UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Klaus Victor Kühr

O USO DE IA PARA CRIAÇÃO PROCEDURAL DE CONTEÚDO ESPACIAL EM JOGOS

Florianópolis

2018

Klaus Victor Kühr

O USO DE IA PARA CRIAÇÃO PROCEDURAL DE CONTEÚDO ESPACIAL EM JOGOS

Trabalho de Conclusão de Curso submetido ao Curso de Sistemas de Informação para a obtenção do Grau de Bacharel em Sistemas de Informação.
Orientadora: Prof$^{\underline{a}}$. Dr$^{\underline{a}}$. Jerusa Marchi

Florianópolis

2018

Klaus Victor Kühr

O USO DE IA PARA CRIAÇÃO PROCEDURAL DE CONTEÚDO ESPACIAL EM JOGOS

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de "Bacharel em Sistemas de Informação", e aprovado em sua forma final pelo Curso de Sistemas de Informação.

Florianópolis, 05 de dezembro 2018.

Prof. Cristian Koliver

Banca Examinadora:

Profª Drª. Jerusa Marchi
Presidente

Prof. Dr. Elder Rizzon Santos

Prof. Dr. Ricardo Azambuja Silveira

Dedico este trabalho aos meus pais, que me apoiaram apesar de tudo neste período de minha vida.

O maior atalho na vida é o esforço constante.

Kamokawa Genji

RESUMO

A medida que o mercado de jogos cresce, a competição entre os desenvolvedores e as exigências dos consumidores também sobe. O custo por hora de um time de desenvolvimento pode tornar um jogo proibitivamente caro para muitas empresas e as demandas dos consumidores por mais conteúdo por menos dinheiro fazem grande pressão. Para solucionar a demanda competitiva e de mercado, pode ser de interesse a automatização da produção de conteúdo. Por meio de estudos e técnicas de IA, é possível a produção procedural de ambientes de qualidade para uma grande variedade de jogos, poupando tempo de desenvolvimento ou garantindo ao jogador maior variedade de experiências. Tendo isso em mente, foi realizado um levantamento de diferentes técnicas utilizaveis na criação procedural de espaços em jogos digitais. Para demonstrar como estas podem ser colocadas em prática, foi implementado um Algoritmo Evolutivo que realiza a criação procedural de conteúdo espacial no jogo *Infinite Mario Bros*, gerando fases de qualidade e com parâmetros controláveis.
Palavras-chave: Geração Procedural de Conteúdo. Inteligência Artificial. Game Design. Level Design.

ABSTRACT

As the games industry grows, so too does the competition between developers and the expectations from the customer base. The developing costs for a team of full time developers may be exceedingly high for many start-ups and the customer's ever growing demand for high quality content for the lowest possible price put many teams on the spot. To solve this demand from both the market and competitors, it may be of interest to invest in the full or partial automation of content generation. By using Artificial Intelligence it is possible to solve some of the demands, saving developing time, increasing replay value and bringing players new, interesting and tailor made content. With this in mind, a study was done to expose these ideas to both the academic and gaming communities. In order to show how can such an algorithm may be put on a game, an Evolutionary Algorithm designed to make spatial content was implemented in the game *Infinite Mario Bros*, generating levels for it with a measure of control.
Keywords: Procedural Content Generation. Artificial Intelligence. Game Design. Level Design.

LISTA DE FIGURAS

Figura 1 *Rogue*: exemplo de uma masmorra[1] 28
Figura 2 *Search Based PCG*: Funcionamento (DAHLSKOG; TOGELIUS, 2012). 30
Figura 3 Gráfico representando a relação de formas de representação (*sequence* são sequências, *graph* são grafos e *grid* são matrizes) e técnicas para a utilização de dados(SUMMERVILLE et al., 2017). . 33
Figura 4 *Super Mario Bros*: Um dos exemplos encontrados para *Design Patterns*(DAHLSKOG; TOGELIUS, 2012) 34
Figura 5 *Super Mario Bros*: Outro exemplo de padrão encontrado. (DAHLSKOG; TOGELIUS, 2012) 35
Figura 6 *Infinite Mario Bros*: Tela inicial 38
Figura 7 *Infinite Mario Bros*: Blocos de Tijolos e Moedas 38
Figura 8 *Infinite Mario Bros*: Cogumelo Vermelho e Flor 39
Figura 9 *Infinite Mario Bros*: Formas Grande e de Fogo 39
Figura 10 *Infinite Mario Bros*: Tartaruga Vermelha, Goomba, Goomba Alado, Tartaruga Verde e Espinhosa, Flor Carnívora. 40
Figura 11 *Infinite Mario Bros*: Representação de um alelo. 46
Figura 12 Gráfico representando a variação de *fitness* ao longo de 100 gerações para uma população de 100 indivíduos. 53
Figura 13 Gráfico representando a variação de *fitness* ao longo de 500 gerações para uma população de 100 indivíduos. 54
Figura 14 Gráfico representando a variação de *fitness* ao longo de 1000 gerações para uma população de 100 indivíduos. 54
Figura 15 Gráfico representando a variação de *fitness* ao longo de 100 gerações para uma população de 500 indivíduos. 55
Figura 16 Gráfico representando a variação de *fitness* ao longo de 500 gerações para uma população de 500 indivíduos. 55
Figura 17 Gráfico representando a variação de *fitness* ao longo de 1000 gerações para uma população de 500 indivíduos. 56
Figura 18 Gráfico representando a variação de *fitness* ao longo de 100 gerações para uma população de 1000 indivíduos. 56
Figura 19 Gráfico representando a variação de *fitness* ao longo de 500 gerações para uma população de 1000 indivíduos. 57
Figura 20 Gráfico representando a variação de *fitness* ao longo de

1000 gerações para uma população de 1000 indivíduos. 57
Figura 21 Padrões repetitivos em indivíduo de *fitness* igual a 44. . 60
Figura 22 Padrões repetitivos em indivíduo de *fitness* igual a 92,50. 61

LISTA DE TABELAS

Tabela 1 Padrões presentes no jogo original do *Super Mario Bros* 45

Tabela 2 Padrões contendo inimigos capazes de locomoção criados para a implementação do algoritmo evolutivo no jogo *Infinite Mario Bros* 47

Tabela 3 Padrões contendo abismos 47

Tabela 4 Padrões contendo Vales 48

Tabela 5 Padrões formativos de outros 48

Tabela 6 Padrões contendo recompensas 49

Tabela 7 Tabela de *fitness* das combinações dos diferentes padrões, simplificada 50

Tabela 8 Tempo médio em segundos / desvio padrão para o processamento de diferentes populações e número de gerações 58

Tabela 9 *Fitness* médio / desvio padrão dos melhores indivíduos na geração de número 100, 500 e 1000 para diferentes populações . 59

LISTA DE ABREVIATURAS E SIGLAS

PCG	Procedural Content Generation	23
SBPCG	Search Based Procedural Content Generation	29
PCGML	Procedural Content Generation via Machine Learning	32
WRF	Weighted Random Forest	42

SUMÁRIO

1 INTRODUÇÃO 23
1.1 OBJETIVOS 25
1.1.1 Objetivo Geral 25
1.1.2 Objetivos Específicos 25
1.2 PROBLEMÁTICA 25
1.3 JUSTIFICATIVA 26
1.4 ESTRUTURA 26
2 FUNDAMENTAÇÃO TEÓRICA 27
2.1 INTRODUÇÃO AO *PCG* 27
2.2 TÉCNICAS APLICADAS A GERAÇÃO DE CONTEÚDO . 28
2.2.1 Baseados em algoritmos evolutivos 29
2.2.2 Baseados em Regras 31
2.2.3 Baseados em *Machine Learning* 32
2.2.4 Baseados em *Design Patterns* 34
2.3 TÉCNICAS APLICADAS A AVALIAÇÃO DE CONTEÚDO 35
2.3.1 Preditiva 36
2.3.2 Automático 36
2.4 PLATAFORMA DE TESTES 37
2.5 TRABALHOS CORRELATOS 41
3 PROPOSTA E IMPLEMENTAÇÃO 43
3.1 PREPARO DA PLATAFORMA DE TESTES 43
3.2 ALELOS 44
3.3 FUNÇÃO DE *FITNESS* 49
3.4 *CROSSOVER* E MUTAÇÃO 50
4 RESULTADOS 53
4.1 REPRESENTAÇÕES GRÁFICAS 53
4.2 TEMPO DE PROCESSAMENTO 58
4.3 ANÁLISE DOS RESULTADOS 59
5 CONCLUSÃO 63
6 TRABALHOS FUTUROS 65
REFERÊNCIAS 67
7 ANEXOS 69
7.1 CÓDIGO FONTE 69
7.2 ARTIGO 107

1 INTRODUÇÃO

Jogos acompanharam a humanidade desde o começo da sua história. Durante os séculos, seres humanos criaram maneiras de competirem entre si de maneira não violenta(TYLOR, 1879). Esportes são exemplos de jogos, assim como o xadrez, a bocha, o esconde-esconde, os videogames. Essencialmente, qualquer pessoa já jogou ou vai jogar algum jogo um dia (CRAIG, 2002).

Jogos são elementos culturais importantes que possuem elementos próprios, capazes de distinguir uns dos outros. Um jogo é composto basicamente por regras, objetivos, participantes e opcionalmente, aparatos próprios(KRAMER, 2000). O xadrez, por exemplo, possui como regras: as regras de movimento, a ordem das jogadas e as regras de xeque, cujo objetivo é a tomada da peça rei do oponente. Os participantes são dois jogadores. Os aparatos são as peças e o tabuleiro.

Com a invenção dos computadores e a eventual concepção de jogos digitais, o estudo de jogos cresceu de forma acelerada. Uma indústria bilionária formou-se ao redor dos videogames e eles tornaram-se o passatempo predileto das novas gerações. Com um mercado aquecido, desenvolvedores competem para criar os novos best sellers(WILLIAMS, 2002).

Com um custo elevado da mão de obra, muitas empresas se voltaram à técnicas de IA para gerar conteúdo para seus jogos e fornecerem mais coisas para seus clientes com um custo menor. A geração procedural de conteúdo, *PCG* (do inglês *Procedural Content Generation*), é justamente isso.

O *PCG* é uma técnica que quando usada corretamente pode facilitar o desenvolvimento de jogos ou até torná-los mais atraentes aos jogadores. Com um elemento de aleatoriedade, um jogo pode ser jogado várias vezes pela mesma pessoa e cada vez a experiência ser diferente. Por exemplo: Em um jogo de desvendar mistérios, para um jogador uma porta pode ser vermelha e aberta com uma chave triangular, encontrada na coleira de um cachorro. Para outro jogador, jogando o mesmo jogo, esta porta pode ser azul, com uma chave encontrada atrás de um quadro.

Essa variação em conteúdo pode parecer simples e não alterar muito o jogo, mas ela efetivamente personaliza a experiência de jogo para aquele jogador, não sendo possível para ele procurar dicas na internet, por exemplo.

O conteúdo possível de ser automatizado em um jogo pode ser di-

vidido em: *Game Bits, Game Space, Game Systems, Game Scenarios, Game Design* e *Derived Content.* Conforme Hendrikx et al (HENDRIKX et al., 2013), eles podem ser definidos da seguinte forma: (SMITH, 2015)

Game Bits são as menores partes atômicas de um jogo, e podem ser divididos em duas categorias: abstratos e concretos. Game Bits concretos são itens capazes de interação com o jogador, por exemplo vegetação, construções, fogo, água, nuvens e pedras. *Game Bits* abstratos, por exemplo sons, texturas e comportamentos de animais, precisam ser combinados com outros elementos para se tornarem concretos e consequentemente interagir com o jogador.

Game Space é o espaço de interação do jogador com o jogo, povoado por Game Bits. Ele também pode ser dividido em abstrato e concreto, além de suas sub definições, como espaços internos e externos, por exemplo. O tabuleiro de xadrez é um espaço abstrato, supostamente representando um campo de batalha entre dois reinados ou famílias inimigas. Espaços concretos são espaços notavelmente reconhecidos por humanos, seguindo na linha de como nós percebemos a realidade, como por exemplo uma sala com mesas e cadeiras.

Game Systems são os ecossistemas presentes em um jogo. Um ecossistema simulado, um padrão de ruas em uma cidade, o comportamento e reação de um grupo de habitantes de uma cidade ao jogador são exemplos de sistemas possíveis. A natureza automatizável destes elementos torna os reativos às escolhas do jogador. Por exemplo, se o jogador caça muitos ursos em uma floresta, cervos se reproduzem descontroladamente e a produção de ervas medicinais da vila ao lado cai, gerando inimizade. Esse tipo de interação torna o mundo do jogo um lugar vivo e "real".

Game Scenarios descrevem os eventos do jogo e sua sequência. A história do jogo, conceitos para níveis e enigmas, são exemplos de cenários de jogo. Apesar de não serem imprescindíveis a um bom jogo, geralmente bons jogos possuem boas histórias. A história dá motivação ao jogador de continuar jogando e o contextualiza, manipula e transforma o jogador em um personagem do jogo. Um bom jogo com uma boa história pode ser comparado a um bom livro ou filme.

Derived Content são conteúdos que não estão no jogo, mas são derivados dele. Uma análise do perfil dos jogadores, noticiários e comunicações diretas aos jogadores, rankings de melhores jogadores e sites de fãs são conteúdos derivados. Ou seja, sem o jogo, não existiria a possibilidade de existência de sites dedicados à ele.

Como dentro destes aspectos, as possibilidades de automação são muitas, este trabalho concentra-se na geração de Game Space com uma

ênfase em técnicas de Inteligência Artificial, ou seja, não randômicos.

1.1 OBJETIVOS

Os objetivos deste trabalho estão divididos em objetivo geral e objetivos específicos.

1.1.1 Objetivo Geral

O objetivo geral deste trabalho é abordar o tema de *Game Space PCG* e implementar a técnica de inteligência artificial de algoritmos genéticos para geração de conteúdo espacial no jogo *Infinite Mario bros*[1], a fim de comprovar sua viabilidade.

1.1.2 Objetivos Específicos

- Descrever o que é *Game Space PCG* e quais são as formas de fazer o mesmo.
- Descrever formas de avaliar o conteúdo criado automaticamente.
- Identificar técnicas de IA aplicáveis a *Game Space PCG*.
- Implementar um Algoritmo Evolutivo que realiza a criação procedural de conteúdo espacial na plataforma escolhida.
- Avaliar a técnica e os resultados obtidos

1.2 PROBLEMÁTICA

O custo para desenvolver um jogo pode ser proibitivamente elevado para equipes pequenas, sem o apoio de grandes empresas ou patrocinantes.

Um funcionário engajado em um projeto de jogo custa cerca de US$ 10.000,00 por mês[2]. Neste custo estão incluídos encargos trabalhistas, aluguel ou manutenção de espaços, custos de energia e de equipamento.

[1] Disponível em: http://www.marioai.org/MarioLevelComp2011.zip

[2] Fonte: https://kotaku.com/why-video-games-cost-so-much-to-make-1818508211, acesso em 21/06/2018

Outro ponto é que a natureza imutável de um jogo pode torná-lo repetitivo e desgastante, causando o seu abandono pelo jogador antes de seu fim ou desencorajando sessões subsequentes.

1.3 JUSTIFICATIVA

Uma maneira de aliviar a carga de trabalho e consequentemente reduzir o número de horas trabalhadas necessárias para a conclusão do projeto é a utilização de técnicas de inteligência artificial que agilizam a criação de espaços de interação com o jogador, seja de maneira supervisionada ou não.

Se realizada de maneira correta, técnicas desta natureza garantem ao jogador horas extras de jogabilidade ao variar os espaços de interação, garantindo uma maior satisfação com o produto, sem comprometer o orçamento.

Também vale ressaltar que o orçamento de um jogo é destinado em grande parte ao marketing do mesmo[3], portanto, é interessante reduzir os custos onde possível.

1.4 ESTRUTURA

O trabalho está estruturado em 5 capítulos. O primeiro sendo a introdução, além dos objetivos, problemática e justificativa. O segundo contém a fundamentação teórica do trabalho, necessária para a compreensão da proposta. Neste estão contídos os métodos de geração e avaliação de conteúdo encontrados. O terceiro capítulo trata da proposta e implementação da mesma na plataforma escolhida. O capítulo subsequênte contém os resultados encontrados pela implementação e análise dos mesmos e por último estão as conclusões do trabalho.

[3]Fonte: https://kotaku.com/how-much-does-it-cost-to-make-a-big-video-game-1501413649, acesso em 21/06/2018

2 FUNDAMENTAÇÃO TEÓRICA

A fundamentação teórica foi dividida em 5 partes. A primeira parte é o histórico do *PCG*, seguido por algoritmos de geração de conteúdo, algoritmos de avaliação de conteúdo, a apresentação da plataforma de testes e trabalhos correlatos.

2.1 INTRODUÇÃO AO *PCG*

PCG pode ser definido como o uso de um algoritmo formal para gerar um conteúdo que seria produzido normalmente por um ser humano (SMITH, 2015). No campo de estudo de jogos, a pesquisa de *PCG* é uma subdivisão do campo de estudos de inteligência artificial em jogos, visto primariamente como uma solução para o problema de replicar a criatividade e visão de um designer humano, como avaliar a qualidade de um conteúdo existente e como criar sistemas capazes de interagir com um designer de jogos, auxiliando-o.

Apesar de ser uma ferramenta utilizada na indústria de jogos digitais, suas raízes estão em jogos de tabuleiro e papel e caneta. Um dos primeiros surgimentos de *PCG* data de 1976, em um suplemento de *Dungeons & Dragons* (GYGAX; ARNESON, 1974), que é um jogo de interpretação de papéis. No jogo, os jogadores são aventureiros em um mundo medieval fantástico, povoado por dragões, intrigas, calabouços, monstros e masmorras.

Um aspecto importante de um jogo de *Dungeons & Dragons* é a exploração de lugares perigosos, normalmente subterrâneos. Esses lugares são montados por um dos jogadores, que é o narrador da história. Esse narrador define o espaço de jogo e o que os jogadores irão encontrar, possíveis passagens secretas, monstros e tesouros. É um processo que pode ser demorado e muitas vezes feito em vão, se os jogadores nunca chegarem a entrar na caverna em primeiro lugar.

O suplemento *Dungeons Geomorphs*(TSR, 1976) buscava ajudar o narrador a montar esses lugares de maneira simplificada. Ele dispunha de pedaços de mapas, que poderiam ser embaralhados pelo narrador e expostos aos jogadores à medida que eles exploravam o ambiente. O suplemento informava ao narrador como fazer o uso correto do mesmo, e com essa informação o narrador conseguiria gerar um número quase infinito de masmorras diferentes.

Próximas versões do jogo *Dungeons & Dragons* trouxeram con-

sigo regras que poderiam ser utilizadas pelos narradores menos criativos para montar, com base em resultados de rolamento de dados multifacetados, não apenas masmorras, como também os monstros que as habitam e tesouros contidos nas mesmas.

O avanço da tecnologia e os computadores pessoais trouxe o *PCG* para o meio digital. Alguns dos primeiros jogos procedurais foram jogos de aventura e ação inspirados nas regras procedurais de *Dungeons & Dragons*. *Rogue* é um dos jogos mais memoráveis desta era, gerando o termo em inglês *roguelike* (similar ao *Rogue*), que até hoje é utilizado na indústria para categorizar jogos com conteúdo gerado proceduralmente, diferenças entre cada partida e sem a capacidade de “voltar” uma jogada para impedir a morte do personagem.

A Figura 1 é uma tela de *Rogue*, mostrando uma masmorra criada proceduralmente, de acordo com as regras implementadas pelos desenvolvedores.

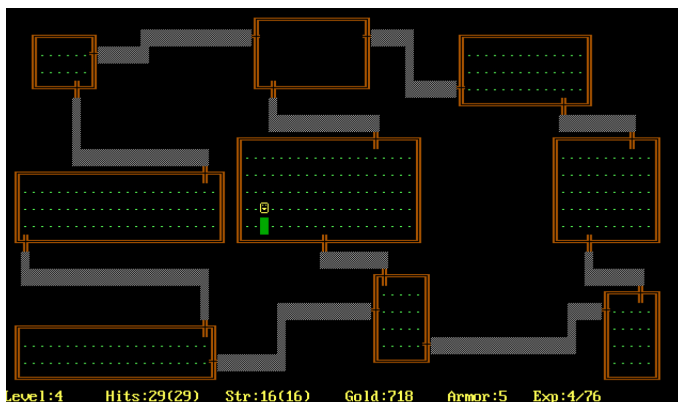

Figura 1 – *Rogue*: exemplo de uma masmorra[1]

2.2 TÉCNICAS APLICADAS A GERAÇÃO DE CONTEÚDO

Apesar de ser um artefato relativamente antigo para a formulação de conteúdo em jogos, o estudo acadêmico de *PCG* é mais recente do que o esperado. Muitas implementações anteriores eram puramente

[1] Fonte: https://commons.wikimedia.org/w/index.php?curid=36978065. Acesso em: 20/06/2018

aleatórias, sem uma intenção definida. Com a diversificação das técnicas de IA, pesquisadores foram capazes de encontrar soluções para problemas antigos de *PCG*. Algumas das soluções para o *PCG* foram algoritmos evolutivos, baseados em regras, *Design Patterns* e *Machine Learning*, os quais serão apresentados neste trabalho.

Um desafio do campo é a avaliação do resultado gerado por *PCG*. Para que seja executado um algoritmo evolutivo, por exemplo, é necessário que haja uma função de fitness capaz de avaliar os resultados de cada geração. O resultado do processo de *PCG* pode possuir a obrigação ou não de ser "completável", sendo assim, a importância da avaliação de possibilidade de completude e qualidade do resultado é variável (TOGELIUS et al., 2011b).

2.2.1 Baseados em algoritmos evolutivos

Algoritmos baseados em algoritmos evolutivos comumente utilizados em *PCG*, também chamados de baseados em Buscas (*SBPCG*) neste contexto, foram objetos de estudo por Togelius et al. (TOGELIUS et al., 2011b). Eles são compostos por populações, indivíduos e seus genótipos e fenótipos, além de uma função de avaliação. Sua maneira de formulação de *Game Space* é por meio de geração de indivíduos e avaliação do conteúdo gerado, até que seja satisfatório. A Figura 2 representa em sua seção superior o funcionamento do *SBPCG*. O motivo de não utilizar o nome "evolutivo" na literatura para este método é para evitar a exclusão de meta-heurísticas comuns, como *Simulated Annealing* e Otimização por enxame de partículas, por exemplo.

Inicialmente, genes são mapeados às expressões dos mesmos. Os genes, que podem sofrer mutações, podem ser mapeados diretamente ou indiretamente aos fenótipos. Um exemplo de mapeamentos são, do mais direto para o mais indireto:

1. Uma célula de uma masmorra, podendo ser uma parede ou espaço vazio.

2. Uma parede inteira, tanto em largura e altura.

3. Um padrão utilizável em uma sala, como uma janela, cortinas e paredes ao redor da mesma.

4. Uma lista de propriedades desejáveis de uma sala, como número de portas, janelas, tapetes e inimigos.

5. Um número aleatório.

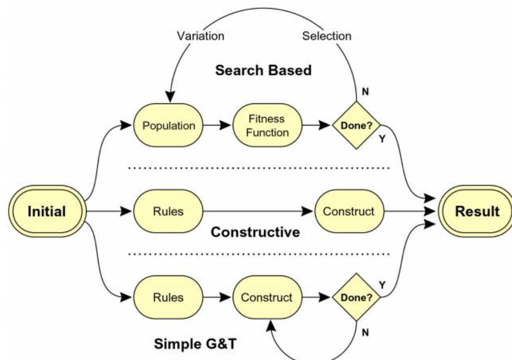

Figura 2 – *Search Based PCG*: Funcionamento (DAHLSKOG; TOGELIUS, 2012).

Uma propriedade que deve ser levada em consideração na escolha do mapeamento é a localidade. A localidade de um gene é capaz de informar o tamanho da mudança que a alteração em um gene apenas é capaz de afetar a fase como um todo. Quanto mais direto o mapeamento, maior será a sua localidade. Por exemplo, a alteração de uma parede lisa por uma parede com uma fonte de iluminação não altera a fase drasticamente, porém uma mudança no número de salas e inimigos alteraria a experiência do jogador.

Outra propriedade que deve ser considerada é a dimensão da fase. Se uma fase for composta por muitos genes, é possível que o cálculo da geração da mesma tome um tempo muito elevado, inviabilizando o processo em tempo de execução ou em desenvolvimento. Por exemplo, se for escolhido um algoritmo de alta localidade e mapeamento direto em que cada gene é uma célula, se uma fase for composta por uma matriz de 1000x1000 células, a dimensionalidade pode causar a inviabilização do algoritmo.

Indivíduos, por sua vez, são compostos por genes. A cada geração, indivíduos são gerados com base na população e suas aptidões são calculadas. Indivíduos com aptidões altas são selecionados para a geração da próxima geração, por meio de cruzamentos com outros indivíduos e mutações aleatórias de genes, assim sucessivamente até que a fitness máxima ou número máximo de gerações seja excedido.

Shaker et al.(SHAKER et al., 2012) apresentaram um modelo de geração de conteúdo adaptativo ao estilo de jogo de cada jogador, uti-

lizando um algoritmo “Gramatical Evolutivo”. A plataforma escolhida por eles para testes foi o *Infinite Mario Bros*(PERSSON, 2008), uma cópia em código aberto do jogo *Super Mario Bros.* O algoritmo gramatical evolutivo descrito utilizou como “indivíduo” da população uma fase composta por genes equivalentes ao número 2 da lista de exemplos descrita nesta subseção, onde um gene contém um atributo de sua posição na fase, sua natureza e sua largura. A função de fitness utilizada na avaliação dos genes foi feita com base em trabalhos anteriores, nos quais foram deduzidas fórmulas para o cálculo de emoções prováveis sentidas pelos jogadores com base nos elementos das fases.

2.2.2 Baseados em Regras

A geração de conteúdo baseada em regras pode ser vista como uma resposta alternativa ao comportamento do jogador. No percurso de uma fase uma sequência de desafios é proposta perante o jogador, que responde aos desafios da maneira que considera mais apropriada. A geração de conteúdo com base em regras toma as ações do jogador como base para geração de conteúdo.

Por exemplo, se no decorrer de uma fase o jogador tende a desviar dos inimigos e o gerador toma os movimentos do jogador como entrada, ele poderá na próxima fase gerada criar caminhos alternativos para o jogador, ou reduzir os mesmos, forçando o confronto.

Togelius et al.(TOGELIUS et al., 2011a) implementaram um sistema de geração de fases em *Super Mario Bros* baseado em regras e com duas versões, uma offline e uma online. As entradas dos sistemas são as ações do jogador. Por exemplo quando ele aperta o botão de pular ou deixa de apertá-lo. Essas ações são gravadas numa linha de tempo e posição e com base nelas o gerador altera a próxima fase, colocando mais ou menos inimigos e alterando a posição de blocos e espaços vazios.

No método apresentado, o modo offline apresentava a geração de conteúdo apenas na próxima fase, enquanto o modo online além das alterações na próxima fase, alterava a fase atual, criando monstros em resposta à coleta de moedas e moedas em resposta à derrota de monstros.

2.2.3 Baseados em *Machine Learning*

O reaparecimento de redes neurais sob a alcunha de *Deep Learning* e a utilização de *Big Data* para treinamento das mesmas serviu para a cogitação e implementação de técnicas de geração de conteúdo espacial em jogos. Dos usos possíveis para *PCG*, a abordagem baseada em *Machine Learning* (*PCGML*) demonstra excelência atualmente, dentre outras áreas, na Geração Automática e Co-Autoria de Conteúdo.

No caso da Geração Automática (não supervisionada), diferentemente de outros métodos de *PCG*, como o *SBPCG*, o desenvolvedor não precisa codificar extensivamente a natureza do problema por meio de fórmulas de fitness, por exemplo. Basta ele treinar o algoritmo com exemplos do que ele deseja e o algoritmo será capaz de gerar conteúdo.

Para a Co-Autoria de Conteúdo, é possível o desenvolvedor utilizar ferramentas de machine learning para completar seções do jogo, agilizando o processo de desenvolvimento. Por exemplo uma fase pode ter seu início, fim e trechos feitos pelo desenvolvedor, com as áreas deixadas em branco preenchidas pelo algoritmo. Uma vez preenchidas essas áreas, o desenvolvedor avalia se foi aceitável ou não.

Summervile et al. (SUMMERVILLE et al., 2017) organizaram as técnicas de *PCGML* com base em duas características: A representação dos dados e técnica de treinamento. As representações possíveis são:

- Sequências: Os dados são sequências de caracteres, como textos ou fases de jogos simples.
- Matrizes: Dados são representados como informações em matrizes. Geralmente sendo matrizes de 2 dimensões, capazes de representar a posição de cada elemento nela contido. Muito utilizado para representação de Fases.
- Grafos: Representações gerais de dados. Essas representações são genéricas, porém normalmente custosas pela falta de estruturas intrinsecamente definidas.

As diferentes técnicas de treinamento dependem da representação para que sejam efetivas ou utilizáveis. As técnicas levadas em consideração foram:*Backpropagation*, *Evolution*, *Frequency Counting*, *Expectation Maximization*, e *Matrix Factorization.*

A Figura 3 demonstra os resultados encontrados por Summerville *et al.* (SUMMERVILLE et al., 2017). Os resultados estão anotados na seguinte forma: Circulos vermelhos são exemplos de jogos de plataforma, Quadrados Laranjas são masmorras, O 'X' Azul é um jogo

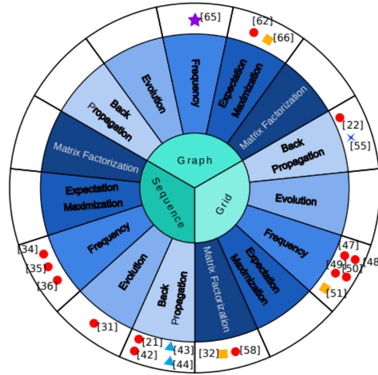

Figura 3 – Gráfico representando a relação de formas de representação (*sequence* são sequências, *graph* são grafos e *grid* são matrizes) e técnicas para a utilização de dados(SUMMERVILLE et al., 2017).

de estratégia em tempo real, Triangulos Azuis são jogos de cartas e a Estrela roxa é um jogo de ficção interativa.

Dahlskog et al.(DAHLSKOG; TOGELIUS; NELSON, 2014) dividiram fases do clássico *Super Mario Bros*(NINTENDO, 1985) em fatias e transformaram-nas em sequências de treinamento para cadeias de Markov em n-gramos [2]. Um n-gramo é uma sequência de valores de tamanho ‘n’, onde cada posição é predita com base nos ‘n’ valores anteriores à mesma.

Para o teste dos resultados, foram feitas fases onde o ‘n’ variava de 0 até 3. Foi observado que quando n=0, as fases eram sequências aleatórias de fatias, muitas vezes impossíveis de serem completadas pelo jogador e a medida que ‘n’ era aumentado, mais atraente e jogável as fases se tornavam. Sequências também foram utilizadas com o *Backpropagation*, para a representação de dados em uma *Long Short-Term Memory Recurrent Neural Network*, ou seja, Rede Neural Recorrente capaz de armazenamento de estados passados.

[2]Mais informações: https://sookocheff.com/post/nlp/ngram-modeling-with-markov-chains/

2.2.4 Baseados em *Design Patterns*

Design Patterns são maneiras de estruturar o *design* e o processo do mesmo em elementos recorrentes (DAHLSKOG; TOGELIUS, 2012). Originado na década de 1970 no campo da Arquitetura, eles foram definidos como problemas que o arquiteto poderia encontrar em seus projetos e como solucioná-los.

Um exemplo simples para um *Design Pattern* é: Em uma festa não há comida suficiente para todos os convidados. A solução é tornar a relação comida / convidado maior. Uma resolução que pode ser tomada é aumentar a quantidade de comida, outra é reduzir o número de convidados.

A ideia era a resolução rápida de problemas, possibilitando que o arquiteto se concentre em outros aspectos do processo de *design*. Eventualmente esta solução foi adaptada para soluções no campo de desenvolvimento de *Software*(DAHLSKOG; TOGELIUS, 2013).

O *design* de jogos não ignorou os *Design Patterns*, adaptando-os ao seu campo. Em jogos, *Design Patterns* podem ser vistos como respostas para problemas do jogo ou do *designer* de fases, assim como um problema a ser resolvido pelo jogador e proposto pelo *designer*.

Figura 4 – *Super Mario Bros*: Um dos exemplos encontrados para *Design Patterns*(DAHLSKOG; TOGELIUS, 2012)

As aplicações de *design patterns* em *Game Space PCG* são duas: Como blocos de conteúdo que podem ser alinhados em uma, duas ou mais dimensões, gerando o conteúdo a ser jogado. A outra aplicação é como avaliação de conteúdo gerado por outros métodos, como o baseado em buscas.

Smith et al. (SMITH; CHA; WHITEHEAD, 2008) propuseram um

método de análise de "ritmo" em fases de jogos 2D. Dividindo cada fase em seções compostas por inimigos, desafios e recompensas. Cada uma dessas subdivisões da fase pode então ser analisada e transformada em um design pattern. Dahlskog e Togelius (DAHLSKOG; TOGELIUS, 2012) analisaram o jogo *Super Mario Bros* com base no método proposto por Smith et al. (SMITH; CHA; WHITEHEAD, 2008) e encontraram 23 padrões que se repetiam nas diversas fases, como os padrões representados nas Figuras 4 e 5. Um gerador de fases então escolhia estes 23 padrões aleatoriamente.

Figura 5 – *Super Mario Bros*: Outro exemplo de padrão encontrado. (DAHLSKOG; TOGELIUS, 2012)

Em um trabalho posterior (DAHLSKOG; TOGELIUS, 2013), Dahlskog e Togelius aliaram aos *Design Pattern* encontrados uma função de *fitness* e um algoritmo evolutivo novos, gerando fases mais divertidas e desafiadoras aos jogadores do que o gerador original, também baseado em buscas.

2.3 TÉCNICAS APLICADAS A AVALIAÇÃO DE CONTEÚDO

Um dos problemas do campo de *PCG* é a avaliação do conteúdo gerado. Para um desenvolvedor ou *designer*, pode ser fácil reconhecer se uma fase é esteticamente agradável e se ela é possível de ser completada. Para que seja possível a avaliação do mesmo conteúdo de maneira automática, no entanto, é necessário codificar o reconhecimento da qualidade da fase. Foram encontradas duas maneiras de se abordar este problema: uma tenta prever a emoção que o jogador terá ao jogar a fase e outra avalia se a fase é interessante ou não.

2.3.1 Preditiva

Pedersen *et al.*(PEDERSEN; TOGELIUS; YANNAKAKIS, 2009) em seu trabalho buscaram quantificar a experiência do jogador em três parâmetros: Diversão, Desafio e Frustração. Com base na experiência quantificada, eles buscaram criar um método capaz de predizer qual foi a experiência de um jogador sem que seja necessário perguntar-lhe de modo explícito. A plataforma de testes escolhida pelos mesmos foi o *Infinite Mario Bros*, já citado anteriormente neste trabalho. A quantificação da experiência é feita com base na coleta das ações do jogador, como o número de vezes que ele saltou, o número de inimigos derrotados, moedas coletadas e vidas gastas, além de um questionário.

A coleta de dados para a validação do trabalho se deu por meio de uma página Web contendo o jogo. Jogadores eram expostos a quatro fases, sendo que após a segunda e quarta fase, o questionário era exibido. O questionário em questão pedia ao jogador que compare as fases jogadas. As perguntas utilizadas foram de múltipla escolha com 4 alternativas, sendo as seguintes:

1. A fase 'A' era/pareceu mais 'E' do que a fase 'B';
2. Ambas as fases foram/pareceram igualmente 'E';
3. Nenhuma das fases foi/pareceu 'E'.

Onde 'E' é a emoção indagada: divertida, desafiadora, entediante, predizível e angustiante.

O número total de jogadores que participaram da pesquisa foram 181, sendo que o mínimo necessário para a pesquisa foram 120, onde seriam utilizados 240 questionários e 480 fases. A análise em foco foi sobre as emoções anteriormente citadas: Diversão, Desafio e Frustração. Utilizando 3 algoritmos preditivos diferentes, com dados de entrada as ações dos jogadores, as fases e os questionários, eles foram capazes de prever a Diversão com 67.92%, o Desafio com 77.77% e Frustração com 88.66% de acurácia.

2.3.2 Automático

Liapis *et al.*(LIAPIS; YANNAKAKIS; TOGELIUS, 2014) apresentaram um método genérico e automatizável de determinar a qualidade de fases, independentemente da maneira que ela foi feita e do jogo em

questão. A avaliação é feita sobre rascunhos do mapa da fase, ou seja, abstrações do espaço de jogo, e pela aplicação de fórmulas matemáticas.

A base para a avaliação são os design patterns em jogos, previamente apresentados. Uma fase pode conter centenas de padrões, porém os padrões em questão são: Simetria, Controle de Área e Exploração. Neste contexto, Simetria é a oportunidade que jogadores diferentes possuam a mesma chance de vitória; Controle de Área dá acesso ao jogador a ações anteriormente impossíveis; e Exploração é o objetivo da compreensão do mapa ou a descoberta de localizações específicas. Sendo que para serem relevantes, tanto o Controle de Área quanto a Exploração necessitam de referências espaciais dentro do mapa.

Minimamente intrusivo para a implementação pelo desenvolvedor, este algoritmo demonstra excelência na análise de fases de jogos de estratégia, mas sua utilidade em outros gêneros, como jogos de tiros ou até mesmo Rogue previamente citado, não deve ser ignorada. A aplicação desta técnica possibilita ao designer da fase em questão avaliar a qualidade da mesma em tempo de desenvolvimento ou como função de fitness para algoritmos genéticos e controle de qualidade em outras técnicas.

2.4 PLATAFORMA DE TESTES

Para a execução do trabalho proposto, a plataforma de testes escolhida foi o *Infinite Mario Bros*(PERSSON, 2008), um clone em código aberto escrito na linguagem *java* do clássico da Nintendo, *Super Mario Bros*(NINTENDO, 1985). Esta cópia foi utilizada por muitos anos como plataforma para competições de IA em jogos, tanto em competições para geração procedural de conteúdo quanto para competições de técnicas de IA para jogar o jogo, além de um grande número de artigos e referências deste trabalho.

A Figura 6 representa a tela inicial doo jogo escolhido, mostrando as informações disponíveis ao jogador. O jogador começa no canto esquerdo da fase, tendo o objetivo de alcançar o canto direito. Uma seta vermelha aponta a direção a ser seguida. Cada fase possui um tempo limite de 200 segundos, pouco mais do que 3 minutos, demonstrados sob a palavra *TIME* (tempo, em inglês), no canto superior direito da tela. O número de vidas disponíveis ao jogador estão exibidos no canto superior esquerdo, depois da palavra *MARIO*. O jogador começa com 3 vidas e quando ele perder sua última ou ele alcançar o fim da fase, o jogo acaba.Vidas podem ser perdidas nas seguintes situações: Caindo

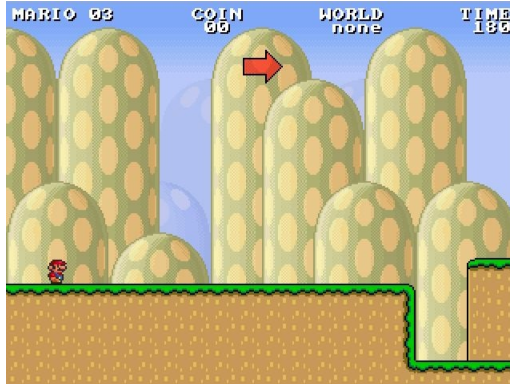

Figura 6 – *Infinite Mario Bros*: Tela inicial

em um buraco e encostando em um inimigo, exceto quando em um ataque, enquanto na forma pequena do Personagem.

Figura 7 – *Infinite Mario Bros*: Blocos de Tijolos e Moedas

O personagem começa o jogo em sua forma pequena. Nesta forma, ele não é capaz de quebrar blocos, como os exibidos na Figura 7. Se o jogador entrar em contato com um "Cogumelo Vermelho", ele cresce para a sua forma grande. Em sua forma grande, ele é capaz de destruir blocos de tijolo e caso for tocado por um inimigo, reverte a forma pequena. Caso o Personagem entre em contato com uma "Flor", sua forma grande se altera para a forma de "Fogo".

A Figura 8 representa tanto o "Cogumelo Vermelho"quanto as

"Flores"citadas anteriormente. Ambos emergem de blocos com interrogações, sendo que a "Flor"é estática e o "Cogumelo"desliza imediatamente para a direita, sendo possível este se perder em buracos na fase. Nesta forma, seu tamanho permanece o mesmo, mas suas roupas vermelhas trocam para a cor branca e sua calça azul torna-se vermelha. Nesta forma, ele é capaz de soltar bolas de fogo. Ambas as formas estão representadas na Figura 9. Caso ele seja tocado por um inimigo nesta forma, ele reverte para a forma Grande.

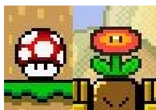

Figura 8 – *Infinite Mario Bros*: Cogumelo Vermelho e Flor

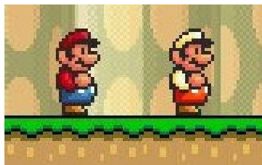

Figura 9 – *Infinite Mario Bros*: Formas Grande e de Fogo

Os inimigos presentes no jogo estão representados na Figura 10 e são: Goombas, Tartarugas Vermelhas e Verdes, Tartarugas Espinhosas e Flores Carnivoras, além de balas de canhões. Goombas e Tartarugas ainda possuem uma variação com asas, capaz de pular. Cada inimigo possui seus comportamentos e naturezas únicos. Por exemplo, tartarugas vermelhas não caem em buracos, enquanto Goombas e Verdes, sim. As tartarugas Vermelhas e verdes entram em seus cascos se forem pisadas pelo jogador. Este casco pode ser carregado pelo jogador e arremessado em outros inimigos.

Em qualquer uma das formas, se o Personagem por algum motivo pisar em um inimigo que não seja uma Tartaruga Espinhosa ou uma Flor Carnívora, o inimigo é derrotado.

Fases possuem tamanhos variados, que podem ser controlados através de parametros de inicialização. Cada fase no gerador original possui de 2 até 11 blocos como zona inicial, onde o Jogador começa e o mesmo para sua zona final, onde a fase termina. O tamanho padrão

Figura 10 – *Infinite Mario Bros*: Tartaruga Vermelha, Goomba, Goomba Alado, Tartaruga Verde e Espinhosa, Flor Carnívora.

da fase são 320 blocos de largura e 15 de altura.

2.5 TRABALHOS CORRELATOS

Shi e Chen (SHI; CHEN, 2016) elaboraram, com base em algoritmos baseados em regras e Machine Learning, um método novo de geração de conteúdo que eles nomearam Primitivas de Construção (do Inglês: Constructive Primitives, abreviado para CPs). Neste método inovador, regras fáceis de serem feitas são utilizadas para remover conteúdo ruim das fases, com o auxílio subsequente de uma análise de qualidade baseado em dados tangíveis, utilizando Machine Learning.

A plataforma de testes utilizada foi o Super Mario Bros e eles buscaram resolver alguns problemas inerentes ao PCG, como fases impossíveis de serem completadas, fases de aspecto feio, curva de dificuldade repentina e conteúdo inalcançável. Outro desafio era como coordenar e controlar elementos da fase, como inimigos, e propriedades da mesma, como a sua linearidade e dificuldade.

Algumas das soluções para os problemas são as regras aplicadas à fase. Por exemplo, por meio de regras um desenvolvedor pode impedir que hajam paredes de blocos bloqueando completamente a passagem do jogador ou prevenir que elementos do mapa interfiram uns com os outros. Para resolver outros problemas, como o estético, eles focaram em pedaços menores do mapa, dividindo o mesmo em segmentos de igual tamanho, estes definidos e referenciados como blocos. Um rígido controle de qualidade automático posterior garante que a fase apresentada ao jogador seja atraente.

Shi e Chen observaram 85 elementos codificáveis de um bloco de Super Mario Bros. Entre eles estão os números máximos de: abismos, elevações, canos, canhões, caixas, moedas e inimigos presentes por bloco. Estes números máximos foram então definidos pelos pesquisadores de modo que foi formado um espaço amostral de blocos composto por $9.72 * 10^{37}$ blocos possíveis.

Como outro elemento controlável são as posições dos elementos das fases, regras foram utilizadas para remover quaisquer blocos onde elementos, exceto as elevações, entram em conflito de posição uns com os outros ou blocos impossíveis de serem completos, por exemplo com uma parede maior do que o alcance do pulo do jogador.

Uma vez que ocorreu a filtragem por meio de regras do espaço amostral, foi necessário treinar o sistema a reconhecer quais blocos são atraentes e interessantes e quais não são. Os blocos foram quantificados em objetos com 85 atributos e então 19.000 deles foram selecionados aleatoriamente. Essa seleção aleatória passou por um processo de Clusterização, gerando 106 clusters.

Foram então coletados 800 blocos aleatoriamente para a formulação do conjunto de testes, sendo que a cada iteração do algoritmo 100 deles são escolhidos e cada um destes analisado pelos pesquisadores, avaliando-os como blocos de alta qualidade ou baixa qualidade. Foi utilizado para classificação o algoritmo *weighted random forests* (*WRFs*) , principalmente pela sua característica de análise de custo, penalizando fortemente erros "Falsos Positivos", onde um segmento de baixa qualidade é predito como um de alta qualidade.

Foi observado que após o treinamento, a taxa de "Falsos Negativos", ou seja, onde blocos de alta qualidade foram preditos como de baixa qualidade ficou em 19.66%, enquanto "Falsos Positivos", cuja presença é muito mais danosa, ficou 3.69%, destes, 0.74% possuíam elementos inalcançáveis, mas todos eram completáveis.

O algoritmo de classificação então percorreu todo o dataset de 19.000 blocos, removendo todos os blocos de baixa qualidade encontrados. O resultado do trabalho foi um gerador de fases de altíssima qualidade, validado pelo classificador automático proposto por Liapis et al.(LIAPIS; YANNAKAKIS; TOGELIUS, 2014) em comparação com outros algoritmos de geração de fases em Super Mario Bros propostos e implementados por outros.

Summerville et al.(SUMMERVILLE; MATEAS, 2016) em um trabalho recente, criaram maneiras de definir fases da plataforma de teste como Strings, sequências de caracteres. Com essas sequências, eles foram capazes de utilizar redes neurais recorrentes para a geração de fases novas. O dataset utilizado foram as fases originais do jogo. O algoritmo utilizou também a sequência provável de posições do jogador para a elaboração da fase, tendo uma porcentagem muito maior de fases capazes de serem finalizadas. Um dos grandes problemas observados neste caso foi a escassez de material para treinamento do modelo, uma vez que foram utilizados apenas as fases originais do jogo.

3 PROPOSTA E IMPLEMENTAÇÃO

Para mostrar como pode ser feito o uso de *Game Space PCG* de modo prático, é proposta a implementação do mesmo com um algoritmo evolutivo especializado no jogo *Infinite Mario Bros.* O indivíduo será composto por um genoma contendo *Design Patterns*, facilitando a avaliação. A função de *fitness* será implementada com base na compatibilidade de diferentes padrões entre si.

A implementação do trabalho foi feita utilizando o *NetBeans IDE* 8.2[1], utilizando a linguagem *Java*. A versão de *Java* utilizada foi a 1.8.0_131.

3.1 PREPARO DA PLATAFORMA DE TESTES

Para a implementação do algoritmo evolutivo na plataforma, foi necessário realizar alguns procedimentos iniciais. Foram eles:

1. Compreender a representação de uma fase, assim como seus elementos;

2. Estipular os limites da fase;

3. Analisar o comportamento do jogo em relação aos inimigos;

4. Descobrir como ocorre a definição da posição inicial de um inimigo;

5. Descobrir com quais métodos o jogo realiza a chamada para o gerador de fases;

A análise destes se deu por meio de estudo do código fonte. O resultado das analises preliminares foram, respectivamente:

1. A fase é representada por três elementos principais, uma matriz de *Bytes* e uma matriz de iguais dimensões contendo a posição inicial e tipo de cada inimigo na mesma e a localização da saída da fase. Isto foi observado na classe *Level* do jogo;

[1] Build: 201609300101. Disponível em: https://download.netbeans.org/netbeans/8.2/final/zip/netbeans-8.2-201609300101.zip

2. O limite da fase é principalmente em altura, uma vez que não é possível inserir blocos acima da altura máxima. Sua altura máxima é por padrão 15, o motivo do mesmo é o limite do tamanho da tela do jogo, cuja câmera não se movimenta no eixo vertical. Isto foi observado nas classes *Level* e *LevelRenderer* do jogo;

3. Inimigos possuem números máximos controláveis, na versão base do jogo, o mesmo é de 10.000 inimigos. É possível criar um inimigo em qualquer posição no mapa que ele irá agir de acordo com suas restrições, mesmo que não faça sentido que ele esteja ali. Por exemplo, uma planta carnívora pode ser colocada no chão, sem ser em um cano. Isto foi observado na classe *SpriteTemplate* do jogo;

4. A definição da posição inicial do inimigo é pela inserção da mesma, juntamente com o tipo de inimigo e se ele possúi asas em uma posição na matriz de representação da fase. Isto foi observado na classe *SpriteTemplate* do jogo;

5. A chamada para o gerador de fase é feita após a chamada para geração da janela do jogo. Isto foi observado na classe *MarioComponent* e no método *init* da classe *LevelSceneTest* do jogo.

Tendo em vista os três elementos principais de uma fase na plataforma de testes, a matriz de *bytes*, a matriz de posição inimigos e a posição da saída, foi criada uma classe "Fase" que estende a classe original (*Level*). Esta classe contém além dos elementos originais um *ArrayList* de números inteiros contendo seu genoma e um objeto "Controlador", que armazena todos os padrões dos alelos e realiza operações sobre eles.

3.2 ALELOS

Os alelos utilizados são derivados de um trabalho de 2012 por Dahlskog *et al* (DAHLSKOG; TOGELIUS, 2012). Foram observados neste trabalho 23 padrões presentes no jogo original do *Super Mario Bros*, os quais foram catalogados e definidos, com breves explicações. Estes padrões foram transcritos e traduzidos na Tabela 1. Os alelos resultantes foram 30, cada um deles servindo um a propósito. Para a adequação aos padrões originais, alguns elementos foram separados em diversos alelos e a função de *fitness* foi calibrada para incentivar a sua reprodução correta.

Tabela 1 – Padrões presentes no jogo original do *Super Mario Bros*

Inimigos	
Inimigo	Um inimigo apenas;
Horda de 2	Dois inimigos seguidos;
Horda de 3	Três inimigos seguidos;
Horda de 4	Quatro inimigos seguidos;
Teto	Inimigos embaixo de uma plataforma, forçando o jogador a bater na mesma quando pula;
Abismos	
Abismos	Um abismo no chão da fase;
Abismos multiplos	Mais de um abismo no chão da fase, com espaçamentos fixos entre eles;
Abismos variaveis	Abismos com plataformas entre eles de tamanho variavel;
Abismos e inimigos	Inimigos no ar acima de abismos;
Abismos e obstáculos	Pilares (canos ou blocos) nas bordas de abismos;
Vales	
Vale	Um vale criado por blocos verticais ou canos, sem plantas carnivoras nos mesmos;
Vale de canos	Um vale criado por canos, com plantas carnivoras em um ou dois dos mesmos;
Vale vazio	Um vale sem inimigos;
Vale e inimigos	Um vale com inimigos;
Vale e teto	Um vale com inimigos e um teto acima dos mesmos;
Multiplos	
2 caminhos	Uma plataforma que divide o caminho da fase em 2;
3 caminhos	Duas plataformas que dividem o caminho da fase em 3;
Risco e recompensa	Um caminho multiplo onde um caminho leva a uma recompensa e um abismo e outro que leva a um inimigo;
Escadas	
Escada para cima	Uma escada ascendente;
Escada para baixo	Uma escada descendente;
Vale de escadas vazio	Um vale com escadas nas bordas;
Vale de escadas	Um vale com escadas nas bordas e inimigos dentro;
Escadas com abismo	Um vale com escadas nas bordas e um abismo no centro;

Os alelos são de composição similar à fase, uma vez que eles compõem a mesma. São atributos dos alelos: duas matrizes, uma delas contendo os componentes do alelo, como onde fica o chão e onde estão blocos, canos e moedas. A outra matriz define onde estão e de que tipo são os inimigos presentes nele, se aplicável. Para evitar repetições, toda vez que um alelo é aplicado à uma fase em construção ele é refeito, com inimigos sorteados em posições fixas predefinidas. Caso não fosse feito isso, todos os padrões que contém um inimigo apenas, por exemplo, teriam o mesmo inimigo em todas as suas aparições na fase.

A definição dos alelos se dá em linha de código, com a definição da matriz espacial que o compõe. A representação é de um *Array* de *Arrays* de *Bytes* (*byte*[ ][ ]). Cada "linha" ou *array*, neste caso, representa uma fatia vertical da fase. A Figura 11 representa o mesmo elemento da fase em duas situações, a primeira como ele aparece para jogador e a segunda como seria a divisão da mesma em código.

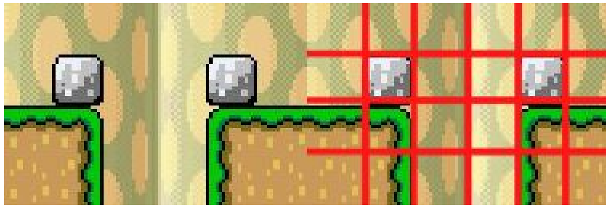

Figura 11 – *Infinite Mario Bros*: Representação de um alelo.

Devido a falta de detalhamento nos padrões encontrados por Dahlskog *et al*(DAHLSKOG; TOGELIUS, 2012), uma interpretação foi feita sobre eles e foram criados 30 padrões com base nos mesmos. Os padrões criados estão descritos nas Tabelas 2, 3, 4, 5 e 6. A tabela 2 descreve os padrões com inimigos capazes de locomoção, ou seja, os padrões que apresentam inimigos que não sejam flores carnivoras.

A tabela 3 contém todos os padrões com um abismo, que podem variar de acordo com sua largura e os blocos em suas bordas. Suas larguras podem variar entre 2, 3 e 4.

A tabela 4 descreve padrões de vales sem inimigos dentro. Assim como os padrões de abismos, os de vales podem variar em largura e blocos nas bordas.

Tabela 2 – Padrões contendo inimigos capazes de locomoção criados para a implementação do algoritmo evolutivo no jogo *Infinite Mario Bros*

Inimigos	
Inimigo	Um inimigo apenas;
Horda de 2	Dois inimigos seguidos;
Horda de 3	Três inimigos seguidos;
Horda de 4	Quatro inimigos seguidos;
Teto	Quatro inimigos embaixo de uma plataforma, o primeiro bloco da plataforma é um bloco que contém um cogumelo ou flor;
Inimigos e moedas	Três blocos de moedas espaçados entre si, com quatro inimigos embaixo.

Tabela 3 – Padrões contendo abismos

Abismos	
Abismo 2	Um Abismo com dois blocos de largura;
Abismo 3	Um Abismo com três blocos de largura;
Abismo 4	Um Abismo com quatro blocos de largura;
Abismo 2 com pedras	Um Abismo com dois blocos de largura com uma pedra em cada borda;
Abismo 3 com pedras	Um Abismo com três blocos de largura com uma pedra em cada borda;
Abismo 4 com pedras	Um Abismo com quatro blocos de largura com uma pedra em cada borda;
Abismo 2 com escadas	Um Abismo com dois blocos de largura com escadas nas bordas;
Abismo 3 com escadas	Um Abismo com três blocos de largura com escadas nas bordas;
Abismo 4 com escadas	Um Abismo com quatro blocos de largura com escadas nas bordas;

Tabela 4 – Padrões contendo Vales

Vales	
Vale de blocos 3	Um vale com três de largura com uma torre de três blocos de altura em cada canto;
Vale de blocos 4	Um vale com quatro de largura com uma torre de três blocos de altura em cada canto;
Vale de blocos 5	Um vale com cinco de largura com uma torre de três blocos de altura em cada canto;
Vale de canos 3	Um vale com três de largura com um cano em cada canto, sendo que um deles contém uma flor;
Vale de canos 4	Um vale com quatro de largura com um cano em cada canto, sendo que um deles contém uma flor;
Vale de canos 5	Um vale com cinco de largura com um cano em cada canto, sendo que um deles contém uma flor;
Vale de escadas	Um vale com três de largura com uma escada virada para frente numa borda e outra virada para trás na outra;

Os padrões de vales que contém inimigos seriam proibitivamente complicados de serem descritos diretamente, com todas as variações de inimigos e bordas possíveis. Por exemplo, temos 5 blocos possíveis de início de vale, 6 tipos diferentes de inimigos e outros 5 blocos finais, gerando 150 combinações diferentes. Por isso os elementos de início e fim de vales foram separados em alelos individuais e descritos na tabela 5.

Para aumentar a variabilidade e presença de conteúdos de recompensa, alelos contendo exclusivamente os mesmos foram criados. Estes estão descritos na tabela 6.

Tabela 5 – Padrões formativos de outros

Padrões formativos	
Cano	Um cano de três blocos de altura;
Cano com flor	Um cano de três blocos de altura com uma flor carnívora;
Pilar	Um pilar de três blocos de altura;
Escada ascendente	Uma escada virada para frente de três blocos de altura;
Escada descendente	Uma escada virada para trás de três blocos de altura;

Tabela 6 – Padrões contendo recompensas

Recompensas	
Bloco de *powerup*	Um bloco contendo um cogumelo ou uma flor;
Moedas	Quatro moedas em formato de cruz, acima do chão;
Blocos de moedas	Três blocos de moedas, com espaçamento de um entre eles;

3.3 FUNÇÃO DE *FITNESS*

A função de *fitness* a ser utilizada deverá ser capaz de avaliar os indivíduos em questão rapidamente e ser capaz de realizar três funções:

1. Coibir a repetição direta de padrões similares. Por exemplo: vários padrões com inimigos seguidos;

2. Estimular padrões observados nos *design patterns* da literatura. Por exemplo: Vales com inimigos formados por componentes diferentes;

3. Providenciar desafios e recompensas ao jogador em proporções similares. Por exemplo: Um bloco de *powerup* seguindo um padrão de inimigos;

A rapidez de avaliação neste caso é muito importante, para que seja feita a geração de fases em tempo de execução, portanto, foram descartadas neste momento quaisquer funções baseadas em Inteligência Artificial. Uma fórmula foi considerada apropriada para a rapidez, porém, seria extremamente complexa para realizar os pontos propostos.

Foi utilizada então uma tabela, que levaria em consideração diferentes combinações de tipos diferentes de alelos e daria um *fitness* apropriado para cada uma das mesmas. Desta forma, padrões de alto valor para o jogo seriam estimulados. A nota atribuída a cada combinação vai de 0,1 até 2 e foi definida com base em padrões presentes na Tabela 1, observações feitas pelo autor e seus conhecimentos. Na tabela, o *fitness* de uma combinação é feita com a linha representando o padrão a esquerda e a coluna o padrão a direita. A tabela de *fitness* está resumida e simplificada na Tabela 7. Na tabela original, os padrões de número 1 até 5 são inimigos, 6 até 14 são abismos, 15 até 20 são vales, 21 até 26 são formadores de vales e 27 até 30 são padrões de recompensa.

Tabela 7 – Tabela de *fitness* das combinações dos diferentes padrões, simplificada

	Inimigo	Abismo	Vale	Formativo	Recompensas
Inimigo	0,1	1	1	2	1,5
Abismo	1,9	0,1	1	1	1
Vale	1	1	0,1	0,1	1
Formativo	2	1	0,1	0,1	1
Recompensa	1	1	1	1	0,1

Além da tabela, o número de ocorrências do mesmo alelo no cromossomo é contado e caso este exceda 15% do número total de alelos no cromossomo, é descontado do *fitness* resultante da tabela duas vezes o número de ocorrências.

Para melhor compreensão, seu funcionamento será explicado por meio de exemplos:

1. Desconsiderando a porcentagem que indica repetição, um cromossomo com os padrões : 1, 6, 2, 22 (Inimigo, abismo, inimigo, formativo); teria o *fitness* destes padrões igual a "4,9", pois uma combinação de inimigo seguido por abismo possui nota "1", um abismo seguido por inimigo nota "1,9"e um inimigo seguido por formativo "2", para um total de "4,9".

2. Considerando repetições, um cromossomo de tamanho 10 com os padrões : 1, 1, 2, 22, 3, 29, 30, 27, 17, 5 (inimigo, inimigo, inimigo, formativo, inimigo, recompensa, recompensa, recompensa, vale, abismo); teria o *fitness* destes padrões igual a "3,9", pois suas combinações seriam: $0,1+0,1+2+2+1,5+0,1+0,1+1+1 = 7,9$, porém, pelo padrão de número "1" se repetir em 20% do cromossomo, é descontado esse número de repetições em dobro, "4", do *fitness* resultante da tabela, acarretando um *fitness* final de "3,9".

3.4 *CROSSOVER* E MUTAÇÃO

O *Crossover* é feito a cada geração, após a normalização do valor do *fitness* de cada fase. Para a normalização do *fitness*, o valor do *fitness* total da população é calculado e em seguida os valores individuais são

divididos pelo mesmo. Um gerador de números aleatórios gera então seleções de pares de fases, cujos cromossomos são enviados para o objeto que controla a operação.

A operação é feita com um *Single Point Crossover*, ou seja, um ponto de corte apenas é selecionado. Este ponto é um número que se encontra entre 0 e o tamanho do menor cromossomo utilizado. Ocorre então a troca de "material genético" dos cromossomos, do primeiro elemento até o elemento do ponto de *Crossover*, gerando dois cromossomos novos de tamanhos iguais aos originais e contendo pedaços dos mesmos.

Depois do processo de *crossover*, é feito o cálculo da mutação. A mutação possui 0.1% de chance de ocorrer por alelo do cromossomo de cada indivíduo. Caso uma mutação ocorra em determinado alelo, sua posição no cromossomo é alterada para conter um alelo aleatório que não exclui o original, causando a probabilidade de 1/30 de que uma mutação seja efetivamente sem efeitos.

A taxa de mutação foi definida em 0.1%, apesar de possivelmente não ser ideal, após testes com diversas outras em que ocorriam duas possibilidades: A stagnação do *fitness* populacional caso muito baixa e a dificuldade em aumentar o mesmo caso muito alta. Foi observado que uma taxa acima de 1% acarreta em um número muito maior de gerações para o mesmo resultado que a taxa escolhida, enquanto uma abaixo de 0.01% apesar de alcançar os resultados desejados ocasionalmente, apresentou uma média menor do que a de 0,01%. Para 100 execuções do algoritmo, a média do *fitness* dos melhores indivíduos para as taxas 1%, 0.1% e 0.01% para uma população de 500 indivíduos no final de 500 gerações foram, respectivamente: 74,10, 85,88 e 83,20.

4 RESULTADOS

Os resultados decorrentes do processo do algoritmo evolutivo proposto são dependentes de dois fatores: o número de gerações que irão decorrer e o tamanho da população. Estes fatores impactam principalmente o tempo de processamento e a qualidade final do produto. Para ser possível determinar qual é a combinação ideal para geração de conteúdo *online*, ou seja, em tempo de execução, foi feita uma análise dos resultados, incluíndo nesta o *fitness* do melhor indivíduo de cada geração, a média, o desvio padrão da mesma e o tempo de processamento.

4.1 REPRESENTAÇÕES GRÁFICAS

As figuras 12, 13, 14, 15, 16, 17, 18, 19 e 20 representam gráficos da variação do *fitness* ao longo de gerações para populações de tamanhos diferentes. Os pontos azuis são os melhores indivíduos de cada geração e as linhas laranja e verde representam a média populacional e o desvio padrão, repespectivamente. As populações e as gerações variam entre 100, 500 e 1000.

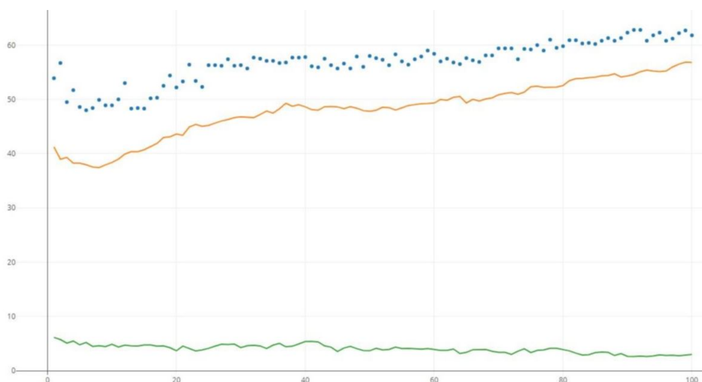

Figura 12 – Gráfico representando a variação de *fitness* ao longo de 100 gerações para uma população de 100 indivíduos.

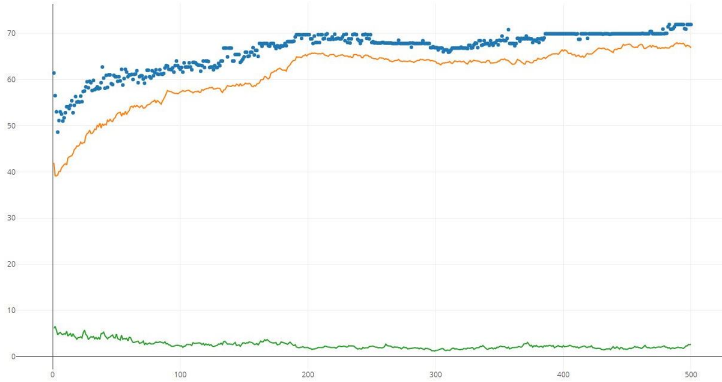

Figura 13 – Gráfico representando a variação de *fitness* ao longo de 500 gerações para uma população de 100 indivíduos.

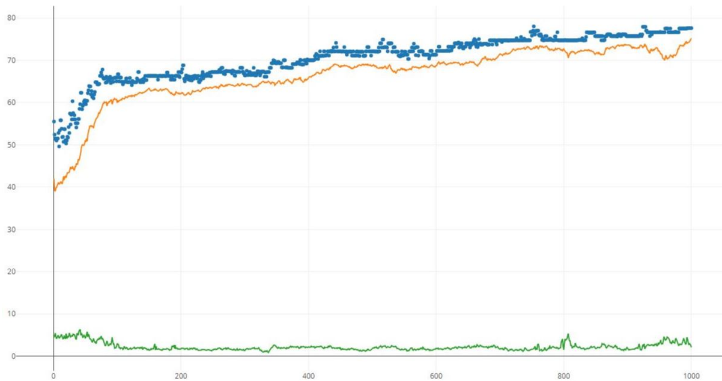

Figura 14 – Gráfico representando a variação de *fitness* ao longo de 1000 gerações para uma população de 100 indivíduos.

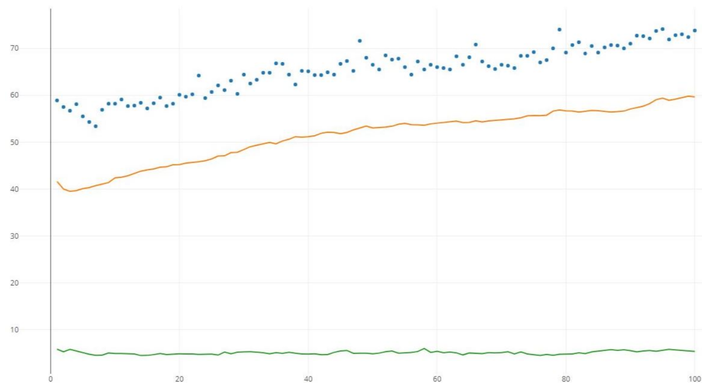

Figura 15 – Gráfico representando a variação de *fitness* ao longo de 100 gerações para uma população de 500 indivíduos.

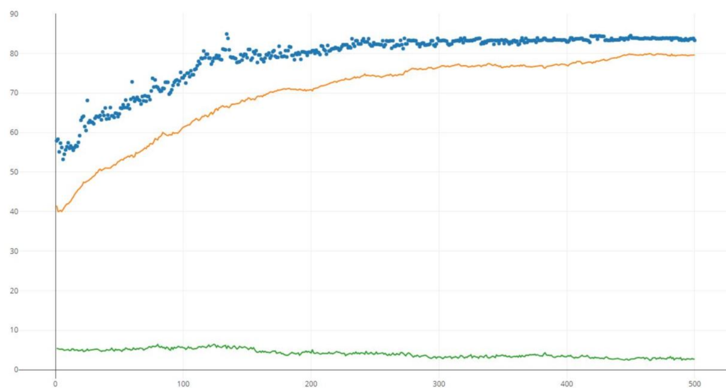

Figura 16 – Gráfico representando a variação de *fitness* ao longo de 500 gerações para uma população de 500 indivíduos.

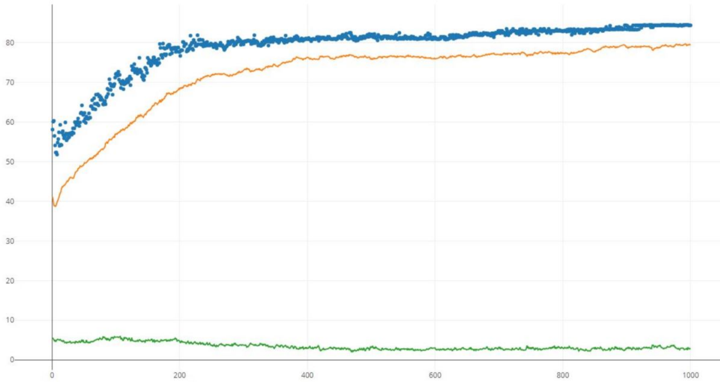

Figura 17 – Gráfico representando a variação de *fitness* ao longo de 1000 gerações para uma população de 500 indivíduos.

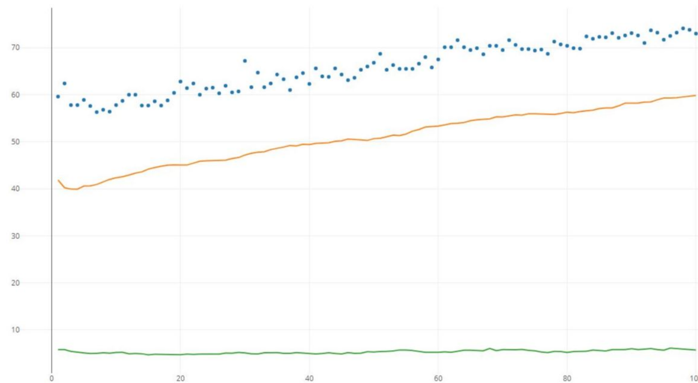

Figura 18 – Gráfico representando a variação de *fitness* ao longo de 100 gerações para uma população de 1000 indivíduos.

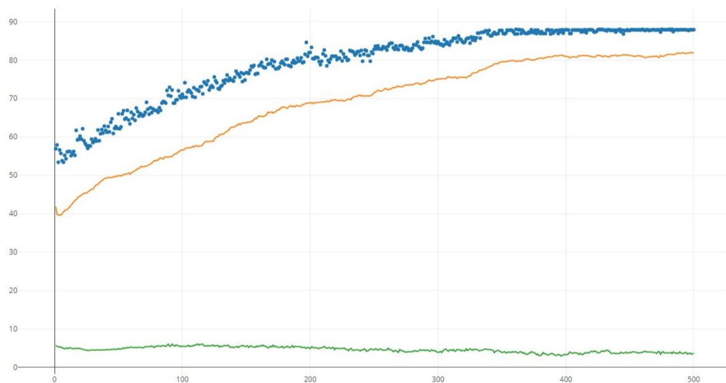

Figura 19 – Gráfico representando a variação de *fitness* ao longo de 500 gerações para uma população de 1000 indivíduos.

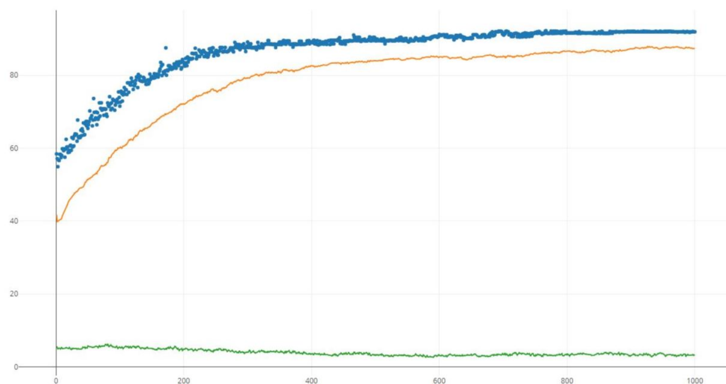

Figura 20 – Gráfico representando a variação de *fitness* ao longo de 1000 gerações para uma população de 1000 indivíduos.

4.2 TEMPO DE PROCESSAMENTO

Como é observado nos gráficos, tanto o aumento da população quanto o número de gerações possuem correlação direta tanto no aumento do *fitness* da fase resultante, até o ponto onde o mesmo entra em estado de estagnação. Para a análise do tempo de processamento, o algoritmo foi executado 100 vezes para cada combinação de geração e população, resultando na Tabela 8, que indica o tempo médio em segundos, além do desvio padrão dos mesmos, auxiliando na compreensão dos limites do tempo. É necessário ter em mente que o algoritmo não está otimizado para execução paralela, sendo executado sequencialmente.

Como é possível observar na tabela 8, existe tanto uma correlação direta entre o aumento do número de gerações quanto de tamanho da população e o tempo de processamento. Vale ressaltar que o tempo de processamento dos pares de população 100 e geração 500 é quase idêntico ao seu contrário, de população 500 e geração 100. Mesmo assim, não podemos afirmar que a igualdade no produto do número de gerações e tamanho de população e similaridade no tempo de processamento acarreta na igualdade de *fitness* e indivíduo resultante. A razão do mesmo é a aleatoriedade de resultados do processo do algoritmo evolutivo.

Para testar a influência dos diferentes parâmetros no tempo de processamento e *fitness* resultante, foram feitos alguns testes com populações e gerações de diferentes tamanhos, porém cujo produto dos mesmos seja igual. Os resultados foram: Para uma população/geração de 5000/10, onde o produto dos mesmos é igual às combinações 100/500 e 500/100, o tempo de processamento foi de 2,031 segundos, com um *fitness* resultante de aproximadamente 60. A combinação contrária, 10/5000 acarretou num tempo de processamento de 1,544 segundos e 43 de *fitness*.

Tabela 8 – Tempo médio em segundos / desvio padrão para o processamento de diferentes populações e número de gerações

	Número de Gerações		
População	100	500	1000
100	0,219 / 0,012	1,067 / 0,022	2,140 / 0,038
500	1,102 / 0,032	5,520 / 0,106	10,919 / 0,165
1000	2,239 / 0,047	10,963 / 0,070	21,981 / 0,184

Tabela 9 – *Fitness* médio / desvio padrão dos melhores indivíduos na geração de número 100, 500 e 1000 para diferentes populações

População / Geração	Geração 100	Geração 500	Geração 1000
100 / 100	62,61/3,22	N.A.	N.A.
100 / 500	61,75/3,70	67,60/3,96	N.A.
100 / 1000	62,52/3,60	68,13/3,79	70,83/3,69
500 / 100	72,22/3,24	N.A.	N.A.
500 / 500	72,90/2,95	85,88/2,76	N.A.
500 / 1000	72,92/2,98	85,70/3,01	87,31/3,10
1000 / 100	74,36/2,63	N.A.	N.A.
1000 / 500	75,05/2,50	89,63/2,18	N.A.
1000 / 1000	74,43/2,47	89,32/2,35	90,36/2,39

A tabela 9 mostra o *fitness* médio e o desvio padrão dos melhores indivíduos nas gerações 100, 500 e 1000 para 900 diferentes execuções do programa (100 para cada combinação), onde foram variados como parametro de entrada tanto o tamanho da população quanto o número de gerações que deveriam ocorrer. A primeira coluna contém a combinação de População e Geração de cada execução. As colunas 2, 3 e 4 contém o *fitness* médio e o desvio padrão dos melhores indivíduos de cada execução nas gerações 100, 500 e 1000, respectivamente. Caso alguma geração em questão não seja alcançável por aquela combinação em particular, um "N.A." significando "Não se aplica" é colocado no lugar do *fitness*.

4.3 ANÁLISE DOS RESULTADOS

Pelos gráficos resultantes do processo de análise estatística, é possível observar que para as populações de 500 e 1000 indivíduos a variação do *fitness* do melhor indivíduo é pequena após a marca da geração de número 400, apesar do aumento da média de cada geração subsequente.

Como já foi citado anteriormente, a avaliação automática dos resultados do processo de *PCG* é um grande problema da área. Este problema também afetou o andamento deste trabalho, uma vez que a implementação dos algoritmos exibidos em capitulos anteriores acarretaria numa fuga do escopo do mesmo. Portanto, foi feita uma análise

manual dos resultados por parte do autor, buscando encontrar pontos positivos e negativos em diferentes fases.

Para se obter uma noção de uma fase com *fitness* baixo e puramente aleatória, foram criadas 5 fases com os parâmetros de tamanho de população 2 e 1 geração, o mínimo permitido pelo algoritmo. Alguns pontos foram observados nelas:

1. Muitos abismos, vales e padrões formadores dos mesmos, uma vez que estes representam 70% de todos os tipos de padrões;
2. Curva de dificuldade variada, duas das fases com muitos abismos consecutivos necessitando alta cautela, enquanto outras duas continham muitos vales vazios, sem desafios em forma de inimigos, uma delas possuiu boa variedade, mas com alta repetição de elementos. A Figura 21 contém uma imagem da mesma, exibindo a repetitividade.
3. No geral, foram fases entediantes.

Figura 21 – Padrões repetitivos em indivíduo de *fitness* igual a 44.

Para comparação, foram feitas 5 fases com parâmetros de população 5000 e geração 1000, com *fitness* médio de 93,60 e tempo de processamento médio de cerca de 130 segundos. Foi observado que o *fitness* médio do melhor indivíduo na geração 1000 já havia sido quase alcançado pela geração 400 em todos os casos sendo este em média 93,50. É possível observar que apesar da pouca variação no melhor indivíduo, a média populacional sofreu um aumento médio de 80,06 para 88,87 nas 600 gerações que passaram entre estas. O resultado das observações foram:

1. Muitos vales com inimigos.
2. Predominância de padrões de tamanhos pequenos.
3. Dificuldade acentuada.
4. Repetições eventuais.

Um elemento não esperado era o de repetições de padrões iguais, uma vez que o *fitness* destas combinações na tabela de *fitness* é baixíssimo. Mesmo assim foram observados e a Figura 22 mostra o mesmo ocorrendo.

Figura 22 – Padrões repetitivos em indivíduo de *fitness* igual a 92,50.

Uma vez expostos e analisados estes indivíduos "extremos" foram estudadas as combinações da Tabela 9, tendo a Tabela 8 em mente, combinações com tempo de processamento acima de 15 segundos foram excluídas. As fases resultantes foram divididas em 3 faixas de *fitness*: até 69,99; De 70 até 79,99; e maiores que 80.

Os resultados da análise para indivíduos até 69,99:

1. Vales com inimigos são incomuns, mas não raros.
2. Alta Repetição de padrões.
3. Dificuldade baixa.
4. Tempo de processamento da fase quase instantâneo.
5. Alta variação de padrões utilizados.

Os resultados da análise para indivíduos entre 70 e 79,99 são:

1. Vales com inimigos são comuns.
2. Repetição de padrões.
3. Dificuldade mediana.
4. Tempo de processamento da fase rápido.

Os resultados da análise para indivíduos acima de 80 são:

1. Vales com inimigos são abundantes.
2. Repetição de padrões rara.
3. Dificuldade mediana média para alta.
4. Tempo de processamento da fase relativamente demorado.
5. Padrões de largura baixa utilizados com frequência.

Portanto, com base nesta análise, são desejáveis indivíduos de *fitness* acima de 80. A combinação ideal para gerar tais indivíduos de acordo com a Tabela 8, ou seja, levando em consideração o tempo de processamento, é de uma população de 500 indivíduos, ao longo de 500 gerações.

5 CONCLUSÃO

Este trabalho serviu para abordar este tema que pode se beneficiar da atenção da comunidade acadêmica. Mais estudos nesta área serviriam para otimizar os processos e gerar uma redução do custo de produção de jogos, tanto eletrônicos quanto tradicionais.

Na fundamentação teórica, foi possível definir o que é o *Game Space PCG*, além de sua origem histórica. As maneiras de realizar a geração de conteúdo descritas neste trabalho servem como inspiração para o aprofundamento no tema.

Duas maneiras de avaliar o conteúdo gerado por diferentes algoritmos foram descritas, uma delas em particular, a proposta por Liapis *et. al.*(LIAPIS; YANNAKAKIS; TOGELIUS, 2014) é extremamente interessante, pela sua natureza genérica.

A implementação do algoritmo evolutivo na plataforma de testes se deu de maneira bem sucedida. O maior gargalo na implementação foi a função de *fitness* escolhida. Originalmente seria feito uso do algoritmo proposto por Liapis *et al.*, porém, devido a sua complexidade ele seria proibitivamente custoso em tempo de processamento para exercer sua função de maneira satisfatória. A tabela implementada no entanto provou-se apropriada, com resultados satisfatórios.

Game Space PCG demonstrou ser um amplo campo de estudos, com alto potencial para aplicação de técnicas de inteligência artificial. A emulação da criatividade humana é algo possível nesta área, que pode servir como fronteira de estudos por muitos anos.

6 TRABALHOS FUTUROS

Para trabalhos futuros, é proposto:

- Aumento da variedade de padrões formativos (alelos) dos indivíduos, incluindo abismos com blocos acima dos mesmos, que foram observados na Tabela 1, porém não implementados.
- Alteração na tabela de *fitness*, estimulando padrões diferentes aos atuais e comparando os resultados com os deste trabalho.
- Criação de conjuntos de treinamento para uma rede neural recorrente com a matriz de *fitness* proposta e com matrizes diferentes.
- Avaliação do resultado deste trabalho ou de derivados com o algoritmo proposto por Liapis *et al.*(LIAPIS; YANNAKAKIS; TOGELIUS, 2014), comparando os resultados com os obtidos por Shi e Chen (SHI; CHEN, 2016).
- Alterar o algoritmo genético implementado, utilizando técnicas de *crossover* diferentes e o implicação no *fitness* resultante.

REFERÊNCIAS

CRAIG, S. *SPORTS AND GAMES OF THE ANCIENTS*. Westport, EUA: Greenwood Press, 2002. 288 p.

DAHLSKOG, S.; TOGELIUS, J. Patterns and procedural content generation: Revisiting mario in world 1 level 1. In: *Proceedings of the First Workshop on Design Patterns in Games*. New York, NY, USA: ACM, 2012. (DPG '12), p. 1–8.

DAHLSKOG, S.; TOGELIUS, J. Patterns as objectives for level generation. In: . [S.l.: s.n.], 2013.

DAHLSKOG, S.; TOGELIUS, J.; NELSON, M. J. Linear levels through n-grams. In: LUGMAYR, A. (Ed.). *MindTrek*. [S.l.]: ACM, 2014. p. 200–206. ISBN 978-1-4503-3006-0.

GYGAX, G.; ARNESON, D. *Dungeons & Dragons: Rules for Fantastic Medieval Wargames Campaigns Playable with Paper and Pencil and Miniature Figures*. [S.l.]: TSR Hobbies, 1974.

HENDRIKX, M. et al. Procedural content generation for games: A survey. *ACM Trans. Multimedia Comput. Commun. Appl.*, ACM, New York, NY, USA, v. 9, n. 1, p. 1–22, fev. 2013.

KRAMER, W. *What Is a Game?* dez. 2000. The Games Journal. <http://www.thegamesjournal.com/articles/WhatIsaGame.shtml>. Acessado em 08/05/2018.

LIAPIS, A.; YANNAKAKIS, G. N.; TOGELIUS, J. Towards a generic method of evaluating game levels. In: *Proceedings of the Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. [S.l.]: AAAI Press, 2014. (AIIDE'13), p. 30–36.

NINTENDO. Jogo Digital, *Super Mario Bros*. 1985.

PEDERSEN, C.; TOGELIUS, J.; YANNAKAKIS, G. N. Modeling player experience in super mario bros. In: *2009 IEEE Symposium on Computational Intelligence and Games*. Milão, Italia: [s.n.], 2009. -, p. 132–139.

PERSSON, M. A. Jogo Digital, *Infinite Mario Bros*. 2008.

SHAKER, N. et al. Evolving personalized content for super mario bros using grammatical evolution. In: *Proceedings of the 8th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2012*. [S.l.: s.n.], 2012. p. 75–80.

SHI, P.; CHEN, K. Online level generation in super mario bros via learning constructive primitives. In: *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. [S.l.: s.n.], 2016. p. 1–8.

SMITH, G. An analog history of procedural content generation. In: *Proceedings of the 10th International Conference on the Foundations of Digital Games*. [S.l.: s.n.], 2015. p. 22–25.

SMITH, G.; CHA, M.; WHITEHEAD, J. A framework for analysis of 2d platformer levels. In: *Proceedings of the 2008 ACM SIGGRAPH Symposium on Video Games*. New York, NY, USA: ACM, 2008. (Sandbox '08), p. 75–80.

SUMMERVILLE, A.; MATEAS, M. Super mario as a string: Platformer level generation via lstms. *CoRR*, abs/1603.00930, 2016.

SUMMERVILLE, A. et al. Procedural content generation via machine learning (pcgml). *CoRR*, abs/1702.00539, 2017.

TOGELIUS, J. et al. What is procedural content generation?: Mario on the borderline. In: *Proceedings of the 2Nd International Workshop on Procedural Content Generation in Games*. New York, NY, USA: ACM, 2011. (PCGames '11), p. 3:1–3:6.

TOGELIUS, J. et al. Search-based procedural content generation: A taxonomy and survey. *IEEE Trans. Comput. Intellig. and AI in Games*, v. 3, n. 3, p. 172–186, 2011.

TSR, H. *Dungeons & Dragons: Dungeon Geomorphs*. 1976.

TYLOR, E. B. The history of games. *Fortnightly review*, Londres, Reino Unido, v. 25, n. 149, p. 735–747, maio 1879.

WILLIAMS, D. Structure and competition in the u.s. home video game industry. *The International Journal on Media Management*, v. 4, n. 1, p. 41–54, 2002.

7 ANEXOS

7.1 CÓDIGO FONTE

Classe executora do processo de seleção, envia o objeto para o jogo principal:

```
package tcc.klauskuhr.AG;

import dk.itu.mario.level.Level;
import java.sql.Time;
import java.time.Duration;
import java.time.Instant;
import java.util.Locale;
import java.util.Random;
import tcc.klauskuhr.AG.data.Controlador;
import tcc.klauskuhr.AG.data.Gerador;
import tcc.klauskuhr.AG.entidades.Fase;
import
    tcc.klauskuhr.Avaliador.Estatistica.Estatistica;

/**
 *
 * @author Klaus
 */
public class Executor {

    private int poplen = 500;
    private int gernum = 500;

    private Fase[] pop;
    private Controlador cont;
    private double[] fitnesses;

    private Printf writer;
    private Estatistica est;

    private String[] resultados;
```

```
    private long[] tempos;
    private double[][] medias;
    private double[][] desvios;



    public Executor() {
        Gerador ger = new Gerador();
        cont = new
            Controlador(ger.inicializarGenes(),
            ger);

        writer = new Printf();
        est = new Estatistica();

    }

    public Level selecionaFase() {
        //Caso for de interesse realizar a analise
        //estatistica, aumentar este numero.
        int iteracoes = 1;

        if (gernum < 1) {
            gernum = 1;
        }
        if (poplen < 2) {
            poplen = 2;
        }
        pop = new Fase[poplen];
        fitnesses = new double[poplen];
        resultados = new String[gernum];

        tempos    = new long[iteracoes];
        medias = new double[gernum][iteracoes];
        desvios = new double[gernum][iteracoes];
        String[] resu = new String[gernum];


        Level ret = null;

        for (int i = 0; i < iteracoes; i++) {
```

```
String fileName = ".txt";

Instant start = Instant.now();

ret = selectBestIndividual(i);
Instant finish = Instant.now();
long timeElapsed = Duration.between(start,
    finish).toMillis();
System.out.println("Tempo : " +
    timeElapsed + " ms");
tempos[i]=timeElapsed;

writer.write(resultados,"p"+poplen+"g"+gernum+"i"+i

}

for (int i = 0; i < resu.length; i++) {
     resu[i] = calcResultados(i);
}


//tempos
double[] aux = new double[iteracoes];
for (int i = 0; i < iteracoes; i++) {
     Long l = new Long(tempos[i]);
     aux[i] = l.doubleValue();
}
double mediaTempo = est.calcMedia(aux);
double desvioTempo = est.calcDesvioP(aux,
    mediaTempo);

String[] tempoAux = {String.format("%.3f",
    mediaTempo)+" , "+
    String.format("%.3f", desvioTempo)};


writer.write(tempoAux,
    "p"+poplen+"g"+gernum+"t"+iteracoes+".txt");

writer.write(resu,
    "p"+poplen+"g"+gernum+"iteracoes"+iteracoes+".t
```

```
        return ret;
    }

    private Level selectBestIndividual(int it) {
        //primeira geracao

        for (int i = 0; i < pop.length; i++) {
            pop[i] = new Fase(320, 15, cont);

        }
        calcFitnessIndividual();
        // escreverTudo();
        //writer.write("Numero de Individuos: " +
            poplen);
        //writer.write("Numero de geracoes  : " +
            gernum);

        //gernum eh o numero maximo de geracoes.
        //geracoes subsequentes
        for (int i = 0; i < gernum; i++) {
            passarGeracao(i,it);
            //escreverTudo();
        }

        return pop[best()];

        //       throw new
            UnsupportedOperationException("Not
            supported yet."); //To change body of
            generated methods, choose Tools |
            Templates.
    }

    private void passarGeracao(int n,int it) {
        calcFitnessIndividual();

        int[][] newPop = new int[pop.length][];
        int[][] aux;
```

```
        double fitnessTotal = calcFitnessTotal();
        double[] fitnessRelativo = new
            double[fitnesses.length];

        for (int i = 0; i <
            fitnessRelativo.length; i++) {
             fitnessRelativo[i] = fitnesses[i] /
                 fitnessTotal;
        }

    for (int i = 0; i < pop.length - 1; i += 2) {
    aux = cont.crossover
    (pop[this.roleta(fitnessRelativo)].getGenoma(),
        pop[this.roleta(fitnessRelativo)].getGenoma());
    newPop[i] = aux[0];
    newPop[i + 1] = aux[1];
        }

        for (int i = 0; i < pop.length; i++) {

             pop[i] = new Fase(320, 15, cont,
                 newPop[i], n == gernum - 1);

        }

        int auxil = best();

        //System.out.println("Melhor fitness da
            geracao " + (n + 1) + " :" + " " +
            fitnesses[auxil]);
        double media, desvio;
        media = est.calcMedia(fitnesses);
        desvio = est.calcDesvioP(fitnesses, media);

        String texto = "Melhor fitness da geracao
            " + (n + 1) + " :" + " " +
            String.format("%.2f",
            fitnesses[auxil]) + " Media: " +
            String.format("%.2f", media) + "
            Desvio Padrao: " +
            String.format("%.2f", desvio);
```

```
        // System.out.println(texto);

        medias[n][it]=fitnesses[auxil];




    // writer.write(texto);
        String escrever = (n + 1) + " , " +
            String.format(Locale.US, "%.2f",
            media) + " , " +
            String.format(Locale.US, "%.2f",
            desvio) + " , " +
            String.format(Locale.US, "%.2f",
            fitnesses[auxil]);
        //System.out.println(texto);
        resultados[n] = escrever;
        // writer.write(escrever);
        // writer.novaLinha();

    }

    private int best() {
        int ret = -1;
        double aux = Double.MIN_VALUE;

        for (int i = 0; i < pop.length; i++) {
            if (fitnesses[i] > aux) {
                ret = i;
                aux = fitnesses[i];
            }
        }

        //  System.out.println("Fitness do melhor:
            " + aux);
        return ret;
    }

    private double calcFitnessTotal() {
        double ret = 0;
```

```
        for (int i = 0; i < fitnesses.length; i++)
            {
             ret += fitnesses[i];
        }
        return ret;
    }

    private void calcFitnessIndividual() {
        for (int i = 0; i < pop.length; i++) {
            fitnesses[i] = pop[i].calcFitness();
        }
    }

    private int roleta(double[] fitnessRelativo) {
        Random r = new Random();
        double escolha = r.nextDouble();
        double aux = 0;

        for (int i = 0; i <
            fitnessRelativo.length; i++) {
             aux += fitnessRelativo[i];
             if (escolha <= aux) {
                 return i;
             }
        }

        throw new
            UnsupportedOperationException("Erro na
            roleta"); //To change body of
            generated methods, choose Tools |
            Templates.
        //return -1;
    }

    /*private void escreverTudo() {
        String aux = "";
        for (int i = 0; i < poplen - 1; i++) {

             aux += fitnesses[i] + ",";
        }
        aux += fitnesses[poplen - 1] + "";
```

```
            writer.write(aux);
            writer.novaLinha();
    }*/

    private String calcResultados(int atual) {

        String ret = new String();

        //precisa saber: a media do melhor
            individuo de cada geracao E desvio
            padrao dos mesmos
       double[] mediaGer ;
       double[] desvioGer ;

       mediaGer = medias[atual];
       desvioGer = desvios[atual];

       double media = est.calcMedia(mediaGer);


        ret+= String.format("%.2f", media)+" ,
            "+String.format("%.2f",
            est.calcDesvioP(medias[atual],media));





        return ret;
    }
}
```

Classe responsavel pelo cálculo da média e desvio padrão:

```
package tcc.klauskuhr.Avaliador.Estatistica;

/**
 *
 * @author Klaus
 */
public class Estatistica {
```

```
    public Estatistica() {
    }

    public double calcMedia(double[] elementos) {
        double ret = 0;

        for (double elemento : elementos) {
            ret += elemento;
        }
        ret = ret / elementos.length;
        return ret;
    }

    public double calcDesvioP(double[] elementos,
       double media) {
        double ret = 0;

        for (double elemento : elementos) {
            ret += Math.pow(elemento - media, 2);
        }
        ret = ret / elementos.length;

        ret = Math.sqrt(ret);

        return ret;
    }

}
```

Classe dos Alelos:

```
package tcc.klauskuhr.AG.data;

import dk.itu.mario.engine.sprites.SpriteTemplate;

/**
 *
 * @author Klaus
 */
public class Gene {
```

```
private byte[][] padrao;
private SpriteTemplate[][] spriteTemplates;

   public SpriteTemplate getSpriteTemplate(int
      x, int y)
{
    if (x < 0) return null;;
    if (y < 0) return null;;
    if (x >= padrao.length) return null;;
    if (y >= padrao[0].length) return null;;
    return spriteTemplates[x][y];
}


public void setSpriteTemplate(int x, int y,
   SpriteTemplate spriteTemplate)
{
    if (x < 0) return;
    if (y < 0) return;
    if (x >= padrao.length) return;
    if (y >= padrao[0].length) return;
    spriteTemplates[x][y] = spriteTemplate;
}

public Gene(byte[][] padrao) {
    this.padrao = padrao;
    this.spriteTemplates = new
       SpriteTemplate[padrao.length][padrao[0].length];
}



public byte[][] getPadrao() {
    return padrao;
}

public void setPadrao(byte[][] padrao) {
    this.padrao = padrao;
}

public int getLength(){
```

```
        return padrao.length;
        }
}
```

Classe que gera os Alelos

```
package tcc.klauskuhr.AG.data;

import dk.itu.mario.engine.sprites.Enemy;
import dk.itu.mario.engine.sprites.SpriteTemplate;
import dk.itu.mario.level.Level;
import java.util.Random;

/**
 *
 * @author Klaus
 */
public class Gerador {

    protected static final byte BLOCK_EMPTY =
        (byte) (0 + 1 * 16);
    protected static final byte BLOCK_POWERUP =
        (byte) (4 + 2 + 1 * 16);
    protected static final byte BLOCK_COIN =
        (byte) (4 + 1 + 1 * 16);
    protected static final byte GROUND = (byte) (1
        + 9 * 16);
    protected static final byte ROCK = (byte) (9 +
        0 * 16);
    protected static final byte COIN = (byte) (2 +
        2 * 16);

    protected static final byte LEFT_GRASS_EDGE =
        (byte) (0 + 9 * 16);
    protected static final byte RIGHT_GRASS_EDGE =
        (byte) (2 + 9 * 16);
    protected static final byte
        RIGHT_UP_GRASS_EDGE = (byte) (2 + 8 * 16);
    protected static final byte LEFT_UP_GRASS_EDGE
        = (byte) (0 + 8 * 16);
    protected static final byte LEFT_POCKET_GRASS
```

```
        = (byte) (3 + 9 * 16);
    protected static final byte RIGHT_POCKET_GRASS
        = (byte) (3 + 8 * 16);

    protected static final byte HILL_FILL = (byte)
        (5 + 9 * 16);
    protected static final byte HILL_LEFT = (byte)
        (4 + 9 * 16);
    protected static final byte HILL_RIGHT =
        (byte) (6 + 9 * 16);
    protected static final byte HILL_TOP = (byte)
        (5 + 8 * 16);
    protected static final byte HILL_TOP_LEFT =
        (byte) (4 + 8 * 16);
    protected static final byte HILL_TOP_RIGHT =
        (byte) (6 + 8 * 16);

    protected static final byte HILL_TOP_LEFT_IN =
        (byte) (4 + 11 * 16);
    protected static final byte HILL_TOP_RIGHT_IN
        = (byte) (6 + 11 * 16);

    protected static final byte TUBE_TOP_LEFT =
        (byte) (10 + 0 * 16);
    protected static final byte TUBE_TOP_RIGHT =
        (byte) (11 + 0 * 16);

    protected static final byte TUBE_SIDE_LEFT =
        (byte) (10 + 1 * 16);
    protected static final byte TUBE_SIDE_RIGHT =
        (byte) (11 + 1 * 16);

    private Gene[] geneList;

    public Gerador() {
    }

    public Gene[] inicializarGenes() {
        Random r = new Random();
        Gene[] lista = new Gene[30];
```

```
//**************************************************
    Gene[] lista = new Gene[23];
//Mapas padrao inimigos
byte[][] gen0 = {{GROUND, HILL_TOP, 0, 0},
    {GROUND, HILL_TOP, 0, 0}, {GROUND,
    HILL_TOP, 0, 0}};
byte[][] gen1 = {{GROUND, HILL_TOP, 0, 0},
    {GROUND, HILL_TOP, 0, 0}, {GROUND,
    HILL_TOP, 0, 0}, {GROUND, HILL_TOP, 0,
    0}};
byte[][] gen2 = {{GROUND, HILL_TOP, 0, 0},
    {GROUND, HILL_TOP, 0, 0}, {GROUND,
    HILL_TOP, 0, 0}, {GROUND, HILL_TOP, 0,
    0}, {GROUND, HILL_TOP, 0, 0}};
byte[][] gen3 = {{GROUND, HILL_TOP, 0, 0},
    {GROUND, HILL_TOP, 0, 0}, {GROUND,
    HILL_TOP, 0, 0}, {GROUND, HILL_TOP, 0,
    0}, {GROUND, HILL_TOP, 0, 0}, {GROUND,
    HILL_TOP, 0, 0}};
byte[][] gen4 = {{GROUND, HILL_TOP, 0, 0,
    0, 0, 0}, {GROUND, HILL_TOP, 0, 0,
    BLOCK_POWERUP, 0, 0}, {GROUND,
    HILL_TOP, 0, 0, BLOCK_EMPTY, 0, 0},
{GROUND, HILL_TOP, 0, 0, BLOCK_EMPTY, 0,
    0}, {GROUND, HILL_TOP, 0, 0,
    BLOCK_EMPTY, 0, 0}, {GROUND, HILL_TOP,
    0, 0, 0, 0, 0}};

//Tipos de inimigos: 1 Tartaruga Vermelha,
    1 Tartaruga Verde, 2 Goompa
//Boolean alado ou nao no final
lista[0] = new Gene(gen0);
lista[0].setSpriteTemplate(1, 2, new
    SpriteTemplate(r.nextInt(3),
    r.nextDouble() < 0.05));

lista[1] = new Gene(gen1);
lista[1].setSpriteTemplate(1, 2, new
    SpriteTemplate(r.nextInt(3),
    r.nextDouble() < 0.05));
lista[1].setSpriteTemplate(2, 2, new
```

```
        SpriteTemplate(r.nextInt(3),
        r.nextDouble() < 0.05));

    lista[2] = new Gene(gen2);
    lista[2].setSpriteTemplate(1, 2, new
        SpriteTemplate(r.nextInt(3),
        r.nextDouble() < 0.05));
    lista[2].setSpriteTemplate(2, 2, new
        SpriteTemplate(r.nextInt(3),
        r.nextDouble() < 0.05));
    lista[2].setSpriteTemplate(3, 2, new
        SpriteTemplate(r.nextInt(3),
        r.nextDouble() < 0.05));

    lista[3] = new Gene(gen3);
    lista[3].setSpriteTemplate(1, 2, new
        SpriteTemplate(r.nextInt(3),
        r.nextDouble() < 0.05));
    lista[3].setSpriteTemplate(2, 2, new
        SpriteTemplate(r.nextInt(3),
        r.nextDouble() < 0.05));
    lista[3].setSpriteTemplate(3, 2, new
        SpriteTemplate(r.nextInt(3),
        r.nextDouble() < 0.05));
    lista[3].setSpriteTemplate(4, 2, new
        SpriteTemplate(r.nextInt(3),
        r.nextDouble() < 0.05));

    lista[4] = new Gene(gen4);
    lista[4].setSpriteTemplate(1, 2, new
        SpriteTemplate(r.nextInt(3),
        r.nextDouble() < 0.05));
    lista[4].setSpriteTemplate(2, 2, new
        SpriteTemplate(r.nextInt(3),
        r.nextDouble() < 0.05));
    lista[4].setSpriteTemplate(3, 2, new
        SpriteTemplate(r.nextInt(3),
        r.nextDouble() < 0.05));
    lista[4].setSpriteTemplate(4, 2, new
        SpriteTemplate(r.nextInt(3),
        r.nextDouble() < 0.05));
```

```
//Abismos (3 tamanhos)
byte[][] gen5 = {{RIGHT_GRASS_EDGE,
    RIGHT_UP_GRASS_EDGE, 0, 0}, {0, 0, 0,
    0}, {0, 0, 0, 0}, {LEFT_GRASS_EDGE,
    LEFT_UP_GRASS_EDGE, 0, 0}};
byte[][] gen6 = {{RIGHT_GRASS_EDGE,
    RIGHT_UP_GRASS_EDGE, 0, 0}, {0, 0, 0,
    0}, {0, 0, 0, 0}, {0, 0, 0, 0},
    {LEFT_GRASS_EDGE, LEFT_UP_GRASS_EDGE,
    0, 0}};
byte[][] gen7 = {{RIGHT_GRASS_EDGE,
    RIGHT_UP_GRASS_EDGE, 0, 0}, {0, 0, 0,
    0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0,
    0, 0}, {LEFT_GRASS_EDGE,
    LEFT_UP_GRASS_EDGE, 0, 0}};

lista[5] = new Gene(gen5);
lista[6] = new Gene(gen6);
lista[7] = new Gene(gen7);

//Abismos com pedras ( 3 tamanhos, 2
    variacoees)
byte[][] gen8 = {{RIGHT_GRASS_EDGE,
    RIGHT_UP_GRASS_EDGE, ROCK, 0}, {0, 0,
    0, 0}, {0, 0, 0, 0}, {LEFT_GRASS_EDGE,
    LEFT_UP_GRASS_EDGE, ROCK, 0}};
byte[][] gen9 = {{RIGHT_GRASS_EDGE,
    RIGHT_UP_GRASS_EDGE, ROCK, 0}, {0, 0,
    0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0},
    {LEFT_GRASS_EDGE, LEFT_UP_GRASS_EDGE,
    ROCK, 0}};
byte[][] gen10 = {{RIGHT_GRASS_EDGE,
    RIGHT_UP_GRASS_EDGE, ROCK, 0}, {0, 0,
    0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0,
    0, 0, 0}, {LEFT_GRASS_EDGE,
    LEFT_UP_GRASS_EDGE, ROCK, 0}};

lista[8] = new Gene(gen8);
lista[9] = new Gene(gen9);
lista[10] = new Gene(gen10);
```

```
byte[][] gen11 = {{GROUND, HILL_TOP, 0,
    0}, {GROUND, HILL_TOP, ROCK, 0},
    {RIGHT_GRASS_EDGE,
    RIGHT_UP_GRASS_EDGE, ROCK, ROCK}, {0,
    0, 0, 0}, {0, 0, 0, 0},
    {LEFT_GRASS_EDGE, LEFT_UP_GRASS_EDGE,
    ROCK, ROCK}, {GROUND, HILL_TOP, ROCK,
    0}, {GROUND, HILL_TOP, 0, 0}};
byte[][] gen12 = {{GROUND, HILL_TOP, 0,
    0}, {GROUND, HILL_TOP, ROCK, 0},
    {RIGHT_GRASS_EDGE,
    RIGHT_UP_GRASS_EDGE, ROCK, ROCK}, {0,
    0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0},
    {LEFT_GRASS_EDGE, LEFT_UP_GRASS_EDGE,
    ROCK, ROCK}, {GROUND, HILL_TOP, ROCK,
    0}, {GROUND, HILL_TOP, 0, 0}};
byte[][] gen13 = {{GROUND, HILL_TOP, 0,
    0}, {GROUND, HILL_TOP, ROCK, 0},
    {RIGHT_GRASS_EDGE,
    RIGHT_UP_GRASS_EDGE, ROCK, ROCK}, {0,
    0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0},
    {0, 0, 0, 0}, {LEFT_GRASS_EDGE,
    LEFT_UP_GRASS_EDGE, ROCK, ROCK},
    {GROUND, HILL_TOP, ROCK, 0}, {GROUND,
    HILL_TOP, 0, 0}};

lista[11] = new Gene(gen11);
lista[12] = new Gene(gen12);
lista[13] = new Gene(gen13);

//Vales sem inimigos (3 blocos de altura
    nas bordas, de 3 ate 5 de espacamento
    entre cada
byte[][] gen14 = {{GROUND, HILL_TOP, ROCK,
    ROCK, ROCK}, {GROUND, HILL_TOP, 0, 0,
    0}, {GROUND, HILL_TOP, 0, 0, 0},
    {GROUND, HILL_TOP, 0, 0, 0}, {GROUND,
    HILL_TOP, ROCK, ROCK, ROCK}};
byte[][] gen15 = {{GROUND, HILL_TOP, ROCK,
    ROCK, ROCK}, {GROUND, HILL_TOP, 0, 0,
```

```
    0}, {GROUND, HILL_TOP, 0, 0, 0},
    {GROUND, HILL_TOP, 0, 0, 0}, {GROUND,
    HILL_TOP, 0, 0, 0}, {GROUND, HILL_TOP,
    ROCK, ROCK, ROCK}};
byte[][] gen16 = {{GROUND, HILL_TOP, ROCK,
    ROCK, ROCK}, {GROUND, HILL_TOP, 0, 0,
    0}, {GROUND, HILL_TOP, 0, 0, 0},
    {GROUND, HILL_TOP, 0, 0, 0}, {GROUND,
    HILL_TOP, 0, 0, 0}, {GROUND, HILL_TOP,
    0, 0, 0}, {GROUND, HILL_TOP, ROCK,
    ROCK, ROCK}};

lista[14] = new Gene(gen14);
lista[15] = new Gene(gen15);
lista[16] = new Gene(gen16);

//Vales sem inimigos (Canos nas bordas,
    uma flor em um dos canos, de 3 ate 5
    de espacamento entre cada
byte[][] gen17 = {{GROUND, HILL_TOP,
    TUBE_SIDE_LEFT, TUBE_SIDE_LEFT,
    TUBE_TOP_LEFT}, {GROUND, HILL_TOP,
    TUBE_SIDE_RIGHT, TUBE_SIDE_RIGHT,
    TUBE_TOP_RIGHT}, {GROUND, HILL_TOP, 0,
    0, 0}, {GROUND, HILL_TOP, 0, 0, 0},
    {GROUND, HILL_TOP, 0, 0, 0}, {GROUND,
    HILL_TOP, TUBE_SIDE_LEFT,
    TUBE_SIDE_LEFT, TUBE_TOP_LEFT},
    {GROUND, HILL_TOP, TUBE_SIDE_RIGHT,
    TUBE_SIDE_RIGHT, TUBE_TOP_RIGHT}};
byte[][] gen18 = {{GROUND, HILL_TOP,
    TUBE_SIDE_LEFT, TUBE_SIDE_LEFT,
    TUBE_TOP_LEFT}, {GROUND, HILL_TOP,
    TUBE_SIDE_RIGHT, TUBE_SIDE_RIGHT,
    TUBE_TOP_RIGHT}, {GROUND, HILL_TOP, 0,
    0, 0}, {GROUND, HILL_TOP, 0, 0, 0},
    {GROUND, HILL_TOP, 0, 0, 0}, {GROUND,
    HILL_TOP, 0, 0, 0}, {GROUND, HILL_TOP,
    TUBE_SIDE_LEFT, TUBE_SIDE_LEFT,
    TUBE_TOP_LEFT}, {GROUND, HILL_TOP,
    TUBE_SIDE_RIGHT, TUBE_SIDE_RIGHT,
```

```
        TUBE_TOP_RIGHT}};
    byte[][] gen19 = {{GROUND, HILL_TOP,
        TUBE_SIDE_LEFT, TUBE_SIDE_LEFT,
        TUBE_TOP_LEFT}, {GROUND, HILL_TOP,
        TUBE_SIDE_RIGHT, TUBE_SIDE_RIGHT,
        TUBE_TOP_RIGHT}, {GROUND, HILL_TOP, 0,
        0, 0}, {GROUND, HILL_TOP, 0, 0, 0},
        {GROUND, HILL_TOP, 0, 0, 0}, {GROUND,
        HILL_TOP, 0, 0, 0}, {GROUND, HILL_TOP,
        0, 0, 0}, {GROUND, HILL_TOP,
        TUBE_SIDE_LEFT, TUBE_SIDE_LEFT,
        TUBE_TOP_LEFT}, {GROUND, HILL_TOP,
        TUBE_SIDE_RIGHT, TUBE_SIDE_RIGHT,
        TUBE_TOP_RIGHT}};

    lista[17] = new Gene(gen17);
    if (r.nextBoolean()) {
         lista[17].setSpriteTemplate(0, 4, new
             SpriteTemplate(Enemy.ENEMY_FLOWER,
             false));
    } else {
         lista[17].setSpriteTemplate(5, 4, new
             SpriteTemplate(Enemy.ENEMY_FLOWER,
             false));
    }

    lista[18] = new Gene(gen18);
    if (r.nextBoolean()) {
         lista[18].setSpriteTemplate(0, 4, new
             SpriteTemplate(Enemy.ENEMY_FLOWER,
             false));
    } else {
         lista[18].setSpriteTemplate(6, 4, new
             SpriteTemplate(Enemy.ENEMY_FLOWER,
             false));
    }

    lista[19] = new Gene(gen19);
    if (r.nextBoolean()) {
         lista[19].setSpriteTemplate(0, 4, new
             SpriteTemplate(Enemy.ENEMY_FLOWER,
```

```
        false));
} else {
    lista[19].setSpriteTemplate(7, 4, new
        SpriteTemplate(Enemy.ENEMY_FLOWER,
        false));
}

byte[][] gen20 = {{GROUND, HILL_TOP,
    TUBE_SIDE_LEFT, TUBE_SIDE_LEFT,
    TUBE_TOP_LEFT}, {GROUND, HILL_TOP,
    TUBE_SIDE_RIGHT, TUBE_SIDE_RIGHT,
    TUBE_TOP_RIGHT}};
byte[][] gen21 = {{GROUND, HILL_TOP,
    TUBE_SIDE_LEFT, TUBE_SIDE_LEFT,
    TUBE_TOP_LEFT}, {GROUND, HILL_TOP,
    TUBE_SIDE_RIGHT, TUBE_SIDE_RIGHT,
    TUBE_TOP_RIGHT}};

lista[20] = new Gene(gen20);
lista[21] = new Gene(gen21);

lista[21].setSpriteTemplate(0, 4, new
    SpriteTemplate(Enemy.ENEMY_FLOWER,
    false));

byte[][] gen22 = {{GROUND, HILL_TOP, 0, 0,
    0}, {GROUND, HILL_TOP, ROCK, ROCK,
    ROCK}, {GROUND, HILL_TOP, 0, 0, 0}};
lista[22] = new Gene(gen22);

byte[][] gen23 = {{GROUND, HILL_TOP, 0, 0,
    0}, {GROUND, HILL_TOP, ROCK, 0, 0},
    {GROUND, HILL_TOP, ROCK, ROCK, 0},
    {GROUND, HILL_TOP, ROCK, ROCK, ROCK},
    {GROUND, HILL_TOP, 0, 0, 0}, {GROUND,
    HILL_TOP, 0, 0, 0}, {GROUND, HILL_TOP,
    0, 0, 0}, {GROUND, HILL_TOP, ROCK,
    ROCK, ROCK}, {GROUND, HILL_TOP, ROCK,
    ROCK, 0}, {GROUND, HILL_TOP, ROCK, 0,
    0}, {GROUND, HILL_TOP, 0, 0, 0}};
lista[23] = new Gene(gen23);
```

```
byte[][] gen24 = {{GROUND, HILL_TOP, ROCK,
    0, 0}, {GROUND, HILL_TOP, ROCK, ROCK,
    0}, {GROUND, HILL_TOP, ROCK, ROCK,
    ROCK}};
byte[][] gen25 = {{GROUND, HILL_TOP, ROCK,
    ROCK, ROCK}, {GROUND, HILL_TOP, ROCK,
    ROCK, 0}, {GROUND, HILL_TOP, ROCK, 0,
    0}};
lista[24] = new Gene(gen24);
lista[25] = new Gene(gen25);

byte[][] gen26 = {{GROUND, HILL_TOP, 0, 0,
    0, 0, 0}, {GROUND, HILL_TOP, 0, 0,
    BLOCK_POWERUP, 0, 0}, {GROUND,
    HILL_TOP, 0, 0, 0, 0, 0}};
lista[26] = new Gene(gen26);

byte[][] gen27 = {{GROUND, HILL_TOP, 0, 0,
    COIN, 0, 0}, {GROUND, HILL_TOP, 0,
    COIN, COIN, COIN, 0}, {GROUND,
    HILL_TOP, 0, 0, COIN, 0, 0}};
lista[27] = new Gene(gen27);
byte[][] gen28 = {{GROUND, HILL_TOP, 0, 0,
    BLOCK_COIN, 0, 0},{GROUND, HILL_TOP,
    0, 0, 0, 0, 0}, {GROUND, HILL_TOP, 0,
    0, BLOCK_COIN, 0, 0},{GROUND,
    HILL_TOP, 0, 0, 0, 0, 0}, {GROUND,
    HILL_TOP, 0, 0, BLOCK_COIN, 0, 0}};
lista[28] = new Gene(gen28);

byte[][] gen29 = {{GROUND, HILL_TOP, 0, 0,
    0, 0, 0}, {GROUND, HILL_TOP, 0, 0,
    BLOCK_COIN, 0, 0}, {GROUND, HILL_TOP,
    0, 0, 0, 0, 0}, {GROUND, HILL_TOP, 0,
    0, BLOCK_COIN, 0, 0}, {GROUND,
    HILL_TOP, 0, 0, 0, 0, 0}, {GROUND,
    HILL_TOP, 0, 0, BLOCK_COIN, 0, 0},
    {GROUND, HILL_TOP, 0, 0, 0, 0, 0}};
lista[29] = new Gene(gen29);
lista[29].setSpriteTemplate(1, 2, new
```

```
            SpriteTemplate(r.nextInt(3),
            r.nextDouble() < 0.05));
        lista[29].setSpriteTemplate(2, 2, new
            SpriteTemplate(r.nextInt(3),
            r.nextDouble() < 0.05));
        lista[29].setSpriteTemplate(3, 2, new
            SpriteTemplate(r.nextInt(3),
            r.nextDouble() < 0.05));
        lista[29].setSpriteTemplate(4, 2, new
            SpriteTemplate(r.nextInt(3),
            r.nextDouble() < 0.05));

        geneList = lista;

        return lista;
    }

    public Gene getGene(int i) {

        Gene ret = new
            Gene(geneList[i].getPadrao());
        ret = tratarTemplate(ret, i);
        return ret;
    }

    private Gene tratarTemplate(Gene g, int i) {

        Random r = new Random();
        switch (i) {
            case 0:
                g.setSpriteTemplate(1, 2, new
                    SpriteTemplate(r.nextInt(3),
                    r.nextDouble() < 0.05));
                break;
            case 1:
                g.setSpriteTemplate(1, 2, new
                    SpriteTemplate(r.nextInt(3),
                    r.nextDouble() < 0.05));
                g.setSpriteTemplate(2, 2, new
                    SpriteTemplate(r.nextInt(3),
                    r.nextDouble() < 0.05));
```

```
        break;
    case 2:
        g.setSpriteTemplate(1, 2, new
            SpriteTemplate(r.nextInt(3),
            r.nextDouble() < 0.05));
        g.setSpriteTemplate(2, 2, new
            SpriteTemplate(r.nextInt(3),
            r.nextDouble() < 0.05));
        g.setSpriteTemplate(3, 2, new
            SpriteTemplate(r.nextInt(3),
            r.nextDouble() < 0.05));
        break;
    case 3:
        g.setSpriteTemplate(1, 2, new
            SpriteTemplate(r.nextInt(3),
            r.nextDouble() < 0.05));
        g.setSpriteTemplate(2, 2, new
            SpriteTemplate(r.nextInt(3),
            r.nextDouble() < 0.05));
        g.setSpriteTemplate(3, 2, new
            SpriteTemplate(r.nextInt(3),
            r.nextDouble() < 0.05));
        g.setSpriteTemplate(4, 2, new
            SpriteTemplate(r.nextInt(3),
            r.nextDouble() < 0.05));
        break;
    case 4:
        g.setSpriteTemplate(1, 2, new
            SpriteTemplate(r.nextInt(3),
            r.nextDouble() < 0.05));
        g.setSpriteTemplate(2, 2, new
            SpriteTemplate(r.nextInt(3),
            r.nextDouble() < 0.05));
        g.setSpriteTemplate(3, 2, new
            SpriteTemplate(r.nextInt(3),
            r.nextDouble() < 0.05));
        g.setSpriteTemplate(4, 2, new
            SpriteTemplate(r.nextInt(3),
            r.nextDouble() < 0.05));
        break;
    case 17:
```

```
        if (r.nextBoolean()) {
            g.setSpriteTemplate(0, 4, new
                SpriteTemplate(Enemy.ENEMY_FLOWER,
                false));
        } else {
            g.setSpriteTemplate(5, 4, new
                SpriteTemplate(Enemy.ENEMY_FLOWER,
                false));
        }
        break;
    case 18:

        if (r.nextBoolean()) {
            g.setSpriteTemplate(0, 4, new
                SpriteTemplate(Enemy.ENEMY_FLOWER,
                false));
        } else {
            g.setSpriteTemplate(6, 4, new
                SpriteTemplate(Enemy.ENEMY_FLOWER,
                false));
        }
        break;
    case 19:

        if (r.nextBoolean()) {
            g.setSpriteTemplate(0, 4, new
                SpriteTemplate(Enemy.ENEMY_FLOWER,
                false));
        } else {
            g.setSpriteTemplate(7, 4, new
                SpriteTemplate(Enemy.ENEMY_FLOWER,
                false));
        }
        break;
    case 21:
        g.setSpriteTemplate(0, 4, new
            SpriteTemplate(Enemy.ENEMY_FLOWER,
            false));
        break;

    case 29:
```

```
                g.setSpriteTemplate(1, 2, new
                    SpriteTemplate(r.nextInt(3),
                    r.nextDouble() < 0.05));
                g.setSpriteTemplate(2, 2, new
                    SpriteTemplate(r.nextInt(3),
                    r.nextDouble() < 0.05));
                g.setSpriteTemplate(3, 2, new
                    SpriteTemplate(r.nextInt(3),
                    r.nextDouble() < 0.05));
                g.setSpriteTemplate(4, 2, new
                    SpriteTemplate(r.nextInt(3),
                    r.nextDouble() < 0.05));
                break;
            default:
                return g;
        }

        return g;
    }

}
```

Classe que controla o calculo do fitness, mutação e cruzamento:

```
package tcc.klauskuhr.AG.data;

import dk.itu.mario.engine.sprites.SpriteTemplate;
import dk.itu.mario.level.Level;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import tcc.klauskuhr.AG.entidades.Fase;

/**
 *
 * @author Klaus
 */
public class Controlador {

    private Gene[] genes;
    private Random r = new Random();
```

```
private Gerador gera;

public Controlador(Gene[] genes, Gerador
    gerador) {
     this.genes = genes;
     gera = gerador;
}

public byte[][] getGene(int i) {
     return genes[i].getPadrao();
}

public void resetGene(int i)
{
     genes[i] = gera.getGene(i);

}

public SpriteTemplate getST(int i, int x, int
    y) {

     return genes[i].getSpriteTemplate(x, y);
}

public double calcFitness(int[] cromossomo) {

     double fitness = 0;
     int[] repeticao = new int[30];

     //Sao 30 padroes observados, logo, e
         necessario uma matriz 23x23
     //double[][] mat = new double[30][30];
     double[][] mat = {

     //    0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 ,
         9 , 10, 11, 12, 13, 14, 15, 16, 17,
         18, 19, 20, 21, 22, 23, 24, 25, 26,
         27, 28,29
          //0, 1 inimigo
          {0.1,0.1,0.1,0.1, 1 , 1 , 1 , 1 , 1 ,
              1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1
```

```
    , 1 , 1 , 2 , 2 , 2 , 1 , 2 , 2 ,
    1 , 1 ,1.5, 1},
//1, 2 inimigos
{0.1,0.1,0.1,0.1, 1 , 1 , 1 , 1 , 1 ,
    1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1
    , 1 , 1 , 2 , 2 , 2 , 1 , 2 , 2 ,
    1 , 1 ,1.5, 1},
//2, 3 inimigos
{0.1,0.1,0.1,0.1, 1 , 1 , 1 , 1 , 1 ,
    1 , 1 , 1 , 1 , 1 , 1 ,1.5, 1 , 1
    , 1 , 1 , 2 , 2 , 2 , 1 , 2 , 2 ,
    1 , 1 ,1.5, 1},
//3, 4 inimigos
{0.1,0.1,0.1,0.1, 1 , 1 , 1 , 1 , 1 ,
    1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1
    , 1 , 1 , 2 , 2 , 2 , 1 , 2 , 2 ,
    1 , 1 ,1.5, 1},
//4, 4 inimigos e recompensa
{ 1 , 1 , 1 , 1 , 1 , 2 , 2 , 2 , 2 ,
    2 , 2 , 2 , 2 , 2 , 1 , 1 , 1 , 1
    , 1 , 1 , 2 , 2 , 2 , 1 , 2 , 2 ,
    1 , 1 , 1 , 1},
//5, abismo tamanho 2
{1.9,1.9,1.9,1.9,1.9,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.
    1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1
    , 1 , 1 , 1 , 1 , 1 , 1 , 1},
//6, abismo tamanho 3
{1.9,1.9,1.9,1.9,1.9,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.
    1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1
    , 1 , 1 , 1 , 1 , 1 , 1 , 1},
//7, abismo tamanho 4
{1.9,1.9,1.9,1.9,1.9,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.
    1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1
    , 1 , 1 , 1 , 1 , 1 , 1 , 1},
//8, abismo com pedra tamanho 2
{1.9,1.9,1.9,1.9,1.9,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.
    1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1
    , 1 , 1 , 1 , 1 , 1 , 1 , 1},
//9, abismo com pedra tamanho 3
{1.9,1.9,1.9,1.9,1.9,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.
    1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1
```

```
    , 1 , 1 , 1 , 1 , 1 , 1 , 1},
//10, abismo com pedra tamanho 4
{1.9,1.9,1.9,1.9,1.9,0.1,0.1,0.1,0.1,0.1,0.1,0.
    1 , 1 , 1 , 1 , 1 , 1
    ,1.5,1.5,1.5, 1 , 1 , 1 , 1 , 1 ,
    1 , 1},
//11, abismo com escada tamanho 2
{1.9,1.9,1.9,1.9,1.9,0.1,0.1,0.1,0.1,0.1,0.1,0.
    1 , 1 , 1 , 1 , 1 , 1
    ,1.5,1.5,1.5, 1 , 1 , 1 , 1 , 1 ,
    1 , 1},
//12, abismo com escada tamanho 3
{1.9,1.9,1.9,1.9,1.9,0.1,0.1,0.1,0.1,0.1,0.1,0.
    1 , 1 , 1 , 1 , 1 , 1
    ,1.5,1.5,1.5, 1 , 1 , 1 , 1 , 1 ,
    1 , 1},
//13, abismo com escada tamanho 4
{1.9,1.9,1.9,1.9,1.9,0.1,0.1,0.1,0.1,0.1,0.1,0.
    1 , 1 , 1 , 1 , 1 , 1
    ,1.5,1.5,1.5, 1 , 1 , 1 , 1 , 1 ,
    1 , 1},
//14 Vale de 3 sem inimigos (3 blocos
    de altura nas bordas)
{ 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 ,
    1 , 1 , 1 , 1 , 1
    ,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.
    1 , 1 , 1 , 1},
//15 Vale de 4 sem inimigos (3 blocos
    de altura nas bordas)
{ 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 ,
    1 , 1 , 1 , 1 , 1
    ,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.
    1 , 1 , 1 , 1},
//16 Vale de 5 sem inimigos (3 blocos
    de altura nas bordas)
{ 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 ,
    1 , 1 , 1 , 1 , 1
    ,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.
    1 , 1 , 1 , 1},
//17 Vale de 3 com flor (Canos nas
    bordas)
```

```
{ 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 ,
    1 , 1 , 1 , 1 , 1
    ,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1
    1 , 1 , 1 , 1},
//18 Vale de 4 com flor (Canos nas
    bordas)
{ 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 ,
    1 , 1 , 1 , 1 , 1
    ,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1
    1 , 1 , 1 , 1},
//19 Vale de 5 com flor (Canos nas
    bordas)
{ 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 ,
    1 , 1 , 1 , 1 , 1
    ,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1
    1 , 1 , 1 , 1},
//20 Cano solitario
{ 2 , 2 , 2 , 2 , 2 , 1 , 1 , 1 , 1 ,
    1 , 1 , 1 , 1 , 1
    ,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1
    1 , 1 , 1 , 1},
//21 Cano solitario com flor
{ 2 , 2 , 2 , 2 , 2 , 1 , 1 , 1 , 1 ,
    1 , 1 , 1 , 1 , 1
    ,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1
    1 , 1 , 1 , 1},
//22 Pilar de 3 solitario
{ 2 , 2 , 2 , 2 , 2 , 1 , 1 , 1 , 1 ,
    1 , 1 , 1 , 1 , 1
    ,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1
    1 , 1 , 1 , 1},
//23 Vale de escadas, 3 de altura, 3
    de espacamento
{ 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 ,
    1 , 1 , 1 , 1 , 1
    ,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1
    1 , 1 , 1 , 1},
//24 Escada virada para frente, 3 de
    altura
{ 2 , 2 , 2 , 2 , 2 , 1 , 1 , 1 , 1 ,
    1 , 1 , 1 , 1 , 1
```

```
,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.
    1 , 1 , 1 , 1},
//25 Escada virada para tras, 3 de
    altura
{ 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 ,
    1 , 1 , 1 , 1 , 1
    ,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.
    1 , 1 , 1 , 1},
//26 Bloco de powerUp solitario
{ 2 , 2 , 2 , 1 , 1 , 1 , 1 , 1 , 1 ,
    1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1
    , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1
    ,0.1,0.1,0.1,0.1},
//27 Moedas
{ 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 ,
    1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1
    , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1
    ,0.1,0.1,0.1,0.1},
//28 Blocos de moedas
{ 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 ,
    1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1
    , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1
    ,0.1,0.1,0.1,0.1},
//29 Blocos de moedas com inimigos
{ 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 ,
    1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1
    , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1
    ,0.1,0.1,0.1,0.1}
};

int aux = cromossomo.length-1;
for (int i = 0; i < aux; i++) {
    //Fitness e o somatorio da
        multiplicacao por alpha da fitness
        2 a 2 de 2 padroes subsequentes.
    fitness
        +=mat[cromossomo[i]][cromossomo[i+1]];

    repeticao[cromossomo[i]]++;
}
   for (int i : repeticao) {
```

```
            if(i>aux*0.15)
                fitness -= i*2;
        }


        //System.out.println(fitness);
        return fitness;
    }

    public int[][] crossover(int[] cromoA, int[]
        cromoB) {
        //Primeiro descobre-se qual e o tamanho do
            menor cromossomo
        int l = cromoA.length;
        int aux = cromoB.length;

       //  System.out.println("Tamanho de
          cromossomo: " + l + " e "+aux);

        if (cromoB.length <= cromoA.length) {
            l = cromoB.length;
            aux = cromoA.length;
        }
        //Gera-se o objeto a ser retornado, de
            modo que o menor cromossomo esta na
            posicao 0 e o maior na 1.
        int[][] ret = new int[2][];
        if (cromoA.length <= cromoB.length) {
            ret[0] = cromoA;
            ret[1] = cromoB;

        } else {
            ret[0] = cromoB;
            ret[1] = cromoA;
        }

       //Para o CrossOver propriamente dito,
          precisamos de um ponto de CrossOver
          menor que o tamanho do menor gene e a
          determinacao de que gene vai para qual
          retorno.
```

```
        int pco = r.nextInt(l);

        for (int i = 0; i < pco; i++) {
            aux = ret[0][i];
            ret[0][i] = ret[1][i];
            ret[1][i] = aux;

        }

        /*
        for (int i = pco; i < l; i++) {

                aux = ret[0][i];
                ret[0][i] = ret[1][i];
                ret[1][i] = aux;

        }
        */
        //Mutacao
        aux = ret[0].length;
        for (int i = 0; i < aux ; i++) {
            if(calcMuta())
            {
                ret[0][i] = this.getRandomGene();
            }
        }
        aux = ret[1].length;
        for (int i = 0; i < aux ; i++) {
            if(calcMuta())
            {
                ret[1][i] = this.getRandomGene();
            }
        }


        return ret;
    }

    public int getLenghtGene(int i) {
        return genes[i].getLength();
    }
```

```
    public int getRandomGene() {
        return r.nextInt(genes.length);

    }

    public int genesRandomMenorQue(int i) {
        if (i > 10) {
            return getRandomGene();
        }
        List<Integer> aux = new ArrayList<>();

        for (int j = 0; j < genes.length; j++) {
            if (genes[j].getLength() < i) {
                aux.add(j);
            }
        }
        if (aux.size() != 0) {
            return aux.get(r.nextInt(aux.size()));
        }

        return -1;

    }


    private boolean calcMuta(){
        return r.nextDouble()<0.001;



    }


}
```

Classe da Fase

```
/*
 * To change this license header, choose License
     Headers in Project Properties.
 * To change this template file, choose Tools |
```

```
    Templates
 * and open the template in the editor.
 */
package tcc.klauskuhr.AG.entidades;

import dk.itu.mario.MarioInterface.*;
import dk.itu.mario.engine.sprites.SpriteTemplate;
import dk.itu.mario.level.*;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import tcc.klauskuhr.AG.data.Controlador;

/**
 *
 * @author Klaus
 */
public class Fase extends RandomLevel implements
   LevelInterface {

    private List<Integer> genoma;

    private Controlador cont;
    private Random random = new Random();
    private int length;

    public Fase(int width, int height, Controlador
       controlador) {
        super(width, height);
        cont = controlador;
        genoma = new ArrayList<Integer>();
        inicializar();
    }

    public Fase(int width, int height, Controlador
       controlador, int[] gens, boolean g) {
        super(width, height);
        genoma = new ArrayList<Integer>();
        for (int gen : gens) {
            genoma.add(gen);
        }
```

```
        cont = controlador;
        inicializarGeracao(g);
    }

    public void inicializar() {
        //"Ponteiro" para a posicao atual de
           geracao
        length = 0;

        //Primeiramente e gerada a plataforma
           inicial.
        buildStraight(0, false);

        //e feita uma selecao aleatoria de genes,
           ate 300 blocos
        int aux = 0;

        while (length < width - 40) {
            aux = cont.genesRandomMenorQue(width -
               40 - length);
            if (aux == -1) {
                break;
            }
            genoma.add(aux);

           //
              this.AcrescerAFase(genoma.get(genoma.size()
              - 1), length);
            length +=
               cont.getLenghtGene(genoma.get(genoma.size()
               - 1));
        }

        buildStraight(length, true);

        yExit = height - 2;
        xExit = length - 15;

    }

    private void buildStraight(int xo, boolean f) {
```

```
        length += 10 /** random.nextInt(5)*/;

        int floor = height - 2;
        // int floor = ;

        //runs from the specified x position to
            the length of the segment
        if (!f) {
            for (int x = xo; x < length; x++) {
                for (int y = 0; y < height; y++) {
                    if (y == floor) {
                        setBlock(x, y, HILL_TOP);
                    }
                    if (y > floor) {
                        setBlock(x, y, GROUND);
                    }
                }
            }
        } else {
            for (int x = xo; x < length + 10; x++)
                {
                for (int y = 0; y < height; y++) {
                    if (y == floor) {
                        setBlock(x, y, HILL_TOP);
                    }
                    if (y > floor) {
                        setBlock(x, y, GROUND);
                    }
                }
            }
        }

    }

    //Recebe a fase, o indice do gene a ser
        inserido e a posicao no eixo X a inserir.
    private void AcrescerAFase(int gene, int pos) {
        int floor = height - 1;
        byte[][] insert = cont.getGene(gene);
        cont.resetGene(gene);
        int size = insert.length;
```

```
        int altura = insert[0].length;

        int aux = 0;
        int auy = 0;

        for (int i = pos; i < pos + size; i++) {
            auy = 0;
            for (int j = 0; j < altura; j++) {

                this.setBlock(i, floor - j,
                    insert[aux][auy]);
                this.setSpriteTemplate(i, floor -
                    j, cont.getST(gene, aux, auy));
                auy++;
            }
            aux++;
        }
    }

    private void inicializarGeracao(boolean g) {
        //"Ponteiro" para a posicao atual de
           geracao
        length = 0;

        //Primeiramente e gerada a plataforma
           inicial.
        buildStraight(0, false);

        while (this.tamanhoGenes() > width - 60) {
            this.removerUltimoGene();
        }

        //Constroi a fase com base nos genes
        for (int i = 0; i < genoma.size(); i++) {
            if (g) {
                this.AcrescerAFase(genoma.get(i),
                    length);
            }
            length +=
                cont.getLenghtGene(genoma.get(i));
```

```
        }

        buildStraight(length, true);

        yExit = height - 2;
        xExit = length - 6;

    }

    public double calcFitness() {
        int[] aux = getGenoma();

        return cont.calcFitness(aux);
    }

    public int[] getGenoma() {

        int[] aux = new int[genoma.size()];

        for (int i = 0; i < aux.length; i++) {
            aux[i] = genoma.get(i);
        }

        return aux;

    }

    private int tamanhoGenes() {
        int ret = 0;

        for (int i = 0; i < genoma.size(); i++) {
            ret +=
                cont.getLenghtGene(genoma.get(i));
        }
        return ret;

    }

    private void removerUltimoGene() {
        genoma.remove(genoma.size() - 1);
    }
```

```
}
```

Classe para a escrita dos resultados da execução

```
package tcc.klauskuhr.AG;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

/**
 *
 * @author Klaus
 */
public class Printf {

    private String fileName;

    public Printf() {
        fileName = "temporario.txt";
    }

    public void write(String[] texto, String
        fileName) {
        this.fileName = fileName;
        try {
            FileWriter fileWriter
                    = new FileWriter(fileName,
                        true);

            BufferedWriter bufferedWriter = new
                BufferedWriter(fileWriter);


            for (String string : texto) {
                bufferedWriter.write(string);
                bufferedWriter.newLine();
            }
```

```
            //bufferedWriter.write(texto);
            //bufferedWriter.newLine();

            bufferedWriter.close();
        } catch (IOException ex) {
            System.out.println(
                    "Error writing to file '"
                    + fileName + "'");

        }
    }
    public void novaLinha() {

        try {
            FileWriter fileWriter
                    = new FileWriter(fileName,
                      true);

            BufferedWriter bufferedWriter = new
               BufferedWriter(fileWriter);

            bufferedWriter.newLine();

            bufferedWriter.close();
        } catch (IOException ex) {
            System.out.println(
                    "Error writing to file '"
                    + fileName + "'");

        }
    }


}
```

7.2 ARTIGO

Técnicas IA para a criacão procedural de conteúdo espacial em jogos

Klaus Victor Kühr[1], Jerusa Marchi[1]

[1]Departamente de Informatica e Estatistica (INE) – Universidade Federal de Santa Catarina (UFSC)

Klaus.kuhr@grad.ufsc.br, jerusa.marchi@ufsc.br

Abstract. *As the games industry grows, so too does the competition between developers and the expectations from the customer base. The developing costs for a team of full-time developers may be exceedingly high for many start-ups and the customer's ever-growing demand for high quality content for the lowest possible price put many teams on the spot. To solve this demand from both the market and competitors, it may be of interest to invest in the full or partial automation of content generation. A study was done to expose these techniques to both the academic and gaming communities.*

Resumo. *A medida que o mercado de jogos cresce, a competição entre os desenvolvedores e as exigências dos consumidores também sobe. O custo por hora de um time de desenvolvimento pode tornar um jogo proibitivamente caro para muitas empresas e as demandas dos consumidores por mais conteúdo por menos dinheiro fazem grande pressão. Para solucionar a demanda competitiva e de mercado, pode ser de interesse a automatização da produção de conteúdo. Tendo isso em mente, foi realizado um levantamento de diferentes técnicas utilizáveis na criação procedural de espaços em jogos digitais.*

1. Introdução

Jogos acompanharam a humanidade desde o começo da sua história. Pelos séculos, seres humanos criaram maneiras de competirem entre si de maneira não violenta[Tylor 1879]. Esportes são exemplos de jogos, assim como o xadrez, a bocha, oesconde-esconde, os videogames. Essencialmente, toda pessoa já jogou ou vai jogaralgum jogo um dia [Craig 2002].

Jogos são elementos culturais importantes que possuem elementos próprios, capazes de distinguir uns dos outros. Um jogo é composto basicamente por regras, objetivos, participantes e opcionalmente, aparatos próprios[Kramer 2000]. O xadrez, por exemplo, possui como regras: as regras de movimento, a ordem das jogadas e as regras de xeque e seu objetivo é a tomada da peça rei do oponente. Os participantes são dois jogadores. Os aparatos são as peças e o tabuleiro.

Com a invenção dos computadores e a eventual concepção de jogos digitais, o estudo de jogos cresceu de forma acelerada. Uma indústria bilionária formou-se ao redor dos videogames e eles tornaram-se o passatempo predileto das novas gerações.

Com um mercado aquecido, desenvolvedores competem entre si para criar os novos best sellers [Williams 2002].

Com um custo elevado da mão de obra, muitas empresas se voltaram à técnicas de IA para gerar conteúdo para seus jogos e fornecerem mais coisas para seus clientes com um custo menor. A geração procedural de conteúdo, *PCG* (do inglês *Procedural Content Generation*), é justamente isso.

O *PCG* é uma técnica que quando usada corretamente pode facilitar o desenvolvimento de jogos ou até torná-los mais atraentes aos jogadores. Com um elemento de aleatoriedade, um jogo pode ser jogado várias vezes pela mesma pessoa e cada vez a experiência ser diferente. Por exemplo: Em um jogo de desvendar mistérios, para um jogador uma porta pode ser vermelha e aberta com uma chave triangular, encontrada na coleira de um cachorro. Para outro jogador, jogando o mesmo jogo, esta porta pode ser azul, com uma chave encontrada atrás de um quadro.

Conforme Hendrikx *et al.* [Hendrikx 2013], o conteúdo possível de ser automatizado em um jogo pode ser dividido em: *Game Bits, Game Space, Game Systems, Game Scenarios Game Design* e *Derived, Content*. Neste trabalho será tratado o *Game Space*:

- *Game Space* é o espaço de interação do jogador com o jogo. Ele também pode ser dividido em abstrato e concreto, além de suas sub definições, como espaços internos e externos, por exemplo. O tabuleiro de xadrez é um espaço abstrato, supostamente representando um campo de batalha entre dois reinados ou famílias inimigas. Espaços concretos são espaços notavelmente reconhecidos por humanos, seguindo na linha de como nós percebemos a realidade, como por exemplo uma sala com mesas e cadeiras.

2. Introdução ao *PCG*

PCG pode ser definido como o uso de um algoritmo formal para gerar um conteúdo que seria produzido normalmente por um ser humano [Smith 2015]. No campo de estudo de jogos, a pesquisa de *PCG* é uma subdivisão do campo de estudos de inteligência artificial em jogos, visto primariamente como uma solução para os problemas de replicar a criatividade e visão de um designer humano, como avaliar a qualidade de um conteúdo existente e como criar sistemas capazes de interagir com um designer de jogos, auxiliando-o.

Apesar de ser uma ferramenta utilizada na indústria de jogos digitais, suas raízes estão em jogos de tabuleiro e papel e caneta. Um dos primeiros surgimentos de *PCG* data de 1976, em um suplemento de *Dungeons & Dragons* [Gygax 1974], que é um jogo de interpretação de papéis. No jogo, os jogadores são aventureiros em um mundo medieval fantástico, povoado por dragões, intrigas, calabouços, monstros e masmorras.

Um aspecto importante de um jogo de *Dungeons & Dragons* é a exploração de lugares perigosos, normalmente subterrâneos. Esses lugares são montados por um dos jogadores, que é o narrador da história. Esse narrador define o espaço de jogo e o que os jogadores irão encontrar, possíveis passagens secretas, monstros e tesouros. É um processo que pode ser demorado e muitas vezes feito em vão, se os jogadores nunca chegarem a entrar na caverna em primeiro lugar.

O suplemento *Dungeons Geomorphs* [TSR 1976] buscava ajudar o narrador a montar esses lugares de maneira simplificada. Ele dispunha de pedaços de mapas, que poderiam ser embaralhados pelo narrador e expostos aos jogadores à medida que eles exploravam o ambiente. O suplemento informava ao narrador como fazer o uso correto do mesmo, e com essa informação o narrador conseguiria gerar um número quase infinito de masmorras diferentes.

O avanço da tecnologia e os computadores pessoais trouxe o *PCG* para o meio digital. Alguns dos primeiros jogos procedurais foram jogos de aventura e ação inspirados nas regras procedurais de *Dungeons & Dragons*. *Rogue* é um dos jogos mais memoráveis desta era, gerando o termo em inglês *roguelike* (similar ao *Rogue*), que até hoje é utilizado na indústria para categorizar jogos com conteúdo gerado proceduralmente, diferenças entre cada partida e sem a capacidade de "voltar" uma jogada para impedir a morte do personagem.

A Figura 1 é uma tela de *Rogue*, mostrando uma masmorra criada proceduralmente, de acordo com as regras implementadas pelos desenvolvedores:

Figura 1. *Rogue*: exemplo de uma masmorra[1]

3. Técnicas aplicadas a geração de conteúdo

Apesar de ser um artefato relativamente antigo para a formulação de conteúdo em jogos, o estudo acadêmico de *PCG* é mais recente do que o esperado. Muitas implementações anteriores eram puramente aleatórias, sem uma intenção definida. Com a diversificação das técnicas de IA, pesquisadores foram capazes de encontrar soluções para problemas

[1]Fonte: https://commons.wikimedia.org/w/index.php?curid=36978065. Acesso em: 20/06/2018

antigos de *PCG*. Algumas das soluções para o *PCG* foram algoritmos evolutivos, baseados em regras, *Design Patterns* e *Machine Learning*, os quais serão apresentados a seguir:

3.1. Baseados em algoritmos evolutivos

Algoritmos baseados em algoritmos evolutivos comumente utilizados em *PCG*, também chamados de baseados em Buscas (*SBPCG*) neste contexto, foram objetos de estudo por Togelius *et al.* [Togelius 2011b]. Eles são compostos por populações, indivíduos e seus genótipos e fenótipos, além de uma função de avaliação. Sua maneira de formulação de *Game Space* é por meio de geração de indivíduos e avaliação do conteúdo gerado, até que seja satisfatório ou que outra condição seja atingida. O motivo de não utilizar o nome "evolutivo" na literatura original para este método é para evitar a exclusão de meta-heurísticas comuns, como *Simulated Annealing* e Otimização por enxame de partículas, por exemplo. A Figura 2 representa o funcionamento do algoritmo evolutivo:

Figura 2. Funcionamento de um algoritmo evolutivo

Inicialmente, genes são mapeados às expressões dos mesmos. Os genes, que podem sofrer mutações, podem ser mapeados diretamente ou indiretamente aos fenótipos. Um exemplo de mapeamentos são, do mais direto para o mais indireto:

- Uma célula de uma masmorra, podendo ser uma parede ou espaço vazio.
- Uma parede inteira, tanto em largura e altura.
- Um padrão utilizável em uma sala, como uma janela, cortinas e paredes ao redor da mesma.
- Uma lista de propriedades desejáveis de uma sala, como número de portas, janelas, tapetes e inimigos.
- Um número aleatório.

Uma propriedade que deve ser levada em consideração na escolha do mapeamento é a localidade. A localidade de um gene é capaz de informar o tamanho da mudança que a alteração em um gene apenas é capaz de afetar a fase como um todo. Quanto mais direto o mapeamento, maior será a sua localidade. Por exemplo, a alteração de uma parede lisa por uma parede com uma fonte de iluminação não altera a fase drasticamente, porém uma mudança no número de salas e inimigos alteraria a experiência do jogador.

Outra propriedade que deve ser considerada é a dimensão da fase. Se uma fase for composta por muitos genes, é possível que o cálculo da geração da mesma tome um tempo muito elevado, inviabilizando o processo em tempo de execução ou em desenvolvimento. Por exemplo, se for escolhido um algoritmo de alta localidade e mapeamento direto em que cada gene é uma célula, se uma fase for composta por uma

matriz de 1000x1000 células, a dimensionalidade pode causar a inviabilização do algoritmo.

Indivíduos, por sua vez, são compostos por genes. A cada geração, indivíduos são gerados com base na população e suas aptidões são calculadas. Indivíduos com aptidões altas são selecionados para a geração da próxima geração, por meio de cruzamentos com outros indivíduos e mutações aleatórias de genes, assim sucessivamente até que a fitness máxima ou número máximo de gerações seja excedido ou a população convergir e estagnar.

3.2. Baseados em regras

A geração de conteúdo baseada em regras pode ser vista como uma resposta alternativa ao comportamento do jogador. No percurso de uma fase uma sequência de desafios é proposta perante o jogador, que responde aos desafios da maneira que considera mais apropriada. A geração de conteúdo com base em regras toma as ações do jogador como base para geração de conteúdo.

Por exemplo, se no decorrer de uma fase o jogador tende a desviar dos inimigos e o gerador toma os movimentos do jogador como entrada, ele poderá na próxima fase gerada criar caminhos alternativos para o jogador, ou reduzir os mesmos, forçando o confronto.

Togelius et al. [Togelius 2011a] implementaram um sistema de geração de fases em *Super Mario Bros* [Nintendo 1985] baseado em regras e com duas versões, uma offline e uma online. As entradas dos sistemas são as ações do jogador. Por exemplo quando ele aperta o botão de pular ou deixa de apertá-lo. Essas ações são gravadas numa linha de tempo e posição e com base nelas o gerador altera a próxima fase, colocando mais ou menos inimigos e alterando a posição de blocos e espaços vazios.

No método apresentado, o modo offline apresentava a geração de conteúdo apenas na próxima fase, enquanto o modo online além das alterações na próxima fase, alterava a fase atual, criando monstros em resposta à coleta de moedas e moedas em resposta à derrota de monstros.

3.3. Baseados em *Machine Learning*:

O reaparecimento de redes neurais sob a alcunha de *Deep Learning* e a utilização de *Big Data* para treinamento das mesmas serviu para a cogitação e implementação de técnicas de geração de conteúdo espacial em jogos.

Dos usos possíveis para *PCG*, a abordagem baseada em *Machine Learning PCGML* demonstra excelência atualmente, dentre outras áreas, na Geração Automática e Coautoria de Conteúdo.

No caso da Geração Automática (não supervisionada), diferentemente de outros métodos de *PCG*, como o *SBPCG*, o desenvolvedor não precisa codificar extensivamente a natureza do problema por meio de fórmulas de fitness, por exemplo. Basta ele treinar o algoritmo com exemplos do que ele deseja e o algoritmo será capaz de gerar conteúdo.

Para a Coautoria de Conteúdo, é possível o desenvolvedor utilizar ferramentas de *machine learning* para completar seções do jogo, agilizando o processo de desenvolvimento. Por exemplo uma fase pode ter seu início, fim e trechos feitos pelo

desenvolvedor, com as áreas deixadas em branco preenchidas pelo algoritmo. Uma vez preenchidas essas áreas, o desenvolvedor avalia se foi aceitável ou não.

Summervile et al. [Summerville 2017] organizaram as técnicas de *PCGML* com base em duas características: A representação dos dados e técnica de treinamento. As representações possíveis são:

- Sequências: Os dados são sequências de caracteres, como textos ou fases de jogos simples.
- Matrizes: Dados são representados como informações em matrizes. Geralmente sendo matrizes de 2 dimensões, capazes de representar a posição de cada elemento nela contido. Muito utilizado para representação de Fases.
- Grafos: Representações gerais de dados. Essas representações são genéricas, porém normalmente custosas pela falta de estruturas intrinsecamente definidas.

As diferentes técnicas de treinamento dependem da representação para que sejam efetivas ou utilizáveis. As técnicas levadas em consideração foram: *Backpropagation, Evolution, Frequency Counting, Expectation Maximization,* e *Matrix Factorization.*

Dahlskog et al. [Dahlskog 2014] dividiram fases do clássico *Super Mario Bros* [Nintendo 1985] em fatias e transformaram-nas em sequências de treinamento para cadeias de Markov em n-gramos. Um n-gramo é uma sequência de valores de tamanho 'n', onde cada posição é predita com base nos 'n' valores anteriores à mesma.

Para o teste dos resultados, foram feitas fases onde o 'n' variava de 0 até 3. Foi observado que quando n=0, as fases eram sequências aleatórias de fatias, muitas vezes impossíveis de serem completadas pelo jogador e à medida que 'n' era aumentado, mais atraente e jogável as fases se tornavam.

3.4. Baseados em *Design Patterns*

Design Patterns são maneiras de estruturar o *design* e o processo do mesmo em elementos recorrentes [Dahlskog 2012]. Originado na década de 1970 no campo da Arquitetura, eles foram definidos como problemas que o arquiteto poderia encontrar em seus projetos e como solucioná-los.

Um exemplo simples para um *Design Pattern* é: Em uma festa não há comida suficiente para todos os convidados. A solução é tornar a relação comida / convidado maior. Uma resolução que pode ser tomada é aumentar a quantidade de comida, outra é reduzir o número de convidados.

A ideia era a resolução rápida de problemas, possibilitando que o arquiteto se concentre em outros aspectos do processo de *design*. Eventualmente esta solução foi adaptada para soluções no campo de desenvolvimento de *Software* [Dahlskog 2012].

O *design* de jogos adaptou-os ao seu campo. Em jogos, *Design Patterns* podem ser vistos como respostas para problemas do jogo ou do *designer* de fases, assim como um problema a ser resolvido pelo jogador e proposto pelo *designer*.

As aplicações de *design patterns* em *Game Space PCG* são duas: Como blocos de conteúdo que podem ser alinhados em uma, duas ou mais dimensões, gerando o conteúdo a ser jogado. A outra aplicação é como avaliação de conteúdo gerado por outros métodos, como o baseado em buscas.

4. Conclusões

Este trabalho serviu para abordar este tema que pode se beneficiar da atenção da comunidade acadêmica. Mais estudos nesta área serviriam para otimizar os processos e gerar uma redução do custo de produção de jogos, tanto eletrônicos quanto tradicionais.

Game Space PCG demonstrou ser um amplo campo de estudos, com alto potencial para aplicação de técnicas de inteligência artificial. A emulação da criatividade humana é algo possível nesta área, que pode servir como fronteira de estudos por muitos anos.

Referências

Craig, S. (2002). SPORTS AND GAMES OF THE ANCIENTS. Greenwood Press, Westport, EUA.

Dahlskog, S. and Togelius, J. (2012). Patterns and procedural content generation: Revisiting mario in world 1 level 1. In Proceedings of the First Workshop on Design Patterns in Games, DPG '12, pages 1–8, New York, NY, USA. ACM.

Dahlskog, S., Togelius, J., and Nelson, M. J. (2014). Linear levels through n-grams. In Lugmayr, A., editor, MindTrek, pages 200–206. ACM.

Gygax, G. and Arneson, D. (1974). Dungeons & dragons: Rules for fantastic medieval wargames campaigns playable with paper and pencil and miniature figures.

Hendrikx, M., Meijer, S., Van Der Velden, J., and Iosup, A. (2013). Procedural content generation for games: A survey. ACM Trans. Multimedia Comput. Commun. Appl., 9(1):1–22.

Kramer, W. (2000). What is a game? The Games Journal. Nintendo (1985). Super mario bros.

NINTENDO. Jogo Digital, Super Mario Bros. 1985.

Smith, G. (2015). An analog history of procedural content generation. In Proceedings of the 10th International Conference on the Foundations of Digital Games, pages 22–25.

Summerville, A., Snodgrass, S., Guzdial, M., Holmgard, C., Hoover, A. K., Isaksen, ˚ A., Nealen, A., and Togelius, J. (2017). Procedural content generation via machine learning (pcgml). CoRR, abs/1702.00539.

Togelius, J., Kastbjerg, E., Schedl, D., and Yannakakis, G. N. (2011a). What is procedural content generation?: Mario on the borderline. In Proceedings of the 2Nd International Workshop on Procedural Content Generation in Games, PCGames '11, pages 3:1–3:6, New York, NY, USA. ACM.

Togelius, J., Yannakakis, G. N., Stanley, K. O., and Browne, C. (2011b). Search-based procedural content generation: A taxonomy and survey. IEEE Trans. Comput. Intellig. and AI in Games, 3(3):172–186.

TSR, H. (1976). Dungeons & dragons: Dungeon Geomorphs.

Tylor, E. B. (1879). The history of games. Fortnightly review, 25(149):735–747.

Williams, D. (2002). Structure and competition in the U.S. home video game industry. The International Journal on Media Management, 4(1):41–54.