UNIVERSIDADE DO ESTADO DO RIO DE JANEIRO

INSTITUTO POLITÉCNICO

GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

Luis Henrique Costa Vogt

Posicionamento procedural de itens em mapas de jogos através de algoritmos genéticos

Nova Friburgo

2018

UNIVERSIDADE DO ESTADO DO RIO DE JANEIRO

INSTITUTO POLITÉCNICO

GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

Luis Henrique Costa Vogt

Posicionamento procedural de itens em mapas de jogos através de algoritmos genéticos

Trabalho de Conclusão de Curso apresentado - como pré-requisito para obtenção do título de Engenheiro de Computação, ao Departamento de Modelagem Computacional do Instituto Politécnico, da Universidade do Estado do Rio de Janeiro.

Orientadores: Prof. Edirlei Soares de Lima

Profa. Silvia Cristina Dias Pinto

Nova Friburgo

2018

Ficha elaborada pelo autor através do
Sistema para Geração Automática de Ficha Catalográfica da Rede Sirius - UERJ

V886 Vogt, Luis Henrique Costa
Posicionamento procedural de itens em mapas de jogos através de algoritmos genéticos / Luis Henrique Costa Vogt. - 2018.
58 f.

Orientador: Edirlei Soares de Lima
Trabalho de Conclusão de Curso apresentado à Universidade do Estado do Rio de Janeiro, Instituto Politécnico, para obtenção do grau de bacharel em Engenharia da Computação.

1. Geração Procedural - Monografias. 2. Algoritmo Genético - Monografias. 3. Geração de mapas de jogos - Monografias. I. Lima, Edirlei Soares de. II. Universidade do Estado do Rio de Janeiro. Instituto Politécnico. III. Título.

CDU 004.41

Luis Henrique Costa Vogt

Geração de itens em um mapa através de um algoritmo genético

Trabalho de Conclusão de Curso apresentado como pré-requisito para obtenção do título de Engenheiro de Computação, ao Departamento de Modelagem Computacional, do Instituto Politécnico, da Universidade do Estado do Rio de Janeiro.

Aprovado em 21 de setembro de 2018.

Orientador: Prof. Dr. Edirlei Soares de Lima
Instituto Politécnico – UERJ

Banca Examinadora:

_____

Profª. Drª. Sílvia Cristina Dias Pinto (Co-Orientador)
Instituto Politécnico - UERJ

_____

Profª. Drª. Lis Ingrid Roque Lopes Custódio
Instituto Politécnico - UERJ

_____

Prof. Dr. Roberto Pinheiro Domingos
Instituto Politécnico - UERJ

Nova Friburgo
2018

DEDICATÓRIA

Aos meus amigos, Leonardo, Raphael e Victor, que mesmo em épocas ruins me dão suporte. Aos meus pais. À Tristana. Ao rururu. Aos meus avós.

AGRADECIMENTOS

Aos meus amigos. Aos meus pais. Ao meu orientador, Edirlei. Aos bons professores do IPRJ.

RESUMO

VOGT, Luis Henrique Costa. *Posicionamento procedural de itens em mapas de jogos através de algoritmos genéticos.* 2018. 54 f. Trabalho de Conclusão de Curso (Graduação em Engenharia de Computação) - Instituto Politécnico, Universidade do Estado do Rio de Janeiro, Nova Friburgo, 2018.

Neste trabalho são apresentados os resultados obtidos por um programa desenvolvido usando algoritmos genéticos, cujo objetivo é otimizar o posicionamento de itens em um mapa de jogo sem a interação do usuário. O benefício que este trabalho traz é uma maior automação no processo de criação de mapas em jogos, permitindo que o desenvolvedor destine os recursos que seriam dedicados a esta tarefa em outras incumbências. O método proposto fornece, como resultado final, duas imagens. A primeira sendo a representação da posição de cada item, em um formato de arquivo considerado fácil do quesito de importação por outras plataformas e, a segunda sendo um mapa com ícones nas posições dos itens.

Palavras-chave: Geração procedural. Algoritmo genético. Geração de mapas de jogos.

ABSTRACT

VOGT, Luis Henrique Costa. *Posicionamento procedural de itens em mapas de jogos através de algoritmos genéticos.* 2018. 54 f. Trabalho de Conclusão de Curso (Graduação em Engenharia de Computação) - Instituto Politécnico, Universidade do Estado do Rio de Janeiro, Nova Friburgo, 2018.

This work presents the results obtained by a program developed using genetic algorithms, with the goal of optimizing the placement of items in a game map without further input from the user. The benefit of this work is a further automation of the map making processes in games, allowing the developer to allocate resources that would be used in this task to other assignments. The proposed method generates, as the final result, two images. The first one is the representation of each item's position, in a file type that is considered easy to import by other platforms. The second is a map with icons on the items' position.

Keywords: Procedural generation. Genetic algorithm. Game map generation.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

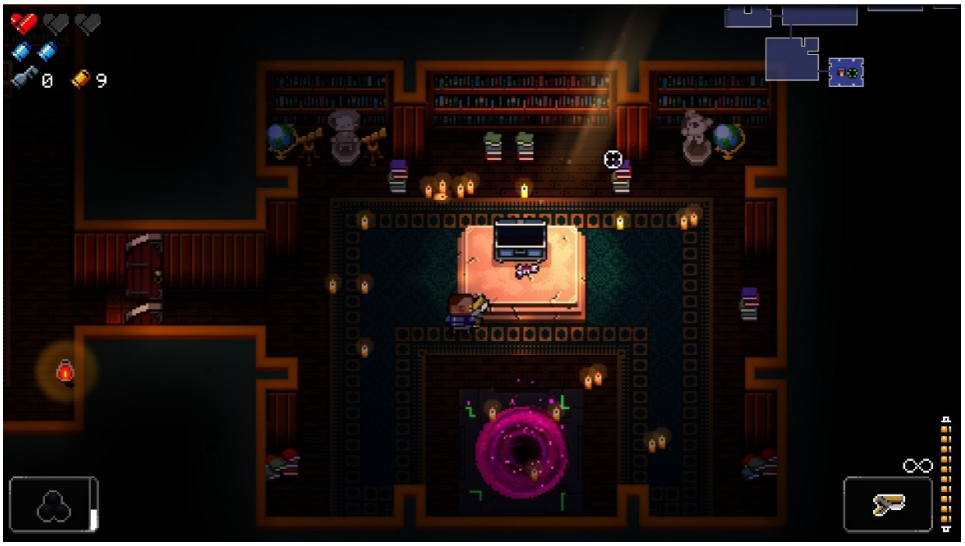
INTRODUÇÃO

Com o avanço rápido da indústria de jogos eletrônicos, está cada vez mais fácil publicar jogos criados por equipes relativamente pequenas, conforme o texto de PEPE (2014). Porém, mesmo com o desenvolvimento de ferramentas que facilitam a produção desses jogos, eles ainda precisam de uma quantidade significante de trabalho para atingirem um estado aceitável para divulgação.

Muitos jogos se passam em um espaço explorável que nós chamamos de mapa. Normalmente existem dois tipos principais de mapa que podem ser utilizados em um jogo: o mapa do mundo e o mapa de masmorras. Enquanto o mapa do mundo costuma conter personagens com quem o jogador pode interagir, cidades e outros conteúdos menos desafiadores, dentro das masmorras o jogador encontrará os desafios mais tradicionais, como inimigos, quebra cabeças, etc. As masmorras costumam ser compostas por várias salas e corredores, onde o jogador também encontrará diversos itens que podem ajudar na sua jornada, como pode-se ver na Figura 1.

Um item é um nome genérico para um objeto que, ao ser adquirido pelo jogador, pode ser utilizado imediatamente, onde o jogador escolher ou em circunstancias específicas, dependendo do tipo de item. Suas utilidades podem variar de curar dano recebido pelo jogador ao enfrentar um inimigo, deixa-lo mais forte para enfrentar esses inimigos, atravessar caminhos bloqueados por um obstáculo tal como uma porta trancada, etc.

Figura 1 - Exemplo de masmorra, com o jogador prestes a adquirir um item



Fonte: ENTER THE GUNGEON, 2016.

Uma das etapas mais trabalhosas desse tipo de desenvolvimento é projetar os mapas onde o jogo se passará e povoar tais mapas com itens distribuídos adequadamente por suas salas. Um método de diminuir o tempo de desenvolvimento e os custos envolvidos com essa etapa é gerar o mapa e o posicionamento dos itens aleatoriamente. Enquanto um desenvolvedor pode escolher os mapas e as distribuições de itens que mais se adequam com o propósito do mesmo, vários jogos incorporam esse aspecto como mecânica fundamental. Jogos como The Binding Of Isaac (2011) criam cada sala como uma peça de um quebra cabeça que podem ser reorganizadas de inúmeras maneiras, de modo que a chance de certa fase ser igual de partida em partida é mínima. Outros jogos, como Rimworld (2018) e Terraria (2011) utilizam algoritmos de geração de mundo que fazem com que o resultado final do mapa seja algo que faça sentido para o jogador.

O objetivo desse trabalho é desenvolver um método para povoar um mapa previamente gerado com itens cujos parâmetros sejam fornecidos. O método proposto calcula os *tiles*, células quadradas de um mapa que formam cada sala, separando-as em entidades apropriadas. Ele escolhe uma sala para ser a entrada da fase e outra para ser o fim, calculando o menor caminho entre essas duas salas utilizando as salas como vértices de um grafo. Os itens são distribuídos em várias configurações aleatórias ao redor do mapa, utilizando os parâmetros adequados a essa etapa. Em seguida, o método utiliza o resto dos parâmetros para julgar a qualidade de cada mapa para, então, utilizar um

algoritmo genético e recombinar a posição dos itens através de várias gerações. O resultado é o mapa com a melhor pontuação, considerando a distância dos itens em relação ao caminho principal e o quão espalhados eles estão pelo mapa.

As etapas desenvolvidas neste trabalho são discutidas a seguir da seguinte maneira: o Capítulo 1 faz um apanhado dos trabalhos relacionados, seguido do Capítulo 2 que apresenta a metodologia aplicada neste trabalho. O Capítulo 3 trata-se do projeto e da arquitetura escolhidos para o desenvolvimento deste trabalho. Detalhes da implementação são apresentados no Capítulo 4. Finalizando então com os Capítulos 5 e 6, os quais apresentam os resultados obtidos e as conclusões finais deste projeto de conclusão de curso.

1 TRABALHOS RELACIONADOS

1.1 Binding of Isaac: Rebirth

Lançado em 2014 pela desenvolvedora e distribuidora Nicalis, o *remake* do The Binding of Isaac original (2011) tem como objetivo percorrer diversos andares, procurando um alçapão que levará o jogador ao próximo nível, derrotando inimigos e coletando itens pelo caminho. Ele utiliza técnicas de geração procedural com base em sementes, combinações de números e letras que definem como o gerador do jogo irá se comportar. Com 2^{32} possíveis sementes, de acordo com RODRIGUEZ (2014), a chance de um jogador participar de duas partidas geradas com a mesma semente é extremamente baixa, a não ser que ele escolha a semente antes de começar a partida. Mesmo assim, dois jogadores que não possuem os mesmos itens desbloqueados podem encontrar itens diferentes, que podem fazer com que aquela partida seja jogada de modo completamente diferente.

O designer do jogo MCMILLEN (2014), entra em alguns detalhes de como o jogo gera o mapa. Os mapas são gerados escolhendo salas aleatórias em uma configuração que gera um nível lógico e completável. Cada sala é gerada em um editor, podendo ter sua chance ajustada de estar presente em um mapa, além de serem configuráveis para não serem acessíveis de certos lados, e o gerador garante que essas salas serão colocadas em locais apropriados, conforme mostra a Figura 2.

Figura 2- Exemplo de um mapa da primeira fase do Binding of Isaac: Rebirth

Fonte: BINDING OF ISAAC: REBIRTH, 2014.

1.2 **Wasteland Kings/Nuclear Throne**

Nuclear Throne, lançado em 2015 pela desenvolvedora e distribuidora Vlambeer, foi inicialmente desenvolvido como Wasteland Kings através de um desenvolvimento de apenas 4 dias para a *gamejam*[1] Mojam, em 2013, como uma parceria entre o estúdio Mojang, criador de Minecraft, adquirido pela Microsoft em 2014, e o site Humble Bundle, que cria diversas ofertas por vários jogos com o objetivo de doar o dinheiro adquirido para caridades. O objetivo do jogo é atravessar várias fases, derrotando todos os inimigos em cada fase para encontrar o portal da próxima, até alcançar o Trono Nuclear.

JW (2013) descreve como eles fazem a geração do mapa e de baús em cada nível. No começo de cada geração, uma instância de um criador de fase, o *FloorMaker* é criado. Ela se move 1 *tile* para frente e cria um *Floor*, um chão. Dependendo da área que está sendo gerada, essa entidade terá diferentes chances de mudar sua direção em 90°, -90° ou até em 180°. A cada *frame* o *FloorMaker* terá uma chance criar um outro *FloorMaker*. Esta chance depende do número de *FloorMaker* ativos na área. Esta criação de múltiplos

[1] Competição onde desenvolvedores devem criar um jogo desde o começo em um curto período de tempo.

FloorMakers muda as características da área, adicionando, por exemplo, corredores. Se mais de um *FloorMaker* existir, a chance de cada um se destruir será maior.

Existem 3 itens que podem ser criados por nível: baús de armas, baús grandes de munição e vasilhas de experiência. Quando o *FloorMaker* muda sua direção em 180°, ele cria um baú de armas. Quando ele se destrói, ele cria um baú de munição. E, quando o nível está se aproximando do tamanho máximo, ele cria uma vasilha de experiência. A Figura 3 apresenta um exemplo deste mapa, depois de todos os inimigos serem derrotados.

Figura 3 - Exemplo de mapa do Nuclear Throne.

Fonte: NUCLEAR THRONE, 2015.

1.3 Rápido Povoamento Procedural de Níveis com Restrições de Jogabilidade

No trabalho apresentado por HORNSWILL & FOGED (2012), introduz-se a noção de restrição de caminhos, que liga funções aos possíveis caminhos que o jogador pode percorrer, e discute como implementar o método.

Os autores discutem como o posicionamento de itens é um passo importante do *design* de um nível. Objetos devem ser colocados não apenas nas quantidades certas, mas em locais apropriados para o nível poder ser concluído.

Utilizando o conceito de restrição de probabilidade, eles consideram, por exemplo, a perda média de vida do jogador em cada sala onde ele enfrenta um inimigo e a vida que ele ganha ao adquirir itens de cura. Considerando a vida inicial e a diferença possível de vida em um caminho é calculada a sobrevivência dos caminhos disponíveis. Se a sobrevivência for maior que zero, o caminho é considerado seguro, e o nível possível de ser completado. Este é um parâmetro que pode ser alterado de acordo com o gosto do desenvolvedor. Além de vida, portas trancadas e chaves também podem ser consideradas restrições, considerando uma porta trancada com pontuação de -1 e uma chave com pontuação de 2. Se a função de caminho for não negativa e tem um valor de 1 na saída, o caminho é considerado completável.

Utilizando programação dinâmica integrado com um sistema de restrição de propagação, eles posicionam os itens em locais adequados.

2 METODOLOGIA

Para conseguir que o programa desenvolvido distribua os itens pelo mapa da forma desejada, é importante que ele consiga alcançar os melhores resultados possíveis sem a interferência do usuário, uma vez definidos os parâmetros iniciais. Para alcançar este objetivo, o conceito de aprendizado de máquina para a criação de um sistema autônomo o suficiente foi escolhido conforme apresentado neste capítulo.

2.1 Métodos de aprendizado de máquina

De acordo com NORVIG e RUSSEL (1995), o campo de inteligência artificial, conhecida pela sigla IA, tem como objetivo entender entidades inteligentes. Ao contrário de filosofia e psicologia, que também se preocupam em entender entidades inteligentes, o objetivo da IA é construir entidades inteligentes além de entendê-las.

O livro de MARSLAND (2015) define a inteligência como aprendizado por experiência, com o aprendizado dando flexibilidade para um indivíduo a se ajustar e adaptar a novas circunstâncias. Aprendizado de máquina se preocupa em fazer o computador modificar ou adaptar suas ações, de forma que essas ações fiquem mais precisas, utilizando uma quantitativa de precisão como uma medida de como as ações escolhidas refletem as ações corretas.

Também de acordo com MARSLAND (2015), os métodos de aprendizado de máquina associam ideias de neurociência, biologia, estatística, matemática e física, proporcionando várias maneiras de classificar métodos de aprendizado de máquina.

Existem vários métodos utilizados para informar à máquina se ela está executando uma tarefa mais efetivamente quando comparada a períodos anteriores. Pode-se informar à máquina se ela alcançou a resposta certa sempre, ou apenas algumas vezes de modo que ela aprenda como chegar na resposta. Também pode-se informar à máquina como chegar neste resultado, ou apenas deixa-la alcança-lo sozinha. Também é possível julgar a resposta dada, atribuindo a ela uma pontuação. Por fim, é possível que não haja respostas

corretas, e que somente se deseja que o algoritmo agrupe as entradas baseando-se no que elas têm em comum. Tendo isso em mente, é possível distinguir quatro métodos de aprendizado: supervisionado, não supervisionado, por reforço e evolucionário.

O aprendizado supervisionado ocorre quando um conjunto de exemplos com as respostas corretas é dado na entrada e, com base nesse conjunto de treinamento, o algoritmo generaliza para responder corretamente a todas as entradas possíveis (MARSLAND, 2015).

O aprendizado não supervisionado não informa as respostas corretas, mas tenta agrupar entradas que possuem atributos em comum. A abordagem estatística para o aprendizado não supervisionado é conhecida como estimação de densidade (MARSLAND, 2015).

O treinamento por reforço é um meio termo entre os aprendizados supervisionados e não supervisionados. O algoritmo é informado se a resposta está errada, mas não como corrigi-la. Ele precisa explorar e tentar possibilidades diferentes até descobrir como encontrar a resposta correta (MARSLAND, 2015).

Aprendizado evolucionário funciona utilizando princípios de evolução biológica. Organismos biológicos se adaptam para melhorar seus índices de sobrevivência e chance de ter um filho em seus respectivos ambiente. Os algoritmos são feitos utilizando a ideia de aptidão, que corresponde a uma pontuação para o quão boa a solução atual é (MARSLAND, 2015).

Dentre os exemplos de aprendizado de máquina, alguns dos citados por MARSLAND (2015) são cadeias de markov, redes neurais, e algoritmos genéticos. Como redes neurais e algoritmos genéticos são alguns dos mais utilizados e podem ser utilizados em conjunto, um foco maior será dado para eles.

Uma rede neural é uma máquina que é projetada para modelar a maneira como o cérebro realiza uma tarefa particular ou função de interesse; a rede é normalmente implementada utilizando-se componentes eletrônicos ou é simulada com o uso de um computador (HAYKIN, 2003).

Após o algoritmo ser implementado, ele precisa passar por um processo de aprendizagem, cuja função é modificar os pesos sinápticos da rede de uma forma ordenada para alcançar um objetivo de projeto desejado.

De acordo com MARSLAND (2015), um bom método para se encontrar boas estruturas para a rede neural é utilizar um algoritmo genético.

Algoritmos genéticos são métodos de otimização e busca inspirados nos mecanismos de evolução de populações de seres vivos. Estes algoritmos seguem o princípio da seleção natural e sobrevivência do mais apto, declarado em 1859 pelo naturalista e fisiologista inglês Charles Darwin em seu livro A Origem das Espécies. De acordo com Charles Darwin "quanto melhor um indivíduo se adaptar ao seu meio ambiente, maior será sua chance de sobreviver e gerar descendentes" (LACERDA, CARVALHO, 1999). Este método de aprendizagem de máquina é um dos focos principais deste trabalho e o tópico será mais aprofundado nas próximas seções.

2.2 **Evolução**

Segundo LINDEN (2003), até o século XIX eram aceitos na comunidade científica duas teorias principais para explicar a existência do mundo ao nosso redor. A primeira é a teoria do criacionismo, que dizia que Deus criou o universo do modo que ele é hoje, e a outra é a teoria da geração espontânea, que dizia que a vida surge de essências presentes no ar. Porém, perto de 1850, Charles Darwin fez uma viagem no navio HMS Beagle que o levou a vários lugares, e a partir de sua observação percebeu que animais de certas espécies eram ligeiramente diferentes de seus parentes em outros ecossistemas, sendo mais adaptados a sobreviver no seu próprio. A partir dessas e outras observações, ele desenvolveu a teoria da evolução das espécies, que foi descrita detalhadamente em seu livro "A Origem das Espécies".

"A teoria da evolução diz que na natureza todos os indivíduos dentro de um ecossistema competem entre si por recursos limitados, tais como comida e água. Aqueles dentre os indivíduos (animais, vegetais, insetos etc.) de uma mesma espécie que não obtêm êxito tendem a ter uma prole menor e esta descendência reduzida faz com que a probabilidade de ter seus genes propagados ao longo de sucessivas gerações seja menor, processo este que é denominado de seleção natural" (LINDEN, 2003).

A combinação de características dos indivíduos que sobrevivem pode resultar em um novo indivíduo melhor adaptado ao seu ambiente. Porém, se características não favoráveis foram herdadas por esse novo indivíduo, elas podem causar um impacto negativo na sua capacidade de sobrevivência. Além disso, podem acontecer mutações que

alteram uma ou mais características, de modo que o filho tenha características que os pais não possuem.

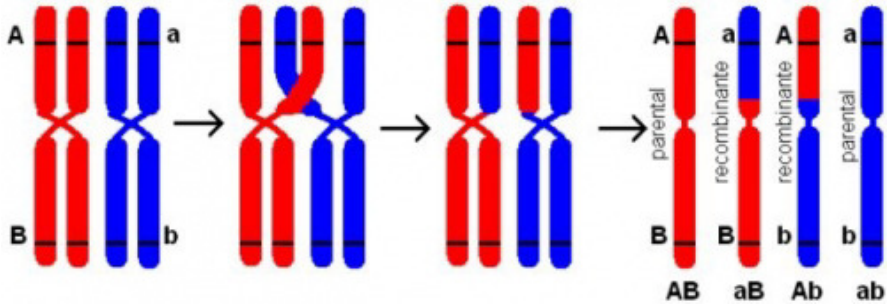
2.3 Genética Básica

LINDEN (2003) fala explica que no século XIX o padre Gregor Mendel compreendeu como o processo de transmissão de características estava associada com uma unidade básica de informação, o gene. Genes contém as informações necessárias para a manifestação de características específicas, e a descoberta de como essas características eram fisicamente armazenadas dentro das células dos indivíduos demorou quase um século, culminando no que hoje nós conhecemos como DNA.

Cada indivíduo vivo é formado por múltiplas células, e cada célula possui uma cópia completa do conjunto de um ou mais cromossomos que descrevem o organismo. Este conjunto é denominado genoma. Cada cromossomo é composto de genes, que são blocos de sequências de DNA e controlam uma característica hereditária específica do indivíduo.

O mecanismo de transmissão que é a base dos Algoritmos Genéticos é a reprodução sexuada (LINDEN, 2003). Nela, dois organismos fornecem um pedaço de material genético chamado gametas. Esses gametas são resultados de um processo chamado *crossover*, ou recombinação genética, demonstrado na Figura 4.

Figura 4 - Exemplo de recombinação genética



Fonte: INFOESCOLA, 2009.

Esse processo é altamente complexo, com a possibilidade de pequenos erros ocorrerem ao longo do tempo, gerando mutações. Essas mutações podem causar mudanças positivas, neutras ou negativas à característica que o gene representa.

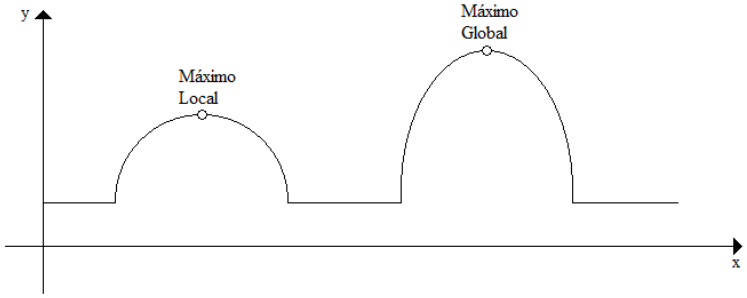
Graças a essas mecânicas de reprodução, os filhos podem herdar diversas características dos pais, benéficas ou não. Espera-se então, através da seleção natural, que os indivíduos que herdaram as características negativas se reproduzam menos, causando um lento desaparecimento dessas características, enquanto os indivíduos com as características positivas as propagam cada vez mais.

2.4 **Algoritmos Genéticos**

Conforme PACHECO (1999), "Algoritmos Genéticos (*GA-Genetic Algorithms*) constituem uma técnica de busca e otimização, altamente paralela, inspirada no princípio Darwiniano de seleção natural e reprodução genética". Estes princípios são imitados na construção de algoritmos computacionais que buscam uma melhor solução para um determinado problema, através da evolução de soluções codificadas através de cromossomos artificiais. No método Algoritmo Genético, cada cromossomo é uma estrutura que representa uma possível solução para o problema. Estes cromossomos são submetidos a um processo evolucionário que envolve a avaliação dos indivíduos, a seleção dos mais aptos, a recombinação sexual e mutações. Após múltiplas gerações, a população deverá conter indivíduos mais aptos.

LINDEN (1999) define algoritmos genéticos como técnicas heurísticas de otimização global. Isso opõe Algoritmos Genéticos a métodos com características gradiente (*hill climbing*) que, com o objetivo de encontrar o valor máximo ou mínimo de uma função, investigam sua derivada, justificando a retenção do algoritmo em máximos locais da função. A Figura 4 dá um exemplo de máximo local e máximo global.

Figura 5 - Exemplo de máximo local e global



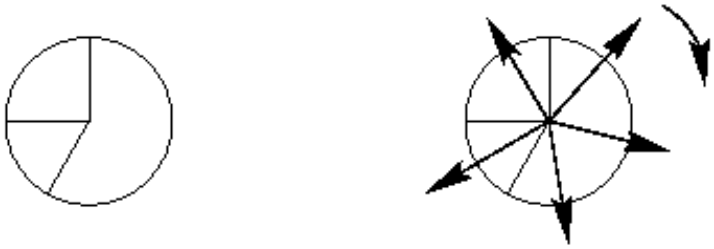
Fonte: O autor, 2018.

Já os Algoritmos Genéticos não ficam estagnados apenas por estarem no máximo local. Como cada geração de indivíduos é igualmente ou mais apta que a anterior, caso algum indivíduo tenha um valor maior do que os que estão próximos de um máximo local, as características que proporcionam este valor se propagarão pelas próximas gerações.

A codificação das informações pertinentes ao problema é um ponto crucial para o funcionamento do Algoritmo Genético. Ela precisa ser feita de modo inteligente, evitando que uma combinação de circunstâncias impossíveis aconteça, quando essas informações forem decodificadas para encontrar a solução final. Assim como na natureza, essas informações serão codificadas no cromossomo.

De acordo com MIRANDA (2018), os indivíduos mais aptos da geração atual são escolhidos para gerar uma nova geração através de recombinação genética entre eles. Através de um método denominado amostragem universal estocástica, é possível fazer uma seleção de indivíduos que dê mais peso para indivíduos mais adaptados, mas sem ignorar indivíduos que não estão entre os melhores avaliados. A maneira como esse método funciona se assemelha a uma roleta sobreposta a um círculo que possui uma porcentagem de sua área proporcional a sua avaliação, assim como demonstrado na Figura 6. Então a roleta é girada para cada pai que será selecionado.

Figura 6 - Roleta sendo girada no sentido horário

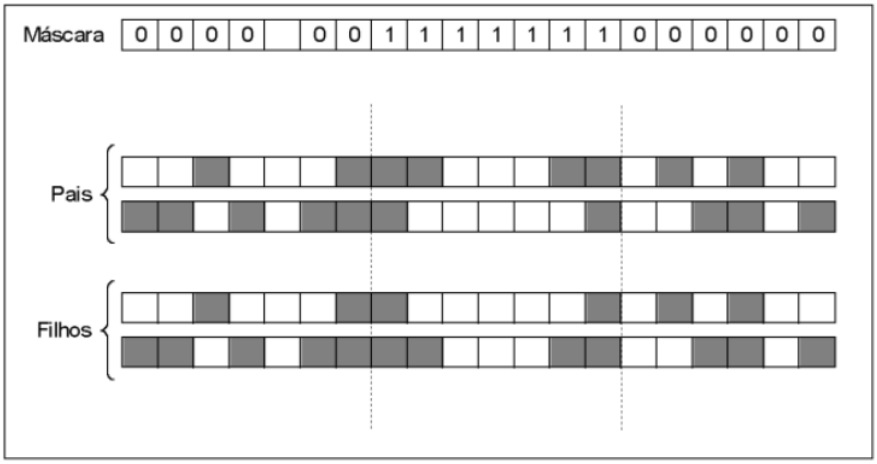


Fonte: MIRANDA, 2018.

O cruzamento ocorre utilizando a recombinação, onde o filho herda os genes de um pai até alcançar um ponto de corte, onde ele começa a herdar os genes de outro pai.

Existem múltiplas maneiras similares, mas distintas, para fazer a recombinação. A que será utilizada neste trabalho é demonstrada na Figura 7.

Figura 7 - Exemplo de *crossover*, com uma máscara que define de qual pai os filhos herdarão os genes



Fonte: LUCAS, 2002.

De acordo com LUCAS (2002), "A mutação opera sobre os indivíduos resultantes do processo de cruzamento com uma probabilidade pré-determinada efetua algum tipo de alteração em sua estrutura". Essa mutação garante que diversas alternativas sejam exploradas, mantendo um nível de abrangência na busca.

Segundo LUCAS (2002), três exemplos de operadores de mutação são: a mutação flip, onde cada gene a sofrer mutação recebe um valor válido para o problema; a mutação

por troca, onde são sorteados pares de genes para serem trocados de lugar um com o outro; e a mutação *creep*, onde um valor aleatório é somado ou subtraído do valor do gene.

Por fim, segundo LUCAS (2002), o processo dos Algoritmos Genéticos é repetido por várias gerações, utilizando como critério de parada desde o número de gerações até o grau de convergência da atual população, ou seja, o grau de proximidade dos valores da avaliação de cada indivíduo da população, entre outros.

3 PROJETO E ARQUITETURA

O objetivo do projeto é utilizar diversas técnicas para: reconhecer as salas presentes na masmorra; investigar quais salas são adjacentes umas das outras através de portas; criar diversas configurações de posição possíveis para cada item; pontuar os itens através de vários métodos de avaliação; e, por fim, utilizar um Algoritmo Genético nestas configurações através de diversas gerações para encontrar um resultado otimizado.

A linguagem escolhida foi C++, graças a sua flexibilidade para o uso de ponteiros, juntamente com a biblioteca Allegro[2] para a manipulação de imagens.

3.1 Allegro

Allegro é uma biblioteca multiplataforma focada primariamente no desenvolvimento de jogos e programação multimídia. Ela se preocupa em realizar tarefas de baixo nível como: criar janelas; receber *input* do usuário; carregar dados; desenhar imagens; reproduzir sons; etc. e no geral abstrair a plataforma subjacente.[3]

Allegro, apesar de no geral ser focada no desenvolvimento de jogos 2D, tem várias ferramentas que podem ser úteis em inúmeros projetos. No caso deste, foca-se na habilidade de manipular imagens pixel a pixel e de importar e exportar imagens.

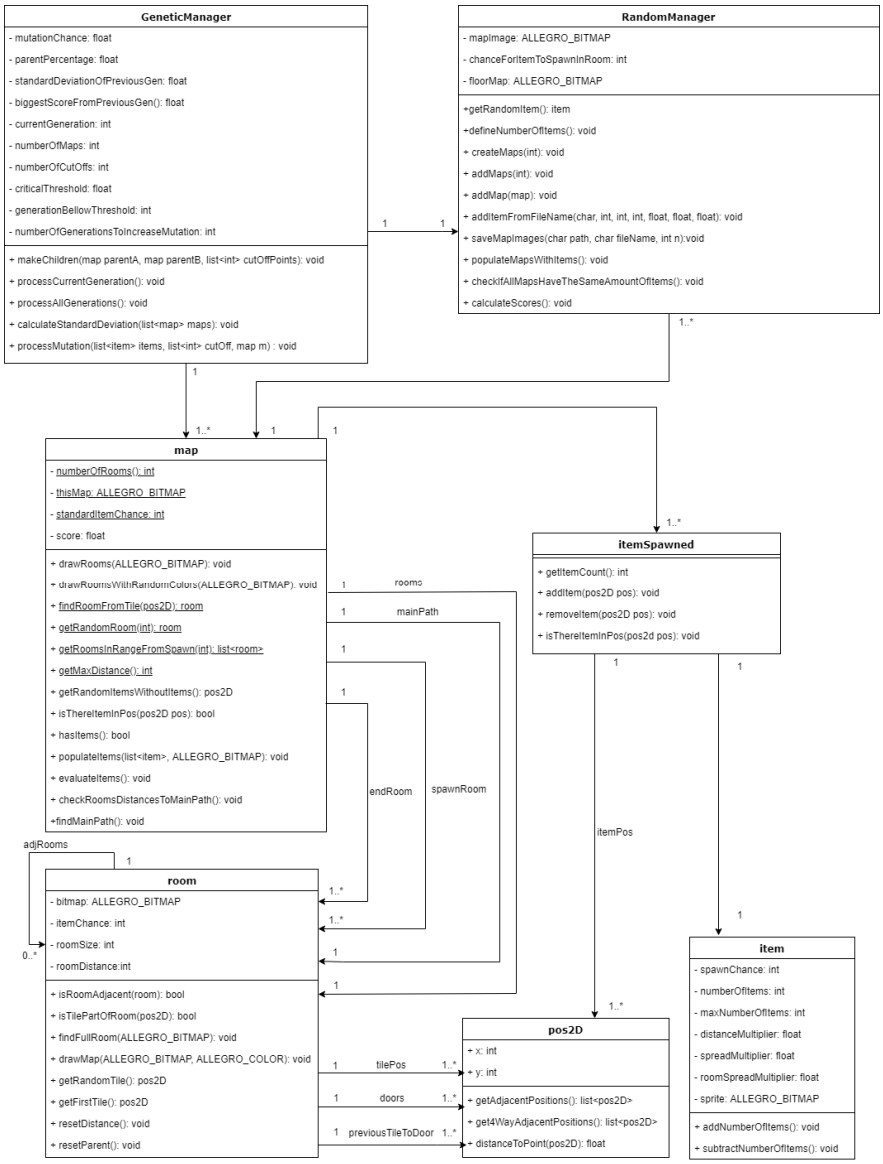
---

[2] http://liballeg.org

[3] http://liballeg.org/index.html

3.2 **Diagrama de classes**

Figura 8 - Diagrama de classes



Fonte: O autor, 2018.

3.3 **Descrição das classes**

3.3.1 Pos2D

A classe Pos2D é vital para representar todas as posições representadas no programa. Utilizando um sistema em grade, com cada célula tendo uma posição *x* e uma posição *y*, esta classe possui métodos para encontrar posições adjacentes em 4 e 8 direções, além de distâncias e outras operações básicas.

3.3.2 Item

Esta classe tem como objetivo guardar as informações de cada tipo diferente de item. Além de guardar os modificadores de pontuação, ela guarda as informações necessárias para definir a quantidade de itens e o *sprite* (representação da imagem pelo computador) do item.

3.3.3 ItemSpawned

O objetivo principal desta classe é guardar as posições de cada instância do item associado a ela, adicionando e removendo posições através de métodos. Além disso, existem métodos para verificar se existe algum item em uma posição específica ou adquirir a quantidade total de itens presentes no mapa.

3.3.4 Room

Com o mapa em si dividido em células e o acesso de um possível jogador estando limitado a um conjunto dessas células, uma classe que às agregue é necessária. A lista *tilePos* serve esta função, guardando todas as posições que constituem à sala. Existe também uma lista de portas e de posições de *tiles* pertencentes à sala em frente às respectivas portas.

Além de listas guardando os componentes da sala, existe uma lista de salas adjacentes, essencial para algoritmos de busca de caminho, uma referência ao *bitmap* do mapa ao qual a sala está associada e respectivos métodos para desenhar a sala.

A maioria dos métodos tem como objetivo encontrar *tiles*, verificar se salas são adjacentes ou modificar as variáveis referentes aos algoritmos de busca.

3.3.5 Map

A classe *map* é um agregado de *itemSpawned*, salas e métodos responsáveis por gerar o mapa em si, os itens e calcular a pontuação da distribuição destes itens. As salas individuais são guardadas como variáveis estáticas, permitindo que todos os mapas tenham acesso a elas sem a necessidade de gastar memória adicional. Como a única coisa que terá variações depois que as salas forem geradas são as posições dos itens, não há necessidade de gerar o mapa múltiplas vezes.

Além dos métodos descritos, a classe possui métodos para encontrar o caminho mais curto entre duas salas, as quais são definidas aleatoriamente. Este caminho é necessário para um dos métodos de avaliação dos itens.

3.3.6 RandomManager

Esta classe é a responsável principal por definir o número de itens que cada mapa terá, além de criar os itens a partir dos parâmetros dados na inicialização do programa. Contém uma lista de mapas, gerenciando cada um e garantindo que eles possuem a mesma quantidade de itens.

3.3.7 GeneticManager

O único objetivo desta classe é executar o Algoritmo Genético. Todos os parâmetros e métodos têm este propósito, ou seja, executa o algoritmo em gerações ou executa partes deste algoritmo.

4 IMPLEMENTAÇÃO

Com a intenção de encontrar a melhor configuração de itens em um dado mapa, várias configurações são geradas no começo do programa, e o Algoritmo Genético é utilizado para este propósito. Os itens são julgados pela sua posição em relação ao caminho principal, que é definido pelo programa, através de uma medida que dirá o quão espalhados os itens estão pelo mapa utilizando a informação do desvio padrão e em relação à quantidade dos mesmos itens presentes na mesma sala.

4.1 Criação da imagem do mapa

A primeira etapa executada pelo programa é abrir um arquivo de texto constituído de números: 0, 1, 2 e 3, conforme visto na Figura 9. O dígito 0 (zero) representa um *tile* intransponível, fora do mapa em si, o dígito 1 representa lugares acessíveis, os dígitos 2 e 3 representam paredes e portas, respectivamente. Essa conversão é importante, pois após cada execução uma imagem representando os itens e o mapa é gerada, para que o usuário do programa possa decidir se a configuração de itens é adequada para os seus propósitos.

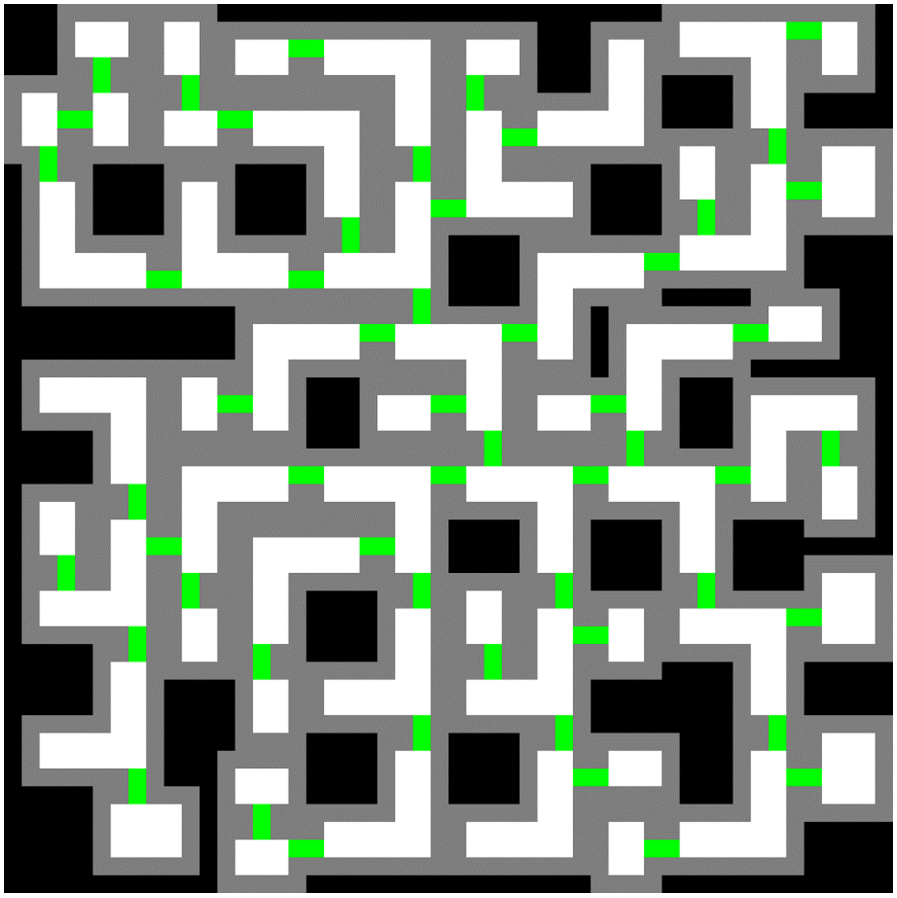
Figura 9 - Fragmento do arquivo utilizado para gerar o mapa utilizado neste trabalho

```
00022222222200000000000000000000000002222222222220
00021112211222222222222222222000222221111113311 20
00021112211221113311111122111200021122111111221120
00022322211221112211111122111200021122222211221120
22222322223222222222221122322200021120000211222220
21122112223222222222211223222222211200002112000 0
21133112211133111112211221122111111200002112000 0
21122112211122111112211221133111111222222232222 2
22322222222222222211222322112222222222112223221112
02322000022220000211222322112222200002112211221112
02112000021120000211221122111112000021122113311112
02112000021120000211221133111112000022322112211112
02112000021120000223221122222222200002232211222222
02112222221122222223221120000222222221111112000000
02111111221111112211111120000211111133111111200000
02111111331111113311111120000211111122222222200000
02222222222222222222223200002112222200000222220000
00000000000002222222223222222112022222222211120000
00000000000002111111331111113311202111111331112000
00000000000002111111221111112211202111111222222000
```

Legenda: (dígito 0) - representa espaços intransponíveis; (1) - representa lugares acessíveis; (2) - representa paredes; (3) - representa portas.
Fonte: O autor, 2018.

A conversão é feita de modo extremamente simples. Um *bitmap* completamente preto é formado com as dimensões fornecidas. O programa então lê o arquivo de texto linha por linha, caractere por caractere, traduzindo os números para a cor adequada. O resultado se torna um mapa similar ao mostrado na Figura 10.

Figura 10 - Mapa formado



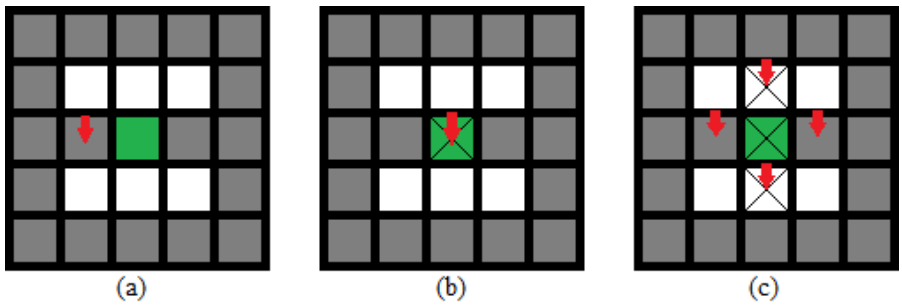
Legenda: (*Tile*s pretos) - representam lugares intransponíveis; (brancos) - lugares acessíveis; (cinzas) - paredes; (verdes) - portas.
Fonte: O autor, 2018.

A imagem utilizada pelo programa tem como tamanho de cada *tile* apenas um pixel, mas isso deixaria a imagem muito pequena. Logo foi necessário salvar a imagem com cada *tile* ocupando 48x48 pixels de modo a ser facilmente visível. Porém, um mapa com apenas estas características não é o suficiente para o propósito deste trabalho, sendo necessário encontrar as salas individuais e armazená-las em uma estrutura de dados adequada.

4.2 **Encontrando as salas**

Para encontrar as salas é necessário ter pelo menos um dos *tiles* que as compõem. Utilizaremos os *tiles* de chão adjacentes aos *tiles* de porta. O programa verifica cada pixel da imagem, e se esse encontrar um pixel verde, ele verifica se existem pixels brancos ao seu redor. Como cada porta está cercada por duas paredes e a mesma é a divisória entre duas salas, ela será adjacente a dois *tiles* de duas salas diferentes. Esse processo pode ser visto na Figura 11.

Figura 11 – Lógica da tarefa de encontrar as portas e salas adjacentes



Legenda: (a) - o programa avalia o *tile* apontado pela seta vermelha, sendo este uma parede; (b) - ele avalia o *tile* verde e, reconhecendo que ele é uma porta;
(c) - ele avalia os 4 *tiles* adjacentes à porta, encontrando os *tiles* brancos, que representam o chão.
Fonte: O autor, 2018.

Uma vez encontrados os pares de *tiles* para cada sala, é verificado se um ou ambos dos *tiles* se encontram em uma sala já encontrada. Se isso ocorrer, o ponteiro referente àquela sala é adquirido. Se não ocorrer, um novo objeto do tipo sala é criado, e seu ponteiro adquirido. Com ambos ponteiros adquiridos, eles são colocados na lista de salas adjacentes da sala analisada.

Ao criar um objeto do tipo sala, uma busca em largura é executada.

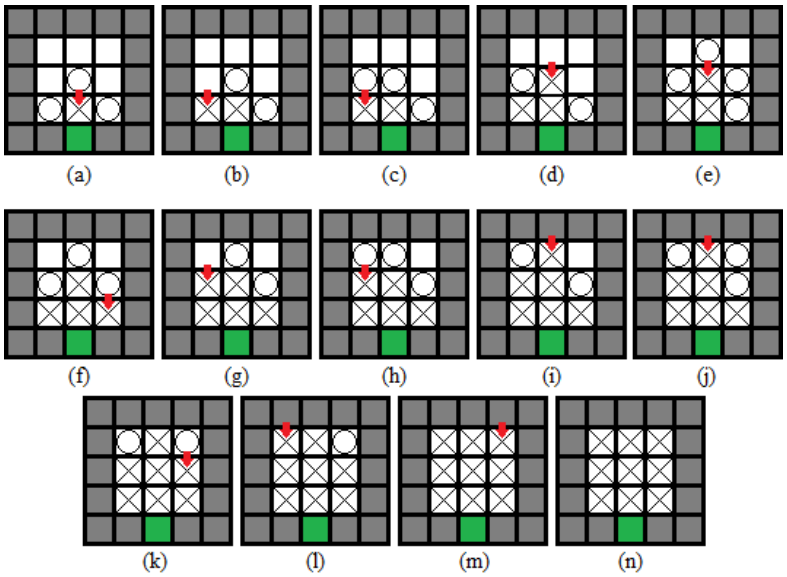
4.2.1 Busca em Largura

De acordo com LEVITIN (2012), a busca em largura é um percurso dos cuidadosos. Ela procede de uma maneira concêntrica visitando primeiro todos os vértices que estão adjacentes ao vértice inicial, depois visitando todos os vértices que estão duas arestas de distância do inicial, e etc., até que todos os vértices do mesmo componente que o vértice inicial tenham sido visitados. Caso ainda existam vértices não visitados, o algoritmo é reiniciado em um vértice arbitrário de outro componente conectado do grafo.

É conveniente usar uma fila para seguir as operações da busca em largura. A fila é iniciada com o vértice inicial, que é marcado como visitado. Em cada iteração, o algoritmo identifica todos os vértices que são adjacentes ao vértice frontal, marca eles como visitados e os adiciona à fila, removendo então o vértice frontal da fila.

A busca em largura possuí eficiência de $\theta(|V|^2)$ para a representação em matriz de adjacência e $\theta(|V| + |A|)$, onde V é o número de vértices e A o número de arestas, para a representação em lista de adjacência. Para este trabalho, será utilizada a representação em lista de adjacência, demonstrada pela Figura 12.

Figura 12 - Exemplo de busca em largura encontrando uma sala.



Fonte: O autor, 2018.

A Figura 12 demonstra a busca em largura para encontrar toda a sala. Ela começa no *tile* adjacente à porta, que já fora previamente reconhecido como pertencente à sala. Depois disso, ele verifica todos os *tiles* adjacentes a ele que também representam chãos e os guarda numa lista de elementos não visitados, representadas por círculos em (a). Em seguida, ele percorre esta lista na ordem que seus elementos foram adicionados à ela, como visto em (b), e coloca o *tile* em uma lista de visitados, retirando-o da lista anterior. Em (c) é verificado as adjacências daquele elemento, e em (d) ele vai para o próximo elemento da lista de não visitados. Esse processo se repete até que não existam mais *tiles* não visitados.

Utilizando os *tiles* como vértices em um grafo, onde apenas seus vizinhos brancos são vértices adjacentes, é possível encontrar todos os *tiles* pertencentes a sala que são válidos como localidade para os itens.

4.3 **Encontrando o caminho principal**

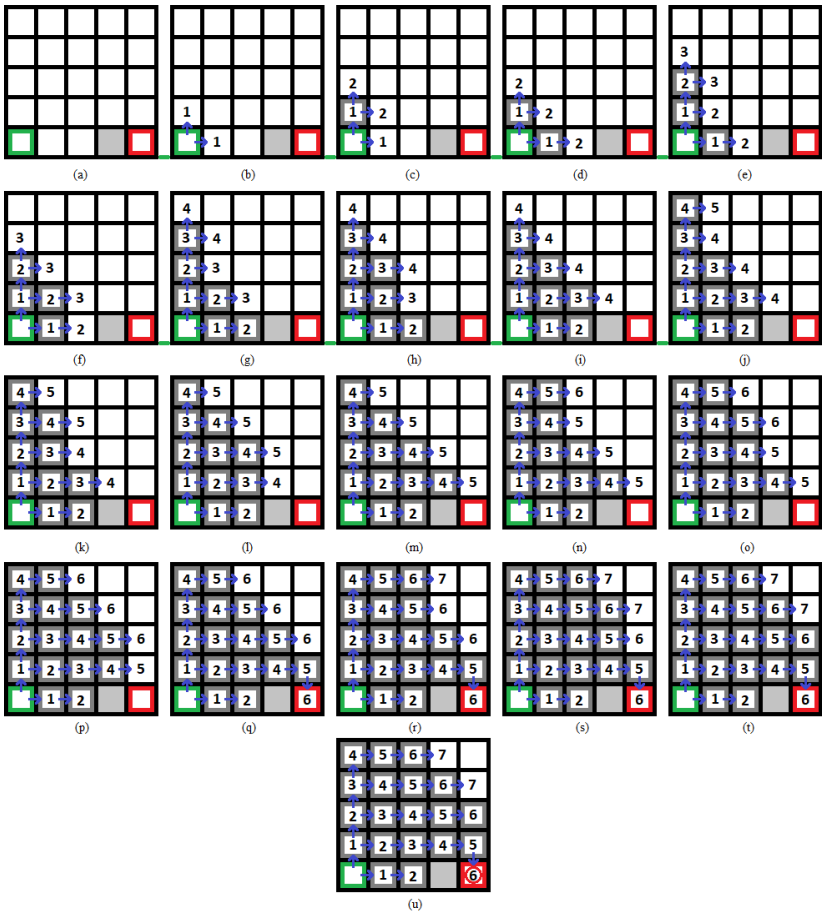
Para efetuar avaliações em etapas posteriores, é necessário que o mapa tenha um caminho principal, que é o menor caminho entre a sala de entrada e de saída do mapa. Para fazer isso, uma sala é escolhida aleatoriamente como sala de entrada. O programa então verifica a distância entre todas as salas e a sala de entrada, escolhendo uma sala aleatória com a maior distância possível para ser a sala de saída. O algoritmo de Dijkstra, apresentado por LEVITIN (2012), é utilizado para encontrar tanto a distância das salas para a sala de entrada como para encontrar o caminho entre a sala de entrada e de saída.

4.3.1 Algoritmo de Dijkstra

De acordo com LEVITIN (2012), o Algoritmo de Dijkstra só é aplicável para grafos direcionados ou não direcionados, com pesos não negativos. Como a maioria das aplicações satisfazem esta condição, a limitação não afetou a usabilidade do algoritmo.

O Algoritmo de Dijkstra acha a menor distância entre um vértice de um grafo e um vértice inicial. Primeiramente, ele acha o menor caminho da fonte para um vértice próximo a ela, depois para o segundo vértice mais próximo a fonte, etc. Em geral, antes da i-ésima iteração, o algoritmo identifica o menor caminho até os i-1 vértices percorridos.

Figura 13 - Demonstração do algoritmo de Dijkstra



Nota: Neste caso, o algoritmo busca a distância entre o tile verde e o vermelho.
Fonte: O autor, 2018.

A Figura 13 demonstra o processo para encontrar o caminho mais curto entre o *tile* verde para o *tile* vermelho utilizando o Algoritmo de Dijkstra. A Figura 13 (a) representa o mapa antes do algoritmo ser executado (b) representa o primeiro passo, onde as distâncias entre os *tiles* adjacentes ao inicial são calculadas e salvas. As setas azuis indicam o caminho que é utilizado para chegar nos respectivos *tiles*. Em (c), os vértices adjacentes ao primeiro vértice analisado tem suas distâncias calculadas, sendo este cálculo feito pela distância entre o vértice analisado e o adjacente mais a distância salva do vértice analisado. Este processo é repetido até o vértice final for analisado, sendo que caso um vértice adjacente previamente analisado for reanalisado e possuir uma distância menor do que ele possuía antes, a nova distância calculada é salva e o caminho é redirecionado para ser entre o vértice analisado e o novo vértice.

4.4 Criando a primeira geração

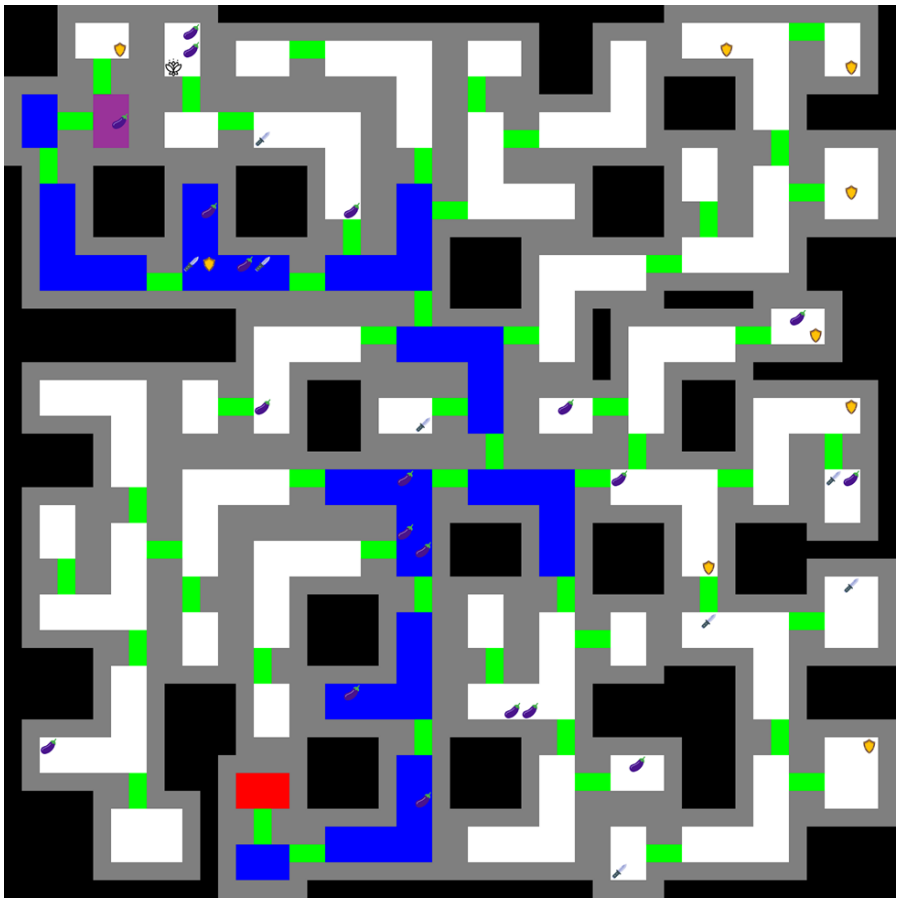
Para criar a primeira geração e começar a utilizar o Algoritmo Genético, é necessário um certo nível de preparo. Primeiramente, *n* objetos do tipo mapa são criados, sendo *n* o número da população. Como a lista de salas, a sala inicial e a sala final são variáveis estáticas, todos os objetos as possuem. Depois o número de itens de cada tipo que irão povoar os mapas precisa ser definido, pois todos os mapas precisam ter o mesmo número de itens para que eles possam ser comparados efetivamente.

Cada item pode possuir um limite mínimo e/ou máximo de instâncias. Um peso, representado por um número inteiro, também pode ser atribuído, o qual definirá as chances desse item ser instanciado ao invés de outros. Para decidir quantos itens serão instanciados nos mapas individuais, o seguinte processo ocorre:

a) Para cada sala contida no mapa, um número aleatório entre 1 e 100 é gerado. Se esse número for menor que a chance geral de um item for instanciado, o programa vai para o próximo passo.
b) Um outro número aleatório é gerado e, utilizando amostragem universal estocástica, um item é escolhido e seu número de instâncias incrementado em um. Caso tenha passado do número máximo, nada acontece. O valor inicial das instâncias é 0, caso não haja mínimo, ou o valor mínimo.

O número total de itens então é escolhido. Após esse processo, um *tile* válido aleatório é escolhido para cada instância de cada item em cada mapa, sendo armazenado no objeto *itemSpawn*. O resultado é parecido com o mostrado pela Figura 14, com os itens não tendo nenhum critério para sua localização.

Figura 14 - Um exemplo de um mapa populado com itens



Legenda: (vermelho e roxo) - são as salas de entrada e saída; (azul) - as salas em azul fazem parte do caminho principal.
Fonte: O autor, 2018.

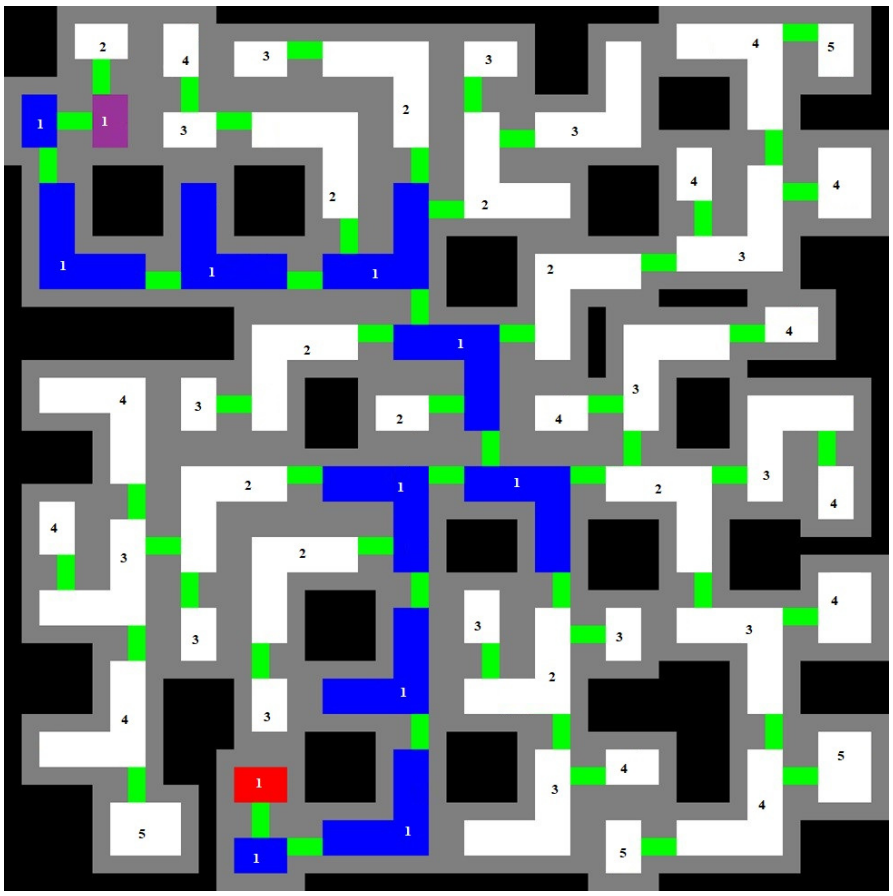
4.5 Calculando a pontuação dos mapas

Para calcular a pontuação dos mapas, três métodos de avaliação são utilizados. Cada método foi multiplicado por certas constantes para tentar fazer com que os valores sejam equivalentes, de modo a não dar mais importância a um em relação aos outros.

4.5.1 Método 1: Medir a distância entre a sala que contêm o item e o caminho principal

O primeiro método é achar a distância mais curta de cada item para o caminho principal. Essa distância é baseada em sala, sendo que se o item está em uma sala no caminho principal, a pontuação base é um. Se ele estiver em uma sala adjacente ao caminho principal, a pontuação base é dois, etc. Essa distância é calculada pelo Algoritmo de Dijkstra, previamente explicado. O caminho entre a sala em que o item se encontra e cada sala do caminho principal é calculado, com o menor caminho sendo escolhido e o número de salas neste caminho sendo a pontuação. Para facilitar estes cálculos, ao invés de calcular a distância da sala dos itens para o caminho principal para todos os itens, a distância de cada sala para o caminho principal é calculada e armazenada nos objetos das respectivas salas. Para aumentar a versatilidade do programa, um parâmetro que multiplica a pontuação por distância pode ser utilizado, dando mais importância para a distância de itens que tenham esse multiplicador maior, e menos para um multiplicador menor. Isso permite que o usuário escolha quais itens tenderão a estar mais distantes do caminho principal. Para o mapa utilizado para os testes, as distâncias de cada sala até o caminho principal são as mostradas na Figura 15.

Figura 15 - Mapa com as distâncias em cada sala



Legenda: (1) - As salas com a distância 1 fazem parte do caminho principal.
Fonte: O autor, 2018.

4.5.2 Método 2: verificar o quão espalhados os itens estão em relação a uns aos outros

O segundo método trata de verificar o agrupamento dos itens. Quanto mais espalhados os itens estiverem, melhor. Para verificar esse espalhamento, é calculado o desvio padrão da posição de cada item no eixo *x* e no eixo *y* do mapa, separadamente.

O desvio padrão, de acordo com CORREA (2003), é a raiz quadrada da variância, que por sua vez "é a medida que dá o grau de dispersão (ou de concentração) de probabilidade em torno da média".

$$Dp = \sqrt{\sum_{i=1}^{n}(x_i - \bar{x})} + \sqrt{\sum_{i=1}^{n}(y_i - \bar{y})}$$

Equação 1 – Desvio padrão.

Como esse cálculo só envolve as coordenadas, a configuração das salas não vai influenciar o resultado. Além disso, apenas um tipo de item é comparado por vez, de modo que a posição de cada tipo de item não interfere com os outros.

Como a medida do desvio padrão não é linear, balancear esse método com os outros foi difícil. A solução encontrada foi multiplicar a parte da pontuação referente ao desvio padrão por uma constante arbitrária.

4.5.3 Método 3: verificar a quantidade de itens em cada sala

O último método é um complemento para o segundo método. Para cada item *i*, ele eleva o multiplicador atribuído ao item, m_i, pelo número *n* de itens do mesmo tipo presentes na sala, até fazer isso para as *j* variedades de itens presentes, ou seja:

$$Pontuação\ da\ sala = \sum_{i=0}^{j} m_i^n$$

Equação 2 – Pontuação da sala.

Desta forma, se o multiplicador for entre 0 e 1, quanto mais itens estiverem na sala, menor a sua pontuação e o Algoritmo Genético tenderá a espalha-los mais entre as salas. Se o multiplicador for maior que 1, o Algoritmo Genético tenderá a agrupar mais destes itens na mesma sala. Se o multiplicador for 1, não haverá diferença entre esta sala e uma sala vazia para este item.

Como ter um item na sala cujo multiplicador seja entre 0 ou 1 diminuiria a pontuação referente àquela sala, apenas o segundo item localizado em cada sala afetará a pontuação. Isso garante que salas vazias não tenham uma pontuação maior que salas com

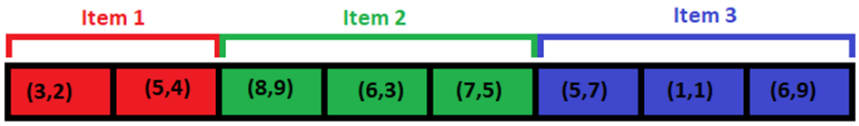
apenas um item, o que faria que o Algoritmo Genético agrupasse itens em salas mesmo com um multiplicador que tenha o objetivo de espalha-los.

Como o cálculo da pontuação da sala não é linear, dificuldades foram encontradas para balancear essa pontuação com as outras. Como solução, ela foi multiplicada por uma constante arbitrária.

4.6 Implementação do Algoritmo Genético

Para a implementação do Algoritmo Genético, a classe *itemSpawn* representa os genes. Sua estrutura é simples, possuindo apenas um ponteiro para o item pai e uma lista de posição. Cada mapa possui uma lista de *itemSpawn*, sendo que cada um desses objetos representa instâncias de itens diferentes. Desta forma pode-se ver cada *itemSpawn* como um cromossomo diferente, capaz de fazer o *crossover* entre os *itemSpawns* equivalentes, mas nunca entre dois que representam itens diferentes. A Figura 16 representa como esta lista é armazenada.

Figura 16 - Representação de vários *itemSpawns* dentro de um mapa

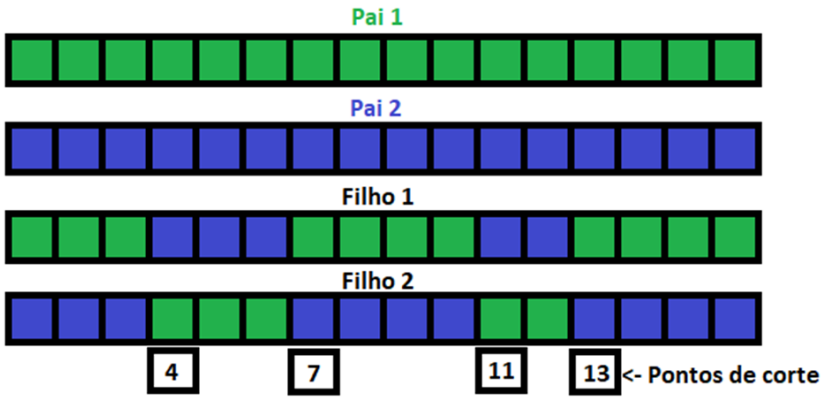


Fonte: O autor, 2018.

O primeiro passo para calcular uma nova geração é decidir os pontos de corte. O número de pontos de corte é definido pelo usuário, mas os pontos de corte em si são decididos aleatoriamente em um intervalo entre 1 e o número total de itens no mapa.

Com os pontos de corte decididos, o próximo passo é gerar os filhos. Os pais são aleatórios, sendo que todos os pais se reproduzem, gerando dois filhos. O primeiro filho herda os genes do primeiro pai e o segundo filho herda os genes do segundo pai até o primeiro ponto de corte, onde essa herança é invertida. Pode-se ver um exemplo disso na Figura 17.

Figura 17 - Exemplo de como o *crossover* é feito



Fonte: O autor, 2018.

Uma vez feito o *crossover*, os dois filhos são salvos em uma lista e a pontuação do mapa é gerada. Essa etapa é repetida até todos os indivíduos da geração anterior gerarem descendentes. Caso o número de indivíduos for ímpar, o último indivíduo funciona como seu próprio filho. Caso ocorra uma mutação, um gene aleatório é escolhido e seu valor é substituído por um outro valor válido, como visto na Figura 18.

Figura 18 - Exemplo de mutação

Fonte: O autor, 2018.

Tendo a lista completa de filhos, o próximo passo é utilizar outro parâmetro decidido pelo usuário, que decide a porcentagem de pais que se manterão na geração

seguinte. Se o parâmetro for 0.5, por exemplo, metade dos pais se manterão na próxima geração. Independentemente do valor deste parâmetro, a pontuação dos pais que permanecerão é sempre a maior possível.

Esse processo é repetido até o número de gerações atingir o limite máximo.

5 RESULTADOS

O programa pode definir a quantidade de itens dinamicamente ou criar um número fixo, dependendo dos parâmetros. Os resultados aqui obtidos são para uma quantidade fixa de itens. Assim, é possível garantir uma comparação consistente entre várias instâncias.

Conforme discutido anteriormente, a Tabela 1 apresenta a configuração dos parâmetros utilizados para criar os itens.

Tabela 1 - Configurações utilizadas para criar os itens

Item	Peso	Número mínimo	Número máximo	Multiplicador de distância	Multiplicador de distribuição	Multiplicador de sala
Berinjela	0	15	15	1.0	1.0	4.0
Faca	0	10	10	2.0	10.0	0.5
Escudo	0	5	5	5.0	10.0	0.1
Lotus	0	1	1	20.0	100.0	1.0

Fonte: O autor, 2018.

Com esses parâmetros, temos itens que vão adquirir uma pontuação maior se sua distância do caminho principal for maior ou menor, que tenderão a estar mais ou menos espalhados e que receberão uma pontuação maior ou menor dependendo da quantidade de itens desse tipo em cada sala.

Além dos itens, a classe responsável pela execução do Algoritmo Genético foi configurada com 50% de chance de uma mutação em uma geração, 20% de preservação de pais para a próxima geração, 100 gerações, 10 pontos de corte no genoma, um desvio padrão de menos de 10% em relação à geração anterior como limite crítico e 2 gerações seguidas onde ocorre esse limite crítico para aumentar a porcentagem de mutação. Cada geração tem 10 indivíduos.

A chance de mutação utilizada foi de 50% para acelerar a saída de máximos locais. Como existe uma preservação dos pais com a maior pontuação em cada geração, a pontuação nunca diminuirá. O limite crítico e o número de gerações onde a pontuação se

mantém dentro desse limite crítico para que a mutação seja 100% foram escolhidos para definir se os indivíduos estavam presos em um máximo local e tentar força-los a sair deste máximo o mais rápido possível.

Os testes a seguir foram feitos com o objetivo de mostrar as diferenças entre os métodos de avaliação disponíveis, começando considerando apenas a distância dos itens ao caminho principal no primeiro teste, adicionando a consideração do desvio padrão no segundo teste e, no terceiro teste, utilizando todos os métodos disponíveis. Assim, é possível que o leitor olhe cada uma destas etapas e decida se os métodos utilizados são satisfatórios para seus propósitos.

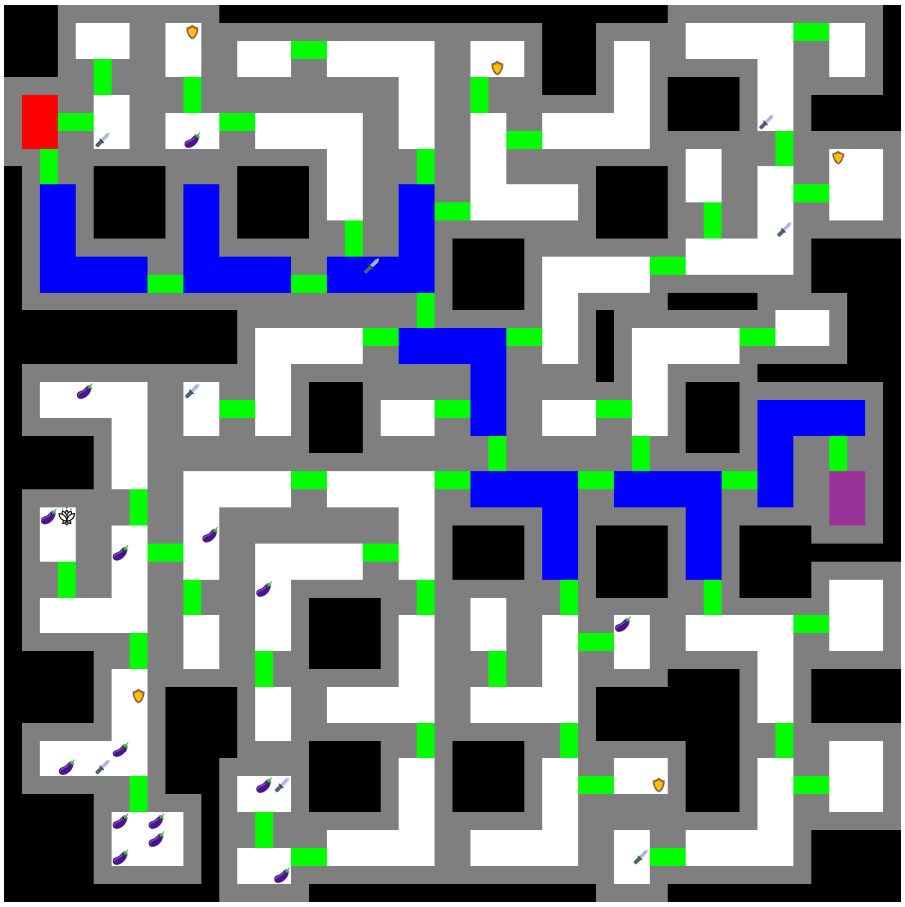
As Figuras 19 e 20 são configurações criadas apenas considerando a distância de cada item até o caminho principal.

Como o caminho principal atravessa o mapa de forma que o canto inferior esquerdo é o mais distante dele, é possível ver uma concentração grande de itens nesta localidade, já que a distribuição no mapa não está sendo levada em consideração. Além disso, os itens tendem a se concentrar próximos de becos sem saída. No geral não é satisfatório utilizar apenas esta forma de avaliação, pois ela deixa muito espaço vazio no mapa, e deseja-se uma distribuição mais uniforme.

Cada um dos métodos de avaliação foi testado pelo menos 20 vezes antes de se escolher os resultados mostrados aqui.

Figura 19 - Resultado 1: Mapa de distribuição de itens considerando apenas a distância.

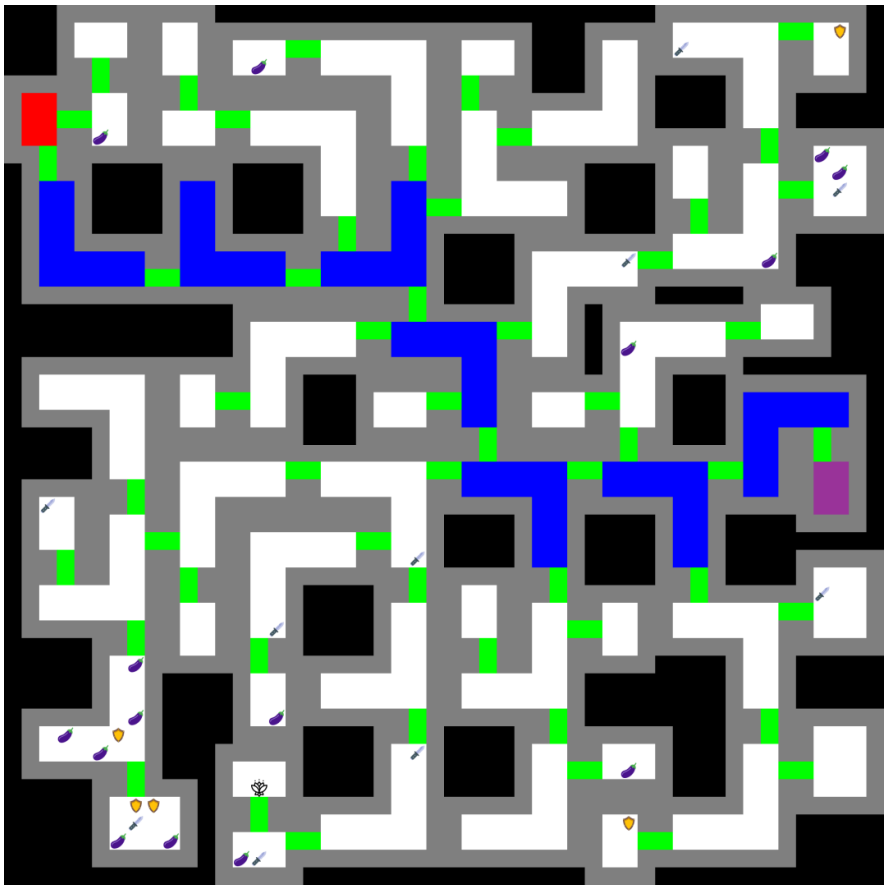
Pontuação 31,8571



Fonte: O autor, 2018.

Figura 20 - Resultado 2: Mapa de distribuição de itens considerando apenas a distância.

Pontuação 37

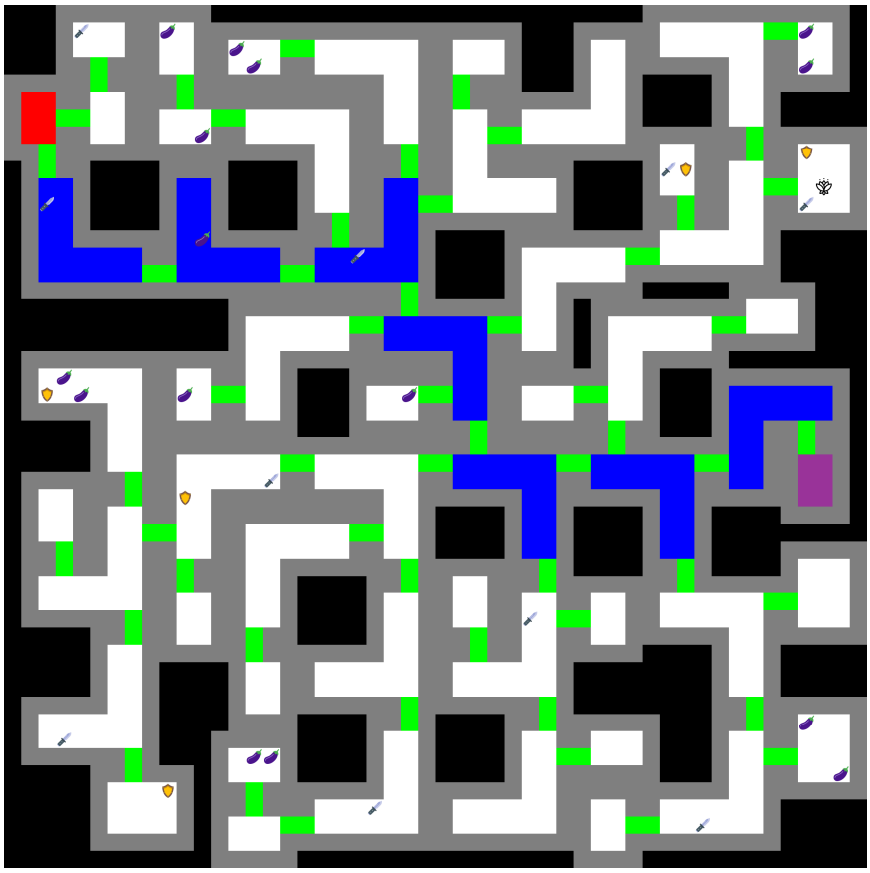


Fonte: O autor, 2018.

Para o segundo teste, foram gerados mapas que consideravam tanto a distância entre os itens e o caminho principal, como o desvio padrão para cada tipo de item, conforme mostram as Figura 21 e 22.

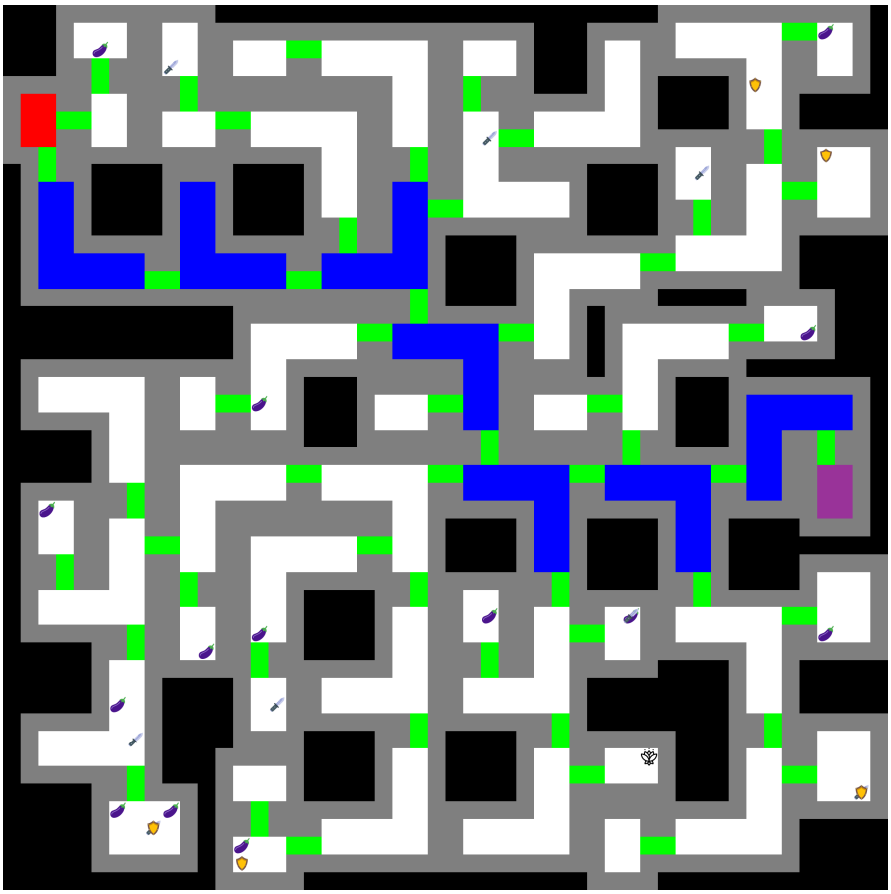
Utilizar o desvio padrão da localização dos itens junto com a distância compensa problemas de utilizar apenas a distância para a pontuação. Agora os itens se encontram mais espalhados, tendo uma tendência menor de se acumularem em cantos do mapa mas ainda tendendo a se distanciarem do caminho principal.

Figura 21 - Resultado 3: Mapa de distribuição dos itens considerando distância e desvio padrão. Pontuação 70.1607



Fonte: O autor, 2018.

Figura 22 - Resultado 3: Mapa de distribuição dos itens considerando distância e desvio padrão. Pontuação 77.7806

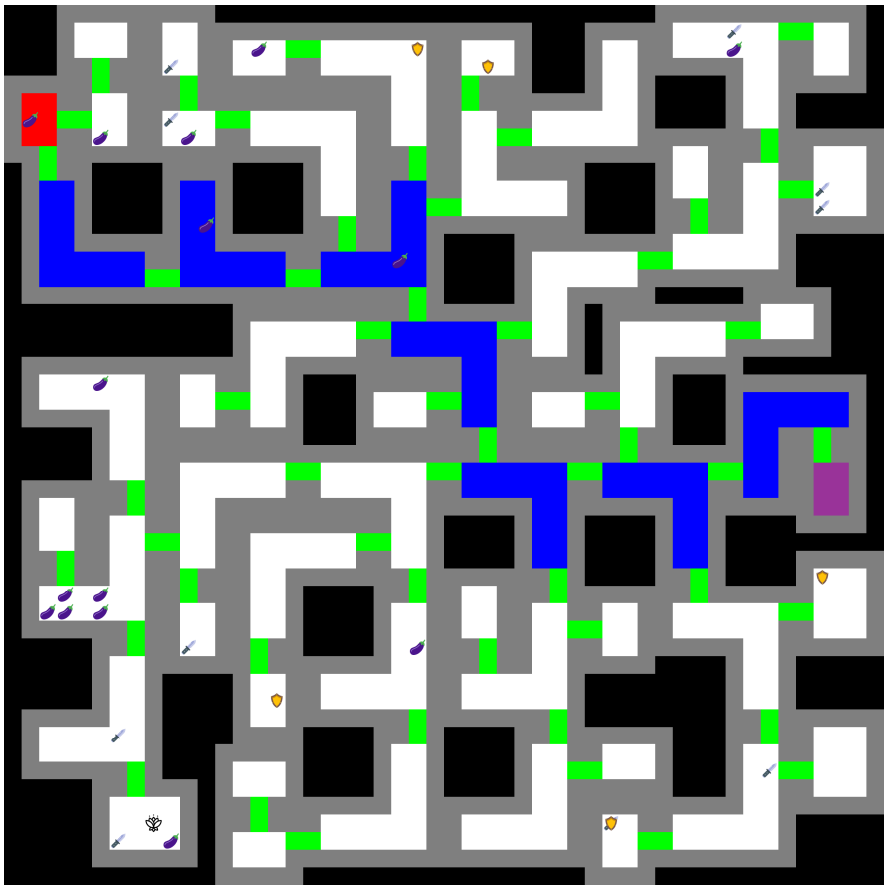


Fonte: O autor, 2018.

Para o terceiro e último teste, foram considerados todos os métodos de pontuação: distância de cada item para o caminho principal, desvio padrão para cada tipo de item e a quantidade de itens iguais na mesma sala, conforme mostram as Figuras 23 e 24.

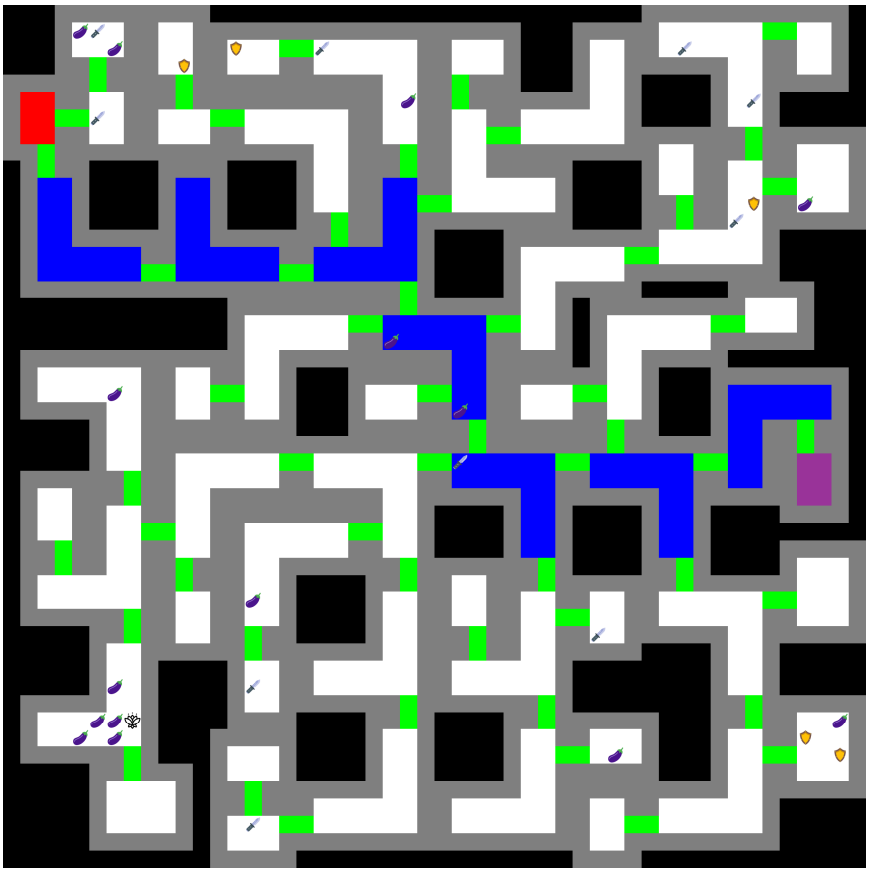
Considerando a quantidade de itens por sala na avaliação é vital para garantir que certos itens sejam povoados de forma mais completa. Deste modo, é possível fazer com que itens de um tipo sejam gerados em grupos, mas estes grupos ainda estejam espalhados pelo mapa graças ao desvio padrão, e que eles tenham tendência de estar próximos ou distantes do caminho principal, graças a avaliação da distância.

Figura 23 - Resultado 5: Utilizando todos os métodos de avaliação - Pontuação de 131.8784



Fonte: O autor, 2018.

Figura 24 - Resultado 6: Utilizando todas as formas de avaliação - Pontuação de 135.0875



Fonte: O autor, 2018.

CONCLUSÃO

Os mapas gerados com a ferramenta programada alcançam um certo equilíbrio entre ter grupos de itens em certas salas, esses grupos estarem espalhados pelo mapa e tenderem a se afastarem do caminho principal, recompensando exploração do eventual jogador. Apesar disso, não é incomum que itens que tendem a estarem agrupados sejam encontrados sozinhos ou que itens que tendem a se manter distantes do caminho principal se encontrem razoavelmente próximos do mesmo. Assim, é necessário buscar um equilíbrio entre recompensar exploração e não deixar o jogador sem itens caso ele não se afaste do caminho principal, conhecendo este caminho ou não.

Uma das dificuldades encontradas foi o balanceamento dos meios de avaliação. Como existem três pontuações que são calculadas utilizando métodos diferentes, eles não possuem pesos iguais na maioria das vezes. Para remediar isso, a pontuação máxima de cada meio de avaliação foi calculada, dividindo-se então a pontuação encontrada pela máxima e encontrando uma pontuação relativa, de 0 a 100. Porém, o desvio padrão utilizado para encontrar o quão espalhados os itens estão não é uma função linear, o que faz com que esta não seja a melhor solução. Para resolver este problema, as pontuações foram multiplicadas por números arbitrários de modo a encontrar um balanceamento razoável. O melhor jeito de resolver isso seria achar uma função para cada meio de avaliação de modo que o resultado destas seja proporcional entre elas.

Apesar de ser focada em gerar itens, a ferramenta, em seu estado atual, pode ser utilizada para gerar inimigos. Neste caso, ao utilizá-la desta maneira, a mesma não é capaz de distribuir os inimigos em relação aos itens existentes, sendo que configurações onde os inimigos estão bloqueando o caminho para itens ou protegendo eles na sala que eles estão localizados seriam apenas coincidências. Tais funcionalidades poderiam ser desenvolvidas no código atual ou em um novo programa que utiliza as localizações de itens geradas neste.

Além disso, não são consideradas formas de bloqueio de progresso, como portas trancadas com chaves e similares. Do mesmo jeito que para os inimigos, o código pode ser adaptado para considerar tais bloqueios e posicionar os meios necessários para o jogador desbloqueá-los em locais acessíveis, ou uma nova ferramenta pode ser criada para fazer isso.

Esta ferramenta poderia ser utilizada tanto por um grupo que deseja criar um jogo que gera os itens em seus mapas dinamicamente, quanto por um que quer que seus mapas tenham um equilíbrio delicado. Ao gerar múltiplos mapas, este desenvolvedor poderia escolher um que mais se aproxima de sua visão e adaptá-lo de acordo.

REFERÊNCIAS

ALLEGRO. Disponível em < http://liballeg.org/ > Acesso em: 02 setembro 2018.

CORREA, Sonia Maria Barros Barbosa. Probabilidade e estatística. 2.ed. Belo Horizonte: E.PUC Minas Virtual, 2003. 116p.

DARWIN, Charles. A Origem das Espécies. Porto: E. Lello & Irmão, 2003. 572p.

ENTER THE GUNGEON. Disponível em < https://store.steampowered.com/app/311690/Enter_the_Gungeon/ > Acesso em: 26 setembro 2018.

FW. Random level generation in Wasteland Kings. 02 abril 2013. Disponível em < http://web.archive.org/web/20160404170856/http://www.vlambeer.com/2013/04/02/random-level-generation-in-wasteland-kings/ > Acesso em: 02 setembro 2018.

HAYKIN, Simon. Redes Neurais: Princípios e Prática. 2.ed. Porto Alegre: E.Bookman, 2003. 898 p.

LACERDA, Estéfane G. M.; DE CARVALHO, André Carlos P.L.F. Sistemas inteligentes: Aplicações a Recursos Hídricos e Ciências Ambientais. [S.l.]:E.UFRGS, 1999. cap.3, p. 87-148.

LEVITIN, Anany. Introduction to the design and analysis of algorithms. 3.ed. [S.l.]: E.Pearson, 2012. 565p.

LINDEN, Ricardo. Algoritmos Genéticos – Uma importante ferramenta da inteligência computacional. 2.ed. Rio de Janeiro: E. Brasport. 2006. 428p.

LUCAS, Diogo C. Algoritmos genéticos: uma introdução. 2002. 47p. Disponível em: < http://www.inf.ufsc.br/~luis.alvares/INE5633/ApostilaAlgoritmosGeneticos.pdf >. Acesso em: 02 setembro 2018.

MARSLAND, Stephen. Machine learning: an algorithmic perspective, 2.ed. Boca Raton: E. CRC Press, 2015. 406 p.

MCMILLEN, Edmund. (insert size matters joke here). [2014] < http://bindingofisaac.com/post/90431619124/insert-size-matters-joke-here > Acesso em: 02 setembro 2018.

MIRANDA, Marcio Nunes de. Algoritmos genéticos: Fundamentos e aplicações. Disponível em: < http://www.nce.ufrj.br/GINAPE/VIDA/alggenet.htm >. Acesso em: 02 setembro 2018.

MOJAN. < https://mojang.com/2013/02/mojam-is-over-donate-download-and-play/ > Acesso em: 02 setembro 2018.

NICALIS. Disponível em < http://nicalis.com > Acesso em: 02 setembro 2018.

NUCLEAR Throne. Disponível em < https://store.steampowered.com/app/242680/Nuclear_Throne/ > Acesso em: 02 setembro 2018.

PACHECO, Marco Aurélio Cavalcanti. Algoritmos genéticos: princípios e aplicações. Disponível em: < http://www2.ica.ele.puc-rio.br/Downloads%5C38/CE-Apostila-Comp-Evol.pdf >. Acesso em: 02 setembro 2018.

PEPE, Felipe. 1975–2014: A Evolução da Indústria dos Computadores & Video Games. Disponível em < https://medium.com/@felipepepe/1975-2014-a-evolu%C3%A7%C3%A3o-da-ind%C3%BAstria-dos-computadores-video-games-ccbee83bb62b > Acesso em: 03 setembro 2018.

RODRIGUEZ, Tyrone. The Birth of Rebirth. [2014]. Disponível em < http://bindingofisaac.com/post/93169913069/the-birth-of-rebirth > Acesso em: 02 setembro 2018.

RUSSEL, Stuart; NORVIG, Peter. Artificial intelligence: a modern approach, 1.ed. Englewood Cliffs: E. Prentice-Hall, 1995. 1152p.

STANFORD, Ian; FOGED, Leif. Fast Procedural Level Population with Playability Constraints. In: AAAI CONFERENCE ON ARTIFICIAL INTELIGENCE AND INTERACTIVE DIGITAL ENTERTAINMENT, 2012, Palo Alto. Proceedings.. Palo Alto: AAAI, 2012. p. 20-25.

THE BINDING of Isaac. Disponível em < http://store.steampowered.com/app/113200/The_Binding_of_Isaac > Acesso em: 02 setembro 2018.

VLAMBEER. Disponível em < http://www.vlambeer.com/ > Acesso em: 02 setembro 2018.

APÊNDICE – Código fonte

Código fonte disponível em https://github.com/LuisVogt/Item-Location-Generator