OTTO-VON-GUERICKE UNIVERSITY MAGDEBURG

FACULTY OF COMPUTER SCIENCE
Institute for Intelligent Cooperating Systems

MASTER THESIS

Multi-objective Procedural Level Generation for General Video Game Playing

Author: Jens Dieskau
Supervisors Prof. Dr. Sanaz Mostaghim
Dr. Diego Perez-Liebana
Date: 08.12.2016

Abstract

This thesis presents a new way to automatically generate levels for arbitrary games that are described in the Video Game Description Language (VGDL). A large number of different approaches to procedural generated content have emerged in recent years. Most of these generators are specialized towards a specific level design. They either need some manual adjustments or utilize a lot of specific rules to generate reasonable levels for different kinds of games. Using them in the context of *general game playing* would not be possible immediately. Therefore, a more flexible method to generate a wide variety of vastly different level layouts is proposed here. A handful of parameters can be used to control the generated level design.

To realize this, a set of different "*Likeliness-Matrices*" are used to guide the algorithm where game objects should be placed in a level. Each matrix encodes one specific property of the desired level. Some properties can be directly extracted from the game description, whereas others must be either set by hand or chosen randomly. In order to avoid manual interaction, an Evolutionary Algorithm (EA) was used for the experiment to automatically find suitable values to fill out the missing information.

The experiment used 20 different games that are provided by the *GVG-AI* framework to demonstrate the performance of this approach. The results confirm that the proposed method is expressive enough to generate a large variety of playable and interesting levels.

Contents

List of Abbreviation

1. Introduction

In the ever-growing market of computer games [Vid16], there is an insatiable thirst for more and novel game content. This is aggravated by the fact that modern game development needs more and more money. A game for the original *PlayStation* took around $800,000 to $1.7 million to develop, whereas games for its successor *PlayStation 2* needed a budget of $5 to $10 million [LW05]. Games for more recent consoles, like *PS3* or *XBox 360*, cost between $20 and $30 million [Gib09] – rising tendency. Traditionally, besides marketing expenses, one of the biggest parts of a game budget is allocated to developers and game designers.

A remedy could be to either reduce the need of additional employees or to increase their efficiency – and this is where *Procedural Content Generation (PCG)* comes into play. PCG is a way to automatically generate new game content or even to design complete new games. This means, it is an algorithmic way to create novel game elements with limited or without any user interaction. Game content could be anything a player can interact with in a video game, directly or indirectly, e.g. from levels, maps, textures to vegetation, music or even complete storylines. This greatly helps to reduce the needed number of artists and opens an opportunity for smaller teams to develop competitive video games. PCG can also be used as a way to help guide an artist or novice user, to find new and innovative designs or simply use it as a source of inspiration.

This thesis will focus on automatically creating game levels. To be more precise: to generate game levels in the context of *General Game Playing (GGP)*. GGP is part of the wide area of *Artificial Intelligence (AI)* research and focuses on creating programs (called *agents*) that are able to play different kinds of games without prior knowledge about the specific game rules. Therefore, the level generator should be able to create levels for multiple kinds of different games without any special adjustments.

1.1. Problem Statement

The goal of this thesis is to create multiple levels for various games with the help of different agents. At least one agent should be a *Single-Objective (SO)* agent and one should be a *Multi-Objective (MO)* agent. Investigations should then be made between the levels to try to identify distinct level characteristics. The main question is therefore: Can any apparent difference in the level structure or design be determined between levels generated by different kinds of agents?

Creating these levels consists of two parts – generating levels and testing them. For this, an *Evolutionary Algorithm (EA)* is used. The level generation itself is an iterative constructive algorithm, whereas the testing uses a simulation-based approach with the help of multiple predefined agents.

1.2. Thesis Structure

The thesis starts with an overview of state-of-the-art related works in the level generation research area. The next chapter will provide background information that is necessary to understand the presented algorithms. A special focus is put on the used software from GGP. Chapter 4 describes the components of the algorithm to create new levels, such as the EA or a detailed explanation of the evaluation function. After all these theoretical information, the next chapter will present the experiment. Both the setup and the results of the practical work are shown here. This is followed by a conclusion and discussion, before the final chapter will give the reader an outlook on possible future enhancements.

2. Related Works

The main work of this paper revolves around procedural level generation. Therefore, this chapter will review foundational work done in research as well as from industry projects about PCG.

As already mentioned in the introduction, the interest in PCG is neither new nor untested in the gaming industry. The first procedural games are quite old and were not even computer games. They were analogue and a human player had to follow instructions to generate the content [Smi15]. These can be seen as pioneers in the field of PCG. However, the interest here lies more in digital games. The earliest were invented in the 80s. Well-known representatives are *Rogue* [CMJ15] and *Elite*. They both use PCG to create the game environments for a player. More recent examples are *Darkspore* (enemy generation) or *Borderlands* (weapon creation). There are numerous more examples from middleware to generate realistic looking trees (*SpeedTree*) to games that use whole generated universes (*Elite Dangerous*, *No Man's Sky*). Besides the interests of the game industry, it is also an active research area. The goal here is not only to find completely new generating methods, but also try to find better ways to control the involved random processes and make the outcome, on the one hand, more reliable, but also more varying on the other hand.

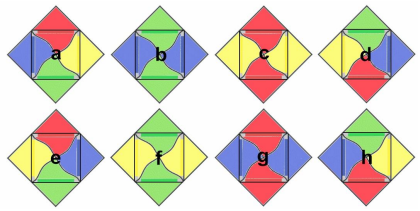
One of the most detailed overviews about the topic was published as a survey by J. Togelius et al. [TYSB11]. The authors also provide a excellent taxonomy. It is worthwhile to read this paper to get an overview about the topic. Additionally, they list desired properties that a procedural content generator should have. The later described algorithm will use this properties to justify particular design decisions. Thus, details about this can be found in Section 4.

The next sections will briefly introduce different PCG techniques and provide useful references for further details.

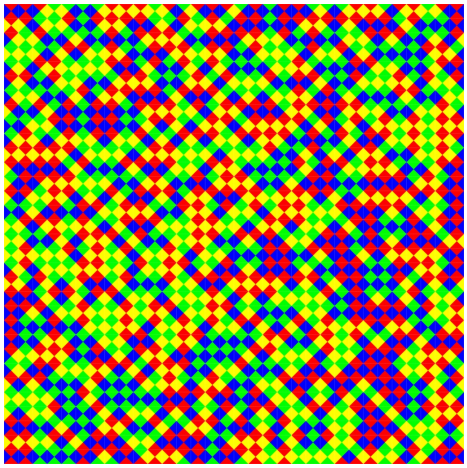
2.1. Combination of pre-made parts

One of the simplest and widely used methods to create novel content is to use pre-made parts and combine them in a new way. This technique uses some kind of rules to define which parts are repeated and where they are placed. Hao Wang invented *Wang tiles* in 1961 [Wan61] that nowadays are used, amongst a lot of other things, to generate levels. Such tile is typically a square with four different colors on each edge. A set of such tiles are arranged to form a pattern without using any rotation or reflection. Neighbouring tiles must have the same color. The left

image in Figure 2.1 shows eight such tiles that were invented by M. F. Cohen et al. [CSHD03]. The right image shows an example tiling created with these tiles. A similar method called *Occupancy-Regulated Extension* was recently successfully used to create Super Mario levels [MM10].



(a) An example set of eight *Wang tiles* [CSHD03, Fig. 4b]



(b) A tiling created with the eight tiles from ***(a)*** [CSHD03, Fig. 5]

Figure 2.1.: Wang tiles proposed by M. F. Cohen et al.

Another common method here is to use a *rhythm based* approach. *Rhythm based* methods try to interpret level generation like a piece of music. Whereas music is composed of different notes, a level also consists of small (atomic) pieces. This is especially useful for 2D platformers [STWM09], since levels for this games are linear just like a song.

2.2. Search-based Methods

Search-based techniques try to find an optimal solution to an optimization problem. That means, these kinds of algorithms search for an optimal solution throughout a search space. The search space contains all possible solutions – good and bad ones. Furthermore, a fitness function is needed to determine the quality of a solution. Implementing such methods is often quite challenging. Finding a meaningful fitness function can be difficult. Additionally, a proper encoding of a solution is needed. Such representation could be as simple as a vector of numbers or up to a sequence of level parts or patterns. An common search method is an Evolutionary Algorithm. A big disadvantage of such methods is their poor efficiency, especially for larger search spaces. Metaheuristics like Evolutionary Algorithm or Genetic Algorithm have no guarantee that they will find the global optimal solutions. In most cases they will only find local maxima.

Again, there are countless examples of successful applications of search-based methods to generate levels. Valtchanov et al. encoded their solutions in a tree structure [VB12]. A node in their tree represents a partial level (mostly rooms) and an edge represents a connection between the rooms. A similar method was used for the game Super Mario. They used a sequence of micro-patterns as the level encoding [DT14].

An interesting experiment was published by Ashlock et al. [ALM11]. They used four different representations to generate maze-like levels. In their case, a direct representation (using bits to encode where a wall is) resulted in more interesting mazes. They also experimented with different fitness functions and their impact on the result. They found out that changes on the fitness function had the biggest impact in comparison to other EA parameters.

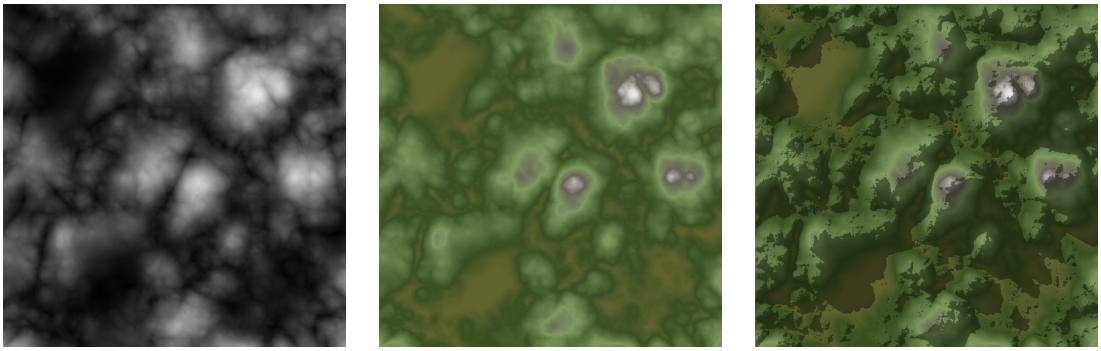
Since the algorithm used in this paper will utilize an EA, a detailed explanation will later be described in Section 4.6.

2.3. Noise Generators

Simply said, noise is just a series of random numbers and a noise generator is a function to create (semi-)random data sets. Natural objects have almost always some kind of noise – blurred or grainy images, noisy sound and so on. Noise is what makes natural objects to appear *natural*. Therefore, to procedurally generate something that looks natural, noise is necessary. Just using random numbers would not lead to the desired result in most cases. For example in the case of level generators, placing objects at random positions is not what a designer would do. A designer would group certain objects together and scatter others around the map. There is a kind of regularity, a pattern, in which objects are arranged.

This is exactly the reason why Ken Perlin developed the first noise generator called *Perlin noise* in 1983 [Per85]. It was one of the first algorithms used for procedural generation. The noise is generated by generating a lattice of (pseudo) random values, which are then interpolated to fill out the space between the lattices. Ken Perlin later developed the *Simplex noise* [Per01] to resolve some disadvantages. *Simplex noise* has a lower computational complexity and more importantly it does not suffer from noticeable directional artefacts. Details can be found in [KKS08]. A slightly different version of *Simplex noise* is *OpenSimplex noise*. *OpenSimplex* uses some smaller tweaks to avoid patent-related issues. The patent is mostly about the tessellation function for higher dimensional noise generation. Therefore, 3D *OpenSimplex* uses a tetrahedral-octahedral honeycomb instead of a tetragonal disphenoid honeycomb from the 3D version of *Simplex noise*.

Originally developed to generate textures, noise generators are now used for a lot of more



(a) Perlin noise

(b) Coloured noise

(c) Noise with simulated shadow

Figure 2.2.: Example Perlin noise used to create a landscape image.

content. For example, the texture could be interpreted as a height to create a landscape like the one shown in Figure 2.2. The different noise values can also be used for placing game objects or to determine different game areas. The famous game *Minecraft* uses such a technique to define biomes and to ensure a smooth transition between them.

2.4. Constraint-based Methods

Constraint-based methods try to define desired properties of a solution instead of describing a sequence of steps to reach a useful solution. Some algorithms can differentiate between hard and soft constraints. Hard constraints must be fulfilled, whereas soft constraints are optional. An advantage is that only feasible solutions are generated and a fitness function is not needed. Additionally, the search space is way smaller compared to methods that can also produce infeasible solutions. A disadvantage is that defining the constraints needs sufficient knowledge of the game mechanics. Finding general constraints that are applicable to a lot of different games and are able to produce playable levels is not an easy task.

Answer Set Programming (ASP) is a widely used method to generate new content. It uses a declarative logic language called *AnsProlog*. *AnsProlog* can be used to describe how a solution to a given problem should look like with the help of simple logic rules and statements. It was already shown that this method is applicable in generating levels [NS16] [SM11], even generating 3D structures is possible [RP04].

2.5. Grammar-based Methods

Originally invented to describe natural language [Cho68], grammar-based methods are applied to all kinds of problems in computer science nowadays. An alphabet and a set of rules are needed to define a grammar. A rule describes which symbol of the alphabet must be replaced with one or more other symbols. The rules are applied iteratively until a termination condition is reached, i.e. until no symbol can be replaced anymore. A simple example of a result of such a system from the works of O. Šťava et al. can be seen in Figure 2.3.

A distinction is made between deterministic and non-deterministic grammars. Rules in the first case are always unambiguous. However, non-deterministic grammars can have multiple rules for the same symbol sequence.

Generating content with such methods is very efficient and therefore applicable in real-time. However, a particular drawback is that the grammar must be customized for each game. There are no general rules to cover several different games. Thus, using such techniques for GGP is limited.

(a) ***(b)*** ***(c)*** ***(d)***

Figure 2.3.: Branching-like structure generated with different rule systems [ŠBM+10, Fig. 1]

Nonetheless, grammars can be used to create a wide variety of different kinds of content. From creating vegetation (e.g. *SpeedTree* [1]), modelling buildings [MWH+06] up to creating complete game missions [Dor10]. Levels have also been generated with such methods, e.g. Super Mario levels [SNY+12] or dungeons [DB11].

2.6. Constructive Methods

Constructive-based methods generate content step-by-step. One run results in only one solution. Therefore, there is no iterative process to re-evaluate and to improve the generated content. Constructive algorithms can be differentiated into two categories: *space partitioning* and *cellular automata*.

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

Space partitioning algorithms divide the 2D or 3D game space into smaller disjoint subareas (*cells*). Sometimes the subdivisions are applied recursively. Thus, each cell can be split up in even smaller cells. This results in a hierarchy of cells – a tree.

The basic concept of cellular automata was already discovered in the 1940s, but a real breakthrough brought *Conway's Game of Life*. It was developed by John Horton Conway in 1970 [Gar70]. A cellular automation is a self-organizing structure consisting of a regular grid of cells. Each cell has a number of states, like *on* and *off* in the simplest case, and a reference of their neighbour cells. They also have some initial state at the beginning ($t = 0$). At last, there is a rule set (oftentimes a mathematical function) that defines the next state ($t + 1$) of each cell according to the current state. Repeatedly applying the function will result in different patterns. The resulting pattern highly depends on the initial state, the rules and how many iterations were performed. To actually use this as a level generator, the *on* / *off* states could be, for example, interpreted as *walls* / *free passage*. Johnson et al. defined an automation to generate cave like structures [JYT10] with just three states (*floor*, *rock*, *wall*) and two simple rules. Other publications used similar rules to create dungeons [vdLLB14] or landscapes [Ols04].

Using cellular automata has several advantages. For one, the implementation is mostly simple and the resulting algorithm quite efficient. It also has the advantage that infinitive levels can be generated and due to its efficiency, it can also be done in realtime while the game is played. A big drawback is the lack of direct control of such algorithms. For example, it is either very hard or not possible to ensure that generated dungeons are reachable by the player. Single dungeons could be cut off from the rest of the level. An additional method is needed to ensure that this kind of cases gets fixed.

2.7. Discussion

As presented in the previous sections, there are many different techniques available to generate content. The brief introduction is just to get an overview. Surveying all algorithms in detail is out of the focus of this paper. The given references should be sufficient to start digging deeper into the subject.

Most of the shown techniques are applicable for creating level designs, but none of the them is, in general, really better than another. All of them have different pros and cons. J. Togelius et al. described desirable properties that a PCG system should have: speed, reliability, controllability, expressivity and diversity, creativity and believability [YT15]. These properties could be used to categorize the presented approaches, but some of them are rather subjective and therefore difficult to classify properly. However, one can easily recognize by this, that fulfilling all

desired properties is almost impossible with the current state-of-the-art. Current algorithms are only able to achieve some of these properties to a certain degree. An improvement can be made by combining multiple methods to form a new approach. Almost all properties, besides maybe speed, can be enhanced with such hybrid-methods. However, hybrid-methods also must try to find a good balance between these quality characteristics.

Furthermore, J. Togelius et al. provide a detailed taxonomy for procedural content generators [TYSB11]. This is useful to categorise or describe a content generator. Since this thesis wants to find and describe differences between level generators that either use a SO or a MO agent, this work could be useful. However, the used main approach is mostly the same here. Only a small component of the whole system will be changed. Therefore, the categorisation according to this taxonomy will (almost) be the same for both variants. Most other authors that propose new generators usually compared their approach with other methods from relevant literature. Unfortunately, most of the utilized comparison methods are specific for one use case.

3. Background

This section explains some essential background information that are important to understand this thesis. At first, an overview about the general functioning of an Evolutionary Algorithm is provided. After that, a detailed explanation of GGP is given. This section also gives some information about the differences between Single-Objective- and Multi-Objective-agents, as well as an introduction about their functionality. Further readings about specific implementations, which are later used for the experiment, are also provided for the interested reader. At last, the used framework for the implementation is presented.

3.1. Evolutionary Algorithm

An Evolutionary Algorithm is a metaheuristic optimization algorithm. It is loosely inspired by the natural evolution process. Have a look at Figure 3.1 to see the most important components.

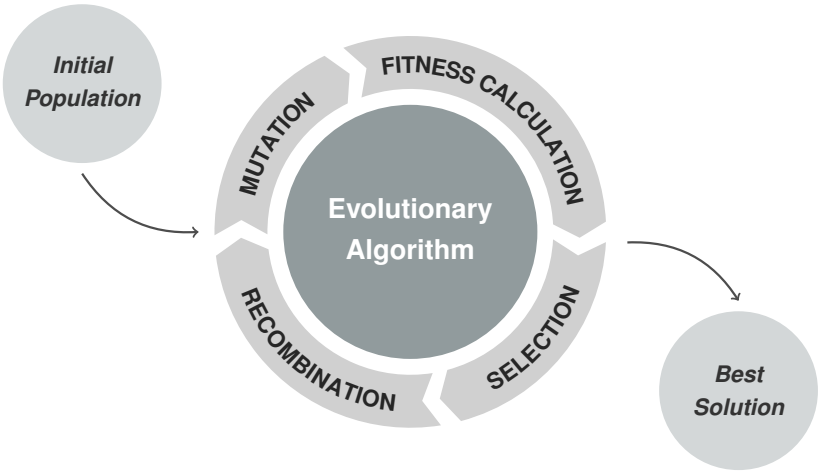


Figure 3.1.: Typical procedure of an evolutionary algorithm to solve an optimization problem.

Possible solutions are encoded in a chromosome and one particular chromosome belongs to an individual. Every individual has a fitness that indicates how useful or good this solution is. A set of individuals is called a *population*. A population undergoes various steps to generate a new generation with slightly altered individuals. Different breeding operations, like mutation and recombination, are responsible for these changes. A selection method chooses which individuals are used for the next generation.

The basic idea is to pick already good solutions and change them to some extent to find a marginally better variant. This method will find mostly only local maxima. Some mutations or

recombinations are more disturbing to better explore the search space. An advantage of an EA is the relatively simple implementation and its easy adaptability to all kinds of search problems. On the downside, there is no guarantee that this method will find the global best solution; but this is true for all metaheuristic algorithms and could only be avoided by reviewing each possible solution.

The influential difficult part for every EA implementation is the fitness function. The fitness function provides information how good or bad a solution candidate is. In most cases the fitness is represented by a single value. One important property of this function is, that a small change on an individual should also result only in a small change of the fitness value. There is no general function or formula; the concrete implementation depends highly on the specific problem. It is quite different for the breeding operations. The used methods here are oftentimes the same and do not require that many adjustments besides some simple parameter changes like the mutation rate. The same applies to the selection method, even if there are far more different variants available. The later experiment will use *Stochastic Universal Sampling (SUS)* [Bak87]. This method selects better individuals with a higher probability. Nevertheless, worse individuals have also a small chance to get selected. This ensures that the gene pool is sufficiently large.

Algorithm 3.1: Stochastic Universal Sampling

```
function SUS(population,N)
  Ω ← {}
  F ← fitness sum of population
  Δ ← F/N
  S ← random(0, P)
  k ← 0
  for i ← 1 to N−1 do
    P ← S+i·Δ
    while fitness sum of population[0..k] < P do
      k ← k+1
    Ω ← Ω∪{population[k]}
  return Ω
```

SUS is very similar to *Roulette Wheel Selection (RWS)*. Whereas RWS chooses multiple solutions by spinning a roulette wheel numerous times, SUS is more like a roulette wheel with multiple evenly spaced pointers that is only spun one time. Therefore, one wheel turn is enough to select multiple candidates. SUS is consequently more efficient. For further details have a look at Algorithm 3.1.

3.2. General Game Playing

Most game playing computer programs are build and trained for exactly one game or at least for one specific type of game. Such a computer program is often called a *bot* or *agent*. The best known examples are probably the world's best chess program *"Deep Blue"* from IBM [CHFh01] or the far more recent example of *"AlphaGo"* from Google [SHM+16] that is able to win against the best human Go players. This research area has a long track record and goes way back to the 50s [Dys12, 315].

In contrast, GGP goes one step further and tries to build AI systems that are able to play multiple different games [GLP05]. Not only that, but a general agent should also be able to play a game without any prior training on this specific game. Therefore, the used algorithm can not be trimmed to a specific set of game rules. They must be able to adapt to new environments and learn on-the-fly how to react to completely new input. The next sub section will give an overview about such agents and how they work.

These agents need two things to achieve all this: A computer readable formal game description and a way to automatically play these games according to the provided rules.

All of these things make GGP an interesting and very challenging interdisciplinary research area. The first concrete analyses are already from the 60s [Pit68] [Pit71]. Although, it really picked up only in the last few years.

Nowadays, there are even international competitions for GGP researchers and amateurs. They provide an incentive to implement novel ideas and to test their performance in comparison to other concepts. One of the oldest is *The International General Game Playing Competition* [1] organized by the Stanford Logic Group of the Stanford University since 2005 [GB13]. Another annually occurring event is the *General Video Game AI Competition (GVG-AI)*[2].

This thesis uses the *Software Development Kit (SDK)* provided by GVG-AI for the experimental part. Thus, a detailed explanation is given in the following section.

[1] `http://games.stanford.edu`
[2] `http://www.gvgai.net`

3.2.1. Agents

As already mentioned, a simulation-based approach is used to evaluate the fitness of a solution candidate. The simulation will use different kinds of agents that are able to play the generated levels. The fitness value is based on the outcome of the simulation.

The task of an agent is to determine a suitable action at any given time in the simulation. That means, the input for such an algorithm is the current state of the game and the output is the action that the game character executes. Seen in the long term, the outcome should be positive for the player. For most agents the positive outcome would be quite simple - winning the game. Although, in some cases winning alone is not good enough. Other objectives could also be relevant, like maximizing the game score, considering time constraints or collecting special items. Therefore, the agent would have more than one objective to fulfil at the same time. Nonetheless, the most common methods are using only one single value to define the quality of a solution. They use a heuristic to map all these different goals to one number. In spite of the fact that these agents have several objectives, their evaluation still uses only one objective – maximizing this value. Consequently, these agents are further called SO-agents.

A problem with this approach is, that sometimes goals conflict with each other. A better way to address this problem is to use a Multi-Objective method. This means that the optimization problem tries to find a well balanced solution for multiple variables at the same time. As recent research suggests, MO-agents have the potential to outperform their SO counterparts [PlML16].

The experiment in this thesis will use both kinds of agents. Two different SO-agents – *sampleMCTS* that just uses the MCTS algorithm and a more sophisticated one named *random42*. Additionally a MO-agent called *paretoMCTS* will be used. The main algorithms that these three agents utilize are described below.

i. Monte Carlo Tree Search

A common method for both approaches is to use a *Tree Search (TS)*. The TS technique iteratively builds up a *game tree*. Each node in the tree represents a game state and every edge stands for an action. The nodes can also hold more information, like game statistics. A terminal node (a leaf) expresses the end of a game and includes information whether the game has been won or lost. The overall number of all possible game states can be huge. For almost all non-trivial games it is either very difficult to evaluate each state or even impossible. Therefore, a guided tree traversal method is needed. Traditionally, method like *Minimax* or $\alpha - \beta$ search are often used for board or video games. More recently, Monte Carlo Tree Search has gained

some momentum, mostly due to its outstanding performance in the game *Go* [SHM+16].

There are multiple variants and modifications for *Monte Carlo Tree Search (MCTS)* available [BPW+12]. The basic idea is quite simple: There is not much information to gain from a single randomly played game, but it is possible to come up with a good strategy if you play multiple random games and gather some statistics. MCTS can be divided into four different steps as shown in Figure 3.2.

MCTS starts with a single root node and expands the tree iteratively. Edges in this tree are transitions from one game state to another. Each edges also holds information about which action was performed in this step. In every node of the tree, some statistics are saved that are continuously updated throughout the whole process. The statistics are how often an action was performed from a state ($N(s,a)$), the number of times each action was played from here ($N(s)$) and the win/loss ratio according to the outcome when a given action was played in a certain state ($Q(s,a)$).

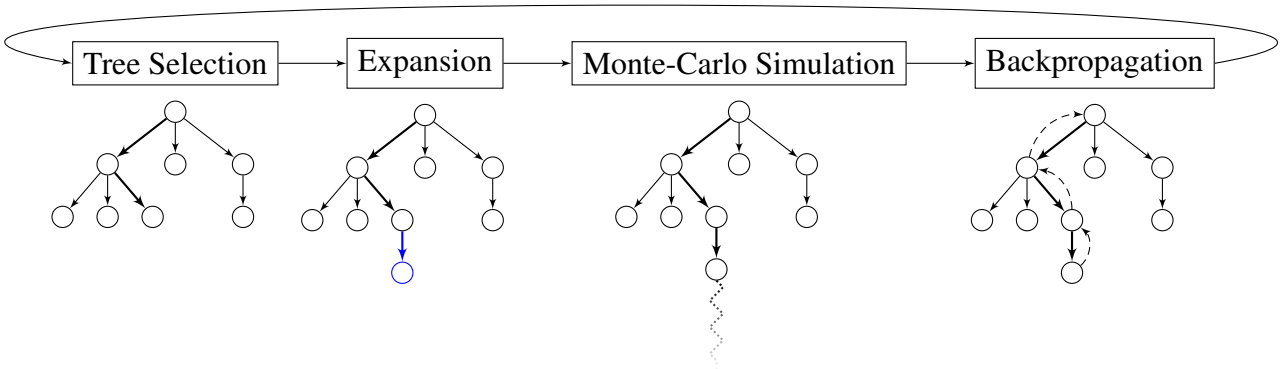


Figure 3.2.: The four steps of a Monte Carlo Tree Search.

Tree Selection A *Tree Policy* is used to navigate through the tree from the root up to the first node that has at least one unpicked action left (known as a non totally expanded node). The policy has the goal to find a good balance between exploitation (using actions that leaded to good results) and exploration (using less promising actions to mitigate the effect of late reward and simulation uncertainty).

Expansion Whenever the algorithm reaches a non-totally expanded node this state is added to the tree and the Monte-Carlo simulation (*roll-out*) starts. A roll-out is a sequence of action selections picked uniformly at random.

Monte-Carlo Simulation The rest of the game uses the *Default Policy* to select the actions. In the simplest case, all actions are weighted equally and a uniformly at random roll-out is played until the game terminates or a given depth is reached. It is often useful to use a heuristic to determine more promising actions instead of choosing them randomly.

Backpropagation As soon as the termination condition is reached, the backpropagation phase is performed. The statistics are all updated on each visited node.

All these steps are repeated until an end condition is met (e.g. a time constraint). The actual step for the real game is then chosen by the *Recommendation Policy*.

Each policy is interchangeable and can be adapted to the particular problem that the users want to solve. The most influential policy is the *Tree Policy*. Every action decision in MCTS needs to find a balance between exploitation and exploration. That means, the algorithm must choose between selecting actions that lead to better outcomes (as far as known at this moment) and selecting actions that it has not explored yet. A possible solution for this dilemma was proposed by L .Kocsis C. Szepesvári [KSW06]. They demonstrated the usefulness of using *Upper Confidence Bound (UCB1)*. The equation to select a new action with UCB1 is:

$$a^* = \underset{a \in A(s)}{\arg\max} \left\{ Q(s,a) + C\sqrt{\frac{\ln N(s)}{N(s,a)}} \right\}$$

The first part $Q(s,a)$ is for exploitation and the right side represents exploration. The constant C is used to weigh both terms.

ii. Pareto MCTS

The goal of all optimization algorithms is to find a feasible non-dominated solution, i.e. a solution were no single objective could be further improved without degrading the value of another objective. Figure 3.3 illustrates this and also shows an example Pareto front. In most cases, there are multiple non-dominated solutions. Single-Objective algorithms often try to find a balance between the different objectives by weighing them somehow and adding them up to a single value. This approach does not work properly for non-convex Pareto fronts.

A possible solution was developed by D. Perez-Liebana et al. [DPL15] and is called The *Pareto MCTS*. They directly use the pareto front to find better solution candidates. The same author also benchmarked his implementation and demonstrated that this algorithm is applicable in the context of GGP [PlML16].

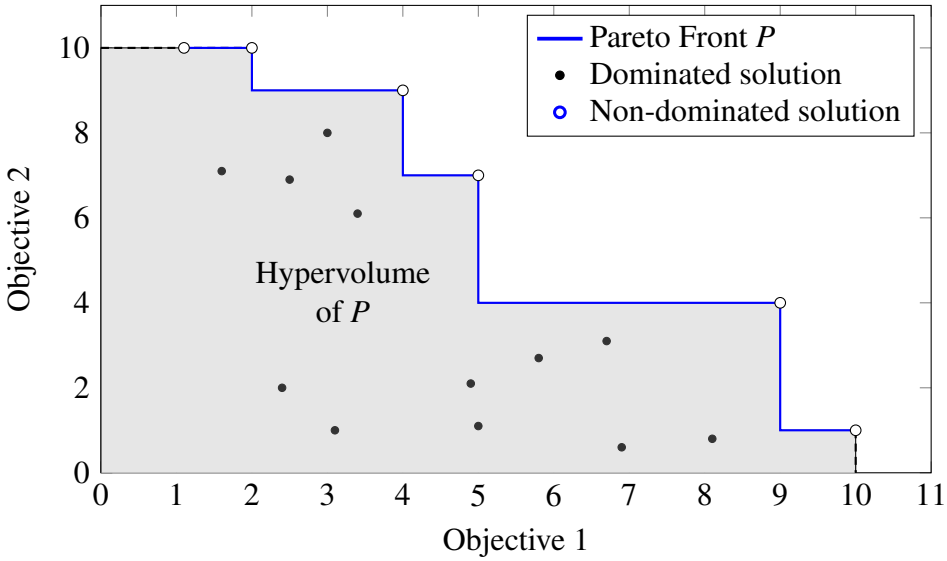


Figure 3.3.: Example Pareto front P for solutions with two different objectives.

Instead of using just one fitness value, this variant uses a vector $\overline{r}$ to represent m values for each of the m objectives. $Q(s,a)$ now contains such a vector to store the average award of each objective. Instead of updating a single value, an approximation of the Pareto front is saved in each node. Since each node has information about the Pareto front, the quality of the reachable states starting from this point can be estimated. The backpropagation updates the Pareto front by either adding a new non-dominated solution and removing all newly dominated ones or by ignoring the solution if it is dominated by an already containing one. The second case can actually also be used to stop the backpropagation at this child since all parents contain only better or equally good solutions. For the same reason, the root node automatically contains the overall best solution that was ever found in this search.

At last, the UCB1 equation must be adapted to use the new reward vector. The *Hypervolume (HV)* of the stored Pareto from each node can be used as a quality measurement. HV is just the objective space of a Pareto front. Figure 3.3 also shows an example HV. The HV can be used to compare different sets of non-dominated solutions. Thus, $HV(P)/N(s)$ can be used as the exploitation term, resulting in the new equation:

$$a^* = \arg\max_{a \in A(s)} \left\{ \frac{HV(P)}{N(s)} + C\sqrt{\frac{\ln N(s)}{N(s,a)}} \right\}$$

The experiment will use this algorithm to optimize two objectives. $Objective_1$ is to maximize the score. The heuristic uses the game score in conjunction with a bonus for winning the game and a penalty for losing. The *sampleMCTS* controller utilize the same heuristic. $Objective_2$ tries to maximize the exploration of the map. The heuristic function is nature-inspired and uses

a kind of *pheromone* diffusion map. Further details can be found in [PDH+15].

iii. Return42

The last used agent is *Return42*. This agent has a high rank in the GVG-AI competition and was created by Tobias Welther, Oliver Welther, Frederik Buss-Joraschek and Stefan Hbecker. The agent uses a hyperheuristic. That means, it combines different heuristic functions and selects one of them according to certain game characteristics. If a game is deterministic, it uses an A^*-algorithm to select actions that lead to a higher score or (possible) wins. Random walks with a handmade heuristic will be used if a game is stochastic. This heuristic takes the score, resources and NPCs into account.

3.2.2. GVG-AI

The GVG-AI project provides two valuable things for researchers around the world. First, a platform to show their current work and to prove or disprove its efficiency. Due to the relatively low entry barrier and the good visualisation, this is also approachable to newcomers, especially young students. It also greatly helps to introduce them to the wide research area of AI.

On the other hand, the GVG-AI framework itself is a great starting point for all kinds of different AI related experimentations. For example, although that this work is not directly about GGP, it nevertheless needs the same building blocks for the evaluation part. The provided framework is an incredibly helpful tool for this.

The framework is written in Java. Its main purpose is to simulate games that are written in *Video Game Description Language (VGDL)* and to provide an interface to write agents that can play these games. It even has an optional *Graphical User Interface (GUI)* to give more and easier to understand feedback to the developer. For testing purpose, it is also possible that a human instead of a computer program plays these games. More recently it also has a native interface built in to directly write level generators for VGDL.

The advantage of using this framework is, that, for one, VGDL is a fairly known language in the scene and that the parsing of this language is already implemented. Hence, the level generator has direct access to a high level abstraction to all relevant game information in form of a Java API. Another big advantage are the included finished game descriptions. The current version provides 72 usable single player games. This will greatly help evaluate the later presented generator with sufficiently different game designs.

3.2.3. Video Game Description Language

Video Game Description Language (VGDL) is a domain specific markup language to define two dimensional arcade-like games. It was originally developed by Tom Schaul [Sch13]. The language itself is very compact and allows to define a wide range of different games with just a few lines of code. Have a look at the example description for the game *Sokoban* at Listing 3.1. The whole game is defined in just 15 short lines. The details of the shown VGDL example are not that important here. Thus, the following paragraphs will only briefly describe the basic structure of the language.

Listing 3.1: VGDL description of the game *Sokoban*

```
1  BasicGame key_handler=Pulse square_size=50
2      SpriteSet
3          hole   > Immovable color=DARKBLUE img=hole
4          avatar > MovingAvatar
5          box    > Passive img=box
6      LevelMapping
7          0 > hole
8          1 > box
9      InteractionSet
10         avatar wall  > stepBack
11         box avatar   > bounceForward
12         box wall box > undoAll
13         box hole     > killSprite scoreChange=1
14     TerminationSet
15         SpriteCounter stype=box limit=0 win=True
```

Each VGDL game definition is divided into four sections – *SpriteSet*, *LevelMapping*, *InteractionSet* and *TerminationSet*.

The *SpriteSet* part describes all properties of each sprite. A Sprite is, in this case, a single object in a game world. It could be a background tile, an enemy, an obstacle or something useful to pickup. The most important sprite type that is always available is the *avatar*. This is the player-controlled figure in a game. *Sokoban* has two more sprite types, *hole* and *box*. In addition to that, the *wall* sprite is also always implicitly defined.

The *InteractionSet* section describes what happens when two sprites collide. The shown example defines, amongst other things, that the player can move boxes around by colliding with them and that putting a box into a hole will destroy the box and raise the players score by one point.

The *TerminationSet* defines the winning and losing conditions. In this case the player would win if there are no more boxes left. At this point it is worth mentioning, that there is always an implicit losing rule in the implementation of GVG-AI. If the agent could not win the game after

a certain amount of steps, he will simply lose the game. The competition uses a maximum of 2000 steps. Furthermore, if an agent exceeds the maximum allowed computation time per step, he also gets disqualified and loses this round. The competition limit is set to just 40ms.

One last thing is missing to actually play a game like *Sokoban*: A level layout. A level is just a description of the size of the map and all initial positions of the different sprites. This is the reason for the *LevelMapping* section. It is used to define another text file, like the one shown on the left side of Figure 3.4.

```
w w w w w w w w w w w w w
w w         w                 w w
w w     w 0 1     w w     1 w w
w w         1                 w w
w w w w w w 0                 w w
w w w                         w w
w w w       w       w     A w w
w w w       w       w     0 w w
w w w w w w w w w w w w w
```

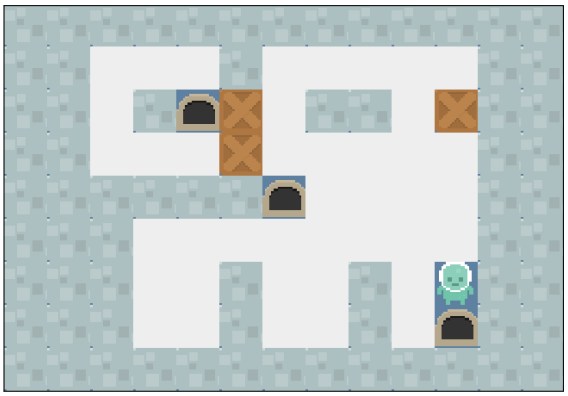


Figure 3.4.: A text-based description of a level for *Sokoban* is shown on the left side. An image of the same level rendered with the GVG-AI GUI is displayed right.

The level description file has one letter at each position in the map. The letters are mapped to sprites according to the definition from the *LevelMapping* section. One letter could also stand for multiple sprites at once. Have a look at the right side of Figure 3.4 to see how the text file would look like in an graphical representation. The letters *A* (*avatar*) and *w* (*wall*) need not to be defined explicitly. They are predefined for each game and can therefore always be used.

4. Procedural Level Generation

After having introduced all necessary background information, this chapter will describe any algorithm to generate game levels. However, just before that, the first section will present some restrictions and their justification. Right after that, an overview of the general procedure methods will be given. This should help to better understand the latter details from the section below.

4.1. Limitations

In regards to the usage case in this thesis, some restrictions to our algorithm can be formulated beforehand. First, because the context of this work is GGP, no game specific rules should be used. Although it would be totally okay to use such rules while generating levels, it would make adjustments in the future much more difficult. Developing and using new game descriptions in GGP is very common, oftentimes even necessary. A complex adjustment should be avoided.

Also, prior knowledge of the games, besides their description, should not be used. For example, the algorithm could use human made levels as a kind of template. Again, it would be fine to do so, but in regards to simplicity, external dependencies should be used as little as possible. Another reason to avoid this is to circumvent bias. Oftentimes, human already have a concrete idea of how a level should look like – in their opinion; mostly based on prior experiences. A computer program with little to no given human background information will be able to find completely new solutions. Solutions no human ever thought about. A solution that nevertheless works according to all given rules, but looks or feels totally different than anything from before. It could then be used to inspire human designers. This is not really the focus of this work, but it could nevertheless be adapted for this usage scenario.

The last restriction is not to use human interaction. A typical job for a person could be to rate all generated levels. But this is highly subjective and would therefore require a lot of manual work from different persons. Using a computer algorithm to solve all aspects of level generation is, in this case, more straightforward.

4.2. Algorithm Overview

The level generator consists of three main stages. The first one is an one-time operation. In this stage, all important information from the game description is parsed. The actual procedure is explained below in Section 4.3.

After having basic information about the game, an indirect level representation gets created. A new kind of heuristic is used here to guide the later explained placing algorithm. Further explanation can be found in Section 4.4. The heuristic data and the game information from the previous stage together form a *level descriptor* – an indirect representation of a level. In regards to the later used EA, it is also called the *genotype*. With the help of this descriptor one or more actual levels can be instantiated. To create such level instance, another algorithm is needed. This one is explained in Section 4.5.

The last stage uses a metaheuristic to find suitable levels. This method tries to find usable level descriptors by evaluating its generated level instances. This is an iterative improvement process and takes the most calculation time. The end result will be a functional level for the given game, optimized with regards to the chosen parameters.

4.3. Retrieve game information

At first we need to gain some general knowledge of the game. Since we do not want to use any other input but the VGDL we have to parse the description and try to deduce as much data as possible. The GVG-AI framework already has an implementation to read in most information and provides them as Java objects. Information about which types of sprites exist, the interactions among each other and the necessary conditions to win or lose a game is always included. In some cases, depending on the game rules, even more information is available, like the lifetime of sprites, speed or their movement direction.

There is other information that is only available indirectly. Some games have a background sprite, a floor-like tile or a tile to represent space, for example. To find out if there is such a sprite, we use the level mapping data from VGDL. The level mapping describes which sprites we can set on the map and, more importantly, how we can stack different sprites. The hypothesis here is, that if there is a sprite type that can be placed together with every other sprite type, then this must be a background sprite.

Most games are surrounded by a wall, but that is not always true. We can deduce this property indirectly from the game description. Namely, if the avatar sprite has no interaction with the border of the map (called *EOS* in VGDL) he would be able to run "outside" of the map. Leaving

the level and going, for example, to a negative position should not be possible. Therefore, we need some other obstacle to prevent this from happening. And this is always a wall.

Another important property of the VGDL are *subtypes*. That means that the same sprite can have different characteristics. Only one subtype is active at any given time. Due to some in-game effects, a sprite could transform from one subtype to another. This information is also already parsed by the framework, but it is necessary to take this into account while the algorithm looks at the properties of each sprite. Then there are also implicit information and assumptions. For example, since we only evaluate single player games, we always need to set exactly one avatar.

Unfortunately the game description tells us nothing about maximum or minimum size of a level. Evaluating big levels will take a lot of time. Therefore, the size of the generated levels must be restricted. There are 365 example levels provided by the GVG-AI framework. From these human generated levels we can get an idea of typical level sizes. The width of all given examples varies between 5 and 50. The height is between only 2 and 36. These parameters are used here to define the size boundaries.

Although this is already a lot of information, there are still some knowledge gaps. It is in no way enough to generate a whole new level. For example there is no clue about the actual number of sprites in a game description or where we have to place each object. The reason is simple, there is not one true number for this. Every level could be totally different. These missing numbers are our parameters. The next task is therefore to find a good combination of all the missing values in such a way that the result is a good and useful level.

4.4. Likeliness-Matrices

After having all necessary basic information at hand, we need to decide how to place a sprite on a map. One possible solution would be to put down sprites at random positions. Do this thousands of times and with some luck one would find a few good levels. This would be very inefficient and time consuming. Therefore we need to guide our placing algorithm.

The solution here is to use a *Likeliness-Matrix* $M_{t,i}[x,y] \in [0,1]$. This matrix describes the likeliness to place the sprite of type t on position x,y on iteration i. An example matrix can be seen on the left side in Figure 4.1. There is one value for each position on the game level. For this reason the dimension of the matrix is the same as the level size that we want to generate. The values are between 0 (impossible to set sprite t here) and 1 (placing the sprite here would probably be very good). It is important to realize that, although the values look like typical probability values, they are not. Even a 1.0 does not mean that the algorithm will definitely

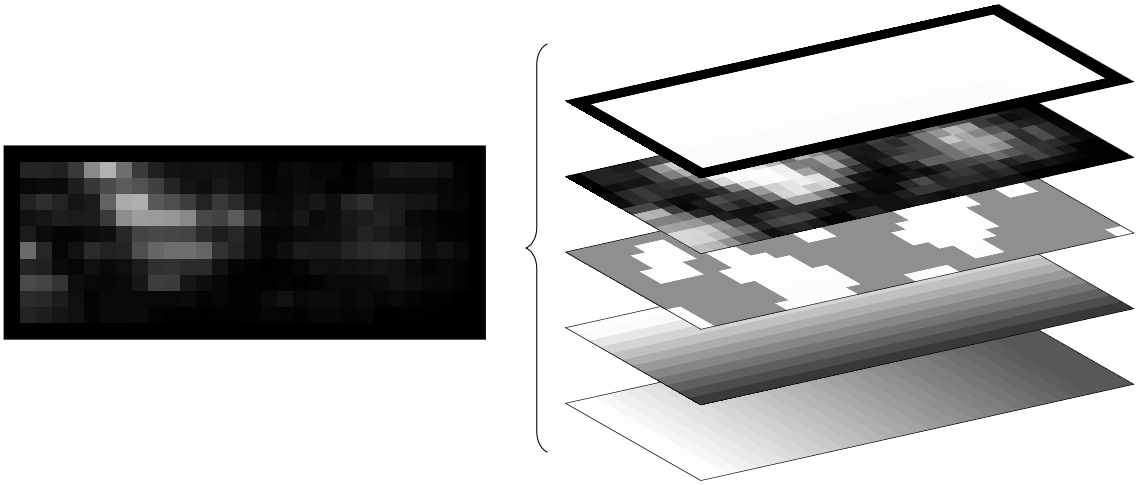


Figure 4.1.: Left: Example 30x12 Likeliness-Matrix
Right: Different sub-matrices that are used to compose the left matrix
(matrices shown as grayscale image; 0 to 1 are mapped between black and white)

place a sprite on this position. It also does not mean that this position is actually a good position, only that it is probably a good one.

The matrix itself is a composition of different matrices as shown in Figure 4.1. All these sub-matrices are multiplied element-wise to create the final matrix M. The sub-matrices have different parameters that are encoded in the chromosome of an individual (see section 4.6). The following subsections will describe all sub-matrices in detail.

4.4.1. Placeable-Matrix

The first one has actually just binary values (0 or 1). It describes if this type of sprite is placeable on the corresponding position on the level at all. For example, for most games we can only place the avatar sprite on a free position of the map. If the map is already surrounded by *walls* (a very common characteristic for a lot of games) a matrix could look like the one shown in Figure 4.2.

Figure 4.2.: Example 30x12 Placeable-Matrix. Black values mean 0 and white 1.

This matrix is important, because in some cases specific sprites can be stacked. The stacking combinations are well defined in the VGDL.

4.4.2. Cluster-Matrix

The next matrix is the Cluster-Matrix. In contrast to the previous one, this one has a parameter. This parameter defines how clustered (or chaotic) the values are distributed and therefore determines the distribution of a sprite among the level.

The idea behind this matrix is, that for the same games, the same sprites are, more or less, evenly spread throughout the whole map and others are more concentrated to only a few places. An example is shown in Figure 4.3. This matrix has approximately three clusters. It is therefore more likely that the associated sprite is placed within these clusters.

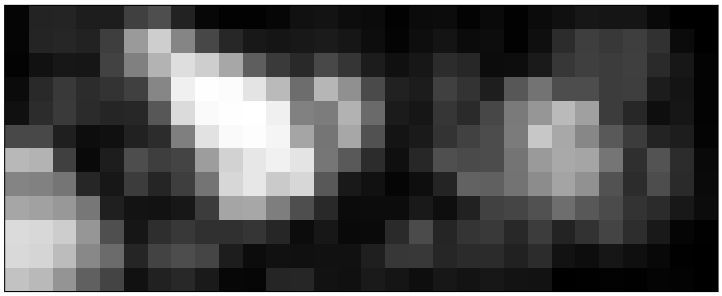


Figure 4.3.: Example 30x12 Cluster-matrix with roughly three clusters.

The data is generated by an OpenSimplex noise generator [Spe14]. The parameter that was already mentioned describes a stretch factor. The value roughly correlates with the number of clusters that the generator creates. For example, a value of 1 means that there is only one big cluster. The matrix values would be evenly distributed. A value of 5 would generate around five randomly positioned clusters.

In this case the 3-dimensional version is used. Instead of having only 2D clusters, the algorithm actually generates 3D clusters. However, the actual matrix is only one slice from the 3D data. The z-axis is determined by an additional second parameter. Small changes of this parameter will only alter the resulting matrix minimally. This parameter is particularly useful for the EA that is later used to generate the level (see Section 4.6).

4.4.3. Pattern-Matrix

The Pattern-Matrix has a similar objective as the Cluster-Matrix – to structure the placement of a sprite in a particular way. Objects in human generated levels are often arranged in some

specific patterns. This matrix tries to archive exactly this behaviour.

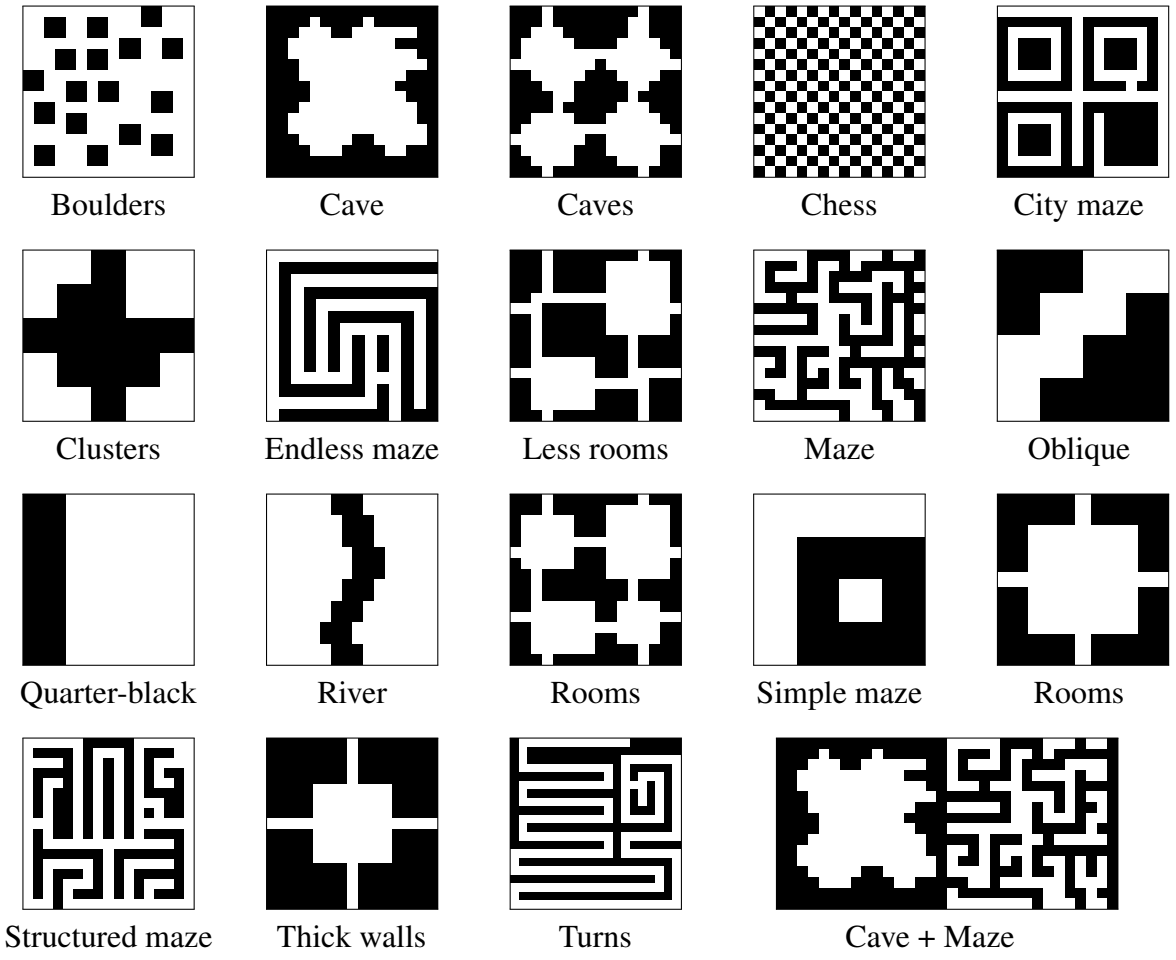


Figure 4.4.: 19 different sample patterns used to generate the Pattern-matrix.

At first we need some predefined patterns that a human would also use. An overview of all used patterns is shown in Figure 4.4. The idea for using this was inspired from [Cha16]. These patterns act as samples to generate patterns big enough to cover the whole map. The algorithm used to archive this is called ConvChain [Mxg16]. It is based on a Markov chain Monte Carlo simulation.

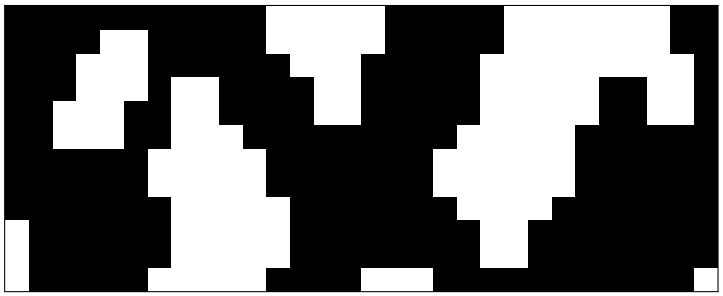


Figure 4.5.: Example Pattern-matrix using the "cave" sample.

An example is shown in Figure 4.5. This matrix was created from the "maze" sample pattern. The actual likeliness of all non-pattern values is determined by another parameter. That means that it is still possible to set sprites outside the pattern structure. This parameter is also subject to the EA algorithm. A smaller parameter value would enforce a distribution according to the pattern more than a higher value.

4.4.4. Constraint-Matrix

The Constraint-Matrix actually describes a whole series of another matrices. Some sprite types have specific properties from which we can deduce placing constraints. These constrains are more like a hypothesis than a given truth, because of the ambiguous nature of such implicit deduced rules.

An example would be a moving sprite with a fixed direction. Imagine the sprite can only move downwards. A logical conclusion would be that the sprite needs space to move into this direction and should therefore be placed on top of the level. Obviously, only placing this sprite on top is not a good solution, but nevertheless the preferred position to place this sprite should at least be somewhere in the top region.

In this case we go even one step further and try to place a sprite as far away from its next collision point as possible. Have a look at Figure 4.6 for an example. Here we have some obstacles (e.g. a wall) in the center of the level. The sprite we want to place can only move downwards and would collide with this obstacle. The shown matrix would ensure that we would most likely not place the sprite directly in front of an obstacle.

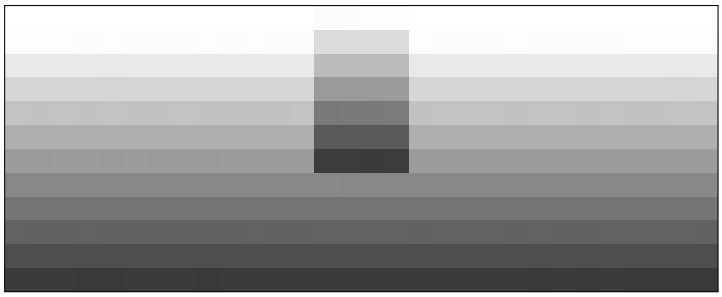


Figure 4.6.: Example Constraint-Matrix for the directional constraint "down" and some obstacles in the middle of the level.

Another interesting implicit information which we can represent with this kind of matrix is the "free track" constraint. This means that there should be no non-moveable, non-transformable, non-destroyable objects in the path of a movable sprite that has no defined effect on this object. Otherwise this sprite would eventually collide with the other one and the result would be undefined.

One important thing here is that these constrains can be inherited. For example, imagine a game with some kind of portal that spawns sprites of type t_1. These sprites have a fixed direction, they can only move to the right side. Therefore, they would have a "right"-constraint. But since they get spawned by another portal sprite, we have to hand this information down to this sprite. Multiple inheritances are also possible. Consequently, a single sprite type can have multiple directional constraints at once. Conflicting ones are cancelled out.

4.5. Genotype-Phenotype Mapping

Until now, we only have a vector of numbers (the chromosome) which describes some basic properties of a level. This vector is often called *Genotype*. An actual level file like the example from Section 3.2.3 is called *Phenotype*. There must be a method to transform this indirect representation to a concrete level description. In this case, it means there is an algorithm needed that determines how to place the sprite on a map according to the map size, quantities and game rules. Additionally the just described Likeliness-Matrices will be used. Therefore, the mapping function is basically a placing algorithm.

The used method works iteratively and begins with an empty map with a size given by the chromosome. Furthermore, there is a set that contains all sprites and their corresponding desired quantity. Again, this information is directly decoded from the individual's chromosome.

In each step, for each remaining sprite in the set, the Likeliness-Matrices are updated. The algorithm then finds a position, where, according to the Likeliness-Matrices from each sprite, the difference between the most likely and the second most likely is maximized. The most likely sprite on this selected position is then placed there. The equation to select a position is therefore:

$$\arg\max_{x,y}(max(\{M_t[x,y] : t \in T\}) - max_2(\{M_t[x,y] : t \in T\})$$

where *max* returns the maximum value of a given set and max_2 the second largest value.

It is important to realize that this algorithm is completely deterministic up until now. Multiple runs will output the exact same phenotype. However, the most important component of the placing algorithm, the Likeliness-Matrices, is not able to provide a definitive prediction of a good placement of a sprite. Its imprecise nature should be taken into account. Therefore, noise is added to the final matrix. In this case, a 10% range of variation is allowed. This means, if the second highest likeliness on a position is less than 10% lower than the absolute maximum, then there is an chance that this value is chosen as the highest instead. This makes the output stochastic. Every run will now yield in a slightly different level.

Noteworthy is, that it can happen that the given number of sprites can not be placed down fully. In this situation the remaining sprites will just be ignored. In the worst case, it can even happen that there is no avatar in the generated level. The next section will describe the EA algorithm that will be used to screen out this kind of levels.

4.6. Evolutionary Algorithm

As already mentioned in the previous section, there are several different parameters for which we need to find good values to actually generate meaningful levels. Each chromosome encodes the following information:

- level width $\in [5 \dots 50]$
- level height $\in [2 \dots 36]$
- mutation variance $\in [1.0 \dots 10.0]$

  for each sprite type:
  - quantity $\in \mathbb{N}$
  - pattern $\in [0 \dots 31]$
  - pattern receptor size $\in [1 \dots 4]$
  - pattern weight $\in [0.0 \dots 1.0]$
  - pattern temperature $\in [0.0 \dots 2.0]$
  - cluster $\in [0.0 \dots 10.0]$
  - cluster noise $\in [0.0 \dots 1.0]$

Consequently, the search space is huge and it is not possible to try every single combination. An EA is used to cover the search space as thoroughly as possible. To be more precisely, in this case a *Genetic Algorithm (GA)* is utilised [Hol92]. Thus, initializing a population, selection and both breeding operators (mutation and recombination) are more or less what you would find in standard literature [SP94]. The following subsections will explain every component in detail.

4.6.1. Initial Population

A uniform random number generator is used to generate an initial population. Every parameter gets initialised with a random value. The range of the values has to take all restrictions into account.

For some games and some parameters there are actually no restrictions given. The quantity of a sprite is such a parameter. One way to solve this is to use an approximation. The absolute maximum number of one sprite type equals the area of a level. But unfortunately, due to the stacking possibility of sprites, this is not the maximum number of all combined sprites. The quantity gets determined individually for each sprite in succession. For the first sprite, the upper limit is the map area. For the next sprites, it is the map area minus the occupied space. The occupied space is only an approximation. Not stackable sprites occupy exactly one tile. Stackable sprites, however, sometimes consume no additional tile. Here we approximate the occupied space with the quantity divided by the number of different stackable combinations for this sprite. So that the same sprite type is not always able to get a high quantity number, the order of the sprites is always random. Due to the approximation, it is possible that the sum of all sprites is higher than what is really feasible. This means that it is not always possible to place the given number of sprites. In the end, this is not a problem at all. The algorithm explained before in Section 4.5 just ignores the surplus.

4.6.2. Breeding Operations

Finding new solutions is an important aspect of every EA. One possibility is just to add random new individuals to the population. Although this is an adequate way to find totally new solutions in a completely new area of the search space, we also need a more fine granular evolutionary method. Hence, two different operations were used here to generate new individuals.

The first one is a simple one-point crossover. Two random, but different individuals are selected to create a new one. Since the chromosome consists of parameters grouped by each sprite type, the cutting point is between the two randomly selected sprites. Therefore, the new individual has the parameters from some sprites from the first parent and the rest from the second.

Crossover alone is a very limited way to create new solution candidates. There are simply not that many possible crossover points. To increase the diversity, a mutation operation is needed. A mutation is an addition of a normally distributed value to a random selected parameter. The variance of the normal distribution is adaptable. It is not trivial to find a good variance value. If it is too high, then the generated individuals would be too different from their original. The consequence would be, that the offspring could have a way worse fitness. A too low variance will have almost no effect on the fitness. For this reason, the variance itself is a parameter which is influenced by the EA.

4.6.3. Fitness Function

The last ingredient for a working EA is a fitness function. A fitness function defines how well an individual performs. In this case, it means that it measures how good our generated level is. Finding a proper definition of what makes a level a good one is complicated. The definition will always be incomplete and subjective. Yet, an approximation is good enough for the fitness function.

The fitness function simulates the game with a given level and draws conclusions from this. The simulation uses different skilled agents – computer programs that are able to play the game. Three fairly simple agents, one that does only random actions, one that looks just one step into the future and the *sampleMCTS* that is provided by the GVG-AI framework. And then, there will be one more advanced agent depending on the experiment (see Section 5.1 for more information).

Figure 4.7.: Pyramid of desirable qualities of a game level.
Most important and easier to accomplish qualities are at the bottom; every step upwards is harder to implement, but (most times) also less important.

There are at least three immediately available attributes that are usable to create a fitness function – the score of each agent, the number of steps they needed and if they won or lose. To define the fitness function, have a look at Figure 4.7. It is loosely based on the works of Togelius and Hartzen [TH12, 2]. This pyramid shows some desirable properties that a level should have. At the bottom is the most important feature a good level should always have. With every step upwards, the property gets less important and at the same time harder to achieve. This pyramid is helpful to actually understand the objective behind every part of the fitness function. Additionally, the whole fitness function can be seen as pseudocode in Algorithm 4.1. As shown in line 1, three different agents are used to compute the fitness value. The *goodAgent* is one of the three previously explained agents (see Section 3.2.1). For the *naiveAgent* the *sampleonesteplookahead*-Agent from the GVG-AI framework was chosen. The *randomAgent*

selects just a random action in each step. The simulation is done in line 3-5 and the results are stored in corresponding objects. Each agent is run multiple times to prevent outliers due to the stochastic nature of some games. More details about this can be found in Section 5.1.

The most important attribute is *playable*. If a level does not follow the game rules, then it is practically useless. Therefore, if it is not playable, the fitness value will be zero.

A game should also be winnable. Nobody would even bother to play a game, if he can not win it. That means, if one of the agents is able to win the game, then the level is obviously winnable and the fitness value should be raised (line 7). Note that, if no agent can win the game, it does not really mean that nobody can win it. Maybe a human or a better algorithm could win. These false negatives are mostly acceptable, since this is only an approximation.

Next, the level should be challenging. A too easy level would be uninteresting and dull, whereas a too hard level could lead to frustration. The main idea here is, that the simple agents should not be able to win the game or to get a high score, but the advanced ones should. If a simple agent is able to win the game anyway, then half of the winnable bonus will be subtracted (line 10). However, winning a game after just a few steps is also not very challenging. Therefore, the number of steps should also be taken into account. Needing too few steps is bad. But in regards to the next property, *balance*, a game should not need the maximum number of allowed steps (2000). Somewhere in-between would be optimal. To achieve this, a very simple formula is used that peaks at 1500 (¾) steps: $1500 - abs(steps - 1500)$ (line 8). The normalized average number of steps of all simple agents are subtracted from the fitness value (line 11). The longer the bad agents survive, the easier the level.

Algorithm 4.1: Pseudocode of the used fitness function

```
1  function fitness(goodAgent, naiveAgent, randomAgent)
2    runs ← 5
3    resultGood ← run(goodAgent, runs)
4    resultNaive ← run(naiveAgent, runs)
5    resultRandom ← run(randomAgent, runs)
6
7    ω ← resultGood.averageWon * feasibleBonus
8       + (targetSteps - abs(resultGood.steps - targetSteps) / runs)
9       + resultGood.score
10      - (resultNaive.won + resultRandom.won) / 2 * feasibleBonus * 0.5
11      - (resultNaive.steps + resultRandom.steps) / 2) / runs
12      - (resultNaive.score + resultRandom.score) / 2
13
14   return ω
```

The score is the last element to define the fitness. Reaching a higher score is, at first glance, better. Simply increasing the fitness value according to the collected score will not work. Tests have shown that in this case larger levels will always be preferred by the EA. The reason

is simple: larger levels can contain more sprites which can increase the score. This method would therefore only create large levels, mostly full of resources. To still use the score, a normalisation has to be done. The overall score of an agent is simply divided by the game area to make the value independent of the map size. This value from the good agent is added to the fitness (line 9). The average values of the simple agents are subtracted (line 12).

The last element of our pyramid, fun, is very hard to define. It is mostly subjective and therefore difficult to evaluate with a computer program. Since, in our case, the levels are only played by computers and are not explicitly designed for human, this property is not that important and is left out here.

5. Experiment

The task for this thesis was to generate new levels for arbitrary games with the help of different kinds of agents and try to identify distinct structures, design elements or other properties. An algorithm was already proposed to generate levels. What is missing is *a)* an experiment to proof that this algorithm works and *b)* to find the differences between the agent variants. Therefore, the final experiment is also two-fold. The first task is to generate levels with SO and MO agents. The second part is presenting the results and compare both level sets.

However, before the results can be presented, the setup must be explained first. Thus, the first section will show which parameters were used and why they were chosen. After that, some technical details about the evaluation are given, especially with a focus on how to do this in an efficient manner by using a distributed concept.

5.1. Parameterization

Since the goal of the final analysis is to find differences between levels generated with a SO-agent and levels created with the help of MO-agents, we need at least two agents of each kind.

The choice of usable MO-agents is very limited. One such agent was developed by D. Perez-Liebana [PlML16] and uses a modified version of MCTS [BPW$^+$12]. This agent will be compared with the SO *sampleMCTS*-agent that is provided by the GVG-AI framework. Since one can assume that the MO variant will perform better, another more complex SO-agent was chosen – called *Return42*.

5.1.1. Fitness Function

Another parameter for the fitness function is the number of needed simulation passes to get a stable and reliable fitness value. Since the games are not deterministic, it can happen that playing the exact same level with the same agent results in different outcomes. One way to mitigate this, is to play each level multiple times. “The more, the better” should also apply here, – at least in regards to the accuracy. However, simulating agents takes a very long time and is therefore a limiting factor. A small experiment is needed to find a compromise between accuracy and calculation time.

Intuitively, using the average of all runs should smooth the result enough to be reliable. To test this hypothesis, the exact same level is played multiple times. Then, the variance between all fitness values should be sufficient to see how well this solution works. In practice, ten individuals were randomly generated for four different games. Using different games is to assure that the results are not tailored to one specific game. The ten individuals are from a 20 generation long evaluation. The parameters for this are not of any interest here. The EA is only used to ensure that the tested individuals have a higher fitness than zero. Each of these ten individuals were then evaluated 100 times with four different simulation passes. This is done for all four games. Overall, the fitness of 16,000 individuals was computed.

Have a look at Figure 5.1 to see an example result from the game *aliens* and a 2-pass simulation. Every individual has another color. The markers are the fitness values – 100 per individual. Additionally, a box plot is displayed to easily see the median, the lower (0.25) and upper (0.75) quartile, as well as the lower [1] and upper whiskers [2] [RM78]. The actual values should be perceived with great care. They depend on a lot of other factors, like the used agent or the speed of the used computers. Still, it is sufficient as a comparison.

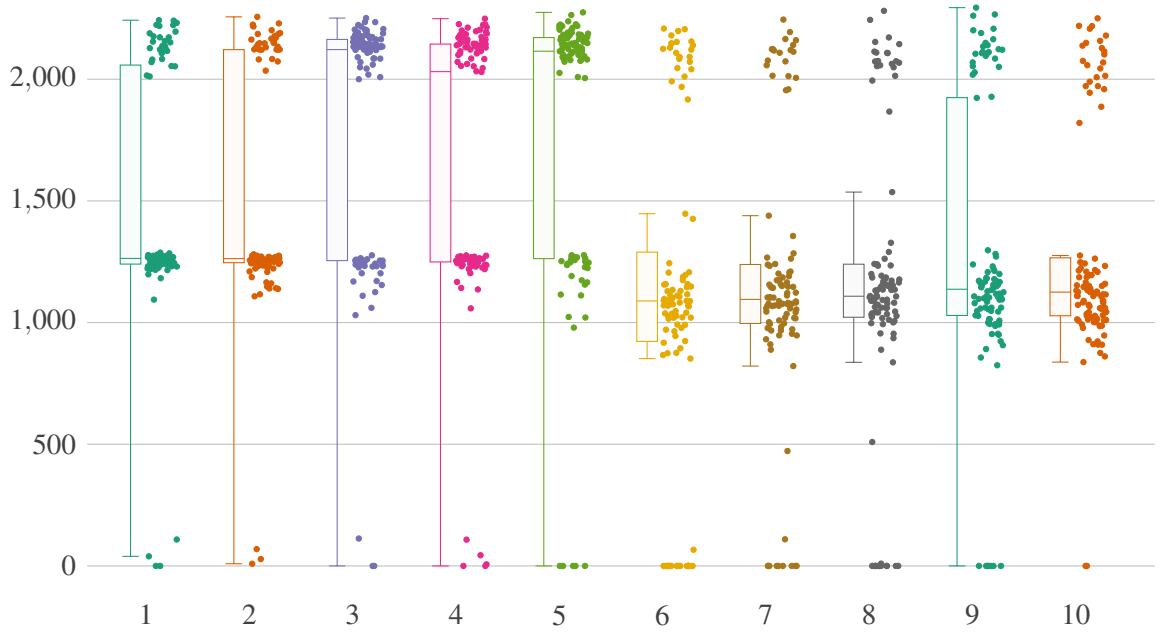


Figure 5.1.: Plotted fitness values of ten different individuals from the game *Aliens* played 100 times with two simulation passes.

As you can see, the difference in fitness is disappointingly high. The big gap between 1200 and 2000 is due to the *feasible*-bonus points (see Section 4.6.3). You can compare the results

[1] smallest value which is larger than $lowerquartile - 1.5 \cdot IQR$, where IQR is the difference between the upper quartile and lower quartile

[2] the largest value which is smaller than $upperquartile + 1.5 \cdot IQR$

for different numbers of simulations in Figure 5.2. Details about the other tested games can be found in Appendix B.1.1. It becomes immediately evident that running the simulation only one time is very unreliable and practical useless, whereas a higher number of runs helps to reduce the variance.

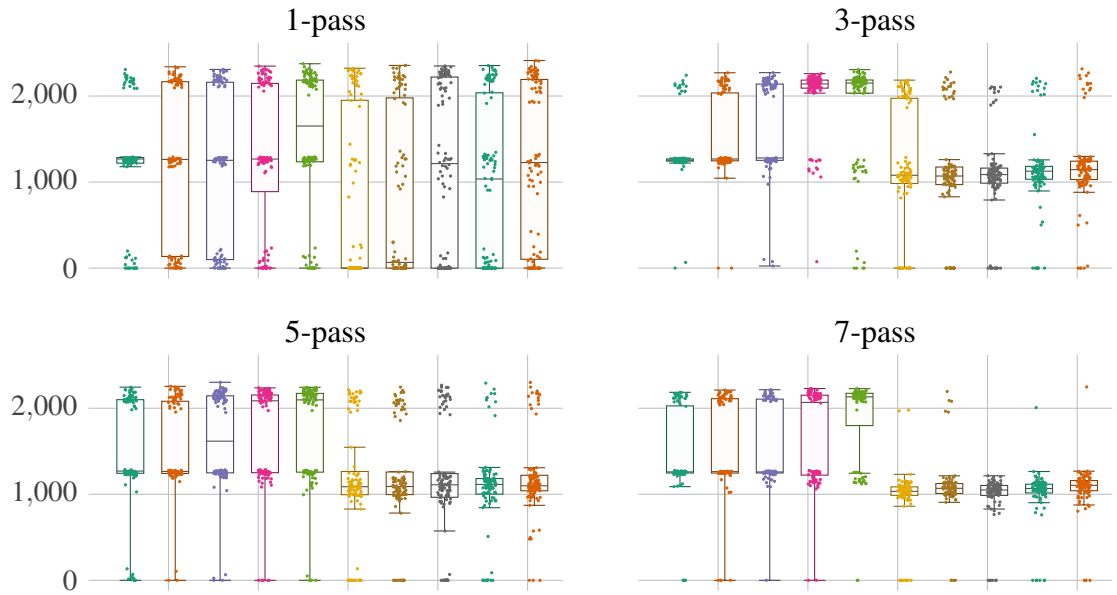


Figure 5.2.: Plotted fitness values of ten different individuals that played the game *aliens* 100 times with 1,3,5 and 7 simulation passes.

In some cases, like *aliens* or *eggomania*, the averaged values are reasonable. However, the results from the other two games are disillusioning. Even a very time consuming high number of passes is not enough to further stabilize the fitness. Nevertheless, it can be assumed that larger numbers of runs will result in slightly better outcomes. But even now, the difference between five and seven runs is not that high. Further increasing the passes is not a practical solution. The advantages compared to the way longer evaluation time does not justify these insignificant improvements. Some other adjustments must be made. Different improvements were tested to find a more reliable fitness function. See Appendix B.1 for details and benchmarks.

5.1.2. Mapping Function

As described in the genotype-phenotype mapping section (4.5), the level generation is not deterministic. That means that multiple runs will yield in slightly different levels. Exactly as in the previous section, an experiment is needed to determine how many levels must be created to get a meaningful fitness value for a chromosome.

This experiment also uses ten different individuals and evaluates each 100 times. However,

instead of always using the exact same level 100 times, like in the experiment before, this time, 100 different levels are created from the same chromosome. For each run, either 1, 2, 3 or 4 levels are generated and then evaluated. The average of all runs is the fitness value. Since the generated levels differ only slightly, the results for one run should be about the same as from the last experiment or at least only marginally worse.

Table 5.1 shows the standard errors. It uses the exact same method as before to calculate them. As you can see, one run produces almost the same standard error as the last experiment (see Table B.3). More runs greatly reduce the variances. Detailed results are shown in Appendix B.2.

#	Aliens	Bolo Adventures	Eggomania	Survive Zombies
1	52.59	35.77	30.55	66.22
2	36.37	24.65	21.88	51.14
3	31.27	20.43	17.24	44.41
4	28.08	19.73	15.29	42.15

Table 5.1.: Standard error of fitness values from 100 different individuals of four different games.

Be aware that each run is especially expensive, since all of them will need to simulate 20 games (four agents with five simulation passes). As a compromise between a stable fitness value and computing time, three runs were chosen for the final evaluation.

5.1.3. Evolutionary Algorithm

For this experiment, a population size of 50 individuals were chosen. An elitism approach was used to always take over the best individuals to the next generation. An elitism approach made sure that the best individual was always chosen to be in the next generation. Crossover generated 15 new individuals from the selected ones and mutation mutated 50% of the new population.

All agents for the fitness evaluation were allowed to play a maximum of 2000 steps. They had 40ms time for each steps. This complies with the competition rules of GVG-AI.

Random tests have shown that for most games the EA finds a fitness plateau after way less than 50 generations. Although that more generations are generally better, the time needed to evaluate each additionally generation should be taken into account. 50 generation was a good compromise between computation time and providing meaningful output.s

5.2. Distributed Computation

Due to the nature of simulation-based evaluation, the level generation needs a lot of computing power. As just described, every individual creates three different levels. Calculating the fitness of one such level needs three different agents; each of them needs five runs to output a stable fitness value. In a worst case scenario, every agent uses the maximum of 2000 steps á 40ms – overall 80s. In that event, evaluating only one individual will need $3 \cdot 3 \cdot 5 \cdot 80s = 3600 = 60min$! It is fair to say that most agents will not need the whole 2000 steps, nor the 40ms per step. On average the computation time will be much lower, but still in the range of several minutes.

Since we want to have 50 individuals in each of the 50 generations for multiple games, a sequential processing could take months, maybe even years. The only solution is to use a highly parallel approach. Thus, the generating algorithm is divided into a master and an evaluation process. The master process is responsible to take care of the main EA tasks. That means, that this process generates all individuals, does the selection and the level instantiation. Additionally, the master sends the levels created by the individuals to all registered evaluation servers. The evaluation servers compute the fitness of these levels and send the results back to the master.

One important thing to consider when multiple systems are in use, is that the used agents have a fixed amount of CPU time per step. Different CPU speeds would therefore skew the results. For this reason, it is critical to always use the same system configuration for the whole experiment. Otherwise the results would not be comparable.

For this experiment the master process runs on an Intel Core i7 5820K processor with overall 12 threads @ 4.20GHz and 64GiB DDR4-2666 memory. All 12 threads could be used simultaneously to instantiate new levels. The evaluation servers utilized a Intel Xeon X5650 with 24 threads @ 2.67GHz and 48GiB DDR3-1333 memory. Ten such servers were in use to concurrently evaluate 22 levels each. The two remaining threads are exclusively reserved for background tasks, like from the operation system itself. To further restrict the interference, every evaluation process was pinned to exactly one CPU core.

In the end, this distributed server-client architecture speeds up the evaluation by a factor of around 220. It is just a theoretical number and ignores things like overhead and non-parallelizable parts.

5.3. Analysing Levels

The intend of the fitness function design was not to be able to compare the output of different agents or games. The function's only goal is to provide guiding assistant for the EA to evolve levels in the right direction. Therefore, each agent and each game should be treated independently. That also means, that the absolute values of an EA from one game / agent can not be compared to the output from another EA. A high fitness value only means that the level is playable, maybe winnable, and that the used agent can solve it better than a random agent. A higher fitness value of an $agent_a$ compared to an $agent_b$ does not mean that the level from one of the agents is *better* or *worse* in any way. Better could mean, for example, that a level is more difficult to solve (but still solvable), that the level is more aesthetically pleasing or that it is more fun. The last two points are subjective and therefore notoriously hard to measure. What is measurable to a certain degree, is how difficult or complex a game level is.

A lot of different ways to measure complexity can be found in literature of combinatorial game theory – from state-space complexity [All94] to computing the asymptotic difficulty. Unfortunately, almost all of these methods are limited to actual *games* and are hard to adapt for measuring game *level* complexity. For some of the used games, the map size or the relation of the number of different sprites could be a way to measure the difficulty, especially the ratio between enemies and friendly or useful objects. However, this greatly depends on the actual game description and is sometimes not easy to identify. Even comparing the difficulty of similar levels could be troublesome, because a lot of other factors, like the actual position of enemies / friends, could play a role.

Another idea is to play each generated level with all used agents and then compare the output. Levels that can be won by more agents are easier than levels that almost no agent wins. The number of needed steps can also be considered. Less needed steps to win a game makes a level easier. On the other hand, needing less steps before an agent dies is a sign that the level is harder. As a bonus of testing all the levels with each agent, it also provides insights about when a level is particularly fitted towards one specific agent. This measurement method is also not perfect. Some agents are simply not that good at playing certain levels. Therefore, two example human made levels from the framework will also be tested – One easy level (the first one) and one more challenging (the last provided level for each game [3]). The results will have an opportunity for interpretation. They will be presented in a table and discussed in detail. The readers are free to make up their own minds of how to interpret the results.

[3] The provided levels are more or less sorted by difficulty, beginning with the easiest.

5.4. Results

Levels were generated for 20 games of the first two game sets from the GVG-AI framework. This section here presents a high level overview about all results. The details are rather long and therefore presented in the auxiliary Chapter A. Each of the 20 games is presented there in an extra section. For all them, the generated levels are displayed as well as the results from the test runs as described in Section 5.3. Additionally, the results from the EA will be discussed and evaluated.

As presented in Section 4.6.3, the fitness function strongly focuses on winnable levels. Therefore, the fitness value is heavily influenced from this factor. The score and the number of steps that an agent needs play a minor role. A fitness value above 500 almost always means, that the game can be won at least sometimes by the used agent. Over 2000 oftentimes means, that the agent was able to win the games almost all times. Exceptions are games where the possible score is extremely high. These special cases are mentioned in the detail chapter next to the corresponding result table.

The proposed level generator was able to produce winnable levels for 19 out of the 20 games. The fitness value from different agents within one game did not fluctuate that much, besides the fact that the generated levels were completely different in most cases. However, there were some exceptions like *Dig Dug*. The generator variant with the MO agent *paretoMCTS* created significantly better solutions. The final level was also heavily fitted towards this agent. That means, only this agent was able to win this level reliably. The levels created with the other two SO agents were rather uninteresting, because of their extreme simplicity. Such games, were the MO agent created levels that only he could win occurred multiple times. Another examples are the games *Survive Zombie* and *Frogs*. Interestingly, for the game *Frogs*, the *return42* agent is really good at playing the provided example levels and the generated levels from both SO variants, but he was not able to reliably solve the MO generated level.

The fitness value of different games could vary a lot. Although that most times the EA found a fitness plateau at around 2000, some game level did not even reach a value of 1300. An example is the game *Infection*. This either means, that no simpler level could be found to win the game more often or that the used weaker agents perform similar in this game.

It is interesting to not only look at the final results, but also what happens during the creation process. The evolution of the fitness value helps to understand how difficult it was to find a suitable level for a specific game. For some games like *Butterflies* or *Eggomania*, the generator found good levels almost instantly. In one case for the game *Eggomania*, the generator found a level with a fitness of around 2000 in the first generation. That means, that a winnable level was found by pure chance. This is a sign for a good (indirect) representation of a level and

of course also that the game has rather simple rules. *Eggomania* has only three game specific sprite types and a handful of defined interactions. It is also beneficial for the generator that the game description is well defined. Thanks to the hardcoded orientation of most objects in this game, the generator was able to deduce placing constraints. The game *Butterflies* has even less game objects, but does not provide any hints for their placing.

Examples for games where the evolution took way longer, are *Firestorms* (Section A.2.3) and *Dig Dug* (Section A.2.2). As already mentioned, the MO agent was able to produce relatively good levels for the game *Dig Dug*. However, both other SO agents not only needed way longer to find a good level, but also produced levels with a far lower fitness value. The *sampleMCTS* agent found two significantly improved levels in the last two generations. This suggests that more generations are needed to create better solutions.

The algorithm could easily be adapted as a tool for a designer. Instead of a fully autonomous system, that has no external input besides the game description, the algorithm could be input from a human. The human could take over parts of the Evolutionary Algorithm or even the whole process. For example, he could just score the levels to better guide the search process. He could also alter the parameters that are encoded in the chromosome like the level size, the number of sprites or the different arguments for the Likeliness-Matrices.

Thanks to the *Pattern-Matrix*, the generator is able to produce levels with certain patterns in it. It actually did happen in a lot of cases, that generated levels had sprites arranged in some pattern. Unfortunately, the fitness function or the game itself give no reward for specific structures. There are also no hints in the game description that could be used to recognize if some pattern is desired or even needed in a game. For this reason, even if a level had a pattern, it was mostly either unusual for the game or it was applied to the wrong sprite type. An example would be randomly moving NPCs that are arranged in a cave-like structure. This happened in the case of *Zelda* (Section A.1.10). The *paretoMCTS* variant grouped monsters together to make up a cave (see Figure A.20c). The structure is fairly useless, since all of the monsters move away immediately after the game starts. However, another level from the same game created with the *return42* agent, also created a similar pattern, but this time made out of walls. Therefore, the Pattern-Matrix is useful, but hard to get right in an automatic way without any other external input.

Another unexpected result was found for the game *Camel Race* (Section A.2.1). All of the generated levels look highly different from the provided examples. Not a single property was similar to the human chosen one. Normally, the objective of this game is to reach a goal before other computer controlled NPCs (*camels*) are able to arrive there. The surprising part is, that all levels had a huge amount of camels placed in close proximity to different goals. These levels should therefore not be winnable, but remarkable in most cases the agent were able to win this

level. What was found here, is a bug in the game description. The game description says, that the level is won as soon as one avatar or one camel reaches a goal. The emphasis lies on *one*. According to this definition, if two or more camels arrive at a goal at the same time step, then nothing will happen. The player has then enough time to reach the goal regarding where it is. The generator found a completely unpredictable new solution that a human probably would not have found. The bug effects all levels, even the predefined. It is a lot less probable due to the lower number of camels, but nonetheless possible.

A main difference between levels generated with a SO agent and a MO agent is the size of the level. In a lot of cases the levels from the MO agent were far larger. The reason for this is the utilized *exploration* objective. This second objective is immensely beneficial in a large number of games. A good example is the game *Whackamole* (Section A.2.9). In this game, the player has to catch moles that pop out from portals that are mostly scattered around on the map. The MO agent produced a 12 to 20 times larger level than the SO variants. The SO agents were unsurprisingly not that good at playing such an enormous level. However, this feature is not always an advantage. There are games where larger level make winning them too easy. This is often the case for games where the player has to avoid something rather than to find or collect objects. For example in the game *Aliens*, the player has to avoid getting hit by bombs that are dropped from approaching hostile aliens (Section A.1.1). In general, a larger level means that the player has more time to kill the aliens and and that it is easier to dodge the bombs. Although that the MO generated level has the most aliens in it, it was also the easiest level to win for all agents due its large size.

In some cases, a particular generator variant produced such a different level design that only the used agent was able to win it. That means, that the level was overfitted towards a specific agent. This phenomena can be observed in the game *Frogs* (Section A.1.5). The level that was generated with the MO agent can also only be won reliable with the MO agent *paretoMCTS*. The same happened in the game *Dig Dug* (Section A.2.2). In both cases, the *sampleMCTS* controller could not even win a handful of times out of 100. Therefore, the wins were more a coincidence. The *return42* could at least solve the level a few times more, but still much worse than their average victory rates on other levels. The reason for the overfitting is in both cases game specific. Unfortunately, in this case, a generalized rule to differentiate between specific MO and SO characteristics could not be deduced.

The difficulty of the games were another investigation in this thesis. Although that the main focus lied on generating playable and winnable levels, the difficulty was also of interest to understand possible differences between a level generated by a SO agent and level created with a MO agent. Measuring the difficulty of a level is not an easy task. There is no general method to do this for arbitrary games. In some cases, game specific characteristics were found that

could be used to determine when a level is either easier or more challenging. A more objective approach was to simply let the agents play the level and investigate the results. This strategy can be used to compare the difficulty of different levels from one game. A comparison between different games is not possible. No general statement could be made about which generator variant produced the most difficult level. It highly depends on the game, but in a large majority of cases, the MO variant were able to generate levels that were hard to win for the SO agents. The opposite also occurred in some games like *Overload* (Section A.2.6).

Finally, it can be said that using either a MO or a SO agent for the level generation does make a big difference. In a lot of cases both used SO agents produced very similar levels, whereas the level from the MO agent was vastly different. Good examples for this can immediately be seen in the levels from *aliens* (see Figure A.2) or *Butterflies* (see Figure A.6).

6. Conclusion

A new procedural level generator was presented in this thesis. It is able to produce solvable and sufficiently challenging levels from an arbitrary game that is described in the Video Game Description Language (VGDL). Not a single human interaction is needed throughout the whole processing pipeline. Everything can be done automatically. Nonetheless, a human designer could also intervene and guide the generator to have a better control over the output.

The proposed approach uses the GVG-AI framework to parse the game description. The game description contains valuable information that are then used to build up an indirect representation of a level. This representation is not a playable level. It is an abstract description about how a possible level could look like. Encoded are information like the size, which and how many sprites should be in the level and how they are distributed. The distribution describes the *structure* or *design* of a level. It "answers" the question that every game designer would have, like: *Are there any clusters of a certain sprite type? Are they arranged in a specific form like a maze or a chess field?* This arrangement is achieved by using a new technique called *Likeliness-Matrix*. Different kinds of Likeliness-Matrices are used to guide the placement of each sprite. Additionally, an Evolutionary Algorithm is used to refine the generated levels. It searches through a lot of possible solutions and try to alter them in some way to create even better levels. The fitness function to compare each solution is simulation-based. It uses different agents to play each solution candidate. The outcome defines the quality of a level.

An experiment was performed to validate the functioning of the used approach. 20 different games from the first two game sets from GVG-AI were used to generate new levels from the ground up. The EA evolved 50 generations with 50 individuals each. Three different agents were utilized to generate three levels for each game. Two of them were Single-Objective agents and one was a Multi-Objective agent. The experiment should also determine if there is a difference between levels generated with either a SO agent or a MO agent.

The results of the experimental work were presented in detail and has been extensively discussed. Each of the three variants of the generator was able to generate winnable levels for any game but one. Although that the fitness from all three variants were more less the same within one specific game, the produced levels were generally fairly different. One of the more obvious differences is the size of the level. In a large number of cases, the generator variant that utilized the MO agent produced larger levels than the variant with a SO agent. A lot of other differences were found. Unfortunately, most of them were rather game specific and could not be generalized.

Another important thing to mention about the generator is, that the creation process takes a very long time. The experimental work in this thesis used a distributed approach with a fleet of servers to create enough levels in a timely manner. Therefore, this approach is not applicable in realtime scenarios. However, the main algorithm could be used in such case since only the EA process takes a long time. If a designer chooses fitting parameter or parameter sets, the algorithm could output playable level on the first run. Under this conditions, the evolutionary process is either not necessary or could be dumped down to speed up the process. Of course, the chosen parameters would highly depend on the specific game and must be adapted whenever some rules of the game change.

7. Future Work

Since this was the first implementation of this algorithm in GGP context, the main focus was on generating playable and winnable levels first. It was also intended to overfit the generated level towards a specific agent to better see the differences between them. Nonetheless, the fitness function could be altered in future works in order to create levels that are even better suited to the desired criteria. As discussed in Section 4.6.3 about the fitness function, the next desirable property of a level is *challenging*. More difficult levels offer a greater challenge for a player. They would also be useful for the GVG-AI competition. Therefore, altering the fitness function could be a good way to help increasing the difficulty. The current function rewards levels more that can be won more often by an agent. Winning the same level just 1 out of 5 times is worse than winning it all five times. This is good to generate levels that could be won in any case by a sufficiently capable agent. Strictly speaking however, a single win would be enough to mark a level as *winnable*. Being able to win a level more often just means that this level is probably too easy. Thus, changing the fitness function to reward a single win and penalty multiple wins, could increase the difficulty of the generated levels.

Another desirable quality of a level was *fun*. Fun is quite a subjective property and is therefore notoriously hard to evaluate with a computer. For that reason, a human component could be integrated somehow in the evaluation pipeline. This is also difficult to realize since a huge number of levels were generated by the EA. Scoring them all could be too much work. An adequate number of human players would be needed. Maybe some kind of interactive online voting portal would be helpful.

In general, the fitness function could be improved further. The current one optimizes multiple objectives at once. The handcrafted function tries to find an optimal balance of the score difference, the step difference and also the win/loss ratio. In some cases, it is desirable to change the weight of these properties to better control the output. The manual approach makes this quite difficult and error prone. A Multi-Objective EA could be used to better handle this. Instead of trying to alter the fitness function every time when the requirements change, a set of simple constraints could be used to adjust the weights.

Interestingly, the EA component of the demonstrated algorithm could be omitted. All parameters like the map size, sprite quantities or the ones for each Likeliness-Matrix could be selected by a human designer. The designer would manually choose fitting values to generate appealing levels. He could also just use the created levels as a starting point and further refine them by hand. Thus, the algorithm would then be more a tool to spark creativity or to kick-start the design process.

APPENDIX CHAPTER A

Detailed Results

A.1. Set 1

A.1.1. Aliens

Generator	Width	Height	Victories in % (+avg. steps)				*wall*	*background*	*base*	*portalSlow*	*portalFast*	Σ
Example 1	30	11	**89** (618.20)	**100** (531.66)	**97** (549.55)		0	281	47	1	0	330
Example 2	30	11	**89** (618.20)	**100** (531.66)	**97** (549.55)		0	292	36	0	1	330
paretoMCTS	23	15	**84** (662.60)	**93** (629.74)	**85** (614.89)		0	325	14	3	2	345
sampleMCTS	4	3	**0** (n/a)	**100** (306.21)	**1** (306.00)		0	6	2	2	1	12
return42	4	6	**0** (n/a)	**99** (308.47)	**85** (310.22)		0	19	0	4	0	24

Table A.1.: Statistics about two examples and three generated levels for the game *Aliens*. Victories are from 100 random runs of this following three agents: ***paretoMCTS***, ***sampleMCTS***, ***return42*** (in this order) Number of steps is the average of all won games. Last columns are the numbers of all placed sprites (excluding the avatar).

In this game, the player is attacked by *aliens*. Aliens are spawned from one or more spawn points and continuously drop *bombs* that are able to destroy *bases* and the players avatar. Aliens only move horizontal and go one row down if they collide with the left or right level border. The avatar can shoot missiles to destroy the enemy aliens and bases. The game is lost whenever the avatar is destroyed, which also happens as soon as an alien collides with the avatar. By shooting all aliens, the player wins the game. The game has a fair amount of opportunities to collect points. Killing an alien increases the score by 2 points. Shooting down a base brings 1 point. The given levels all have just one spawn point. The bases are placed between the aliens and the avatars starting points and serve as a protection against the bombs.

As shown in Figure A.1, all three fitness functions were able to generate winnable levels. Neither of the three variants stand out, all of them found a feasible solution after just a few generations. The highest score was reached by *return42* with 2347.49 points, closely followed by *sampleMCTS* (2252.32) and *paretoMCTS* (2124.82). It did not even take 10 generations to find a plateau. This shows that finding a feasible level for this game is quite easy – even a random initialization could be enough.

The generated levels, plus a human-made one, are shown in Figure A.2. Statistics are listed at the beginning in Table A.1. Eye-catching is, that all generated levels have more alien spawn points than the example levels. The reason for this is, that more aliens also mean that there is

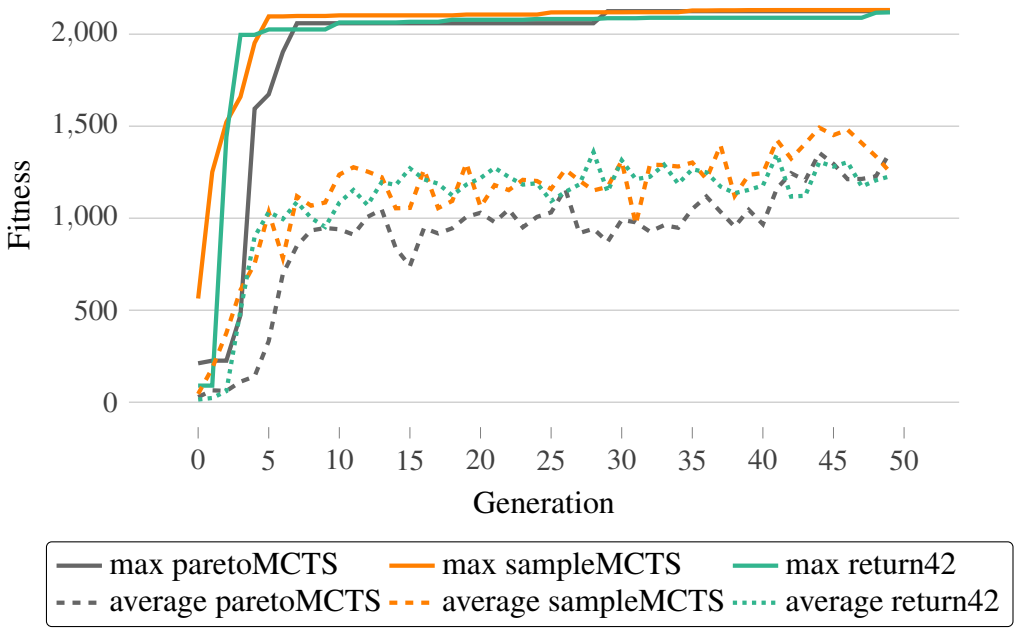
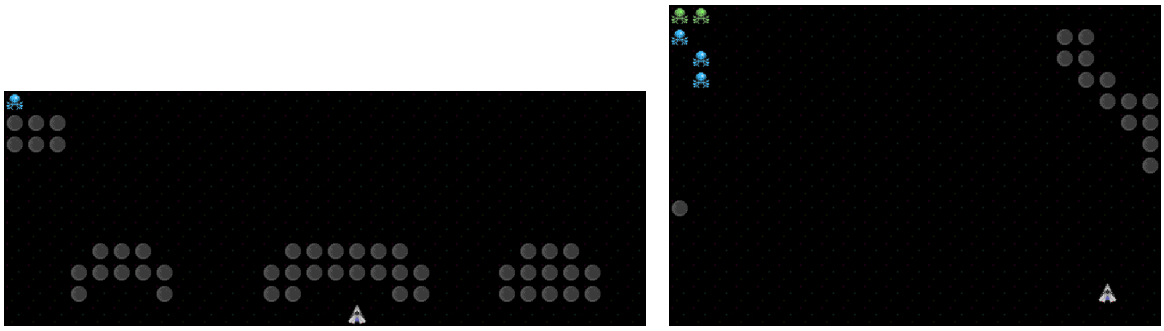
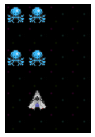


Figure A.1.: Average and maximum fitness values for the game *Aliens* for all 50 generations using three different agents for the simulation.



(a) Example

(b) paretoMCTS



(c) return42

(d) sampleMCTS

Figure A.2.: Generated levels for the game *Aliens* and one human-made example level design for comparison.

an opportunity to gather more points. Especially with regards to the fact that the naive agents are probably not capable of killing as many aliens as the more skilled agents.

Noticeable is the different size of the level. A higher level usually means, that the player has more time to avoid being hit by a bomb. The player also has more time to destroy the aliens. A narrower level has probably the opposite effect, since the avatar would have less space to dodge the falling bombs. The *return42* and *sampleMCTS* levels are very small, especially in comparison with the example levels. However, the level from the MO variant *paretoMCTS* has a acceptable size.

Quite impressive is the result of *sampleMCTS*. The level looks extremely simple, but none of the other agents is able to beat it. This is a perfect example of overfitting – it works just for one agent. In general, *paretoMCTS* performs better than *sampleMCTS* for the game *Aliens* [e.g. PlML16], but this agent has also problems with the other level from *return42*. The reason is probably the exploration objective. It is absolutely not helpful for such mini-levels.

Overall, the generated levels are quite useful. The *paretoMCTS* variant was the closest to the example levels. They do not have the same aesthetics as a human-made level. It is a pity that not more levels with a symmetric or ordered sprite distribution were generated. Nonetheless, generating such levels would be possible, thanks to *Pattern*-Matrix.

A.1.2. Boulderdash

Generator	Width	Height	Victories in % (+avg. steps)			wall	dirt	background	exitdoor	boulder	diamond	crab	butterfly	Σ
Example 1	26	13	0 (n/a)	0 (n/a)	8 (645.62)	93	167	16	1	33	23	2	2	338
Example 2	26	13	0 (n/a)	0 (n/a)	8 (645.62)	83	179	16	1	33	21	2	2	338
paretoMCTS	42	23	49 (334.37)	56 (261.29)	98 (69.59)	141	32	0	2	7	447	0	2	632
sampleMCTS	9	32	80 (281.39)	92 (323.88)	99 (23.15)	133	0	0	2	0	58	14	2	210
return42	15	22	74 (109.41)	100 (105.27)	95 (27.59)	132	9	0	1	0	180	5	2	330

Table A.2.: Statistics about two examples and three generated levels for the game *Boulderdash*. Victories are from 100 random runs of this following three agents: ***paretoMCTS***, ***sampleMCTS***, ***return42*** (in this order) Number of steps is the average of all won games. Last columns are the numbers of all placed sprites (excluding the avatar).

In *Boulderdash*, the player has to collect a certain number of *diamonds* before he can go through an *exit* to win the game. The provided example levels always have exactly one exit and multiple diamonds. The game description dictates that all exits must be destroyed to win a game. Originally, the level should be build up like a cave consisting of *dirt* and *wall* tiles. Two enemy types are hidden in the cave that are able to kill the player: *butterflies* and *crabs*. The player is able to remove the dirt and gather diamonds, but he must be careful with the *boulders* that are scattered around the map. These boulders will fall down as soon as the underlying dirt gets removed. A falling boulder can also kill the player. The most points can be earned with diamonds (2 points). Another way to increase the score by 1 is when a butterfly collides with a crab. This event also creates an additional diamond.

Figure A.3 shows, that all generator variants were able to create winnable levels rather quickly. However, it took way longer to find a fitness plateau for this game. *Return42* needed the most iterations but found the best result in the end.

Unfortunately, as you can see in Figure A.4, none of the generated levels have great similarity to the provided example levels. Whereas the example level had a lot of dirt, the generated levels had either none or just a few *dirt* tiles. The reason is simple: *Dirt* has no immediate reward. It is rather disadvantageous, since there is the danger of being hit by a falling *boulder*. It has to be said that this game is very difficult for all used agents [see PlML16, fig. 3]. Table A.2 with the results from the played games demonstrates this very clearly. This is probably also the reason why the levels look so different compared to the given examples. None of the tested

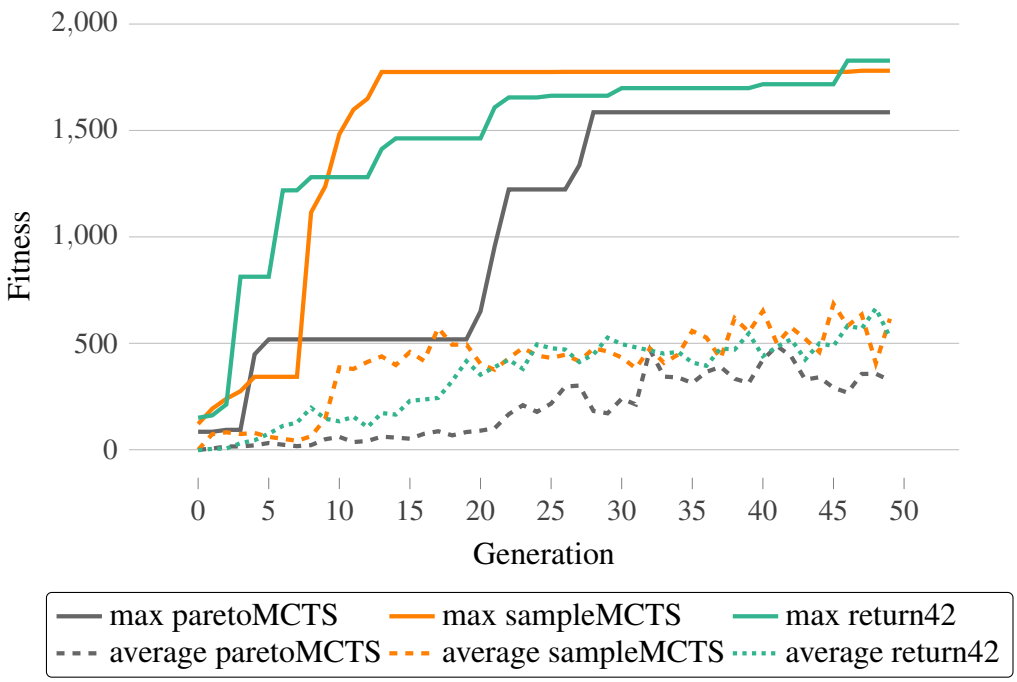


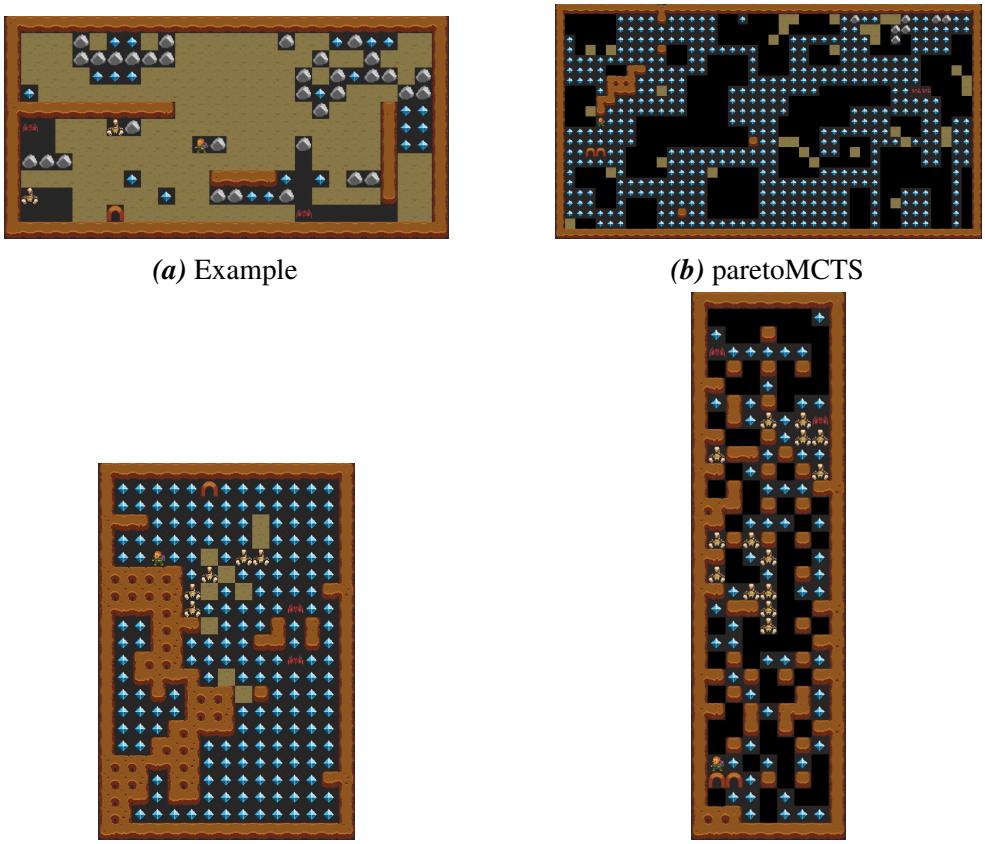
Figure A.3.: Average and maximum fitness values for the game *Boulderdash* for all 50 generations using three different agents for the simulation.

agents can successfully play them. *return42* is the only one that wins at least a few of them. Whereas the generated level can be played by all agents. Based on the average number of steps that each agent needed to complete a game, the levels are probably way to easy to solve.

Also, it is striking that there are no enemies in the level generated by *paretoMCTS*. The two other agents had absolutely no problem winning this level. Therefore, it is the easiest of all generated levels. The *paretoMCTS* agent himself was not that good at playing the generated levels from the other agents. From this, we can conclude that the *paretoMCTS* has problems with this game in general.

Surprisingly, all levels had the right amount of exit doors. For most other games that have such exits, the number is way off. They mostly have a lot more exits than needed. In a lot of cases, the exit door is a way to gather points. The original design of this games was not to provide a resource, but reward winning.

At the end, the generated levels are all winnable, but they do not correspond to what a human designer would have produced. Additionally, whereas the example levels are mostly *very* difficult, the generated levels are way to easy. In this case, no single agents stands out. All are quite similar.



(a) Example

(b) paretoMCTS

(c) return42

(d) sampleMCTS

Figure A.4.: Generated levels for the game *Boulderdash* and one human-made example level design for comparison.

A.1.3. Butterflies

Generator	Width	Height	Victories in % (+avg. steps)				*wall*	*butterfly*	*cocoon*	Σ
Example 1	28	11	**32** (517.69)	**58** (382.71)	**100** (113.79)		102	6	27	136
Example 2	28	12	**32** (517.69)	**58** (382.71)	**100** (113.79)		106	7	3	117
paretoMCTS	19	14	**85** (574.07)	**100** (423.26)	**100** (168.97)		81	52	46	180
sampleMCTS	6	10	**36** (51.22)	**97** (16.36)	**97** (16.72)		29	2	22	54
return42	13	5	**23** (61.43)	**86** (31.38)	**44** (22.89)		32	9	16	58

Table A.3.: Statistics about two examples and three generated levels for the game *Butterflies*. Victories are from 100 random runs of this following three agents: ***paretoMCTS***, ***sampleMCTS***, ***return42*** (in this order) Number of steps is the average of all won games. Last columns are the numbers of all placed sprites (excluding the avatar).

The goal of this game is to catch every *butterfly*. Butterflies are moving randomly around the map. As soon as they touch a *cocoon*, a new butterfly gets spawned. The player loses when all cocoons are opened. There is only one way to get points in this game: by destroying butterflies (2 points). The example levels for this game have almost always a balanced number of butterflies and cocoons.

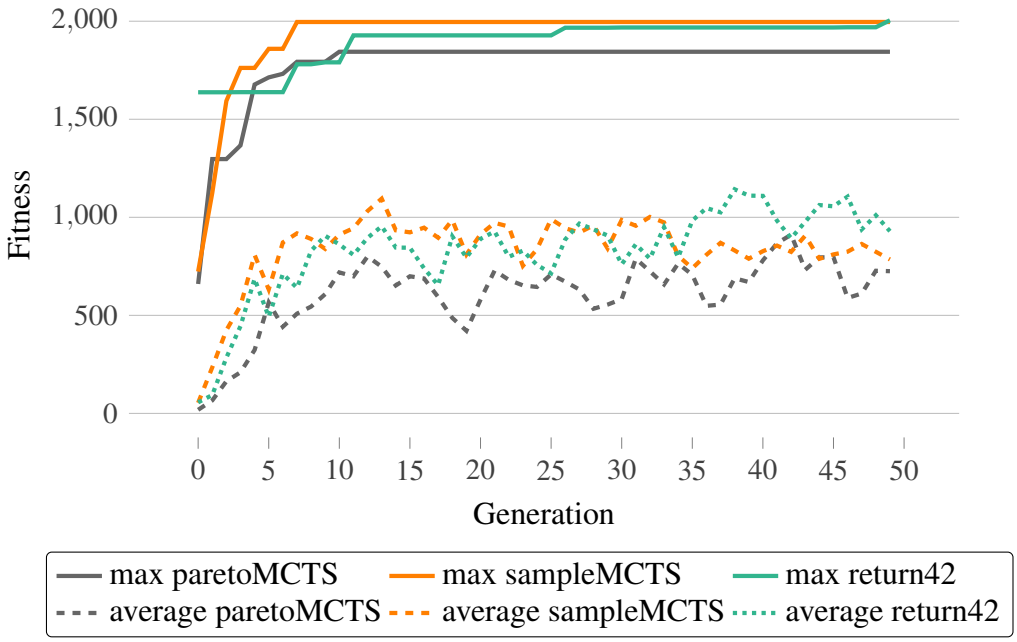
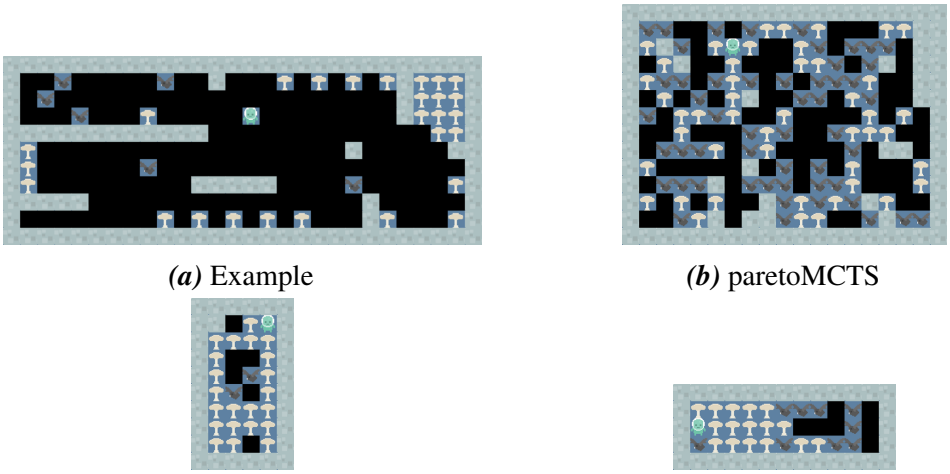


Figure A.5.: Average and maximum fitness values for the game *Butterflies* for all 50 generations using three different agents for the simulation.

The result of the EA is plotted in Figure A.6. Once again it did not take long before winnable levels were generated. The *return42* variant even found a relatively good solution in the first generation by chance. After around 10 generations a plateau was found to each variant. Although that this game is not that difficult and that it offers the possibility to get a very high score, the overall fitness is relatively low. The reason is probably, that the game rules are too simple and even a naive agent is able to collect a lot of butterflies.

paretoMCTS stands out strongly from the other two agents. This level has much less butterflies and cocoons and it is way larger. This makes it much more similar to the provided examples. It is likely that the second objective of this agent, the exploration, greatly helps the agent to find butterflies even in the most remote corners.



(a) Example ***(b)*** paretoMCTS

(c) sampleMCTS ***(d)*** return42

Figure A.6.: Generated levels for the game *Butterflies* and one human-made example level design for comparison.

Having more cocoons makes a level easier. The shown level from *paretoMCTS* has around the same number of cocoons as butterflies. Both other variants have a lot more cocoons than butterflies. This is not entirely unusual, since some of the human-designed levels also have way more cocoons than butterflies.

The test results from Table A.3 show, that no level was particularly harder than another one. Only the level from *return42* was a little bit more difficult. *paretoMCTS* performed rather poorly for the small levels that the other two agents generated, which is to be expected due to the exploration objective.

None of the levels have the same design characteristic as the example levels. *paretoMCTS* comes close and provides a good balance between aesthetic and difficulty. *return42* generated the most difficult level. It is hard to win, but it is nonetheless quite short. The number of steps needed is way too low to call this level challenging.

A.1.4. Chase

Generator	Width	Height	Victories in % (+avg. steps)				*wall*	*scared*	Σ
Example 1	24	11	**0** (n/a)	**0** (n/a)	**69** (217.49)		129	7	137
Example 2	24	11	**0** (n/a)	**0** (n/a)	**69** (217.49)		100	12	113
paretoMCTS	4	22	**95** (182.54)	**98** (196.92)	**100** (22.36)		59	3	63
sampleMCTS	10	6	**39** (73.44)	**73** (300.36)	**51** (16.69)		42	6	49
return42	4	8	**49** (62.16)	**85** (53.22)	**58** (12.16)		23	6	30

Table A.4.: Statistics about two examples and three generated levels for the game *Chase*. Victories are from 100 random runs of this following three agents: ***paretoMCTS***, ***sampleMCTS***, ***return42*** (in this order) Number of steps is the average of all won games. Last columns are the numbers of all placed sprites (excluding the avatar).

The goal of a player in this game is to chase *goats*. Goats are the only defined entity in the game and can either be *scared* or *angry*. At the beginning of the game, all goats are scared. Catching such runaway goat scores 1 point and makes the goat angry at the same time. An angry goat in turn tries to catch the avatar and to kill him. The game is won as soon as all goats are angry. As usable, walls can be placed as obstacles.

Although that the game rules are quite simple, winning such a game is not. As shown in Table A.4, *return42* is the only agent that can successfully play the example level. As presented in Figure A.7, the fitness values are relatively low, even after 50 generations. A very high fitness value is not to be expected, since the agent can only get 1 point per goat.

A look at the generated levels at Figure A.8 helps to understand the reason for the low fitness values. Since there are only two possible sprite types to place (besides the one-time avatar starting position), the probability to set a high number of walls is quite high. As a result, the algorithm places either too many walls or too many goats. Every goats increases the difficulty of a level a lot. Dodging them all becomes very difficult or even impossible. Additionally, placing too many walls results in cut off and inaccessible spaces as the *paretoMCTS* and *sampleMCTS* levels impressively demonstrate. These levels are therefore hard to evolve. Slightly changing the number of walls will oftentimes not change the fitness. Removing or adding just one or maybe a handful of wall tiles is generally not enough to solve the isolated space problem. Furthermore, adding a goat in a unreachable area will instantly make the whole level unwinnable. The game or the fitness function lacks of an incentive to set fewer walls at the beginning.

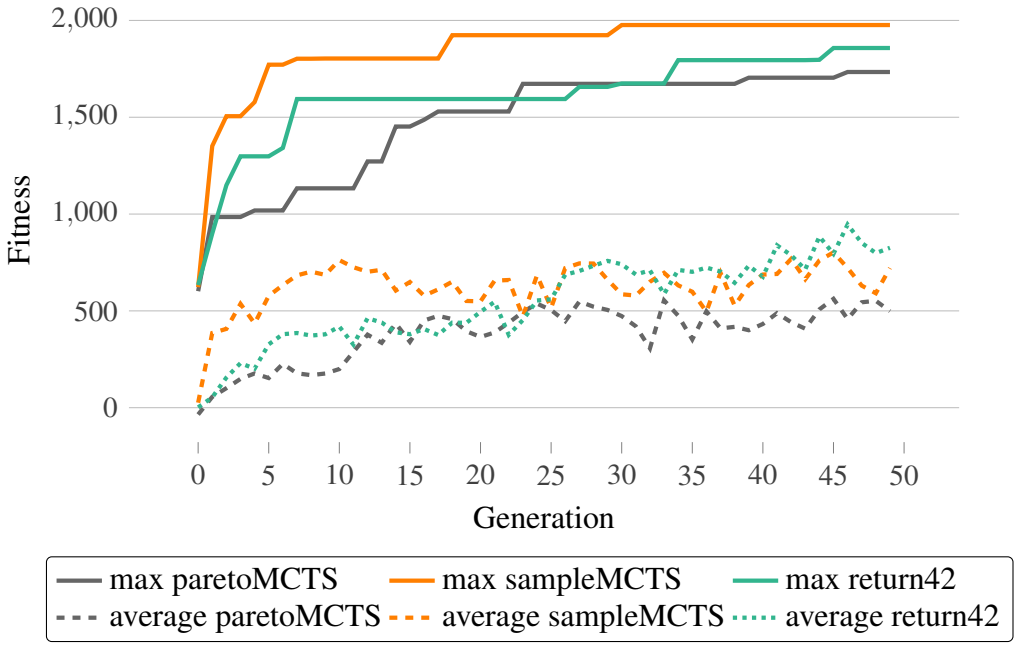
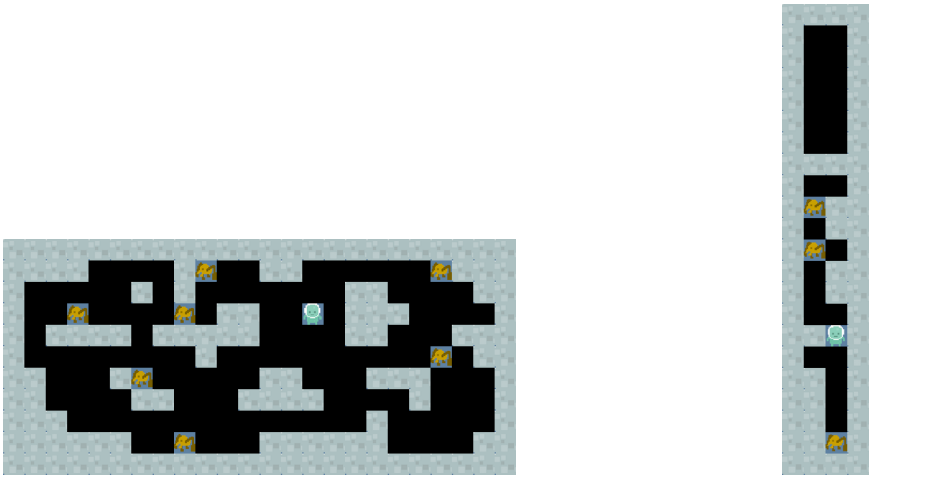


Figure A.7.: Average and maximum fitness values for the game *Chase* for all 50 generations using three different agents for the simulation.



(a) Example

(b) paretoMCTS

(c) return42

(d) sampleMCTS

Figure A.8.: Generated levels for the game *Chase* and one human-made example level design for comparison.

None of the generated levels look like the examples. They are also all way easier to solve. Unfortunately, none of them provides challenge to another agent. Great differences between the level generator variants are not apparent.

A.1.5. Frogs

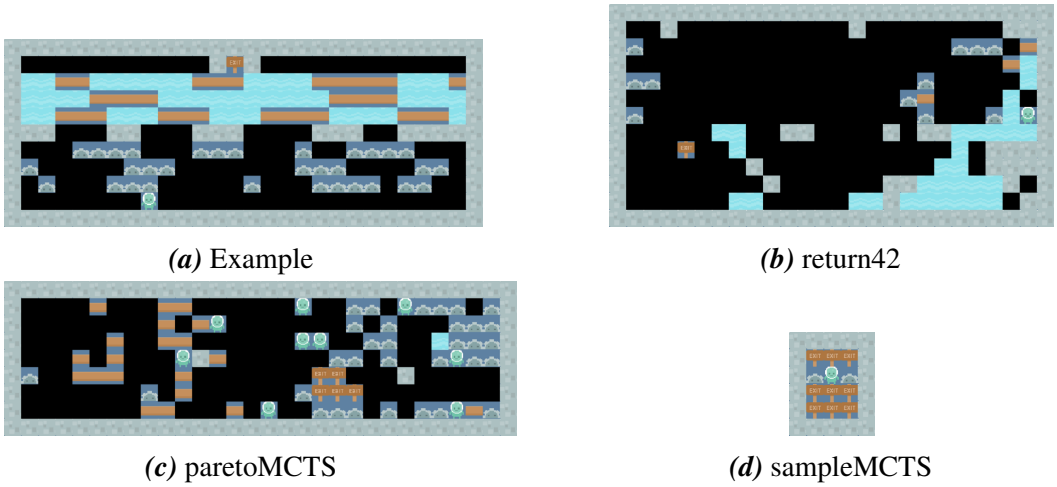
Generator	Width	Height	Victories in % (+avg. steps)			wall	goal	water	forestDense water	forestDense wall log	forestSparse water	forestSparse wall log	slowRtruck	fastRtruck	slowLtruck	fastLtruck	log water	avatar log	Σ
Example 1	28	11	**39** (366.56)	**21** (439.71)	**83** (241.27)	87	1	45	1	3	0	0	24	8	0	0	32	0	202
Example 2	28	10	**39** (366.56)	**21** (439.71)	**83** (241.27)	70	1	89	1	4	0	0	0	0	0	0	39	1	206
paretoMCTS	30	9	**93** (41.57)	**1** (11.00)	**36** (13.69)	68	5	0	1	8	0	0	0	0	9	23	20	9	144
sampleMCTS	5	6	**98** (56.13)	**100** (84.53)	**75** (12.08)	16	9	0	0	2	0	0	0	1	1	0	0	1	31
return42	26	13	**78** (283.82)	**87** (324.59)	**89** (48.84)	67	1	26	4	10	3	19	0	4	7	0	0	0	142

Table A.5.: Statistics about two examples and three generated levels for the game *Frogs*. Victories are from 100 random runs of this following three agents: ***paretoMCTS***, ***sampleMCTS***, ***return42*** (in this order) Number of steps is the average of all won games. Last columns are the numbers of all placed sprites (excluding the avatar).

In this game, the avatar is a *frog* that has to cross *rivers* and *streets* to reach the goal. *Trucks* are driving on the street and *logs* are swimming through the rivers. The player must not be overrun by a truck and has to jump over logs to cross a river. As shown in the example level in A.9, the human-made level design has clearly defined streets and rivers.

Reaching a goal will increase the players score by one point. There are no other opportunities to gather points. This is the reason why the level generator "interpreted" the game quite different than a human designer would do. Whereas the provided levels have only exactly one goal, the generated levels have a lot more. Having more goals increases the maximum possible score. Placing only goals on the map would be seen as a disadvantage for the used fitness function.

All generator variants were able to create winnable levels as presented in Figure A.9. A plateau could not be found in just 50 generations. *paretoMCTS* even found a better solution in the last iteration.



(a) Example

(b) return42

(c) paretoMCTS

(d) sampleMCTS

Figure A.9.: Generated levels for the game *Frogs* and one human-made example level design for comparison.

The difficulty in this game usually consists of avoiding trucks on the road and to correctly take advantage of the logs to cross the rivers. Both challenges are mostly not present in the generated levels. No level generator created some kind of river. The only other task that is left to do, is to collect the goal sprites. Only *paretoMCTS* placed a lot of trucks and therefore makes the level more difficult. As shown in Table A.5, it is indeed the most difficult level. The *sampleMCTS* controller had a lot of trouble winning this level, whereas he had no problem with both other generated levels.

Together they all have in common that the levels are quite different from their pre-made counterparts. There are no river-like structure or street. At least, the starting positions of the trucks are mostly right. Most trucks that drive left are placed on the right side and trucks that drive to the right are placed left. Overall, the *sampleMCTS* level performed significantly worse than the other two. The level is extremely small and has hardly any enemies. *paretoMCTS* is quite the opposite. A lot of enemies and much more space. The level from *return42* looks best from a subjective point of view. It has everything: enemies, enough space to explore and at least some structure.

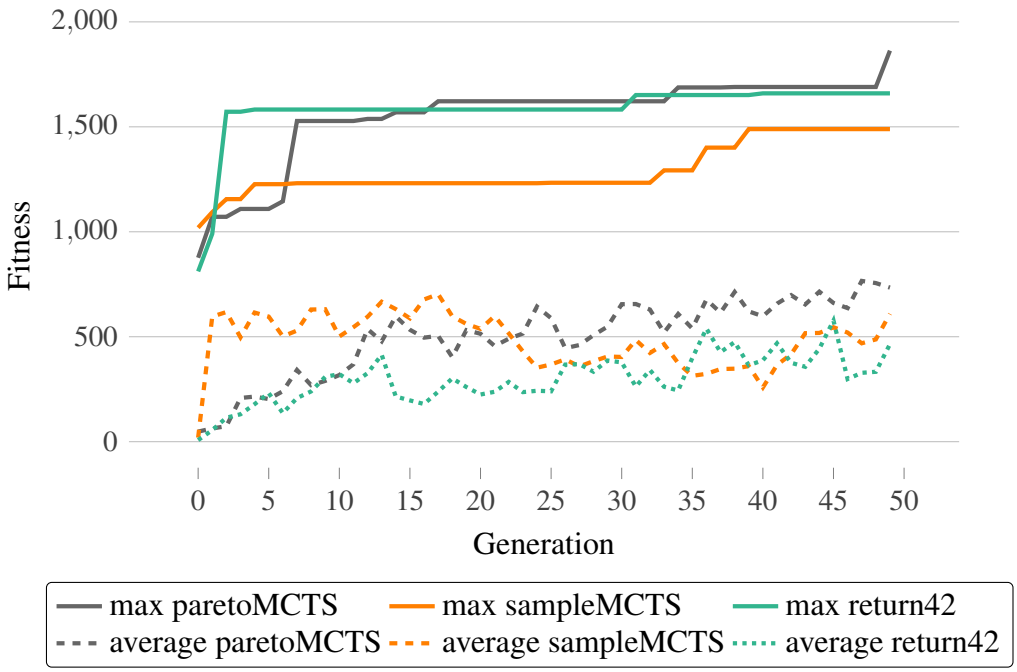


Figure A.10.: Average and maximum fitness values for the game *Frogs* for all 50 generations using three different agents for the simulation.

A.1.6. Missile Command

Generator	Width	Height	Victories in % (+avg. steps)				wall	city	incoming_slow	incoming_fast	Σ
Example 1	24	12	**52** (154.27)	**55** (142.27)	**100** (88.31)		46	3	4	0	54
Example 2	24	12	**52** (154.27)	**55** (142.27)	**100** (88.31)		46	8	7	0	62
paretoMCTS	33	21	**72** (137.47)	**42** (151.67)	**99** (84.29)		131	4	13	1	150
sampleMCTS	16	10	**100** (47.93)	**88** (52.43)	**100** (48.26)		48	28	0	42	119
return42	27	3	**100** (171.62)	**95** (156.25)	**98** (30.26)		56	2	4	6	69

Table A.6.: Statistics about two examples and three generated levels for the game *Missile Command*. Victories are from 100 random runs of this following three agents: ***paretoMCTS***, ***sampleMCTS***, ***return42*** (in this order)
Number of steps is the average of all won games. Last columns are the numbers of all placed sprites (excluding the avatar).

The player has to protect *cities* from incoming *missiles*. Cities are usually placed at the bottom and missiles at the top of the map. The starting position of the avatar is somewhere in-between. Destroying a missile increases the score by 2 points. Every destroyed city is punished with -1 point. The player wins as soon as all missiles are wiped out. If the player can not stop the missiles before every city is razed, he loses the game.

The EA results are presented in Figure A.11. All variants were able to generate winnable levels. The *sampleMCTS* variant needed way more generation to find a good level, but the end result is nearly the same for each agent.

The generated levels from *return42* and *sampleMCTS* do not look like anything from the pre-made levels (see Figure A.12). The distribution of the cities and missiles is quite different. The reason is simply that there are no clues for this in the game description. It is not necessarily negative. Placing missiles and cities closer together increases the difficulty. Also, it is one of the goals to generate novel ideas with procedural algorithms.

The *paretoMCTS* variant is close to the provided levels. The position of the cities and the missile is only exactly reversed, apart from one missile. Even the starting position of the avatar is between the missiles and the cities. In comparison with the other two generated levels, this one is also the most difficult (see Table A.6). Nonetheless, it is rather easy in comparison with the example levels.

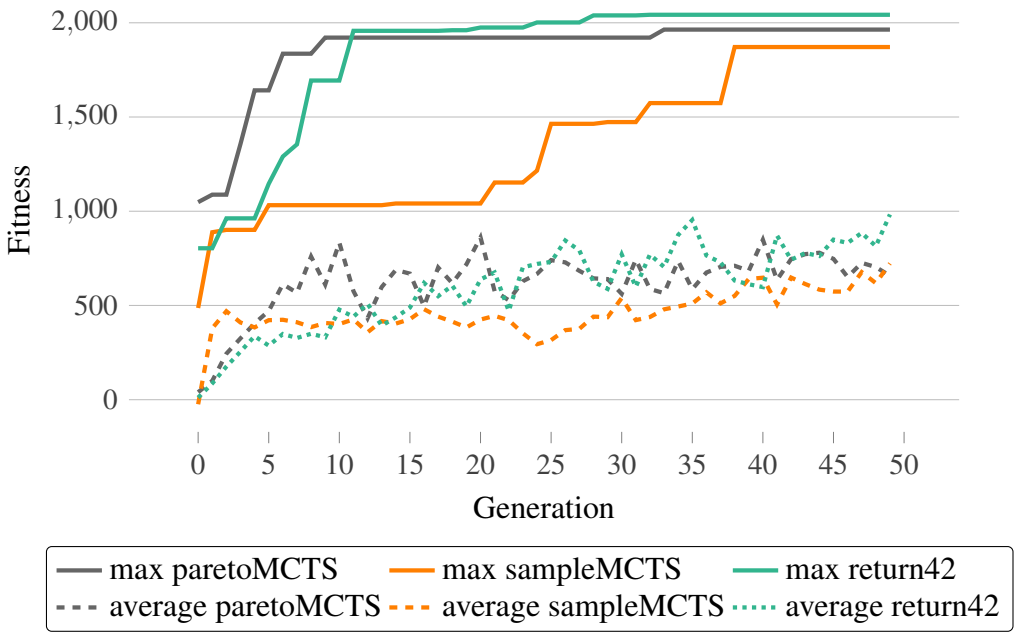
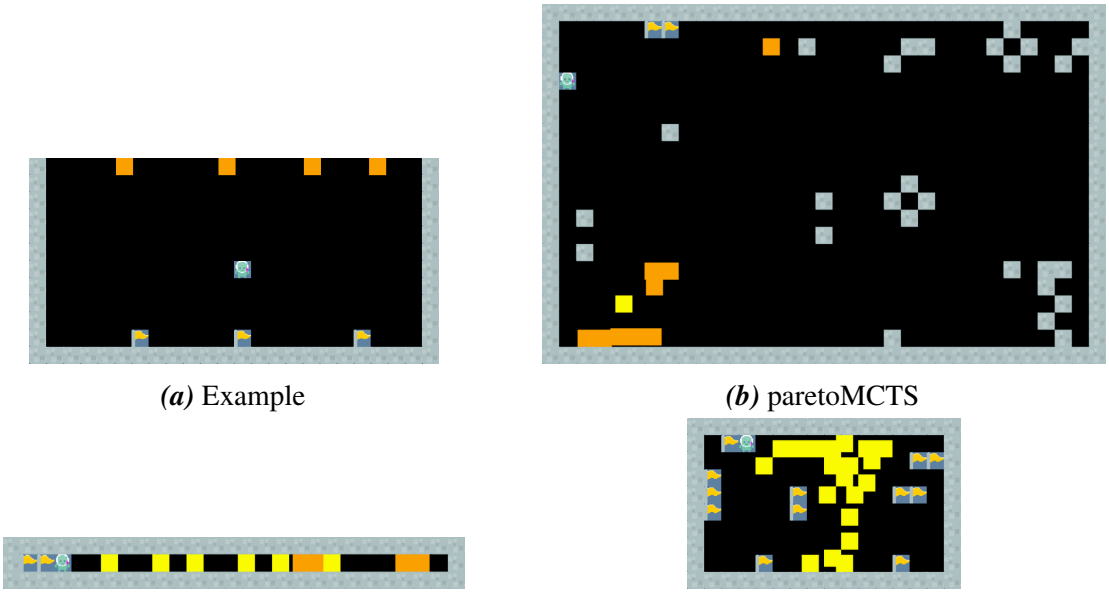


Figure A.11.: Average and maximum fitness values for the game *Missile Command* for all 50 generations using three different agents for the simulation.



(a) Example

(b) paretoMCTS

(c) return42

(d) sampleMCTS

Figure A.12.: Generated levels for the game *Missile Command* and one human-made example level design for comparison.

The ratio between cities and missiles can be used to measure the difficulty of a level for this game. More missiles are more difficult to intercept and at the same time, fewer cities can be destroyed faster. *paretoMCTS* and *return42* variants produced levels with an average of around 4 missiles per city. The *sampleMCTS* agent generated easier levels with a ratio of 2:1. At first sight, this level looks quite challenging, due to all these missiles near the cities. On closer inspection however, this level is quite easy due to the many cities. Yes, there are a lot of scary missiles, but they do not pose a real threat.

At the end, the algorithm could find quite acceptable solutions, but they probably are all too easy. Only the *paretoMCTS* level is at least a little bit challenging. The win ratio is close to the one from the example levels. Subjectively, this variant also looks the most attractive. The arrangement of the sprites and the overall structure is quite close to what a human designer would create.

A.1.7. Portals

Generator	Width	Height	Victories in % (+avg. steps)			wall	horizontal	vertical	sitting	random	goal	entry1	entry2	exit1	exit2	Σ
Example 1	19	11	**10** (518.50)	**0** (n/a)	**83** (323.75)	91	3	2	3	2	1	2	1	3	1	110
Example 2	19	11	**10** (518.50)	**0** (n/a)	**83** (323.75)	73	3	3	0	16	1	0	1	0	1	99
paretoMCTS	15	10	**92** (21.84)	**100** (18.17)	**99** (8.77)	56	0	5	5	15	5	0	43	6	2	138
sampleMCTS	50	14	**72** (74.93)	**93** (43.67)	**98** (18.30)	124	3	6	5	5	1	2	0	0	209	356
return42	38	8	**98** (85.16)	**100** (95.40)	**90** (53.71)	89	0	3	1	0	1	56	2	92	0	245

Table A.7.: Statistics about two examples and three generated levels for the game *Portals*. Victories are from 100 random runs of this following three agents: ***paretoMCTS***, ***sampleMCTS***, ***return42*** (in this order) Number of steps is the average of all won games. Last columns are the numbers of all placed sprites (excluding the avatar).

The player has to reach a *goal* while he has to deal with different enemies at the same time. The map is divided into smaller rooms by walls. Different *portals* are placed around the map that can teleport a the avatar from one room into another. The player has no prior knowledge about which portals are connected. He has to use them to find out which portals belong together. Usually, a level has only one goal. Reaching the goal is the only way to get a point. If there are multiple goals, then all goals must be visited. The game description has no upper limit defined that out algorithm could use as a constraint. As shown in Table A.7, surprisingly few goals were used. Only *paretoMCTS* had more than one.

A winnable level could be found extremely fast. The presented EA results in Figure A.13 show clearly that a winnable level could even be found by pure chance. After only around 15 generations, a fitness plateau was found. None of the three agents stand out particularly. At least the fitness of all generators were almost the same.

The generated levels look only slightly different to each other (see Figure A.14. All of them have numerous exits and entries, way more than the provided levels. One characteristic of the example levels is, that the number of exits and entries are more or less the same. Such balance is not given in the generated levels. The portal should also connect different rooms. Unfortunately, the generator created no room structure for these levels.

No clear best solution can be nominated here. There are all quite different to the given levels. Furthermore, all of them are too easy. None of the agents had any trouble winning them.

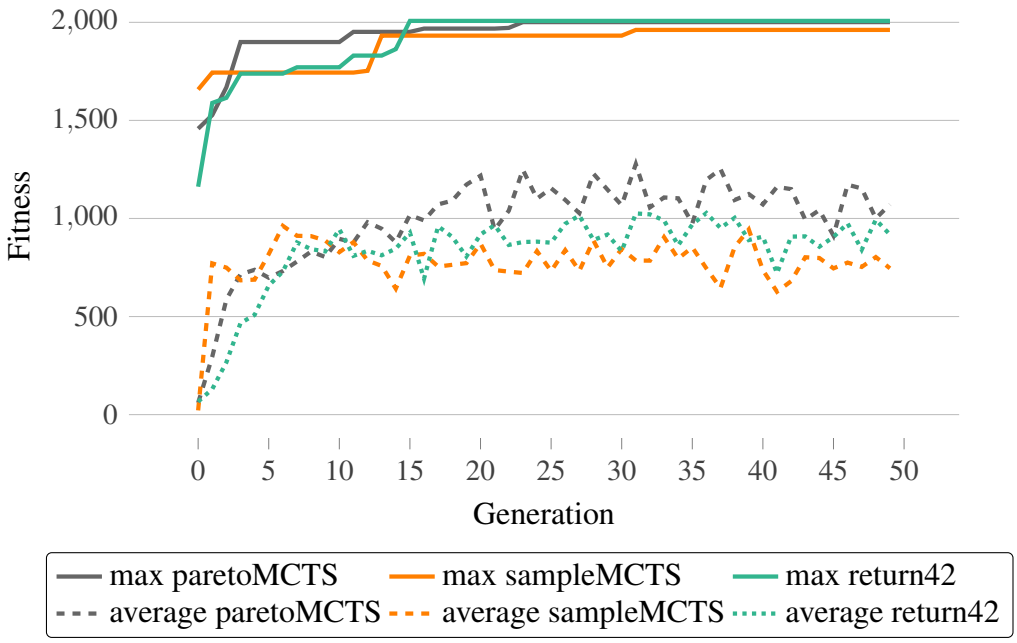
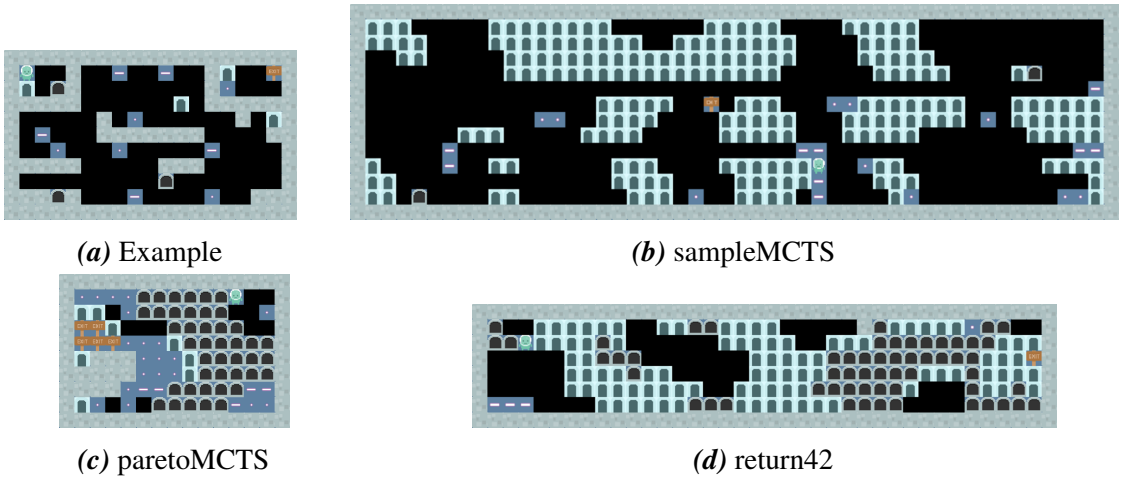


Figure A.13.: Average and maximum fitness values for the game *Portals* for all 50 generations using three different agents for the simulation.



(a) Example

(b) sampleMCTS

(c) paretoMCTS

(d) return42

Figure A.14.: Generated levels for the game *Portals* and one human-made example level design for comparison.

A.1.8. Sokoban

Generator	Width	Height	Victories in % (+avg. steps)				wall	hole	box	Σ
Example 1	13	9	**0** (n/a)	**0** (n/a)	**0** (n/a)		53	2	4	60
Example 2	13	9	**0** (n/a)	**0** (n/a)	**0** (n/a)		86	2	2	91
paretoMCTS	6	44	**93** (290.37)	**77** (351.09)	**100** (56.97)		105	144	4	254
sampleMCTS	21	5	**94** (288.61)	**96** (296.47)	**100** (30.17)		51	8	3	63
return42	31	24	**92** (181.58)	**75** (239.79)	**100** (31.79)		120	112	3	236

Table A.8.: Statistics about two examples and three generated levels for the game *Sokoban*. Victories are from 100 random runs of this following three agents: ***paretoMCTS***, ***sampleMCTS***, ***return42*** (in this order) Number of steps is the average of all won games. Last columns are the numbers of all placed sprites (excluding the avatar).

Levels in *Sokoban* consist of *holes* and *boxes*. The players task is to move each box into a hole. As a reward the player gets 1 point. Once there are no boxes left, the game is won.

The challenge in this game is, that boxes can only be moved forward by the avatar. That means, as soon as a box is moved against a wall or, even worse, in a corner, the player is either limited in which direction he can move the box or it is even impossible to move the box altogether. Therefore, just one wrong step could easily cost victory.

Details about the EA are plotted in Figure A.15. The EA was not able to find a solution with a high fitness value. It can also be seen that the first few generations were exceptionally bad compared to most other games. This and the fact that the *return42* variant found a better level in the last generation, means that a longer evaluation time could further improve the output.

The provided levels have more or less the same number of holes and boxes. More boxes in relation to holes make a level easier. Paradoxically, the reversed statement is also true. Having way more holes than boxes can also make a level easier. In the first case, the player could move a box into an edge and the game would still be solvable. Surplus boxes are a kind of backup. For the second case, having more holes would shorten the way a box needs to be moved. In all cases here, the generator chose to place way more holes than boxes. Have a look at the generated levels in Figure A.16 to see the the full extent of the problem.

Nonetheless, the levels from *return42* and *sampleMCTS* look reasonable. Whereas the arrangement of boxes and holes is probably too easy in the *return42* variant, the *sampleMCTS* provides a fair amount of challenge. This is also reflected in the results in Table A.8. It can also be seen, that both generated level are way easier to solve than any given example level.

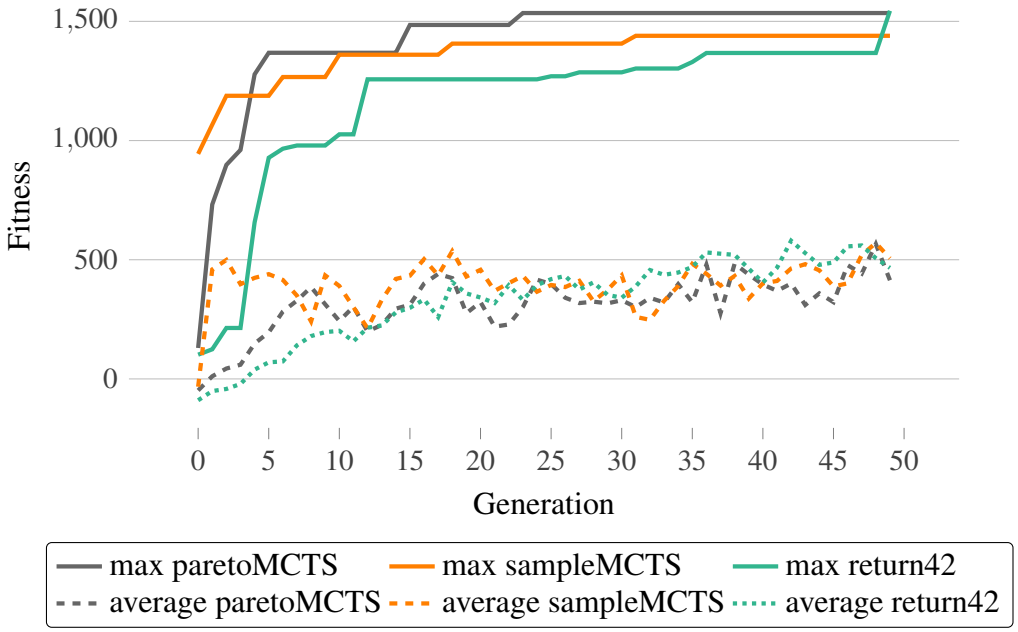
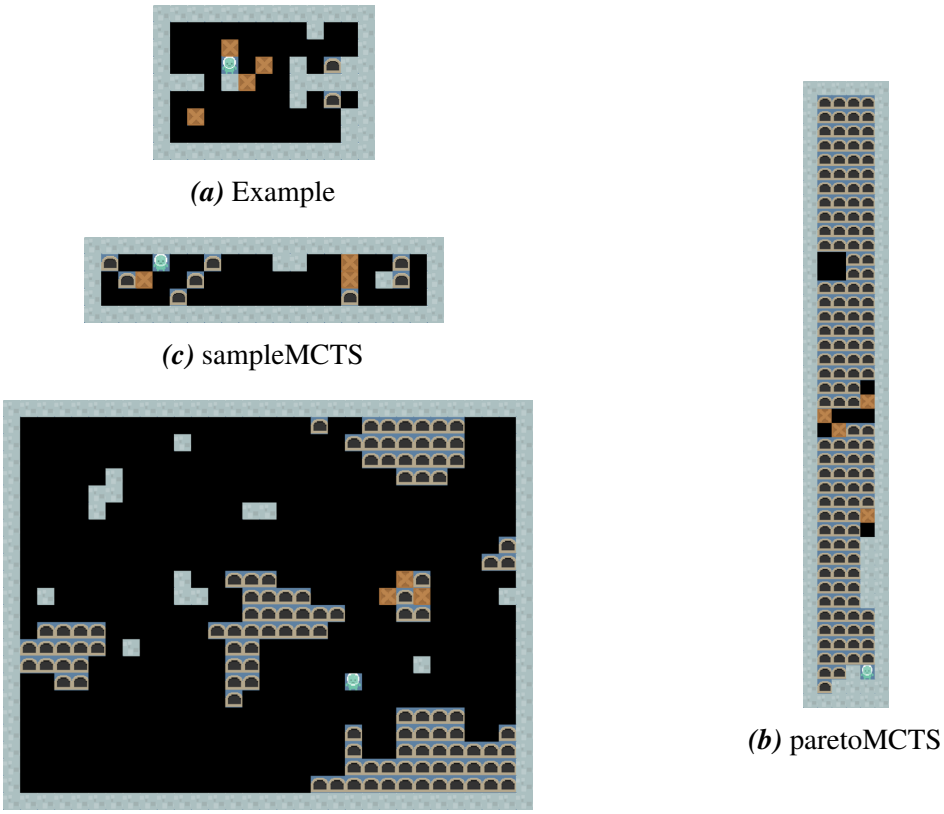


Figure A.15.: Average and maximum fitness values for the game *Sokoban* for all 50 generations using three different agents for the simulation.



(a) Example

(c) sampleMCTS

(d) return42

(b) paretoMCTS

Figure A.16.: Generated levels for the game *Sokoban* and one human-made example level design for comparison.

A.1.9. Survive Zombies

Generator	Width	Height	Victories in % (+avg. steps)				*wall*	*flower*	*slowHell*	*fastHell*	*honey*	*zombie*	Σ
Example 1	19	11	**0** (n/a)	**0** (n/a)	**14** (1000.00)		85	3	3	0	14	1	107
Example 2	19	11	**0** (n/a)	**0** (n/a)	**14** (1000.00)		78	0	10	2	9	6	106
paretoMCTS	31	50	**51** (1000.00)	**7** (1000.00)	**5** (1000.00)		158	79	13	6	18	0	275
sampleMCTS	11	37	**11** (1000.00)	**32** (1000.00)	**3** (1000.00)		92	110	0	37	10	3	253
return42	20	24	**78** (1000.00)	**89** (1000.00)	**0** (n/a)		228	119	0	0	45	82	475

Table A.9.: Statistics about two examples and three generated levels for the game *Survive Zombies*. Victories are from 100 random runs of this following three agents: ***paretoMCTS***, ***sampleMCTS***, ***return42*** (in this order)
Number of steps is the average of all won games. Last columns are the numbers of all placed sprites (excluding the avatar).

In this game, the player has to fight for their very survival. *Zombies* and creatures from *hell* are broken out and they try to kill the avatar. The player is not able to fight against the enemies, but he can try to gather *honey* which shields him from the deadly zombies. Honey is produced by *bees*. Bees are not directly placed in a level, but *flower* sprites are used. These flowers spawn bees. There is no protection against other enemy types. If the player is able to survive a certain amount of time, he automatically wins the game. Every collected honey brings 1 point. If the player has honey and collides with a Zombie, he gets a point deducted.

Evolving levels for this game took a lot longer than for most other games (see Figure A.17). None of the agents needed longer than another. All are quite similar. *Return42* could reach the highest fitness value. The other both agents found their best solution relatively late. This indicates that a longer evaluation could result in even better levels.

Table A.9 shows that none of the used agents were able to solve the provided levels. It is worth mentioning that the *return42* agent is especially bad at this game. He could not even play his own generated level. This is also the reason why the generated level of this agent looks so different (see Figure A.18).

None of the generated levels look like the provided. The *paretoMCTS* level has some structure, but it does not resemble the look of the example levels. This level also has no single zombie in it. It should therefore be easier to win. The *sampleMCTS* variant looks quite chaotic. Nonetheless, it has everything a good level needs. The test runs have also shown that this is

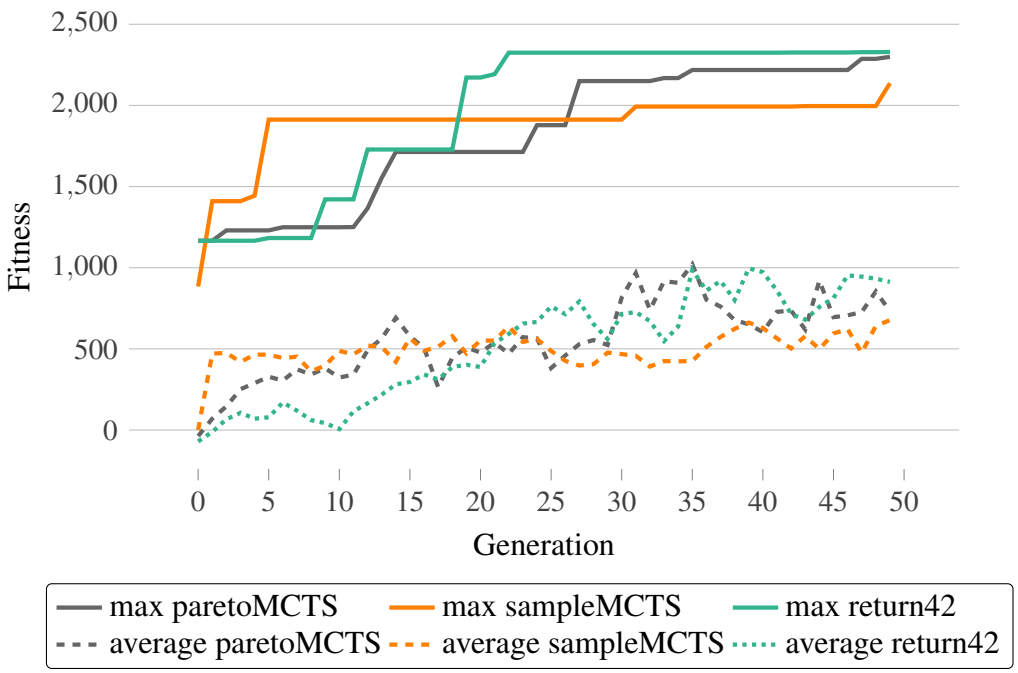
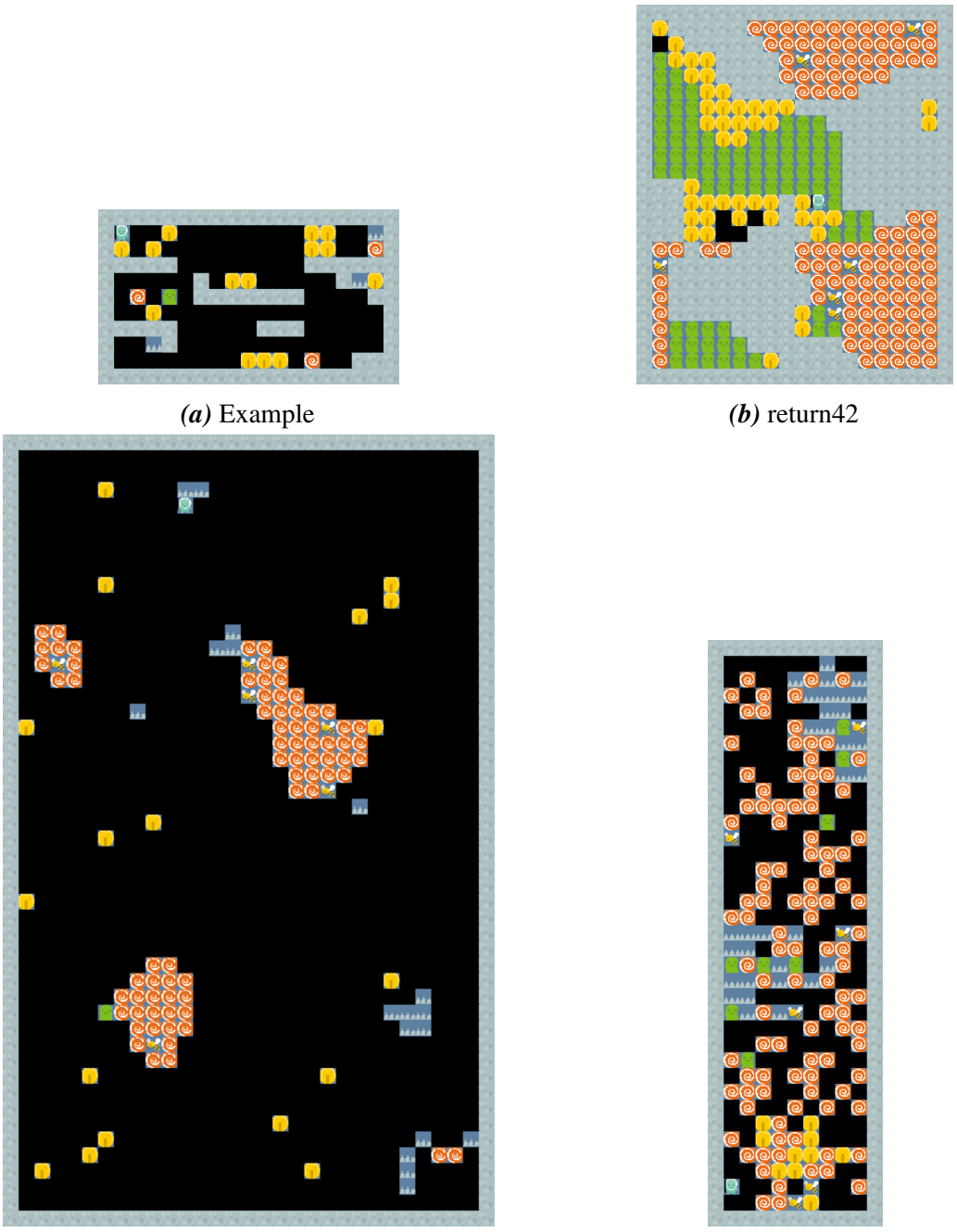


Figure A.17.: Average and maximum fitness values for the game *Survive Zombies* for all 50 generations using three different agents for the simulation.

the most difficult of all generated levels. It has way more flowers than any given level. This is probably the only reason why most agents can win this level at least a few times in comparison to the examples.



(a) Example

(b) return42

(c) paretoMCTS

(d) sampleMCTS

Figure A.18.: Generated levels for the game *Survive Zombies* and one human-made example level design for comparison.

A.1.10. Zelda

Generator	Width	Height	Victories in % (+avg. steps)			wall	floor	goal	key	nokey	monsterQuick	monsterNormal	monsterSlow	Σ
Example 1	13	9	11 (557.00)	2 (687.00)	99 (230.54)	53	58	1	1	1	0	3	0	118
Example 2	13	9	11 (557.00)	2 (687.00)	99 (230.54)	57	54	1	1	1	1	2	0	118
paretoMCTS	26	22	63 (80.52)	62 (89.44)	87 (73.61)	93	0	1	17	1	18	0	221	352
sampleMCTS	26	6	80 (190.57)	76 (340.66)	96 (51.49)	61	0	3	22	1	5	1	12	106
return42	48	17	78 (384.36)	61 (426.39)	96 (187.41)	420	0	1	7	1	2	13	14	459

Table A.10.: Statistics about two examples and three generated levels for the game *Zelda*. Victories are from 100 random runs of this following three agents: ***paretoMCTS***, ***sampleMCTS***, ***return42*** (in this order) Number of steps is the average of all won games. Last columns are the numbers of all placed sprites (excluding the avatar).

In this game, the player has to escape a dungeon. There are enemies all around the dungeon that the player can kill by using his *sword*. To win the game, the user needs to pick up a *key* and go through an exit door. The game provides a fair amount opportunities to score. Killing an enemy brings 2 points and opening the goal with a key 1 point.

It took awhile, but all variants produced a winnable level. As presented in Figure A.19, the final fitness was relatively low for all agents. Additionally, *paretoMCTS* found its best solution fairly late. This suggests that more generations could yield better solutions.

Although that goals provide a way to gather points, all of the generated levels used them rather sparse. A difference to the human made level is, that the generated levels have way more keys. A key is needed to win the game. Therefore, more keys make the game easier. The fewest keys were used by the *return42* variant. It is also the largest level. Intuitively, this game should be quite challenging. However, the test runs, as shown in Table A.10, show a slightly different picture. The difference is not that big, but the *paretoMCTS* level seems to be a little bit harder to solve. This is probably due to the enormous horde of monsters in this level. They are all slow monsters and can be killed easily to earn points.

The *sampleMCTS* level looks way simpler due to its smaller size and less enemies. Yet, the performance difference from all three tested agents was not that big. A smaller map is not necessary easier. It is harder to avoid enemies on smaller levels. Another difference between the generated levels and the examples is the use of the floor tile. None of the generated levels

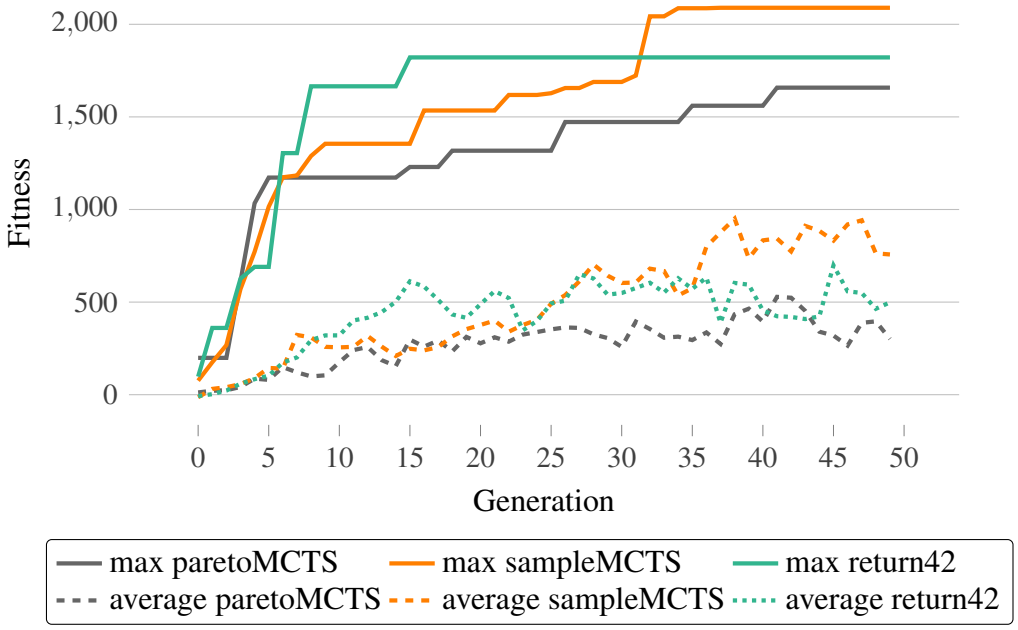
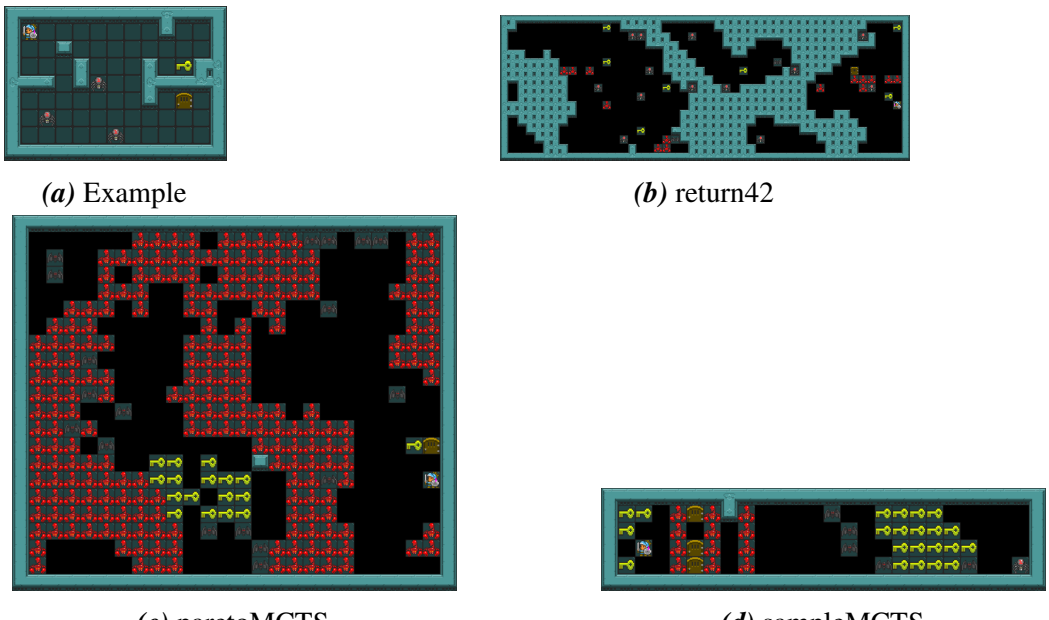


Figure A.19.: Average and maximum fitness values for the game *Zelda* for all 50 generations using three different agents for the simulation.



(a) Example

(b) return42

(c) paretoMCTS

(d) sampleMCTS

Figure A.20.: Generated levels for the game *Zelda* and one human-made example level design for comparison.

used them as a background. There is no encouragement to use them.

Overall, the levels are quite useful. The generator found a completely new interpretation of the game description. One that is easier to solve for the computer, but that is nonetheless sufficiently challenging. Subjectively, the *return42* and *paretoMCTS* levels look better than the *sampleMCTS* level, even if the used pattern for the enemies is rather useless. It would be great if this pattern had been applied to the wall. This would make the levels look more like a dungeon.

A.2. Set 2

A.2.1. Camel Race

Generator	Width	Height	Victories in % (+avg. steps)			wall	goal	randomCamel	fastR	mediumR	slowR	fastL	mediumL	slowL	Σ
Example 1	49	9	**0** (n/a)	**0** (n/a)	**56** (59.55)	110	7	3	1	1	1	0	0	0	124
Example 2	49	9	**0** (n/a)	**0** (n/a)	**56** (59.55)	131	7	0	3	1	0	0	0	0	143
paretoMCTS	43	45	**94** (523.34)	**92** (381.77)	**0** (n/a)	198	17	62	13	1	1010	0	2	142	1446
sampleMCTS	42	50	**95** (473.16)	**97** (346.71)	**0** (n/a)	382	119	329	1	81	40	1130	17	0	2100
return42	43	36	**94** (493.16)	**97** (262.57)	**0** (n/a)	790	22	44	19	3	4	11	653	1	1548

Table A.11.: Statistics about two examples and three generated levels for the game *Camel Race*. Victories are from 100 random runs of this following three agents: ***paretoMCTS***, ***sampleMCTS***, ***return42*** (in this order)
Number of steps is the average of all won games. Last columns are the numbers of all placed sprites (excluding the avatar).

In this game, the player has to win a race against multiple *camels*. Whoever first reaches a *goal* wins the game. The provided levels have many goals and usually a similar number of camels on the opposite site of the map. Multiple different camel variants can be used in this game. Some of them move randomly and some are able to run in a straight line to reach a goal faster. They also can have different speeds. Like in many other games, walls are used as obstacles.

Figure A.21 shows the results of the EA. As shown there, the generator could find a winnable solution in no time. It did not take long before a fitness plateau was reached. At the end, all variants generated levels with almost the same fitness. This is not surprising, since the game does not provide any way to collect points besides the one time event of winning.

Unlike many other games, it is enough to reach just one goal to win. The player does not need to visit each goal. As a disadvantage, this will not lead to a way to gather points like in *Frogs*. If you look at the generated levels in Figure A.22, you will wonder how such a game is winnable. However, the results in Table A.11 clearly show that almost all of these levels can be solved without a hassle with all agents but *return42*.

The overview table from the beginning also shows that a huge amount of camels were placed

in each map. *How can such a game be winnable?* The reason is simple, the game description has an error in it. It is defined that the game is won as soon as one camel reaches a goal. The emphasis is on *one*. Therefore, nothing will happen if two or more camels reach the goal at the same time. The generator could place as many camels as he liked. There must be only a minimum number of camels near a goal. This was obviously not the intention of the writer of the game description, but it explains the strange level design that was created.

The generator found a loophole and mercilessly exploited it. Thus, the generated levels are not even remotely similar to the examples. The difference is staggering. The level description should be fixed before this or any other procedural level generator can be applied again.

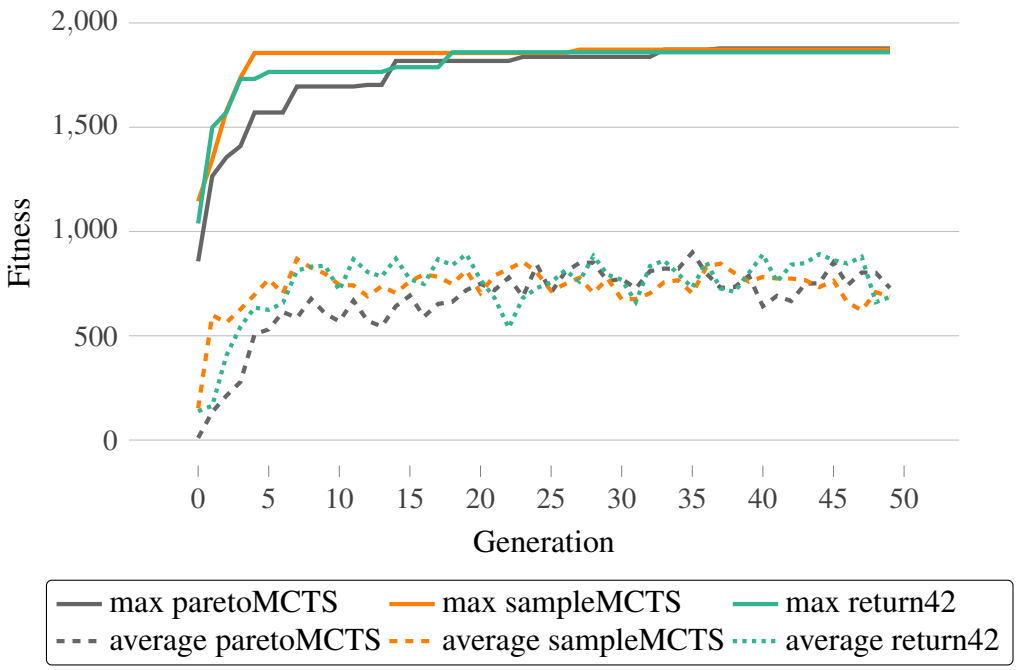
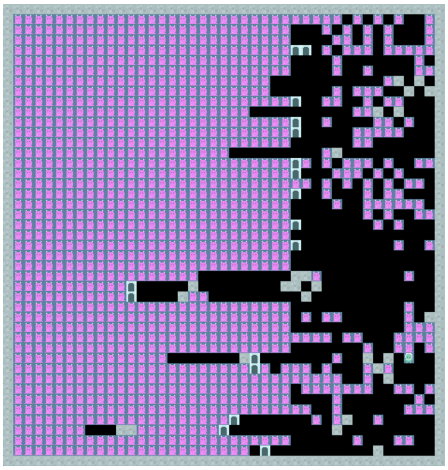
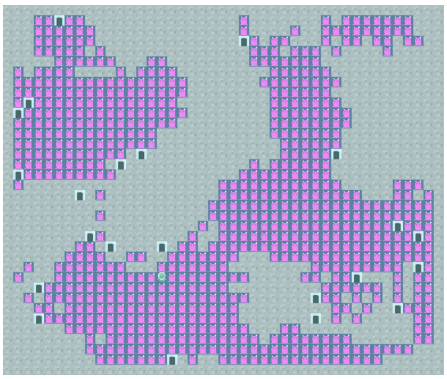


Figure A.21.: Average and maximum fitness values for the game *Camel Race* for all 50 generations using three different agents for the simulation.

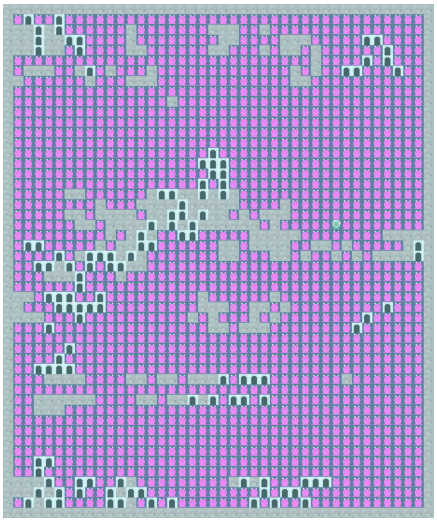
(a) Example



(b) paretoMCTS



(c) return42



(d) sampleMCTS

Figure A.22.: Generated levels for the game *Camel Race* and one human-made example level design for comparison.

A.2.2. Dig Dug

Generator	Width	Height	Victories in % (+avg. steps)				*wall*	*gem*	*gold*	*entrance*	*monster*	Σ
Example 1	28	15	**0** (n/a)	**0** (n/a)	**0** (n/a)		267	20	7	2	2	299
Example 2	28	15	**0** (n/a)	**0** (n/a)	**0** (n/a)		103	25	25	1	9	164
paretoMCTS	4	6	**90** (178.00)	**5** (389.80)	**38** (34.92)		16	2	0	1	2	22
sampleMCTS	4	4	**97** (40.65)	**99** (80.16)	**74** (9.59)		12	0	1	0	1	15
return42	9	7	**89** (297.45)	**53** (330.25)	**59** (102.59)		53	1	1	0	1	57

Table A.12.: Statistics about two examples and three generated levels for the game *Dig Dug*. Victories are from 100 random runs of this following three agents: ***paretoMCTS***, ***sampleMCTS***, ***return42*** (in this order) Number of steps is the average of all won games. Last columns are the numbers of all placed sprites (excluding the avatar).

The game *Dig Dug* takes place in a kind of dungeons created from *walls*. The player has to dig through the walls, collect *gem* and *gold* (+1 point each) and kill all *enemies* (+2 points) to win this game. The enemies are spawned from different *entrances* and are able to kill the player. The player can shoot *boulders*, that he has to collect first, to kill the enemies.

The EA results are presented in Figure A.23. For this game, all but the *paretoMCTS* needed a very long time to generate good levels. The *sampleMCTS* variant even found its best solution only at the end in the last two generations. In general, the fitness of both SO agents is disappointingly low. Whereas the MO variant could reach a fitness value of above 2021, both others just scratched on the 1500 mark.

A look at the results in Figure A.24 shows that all generated levels are very small. It can be seen that the level generator had a similar problem as for the game *Chase*. Since there are only a handful of sprite types available, the probability to place a lot of walls is quite hight. The cut off room problematic can especially be seen in the level from the *return42* variant. Since the agent has to either gather all goodies or kill any enemy, unreachable sections can make a level unwinnable. These kinds of games are hard to evolve through random mutations. Just adding or removing one wall tile on the wrong position can alter the fitness enormously. This is probably the reason why the generator preferred such small levels.

Interestingly, Table A.12 reveals that *paretoMCTS* was not only the best level regarding the fitness value, but also regarding the difficulty. Both other SO agents were not able to reliably win this level. Even if they won, they needed mostly more steps on average as for the other generated levels. This level was also the only one with an entrance in it that spawns more

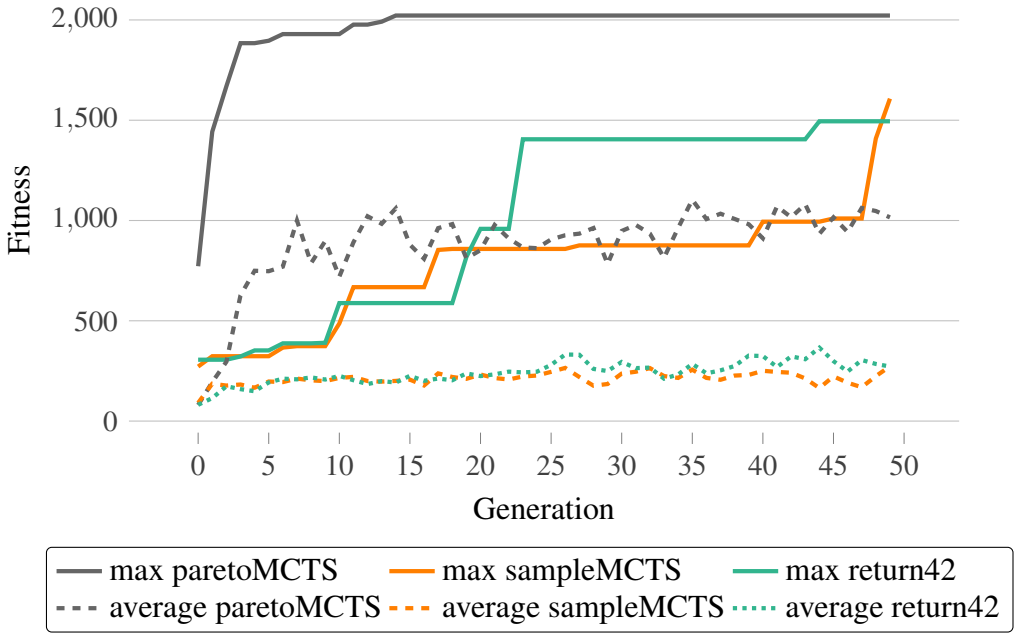
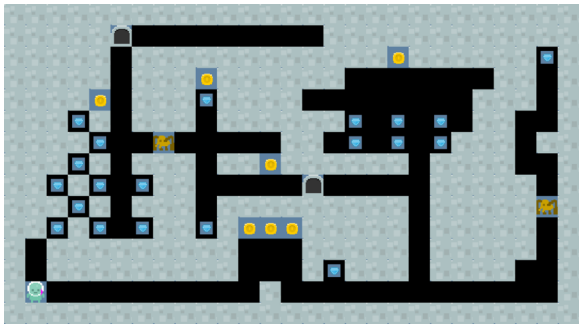


Figure A.23.: Average and maximum fitness values for the game *Dig Dug* for all 50 generations using three different agents for the simulation.



(a) Example

(b) paretoMCTS

(c) return42

(d) sampleMCTS

Figure A.24.: Generated levels for the game *Dig Dug* and one human-made example level design for comparison.

enemies. Therefore, this level is specially optimized towards the MO agent. This is especially astonishing since the level is so small, that one might suppose that the exploration objective would not be helpful.

It must also be said, that none of the agents is good at playing this game. Both example level could not be solved by any agent. Thus, it is hardly surprising that the generated levels are quite different to the human-made ones. It is nonetheless disappointing, that no larger levels were created.

A.2.3. Firestorms

Generator	Width	Height	Victories in % (+avg. steps)				*wall*	*escape*	*seed*	*water*	Σ
Example 1	29	11	**16** (577.81)	**0** (n/a)	**10** (576.20)		114	1	8	6	130
Example 2	29	11	**16** (577.81)	**0** (n/a)	**10** (576.20)		163	1	1	3	169
paretoMCTS	6	9	**81** (178.52)	**44** (140.11)	**10** (200.30)		26	8	4	12	51
sampleMCTS	7	31	**87** (358.10)	**73** (392.37)	**9** (322.56)		84	2	7	32	126
return42	16	6	**85** (247.98)	**99** (271.31)	**58** (206.31)		40	3	4	46	94

Table A.13.: Statistics about two examples and three generated levels for the game *Firestorms*. Victories are from 100 random runs of this following three agents: ***paretoMCTS***, ***sampleMCTS***, ***return42*** (in this order) Number of steps is the average of all won games. Last columns are the numbers of all placed sprites (excluding the avatar).

The player has to reach a *goal* to win this game. The challenge is to not get burned from deadly *flames* that are spawned from *portals* (called *seeds* in the game description). There is also *water* distributed around the map that the player can collect to shield him against the flames. One unit of water shields the avatar against one hit by a flame, but at the expense of 1 point. As shown in Figure A.26 in the example level, the starting position and goal is as far away from each other as possible. Furthermore, the number of water sprites corresponds approximately to the number of fire portals. As usually, walls are used as obstacles.

As the data from Figure A.25 shows, the generator had absolutely no problem in finding a winnable solution. Even a random initialisation was sometimes enough to create good enough levels. The final fitness value is relative low due to the fact that this game does not provide any way to collect points. None of the agents performed significantly better than another one. All reached a fitness value of around 2000.

Striking in this game is that the player can only get a negative score. Thus, it is not enough

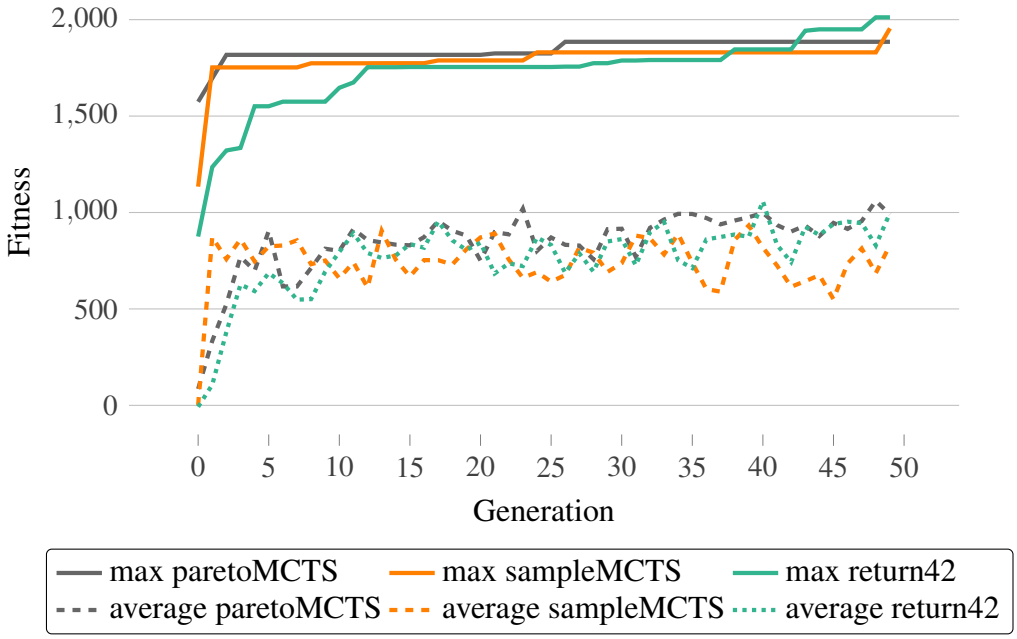


Figure A.25.: Average and maximum fitness values for the game *Firestorms* for all 50 generations using three different agents for the simulation.

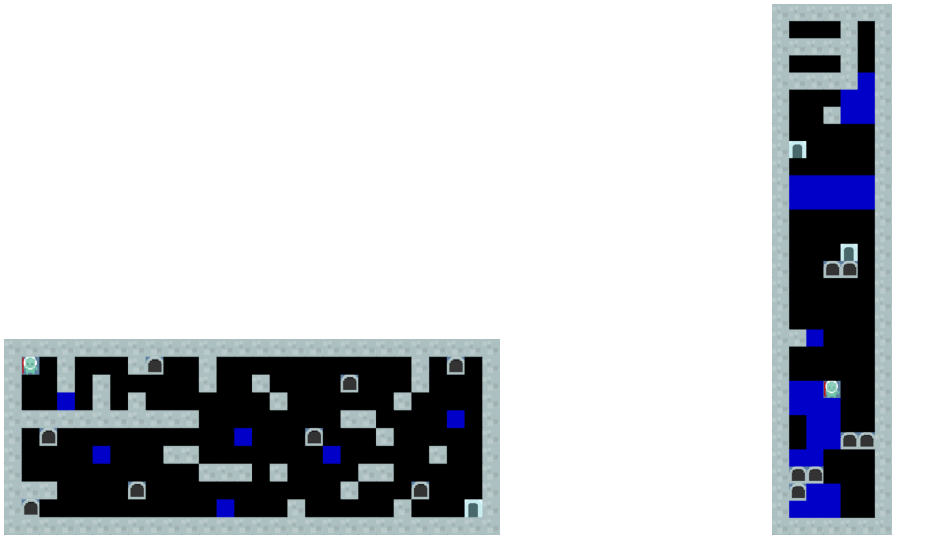
that the good agent avoids being hit by a flame, but the weak agents must be hit more often. Otherwise the score difference would be zero. This is quite a difference to a lot of other games. You also need to gather all *escapes* to win, but this time there are no points to reward this behaviour. This is probably the reason why all example levels have only one goal sprite.

As presented in Table A.13, all agents are in general pretty bad at playing this game. The *sampleMCTS* agent was not able to win a single game from the example levels. Although that *return42* was able to sometimes solve the example levels, this agent reached the worst victory rate on the generated levels. Both other agents could play these levels reasonably well.

A rather big difference between the created levels and the provided examples is the number of water tiles. The water is helpful against flames and they therefore make a level easier. The *return42* variant used the most amount of water, almost four times as many as the *paretoMCTS* level. This is also reflected in the victory rate in Table A.13. The combination of just a few flame throwers and a lot of water makes this level too easy.

As shown in Figure A.26, the *return42* level is almost completely covered with water tiles – 82% of the usable game field (i.e. the map without the pre-placed walls) is water. Both other agents have significantly less water (*sampleMCTS*: 22%; *paretoMCTS*: 42%). They are therefore more difficult.

Another property to measure the difficulty for this game is the number of escapes. There is no incentive for the agents to collect them. Therefore they only need to survive long enough and



(a) Example

(b) sampleMCTS

(c) return42

(d) paretoMCTS

Figure A.26.: Generated levels for the game *Firestorms* and one human-made example level design for comparison.

explore the map sufficiently to find them all by chance. This is were *paretoMCTS* can play to its strength. Due to the exploration objective, it was able to find all escapes before it died in almost all cases. The generated level from this agent also used the most escapes.

In the end, only *sampleMCTS* resembled the design of the example levels. This level is also sufficiently difficult, has few escapes and a good balance between water and frame spawn points. The *return42* agent for example was not able to win it in most cases. *paretoMCTS* has used its exploration ability and generated a level that is fairly challenging for the other agents.

A.2.4. Infection

Generator	Width	Height	Victories in % (+avg. steps)				*wall*	*doctor*	*host*	*virus*	*entrance*	Σ
Example 1	29	11	**2** (613.50)	**6** (678.50)	**95** (426.05)		121	4	17	6	2	151
Example 2	29	11	**2** (613.50)	**6** (678.50)	**95** (426.05)		86	1	4	6	4	102
paretoMCTS	21	43	**57** (580.82)	**57** (605.82)	**61** (556.64)		212	0	226	56	281	776
sampleMCTS	38	28	**86** (496.44)	**82** (507.51)	**81** (397.16)		199	16	366	155	27	764
return42	46	37	**93** (136.05)	**95** (146.74)	**93** (123.15)		453	23	7	383	528	1395

Table A.14.: Statistics about two examples and three generated levels for the game *Infection*. Victories are from 100 random runs of this following three agents: ***paretoMCTS***, ***sampleMCTS***, ***return42*** (in this order) Number of steps is the average of all won games. Last columns are the numbers of all placed sprites (excluding the avatar).

The goal of this game is to infect all *hosts* with a *virus*. Hosts and players can get infected by either a virus sprite or by contact with other infected hosts. There are also some *entrances* were *doctors* get spawned. A doctor is able to heal infected hosts (including the avatar). The player can use his sword to kill a doctor. The game is won as soon as all hosts are infected.

The game provides quite an amount of ways to alter the score. The player will get a bonus of 2 points for killing a doctor. However, just colliding with a doctor decreases the score by 1. Infecting other hosts gets 2 points. All level are presented in Figure A.28. Green tiles are healthy hosts and doctors are symbolised in blue. Infected hosts would be orange (not shown in any of the presented levels). Having a lot of hosts, especially in the narrowest space, makes the game easier since the infection can be directly transmitted from host to host. Having more viruses also makes it easier to infect everyone.

The generated levels have a pretty low fitness value as shown in Figure A.27. A fitness plateau was found after around 20 generations. The *paretoMCTS* has found a solution with the highest fitness. The difference in the end is rather marginal.

As presented in Table A.14, the *paretoMCTS* variant produced the most difficult level. The balance between the number of doctors and hosts defines the difficulty of a level. Although that no doctor was directly placed in this level, there are a lot of spawn points that continuously output new ones. The images of the level in Figure A.28 do not show the entrances, but the doctors. This is due to the fact that the screenshots were made at step one of the level. At this very moment, all the doctors were spawned.

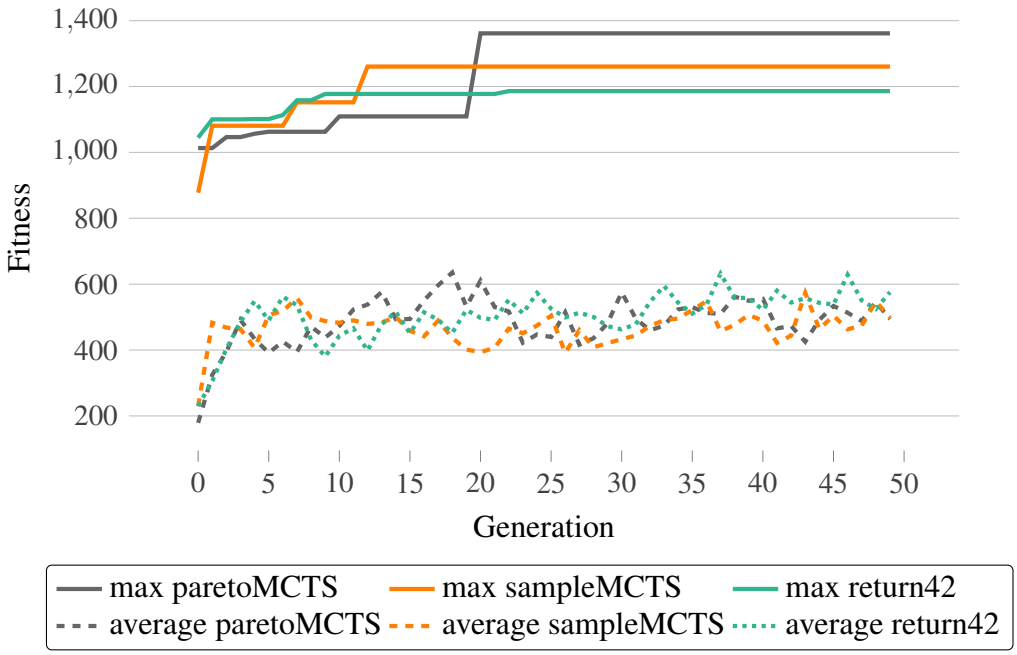
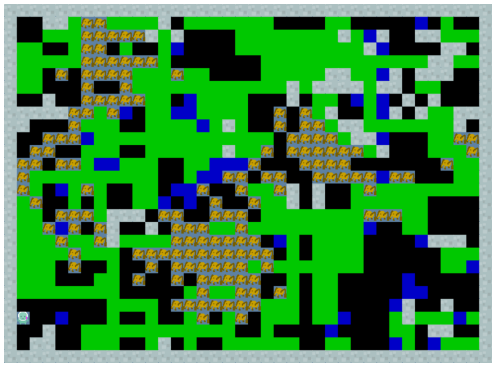


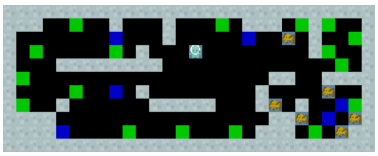
Figure A.27.: Average and maximum fitness values for the game *Infection* for all 50 generations using three different agents for the simulation.

The generated levels do not have that much in common with the provided examples. They have more of everything – more walls, more doctors, way more hosts, viruses, entrances, ... Therefore, the generator was not able to find a well balanced solution. The *return42* variant produced the easiest level. It has only seven hosts and a ton of viruses. The fact that it also had the most entrances did not help for so few hosts. All agents were able to infect them all in less than 150 steps on average. The *sampleMCTS* level has the least amount of entrances and the most hosts. Although that this level is a little bit more difficult to solve for the agents, it has probably to many hosts. The agent has hardly anything to do. The hosts will infect each other without any agent intervention.

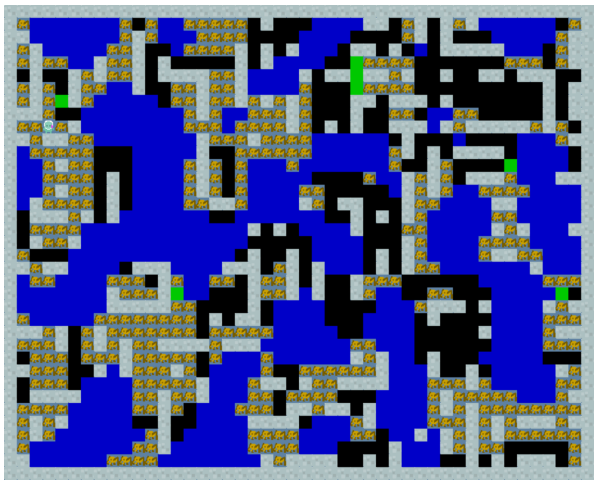
Overall, the generated levels are quite different to the human-made ones. Nonetheless, playable and winnable games with sufficient difficulty could be created with the proposed generator. The levels all look quite chaotically, but they are not that different from the examples. They all have a similar structured wall placement.



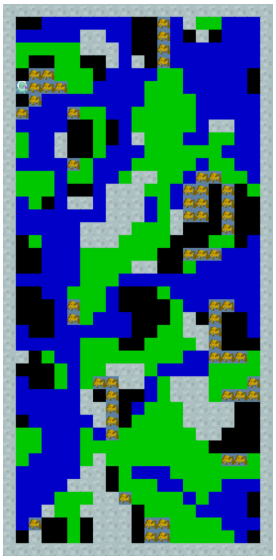
(a) sampleMCTS



(b) Example



(c) return42



(d) paretoMCTS

Figure A.28.: Generated levels for the game *Infection* and one human-made example level design for comparison.

A.2.5. Firecaster

Generator	Width	Height	Victories in % (+avg. steps)				wall	goal	box	mana	Σ
Example 1	20	11	0 (n/a)	0 (n/a)	0 (n/a)		56	1	64	10	132
Example 2	20	11	0 (n/a)	0 (n/a)	0 (n/a)		56	1	48	4	110
paretoMCTS	44	8	92 (219.59)	100 (146.77)	100 (20.51)		111	1	0	237	350
sampleMCTS	10	13	66 (465.20)	79 (394.68)	99 (61.83)		42	2	16	13	74
return42	19	29	82 (345.40)	87 (346.72)	100 (58.26)		241	1	66	107	416

Table A.15.: Statistics about two examples and three generated levels for the game *Firecaster*. Victories are from 100 random runs of this following three agents: ***paretoMCTS***, ***sampleMCTS***, ***return42*** (in this order) Number of steps is the average of all won games. Last columns are the numbers of all placed sprites (excluding the avatar).

This is another game in which the player has to reach a *goal*. However, in this case, the way to the goal is blocked by wooden *boxes*. The player can burn down the boxes by shooting at them. Shooting needs *mana* that must be collected beforehand. Collecting mana and destroying boxes are rewarded with 1 point. Flames can spread from one box to another, but the player must be careful not to get burned himself. He has a certain number of health points that gets decreased by each hit from a flame. The player loses as soon as the health reaches zero. Furthermore, each hit by a flame costs the player 2 points.

Figure A.29 shows the result of the EA. All variants produced, more or less, levels with the same fitness. Only a few generations were needed to create winnable levels. In the end, the fitness were around 1700 for all agents.

Normally, the way to the goal is blocked by multiple boxes. The player has to first burn them down before he can reach the goal. A look at Figure A.30 reveals, that this is not true for the generated levels. In all cases, the goals can be reached immediately. Important to know is, that all goals must be visited by the avatar. The example levels have always one goal, but the game description has no hardcoded limit. However, visiting a goal has no immediate advantage. This game does not provide any points for this. Having multiple goals is therefore a disadvantage since almost all agents will not be able to "learn" that a goal is useful. Only the *sampleMCTS* agent had two goals placed. It can be seen in Table A.15 that this is probably also the most difficult level.

The easiest generated level is from the *paretoMCTS* agent. There is not a single challenge in it. The player can only get a lot of points from all the mana. The only reason why this agent is

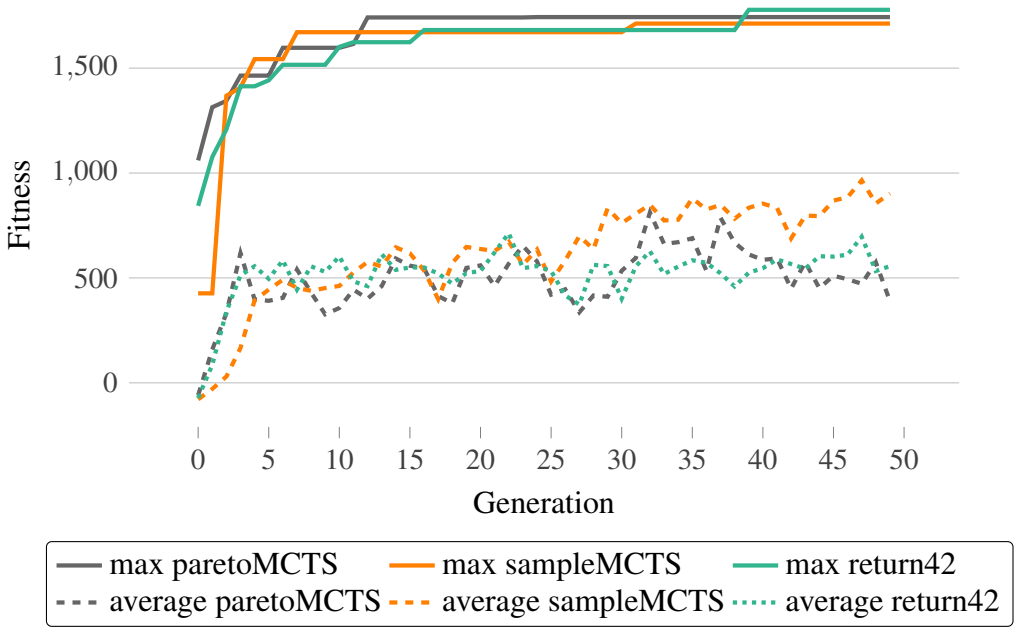
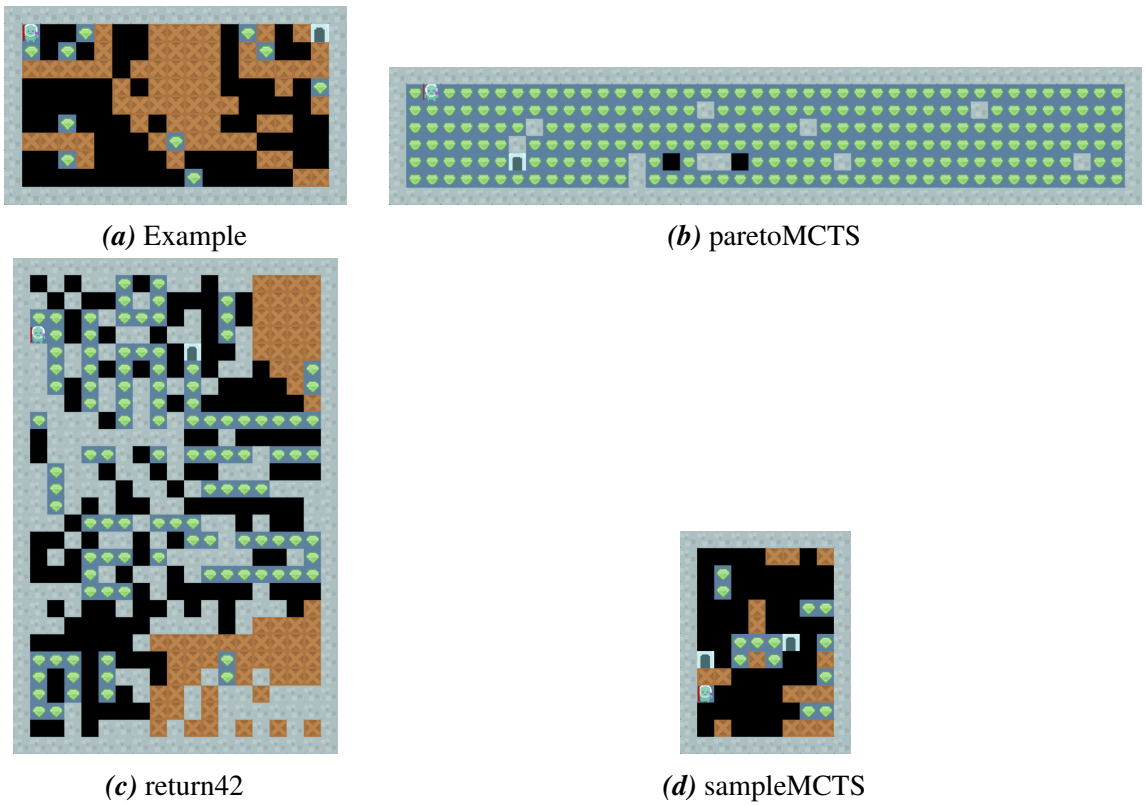


Figure A.29.: Average and maximum fitness values for the game *Firecaster* for all 50 generations using three different agents for the simulation.



(a) Example

(b) paretoMCTS

(c) return42

(d) sampleMCTS

Figure A.30.: Generated levels for the game *Firecaster* and one human-made example level design for comparison.

better than the weaker agents (used for the fitness function), is probably that the exploration objective forced him to visit more places. Therefore, he did not run directly towards the goal and had enough possibilities to collect mana. The *return42* level also does not force the player to burn down boxes, but at least it has a lot of normal walls that block the way to the goal. Since this agent uses, amongst other things, the A^* algorithm to find its way to the goal, it has the least problems in solving such a level.

Overall, none of the generator variants could catch the spirit of the game – burning down boxes to free the way towards the goal. *return42* provided at least some challenges by building up a kind of maze to make it more difficult to get to the goal. The *paretoMCTS* variant produced a rather bad level. It is playable and winnable, but does not provide any challenge. Even a random agent could win this level.

A.2.6. Overload

Generator	Width	Height	Victories in % (+avg. steps)				*wall*	*goal*	*marsh*	*gold*	*random*	*weapon*	Σ
Example 1	20	11	**0** (n/a)	**0** (n/a)	**100** (223.09)		85	1	6	20	1	1	115
Example 2	20	11	**0** (n/a)	**0** (n/a)	**100** (223.09)		57	1	9	66	3	1	138
paretoMCTS	24	33	**63** (241.29)	**59** (291.64)	**100** (59.32)		114	1	8	406	42	0	572
sampleMCTS	42	11	**75** (232.07)	**76** (222.33)	**100** (39.07)		112	1	41	120	1	143	419
return42	16	6	**33** (237.55)	**93** (332.16)	**8** (40.88)		40	1	28	15	0	0	85

Table A.16.: Statistics about two examples and three generated levels for the game *Overload*. Victories are from 100 random runs of this following three agents: ***paretoMCTS***, ***sampleMCTS***, ***return42*** (in this order) Number of steps is the average of all won games. Last columns are the numbers of all placed sprites (excluding the avatar).

Once again the player must reach a *goal*. This time he has to collect a certain number of *gold* in advance to win. However, if he collects more gold than a second certain number defines, then he can get vulnerable on *marsh* tiles. The avatar can also collect *weapons*. Doing so will give him 2 bonus points. The weapon can be used to destroy marsh. Additionally, there are randomly moving NPCs (called *random* in the game description) that are also able to gather the gold.

As presented in Figure A.31, the generated levels all have a fairly low fitness. A lot of generations were needed to find suitable levels. The *paretoMCTS* and *return42* variant both found their best solution relatively late. This indicates, that a longer evaluation might result in

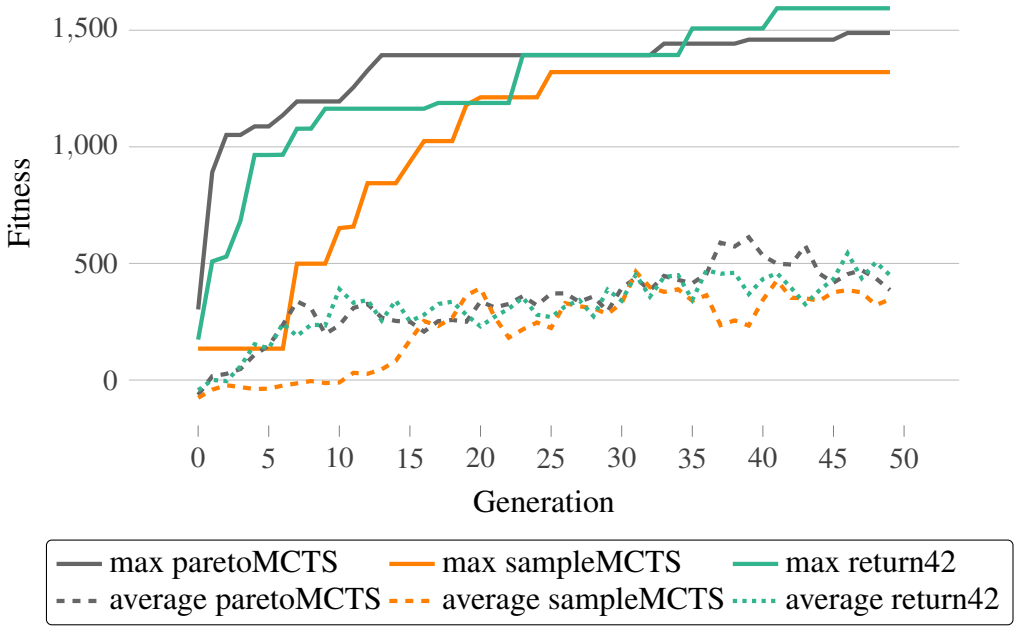
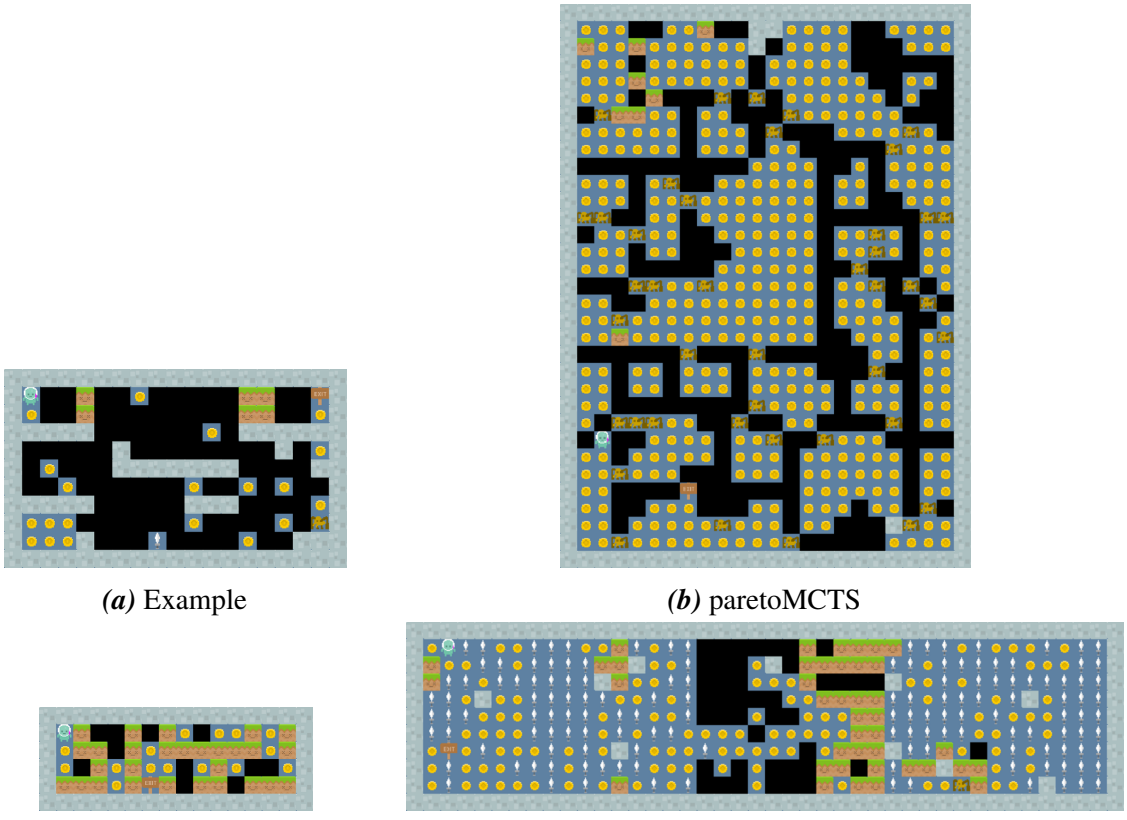


Figure A.31.: Average and maximum fitness values for the game *Overload* for all 50 generations using three different agents for the simulation.



(a) Example

(b) paretoMCTS

(c) return42

(d) sampleMCTS

Figure A.32.: Generated levels for the game *Overload* and one human-made example level design for comparison.

better levels.

Most of the generated levels, as shown in Figure A.32, have a lot more gold than the example levels, especially the *paretoMCTS* variant. This agent has a big advantage in such a case compared to the other used agents. Its exploration objective helps him to find the gold even in the furthest corners of the map. This is also the reason why this variant generated the largest levels. Interestingly, this is also the only level with a lot of NPCs. However, as a compensation for this, this level also has the most gold coins, which lowers the difficulty of the level.

A look in Table A.16 reveals, that the level from *return42* is the most difficult for all tested agents. This level has exactly the right amount of gold. It is enough to reach the goal and enough to trigger the changes that make the marsh deadly. They are also a lot of marsh tiles between the avatar and the goal. The player has to go through marsh to reach the goal. Striking is, that the same agent is not even that good at playing its own generated game. This suggests that the used simple agents for the fitness function had even more problems solving this level.

The level produced with *sampleMCTS* has a lot of gold and a relatively low number of marsh tiles compared to its size. Additionally, it is the only level with a lot of weapons. This combination makes the level very easy. It is also pretty easy to get a high score in this level due to all the weapons. Using the weapon increases the score even further. And due to all the weapon power, collecting more gold coins is relatively harmless.

In contrast to a game like *Frogs*, this game does not reward visiting a goal. Therefore, it is not surprising that at least the number of goals is the same as in all example levels. In the end, the *return42* variant is closest to what a designer would create. It is fairly challenging and has a good balance of all important game elements. If it was a little bit larger and one or two NPCs more would make this level perfect. It is a shame that the other two agents are too bad in this game.

A.2.7. Pacman

Generator	Width	Height	Victories in % (+avg. steps)				*wall*	*power*	*pellet*	*hungry*	*redspawn*	*orangespawn*	*bluespawn*	*pinkspawn*	*fruit*	Σ
Example 1	29	31	**0** (n/a)	**0** (n/a)	**0** (n/a)		546	4	237	1	1	1	1	1	4	797
Example 2	29	31	**0** (n/a)	**0** (n/a)	**0** (n/a)		636	4	166	1	1	1	1	1	1	813
paretoMCTS	39	9	**73** (194.82)	**85** (281.60)	**40** (584.42)		264	9	0	1	14	60	2	1	0	352
sampleMCTS	8	49	**0** (n/a)	**73** (425.14)	**93** (291.18)		110	2	1	1	0	1	0	0	0	116
return42	21	20	**0** (n/a)	**0** (n/a)	**77** (634.57)		79	146	105	1	7	0	2	5	28	374

Table A.17.: Statistics about two examples and three generated levels for the game *Pacman*. Victories are from 100 random runs of this following three agents: ***paretoMCTS***, ***sampleMCTS***, ***return42*** (in this order) Number of steps is the average of all won games. Last columns are the numbers of all placed sprites (excluding the avatar).

In this game, the player has to collect all goodies such as *pellets*, *fruits* and *power pills* to win the game. Collecting them will increase the score by 1, 5 and 10 respectively. He also has to avoid *ghosts* while he gets all these items. The ghosts are deadly as long as the player did not eat a power pill. The pill works only for a certain time. The player can kill a ghost in this time frame. Killing a ghost gets rewarded with 40 points. Ghost are spawned from different spawn points.

The results from the EA are shown in Figure A.33. All generator variants were able to produce winnable levels. A fitness plateau was found fairly fast for both SO agents. The *paretoMCTS* variant needed some more generations, but the final fitness values are almost the same. The many ways to gain score are reflected in the relative high fitness values.

Have a look at Table A.17 to see the experimental results. As you can see there, none of the agents is very good at this game. They all lose the game every single time. Have a look at the example level in Figure A.34 to see the reason for this. The difficulty in this game is the limited freedom of movement. Only a small path can be entered by the agent. Furthermore, this path is build up like a maze. The deadly ghost can come from each side, sometimes at the same time. They are normally also slightly faster than the avatar which makes it even more difficult to run away.

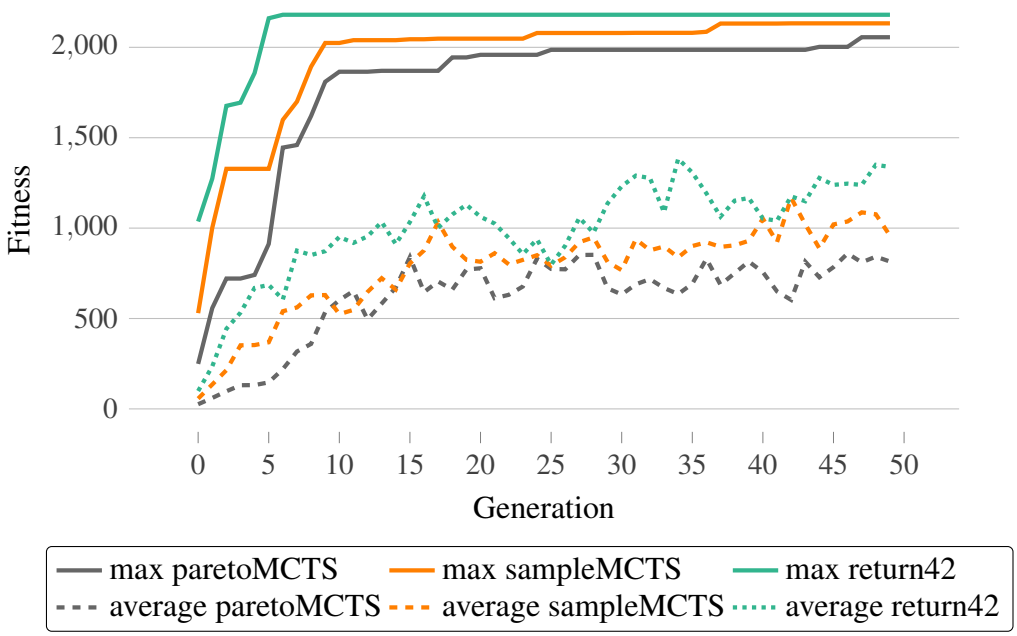
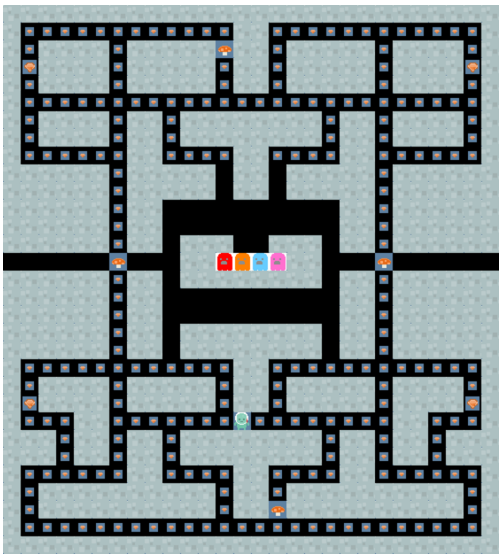
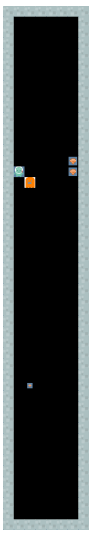


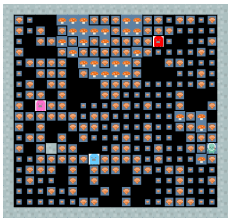
Figure A.33.: Average and maximum fitness values for the game *Pacman* for all 50 generations using three different agents for the simulation.



(a) Example



(b) sampleMCTS



(c) return42

(d) paretoMCTS

Figure A.34.: Generated levels for the game *Pacman* and one human-made example level design for comparison.

None of the generated levels look like the provided, especially the typical maze-like structure is not there. The *return42* variant used a lot of goodies. It is easy to survive the ghosts with all those power pills. There are also no obstacles that the player must avoid. He can easily flee from any ghost.

The *sampleMCTS* level is easy to solve for a completely different reason. There is only one goody to gather. The long hose-like structure of this level makes it a little bit more difficult to avoid the ghosts. This is probably the reason why the weaker agent could not win the game.

The results for this game are disappointing. It does not really come unexpected, since none of the agents can play the original levels.

A.2.8. Seaquest

Generator	Width	Height	Victories in % (+avg. steps)				*wall*	*sky*	*sharkhole*	*whalehole*	*normaldiverhole*	*offendiverhole*	Σ
Example 1	22	9	**0** (n/a)	**0** (n/a)	**0** (n/a)	_ _ _	0	21	3	3	2	0	30
Example 2	22	9	**0** (n/a)	**0** (n/a)	**0** (n/a)	_ _ _	0	21	6	0	0	2	30
paretoMCTS	10	10	**0** (n/a)	**0** (n/a)	**0** (n/a)	_ _ _	0	34	1	0	0	64	100
sampleMCTS	4	5	**0** (n/a)	**0** (n/a)	**0** (n/a)	_ _ _	0	1	0	0	0	18	20
return42	7	5	**0** (n/a)	**0** (n/a)	**0** (n/a)	_ _ _	0	1	0	0	0	33	35

Table A.18.: Statistics about two examples and three generated levels for the game *Seaquest*. Victories are from 100 random runs of this following three agents: ***paretoMCTS***, ***sampleMCTS***, ***return42*** (in this order) Number of steps is the average of all won games. Last columns are the numbers of all placed sprites (excluding the avatar).

The player controls a submarine in this game. He must avoid getting killed by animals such as *whales* and *sharks*. Another danger is to have too few *air*. The submarine must turn up to the surface to refill the air before the player suffocates. The players submarine also has capacity for 4 divers that he can rescue. Each rescued diver gets rewarded with 1000 points. He can also shoot down an animal. This action gets rewarded with 1 point. The game is won after 1000 steps.

Due to all the points that can be collected in this game, the fitness of a level gets quite high

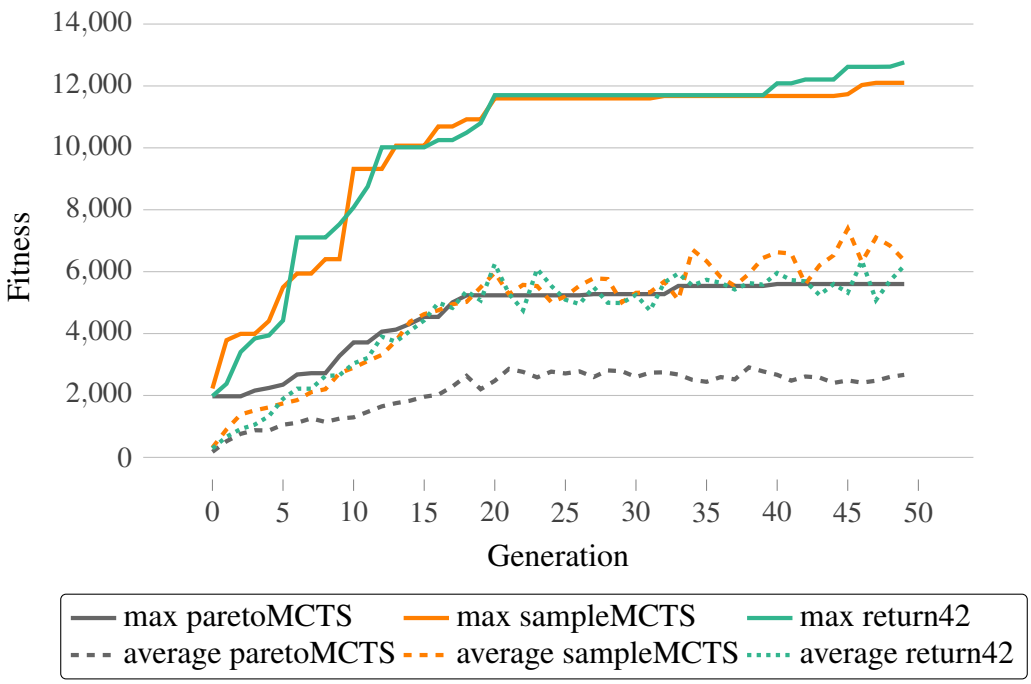
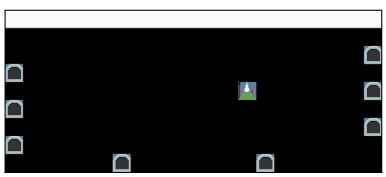


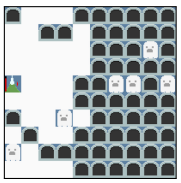
Figure A.35.: Average and maximum fitness values for the game *Seaquest* for all 50 generations using three different agents for the simulation.

as shown in Figure A.35. This is one of the games, where it is hard to differentiate between a high score and a high win rate. Nonetheless, it can be seen that both SO agents reached an extremely high fitness value. The fitness of the generated level from *paretoMCTS* is also pretty high. The difference, however, between both approaches is remarkable. This suggests that the *paretoMCTS* agent is rather bad at playing this game.

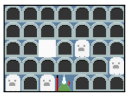
A look at Table A.18 reveals, that none of the tested agents are able to play this games. Not a single level was won. Interestingly, the fitness value was still high. The reason for this is simple. The points for rescuing a diver are so high, that the generator optimized the level towards this goal.



(a) Example



(b) paretoMCTS



(c) return42

(d) sampleMCTS

Figure A.36.: Generated levels for the game *Seaquest* and one human-made example level design for comparison.

As shown in Figure A.36, all of the generated levels have a lot of spawn points for divers (the portal-like objects on the map). The divers must be brought to the *sky* (white rectangle) by the avatar. Since the sky is always very close to the spawn points, the avatar does not even need to move that far. Due to the sheer amount of available divers, the score just continues to skyrocket.

The *paretoMCTS* agent produced as the only one spawn points for animals. Killing an animal is only rewarded with 1 point – i.e. 0.1% of what a rescue mission brings in. Additionally, each animal increases the chance to die. Therefore, the generator has no incentive to place spawnpoints. This level can not be created with the proposed algorithm. At least it needs to overhaul the balance of the different rewards. The algorithm itself would need some information or mechanism to avoid placing diver spawn points and sky tiles close together.

A.2.9. Whackamole

Generator	Width	Height	Victories in % (+avg. steps)				*wall*	*wide*	*tight*	*cat*	Σ
Example 1	15	5	**34** (500.00)	**94** (500.00)	**98** (500.00)		44	16	8	1	70
Example 2	15	5	**34** (500.00)	**94** (500.00)	**98** (500.00)		61	3	3	2	70
paretoMCTS	43	37	**59** (500.00)	**70** (500.00)	**70** (500.00)		337	747	7	9	1101
sampleMCTS	4	19	**37** (500.00)	**83** (500.00)	**95** (500.00)		48	0	0	1	50
return42	21	6	**40** (500.00)	**78** (500.00)	**96** (500.00)		62	4	0	2	69

Table A.19.: Statistics about two examples and three generated levels for the game *Whackamole*. Victories are from 100 random runs of this following three agents: ***paretoMCTS***, ***sampleMCTS***, ***return42*** (in this order)
Number of steps is the average of all won games. Last columns are the numbers of all placed sprites (excluding the avatar).

Here, the player must try to catch *moles* that periodically pop out of *tight* or *wide* portals. Additionally, there is a *cat* that also tries to catch moles. If the player catches a mole, then he scores 1 point. The player loses the game as soon as the cat collides with the avatar. The player has to manage to stay alive for a certain amount of time steps. It is worth mentioning here, that the walls are no obstacles like in many other games. They are simple background tiles.

The EA results are presented in Figure A.37. All generator variants were able to produce winnable levels. The difference between both SO agents and the MO agent is pretty significant. It can therefore be concluded that the *paretoMCTS* agent is worse at playing this game. The

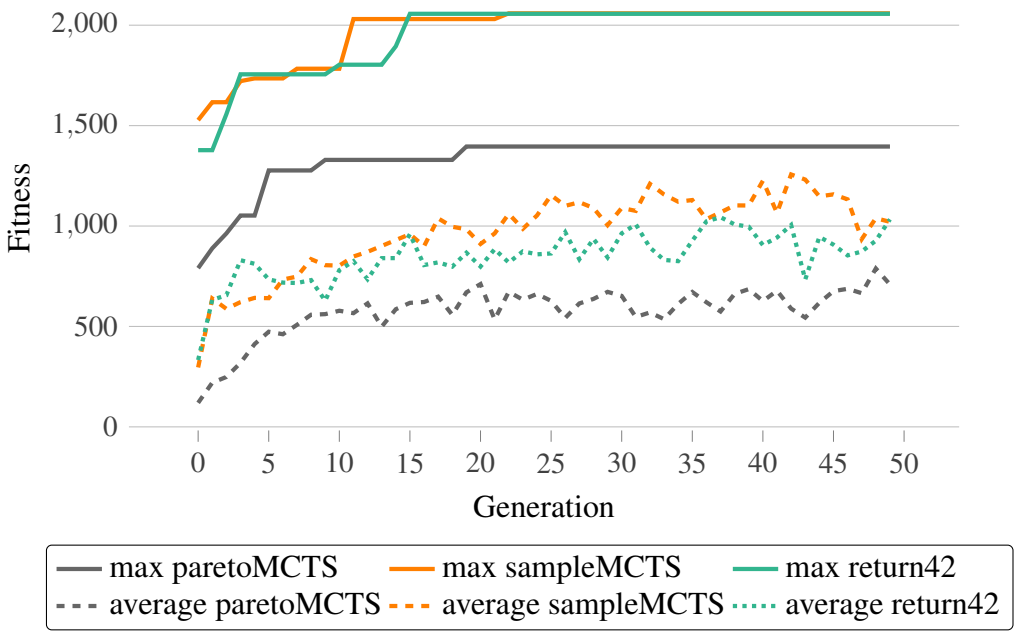
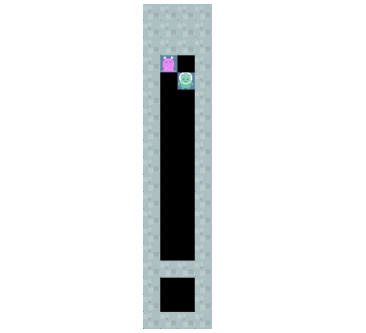


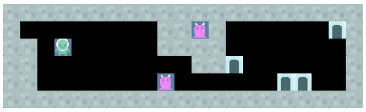
Figure A.37.: Average and maximum fitness values for the game *Whackamole* for all 50 generations using three different agents for the simulation.

results from Table A.19 confirm the assumption. The example levels can be found by almost both SO agents without bigger problems.

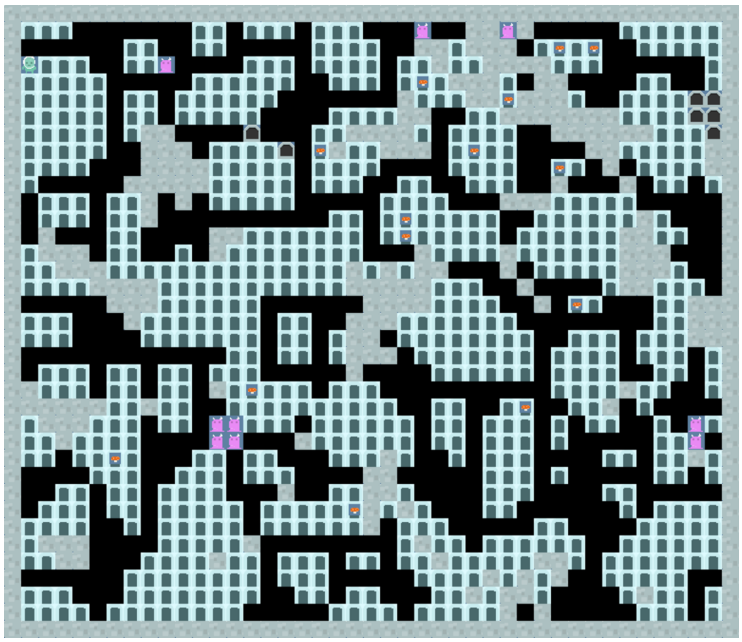


(a) sampleMCTS

(c) Example



(d) return42



(b) paretoMCTS

Figure A.38.: Generated levels for the game *Whackamole* and one human-made example level design for comparison.

Two surprising things can be seen in the generated levels that are presented in Figure A.38. First, the *sampleMCTS* level has not a single spawn point. The only task for the player here is to just keep out of the cat's way and wait until the end of the game. This is a task, where the *paretoMCTS* controller has a disadvantage. His exploration objective forces him to constantly stay in motion. Other agents could crawl in a corner and just wait.

The other surprising finding is, that the *paretoMCTS* level is enormously larger than both other levels. It has a lot of mole spawn points and overall nine cats. This makes the level a little bit challenging. Overall, the SO agents lost this level more than any other (including the examples). However, the MO agent performed best here. This level is therefore fitted towards a MO agent or at least an agent that incorporates exploration in their evaluation.

Finally, all levels are pretty different from the provided ones. The *sampleMCTS* variant is way too easy and misses the point of the game. The level from *return42* looks quite reasonable. At last, the *paretoMCTS* level is the most difficult level. Only the *paretoMCTS* agent itself could handle it well. It also has kind of acceptable structure. It is a pity that the generator does not recognize the wall sprites as a background tile.

A.2.10. Eggomania

Generator	Width	Height	Victories in % (+avg. steps)				*wall*	*slowChicken*	*fastChicken*	*trunk*	Σ
Example 1	33	14	**0** (n/a)	**2** (292.50)	**42** (549.71)		88	1	0	30	120
Example 2	33	14	**0** (n/a)	**2** (292.50)	**42** (549.71)		88	1	1	30	121
paretoMCTS	8	43	**82** (394.11)	**89** (370.47)	**93** (461.97)		98	190	1	0	290
sampleMCTS	7	5	**44** (413.32)	**100** (452.02)	**66** (421.97)		23	4	0	0	28
return42	9	10	**41** (334.07)	**77** (161.38)	**57** (229.07)		34	11	3	5	54

Table A.20.: Statistics about two examples and three generated levels for the game *Eggomania*. Victories are from 100 random runs of this following three agents: ***paretoMCTS***, ***sampleMCTS***, ***return42*** (in this order) Number of steps is the average of all won games. Last columns are the numbers of all placed sprites (excluding the avatar).

The player in this game has to try to catch all *eggs*. Catching an egg is rewarded with 1 point. Eggs are fragile and break as soon as they collide with the floor (*wall*). The player loses the game if this happens. One or more *chicken* are on top of the level and throw their eggs down. The player can only move horizontal at the bottom of the map. When he managed

to catch enough eggs, he can shoot back with them at the chicken and get 100 points per hit. Furthermore, the game is won as soon as all chickens were hit.

All generators were able to create winnable levels fairly fast as shown in Figure efplot:exp:result:eggomania. After less than 10 generations, each variant found a fitness plateau. A marginal improvement could only be found by *sampleMCTS* in generation 36. All of the variants had almost the same final fitness (slightly below 2200).

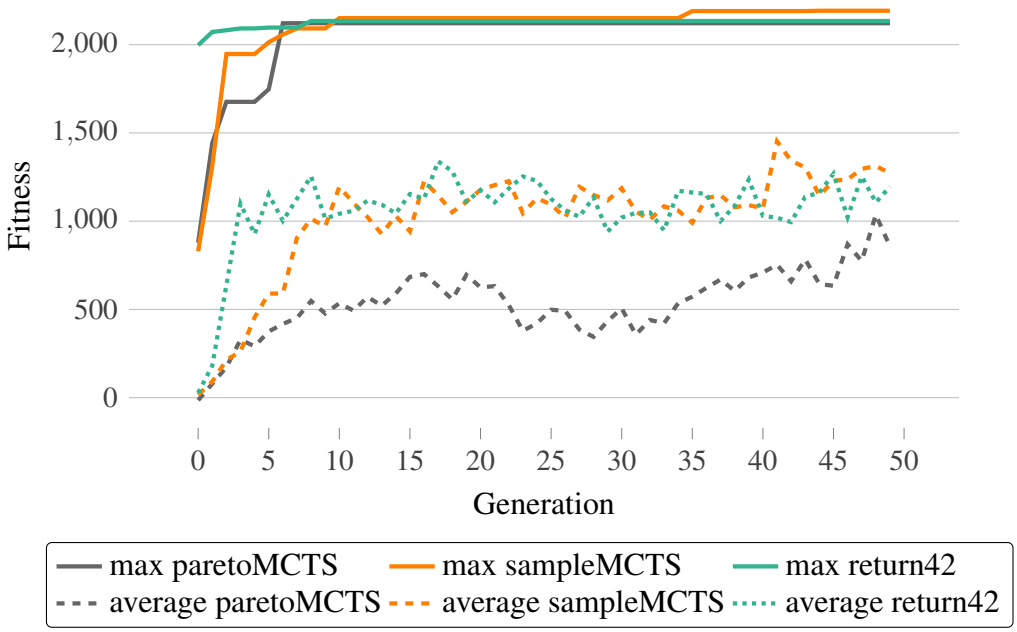


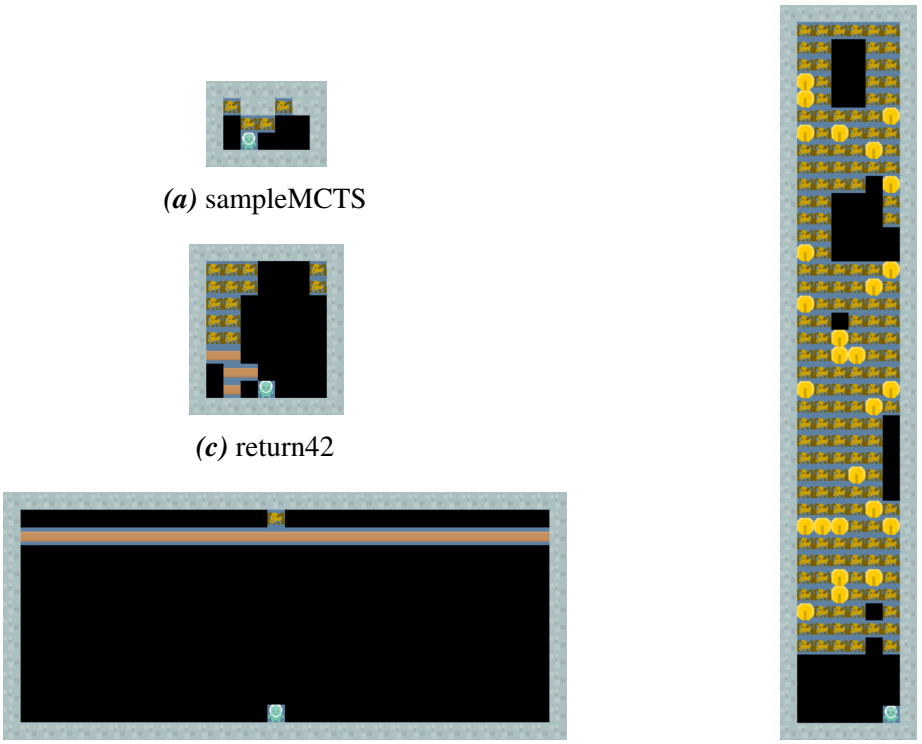
Figure A.39.: Average and maximum fitness values for the game *Eggomania* for all 50 generations using three different agents for the simulation.

Have a look at the levels in Figure A.40. The one example level shown and the other four provided examples look extremely similar. Normally, the chickens sit on a trunk made up of single brown logs. None of the generated levels could replicate this aspect. There is simply no single incentive of setting these logs. However, this does not influence the game in any way.

Broader levels increase the difficulty since it would be harder to catch an egg that falls down far away from the player's current position. The opposite is true for the height of an level. Higher levels give the player more time to react on falling down eggs. The largest level was generated with the *paretoMCTS* agent, but this one is rather narrow. Indeed, the results from the test runs confirm this assumption. As presented in Table A.20, the *paretoMCTS* level was relatively easy to win for all agents, despite the fact that this level has the most chickens. The most challenging level was created with the *return42* agent. It has a fair amount of chickens and is quite small.

None of the generated levels used any walls (besides the default border). Placing a wall in this game is not an easy task. The avatar must be either above the wall or the chicken should not be able to move that far right or left that their eggs could hit this wall. The generator handled the positioning of all other sprites very well due to the placing constraints that are encoded into the Likeliness-Matrices. The avatars starting position is always on the bottom and the chickens are at the top.

Overall, none of the levels could replicate the aesthetical aspect of the example levels. Nonetheless, all variants were able to produce winnable levels and placed all sprites on the correct positions. The levels are probably all a little bit too easy, especially the *paretoMCTS* level focuses too much on collecting points.



(a) sampleMCTS

(c) return42

(d) Example

(b) paretoMCTS

Figure A.40.: Generated levels for the game *Eggomania* and one human-made example level design for comparison.

APPENDIX CHAPTER B

Interim Experimental Results

B.1. Experiment - Fitness Reliability

As shown in Section 5.1.1 a single run of the fitness function results in unreliable fitness values. This section will show and test some improvements to further stabilize the outcome of the fitness function.

Yet, before any improvements can be tested, a method to measure and compare the quality of the reliability of the fitness function is needed. For this, the *Standard Error of the Mean (SEM)* will be used [AB05]. SEM is the standard deviation of the sample-mean's estimate of a population mean [Eve02]. That is:

$$\mathrm{SE}_{\bar{x}} = \frac{\sqrt{\sigma_1^2 + \sigma_2^2 + \ldots + \sigma_{10}^2}}{\sqrt{N}}$$

where N is the number of samples and σ_n the variance of the nth individual. That means that the variances of all fitness values from each individual are averaged and the square root from this is then the average standard deviation of the whole population. At last, a normalization according to the population size is done.

#	Aliens	Bolo Adventures	Eggomania	Survive Zombies
1	88.26	71.09	27.27	111.00
3	51.78	84.34	16.07	101.56
5	57.84	81.35	15.59	102.68
7	42.05	78.25	15.28	92.34

Table B.1.: Standard errors of four games with different numbers of simulation passes using a simple average method.

Have a look at Table B.1 for an overview of the results. As assumed, in most cases more passes means a lower standard error. Unfortunately, it is also clear that increasing the number of runs will not be enough. Particularly with regard to the extra required computation time.

The first possible improvement is to ignore extreme outliers. Ignoring the min and max fitness values of all runs and averaging the rest should result in a more stable fitness function. This is usually called a *centred average*. A disadvantage is that now at least four passes are needed. From the previously gained knowledge we know that averaging more values should result in a more reliable outcome. To be able to better compare the results, this test uses five and seven passes. All results can be found in Appendix B.1.2. Again, the outcome is rather mixed. Nonetheless, as demonstrated in Table B.2, the results are slightly better than before, especially when using five passes. The improvement of more passes are negligible.

Still striking are the large jumps in between the fitness values. This is due to the *feasible*-bonus

#	Aliens	Bolo Adventures	Eggomania	Survive Zombies
5	46.33	80.41	13.28	95.86
7	43.16	78.97	12.35	96.32

Table B.2.: Standard errors of fitness values from four games using a centred average.

when an agent wins a game. To minimize this effect and further stabilize the fitness value, this bonus will now be awarded proportionally. This means, when, for example, five simulation passes are used and the agent wins only in two of the five cases, the *feasible*-bonus will be 2/5 of the actual value.

#	Aliens	Bolo Adventures	Eggomania	Survive Zombies
5	46.44	33.74	22.14	65.79
7	40.21	29.33	19.72	61.96

Table B.3.: Standard errors of fitness values from four games with five simulation passes using a centred average and an averaged feasible bonus.

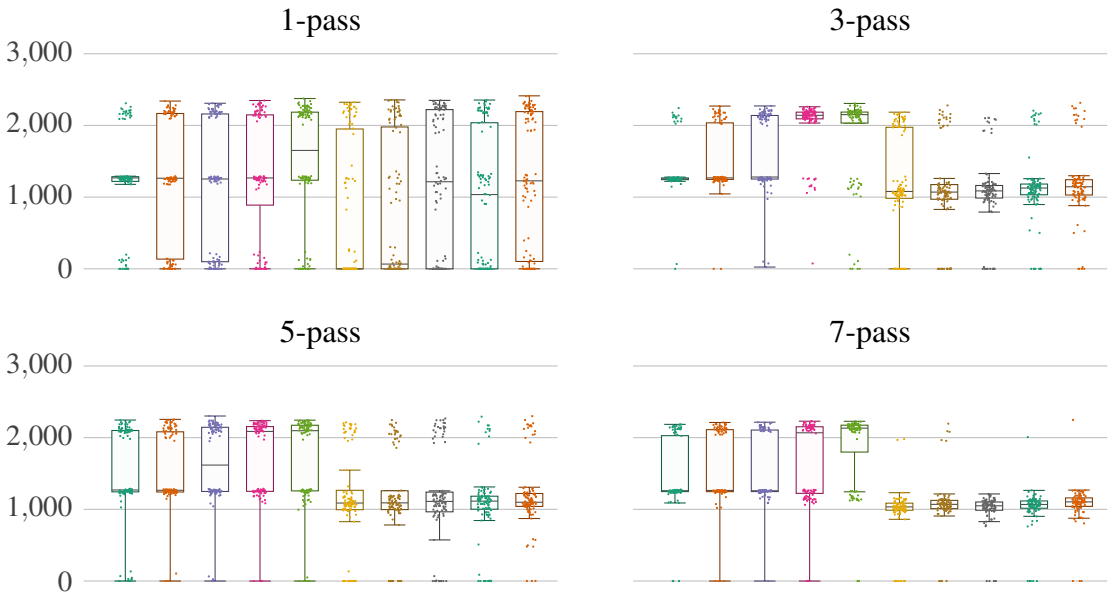
The results, as presented in table B.3, show a significant improvement, especially for the games that previously were particularly bad, like *Bolo Adventures*. Using seven, instead of five passes, will slightly improve the results further, but at the expense of a 40% longer computation time.

The final evaluation will use the last introduced method with five simulation passes. Using more passes has too few advantages to justify the way longer computing time.

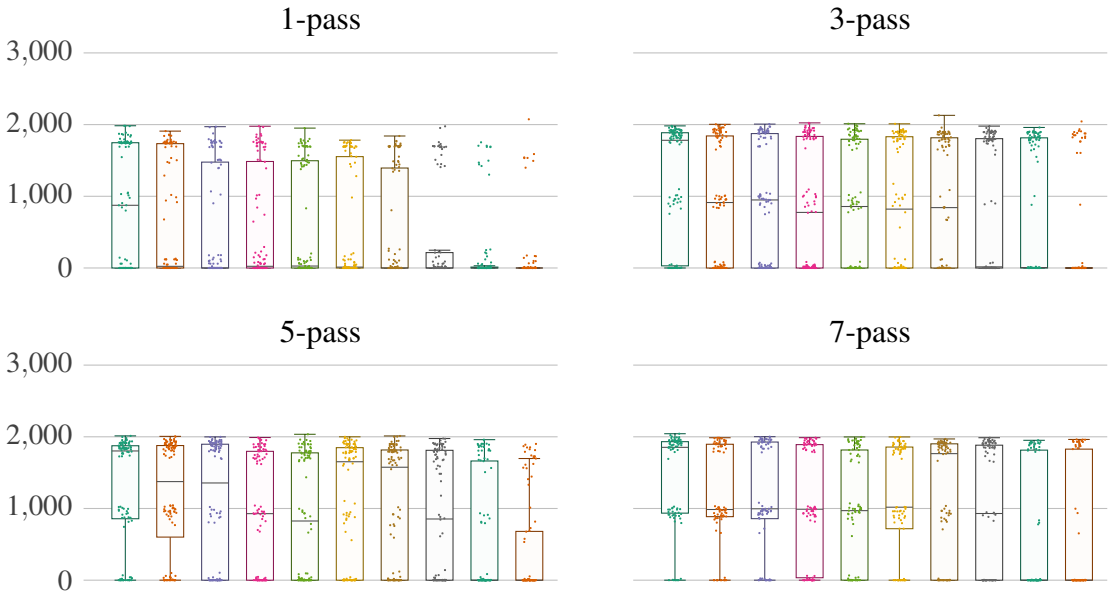
The following figures present the fitness values of ten individuals for four games by using a slightly different fitness function as explained in section 5.1. Every fitness function was evaluated with 1, 3, 5 and 7 simulation passes and every individual was played 100 times.

B.1.1. Averaged Fitness

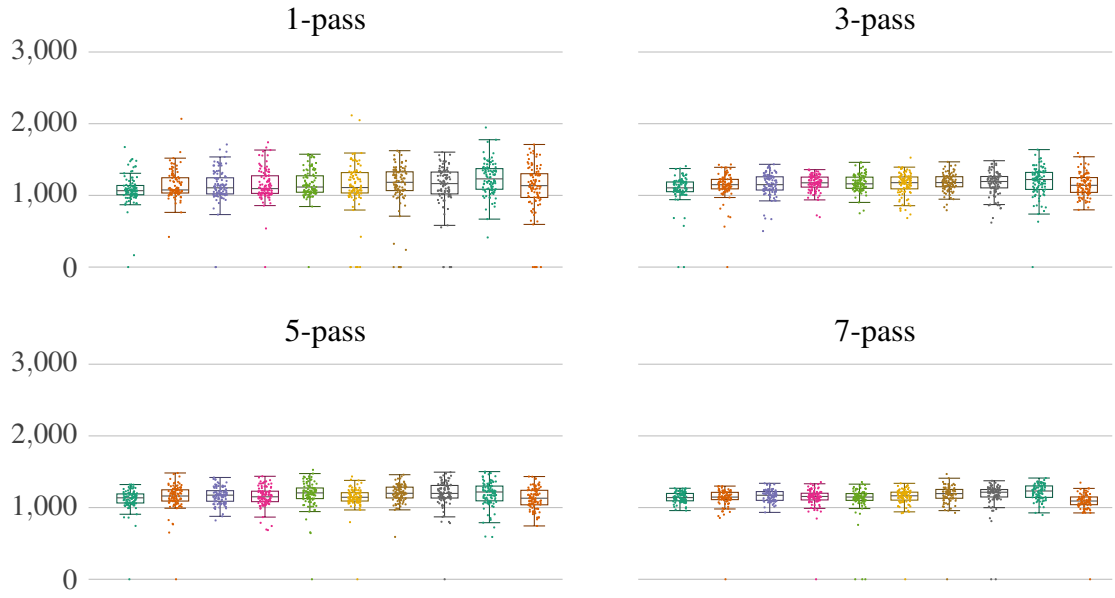
i. Aliens



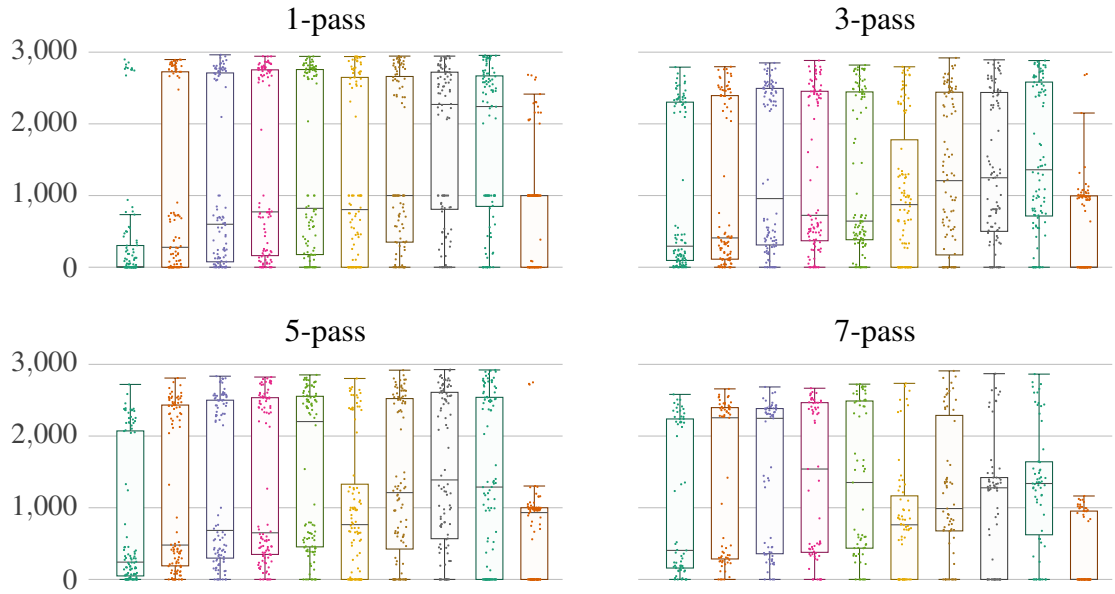
ii. Bolo Adventures



iii. Eggomania

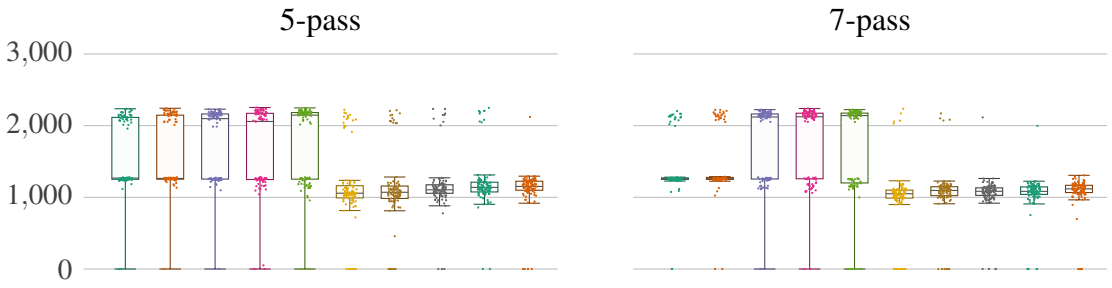


iv. Survive Zombies

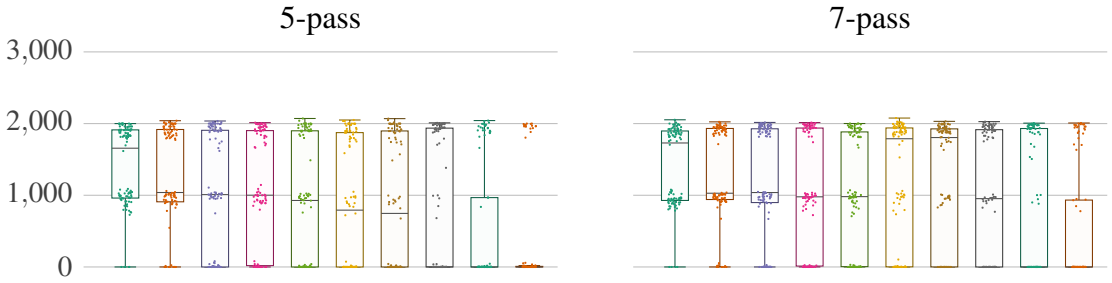


B.1.2. Centred Averaged Fitness

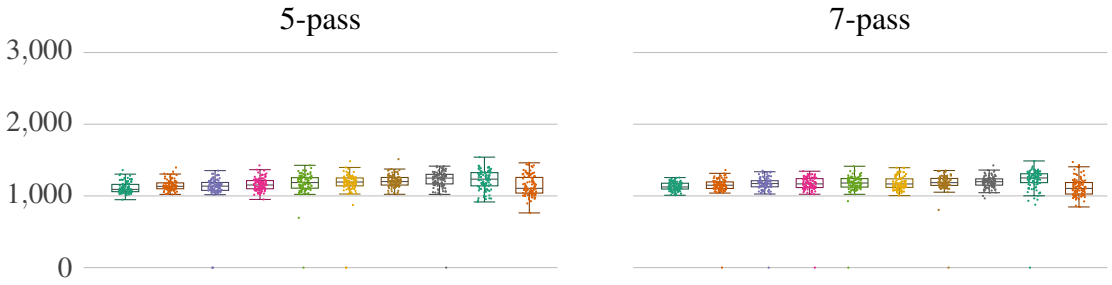
i. Aliens



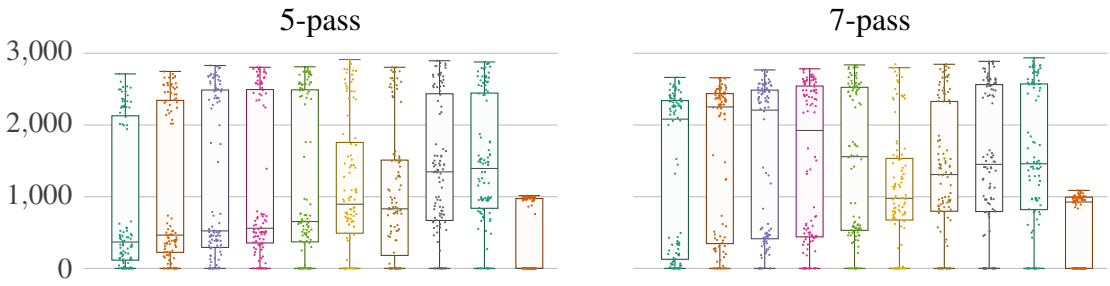
ii. Bolo Adventures



iii. Eggomania

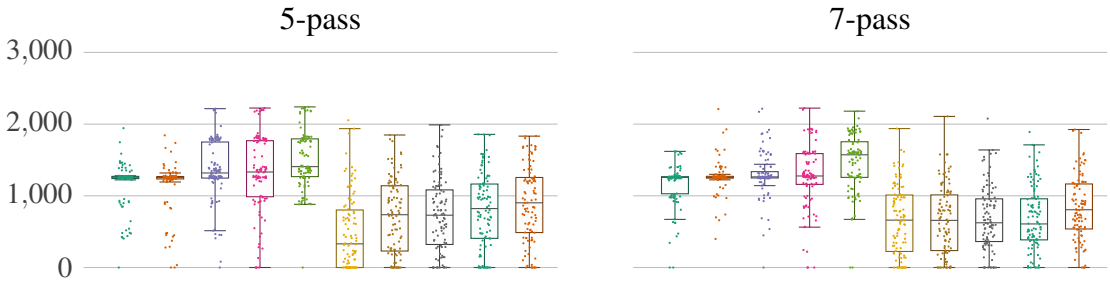


iv. Survive Zombies

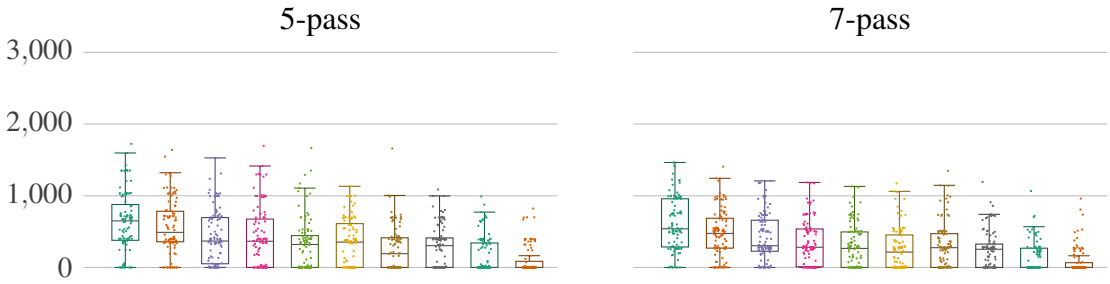


B.1.3. Centred Averaged Fitness with Average Feasible-Bonus

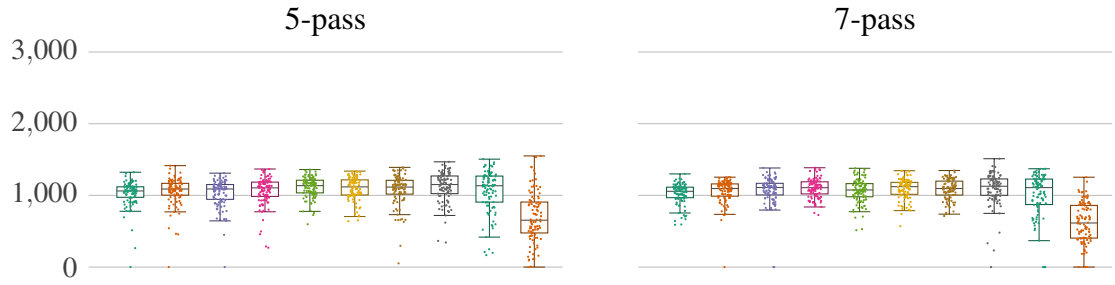
i. Aliens



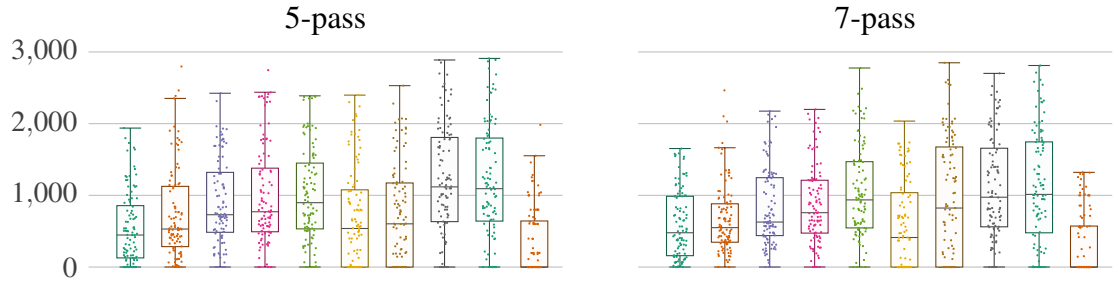
ii. Bolo Adventures



iii. Eggomania

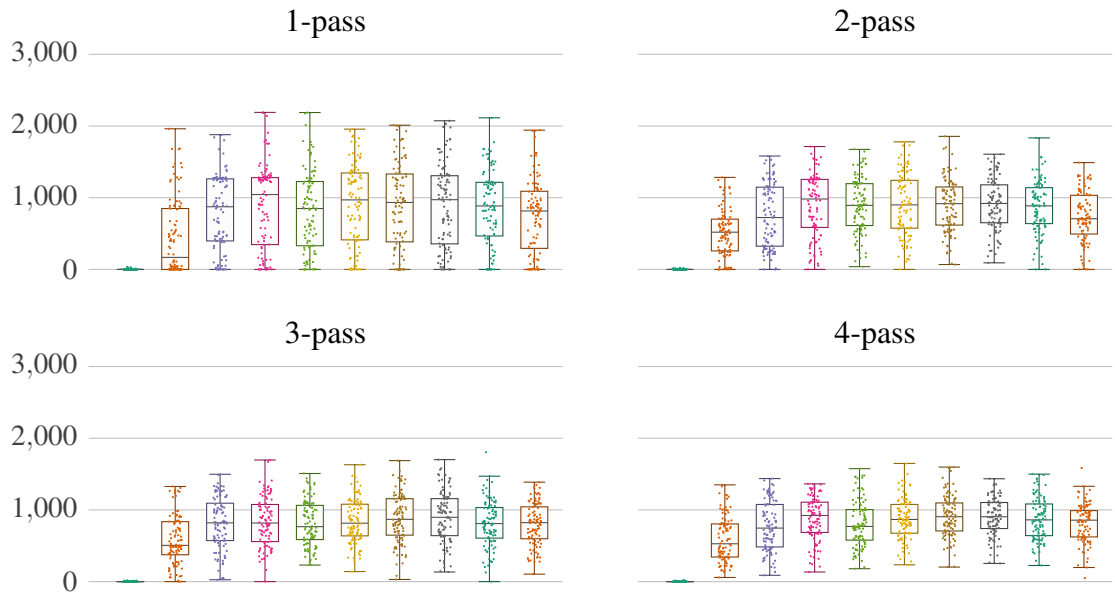


iv. Survive Zombies

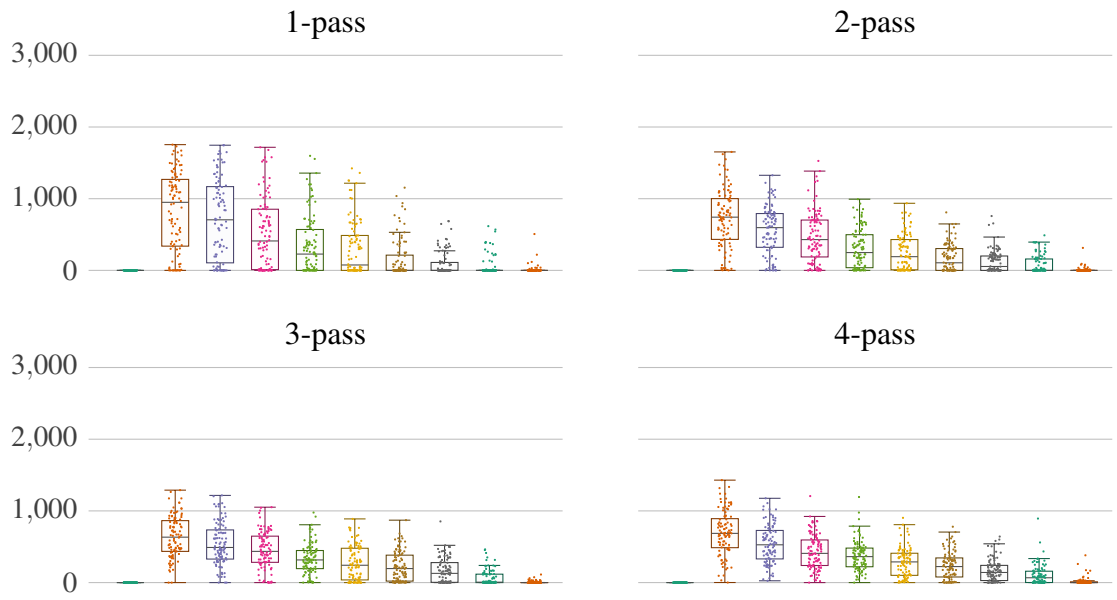


B.2. Experiment - Mapping Reliability

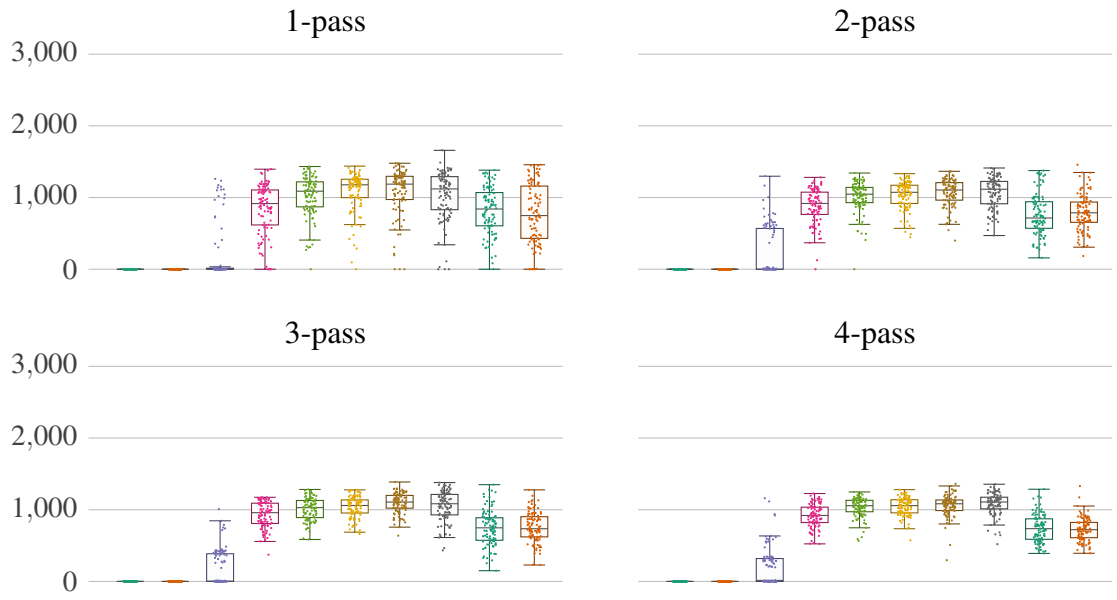
i. Aliens



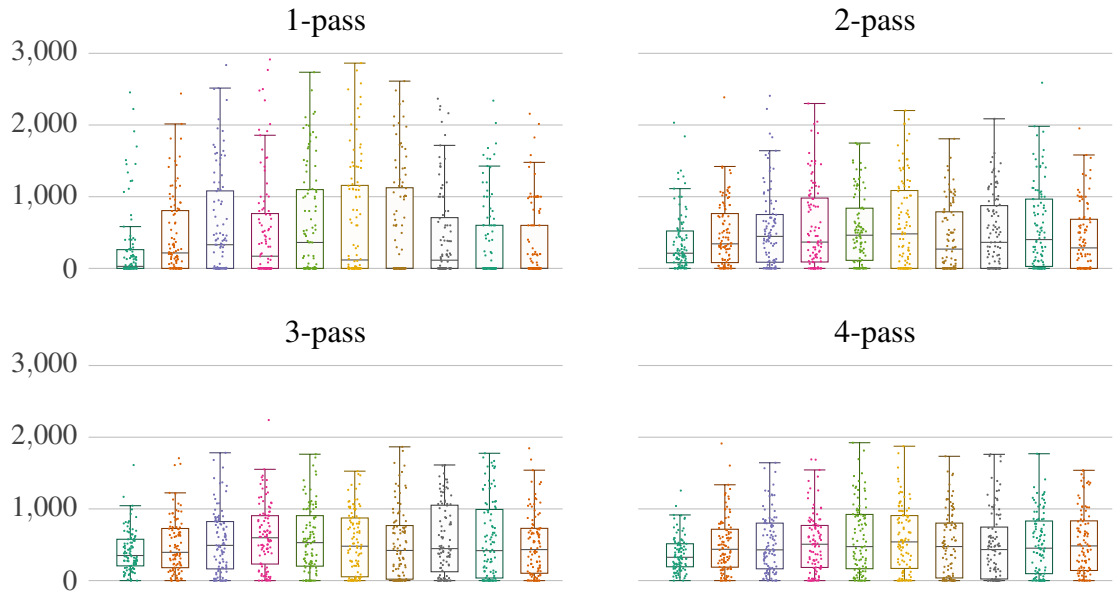
ii. Bolo Adventures



iii. Eggomania



iv. Survive Zombies



Bibliography

[AB05] Douglas G Altman and J Martin Bland. Standard deviations and standard errors, oct 2005.

[All94] L. Victor Searc Allis. *Searching for Solutions in Games and Arti cial Intelligence*. 1994.

[ALM11] Daniel Ashlock, Colin Lee, and Cameron McGuinness. Search-based procedural generation of maze-like levels. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):260–273, 2011.

[Bak87] James E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*, pages 14–21, Hillsdale, NJ, USA, 1987. L. Erlbaum Associates Inc.

[BPW+12] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012.

[Cha16] Kevin Chapelier. Convchain Example: Continuous. `http://www.kchapelier.com/convchain-demo/continuous.html`, 2016. (Accessed on: 2016-11-16).

[CHFh01] Murry Campbell, Joseph Hoane, and Hsu Feng-hsiung. Deep Blue. pages 1–31, 2001.

[Cho68] N. Chomsky. *Language and mind*. Harcourt, Brace & world. Harcourt, Brace & World, 1968.

[CMJ15] D.L. Craddock, A. Magrath, and M. Jaram. *Dungeon Hacks: How NetHack, Angband, and Other Roguelikes Changed the Course of Video Games*. Press Start Press, 2015.

[CSHD03] Michael F Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. *Wang tiles for image and texture generation*, volume 22. ACM, 2003.

[DB11] J. Dormans and S. Bakkes. Generating missions and spaces for adaptable play experiences. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):216–228, Sept 2011.

[Dor10] Joris 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*, page 1. ACM, 2010.

[DPL15] S. Samothrakis D. Perez, S. Mostaghim and S. M. Lucas. Multiobjective monte carlo tree search for real-time games. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(4):347–360, Dec 2015.

[DT14] Steve Dahlskog and Julian Togelius. *Procedural Content Generation Using Patterns as Objectives*, pages 325–336. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[Dys12] George Dyson. *Turing's Cathedral: The Origins of the Digital Universe (Vintage)*. Vintage Books, 2012.

[Eve02] Brian Everitt. *The Cambridge dictionary of statistics*. Cambridge University Press, Cambridge, UK; New York, 2002.

[Gar70] Martin Gardner. Mathematical Games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223:120–123, 1970.

[GB13] Michael Genesereth and Yngvi Björnsson. The International General Game Playing Competition. *AI Magazine*, 34(2):107–111, 2013.

[Gib09] Ellie Gibson. Games to cost $60m, says ubisoft boss. `http://www.eurogamer.net/articles/games-to-cost-USD60m-says-ubisoft-boss`, 2009.

[GLP05] Michael Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the AAAI competition. *AI Magazine*, 26(2):62–72, 2005.

[Hol92] John H Holland. Genetic algorithms. *Scientific american*, 267(1):66–72, 1992.

[JYT10] Lawrence Johnson, Georgios N. Yannakakis, and Julian Togelius. Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, pages 10:1–10:4, New York, NY, USA, 2010. ACM.

[KKS08] Andrew Kensler, Aaron Knoll, and Peter Shirley. Better gradient noise. Technical report, Tech. Rep. UUSCI-2008-001, SCI Institute, University of Utah, 2008.

[KSW06] Levente Kocsis, Csaba Szepesvári, and Jan Willemson. Improved Monte-Carlo Search. *White paper*, (1):22, 2006.

[LW05] Cathy Leach Waters. The united states launch of the sony playstation2. *Journal of Business Research*, 58(7):995–998, 2005.

[MM10] Peter Mawhorter and Michael Mateas. Procedural level generation using occupancy-regulated extension. In *2010 IEEE Conference on Computational Intelligence and Games*, Copenhagen, Denmark, 2010. IEEE, IEEE.

[MWH+06] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In *Acm Transactions On Graphics (Tog)*, volume 25, pages 614–623. ACM, 2006.

[Mxg16] Mxgmn. ConvChain. `https://github.com/mxgmn/ConvChain`, 2016. (Accessed on: 2016-11-16).

[NS16] Mark J. Nelson and Adam M. Smith. *ASP with Applications to Mazes and Levels*, pages 143–157. Springer International Publishing, Cham, 2016.

[Ols04] Jacob Olsen. Realtime procedural terrain generation. *Department of Mathematics And Computer Science (IMADA), University of Southern Denmark*, page 20, 2004.

[PDH+15] D. Perez, J. Dieskau, M. Hünermund, S. Mostaghim, and S.M. Lucas. Open loop search for general video game playing. In *GECCO 2015 - Proceedings of the 2015 Genetic and Evolutionary Computation Conference*, 2015.

[Per85] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, July 1985.

[Per01] Ken Perlin. Noise Hardware. *SIGGRAPH Course Notes*, 2001.

[Pit68] Jacques Pitrat. Realization of a general game-playing program. In *IFIP Congress*, pages 1570–1574, 1968.

[Pit71] Jacques Pitrat. Realization of a general game-playing program. *Information Processing*, 68:1570–1574, 1971.

[PlML16] Diego Perez-liebana, Sanaz Mostaghim, and Simon M Lucas. Multi-Objective Tree Search Approaches for General Video Game Playing. *Proceedings of the IEEE Congress on Evolutionary Computation*, 2016.

[RM78] Wayne A. Larsen Robert McGill, John W. Tukey. Variations of box plots. *The American Statistician*, 32(1):12–16, 1978.

[RP04] Timothy Roden and Ian Parberry. *From Artistry to Automation: A Structured Methodology for Procedural Content Creation*, pages 151–156. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[ŠBM+10] O. Šťava, B. Beneš, R. Měch, D. G. Aliaga, and P. Krištof. Inverse procedural modeling by automatic generation of L-systems. *Computer Graphics Forum*, 29(2):665–674, 2010.

[Sch13] Tom Schaul. A video game description language for model-based or interactive learning. *IEEE Conference on Computatonal Intelligence and Games, CIG*, 2013.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature Publishing Group*, (1):1–37, 2016.

[SM11] A. M. Smith and M. Mateas. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):187–200, Sept 2011.

[Smi15] Gillian Smith. An Analog History of Procedural Content Generation. *Foundations of Digital Games*, 2015.

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

[SP94] Mandavilli Srinivas and Lalit M Patnaik. Genetic algorithms: A survey. *Computer*, 27(6):17–26, 1994.

[Spe14] K Spencer. OpenSimplexNoise.java. `http://gist.github.com/KdotJPG/b1270127455a94ac5d19`, 2014. (Accessed on: 2016-11-16).

[STWM09] Gillian Smith, Mike Treanor, Jim Whitehead, and Michael Mateas. Rhythm-based level generation for 2d platformers. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, pages 175–182. ACM, 2009.

[TH12] Julian Togelius and Anders Hartzen. Compositional procedural content generation. *Proceedings of the FDG Workshop on Procedural Content Generation (PCG).*, 2012.

[TYSB11] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, 2011.

[VB12] Valtchan Valtchanov and Joseph Alexander Brown. Evolving dungeon crawler levels with relative placement. In *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, C3S2E '12, pages 27–35, New York, NY, USA, 2012. ACM.

[vdLLB14] R. van der Linden, R. Lopes, and R. Bidarra. Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1):78–89, March 2014.

[Vid16] Video Game Market Report. `http://www.woodsidecap.com/wp-content/uploads/2015/12/WCP-Video-Game-Report-20151104.pdf`, April 2016. (Accessed: 2016-11-16).

[Wan61] H. Wang. Proving theorems by pattern recognition ii. *The Bell System Technical Journal*, 40(1):1–41, Jan 1961.

[YT15] G. N. Yannakakis and J. Togelius. A panorama of artificial and computational intelligence in games. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(4):317–335, Dec 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, 08.12.2016 *Jens Dieskau*