General Video Game Evaluation Using Relative Algorithm Performance Profiles

Thorbjørn S. Nielsen[1], Gabriella A.B. Barros[1], Julian Togelius[2], and Mark J. Nelson[3(✉)]

[1] Center for Computer Games Research, IT University of Copenhagen, Copenhagen, Denmark
{thse,gbar}@itu.dk
[2] Department of Computer Science and Engineering, New York University, New York, NY, USA
julian@togelius.com
[3] Anadrome Research, Copenhagen, Denmark
mjn@anadrome.org

Abstract. In order to generate complete games through evolution we need generic and reliably evaluation functions for games. It has been suggested that game quality could be characterised through playing a game with different controllers and comparing their performance. This paper explores that idea through investigating the relative performance of different general game-playing algorithms. Seven game-playing algorithms was used to play several hand-designed, mutated and randomly generated VGDL game descriptions. Results discussed appear to support the conjecture that well-designed games have, in average, a higher performance difference between better and worse game-playing algorithms.

1 Introduction

How well do knowledge-free algorithms play really bad video games? This might not be a question that has kept you awake at night, but as we shall show there are excellent reasons to consider it. Reasons having to do with understanding fundamental design characteristics of a broad class of simple video games, and laying the groundwork for automatically generating such games.

One way to generate complete games might be to search a space of games represented in a programming language like C or Java. However, the proportion of programs in such languages that can in any way be considered a game is quite small. Increasing the density of games in the search space can be achieved by searching programs defined in a game description language (GDL) designed to encode games.

Even searching a reasonably well defined space of games still supposes that we have a way of automatically telling good games from bad games (or not-quite-so-bad games from really bad games). In other words, we need a fitness function. Part of the fitness function could consist in inspecting the rules as expressed in the GDL, e.g. to make sure that there are winning conditions which could

A.M. Mora and G. Squillero (Eds.): EvoApplications 2015, LNCS 9028, pp. 369–380, 2015.
DOI: 10.1007/978-3-319-16549-3_30

in principle be fulfilled. But there are many bad games that fulfil such criteria. To really understand a game, you need to play it. It seems the fitness function therefore needs to incorporate a capacity to play the games it is evaluating.

This game-playing capacity needs to be *general*, because we know almost nothing about the games that will be evaluated. We can therefore not incorporate any domain knowledge about these games; we need algorithms that are as *knowledge-free* as possible. Examples of such algorithms are the various tree-search algorithms, such as Minimax and Monte Carlo tree search (MCTS), that have been widely used for playing various games. But online evolutionary algorithms might also be used as knowledge-free algorithms. In case a heuristic representing the quality of a particular in-game state is need, such a heuristic should be as neutral as possible, e.g. the score of the game.

Just being able to play a game does not in itself tell us how good the game is. Many boring games are perfectly playable by an algorithm. And because we don't know the game, we don't know what constitutes good or bad play, compared to how well or badly the game could be played. Instead we propose a measure of *relative* performance between algorithms, the Relative Algorithm Performance Profile (RAPP). The intuition is that good games are likely to have high skill differentiation: good players get better outcomes than bad players. A game that is *insensitive* to skill, by contrast, is not likely to be a good one. We therefore formulate the following hypothesis: the performance difference (measured as score and/or win-rate) between generally better game-playing algorithms and generally worse game-playing algorithms is on average higher for well-designed games than for poorly designed games. But there might very well be other interesting differences between classes of games that can be discerned by looking at the performance profiles of sets of game-playing algorithms; this study is intended as a preliminary investigation into the hypothesis that RAPP is a productive way of differentiating games.

We carry out this investigation using the General Video Game Playing platform (GVG-AI) and its associated Video Game Description Language (VGDL). This framework makes 20 hand-designed games available, mostly versions of well-known arcade games. We contrast those games with a large number of randomly generated games in the same language, and with a large number of "mutations" of the hand-designed games. A core assumption we make is that the hand-designed games are, on average, better designed than the randomly generated ones. We calculate a performance profile using several game-playing algorithms available in the GVG-AI framework and some new algorithms. The concrete contributions of this paper thus include two new variations on Monte Carlo-based game-playing, as well as a quantitative investigation of the performance profiles of these algorithms and the associated methodology for performing this study. However, we primarily see this work as groundwork for a reliable game fitness function, that will eventually allow us to generate good new sets of game rules.

2 Background

The idea of generating complete games through algorithms is not itself new. The problem in full generality is quite large, so usually a subset of the general

problem is tackled. Videogames may be comprised of a large number of tangible and intangible components, including rules, graphical assets, genre conventions, cultural context, controllers, character design, story and dialog, screen-based information displays, and so on [2,7,8].

In this paper we look specifically at generating game rules; and more specifically the rules of arcade-style games based on graphical movement and local interaction between game elements, represented in VGDL. The two main approaches that have been explored in generating game rules are reasoning through constraint solving [11] and search through evolutionary computation or similar stochastic optimisation [1,4,13].

Generating a set of rules that makes for an interesting and fun game is a hard task. The arguably most successful attempt so far, Browne's Ludi system, produced a new board game of sufficient quality to be sold as a boxed product [1]. However, it succeeded partly due to restricting its generation domain to the rules of a rather tightly constrained space of board games. A key stumbling block for search-based approaches to game generation is the fitness/evaluation function. This function takes a complete game as input and outputs an estimate of its quality. Ludi uses a mixture of several measures based on automatic play of games, including balance, drawishness and outcome uncertainty. These measures are well-chosen for two-player board games, but may not transfer well to video games or single-player games, which have in a separate analysis been deemed to be good targets for game generation [12]. Other researchers have attempted evaluation functions based on the learnability of the game by an algorithm [13] or an earlier and more primitive version of the characteristic that is explored in this paper, performance profile of a set of algorithms [4].

2.1 Game Description Languages

Regardless of which approach to game generation is chosen, one needs a way to represent the games that are being created.[1] For a sufficiently general description of games, it stands to reason that the games are represented in a reasonably generic language, where every syntactically valid game description can be loaded into a specialised game engine and executed. There have been several attempts to design such GDLs. One of the more well-known is the Stanford GDL, which is used for the General Game Playing Competition [5]. That language is tailored to describing board games and similar discrete, turn-based games; it is also arguably too verbose and low-level to support search-based game generation. The various game generation attempts discussed above feature their own GDLs of different levels of sophistication; however, there has not until recently been a GDL for suitably large space of video games.

2.2 VGDL

The Video Game Description Language (VGDL) is a GDL designed to express 2D arcade-style video games of the type common on hardware such as the Atari

[1] See [9] for a discussion of game-rule representation choices.

2600 and Commodore 64. It can express a large variety of games in which the player controls a moving avatar (player character) and where the rules primarily define what happens when objects interact with each other in a two-dimensional space. VGDL was designed by a set of researchers [3,6] (and implemented by Schaul [10]) in order to support both general video game playing and video game generation. The language has an internal set of classes, properties and types that each object can defined by.

Objects have physical properties (i.e. position, direction) which can be altered either by the properties defined, or by interactions defined between specific objects. A VGDL description has four parts: the SpriteSet, which defines the ontology of the game – which sprites exist and what can they do; the LevelMapping, which maps from level description to game state; the InteractionSet, which defines what happens when sprites overlap, and the TerminationSet which defines how the game can be won or last.

2.3 The GVG-AI Framework

The GVG-AI framework is a testbed for testing general game playing controllers against games specified using VGDL. Controllers are called once at the beginning of each game for setup, and then once per clock tick to select an action. Controllers do not have access to the VGDL descriptions of the games. They receive only the game's current state, passed as a parameter when the controller is asked for a move. However these states can be forward-simulated to future states. Thus the game rules are not directly available, but a simulatable model of the game can be used.

```
BasicGame
   SpriteSet
      city > Immovable color=GREEN img=city
      explosion > Flicker limit=5 img=explosion
      movable >
         avatar > ShootAvatar stype=explosion
         incoming >
            incoming_slow > Chaser stype=city color=ORANGE speed=0.1
            incoming_fast > Chaser stype=city color=YELLOW speed=0.3

   LevelMapping
      c > city
      m > incoming_slow
      f > incoming_fast

   InteractionSet
      movable wall > stepBack
      incoming city > killSprite
      city incoming > killSprite scoreChange=-1
      incoming explosion > killSprite scoreChange=2

   TerminationSet
      SpriteCounter stype=city win=False
      SpriteCounter stype=incoming win=True
```

Fig. 1. Example of VGDL description - a simple implementation of the game Missile Command

The framework additionally contains 20 hand-designed games, which mostly consist of interpretations of classic video games. Figure 1 shows the VGDL description of the game *Missile Command*.

3 Method

There are three types of games tested: human-written VGDL games, mutated versions of those games, and randomly generated games.

3.1 Example Games

Two of the 20 games from the GVG-AI framework were deemed too monotonous after initial tests. In these two games the controllers all had similar scores for each run—or only one controller was able to increase its score. The remaining 18 hand-designed VGDL game descriptions are used as the baseline. Most are inspired

Fig. 2. A visual representation of a few of the VGDL example games. From top-left: *Zelda*, *Portals* and *Boulderdash*

by classic arcade (e.g. Boulderdash, Frogger, Missile Command and Pacman). The player controls a single avatar which must be moved quickly around in a 2D setting to win, or to get a high score. The player can increment a score counter in all of the games. A brief descriptions of each game (Fig. 2):

Aliens Based on *Space Invaders*. Aliens are spawned from the top of the screen; the player wins by shooting them all. **Boulderdash** Based on *Boulder Dash*. The avatar has to dig through a cave to collect diamonds while avoiding being smashed by falling rocks or killed by enemies. **Butterflies** The avatar has to capture all butterflies before all the cocoons are opened. Cocoons open when a butterfly touches them. **Chase** Chase and kill fleeing goats. However, if a fleeing goat encounters the corpse of another, it gets angry and starts chasing the player instead. **Digdug** Base on *Dig Dug*. The avatar collects gold coins and gems, digs through a cave, and avoids or shoots boulders at enemies. **Eggomania** Based on *Eggomania*. The avatar moves from left to right collecting eggs that fall from a chicken at the top of the screen, in order to use these eggs to shoot at the chicken, killing it. **Firecaster** The goal is to reach the exit by burning wood that is on the way. Ammunition is required to set things on fire. **Firestorms** The player must avoid flames from hell gates until reaching the exit of a maze. **Frogs** Based on *Frogger*. The player is a frog that has to cross a road and a river, without getting killed. **Infection** The objective is to infect all healthy animals. The player gets infected by touching a bug. Medics can cure infected animals. **Missile Command** Based on *Missile Command*. The player has to destroy falling missiles, before they reach their destinations. The player wins if any cities are saved. **Overload** The player must get to the exit after collecting coins. But too many coins make the player too heavy to pass through the exit. **Pacman** Based on *Pac-Man*. The goal is to clear a maze full of power pills and pellets, and avoid or destroy ghosts. **Portals** The objective is to get to a certain point using portals to go from one place to another, while at the same time avoiding lasers. **Seaquest** Based on *Seaquest*. The avatar is a submarine that rescue divers and avoids sea animals that can kill it. The goal is to maximise score. **Survive Zombies** The player has to flee zombies until time runs out, and can collect honey to kill the zombies. **Whackamole** Based on *Whac-a-Mole*. Must collect moles that appear from holes, and avoid a cat that mimics the moles. **Zelda** Based on *Legend of Zelda*. The objective is to find a key in a maze and leave the level. The player also has a sword to defend against enemies.

3.2 Controllers

Seven general videogame controllers were used to test the games. The controllers use different approaches, with a varying degree of intelligence. Three of the controllers are included in the GVG-AI framework, while the remaining were implemented for this work. Except for *OneStep-Heuristic*, the controllers only evaluate a given state according to its score and win/loss status.

MCTS. GVG-AI sample controller. "Vanilla" MCTS using UCT.

GA. GVG-AI sample controller. Uses a genetic algorithm to evolve a sequence of actions.

OneStep-Heuristic. GVG-AI sample controller. Heuristically evaluates the states reachable through one-step lookahead. The heuristic takes into account the locations of NPCs and certain other objects.

OneStep-Score. Similar to *OneStep-heuristic*, but only uses the score and win/loss status to evalue states.

Random. Chooses a random action from those available in the current state.

DoNothing. Returns a nil action. Literally does nothing.

Explorer. Design specifically to play the arcade-style games of the GVG-AI framework. Unlike the other controllers which utilise open-loop searches, it stores information about visited tiles and prefers visiting unvisited locations. Also addresses a common element of the VGDL example games, randomness. The controller gains an advantage by simulating the results of actions repeatedly, before deciding the best move.

3.3 Mutation of Example Games

A mutation process was repeatedly applied for each of the 20 example games mentioned in Sect. 3.1. The process consisted of changing the set of interaction rules (i.e. lines from the InteactionSet) defined in each game description. For each mutation, each interaction rule had a 25 % chance of being mutated, but with a requirement that at least one rule were changed. Mutation occurred by changing the objects in that interaction rule, the function on collision between said objects, and/or the function's parameters.

Several contrains were used during each mutation to avoid games with non-valid descriptions (which can cause crashes in the GVG-AI framework). Additionally, several constraints were used for the different function parameters, as to only allow "realistic" values. The range of these constraints were extrapolated (and slightly extended) from the example games. For instance, the parameter *limit* used by certain rules was limited to values between 0 and 10, as the same is true for the rules of the example games. This process was applied 10 times for each example game description, resulting in 200 generated game descriptions.

When testing the mutated games the same level descriptions as for their original counterparts were used (those mentioned in Sect. 3.1).

3.4 Random Game Generation

A set of 400 random VGDL game descriptions were generated by constructing the textual lines for different parts of a VGDL description: Generating an array of sprites (for the SpriteSet), interaction-rules (InteractionSet), termination-rules (TerminationSet) and level mappings (LevelMapping) (Fig. 3).

Before generating descriptions, we used similar constraints to those in Sect. 3.3, partly to avoid generating descriptions with invalid elements, and partly to increase the proportion of interesting outcomes. The number of sprites, interaction, and termination rules were randomly chosen, limited to 25, 25, and 2,

Fig. 3. Visual representation of one of the 400 randomly generated VGDL games

respectively. A simple level (only containing one of each sprite) was generated for each of the generated games for test purposes.

4 Results

The seven controllers mentioned in Sect. 3.2 were used to play through a set of example-, mutated and randomly generated games. Because of CPU budget limitations, each game was played for a maximum of 800 clock ticks, and each controller was restricted to use 50 ms on each tick. In the following sections, we show results of these tests, analyse the average of all play-throughs for each controller, and compare the results with each other.

To more accurately compare the score for the different controllers across the range of different games, we normalise each score using a max-min normalisation. Normalised averages and win rate averages are shown in Figs. 7 and 8, respectively. In Fig. 7, it is possible to see that the difference between the highest and lowest scores is greater in the example and mutated games than in the generated games. On the other hand, the average win rate of generated games surpasses both examples and mutated games, as shown in Fig. 8.

In addition to the score and win-rate, the average entropy of actions chosen for the player avatar is shown in the tables below.

4.1 Example Games

Averages and win-rates from the 18 human-designed example games are shown in Fig. 4. The distributions of normalised scores show that more intelligent controllers tend to have more success. It is worth noticing that the *score mean* and *normalised score mean* have slightly different orderings. Notice also that distributions are slightly different when analysing the results of individual games. For instance, in *Aliens*, Random has a higher average than Onestep.

controller	*score mean*	*std.dev.*	*normalised-mean*	*winrate*	*act-entropy*
Explorer	42.94	121.55	0.7966	0.4467	0.8860
MCTS	23.14	86.39	0.4935	0.2489	0.9006
GA	11.98	51.08	0.4010	0.1533	0.7052
Onestep-S	14.48	27.28	0.4149	0.0844	0.8848
Onestep-H	3.73	9.81	0.2350	0.1111	0.1943
Random	7.52	17.01	0.2493	0.0556	0.9016
DoNothing	0.39	4.02	0.1317	0.0556	0

Fig. 4. Results from the 20 example games

4.2 Mutated Example Games

When mutating games, two types of games are problematic: Games where the controllers never increase their score (and never win), and games where too many objects are created and each frame end up taking too long (>50 ms). We exclude both types of games in the following analysis.

Averages from playing the remaining 146 mutated games (of 200 total) are shown in Fig. 5. The scores have higher means and standard deviations, indicating outliers in the data. The ordering of the *normalised score mean*, however, shows a similar pattern as for the example games, with Explorer again excelling.

controller	*score mean*	*std.dev.*	*normalised-mean*	*win-rate*	*act-entropy*
Explorer	392.08	6441.77	0.8361	0.3055	0.8372
MCTS	140.49	1097.93	0.4799	0.1510	0.9012
GA	88.96	756.92	0.4254	0.1274	0.6693
Onestep-S	128.82	706.79	0.4418	0.1049	0.8444
Onestep-H	82.67	762.93	0.2580	0.1866	0.2033
Random	69.70	666.81	0.2581	0.1077	0.9030
DoNothing	81.55	945.71	0.1819	0.0959	0

Fig. 5. Results from mutated games

4.3 Randomly Generated Games

Figure 6 shows results for the 65 randomly generated games, with problematic games removed according to the same criteria as in the previous section.

First of all, *score std. deviations* are much higher than in the previous games, with the minimum being 199,406.58, over 1500 times larger than the highest in the set of example games (i.e. 121.55, by Explorer). Clearly, only the *normalised mean* can be on this set to compare scores across the different game types. The *normalised score means* and *win-rates* both have values that are more closely clustered together, than in the previous game sets.

controller	*score mean*	*std.dev.*	*normalised-mean*	*win-rate*	*act-entropy*
Explorer	−18 207.91	227 282.72	0.6193	0.2566	0.7193
MCTS	−4035.85	258 890.78	0.4395	0.2769	0.8392
GA	−3501.65	262 508.97	0.4399	0.2480	0.5767
Onestep-S	−16 680.67	231 600.95	0.3916	0.2191	0.8197
Onestep-H	−25 728.97	195 365.78	0.3640	0.2025	0.4022
Random	−23 348.24	199 405.58	0.3195	0.2105	0.8553
DoNothing	−3051.34	259 562.33	0.3747	0.1846	0

Fig. 6. Results from randomly generated games

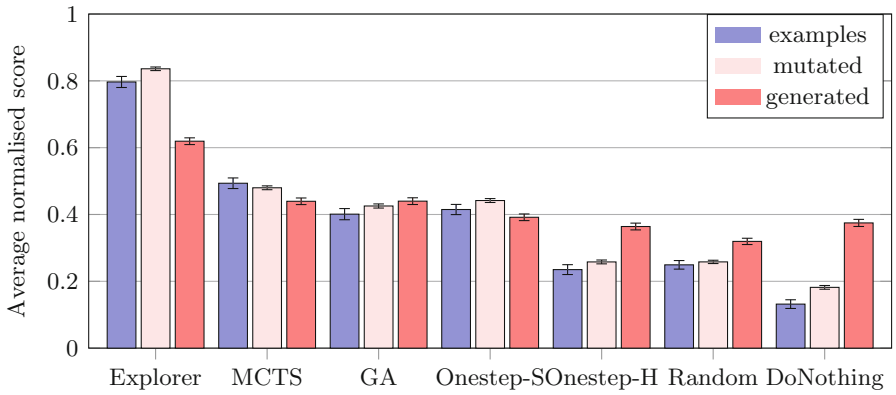

Fig. 7. Average normalised score

5 Discussion

The results in Sect. 4 display some interesting patterns. Win rates suggest a relationship between intelligent controllers' success and better game design; for better designed games, the relative performance of different types of algorithms differ more. This corroborates our hypothesis that RAPPs can be used to differentiate between games of different quality. In randomly generated games, which arguably tend to be less interesting than the others, smarter controllers (e.g. Explorer and MCTS) do only slightly better than the worse ones (i.e. Random and DoNothing). This is due to a general a lack of consistency between rules generated in this manner. Mutated games, however, derive from a designed game. Therefore, they maintain some characteristics of the original idea, which can improve the VGDL description's gameplay and playability. Furthermore, it is interesting that Random and DoNothing do well in some games, as seen in Fig. 8. While it is possible that random actions can result in good outcomes, this chance is very low, especially when compared to the chance of making informed decisions. In spite of that both Random and DoNothing do fairly well in randomly generated games. The performance of DoNothing emerges as a secondary

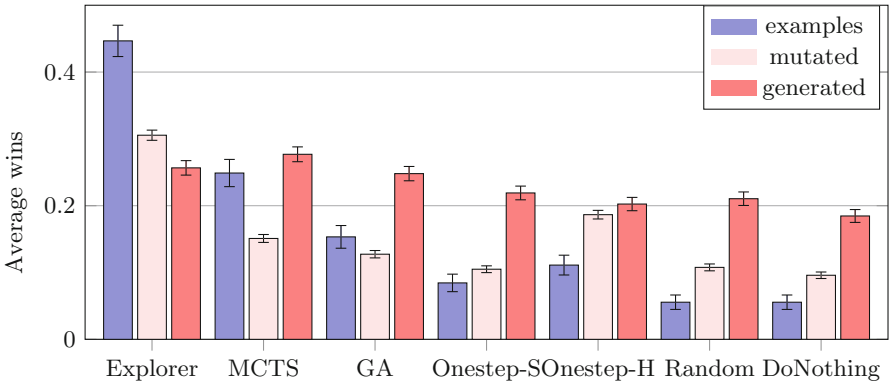

Fig. 8. Average wins

indicator of (good) design: in human-designed games, DoNothing very rarely wins or even scores.

6 Conclusion

Our intent has been to investigate evaluating video games via the performance of game-playing algorithms. We hypothesised that the performance difference between good and bad game-playing algorithms is higher on well-designed games, and therefore can be used as at least a partial proxy for game quality. To test this theory, we had seven controllers with varying levels of skill play 18 human-designed, 146 mutated, and 65 randomly generated VGDL games. The results seem to corroborate our initial conjecture, showing a clear distinction between results of more and less intelligent controllers for human-designed games but not for random games. We also suggest new controllers for GVG-AI: Explorer, OneStep-Score and DoNothing. The first one in particular shows strong overall performance compared to existing baselines such as "vanilla" MCTS.

Acknowledgments. Gabriella A.B. Barros acknowledges financial support from CAPES Scholarship and Science Without Borders program, Bex 1372713-3. Thanks to Diego Perez, Spyros Samothrakis, Tom Schaul, and Simon Lucas for useful discussions.

References

1. Browne, C.: Automatic generation and evaluation of recombination games. Ph.D. thesis, Queensland University of Technology (2008)
2. Cook, M., Colton, S.: Ludus ex machina: building a 3d game designer that competes alongside humans. In: Proceedings of the 5th International Conference on Computational Creativity (2014)

3. Ebner, M., Levine, J., Lucas, S.M., Schaul, T., Thompson, T., Togelius, J.: Towards a video game description language. Dagstuhl Follow-Ups, vol. 6 (2013). http://drops.dagstuhl.de/opus/volltexte/2013/4338/
4. Font, J.M., Mahlmann, T., Manrique, D., Togelius, J.: Towards the automatic generation of card games through grammar-guided genetic programming. In: FDG, pp. 360–363 (2013)
5. Genesereth, M., Love, N., Pell, B.: General game playing: overview of the AAAI competition. AI Mag. **26**(2), 62–72 (2005)
6. Levine, J., Congdon, C.B., Ebner, M., Kendall, G., Lucas, S.M., Miikkulainen, R., Schaul, T., Thompson, T.: General video game playing. Dagstuhl Follow-Ups, vol. 6 (2013). http://drops.dagstuhl.de/opus/volltexte/2013/4337/
7. Liapis, A., Yannakakis, G.N., Togelius, J.: Computational game creativity. In: Proceedings of the 5th International Conference on Computational Creativity (2014)
8. Nelson, M.J., Mateas, M.: Towards automated game design. In: Basili, R., Pazienza, M.T. (eds.) AI*IA 2007. LNCS (LNAI), vol. 4733, pp. 626–637. Springer, Heidelberg (2007)
9. Nelson, M.J., Togelius, J., Browne, C., Cook, M.: Chapter 6: Rules and mechanics. In: Procedural Content Generation in Games: A Textbook and an Overview of Current Research. Springer (2015, to appear). http://www.pcgbook.com
10. Schaul, T.: A video game description language for model-based or interactive learning. In: Proceedings of the 2013 IEEE Conference on Computational Intelligence in Games, pp. 1–8 (2013)
11. Smith, A.M., Mateas, M.: Variations forever: flexibly generating rulesets from a sculptable design space of mini-games. In: Proceedings of the 2010 IEEE Symposium on Computational Intelligence and Games, pp. 273–280 (2010)
12. Togelius, J., Nelson, M.J., Liapis, A.: Characteristics of generatable games. In: Proceedings of the 5th Workshop on Procedural Content Generation in Games (2014)
13. Togelius, J., Schmidhuber, J.: An experiment in automatic game design. In: Proceedings of the 2008 IEEE Symposium on Computational Intelligence and Games, pp. 111–118 (2008)