Uma abordagem evolutiva para geração procedural
de níveis em jogos de quebra-cabeças baseados em
física

Lucas Nascimento Ferreira

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Lucas Nascimento Ferreira

Uma abordagem evolutiva para geração procedural de níveis em jogos de quebra-cabeças baseados em física

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Claudio Fabiano Motta Toledo

USP – São Carlos
Setembro de 2015

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

F383u Ferreira, Lucas Nascimento
 Uma abordagem evolutiva para geração procedural de
 níveis em jogos de quebra-cabeças baseados em física
 / Lucas Nascimento Ferreira; orientador Claudio
 Fabiano Motta Toledo. - São Carlos - SP, 2015.
 82 p.

 Dissertação (Mestrado - Programa de Pós-Graduação
 em Ciências de Computação e Matemática Computacional)
 - Instituto de Ciências Matemáticas e de Computação,
 Universidade de São Paulo, 2015.

 1. Geração Procedural de Conteúdo. 2. Algoritmo
 Genético. 3. Funções de Aptidão ruidosa.
 4. Otimização. 5. Agente Inteligente. 6. Imersão.
 7. Jogos de Quebra-cabeças Baseados em Física . I.
 Toledo, Claudio Fabiano Motta, orient. II. Título.

Lucas Nascimento Ferreira

**An evolutionary approach for procedural generation of levels
in physics-based puzzle games**

Master dissertation submitted to the Instituto de
Ciências Matemáticas e de Computação – ICMC-
USP, in partial fulfillment of the requirements for the
degree of the Master Program in Computer Science
and Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and
Computational Mathematics

Advisor: Prof. Dr. Claudio Fabiano Motta Toledo

USP – São Carlos
September 2015

*Este trabalho é dedicado a todos os professores que,
direta ou indiretamente, me influenciaram a seguir a carreira acadêmica.*

AGRADECIMENTOS

Os agradecimentos principais são direcionados à minha família, que sempre me forneceu apoio emocional e financeiro para que eu realizasse meus sonhos. Agradeço muito também ao Prof. Dr. Claudio Toledo pela confiança e dedicação. Em especial por ter me dado liberdade para a escolha e desenvolvimento de um tema de pesquisa até então pouco explorado no Brasil. Por fim, agradeço aos grandes amigos que fiz em São Carlos, os quais me ensinaram muito e me proporcionaram momentos que lembrarei para o resto da vida.

“A Wop-Boppa Loo-Bop a Wop-Bam Boom”
(Little Richard, Tutti Frutti)

RESUMO

FERREIRA, L. N.. **Uma abordagem evolutiva para geração procedural de níveis em jogos de quebra-cabeças baseados em física**. 2015. 82 f. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

Na última década diversos algoritmos baseados em busca foram desenvolvidos para a geração de níveis em diferentes tipos de jogos. O espaço de busca para geração de níveis geralmente possui restrições, uma vez que a mecânica de um jogo define regras de factibilidade para os níveis. Em alguns métodos, a avaliação de factibilidade requer uma simulação com um agente inteligente que controla o jogo. Esse processo de avaliação geralmente possui ruído, causado por componentes aleatórios no simulador ou na estratégia do agente. Diversos trabalhos têm utilizado simulação como forma de avaliação de conteúdo, no entanto, nenhum deles discutiu profundamente a presença de ruído neste tipo de abordagem. Assim, esse trabalho apresenta um algoritmo genético capaz de gerar níveis factíveis que são avaliados por um agente inteligente em uma simulação ruidosa. O algoritmo foi aplicado a jogos de quebra-cabeças baseados em física com a mecânica do *Angry Birds*. Uma representação dos níveis em forma de indivíduos é introduzida, a qual permite que o algoritmo genético os evolua com características diferenciadas. O ruído na função de aptidão é tratado por uma nova abordagem, baseada em uma sistema de cache, que auxilia o algoritmo genético a encontrar boas soluções candidatas. Três conjuntos de experimentos foram realizados para avaliar o algoritmo. O primeiro compara o método de cache proposto com outros métodos de redução de ruído da literatura. O segundo mede a expressividade do algoritmo genético considerando as características estruturais dos níveis gerados. O último avalia os níveis gerados considerando aspectos de design (como dificuldade, imersão e diversão), os quais são medidos por meio de questionários respondidos por jogadores humanos via Internet. Os resultados mostraram que o algoritmo genético foi capaz de gerar níveis distintos que são tão imersíveis quanto níveis produzidos manualmente. Além disso, a abordagem de cache lidou apropriadamente com o ruído nos cálculos de aptidão, permitindo uma correta evolução elitista.

Palavras-chave: Geração Procedural de Conteúdo, Algoritmo Genético, Funções de Aptidão ruidosa, Otimização, Agente Inteligente, Imersão, Jogos de Quebra-cabeças Baseados em Física .

ABSTRACT

FERREIRA, L. N.. **Uma abordagem evolutiva para geração procedural de níveis em jogos de quebra-cabeças baseados em física.** 2015. 82 f. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

In the last decade several search-based algorithms have been developed for generating levels in different types of games. The search space for level generation is typically constrained once the game mechanics define feasibility rules for the levels. In some methods, evaluating level feasibility requires a simulation with an intelligent agent which plays the game. This evaluation process usually has noise, caused by random components in the simulator or in the agent strategy. Several works have used a simulation for content evaluation, however, none of them have deeply discussed the presence of noise in this kind of approach. Thus, this paper presents a genetic algorithm capable of generating feasible levels that are evaluated by an intelligent agent in a noisy simulation. The algorithm was applied to physics-based puzzle games with the Angry Birds mechanics. A level representation in the form of individuals is introduced, which allows the genetic algorithm to evolve them with distinct characteristics. The fitness function noise is handled by a new approach, based on a cache system, which helps the genetic algorithm finding good candidate solutions. Three sets of experiments were conducted to evaluate the algorithm. The first one compares the proposed cache approach with other noise reduction methods of the literature. The second one measures the expressivity of the genetic algorithm considering the structural characteristics of the levels. The last one evaluates design aspects (such as difficulty, immersion and fun) of the generated levels using questionnaires answered by human players via Internet. Results showed the genetic algorithm was capable of generating distinct levels that are as immersive as levels manually designed. Moreover, the cache approach handled properly the noise in the fitness calculations, allowing a correct elitist evolution.

Key-words: Procedural Content Generation, Genetic Algorithm, Noisy Fitness Function, Optimization, Intelligent Agent, Immersion, Physics-based Puzzle Games .

LISTA DE ILUSTRAÇÕES

Figura 1 – Um nível do jogo Rogue em um computador pessoal da IBM de 1984. O caractere “@” representa a posição atual do personagem, “!” é uma poção mágica e “k” é um inimigo do tipo Kobold. Os pontos no meio das salas representam os locais por onde o personagem pode andar, o “+” representa uma porta e os caracteres “#” e “-” representam as conexões e as paredes das salas, respectivamente.	32
Figura 2 – Uma instância de mapa do jogo Minecraft.	33
Figura 3 – Dois exemplos de armas geradas por procedimentos no jogo <i>Borderlands</i>	33
Figura 4 – Representação direta de um nível do Super Mario Bros. (DAHLSTROM; TOGELIUS, 2012).	34
Figura 5 – Representação indireta para a construção de uma pista de corrida (TOGELIUS; NARDI; LUCAS, 2007).	35
Figura 6 – Um exemplo de mapa, disposição de inimigos e conjunto de regras utilizados no Angelina (COOK; COLTON, 2011).	37
Figura 7 – Interface gráfica do <i>framework</i> para avaliação das pistas (CARDAMONE; LOIACONO; LANZI, 2011).	38
Figura 8 – Exemplo de um nível do <i>Angry Birds</i> com os seus principais elementos: estilingue, pássaros, pilha de blocos e porcos	45
Figura 9 – Exemplos de blocos de diferentes classes, os três primeiros são compostos e os outros são elementares.	47
Figura 10 – Um grafo de relações seguras entre as diferentes classes de blocos.	47
Figura 11 – Um exemplo de genótipo e o seu fenótipo correspondente.	48
Figura 12 – Exemplo de blocos duplicados.	49
Figura 13 – Um exemplo de operação de recombinação. Os dois indivíduos na parte de cima da figura são os pais e os outros, na parte de baixo, são os filhos gerados.	54
Figura 14 – Exemplos de operação de mutação aplicados sobre os dois indivíduos gerados na Figura 13.	54
Figura 15 – Média da aptidão do melhor indivíduo (eixo y) por geração (eixo x).	56
Figura 16 – Média da aptidão do melhor indivíduo (eixo y) por geração (eixo x).	58

Figura 17 – Exemplos de níveis com uma grande diferença de linearidade. O nível (a) possui alta linearidade porque possui pilhas com alturas similares. Por outro lado, as pilhas do nível (b) possuem diferentes alturas, por isso a sua linearidade é baixa.	60
Figura 18 – Um exemplo de dois níveis com grandes diferenças de densidade. O nível (a) possui alta densidade porque as suas pilhas ocupam quase toda a sua área útil. Por outro lado, o nível (b) possui densidade baixa pois suas pilhas ocupam apenas uma pequena porção da área útil. . .	61
Figura 19 – Impacto na frequência de pássaros e porcos quando os parâmetros b_n e l_n são variados.	61
Figura 20 – Impacto do parâmetro b_n nas métricas de linearidade e densidade. . . .	62
Figura 21 – Impacto do parâmetro l_n nas métricas de linearidade e densidade. . . .	63
Figura 22 – Distribuição de idade, gênero, experiência com jogos em geral, experiência em Angry Birds e nível de educação dos participantes.	65
Figura 23 – O impacto da experiência dos jogadores na acurácia para descobrir qual dos dois grupos foi gerado por computador.	66

LISTA DE ALGORITMOS

Algoritmo 1 – Um Algoritmo Genético clássico.	40
Algoritmo 2 – Gerando níveis aleatórios.	50

LISTA DE TABELAS

Tabela 1 – Média μ e desvio padrão σ das pontuações obtidas no experimento de imersão.	65
Tabela 2 – Média μ e desvio padrão σ das pontuações obtidas para similaridade e percepção no experimento de similaridade.	67

LISTA DE ABREVIATURAS E SIGLAS

AG		Algoritmo Genético
CE		Computação Evolutiva
IA		Inteligência Artificial
ME		Médias Explícitas
ME-2		Médias Explícitas com amostra de tamanho 2
ME-4		Médias Explícitas com amostra de tamanho 4
ME-8		Médias Explícitas com amostra de tamanho 8
MI		Médias Implícitas
MI-200	...	Médias Implícitas com população de tamanho 200
MI-400	...	Médias Implícitas com população de tamanho 400
MI-800	...	Médias Implícitas com população de tamanho 800
NPCs		<i>Non-Player Characters</i>
PCG		<i>Procedural Content Generation</i>
SBPCG	..	<i>Search-based Procedural Content Generation</i>

LISTA DE SÍMBOLOS

- T — Classe dos blocos que possuem formatos de pódio (similares à letra “T”)
- C — Classe dos blocos que possuem formatos circulares
- R — Classe dos blocos que possuem formatos retangulares
- X — Classe dos blocos que possuem formatos de caixa
- H — Constante que define a altura máxima dos níveis
- W — Constante que define a largura da área útil dos níveis
- B — Constante que define a quantidade máxima de pássaros permitida
- L — Constante que define a quantidade máxima de blocos permitida
- b_n — Parâmetro do algoritmo genético que define a porcentagem de pássaros que o jogador precisa usar para completar o nível
- l_n — Parâmetro do algoritmo genético que define a porcentagem de blocos que o nível deve ter
- b_u — Quantidade de pássaros lançados durante a simulação
- l_b — Quantidade de blocos no início da simulação
- p_f — Quantidade de porcos ao final da simulação
- s — Estabilidade dos blocos, medida por meio da soma de suas velocidades

SUMÁRIO

1	INTRODUÇÃO	27
2	REVISÃO BIBLIOGRÁFICA	31
2.1	Geração Procedural de Conteúdo	31
2.1.1	<i>Algoritmos Baseados em Busca</i>	34
2.2	Computação Evolutiva	39
2.2.1	<i>Funções de Aptidão com Ruído</i>	41
3	GERAÇÃO DE NÍVEIS COM ALGORITMOS GENÉTICOS	45
3.1	Representação dos Níveis	46
3.2	Inicialização Aleatória	49
3.3	Função de Aptidão	51
3.4	Operadores Genéticos	53
4	EXPERIMENTOS E RESULTADOS	55
4.1	Avaliação do Impacto do Ruído	55
4.2	Análise de Expressividade	59
4.3	Análise de Imersão	63
5	CONCLUSÃO	69
	Referências	73
APÊNDICE A	ADAPTAÇÕES DOS NÍVEIS COLETADOS DO JOGO ORIGINAL	79
APÊNDICE B	QUESTIONÁRIOS UTILIZADOS NO EXPERIMENTO DE IMERSÃO	81

INTRODUÇÃO

O grande crescimento do número de jogadores e o aumento na complexidade dos jogos digitais ([ESA, 2015](#)) trouxeram um problema de escalabilidade na geração de conteúdo para a indústria de jogos ([IOSUP, 2011](#)). Dessa forma, a produção de produtos comerciais de alta qualidade, atualmente, envolve o trabalho de uma equipe de centenas de pessoas incluindo artistas, designers, programadores e engenheiros de áudio, os quais trabalham em sua maioria na produção de conteúdo. Como consequência, esta tarefa se tornou um fator de grande impacto no custo e tempo de produção ([HENDRIKX et al., 2013](#)) desses produtos. Por exemplo, o *World of Warcraft* é um jogo que possui um mundo virtual complexo onde existem dois continentes principais, os quais são aproximadamente do tamanho de Manhattan. O ambiente dos continentes é formado por paisagens detalhadas, um vasto ecossistema, grandes cidades e uma complexa malha de vias. Para compor esse ambiente foi necessária a criação de uma grande variedade de conteúdos, tais como texturas, terrenos, prédios, ruas, vegetações e efeitos sonoros. A maioria deste grande volume de conteúdo foi produzida de forma manual, ao longo de cinco anos de desenvolvimento.

Uma alternativa para acelerar o processo de produção dos jogos é a Geração Procedural de Conteúdo (*Procedural Content Generation - PCG*), que consiste em utilizar algoritmos para gerar conteúdo de maneira automática ou semi-automática ([TOGELIUS et al., 2011](#)). Esses algoritmos têm a capacidade de criar grandes volumes de conteúdo que podem ser utilizados diretamente ou como ponto de partida para os designers humanos. Além disso, os algoritmos podem proporcionar conteúdos personalizados e sob demanda para o jogador, por meio da coleta de informações sobre o seu comportamento em tempo real. Esse tipo de abordagem pode aumentar o entretenimento do jogador, uma vez que ele terá experiências diferentes a cada nova partida ([SMITH et al., 2011](#)).

De acordo com ([SHAKER; TOGELIUS; NELSON, 2013](#)), conteúdo é tudo que está inserido em um jogo, exceto pelo seu motor (*game engine*) e pela Inteligência Artificial

(IA) dos Personagens Não-Jogáveis (*Non-Player Characters* - *NPCs*) . Exemplos que fazem parte dessa definição são níveis, mapas, regras, texturas, narrativas, itens, missões, músicas, armas, veículos, personagens, etc. Segundo os autores mencionados, os métodos de IA aplicados ao desenvolvimento de comportamentos para NPCs não fazem parte dessa definição pois eles estão inseridos em uma outra linha de pesquisa. Outro conceito importante nesse trabalho é o do “nível”, que deve ser interpretado como um estágio (seção, parte ou fase) do jogo e não deve ser confundido com o grau de dificuldade do mesmo.

A PCG em jogos pode ser abordada como um problema de otimização e busca (OTHMAN *et al.*, 2012). Neste caso, define-se um espaço de soluções que represente os conteúdos possíveis para um determinado contexto do jogo, no qual os algoritmos propostos devem realizar a busca pelo melhor conteúdo (RAFFE; ZAMBETTA; LI, 2011). Uma das principais vantagens dessa abordagem é que ela permite aos designers formularem as propriedades dos conteúdos a serem gerados de forma mais explícita do que outros métodos (TOGELIUS *et al.*, 2010). Diversas técnicas deste tipo utilizam um sistema evolutivo como mecanismo de busca (HASTINGS; GUHA; STANLEY, 2009; CARDONA; TOGELIUS; NELSON, 2013; LIAPIS; YANNAKAKIS; TOGELIUS, 2011; MAHLMANN; TOGELIUS; YANNAKAKIS, 2012; VALTCHANOV; BROWN, 2012), no qual uma população de conteúdos é evoluída durante um determinado número de gerações. Para isso, operadores de recombinação e mutação são definidos e aplicados aos conteúdos selecionados de acordo com sua qualidade.

Definir uma forma de avaliação de qualidade é uma das principais tarefas para a criação de métodos de PCG baseados em busca (TOGELIUS *et al.*, 2011). A avaliação de qualidade deve considerar a factibilidade dos conteúdos gerados, a qual é definida pelas mecânicas do jogo (LIAPIS; YANNAKAKIS; TOGELIUS, 2013). Por exemplo, um nível gerado para um jogo *rogue-like* precisa ter uma sala inicial, outra sala final e um caminho factível entre essas duas áreas. A avaliação de factibilidade é geralmente feita com base na estrutura interna do conteúdo gerado (SHAKER; TOGELIUS; NELSON, 2013). No entanto, em alguns casos é mais adequado (ou necessário) avaliar os conteúdos por meio de uma simulação do jogo, na qual o jogador é controlado por um agente inteligente (SHAKER; SHAKER; TOGELIUS, 2013b). Por exemplo, em jogos de quebra-cabeças baseados em física, a mecânica é extremamente dependente das características físicas do mundo virtual que foi definido. Nesse caso, é apropriado avaliar a factibilidade dos níveis por meio de uma simulação do jogo com um agente que considera rigorosamente essas características. Assim, um nível é factível se e somente se o agente é capaz de completá-lo. Essa abordagem também pode ser utilizada para coletar informações durante o jogo, estimando a dificuldade e a qualidade dos níveis. Apesar das vantagens, esta solução pode consumir muito tempo computacional e sua acurácia depende muito da estratégia utilizada pelo agente para controlar o jogador.

No contexto de computação evolutiva, o agente inteligente é responsável por definir os valores das variáveis de uma função de aptidão, a qual precisa retornar o mesmo valor sempre que o mesmo nível é avaliado (SHAKER; SHAKER; TOGELIUS, 2013a). No entanto, tanto o ambiente de simulação quanto a estratégia do agente geralmente possuem componentes estocásticos, o que adiciona ruído no cálculo de aptidão. O ruído, por sua vez, impede que a função de aptidão tenha o comportamento determinístico esperado. Esse problema é conhecido como otimização com funções de aptidão ruidosas (SANO; KITA, 2002). Ele é tipicamente encontrado em domínios onde o sistema utilizado para avaliação das soluções é modificado ao longo do tempo ou em otimizações que utilizam simulações com componentes aleatórios (SANO; KITA, 2002). Como exemplo, podem ser citados jogos de probabilidade, jogos de esporte com física não determinística e simulações industriais. A presença de ruído na função de aptidão prejudica o processo evolutivo porque uma boa solução pode ser interpretada como ruim. Esse problema pode afetar negativamente o desempenho dos algoritmos, levando à redução das taxas de convergência ou até mesmo à comportamentos de divergência (PIETRO; WHILE; BARONE, 2004).

O objetivo geral deste trabalho consiste na geração de níveis para jogos de quebra-cabeças baseados em física. A solução proposta é um Algoritmo Genético (AG) definido no contexto do *Angry Birds*, um dos jogos mais famosos do gênero em estudo. Esse algoritmo evolui uma população de níveis que são avaliados por uma simulação com um agente inteligente. Como destacado anteriormente, as simulações neste caso geralmente são estocásticas e adicionam ruído no processo de avaliação. Dessa forma, esse trabalho tem como objetivo específico propor um método capaz de lidar com funções de aptidão ruidosas baseadas em simulação. O método desenvolvido utiliza uma memória cache para evitar a propagação de ruído durante a evolução. Essa solução é uma inovação entre os métodos de PCG, uma vez que o problema de funções de aptidão ruidosas foi pouco explorado neste campo de pesquisa (MAHLMANN; TOGELIUS; YANNAKAKIS, 2011; TOGELIUS; NARDI; LUCAS, 2006; SHAKER; TOGELIUS; NELSON, 2013).

Três conjuntos de experimentos foram definidos para avaliar o algoritmo, os quais foram realizados utilizando uma versão clone do *Angry Birds* que foi desenvolvida ao longo deste trabalho. O primeiro conjunto de experimentos tem o intuito de avaliar o método de cache em comparação com outros métodos da literatura. O segundo conjunto avalia a expressividade do algoritmo enquanto um gerador de níveis. O último conjunto é composto por uma série de questionários que foram aplicados em jogadores humanos para avaliar aspectos de design (tais como, dificuldade, imersão e diversão) dos níveis gerados. Os resultados mostraram que o AG foi capaz de gerar níveis tão imersíveis quanto níveis produzidos manualmente, porém, com características distintas. Além disso, o método de cache permitiu uma correta evolução elitista e economizou tempo durante a execução do AG, uma vez que evitou um número considerável de simulações.

As principais contribuições desse trabalho são a representação dos níveis em forma de indivíduos, um método de inicialização de população com alta probabilidade de produzir níveis estáveis e uma função customizável de aptidão baseada em simulação. No âmbito da função de aptidão, contribuições específicas são um método para lidar com ruído e um agente inteligente capaz de avaliar a factibilidade dos níveis. Além disso, esse trabalho contribui para a área de PCG apresentando uma metodologia de avaliação de níveis por meio de experimentos com jogadores humanos. A versão clone do jogo *Angry Birds* é a principal contribuição técnica desse trabalho. O código-fonte do jogo dessa versão é aberto e está disponível para acesso online ¹.

O restante deste trabalho está organizado da seguinte forma: o Capítulo 2 contém uma revisão bibliográfica das áreas de geração procedural de conteúdo, computação evolutiva e funções de aptidão ruidosas. O Capítulo 3 apresenta o algoritmo genético proposto para geração de níveis do jogo *Angry Birds*, destacando a representação dos níveis, a função de aptidão e os operadores genéticos utilizados. O Capítulo 4 mostra os experimentos realizados para avaliar o método de cache definido, a expressividade do algoritmo proposto e aspectos de design dos níveis gerados. Por fim, o Capítulo 5 conclui esse trabalho com uma discussão dos resultados e possíveis trabalhos futuros desse projeto.

¹ <<https://github.com/lucasnfe/AngryBirdsCover>>

REVISÃO BIBLIOGRÁFICA

Esse trabalho utiliza computação evolutiva para a geração procedural de níveis em jogos de quebra-cabeças baseados em física. A avaliação dos níveis é feita por meio de simulações ruidosas do jogo com um agente inteligente. Dessa forma, esse capítulo apresenta uma revisão bibliográfica na área de geração procedural de conteúdo, computação evolutiva e funções de aptidão ruidosas.

2.1 Geração Procedural de Conteúdo

A geração procedural de conteúdo em jogos teve início na década 1980 e sua motivação inicial foi a limitação de memória dos computadores da época (KHALED; NELSON; BARR, 2013). A pequena capacidade de armazenamento dos computadores pessoais restringia a quantidade de conteúdo que um jogo podia conter. Sendo assim, os designers de jogos foram forçados a encontrar outras formas para gerar e armazenar conteúdo. O jogo *Elite* (BELL; BRABEN, 1984), por exemplo, foi um dos primeiros a lidar com esse problema. O universo do jogo é composto por 8 galáxias, sendo que cada uma possui 256 planetas com características únicas. Para suportar um universo deste tamanho, os planetas tiveram que ser gerados dinamicamente, uma vez que todo esse volume de conteúdo não podia ser armazenado em memória. Outro exemplo clássico que lidou com o mesmo problema é o *Rogue* (TOY, 1983), um jogo do gênero *dungeon crawler* em que os níveis são gerados sob demanda por meio de procedimentos computacionais. Para realizar essa tarefa, os procedimentos definem as seguintes características dos níveis: o tamanho de cada sala, as conexões entre elas, as posições dos itens, a disposição dos inimigos e os locais de entrada e saída do nível. A Figura 1 mostra como esses elementos são utilizados para formar um nível.

Recentemente, mesmo com o aumento da capacidade de armazenamento dos computadores, os jogos comerciais continuam utilizando técnicas de PCG. Um dos principais

Figura 1 – Um nível do jogo Rogue em um computador pessoal da IBM de 1984. O caractere “@” representa a posição atual do personagem, “!” é uma poção mágica e “k” é um inimigo do tipo Kobold. Os pontos no meio das salas representam os locais por onde o personagem pode andar, o “+” representa uma porta e os caracteres “#” e “-” representam as conexões e as paredes das salas, respectivamente.

motivos para isso vem do fato de que a geração dinâmica de conteúdo faz com que o jogador tenha experiências diferentes a cada nova partida (SMITH *et al.*, 2011). Essa característica é muito importante comercialmente, pois ela aumenta a chance de um jogador jogar o mesmo jogo diversas vezes (*replayability*). Um exemplo recente de uso de PCG é o *Spelunky* (YU; HULL; SUHRKE, 2013), um jogo de plataforma 2D em que variações de níveis são geradas automaticamente. O *Civilization IV* (FIRAFIX, 2005) é outro exemplo deste caso, um jogo de estratégia em turnos, cujos mapas são gerados aleatoriamente antes do início de cada partida. O *Minecraft* (MOJANG, 2009) também é outro exemplo bastante atual. Nesse jogo, todo o ambiente em que o jogador está inserido é gerado sob demanda. A Figura 2 ilustra uma instância de ambiente do *Minecraft*. Quando o jogo inicia, uma parte do ambiente é criada, conforme o jogador se move para fora dessa área, novas áreas são geradas.

Outra motivação recente para se utilizar PCG é o grande crescimento do tamanho e da complexidade dos jogos. Atualmente os jogos digitais são compostos por diversos tipos de conteúdos, tais como terrenos, mapas, níveis, narrativas, diálogos, missões, personagens, conjuntos de regras, armas, veículos, etc. Gerar tais conteúdos pode demandar muito tempo para os artistas e designers. Para auxiliar as tarefas desses profissionais, é possível construir ferramentas baseadas em métodos de PCG que geram uma grande variedade de conteúdo de forma rápida. Assim, os profissionais podem utilizar os conteúdos gerados dessa forma como base para uma versão final, o que ajuda a acelerar o processo de desenvolvimento de conteúdo (SMITH; WHITEHEAD; MATEAS, 2010). Por exemplo, esse tipo de abordagem foi utilizada no jogo *Just Cause* (AVALANCHE, 2006) para criar uma grande variedade

Figura 2 – Uma instância de mapa do jogo Minecraft.

de ilhas para o ambiente do jogo. Dentre elas, algumas foram escolhidas e manipuladas por designers até que chegassem em uma versão final. Outro jogo recente que usou essa abordagem foi o *Borderlands* (GEARBOX, 2009). Durante o seu desenvolvimento, uma grande diversidade de armas foi gerada por procedimentos computacionais, sendo que a Figura 3 mostra dois exemplos delas.

Figura 3 – Dois exemplos de armas geradas por procedimentos no jogo *Borderlands*.

A geração procedural de conteúdo também tem chamado a atenção dos pesquisadores da área de Inteligência Artificial em Jogos, sendo considerada atualmente como um dos tópicos mais promissores dentro desse campo de pesquisa (YANNAKAKIS; TOGELIUS, 2014). Os métodos baseados em busca formam uma das principais sub-áreas deste campo. Neste caso, a PCG é tratada como um problema de otimização e os métodos têm como objetivo encontrar o ótimo global de uma função de aptidão que representa o valor de entretenimento de um determinado conteúdo (TOGELIUS *et al.*, 2011). Por exemplo, (TOGELIUS *et al.*, 2013) propôs um algoritmo evolutivo multi-objetivo para a geração de mapas para o clássico jogo de estratégia *Star Craft*. (BROWNE; MAIRE,

2010) demonstrou que é possível gerar automaticamente jogos de tabuleiros completos com qualidade similar a de jogos presentes no mercado. (MOURATO; SANTOS; BIRRA, 2011), (SHAKER; YANNAKAKIS; TOGELIUS, 2010) e (FERREIRA; PEREIRA; TOLEDO, 2014) propuseram diferentes algoritmos genéticos para geração de níveis em jogos de plataforma.

2.1.1 Algoritmos Baseados em Busca

As principais características de um método de PCG baseado em busca (*Search-based Procedural Content Generation* - SBPCG) é a presença de uma função de aptidão, uma forma de representar o conteúdo e um mecanismo de busca capaz de encontrar conteúdos desejados. Nesta abordagem, o algoritmo gera conteúdos candidatos e os avalia segundo algum critério de qualidade para o contexto em que ele está sendo gerado. Se eles não forem bem avaliados, eles são descartados e novos candidatos são gerados, até que sejam bons o suficiente. Neste caso, a função de avaliação não é utilizada apenas para aceitar ou rejeitar os conteúdos candidatos. Ela atribui uma nota para cada um deles, de tal forma que a nota é a sua aptidão. Os novos candidatos são baseados nos que foram gerados anteriormente, visando obter soluções com maior aptidão (TOGELIUS *et al.*, 2011).

Em SBPCG, uma solução do espaço de busca é representada pelas informações necessárias para a construção do conteúdo (SHAKER; TOGELIUS; NELSON, 2013). Essa representação pode ser feita por meio de uma codificação direta ou indireta. Na codificação direta, o tamanho da representação é linearmente proporcional ao tamanho do conteúdo e cada parte da representação mapeia uma parte específica do conteúdo. Isto implica uma complexidade computacional relativamente baixa na transformação de um em outro. Por exemplo, em (DAHLSSKOG; TOGELIUS, 2012), um algoritmo evolutivo é utilizado para a geração de níveis do jogo Super Mario Bros. Cada nível é representado por um vetor de segmentos verticais pré-definidos contendo objetos do jogo. Para decodificar essa cadeia em um nível, basta conectar todos os segmentos do vetor. A Figura 4 mostra um trecho de um nível do jogo que contém apenas três inimigos. Esse trecho pode ser representado por um vetor de tamanho três, no qual todos os elementos possuem o mesmo segmento vertical. Para decodificar esse indivíduo no trecho ilustrado, basta interpretar o vetor da esquerda para a direita.

Na codificação indireta, a representação mapeia de forma não linear o conteúdo, assim, eles não precisam ter tamanhos proporcionais. Geralmente esse tipo de codificação envolve uma grande complexidade computacional (BENTLEY; KUMAR, 1999). Por exemplo, uma codificação indireta foi utilizada em (TOGELIUS; NARDI; LUCAS, 2007) para representar pistas de jogos de corrida em um algoritmo evolutivo. A representação da pista é formada por um conjunto de pontos de controle que ela tem que cobrir. A pista propriamente dita é gerada utilizando curvas de Bezier com os pontos de controle

Figura 4 – Representação direta de um nível do Super Mario Bros. (DAHLKOG; TOGELIUS, 2012).

da representação. A Figura 5 ilustra uma pista codificada por um conjunto de pontos de controle (pontos vermelhos), os quais são representados em um sistema de coordenadas polares, sendo que o ponto azul é a origem do sistema.

É muito importante que a codificação escolhida seja capaz de representar bem todas as soluções interessantes. Esta qualidade é difícil de ser garantida com as codificações indiretas, pois podem haver regiões do espaço de conteúdos que não são mapeados pela representação definida (SHAKER; TOGELIUS; NELSON, 2013). Na codificação direta é geralmente mais simples de garantir quais áreas do espaço de soluções podem ser encontradas. As representações também devem seguir o princípio da localidade, ou seja, pequenas mudanças na representação devem resultar em uma pequena alteração do conteúdo e uma pequena diferença de aptidão (TOGELIUS *et al.*, 2011).

De uma maneira geral, o espaço de busca de conteúdos possui diversas restrições, as quais são definidas pela própria mecânica do jogo. Por exemplo, em um jogo de plataforma, as plataformas que formam o caminho entre o início e o fim do nível devem ser todas alcançáveis, possibilitando que o jogador chegue ao seu destino final. Os métodos de busca precisam lidar com essas restrições, uma vez que soluções infactíveis serão decodificadas em conteúdos que não podem ser jogados. Uma forma de evitar soluções infactíveis consiste em desenvolver uma representação e um mecanismo de busca que satisfaçam completamente as restrições do problema. Em alguns casos é muito difícil evitar a geração de conteúdos infactíveis, assim outras soluções devem ser empregadas. Uma maneira trivial de lidar com este problema consiste em simplesmente descartar os conteúdos infactíveis sempre que eles são produzidos (CARDAMONE; LOIACONO; LANZI, 2011). Essa solução é simples mas pode reduzir a capacidade do algoritmo de explorar o espaço de busca.

Figura 5 – Representação indireta para a construção de uma pista de corrida (TOGELIUS; NARDI; LUCAS, 2007).

Uma maneira comum de resolver esse problema consiste em penalizar, via função de aptidão, os conteúdos que não satisfazem as restrições (LOIACONO; CARDAMONE; LANZI, 2011). Uma abordagem mais elaborada consiste em desenvolver uma função de reparo que corrige um indivíduo antes da sua avaliação, transformando-o em um conteúdo factível (CACHIA; LIAPIS; YANNAKAKIS, 2015). Outra abordagem, bastante difundida na área de PCG, consiste em utilizar heurísticas completamente separadas para avaliar indivíduos factíveis e infactíveis. Esse tipo de abordagem é utilizada no algoritmo FI-2Pop GA, o qual mantém duas populações de indivíduos, cada uma com uma função de aptidão e um esquema de seleção diferentes e independentes (LIAPIS; YANNAKAKIS; TOGELIUS, 2013). A primeira população possui apenas indivíduos factíveis e sua avaliação está relacionada com o tipo de conteúdo que se deseja. A segunda população possui apenas soluções infactíveis, sendo que sua função de aptidão avalia o nível de factibilidade dos conteúdos (KIMBROUGH *et al.*, 2008). Assim, qualquer solução da segunda população que passar a satisfazer todas as restrições, é movida para a primeira população. Por outro lado, qualquer solução da primeira população que violar uma restrição (por um resultado de um operador genético) é movida para a primeira população (KIMBROUGH *et al.*, 2008). A migração de indivíduos dessa maneira permite operações de cruzamento que aumentam a diversidade em ambas as populações.

Uma vez que o conteúdo é criado, ele precisa ser avaliado e receber uma nota que representa com precisão a sua utilidade no jogo. Definir uma função de avaliação demanda decidir que aspecto do jogo precisa ser otimizado e como ele pode ser formalizado. Por exemplo, um método de SBPCG pode ser definido para otimizar a diversão do jogo. Desta forma, a função de avaliação deve ser capaz de identificar o quanto um determinado conteúdo gerado por esse método contribui para a sensação de diversão do jogador durante o jogo. De acordo com (YANNAKAKIS; TOGELIUS, 2011), as funções de avaliação podem

ser classificadas em três tipos: direta, baseada em simulação e interativa.

Na avaliação direta, apenas as características do conteúdo são utilizadas para o cálculo da aptidão, sendo que as funções para esse cálculo podem ser baseadas em teorias (*theory-driven*) ou em dados (*data-driven*) (TOGELIUS *et al.*, 2011). No primeiro caso, o desenvolvedor da função de avaliação é guiado por teorias qualitativas de *game design* sobre geração e avaliação de experiências em jogadores. No segundo caso, são utilizados dados coletados sobre o efeito de diversos conteúdos na experiência do jogador. Diferentes exemplos de funções de avaliação diretas são apresentadas em (COOK; COLTON, 2011). Este trabalho utiliza um algoritmo genético co-evolutivo para gerar jogos inteiros no estilo *arcade*. O algoritmo explora três aspectos do jogo: mapa, posição dos inimigos e um conjunto de regras. A Figura 6 ilustra um exemplo desses três aspectos. Cada um deles possui uma função de avaliação direta própria. Por exemplo, a qualidade da disposição dos inimigos é calculada utilizando a esparsidade e o volume da matriz que a representa. Dado um mapa m com n inimigos, essas métricas podem ser calculadas da seguinte forma:

$$esparsidade(m) = 2 * (\sum_{i=1}^n \sum_{j=1}^n dist(i, j)) * \frac{1}{n*n-1}$$

$$vol(m) = |vermelhos(m)| + |azuis(m)| + |verdes(m)|$$

Sendo que $dist(i, j)$ é a distância euclidiana entre os inimigos i e j , $|vermelhos(m)|$ é a quantidade de inimigos da cor vermelha, $|azuis(m)|$ é a quantidade de inimigos da cor azul e $|verdes(m)|$ é a quantidade de inimigos da cor verde. É importante ressaltar que a função de qualidade da disposição dos inimigos é calculada utilizando apenas o estado da matriz que representa a disposição. Por isso, este tipo de função é dita como avaliação de conteúdo direta.

```
<scorelimit>44</scorelimit>
<timelimit>72</timelimit>
<redmovement>CLOCKWISE</redmovement>
<greenmovement>RANDLONG</greenmovement>
<bluemovement>RANDLONG</bluemovement>
<rules>
  <rule>PLAYER, RED, DEATH, DEATH, 1</rule>
  <rule>GREEN, RED, TELEPORT, DEATH, 0</rule>
  <rule>OBSTACLE, RED, NOTHING, TELEPORT, 0</rule>
</rules>
```

Figura 6 – Um exemplo de mapa, disposição de inimigos e conjunto de regras utilizados no Angelina (COOK; COLTON, 2011).

A função de avaliação também pode ser do tipo interativa e, neste caso, o conteúdo é avaliado por um jogador (ou designer) humano durante o processo de geração do

conteúdo (SMITH; WHITEHEAD, 2010). Os dados sobre a avaliação podem ser captadas de maneira explícita ou implícita. As avaliações explícitas podem ser feitas por meio de questionários e as implícitas por meio de métricas, coletadas diretamente pelo jogo, que reflitam a experiência do jogador. Por exemplo, calcular quantas interações o usuário teve com um determinado tipo de conteúdo, o momento em que o jogador saiu do jogo ou a pressão utilizada para apertar um botão (TOGELIUS *et al.*, 2011). Em (CARDAMONE; LOIACONO; LANZI, 2011) é apresentado um *framework* para geração de pistas para jogos de corrida, as quais são avaliadas de forma interativa pelos jogadores. Este *framework* é composto por duas partes, uma interface *web* responsável pela interação com o usuário para avaliação das pistas e um servidor que aplica os operadores evolutivos de seleção, recombinação e mutação. A Figura 7 ilustra a interface gráfica do *framework*, que possibilita que o jogador avalie a qualidade das pistas geradas. A representação das pistas utiliza uma codificação indireta similar à apresentada em (TOGELIUS; NARDI; LUCAS, 2007), portanto as pistas geradas sempre são factíveis. O processo evolutivo é guiado pelo usuário, ou seja, os operadores evolutivos são executados pelo servidor apenas depois que ele avalia as pistas da população corrente e pressiona o botão “*Evolve!*”.

Figura 7 – Interface gráfica do *framework* para avaliação das pistas (CARDAMONE; LOIACONO; LANZI, 2011).

Como nem sempre é simples calcular a qualidade de um conteúdo com base apenas em suas características (BIRD; FELTWELL; CIELNIAK, 2012), é possível avaliar como o conteúdo agrega valor ao jogo utilizando uma simulação. Assim, o conteúdo é inserido no jogo e um agente inteligente é utilizado para simular um jogador real. Nesse caso, características são extraídas durante a execução da simulação para avaliação do conteúdo (TOGELIUS *et al.*, 2011). Este tipo de avaliação pode ser feita de maneira dinâmica ou estática (SHAKER; TOGELIUS; NELSON, 2013). Quando o agente utilizado na

simulação não é alterado durante o jogo, a avaliação é dita estática. Por exemplo, em (SHAKER; SHAKER; TOGELIUS, 2013a) são gerados níveis para o *Cut the Rope*, um jogo de quebra-cabeças baseados em física. Os autores utilizaram um agente capaz de resolver os quebra-cabeças do jogo, o qual foi integrado a um gerador de níveis chamado *Ropossum* (SHAKER; SHAKER; TOGELIUS, 2013b) que verifica, por meio de simulações, se os níveis produzidos são jogáveis ou não. Na avaliação dinâmica o agente muda durante a execução da simulação e a aptidão do conteúdo leva em consideração as mudanças ocorridas. Por exemplo, um agente pode ser implementado utilizando algum algoritmo de aprendizado, dessa forma a aptidão do conteúdo seria relativa à quantidade de conhecimento que o agente obteve na simulação.

A avaliação por simulação pode adicionar um alto tempo computacional para um gerador de conteúdo, sendo que a sua acurácia depende muito da estratégia utilizada pelo agente para completar os níveis. Além disso, esse tipo de avaliação geralmente possui ruídos, que podem ser causados por componentes estocásticos tanto no simulador quanto na estratégia utilizada pelo agente. A presença de ruído na avaliação dificulta o desenvolvimento de algoritmos de SBPCG, pois um conteúdo pode obter valores diferentes de qualidade quando avaliado diversas vezes. Mesmo com esses problemas, muitos algoritmos baseados em busca utilizam essa abordagem de avaliação (MAHLMANN; TOGELIUS; YANNAKAKIS, 2011; TOGELIUS; NARDI; LUCAS, 2006; SHAKER; SHAKER; TOGELIUS, 2013a), porém poucos deles definiram algum mecanismo para lidar com a presença de ruídos. Como destacado no Capítulo 1, essa dissertação tem como objetivo específico propor uma solução para lidar adequadamente com ruído em métodos de SBPCG. Portanto, a seção seguinte contém uma revisão sobre computação evolutiva e otimização com funções de aptidão ruidosas, apresentando as características do problema e algumas das principais soluções.

2.2 Computação Evolutiva

A Computação Evolutiva (CE) é um ramo da Inteligência Artificial que compreende os métodos de otimização baseados na evolução natural das espécies, sendo que otimização consiste na busca pela melhor solução de um determinado problema. Os métodos de CE utilizam o processo de evolução natural como um paradigma para solução de problemas de otimização complexos. Uma grande vantagem da CE é a facilidade com que os métodos podem ser adaptados e aplicados em diferentes problemas de otimização (EIBEN; SMITH, 2003). Essa característica é obtida pela forma como os métodos são estruturados e, de acordo com (JONG, 2008), os componentes básicos de um método evolutivo são:

- Uma ou mais populações de indivíduos competindo por recursos limitados
- Atualização dinâmica de populações por meio de nascimento e morte de indivíduos

- Cada indivíduo possui uma habilidade para sobreviver e reproduzir (aptidão)
- Herança: as gerações herdam características de seus antecessores

Uma característica importante dos métodos de CE é a forma como eles representam os indivíduos que compõe as populações (JONG, 2008). Uma forma bastante tradicional e simples consiste em utilizar um vetor de tamanho N , sendo que N é o número de características do indivíduo e cada posição do vetor representa uma delas. Por exemplo, as características físicas de um ser humano podem ser representados da seguinte maneira:

< cor de cabelo, cor dos olhos, cor da pele, altura, peso >

Desta forma, dizemos que o vetor com os valores para as características do indivíduo é o seu genótipo e as características em si (que observamos) são seu fenótipo, de maneira análoga aos conceitos da Genética (ALLEN, 1979). Além de definir a representação dos indivíduos, os métodos estabelecem um mecanismo de evolução que utiliza operadores inspirados em conceitos biológicos como seleção e reprodução dos indivíduos. Esses mecanismos acabam por gerar e manter na população de indivíduos aqueles que apresentam uma melhor aptidão. Para isso, funções de avaliação são definidas visando qualificar os indivíduos da população.

Um dos mecanismos de evolução mais tradicionais é o AG (GOLDBERG, 1989). De uma maneira geral, os AGs são compostos por um número pré-definido de gerações sendo que, em cada geração, um conjunto de pais é escolhido para reprodução e seus filhos os substituem na próxima geração. Os pais são escolhidos estocasticamente com relação a sua aptidão, assim, os indivíduos mais aptos têm maior chance de contribuir com seu material genético. Os filhos são formados pela recombinação de seus pais, seguido de uma operação de mutação. Este processo está sintetizado no Algoritmo 1.

Os métodos mais comuns de seleção estocástica com base na aptidão dos indivíduos são a roleta, o torneio e o ranking (JONG, 2008). No método da roleta, para cada indivíduo que será selecionado, gera-se um número aleatório $r \in [0, S]$ por meio de uma distribuição uniforme, onde S é a soma das aptidões de todos os indivíduos. Um acumulador é incrementado com o valor da aptidão do indivíduo atual enquanto o algoritmo percorre os indivíduos. Desta forma, o indivíduo selecionado é o primeiro que possui aptidão acumulada maior que r (LIPOWSKI; LIPOWSKA, 2011). Esse algoritmo é executado até que sejam escolhidos M indivíduos, onde M é o tamanho da população.

O principal problema do algoritmo da roleta é que ele não suporta valores negativos de aptidão. A seleção por torneio é uma alternativa para solucionar este problema. Nesse caso, são escolhidos aleatoriamente n indivíduos da população e o indivíduo com maior aptidão dentre eles é selecionado para reprodução. O processo se repete até preencher uma população intermediária com o mesmo tamanho M da população do AG (MILLER et

Algoritmo 1: Um Algoritmo Genético clássico.

```
1  $ger \leftarrow 0$ ;  
2 Gerar população inicial aleatoriamente com tamanho M;  
3 Calcular a aptidão dos indivíduos da população inicial;  
4 enquanto  $ger < maxGer$  faça  
5   Selecionar os pais para reprodução;  
6   Aplicar operação de recombinação sobre os pais selecionados;  
7   Aplicar mutação sobre os filhos gerados na operação anterior;  
8   Calcular a aptidão desses filhos;  
9   Substituir todos os indivíduos da população atual pelos filhos gerados;  
10   $ger \leftarrow ger + 1$ ;
```

[al., 1995](#)). Outra forma de selecionar os indivíduos para reprodução é o ranking ([EIBEN; SMITH, 2003](#)). Nesse método, os indivíduos são primeiramente ordenados com base em sua aptidão, depois recebem um novo valor de aptidão de acordo com sua posição na lista ordenada. Em seguida, são selecionados indivíduos de acordo com suas posições no ranking. Tal procedimento deve dar mais chance de seleção para aqueles que estão no topo do ranking.

Tradicionalmente, os AGs utilizam dois tipos de operadores genéticos, a mutação e a recombinação. A recombinação se dá por meio de operações de cruzamento ([DAVIS; MITCHELL, 1991](#)) realizadas sob pares de indivíduos, chamados de pais. Geralmente, este tipo de operação produz dois indivíduos, que são chamados de filhos. Uma das formas mais básicas de realizar o cruzamento consiste em escolher um ponto de corte nos vetores que representam cada um dos dois pais. Assim, o primeiro filho é formado pela concatenação da sequência inicial (antes do ponto de corte) do primeiro pai e a sequência final (depois do ponto de corte) do segundo. Da mesma maneira, o segundo filho é formado pela concatenação da sequência final (depois do ponto de corte) do primeiro pai e a sequência inicial (antes do ponto de corte) do segundo. A mutação pode ser realizada alterando, com uma certa probabilidade, um ou mais genes dos indivíduos produzidos pela operação de recombinação ([GOLDBERG, 1989](#)). A probabilidade para alteração dos genes é geralmente baixa e é chamada de taxa de mutação.

As aptidões dos indivíduos são calculadas por meio de uma função, que mede o quão próximo um indivíduo está da solução desejada ou quão boa é esta solução ([GOLDBERG, 1989](#)). É essencial que esta função seja representativa e diferencie corretamente as soluções boas das ruins. Se houver pouca precisão na avaliação, uma boa solução pode ser descartada e o algoritmo pode passar muito tempo explorando soluções pouco promissoras. Além disso, a função de aptidão deve retornar o mesmo resultado sempre que o mesmo indivíduo

for avaliado. Porém, caso o cálculo da função dependa de variáveis do mundo real ou de simulações, é possível que o determinismo não seja garantido. Esse problema é conhecido como otimização com funções de aptidão ruidosas.

2.2.1 Funções de Aptidão com Ruído

Problemas de otimização com funções de aptidão ruidosas são aqueles nos quais diferentes valores de aptidão podem ser obtidos a partir do mesmo genótipo. Esses problemas aparecem geralmente em aplicações do mundo real que precisam de um sistema externo ou de uma simulação para avaliar as soluções candidatas (SANO; KITA, 2002). O ruído pode ser proveniente de diversas fontes, tais como erros em sensores de medição ou simulações com componentes aleatórios. Uma função de aptidão ruidosa causa comportamentos indesejados durante a execução de um algoritmo de otimização. Por exemplo, uma solução superior pode ser erroneamente interpretada como inferior e vice-versa. Neste caso, uma boa solução pode ser eliminada e uma solução ruim pode sobreviver e se reproduzir. Esse comportamento causa problemas como redução da taxa de aprendizado, dificuldade em reter o que o algoritmo já aprendeu sobre o problema, limite na capacidade de exploração do espaço de busca e falta de garantia de que a aptidão irá melhorar durante as gerações (mesmo com elitismo) (PIETRO; WHILE; BARONE, 2004). Existem diversos métodos capazes de lidar com funções de aptidão ruidosas e alguns deles estão detalhados a seguir.

Uma das formas mais comuns de reduzir o impacto de ruídos na função de aptidão consiste em avaliar as soluções candidatas por meio da média de diversos cálculos de aptidão (JIN; BRANKE, 2005). Calcular a média da aptidão aumenta a acurácia do método de seleção por meio da redução do erro padrão, durante a avaliação das soluções. Na versão canônica desse método, a média é calculada a partir de um número fixo de amostras. Uma vez que cada avaliação de aptidão pode consumir muito tempo (ou “custar caro”), é desejado reduzir o número de amostras o máximo possível, sem comprometer o desempenho. Isso pode ser feito adaptando a quantidade de amostras durante a execução do algoritmo. Por exemplo, aumentando gradativamente o tamanho da amostra com base no número de iterações (gerações) que o algoritmo já executou. Outra forma de controlar o número de amostras consiste em defini-lo com base na variância estimada da aptidão, isto é, quanto maior a variância, maior será o tamanho da amostra (JIN; BRANKE, 2005). A aplicabilidade e a acurácia do método de média depende de quão “caro” e de quanto tempo demora para calcular uma função de aptidão.

Em algoritmos com mecanismo de busca populacional, como os AGs, as boas soluções candidatas têm maior probabilidade de sobrevivência, quando comparadas às piores soluções. Esse efeito pode ser tratado como um tipo de “média implícita” em termos de funções de aptidão com ruído. Assim, um simples método para reduzir a influência de ruído em algoritmos de otimização consiste em utilizar um tamanho grande

de população (FITZPATRICK; GREFENSTETTE, 1988). Isso reduz, de uma maneira geral, o impacto de cálculos de aptidão realizados de forma incorreta, dificultando que soluções inferiores (superestimadas) dominem a população. (MILLER; GOLDBERG, 1996) mostrou teoricamente que, quando o tamanho da população tende ao infinito, o ruído não afeta o mecanismo de seleção com base na aptidão das soluções. Em (RATTRAY; SHAPIRO, 1997), resultados de experimentos com tamanhos diferentes de população indicaram que tamanhos grandes de população reduzem o efeito de ruídos sintéticos gerados por uma função gaussiana.

No contexto das Estratégias Evolutivas, (BEYER, 1998) desenvolveu um método baseado no princípio de “mutar muito, herdar pouco”. Esse método inicializa a população com uma taxa grande de mutação, visando gerar pontos que estão distantes no espaço de busca. Depois, em cada geração, ele reduz essa taxa por um fator constante k para produzir os indivíduos filhos. Nesse processo é provável que a diferença de aptidão entre os pais e filhos aumente, o que deveria superar o nível de ruído. Teoricamente, para cada nível de ruído, existe um valor k ótimo que maximiza o desempenho do método. Diferente dos métodos baseados em média, este não requer avaliações de aptidão adicionais e, conseqüentemente, demanda menos tempo computacional.

Existem uma série de outros métodos que lidam com ruídos na função de aptidão por meio de modificações no mecanismo de seleção do processo evolutivo. Por exemplo, no método de (MARKON *et al.*, 2001), uma solução candidata é selecionada apenas se a sua aptidão for maior que a de seus pais em, no mínimo, um determinado valor limite (*threshold*). Essa técnica aumenta a chance de preservação da melhor solução durante as iterações evolutivas. (RUDOLPH, 2001) propôs uma técnica de seleção considerando que nem sempre é possível determinar se uma solução é realmente melhor que outra. Assim, também não é possível definir uma ordem total entre as soluções candidatas. Considerando esse problema e assumindo que o ruído não pode exceder um determinado limite, a abordagem proposta define uma ordem parcial no conjunto de valores de aptidão ruidosos e seleciona indivíduos mínimos de conjuntos parcialmente ordenados. Outro exemplo de modificação de seleção consiste em reduzir a aleatoriedade no processo de seleção (BRANKE; SCHMIDT, 2003). O fundamento deste método é balancear a incerteza, causada pelo ruído, com um método de seleção mais determinístico.

Outra abordagem consiste em estimar a aptidão de um indivíduo considerando outras soluções próximas a ele. Essa técnica requer a suposição de que a aptidão de um indivíduo é similar às aptidões das soluções próximas a ele. Por exemplo, o método proposto por (TANOOKA *et al.*, 1999) introduziu um atributo auxiliar chamado “idade” para os indivíduos, o qual é utilizado para estimar a aptidão considerando valores antigos calculados durante o seu período de sobrevivência na população. (BRANKE, 1998) propôs um método de médias, que considera a vizinhança de um indivíduo na geração atual e

em gerações passadas para estimar sua aptidão. Nesse método, o número de amostras usado na estimativa é fixo e a aptidão é estimada por uma média ponderada. (SANO; KITA, 2002) também utilizou informações de gerações passadas para estimar a aptidão de um indivíduo. Nesse caso, amostras de aptidões são armazenadas em uma memória como um histórico de busca e um modelo estocástico é utilizado para estimar a aptidão considerando esse histórico.

GERAÇÃO DE NÍVEIS COM ALGORITMOS GENÉTICOS

Algoritmos que automatizam design de níveis devem considerar a mecânica do jogo para gerar níveis factíveis com uma determinada característica de jogabilidade (*gameplay*). Nesse trabalho é proposto um algoritmo genético capaz de gerar níveis factíveis para jogos de quebra-cabeças baseados em física com a mecânica do *Angry Birds*. Essa mecânica consiste em atirar pássaros, usando um estilingue, contra uma pilha de blocos que proteje porcos (inimigos). A Figura 8 mostra um nível do jogo com todos os seus elementos básicos. O objetivo do jogo é matar todos os porcos com uma quantidade limitada de pássaros. Dessa forma, um nível factível desse jogo precisa ter dois componentes principais:

Figura 8 – Exemplo de um nível do *Angry Birds* com os seus principais elementos: estilingue, pássaros, pilha de blocos e porcos

- Uma pilha estável de blocos que projete porcos.
- Uma quantidade de pássaros que permita que o jogador mate todos os porcos.

O AG proposto constrói esses dois componentes por meio da evolução de uma população de níveis em direção ao valor ótimo de uma função de aptidão. Essa função avalia a estabilidade da pilha de blocos e a factibilidade de matar todos os porcos com a quantidade de pássaros disponível. Ambas as avaliações são realizadas por meio de uma simulação do jogo. A estabilidade é medida a partir da velocidade total dos blocos durante o início da simulação. A factibilidade de matar todos os porcos é definida por um agente inteligente que controla o jogador durante a simulação. Além disso, a função de aptidão possui parâmetros para definir algumas características estruturais desejadas nos níveis, como o número de blocos e pássaros.

O algoritmo inicia com uma população composta por níveis aleatórios, isto é, tanto a pilha de blocos quanto a quantidade de pássaros são geradas aleatoriamente. Durante cada geração, os níveis são avaliados pela função de aptidão, que mede a estabilidade e a factibilidade deles. Os valores de aptidão calculados são utilizados em um algoritmo de seleção por torneio que define os níveis que serão utilizados para reprodução. Se um nível é infactível, ele é penalizado de maneira a ter uma pequena probabilidade de ser selecionado para reprodução. Novos níveis são construídos a partir de operações de recombinação e mutação, que são realizadas com os níveis selecionados. Essas operações visam manter a estabilidade e a factibilidade dos níveis presentes na população. Todos os níveis novos formam a próxima geração do algoritmo, exceto aquele com menor aptidão, que é substituído pelo nível com maior aptidão da geração corrente (elitismo). O algoritmo termina quando uma certa quantidade de gerações é alcançada ou caso a aptidão do melhor nível não melhore depois de dez gerações.

3.1 Representação dos Níveis

Os níveis são codificados em um genótipo composto por pilhas de blocos, de forma similar às codificações apresentadas em (FERREIRA; TOLEDO, 2014b) e (FERREIRA; TOLEDO, 2014a). Nesses dois trabalhos, o genótipo é definido como um vetor de pilhas de blocos, sendo que cada bloco é representado por um número inteiro. O tamanho do vetor pode ser diferente para cada nível e a altura de cada pilha pode ser variável dentro do mesmo nível. Um bloco pode ser elementar ou composto, sendo que os elementares são peças unitárias que podem ser agrupadas para formar blocos compostos. A quantidade de pássaros não foi codificada nesses dois artigos, assim, todos os níveis possuíam o mesmo valor para essa propriedade.

A representação de níveis nesse trabalho estende as que foram apresentadas em (FERREIRA; TOLEDO, 2014b) e (FERREIRA; TOLEDO, 2014a). Nessa nova versão, a representação contempla a quantidade de pássaros do nível e, além disso, os blocos das pilhas podem ser duplicados. Outra extensão é a classificação dos blocos com base em suas formas. Cada bloco pertence, necessariamente, a uma das seguintes classes: R , T , C ou X . Os blocos da classe R possuem formas retangulares, os da classe T possuem formas de pódios (similares à letra “T”), os da classe C possuem formas circulares e os blocos da classe X são caixas, isto é, blocos retangulares que podem cobrir outros blocos.

Figura 9 – Exemplos de blocos de diferentes classes, os três primeiros são compostos e os outros são elementares.

Esse sistema de classificação foi definido para ajudar a construção de pilhas estáveis, uma vez que alguns blocos irão com certeza prejudicar a estabilidade das pilhas quando colocados em cima de outros. Assim, cada classe possui um conjunto de “relações seguras” com outras classes, as quais estão ilustradas na Figura 10. Um nó no grafo representa uma classe e uma aresta do nó a para b representa que um bloco da classe a pode ser empilhado em cima de outro da classe b . Como mostrado na Figura 10, apenas caixas (X) podem ser empilhadas em cima de blocos do tipo C e qualquer um pode ser empilhado em cima de blocos do tipo T , R ou X . Dessa forma, sempre que o algoritmo for empilhar um bloco em cima de outro, ele primeiro verifica se um possui uma relação segura com o outro. Esse grafo de relações, por si só, não garante a estabilidade das pilhas, porém ele evita que blocos que certamente iriam desestabilizar a pilha sejam empilhados.

Figura 10 – Um grafo de relações seguras entre as diferentes classes de blocos.

Como descrito anteriormente, os blocos dessa nova representação podem ser duplicados. Um bloco duplicado é adicionado na pilha ao lado de outro exatamente igual a

ele, assim como mostra a Figura 12b. Essa representação também codifica a quantidade de pássaros do nível, adicionando um número inteiro para o genótipo. Para decodificar o genótipo em fenótipo, primeiramente, a quantidade de pássaros representada é adicionada ao nível, sendo que o primeiro é posicionado no estilingue e os restantes são posicionados atrás dele. O segundo passo consiste em posicionar as pilhas de blocos, uma na frente da outra, sem lacunas entre elas. Haverão lacunas no nível apenas se uma ou mais pilhas forem vazias. Uma pilha vazia adiciona uma lacuna de largura constante entre as suas pilhas adjacentes. Para adicionar uma pilha não vazia, o primeiro bloco é posicionado sobre o chão e os restantes são empilhados uns em cima dos outros, sempre em uma posição centralizada. A posição das pilhas é definida de tal maneira que a distância entre a primeira pilha e o estilingue seja igual à distância entre a última pilha e o final do nível.

A Figura 11 ilustra um genótipo de nível e o seu fenótipo correspondente. O primeiro elemento do genótipo representa a quantidade de pássaros e todos os outros representam as pilhas. Cada elemento de uma pilha codifica informações sobre um bloco, o qual contém um número inteiro que representa o índice do bloco e uma variável booleana que define se o bloco será duplicado. Nesse caso particular, o nível possui quatro pássaros e oito pilhas de blocos, sendo que a terceira, a quinta e a sétima pilhas estão vazias. Todas as restantes possuem um porco no topo (elemento 31). Analisando as pilhas de baixo para cima, o segundo bloco da primeira, os três primeiros blocos da segunda e o primeiro bloco da sexta estão duplicados.

3.2 Inicialização Aleatória

A primeira população do AG é inicializada com níveis que possuem uma quantidade aleatória, maior que zero, de pilhas e pássaros. Cada pilha também é formada por uma quantidade aleatória, maior ou igual a zero, de blocos. Em uma pilha não vazia, ao menos um elemento precisa ser um porco. O Algoritmo 2 descreve em detalhes o processo de geração de um nível aleatório. Inicialmente, ele gera um número inteiro aleatório $1 \leq b \leq B$ utilizando uma distribuição uniforme (linha 2), sendo que B é uma constante que define a quantidade máxima de pássaros permitida. Em seguida, o algoritmo gera a largura (no espaço do mundo do jogo) do vetor de pilhas usando uma distribuição normal com μ e σ iguais a metade da largura W da área útil do nível (linha 4). Depois disso, o algoritmo iterativamente gera as pilhas até que a largura definida anteriormente seja preenchida (linha 5-21). Para formar uma pilha, o algoritmo primeiro gera a sua altura (também no espaço do mundo do jogo) usando uma distribuição normal com μ e σ iguais a metade da altura máxima H no nível (linha 9). Com a altura definida, o algoritmo começa a empilhar blocos aleatórios um a um, sempre considerando o grafo de relações seguras. Esse processo é realizado até que a pilha alcance a altura definida ou até que não existam mais blocos que possam ser empilhados de maneira segura. (linha 10).

Figura 11 – Um exemplo de genótipo e o seu fenótipo correspondente.

Para adicionar blocos em uma pilha, o algoritmo aleatoriamente seleciona uma classe c_{prox} com base em suas relações (veja Figura 10) com o último bloco empilhado c_{ant} (linha 11). O novo bloco l_{prox} é aleatoriamente escolhido dentre os elementos do vetor V que são da classe c_{prox} (linha 12). A função **EscolheBlocoAleatorioDaClasse** remove o bloco selecionado de V , o que garante uma seleção sem reposição. Se esse for o primeiro bloco da pilha, qualquer classe pode ser selecionada e, consequente, qualquer bloco também. Se o bloco selecionado não for uma caixa (linha 13), o próximo passo é verificar se ele também não é um pódio e “lançar uma moeda” para definir se ele será duplicado ou não (linha 14). Pódios não podem ser duplicados porque eles podem facilmente desestabilizar a pilha. A Figura 12a mostra um exemplo de um bloco do tipo T (pódio) duplicado que prejudica a estabilidade da pilha.

Depois de definir se um bloco será duplicado, o algoritmo compara as áreas dos quadrados delimitadores (*bounding boxes*) do bloco anterior e do próximo bloco (se esse não for o primeiro). Essa comparação é uma solução simplificada para o problema de descobrir se dois blocos não quadrados empilhados cairão quando submetidos à força da gravidade (WANG *et al.*, 2009). Ao invés de analisar as formas reais dos blocos, o algoritmo assume que todos eles possuem formas quadradas. Assim, se a área do quadrado delimitador do bloco anterior for maior que a do próximo (linha 16), então l_{prox} é adicionado na pilha

(a) Blocos duplicados de maneira insegura. (b) Blocos duplicados de maneira segura.

Figura 12 – Exemplo de blocos duplicados.

corrente S (linha 17) e o contador da altura preenchida $h_{preenchida}$ é incrementado com a altura h_{prox} desse bloco (linha 18). Por outro lado, se o bloco selecionado é uma caixa, o algoritmo tenta cobrir o número máximo de blocos do topo da pilha que cabem dentro dele

Algoritmo 2: Gerando níveis aleatórios.

```

1   $A \leftarrow \{\}$ ;
2   $b \leftarrow U(1, B)$ ;
3   $w_{preenchida} \leftarrow 0$ ;
4   $w_A \leftarrow N(W * 0.5, W * 0.5)$ ;
5  enquanto  $w_{preenchida} < w_A$  faça
6       $S \leftarrow \{\}$ ;
7       $V \leftarrow$  todos os blocos;
8       $h_{preenchida} \leftarrow 0$ ;
9       $h_s \leftarrow N(H * 0.5, H * 0.5)$ ;
10     enquanto  $h_{preenchida} < h_i$  e  $V$  não está vazio faça
11          $c_{prox} \leftarrow \text{EscolheClasseSeguraAleatoria}(c_{ant})$ ;
12          $l_{prox} \leftarrow \text{EscolheBlocoAleatorioDaClasse}(c_{prox}, V)$ ;
13         se  $c_{prox}$  não é uma caixa então
14             se  $c_{prox}$  não é um pódio e  $U(0, 1) < 0.5$  então
15                 DuplicaBloco( $l_{prox}$ );
16             se  $\text{AreaQuadDelimitador}_{ant} > \text{AreaQuadDelimitador}_{prox}$  então
17                 adiciona  $l_{prox}$  em  $S$ ;
18                  $h_{preenchida} \leftarrow h_{preenchida} + h_{prox}$ ;
19         senão se blocos no topo de  $S$  cabem dentro de  $l_{prox}$  então
20             adiciona  $l_{prox}$  em  $S$ ;
21              $h_{preenchida} \leftarrow h_{preenchida} + h_{prox}$ ;
22     InserePorcos( $S$ );
23     se  $h_{preenchida} > 0$  então
24          $w_{preenchida} \leftarrow w_{preenchida} + w_s$ ;
25     senão
26          $w_{preenchida} \leftarrow w_{preenchida} + k$ ;
27     adiciona  $S$  em  $A$ ;

```

(linha 19). Dependendo da caixa selecionada e do estado da pilha, a caixa não será capaz de cobrir os blocos e, nesse caso, ela não é adicionada. Caso contrário, ela é adicionada (linha 20) e o contador de altura é incrementado (linha 21).

O último passo para a criação de uma pilha é adicionar os porcos em espaços disponíveis, os quais podem ser espaços vazios dentro de caixas ou o topo de uma pilha. A função **InserePorcos** (linha 22) adiciona uma quantidade aleatória de porcos em uma pilha, iterando entre todos os seus blocos e procurando por espaços disponíveis. Se essa função receber uma pilha vazia como entrada, ela simplesmente “lança uma moeda” para decidir se o porco será adicionado ou não. Se o porco for adicionado, ele será posicionado diretamente no chão do nível. Se a pilha de entrada não for vazia, a função itera entre seus elementos e se um espaço disponível for encontrado, ela “lança uma moeda” para decidir se o porco será adicionado ou não naquela posição. Se a função alcançar o topo da pilha sem ter adicionado porcos, ela automaticamente adiciona um porco no topo da pilha. Nesse caso, se o último bloco da pilha tiver uma forma circular, o algoritmo substitui esse bloco pelo porco.

Uma vez que uma pilha está completamente criada, o algoritmo incrementa o contador de largura preenchida $w_{preenchida}$ com a largura da pilha w_s (linha 24) ou com uma constante k (linha 26). A segunda opção acontece apenas se a pilha criada for vazia. Finalmente, o algoritmo adiciona a nova pilha no vetor A (linha 27) e tenta gerar a próxima. Se, ao final da execução, todas as pilhas estiverem vazias, o algoritmo escolhe aleatoriamente uma pilha e a substitui por outra com altura igual a $H * 0.5$, composta apenas por blocos retangulares e por um porco no topo.

3.3 Função de Aptidão

O algoritmo de inicialização não garante que as pilhas criadas são estáveis e nem que a quantidade de pássaros adicionados é suficiente para matar todos os porcos. Nesse trabalho, a função de aptidão é responsável por medir se um nível satisfaz essas duas restrições. Essa medida é calculada via uma simulação do jogo com um agente inteligente (Bot) que joga os níveis. A estabilidade é definida pela velocidade total dos blocos durante o início da simulação. Quando ela inicia, o agente espera para atirar o primeiro pássaro até que os blocos estejam todos estáveis, isto é, até que a magnitude de seus vetores velocidade sejam iguais a zero. Durante esse período, o algoritmo mede a velocidade de cada bloco e acumula os valores obtidos em uma variável s .

A factibilidade de matar todos os porcos é definida pela estratégia de jogo utilizada pelo agente. Um nível é considerado factível se a quantidade de porcos p_f no final da simulação é igual a zero. Na estratégia proposta, o agente primeiro seleciona um porco aleatório do nível e tenta lançar um pássaro nele. Se o agente não tiver sucesso, um porco

é escolhido aleatoriamente de novo. Se o mesmo porco for escolhido duas vezes seguidas, o agente adiciona um ruído na posição do porco e atira o pássaro nessa direção. A ideia do ruído é tentar quebrar blocos em volta que possam estar protegendo o porco. Por outro lado, se o porco p_i selecionado no i -ésimo lançamento for diferente do porco p_{i-1} anterior, o agente irá atirar diretamente na direção do porco p_i . Quando o agente lança um pássaro em direção a um porco, ele não considera se existem blocos em sua frente. Além disso, um pássaro é sempre lançado com a maior velocidade possível.

Como mencionado anteriormente, além de avaliar estabilidade e factibilidade, a função de aptidão também avalia algumas características estruturais dos níveis. Mais especificadamente, ela mede a quantidade total de blocos e o número de pássaros que são necessários para matar todos os porcos. Essas métricas são usadas para controlar a saída do algoritmo. Existem dois parâmetros $0 \leq b_n \leq 1$ e $0 \leq l_n \leq 1$ que representam a porcentagem de pássaros que o jogador precisa usar para completar o nível e a porcentagem de blocos que o nível deve ter, respectivamente. A função de aptidão avalia os níveis considerando esses parâmetros, assim, ela pode ser matematicamente descrita pela Equação 3.1

$$aptidao(x) = |[b_n * B] - b_u| + |[l_n * L] - l_b| + p_f + s \quad (3.1)$$

Na função proposta, B é uma constante que define a quantidade máxima de pássaros permitida, b_u é a quantidade de pássaros lançados durante a simulação, L é outra constante que define a quantidade máxima de blocos permitida e l_b é a quantidade de blocos no início da simulação. O primeiro termo da equação calcula a distância entre o número de pássaros que deveria ter sido usado para matar os porcos e o número que foi, de fato, utilizado. O segundo termo calcula a distância entre o número de blocos desejados e o número de blocos presentes no nível logo após sua inicialização. Se ambos os termos forem zero, o nível possui todas as características estruturais desejadas. Os termos restantes da equação avaliam o número p_f de porcos ao final da simulação e a estabilidade s dos blocos, respectivamente. Um nível é considerado estável e factível quando essa duas variáveis valem ambas zero. O design da Equação 3.1 mostra que o objetivo do AG proposto é encontrar o valor mínimo da função de aptidão.

Um dos principais problemas do uso de simulação para avaliação de níveis em jogos com mecânicas baseadas em física é que o seu resultado não é determinístico. Mesmo que a estratégia utilizada pelo agente para jogar o nível seja determinística, os motores de física normalmente não são. Assim, não há uma garantia de que o mesmo nível terá a mesma aptidão se ele for avaliado duas vezes. Esse problema é conhecido como otimização com função de aptidão ruidosa, assim como descrito no Capítulo 2 (PIETRO; WHILE; BARONE, 2004). A solução proposta nesse trabalho consiste em nunca recalculer a aptidão de um mesmo indivíduo. Isso é feito por meio de uma memória cache, estruturada como

uma tabela hash, que armazena os níveis como chaves e suas aptidões como valores. Dessa forma, toda vez que o AG precisa avaliar um nível, ele primeiro procura pelo nível na memória. Caso ele o encontre, o valor de aptidão armazenado é retornado, caso contrário, uma simulação é executada e uma nova instância é adicionada na memória. Para armazenar e acessar elementos na cache, o algoritmo cria as chaves por meio de uma função hash que considera todos os elementos da representação (genótipo) do nível. Essa solução para tratar ruído possui duas vantagens principais: redução no número de simulações e liberdade para desenvolver a estratégia do agente inteligente. Por outro lado, ela pode aumentar o consumo de memória do algoritmo, dependendo da sua acurácia em explorar o espaço de busca.

3.4 Operadores Genéticos

A função de aptidão avalia cada nível considerando a sua estabilidade, factibilidade e características estruturais. O valor de aptidão atribuído por essa função é utilizado pelo AG para selecionar n indivíduos para reprodução, sendo que n é o tamanho da população. Esse processo é realizado iterativamente e, em cada uma das n iterações, um indivíduo é selecionado por meio de um torneio composto por dois pais aleatórios diferentes. Uma nova geração é então formada por meio da aplicação de uma operação de recombinação em cada par de pais selecionados, seguida de uma mutação em cada filho gerado.

Ambos os operadores genéticos são compostos por duas etapas, sendo que a primeira é aplicada na quantidade de pássaros e a segunda nas pilhas de blocos. As quantidades de pássaros de cada nível são recombinadas por meio de um cruzamento aritmético. Assim, considerando dois pais p_1 e p_2 com x_1 e x_2 pássaros, respectivamente, o cruzamento aritmético produz duas novas quantidades de pássaros y_1 e y_2 utilizando a Equação 3.2, sendo que $0 \leq r \leq 1$ é um fator aleatório (peso) escolhido antes de cada operação. Uma vez que a quantidade de pássaros é um número inteiro, os dois resultados da recombinação são transformados em inteiros usando a função *piso*.

$$\begin{aligned} y_1 &= \lfloor r * x_1 + (1 - r) * x_2 \rfloor \\ y_2 &= \lfloor (1 - r) * x_1 + r * x_2 \rfloor \end{aligned} \tag{3.2}$$

A outra parte da operação de recombinação tem como objetivo a geração de dois novos vetores de pilhas de blocos a partir de dois pais p_1 e p_2 com m e n pilhas, respectivamente. Essa operação gera a pilha $i \in [1, \max(m, n)]$ de um novo indivíduo (filho) selecionando aleatoriamente uma pilha de p_1 ou p_2 com igual probabilidade. Se o tamanho dos vetores dos dois pais forem diferentes ($m \neq n$) e assumindo que $m > n$, cada uma das $m - n$ colunas de p_1 terá 50% de chance de ser incluída no novo indivíduo. Caso contrário

($n > m$), cada uma das $n - m$ colunas de p_2 terá 50% de chance de ser escolhida. A Figura 13 mostra um exemplo dessa operação.

Figura 13 – Um exemplo de operação de recombinação. Os dois indivíduos na parte de cima da figura são os pais e os outros, na parte de baixo, são os filhos gerados.

Nesse exemplo, o fator aleatório r do cruzamento aritmético possui valor 0.2 e as quantidades de pássaros de cada indivíduo são 1 e 4. Assim, o resultado gerou o valor 3 para o primeiro filho e 2 para o segundo. Para gerar as pilhas de blocos do primeiro filho, a recombinação selecionou a primeira pilha do primeiro pai e as últimas duas pilhas do segundo pai. Para gerar as pilhas do segundo filho, foram selecionadas a primeira pilha do segundo pai e a segunda pilha do primeiro pai. Nesse caso, a recombinação não selecionou a terceira pilha do segundo pai, assim, a terceira pilha do segundo filho ficou vazia.

Figura 14 – Exemplos de operação de mutação aplicados sobre os dois indivíduos gerados na Figura 13.

Depois que operação de recombinação gera os n novos indivíduos, a mutação é aplicada sobre cada um deles. A mutação altera de forma aleatória a quantidade de pássaros e cada pilha de blocos com uma certa probabilidade. Se a quantidade de pássaros for selecionada para mutação, o seu valor é trocado por outro inteiro aleatório $1 \leq b \leq B$, sendo que B é a constante do AG que define o número máximo de pássaros que os níveis podem ter. Por outro lado, se um elemento do vetor de pilhas for selecionado para mutação, ele será trocado por outra pilha gerada aleatoriamente, de maneira similar ao algoritmo de inicialização do AG (veja Seção 3.2). A Figura 14 ilustra um exemplo de operação de mutação na qual a quantidade de pássaros do primeiro filho da Figura 13 foi alterada de 3 para 5. Essa operação também alterou a segunda pilha desse mesmo filho. Uma outra mutação pode ser vista sobre o segundo filho da Figura 14, a qual alterou apenas a sua terceira pilha.

EXPERIMENTOS E RESULTADOS

O AG definido neste trabalho foi avaliado por três conjuntos de experimentos. O primeiro deles avalia o mecanismo de controle de ruído usando cache. O segundo analisa a expressividade do AG enquanto um gerador de níveis. O último experimento é um teste com jogadores humanos para medir aspectos de design dos níveis gerados. Dentre os aspectos avaliados estão dificuldade, imersão e diversão. Em todos os experimentos, o AG foi configurado com os seguintes valores de parâmetros: tamanho da população igual a 100, torneio de tamanho 2, taxa de recombinação igual a 95% e taxa de mutação de 10%. Além disso, as quantidades máximas de pássaros (B) e blocos permitidas (L) foram definidas em 5 e 100, respectivamente. Esses valores foram escolhidos empiricamente com base em experimentos anteriores.

4.1 Avaliação do Impacto do Ruído

Nesse conjunto de experimentos, a abordagem de cache proposta foi comparada com outros dois métodos da literatura: **Médias Explícitas** (ME) (SANO; KITA, 2002) e **Médias Implícitas** (MI) (RATTRAY; SHAPIRO, 1997). Os métodos foram comparados por meio da evolução das médias, de 5 execuções, das aptidões dos melhores indivíduos. A média e o desvio padrão do tamanho da cache bem como o tempo de execução de cada método também foram considerados na comparação. O AG foi configurado para gerar níveis com os parâmetros $b_n = 0.25$ e $l_n = 1$. Dessa forma, os níveis gerados deveriam possuir 100 blocos e apenas um quarto dos pássaros disponíveis deveria ser suficiente para que o jogador matasse todos os porcos. Todos os métodos foram executados até que o algoritmo atingisse 50 gerações ou caso a aptidão do melhor indivíduo da população não aumentasse durante 10 gerações.

A primeira análise compara a abordagem de cache com o método ME. Três diferentes tamanhos de amostras foram utilizados pelo método ME para calcular a média dos valores

de aptidão: **2**, **4** e **8**. A Figura 15 mostra os resultados obtidos. No método ME o impacto do ruído foi reduzido com o aumento da amostra, o que pode ser evidenciado pela quantidade de vales de cada gráfico. Com o tamanho da amostra igual a 2 (ME-2), o algoritmo de médias rapidamente melhorou a sua solução inicial, porém ele não foi capaz de convergir para a solução ótima (aptidão igual a zero). No início das execuções, a melhor solução teve uma média de aptidão igual a 7 e alcançou uma aptidão com valor médio igual a 3 logo na terceira geração. Durante as próximas gerações houve uma grande ocorrência de ruído que se destacou entre a sexta e a vigésima geração. A partir da vigésima até a última geração, as execuções mantiveram uma aptidão média entre 1 e 2. A grande presença de ruído e a estagnação no valor médio de aptidão durante mais da metade das gerações, mostra que o AG com ME-2 não foi capaz de evoluir efetivamente os níveis.

Figura 15 – Média da aptidão do melhor indivíduo (eixo y) por geração (eixo x).

A presença de ruído foi suavizada no método ME com o tamanho da amostra igual a 4 (ME-4). Novamente, a aptidão do melhor indivíduo teve uma melhoria considerável nas primeiras cinco gerações. O número de vales no gráfico entre a sexta e a vigésima geração é menor, o que demonstra um número menor de ruído no cálculo da aptidão. No entanto, o algoritmo novamente não convergiu para a solução ótima. Ao final das gerações, o AG com ME-4 alcançou uma aptidão média de aproximadamente 1, sendo ligeiramente melhor que o ME-2. O impacto do ruído na função de aptidão foi ainda menor com o ME utilizando 8 amostras para o cálculo das médias (ME-8). A aptidão média obtida nas 5 execuções seguiu um comportamento similar às versões ME-2 e ME-4, pois nesse caso também houve uma otimização rápida nas primeiras gerações e, ao final, a aptidão média foi aproximadamente igual a 1. Porém, o gráfico do ME-8 mostra uma redução relevante

na presença de ruído, principalmente a partir da trigésima geração. Apesar do método ME ter reduzido a quantidade de ruído da função de aptidão, ele não foi capaz de atingir o seu ótimo global. Isso pode ser explicado pela dificuldade em garantir a propriedade do elitismo na evolução.

Analisando o gráfico referente ao AG com cache na Figura 15, observa-se que o seu comportamento evolutivo seguiu um padrão similar aos outros. Mais uma vez ocorreu uma evolução considerável nas primeiras gerações e uma pequena otimização durante quase todas as gerações. Porém, nesse último caso, o AG foi capaz de escapar de um possível ótimo local (região com soluções de aptidão igual a 1) e alcançar a solução ótima nas últimas gerações. Além de ser o único a encontrar uma solução com aptidão ótima em todas as cinco execuções, o método de cache proposto possibilitou uma correta evolução elitista. O que é explicado pelo fato de que as aptidões dos indivíduos são calculadas apenas uma vez durante todo o algoritmo.

A abordagem ME utiliza um maior número de avaliações de aptidão para lidar com o problema do ruído, assim, mais tempo computacional é necessário durante a evolução do AG. O tempo de execução médio das 5 execuções é mostrado na Figura 15 abaixo de cada título dos gráficos. Como esperado, quanto maior o tamanho da amostra do ME, maior o seu tempo de execução. O ME-2 levou 9495 segundos em média para encontrar soluções com aptidões entre 1 e 2, enquanto o ME-4 e o ME-8 gastaram, respectivamente, 18367 e 25445 em média. A abordagem baseada em cache gastou apenas 2755 segundos em média e, ainda assim, foi o único dos métodos a encontrar soluções ótimas em suas execuções. Esses resultados mostraram que o AG com cache foi capaz de evoluir muito mais rápido que os outros métodos e com maior acurácia.

O desempenho do método de cache, em tempo de execução, foi melhor que os outros métodos avaliados. Porém, ele necessita de uma quantidade extra de memória para atingir estes resultados. Se o tamanho da memória cache for muito grande, o método pode não ser aplicável em sistemas com pouca memória física. Dessa forma, o tamanho médio da memória cache foi avaliado durante as mesmas cinco execuções dos experimentos anteriores. A Figura 15 mostra que o tamanho médio da cache foi de 4881.4 indivíduos e que a cache foi acessada 218.6 vezes em média. A proporção entre o tamanho da cache e o número de vezes que ela foi acessada mostra que apenas uma pequena porção de indivíduos foram gerados repetidamente ao longo das gerações. Isso indica que o AG teve uma boa capacidade de exploração do espaço de busca. No entanto, o tamanho da cache foi um ponto negativo do método, pois ela chegou a ter quase o mesmo tamanho da população.

A segunda análise desse conjunto de experimentos compara o método de cache com o de médias implícitas (MI). Três diferentes tamanhos de populações foram avaliados com o método MI: **200** (MI-200), **400** (MI-400) e **800** (MI-800). Os resultados estão reportados na Figura 16, que ilustra a média da evolução do melhor indivíduo depois de

cinco execuções do AG. No MI-200, a aptidão média do melhor indivíduo foi igual a 5 na primeira geração. Esse valor diminuiu para aproximadamente 2.5 logo na segunda geração. Essa abordagem teve um período muito ruidoso durante a primeira metade das execuções. Na segunda metade o ruído diminuiu e o algoritmo foi capaz de alcançar, nas últimas gerações, uma aptidão média entre 1 e 2.

Figura 16 – Média da aptidão do melhor indivíduo (eixo y) por geração (eixo x).

O método MI-400 começou com uma aptidão média de aproximadamente 3. Nas próximas 5 gerações ocorreu um período de melhoria significativa, onde a aptidão média chegou a valer 1. A partir da quinta geração até o final das execuções houve uma grande presença de ruído, com valores médios de aptidão oscilando entre 0.2 e 1. Os resultados obtidos com as execuções do método MI-800 tiveram um comportamento muito similar aos resultados do MI-400. Porém o impacto do ruído foi menor com a população de 800 indivíduos, o que pode ser evidenciado pela menor oscilação de aptidão entre as gerações.

Assim como as execuções com o método ME, as três configurações de MI obtiveram uma acurácia pior que o AG com cache, pois elas também não alcançaram o ótimo global. Os tempos de execução das abordagens MI-200, MI-400 e MI-800 foram 5918, 16248 e 36448, respectivamente. Esses tempos mais do que dobraram nas duas vezes que o tamanho da população também foi dobrado. Esses resultados mostram que o desempenho do AG com cache também foi muito melhor que qualquer uma das variações de MI, uma vez que a abordagem de cache chegou a ser aproximadamente 2.5 vezes mais rápida do que a versão com menor população do MI.

4.2 Análise de Expressividade

Os resultados da última seção analisaram o desempenho do método de cache proposto em relação a outros métodos da literatura. Essa seção contém experimentos para avaliar a expressividade do AG, ou seja, a diversidade de níveis que ele é capaz de gerar. A análise de expressividade foi proposta por (SMITH; WHITEHEAD, 2010) como uma forma de avaliar a variedade de níveis que um gerador é capaz de criar, bem como o impacto que os seus parâmetros de entrada causam nesses níveis. A metodologia para essa análise foi definida da seguinte maneira:

- **Determinar métricas apropriadas.** Encontrar métricas para descrever os níveis visando analisar a diversidade de conteúdos que podem ser criados.
- **Gerar conteúdo.** Coletar uma amostra representativa de níveis executando o gerador um grande número de vezes. Essa amostra deve conter níveis gerados com diferentes variações de parâmetros, incluindo o caso no qual todos eles são ponderados igualmente.
- **Visualizar o espaço de níveis gerados.** Apresentar os resultados por meio de gráficos, nos quais os eixos são definidos pela variação de valores das métricas definidas.
- **Analisar o impacto dos parâmetros.** Comparar como os níveis gerados são influenciados pela mudanças de parâmetros. Utilizar as amostras geradas a partir de uma parametrização ponderada igualmente como modelo de comparação.

Os níveis gerados, baseados no *Angry Birds*, foram avaliados a partir da adaptação de três métricas da literatura: frequência (SHAKER; TOGELIUS; NELSON, 2013), linearidade (SMITH; WHITEHEAD, 2010) e densidade (SMITH; WHITEHEAD, 2010). A frequência representa a proporção de determinados tipos de elementos em um nível. Ela é calculada somando as instâncias de elementos do tipo desejado e dividindo pela quantidade total de elementos. Nesse caso particular, os elementos de interesse são os pássaros e os porcos que foram adicionados. A frequência de blocos não é considerada porque a quantidade de blocos é um parâmetro do algoritmo (l_n).

A métrica de linearidade é calculada por meio da variância das alturas (no espaço do mundo do jogo) das pilhas de blocos, incluindo as que são vazias. Os resultados são normalizados em $[0, 1]$, sendo que 1 significa que o nível é totalmente linear e 0 que ele é totalmente não-linear. Assim, níveis com pilhas de diferentes tamanhos terão baixa linearidade e níveis com pilhas de tamanhos similares terão alta linearidade. A Equação 4.1 mostra como calcular essa métrica para um nível, sendo que h_i é a altura da pilha i , $\bar{h}$

é a altura média das pilhas e H é a altura máxima de um nível. A Figura 17 ilustra dois níveis com diferentes valores de linearidade.

$$linearidade = 1 - \frac{\sqrt{\sum_{i=1} (h_i - \bar{h})^2}}{\sqrt{\sum_{i=1} (h_i - H)^2}} \quad (4.1)$$

(a) Um nível com alta linearidade (0.98).

(b) Um nível com baixa linearidade (0.14).

Figura 17 – Exemplos de níveis com uma grande diferença de linearidade. O nível (a) possui alta linearidade porque possui pilhas com alturas similares. Por outro lado, as pilhas do nível (b) possuem diferentes alturas, por isso a sua linearidade é baixa.

A densidade mede a disposição das pilhas ao longo da área útil do nível. Essa métrica é calculada somando a largura de todas as suas pilhas não-vazias, dividindo pela largura W de sua área útil. Os valores de densidade são normalizados em $[0,1]$, sendo que densidade 1 significa que a área útil do nível está totalmente preenchida com pilhas. Por outro lado, densidade 0 significa que o nível não possui pilhas ou que todas as suas pilhas estão vazias. A Equação 4.2 descreve como a densidade de um nível é calculada, sendo que w_i é a largura da i -ésima pilha não-vazia e W é a largura da área útil do nível. A Figura 18 ilustra dois níveis com diferentes densidades.

$$densidade = \frac{\sum_{i=1} w_i}{W} \quad (4.2)$$

Uma amostra de 500 níveis foi utilizada para avaliar a expressividade do AG proposto. Para gerar essa amostra foram utilizadas diferentes combinações para os parâmetros b_n e l_n . Primeiramente, o valor de b_n foi fixado em 0.5 e 100 níveis foram gerados para cada uma das seguintes variações de l_n : 0.25, 0.5 e 1.0. Em seguida, o valor de l_n foi fixado em 0.5 e 100 níveis foram gerados tanto para $b_n = 0.25$ quanto para $b_n = 1.0$. A primeira análise avalia a frequência de porcos e pássaros nos níveis gerados. Esses valores foram normalizados considerando a soma total de blocos e porcos dos níveis. A Figura 19 mostra

(a) Um nível com alta densidade (0.98).

(b) Um nível com baixa densidade (0.14).

Figura 18 – Um exemplo de dois níveis com grandes diferenças de densidade. O nível (a) possui alta densidade porque as suas pilhas ocupam quase toda a sua área útil. Por outro lado, o nível (b) possui densidade baixa pois suas pilhas ocupam apenas uma pequena porção da área útil.

cinco gráficos contendo a média e o desvio padrão das frequências para cada variação dos parâmetros b_n e l_n .

Figura 19 – Impacto na frequência de pássaros e porcos quando os parâmetros b_n e l_n são variados.

Em todos os gráficos, a frequência de porcos se manteve entre 10% e 20% da quantidade total dos objetos do nível. A quantidade de pássaros necessária para completar o nível não influencia o processo de adicionar porcos, por isso, o parâmetro b_n não influenciou a frequência dos porcos. Por outro lado, os resultados mostraram que a proporção de porcos foi mantida quando o número de blocos aumentou. Esse resultado era esperado, pois os porcos são inseridos nas pilhas, com uma certa probabilidade, dentro de caixas ou nos topos. Assim, o número de porcos no nível é diretamente proporcional ao número de blocos adicionados. Também pode-se concluir desses resultados que a porcentagem de caixas nos níveis foi menor do que 20%.

A frequência de pássaros foi normalizada pelo número máximo B de pássaros, que foi pré-determinado como 5. Os resultados da Figura 19 mostram que a frequência de pássaros variou entre 0.5 e 1 na maioria dos casos. Isso significa que a maioria dos níveis eram compostos por 2 ou mais pássaros (no máximo 5), ou seja, eles dificilmente tiveram apenas um pássaro. Quando o parâmetro b_n foi fixado em 0.5 e o l_n foi variado, nota-se que a frequência média de pássaros aumentou, a medida que o valor de l_n também aumentou. Esse resultado leva a uma conclusão esperada de que níveis com mais blocos precisam de mais pássaros para serem finalizados.

Quando o parâmetro l_n foi fixado e b_n foi variado, a quantidade de pássaros não foi afetada diretamente (veja as colunas da direita dos três primeiros gráficos). O AG gerou níveis de 50 blocos ($l_n = 0.5$) com diferentes porcentagens de pássaros que são necessários para finalizar os níveis (b_n). Independentemente do valor de b_n , os níveis apresentaram uma quantidade de pássaros similar (entre 2 e 5). No entanto, é importante destacar que a quantidade total de pássaros afeta a dificuldade do nível. Por exemplo, em um nível com $b_n = 0.25$ e 5 pássaros o jogador tem cinco tentativas para matar todos os porcos. Em um nível idêntico, porém com 2 pássaros, o jogador tem apenas duas tentativas para matar os porcos.

Níveis com uma quantidade maior de blocos tendem a possuir pilhas mais altas, o que pode aumentar a fragilidade de toda a estrutura do nível. Se a diferença de altura entre essas pilhas for muito grande, a chance de um pássaro lançado destruir blocos e porcos é grande. Essa chance aumenta ainda mais se existirem espaços vazios entre as pilhas. No entanto, os resultados mostraram que níveis com mais blocos precisam de mais pássaros para matar os porcos. Isso indica que as pilhas dos níveis possuem alturas similares e que não há muitos espaços vazios entre elas. Porém, é difícil concluir as características estruturais dos níveis apenas com base nos resultados de frequência. Essas características podem ser discutidas com maior profundidade a partir dos resultados de linearidade e densidade.

As Figuras 20 e 21 ilustram como a variação de parâmetros impacta na linearidade e na densidade dos níveis. A primeira figura mostra que a maioria dos níveis com 50 blocos possuem linearidade alta (entre 0.8 e 1) e um valor intermediário de densidade (entre 0.4 e 0.6). Esses resultados confirmam que a maioria dos níveis possui pilhas com alturas similares. Além disso, os valores de densidade mostram que as pilhas ocupam aproximadamente metade da área útil dos níveis. Isso indica que o AG possui um viés em distribuir os blocos entre poucas pilhas, ao invés de criar mais pilhas com alturas menores. É importante notar que as alterações no parâmetro b_n não causaram modificações significativas nos valores de linearidade e na densidade dos níveis.

Figura 20 – Impacto do parâmetro b_n nas métricas de linearidade e densidade.

A Figura 21 mostra que o parâmetro l_n teve um impacto relevante na linearidade e na densidade dos níveis. A maioria dos níveis com 25 blocos ($l_n = 0.25$) possuem linearidade

alta (entre 0.8 e 1.0) e densidade baixa (entre 0.2 e 0.4). Como mostrado anteriormente, os níveis com 50 blocos ($l_n = 0.5$) também possuem linearidade alta (entre 0.8 e 1), mas seus valores de densidade são intermediários (entre 0.4 e 0.6). Os níveis com 100 blocos ($l_n = 1$) apresentaram linearidade intermediária e alta densidade. Esses resultados mostram que a densidade dos níveis aumenta de maneira proporcional ao número de blocos. Além disso, o aumento do número de blocos diminui levemente a linearidade. Esse resultado era esperado, uma vez que níveis com muitos blocos precisam de mais área útil.

Figura 21 – Impacto do parâmetro l_n nas métricas de linearidade e densidade.

4.3 Análise de Imersão

Essa seção apresenta o último experimento desse trabalho, que tem como objetivo analisar aspectos de imersão dos níveis gerados. De acordo com (JENNETT *et al.*, 2008), imersão pode ser descrita como o grau de envolvimento de um jogador com um jogo digital. Dessa forma, um jogo imersivo tem a capacidade de prender a atenção das pessoas, fornecendo uma distração atraente para as preocupações do dia-a-dia. O grau de envolvimento de um jogador depende de uma combinação de fatores humanos, computacionais e contextuais como as suas preferências de jogos, a construção do jogo e as distrações do ambiente. Considerando esses fatores, (JENNETT *et al.*, 2008) propôs um questionário composto por 31 perguntas para avaliar imersão em jogos. Uma versão menor desse questionário (veja o Apêndice B), contendo apenas 6 perguntas, foi utilizada nesse trabalho para avaliar a imersão que os níveis gerados pelo AG proporcionam aos jogadores. A mesma avaliação foi realizada com níveis adaptados da versão original do *Angry Birds*. Assim, os resultados desse experimento são apresentados em comparação com níveis do jogo original.

O experimento dessa seção foi realizado utilizando a versão clone do *Angry Birds* desenvolvida durante esse projeto. O questionário foi inserido nessa implementação do jogo, que foi compilada em uma versão web e disponibilizada para acesso via Internet ¹. Essa versão do jogo contém dois conjuntos de níveis, *P* e *R*, que são apresentados para o jogador em uma ordem aleatória. Cada conjunto possui cinco níveis, sendo que os de *P* foram gerados pelo AG e os de *R* foram coletados e adaptados do jogo original. Para formar o

¹ <<http://www.lucasnferreira.com/AngryBirdsWeb/AngryBirdsWeb.html>>

conjunto P , foram gerados três níveis para cada uma das cinco configurações de parâmetros mostradas na Figura 19. Para cada configuração foi selecionado aleatoriamente apenas um dos três, dessa forma, o conjunto P foi composto por 5 níveis com diferentes características. Para popular o conjunto R , foram selecionados 5 níveis aleatórios do primeiro mundo do jogo original, os quais foram adaptados para a versão clone do jogo. Uma das limitações do clone é a variedade de pássaros, pois ele suporta apenas os pássaros vermelhos. Assim, todos os outros foram substituídos por pássaros vermelhos. O clone também não suporta blocos de vidro, dessa forma, eles foram substituídos por blocos de madeira ou pedra. Em alguns casos a quantidade de pássaros também foi modificada, dependendo de como a falta de pássaros não-vermelhos impactou na jogabilidade dos níveis. Todos os níveis adaptados, reportados no Apêndice A, foram testados para garantir factibilidade. Com esses dois conjuntos definidos, a estrutura do experimento de imersão foi a seguinte:

1. **O participante abre a URL do jogo em seu navegador e uma tela contendo a descrição do experimento é apresentada.**

Essa tela informa ao participante que ele irá jogar dois conjuntos diferentes de níveis e avaliar cada um separadamente após cada jogo. O participante não é informado sobre a diferença entre os dois conjuntos.

2. **O questionário de perfil, descrito no Apêndice B, é apresentado para o participante.**

3. **O primeiro conjunto de níveis é apresentado ao participante.**

O conjunto é selecionado aleatoriamente dentre P ou R

4. **O participante avalia, usando o questionário de imersão do Apêndice B, o conjunto que ele acabou de jogar.**

5. **O segundo conjunto de níveis é apresentado ao participante.**

Se P já foi jogado, então R é apresentado. Caso contrário, P é escolhido.

6. **O participante repete a tarefa 4 para o segundo conjunto**

7. **Uma nova tela aparece perguntando ao jogador qual dos dois conjuntos ele acha que foi gerado por computador.**

Antes da pergunta, a tela informa ao participante que um dos conjuntos foi gerado manualmente e que outro foi gerado por computador.

8. **Outra tela aparece revelando ao participante qual dos grupos foi gerado por computador.**

9. **O jogador avalia, usando o último questionário do Apêndice B, a similaridade entre os dois conjuntos e a sua percepção sobre os níveis gerados por computador.**

As últimas três etapas do experimento não são destinadas a medir imersão, mas sim a medir similaridade entre os dois grupos de níveis. A etapa 7 é um teste de Turing para medir o número de participantes que são capazes de descobrir qual dos grupos foi gerado por computador. Na etapa 8, a resposta correta do teste de Turing é dada e, na etapa 9, o participante responde questões para descrever o quão similar os dois grupos são. Esse questionário final também pergunta ao jogador sobre a sua percepção de que os níveis do conjunto P tinham sido gerados por um computador.

Figura 22 – Distribuição de idade, gênero, experiência com jogos em geral, experiência em Angry Birds e nível de educação dos participantes.

Um total de 100 pessoas participaram do teste por meio da Internet até o momento da coleta de dados. A Figura 22 mostra a distribuição de idade, gênero, experiência com jogos, experiência em *Angry Birds* e nível de educação desses participantes. A maioria deles são homens entre 21 e 30 anos de idade que possuem o ensino superior completo. A experiência com jogos e a experiência em *Angry Birds* foram medidas em uma escala de 1 a 5, onde 1 significa “nenhuma experiência” e 5 significa “muita experiência”. Aproximadamente 60% dos participantes possuem bastante experiência com jogos (responderam 4 e 5) e a maioria deles possui uma experiência regular (responderam 3) com o jogo *Angry Birds*.

As respostas das perguntas do questionário de imersão também foram definidas em uma escala de 1 a 5, sendo que 1 significa “quase nada” e 5 significa “muito”. Dessa forma, o grau total de imersão $6 \leq imm \leq 30$ de um grupo de níveis é calculado somando as seis respostas do questionário. Os resultados obtidos para cada um dos dois grupos estão reportados na Tabela 1. A pontuação média de imersão dos dois grupos foi aproximadamente igual (17.57 e 17.27). Sendo que o grupo gerado pelo AG teve uma pontuação ligeiramente mais alta. Isso significa que o AG foi capaz de gerar níveis tão imersíveis quanto os níveis do primeiro mundo da versão original do jogo. As duas pontuações são valores intermediários (considerando os limites de imm), o que significa que os dois grupos proporcionaram um nível de imersão regular para os jogadores.

	Grupo P	Grupo R
$imm \mu$	17.57	17.27
$imm \sigma$	4.34	4.10

Tabela 1 – Média μ e desvio padrão σ das pontuações obtidas no experimento de imersão.

Depois de responder o questionário de imersão para ambos os grupos, o jogador é avisado que um deles foi gerado por computador e que ele deve descobrir qual foi. A Figura 23 mostra a quantidade de participantes que passaram e falharam no teste e a porcentagem de jogadores que falharam no teste considerando as suas experiências em jogos e em *Angry Birds*. O gráfico da esquerda mostra que a quantidade de jogadores que falharam e passaram no teste foi praticamente a mesma (aproximadamente 50), sendo que a quantidade de falhas foi ligeiramente maior.

Figura 23 – O impacto da experiência dos jogadores na acurácia para descobrir qual dos dois grupos foi gerado por computador.

O gráfico da direita mostra que houve erros em todas as classes de experiência com uma taxa de no mínimo 30%. Devido a pouca quantidade de participantes em determinadas classes, não se pode fazer uma comparação equilibrada entre as taxas de erros. Porém, considerando apenas os jogadores que possuem muita experiência em jogos (igual a 5), que são a maioria, nota-se que 60% deles falharam no teste. Segmentando a população por experiência em *Angry Birds*, a maioria passa a ter experiência regular (igual a 3). Nesse caso, aproximadamente 50% deles também falharam no teste. Esses resultados indicam que o AG produziu níveis com qualidade considerável, uma vez que pouco mais da metade dos jogadores, incluindo os experientes, não conseguiu distingui-los dos níveis do jogo original.

Na última etapa do experimento, os participantes tiveram que responder perguntas sobre a similaridade entre os grupos de níveis. Nesse momento, eles já sabiam qual dos grupos tinha sido gerado por computador. As pontuações do questionário de similaridade usam a mesma escala do questionário de imersão. No entanto, ele coleta duas medidas diferentes: similaridade $1 \leq sim \leq 5$ dos grupos e percepção $2 \leq per \leq 10$ dos participantes de que os níveis do conjunto P tinham sido gerados por um computador. A primeira medida é calculada usando apenas a questão 2 do questionário de similaridade. A segunda é calculada somando as pontuações obtidas nas questões 1 e 3 desse mesmo questionário. Os resultados obtidos para essas medidas estão reportados na Tabela 2.

A média de pontuação de similaridade e o grau médio de percepção são aproxima-

	Similaridade	Percepção
μ	2.9	5.51
σ	1.25	2.19

Tabela 2 – Média μ e desvio padrão σ das pontuações obtidas para similaridade e percepção no experimento de similaridade.

mente 3 e 5, respectivamente. Isso significa que os participantes perceberam que os grupos possuíam alguma similaridade, mas eles também reconheceram algumas diferença entre os grupos. Considerando essas diferenças e os resultados da Tabela 1, podemos concluir que o AG foi capaz de gerar níveis consideravelmente diferentes dos originais, mas com um nível equivalente de imersão. Dessa forma, o AG proposto pode ser considerado como uma boa ferramenta para gerar níveis para o primeiro mundo do *Angry Birds*.

CONCLUSÃO

Esse trabalho apresentou um algoritmo genético (AG) para geração procedural de níveis em jogos de quebra-cabeças baseados em física. O algoritmo foi aplicado em uma versão clone do jogo *Angry Birds*. Os algoritmos de geração de níveis neste tipo de jogo geralmente avaliam a factibilidade por meio de um agente inteligente em uma simulação do jogo. Nesse caso, a simulação é utilizada dentro do processo de cálculo de aptidão do algoritmo genético. Dessa forma, dados são coletados na simulação para definir os valores das variáveis da função de aptidão. O problema dessa abordagem é que tanto o motor de física quanto o agente podem conter componentes não determinísticos, o que pode gerar ruído no cálculo de aptidão. Esse tipo de problema foi pouco explorado dentro do campo de geração procedural de conteúdo. No entanto, nesse trabalho, foi proposta uma abordagem baseada em cache que possibilitou evoluções elitistas corretas e, além disso, melhorou o desempenho do AG.

A função de aptidão considera a factibilidade do nível e a estabilidade de seus elementos. A estabilidade foi medida pela velocidade total dos objetos durante o início da simulação e a factibilidade foi medida por um agente inteligente simples que escolhe porcos aleatórios para atirar. A cache foi aplicada nesse processo, assim, antes da avaliação de um nível, o AG primeiro procura por ele na cache. Se ele for encontrado, a aptidão armazenada é retornada, caso contrário, uma simulação é executada e uma nova instancia é adicionada na cache. As principais vantagens desse método são a redução na quantidade de simulações e a liberdade para desenvolver agentes inteligentes que usam estratégias estocásticas.

O AG utiliza esse sistema de cache para evoluir uma população de níveis que são representados por vetores de pilhas de blocos do jogo. Esse AG foi avaliado por três conjuntos de experimentos, sendo que o primeiro mede o impacto da cache no desempenho e acurácia do AG. Outros dois métodos tradicionais da literatura para lidar com ruído

foram utilizados para comparação: médias explícitas e médias implícitas. Os resultados mostraram que a abordagem de cache foi o único método capaz de encontrar soluções com aptidões ótimas em todas as execuções. Além disso, ele foi aproximadamente 3.5 e 2.15 vezes mais rápido que as versões mais rápidas dos métodos de médias explícitas e implícitas, respectivamente.

O segundo conjunto de experimentos analisou a expressividade do AG enquanto um gerador de níveis. Expressividade, nesse caso, significa o espaço de todos os níveis que ele consegue gerar. Para esta análise, foram definidas três métricas que representam as características dos níveis: frequência, linearidade e densidade. Essas métricas foram calculadas para 500 níveis que foram gerados com 5 configurações diferentes, sendo 100 níveis para cada configuração. Os valores obtidos mostraram que a frequência de porcos não é diretamente afetada pela variação das entradas do algoritmo. Além disso, o valor de frequência dos porcos indicou que a porcentagem de caixas nos níveis é menor que 20%. A frequência de pássaros mostrou que os níveis dificilmente terão apenas um pássaro para que o jogador destrua os porcos. Os resultados de densidade e linearidade confirmaram a hipótese de que níveis com mais blocos precisam de mais pássaros para serem finalizados. Os valores de densidade e linearidade também indicaram que o AG possui um viés na distribuição de pilhas nos níveis. Ele distribui os blocos em poucas pilhas, ao invés de criar mais pilhas com alturas menores. Esses resultados também mostraram que o aumento do número de blocos aumenta diretamente a densidade do nível e diminui levemente a sua linearidade.

O último conjunto de experimentos foi definido para avaliar aspectos de imersão dos níveis gerados, em comparação com níveis criados manualmente. Dessa forma, dois grupos de níveis foram definidos, sendo que o primeiro possui apenas níveis gerados pelo AG e outro possui níveis que fazem parte do primeiro mundo da versão original do *Angry Birds*. Esses dois grupos foram jogados por 100 participantes que responderam questionários sobre imersão e similaridade dos grupos. Os valores de imersão dos dois grupos foram muito próximos e os valores de similaridade indicaram que há uma diferença perceptível entre as características estruturais dos níveis de cada grupo. Os participantes não sabiam que um dos grupos tinha sido gerado por computador, assim, eles foram desafiados a adivinhar qual dos dois era esse grupo. O número de participantes que passaram no teste foi muito próximo ao número de participantes que falharam.

Os resultados dos experimentos indicaram que o AG foi capaz de gerar níveis diferentes dos que fazem parte do primeiro mundo da versão original do *Angry Birds*, mas que são tão imersíveis quanto eles. Porém, esses níveis possuem estruturas muito simples, quando comparados com outros que estão em mundos mais avançados do jogo original. Assim, um dos trabalhos futuros desse projeto consiste em melhorar a representação dos níveis, visando suportar estruturas mais complexas de blocos. Além disso, o agente

inteligente utilizado possui uma estratégia muito simples, que influencia muito na avaliação de factibilidade dos níveis. Definir uma estratégia melhor para o agente é outro trabalho futuro que pode melhorar muito a qualidade dos níveis gerados. Finalmente, tanto a estabilidade quanto a factibilidade estão sendo avaliadas de maneira conjunta em uma função polinomial. Separar esses dois aspectos em dois objetivos é outro trabalho futuro que pode melhorar muito a capacidade do AG em explorar o espaço de busca.

REFERÊNCIAS

- ALLEN, G. E. Naturalists and experimentalists: The genotype and the phenotype. **Studies in the History of Biology**, v. 3, p. 179 – 209, 1979 1979. Citado na página 39.
- AVALANCHE, S. Jogo Digital, **Just Cause**. 2006. <<http://www.justcause.com>>. Acessado em: 2014-01-30. Citado na página 32.
- BELL, I.; BRABEN, D. Jogo Digital, **Elite**. 1984. <<http://www.iancgbell.clara.net/elite>>. Acessado em: 2014-01-30. Citado na página 31.
- BENTLEY, P.; KUMAR, S. Three ways to grow designs: A comparison of evolved embryogenies for a design problem. In: **Genetic and Evolutionary Computation Conference**. [S.l.]: Morgan Kaufmann, 1999. p. 35–43. Citado na página 35.
- BEYER, H.-G. Mutate large, but inherit small! on the analysis of rescaled mutations in 1-lambda-es with noisy fitness data. In: **Proceedings of the 5th International Conference on Parallel Problem Solving from Nature**. London, UK, UK: Springer-Verlag, 1998. (PPSN V), p. 109–118. ISBN 3-540-65078-4. Citado na página 42.
- BIRD, J.; FELTWELL, T.; CIELNIAK, G. Real-time adaptive track generation in racing games. In: **13th International Conference on Intelligent Games and Simulation, GAME-ON 2012**. School of Computer Science, University of Lincoln, United Kingdom: [s.n.], 2012. p. 48–52. Citado na página 38.
- BRANKE, J. Creating robust solutions by means of evolutionary algorithms. **Parallel Problem Solving from Nature — PPSN V**, Springer Berlin Heidelberg, v. 1498, p. 119–128, 1998. Citado na página 43.
- BRANKE, J.; SCHMIDT, C. Selection in the presence of noise. **Genetic and Evolutionary Computation — GECCO 2003**, Springer Berlin Heidelberg, v. 2723, p. 766–777, 2003. Citado na página 43.
- BROWNE, C.; MAIRE, F. Evolutionary game design. **Computational Intelligence and AI in Games, IEEE Transactions on**, v. 2, n. 1, p. 1–16, March 2010. Citado na página 33.
- CACHIA, W.; LIAPIS, A.; YANNAKAKIS, G. N. Multi-level evolution of shooter levels. In: **Proceedings of the AAAI Artificial Intelligence for Interactive Digital Entertainment Conference**. [S.l.: s.n.], 2015. Citado na página 36.
- CARDAMONE, L.; LOIACONO, D.; LANZI, P. L. Interactive evolution for the procedural generation of tracks in a high-end racing game. In: **Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation**. New York, NY, USA: ACM, 2011. (GECCO '11), p. 395–402. ISBN 978-1-4503-0557-0. Citado 4 vezes nas páginas 15, 36, 37 e 38.

- CARDONA, A. B.; TOGELIUS, J.; NELSON, M. J. Competitive coevolution in Ms. Pac-Man. In: **Evolutionary Computation (CEC), 2013 IEEE Congress on**. [S.l.: s.n.], 2013. p. 1403–1410. Citado na página 28.
- COOK, M.; COLTON, S. Multi-faceted evolution of simple arcade games. In: **Computational Intelligence and Games (CIG), 2011 IEEE Conference on**. [S.l.: s.n.], 2011. p. 289–296. Citado 3 vezes nas páginas 15, 36 e 37.
- DAHLISKOG, S.; TOGELIUS, J. Patterns and procedural content generation: Revisiting Mario in world 1 level 1. In: **ACM International Conference Proceeding Series**. [S.l.: s.n.], 2012. Citado 2 vezes nas páginas 15 e 34.
- DAVIS, L. D.; MITCHELL, M. Handbook of Genetic Algorithms. **Van Nostrand Reinhold**, 1991. Disponível em: <<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.87.3586>>. Citado na página 41.
- EIBEN, A. E.; SMITH, J. E. **Introduction to Evolutionary Computing**. [S.l.]: SpringerVerlag, 2003. ISBN 3540401849. Citado 2 vezes nas páginas 39 e 40.
- ESA. **ESA, 2015 Essential facts about the computer and video game industry: Sales, demographics and usage data**. 2015. <<http://www.theesa.com/>>. Acessado: 15-06-2015. Citado na página 27.
- FERREIRA, L.; PEREIRA, L.; TOLEDO, C. A multi-population genetic algorithm for procedural generation of levels for platform games. In: **Proceedings of the 2014 Conference Companion on Genetic and Evolutionary Computation Companion**. New York, NY, USA: ACM, 2014. (GECCO Comp '14), p. 45–46. ISBN 978-1-4503-2881-4. Citado na página 33.
- FERREIRA, L.; TOLEDO, C. Generating levels for physics-based puzzle games with estimation of distribution algorithms. In: **Proceedings of the 11th International Conference on Advances in Computer Entertainment**. [S.l.: s.n.], 2014. (ACE'14). Citado na página 46.
- _____. A search-based approach for generating angry birds levels. In: **Proceedings of the 9th IEEE International Conference on Computational Intelligence in Games**. [S.l.: s.n.], 2014. (CIG'14). Citado na página 46.
- FIRAFIX, G. Jogo Digital, **Civilization IV**. 2005. <<http://www.2kgames.com/civ4/>>. Acessado em: 2014-01-30. Citado na página 32.
- FITZPATRICK, J.; GREFFENSTETTE, J. Genetic algorithms in noisy environments. **Machine Learning**, Kluwer Academic Publishers, v. 3, n. 2-3, p. 101–120, 1988. ISSN 0885-6125. Citado na página 42.
- GEARBOX, S. Jogo Digital, **Borderlands**. 2009. <<http://www.borderlandsthegame.com>>. Acessado em: 2014-01-30. Citado na página 32.
- GOLDBERG, D. E. **Genetic Algorithms in Search, Optimization and Machine Learning**. 1st. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN 0201157675. Citado 2 vezes nas páginas 40 e 41.

HASTINGS, E.; GUHA, R.; STANLEY, K. Automatic content generation in the galactic arms race video game. **Computational Intelligence and AI in Games, IEEE Transactions on**, v. 1, n. 4, p. 245–263, Dec 2009. ISSN 1943-068X. Citado na página 28.

HENDRIKX, M.; MEIJER, S.; VELDEN, J. V. D.; IOSUP, A. Procedural content generation for games: A survey. **ACM Trans. Multimedia Comput. Commun. Appl.**, ACM, New York, NY, USA, v. 9, n. 1, p. 1:1–1:22, 2013. ISSN 1551-6857. Citado na página 27.

IOSUP, A. POGGI: generating puzzle instances for online games on grid infrastructures. **Concurrency and Computation: Practice and Experience**, John Wiley and Sons Ltd., v. 23, n. 2, p. 158–171, feb 2011. ISSN 15320626. Citado na página 27.

JENNETT, C.; COX, A. L.; CAIRNS, P.; DHOPAREE, S.; EPPS, A.; TIJS, T.; WALTON, A. Measuring and defining the experience of immersion in games. **Int. J. Hum.-Comput. Stud.**, Academic Press, Inc., Duluth, MN, USA, v. 66, n. 9, p. 641–661, sep 2008. ISSN 1071-5819. Citado na página 63.

JIN, Y.; BRANKE, J. Evolutionary optimization in uncertain environments-a survey. **Evolutionary Computation, IEEE Transactions on**, v. 9, n. 3, p. 303–317, June 2005. ISSN 1089-778X. Citado na página 42.

JONG, K. D. Evolutionary computation: a unified approach. In: **GECCO '08: Proceedings of the 2008 GECCO conference companion on Genetic and evolutionary computation**. ACM, 2008. p. 2245–2258. ISBN 978-1-60558-131-6. Disponível em: <<http://dx.doi.org/10.1145/1388969.1389054>>. Citado 2 vezes nas páginas 39 e 40.

KHALED, R.; NELSON, M. J.; BARR, P. Design metaphors for procedural content generation in games. In: **Proceedings of the SIGCHI Conference on Human Factors in Computing Systems**. New York, NY, USA: ACM, 2013. (CHI '13), p. 1509–1518. ISBN 978-1-4503-1899-0. Disponível em: <<http://doi.acm.org/10.1145/2470654.2466201>>. Citado na página 31.

KIMBROUGH, S. O.; KOEHLER, G. J.; LU, M.; WOOD, D. H. On a feasible-infeasible two-population (fi-2pop) genetic algorithm for constrained optimization: Distance tracing and no free lunch. **European Journal of Operational Research**, v. 190, n. 2, p. 310–327, 2008. Citado na página 36.

LIAPIS, A.; YANNAKAKIS, G. N.; TOGELIUS, J. Optimizing visual properties of game content through neuroevolution. In: **Proceedings of the 7th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2011**. [S.l.: s.n.], 2011. p. 152–157. Citado na página 28.

_____. Enhancements to constrained novelty search: Two-population novelty search for generating game content. In: **Proceedings of Genetic and Evolutionary Computation Conference**. [S.l.: s.n.], 2013. p. 343–350. Citado 2 vezes nas páginas 28 e 36.

LIPOWSKI, A.; LIPOWSKA, D. Roulette-wheel selection via stochastic acceptance. **CoRR**, abs/1109.3627, 2011. Disponível em: <<http://dblp.uni-trier.de/db/journals/corr/corr1109.html#abs-1109-3627>>. Citado na página 40.

LOIACONO, D.; CARDAMONE, L.; LANZI, P. Automatic track generation for high-end racing games using evolutionary computation. **Computational Intelligence and AI in Games, IEEE Transactions on**, v. 3, n. 3, p. 245–259, Sept 2011. Citado na página 36.

MAHLMANN, T.; TOGELIUS, J.; YANNAKAKIS, G. Towards procedural strategy game generation: Evolving complementary unit types. **Applications of Evolutionary Computation**, Springer Berlin Heidelberg, v. 6624, p. 93–102, 2011. Citado 2 vezes nas páginas 29 e 39.

MAHLMANN, T.; TOGELIUS, J.; YANNAKAKIS, G. N. Evolving card sets towards balancing dominion. In: **Evolutionary Computation (CEC), 2012 IEEE Congress on**. [S.l.: s.n.], 2012. p. 1–8. Citado na página 28.

MARKON, S.; ARNOLD, D.; BACK, T.; BEIELSTEIN, T.; BEYER, H.-G. Thresholding-a selection operator for noisy es. In: **Evolutionary Computation, 2001. Proceedings of the 2001 Congress on**. [S.l.: s.n.], 2001. v. 1, p. 465–472. Citado na página 43.

MILLER, B. L.; GOLDBERG, D. E. Genetic algorithms, selection schemes, and the varying effects of noise. **Evolutionary Computation**, v. 4, p. 113–131, 1996. Citado na página 42.

MILLER, B. L.; MILLER, B. L.; GOLDBERG, D. E.; GOLDBERG, D. E. Genetic algorithms, tournament selection, and the effects of noise. **Complex Systems**, v. 9, p. 193–212, 1995. Citado na página 40.

MOJANG. Jogo Digital, **Minecraft**. 2009. <<https://minecraft.net>>. Acessado em: 2014-01-30. Citado na página 32.

MOURATO, F.; SANTOS, M. P. dos; BIRRA, F. Automatic level generation for platform videogames using genetic algorithms. In: **Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology**. New York, NY, USA: ACM, 2011. (ACE '11), p. 8:1–8:8. ISBN 978-1-4503-0827-4. Disponível em: <<http://doi.acm.org/10.1145/2071423.2071433>>. Citado na página 33.

OTHMAN, N.; DECRAENE, J.; CAI, W.; HU, N.; LOW, M.; GOUAILLARD, A. Simulation-based optimization of starcraft tactical ai through evolutionary computation. In: **Computational Intelligence and Games (CIG), 2012 IEEE Conference on**. [S.l.: s.n.], 2012. p. 394–401. Citado na página 28.

PIETRO, A. D.; WHILE, L.; BARONE, L. Applying evolutionary algorithms to problems with noisy, time-consuming fitness functions. In: **Evolutionary Computation, 2004. CEC2004. Congress on**. [S.l.: s.n.], 2004. v. 2, p. 1254–1261 Vol.2. Citado 3 vezes nas páginas 29, 42 e 52.

RAFFE, W. L.; ZAMBETTA, F.; LI, X. Evolving patch-based terrains for use in video games. In: **Genetic and Evolutionary Computation Conference, GECCO'11**. [S.l.: s.n.], 2011. p. 363–370. Citado na página 28.

RATTRAY, M.; SHAPIRO, J. Noisy fitness evaluation in genetic algorithms and the dynamics of learning. In: **Foundations of Genetic Algorithms**. [S.l.: s.n.], 1997. Citado 2 vezes nas páginas 42 e 55.

RUDOLPH, G. A partial order approach to noisy fitness functions. In: **Evolutionary Computation, 2001. Proceedings of the 2001 Congress on**. [S.l.: s.n.], 2001. v. 1, p. 318–325. Citado na página 43.

SANO, Y.; KITA, H. Optimization of noisy fitness functions by means of genetic algorithms using history of search with test of estimation. In: **Evolutionary Computation, 2002. CEC '02. Proceedings of the 2002 Congress on**. [S.l.: s.n.], 2002. v. 1, p. 360–365. Citado 4 vezes nas páginas 29, 41, 43 e 55.

SHAKER, N.; SHAKER, M.; TOGELIUS, J. Evolving playable content for cut the rope through a simulation-based approach. In: SUKTHANKAR, G.; HORSWILL, I. (Ed.). **AIIDE**. AAAI, 2013. Disponível em: <<http://dblp.uni-trier.de/db/conf/aiide/aiide2013.html#ShakerST13>>. Citado 3 vezes nas páginas 28, 38 e 39.

_____. Ropossum: An authoring tool for designing, optimizing and solving cut the rope levels. In: SUKTHANKAR, G.; HORSWILL, I. (Ed.). **AIIDE**. AAAI, 2013. Disponível em: <<http://dblp.uni-trier.de/db/conf/aiide/aiide2013.html#ShakerST13a>>. Citado 2 vezes nas páginas 28 e 38.

SHAKER, N.; TOGELIUS, J.; NELSON, M. J. (Ed.). **Procedural Content Generation in Games: a Textbook and an Overview of Current Research**. [S.l.]: pcgbook.com, 2013. Citado 7 vezes nas páginas 27, 28, 29, 34, 35, 38 e 59.

SHAKER, N.; YANNAKAKIS, G. N.; TOGELIUS, J. Towards automatic personalized content generation for platform games. In: YOUNGBLOOD, G. M.; BULITKO, V. (Ed.). **Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment**. [S.l.]: The AAAI Press, 2010. Citado na página 33.

SMITH, G.; GAN, E.; OTHENIN-GIRARD, A.; WHITEHEAD, J. Pcg-based game design: enabling new play experiences through procedural content generation. In: **Proceedings of the 2nd International Workshop on Procedural Content Generation in Games**. New York, NY, USA: ACM, 2011. (PCGames '11), p. 7:1–7:4. ISBN 978-1-4503-0872-4. Citado 2 vezes nas páginas 27 e 32.

SMITH, G.; WHITEHEAD, J. Analyzing the expressive range of a level generator. In: **Proceedings of the 2010 Workshop on Procedural Content Generation in Games**. [S.l.]: ACM, 2010. p. 4:1–4:7. Citado 2 vezes nas páginas 37 e 59.

SMITH, G.; WHITEHEAD, J.; MATEAS, M. Tanagra: A mixed-initiative level design tool. In: **Proceedings of the Fifth International Conference on the Foundations of Digital Games**. New York, NY, USA: ACM, 2010. (FDG '10), p. 209–216. Citado na página 32.

TANOOKA, K.; TAMAKI, H.; ABE, S.; KITAMURA, S. A continuous age model of genetic algorithms applicable to optimization problems with uncertainties. In: **Systems, Man, and Cybernetics, 1999. IEEE SMC '99 Conference Proceedings**. [S.l.: s.n.], 1999. v. 1, p. 637–642. Citado na página 43.

TOGELIUS, J.; KASTBJERG, E.; SCHEDL, D.; YANNAKAKIS, G. N. What is procedural content generation? Mario on the borderline. In: **ACM International Conference Proceeding Series**. IT University of Copenhagen, Rued Langgaards Vej 7, Copenhagen, Denmark: [s.n.], 2011. Citado na página 27.

TOGELIUS, J.; NARDI, R. D.; LUCAS, S. M. Making racing fun through player modeling and track evolution. In: **Proceedings of the SAB'06 Workshop on Adaptive Approaches for Optimizing Player Satisfaction in Computer and Physical Games**. [S.l.: s.n.], 2006. Citado 2 vezes nas páginas 29 e 39.

_____. Towards automatic personalised content creation in racing games. In: **Proceedings of the IEEE Symposium on Computational Intelligence and Games**. [S.l.: s.n.], 2007. *Togelius is working at IDSIA on SNF grant 21-113364 to J. Schmidhuber*. Citado 3 vezes nas páginas 15, 35 e 38.

TOGELIUS, J.; PREUSS, M.; BEUME, N.; WESSING, S.; HAGELBACK, J.; YANNAKAKIS, G. Multiobjective exploration of the starcraft map space. In: **Computational Intelligence and Games (CIG), 2010 IEEE Symposium on**. [S.l.: s.n.], 2010. p. 265–272. Citado na página 28.

TOGELIUS, J.; YANNAKAKIS, G.; STANLEY, K.; BROWNE, C. Search-based procedural content generation: A taxonomy and survey. **Computational Intelligence and AI in Games, IEEE Transactions on**, v. 3, n. 3, p. 172–186, Sept 2011. Citado 7 vezes nas páginas 28, 33, 34, 35, 36, 37 e 38.

_____. Controllable procedural map generation via multiobjective evolution. **Genetic Programming and Evolvable Machines**, v. 14, n. 2, 2013. ISSN 1389-2576. Citado na página 33.

TOY, M. Jogo Digital, **Rogue**. 1983. <<http://www.wichman.org/roguehistory.html>>. Acessado em: 2014-01-30. Citado na página 31.

VALTCHANOV, V.; BROWN, J. A. Evolving dungeon crawler levels with relative placement. In: **ACM International Conference Proceeding Series**. School of Computer Science, University of Guelph, Guelph, ON N1G 2W1, Canada: ACM Press, 2012. p. 27–35. ISBN 9781450310840. Citado na página 28.

WANG, J.; ROGERS, P.; PARKER, L.; BROOKS, D.; STILMAN, M. Robot jenga: Autonomous and strategic block extraction. In: **Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on**. [S.l.: s.n.], 2009. p. 5248–5253. Citado na página 50.

YANNAKAKIS, G.; TOGELIUS, J. A panorama of artificial and computational intelligence in games. **Computational Intelligence and AI in Games, IEEE Transactions on**, PP, n. 99, p. 1–1, 2014. Citado na página 33.

YANNAKAKIS, G. N.; TOGELIUS, J. Experience-Driven Procedural Content Generation. v. 2, n. 3, p. 147–161, 2011. Citado na página 36.

YU, D.; HULL, A.; SUHRKE, E. Jogo Digital, **Spelunky**. 2013. <<http://www.spelunkyworld.com>>. Acessado em: 2014-01-30. Citado na página 32.

ADAPTAÇÕES DOS NÍVEIS COLETADOS DO JOGO ORIGINAL

Esse apêndice mostra todas as adaptações que foram realizadas nos níveis coletados do primeiro mundo da versão original do *Angry Birds*.

QUESTIONÁRIOS UTILIZADOS NO EXPERIMENTO DE IMERSÃO

Profile Questionnaire

1. **Age**

9 or less 10-20 21-30 31-40 41-50 51-60 60 or more

2. **Gender**

Male Female

3. **Experience as a gamer**

Not at All 1 2 3 4 5 A lot

4. **Experience in Angry Birds**

Not at All 1 2 3 4 5 A lot

5. **Education**

Primary Secondary Bachelor Master Doctor

Immersion Questionnaire

Please answer the following questions by selecting the relevant number. In particular, remember that these questions are asking you about how you felt at the end of the game.

1. **To what extent did these levels hold your attention?**

Not at All 1 2 3 4 5 A lot

2. **To what extent did you lose track of time?**

Not at All 1 2 3 4 5 A lot

3. To what extent did you find these levels challenging?

Not at All 1 2 3 4 5 A lot

4. Were there any times during these levels in which you just wanted to give up?

Not at All 1 2 3 4 5 A lot

5. How much would you say you enjoyed playing these levels?

Not at All 1 2 3 4 5 A lot

6. Would you like to play these levels again?

Definitely not 1 2 3 4 5 Definitely yes

Similarity Questionnaire

Please answer the following questions by selecting the relevant number. In particular, remember that these questions are asking you about how you felt at the end of the game.

1. To what extent do you think the levels from the Group B were generated by the computer?

Not at All 1 2 3 4 5 A lot

2. To what extent the levels from Group B were similar to those from Group A?

Not at All 1 2 3 4 5 A lot

3. To what extent did you percept that the levels from Group B were generated by the computer?

Not at All 1 2 3 4 5 A lot