OTTO-VON-GUERICKE UNIVERSITY MAGDEBURG

FACULTY OF COMPUTER SCIENCE
INSTITUTE OF KNOWLEDGE AND LANGUAGE ENGINEERING

MASTER THESIS

Procedural Level Generation with Answer Set Programming for General Video Game Playing

Author:
Xenija NEUFELD

Supervised by:
Prof. Dr. Sanaz MOSTAGHIM
Dr. Diego PEREZ-LIEBANA

December 1, 2015

Statutory declaration

I assure that this thesis is a result of my personal work and that no other than the indicated aids have been used for its completion. Furthermore I assure that all quotations and statements that have been inferred literally or in a general manner from published or unpublished writings are marked as such. Beyond this I assure that the work has not been used, neither completely nor in parts, to pass any previous examination.

Magdeburg, December 1, 2015 *Xenija Neufeld*

Abstract

This thesis proposes an automatic way of level generation for arbitrary games that are described in Video Game Description Language (VGDL). Procedural Content Generation has become a popular technique in the last years. Different approaches have been investigated and optimized for different purposes. The most of them are search-based and need a fitness function for the evaluation of generated solutions. For that reason, they can only be used for the application they were created for and are not applicable for a *general* generator. To overcome this problem, we use Answer Set Programming (ASP) which is a constraint-solving method. It finds solutions through deductive reasoning and does not require a fitness function.

The proposed approach works as follows: the VGDL description of a game is transformed into ASP rules. Along with some style constraints, these rules provide multiple levels. Though, not all of these levels are solvable and well-designed. For this reason, a simple Evolutionary Algorithm (EA) is used to optimize the quality of the levels. The created ASP generators are evaluated with the help of game-playing agents with different skill levels. Thereby, we use the difference between their performances as a measure of level quality, assuming that the difference should be higher for well-designed levels. Finally, the generators are evolved using simple mutation operators. The experimental work performed in this thesis consists of level generation for 20 games provided by the GVG-AI framework. It shows that it is possible to create interesting level generator using this approach and outlines some possible directions for future work.

Contents

Chapter 1

Introduction

In the last few years, Procedural Content Generation (PCG) has become very popular in the video games industry and in research. This way of automatic creation of game content provides multiple challenges and leads to various interesting outcomes. There are many different methods to generate content procedurally, with *content* being anything that is needed to create a game.

In this thesis, we concentrate on the procedural generation of *game levels* or *maps*. There are multiple different algorithms for generating levels for certain video games. In most cases, these approaches are search-based and need a fitness function for the evaluation of the generated levels. They perform well, creating levels for the game they are built for. Nevertheless, they are hardly applicable to other games because every game has its own game mechanics, rules and goals. For that reason, a unique fitness function is needed for every game and the search space changes depending on the game.

Having to deal with this difficulty, it is even more interesting to create a *general* generator that is able to build levels for *any* game. However, for that purpose, we need an approach that does not rely on a specific fitness function and can work with any search space without being bound to the domain of a specific game.

As a solution of that problem, we propose a level generator that is able to read game descriptions written in Video Game Description Language (VGDL) and can create levels for any of these games fully automatically. Our generator consists of two main parts, using Answer Set Programming (ASP) and an Evolutionary Algorithm (EA). First, it uses the VGDL-descriptions to generate maps with the help of ASP and then the EA is used to optimize the difficulty levels of the maps.

An advantage of ASP is that the domain of a game, including the representation of its content, mechanics, and rules can be defined as logical expressions, which can be used by ASP to find solutions through deductive reasoning. That way, no fitness function is required at this point and the search space can be reduced by adding further constraints to the definition of the problem. Moreover, ASP provides a possibility to optimize certain criteria of the generated solutions.

In this work, we describe the structure of the levels in form of ASP rules. Therefor, we use the VGDL descriptions of games provided by the GVG-AI framework. Currently, there are 60 single-player games available for the framework. It is important to highlight that the generator is kept general, so that not all information contained in the descriptions can be converted into ASP rules. For that reason, we complete the missing information by adding random rules for some unknown features of game objects. This way, we change the shape and the size of the search space guiding the generator into different search directions.

Furthermore, we add some style constraints that optimize the horizontal and vertical balance of game objects inside the levels. We assume that the optimization of these features should increase the aesthetic value of the maps improving the visual impression on players.

After the generation of the levels through ASP, we continue with a simple EA. We use 2 game-playing agents with different skill levels to evaluate the quality of the generated maps. As a measure of map quality we concentrate on the difficulty levels of the maps. Therefor, we compare the game scores achieved by the agents assuming that the difference between the scores should be higher for well-balanced levels. Whereas maps with the difficulty level being too low (or too high) should be solved almost equally well (or bad) by both agents. Additionally, using the agents, we can test whether the generated levels are solvable.

With the help of the agents' scores, we select the best individuals following a simple elitism strategy. Finally, the evolution of the levels is done by mutating the randomly created ASP constraints.

This thesis is structured as follows: the current chapter outlines the motivation for the thesis. Chapter 2 describes some approaches used for PCG. Chapter 3 provides background on VGDL, the GVG-AI framework and ASP. Then, Chapter 4 describes the process of level generation through ASP, followed by Chapter 5 that details the evolution of level generators. Chapter 6 concentrates on the experimental work performed for this thesis. It includes the creation of levels for 20 games provided by the GVG-AI framework and describes some general findings. Finally, Chapter 7 concludes the thesis and outlines some directions for future work.

1.1 Motivation

Traditionally, game levels are created by a game designer. His task is to design different levels in accordance with given game rules and make sure that they are solvable. Furthermore, these levels should be interesting to play providing immersive experiences to players. In some cases, the maps have to show different difficulty levels, starting from an easier one and getting more difficult with growing player skills.

Depending on various factors, such as the size of a map and the variety of assets in a game, the process of designing a level can take from some hours up to several days. This can turn the development pipeline into a long and expensive procedure, especially for games where a large, or even infinite, number of levels is needed. Designing new levels procedurally could shorten the development time and give game designers more time for other tasks.

Furthermore, as the design process is done by a human being, the diversity of game levels is bound to the limits of the designer's creativity. Thus, at some point, a game designer is likely to create similar levels making them more predictable and less interesting for a player. A *procedural* level generator could avoid such repetitions and depending on the underlying algorithm, achieve a considerable diversity of maps. These levels could have unexpected appearances being completely different from those of a human designer.

A *general* level generator that is able to create maps for *any* game is interesting for multiple reasons. Having such a generator as a tool can be an advantage for a game company. Especially, those companies, that specialize at the creation of multiple small games e.g. for mobile devices could benefit from this tool. Often, such games implement different game mechanics and rules having small maps with a lot of different assets. Without being bound to a certain game, the generator could be used in several projects providing appropriate maps for all of them and saving a lot of resources.

Going one step further, this generator could be used for creation of completely new games. Since it should be able to read any game description, new rules, game mechanics and assets could be invented or recombined from old ones and put into the generator. This way, the generator would provide solvable and *interesting* levels for the newly invented game. These levels, in turn, would help game designers to understand how the change of rules would influence the appearance of the game and guide them in further development steps.

Chapter 2

State of the Art

Procedural Content Generation (PCG) is a way of content creation for an application with the help of an algorithm. In general, this algorithm is able to create the content using only some information stored inside of it. In this work, we speak of generating content for video games where *content* can mean any kind of asset that is used to build a game such as textures, models, maps, game mechanics and rules, quests and audio files.

Currently, PCG methods are widely used by the game industry for creation of different kinds of content. There are several tools that help game designers to create game worlds and cities procedurally. Also, more and more games rely on procedural quest generation.

In the academia, PCG is categorized as a sub-area of artificial and computational intelligence in games as it is described in [1]. Especially during the last few years, PCG has gained a lot of interest from researchers. It provides a basis for the research on different techniques such as search-based methods or constraint-solving techniques. The major groups of PCG techniques will be described in sections 2.1-2.4. The most of the current applications of PCG investigate a specific problem and have not (yet) been applied across different areas (games).

For classifying all the different methods, Togelius et al. [2] introduce a taxonomy of PCG describing the following seven dimensions in which an individual method could be located:

- *Online versus offline*: defining whether the content is generated while playing the game or before the game start

- *Necessary versus optional*: defining whether or not the content can be discarded

- *Degree and dimensions of control*: defining the degree of control over the generation space
- *Generic versus adaptive*: defining to what extent player behavior is considered during the generation process
- *Stochastic versus deterministic*: defining whether or not the same content is created given the same starting parameters
- *Constructive versus Generate-and-test*: defining whether the content is generated once or after several runs through the generate-test loop
- *Automatic generation versus mixed authorship*: defining to what extent a human game designer is able to guide the generation algorithm into desired directions

While categorizing different PCG methods in these dimensions, Togelius et al. also describe some *desirable properties of a PCG solution* such as *speed, reliability, controllability, expressivity and diversity, creativity and believability* [2]. Depending on the application, current methods try to optimize these properties to different extents, often, taking tradeoffs between e.g. speed and quality.

2.1 Search-based Methods

Search based methods such as e.g. an Evolutionary Algorithm (EA) or exhaustive search are generate-and-test methods and belong to the mostly investigated methodologies for PCG in academia. The general idea behind such methods consists of applying a search algorithm on a search space representing the content and measuring the quality of found solutions with the help of a fitness function.

Depending on the kind of game content that should be generated, its representation can be e.g. a vector of numbers, a graph or even a sequence of predefined micro-patterns that show parts of the content as described for Super Mario Bros levels in [3, 4]. Levels for 2D games, in most cases, can be described by a grid of cells or *tiles*. That way, grids can be represented by a (two dimensional) array of integers with each integer representing one of the available game objects in the game world. This representation is often used for so called *Rogue-like* games or dungeons as shown in [5–7].

Apart from the content representation, the fitness function plays an important role in a search algorithm. It defines a measure of the quality of solutions and guides the search process towards better results. Though, for the search process to work properly, the fitness function has to be designed carefully modeling desired properties of the content. In some cases, it is possible to define measurable properties of the content such as the number of particular objects on a map or the distance between them. Though, in most cases, finding the right fitness function is a difficult task.

In addition to a fitness function, constraints can be used to discard infeasible contents such as e.g. levels that are not solvable. Though, sometimes, even infeasible solutions are kept for future evolution steps like it was done for a strategy game in [8].

Furthermore, to overcome the difficulties of defining *one* fitness function that contains all necessary features, multi objective optimization can be applied using multiple fitness functions. For example in [8] the weighted sum of six different fitness functions was optimized in the process of creation of levels with better competitive play between two players.

Here, the results have shown that the maps that were optimized on multiple functions were more appropriate for use in strategy games than those optimized on a single function. Although, the simultaneous optimization of multiple functions has shown limited efficiency and could become slower and more difficult with a higher number of functions.

Another experiment has used the NSGA-II algorithm for multi objective optimization creating maps for a similar strategy game [9]. It has shown comparable results having created playable maps.

An alternative search based method that does not require a fitness function is e.g. novelty search. Instead of optimizing a function it aims to reach a great diversity of solutions. A variation of novelty search was used in [7] where levels for the same problem as in the previous methods were generated. Results have shown that using novelty search requires a tradeoff between the diversity of solutions and the number of feasible solutions that are found by the algorithm.

In summary, search based methods can be used for PCG, especially for level generation. Though, they require a good choice of content representation and the fitness function to work properly. Furthermore, their efficiency can decrease with a growing search space and additional constraints are needed to filter out infeasible solutions.

2.2 Formal Grammars

Formal grammars are widely used in many different areas in computer science. A grammar consists of an *alphabet* and a set of *production rules*. These rules describe how a new string is build out of the *symbols* of the alphabet. Therefor, one or more symbols on the left-hand side of a rule are replaced by one or more symbols on the right-hand side of it. A generative grammar can be deterministic, with exactly one rule for each sequence on the left-hand side, or non-deterministic with multiple rules for the same group of symbols.

Some symbols are called *terminals* and cannot be replaced. Terminals are, normally, represented by lowercase characters. An example for a non-deterministic grammar with the terminal *t* would be: $S \rightarrow St;\ S \rightarrow t$. The uppercase character *S* is a *non-terminal* symbol that has two rules for its replacement. Conventionally, the character S is used as a *start symbol* of a grammar.

Originally, grammars were used for modeling natural languages. Though, during the last years, they were introduced in many other fields. For example, a special form of a grammar, called *L-System* is used for graphical interpretation and generation of plants. The mostly known example application of an L-System is *SpeedTree*, a tool that procedurally generates vegetation and is widely used in many games [1].

However, vegetation is not the only thing that can be procedurally generated with the help of a formal grammar. Recently, there have been several examples where grammars were used for generation of missions in a game [10,11] and level generation.

For example, Shaker et al. have used Grammatical Evolution for generation of Super Mario Bros levels [12]. Thereby, levels were represented by a set of *chunks* where a chunk was a game object such as e.g. a canon or a coin with distinguishable properties. A design grammar was created to build levels out of these chunks and the results from this grammar were evolved with an Evolutionary Algorithm. After having resolved conflicts such as overlapping of chunks, the results have shown playable levels.

It has been shown that formal grammars can be used for level generation. However, designing a grammar is a difficult task that requires prior knowledge of the content domain. Having designed the grammar correctly, it can be used to create new content very fast. Though, a grammar is very domain specific and cannot be used across different games.

[1] http://www.speedtree.com

2.3 Constructive Methods

Constructive methods, in contrast to e.g. search-based methods, produce only one solution per run. The general idea behind all constructive methods is that they start at one point and generate the content step-by-step. Shaker et al. distinguish between the two alternative ways of building dungeons or levels for platform games *space partitioning* and *cellular automata* [13].

Space partitioning is based on the idea, that the space (of a level) can be divided into subspaces building a tree. The leaves of that tree can represent a room or an empty space with the root node representing the entire space. Children of the same parent are connected through a corridor so that bigger rooms and dungeons can be created. The generation process of the tree can be random or follow certain rules. Resulting dungeons have a very organized structure without any overlapping areas.

Another way of building a dungeon is by using a cellular automaton. Cellular automata are used in many different areas in computer science. Consisting of a grid, they define a set of states and transition rules. Each cell of the grid is changing its state accordingly to the transition rules taking into account its own previous state and those of its neighbors in the defined neighborhood. That way, clusters of cells with the same state can be build which, used in level generation, can be recognized as dungeons.

Johnson et al. have used cellular automata for real-time generation of cave-like dungeons [14]. Each cell of the grid they used could be a wall, a rock or floor. With certain probabilities cells could change their state e.g. from floor to rock. Running for a certain number of times, the algorithm created tunnels with different widths. The aim of this work was to create infinite caves in real-time, so that a player could play a game forever. Results have shown this method as very effective for this purpose.

In general, constructive methods show good results in terms of efficiency when creating levels in real-time. Though, due to their nature, they provide limited control over the output.

2.4 Constraint Solving Methods

Constraint solving methods, as the name suggests, provide solutions to problems described through constraints. In contrast to search-based methods, where the entire search space is investigated, here, constraints reduce the search space. That way, only feasible solutions are provided with each of them meeting the desired criteria. A fitness function is not needed in that case.

Constraints can be expressed as hard constraints that cannot be violated or soft constraints that may be violated. Besides, a logic that describes the problem domain has to be formulated. This logic and the defined constraints are then passed to a program called *solver* which finds solution candidates to the logic problem. An example for a constraint solving method is *constraint propagation* which was used for level population in [15]. An alternative method is *Answer Set Programming* (ASP) which is described in more detail in section 3.3.

Recently, it has been shown that ASP can be used for procedural generation of game rules creating new variations of games [16]. Furthermore, it can be used to create not only levels for mazes and dungeons [17, 18] but also for puzzle games [19]. Furthermore, it was used to create new game mechanics in [20]. The game mechanics and the structure of the content can be defined through logical expressions.

Having sufficient knowledge about the game, it is possible to formulate constraints which, being solved, result in playable game levels. All formulated constraints are guaranteed to be met in all solutions.

Chapter 3

Background

In this section, we discuss in more detail the background about the essential components of the generator developed in this work. First, we describe VGDL, then, we provide a short description about the GVG-AI Framework and explain the concepts behind ASP.

3.1 Video Game Description Language (VGDL)

Many 2D games can be described with the help of VGDL which was originally implemented by Tom Schaul in Python [21]. Every game object can be defined as a sprite in the *SpriteSet*, including some properties like orientation and movement abilities. Furthermore, a *LevelMapping* section contains the representation of each object on the 2D map. The *InteractionSet* defines which effects are applied to objects when they collide with each other, and finally the *TerminationSet* defines which conditions have to be met for the game to end. An example description of the game *Aliens* (based on *Space Invaders*) is shown in Figure 3.1.

3.2 GVG-AI Framework

The GVG-AI framework was developed for the General Video Game AI Competition which takes place since 2014 [22]. It provides an environment for agents that should be able to play any game that is given to them. Thereby, the agents have access to only a small part of the information about the game without being able to read the game rules directly.

```
BasicGame
    SpriteSet
        base    > Immovable    color=WHITE img=base
        avatar  > FlakAvatar   stype=sam
        missile > Missile
            sam > orientation=UP color=BLUE singleton=True img=spaceship
            bomb > orientation=DOWN color=RED  speed=0.5 img=bomb
        alien   > Bomber stype=bomb   prob=0.01  cooldown=3 speed=0.8 img=alien
        portal  >
            portalSlow > SpawnPoint stype=alien cooldown=16 total=20 img=portal
            portalFast > SpawnPoint stype=alien cooldown=12 total=20 img=portal

    LevelMapping
        0 > base
        1 > portalSlow
        2 > portalFast

    TerminationSet
        SpriteCounter      stype=avatar       limit=0 win=False
        MultiSpriteCounter stype1=portal stype2=alien limit=0 win=True

    InteractionSet
        avatar  EOS  > stepBack
        alien   EOS  > turnAround
        missile EOS  > killSprite
        missile base > killSprite
        base bomb > killSprite
        base sam > killSprite scoreChange=1
        base   alien > killSprite
        avatar alien > killSprite scoreChange=-1
        avatar bomb  > killSprite scoreChange=-1
        alien  sam   > killSprite scoreChange=2
```

Figure 3.1: VGDL description of the game *Aliens*.

Being developed in Java, the framework gets the VGDL description of a game and a low-level map sketch for this game as input to run the game. Figure 3.2 shows a map sketch for the game *Frogs* on the left side and its high-level representation in the game on the right side.

The framework provides a graphical user interface (GUI) visualizing the play of an agent to its developer. Furthermore, it offers a possibility for a human player to play the game himself. In case that multiple runs of a game are required, the framework offers a possibility to do the runs without a GUI providing the desired output through a console.

The advantage of working with the framework instead of the VGDL descriptions directly is that the framework converts the entire game information into Java structures. That way, it is much easier to access certain parameters of game objects, without the need of reading strings from a text file. Furthermore, the frame-

```
wwwwwwwwwwwwwwwwwwwwwwwwwwww
w           wGw            w
w0===0========0============2
w0000==================00012
w00===000===000===========02
wwwwwwwww  wwwwwwwwwwwwwwwww
w   xxxxxx x   xxxxxxxxx   w
w-     xxx       xxx    xx w
w xxx  xxxxxxxx  xxxx  xxx w
w       A                  w
wwwwwwwwwwwwwwwwwwwwwwwwwwww
```

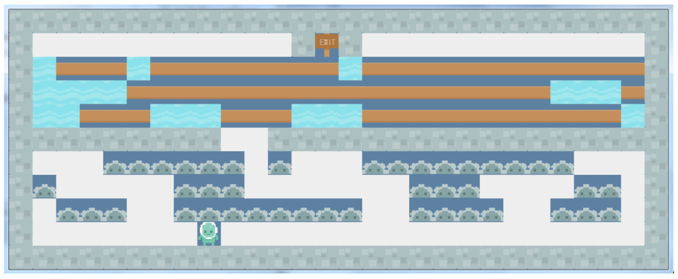


Figure 3.2: Map sketch for the game *Frogs* on the left side and its high-level representation in the GVG-AI framework on the right side.

work creates collections of e.g. all colliding effects in a game which make it possible to iterate through them and make queries.

Games Provided by the Framework

Even though we are speaking of a framework for *general* video game playing, the types of games that can be created and played in this framework are constrained by several criteria. Currently, the framework provides 60 manually designed games. All those games are 2D, one-player games that use grid levels. Each game object has the size of exactly one cell (tile) in that grid.The games are run in real time giving an agent 40ms to perform an action.

A collision between two game objects is recognized when both of them are on the same tile. There are two types of effects of a collision, those that have an impact on *one* of the colliding objects and those that affect *both* participants. Currently, there are 9 binary effects and 13 unary effects defined in the framework that can be used in the VGDL description of a game. Adding a new effect in VGDL would require its Java-definition in the GVG-AI framework.

A player can perform the following 5 actions: *left, right, up, down, use*. Depending on the type of the game, a subset of these actions is actually available, i.e. in some games there is no need for the player to *use* anything, so, only 4 actions are available to him. The types of actions that the player is able to perform define the type of the player avatar. In total, there are 6 different avatar types such as e.g. a *HorizontalAvatar* that is able to move left and right and a *ShootAvatar* that has an orientation in the game world and can perform all actions. The type of the avatar can also be specified in VGDL.

As can be seen on the list of actions, the player can only move vertically or horizontally inside the grid. Thus, games where a *jump* action should be implemented, would require some physics effects. These are already implemented in the framework, though, no sufficient testing has been done on them and no games implementing physics are (yet) provided by the framework.

To make the games interesting to play and provide different difficulty levels, there are 6 different types of non-player-characters (NPCs) implemented in the framework. As well, as the player, NPCs are able to shoot/use objects and move in different directions. Some of them show intelligent behavior trying to chase the player avatar or run away from him and some perform random actions. Also, the types of NPCs can be specified in the VGDL description.

The framework provides some example sprites. Though, since every game object has the size of one tile, it is very easy to change the sprites and create your own assets. Furthermore, new levels for games can easily be created in a text file considering the level mapping given in the VGDL description.

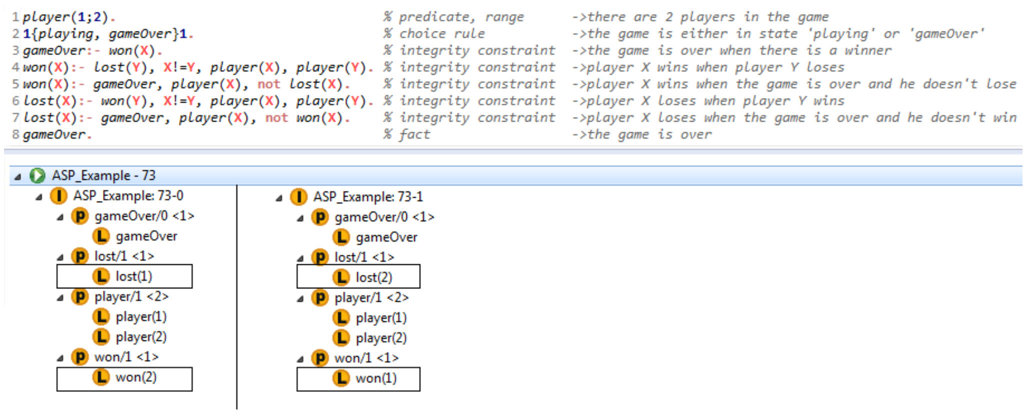
3.3 ASP

Answer Set Programming (ASP) is a declarative programming method which can be used to describe rather *what* problem is to be solved instead of *how* it is to be solved. Therefore, the problem needs to be expressed through logical terms formulated in a specific language.

The language used in ASP is called *AnsProlog*. It is very similar to *Prolog* with the difference that an AnsProlog program always terminates whereas a Prolog program can have infinite loops. The terms formulated in AnsProlog can be atoms/facts (simple statements), predicates (statements with parameters), choice rules (allowing a choice on elements) or integrity constraints (allowing conditions). Figure 3.3 shows a simple ASP program with each type of expression implemented there. Each line has a comment describing the kind of the expression and its meaning.

When the program logic is formulated, it can be passed to a so-called *grounder* program such as *gringo*[1]. A grounder translates the given problem into a propositional logic program which can be then passed to a *solver* e.g. *clasp* or *smodels*[2] A solver provides solutions to the logic problems as answer sets using deductive reasoning. The lower part of figure 3.3 shows the two answers delivered for the problem that was defined on the top of the figure.

[1]Clasp, Gringo, Clingo: http://potassco.sourceforge.net/index.html
[2]Smodels: http://www.tcs.hut.fi/Software



```
player(1;2).                                       % predicate, range      ->there are 2 players in the game
1{playing, gameOver}1.                             % choice rule           ->the game is either in state 'playing' or 'gameOver'
gameOver:- won(X).                                 % integrity constraint  ->the game is over when there is a winner
won(X):- lost(Y), X!=Y, player(X), player(Y).      % integrity constraint  ->player X wins when player Y loses
won(X):- gameOver, player(X), not lost(X).         % integrity constraint  ->player X wins when the game is over and he doesn't lose
lost(X):- won(Y), X!=Y, player(X), player(Y).      % integrity constraint  ->player X loses when player Y wins
lost(X):- gameOver, player(X), not won(X).         % integrity constraint  ->player X loses when the game is over and he doesn't win
gameOver.                                          % fact                  ->the game is over
```

```
ASP_Example - 73
    ASP_Example: 73-0                 ASP_Example: 73-1
        gameOver/0 <1>                    gameOver/0 <1>
            gameOver                          gameOver
        lost/1 <1>                        lost/1 <1>
            lost(1)                           lost(2)
        player/1 <2>                      player/1 <2>
            player(1)                         player(1)
            player(2)                         player(2)
        won/1 <1>                         won/1 <1>
            won(2)                            won(1)
```

Figure 3.3: A simple ASP program on the top and its two solutions on the bottom. The program describes a two-player game that can be in one of the states *playing* or *gameOver* having exactly one winner and one loser.

There are several programs available which combine a grounder and a solver. One example of such a program is *clingo* which was developed at the university of Potsdam. *Clingo* is used in the process of this work since it provides a good documentation and can easily be executed not only from *Eclipse* but also from the command line. A detailed users guide for *clingo* can be found in [23].

Independently from the choice of the solver and the search algorithm behind it, resulting answer sets will be equivalent due to the semantics of ASP [18]. Though, some tools provide ways for simplified search during the execution. For example, when optimizing a certain parameter in the logic program, *Clingo* is able to save the *best* solution found so far and discard all solutions that are worse during the search process [23].

Every solution that is found by a solver is guaranteed to satisfy all constraints defined in the logic program. By formulating certain constraints, the design space can be incrementally specified and the search space is reduced. That way, undesired answers are excluded. When using ASP for procedural level generation, desired properties of a level can be formulated incrementally as constraints and appropriate levels will be generated. For this reason, ASP can be used for PCG without using any evaluation function.

Nevertheless, the desired solutions can only be found if the game logic and all dependencies between game objects are completely known to the user and are correctly expressed in ASP. Some detailed explanations on building the design space for a game are described in [16].

Chapter 4

Map Generation through ASP

Having introduced the basic components of our level generator, we now describe how the generator creates the levels. Therefore, we describe the basic ASP rules and some auxiliary methods that are used in the process of rule formulation.

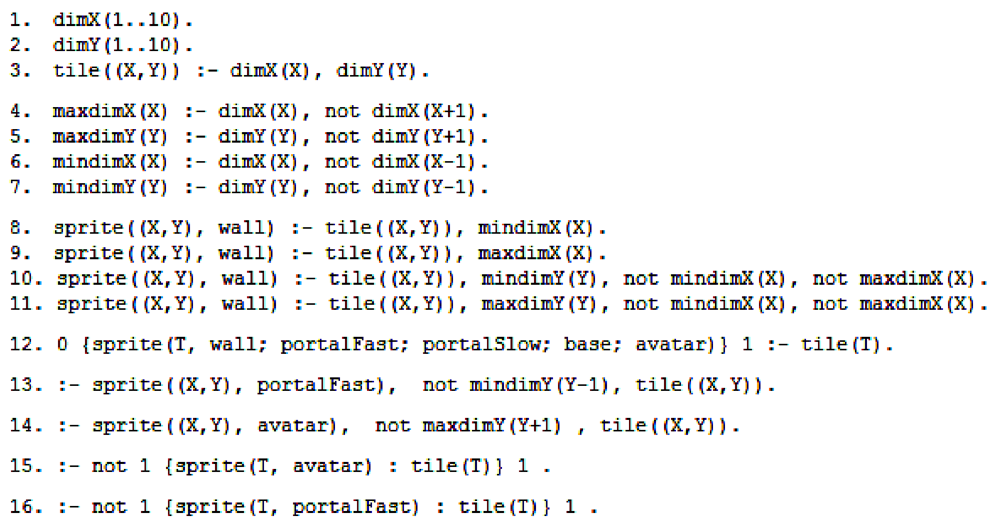
Afterwards, we take a closer look at how the information from VGDL descriptions is read in and handled by the GVG-AI framework. Then, we go into more detail describing how game specific rules are derived from the VGDL descriptions. Furthermore, we outline some rules that are used to improve the visual impression of the generated levels.

Since it is not possible to infer all dependencies between game objects, we then use an evolutionary algorithm which adds missing rules. All rules are written by the generator into a separate file for each game which is finally passed to the ASP solver for map creation.

To find levels that are more suitable for a certain game, we evaluate them by letting two game-playing agents play them. The evolutionary algorithm used in the generator and the evaluation of the generated levels is described in the next chapter.

4.1 Basic ASP Rules

Before concentrating on the game descriptions expressed in VGDL and creating game specific ASP constraints, we create some basic rules. As already mentioned in section 3.2, the games that are provided by the GVG-AI framework are limited by some properties. Therefore, they all have some parameters in common. Thus, there are some basic ASP rules that are common for every game and can be used



```
dimX(1..10).
dimY(1..10).
tile((X,Y)) :- dimX(X), dimY(Y).

maxdimX(X) :- dimX(X), not dimX(X+1).
maxdimY(Y) :- dimY(Y), not dimY(Y+1).
mindimX(X) :- dimX(X), not dimX(X-1).
mindimY(Y) :- dimY(Y), not dimY(Y-1).

sprite((X,Y), wall) :- tile((X,Y)), mindimX(X).
sprite((X,Y), wall) :- tile((X,Y)), maxdimX(X).
sprite((X,Y), wall) :- tile((X,Y)), mindimY(Y), not mindimX(X), not maxdimX(X).
sprite((X,Y), wall) :- tile((X,Y)), maxdimY(Y), not mindimX(X), not maxdimX(X).

0 {sprite(T, wall; portalFast; portalSlow; base; avatar)} 1 :- tile(T).

:- sprite((X,Y), portalFast),  not mindimY(Y-1), tile((X,Y)).

:- sprite((X,Y), avatar),  not maxdimY(Y+1) , tile((X,Y)).

:- not 1 {sprite(T, avatar) : tile(T)} 1 .

:- not 1 {sprite(T, portalFast) : tile(T)} 1 .
```

Figure 4.1: Simple ASP rules for the game *Aliens*.

in further statements.

Basic rules include definitions like e.g. the Manhattan distance measure and adjacency of tiles. Later, the first definition can be used e.g. for maximizing the distance between certain objects. The adjacency of tiles can be used to check whether there is a path between two objects. A path is then defined as a sequence of passable adjacent tiles.

Furthermore, the dimensionalities of a level are defined in the basic rules providing knowledge about level borders. Since without loss of generality levels of all games provided by the framework are surrounded by walls, one rule is formulated at this point, placing wall sprites at corresponding tiles.

4.2 Auxiliary Methods

As already mentioned, all ASP rules for a game are written into a separate file that is passed to the solver program *Clingo*. Since the generator is implemented In Java, there are certain methods that have direct access to that file and are responsible for writing rules into that file. For the Java program, ASP rules, as they are described in figure 4.1, are strings that contain the names of certain game objects, e.g. *portalFast, avatar* etc.

When designing the ASP program for a game, there can be many rules that are written for different game objects being similar in their definition. For example, there could be a rule saying that a *coin* should be reachable by the player, i.e. there should be a path between them, and a rule saying that a *diamond* should also be reachable by the player.In Java this would be one method called with two different parameters (coin and diamond). Though, from the prospective of a Java program, these rules are two strings with different substrings:

:- not reach(C1,C2) , sprite(C1, avatar), sprite(C2, coin). and

:- not reach(C1,C2) , sprite(C1, avatar), sprite(C2, diamond).

For the purpose of reusability, we implemented some auxiliary methods in the generator that can be called with different parameters to write such strings. Hereby, the parameters are written as substrings into the rule strings that are predefined within these methods. These methods can be called whenever new information about a game object is found and should be formulated as a rule. All rules defined in that auxiliary methods are basic rules which can be combined into more complex rules describing the dependencies between game objects in a better way.

For example, in the game *Portals* the player should be able to go through a portal and each portal entry is assigned to its own portal exit. Thus, there should be as many entries as exits in a level.

Combining the simple rule *CreateAsManyOthersAsOnes(exit, entry)* with the rule *CreateMinNumber(entry, 1)* would result in levels having the same amount of entries and exits with *at least* 1 pair of them. A full table of all auxiliary methods with example parameters, a brief explanation about their functionalities and resulting ASP rules can be found in appendix A.

4.3 Information Gain from VGDL Descriptions

The most challenging part of this work is the creation of game specific rules. Here, one important task of the generator is the identification of all properties of game objects and all dependencies between different objects. The next important step is the understanding of the meaning of this information for the game play and the formulation of corresponding rules, which is described in the next section.

Since the VGDL descriptions of games are given in text files, using the given information in form of strings would be very complicated and not well manageable. For that reason, the generator uses the implementation of the GVG-AI framework which reads the textual descriptions and provides corresponding Java representations.

With the help of the GVG-AI framework, the generator first creates an instance of the *Game* object. That way, all information from the description file is read in and provided in form of members of that object. Hereby, the most significant information, contains collections of defined *types of game objects, terminations* and *collision effects*. With each of them represented as an instance of the *Content, Termination* and *Effect* class accordingly, the generator is able to easily access their parameters. Furthermore, the *Game* class provides a mapping of character representations of objects on the map sketch to the names of corresponding types of game objects.

With these collections provided by the framework, all necessary information can be derived by the generator. However, for a better handling of that information, we introduce the two additional classes *SpriteInfo* and *AvatarInfo*. Using these classes, we can save parameters of a certain sprite type or the player avatar that are important for the formulation of ASP rules.

The class *SpriteInfo* saves, amongst other things, values such as the minimum/maximum number of a certain type of game object in the level. Though, the most important parameters that are saved for a sprite type are effects that are applied on collisions with this kind of game object. Collision effects represent the essential part of the VGDL descriptions, since they describe how the game works and which rules are valid in the game. Therefore, it is very important to be able to iterate through the defined effects and make fast queries.

In the VGDL description of a game, an effect is described in the following way: *avatar alien > killSprite scoreChange=-1*

Here, *avatar* and *alien* are the two game objects that collide and the effect of this collision is *killSprite*, i.e. the game object is killed or deleted from the game world with a negative score change for the player. Thereby, the defined effect is applied to the object that is named first, in this case the player avatar. If the *alien* should be killed as well on this collision, there would be another similar effect definition with the *alien* standing first and the *avatar* second.

Given this kind of effect descriptions, it is obvious, that there is always a game object that is *affected* (here, the player avatar) and one that is *affecting* (alien). For that reason, we save the corresponding effects for each type of game objects in two mappings inside the *SpriteInfo* class. One of those mappings maps the IDs of all game objects that are *affected by* this object type to corresponding effects (*HashMap<Integer, ArrayList<Effect>> affectedBy*). The other mapping does the same for objects that *affect* this object (*HashMap<Integer, ArrayList<Effect>> affects*).

That way, we are able to iterate faster though the effects and query the corresponding IDs of game objects. This information can be used e.g. if we know that an object *A* is transformed into an object *B* when it collides with object *C*. Now, we would like to know whether there is any object that transforms object *B* back into object *A*. For that, we iterate through the *affectedBy* mapping of object *B* and check for effects of the type *transformTo*. If there is such an effect defined in that mapping, we can easily get the ID of the object that causes this transformation.

Another class that is added in the generator additionally to the *SpriteInfo* class is called *AvatarInfo*. This class saves some of the necessary information about the player avatar that is not directly provided by the GVG.AI framework. As already mentioned in section 3.2, there are different types of avatars in the GVG-AI framework which are defined through the actions that are available to the avatar and through the specifications whether or not the avatar is oriented.

In case the player avatar is able to shoot in a game, among some other values, we save a list of all the types of game objects that are used as ammunition by the avatar. That way, it is possible to find out not only what the avatar affects directly, but also what can be affected by his weapons.

Furthermore, we save a list of all possible subtypes of the avatar including the corresponding parameters provided by the framework. An avatar can have several subtypes in a game with one of them being active at the beginning at the game. During the game, it is possible that the avatar transforms from one type into another. For that reason, it is important to know all his subtypes and check for effects not only of the one that is active at the beginning, but for those of all of his subtypes. The ID of the subtype that is active at the game start is saved additionally.

4.4 Game-specific ASP Rules

Having all the information described in the previous section, we are, theoretically, able to formulate specific ASP rules that describe the current game. This step could be easily done by a human game designer seeing *all* information at once. However, an automatic generation of rules remains a difficult task. The generator has to check the parameters of every single game objects, look for collision effects with other game objects, probably look for transitive effects and, estimating their meaning for the game, create corresponding ASP rules.

In this section, we outline the processes that take place during the rule creation and give some examples. All rules described here were derived from the game descriptions of the first training set provided by the GVG-AI framework and some from the second set. Further games were not taken into account yet. For that reason, we assume that there could be more rules added at this point to guarantee more generality of the generator.

After the basic rules are created and the generator has the information described in section 4.3, we start with a simple, although very important rule defining which game objects can be placed as a *sprite* on a tile/grid cell. Not all game objects that are listed in the *SpriteSet* of a VGDL description can be drawn on a map sketch. For example, game objects representing the ammo of the player cannot be located on the map at the beginning of the game, since they are produces by the player. For that reason, we check the *LevelMapping* of the VGDL description or rather the *charMapping* provided by the framework and add corresponding game objects to the rule that defines what a *sprite* can be. Additionally, we formulate a constraint that allows *maximal one* sprite to be placed on one tile.

According to the description of the game *Aliens* provided in Figure 3.1, this rule would look the following way:

0 {*sprite(T, wall; portalFast; portalSlow; base; avatar)* } *1:- tile(T).*

After defining which sprites are allowed, we investigate which of them are subtypes of the same object, i.e. they represent the same game object having slightly different parameters. For example, in the *Aliens* description in Figure 3.1, we see the lines:

portal >
  portalSlow > SpawnPoint stype=alien cooldown=16 *total*=20 *img=portal*
  portalFast > SpawnPoint stype=alien cooldown=12 *total*=20 *img=portal*

Here, the game object *portal* has the two subtypes *portalSlow* and *portalFast* with *portalFast* having a higher frequency of spawning *aliens*. However, both of them have the same function of spawning the same enemy entities. Having one subtype in the level would make only a little difference to having the other one. For that reason, we save all possible alternatives of each subtype in its *SpriteInfo* which we described in section 4.3.

While checking for sprites that represent the ammo of the player or an NPC, we also check whether any of them has an orientation. In most cases, a *bullet* is represented by the *Missile* Java class which has a parameter *orientation*. If the object, that fires this missile is also oriented and never changes its orientation and the *missile* also does not change its direction, we formulate a rule that places the shooting object on the map border that allows the longest way for the bullet to fly.

For example, Figure 3.1 shows the following lines:

portal > portalSlow > SpawnPoint stype=alien cooldown=16 total=20
alien > Bomber stype=bomb prob=0.01 cooldown=3 speed=0.8 img=alien
missile > Missile
    bomb > orientation=DOWN color=RED speed=0.5 img=bomb

This sequence shows that a *portal* which is immovable spawns an *alien* that is of the class *Bomber* and thus is oriented and moving right. The *alien* shoots with a *missile* which is also oriented. *Missile's* orientation is *down*. This means that the *aliens* are moving horizontally and dropping bombs down. For that reason, the aliens should be placed as close as possible to the upper border of the map.

However, aliens are not listed in the *LevelMapping* and cannot be drawn on the map directly. That is why, the ASP rules that are created at this point, say that every object of type *portalSlow* or *portalFast* should be placed on the uppermost row of the level grid. To be more exact, the constraints say that it is not possible, that a portal sprite is not on a tile of the topmost row.

:- sprite((X,Y), portalSlow) , not mindimY(Y-1) , tile((X,Y)).
:- sprite((X,Y), portalFast) , not mindimY(Y-1) , tile((X,Y)).

Checking for the *orientation* of the *Missile* is only one example parameter that has an impact on the game play and the structure of the level. For example, entities called *Resource* have parameters *value* and *limit* which can be used defining the desired amount of these objects in the level, while entities called *Flicker* have a limited lifetime which might be taken into consideration when defining their positions on the map. At this point, it is very important to have a good knowledge of all VGDL entities and their parameters to be able to define further important ASP rules.

After defining sprite specific rules, we use the information that is saved in the *AvatarInfo* class (described in section 4.3) and create avatar specific rules. Since all games taken into consideration, are one player games, we define a rule that places exactly one avatar onto the map. For games with more than one player, this rule could be easily adapted. Depending on the type of the avatar and its orientation, we define another rule that places the avatar on a border of the level following the same principles as described in the *Aliens* example above.

In the next important step, we go through all termination conditions defined for the game and create ASP rules that make sure that a game is not terminated at its beginning, i.e. termination conditions are not satisfied. There are two types of terminations defined in the framework, a *Timeout* and a *SpriteCounter*. The former says that a game is over after a certain amount of time and the latter says that it is over when there is a particular number of certain entities in the game.

Since the time condition can hardly be used on level generation, we concentrate on the counter conditions.

A *SpriteCounter* condition defines the end of the game specifying the object type and the limit of these objects in the level. Furthermore, it provides a boolean value defining whether the player wins or loses the game meeting this condition. In most cases, the limit is set to 0, meaning that the game is over when there are no more objects of the given type in the game (i.e. they were collected or destroyed). For example, the following termitation condition says that the player loses the game when there is no avatar in the game:

SpriteCounter stype=avatar limit=0 win=False

However, in some cases the limit is set to a positive number meaning that there should be exactly that many objects in the game for it to end, as e.g. the game *Eggomania* is over when there is *one* broken egg:

SpriteCounter stype1=brokenegg limit=1 win=False

Furthermore, some games can be ended with the same result by satisfying one of multiple conditions. For example, the player can win the game *Aliens* when he either destroys all aliens in the level or when he destroys the portal that spawn the aliens. These two conditions can be defined through a *MultiSpriteCounter* that, in contrast to a *SpriteCounter*, specifies the limit of multiple game objects. Thereby, the limit has to be reached for at least *one* of the specified object types as shown in the following line from the *Aliens* description:

MultiSpriteCounter stype1=portal stype2=alien limit=0 win=True

Having the knowledge about the termination conditions, we can derive the minimum or maximum amounts of the given game objects and save these amounts in the corresponding *SpriteInfo* instances to make sure that these numbers are never changed. Furthermore, we can now directly define ASP rules saying that these conditions should not be satisfied at the beginning of the game by simply negating the condition. For example, the ASP rule for the following description line:

SpriteCounter stype=avatar limit=0 win=False

would look the following way:

counter(avatar, N) :- N = #count{sprite((X,Y), avatar)}.

:- counter(avatar, 0).

saying that it cannot be true that there is no avatar in the level.

Additionally to the numbers of game objects, we define further ASP rules that are meant to make the level a little more difficult and interesting to play. For example, a very common game mechanic is going through a door to end the level. To force the player to perform multiple actions before reaching the door, we make sure that he is not placed close to that door at the beginning of the game.

According to the VGDL description structure, that game rule would be defined as a collision between the door and the avatar with the door sprite being deleted from the level. So, if a termination condition says that the game is over when there is no object of a certain type, such as e.g. an *exitdoor*, we check whether there is such a collision defined for it. Therefore, we access the *affectedBy* array of the *exitdoor* (described in section 4.3) and search for collisions with any subtype of the avatar. In case that we find such a collision, we maximize the distance between the door and the player using the *MaximizeDistance* method described in section 4.2 .

In some cases, the subtype of the avatar that collides with the door is not the same as the subtype that is placed on the map at the beginning of the game. That means that the player has to satisfy a certain condition to be transformed into another type. For example, in the game *Zelda* the player starts the game as the subtype *nokey*. He has to collect/collide with the game object called *key* to be transformed into the subtype *withkey*. Being an entity of the type *withkey*, he is able to end the game by colliding with the entity *goal*. The following lines show the corresponding parts of the VGDL description:

A > nokey
key avatar > killSprite scoreChange=1
nokey key > transformTo stype=withkey
goal withkey > killSprite scoreChange=1
SpriteCounter stype=goal win=True

With these game rules, the player has to perform even more actions to end the game. To make the level even more interesting, we should not only maximize the distance between the avatar and the goal but also the distance between the avatar and the key and distance between the key and the goal. For that purpose, we check whether the subtype of the avatar is different from the subtype that is needed at the end. If so, we search for collisions that transform the avatar and add corresponding rules maximizing the distances between the three object types (avatar, key, goal).

Looking for transformations, we also check if the avatar is ever transformed into any other subtype. In case there is only *one* transformation defined in the VGDL description (e.g. *nokey* transformed into *withkey*) and the entity is never transformed into something else, we can define a rule saying that there should be exactly *one* game object that effects this transformation. As in the case of *Zelda*, there should be only one *key* in the whole level.

After going through all termination conditions, we continue with an investigation of all collision effects defined for a game. As already mentioned, collision effects indirectly define the mechanics and rules of a game. Therefore, it is very important to know how each single effect is defined in the VGDL and the framework and to understand what impact it has on the game play.

In the final part of the definition of game specific ASP rules, we go through all effects and create corresponding rules. At this point, we would like to emphasize, that currently the rules are built based on the effects defined in the first game set of the GVG-AI framework and therefore take into consideration only a subset of all possible effects.

As an example for such an effect-rule pair we take the effect called *KillIfOtherHasMore* as shown in the following line from the description of the game *Boulderdash*:

exitdoor avatar > killIfOtherHasMore resource=diamond limit=9

In this case, the *exitdoor* disappears on collision with the *avatar* if the player has collected more than 9 *diamonds*. This information implies that there should be *at least* 10 *diamonds* in the level. Thus, having an effect *KillIfOtherHasMore* we set the minimum number of diamonds on the map to 10.

Another effect is called *BounceForward*. It causes an object to move into the opposite direction from the one in which the collision took place. That way, the player is able to push boxes (into holes) in the game *Sokoban* as can be seen in the following line:

box avatar > bounceForward

Since the player needs to reach the box from one side and the box has to be pushed into the opposite direction, we create a rule allowing a box being placed only in the following way: at least two opposite neighbors of it should not be immovable and indestructible at the same time. That means a box can have a wall (which is immovable and indestructible) as its left neighbor but then its upper and lower neighbors cannot be immovable. Though, a box can be completely surrounded by other boxes, since all of them are movable so that the space around the box in the middle can be freed by pushing away the surrounding ones. Figure 4.2 shows three examples that can be created following this rule as well as a configuration that is not possible.

The last case that we describe here is the collision effect called *TeleportToExit*. For example, in the game *Portals* the player can enter a door that represents the entity *portalentry* and appear in front of another door that represents the *portalexit*. Thereby, each *portalentry* is assigned to its own *portalexit*. Choosing the right sequence of doors, the player should reach his goal.

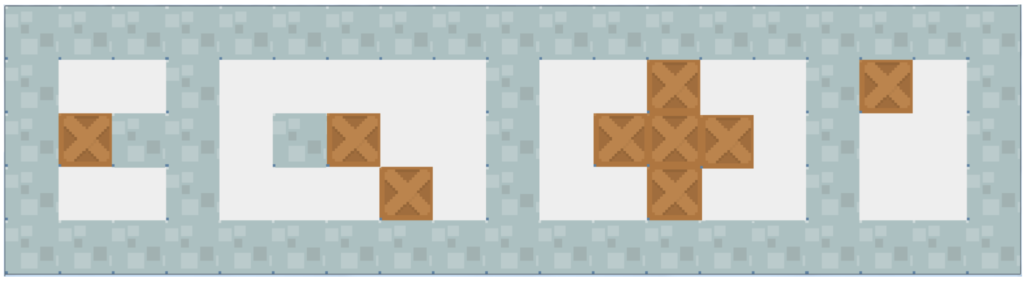


Figure 4.2: Boxes in the first 3 levels on the left can be pushed by the player. The box in the rightmost level cannot be pushed.

Having the following lines of the VGDL description, we see that a player should teleport on a collision with a *portalentry* and we are able to find out how the door pairs are called in the current game (*entry1, exit1*):

portalentry > Portal img=portal

entry1 > stype=exit1 color=LIGHTBLUE

avatar portalentry > teleportToExit

Knowing that the framework assigns an own exit to each entry, we now create an ASP rule saying that there should be the same amount of both door types in the map.

After creating the game specific rules that were partially described in this chapter, we finally try to improve the *visual impression* of the resulting levels. Inspired by the work described in [24], we assume that levels with a balanced distribution of game objects are perceived by the player as more enjoyable and interesting as those having unequally distributed objects on the map.

Therefore, we add two rules for every game object providing that *vertical* and *horizontal balance* of these objects are sustained in every level. *Vertical balance* is achieved through a rule that minimizes the difference of the number of certain game objects on the left half and those on the right half of the map, whereas *horizontal balance* is obtained through a similar ASP rule for the upper half and lower half.

Optimization (in this case minimization) of certain criteria can be defined in ASP directly. Thereby, the rules describing the problem are regarded as soft constraints. Thus, having a rule (hard constraint) that defines that a game object should be placed in the uppermost row of a map, the *horizontal balance* rule will be ignored. The following lines show the two ASP rules balancing the distribution of wall sprites in a level:

hBalance(wall, N) :-

T = #count{sprite((X,Y), wall) : Y<=height/2},

B = #count{sprite((X,Y), wall) : Y>height/2}, N=#abs(T-B).

#minimize [hBalance(wall, N)=N@5].

vBalance(wall, N) :-

L = #count{sprite((X,Y), wall) : X<=width/2},

R = #count{sprite((X,Y), wall) : X>width/2 }, N=#abs(L-R).

#minimize [vBalance(wall, N)=N@5].

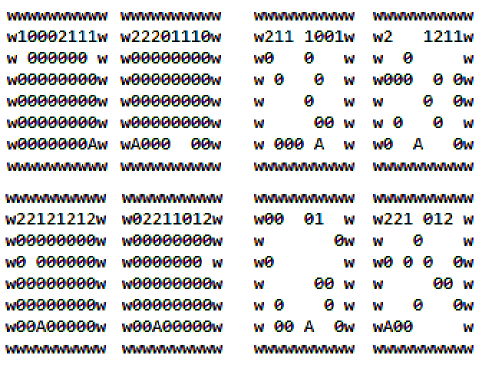
It is possible to optimize multiple functions at the same time in ASP. The importance of each function can either be set by its priority or by setting weights for each function and optimizing the weighted sum. Since the balance of all objects in a level is regarded equally important, we set equal priorities for all balance functions.

With the *visual impression* rules we finish the part of the work where we create rules that are specific for a given game. Following the steps described in this section we are able to create *some* rules that are based on the information from the VGDL description of a game. Though, not all dependencies between game objects can be derived from such a description. That is why the rules created so far are not always sufficient and the search space resulting from these rules is still too big. With the aim to decrease the size of the search space, we continue with additional rules that are described in the following section.

4.5 Additional ASP Rules

In the previous section we have already shown a few examples of how certain properties of game objects such as orientation or minimum amount can be derived from the VGDL description of a game. Though, there are still many properties that are not traceable by the level generator. For example, Figure 4.3 left shows levels of the game *Aliens* that have multiple portals spawning the aliens. A human designer would recognize that such a high number of portals is too difficult for the player to handle. Furthermore, these levels have too many base tiles which have to be destroyed either by the aliens or by the avatar.

Levels like these are created because there is no upper limit given on the number of portals or base sprites. Following the steps described in section 4.4 it is not possible to find out how many objects of these types should be in a level since there are no hints given in the VGDL description that is shown in Figure 3.1. That



```
wwwwwwwwww  wwwwwwwwww   wwwwwwwwww  wwwwwwwwww
w10002111w  w22201110w   w211 1001w  w2   1211w
w 000000 w  w00000000w   w0   0   w  w  0     w
w00000000w  w00000000w   w 0   0  w  w000  0 0w
w00000000w  w00000000w   w    0   w  w    0  0w
w00000000w  w00000000w   w     00 w  w 0   0  w
w0000000Aw  wA000  00w   w 000 A  w  w0  A   0w
wwwwwwwwww  wwwwwwwwww   wwwwwwwwww  wwwwwwwwww

wwwwwwwwww  wwwwwwwwww   wwwwwwwwww  wwwwwwwwww
w22121212w  w02211012w   w00  01  w  w221 012 w
w00000000w  w00000000w   w       0w  w   0    w
w0 000000w  w0000000 w   w0       w  w0 0 0  0w
w00000000w  w00000000w   w     00 w  w     00 w
w00000000w  w00000000w   w 0    0 w  w   0   0w
w00A00000w  w00A00000w   w 00 A  0w  wA00     w
wwwwwwwwww  wwwwwwwwww   wwwwwwwwww  wwwwwwwwww
```

Figure 4.3: Example maps for Aliens. Left: results of basic and game specific rules; right: results after adding horizontal/vertical balance rules and number constraints. (0 - base; 1 - portalSlow; 2 - portalFast; A - avatar; w - wall)

way, the ASP solver searches through a huge search space and creates levels that are not desirable.

To prevent the solver from creating such levels, we propose limiting the number of game objects that have no limits given through VGDL descriptions. Having saved the lower and upper limits of each game object for which it was possible in its *SpriteInfo* instance (described in section 4.3) the generator is able to identify missing limits. Each of the missing amount limits is then set to a random number between *one* and *one fourth of the map size*. That way, we can make sure that there is always *at least* one object of each type, and *at most* one fourth of the map size excluding maps that are full with objects of a single type as shown in Figure 4.3 left.

Furthermore, creating the limits for multiple subtypes of an object, we propose handling them in a single rule. As already described in section 4.4, a *portal* in the game *Aliens* is represented by a *portalSlow* and a *portalFast*. Setting the amount limits for these two subtypes, we create a single rule saying that there should be at least one of the two portalFast *or* portalSlow and having a map size of e.g. 80 grid cells, the generator creates a rule defining an upper limit of 20 for *both* of them *together*.

Limit constraints are one possibility to decrease the search space and exclude levels that are very likely to be unplayable. Another possibility is to artificially create dependencies between arbitrary game objects or add further properties using some of the auxiliary methods described in section 4.2.

For that purpose, we use the method called *CreatePassableCellsAroundObject* adding or deleting a rule for each type of a game object that specifies how many of its four neighbors (adjacent cells) should be *passable*. The chance of adding such a rule to a certain game object is 50%. Thereby, a passable tile is defined as one that the player can step on eventually collecting an item or destroying an object on that tile. Thus, tiles with immovable objects that the player cannot destroy (or collect) are regarded as *impassable*.

The information about passable neighbors is saved as well as every other information in its *SpriteInfo* instance. In some cases, this information can be obtained from the VGDL description, as described in section 4.4 for the boxes from the game *Boulderdash*. Though, if no prior knowledge about the neighbors is given, the generator sets the number of passable neighbors to a random number between 0 and 4.

That way, we can make sure that each game object can be accessed by the player from at least one direction (or cannot be accessed at all). Though, it does not necessarily mean, that the object is being placed in the middle of an empty space because it still can be surrounded by other objects that are movable or destroyable by the player. Setting the number of passable neighbors randomly, we are changing the borders of the search space limiting its size.

At the moment, setting the missing limits randomly and assigning a random number of passable neighbors are the only additional rules implemented in the generator. Although, other constraints could be added at this point using more of the auxiliary methods in future.

Chapter 5

Evolution of ASP Rulesets

In the previous section we have outlined how the generator creates a set of ASP rules for a game, given its VGDL description. Such a ruleset can be then put into the ASP solver which is able to find all possible solutions for the given problem. As already described, each generated ruleset contains basic, game specific and additional rules. Basic and game specific rules are assumed to fit the game description and exclude undesirable solutions. Whereas additional rules are created randomly and cannot guarantee that *the best* solutions are created or excluded through them. Thus, we cannot ensure that the desired subspace of the search space, i.e. the one with the *most interesting* levels, is observed.

As a solution to this problem, we propose evolving the randomly created additional rules through a simple evolutionary algorithm. By changing only the additional rules, we can make sure, that the most important part containing the basic and game specific rules sustains throughout the evolution process. Thereby, we still can slightly change the shape of the observed search space looking for a configuration of additional rules that provides solvable and the most interesting levels.

In this section, we describe the evolutionary process showing how a population of rulesets is created and evaluated. Furthermore, we go into more details describing how the additional rules of selected rulesets are mutated.

5.1 Population Creation

As already mentioned in section 3.3, a solver program provides all possible solutions of a given problem. Considering the relatively small number of constraints that are defined in the ruleset creation phase, the solver finds a very large amount of possible configurations of game objects on a map. Thus, we can obtain from *one* ruleset *many* maps that lie very close to each other in the solution space and have small differences in the positions of game objects. Though, all of these maps satisfy all constraints defined in a ruleset.

Since we are not interested in testing maps that are very similar to each other and we do not have the possibility to test *all* levels generated from a single ruleset, we propose choosing randomly 5 maps from a ruleset. The randomness and the number of desired solution can be set as parameters of the ASP solver. Though, to be able to reconstruct the same levels from a ruleset we set a fixed seed of the random generator as another solver parameter.

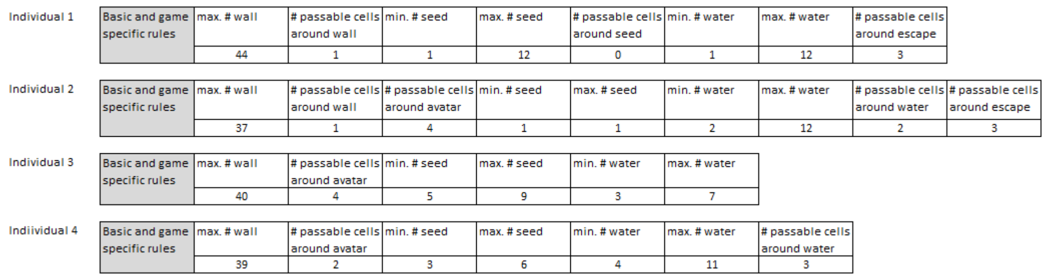
Furthermore, since we use optimization, balancing the distribution of game objects on a map, we set the corresponding parameter of the solver to *–opt-all*. That way, the solver finds only 'answer sets that are not worse than the best one found so far' [23]. Thus, each map provided by the solver is more balanced than the previous one. However, it is not guaranteed that the last map delivered is the optimum solution due to the random choice of solutions.

To further optimize the search for optimal solutions, we set another option of the solver called *–restart-on-model*. With this mode turned on, the solver restarts its search after finding an answer set recording solutions found so far. This way, the search can be sped up and the current optimum is always recorded [23].

Creating 5 levels from each ruleset, we propose generating small populations of 10 rulesets/individuals. Thereby, each ruleset has the same basic and game specific rules and different additional rules. That way, the genes of an individual save the information about the randomly generated amount limits of certain game objects. Furthermore, some genes can contain the information about the number of passable neighbor tiles of an object. As shown in figure 5.1, it is possible that individuals have different lengths, since the rule defining passable neighbors is added with a 50% chance for each object.

5.2 Fitness Evaluation

With all rulesets in a population containing game specific rules, we make sure that the most important rules are taken into consideration in all generated maps. However, due to the randomly created additional rules, we cannot guarantee that all levels are solvable. Furthermore, we do not know which configurations of additional rules provide *more interesting* levels.



Individual 1

Basic and game specific rules	max. # wall	# passable cells around wall	min. # seed	max. # seed	# passable cells around seed	min. # water	max. # water	# passable cells around escape
	44	1	1	12	0	1	12	3

Individual 2

Basic and game specific rules	max. # wall	# passable cells around wall	# passable cells around avatar	min. # seed	max. # seed	min. # water	max. # water	# passable cells around water	# passable cells around escape
	37	1	4	1	1	2	12	2	3

Individual 3

Basic and game specific rules	max. # wall	# passable cells around avatar	min. # seed	max. # seed	min. # water	max. # water
	40	4	5	9	3	7

Indiividual 4

Basic and game specific rules	max. # wall	# passable cells around avatar	min. # seed	max. # seed	min. # water	max. # water	# passable cells around water
	39	2	3	6	4	11	3

Figure 5.1: Example chromosomes/rulesets. All individuals have the same basic and game specific rules (gray) which are never changed. White: additional rules that represent single genes.

Since it is a very difficult question to answer, we do not try to define what *interestingness* regarding a video game means. Nevertheless, we do assume that a game level is more interesting if it requires some efforts from a player than the one that can be won immediately. Furthermore, we presume that a level that cannot be solved by a player at all, is perceived as less interesting. Thus, an interesting level should provide a certain degree of difficulty.

Following these assumptions and being inspired by the work by Nielsen et al. [25], we propose measuring the quality of the generated levels through their difficulty. The difficulty, in turn, can be measured using e.g. the game score provided by almost every game or by the fact whether or not the game could be won. Thus, a level with the desired degree of difficulty should be solved by a skilled player with a higher score than by a player that has a lower skill level.

For that purpose, we let two different agents play each level from a population multiple times. We use two controllers from the GVG-AI competition. The first controller is called *adrienctx*. It ranked first in the competition in the year 2014 and is therefore supposed to be more intelligent. The second controller is *sampleMCTS* which is provided with the GVG-AI framework and scored third in the same years competition. This controller is used as the worse-playing agent.

For each generated ruleset in a population, we measure the difference between the average scores achieved by each agent while playing the maps that were generated from this ruleset. Thereby, we assume that maps with higher score differences have a better difficulty meaning that the agent with a higher skill level could handle the game better than the worse one. Similarly, levels with lower differences are assumed to have a worse difficulty level meaning that either the level was too hard or too easy for both agents.

Since the most games observed in our studies are nondeterministic, we let each agent play each level *n* times with $n \in \{3, 10\}$ and compute then the average score achieved by each agent for each map. Afterwards, we compute the difference between these values. We add a score bonus of 1000 points when at least one of the agents wins the game to favor solvable levels. (In the GVG-AI competition, a player gets a score of -1000 if he is disqualified for some reason. We change this value to -10 to make sure that it does not overwrite the bonus points of 1000 in the fitness function.) Because we generate 5 levels from each ruleset of a population, we finally sum up the score differences of the 5 maps. That way, we get the fitness value of a single ruleset F_R as described in equation (5.1).

$$F_R = \sum_{1}^{5} \left(\frac{1}{n} \sum_{1}^{n} score_{adrienctx} - \frac{1}{n} \sum_{1}^{n} score_{sampleMCTS} + 1000 \right) \qquad (5.1)$$

$$n \in \{3, 10\}$$

Playing each map e.g. 3 times by two agents may take up to hours. For that reason, we limit the maximum number of game steps to 1000 (at the GVG-AI Competition each game lasts for max. 2000 steps). At this point, the evaluation of a map with the help of a human could be a faster method. Nevertheless, we aim to have a completely automatic generator using the two controllers.

5.3 Selection

With the help of the fitness evaluation function described in the previous section, we select the best individuals for the next population. Thereby, we follow a (μ+λ) elitism strategy with μ = λ = 10. That way, we keep the best 10 solutions chosen from the parents and the children ensuring that solutions with better combinations of additional rules are kept for further improvements.

Furthermore, we keep an archive with the best 10 individuals found during all generations. This allows us to determine the evolution of the rulesets. Additionally, is saves the best (but different) solutions found throughout the evolutionary process until we reach the termination criterion.

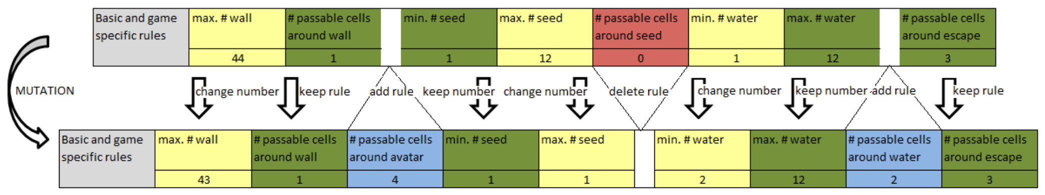

Basic and game specific rules
max. # wall
44
# passable cells around wall
1
min. # seed
1
max. # seed
12
# passable cells around seed
0
min. # water
1
max. # water
12
# passable cells around escape
3
MUTATION
change number
keep rule
add rule
keep number
change number
delete rule
change number
keep number
add rule
keep rule
Basic and game specific rules
max. # wall
43
# passable cells around wall
1
# passable cells around avatar
4
min. # seed
1
max. # seed
1
min. # water
2
max. # water
12
# passable cells around water
2
# passable cells around escape
3

Figure 5.2: Mutation of a chromosome (ruleset) using two operators. Gray: basic and game specific rules that are not mutated; green: unchanged genes (rules); yellow: changed numbers/genes; blue: added rules/genes; red: deleted rules/genes.

5.4 Mutation of additional ASP Rules

As already described in section 5.1, all rulesets of a population have the same basic and game specific rules and different additional rules. The genes of the individuals contain information about the randomly generated amount limits and the number of passable neighbor tiles of certain game objects. However, these numbers are randomly generated and do not guarantee the creation of levels with the desired difficulties.

To change the shape of the search space and to guide the solver into different directions, we propose using a mutation operator that randomly changes the additional rules. For that purpose, each additional rule described in section 4.5 has a 50% chance of being mutated. Thereby, each one of the amount limits is changed to a random value. As already described, this random value is kept between 1 and one fourth of the map size with the upper limit always being equal or higher than the lower limit.

Regarding the rules that set the number of passable neighbors of a game object, the mutator works according to the following rules: if there is no passable-neighbors rule defined for a certain type of game objects, it is added with a chance of 50%. Thereby, the number of passable neighbors is randomly set to a value between 0 and 4. In case, there is already such a rule defined for an object type, the mutator deletes this rule with a possibility of 50%. Figure 5.2 shows an example of the mutation process.

After applying these two kinds of operators to the 10 selected individuals, we have a new population of rulesets. Having this population, we continue with the evolution process described in the previous sections. That way, the generator moves in the search space and creates different combinations of rules. The generated rulesets provide maps with different properties and difficulty levels resulting in different fitness values.

Chapter 6

Experimentation

The experiments in this work contained two parts observing the levels build for different games of the GVG-AI framework. In both parts, the generator evolved the levels over 10 generations. The number of generations was held so low because of the very long time needed for the evaluation. Thereby, each population contained 10 individuals which were sets of ASP rules. Each ruleset provided 5 levels with a relatively small size of 10×8 tiles. These levels were played by two game-playing agents for maximum 1000 game steps each. The sum of the differences between the scores of these agents represented the fitness value of a ruleset. The best 10 individuals were selected into the next generation following a $\mu + \lambda$ elitism strategy. Furthermore, 10 best solutions were kept in an archive throughout the evolution process.

First, the generator evolved levels for the first game set which contained 10 different games. This process was performed twice. In the first run, each agent played each level 3 times. In the next run, we repeated this experiment letting each agent play each level 10 times. A comparison between these results should provide information about the impact of the number of runs per map on the evolution process of the levels.

The second step contained the evolution of levels for the second set of 10 games. Here, the experiments were run with the same parameters as in the first step and each agent playing each level 10 times. Since the generator was developed essentially based on the games from the first set, this step of the experiment should provide information on the generality of the generator.

In the following section, we describe general findings from the results of both experiment parts. For more detailed information about single games, the reader is asked to refer to the descriptions placed in the appendix B.

6.1 Results

Generating rulesets for level creation, our aim was to find sets that were not only solvable but also had *interesting* difficulty levels. According to the fitness function described in section 5.2, a ruleset got a higher fitness value if all of its maps were solvable. The digit in the thousands place represented, in most cases, the number of solvable levels and the digits behind it represented the sum of the differences between the agents scores. Thus, a ruleset having a fitness value above 5000 was expected to have created 5 levels all of which were solvable by at least one of the agents.

That way, the generator preferred rulesets that had more solvable levels to those with a higher score difference. For example, if the sum of differences of a ruleset was e.g. 55 and only 3 levels of this ruleset could be won, the ruleset got a fitness value of 3055. If now, another ruleset had a lower score difference of e.g. 15 but all 5 levels could be won, then the ruleset got a fitness value of 5015. Thus, despite having a lower score difference, the second ruleset was preferred by the generator.

However, this fitness function could be changed depending on the users preferences. If it is desired to have as many solvable levels as possible (as in our case), the fitness function can be used to filter out the desired solutions. If it is more important to have higher score differences and a ruleset that creates at least *one* solvable map, than the fitness function should be adjusted giving e.g. a single bonus of only 1000 points independently of the number of solved levels.

As the results of the experiments performed in this work have shown, the generator was able to find solvable *levels* for all games. For the most games, it also could find *rulesets* with fitness values above 5000. Furthermore, the final rulesets saved in the archive for almost all games had a minimum value above 5000 as shown in table 6.1. The only games for which the fitness value did not reach 5000 were *Dig Dug* and *Eggomania* having different reasons which are described later in this section.

In general, the fitness values varied from game to game depending on the scores that an agent could get in the game. For example, in the game *Camel Race* an agent could only get a score of 1 if he won the game and a score of -1 if he lost it. That way, the maximum possible difference between the two agents was 2. For that reason, the fitness values of rulesets from that game could not exceed 5002. Whereas in the game *Sea Quest*, a player could get 1000 score points for destroying a certain object. Destroying multiple of these objects, the difference between the agents could be far beyond 5000 as can be seen in table 6.1.

In this case, it was more difficult to interpret the results because we could not say for sure, whether the fitness of a ruleset was above 5000 because all 5 levels could be solved or because the better player got a higher score than the worse one. For that reason, we propose to adapt the fitness function and instead of adding 1000 bonus points when a level is solved, add a value that is suitable for the certain game taking into account the maximum game score.

Looking at the fitness curves of the different games, we could not only see that solvable levels were generated but also *how difficult* it was for the generator to find such levels and rulesets for certain games. For example, in the game *Missile Command*, the generator created multiple solvable maps already in the first generation. So, the minimum fitness value in the archive was above 5000 already after the evaluation of the second generation as it can be seen in the top row of figure 6.1.

In contrast to that example, the generator had difficulties to find solvable levels for the game *Boulderdash*. As we can see in the bottom row of figure 6.1, the minimum fitness value in the archive in the experiment with 10 runs per map reached 5000 only in the last generation (right column). With 3 runs per map, there were still rulesets in the archive which had a value below 5000 even after 10 generations (left column). Thus, in this case, not all of the final rulesets produced 5 solvable levels each.

These results can be traced back to the fact that in *Missile Command* the generator only had to find a good balance between the positive and the negative game objects, whereas in *Boulderdash*, it had to adjust a higher number of different game objects. Furthermore, *Boulderdash* has more complicated game mechanics including some physics which make it more difficult to find solvable levels.

Even though we can see such differences between the results of different games, it is not enough to look only at the fitness values. It is also important to have a closer look at the rulesets. For example, comparing the fitness values achieved in the games from the first GVG-set played 3 times by each agent with those where each agent played each level 10 times, we cannot recognize any meaningful differences. However, looking closer at the concrete parameters defined in the rulesets of the two experiments, we can see that in general, the generator created slightly more difficult levels in the experiment with 10 runs per map.

A good example therefor, is the game *Aliens* that was already mentioned several times in this work. In the original maps, there is usually one portal that spawns aliens which the player has to shoot. After examining the final rulesets in the archives of the two experiments, we noticed that in the part with 3 runs per map, 8 out of 10 rulesets had only one portal as expected. In contrast to that, in the

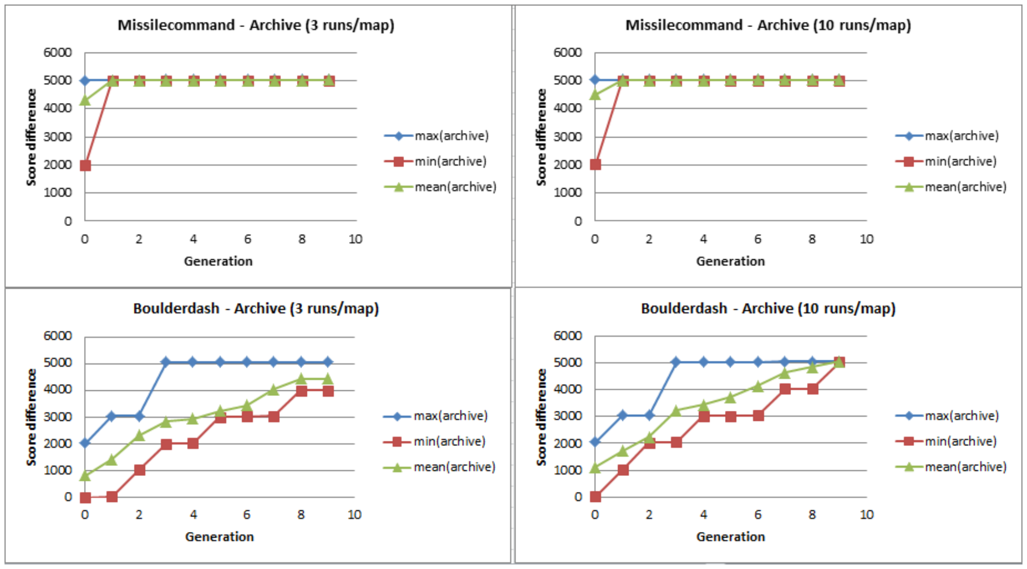


Figure 6.1: Comparison of the fitness curves of the games Missilecommand (top) and Boulderdash (bottom) for the experiments with 3 runs per map (left column) and 10 runs per map (right column).

experiment with 10 runs per map, 9 out of 10 rulesets had 2 portals. Thereby, sets with 1 portal had lower fitness values. Thus, with more runs, the score difference between the two controllers was higher so that the generator preferred more difficult levels with 2 portals. Similar relations could be found in the results of other games such as *Chase* where the generator created more enemies in the experiment with 10 runs than in the one with 3 runs.

Furthermore, we could see that for some games from the second game set where we have performed only experiments with 10 runs, the generator created levels that were more difficult than the original ones. For example, in the game *Eggomania*, there is usually 1 chicken that throws its eggs on the player who has to catch all eggs and kill the chicken. However, 9 out of 10 resulting rulesets had 3 or even more (up to 8) chickens in their levels. A higher number of chickens made the game more difficult and gave the playing agents the chance to increase the difference between their scores.

Speaking about the difficulty in these games, we outline the amounts of game objects that were set in the rulesets (described in section 4.5). Throughout all games from both game sets, we could notice that the generator was able to find acceptable amounts of objects. It performed especially well in those games, where the fitness value directly depended on the amounts of certain game objects. Those were the games where the player's aim was to avoid or destroy some *negative* objects at the same time collecting some *positive* objects. Hereby, the player got score points for both actions which made it possible for the more intelligent agent to gather more score points than the less intelligent one. That way, the evolutionary algorithm could easily filter out solutions with a desirable balance between the objects adjusting their numbers.

A good example for such a case is the game *Infection* where the player has to infect hosts. Therefor, he has to get the infection from a virus and collide with hosts avoiding the guardians that could heal him. In this case, the generator preferred rulesets with a low amount of viruses (1 to 3), so it was not too easy to win the game, and a medium amount of hosts and guardians having approximately 5 of each type in the levels. Rulesets with e.g. only 1 host or more than 3 viruses could be easily solved by both agents which could be clearly seen on their fitness values. Such rulesets had fitness values very close to 5000 which meant that the score difference was around 0 (in contrast to that, the maximum fitness value was 5069.4). Further examples for games with well-balanced levels were *Missile Command* and *Butterflies*.

Furthermore, the previous example of *Infection* has shown that the number of game objects that did not have a direct influence on the score could vary in the final sets. For example, walls were only obstacles in *Infection* but did not give any score, since they could not be destroyed. So, we could see that in the archive, there were rulesets having as well only 1 as up to 10 wall sprites in their levels. However, having more walls in a level meant that the agents had less space to move around which made the level more difficult. This could be the probable reason for the most final sets (6 out of 10) having only 3 or less walls.

Although, the generator performed generally well in balancing the amounts of game objects, there were still exceptions where it could not reach the perfect balance. For example, as already mentioned, in the game *Dig Dug*, none of the rulesets got a fitness value above 5000. In this game, the player has to either collect all goodies or kill all enemies who are spawned by spawn points. Taking into consideration that each spawn point spawns 5 enemies and the created levels had a relatively small size, it would be enough to have only 1 spawn point (and a few enemies) in the level.

However, analyzing the 50 rulesets created throughout the whole evolution process, we noticed that none of them had such a small number of spawn points. That means that many levels were too difficult to solve which is the probable reason why all rulesets had a fitness value below 5000. At this point, we assume that having a mutation chance of 50% and the random number being chosen between 1 and 12 (for the given map size), 10 generations were not enough to create a ruleset with the maximum number of spawn points set to 1. We further assume that more generations (or higher population size) could help the generator to find a better balance for this game.

Furthermore, some results have shown that it was wrong to assume that the amount of every object type should always be at least 1 and at most $1/4$ of the map size. For example, in the original maps of *Eggomania* there are no wall tiles inside the level except from the ones on the borders. If the egg thrown by the chicken hits a wall tile, the player loses the game. Usually, this happens only when an egg hits the bottom of the level.

However, since our generator created rulesets with the maximum amount of wall tiles set to at least 1 more than was needed for the borders, the player often could not prevent the egg from hitting the wall or the wall limited the players movement. Examples of such levels are shown in figure 6.2 where wall tiles are placed in such a way that the player is not able to catch the egg either because the a wall tile is placed between him and the chicken (left) or because he cannot move towards the egg (right). For this game, it would be more suitable to have *no* wall tiles inside the levels.

Examples for games where the amount of certain objects should exceed $1/4$ of the map size are *Boulderdash* and *Dig Dug*. In these games, the maps are normally completely covered with either dirt or walls which the player can destroy. These elements have important functionalities and only through them some game mechanics are made possible, e.g. the player can destroy a wall below a boulder and make the boulder fall onto an enemy. Though, because the generator could

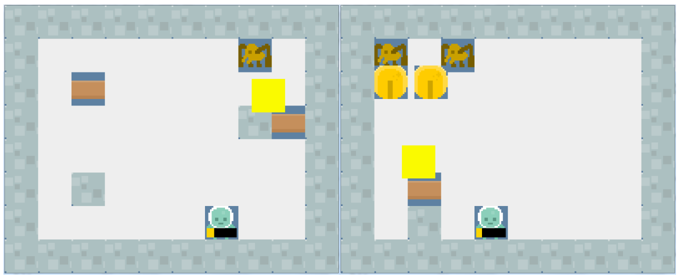


Figure 6.2: Examples of unsolvable levels for Eggomania. Left: the player cannot catch the yellow egg before it collides with the wall tile; right: the player is unable to move left to catch the egg.

not create a high number of these objects, the most levels had a very small amount of them. That way, the agents were forced to use other game mechanics so that the difficulty levels of these maps were not representative for these games. Moreover, the created levels were very different from the original ones.

Another problem that we noticed for multiple games was the wrong amount of exit doors in levels. There are many similar games such as *Boulderdash, Portals* and *Zelda* where the player has to reach a *goal* or an *exit*. The main idea in such games is that there is normally only one exit in the level and it is not easy to reach, e.g. it is hidden behind some walls or needs a key to be opened.

However, in most cases, a VGDL description for such a game does not contain the information that there should be only *one* exit. Usually, it only contains the termination condition which says that the game is over when there are no more exits in the level and that an exit disappears on collision with the avatar. Sometimes, the avatar gets some additional score points colliding with the exit.

For that reason, the generator interpreted the exits as positive game objects (collectables) that gave the agents more score points. Thus, having a higher amount of exits in a level meant that the score difference between the agents could be higher than in levels with less exits. So, the generator preferred levels with multiple exits.

That way, the main idea behind reaching a single hidden exit was not sustained in the generated levels and the agents aim changed to colliding with as many exits as possible to get a higher score. Hereby, the generator could find a suitable amount of exits to have an acceptable difficulty level but it could not produce levels similar to the original ones.

To avoid such problems in future, we propose taking into account not only the game score but also the time needed by the agents to solve the levels. That way, the generator could prefer maps which the agent with a higher skill level could solve faster than the weaker agent. We assume that thereby, in levels with multiple exits both agents would need approximately the same (long) time whereas in a level with only one exit the smarter agent could react faster than the worse one.

A general discovery that we could make looking at the generated levels, was that all of them have shown an equal distribution of game objects having good vertical and horizontal balance. Comparing the generated rulesets and their fitness values, we could not make any significant statement about the mutation operator that changed the number of passable neighbors for a game object. Some of the final rulesets of every game had rules specifying the number of passable neighbors and some did not. The fitness values seemed not affected by these rules. As a possible reason for that, we see the fact that, in general, there were not many objects in the levels that were impassable and could surround the objects.

Furthermore, the generator did not create any formations in the levels. For games like *Frogs, Portals* or *Pacman* that was a serious problem since their game mechanics rely on certain formations. In these games, lines of walls either build labyrinths in which the player has to move wisely or they separate doors from the rest of the environment making them unreachable. In *Frogs*, lines of water and moving trucks represent a river and a highway that the player has to cross carefully.

If these tiles are placed randomly in the level instead of the desired formations, some game mechanics become useless. If there is no labyrinth in *Pacman*, the player can move freely in level and can easily escape the ghosts and if there is no river between the player and the exit in *Frogs* the player can win the game very fast going directly to the exit. However, looking at the levels generated for those games, we have noticed that there were no such structures so the generator adjusted the difficulty of the maps by balancing the amounts of the game objects.

The reason for not creating any structures was that they were not defined in the ASP rulesets. A possible solution for that problem could be the definition of e.g. *blocks* or vertical and horizontal *lines* of objects of the same type and an optimization of the number of such blocks/lines in the levels. An alternative solution could be a constraint forbidding wall tiles (or other objects) to be placed alone in the level, thus each tile would have to have at least one neighbor of the same type.

Although not containing any formations was a problem for some games in our case, it is uncertain whether this is *always* a problem or whether it could be desired in some cases. Especially for new games, for which the user knows only the game mechanics but does not have any original maps, the created levels could provide help at polishing the mechanics.

Another point that did not work as desired in some games, was the placing of oriented objects at the borders of the map. As already mentioned, we created a rule that placed e.g. an object that was oriented left on the right border of the map. That way, this object should be able to move or shoot left using the longest distance. That worked out well in games like *Aliens* where the portals were placed on the top of the map or *Frogs* where the trucks were placed on the left/right side and moved towards the opposite wall.

However, it did not work, for example, in the game *Camel Race*. Here, the VGDL description provided definitions of camels that were moving only left (or only right). That way, the generator did not place all camels on one side of the map. Instead they were placed on the opposite borders of the map depending on theirs orientations having different distances to the goals. To avoid that problem, the orientation rule could be e.g. added to the additional rules instead of the game specific rules.

Even though the generator had problems creating levels for some very specific games, it generated adequate maps for the most games. However, independently from how well it performed in the generation process, it has shown a big problem regarding its time consumption. Having chosen a small amount of only 10 generations for our experiments, we assumed that each evolution process would take up to several hours. Though, we did not expect that evaluating some games would take as long as it did.

The second column of table 6.1 shows the time needed for each game to perform the evolution process. As we can see, even for the experiments where each agent played each map only 3 times, the generator needed from 3 hours 54 minutes (*Missile Command*) up to 17 hours 25 minutes (*Portals*). As for the experiments with 10 runs per map, the time ranged from 1 hour 15 minutes (*Camel Race*) to 79 hours 31 minutes (*Sokoban*).

These periods of time contained as well the time needed by the ASP solver to find the solutions, as the time needed by the agents to play all levels. In general, the time taken by the ASP solver did not exceed 5 minutes. However, if the ruleset contained rules that made it unsolvable, e.g. it had inconsistent amounts of objects, the solver tried to find a solution for a certain period of time before a new ruleset was created. That way, creating a solvable ruleset could take more than 5 minutes.

Nevertheless, the most time-consuming process was the evaluation by the agents. Depending on the positions of certain objects in the map, the game mechanics and the skill level of an agent, it could solve a level immediately (or lose) or consume all 1000 game steps. That way, it was possible to have such a wide range of time spans for different games.

Game	Evolution Time	Min. Fitness 1st gen.	Max. fitness 1st gen.	Min. Fitness 10th gen.	Max. fitness 10th gen.	Min. fitness archive 10th gen.	Max. fitness archive 10th gen.
Aliens (3)	5hrs 52mins	-262.3	5015	930	5014	5014	5109
Boulderdash(3)	8hrs 30mins	9.7	2036.7	13.3	4038.7	4009.7	5053.3
Butterflies (3)	1hr 32mins	-14.7	5006.7	3976.7	4996	5004	5049.3
Chase (3)	12hrs 31mins	4	5005.3	2009.3	5008.7	5007	5012
Frogs (3)	6hrs 8mins	10.7	5002.7	3002	5006.7	5005.3	5017.7
Missile Command (3)	3hrs 54mins	2002.3	5009.3	1.3	5018.7	5009.3	5018.3
Portals (3)	17hrs 25mins	0.7	4023.7	7.3	5024.3	5017.7	5031.3
Sokoban (3)	14hrs 45mins	0.7	4010	1008	5003	5000	5003
Survive Zombies (3)	14hrs 16mins	-2	4233.7	36	5243.7	5187	5393.3
Zelda (3)	7hrs 25mins	3015.7	5092	3034.3	5084	5095	5133.3
Aliens (10)	16hrs 10mins	843	5010.5	4918.6	5049	5020.8	5069.6
Boulderdash (10)	27hrs 59mins	16.4	2035.3	1016.8	5040.2	5014	5048.9
Butterflies(10)	10hrs 21mins	3962.8	5000.8	4982.4	5000.4	5001.6	5012.6
Chase (10)	40hrs 20mins	4.4	5004.8	5001.3	5010.1	5007.9	5011.3
Frogs (10)	12hrs 15mins	1998.3	5033.7	3103.3	5012.7	5022	5037.7
Missile Command (10)	5hrs 9mins	2015.5	5016.8	1996.5	5010.4	5013.5	5021.1
Portals (10)	42hrs 53mins	6.8	4023	19	5022.4	5015.4	5034.7
Sokoban (10)	79hrs 31mins	5.3	5001.6	3001.6	5003.8	5001.6	5007.3
Survive Zombies (10)	27hrs 9mins	4.6	5245.5	3	5419.5	5276.7	5424.1
Zelda (10)	23hrs 26mins	4035.6	5091.8	4032.5	5106.1	5086.2	5106.1
CamelRace (10)	1hr 15mins	4999	5000.6	2999.4	5000.6	5000.6	5001.2
Digdug (10)	52hrs 17mins	-66.9	3899.9	901	4951.8	3930.9	4964.2
Firestorms(10)	19hrs 47mins	1.1	5058.8	996.3	5068.1	5057.1	5072.9
Infection (10)	5hrs 54mins	5001.1	5025.2	5002	5004.1	5046	5069.4
Firecaster (10)	50hrs 49mins	4010.5	5034.6	3018.8	5027.8	5030.7	5045.2
Overload (10)	67hrs 49mins	18.4	4017.8	11.6	5028.1	5022.2	5085.5
Pacman (10)	11hrs 30mins	3053.3	5052.2	4110.4	5107.2	5090.6	5161.4
Seaquest (10)	27hrs 9mins	-22522	24857.4	-18806	32417	24453	35549.3
Whackamole (10)	26hrs 20mins	994.4	5055.5	19997.6	5072.7	5055	5077.1
Egoomania(10)	12hrs 42mins	1872.5	45524.9	1720.7	4726.5	4370	4787

Table 6.1: Results of the level generation for the first 20 games of the GVG-AI framework. First 10 rows: games of the 1st game set with the agents playing each level 3 times; next 10 rows: the same games with 10 runs per map; last 10 rows: experiments on the 10 games from the 2nd set with 10 runs per map. The second column shows the time needed for the experiment. Further columns show the minimum and maximum fitness in the: 1st generation, last generation, final state of the archive.

Chapter 7

Conclusions and Future Work

This chapter summarizes the thesis and its outcomes and outlines possible directions for future work.

7.1 Conclusions

This thesis introduced a procedural level generator that has the aim to create maps for any game described in Video Game Description Language (VGDL). Reading the descriptions of the sprites and collision effects provided for a game, the generator created corresponding rules for Answer Set Programming (ASP). To complete missing information, random rules were added specifying amounts of game objects and defining their neighborhoods. The generated rulesets were then put into an ASP solver which found corresponding game levels through deductive reasoning.

Afterwards, these rulesets were evolved using a simple Evolutionary Algorithm in order to obtain not only solvable but also interesting levels. Therefor, the generator created for each game populations of 10 rulesets with 5 maps each, evolved them over 10 generations and evaluated their fitness with the help of two game playing agents. Hereby, we took the difference between the game scores of the two agents as a measure of level quality assuming that a well-designed level would differentiate between the weaker and the smarter agent. This assumption was based on a hypothesis that can be found in [25, 26].

In the experiments, the generator created levels for the first 2 game sets provided by the GVG-AI framework. Each set had 10 different single player games. For the first set, we performed two steps, first, letting each agent play each map

3 times and second, letting them have 10 runs per map taking the average of the scores achieved in these runs. For the second set, we only performed the experiment with 10 runs per map.

Throughout the whole process of evolution of every game, 10 best solutions were saved in an archive. That way, it was possible to access the best levels at the end of the process and analyze not only the resulted fitness values but also the levels. Even though we did not have many generations due to long evaluation times, the results have shown that the EA still showed good performance. It performed especially well in balancing the amounts of positive and negative game objects finding maps with desired difficulty levels for the most games. Although in some cases, more generations could provide a better fine-tuning of the numbers.

It is interesting that for some games, the generator created levels that were more difficult than the original ones. Thus, levels with the increased difficulty were more manageable for the smarter agent than for the weaker one allowing a higher score difference between them. Also, we noticed that in the experiments with the agents playing each map 10 times, the levels were slightly more difficult than in the experiments performed with 3 runs per map.

Although the generator performed well in adjusting the numbers of game objects, it did not create any formations of game objects in the levels. For the most games, that was not a problem because formations were not important in these games. Considering the small size of the generated levels and the good vertical and horizontal balance of objects, which sustained in all cases, the levels of these games had a structure similar to many of the original maps. However, missing formations were a problem for games that rely on certain structures such as labyrinths or passable streets. Without these structures, some game mechanics became useless, so that the agents had to solve the game in a different way. That way, the generated maps and their fitness values misrepresented the desired solutions.

Although the results of the process described in this thesis were satisfactory, the method proved to take very long time for the evolution. Especially, the evaluation performed with the help of the controllers was very time-consuming. With only 3 runs per map, the process lasted for at least 3 hours per game. The maximum time needed for an experiment with 10 runs, exceeded 79 hours. Even with fewer individuals or a smaller amount of generations, evaluating the maps by letting game agents play them, would not allow using the generator for level creation at runtime. For that reason, we do not recommend using this method if a fast generation of maps is desired.

7.2 Future Work

Although the method proposed in this thesis cannot generate new levels very fast, it is still interesting for many research areas and provides multiple directions for future work. To analyze some minor problems described here, the initial piece of future work would consist of performing experiments with more generations and higher population sizes. Moreover, further experiments could be performed on all 60 games of the GVG-AI framework.

The levels created for this thesis, had a relatively small size. By increasing the size of the levels, also the size of the search space would grow. We assume that this could lead to much longer computation times of the ASP solver. That is why the performance of the ASP solver should be tested on levels with a larger size.

For evaluation of the generated maps, this work concentrated on the differences of scores between the game playing controllers *adrienctx* and *sampleMCTS*. However, there were multiple controllers in the competition of the year 2015 that performed much better than both of these two agents. For this reason, further research could be performed using newer agents with higher skill levels. Moreover, the fitness function could be changed comparing more than only 2 controllers.

Additionally, we recommend taking into consideration not only the game score but also the time needed by each agent to solve a level. That way, other techniques such as Multi-Objective EA could be used to optimize multiple fitness functions.

In this work, we have already implemented some rules optimizing the vertical and horizontal balance of the objects trying to improve the visual impression of the maps. However, the levels did not have any formations. For that reason, we propose adding further ASP rules to the rulesets optimizing the relation between the number of objects and the number of distinct segments of these objects as described in [24].

Furthermore, to evaluate the visual impression of the maps, it is not enough to use game controllers or any fitness function. For that purpose, it would be more interesting to include a human into the loop. Therefor, we propose performing user studies searching not only for solvable levels with the desired difficulty but also for maps with a certain aesthetic value.

Another possibility is to transform the methodology from a purely automatic generator to a generator with mixed authorship. Thereby, the generator could create ASP rules and a human game designer could then include or exclude certain rules before the generator would perform the evaluation step. That way, the generator could be used as a tool creating not only levels but any other game assets.

Appendix A

Auxiliary Methods

Java Method with example parameters	Description	Rule example
Create Two Opposite Neighbors Free (”box”)	Makes sure that two opposite neighboring cells of the object are not impassable.	:- sprite((X,Y), box), impassable((X-1, Y)), not impassable((X+1, Y)).
Create N Passable Cells Around Object (”portal”, 3)	Creates a given number of cells that are not impassable around each obj of given type	0{impassable(T2) : adjacent(T1,T2)}1 :- sprite(T1, portal).
Create Counter Termination Rule (”exit”, 0)	Makes sure that a *SpriteCounter* termination condition is not met.	counter(exit, N) :- N=#count{sprite((X,Y), exit)}. :- counter(exit, 0).
Create Counter Termination Rule For Subtypes(”portal”, 0, {”portalSLow”, ”portalFast”})	Makes sure that a *MultiSprite-Counter* termination condition is not met.	counter(portal, N) :- M = #count{sprite((X,Y), portalSlow}, K = #count{sprite((X,Y), portalFast)}, N = K+M. :- counter(portal, 0).
Create One If Other Exists (”exit”, ”entry”)	If there is a sprite with the second object, then also sprites with the first object are created.	:- 1{sprite(T1, entry):tile(T1)}, {sprite(T2, exit):tile(T2)}0.
Create None If No Other Exists (”exit”, ”entry”)	If there is no sprite with the second object, then no sprite with first object is created.	:- {sprite(T1, entry2):tile(T1)}0, 1{sprite(T2, exit2):tile(T2)}.
Create As Many Others As Ones (”entry”, ”exit”)	If there are n objects of the first type, creates n objects of the second type.	X{sprite(T, exit1):tile(T)}X :- setNumber(exit,X). getNumber(entry1, N) :- N = #count{sprite((X,Y), entry1):tile((X,Y))} . setNumber(exit, N) :- getNumber(entry, N).
Create One Object Rule (”avatar”)	Creates a rule saying that there is exactly one object of the given type.	:- not 1{ sprite(T, avatar): tile(T)}1.
Create Orientation Rule (”avatar”, ”UP”)	Puts the given object on the boundaries of the map (if direction is ”UP”, object is on the bottom etc.)	:- sprite((X,Y),avatar), not maxdimY(Y+1), tile((X,Y)).
Minimize Distance (”avatar”, ”diamond”)	Minimizes the distance between the first and the second object type	dis((X1,Y1), S1, (X2,Y2), S2 ,N) :- sprite((X1,Y1),S1), sprite((X2,Y2),S2), N=#abs(X1-X2)+#abs(Y1-Y2). #minimize[dis(C1, S1, C2, S2, N) : S1 = diamond: S2 = avatar = N @ 5].

Java Method with example parameters	Description	Rule example
Maximize Distance (”avatar”, ”exit”)	Maximizes the distance between the first and the second object type	#minimize[dis(C1, S1, C2, S2, N) : S1 = exit: S2 = avatar = ((width-2)+(height-2))-N @ 5].
Reachable (”avatar”, ”exit”, false)	Creates a rule saying whether there is a path between the first and the second object.	:- not reach(C1,C2), sprite(C1,avatar), sprite(C2,exit).
Maximize Vertical Balance (”wall”)	Minimizes the difference between the amount of objects on the left half and on the right half of the level.	vSymmetry(wall, N) :- L = #count{sprite((X,Y), wall) : X<width/2}, R = #count{sprite((X,Y), wall) : X>=width/2}, N=#abs(L-R). #minimize [vSymmetry(wall, N) =N@5].
Maximize Horizontal Balance (”wall”)	Minimizes the difference between the amount of objects on the top half and on the bottom half.	hSymmetry(wall, N) :- T = #count{sprite((X,Y), avatar) : Y<=height/2}, B = #count{sprite((X,Y), wall) : Y>height/2}, N=#abs(T-B). #minimize [hSymmetry(wall, N) =N@5].
Create Min Dist (”avatar”, ”crab”, 5)	Distance between the first and the second object is minimum X.	:- dist((X1,Y1), avatar, (X2,Y2), crab, N), N<5.
Minimize Difference Between Types ({”crab”,”butterfly”, ”boulder”}, {”diamond”,”coin”})	Minimizes the difference between the amounts of the objects from the first and the second list.	dif(pos, neg , M) :- N=#count{sprite((X,Y), crab;butterfly;boulder)}, P=#count{sprite((X,Y), diamond; coin)}, M = #abs(N-P). #minimize [dif(pos,neg,M) =M@5].
Create Min Number (”diamond”, 3)	Sets the minimum number of given objects.	3 {sprite(T,diamond):tile(T)}.
Create Max Number (”diamond”, 6)	Sets the maximum number of given objects	{sprite(T,diamond):tile(T)} 5.
CreateMinNumberOf Alternatives ({”portalFast”, ”portal-Slow”}, 2)	Sets the minimum number of one or more of the given sprite types	2 {sprite(T, portalFast; portal-Slow):tile(T)}.
CreateMaxNumberOf Alternatives ({”portalFast”, ”portal-Slow”}, 7)	Sets the maximum number of one or more of given sprite types.	{sprite(T, portalFast; portal-Slow):tile(T)} 7.

Table A.1: Auxiliary Methods implemented in the generator with their descriptions and exmples of resulting ASP rules.

Appendix B

Detailed Results of all Games

B.1 Results in the first Game Set

B.1.1 Aliens

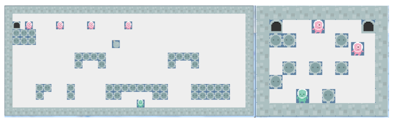


Figure B.1: Aliens levels. Left: level provided by the GVG-AI framework; right: generated level.

In the game *Aliens*, the player is attacked by *aliens* who are spawned by a *portal*. Usually, there is only *one* portal which is placed in one of the upper corners of the level and spawns a limited number of aliens. Those are moving horizontally going one row down when they reach the left/right border of the level. They throw bombs down on the player who is placed on the lower border of the level and is also moving left and right. The player is able to shoot back at the aliens getting 2 score points for each alien. The game is over when either the player or all aliens are killed. Sometimes, there are some *base* tiles placed between the player and the portal as shown in figure B.1 on the left side. Those tiles can be destroyed by the player giving him 1 score point each.

The results of both experiments for *Aliens* have shown that it is possible to create rulesets that provide solvable levels. As can be seen in the upper diagrams of figure B.2, there were rulesets with fitness values above 5000 in all generations.

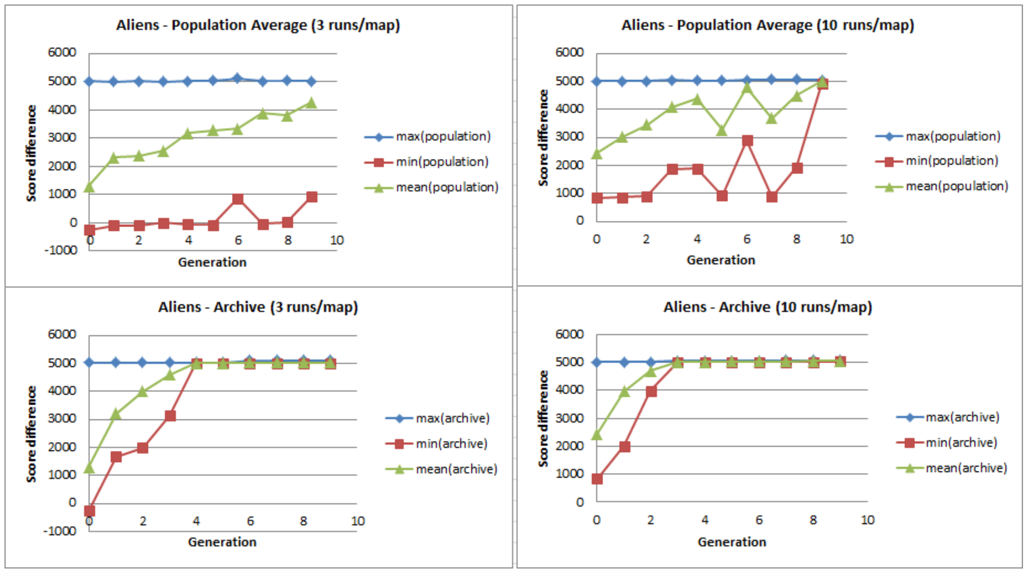


Figure B.2: Score differences for the game Aliens. On the left column, the results of the experiment with the agents playing each level 3 times is shown. The right column shows the results of the experiment with 10 runs per map. The upper row shows the minimum, average and maximum fitness in the population per generation. The lower row shows the corresponding values of the individuals present in the archive at each generation.

This value means that all 5 levels provided by the corresponding ruleset could be solved by at least one of the agents.

The maximum fitness value achieved in the experiment with the agents playing 3 times each level was 5109 meaning that all levels were solved and the mean difference between the agents scores was 109. The experiment with 10 runs per map delivered a maximum difference of 69 with a fitness value of 5069 showing only a slight difference to the first results.

Since the aliens were the only source of danger in this game, the amount of the portals spawning the aliens was the main criteria of the level difficulty. In the first experiment, 8 of the 10 final rulesets had only 1 portal in the levels which corresponds to the original levels. However, letting the agents play each level 10 times has shown that having only 1 portal in the levels provided lower score differences. This means that both agents could solve the levels almost equally well with the levels being too easy for both of them. Hereby, all of the final sets had 2 portals as can be seen on the right side of figure B.1, spawning more aliens and providing a better/higher difficulty level for the agents. This fact can be used to either increase the number of portals or their spawning rate in further levels, generating more aliens and changing the difficulty that way.

Another noteworthy issue is that 9 of the 10 sets had a maximum amount of base tiles being set to 9. Since the portals and the bases were the only objects in the game, all those 9 rulesets provided the same 5 levels even though they had different numbers for the minimum amounts of the base tiles. This fact can be led back to the balancing of the object distribution in the levels showing that these 5 levels have the optimal distribution of the portals and bases in the levels found so far.

B.1.2 Boulderdash

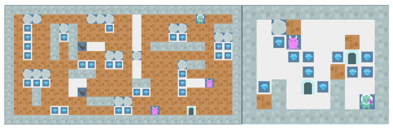


Figure B.3: Boulderdash levels. Left: level provided by the GVG-AI framework; right: generated level.

Boulderdash is a game were the player has to collect a given number of *diamonds* (getting 2 score points each) to be able to go through the *exit* and win the game. Usually, there is only one exit and many diamonds in the level. The left part of figure B.3 shows that the most space of the level is covered with *dirt* which can be destroyed by the player. Furthermore, there are a few *enemies* who can be killed by the player. Therefore he should let a *boulder* fall on the enemy destroying the dirt sprites under the boulder.

Although, the results of both experiments demonstrated in figure B.4 have shown that the generated levels can be solved by the agents, no level had a structure similar to the one from the original levels (see the right part of figure B.3). In both experiment, all final sets had barely any dirt in the levels. Through our upper limit of the amount of an object set to one fourth of the map size it could not be possible to cover the levels completely with dirt. However, the maximum amount was not even set to one fourth but to lower numbers. That can be explained by the fact that the dirt sprites did not have any direct influence on the player score and sets with less dirt and more other objects (e.g. diamonds) provided higher fitness values.

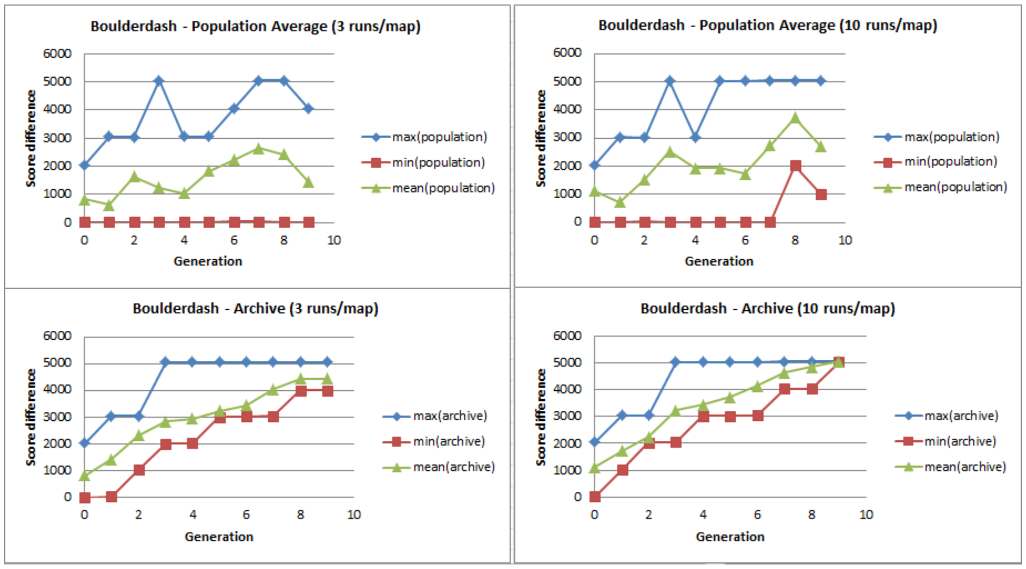


Figure B.4: Score differences for the game Boulderdash. On the left column, the results of the experiment with the agents playing each level 3 times is shown. The right column shows the results of the experiment with 10 runs per map. The upper row shows the minimum, average and maximum fitness in the population per generation. The lower row shows the corresponding values of the individuals present in the archive at each generation.

Since the player was forced to collect at least 9 diamonds to win the game, all final sets had that number of diamonds. However, the most rulesets in both experiments had 2 or more exits because there was no information in the VGDL-description of the game that only 1 exit was required.

The balance between the enemies and boulders which could kill them was limited by the fact that all boulders were placed in the uppermost row of the levels. This happened due to the fact that the boulders could only fall/move *down* and the corresponding rule described in 4.4 placed them on the top of the map. Therefore, only a limited number of boulders could be placed in the level (max. 8). Furthermore, there was almost no dirt which would keep the boulders in their place, so that they fell immediately at the game start and could not be used by the player to kill the enemies. For that reason, all final sets from the first experiment had only a single enemy allowing the agents to win the game by collecting all diamonds and avoiding the enemy. The second experiment delivered sets with slightly higher numbers of enemies providing a little higher difficulty level for the agents.

B.1.3 Butterflies

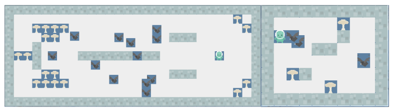


Figure B.5: Butterflies levels. Left: level provided by the GVG-AI framework; right: generated level.

The game *Butterflies* has only 3 kinds of game objects: *walls, butterflies* and *cocoons*. Usually, there is approximately the same amount of both kinds of objects. If a butterfly collides with a cocoon, the first one is cloned and the second one disappears. The players aim is to kill all butterflies by colliding with them (getting 2 score points for each) without losing all cocoons. Structures of wall tiles may represent obstacles for all moving entities.

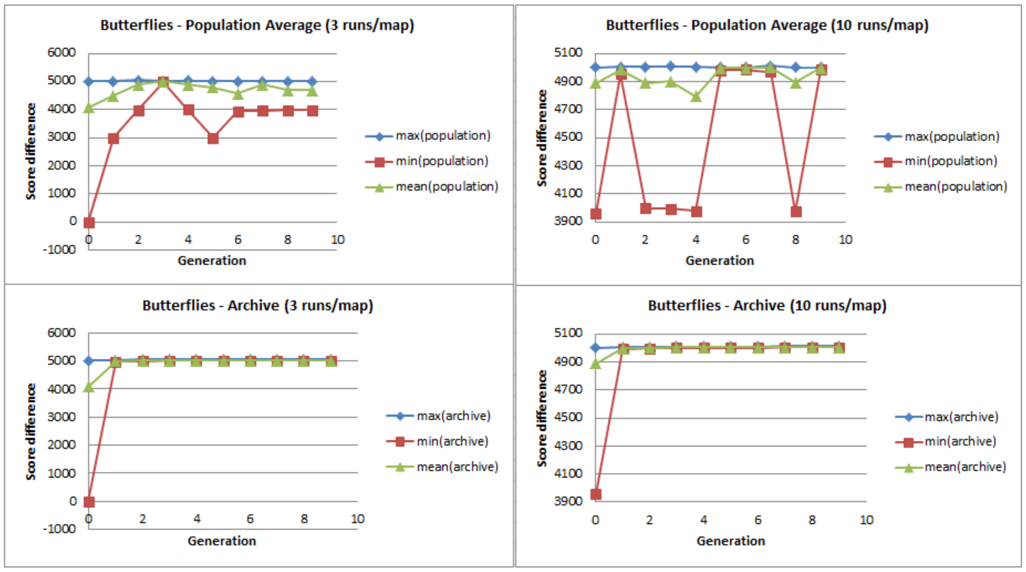


Figure B.6: Score differences for the game Butterflies. On the left column, the results of the experiment with the agents playing each level 3 times is shown. The right column shows the results of the experiment with 10 runs per map. The upper row shows the minimum, average and maximum fitness in the population per generation. The lower row shows the corresponding values of the individuals present in the archive at each generation.

In this case, the results of both experiments that can be seen in figure B.6 also showed that solvable levels can be created by the generator. However, the score differences between the two controllers in the second experiment were very low with a maximum difference of $12,6$ ($49,3$ in the first experiment). This shows that the game could be solved by the two controllers almost equally well. As can be seen on the right side of figure B.5, the generated levels provided no structures of walls as it is the case in the original level on the left side of the figure.

Thereby, the balance between the butterflies and the cocoons was achieved by the most final rulesets in both cases. Only 4 individuals in the first experiment and 2 in the second one had less butterflies than cocoons. Individuals that were too easy for both controllers had fitness values around 5000 meaning that all maps were solved but showing no difference between the agents. In some cases, *sampleMCTS* controller scored even better than *adrienctx* as well in levels that were too easy as in those that were too hard (e.g. had significantly more butterflies than cocoons).

B.1.4 Chase

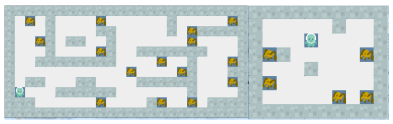


Figure B.7: Chase levels. Left: level provided by the GVG-AI framework; right: generated level.

The only moving entities in the game *Chase* are the player and *scared goats*. The player has to chase the goats and get 1 score point colliding with each of them and making the goats *angry*. Avoiding the angry goats the player can win the game when there are no more scared goats left. Structures of walls can be obstacles and cover for all entities as shown on the left side of figure B.7.

In both experiments, the generator could easily create solvable levels with rulesets having fitness values above 5000 as shown in figure B.8. The number of walls in the levels was similar in both experiments having maximum 3 wall tiles. However, it is noteworthy that the experiment with 3 runs per map provided rulesets with an upper limit of goats set to 5 - 6 and the one with 10 runs per map also had levels with 7 enemies. This shows that similarly to the game *Aliens* described in section B.1.1, the rulesets of the second experiment had a slightly higher difficulty level than those from the first experiment.

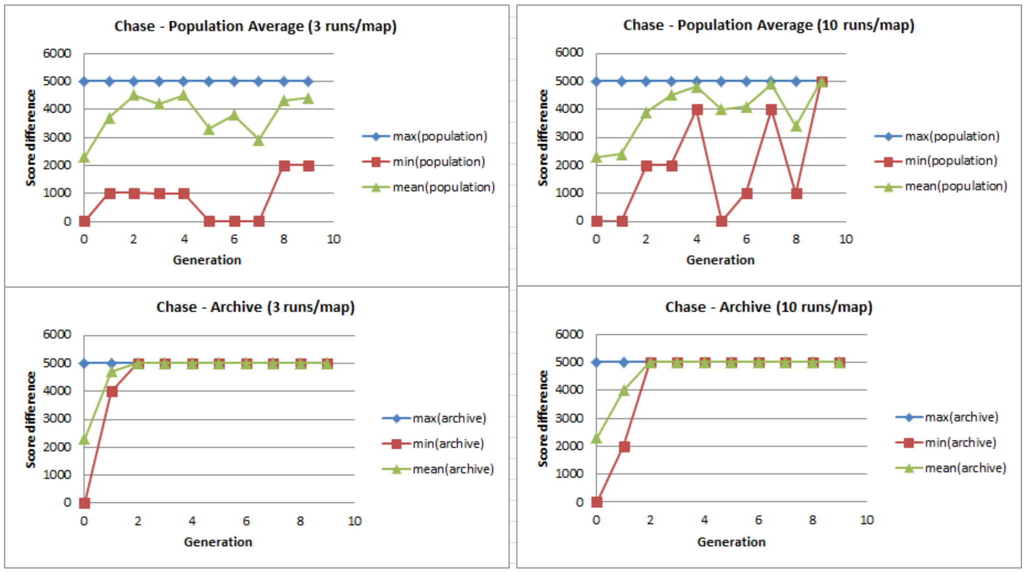


Figure B.8: Score differences for the game Chase. On the left column, the results of the experiment with the agents playing each level 3 times is shown. The right column shows the results of the experiment with 10 runs per map. The upper row shows the minimum, average and maximum fitness in the population per generation. The lower row shows the corresponding values of the individuals present in the archive at each generation.

Furthermore, the results have shown that the levels with less than 5 goats were too easy for both controllers having a fitness value around 5000. Thereby, levels with more than 7 goats had very low fitness values of around 5 being too difficult for both agents. This example shows very well how the generator is able to balance the number of game objects in a simple game. However, it is not able to generate levels with structures, as can be seen on the right side of figure B.7.

B.1.5 Frogs

In the game *Frogs*, the player has to reach a *goal* crossing a *street* and a *river*. Thereby, he has to avoid *trucks* driving on the street and for crossing the river he must jump on *logs* floating in the water. As we can see on the left part of figure B.9, the river and the street are clearly defined as horizontal structures with the frog and the exit being on the different sides of them. The player gets 1 score point when he reaches the goal.

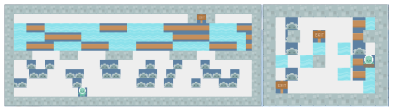


Figure B.9: Frogs levels. Left: level provided by the GVG-AI framework; right: generated level.

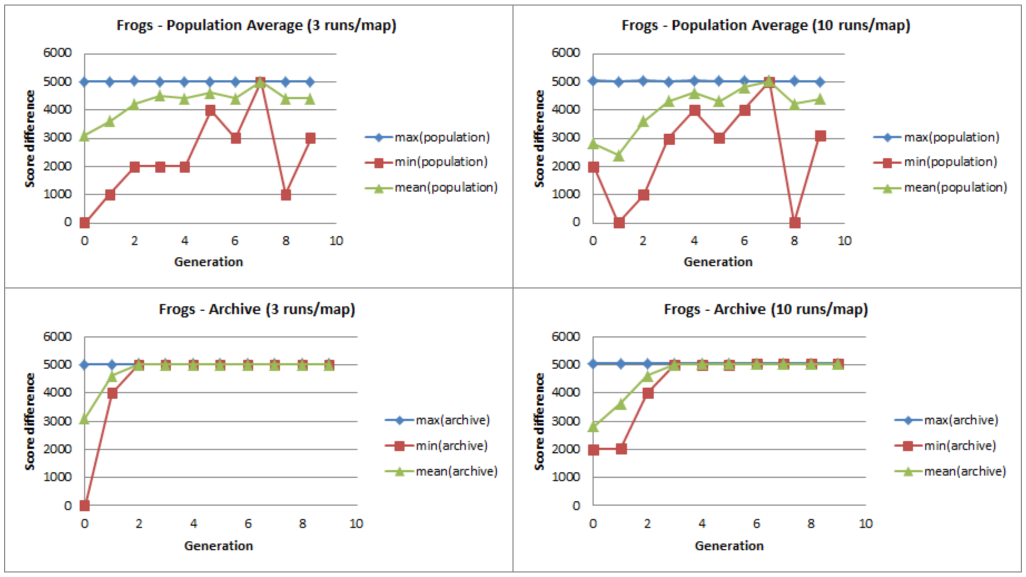


Figure B.10: Score differences for the game Frogs. On the left column, the results of the experiment with the agents playing each level 3 times is shown. The right column shows the results of the experiment with 10 runs per map. The upper row shows the minimum, average and maximum fitness in the population per generation. The lower row shows the corresponding values of the individuals present in the archive at each generation.

As figure B.10 shows, the generator could find solvable levels for this game. However, the score differences were not high. The only type of game objects that brought the agents score points were the goals and because of the way how the goals and water were defined in the VGDL description, the generator handled them both in one rule. So, the minimum and maximum amount was set to the sum of water and goal tiles. That way, these numbers did not have direct influence on the level structure and the fitness value. In most cases, there were multiple goals in a level.

Looking at the structure of the generated levels, we can see on the right part of figure B.9 that the trucks were placed on the borders of the map so that they could move horizontally across the map. That way, the levels represented streets. However, the water tiles were placed randomly in the maps, so that no rivers were created.

B.1.6 Missile Command

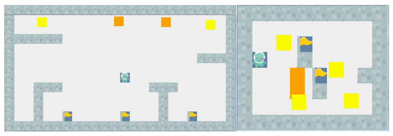


Figure B.11: Missilecommand levels. Left: level provided by the GVG-AI framework; right: generated level.

In *Missile Command*, the player has to defend a few *cities* from *incoming* enemies. Each time that an enemy reaches a city, both disappear and the players score decreases by 1 point. The player can shoot at the enemies getting 2 score points for each killed enemy. He wins if he kills all enemies and loses if all cities are destroyed. Usually, there are a few more enemies than cities and the two types of objects are placed on the opposite sides of the level as it is shown on the left side of figure B.11. The player is placed between the cities and the enemies. There are no walls inside the level.

The results of both experiments of this game have shown that it was possible to find solvable levels (see figure B.12). Furthermore, all final sets had $2-4$ enemies more than cities having a similar balance to the original levels. Rulesets with less enemies than cities or a small amount of both object types were too easy for the agents and had fitness values around 5000. Thereby, levels that had significantly more enemies than cities were too difficult for both agents and got fitness values much lower than 5000.

Although the generator found a good balance of game objects, it could not recreate the level structure from the original levels. Even when the enemies and the cities were spatially separated in the original maps, it was not described in the VGDL description of the game. That way, all objects were placed randomly in the levels as the right part of figure B.11 shows. However, this fact is not necessarily a negative one, because levels where the enemies are closer to the cities may provide a higher difficulty degree.

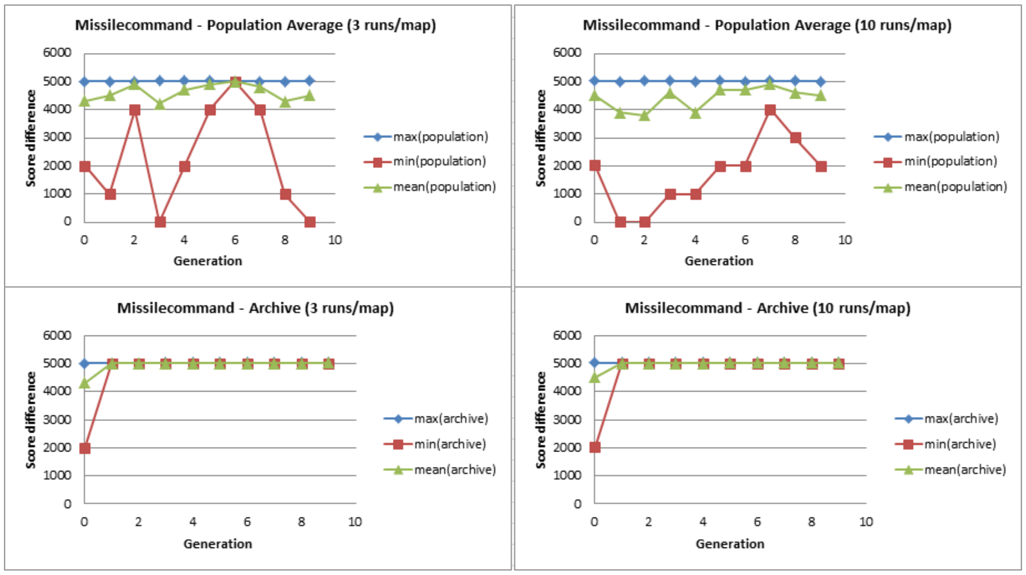


Figure B.12: Score differences for the game Missile Command. On the left column, the results of the experiment with the agents playing each level 3 times is shown. The right column shows the results of the experiment with 10 runs per map. The upper row shows the minimum, average and maximum fitness in the population per generation. The lower row shows the corresponding values of the individuals present in the archive at each generation.

B.1.7 Portals

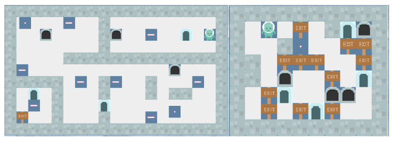


Figure B.13: Portals levels. Left: level provided by the GVG-AI framework; right: generated level.

The game *Portals* belongs to those games where the player has to reach the *goal* which is present only once in a level. It is separated by walls and can be reached by the player going through portals *entries* and coming out from portal *exits*. There is always one special exit placed close to the goal. However, the player does not know which entry leads to which exit. Finding out the connection between the portals the player has to avoid enemies. Reaching the goal, the player gets 1 score point.

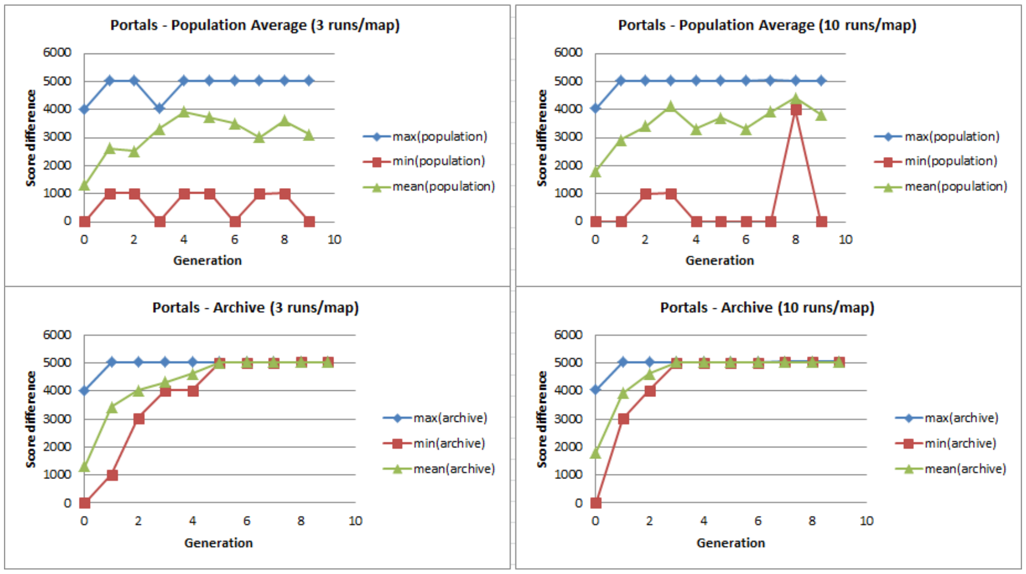


Figure B.14: Score differences for the game Portals. On the left column, the results of the experiment with the agents playing each level 3 times is shown. The right column shows the results of the experiment with 10 runs per map. The upper row shows the minimum, average and maximum fitness in the population per generation. The lower row shows the corresponding values of the individuals present in the archive at each generation.

Since the goal was the only entity that could bring the player some score points in this game, all of the final rulesets produced by the generator had multiple goals, as can be seen on the right side of figure B.13. The upper limit of the goals was not set to 1 in the game specific rules because this information could not be derived from the VGDL description of the game similarly to the game *Boulderdash* described in section B.1.2.

The results of this game, shown in figure B.14, proved once more that the generator is able to find maps with a good difficulty level by changing the number of enemies. Levels which were relatively empty and were too easy had fitness values around 5000. Those, with a high number of enemies got very low fitness values.

However, this game has shown once again that the generator was not able to produce levels with structures. The goals were not separated from the player as in the original levels. Furthermore, there were barely any walls at all so that the player could reach all portals and goals by moving through the level. Here, a rule saying that the goal should not be reachable by the player would be desired. In future, this could be achieved by adding such a rule (among others) randomly to the ruleset as an additional mutation operator.

B.1.8 Sokoban

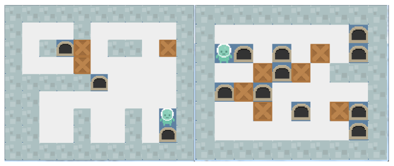


Figure B.15: Sokoban levels. Left: level provided by the GVG-AI framework; right: generated level.

In the game *Sokoban*, a level contains a few *holes* and an equal or slightly higher number of *boxes*. The players aim is to put all boxes into holes. Every time that the player pushes a box into a hole, he gets 1 score point and the box disappears. As the left part of figure B.15 shows, the original levels have a quite small size leaving only a little space for the player to move.

The results of both experiments, have shown that the generator was able to create rulesets with 5 solvable levels (see figure B.16). However, the highest score difference achieved in the experiment with 3 runs per map was only 3 (5003) and in the experiment with 10 runs 7.3 (5007.3). Since the scores in this games depended directly on the number of boxes that the agents could push into holes, the results show that the levels in the second experiment had a slightly higher difficulty level.

In contrast to the original levels, all final sets of both experiments had fewer boxes than holes. However, the levels in the second experiment had a higher amount of both objects types than the levels in the first experiment. This shows again that the maps of the second experiment were more difficult since the agents had less space to move and needed more time to solve levels that were fuller.

Although the agents were able to solve many of the created levels, some maps could not be won by the controllers although they could be solved by a human player. These levels had a more similar structure to the original ones having more boxes than holes. The fact that these maps could not be solved by the controllers

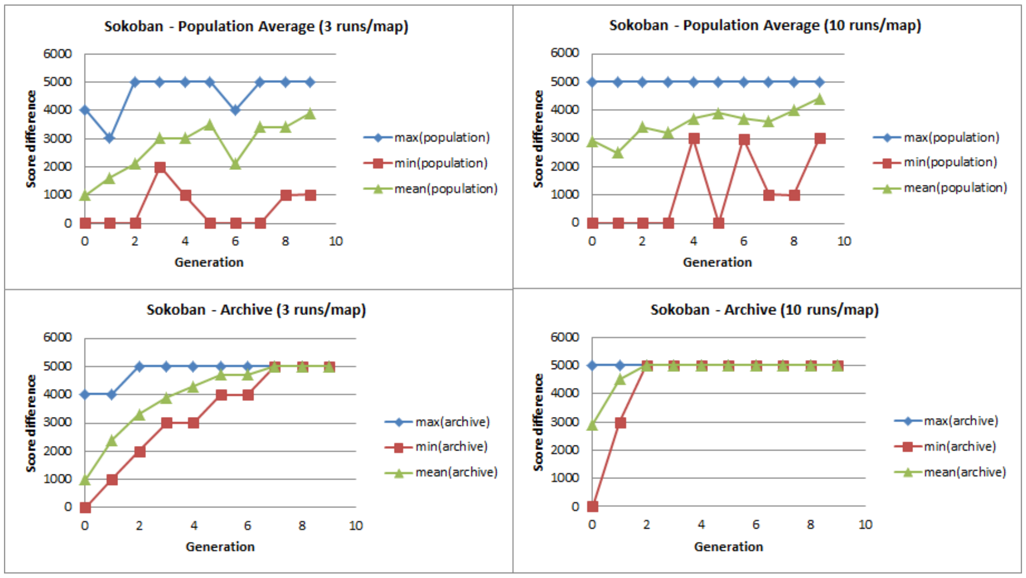


Figure B.16: Score differences for the game Sokoban. On the left column, the results of the experiment with the agents playing each level 3 times is shown. The right column shows the results of the experiment with 10 runs per map. The upper row shows the minimum, average and maximum fitness in the population per generation. The lower row shows the corresponding values of the individuals present in the archive at each generation.

indicates that they do not perform very well in this game. Thus, they can only be used for evaluation to a certain degree.

B.1.9 Survive Zombies

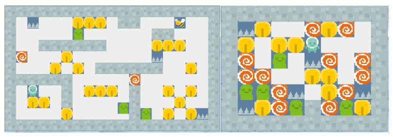


Figure B.17: Survive Zombies levels. Left: level provided by the GVG-AI framework; right: generated level.

In the game *Survive Zombies*, the player has to escape *Zombies*. If he collides with a zombie, he dies, unless he has collected *honey*. With the player having honey, the zombie dies on a collision with a player. Besides the zombies, there are some other creatures from *hell* who can kill the player but are not destroyable. As it is shown on the left part of figure B.17, there are usually only a few zombies and hell-creatures in a level and multiple honey vats. Furthermore, there are some *flowers* (normally, the same amount as zombies) which spawn *bees*. The bees create more honey colliding with zombies and killing them that way. The player can only win the game, if he survives the game round.

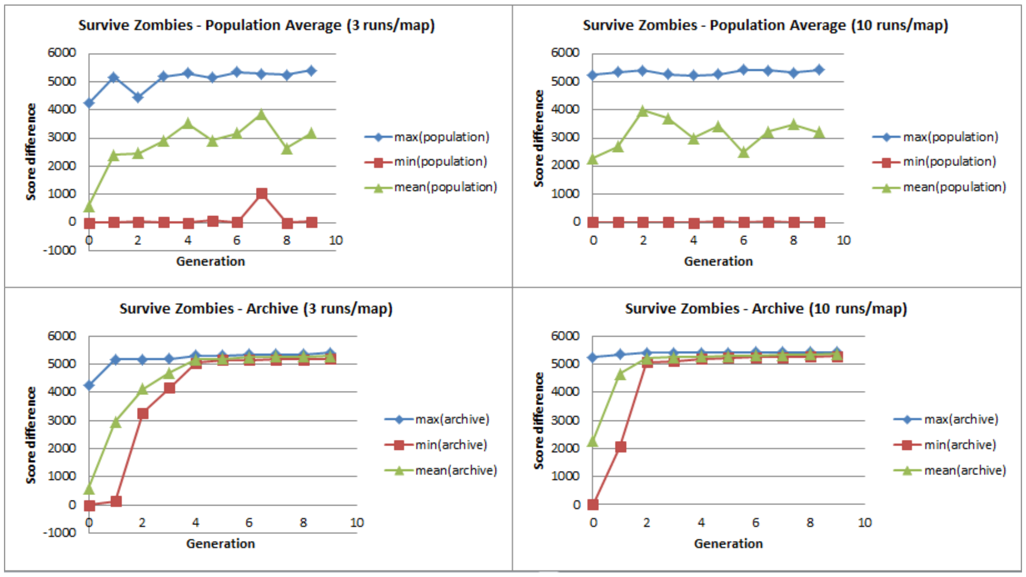


Figure B.18: Score differences for the game Survive Zombies. On the left column, the results of the experiment with the agents playing each level 3 times is shown. The right column shows the results of the experiment with 10 runs per map. The upper row shows the minimum, average and maximum fitness in the population per generation. The lower row shows the corresponding values of the individuals present in the archive at each generation.

The results shown in figure B.18 have proved that the generator could find solvable levels for this game. Thereby, the fitness values in the final state of the archive were higher in the experiment with 10 runs per map.

Looking at the generated rulesets, we have noticed that the generator balanced the number of negative objects (zombies, hell creatures) and positive objects (honey, flowers) very well in both experiments. However, the there was a visible difference between their distribution. In the experiment with 3 runs, the generator changed the difficulty through the number of zombies which can be killed in the game. Here, the number of zombies was always very high (5 to 11) and the number of the positive objects was in most cases lower. Thereby, the amount of hell creatures which could not be killed was never higher than4.

In contrast to that, the most of the final levels in the experiment with 10 runs per map, had relatively low amounts of zombies but higher amounts of hell creatures (5 to 8). Since the hell creatures could not be killed, we regard the levels from the second experiment as more difficult that the ones from the first experiment. Nevertheless, the general distribution of the game objects, that is shown on the right part of figure B.17 was similar to the one in the original levels.

B.1.10 Zelda

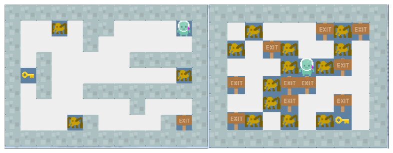


Figure B.19: Zelda levels. Left: level provided by the GVG-AI framework; right: generated level.

In the game *Zelda*, the player has to escape a dungeon through an *exit door* first collecting a *key*. Usually, there is only one of each if these objects placed in the level as shown on the left of figure B.19. Furthermore, there are a few *enemies* which can be killed or kill the player. Collecting the key and reaching the goal gives 1 score point each, whereas killing an enemy gives the player 2 score points.

The fitness values in both experiments of this game did not have any noticeable difference. In both cases, the generator could find solvable levels with the most rulesets having a fitness value above 4000 as shown in figure B.20.

Looking closer at the produced level, we have noticed that similarly to the games *Boulderdash* and *Portals*, the generator created levels with multiple exits (6 to 12). This happened due to the fact that a collision with an exit gave the agents a score point. That way, the selection mechanism preferred levels where the agents had a good chance to increase their score difference through goals.

Furthermore, the generated levels had very high amounts of enemies (10 to 12) as the right part of figure B.19 shows. The enemies were another element that gave the agents more score points. With such a high amount of enemies, the generator selected levels that had a much higher difficulty level than in the levels provided by the GVG-AI framework.

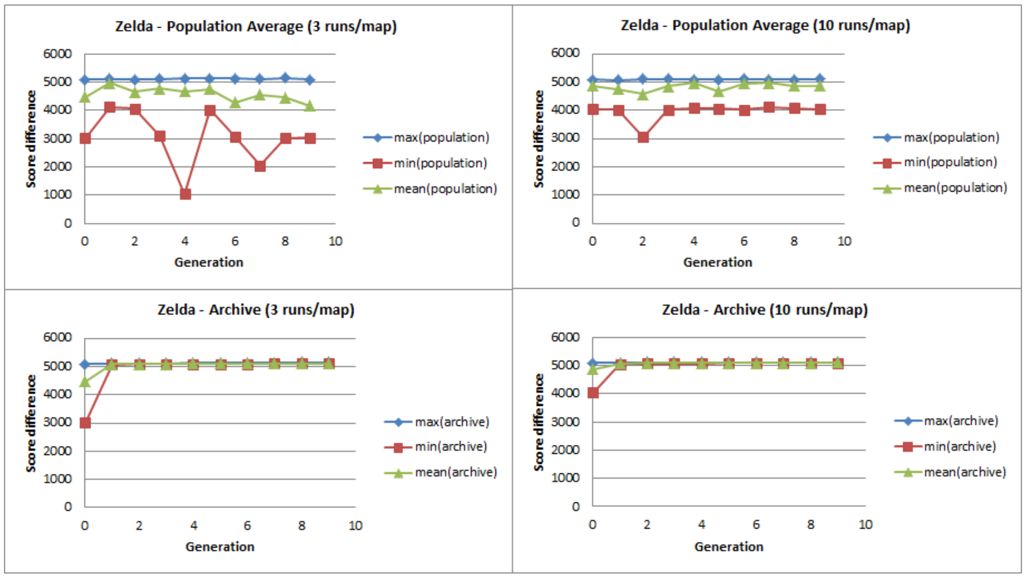


Figure B.20: Score differences for the game Zelda. On the left column, the results of the experiment with the agents playing each level 3 times is shown. The right column shows the results of the experiment with 10 runs per map. The upper row shows the minimum, average and maximum fitness in the population per generation. The lower row shows the corresponding values of the individuals present in the archive at each generation.

B.2 Results in the second Game Set

B.2.1 Camel Race

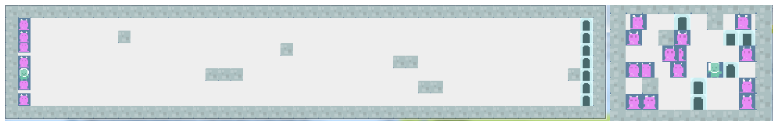


Figure B.21: Camel Race levels. Left: level provided by the GVG-AI framework; right: generated level.

In *Camel Race*, the player has to reach a *goal* before the other *camels* do so. In the original levels, there are as many goals as camels and the goals and camels are placed on the opposite sides of the level. In some cases, structures of walls are placed in the level as obstacles preventing the camels from running straight towards the goals which can be seen on the left side of figure B.21. The player is able to get only 1 score point when he wins (reaches a goal first). If one of the other camels wins, then the player gets a score of -1. Because it is possible to get only these to different scores, the maximum difference that could be reached in this game is 2. That is why the maximum difference in the results of this experiment was only 1.2 with a fitness value of 5001.2.

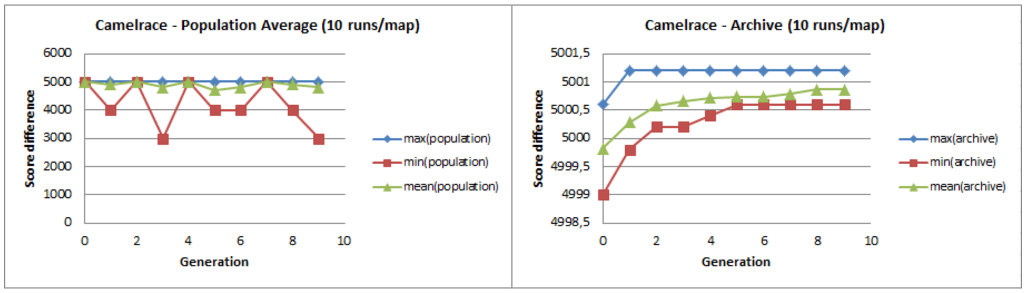


Figure B.22: Score differences for the game Camel Race. On the left, the minimum, average and maximum fitness in the population per generation is shown. On the right, are the corresponding values of the individuals present in the archive at each generation.

Although many rulesets had fitness values around 5000 (see figure B.22), these values were not meaningful because of the way how the levels were structured. In contrast to the original levels, there was not the same amount of camels and goals in the levels. Furthermore, the main difference between the generated and the original levels was that in the generated ones, the camels were not placed on one side of the level. Through the *direction* rule described in section 4.4, all camels were placed in such a way that they could run the longest distance in the direction that was defined in their characteristics. For example, all camels whose direction was *left* were placed on the right border of the level and those with the direction *right* were placed left. That way, goals, randomly moving camels, walls and the player were placed randomly in the middle of the level as shown on the right side of figure B.21. Thus, the distance between some goals and camels was smaller so that they could win the game faster without having a higher skill level in this game. So, the fitness values could be achieved through luck and cannot be regarded as meaningful.

However, the question arises whether the fact that the amount of goals and camels was not equal and that the goals did not have the same distance to all camels is a negative point. Creating unexpected levels from given VGDL descriptions could make the levels more interesting increasing their difficulty and even provide ideas for new games.

B.2.2 Digdug

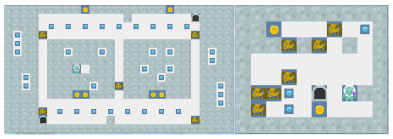


Figure B.23: Digdug levels. Left: level provided by the GVG-AI framework; right: generated level.

The levels of the game *Digdug* look quite similar to those of the game *Boulderdash* described in section B.1.2. They are full of walls which can be destroyed by the player as it is shown on the left of figure B.23. Furthermore, there are some *enemies* in the game which are spawned by a few spawn points. The enemies can be killed (and give 2 score points) or kill the player. Also, there are *goodies* which the player can collect and get 1 point each. The player can shoot at the walls and enemies with a boulder which he has to recollect before being able to reuse it. He wins when he either kills all enemies or collects all goodies.

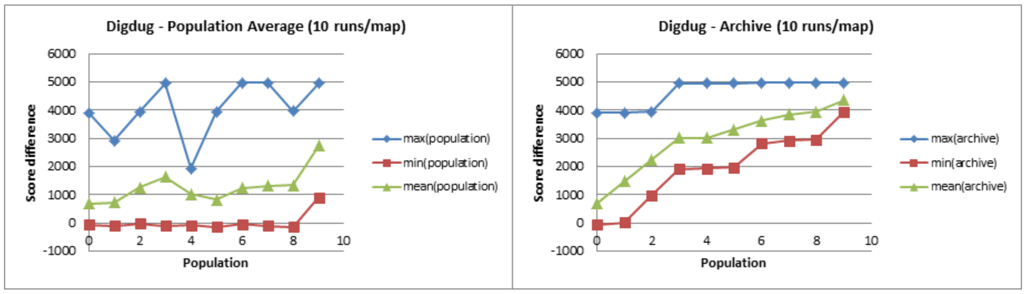


Figure B.24: Score differences for the game Digdug. On the left, the minimum, average and maximum fitness in the population per generation is shown. On the right, are the corresponding values of the individuals present in the archive at each generation.

Since the spawn points and the enemies were the main source of danger in this game, their amounts were very important. As the results in figure B.24 show, there was no ruleset that had a fitness value above 5000 meaning that no set produced 5

maps that could be solved by an agent. Looking closer at all 50 rulesets generated throughout the evolution process, we noticed that there was no set having only 1 spawn point and 1 enemy. These amounts would be enough for levels of such a small size.However, the 10 generations were not enough to get a ruleset with these numbers. Even though, $7/10$ final rulesets had only a single spawn points, they produced too many enemies providing a high difficulty level. That is also the reason why $33/50$ rulesets had a negative fitness value meaning that none of their levels could be solved and in many cases *sampleMCTS* agent could handle the game better than *adrienctx*.

The game could be won not only by killing enemies but also by collecting all goodies which means that a high amount of enemies could be compensated through a low amount of goodies. However, to win the game by collecting a single goodie in a level with a lot of enemies the agent should be placed close to that goodie. That can be seen in the levels of a ruleset which had a fitness value of $1015,6$. This set had a maximum amount of spawn points set to 12 and only 1 goodie. Here, one of the generated levels could be won because the agents were close enough to that goodie. Also, in other 4 levels *adrienctx* could score better than *sampleMCTS*.

However, this individual was not saved in the archive because of the generally low fitness value even though it showed a higher difference in scores. This example shows two important things. First, since the fitness function always prefers rulesets of which all 5 levels could be won, it is possible that sets with less solvable levels but a generally higher score difference are lost throughout the evolution process. Second, a higher population size and/or number of generations could lead to levels which provided a better balance between game objects (enemies and goodies).

B.2.3 Firestorms

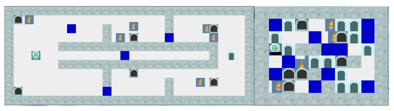


Figure B.25: Firestorms levels. Left: level provided by the GVG-AI framework; right: generated level.

In the game *Firestorms* the player has to reach a *goal* avoiding collisions with *fire*. If the player collides with fire, his score decreases by 1 point. Furthermore, he could die if he did not collect *water* in advance. As I is shown on the left part of figure B.25, there are usually approximately as many water objects as spwan points of fire and the avatar is placed far from the exit with walls building obstacles on his way towards the goal.

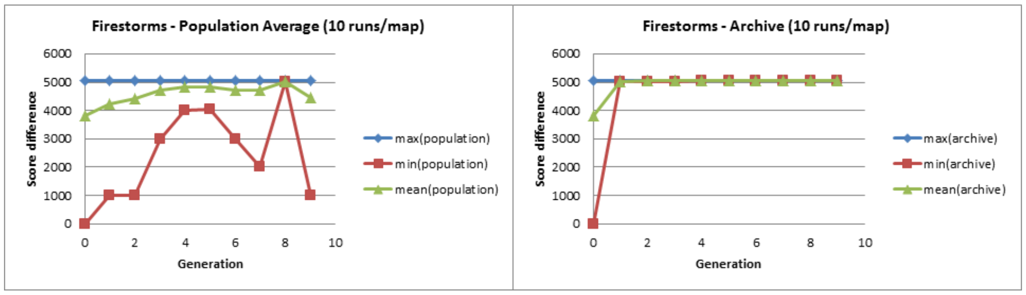


Figure B.26: Score differences for the game Firestorms. On the left, the minimum, average and maximum fitness in the population per generation is shown. On the right, are the corresponding values of the individuals present in the archive at each generation.

Since the only possibility to have a change in the game score was given by the collisions with fire items, the score could only be negative. Thus, the difference between the scores of the agents could only be positive if the weaker agent collided with more fire items than the smarter one. However, to survive the collision, an agent would have to collect water showing intelligent behavior. For that reason, we assume that in the selected solutions in this process the generator selected those levels, in which the weaker agent performed better than the smarter one. This means, that the fitness function was not appropriate for that game and should be changed in future.

Nevertheless, the generator created many solvable levels, as the results in figure B.26 show. However, looking at the final rulesets in the archive, we could see that all of them had a very high amount of water tiles (8 - 12) and the number of spawn points of fire was never higher than 7. These numbers could prove that the maps had low difficulty levels (with such a high number of water) that even the weaker agent could perform well solving them. Thereby, levels with 12 spawn points had very low fitness values around 0 meaning that none of the agent survived in these maps.

As in some other games described in this work, the generated level had multiple exits (see right side of figure B.25). Even though the exits did not give any score points, having a high number of them meant that the agents needed more time to solve the levels. For that reason, we propose taking into consideration the time required by the agents, when computing the fitness value of a level.

B.2.4 Infection

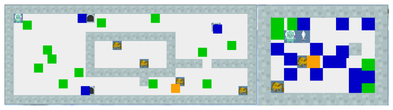


Figure B.27: Infection levels. Left: level provided by the GVG-AI framework; right: generated level.

In the game *Infection*, the player takes the role of the carrier of an infection which he has to distribute among *hosts*. The player and the hosts can be infected from a game object representing the *virus* or one of the *infected* entities. There are a few viruses and hosts in a level. Furthermore, there are some spawn point which spawn *guardians* who can heal as well the infected entities as the player. Colliding with a guardian, the players score decreases by 1 point. When killing a guardian with his sword or infecting a host, the player gets 2 score points. The player wins when all hosts get infected. As can be seen on the left part of figure B.27, walls may build obstacles for all moving entities in the level.

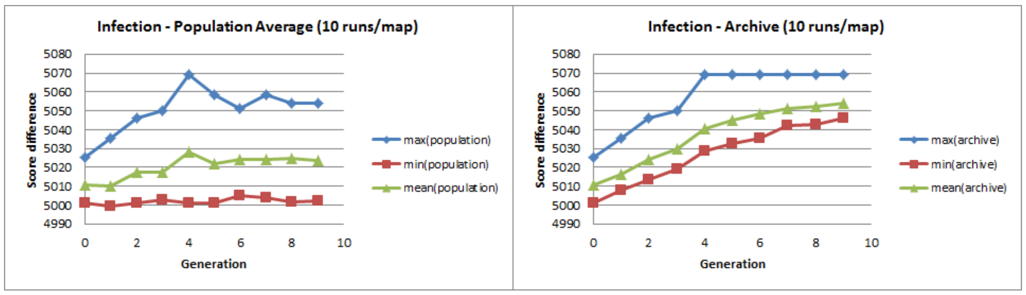


Figure B.28: Score differences for the game Infection. On the left, the minimum, average and maximum fitness in the population per generation is shown. On the right, are the corresponding values of the individuals present in the archive at each generation.

All rulesets generated for this game had a fitness value above 5000 meaning that all levels could be solved by at least one agent. However, there was a recognizable difference between well-balanced levels and those that were too easy or too hard. All final sets had a fitness value above 5046 (see figure B.28) having maximum 6 spawn points and 3 viruses. Whereas, e.g. levels that had more viruses got significantly lower fitness values because the hosts could be infected by the viruses without the agents help. These levels were too easy for both controllers.

Here, the generator has shown again that it is able to find a good balance between the virus objects and the host entities. Even though it wrongly handled the hosts and guardians as subtypes of the same object and created a single rule for both of them. This happened because of the way these agents were defined in the VGDL description of the game. Since the walls did not play an important role in the game, final rulesets had as well only 1 wall as up to 10 walls in their levels. However, these wall tiles did not build any structures as it is shown on the right part of figure B.27.

B.2.5 Firecaster

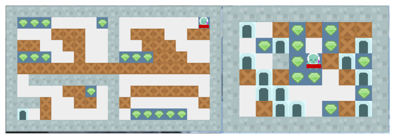


Figure B.29: Firecaster levels. Left: level provided by the GVG-AI framework; right: generated level.

Similarly to many other games, the player has to reach a *goal* in the game *Firecaster*. The goal is hidden behind *walls* and *boxes* which the player has to destroy. For that purpose, the player needs to collect *mana* objects which he then can use to shoot at the boxes. Collecting a mana object and destroying a box gives the player 1 score point each. As the left part of figure B.29 shows, there is usually only one goal in the level and a small amount of mana objects, so that the player has to use them wisely to be able to reach the goal.

The results in this experiment have shown once again that the generator able to find rulesets that provided solvable levels (see figure B.30). However, the main aspect hereby was the balance between the number of the boxes and the mana objects and not the structure of the level. The most sets had almost equal number of boxes and mana. Such levels were preferred by the generator because they gave the more intelligent player the chance to gain a higher score destroying more

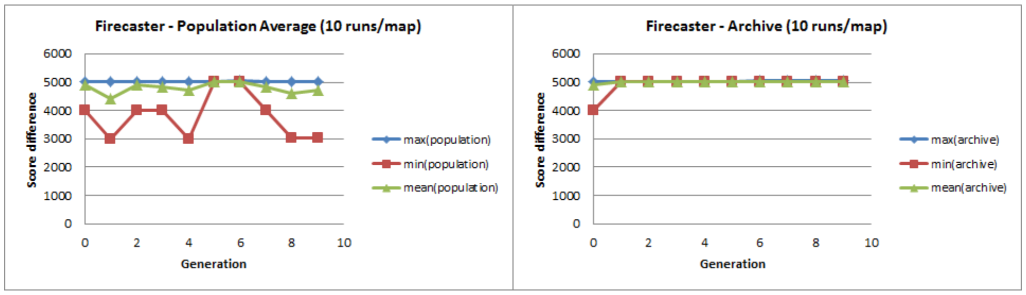


Figure B.30: Score differences for the game Firecaster. On the left, the minimum, average and maximum fitness in the population per generation is shown. On the right, are the corresponding values of the individuals present in the archive at each generation.

boxes. Levels with more mana than boxes were too easy for both players and got fitness values around 5000.

Another interesting point is that the most levels had a high number of goals. Though the goals did not have a direct influence on the score, they affected the time required to solve the level. However, in the most cases, the goals were not hidden behind boxes and could be reached by the agents all the time as it is shown on the right side of figure B.29. Nevertheless, the agents destroyed the boxes gaining more score points. Though, in those levels, were a goal was actually hidden behind boxes and there were less mana objects than boxes, the agents could not win the game. That way, levels that were more similar to the original ones, had a fitness value below 5000 and were not chosen by the selection mechanism.

B.2.6 Overload

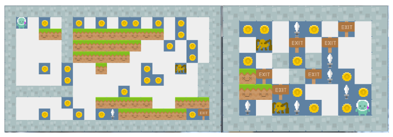


Figure B.31: Overload levels. Left: level provided by the GVG-AI framework; right: generated level.

In the game *Overload*, the player has to reach a *goal* having collected 10 diamonds in advance. When he reaches the goal, he gets 1 score point. However, if he collects 11or more diamonds and collides with a *marsh*, he dies. He can collect a *sword* getting 2 score points and destroy marsh with this sword. Furthermore, there is a NPC (randomly moving entity) who is also able to collect diamonds. As we can see on the left part of figureB.31, there is usually only one sword and one exit in the level with a low number of NPCs but a high number of marsh tiles and diamonds.

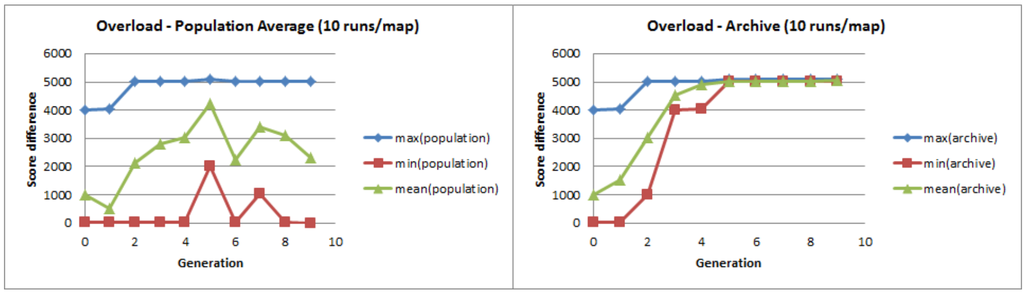


Figure B.32: Score differences for the game Overload. On the left, the minimum, average and maximum fitness in the population per generation is shown. On the right, are the corresponding values of the individuals present in the archive at each generation.

As the fitness curves in figure B.32 show, the generator could find solvable levels for this game. Since having a sufficient number of diamonds was an important criteria for winning the game, all final rulesets had 10 to 12 diamonds (a higher number was not allowed due to the size of the map). Also, the generator performed well in finding levels with a low amount of NPCs making it possible for the agents to solve the maps. 7 out of 10 final rulesets had only 1 NPC in their levels. The other rulesets had 2 to 3 NPCs. Nevertheless, similar to many other games, the generator created multiple exits and weapons in the levels (see right part of figure B.31). These objects gave the agents the possibility to gain more score points.

B.2.7 Pacman

In *Pacman*, the player has to collect all positive items such as *pellets, power pills* and *fruits* getting 1, 10 and 5 score points. Thereby, he has to avoid collisions with *ghosts*. If the player collects a power pill, he can kill a ghost getting 40 score points. The ghosts are spawned in certain *spawn points*. As we can see on the left

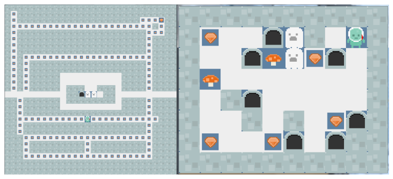


Figure B.33: Pacman levels. Left: level provided by the GVG-AI framework; right: generated level.

part of figure B.33, the original level consist of labyrinth build with *wall* tiles. The size of these levels is relatively big.

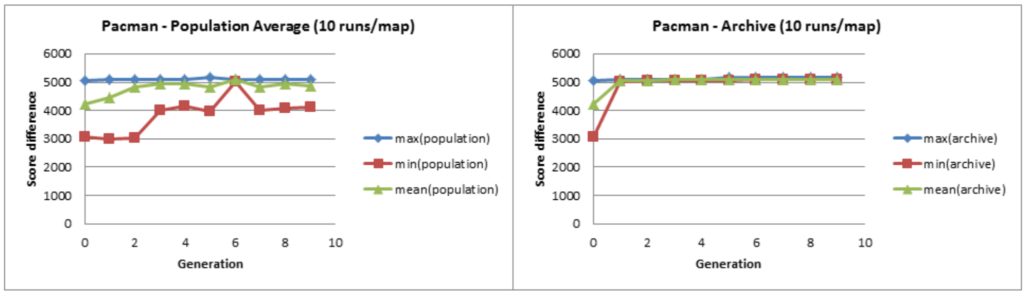


Figure B.34: Score differences for the game Pacman. On the left, the minimum, average and maximum fitness in the population per generation is shown. On the right, are the corresponding values of the individuals present in the archive at each generation.

As we can see in figure B.34, the most levels generated for this game were solvable by at least one of the agents. However, the levels did not have any labyrinths and had only a few wall tiles (see right part of figure B.33). Searching for the right difficulty level, the generator concentrated on balancing the amounts of spawn points and positive items. Thereby, all collectables were handled in a single rule, so that specifying the number of *food*, the ruleset did not specify the numbers of single types of game objects.

Without the labyrinths, the agents could freely move in the levels collecting the items. The only obstacles were the ghosts. However, looking at the final rulesets, we could not find any relation between the amounts of the collectables and the spawn points. There were some relatively empty levels with only 3 spawn points and 5 food items, as well as some maps with 10 spawn points and 8 collectables.

B.2.8 Seaquest

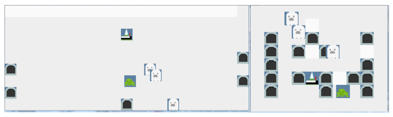


Figure B.35: Seaquest levels. Left: level provided by the GVG-AI framework; right: generated level.

In the game *Seaquest*, the player plays the role of a submarine. There are some dangerous *sharks* and *whales* in the water which can kill the player. He can also shoot at the enemies and get 1 score point killing each of them. Furthermore, the player has to return to the surface to collect *air* which he loses colliding with *bubbles*. Additionally, he has to save *divers* bringing them to the surface. For each diver that he saves, he gets 1000 score points. The left part of figure B.35 shows that the original levels are very empty having only a few holes that spawn the fishes, one hole for the diver and one for the bubbles. The air is placed on the top of a level. The player wins the game if he survives the game round (1000 game steps).

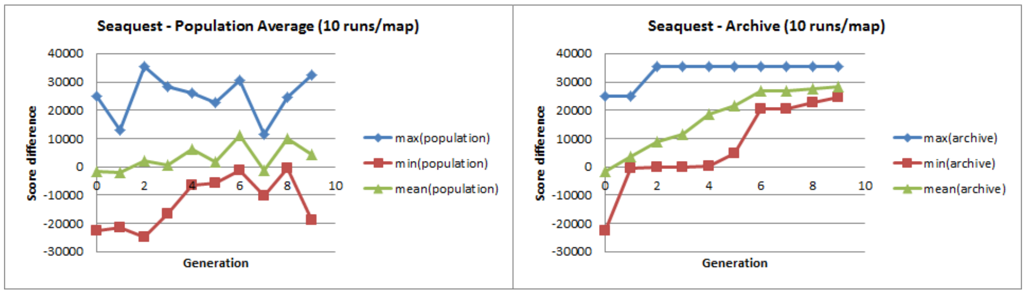


Figure B.36: Score differences for the game Seaquest. On the left, the minimum, average and maximum fitness in the population per generation is shown. On the right, are the corresponding values of the individuals present in the archive at each generation.

As we can see in figure B.36, the fitness values for this game were much higher than 5000 due to the fact that the player got 1000 score points for each diver that he saved. At this point, the fitness function should be adjusted in accordance with the maximum score values. Additionally, the fitness curves show that some rulesets got negative fitness value which means that in some cases the weaker agent performed better than the smarter one.

In this game, the sources of danger were the bubbles and the fishes. Thereby, the fishes could be killed by the agents whereas the bubbles could not be destroyed. The final rulesets reflected this behavior by having only 1 bubble hole in all levels and many fish-spawning holes. Looking at the results of all generated rulesets, we could notice that those rulesets that had more bubble-holes got very low fitness values.

At the same time, the air and the divers were the positive objects that helped the player get more score points. Thereby, the air played the more important role since it could be consumed by the player and was not recovered anymore. Thus, the amount of air tiles specified the difficulty of the level. Therefor, 9 of 10 final rulesets had maximal 4 air tiles being solvable, though not too easy for the agents.

As we can see on the right side of figure B.35, the level structure of the generated maps was similar to the original ones in that sense, that the fish-spawning holes were placed on the left and right borders of the maps. This happened due to their orientations that were specified in the VGDL descriptions. However, the air tiles were placed randomly in the level not representing the surface of the water.

B.2.9 Whackamole

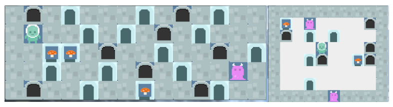


Figure B.37: Whackamole levels. Left: level provided by the GVG-AI framework; right: generated level.

In the game *Whackamole*, the player has to collect moles which periodically come out of *holes*. Thereby, he gets 1 score point for each mole. There is a *cat* which also collects the moles decreasing each time the players score by 1 point. If the player collides with the cat, he dies. To win the game, he has to survive the game round. As we can see on the left side of figure B.37, there is a high number of holes for the moles distributed equally in the level.

The results of this experiment have shown that the generator could find many levels that were solvable by the agents (see figure B.38). With the cats being the single source of danger and the holes being the single source of attraction, the generators aim was to find the best balance between these objects. In contrast to the original levels, all final maps in the archive were more difficult with 2 to 3 cats. The amounts of the holes were between 8 and 12. We could notice that maps

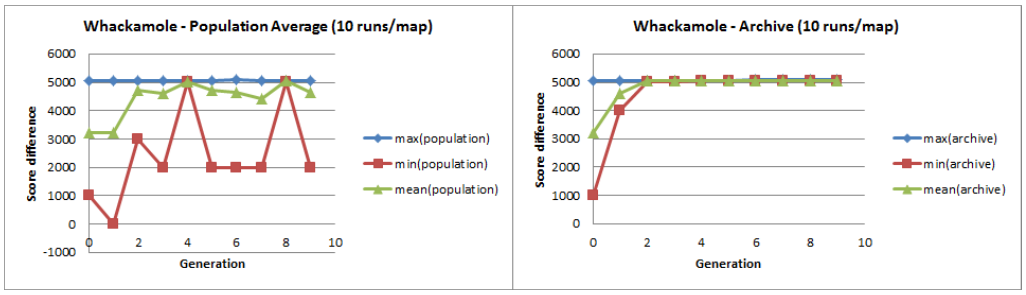


Figure B.38: Score differences for the game Whackamole. On the left, the minimum, average and maximum fitness in the population per generation is shown. On the right, are the corresponding values of the individuals present in the archive at each generation.

with significantly more cats had very low fitness values and could not be solved by the agents.

B.2.10 Eggomania

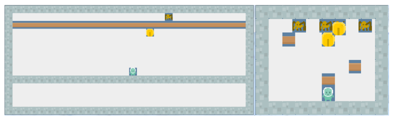


Figure B.39: Eggomania levels. Left: level provided by the GVG-AI framework; right: generated level.

In *Eggomania*, a *chicken* that is placed on the top of the level, throws eggs at the player who is placed on the bottom of the level. There is usually only one chicken, though in one of the original levels, there are two of them. The players aim is to kill all chickens. Furthermore, he may not let any eggs reach the ground (*wall*). Therefore, he has to catch them. When he catches an egg, he gets 1 score point and can use the egg as a weapon against the chicken. Killing a chicken gives the player 100 score points. There are no wall sprites between the avatar and the chicken. Though, there may be some *trunk* sprites that do not have any effect in the game except from an optical one (see right part of figure B.39).

As the results in figure B.40 show, none of the generated rulesets got a fitness value above 5000. This happened because accordingly to the way we set the amounts of game objects, there should always be at least 1 object of each type

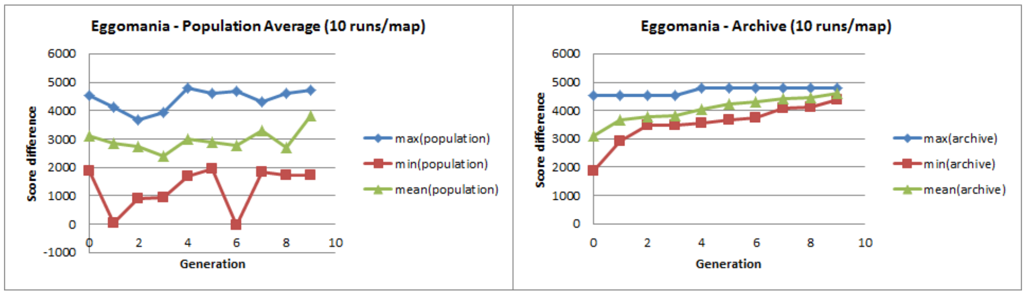


Figure B.40: Score differences for the game Eggomania. On the left, the minimum, average and maximum fitness in the population per generation is shown. On the right, are the corresponding values of the individuals present in the archive at each generation.

in the level. That way, all rulesets had at least 1 wall tile inside their levels. In those levels, where the wall tile was placed between the player and the chicken, an egg could fall onto the wall (which would lead to a game over) without the player being able to prevent this. However, the results show that those rulesets have survived the evolution that had a low number of walls.

Additionally, the final rulesets have shown that the generator was able to filter out levels with a certain degree of difficulty. $9/10$ final individuals had 3 chickens providing more difficult levels than the original ones. Furthermore, killing a chicken would bring an agent 100 score points, meaning that levels with more chickens could provide higher score differences and fitness values. In all levels, the main idea of the game sustained, so that the chickens were always placed on the top and the agents on the bottom of the level as it is shown on the right side of figure B.39.

Bibliography

[1] G. N. Yannakakis and J. Togelius, "A panorama of artificial and computational intelligence in games," 2014.

[2] J. Togelius, N. Shaker, and M. J. Nelson, "Introduction," in *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, N. Shaker, J. Togelius, and M. J. Nelson, Eds. Springer, 2015.

[3] S. Dahlskog and J. Togelius, "Procedural content generation using patterns as objectives," in *Applications of Evolutionary Computation*. Springer, 2014, pp. 325–336.

[4] ——, "A multi-level level generator," in *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*. IEEE, 2014, pp. 1–8.

[5] A. Liapis, C. Holmgård, G. N. Yannakakis, and J. Togelius, "Procedural personas as critics for dungeon generation," in *Applications of Evolutionary Computation*. Springer, 2015, pp. 331–343.

[6] A. Liapis, G. N. Yannakakis, and J. Togelius, "Towards a generic method of evaluating game levels." in *AIIDE*, 2013.

[7] ——, "Enhancements to constrained novelty search," 2013.

[8] ——, *Generating map sketches for strategy games*. Springer, 2013.

[9] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelbäck, G. N. Yannakakis, and C. Grappiolo, "Controllable procedural map generation via multiobjective evolution," *Genetic Programming and Evolvable Machines*, vol. 14, no. 2, pp. 245–277, 2013.

[10] J. Togelius, N. Shaker, and J. Dormans, "Grammars and l-systems with applications to vegetation and levels," in *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, N. Shaker, J. Togelius, and M. J. Nelson, Eds. Springer, 2015.

[11] J. Dormans, "Adventures in level design: generating missions and spaces for action adventure games," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. ACM, 2010, p. 1.

[12] N. Shaker, M. Nicolau, G. N. Yannakakis, J. Togelius, and M. O. Neill, "Evolving levels for super mario bros using grammatical evolution," in *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*. IEEE, 2012, pp. 304–311.

[13] N. Shaker, A. Liapis, J. Togelius, R. Lopes, and R. Bidarra, "Constructive generation methods for dungeons and levels," *Procedural Content Generation in Games: A Textbook and an Overview of Current Research. Springer, Heidelberg*, 2015.

[14] L. Johnson, G. N. Yannakakis, and J. Togelius, "Cellular automata for real-time generation of infinite cave levels," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. ACM, 2010, p. 10.

[15] I. D. Horswill and L. Foged, "Fast procedural level population with playability constraints." in *AIIDE*, 2012.

[16] A. M. Smith and M. Mateas, "Variations forever: Flexibly generating rulesets from a sculptable design space of mini-games," in *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*. IEEE, 2010, pp. 273–280.

[17] M. J. Nelson and A. M. Smith, "Asp with applications to mazes and levels," in *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, N. Shaker, J. Togelius, and M. J. Nelson, Eds. Springer, 2015.

[18] A. M. Smith and M. Mateas, "Answer set programming for procedural content generation: A design space approach," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 3, no. 3, pp. 187–200, 2011.

[19] A. M. Smith, E. Butler, and Z. Popovic, “Quantifying over play: Constraining undesirable solutions in puzzle design.” in *FDG*, 2013, pp. 221–228.

[20] A. Zook and M. O. Riedl, “Automatic game design via mechanic generation,” in *Proceedings of the 28th AAAI Conference on Artificial Intelligence*, 2014.

[21] T. Schaul, “A video game description language for model-based or interactive learning,” in *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*. IEEE, 2013, pp. 1–8.

[22] D. Perez, S. Samothrakis, J. Togelius, T. Schaul, S. Lucas, A. Couetoux, J. Lee, C. Lim, and T. Thompson, “The 2014 general video game playing competition,” *Computational Intelligence and AI in Games, IEEE Transactions on*, 2015. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7038214

[23] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele, “A users guide to gringo, clasp, clingo, and iclingo,” 2008.

[24] M. Preuss, A. Liapis, and J. Togelius, “Searching for good and diverse game levels,” in *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*. IEEE, 2014, pp. 1–8.

[25] T. S. Nielsen, G. A. Barros, J. Togelius, and M. J. Nelson, “General video game evaluation using relative algorithm performance profiles,” in *Applications of Evolutionary Computation*. Springer, 2015, pp. 369–380.

[26] D. Perez, J. Togelius, S. Samothrakis, P. Rohlfshagen, and S. Lucas, “Automated Map Generation for the Physical Travelling Salesman Problem,” *IEEE Transactions on Evolutionary Computation*, vol. 18:5, pp. 708–720, 2014.