Pontifícia Universidade Católica
do Rio de Janeiro

Algoritmos genéticos para geração procedimental de níveis em jogos digitais

Lucas Feijó Lobo de Andrade

Projeto Final de Graduação

Centro Técnico Científico - CTC

Departamento de Informática

Curso de Graduação em Engenharia da Computação

Rio de Janeiro, novembro de 2022

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Lucas Feijó Lobo de Andrade

Algoritmos genéticos para geração procedimental de níveis em jogos digitais

Relatório Final de Projeto de Conclusão de Curso, apresentado como requisito parcial para obtenção do título de Engenheiro da Computação.

Orientador: Augusto Cesar Espíndola Baffa

Rio de Janeiro, novembro de 2022

Resumo

Feijó, Lucas. Baffa, Augusto. Algoritmos genéticos para geração procedimental de níveis em jogos digitais. Rio de Janeiro, 2022. 30p. Relatório Final de Projeto Final II - Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

Este trabalho relata o desenvolvimento de um algoritmo genético para geração procedimental de níveis em jogos digitais 2D, o desenvolvimento de um jogo simples para testar o algoritmo e a avaliação dos resultados produzidos pelo algoritmo.

Palavras-chave

Algoritmo genético, geração procedimental de níveis, geração procedimental de conteúdo

Abstract

Feijó, Lucas. Baffa, Augusto. Genetic algorithms for procedural level generation in digital games. Rio de Janeiro, 2022. 30p. Final Report for Final Project II - Department of Informatics. Pontifical Catholic University of Rio de Janeiro.

This report describes the development of a genetic algorithm for procedural level generation in 2D digital games, the development of a simple game to test the algorithm and the analysis of the results produced by the algorithm.

Keywords

Genetic algorithm, procedural level generation, procedural content generation

Sumário

Terminologia

GPN - geração procedural de níveis

PDG - *procedural dungeon generation* (geração procedural de *dungeons*)

AG - algoritmo genético

1 Introdução

A geração procedimental de níveis (GPN), isto é, a criação automática de níveis através do uso de um algoritmo, é cada vez mais utilizada em jogos. No contexto dos jogos digitais, os níveis são os espaços virtuais através dos quais o jogador deve se locomover, completando eventuais objetivos (TOGELIUS et al., 2011). São os níveis que definem os desafios encontrados pelo jogador, bem como as estratégias que podem ser usadas para enfrentá-los. Portanto, são um dos principais fatores que definem o grau de dificuldade e diversão de um jogo e, consequentemente, a qualidade da experiência do jogador.

O uso da GPN tem vantagens e desvantagens em relação à criação manual de níveis. Ela pode economizar tempo e recursos de desenvolvimento, o que é especialmente interessante para desenvolvedores individuais ou equipes pequenas, que trabalham com recursos mais limitados. Além disso, jogos com níveis criados manualmente têm, obrigatoriamente, uma quantidade finita de níveis. Já os jogos com níveis gerados procedimentalmente não têm essa limitação, podendo proporcionar ao jogador uma experiência efetivamente infinita.

A principal desvantagem da GPN é o menor controle criativo sobre os níveis, já que, por definição, o desenvolvedor do jogo não tem controle sobre a configuração final do nível, e, portanto, sobre a sua qualidade. Daí surge a necessidade de estudar maneiras de maximizar o grau de controle que se tem sobre os níveis gerados por algoritmos de GPN.

Atualmente, a GPN é utilizada frequentemente em jogos de diversos gêneros. Entre eles, destaca-se o gênero *roguelike*. Jogos deste gênero consistem, em geral, da exploração de um *dungeon* (calabouço), definido por van der Linden et al. (LINDEN; LOPES; BIDARRA, 2014) como um nível "labiríntico" onde o jogador encontra diferentes desafios e recompensas. Alguns exemplos são The Binding of Isaac (MCMILLEN; HIMSL, 2011), Crypt of the Necrodancer (BRACE YOURSELF GAMES, 2015) e Spelunky (YU; MOSSMOUTH, 2008). A GPN é adequada para esse estilo de jogo pois, cada vez que o jogador perde e tem que recomeçar, um nível novo pode ser gerado, evitando que a experiência fique repetitiva. Um dos métodos mais utilizados para a geração procedimental de *dungeons* (PDG) é o uso de algoritmos genéticos. (LINDEN; LOPES; BIDARRA, 2014) (VIANA; SANTOS, 2019)

O objetivo principal deste trabalho é implementar um algoritmo genético capaz de gerar níveis no estilo *dungeon* e avaliar o impacto que a utilização de diferentes funções de avaliação tem nos níveis gerados pelo algoritmo. Será desenvolvido, também, um jogo simples, que servirá como base para o algoritmo de geração de níveis.

2 Pesquisa Relacionada

Em um levantamento bibliográfico dos métodos utilizados para a geração procedural de *dungeons* (*procedural dungeon generation* - PDG) feita em 2013 por van der Linden et al. (LINDEN; LOPES; BIDARRA, 2014), foram identificados quatro métodos principais: autômatos celulares, gramáticas generativas, algoritmos genéticos e geração baseada em restrições. Entre esses, os mais comuns foram as gramáticas generativas e os algoritmos genéticos. Mais recentemente, em um levantamento por Viana & Dos Santos de 2019 (VIANA; SANTOS, 2019), 18 dos 26 artigos utilizaram uma abordagem evolutiva, e 14 desses utilizaram um algoritmo genético ou derivado.

2.1 Algoritmos Genéticos

2.1.1 Funcionamento

Algoritmos genéticos (AGs) são algoritmos de otimização baseados em busca cujo funcionamento é inspirado pela seleção natural. Os elementos essenciais de um AG são os **indivíduos** (ou cromossomos) e a **função de aptidão** (ou função objetivo). (LACERDA; CARVALHO, 1999)

Os indivíduos representam as possíveis soluções para o problema de otimização em questão e, geralmente, tomam a forma de uma sequência de bits. Esses bits correspondem, de forma definida pelo desenvolvedor do algoritmo, a uma solução no espaço de busca do algoritmo.

A função de aptidão toma como entrada um indivíduo e dá a ele uma nota, denominada **aptidão**, que é o valor que o algoritmo deseja maximizar. Portanto, idealmente, quanto melhor for a solução para o problema, maior será a aptidão.

Inicialmente, o algoritmo gerará uma população de indivíduos aleatórios. Então, a cada iteração do algoritmo, ocorrerá o processo de ***crossover*** e **mutação**. No *crossover*, partes de indivíduos diferentes são selecionadas para formar um indivíduo novo para a próxima geração. Os indivíduos são selecionados para sofrer o *crossover* com base na sua aptidão, de forma que indivíduos com maior aptidão são selecionados mais frequentemente. Isso faz com que a aptidão média da população tenda a subir com o tempo. Em seguida, os indivíduos novos sofrem mutações aleatórias, com o objetivo de aumentar a variedade genética na população.

Esse processo se repete até que um critério de parada seja alcançado. Esse critério pode ser, por exemplo, um certo valor de aptidão, um certo número de iterações ou um longo período sem melhora na aptidão.

2.1.2 Uso de algoritmos genéticos para PDG

Ashlock et al. (ASHLOCK; LEE; MCGUINNESS, 2011) desenvolveram um algoritmo genético para PDG e compararam os resultados produzidos utilizando diferentes representações genéticas e funções de avaliação. Os resultados demonstram que funções de avaliação e representações diferentes de indivíduos geram resultados com características significativamente diferentes.

Valtchanov & Brown (VALTCHANOV; BROWN, 2012) desenvolveram um AG para PDG utilizando uma representação baseada em árvores. Os níveis gerados pelo algoritmo correspondem à intenção dos autores quando desenvolveram a função de avaliação, o que sugere que é possível representar qualidades desejadas em um nível através de uma função bem escolhida. Além disso, os resultados eram diferentes entre si, sem tender para um único máximo global, o que é desejável no contexto de níveis para jogos.

Baldwin et al. (BALDWIN et al., 2017) desenvolveram um programa de PDG que permite que um usuário defina parâmetros de entrada para um AG. Esses parâmetros são, por exemplo, a frequência com que certos padrões aparecem no nível gerado, o nível de dificuldade e a quantidade de tesouros e inimigos. Os resultados dos experimentos realizados sugerem que, em geral, o AG é capaz de gerar níveis correspondentes aos parâmetros de entrada do usuário.

Esses resultados sugerem que a escolha da representação genética e função de avaliação é uma forma adequada de controlar o resultado de um algoritmo genético e fazer com que os níveis resultantes tenham características desejáveis. Outro ponto positivo é que a função de avaliação pode ser facilmente alterada sem qualquer modificação no resto do algoritmo, o que facilita a experimentação com diferentes funções.

Um ponto negativo é que é necessário conhecimento técnico para desenvolver uma função de avaliação, o que torna o método pouco acessível para designers de jogos que não possuam tal conhecimento. Isso pode ser remediado se for permitida a alteração de parâmetros de uma função de avaliação previamente implementada (através de uma interface gráfica, por exemplo).

3 Planejamento

3.1 Objetivo e Proposta do Trabalho

Os objetivos deste trabalho são:

- **Implementar um algoritmo genético para PDG.** Este algoritmo deverá ser capaz de gerar níveis desejáveis no estilo *dungeon*. Níveis desejáveis são, por exemplo, viáveis (é possível partir do início e chegar ao fim), divertidos e diferentes uns dos outros.
- **Analisar o impacto da utilização de diferentes funções de avaliação nos níveis gerados pelo algoritmo.** Para analisar o grau de controle que se pode ter sobre os níveis gerados, serão desenvolvidas diferentes funções de aptidão para o AG. Essas funções levarão em conta diferentes aspectos dos níveis gerados pelo algoritmo, com o objetivo de realçar certas características em tais níveis. Então, serão analisados os níveis resultantes de cada função, para determinar se funções diferentes geram níveis diferentes umas das outras, e se as características observadas correspondem com a intenção por trás de cada função.
- **Desenvolver um jogo simples para o qual o algoritmo gerará níveis.** Visto que a finalidade de um algoritmo de PDG é gerar níveis para um jogo, será desenvolvido um jogo simples no estilo *roguelike*, através do qual será possível testar os níveis gerados pelo algoritmo.

3.2 Planejamento do jogo

3.2.1 Tecnologias

Para o desenvolvimento do jogo, foi escolhida a *engine* MonoGame (MONOGAME,). O MonoGame é uma implementação do Microsoft XNA, uma *framework* para o desenvolvimento de jogos baseada no Microsoft .NET (MICROSOFT,). A linguagem de programação utilizada é o C#.

3.2.2 Formato do jogo

Foi definido que o jogo seria do gênero *roguelike* (descrito na seção 1). Ou seja, o objetivo do jogo é chegar ao final de cada nível (ou *dungeon*), derrotando os inimigos encontrados pelo caminho. Os níveis são gerados por um algoritmo genético, e é gerado um novo nível cada vez que o jogador vencer um nível.

3.3 Planejamento do algoritmo

3.3.1 Estudo de implementações existentes

Para entender as abordagens mais comuns em algoritmos genéticos para PDG, foram estudadas algumas das implementações mencionadas pelos levantamentos citados no capítulo 2.

Em sua implementação, Ashlock et. al (ASHLOCK; LEE; MCGUINNESS, 2011) utilizaram diversas representações. Na mais simples delas, a representação direta, o nível é representado por uma matriz onde cada entrada corresponde a um quadrado aberto ou fechado, e os cromossomos codificam as entradas da matriz diretamente.

Na representação que chamaram de indireta positiva, o nível ainda é representado pela matriz descrita acima, mas os cromossomos codificam posições de barreiras lineares a serem adicionadas a um nível vazio. De forma similar, na representação indireta negativa, os cromossomos codificam áreas retangulares a serem removidas de um nível inicialmente preenchido.

Por último, a representação chamada cromática representa o nível como uma matriz de inteiros entre 0 e 5, criando barreiras entre quadrados adjacentes se a diferença entre seus valores é maior que 1.

Valtchanov & Brown (VALTCHANOV; BROWN, 2012) também utilizaram um nível representado por uma matriz de quadrados abertos ou fechados, mas os cromossomos codificam uma árvore. Nessa árvore, cada nó representa um dos diferentes modelos de sala criados pelos autores e seus filhos representam as salas conectadas. A árvore é, então, traduzida para um espaço bidimensional de acordo com regras que ditam o posicionamento das salas.

Baldwin et al. (BALDWIN et al., 2017) também representam seus níveis como uma matriz de quadrados, mas estes podem representar tesouros, inimigos, entradas e portas, além de quadrados abertos e fechados. A representação dos indivíduos tem o formato de uma matriz de inteiros indicando o conteúdo de cada quadrado.

3.3.2 Escolha da representação

É necessário escolher um formato para os níveis finais gerados pelo algoritmo e utilizados no jogo, bem como uma forma de codificar este formato em cromossomos para o algoritmo genético.

Os níveis finais utilizados no jogo têm a forma de uma matriz de quadrados, cada um podendo conter, a princípio, uma parede ou um espaço vazio. Essa forma foi escolhida por ser fácil de representar na memória (e, consequentemente, fácil de ser manipulada no desenvolvimento de funções de aptidão) e versátil (não está limitada a salas pré-definidas conectadas por portas, por exemplo). Além disso, como visto acima, essa representação foi utilizada com êxito em trabalhos anteriores.

Para definir o formato dos indivíduos (ou cromossomos), foram implementados e testados dois tipos, baseados nos utilizados por Ashlock et al. (ASHLOCK; LEE; MCGUINNESS, 2011): a representação direta e a indireta negativa. Os resultados dos testes (exibidos na seção 5.2) mostraram resultados mais interessantes com a representação direta, e, por isso, esta foi escolhida.

4 Especificação do Projeto

Foi desenvolvido um jogo 2D com um sistema de geração procedimental de níveis baseada em algoritmos genéticos.

4.1 Jogo

O jogo é do gênero *roguelike*, ou seja, consiste da exploração de níveis com o objetivo de chegar ao final de um nível e prosseguir para o próximo.

Inicialmente, o jogador se encontra em um nível gerado procedimentalmente. O jogador não consegue ver o nível inteiro e, portanto, não sabe onde fica o final do nível. Seu objetivo é navegar o nível para encontrar o final, evitando ou derrotando os inimigos que encontrar no caminho. Ao chegar no final de um nível, um nível novo é gerado.

O jogador começa com 100 pontos de vida, representados pela barra de vida no canto inferior esquerdo. Quando sofre um ataque de um inimigo, perde uma parte desses pontos de vida. Se a vida do jogador chega a 0, o jogador perde e tem a opção de recomeçar o jogo.

Capturas de tela do jogo podem ser vistas na figura 1.

4.2 Estrutura do código

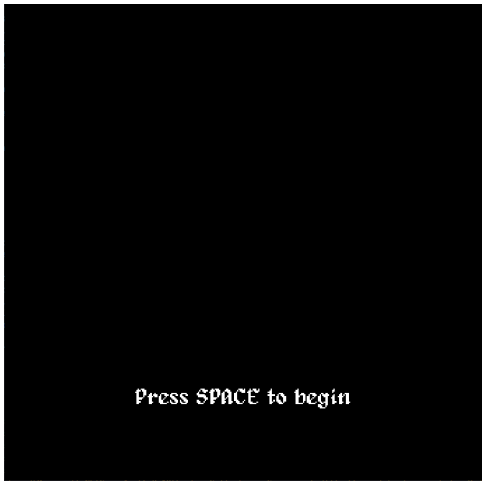
A figura 2 mostra o diagrama de classes do projeto.

O jogo contém um `SceneManager`, responsável por gerenciar as cenas do jogo. Uma cena é um estado do jogo, com suas próprias variáveis e objetos. Existem duas cenas: a tela inicial ou menu (`MenuScene`) e o jogo em si (`GameScene`).

A `MenuScene` simplesmente espera o jogador apertar uma tecla para entrar na cena do jogo. A `GameScene` contém a lógica do jogo, de fato.

A `GameScene` contém um objeto para representar o mundo (`World`). Este objeto guarda todas as informações sobre o nível, como a posição das paredes e as texturas apropriadas para cada bloco de parede, dependendo dos blocos adjacentes. Este objeto também contém um objeto que representa o algoritmo genético em si (`GeneticAlgorithm`).

A classe `GeneticAlgorithm` é responsável por executar o algoritmo genético e gerar os níveis. Ela contém os parâmetros do algoritmo, como tamanho da população e taxa de mutação, além de conter a população em si. A população é composta por indivíduos, representados pela classe abstrata `Individual`.



(a) Tela inicial

(b) Tela de jogo

(c) Tela de derrota

Figura 1 – Capturas de tela do jogo

Cada objeto do tipo `Individual` guarda uma matriz contendo o nível representado por ele e a aptidão atribuída a ele pela função de aptidão do algoritmo genético. A classe `Individual` implementa as diferentes funções de aptidão descritas em 4.5.2.1. As subclasses de `Individual` correspondem às diferentes representações genéticas descritas em 4.5.1.1 e implementam métodos de inicialização, *crossover*, mutação e geração de nível específicas às suas representações genéticas.

A `GameScene` também contém todos os elementos necessários para o funcionamento do jogo em si. Estes elementos são objetos do tipo `Camera`, `GameHUD`, `Player` e `Enemy`.

A classe `Camera` é responsável por guardar informações sobre a posição da "câmera" no mundo do jogo e seu nível de *zoom*. A posição da câmera corresponde ao centro da área visível do nível, e o *zoom* controla o tamanho desta área.

A classe `GameHUD` é responsável por exibir os elementos da interface gráfica: a barra de vida do jogador e a tela indicando que o jogador perdeu.

A classe `Player` representa o jogador e guarda informações como sua posição, velocidade e o retângulo que abrange o jogador, utilizado para detectar colisões com o mundo. Além disso, contém a lógica de controle e movimentação do jogador.

A classe abstrata `Enemy` representa os inimigos. Esta classe foi implementada como uma classe abstrata para permitir a existência de diferentes tipos de inimigos com comportamentos diferentes, mas apenas um tipo foi implementado: `Monster`.

A classe `Monster` contém informações sobre os monstros, além de implementar a IA que define seu comportamento, descrito em 4.6.1.

Toda a lógica do jogo acontece dentro das funções `Initialize`, `LoadContent`, `Update` e `Draw`. Estas funções são implementadas por todas as classes, exceto as referentes à lógica do algoritmo genético. A função `Initialize` realiza as operações que precisam ser executadas uma vez quando o objeto é instanciado. A função `LoadContent` carrega as texturas usadas pela classe correspondente. A função `Update` contém a lógica da classe correspondente e é executada continuamente, assim como a função `Draw`, que exibe o conteúdo gráfico referente à classe correspondente.

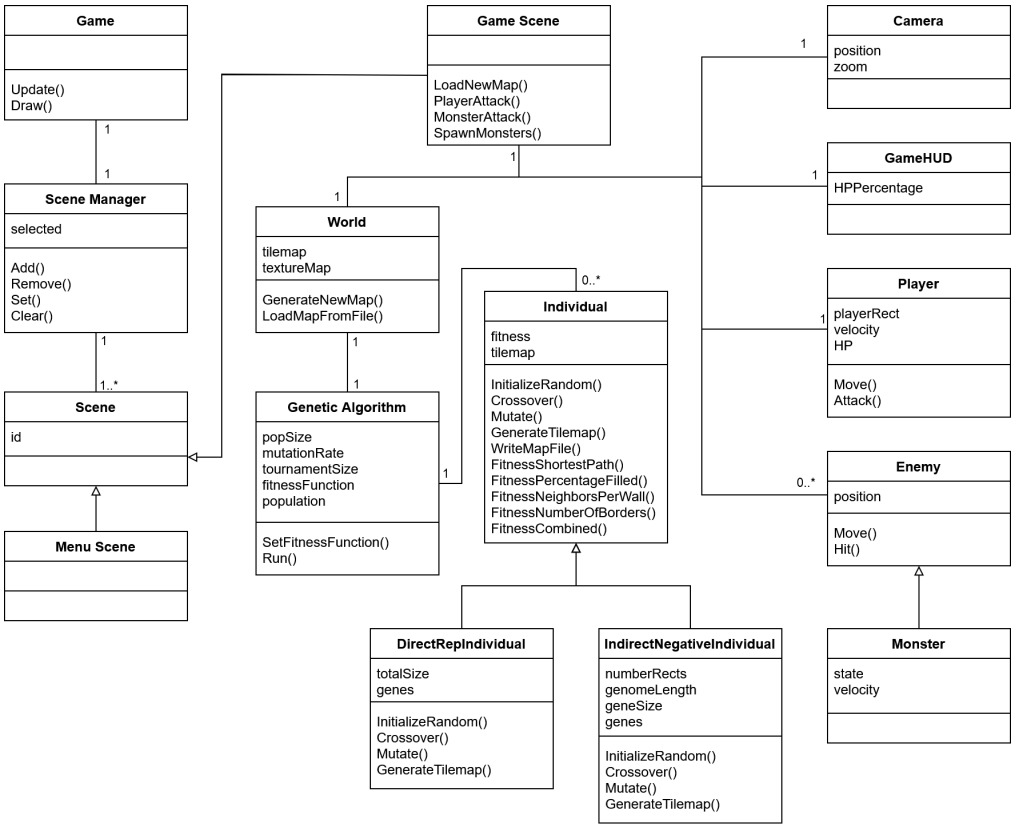


Figura 2 – Diagrama de classes

4.3 Níveis

Os níveis são formados por uma matriz de "blocos" quadrados, que podem representar um espaço vazio ou uma parede. Cada nível tem um bloco verde que representa o começo do nível e um bloco amarelo que representa o fim. Os níveis também contém inimigos que atacam o jogador. Um exemplo de nível pode ser visto na figura 3.

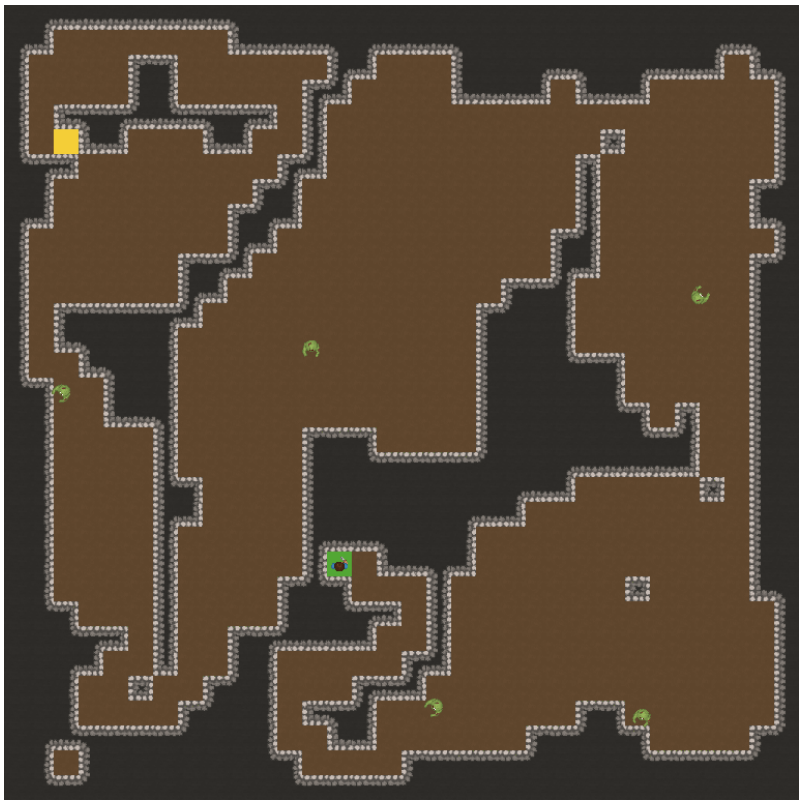


Figura 3 – Exemplo de nível. As áreas marrons representam espaços vazios onde o jogador e os inimigos podem circular. As áreas cinzas representam paredes que bloqueiam o movimento do jogador. Os quadrados verde e amarelo representam o início e fim do nível, respectivamente. O jogador encontra-se no início do nível e os inimigos encontram-se espalhados pelo nível.

4.4 Geração procedimental de níveis

Os níveis são gerados por um algoritmo genético. O *output* do algoritmo genético é uma matriz bidimensional, onde cada entrada é 0 ou 1, representando blocos vazios e paredes, respectivamente.

4.5 Funcionamento do algoritmo genético

4.5.1 Indivíduo

A base do algoritmo genético é o indivíduo, que representa um nível no processo evolutivo do algoritmo. Neste caso, cada indivíduo contém a representação genética de um nível do jogo, que é usada para gerar o *tilemap* no formato descrito acima.

Foram implementados diferentes tipos de indivíduos, cada um com um formato de representação genética diferente. Para cada tipo de indivíduo, foram implementadas, entre outras, funções de *crossover*, mutação e geração do *tilemap*.

4.5.1.1 Representações genéticas

A representação genética é a forma como os níveis são codificados durante o processo de geração. Foram implementados dois tipos de representação genética, baseados na implementação de Ashlock et al (ASHLOCK; LEE; MCGUINNESS, 2011).

4.5.1.1.1 Representação Direta

Na representação direta, o genoma de cada indivíduo é uma sequência de 0s e 1s, onde cada entrada representa diretamente um bloco no nível final.

Crossover

Nessa representação, o *crossover* entre dois indivíduos é realizado, simplesmente, combinando aleatoriamente os genomas dos dois indivíduos: para cada entrada do genoma, é copiada a entrada correspondente de um dos indivíduos, selecionado aleatoriamente.

Mutação

A operação de mutação também é simples: uma mutação consiste em inverter o valor de uma entrada aleatória do genoma.

4.5.1.1.2 Representação Indireta

Nessa representação, em vez de codificar diretamente os blocos, o genoma codifica regiões a serem removidas de um nível inicialmente completamente preenchido. O genoma consiste em uma sequência de inteiros de tamanho fixo, onde cada uma dessas regiões é representada por cinco números: as coordenadas x e y de um dos vértices de um retângulo, as coordenadas x e y do vértice oposto do retângulo, e um último inteiro que indica se a região está ativada ou não.

Para converter o genoma em um nível, inicializa-se um nível completamente preenchido. Então, para cada uma das regiões codificadas no genoma, se a *flag* que indica a ativação da região for igual a 1, todos os blocos dentro do retângulo representado pelos quatro números anteriores são transformados em espaços vazios.

Crossover

A operação de *crossover* na representação indireta é realizada a nível das regiões codificadas. Ou seja, para cada conjunto de cinco inteiros representando uma região, um dos dois indivíduos é selecionado aleatoriamente e os cinco inteiros representando a região são copiados para o indivíduo descendente.

Mutação

A operação de mutação da representação indireta escolhe, aleatoriamente, um dos inteiros da sequência genética. Se o inteiro selecionado corresponde à *flag* de ativação de uma das regiões, ele é invertido (se era 0, vira 1 e vice versa). Caso contrário, o inteiro representa uma coordenada x ou y de um dos vértices do retângulo. Nesse caso, é soma-se ou subtrai-se 1 da coordenada. Se isso resultaria em uma coordenada inválida (fora dos limites do nível), a mutação não é realizada.

4.5.2 Algoritmo genético

O funcionamento geral de um algoritmo genético foi explicado na seção 2.1.1. O algoritmo usado para gerar os níveis funciona da seguinte maneira:

N indivíduos (descritos na seção 4.5.1) são inicializados aleatoriamente. Esses N indivíduos são avaliados pela função de aptidão. Em seguida, ocorre a geração de indivíduos para a próxima geração através de uma seleção por torneio. São selecionados t indivíduos aleatoriamente e, entre estes, são escolhidos os 2 indivíduos com maior aptidão. Então, é realizado o *crossover* entre esses 2 indivíduos duas vezes, gerando dois novos indivíduos para a geração seguinte. Esse processo de seleção se repete até que a população nova tenha o tamanho da população anterior. Por último, são realizadas, no total, M mutações, cada uma em um indivíduo da nova geração selecionado aleatoriamente. Esse processo de avaliação, seleção, *crossover* e mutação se repete g vezes, e, por fim, o algoritmo retorna o indivíduo com maior aptidão encontrado ao longo de todo o processo de evolução. O pseudocódigo 1 contém um resumo deste processo.

4.5.2.1 Funções de aptidão

Foram implementadas diversas funções de aptidão diferentes, além de uma função que combina os valores de mais de uma função, com o objetivo de controlar diferentes aspectos dos níveis gerados. A função escolhida para o jogo é uma combinação das funções descritas abaixo (seção 5.2.2).

Algorithm 1 Algoritmo Genético

Inicializa N indivíduos aleatoriamente

Repita g vezes:

Para cada indivíduo I:

Avalia $F(I)$

Repita N vezes:

Seleciona t indivíduos aleatoriamente

Entre esses, seleciona $I1$ e $I2$, os dois indivíduos com maior aptidão

Aplica o operador de crossover a $I1$ e $I2$, adicionando o resultado à geração seguinte

Repita M vezes:

Aplica uma mutação a um indivíduo aleatório da nova geração

Retorna o indivíduo com maior aptidão, entre todos os indivíduos de todas as gerações

4.5.2.1.1 Porcentagem do nível preenchida

O objetivo desta função é controlar a proporção de blocos vazios para blocos preenchidos no nível, evitando níveis com áreas muito abertas ou muito fechadas. Essa função calcula a fração de blocos totais do nível que estão preenchidos e retorna o seguinte valor:

$$F_{\text{preench.}} = (1 - |\, p - P \,|)^2$$

Onde p é a razão $\frac{\text{blocos preenchidos}}{\text{número total de blocos}}$, e P é o valor desejado para esta razão. Ou seja, esta função tem formato parabólico e seu valor é máximo quando a proporção de blocos preenchidos é exatamente igual à desejada.

4.5.2.1.2 Número de bordas

Essa função calcula a quantidade total de bordas entre blocos vazios e paredes e retorna:

$$F_{\text{bordas}} = 1 - \frac{b}{B}$$

Onde b é a quantidade de bordas no nível, e B é o maior número de bordas possível em um nível. Ou seja, esta função tem valor maior quando há menor quantidade de blocos vizinhos diferentes uns dos outros. O objetivo desta função é minimizar a quantidade de blocos "avulsos"e recompensar níveis com paredes mais lisas.

4.5.2.1.3 Número médio de vizinhos por parede

Essa função simplesmente retorna o número médio de blocos de parede adjacentes a cada parede. Ela tem objetivo semelhante à função de número de bordas, recompensando níveis com grupos coesos de paredes e penalizando níveis com blocos de parede avulsos.

4.5.2.1.4 Comprimento do caminho entre início e fim

Essa função implementa o algoritmo A* para calcular o comprimento em número de blocos do caminho mais curto entre o início e o fim do nível, e retorna:

$$F_{\text{cam.curto}} = \begin{cases} \frac{c}{N}, & \text{se há caminho entre início e fim} \\ 0, & \text{caso contrário} \end{cases}$$

Onde c é o comprimento do caminho, e N é o tamanho total de um nível, ambos medidos em número de blocos. O objetivo desta função é punir níveis onde não há caminho entre o início e o fim e recompensar níveis onde o caminho a ser percorrido pelo jogador para chegar no final do nível não é trivial, aumentando a dificuldade dos níveis.

4.6 Inimigos

Os inimigos estão distribuídos pelos níveis e seu objetivo é atacar o jogador, impedindo que ele complete o nível.

4.6.1 Comportamento dos inimigos

Os inimigos são controlados por uma inteligência artificial básica, implementada como uma máquina de estados. A máquina conta com 5 estados: Patrulha, Espera, Perseguição, Ataque e *Cooldown*, que funcionam da seguinte maneira:

- **Patrulha** - Quando o inimigo entra no estado de patrulha, ele escolhe, aleatoriamente, uma direção para andar. Ele anda nesta direção por um tempo determinado, podendo ser interrompido caso aviste o jogador. Ao encerrar o tempo de patrulha, o inimigo muda para o estado Espera.
- **Espera** - Após finalizar uma patrulha, o inimigo espera um tempo determinado e, depois, volta para o estado de Patrulha. Neste estado, também pode ser interrompido se avistar o jogador.
- **Perseguição** - Quando o inimigo avista o jogador e a distância entre eles é maior que uma distância pré-determinada, ele entra no estado de Perseguição. Neste estado, ele anda em linha reta até o jogador até que esteja próximo o suficiente dele, e então entra no estado Ataque.

- **Ataque** - Quando o inimigo entra neste estado, ele simplesmente ataca o jogador e entra no estado de *Cooldown*.
- **Cooldown** - Após atacar, o inimigo entra no estado de *cooldown*. Neste estado, ele simplesmente espera por um tempo determinado e, depois, entra no estado Ataque, se ainda estiver próximo do jogador, ou no estado Perseguição, caso contrário.

A figura 4 ilustra o funcionamento da máquina de estados.

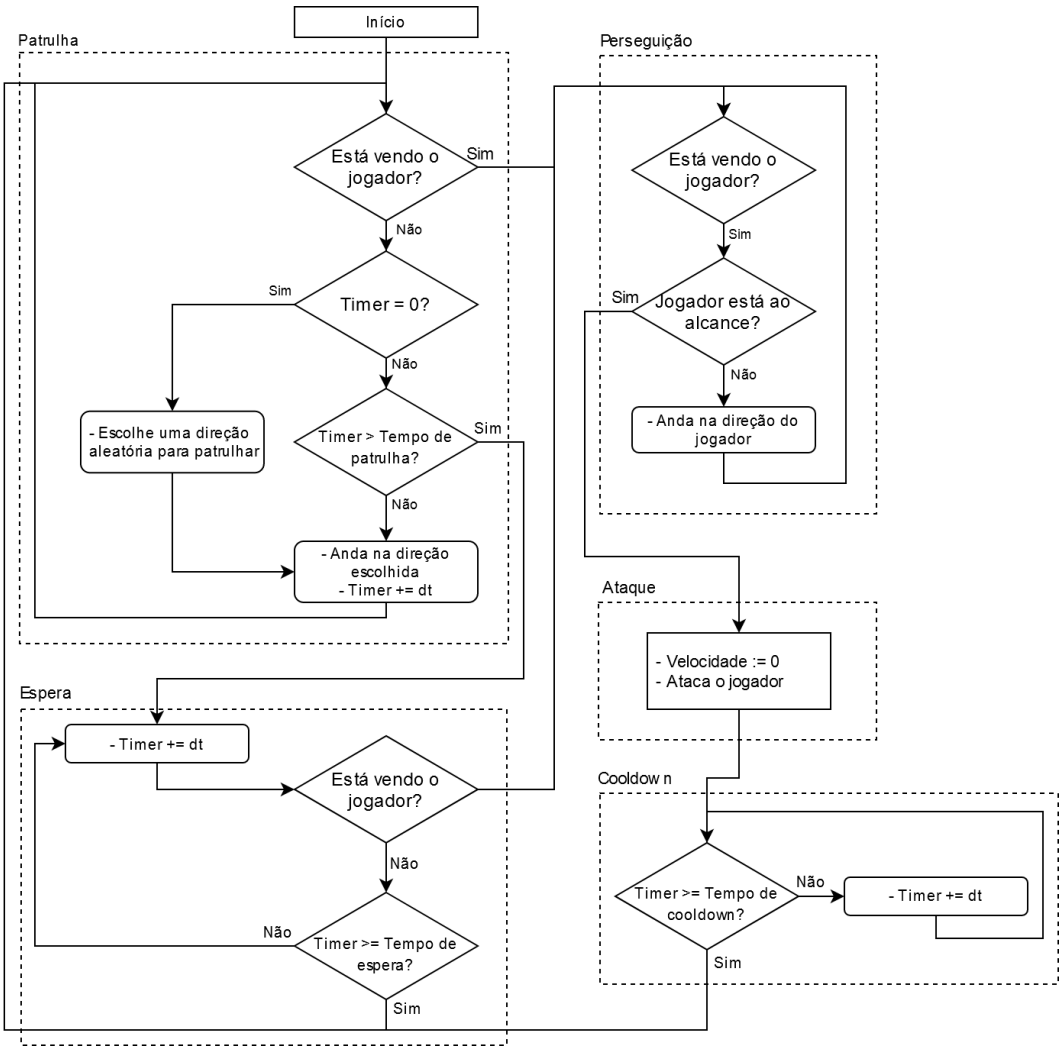


Figura 4 – Máquina de estados que dita o comportamento dos monstros

4.7 Interação com o jogo

O jogador controla o personagem utilizando as setas do teclado e ataca os inimigos utilizando a barra de espaço. Ao realizar um ataque, os inimigos que estiverem próximos o suficiente do personagem na direção para a qual este está olhando serão atingidos pelo ataque.

5 Implementação e Avaliação

5.1 Protótipo do Jogo

Primeiramente, com o objetivo de estudar o MonoGame, foi desenvolvido um protótipo do jogo, que pode ser visto na figura 5. O protótipo tinha as seguintes funcionalidades:

- Leitura e conversão de um arquivo de texto em um nível representado por uma matriz

  O nível exibido no jogo é representado por uma matriz, onde cada entrada é um inteiro que representa um tipo de "bloco" (neste caso, chão ou parede). O nível é carregado de um arquivo de texto onde cada caractere representa uma entrada da matriz.

- Movimentação do personagem baseada no input do jogador

  O jogador pode usar as setas do teclado para movimentar o personagem no jogo.

- Detecção de colisão

  Ao movimentar o personagem, o jogo verifica se há interseção entre o personagem e um bloco de parede. Se for o caso, o jogo impede o movimento na direção correspondente à colisão.

5.2 Testes do algoritmo genético

5.2.1 Testes iniciais

Para testar a implementação do algoritmo genético, foram realizados testes básicos com o objetivo de verificar se os níveis gerados pelo algoritmo correspondiam ao resultado esperado para a função de aptidão escolhida.

Foi utilizada uma função de aptidão F simples: um nível pré-definido A é lido de um arquivo de texto e a função $F(I)$ simplesmente retorna a fração de blocos de I que são iguais ao bloco correspondente em A. Ou seja, quanto maior a semelhança entre I e A, maior o valor de F. Assim, espera-se que o algoritmo gere níveis muito semelhantes a A.

O resultado dos testes para dois níveis diferentes, na figura 6, indicou que o algoritmo funciona como esperado.

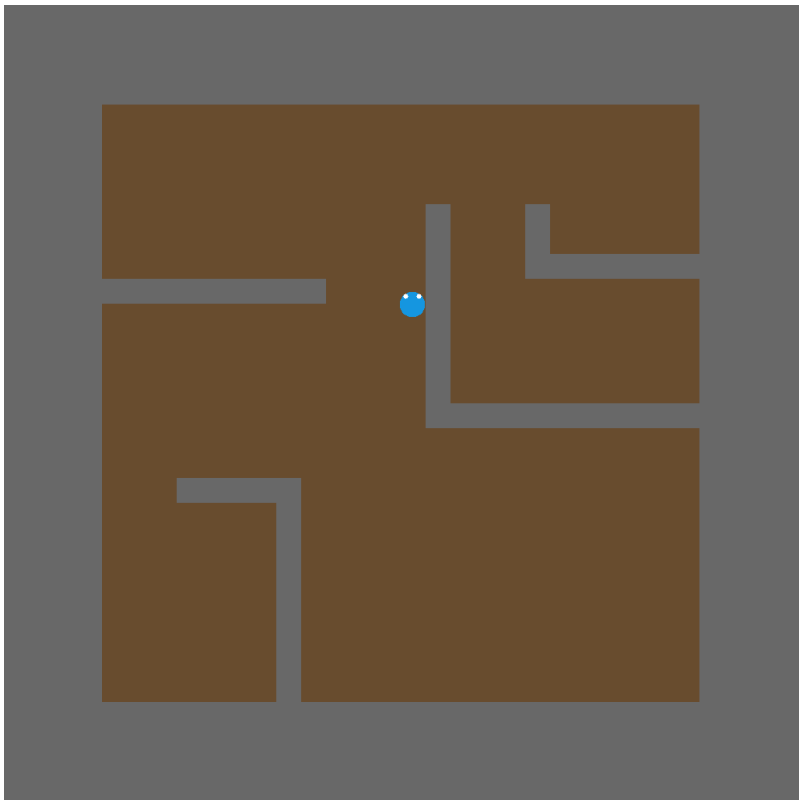


Figura 5 – Protótipo do jogo. O nível exibido nesta imagem tem 32×32 blocos. Os blocos marrons representam o chão, e os cinzas representam a parede. O personagem é representado pelo círculo azul.

5.2.2 Funções de aptidão

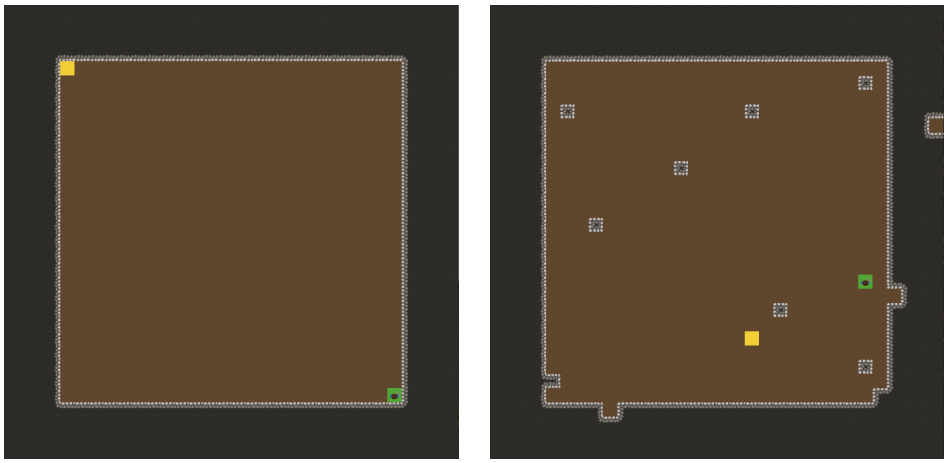
Foram realizados testes com as funções de aptidão descritas em 4.5.2.1 para verificar se os níveis gerados a partir delas possuíam as características esperadas. Os níveis gerados nos testes podem ser vistos nas figuras 7 e 8 e os gráficos da evolução da aptidão das populações pode ser visto nas figuras 9 e 10.

Os parâmetros usados nestes experimentos foram:

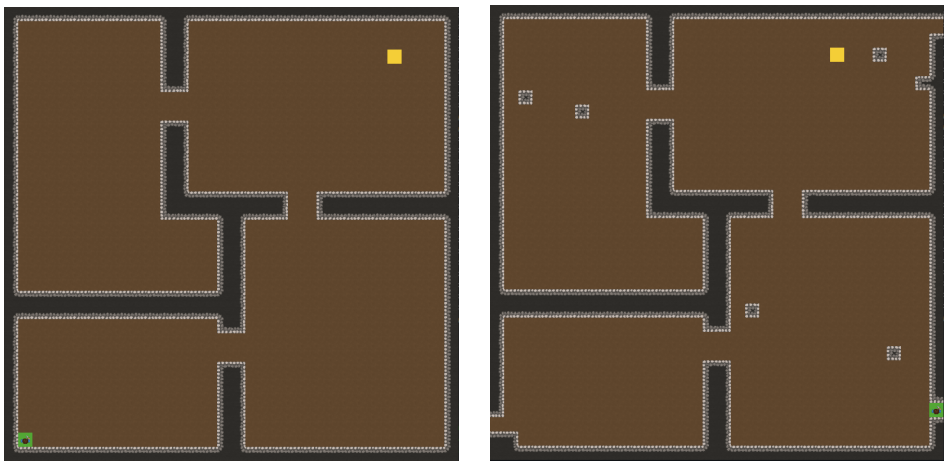
- Tamanho da população: 400
- Número de gerações: 400
- Taxa de mutação: 5 / indivíduo
- Tamanho dos torneios: 16

Porcentagem preenchida

Para a função de porcentagem preenchida (seção 4.5.2.1.1), foi utilizado um objetivo de preenchimento de 50%. O resultado esperado com essa função de aptidão, portanto, é um nível onde aproximadamente metade dos blocos estão preenchidos. Os gráficos correspondentes a essa função, 9a e 10a, indicam que o algoritmo alcançou



(a) nível 1 (b) resultado do AG



(a) nível 2 (b) resultado do AG

Figura 6 – Testes do algoritmo genético. Na coluna da esquerda, estão os níveis definidos através de um arquivo de texto. Na coluna da direita, estão os resultados do algoritmo genético usando a função de aptidão que mede a similaridade aos níveis da esquerda.

aptidão máxima rapidamente. Os níveis gerados com essa função, nas figuras 7a e 8a, têm aproximadamente metade dos blocos preenchidos, como esperado.

Número de bordas

A função de número de bordas tem valor máximo quando o número de bordas entre blocos diferentes é mínimo. Portanto, o resultado esperado utilizando esta função é um nível onde os blocos são, em geral, iguais a seus vizinhos. Níveis com aptidão alta com esta função seriam, por exemplo, níveis completamente preenchidos ou completamente vazios. Como o algoritmo força a existência de uma borda de blocos preenchidos em torno do nível, o nível que maximiza esta função é um nível completamente preenchido.

Com a representação indireta, o algoritmo encontrou a solução ótima (figura 8b). Já com a representação direta, o algoritmo chegou, aproximadamente, em um

máximo local, encontrando um nível quase completamente vazio (figura 7b), exceto pelas bordas obrigatórias e alguns blocos preenchidos.

Vizinhos por parede

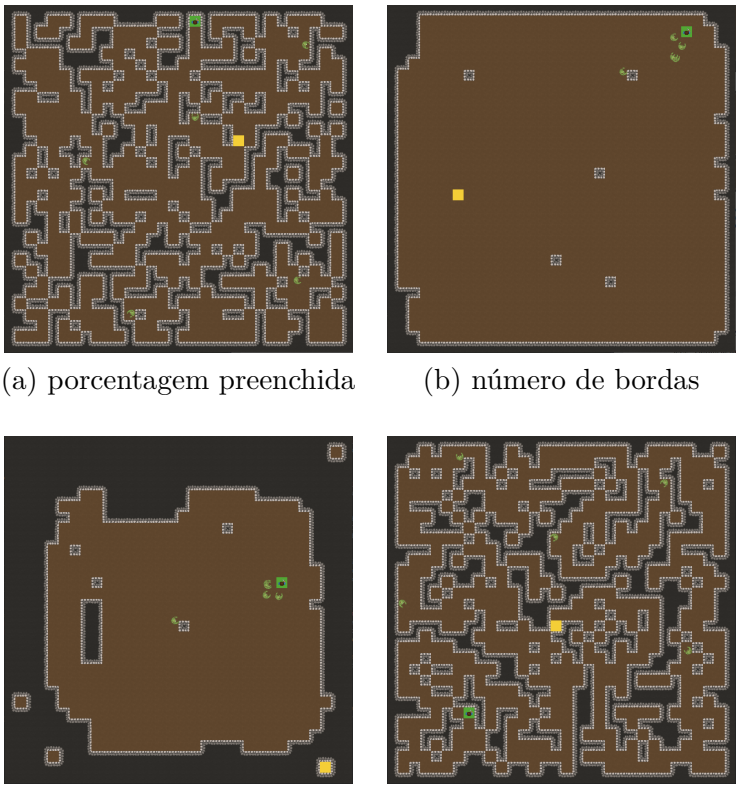
A função de vizinhos por parede é máxima para níveis onde toda parede tem quatro vizinhos, o que acontece quando o nível está completamente preenchido. O gráfico na figura 10c indica que, utilizando a representação indireta, o algoritmo encontrou a solução ótima (figura 8c) rapidamente.

Já a figura 7c mostra que, com a representação direta, o algoritmo não encontrou a solução ótima. Analisando o gráfico correspondente (9c), percebe-se que a aptidão máxima da população não chegou a estabilizar e continuava crescendo quando o teste chegou ao fim. Isso sugere que, dado tempo suficiente, o algoritmo também encontraria a solução ótima neste caso.

Caminho mais curto

Esta é a função mais complexa entre as quatro. Um nível tem aptidão alta segundo esta função quando o menor caminho entre o início e o fim é mais comprido. A figura 8d mostra o resultado do algoritmo utilizando esta função e a representação indireta. Neste caso, o início e o fim do nível estavam relativamente próximos. Ainda assim, o algoritmo encontrou uma forma de maximizar o caminho entre eles.

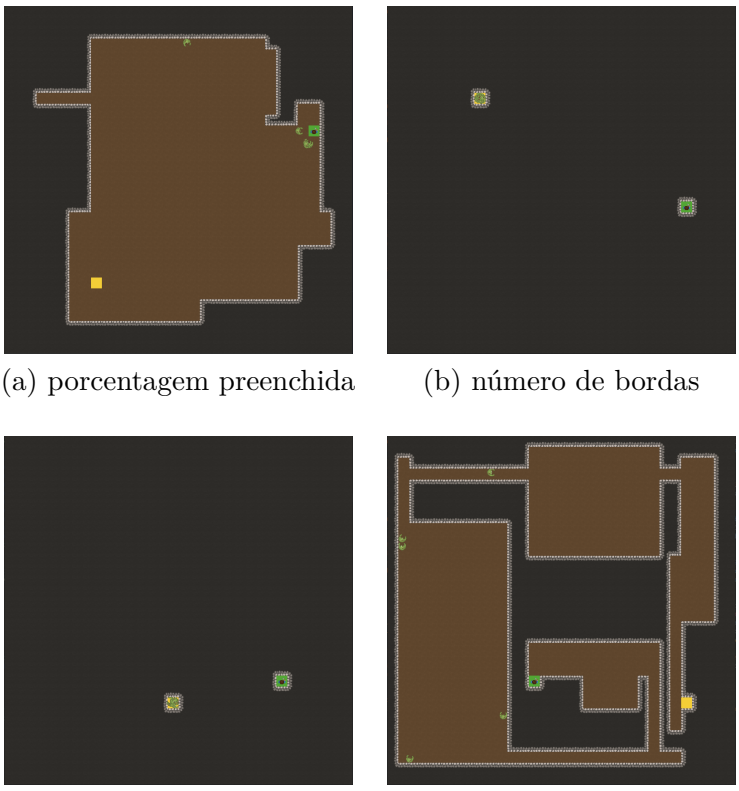
Com a representação direta, o algoritmo alcançou uma aptidão maior ainda, aproximadamente duas vezes maior que com a representação indireta (gráficos 9d e 10d). Observa-se que o nível gerado com essa representação (figura 7d) aproveitou quase todo o espaço do nível para formar o caminho, o que resultou em um caminho estreito e tortuoso.



(a) porcentagem preenchida (b) número de bordas

(c) vizinhos por parede (d) caminho mais curto

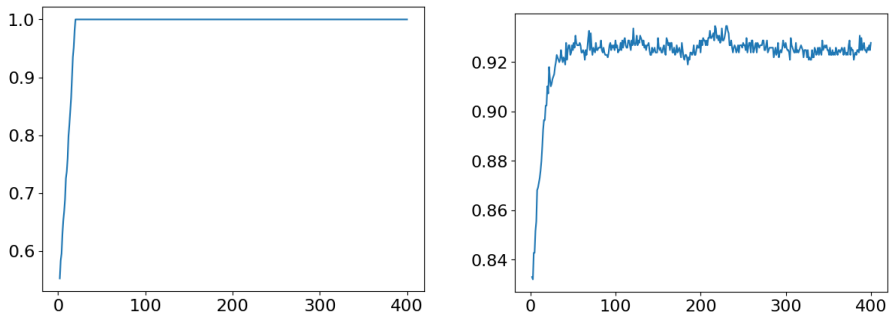
Figura 7 – Testes das funções de aptidão utilizando a representação direta



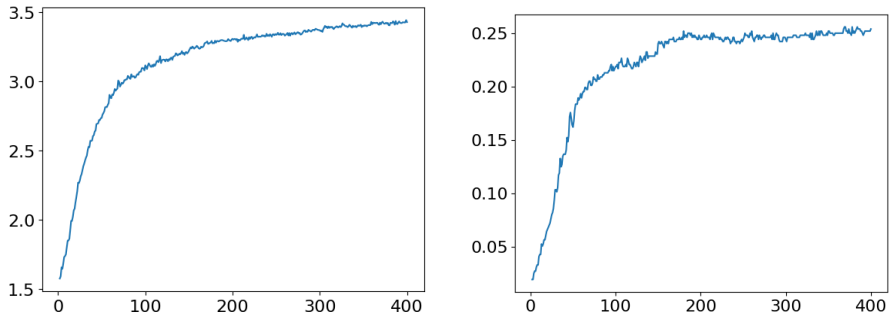
(a) porcentagem preenchida (b) número de bordas

(c) vizinhos por parede (d) caminho mais curto

Figura 8 – Testes das funções de aptidão utilizando a representação indireta

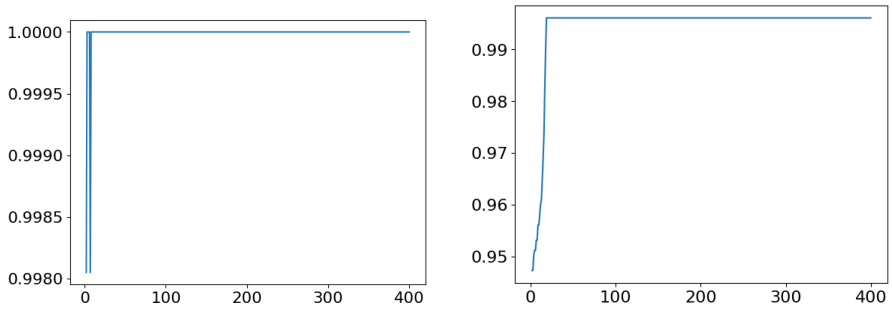


(a) porcentagem preenchida (b) número de bordas

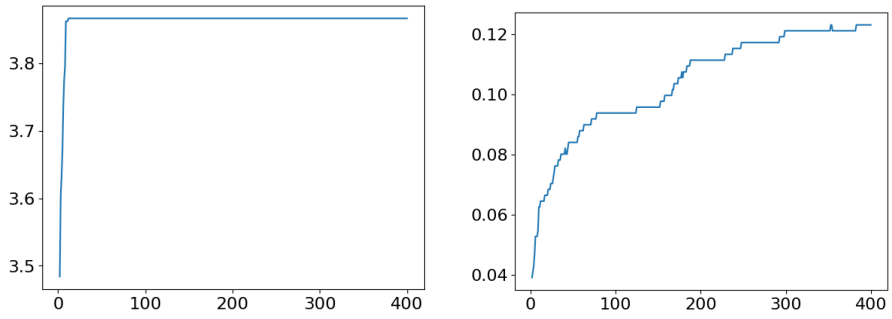


(c) vizinhos por parede (d) caminho mais curto

Figura 9 – Valor da função de aptidão do melhor indivíduo (eixo Y) de cada geração (eixo X) nos testes das funções de aptidão utilizando a representação direta.



(a) porcentagem preenchida (b) número de bordas



(c) vizinhos por parede (d) caminho mais curto

Figura 10 – Valor da função de aptidão do melhor indivíduo (eixo Y) de cada geração (eixo X) nos testes das funções de aptidão utilizando a representação indireta.

5.2.3 Combinações de funções de aptidão

As funções de porcentagem preenchida, número de bordas e vizinhos por parede, por si só, não geram níveis interessantes e não introduzem punições para níveis impossíveis. Já a função de caminho mais curto gera níveis possíveis de serem completados, mas não oferece qualquer controle sobre o resultado final do algoritmo.

É através da combinação destas funções que se pode ter controle sobre as características dos níveis gerados pelo algoritmo. A definição das características desejáveis e indesejáveis é inteiramente subjetiva, por isso a escolha da maneira como estas funções são combinadas deve ser feita através da experimentação de acordo com o julgamento do desenvolvedor.

Na figura 11, pode-se observar os níveis resultantes da execução do algoritmo utilizando a combinação das diferentes funções de aptidão. As funções utilizadas são as seguintes:

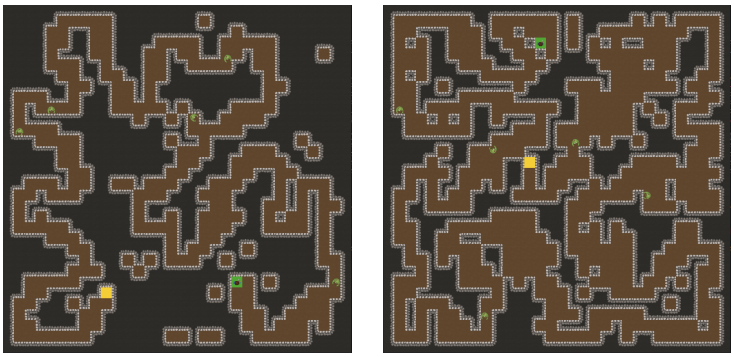
- $F_1 = F_c \cdot F_v$
- $F_2 = F_c \cdot F_b$
- $F_3 = F_c \cdot F_{p,\,0.25}$
- $F_4 = F_c \cdot {F_v}^2 \cdot F_{p,\,0.35}$

Onde F_c é a função de caminho mais curto, F_v é a função de vizinhos por parede, F_b é a função de número de bordas e $F_{p,\,x}$ é a função de preenchimento com razão de preenchimento igual a x. Os testes foram feitos com os mesmos parâmetros que os testes acima e utilizaram a representação direta.

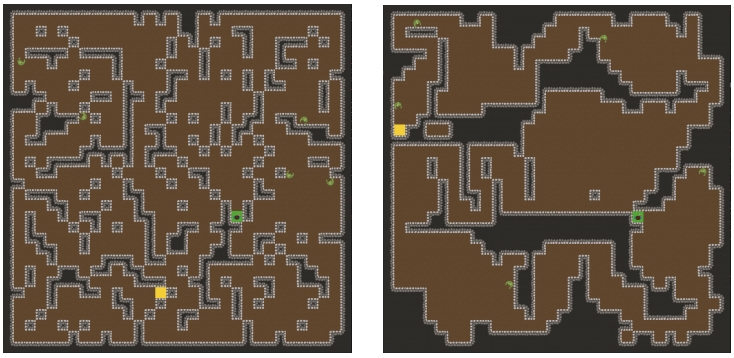
A função de caminho mais curto foi utilizada em todas as combinações, pois é ela que garante que o caminho entre o início e o fim dos níveis existirá, mas não será trivial. As funções F_1, F_2 e F_3 ilustram o efeito de cada uma das funções combinadas com o caminho mais curto isoladamente. A função F_4 foi a função escolhida para gerar os níveis do jogo, pelos motivos explicados a seguir.

A combinação das funções foi feita de forma multiplicativa por dois motivos. Primeiramente, a escala dos valores de aptidão atribuídos pelas funções é diferente. Enquanto a função de vizinhos por parede tem valores na faixa de 0 a 4, a função de caminho mais curto tem valores máximos em torno de 0.3. Logo, se esses valores fossem somados, algumas funções teriam peso maior que outras.

O segundo motivo é incentivar o equilíbrio entre as funções. Se o valor das funções fosse somado, uma ou mais das funções poderia ter valor 0, desde que as outras compensassem por isso. Assim, seria possível, por exemplo, obter um nível impossível de ser completado com aptidão alta, pois a aptidão 0 dada pela função de caminho mais curto seria ignorada.



(a) F_1 (b) F_2



(c) F_3 (d) F_4

Figura 11 – Testes das funções de aptidão utilizando a representação direta

Como não existe um critério formal de qualidade de um nível, características desejáveis devem ser escolhidas de forma subjetiva. Algumas dessas características que foram identificadas são:

- **Desafio:** Um nível que pode ser completado de maneira trivial não é interessante ou engajante para o jogador. Por isso, é interessante que haja um caminho longo entre o início e fim do nível. Isso dá mais oportunidades para o jogador encontrar monstros e tomar decisões sobre como chegar ao fim do nível.

- **Aparência:** Níveis onde os blocos de parede e chão parecem estar espalhados de forma aleatória (como, por exemplo, nas figuras 7a e 7d) aumentam a quantidade de informações a serem processadas pelo jogador. Níveis com áreas abertas e paredes mais coesas, como nas figuras 11a e 11d tendem a ser mais visualmente agradáveis. Ao mesmo tempo, as formas retangulares resultantes da representação indireta têm aparência mais uniforme e menos natural que as formas irregulares formadas pela representação direta, que tendem a ser mais interessantes.

- **Facilidade de movimentação:** Ao testar diferentes funções de aptidão, percebeu-se que corredores estreitos e tortuosos eram mais difíceis de navegar, visto que o jogador colidia mais frequentemente com a parede. Por isso, é

interessante que os níveis tenham áreas abertas onde a movimentação é mais fácil. Isto também dá ao jogador mais versatilidade no combate com inimigos.

Foram testadas diferentes combinações de funções, e a função F_4 exibiu resultados satisfatórios de acordo com as características acima. Ela dá grande peso à função de vizinhos por parede, o que incentiva fortemente que os blocos de parede tenham vizinhos e que as áreas de parede sejam coesas. A função de porcentagem preenchida equilibra esta tendência a ter um grande número de paredes, pois tem um objetivo de preenchimento de 35%. Assim, os níveis têm áreas abertas espaçosas que facilitam a movimentação.

Por esses resultados satisfatórios, a função F_4 foi escolhida para o jogo. Ela é utilizada com os seguintes parâmetros:

- Tamanho da população: 500
- Número de gerações: 100
- Taxa de mutação: 5 / indivíduo
- Tamanho dos torneios: 16

O tempo médio de geração de um nível é de 33.652 s, calculado com base em 10 execuções do algoritmo em um processador Intel Core i5 (4x 3.5 GHz).

6 Considerações Finais

6.1 Conclusão

Neste trabalho, foram desenvolvidos um algoritmo genético para gerar níveis para um jogo procedimentalmente e um jogo simples para testar o algoritmo. O objetivo de controlar características dos níveis gerados foi atingido através da combinação de diferentes funções de aptidão, e a qualidade dos níveis gerados atingiu um nível satisfatório.

O uso de algoritmos como este para geração de níveis é útil para desenvolvedores de jogos, especialmente aqueles com recursos limitados, pois os permite investir em outras áreas do desenvolvimento tempo e mão de obra que, caso contrário, seriam utilizados para desenvolver níveis.

Seu uso também pode beneficiar os jogadores desses jogos, pois a geração procedimental de níveis permite uma variedade praticamente infinita entre os níveis.

6.2 Trabalhos futuros

A principal área para melhora do projeto é a performance. O tamanho escolhido para os níveis do jogo é relativamente pequeno, mas, mesmo assim, o tempo que o algoritmo leva para gerar um nível é longo. Idealmente, os níveis seriam maiores e gerados mais rapidamente.

Uma possibilidade de otimização a ser investigada é a geração de níveis em segundo plano: enquanto o jogador joga, o nível seguinte começa a ser gerado paralelamente.

Outra possível área de melhora é a qualidade dos níveis gerados. Atualmente, os níveis gerados, em sua maioria, têm apenas um caminho linear, sem bifurcações, do início ao fim do nível. Para tornar a exploração dos níveis mais engajante, seria interessante desenvolver funções de aptidão que recompensassem níveis com caminhos mais complexos.

Referências

TOGELIUS, J. et al. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, v. 3, n. 3, p. 172–186, 2011. Citado na página 5.

LINDEN, R. van der; LOPES, R.; BIDARRA, R. Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, v. 6, n. 1, p. 78–89, 2014. Citado 2 vezes nas páginas 5 e 6.

MCMILLEN, E.; HIMSL, F. *The Binding of Isaac*. 2011. Disponível em: <https://bindingofisaac.com/>. Citado na página 5.

BRACE YOURSELF GAMES. *Crypt of the Necrodancer*. 2015. Disponível em: <https://braceyourselfgames.com/crypt-of-the-necrodancer/>. Citado na página 5.

YU, D.; MOSSMOUTH. *Spelunky*. 2008. Disponível em: <https://spelunkyworld.com/>. Citado na página 5.

VIANA, B. M.; SANTOS, S. R. dos. A survey of procedural dungeon generation. In: IEEE. *2019 18th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. [S.l.], 2019. p. 29–38. Citado 2 vezes nas páginas 5 e 6.

LACERDA, E. G. de; CARVALHO, A. D. Introdução aos algoritmos genéticos. *Sistemas inteligentes: aplicações a recursos hídricos e ciências ambientais*, v. 1, p. 99–148, 1999. Citado na página 6.

ASHLOCK, D.; LEE, C.; MCGUINNESS, C. Search-based procedural generation of maze-like levels. *IEEE Transactions on Computational Intelligence and AI in Games*, v. 3, n. 3, p. 260–273, 2011. Citado 4 vezes nas páginas 7, 9, 10 e 15.

VALTCHANOV, V.; BROWN, J. A. Evolving dungeon crawler levels with relative placement. In: *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*. [S.l.: s.n.], 2012. p. 27–35. Citado 2 vezes nas páginas 7 e 9.

BALDWIN, A. et al. Mixed-initiative procedural generation of dungeons using game design patterns. In: *2017 IEEE Conference on Computational Intelligence and Games (CIG)*. [S.l.: s.n.], 2017. p. 25–32. Citado 2 vezes nas páginas 7 e 9.

MONOGAME. Acessado: 24/06/2022. Disponível em: <https://www.monogame.net/>. Citado na página 8.

MICROSOFT. *What is .NET Framework? A software development framework.* Acessado: 26/06/2022. Disponível em: <https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet-framework>. Citado na página 8.