Rogue-like Games as a Playground for Artificial Intelligence - Evolutionary Approach

Vojtech Cerny and Filip Dechterenko

Charles University in Prague, Czech Republic,
woitee@gmail.com, filip.dechterenko@gmail.com

Abstract. Rogue-likes are difficult computer RPG games set in a procedurally generated environment. Attempts have been made at playing these algorithmically, but few of them succeeded. In this paper, we present a platform for developing artificial intelligence (AI) and creating procedural content generators (PCGs) for a rogue-like game Desktop Dungeons. As an example, we employ evolutionary algorithms to recombine greedy strategies for the game. The resulting AI plays the game better than a hand-designed greedy strategy and similarly well to a mediocre player – winning the game 72% of the time. The platform may be used for additional research leading to improving rogue-like games and general PCGs.

Keywords: artificial intelligence, computer games, evolutionary algorithms, rogue-like

1 Introduction

Rogue-like games, as a branch of the RPG genre, have existed for a long time. They descend from the 1980 game "Rogue" and some old examples, such as NetHack (1987), are played even to this day. Many more of these games are made every year, and their popularity is apparent.

A rogue-like is a single-player, turn-based, highly difficult RPG game, featuring a randomized environment and permanent death[1]. The player takes the role of a hero, who enters the game's environment (often a dungeon) with a very difficult goal. Achieving the goal requires a lot of skill, game experience and perhaps a little bit of luck.

Such a game, bordering between RPG and puzzle genres, is challenging for artificial intelligence (AI) to play. One often needs to balance between being reactive (dealing with current problems) and proactive (planning towards the main goal). Attempts at solving rogue-likes by AI have been previously made [9, 2, 7], usually using a set of hand-coded rules as basic reasoning, and being to some extent successful.

On the other hand, the quality of a rogue-like can heavily depend on its procedural content generator (PCG), which usually creates the whole environment.

[1] The game offers no save/load features, it is always replayed from beginning to end.

Procedural generation [14] has been used in many kinds of games [17, 5], and thus, the call for high-quality PCG is clear [8]. However, evaluating the PCG brings issues [4, 16], such as how to balance between the criteria of high quality and high variability.

But a connection can be made to the former – we could conveniently use the PCG to evaluate the artificial player and similarly, use the AI to evaluate the content generator. The latter may also lead to personalized PCGs (creating content for a specific kind of players) [15].

In this paper, we present a platform for developing AI and PCG for a rogue-like game Desktop Dungeons [11]. It is intended as an alternative to other used AI or PCG platforms, such as the Super Mario AI Benchmark [6] or SpelunkBots [13]. AI platforms have even been created for a few rogue-like games, most notably NetHack [2, 7]. However, Desktop Dungeons has some characteristics making it easier to use than the other. Deterministic actions and short play times help the AI, while small dungeon size simplifies the work of a PCG.

And as such, more experimental and resource demanding approaches may be tried. The platform could also aid other kinds of research or teaching AI, as some people create their own example games for this purpose [12, Chapter 21.2], where Desktop Dungeons could be used instead.

The outline of this paper is as follows. First, we introduce the game to the reader, then we proceed to describe our platform, and finally, we will show how to use it to create a good artificial rogue-like player using evolutionary algorithms.

2 Desktop Dungeons Description

Desktop Dungeons by QCF Design [11] is a single-player computer RPG game that exhibits typical rogue-like features. The player is tasked with entering a dungeon full of monsters and, through careful manipulation and experience gain, slaying the boss (the biggest monster).

Disclaimer: The following explanation is slightly simplified. More thorough and complete rules can be found at the Desktop Dungeons wiki page [1].

2.1 Dungeon

The dungeon is a 20×20 grid viewed from the top. The grid cells may contain monsters, items, glyphs, or the hero (player). Every such object, except for the hero, is static - does not move[2]. Only a 3×3 square around the hero is revealed in the beginning, and the rest must be explored by moving the hero next to it. Screenshot of the dungeon early in the game can be seen in Fig. 1.

[2] Some spells and effects move monsters, but that is quite uncommon and can be ignored for our purpose.

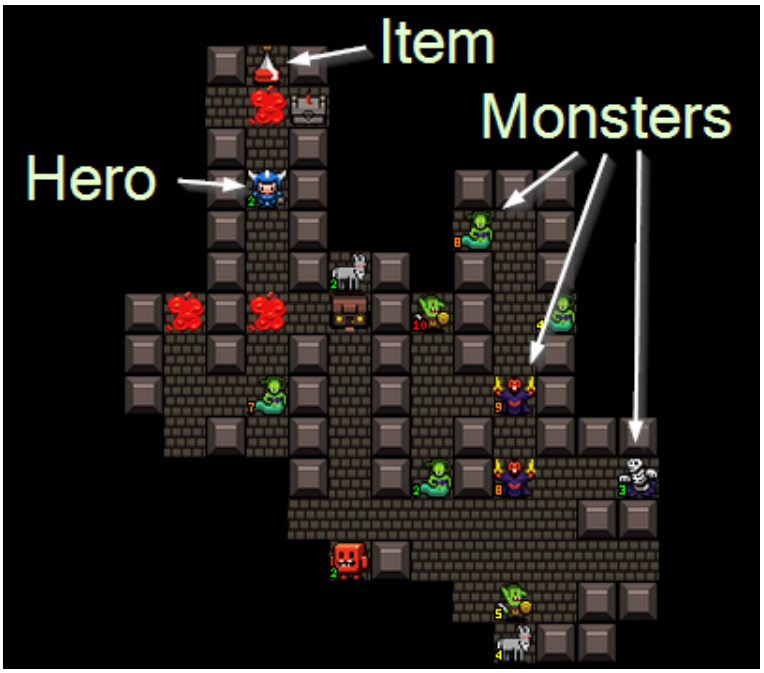

Fig. 1. Screenshot of the dungeon, showing the hero, monsters, and an item (a health potion). The dark areas are the unexplored parts of the dungeon.

2.2 Hero

The hero is the player-controlled character in the dungeon and holds a set of values. Namely: health, mana, attack power, the number of health/mana potions, and his spell glyphs. The hero can also perform a variety of actions. He can attack a monster, explore unrevealed parts of the dungeon, pick up items and glyphs, cast spells or convert glyphs into bonuses.

2.3 Exploring

Unrevealed grid cells can be explored by moving the hero next to them (at least diagonally). Not only does exploration reveal what lies underneath for the rest of the game, but it also serves one additional purpose – restoring health and mana. Every square explored will restore health equal to the hero's level and 1 mana. This means that the dungeon itself is a scarce resource that has to be managed wisely. It shall be noted, though, that monsters heal also when hero explores, so this cannot be used to gain an edge over damaged monsters.

2.4 Combat

Whenever the hero bumps into a monster, a combat exchange happens. The higher level combatant strikes first (monster strikes first when tied). The first attacker reduces his opponent's health by *exactly* his attack power. The other attacker, if alive, then does the same. No other action causes any monster to attack the hero.

2.5 Items

Several kinds of items can be found lying on the ground. These comprise of a Health Powerup, Mana Powerup, Attack Powerup, Health Potion and a Mana Potion. These increase the hero's health, mana, attack power, and amount of health and mana potions respectively.

2.6 Glyphs

Spell glyphs are special items that each allow the hero to cast one kind of spell for it's mana cost. The hero starts with no glyphs, and can find them lying in the dungeon. Common spells include a Fireball spell, that directly deals damage to a monster (without it retaliating), and a Kill Protect spell, that saves the hero from the next killing blow.

Additionally, a spell glyph can be converted to a racial bonus - a specific bonus depending on the hero's race. These are generally small stat increases or an extra potion. The spell cannot be cast anymore, so the hero should only convert glyphs he has little use for.

2.7 Hero Races and Classes

Before entering the dungeon, the player chooses a race (Human, Elf, etc.) and a class (Warrior, Wizard, etc.) of his hero. The race determines only the reward for converting a glyph, but classes can modify the game in a completely unique way.

2.8 Other

The game has a few other unmentioned mechanics. The player can enter special "challenge" dungeons, he can find altars and shops in the dungeon, but all that is far beyond the basics we'll need for our demonstration. As mentioned, more can be found at the Desktop Dungeons wiki [1].

3 AI Platform

Desktop Dungeons has two parameters rarely seen in other similar games. Every action in the game is deterministic[3] (the only unknown is the unrevealed part of the dungeon) and the game is limited to 20×20 grid cells and never extends beyond. These may allow for better and more efficient AI solutions, and may be advantageously utilized when using search techniques, planning, evaluating fitness functions, etc.

On the other hand, Desktop Dungeons is a very interesting environment for AI. It is complex, difficult, and as such can show usefulness of various approaches. Achieving short-term and long-term goals must be balanced, and thus, simple approaches tend to not do well, and must be specifically adjusted for the task. Not much research has been done on solving rogue-like games altogether, only recently was a famous, classic title of this genre — NetHack — beaten by AI [7].

From the perspective of a PCG, Desktop Dungeons is similarly interesting. The size of the dungeon is very limited, so attention to detail should be paid. If one has an artificial player, the PCG could use him as a measure of quality, even at runtime, to produce only the levels the artificial player found enjoyable or challenging.

This is why we created a programming interface (API) to Desktop Dungeons, together with a Java framework for easy AI and PCG prototyping and implementation. We used the alpha version of Desktop Dungeons, because it is more direct, contains less story content and player progress features, runs in a browser, and the main gameplay is essentially the same as in the full version.

The API is a modified part of the game code that can connect to another application, such as our framework, via a WebSocket (TCP) protocol and provide access to the game by sending and receiving messages. A diagram of the API usage is portrayed in Fig. 2.

[3] Some rare effects have probabilistic outcomes, but with a proper game setting, this may be completely ignored.

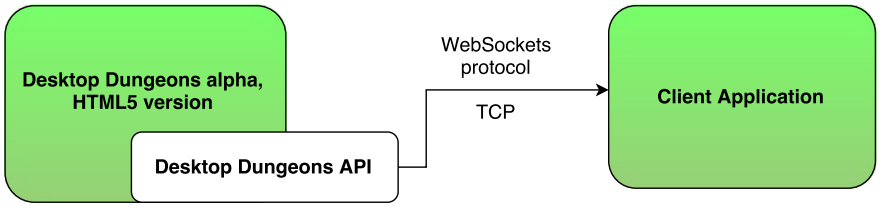

Fig. 2. The API, as a part of the game, connects to an application using a WebSockets protocol and provides access to the game by receiving and sending messages.

The framework allows the user to focus on high-level programming, and have the technical details hidden from him. It efficiently keeps track of the dungeon elements, and provides full game simulation, assisting any search techniques and heuristics that might be desired. The developed artificial players can be tested against the default PCG of the game, which has the advantage of being designed to provide challenging levels for human players, or one can generate the dungeon on his own and submit it to the game. Intermediate ways can also be employed, such as editing the dungeons generated by the game's PCG to e.g. adjust the difficulty or reduce the complexity of the game.

The framework is completely open-source and its repository can be found at `https://bitbucket.org/woitee/desktopdungeons-java-framework`.

4 Evolutionary Approach

To demonstrate the possibilities of the Desktop Dungeons API, we have implemented an evolutionary algorithm (EA) [10] to fine-tune greedy AI. A general explanation of EAs is, however, out of the scope of this paper.

4.1 Simple Greedy Algorithm

The original greedy algorithm was a simple strategy for each moment of the game. It is best described by a list of actions, ordered by priority.

1. Try picking up an item.
2. Try killing a monster (prefer strongest).
3. Explore.

The hero tries to perform the highest rated applicable action, and when none exists, the run ends. Killing the monster was attempted by just simulating attacks, fireballs and drinking potions until one of the participants died. If successful, the sequence of actions was acted out. This can be modeled as a similar list of priority actions:

1. Try casting the Fireball spell.
2. Try attacking.
3. Try drinking a potion.

Some actions have parameters, e.g. how many potions is the hero allowed to use against a certain level of monster. These were set intuitively and tuned by trial and error.

This algorithm has yielded good results. Given enough time (weeks, tens of thousands of runs), this simple AI actually managed to luck out and kill the boss. This was very surprising, we thought the game would be much harder to beat, even with chance on our side. It was probably caused by the AI always calculating how to kill every monster it sees, which is tedious and error-prone for human players to do.

4.2 Design of the Evolution

We used two ordered lists of elementary strategies in the greedy approach, but we hand-designed them and probably have not done that optimally. This would become increasingly more difficult, had we added more strategies to the list. We'll solve this by using evolutionary algorithms.

We'll call the strategies used to select actions in the game **maingame strategies** and the strategies used when trying to kill monsters **attack strategies**. Each strategy has preconditions (e.g. places to explore exists) and may have parameters. We used as many strategies as we could think of, which resulted in a total of 7 maingame strategies and 13 attack strategies.

The evolutionary algorithm was tasked with ordering both lists of strategies, and setting their parameters. It should be emphasized, that this is far from an easy task. Small imperfections in the strategy settings accumulate over the run, and thus only the very refined individuals have some chance of slaying the final boss.

However, the design makes the AI ignore some features of the game. It doesn't buy items in shops nor does it worship any gods. These mechanics are nevertheless quite advanced, and should not be needed to win the basic setting of the game. Using them can have back-biting effects if done improperly, so we just decided to ignore them to keep the complexity low.

On a side note, this design is to a certain extent similar to *linear genetic programming* [3].

4.3 Fitness Function

Several criteria could be considered when designing the fitness function. An easy solution would be to use the game's score, which is awarded after every run. However, the score takes into account some attributes that do not directly contribute towards winning the game, e.g. awarding bonuses for low completion time, or never dropping below 20% of health.

We inspired ourselves by the game's scoring, but simplified it. Our basic fitness function evaluates the game's state at the end of the run and looks like

this:

$$\begin{aligned} fitness = 10 \cdot xp &+ 150 \cdot healthpotions \\ &+ 75 \cdot manapotions \\ &+ health \end{aligned}$$

The main contributor is the total gained XP (experience points, good runs get awarded over a hundred), and additionally, we slightly reward leftover health and potions. We take these values from three runs and add them together. Three runs are too few to have low variance on subsequent evaluations, but it yields far better results than evaluating only one run, and more runs than three would just take too much time to complete.

If the AI manages to kill the boss in any of the runs, we triple the fitness value of that run. This may look a little over the top, but slaying the final monster is very difficult, and if one of the individuals is capable of doing so, we want to spread it's gene in the population. Note, that we don't expect our AI to kill the boss reliably, 5-10% chance is more what we are aiming for.

We have tried a variety of fitness functions, taking into account other properties of the game state and with different weights. For a very long time, the performance of the bots was similiar to the hand-designed greedy strategy. But, by analyzing more of the game, we have constructed roughly the fitness function above and the performance has hugely improved.

The improvement lies in the observation of how can the bots improve during the course of evolution. Strong bots in the early state will probably just use objectively good strategies, and not make complete blunders in strategy priorities, such as exploring the whole level before trying to kill anything. This should already make them capable of killing quite a few monsters. Then, the bots can improve and fine-tune their settings, to use less and less resources (mainly potions) to kill as many monsters as possible. And towards the late state of evolution, the bots can play the game so effectively, they may still have enough potions and other resources to kill the final boss and beat the game. The current fitness function supports this improvement, because the fitness values of the hypothetical bots in subsequent stages of evolution continuously rises.

After implementation, this was exactly the course the bots have evolved through. Note, that saving at least a few potions for the final boss fight is basically a necessary condition for success.

4.4 Genetic Operators

Priorities of the strategies are represented by floating point numbers in the $[0, 1]$ interval. Together with the strategy's parameter values, we can encode it as just a few floating point numbers, integers and booleans.

This representation allows us to use classical operators like one-/two-point crossovers and small change mutations. And they make good sense and work, but they are not necessarily optimal, and after some trial and error, we have

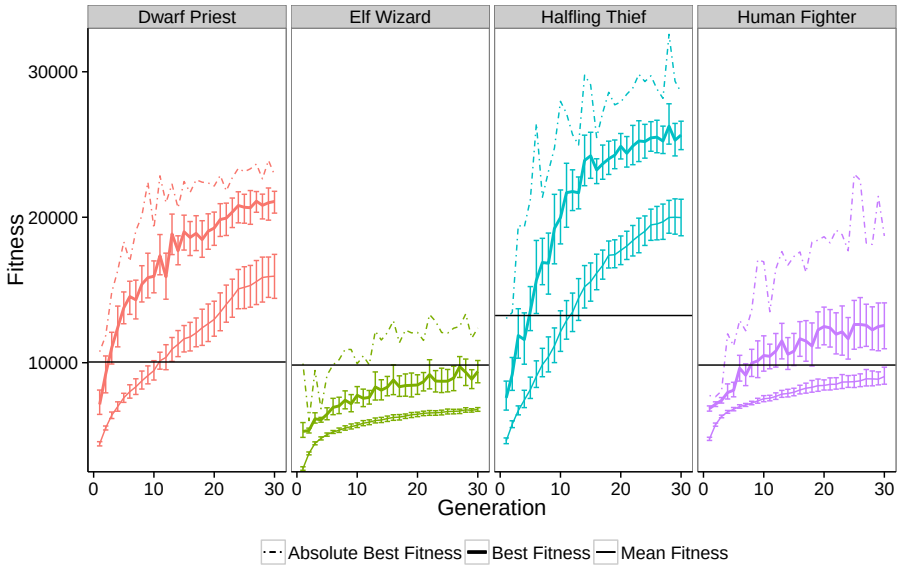

Fig. 3. Graphs describing the fitnesses of the evolution for each of our class-race settings. The three curves describe the total best fitness ever encountered, the best fitnesses averaged over all runs and the mean fitnesses averaged over all runs. The vertical line indicates the point, where the AI has killed the boss and won the game at least once in three attempts. This fitness value is different for each setting, since some race-class combinations can gain more hitpoints or health potions than other, both of which directly increase their fitness (see Section 4.3).

started using a weighted average operator to crossover the priorities for better performance.

The AI evolved with these settings were just a little too greedy, often using all their potions in the early game, and even though they advanced far, they basically had no chance of beating the final boss. These strategies found quite a strong local optimum of the fitness, and we wanted to slightly punish them for it. We did so in two ways. Firstly, we rewarded leftover potions in our fitness value calculation, and secondly, a smart mutation was added, that modifies a few individuals from the population to not use potions to kill monsters of lower level than 5. After some balancing, this has shown itself to be effective.

Mating and natural selection was done by simple roulette, i.e. individuals were chosen with probability proportional to their fitness. This creates a rather low selection pressure, and together with a large enough number of individuals in a generation, the evolution should explore a large portion of the candidate space and tune the strategies finely.

4.5 Results

After experimentation, we settled to do final runs with a population of 100 individuals, evolving through 30 generations. The population seemed large enough to be exploring the field well, and the generations sufficient for the population to converge. We ran the EA on 4 computers for a week, with a different combination of hero class and race on each computer. The result was a total of 62 runs, every hero class and race setting completed a minimum of 12 full runs. A single evaluation of an individual takes about 2 seconds, and a single whole run finishes in about 14 hours (intel i5-3470 at 3.2GHz, 4GB RAM, two instances in parallel).

The data of the results contain a lot of good strategies, their qualities can be seen in Fig. 3. Every combination of hero race and class managed to beat the boss at least once, and the strongest evolved individual kills the boss 72% of time (averaged over 10000 runs). This is definitely more than we expected. Note that no AI can slay the boss 100% of the time, since the game's default PCG sometimes creates an obviously unbeatable level (e.g. all exits from the starting room surrounded by high level monsters).

The evolved strategies also vary from each other. Different race and class combinations employ different strategies, but variance occurs even among runs of the same configuration. This shows that Desktop Dungeons can be played in several ways, and that different initial settings require different approaches to be used, which makes the game more interesting for a human. The different success rates of the configurations can also be used as a hint which race–class combinations are more difficult to play than others, either to balance them in the game design, or to recommend the easier ones to a beginner.

5 Conclusion

We present a platform for creating AI and PCG for the rogue-like game Desktop Dungeons. As a demonstration, we created an artificial player by an EA adjusting greedy algorithms. This AI functioned better than the hand-made greedy algorithm, winning the game roughly three quarters of the time, compared to a winrate of much less than 1%, and being as successful as an average human player.

This shows that the game's original PCG worked quite well, not generating a great abundance of impossible levels, yet still providing a good challenge.

A lot of research is possible with this platform. AI could be improved by using more complex EAs, or created from scratch using any techniques, such as search, planning and others. The PCG may be improved to e.g. create more various challenges for the player, adjust difficulty for stronger/weaker players or reduce the number of levels that are impossible to win. For evaluating the PCG, we could advantageously utilize the AI, and note some statistics, such as winrate, how often are different strategies employed or number of steps to solve a level. A combination of these would then create a rating function.

Also, it would be very interesting to keep improving both the artificial player and the PCG iteratively by each other.

References

1. Desktop Dungeons - DDwiki. `http://www.qcfdesign.com/wiki/DesktopDungeons`, (Accessed: 12 May 2015)
2. Tactical Amulet Extraction Bot (TAEB) - Other Bots. `http://taeb.github.io/bots.html`, (Accessed: 12 May 2015)
3. Brameier, M.F., Banzhaf, W.: Linear Genetic Programming. Springer Science & Business Media (2007)
4. Dahlskog, S., Smith, G., Togelius, J.: A Comparative Evaluation of Procedural Level Generators in the Mario AI Framework. Proceedings of Foundations of Digital Games (2014)
5. Hendrikx, M., Meijer, S., Van Der Velden, J., Iosup, A.: Procedural content generation for games: A survey. ACM Transactions on Multimedia Computing, Communications, and Applications 9(1), 1–22 (2013)
6. Karakovskiy, S., Togelius, J.: The Mario AI Benchmark and Competitions. IEEE Transactions on Computational Intelligence and AI in Games 4(1), 55–67 (2012)
7. Krajíček, J.: NetHack Bot Framework. Master's thesis, Charles University in Prague, Czech Republic (2015), (In Czech)
8. Liapis, A., Yannakakis, G.N., Togelius, J.: Towards a Generic Method of Evaluating Game Levels. In: AIIDE (2013)
9. Mauldin, M.L., Jacobson, G., Appel, A.W., Hamey, L.G.C.: ROG-O-MATIC: a belligerent expert system (1983)
10. Mitchell, M.: An Introduction to Genetic Algorithms. MIT Press (1996)
11. QCF Design: Desktop Dungeons. `http://www.desktopdungeons.net/`, (Accessed: 12 May 2015)
12. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edn. (2009)

13. Scales, D., Thompson, T.: SpelunkBots API - An AI Toolset for Spelunky. IEEE Conference on Computational Intelligence and Games pp. 1–8 (2014)
14. Shaker, N., Togelius, J., Nelson, M.J.: Procedural Content Generation in Games: A Textbook and an Overview of Current Research. Springer (2015)
15. Shaker, N., Yannakakis, G.N., Togelius, J.: Towards Automatic Personalized Content Generation for Platform Games. In: AIIDE (2010)
16. Smith, G.: The Seven Deadly Sins of PCG Research. `http://sokath.com/main/blog/2013/05/23/`, (Accessed: 12 May 2015)
17. Togelius, J., Yannakakis, G.N., Stanley, K.O., Browne, C.: Search-based procedural content generation: A taxonomy and survey. IEEE Transactions on Computational Intelligence and AI in Games 3(3), 172–186 (2011)