Automated Level Generation for General Video Games[1]

E.C. Dallmeier

August 23, 2018

Abstract

This thesis presents alternative methods for tackling the problem of General Video Game Level Generation (GVG-LG). For this purpose three Monte Carlo (MC) search algorithms have been implemented in the General Video Game AI (GVGAI) framework. Also an improvement of the Constructive Generator algorithm available in the framework is proposed and tested. In a user study participants clearly preferred levels generated by the Improved Constructive Generator over all other methods. Additionally the participants were unable to distinguish the levels generated by the MC Search-based Generator from levels generated by the Genetic-Algorithm (GA) Search-based Generator. The work presented here provides valuable insights for further studies in the field of GVG-LG.

1 Introduction

To ensure proper understanding of the following sections it is necessary to provide an introduction to the domain. It is assumed that the definition of the phrases *Video Game*, *Level* and *(Software) Agent*, as well as the terminologies of *Search Algorithms* and *Search Trees* are commonly known.

1.1 Level Generation

Level Generation can be understood as a subset of the broader field *Procedural Content Generation* (PCG) [2, 9, 22]. While the ideas of PCG and *Level Generation* have already been around long for analog games, this thesis assumes the setting of digital or video games. In this context it can be found that this concept first emerged in the 1980s with example games like Rogue, published in 1980, and Elite, published in 1984. At that time the idea of PCG was mainly used to make the most use out of the limited disk capacity by calculating content of the game dynamically during runtime instead of shipping precalculated environments. The concept has come a long way from there as with the number of video games played raising also the need for new content for those games rises. For this reason it has developed from being a space-saving to being a cost-saving factor in the game industry. It occurs in many different contextual aspects of game development, ranging from the simple layout of levels and placement of objects in those up to narrative elements of the story of a game. Most of the commercial PCG-algorithms that emerged in the last decades are all tailored to be used with one specific game only. There are only a few exceptions to this specificity, for example *SpeedTree* [2, 9].

For the remainder of this thesis only the previously mentioned example of the layout of a level and location of objects within it is considered and is referred to as *Level Generation*.

1.2 General Video Game AI Framework

As this kind of specificity also reflects in the field of creating agents for video game playing the need emerged to shift this research into the field of *General* Video Game Playing (GVGP) [16]. The aim of GVGP is to reduce the domain knowledge often included in agents submitted to video game playing competitions that are centred around a certain game, for example StarCraft [18]. Reducing the impact of domain specific knowledge is done by providing a variety of different games instead of providing only one.

The first step towards creating a suitable environment for GVGP was to provide a language capable of describing a large set of video games. Here Ebner et al. [10] introduced the *Video Game Description Language* (VGDL) in 2013. Essentially it breaks a game down into four components, the *SpriteSet*, the *InteractionSet*, the *TerminationSet* and the *LevelMapping*. Using this language it is possible to represent a large subsection of video games, namely arcade-style games. These games are situated in a 2D environment and are subject to real-time changes [13], spanning many different genres (see Figure 1 for examples).

The next step was the creation of the *General Video*

[1] This thesis was prepared in partial fulfilment of the requirements for the Degree of Bachelor of Science in Data Science and Knowledge Engineering, Maastricht University, supervisor: Dr. Mark H.M. Winands.

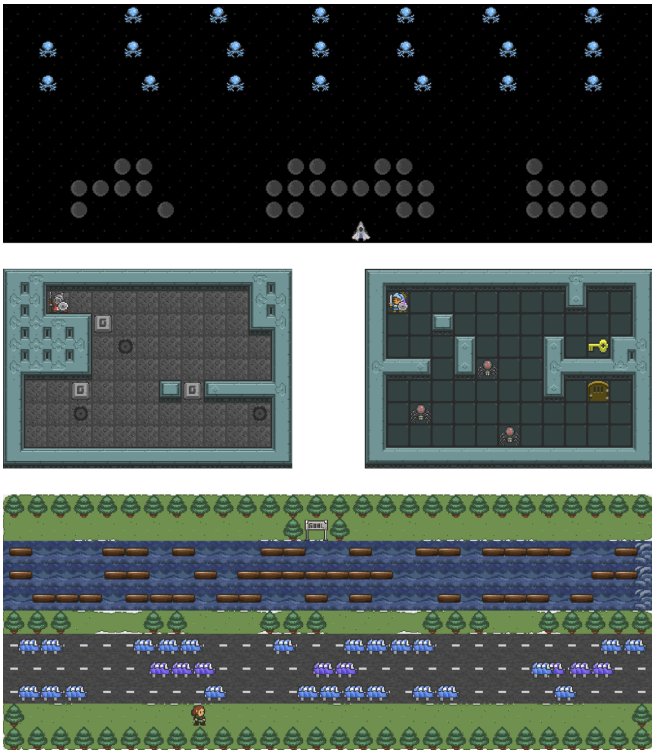


Figure 1: Some examples of games that can be represented in VGDL, top: Aliens (Space Invaders), centre-left: Sokoban, centre-right: Zelda (The Legend of Zelda), bottom: Frogs (Frogger).

Game AI (GVGAI) framework in 2014 [17] which consisted of the components necessary to hold competitions in the field of GVGP. In the following years the GVGAI framework and competition was further extended, including a total of 180 games written in VGDL in 2018 and several other competition tracks [18].

One of these tracks is the *General Video Game Level Generation* (GVG-LG) competition, which was held in 2016 [13, 18] and is designed analogous to the concept of GVGP, asking for submissions of level generators that are capable of generating a level for any single-player game described in VGDL. The work done on that part by Khalifa et al. [13] and the sample level generators presented by them form the cornerstone of the research in this thesis.

1.3 Problem Statement & Research Questions

Given the above environment and taking into account the work performed by Khalifa et al. [13] we can derive the following problem statement:

Considering different games that are available in VGDL-representation, how can we generate a human-enjoyable variety of levels using the same generator?

As becomes clear form the conclusion of their work [13], people that judged the performance of the sample generators were unable to distinguish between a level generated by the Random and the Constructive Generator. Also only one search-based generator, the Genetic-Algorithm (GA) Search-based Generator, was presented and applied to the above problem.

From their findings two research questions have been be formulated:

1. Can the Constructive Generator be improved to be more preferable than the Random Generator?
2. Can Monte Carlo search algorithms be used for level generation?

1.4 Outline

Section 2 gives an overview of the aforementioned work by Khalifa et al. [13] performed on the topic of level generation, including a description of the sample generators that are available in the GVGAI framework. Next, in Section 3, different search-based approaches that might be suitable for the problem at hand are presented, followed by the implementation of these algorithms within the GVGAI framework in Section 4. Experiments on these implementations are then performed in Section 5 and the results with regard to the research questions together with additional insight are presented in Section 6.

2 Previous Work

When the GVG-LG competition track was introduced in 2016 it was accompanied by the paper *General Video Game Level Generation* by Khalifa et al. [13]. Here the historical and scientific background of the competition track gets explained, as well as the GVG-LG programming framework, and three sample level generators.

When working in the GVG-LG framework, which provides a Java interface, the generators have access to certain information about the game they are given in form of the *GameDescription* and *GameAnalyzer* objects [13]. The *GameDescription* object provides raw information encapsulated in the VGDL description of the given game. It includes the occurring sprites from the *SpriteSet*, possible events created by the interaction of those sprites with themselves or the border of the screen from the *InteractionSet*, the game's termination conditions from the *TerminationSet* and lastly the *LevelMapping*, a mapping from character to sprite, for the graphical representation of the game [10]. As the name implies, the *GameAnalyzer* already processes the data resulting from the *SpriteSet*, *InteractionSet*, and *TerminationSet*. After this analysis, sprites are classified according to whether they are an avatar, solid, harmful, collectable or none of the previous. The *LevelMapping* also contributes to the competition by imposing on it the

constraints to only include one avatar and only characters that correspond to a valid sprite representation. Additionally due to relying on the GVGAI framework the competitors have access to a GUI for playing the games as well as several agents capable of doing so.

In the end, the three sample generators described below and the submissions for the 2016 competition were judged by human players being able to state their preference, if any, over two levels chosen randomly from two different generators [13, 18]. In the initial pilot study this resulted in the sample Random and Constructive Generator receiving a similar number of votes and the sample Search-based Generator creating the most appealing levels. These generators are discussed in Subsections 2.1, 2.2, and 2.3, respectively.

2.1 Random Generator

The Random Generator [13, 18] is the most simple approach. It has only one parameter, the probability for each tile of the level being considered for placing a randomly chosen sprite on it. After that, it automatically chooses the size of the level to be generated depending on the number of sprites available for the given game and ensures that each available sprite is at least placed once. Only the avatar, which is to be placed exactly once only, is an exception to this. No additional information about the game at hand is taken into account when creating the level.

2.2 Constructive Generator

A more sophisticated approach is presented with the Constructive Generator [13, 18]. It already utilises the information provided by the *GameAnalyzer* and prioritises sprites based on their occurrences in the *InteractionSet*: the more often a sprite is mentioned there, the more often it is placed in the level. Apart from a pre- and post-processing phase the generator passes through four main phases. First, a general layout of the level is created, if the *SpriteSet* of the game contains a solid sprite. In the second step the avatar is placed. The third step consists of placing harmful sprites either randomly or at a distance from the avatar depending on whether the harmful sprite is moving. Lastly, a random number of collectable and other sprites are placed. This is all done according to the desired cover-percentages calculated in the pre-processing step and concluded with checking that the number of sprites listed as goals is at least as high as required in the *TerminationSet*.

2.3 Search-based Generator

The last presented sample generator falls into the category of search-based generators. According to Perez-Liebana et al. [18] a search-based generator makes use of simulating a playthrough of the generated level to ensure playability. Here Khalifa et al. [13] use a Feasible-Infeasible Two-Population (FI-2Pop) [14] Genetic-Algorithm (GA) approach. For this purpose they created several features that are maximised using the GA. Those features can be divided into either *Direct Fitness Functions* or *Simulation-based Fitness Functions* according to the definition by Togelius et al. [22]. The difference between the two being that the first class of fitness functions can directly be applied to a generated level and the second class making use of one or several agents playing the generated level.

In case of one half of the populations of the GA generator the *Direct Fitness Functions* include features regarding the number of avatars, the number of sprites, the number of goals, as well as the current cover-percentage of the level. Also in this half *Simulation-based Fitness Functions* are applied, consisting of measuring the length of the solution generated by the used agent, checking if the agent did win the game and lastly whether an agent did not die within a predefined time span. The second half of the populations of the GA solely relies on *Simulation-based Fitness Functions*, calculating the score-difference after playthrough between a simple agent and a better agent, as well as inspecting how often the rules defined by the *InteractionSet* are utilised by different agents. For the previously mentioned better agent Khalifa et al. [12] introduced an agent that is a deviation of one of the highest ranking agents for playthrough in the GVGAI framework. It was modified such that it sometimes tends to not perform an action or to repeat the same action several times in order to closer mimic human behaviour. When calculating the score-difference as well as when checking for the agent not dying within a certain number of steps they use an agent that simply performs no actions at all.

Due to the default time for a playthrough of a generated level by means of the agents mentioned before being set to three seconds in the framework, the number of levels that can be generated and tested within a certain amount of time is limited. Because of having a total duration of five hours available for generating a level during the GVG-LG competition [13, 18] the algorithm is still able to generate and test a sufficient number of levels. Five hours is also the allowed duration that is used for the experiments in Section 5.

3 Different Search-based Approaches

The following Subsections 3.1, 3.2, and 3.3 provide an introduction to the functionality of the search-algorithms as they are described in the literature. All three algorithms have in common that they apply Monte Carlo (MC) [3] methods, which initially were used for approx-

imating statistical solutions. The idea behind using MC methods when applied to search problems is to approximate the actual value of a state by pseudo-randomly simulating a sequence of actions [7] and using the resulting value as the estimate. Additionally to pointing out their functionality, also similarities and differences between the different approaches are shown below. Further domain-specific details on the algorithms follow in Section 4.

3.1 Monte Carlo Tree Search

The first algorithm is Monte Carlo Tree Search (MCTS) [8, 15] and consists of four phases [7], *selection*, *expansion*, *rollout*, and *backpropagation* (see Figure 2). Over the time it builds a search tree where

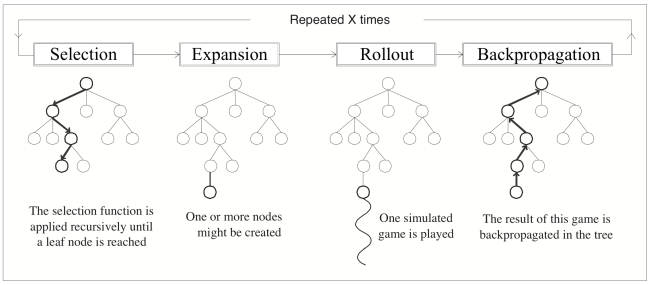


Figure 2: Basic structure of MCTS [7].

each node contains its value and the number of times it has been visited so far. In the first phase the partial search-tree that has been created so far is recursively traversed from the root node to a previously unexpanded node by applying certain means of *selection*. Once such a node is reached it is chosen for *expansion*, consequently being added to the tree. Starting from the chosen child/children, a MC *rollout* is performed until a terminal state is reached. This terminal state can be easily evaluated and the resulting value is *backpropagated* through the nodes that were visited during the first phase, each time updating the two fields inside the node. After those four phases are concluded they are repeated from the beginning as long as the current time- or resource-constraints allow it. Due to using the value of a terminal state as an estimate no intermediate evaluation function is necessary, making MCTS suitable for applications where such an intermediate evaluation function is not available.

Each of the phases can be modified, leading to a large variety of algorithms tailored for certain purposes. One of the most popular modification for the *selection* step, which also is used in this thesis, is Upper Confidence bound applied to Trees (UCT) [15]. The UCT selection strategy states that a move i will be chosen such that it maximises Formula 1:

$$v_i + C \times \sqrt{\frac{\ln N}{n_i}} \tag{1}$$

where v_i represents the average score achieved by node i so far, n_i represents the number of visits of node i, N represents the number of visits for the parent node of node i and C is an adjustable constant. The constant C, which is denoted by $\sqrt{2}$ in the literature, is used to determine how much influence the $\sqrt{\frac{\ln N}{n_i}}$ part of Formula 1 has. A high value of C makes moves that have not been visited often more favourable and a low C value results in choosing moves that previously scored high. Tuning C is also considered balancing between *exploration* of unknown moves and *exploitation* of previously successful moves, respectively. Regarding the expansion phase the strategy proposed by Coulom [8] is used here, stating that each run one node is added to the tree. The rollouts are performed in a random manner and also the backpropagation is handled in the default way as described above by Chaslot et al. [7].

3.2 Nested Monte Carlo Search

The second algorithm for the problem at hand is the Nested Monte Carlo Search (NMCS) [5], which introduces the notion of *nesting levels*. It relies on the idea that an action or a sequence of actions that performs well on a lower-level search also performs well on the higher level. In NMCS the *base level* behaves similar to a rollout in MCTS [7], randomly choosing actions from the given state until reaching a terminal state, which again can be easily evaluated. On *nesting levels* higher than the *base level*, the algorithm recursively performs a lower-level search for all available legal actions, each time returning the value of the terminal state and the sequence that led to this result. This result for example is a random sequence of legal actions, together with its score, in case of the *base level*. By combining the resulting sequences from all lower level searches, a complete sequence of actions from the root to a terminal state is generated. Next, the algorithm considers the first action from the returned sequence as the root node, thus resulting in a possibly different set of legal actions. From there on it tries to improve on the result of the initially found sequence by again performing a lower-level search for all of those actions. In case such an improvement is found, the remaining initial sequence of actions is replaced with the new sequence. This procedure is repeated until the point where the current root node is a terminal state, in other words when every other action of the initial or updated sequence was considered once as root node with the intention of finding a better follow-up sequence using a nested MC rollout (see Figure 3).

In comparison to the previously described MCTS [7], it does not rely on building a partial search tree, but

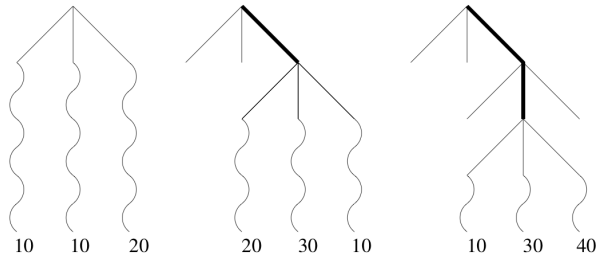


Figure 3: Basic structure of NMCS [6]. A straight line represents all allowed actions at the current node and a wavy line indicates a lower level search. According to the found value, the current node is changed for the next iteration, displayed by a bold line. A total of three iterations are shown above.

instead only memorises the best found sequence so far together with the evaluated result [5], making it more memory-efficient. This sequence is also kept after one complete run of NMCS is concluded and is used as input for the next run, guaranteeing that the next result can only either stay the same or improve, which is not given in case of the default MCTS with UCT [15]. The only parameter that can be used for adjusting the behaviour of the algorithm is l, the number of *nesting levels* that are to be used. With increasing values of l the number of rollouts, as well as the computation time for one run increases, so in the literature usually values for l from 0 to 3 are considered. Apart from the l parameter, there is no direct way of tuning the algorithm towards either more *exploration* or *exploitation*, yet NMCS outperformed variations of MCTS in areas where the later previously had achieved new records [3].

3.3 Nested Rollout Policy Adaption

The third presented algorithm is the Nested Rollout Policy Adaption (NRPA) [19] which also belongs to the family of Monte Carlo algorithms [3]. Contrary to the first two approaches, this algorithm does not solely rely on randomness during rollout, but it utilises a policy π when determining the sequence of actions. For any state S and any legal action a_S for that state, a code $code(S, a_S)$ is computed. It allows to include domain-specific knowledge during NRPA in the form of having $code(S, a_S)$ return the same value for different, but under the consideration of domain-specific knowledge sufficiently similar, state-action pairs. This calculated code is then used as input for the policy, such that during rollout an action is considered proportional to its value $\pi(code(S, a_S))$. Apart from this policy the structure of the algorithm is similar to the structure of NMCS [5], again making use of *nesting levels* l.

On the *base level* NRPA [19] performs a simple rollout, choosing the sequence of actions according to the current policy as described above. During each higher-level search M lower-level searches are initiated recursively, where M represents the number of iterations, an adjustable parameter of NRPA. Additionally, when initiating those lower-level searches, they are given the current policy of the higher-level search that initiated them to be used for either rollout or initiating lower-level searches themselves. Once all M iterations are concluded, the highest scoring sequence of actions and its value is returned back to the higher level. Next the policy is adapted according to a learning rate α, the second adjustable parameter in NRPA, ensuring that future searches are more likely to be performed in favour of the best found sequence so far. Again, similar to NMCS [5], the overall best sequence, value and also the policy are stored after completing a run of the algorithm to be used for subsequent calls, such that the next value can never drop below the best finding so far.

When returning to the notion of *exploration* and *exploitation* mentioned during the description of MCTS [7], analogous statements can be made about NRPA [19] using the available parameters M, α and l. Even though not stated explicitly, choosing a high number of iterations M results in the same policy being used for a higher number of subsequent rollouts, similar to the notion of *exploration*. This can also be achieved by considering a low value for α, causing the policy to converge slower than it would do with a high value for α. The later, as well as choosing a relatively small M, can in turn be considered *exploitation* as the given policy converges faster and does not get many iterations to find a better result. The literature uses sample values of 100 and 1 for M and α, respectively. Adjusting l again influences the total number of rollouts that are performed during one run of the algorithm. NRPA scored even higher than NMCS [5] on similar problems that were previously used for comparing NMCS with MCTS.

4 Implementation of the Level Generator

The following subsections provide insight into the implementations that were made in the course of this thesis. These implementations are split into two parts, each of which arose from one of the two research questions from Subsection 1.3.

4.1 Improved Constructive Generator

To address the first of the two previously stated research questions it is helpful to revise the differences between the Random and the Constructive Generator mentioned in Section 2. Apart from the obvious difference, specifically the Constructive Generator following a more struc-

tured approach than the Random Generator, one other crucial aspect is noteworthy as it was given as a possible explanation by Khalifa et. al. [13] for the issue at hand. This issue is the unexpected indistinguishability between the Constructive and the Random Generator that arose during the initial pilot study.

While the Random Generator ensures that all sprites available through the *GameDescription* object are placed in the level, the Constructive Generator does not do so. Instead, only a random selection from the sprites that are considered non-essential by the Constructive Generator is used to fill the remaining tiles of the level. This behaviour can lead to unplayable levels if the importance of a sprite is hidden in the game's VGDL representation. It happens for example with the game *Zelda*, where there is a dungeon exit, clearly marked as goal, and additionally a key, only considered a collectable sprite, but without which the exit is unusable. Even though the Constructive Generator contains a phase where it checks that the number of goal sprites is acceptable, it does not consider collectable sprites, as their desired number of occurrences can be any value, including zero. When participants of the initial pilot study [13] encountered levels where the above situation showed up, they obviously preferred a playable level over an unplayable one, regardless of its random nature. The reasoning behind this strategy for the Constructive Generator is that if there are several sprites that can be grouped into one class, the generator would not necessarily place one or more of each sprite, but only of each class. Considering again the example of *Zelda* this can be visualised by having three different types of enemy sprites, where a level containing only one type of enemy sprite would appear less cluttered than each level always containing all three types.

The improvement made to the Constructive Generator in this thesis is to include an additional phase where it is ensured that also each collectable sprite is at least placed once. Whether this proposed improvement is sufficient to answer the first research question is discussed in Section 5. The motivation behind this decision is that having a playable level outweighs having a less cluttered level.

4.2 Alternative Search-based Generators

Regarding the second research question the resulting implementations and their discussion require more insight. To justify the usage of the three algorithms described in Section 3, one should consider that all three have been successfully applied to single-player games [3] and that level generation can be considered as a single-player game.

Evaluation Function

For the three algorithms an improved version of the already existing *Direct Fitness Functions* [22] is used, containing more features. Contrary to the GA Search-based algorithm [13] no *Simulation-based Fitness Functions* [22] are employed. The reasoning for this decision is to fully utilise the strength of the MC search-based algorithms, lying in performing as many rollouts as possible. If one would use *Simulation-based Fitness Functions* with the default parameters given in the GVGAI framework, only one rollout every three seconds would be possible.

When ignoring any features that make use of *Simulation-based Fitness Functions* we are initially left with the following direct evaluation function as shown in Equation 2 [13].

$$f_{directScore} = \frac{\begin{array}{c} f_{avatarNumber} + f_{spriteNumber} \\ + f_{goalNumber} + f_{coverPercentage} \end{array}}{4} \quad (2)$$

In the above equation $f_{avatarNumber}$ is 0 if the level contains no or multiple avatars and 1 if it contains exactly one. $f_{spriteNumber}$ is 1 when each sprite from the *SpriteSet*, that is not spawned by another sprite, is placed at least once, and 0 if the level contains none of these sprites. Similar $f_{goalNumber}$ is 1 as soon as the minimum number of occurrences for all goal sprites is reached, and $f_{coverPercentage}$ is 1 as long as the number of covered tiles lies within a predefined range. The fact that all parts of the numerator of Equation 2 are in the range $[0, 1]$, with $f_{avatarNumber}$ being a binary variable, leads to $f_{directScore}$ also being within that range. Additionally in the initial setup all *Direct Fitness Functions* are equally weighted. These four features by themselves seem too imprecise for generating playable levels. This is why they were accompanied by *Simulation-based Fitness Functions* in the original implementation. As those simulations do not take place when applying MCTS [8, 15], NMCS [5] and NRPA [19], the evaluation function has been improved in this thesis to consist of the following features:

- **Accessibility:** A binary variable indicating whether all non-solid tiles of the level are reachable from the current position of the avatar.
- **Avatar number:** Similar binary variable as $f_{avatarNumber}$ mentioned in Equation 2.
- **Connected walls:** The ratio between solid sprites that are directly adjacent to another solid sprite and the total number of solid sprites.
- **Cover-percentage:** Similar real-valued variable as $f_{coverPercentage}$ from Equation 2 indicating how

close the current tile-coverage of the level is to the desired cover-percentage.

- **Ends initially:** The ratio between the number of unsatisfied termination conditions listed in the *TerminationSet* and the total number of termination conditions, inspired by $f_{goalNumber}$.
- **Goal distance:** The distance of the closest currently placed goal sprite to the position of the avatar over the maximum possible distance.
- **Neutral-harmful ratio:** Indicating whether the ratio between neutral and harmful sprites fulfils a predefined ratio.
- **Simplest avatar:** Binary variable ensuring that the currently placed avatar is in the most simplest form and not an *powered-up* avatar that usually requires interaction with another sprite before it becomes available.
- **Sprite number:** Identical to $f_{spriteNumber}$ from Equation 2.
- **Space around avatar:** The number of populated tiles that are reachable by two steps from the avatar's position over the sum of all the tiles within the two step range.
- **Symmetry:** Checking for symmetry along the X- and Y-axis for each tile in each quadrant of the level.

Alternatively to the initial evaluation function the improved version consists of the weighted, $[0, 1]$-normalised, score, resulting in the following equation:

$$f_{directScore} = \frac{\sum_i (w_i f_i)}{\sum_i w_i}, \tag{3}$$

where w_i is the integer weight for the ith feature f_i. These 11 features are chosen in a way to create mostly playable and appealing levels while only using *Direct Fitness Functions* and still trying not overfit on any of the many genres represented in VGDL.

General Information

Concerning the actual implementation of the three algorithms in the GVGAI framework additional remarks are necessary. Also the terminology of the described algorithms needs to be transferred into the domain of level generation. This means that a *node* or *state* can be thought of as the current layout of the level, while all *actions* possible at that *state* or descending from that *node* can be viewed as placing all possible sprites on all possible remaining tiles.

Monte Carlo Tree Search

MCTS with UCT [8, 15] is implemented as in the literature. Due to the existence of more advanced improvements for the different phases of MCTS that would be applicable to the problem at hand [3, 4, 20] one should bear in mind that there is still potential for an increase in performance. Yet those improvements are not implemented in the course of this thesis, such that the only adjustable parameter for MCTS is the value C as shown in Formula 1.

Nested Monte Carlo Search

NMCS [5] is also implemented as in the literature. This includes memorisation of the currently best found sequence and using it as input for subsequent calls, which ensures that the next value found can either only improve or stay the same. Here again only one adjustable parameter, namely the nesting level l, can be specified.

Nested Rollout Policy Adaption

Likewise NRPA [19] is implemented according to the literature, but as its $code(S, a_S)$ function incorporates domain-specific knowledge it needs to be described explicitly. To recall, the purpose of $code(S, a_S)$ is to ensure that, under consideration of domain-specific knowledge, several state-action pairs are treated identically when deciding which policy to use, even though the pairs are actually not the same. To make use of that in the domain of level generation the state S is chosen to not represent the complete current level, but only the last n executed actions. This works analogous to the concept of N-Grams [21], which already has been successfully applied to the field of General Game Playing. When considering the expression of the policy function $\pi(code(S, a_S))$ as a simple English sentence it would translate to: *What is the policy value of action a_S, given that the last n actions were S?* As the order of the n previously executed actions S does not influence the layout of the level, the order of the actions in S is neglected when determining the policy value of a_S. This results in a total of four adjustable parameters for NRPA: the previously introduced n, the number of iterations M, the learning rate α and the nesting level l.

5 Experiments

The first series of experiments deal with defining all aforementioned parameters for the algorithms and weights for the evaluation function. The goal of the second series of experiments is to determine, which of the three algorithms used for level generation in this thesis on average achieves the highest score. The third series of experiments is a user study, where participants are asked to state their preference over two levels. These levels come from four different generators, namely the sample Random Generator [13], the improved constructive generator, the GA Search-Based Generator and the best performing algorithm from the Monte Carlo family. Due to the aim of this thesis being to present comparable

alternative algorithms for level generation the most focus is put on the second and third round of experiments.

5.1 Parameter Tuning

In this subsection first the weights and parameters for the new evaluation function described in Subsection 4.2 are determined, followed by the individual parameters for the three algorithms at hand.

Feature Weights

The weights of the proposed evaluation function were chosen intuitively rather than by means of experiments. This was mainly due to only having a small number of already created levels available, of which some even lacked proper quality that is they were not playable or did not represent the concept of the underlying game. If those levels would have been available, one could, just to mention one possibility, have applied gradient search algorithms on the weights of the evaluation function in order to learn which weights would on average result in the highest score over those example levels. Instead all weights were initialised at 1 and then increased by 1 if the respective feature subjectively seemed more important than the other features. This resulted in the following values for the weights:

- $w_{Accessibility} = 3$
- $w_{AvatarNumber} = 4$
- $w_{ConnectedWalls} = 1$
- $w_{CoverPercentage} = 2$
- $w_{EndsInitially} = 4$
- $w_{GoalDistance} = 1$
- $w_{NeutralHarmfulRatio} = 1$
- $w_{SimplestAvatar} = 4$
- $w_{SpriteNumber} = 2$
- $w_{SpaceAroundAvatar} = 3$
- $w_{Symmetry} = 1$

Those values were chosen such that features that contribute to an increased playability of the level have a larger weight than those that have a mere aesthetic aspect.

Algorithm-specific Parameters

When generating a level according to the initial research as well as during the 2016 GVG-LG competition [13] a time limit of five hours was given. This made tuning the parameters for the individual algorithms very time-consuming. For this reason the experiments were conducted on shorter time periods and from those results assumptions about the parameters influence on longer periods were made. Finally this assumed influence and behaviour was then either confirmed or rejected by a small number of five-hour runs.

Parameters for MCTS

As pointed out in Subsection 4.2 there only is one tuneable parameter for MCTS with UCT [8, 15], namely C, which is used to balance between exploration and exploitation. For these experiments the size of the level to be generated was fixed at width and height of 8 each. MCTS was then used for level generation with a duration of 10 minutes, repeated 30 times, with 0.05, 0.5, 0.9, 1.4 and 3 as possible values for C. Out of those values $C = 0.05$, a value resulting in high exploitation, on average reached the highest score, yet when used for the actual duration of five hours the score did not notably improve. For this reason a value of $C = 1.4$ (as an approximation to $C = \sqrt{2}$), the standard value as proposed in the literature [1, 15], was tried for the five-hour scenario. Because it already achieved higher scores than the previously chosen value, it was decided to continue using $C = 1.4$. The reason for the poor performance of the exploitation-focused value on the long duration can easily be explained with the same. When there only is little time available for finding a solution, of course it is more feasible to exploit a quick acceptable solution. If more time is available it becomes more feasible to spend time on exploring the search space for alternative solutions. This is why a small C was appropriate for the short duration, but not as applicable to the long duration.

Parameters for NMCS

The only adjustable parameter for NMCS [5] is the nesting level l. At a value of $l = 0$ only one rollout is performed to generate a level and its outcome is purely random. With increasing values of l also the number of rollouts increase. Already at $l = 2$ the algorithm was not able to generate a level with a score only remotely as high as with $l = 1$ within a duration of 10 minutes. For this reason $l = 1$ was chosen, as it provided acceptable results and did so in a timely fashion.

Parameters for NRPA

With NRPA [19] as implemented in this thesis there are a total of four tuneable parameters. For the code function of NRPA an N-Gram parameter of $n = 1$ is used, as further increasing n did not improve on the average result. Here $n = 1$ corresponds to considering only the current action and none of the preceding actions. When considering the learning rate parameter α, the literature [19] used a value of $\alpha = 1$, as it provided a stable balance between converging too slow or too fast. This value was also used in the course of this thesis. The remaining parameters, M and l, the number of iterations and the nesting level, determine the number of rollouts that are performed to generate a level. Using small values for M in combination with small values for l lead to a decreasing quality in the generated levels. On the

other hand using high l and M values could result in the algorithm taking too long to return a proper level. Since $M = 10000$ combined with $l = 1$ and $M = 100$ combined with $l = 2$ resulted in similarly good scores that were returned in acceptable time, it was decided to use the later combination.

5.2 Highest Scoring Algorithm

To find out which of the three algorithms is most applicable for level generation they are each used with the parameters determined above. As a time limit for level generation the previously mentioned duration of five hours is applied. From all available games in the GVGAI framework four were chosen, namely *Aliens*, *Frogs*, *Sokoban* and *Zelda*. Two out of these four (*Frogs* and *Zelda*) were also used in the pilot study conducted during the initial research [13]. For each of the four games 30 levels were generated per algorithm, resulting in a total of 120 levels each. These levels had their size fixed at 10×10 tiles. The performance of the three algorithms was compared by three different scores. First the number of rollouts, second the time it took to reach the highest value and third the highest returned value itself. Out of those measurements the last one decides about the highest scoring algorithm, but the other two still present valuable insights. The 95% confidence intervals of those three measures are presented in the plots in Figure 4.

Number of Rollouts

In Subfigure 4a the 95% confidence intervals for the number of rollouts for each of the three algorithms is shown. The means for MCTS [8, 15], NMCS [5] and NRPA [19] are $\bar{x}_{rolloutsMCTS} \approx 105323 \times 10^3$, $\bar{x}_{rolloutsNMCS} \approx 284481 \times 10^3$ and $\bar{x}_{rolloutsNRPA} \approx 6383 \times 10^3$, respectively. Those values are rounded to the nearest thousands. NMCS on average has the highest number of rollouts. This can be explained by the fact that NMCS does neither create a search tree nor has to keep track of a policy. Because of this, more time can be spent on actually performing rollouts, resulting in these values. As already indicated above, MCTS with UCT does build a partial search tree and also has to update all visited nodes of a solution during the backpropagation step. Due to this more sophisticated selection-strategy, MCTS is only able to perform the second highest average number of rollouts. Of the three algorithms NRPA has the lowest average number of rollouts. The reason for this is keeping track of the underlying policy. It includes updating the values of the policy accordingly as well as creating copies of it for the nesting levels. There is a visible trade-off between using more information during the search and the number of performed rollouts.

Time Until Reaching Highest Score

Subfigure 4b shows the 95% confidence intervals for the duration it took the three algorithms to reach the highest returned score. Again the means for MCTS [8, 15], NMCS [5] and NRPA [19] are $\bar{x}_{durationMCTS} \approx 136$, $\bar{x}_{durationNMCS} \approx 117$ and $\bar{x}_{durationNRPA} \approx 134$ minutes, respectively. These values are rounded to the nearest integer value. On average there only is a small difference in duration when comparing the three algorithms. The fact that all three means and also all three confidence intervals lie close to half of the available time of five hours is more interesting. It implies that from this point onward on average the three algorithms did not find a better solution. This leads to the assumption that the parameters determined in Subsection 5.1 are not perfectly fit for the five-hour timeframe. Judging from these averages one could safely decrease the time-limit to three hours and would still receive results of similar quality.

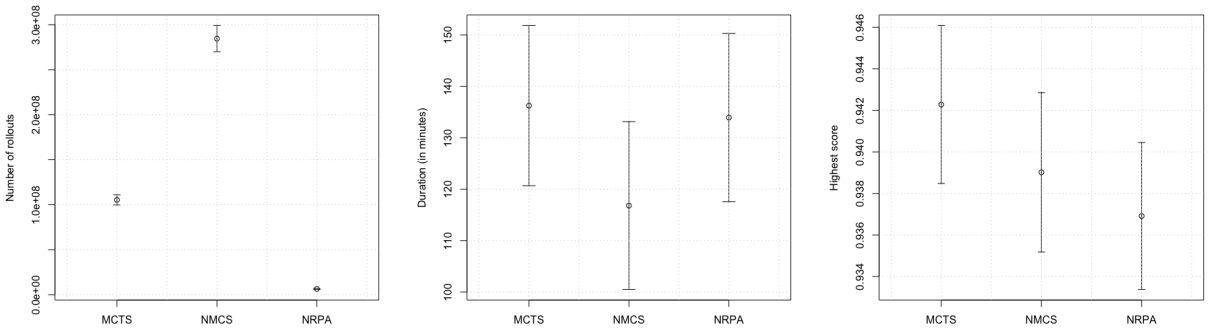
Highest Score

The last plot, Subfigure 4c, shows the 95% confidence intervals for the highest score reached by the three algorithms. Their means are $\bar{x}_{scoreMCTS} \approx 0.9423$, $\bar{x}_{scoreNMCS} \approx 0.939$ and $\bar{x}_{scoreNRPA} \approx 0.9369$ for MCTS [8, 15], NMCS [5] and NRPA [19], respectively. These values are rounded to the nearest ten thousandths. It is visible that all three means and confidence intervals result in a score above 0.93. Considering that the highest possible score is 1 all three algorithms performed well. From this observation one could make the assumption that a more granular and detailed evaluation function would have been applicable, making it harder for the algorithms to reach scores close to the maximum.

As a next step three null hypotheses, namely that for any of the three possible combinations of two algorithms their means are equal, are set up. For all three combinations a Welch Two Sample t-test is performed. The p-values are 0.2327, 0.04179 and 0.4259 for MCTS vs NMCS, MCTS vs NRPA and NMCS vs NRPA, respectively. From this result it is safe to say that, with the exception of MCTS vs NRPA, the null hypotheses can not be rejected. For MCTS vs NRPA the p-value indicates that there is a statistically significant difference between the means of the two algorithms. Additionally taking into concern the data shown in Subfigure 4c it becomes visible that MCTS outperformed NRPA. Following this line of reasoning MCTS with UCT will be considered as a candidate for the user study in the next subsection.

5.3 User Study

To properly answer the research questions of this thesis it is important to let participants play the generated levels. The level generators under consideration here are



(a) Number of rollouts (b) Minutes until highest score (c) Highest score

Figure 4: Results of determining the highest scoring algorithm. The above figures show the 95% confidence intervals with sample size 120.

the sample Random Generator [13], the improved constructive generator, the GA Search-Based Generator and the highest scoring algorithm from Subsection 5.2, the MCTS generator.

Setup of the Experiment

An online-survey was conducted, where participants were asked to download and execute a .jar-file. This survey-program is a modified version of the software used to carry out the original pilot study. Upon starting the program the participants were first informed about the intention of the study, which is to compare preferability of levels generated by different algorithms. Next, one of the four games mentioned in Subsection 5.2 was randomly chosen for them. The participants were then presented with a handmade tutorial-level to get used to the purpose and the controls of the selected game. Afterwards they were presented with one level from one randomly chosen generators and upon finishing it with a level from a different randomly chosen generator. Each of the levels for each generator came from a pool of five possible levels, where again always one was chosen randomly. With four different algorithms there are six different combinations of pairing two of these algorithms. Thus a complete run of the experiment consisted of playing six pairs of levels, where the order of the two paired algorithms was also randomised by the program. After playing one pair, the participants were asked to state their preference, if any, over the two recently played levels. The possible answers to the question *Which level do you prefer?* were *Neither*, *First*, *Second* and *Equal*. After all six combinations of algorithms were presented once, the experiment started over with a different randomly chosen game. It was left to the participants to decide when to stop the program, as it would otherwise continue forever, but it was suggested that they at least finish one complete run.

Setting up and carrying out the experiment was done similarly to the original pilot study conducted by Khalifa et al. [13]. The two studies differ only in the number of algorithms being compared and partly in the games available to the participants. In this user study four algorithms were compared, while the original study only concerned three algorithms. This results in setting up six instead of three null hypotheses, each stating that there is no preference between any two chosen algorithms. Each of these hypotheses is likewise analysed using a two-tailed binomial test between the number of times an algorithm was preferred and not preferred. Contrary to the initial research the *Neither* and *Equal* answers are only discarded for the binomial test, but also discussed. In the initial pilot study these answers were discarded completely.

Results of the Experiment

The survey program, along with a short motivation, was posted in Facebook groups related to studying at DKE as well as distributed to friends and family. From this, the age-range of the participants can be assumed to be between 18 and 55 years. As the survey did not collect any user-specific information, apart from a unique user-id, no statement can be made about the distribution of male and female participants or their occupation. Over the duration of one week 453 responses were collected from 26 different participants. All participants completed at least one whole run (six responses) and in total 74 whole runs were conducted. The remaining 9 responses were not discarded as they only provide additional information. The results of the experiment are shown in Table 1, where each row corresponds to one of the six possible combinations of algorithms.

From the information in Table 1 it becomes clear

	Preferred	Not preferred	Neither	Equal	Binomial p-value
Improved Constructive vs Random	61	4	10	1	3.919×10^{-14}
GA Search-based vs Random	36	8	27	5	2.545×10^{-5}
MCTS Search-based vs Random	37	5	27	5	4.434×10^{-7}
GA Search-based vs Improved Constructive	17	43	7	7	0.001066
MCTS Search-based vs Improved Constructive	14	43	9	10	0.0001539
MCTS Search-based vs GA Search-based	26	19	25	7	0.3713

Table 1: In the table above the rows represent the possible combinations of two algorithms. For each combination the number and kind of collected responses is given. The two-sided binomial test was conducted between the *Preferred* and *Not preferred* responses and expected them to be equal.

that we can reject the null hypotheses for the first five combinations of algorithms. This implies that in these cases the alternative hypothesis, namely that there is a significant difference in preferability, holds. The sixth null hypothesis, stating that there is no significant difference between the GA Search-based algorithm [13] and the MCTS Search-based algorithm can not be rejected.

Upon closer inspecting the data, some interesting observations can be made. The first example is the first combination of algorithms, the improved constructive generator and the Random Generator [13]. It is clearly visible that the number of times the improved constructive generator, implemented in the course of this thesis, was preferred over the Random Generator outweighs the number of times it was not preferred. Also for the second and the third combination of algorithms, the Random Generator versus the GA Search-based and MCTS Search-based generator, respectively, the later were more preferred. For the next two combinations, the improved constructive generator versus the GA Search-based and MCTS Search-based generator, respectively, an interesting observation can be made. While the p-value only indicates that there is a significant difference in preferability, we can only make statements about which algorithm was preferred more in terms of the collected responses. In both cases the improved constructive generator was preferred more often than the two search-based generators. Since we were unable to reject the null hypothesis for the last combination, the MCTS versus the GA Search-based Generator, we can only again argue in terms of the available data. The MCTS Search-based generator, implemented in the course of this thesis, was preferred slightly more often than the original GA Search-based Generator.

Apart from that the high number of *Neither* responses needs to be pointed out. One reason for participants to respond in such a way would be if none of the two presented levels were preferable. With 105 out of 453 responses being *Neither*, almost one fourth of all pairs of levels were either not enjoyable or not playable. *Equal* was only selected as answer in approximately 7% of the responses.

Discussion of the Experiment

In the initial pilot study [13] no significant difference in preference could be observed between the Constructive and the Random Generator. This was the reason for setting up the first research question of this thesis and trying to improve the Constructive Generator. From the above described results it is clearly visible that improving the constructive generator had the desired effect. The preferability of the improved constructive over the Random Generator was the strongest of all possible combinations of algorithms.

The second research question of this thesis can also be answered from the previously presented results. Failure to reject the null hypothesis for the corresponding combination of algorithms does not directly imply that it holds. Yet from the gathered data it becomes clear that the MCTS and the GA Search-based Generator were nearly equally likely preferred, despite using different kinds of evaluation functions. It leads to the assumption that MC search-based algorithms can indeed be applied successfully to the problem of level generation.

Other insights on the problem at hand can be formulated from the results of the experiment. First, the improved constructive generator was the most preferred algorithm over all combinations. This can be explained by the fact that it utilises the information about the games in a different way. As a result the underlying concepts of the games are represented more properly than in the case of the search-based algorithms. One example for this is the game *Aliens*, where the concept of the game is an avatar placed at the bottom of the screen, having to shoot targets that are placed above him. In the VGDL [10] representation this concept is only represented by limiting the avatars movement to a horizontal line. While in the improved constructive generator this can lead to placing the avatar on the bottom of the level, the search-based algorithms tended to place the avatar randomly, violating the idea of the game. This is supported by the fact that the winner of the 2016

GVG-LG competition also was an algorithm that is of constructive nature [18]. *Easablade*, a multi-layered cellular automaton, won the competition by a huge margin. It used the VGDL description of the games to decide on the parameters of the cellular automaton and iteratively constructed a level from that. Additionally this set the foundation for starting to develop *Marahel*, a description language for constructive generators, by Khalifa et al. [11]. The second valuable insight is the high number of *Neither* responses. They indicate that the generated levels are far from being perfect. Here it seems to be unimportant whether the level was generated using *Simulation-based Fitness Functions* or *Direct Fitness Functions* [22]. Obviously with the later it is very hard to ensure playability or even to represent the concept behind games using a measurable quantity. But even with *Simulation-based Fitness Functions* the same problem is observable. While there might be an agent capable of playing the generated level, there is no guarantee that the way he finished the level represented the concept of the underlying game. This problem was already partly tackled during the initial research, by having an agent that behaves more human-like [12]. Yet, until the agent acts according to the idea of the game at hand, there seems to be no huge improvement in quality of the levels when utilising *Simulation-based Fitness Functions* over *Direct Fitness Functions*.

6 Conclusions & Future Research

The first conclusion to draw from the research conducted in this thesis is that the original Constructive Generator can be improved to become distinguishable from the Random Generator. It even becomes the most preferred algorithm of the four algorithms under consideration in the user study. This conclusion at the same time is the answer to the first research question. The second conclusion, in turn answering the second research question, is that MC search algorithms [3] can as equally likely be applied to the problem of level generation as the genetic algorithm. Also results similar to the original GA Search-based generator [13] can be reached by the MC algorithms without needing the full five hour time-window. This thesis also points out that the main issue with search-based level generators in general is not the algorithm itself, but the underlying evaluation functions. In addition, the importance of the information encoded in the VGDL [10] representation of the games was learned in the course of this thesis, which becomes visible from the high preference of the improved constructive level generator. Another conclusion is that all levels, generated in the course of this thesis and the original pilot study, are far from being perfect, with nearly 25% of responses by participants of the user study being *Neither*.

While work on further improving the constructive level generators has already been started [11], the focus for future research can be set into another direction. One obvious next step is to further improve on the evaluation functions utilised in the different search-based generators. Another possible field of study is to apply deep learning algorithms to the problem of level generation. For both of those suggestions it would be helpful to have a larger pool of handcrafted levels available for all games represented in VGDL [10]. From these either a suitable evaluation function or the ideal layout of a level could directly be learned. Alternatively these handcrafted levels could be used to record the actions of a human player playing the level, as well as his experience [23]. In turn this information could then be applied to generate levels that truly incorporate the concept of the game at hand, rather than forcibly making the level playable at the cost of ignoring its concept.

References

[1] Auer, Peter, Cesa-Bianchi, Nicolò, and Fischer, Paul (2002). Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, Vol. 47, No. 3, pp. 235–256.

[2] Bontchev, Boyan (2017). Modern Trends in Automatic Generation of Content for Video Games. *Serdica Journal of Computing*, Vol. 10, No. 2, pp. 133–166.

[3] Browne, Cameron B., Powley, Edward, Whitehouse, Daniel, Lucas, Simon M., Cowling, Peter I., Rohlfshagen, Philipp, Tavener, Stephen, Perez, Diego, Samothrakis, Spyridon, and Colton, Simon (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, No. 1, pp. 1–43.

[4] Cazenave, Tristan and Diemert, Eustache (2018). Memorizing the Playout Policy. *Computer Games. CGW 2017. Communications in Computer and Information Science* (eds. Tristan Cazenave, Mark H. M. Winands, and Abdallah Saffidine), Vol. 818, pp. 96–107, Springer, Cham.

[5] Cazenave, Tristan (2009). Nested Monte-Carlo Search. *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, pp. 456–461, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[6] Cazenave, Tristan (2010). Nested Monte-Carlo Expression Discovery. *Proceedings of the 2010*

Conference on ECAI 2010: 19th European Conference on Artificial Intelligence, pp. 1057–1058, IOS Press, Amsterdam, The Netherlands.

[7] Chaslot, Guillaume M. J.-B., Winands, Mark H. M., Uiterwijk, Jos W. H. M., van den Herik, H. Jaap, and Bouzy, Bruno (2007). Progressive strategies for Monte-Carlo Tree Search. *Information Sciences 2007*, pp. 655–661. World Scientific.

[8] Coulom, Rémi (2007). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Proceedings of the 5th International Conference on Computers and Games* (eds. H. Jaap van den Herik, Paolo Ciancarini, and H. H. L M. Donkers), CG'06, pp. 72–83, Springer-Verlag, Berlin, Heidelberg.

[9] Dieskau, Jens (2016). Multi-objective Procedural Level Generation for General Video Game Playing. M.Sc. thesis, Otto-von-Guericke University Magdeburg, Faculty of Computer Science, Germany.

[10] Ebner, Marc, Levine, John, Lucas, Simon M, Schaul, Tom, Thompson, Tommy, and Togelius, Julian (2013). Towards a video game description language. *Artificial and Computational Intelligence in Games*, Vol. 6 of *Dagstuhl Follow-ups*, pp. 85–100. Dagstuhl Publishing, Wadern.

[11] Khalifa, Ahmed and Togelius, Julian (2017). Marahel: A Language for Constructive Level Generation. Technical report, Tandon School of Engineering, New York University, New York, USA.

[12] Khalifa, Ahmed, Isaksen, Aaron, Togelius, Julian, and Nealen, Andy (2016a). Modifying MCTS for Human-Like General Video Game Playing. *IJCAI 2016*, pp. 2514–2520.

[13] Khalifa, Ahmed, Perez-Liebana, Diego, Lucas, Simon M, and Togelius, Julian (2016b). General video game level generation. *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pp. 253–259, ACM.

[14] Kimbrough, Steven Orla, Koehler, Gary J., Lu, Ming, and Wood, David Harlan (2008). On a Feasible-Infeasible Two-Population (FI-2Pop) genetic algorithm for constrained optimization: Distance tracing and no free lunch. *European Journal of Operational Research*, Vol. 190, pp. 310–327.

[15] Kocsis, Levente and Szepesvári, Csaba (2006). Bandit Based Monte-Carlo Planning. *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, pp. 282–293, Springer-Verlag, Berlin, Heidelberg.

[16] Levine, John, Congdon, Clare Bates, Ebner, Marc, Kendall, Graham, Lucas, Simon M., Miikkulainen, Risto, Schaul, Tom, and Thompson, Tommy (2013). General Video Game Playing. *Artificial and Computational Intelligence in Games* (eds. Simon M. Lucas, Michael Mateas, Mike Preuss, Pieter Spronck, and Julian Togelius), Vol. 6 of *Dagstuhl Follow-Ups*, pp. 77–83. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.

[17] Perez-Liebana, Diego, Samothrakis, Spyridon, Togelius, Julian, Schaul, Tom, Lucas, Simon M., Couetoux, Adrien, Lee, Jerry, Lim, Chong-U, and Thompson, Tommy (2016). The 2014 General Video Game Playing Competition. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 8, No. 3, pp. 229–243.

[18] Perez-Liebana, Diego, Liu, Jialin, Khalifa, Ahmed, Gaina, Raluca D., Togelius, Julian, and Lucas, Simon M. (2018). General Video Game AI: a Multi-Track Framework for Evaluating Agents, Games and Content Generation Algorithms. *arXiv preprint arXiv:1802.10363*.

[19] Rosin, Christopher D. (2011). Nested Rollout Policy Adaptation for Monte Carlo Tree Search. *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence* (ed. Toby Walsh), Vol. 1, pp. 649–654, AAAI Press/International Joint Conferences on Artificial Intelligence, Menlo Park, California.

[20] Schadd, Maarten P. D., Winands, Mark H. M., van den Herik, H. Jaap, Chaslot, Guillaume M. J.-B., and Uiterwijk, Jos W. H. M. (2008). Single-player Monte-Carlo Tree Search. *International Conference on Computers and Games* (eds. H. Jaap van den Herik, H. J. Xinhe, Z. Ma, and Mark H. M. Winands), pp. 1–12, Springer.

[21] Tak, M. J. W., Winands, M. H. M., and Bjornsson, Y. (2012). N-Grams and the Last-Good-Reply Policy Applied in General Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, No. 2, pp. 73–83.

[22] Togelius, Julian, Yannakakis, Georgios N., Stanley, Kenneth O., and Browne, Cameron B. (2010). Search-based procedural content generation. *European Conference on the Applications of Evolutionary Computation*, pp. 141–150, Springer.

[23] Yannakakis, Georgios N. and Togelius, Julian (2011). Experience-driven procedural content

generation. *IEEE Transactions on Affective Computing*, Vol. 2, No. 3, pp. 147–161.