []

[Drag] -> []

Algorithm A.3 Rules used to generate objects at random for Sokoban levels in PuzzleScript Next.

(Level generation)

random [NewLevel] [Spawn] -> [NewLevel] [Wall]

random [NewLevel] [Spawn] -> [NewLevel] [Wall]

random [NewLevel] [Spawn] -> [NewLevel] [Wall]

random [NewLevel] [Spawn] -> [NewLevel] [Wall]

random [NewLevel] [Spawn] -> [NewLevel] [Wall]

random [NewLevel] [Spawn] -> [NewLevel] [Target1]

random [NewLevel] [Spawn] -> [NewLevel] [Target2]

random [NewLevel] [Spawn] -> [NewLevel] [Target3]

startloop

[Wall Spawn] -> [Wall]

endloop

Algorithm A.4 Rules used to generate the Rush Hour pieces in PuzzleScript Next.

(Level generation)

```

random [NewLevel] [FirstWaveSpawn] -> [NewLevel] [VBlock1]
random [NewLevel] [FirstWaveSpawn] -> [NewLevel] [VBlock2]
down [VBlock1 | SpawnVComplement] -> [VBlock1 | VBlock1]
down [VBlock2 | SpawnVComplement] -> [VBlock2 | VBlock2]

random [NewLevel] [SecondWaveSpawn] -> [NewLevel] [HBlock1]
random [NewLevel] [SecondWaveSpawn] -> [NewLevel] [HBlock2]
random [NewLevel] [SecondWaveSpawn] -> [NewLevel] [HBlock3]
right [HBlock1 | SpawnHComplement] -> [HBlock1 | HBlock1]
right [HBlock2 | SpawnHComplement] -> [HBlock2 | HBlock2]
right [HBlock3 | SpawnHComplement] -> [HBlock3 | HBlock3]

```

Algorithm A.5 Example of one of the Ludemes used to push a box into one of the goals and the Ludeme that replaces the box and the goal by an object defined as the overlap between them.

```

(define "MoveBoxUpToGoal"
  (move Step
    (directions {Forward})
    (to
      if: (and
        ("CheckForBox" (to))
        ("IsGoalAt" (ahead (to) steps:1 N))
      )
    )
    (then
      ("PushBoxOnGoal" ("NorthFromPlayer" "FindPlayerPiece"))
    )
  )
)

(define "PushBoxOnGoal"
  (and
    (remove #1)
    (add (piece "G") (to #1))
  )
)

```

Algorithm A.6 Example of the Ludemes describing the pieces that are used for this Rush Hour implementation. Notice that each half of a piece can move to only one direction.

```
(piece "Car2" P1 "MoveLeft")
(piece "Car1" P1 "MoveRight")

(piece "CircleA1" P1 "MoveRight")
(piece "CircleA2" P1 "MoveLeft")

(piece "SquareA1" P1 "MoveDown")
(piece "SquareA2" P1 "MoveUp")
```

Algorithm A.7 Ludeme used to place objects at random when creating a new Sokoban level.

```
(start
  {
    (place Random {"3" "X" "X" "X" "0" "0" "0" "0" "0"})
    (place Random (sites "BoxSpawns") {"Box"} count: 3)
  }
)
```

Algorithm A.8 Excerpt of the Interaction Set for Rush Hour in PyVGD. Functions are only called when two objects interact.

InteractionSet

```
    avatar wall > stepBack
wall wall > teleportInLevel

avatar hori1 > pushSpecificCar
avatar hori2 > pushSpecificCar
avatar ver1 > pushSpecificCar
avatar ver2 > pushSpecificCar
avatar car1 > pushSpecificCar
avatar car2 > pushSpecificCar

hori1 hori2 > carPushingCar
hori2 hori1 > carPushingCar
ver1 ver2 > carPushingCar
ver2 ver1 > carPushingCar
car1 car2 > carPushingCar

car2 hole > transformTo stype=carout
```

Algorithm A.9 Function implemented on effects.py to handle pieces being pushed onto walls.

```
#While the undoAll would normally work for walking into walls
#due to small pieces pushing each other in a chain it results in pieces
#taking more than 1 tick to move properly and the undo to not be
#able to redo everything.
#The global variable chainPushDirection is used to keep track of
#what direction a chain of blocks being pushed would go.
def pushCarBack(sprite,p,game):
    global chainPushDirection
    if sprite.key == 'hori1' or sprite.key == 'car1':
        sprite.rect = pygame.Rect((p.rect.x+1,p.rect.y),p.rect.size)
        chainPushDirection = 1
    if sprite.key == 'hori2' or sprite.key == 'car2':
        sprite.rect = pygame.Rect((p.rect.x-1,p.rect.y),p.rect.size)
        chainPushDirection = 2
    if sprite.key == 'ver1':
        sprite.rect = pygame.Rect((p.rect.x,p.rect.y+1),p.rect.size)
        chainPushDirection = 3
    if sprite.key == 'ver2':
        sprite.rect = pygame.Rect((p.rect.x,p.rect.y-1),p.rect.size)
        chainPushDirection = 4
```

Algorithm A.10 Function implemented on effects.py to handle part of the level generation for Rush Hour.

```

phase = 0
chainPushDirection = 0
goodSpawn = True

def placeCarSpawn(sprite,partner,game, index=0, check=True):
    global phase
    global goodSpawn
    goodSpawn = check
    position = 0
    xlist = [6,5,7,6,4,4,5,6,6,4,3,4,6,7,2,1,1,2,3,4]
    ylist = [1,2,2,3,1,4,5,4,1,2,4,6,5,6,1,2,4,6,5,7]
    if index == 0:
        position = random.randint(0,3)
    elif index == 1:
        if phase < 2:
            return
        position = random.randint(4,8)
    elif index == 2:
        if phase < 5:
            return
        position = random.randint(9,13)
    elif index == 3:
        if phase < 9:
            return
        position = random.randint(14,19)
    xpos = xlist[position]
    ypos = ylist[position]
    sprite.rect = pygame.Rect((xpos,ypos),sprite.rect.size)

```

B. Levels used for testing solvers

All levels are shown in this appendix as they appear in PuzzleScript Next but the location of each object was recreated in the same places for each framework nonetheless.



Figure B.1: The five Sokoban levels used to test the solvers shown in Puzzlescript.

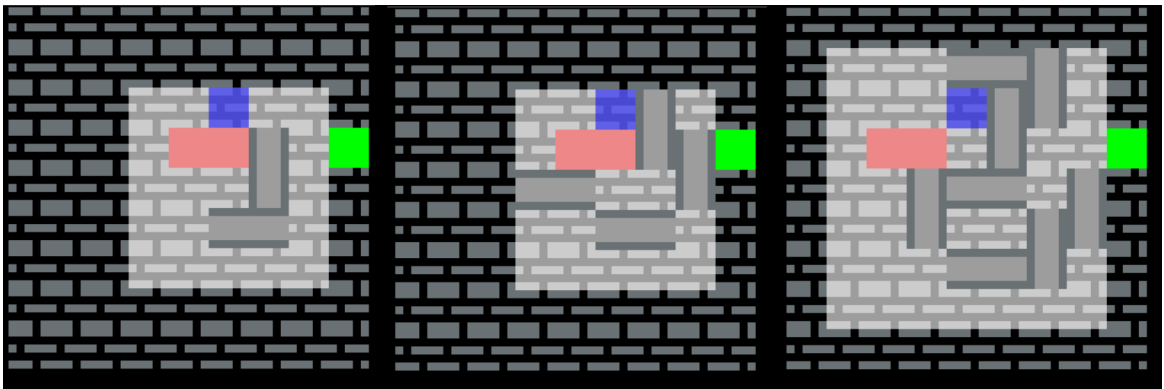


Figure B.2: The three Rush Hour levels used to test the solvers shown in Puzzlescript.