Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Sistemas de Informação

Estudo sobre geração procedural de conteúdo em jogos dos subgêneros Roguelike e Metroidvania.

Lucas de Rezende da Silva

JUIZ DE FORA
MARÇO, 2021

Estudo sobre geração procedural de conteúdo em jogos dos subgêneros Roguelike e Metroidvania.

Lucas de Rezende da Silva

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Sistemas de Informação

Orientador: Marcelo Caniato Renhe

JUIZ DE FORA

MARÇO, 2021

Estudo sobre geração procedural de conteúdo em jogos dos subgêneros Roguelike e Metroidvania.

Lucas de Rezende da Silva

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM SISTEMAS DE INFORMAÇÃO.

Aprovada por:

Marcelo Caniato Renhe
Doutor em Engenharia de Sistemas e Computação

Igor de Oliveira Knop
Doutor em Modelagem Computacional

Luiz Maurílio da Silva Maciel
Doutor em Engenharia de Sistemas e Computação

JUIZ DE FORA
12 DE MARÇO, 2021

Resumo

Jogos eletrônicos atualmente são ótimos exemplos do uso de soluções geradas proceduralmente, mais especificamente aqueles dos gêneros ação e aventura e RPG. As abordagens de geração procedural de conteúdo, ou GPC, não se limitam a jogos, tendo, contudo, um grande impacto e relevância na área. A possibilidade de poupar tempo de desenvolvimento e aumentar o fator rejogabilidade, gerando cenários mais interessantes e desafiadores, é algo muito atrativo para o desenvolvimento de jogos. O foco deste trabalho são os jogos *roguelike* e *metroidvania*, subgêneros de jogos de ação, aventura e RPG. Cada um destes sub-gêneros contém mecânicas que favorecem a criação de conteúdo procedural. A adequação do conteúdo criado, dessa forma, enfrenta alguns desafios relacionados às características do domínio em questão. Neste contexto, este trabalho tem por objetivo estudar e analisar algumas das principais técnicas de geração procedural de conteúdo em uso na atualidade. Serão apresentados resultados comparativos do uso prático de algumas técnicas selecionadas.

Palavras-chave: geração procedural de conteúdo, jogos roguelike, jogos metroidvania, geração de masmorras.

Abstract

Electronic games are currently great examples of using procedurally generated solutions, more specifically those of the adventure and RPG genres. Procedural content generation approaches, or PCG, are not limited to games, but have a great impact and relevance in this area. The possibility of saving development time and increasing replay factor, producing more interesting and challenging scenarios, is something very attractive for game development. This work focuses on roguelike and metroidvania games, subgenres of action and adventure and RPG. Each of these subgenres contains mechanics that favor the procedural content creation. This way, the adequacy of the created content faces some challenges related to the characteristics of the selected domain. In this context, this work aims to study and analyze some of the main techniques of procedural content generation in use today. Comparative results of the practical use of some selected techniques will be presented.

Keywords: Procedural content generation, dungeon generation, roguelike games, metroidvania games.

Conteúdo

Lista de Figuras

Lista de Abreviações e Termos

DCC	Departamento de Ciência da Computação.
UFJF	Universidade Federal de Juiz de Fora.
RPG	*Role-Playing Game*.
GPC	Geração procedural de conteúdo
Tile	Bloco ou polígono que constitui a menor parte na montagem de um plano.
Leiaute	Distribuição e arranjo dos elementos gráficos num determinado espaço.
Hub	Ambiente principal ou central de onde vários outros se ramificam
Bot	É uma aplicação de software concebido para simular ações humanas

1 Introdução

Um elemento comumente usado em jogos digitais é o conceito de *dungeon*, que pode ser traduzido como "calabouço" ou "masmorra". Um *dungeon* é caracterizado por uma vasta área complexa, composta de salas ou cavernas interligadas por corredores ou acessos, que é explorada por aventureiros, sejam eles solitários ou em grupo. Estes elementos encontrados em diversos tipos de jogos podem ser encontrados em jogos *roguelike* e *metroidvania*, que são subgêneros do RPG e do gênero de ação e aventura.

Jogos *roguelike* são em sua essência jogos de exploração livre. O jogador avança pelas áreas geradas em busca de tesouros e pontos. Cada sessão de jogo pode ser única e desafia o jogador a avançar o máximo possível, acumulando recursos e enfrentando desafios cada vez maiores. A morte permanente e a aleatoriedade completa das salas a cada partida são características muito presentes.

Já os jogos do subgênero conhecido como *metroidvania* têm como características principais um maior controle na definição dos caminhos que o jogador pode percorrer e um forte foco na exploração contínua do ambiente. Com pontos de início e fim bem definidos, o caminho entre os dois pontos pode ser algo fixado pelo desenvolvedor ou completamente gerado. Esse subgênero usa itens ou habilidades de movimentação adquiridas em pontos específicos para controlar o acesso do jogador a certas áreas. Os níveis são, de forma geral, grandes áreas interligadas em que o jogador é livre para explorar da forma que bem entender. O jogo permite e encoraja o retorno a salas e áreas já exploradas, uma vez que se tenha adquirido os pré-requisitos ou novas habilidades para acessar novas áreas ou segmentos novos de áreas antigas.

A criação manual de recursos de jogos, como itens, níveis ou *puzzles*, consome muito tempo. Com isso, o volume de conteúdo produzido para um jogo tende a ser limitado, dependendo dos prazos e da equipe disponível para o seu desenvolvimento. A falta de conteúdo pode afetar o fator rejogabilidade de um jogo, deixando-o pouco atrativo e, por vezes, muito repetitivo. Por conta disso, técnicas de geração procedural de conteúdo (GPC) são frequentemente utilizadas para contornar essas restrições. Qualquer conteúdo

presente gerado de forma algorítmica pode ser considerado GPC. Essas técnicas propiciam a equipes menores de desenvolvedores produzir jogos com ambientes mais ricos e variados, mesmo com recursos limitados.

Além das questões levantadas acima, jogos consomem cada vez mais espaço em disco. A possibilidade de gerar cenários, missões e outros elementos de jogo proceduralmente reduz esse consumo. Por outro lado, gerar estes recursos durante a execução pode exigir uma maior demanda por processamento. No entanto, os benefícios citados muitas vezes compensam este gasto.

1.1 Motivação e justificativa

Grande parte do apelo dos subgêneros citados no início deste capítulo está diretamente ligado à interação do jogador com o mapa a ser explorado. Jogos desses subgêneros utilizam ou têm potencial de utilizar várias das técnicas de geração procedural para construção desses mapas. Porém, algumas destas técnicas são bem rígidas e poucos flexíveis (SMITH; PADGET; VIDLER, 2018). Com o passar do tempo, para atender a características específicas e mecânicas de alguns jogos, as técnicas de geração evoluíram e diversas abordagens nasceram. Ainda assim, muitos dos algoritmos usados seguem os mesmos princípios, porém com restrições e melhorias focadas na solução de problemas pontuais.

Embora muito comuns e disseminados pela *Web* atualmente, muitos dos métodos de GPC não estão formalmente descritos, e por isso é difícil analisá-los e compará-los. Nos trabalhos usados como referências para este trabalho, é possível encontrar diversas soluções alternativas que nasceram para preencher lacunas específicas na forma de gerar níveis eficientes. No caso de jogos *roguelike*, um alto grau de aleatoriedade costuma ser suficiente, enquanto em jogos do estilo *metroidvania* os níveis devem ser gerados dentro de algumas restrições, uma vez que a livre movimentação do jogador pelo cenário depende dos pré-requisitos para transpor certos obstáculos.

Uma análise aprofundada destes métodos pode oferecer um melhor panorama das técnicas em uso atualmente na construção de mapas para estes subgêneros. Pretende-se que este trabalho possa ser usado como fonte de referência para estudos futuros sobre o tema, oferecendo dados úteis para a análise de métodos de GPC.

1.2 Objetivos

Este trabalho tem como objetivo principal estudar algoritmos para a geração procedural de níveis em jogos dos subgêneros *roguelike* e *metroidvania*, analisando algumas das atuais soluções usadas e formalizando-as, utilizando critérios objetivos para comparar os resultados de cada método e o seu grau de adequação às necessidades de cada subgênero. Como objetivos específicos, podemos citar:

1. Estudar e selecionar um conjunto de soluções propostas na literatura e estabelecer critérios objetivos e comuns a técnicas de geração procedural;

2. Implementar as soluções analisadas e compará-las utilizando os critérios estabelecidos;

3. Compilar os resultados obtidos e apresentar casos de uso e possíveis propriedades de cada método.

1.3 Trabalhos relacionados

Não é difícil encontrar diversos tipos de abordagens para GPC no cenário atual, não apenas para jogos destes subgêneros. A variedade é grande, e cada abordagem pode ser indicada para um tipo específico de situação. Seja para criação de ambientes, itens, personagens e até mesmo narrativas, existem diferentes técnicas e algoritmos que podem ser usados. Algumas dessas abordagens são discutidas em Baron (2017).

Restrições de exploração, quebra cabeças, busca por itens chaves entre outros, sugerem abordagens mais complexas que apenas explorar de forma aleatória cada mapa. Para geração do mapa e seus objetivos, entender o design permite definir os resultados gerados para uma variedade de conteúdo de jogos e abranger vários gêneros e subgêneros (SMITH; MATEAS, 2011).

Em Smith, Padget e Vidler (2018), é apresentada uma abordagem baseada em grafos que utiliza uma sequência de restrições declarativas para gerar tarefas ou missões dentro do cenário produzido. Já em Forsyth (2016), um algoritmo geral para geração de

dungeons é proposto, de forma que reduza a quantidade de caminhos sem saída e aumente a quantidade de caminhos alternativos para o objetivo.

Os benefícios do uso de GPC em jogos *roguelike* é abordado em Valtchanov e Brown (2012). Já Inferno, Clay e ELAarag (2017) demonstram uma implementação de um gerador de *dungeons* para jogos *roguelike*. Baseado em conjuntos fixos de salas, portas e conexões para criar um ambiente que segue uma hierarquia de relacionamento pai-filho. Com relação ao subgênero *metroidvania*, em Stalnaker (2020) é proposto um sistema para modelar mapas com estruturas de grafos direcionais e validar os percursos que finalizam o jogo.

Como dito anteriormente, o objetivo desta monografia é analisar algoritmos de GPC dentro dos subgêneros *roguelike* e *metroidvania* segundo um conjunto de métricas. Um exemplo de trabalho que analisa algoritmos no contexto de *roguelike* pode ser encontrado em Cruz (2014). O autor define algumas métricas e utiliza um *bot* para simular uma partida e avaliar o mapa gerado .

2 Fundamentação teórica

Neste capítulo serão apresentados os principais conceitos relacionados à geração procedural de conteúdo em jogos. Antes de entender de forma mais profunda as relações entre GPC e jogos digitais, é necessário definir o que é um jogo digital. De acordo com Lucchese e Ribeiro (2009), podemos definir um jogo digital como sendo a união de um universo e um conjunto de regras, criados e gerenciados por um *software*, onde um jogador pode interagir e tomar decisões buscando alcançar objetivos estabelecidos.

Jogos digitais podem ser divididos em diversos gêneros e subgêneros de jogos. Cada gênero tem suas particularidades e características, podendo ser classificados de várias formas. Como destacado em Janssen, Araujo e Baião (2019), uma das formas de se definir um gênero é pelo conjunto de características de *gameplay*, mecânicas e interações próprias que os tornam reconhecíveis. Como exemplos de gêneros, temos aventura, esportes, RPG, entre outros. Os subgêneros, por sua vez, são comumente definidos por características identificáveis que não representam por si só um novo gênero, mas sim algo que possa ser encaixado em diferentes gêneros existentes.

Como mencionado no Capítulo 1, este trabalho foca nos subgêneros *roguelike* e *metroidvania*. Assim, nas seções a seguir serão apresentadas as principais características destes subgêneros.

2.1 Definição de *roguelike*

O termo *roguelike* é inspirado pelo jogo Rogue, criado em 1980 e demonstrado na Figura 2.1a. O jogo se destacou e acabou inspirando outros futuros jogos como os da Figura 2.1 a seguirem suas características. O Rogue original é um jogo de aventura, em que um aventureiro ou um grupo explora complexos calabouços, com a disposição das salas e itens gerados aleatoriamente a cada execução (VALTCHANOV; BROWN, 2012). As características próprias trazidas pelo Rogue foram seguidas por outros jogos, cunhando assim o termo *roguelike*. Mapas gerados de forma procedural, disposição de itens e inimigos

espalhados de forma aleatória e morte permanente com perda do progresso atual foram algumas das características que o definiram na época, podemos ver tais características em jogos como The Binding of Isaac (Fig.2.1b) e Spelunk (Fig.2.1c) por exemplo. O conceito de morte permanente tornou-se menos recorrente com o passar do tempo, e a morte no jogo começou a ser punida com perda de itens ou penalidades, como uma forma de tornar o subgênero mais amigável para jogadores casuais ou novatos.

Características

- Mapas gerados de forma procedural;
- Disposição de itens e inimigos de forma aleatória;
- Morte permanente;
- Exploração do ambiente.

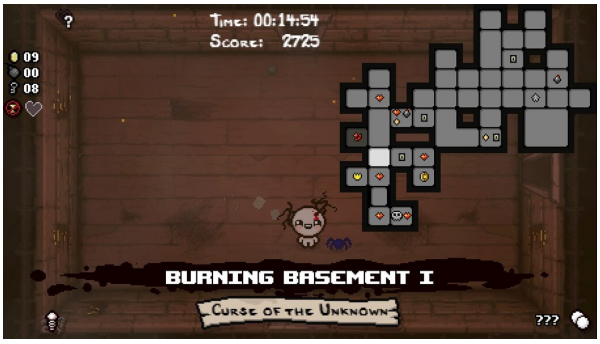
2.2 Definição de *metroidvania*

Metroidvania é um subgênero de jogos baseado na jogabilidade e design das séries de jogos *Metroid* e *Castlevania*, geralmente com uma câmera de rolagem lateral (*side-scrolling*), embora outras perspectivas também possam ser usadas. O subgênero conta com características de jogos de plataforma, aventura e RPGs.

Jogos desse subgênero são ambientados em um grande mapa dividido em zonas interconectadas que o jogador deve explorar(Fig.2.2). Embora o acesso a algumas partes do mapa seja inicialmente limitado por obstáculos, estes podem ser superados com a aquisição de ferramentas, armas ou habilidades obtidas durante o avanço no jogo. Essas melhorias adquiridas também podem ajudar o jogador a derrotar inimigos mais difíceis e localizar atalhos e áreas secretas, o que inclui refazer os próprios passos pelo mapa. Com isso, os jogos *metroidvania* integram o design dos níveis à história, incentivando a exploração e experimentação do ambiente.

Apesar deste subgênero não ter em sua concepção original o uso de GPC como podemos ver nas Figuras 2.2a e 2.2b onde o designer e cuidadosamente criado pela equipe

(a) Rogue (TOY, 1980)



(b) The Binding of Isaac (MCMILLEN, 2011)

(c) Spelunky (YU, 2008)

Figura 2.1: Exemplos de jogos *roguelike* que fazem uso de geração procedural de conteúdo, desde o lançamento do (a) Rogue, que cunhou o termo, até exemplos mais atuais, como (b) The Binding of Isaac e (c) Spelunky.

de desenvolvimento, com alguns cuidados e adaptações é possível criar mapas proceduralmente, mantendo as características principais do subgênero como por exemplo o jogo da Figura 2.2c.

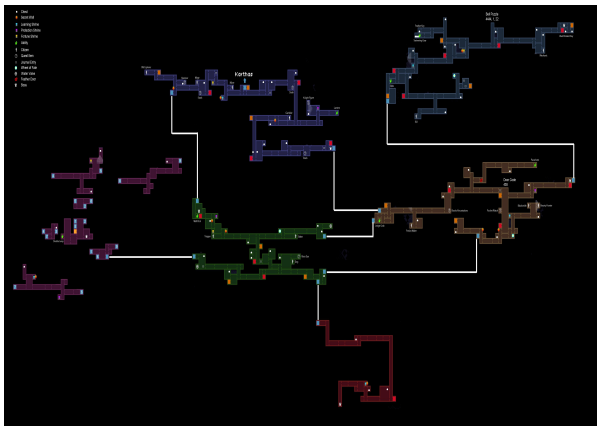
Características

- Grande mapa interconectado;
- Acesso limitado a alguns caminhos, desbloqueados por itens ou habilidades;
- Aquisição de ferramentas, armas ou habilidades no decorrer do jogo;
- Atalhos e áreas secretas;
- Incentivo à experimentação e exploração de ambiente, além do retorno a áreas já visitadas.

- Mapas gerados de forma procedural;
- Disposição de itens e inimigos de forma aleatória;
- Morte permanente;
- Exploração do ambiente.

(a) Castlevania: Symphony of the Night (KONAMI, 1997)

(b) Hollow Knight(TEAMCHERRY, 2017)



(c) Chasm (BITKID, 2018)

Figura 2.2: Exemplos de mapas de jogos *metroidvania*. Castlevania: Symphony of the Night (a) foi um clássico que ajudou a dar origem ao termo. Com planejamento e construção de mapa manual, Hollow Knight (b) tem mapas grandes e complexos, enquanto Chasm (c) tem seus mapas construídos de forma procedural.

2.3 Geração procedural de conteúdo (GPC)

Como pode ser visto em Duarte (2012), todo conteúdo presente em um jogo que pode ser criado por meio de algum algoritmo pode ser definido como Geração Procedural de Conteúdo (GPC). Essa definição simples é necessária para começar a compreender o que é GPC de fato. Todo conteúdo gerado possibilita inúmeras variações de cada estágio, incluindo cenário, inimigos ou itens com os quais o jogador poderá interagir durante a partida.

Ao contrário de uma abordagem tradicional em que cada detalhe da construção de mundo do jogo é decidido por uma equipe, na abordagem GPC os programadores preparam o algoritmo e um conjunto de parâmetros básicos, que são usados para gerar um resultado que poderá ou não ser modificado de forma manual ao final do processo. O processo de desenvolvimento é iterativo e deve ser pensado e lapidado continuamente, criando uma ferramenta cada vez mais refinada para os objetivos do projeto.

Para a construção específica de mapas, existem muitas possibilidades e variações, com diferentes formatos, tamanhos e caminhos. Avaliar as características presentes em cada variação permite definir quando o resultado obtido via GPC atende ou não às expectativas desejadas. De modo geral, é relativamente simples definir alguns pontos que fazem um mapa ser considerado ruim. Caminhos muito lineares, tamanho inadequado (muito grande ou muito pequeno), grau de desafio mal dimensionado e estruturas pouco variadas são exemplos de situações problemáticas. Por outro lado, não é trivial definir o que seria considerado um bom mapa. Na maioria dos casos, isso dependerá do objetivo desejado. Portanto, o que pode ser feito é definir algumas métricas, conforme será visto na Seção 3.4, e através do estudo e observação das mesmas verificar se o método utilizado atende às demandas necessárias.

2.3.1 Gerenciamento da saída

O conflito existente entre a criação manual de conteúdo e qualquer método de GPC se dá pela dificuldade de subverter a tendência desses métodos de gerar padrões simples e facilmente reconhecíveis. Para tentar equilibrar a aleatoriedade e a tendência de padrões,

ordem e caos são dois conceitos de fácil entendimento que se contrapõem, e se bem equilibrados, produzem um resultado satisfatório.(BARON, 2017)

O conceito de ordem abrange o potencial que a saída gerada tem de assumir padrões simples e genéricos. Quando este conceito se sobrepõe, temos uma situação onde o conteúdo final é monótono e pouco interessante. Em contrapartida, o caos representa a aleatoriedade e imprevisibilidade do resultado gerado. Caos em excesso cria uma saída distorcida e sem controle, onde o ambiente e conteúdos criados podem ser extremamente confusos ou inviáveis para uso prático.

A busca pelo equilíbrio entre esses conceitos é o que gera um conteúdo interessante e de qualidade. Para isso, os ajustes nos algoritmos devem buscar garantir que a saída tenha as características desejadas, equilibrando as parcelas de caos e ordem. O uso de restrições visa impedir apenas situações que se deseja evitar, sem introduzir um excesso de ordem que poderia levar a um resultado genérico e pouco expressivo. Um exemplo é o da Figura 2.3, em que são evitadas situações como a presença de árvores na água e cactos fora da areia. O processo de refinamento do algoritmo passa por inúmeras iterações, adicionando cada vez mais restrições para lapidar o resultado final.

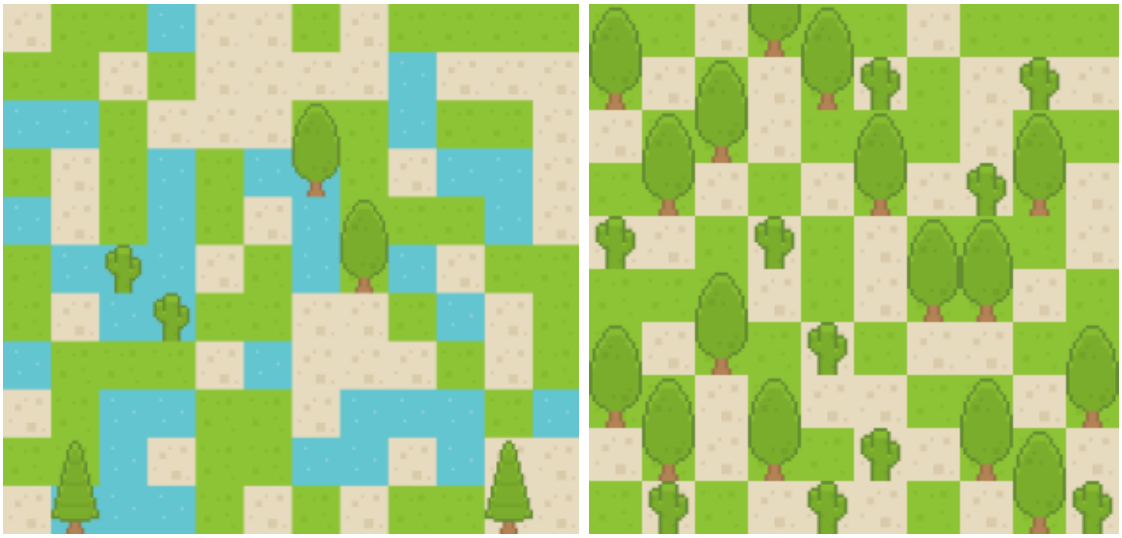


Figura 2.3: À esquerda, temos um exemplo onde a falta de restrições gera situações absurdas; à direita temos a demonstração de como algumas restrições podem organizar melhor o resultado (CAMPOS, 2016).

2.4 Classificação de métodos de GPC

Apesar das diferenças, todas as abordagens de GPC têm o mesmo propósito, que é o de criar conteúdo ou recurso de forma autônoma baseado apenas na combinação ou organização de elementos primitivos. Em Short e Adams (2017), é apresentada uma classificação de métodos de geração procedural em cinco categorias. Alguns métodos são baseados diretamente em sequências obtidas por geradores de números pseudo-aleatórios, enquanto outros são baseados na geração de sequências complexas a partir da combinação de elementos primitivos, como é o caso dos Sistemas Lindenmayer (*L-Systems*) e das Cadeias de Markov. Existem também métodos baseados em mapas de altura, que permitem mapear de forma simples o padrão topográfico de uma área, sendo muito usados para geração procedural de terrenos. Exemplos de métodos desta categoria são o algoritmo do ponto médio (ou diamante-quadrado), formação de falhas e a geração através do ruído de Perlin. Por fim, existem os métodos baseados em particionamento do espaço, que consistem em subdividir sucessivamente uma área inicial até que se atinga um determinado limiar, e os métodos baseados em preenchimento do espaço, que são o foco deste trabalho.

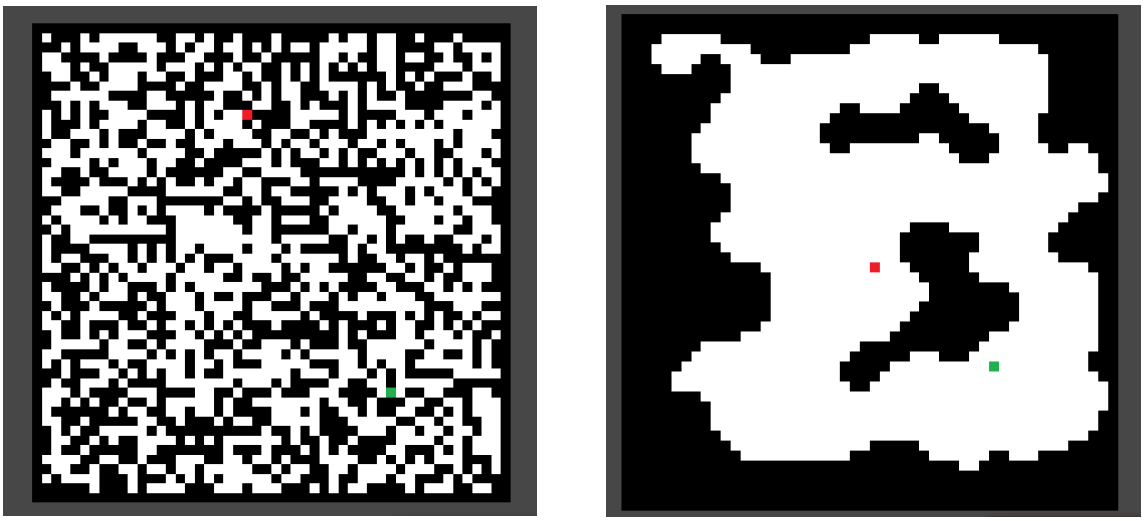
Os métodos de preenchimento do espaço são definidos por conjuntos de soluções que geram e organizam objetos e caminhos no espaço. A partir de um espaço inicialmente vazio, cada abordagem busca iterativamente preencher e organizar de forma estruturada esse espaço. Exemplos de técnicas que se enquadram nesta categoria são os autômatos celulares, o passeio aleatório, o assentamento (geração aleatória de salas que são reposicionadas por um sistema físico de colisão) e o Wang Tiles. Com exceção do assentamento, os outros três algoritmos são estudados e analisados neste trabalho, sendo apresentados nas seções a seguir.

2.4.1 Autômatos celulares

Os autômatos celulares são uma ampla categoria de sistemas que operam em um grafo de células discretas. Cada célula apresenta um conjunto de dois estados, sendo um o estado vivo e o outro o estado morto (Fig.2.4). O sistema inteiro opera sobre um conjunto de regras, descritas como regras de sobrevivência. Essas regras definem como o estado de cada célula se altera (SHORT; ADAMS, 2017).

As regras de sobrevivência determinam as condições para atualização do estado de uma célula. Sabendo quais são os estados das células vizinhas pode-se saber o próximo estado de uma célula. As regras podem ser definidas de acordo com uma função de probabilidades ou definidas com base nos estados das células vizinhas. Existem dois tipos de vizinhança mais comumente adotadas: a vizinhança de von Neumann, que considera apenas os 4 vizinhos imediatos nas direções ortogonais, e a vizinhança de Moore, que inclui também os outros 4 vizinhos nas diagonais.

Como entrada o sistema a largura e altura da matriz, uma semente fixa ou aleatória e a quantidade de vezes que as regras de sobrevivência deverão ser aplicadas. Um conjunto de células é então criado e seu estado e posição definidos de forma aleatória, como visto na Figura 2.4a. A cada iteração, aplica-se um conjunto de regras que pode alterar o estado de cada célula. Após algumas iterações, uma nova distribuição e organização das células no espaço é obtida, conforme ilustrado na Figura 2.4b.



(a) Antes da aplicação das regras de sobrevivência. **(b)** Após a aplicação das regras de sobrevivência.

Figura 2.4: Resultado da geração de um autômato celular (a) antes e (b) após aplicar as regras de sobrevivência. Os pontos verde e vermelho representam, respectivamente, o ponto de inicio e o objetivo a ser alcançado no mapa.

2.4.2 Wang Tiles

Tiles são blocos ou módulos que constituem a menor parte na montagem dos planos. O algoritmo Wang Tiles se baseia em uma matriz criada com largura e altura fornecidas pelo usuário (Fig. 2.5) e uma paleta de *tiles* previamente criada e também fornecida como entrada para o algoritmo. Cada *tile* deste algoritmo é uma peça de formato variado, com

um tipo de encaixe definido para cada aresta. O algoritmo se inicia sorteando um dos *tiles* da paleta e o colocando nas coordenadas $x = 0$ e $y = 0$ para dar inicio ao leiaute. Após definir a peça inicial, o algoritmo segue para a próxima posição. Em cada iteração, são sorteados, dentre o conjunto fornecido, *tiles* que consigam se conectar com aqueles já existentes nas adjacências da atual posição. O *tile* escolhido é então organizado de modo que cada aresta coincida com o encaixe das arestas adjacentes (SHORT; ADAMS, 2017).

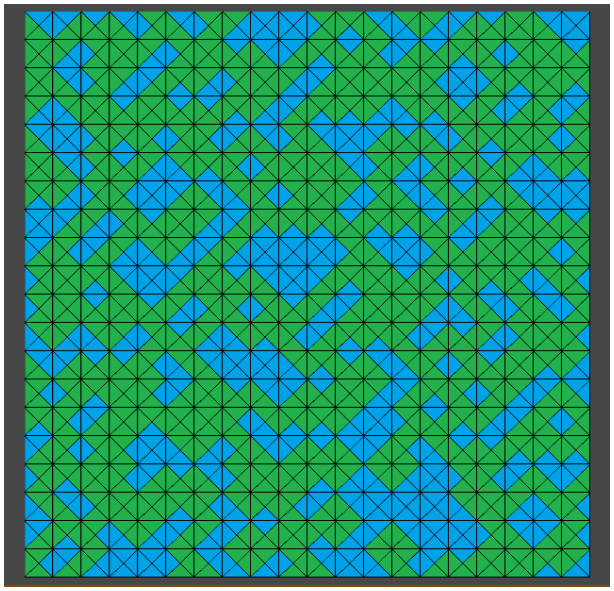


Figura 2.5: Paleta com *tiles* de quatro arestas e duas possibilidades de encaixe (cores), sendo o verde um espaço disponível e o azul um espaço indisponível.

2.4.3 Passeio aleatório

O algoritmo do passeio aleatório propõe aquilo que provavelmente é a solução mais simples das selecionadas. Uma matriz com largura e altura é definida e um ponto aleatório é escolhido nesta matriz. Uma quantidade de passos é então fornecida para que o algoritmo esteja pronto para começar. Partindo do ponto escolhido, o algoritmo avança passo a passo, escolhendo direções aleatórias para criar uma trajetória, até concluir o número de passos requisitados (Fig. 2.6). As tentativas de movimento para as paredes são ignoradas e passar por um *tile* já visitado ainda conta como um novo passo efetuado (SHORT; ADAMS, 2017).

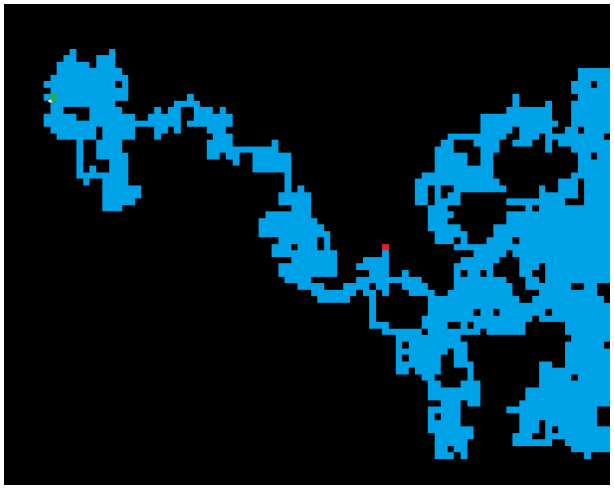


Figura 2.6: O passeio aleatório pode ser facilmente implementado e produzir bons resultados rapidamente.

3 Processo de desenvolvimento

O processo para geração procedural de níveis pode ser dividido em duas partes que podem ou não ocorrer juntas: a criação do leiauteee, que representa a distribuição ou arranjo dos elementos gráficos num determinado espaço, e o preenchimento desse leiaute com conteúdo, como itens, decoração, inimigos etc.. Este trabalho tem foco na primeira parte.

Utilizando os algoritmos selecionados, serão construídos mapas que possam se adequar às necessidades dos subgêneros aqui considerados. Cada um dos algoritmos produz como resultado um leiaute único, utilizando uma perspectiva *top-down*, onde o jogador tem uma visão de cima do ambiente, podendo se mover em qualquer direção.

3.1 *Roguelike* x *metroidvania*

Ambos os subgêneros têm como característica a exploração do ambiente. Assim, o processo de construção do leiaute destes ambientes deve encontrar e levar em conta características em comum que são relevantes em ambos os casos.

Similaridades

A estrutura e organização presentes nos mapas podem assumir várias formas. Cada leiaute pode seguir uma estrutura mais compacta, lembrando uma grande sala ou caverna, um conjunto de várias ramificações com muitas bifurcações ou também pode formar uma estrutura linear, em que os níveis se aproximam mais de grandes corredores.

A quantidade de caminhos possíveis de um ponto ao outro também é uma característica em comum, de forma que informações sobre os caminhos ajudam a entender ou planejar as formas como o jogador irá interagir com o leiaute.

Diferenças

Quanto ao caminho até o objetivo final do mapa, o subgênero *roguelike* preza por caminhos que não sejam tão curtos, com níveis gerados de tamanho mediano, que possibilitem uma

boa distribuição de itens para incentivar uma exploração mínima antes de avançar no mapa. Enquanto o subgênero *metroidvania* busca áreas mais extensas com caminhos mais longos, uma grande variedade de caminhos e a possibilidade de livre movimentação e exploração após liberar acesso integral das áreas no decorrer do jogo.

3.2 Implementação

Nesta seção, os detalhes de implementação de cada algoritmo selecionado são descritos, bem como as entradas utilizadas e como as saídas foram obtidas. O ambiente de desenvolvimento utilizado foi o motor de jogo proprietário Unity versão 2019.3.0f6 (64-bit) na linguagem de programação C, onde todos os algoritmos foram replicados e executados para gerar os dados e algumas das imagens usadas no trabalho.

3.2.1 Autômato celular

Como descrito na Seção 2.4.1, este algoritmo trabalha com um conjunto de células, estados, regras de sobrevivência e alguns parâmetros de entrada. A estrutura usada para representar o conjunto de células foi uma matriz de inteiros contendo um valor 0 ou 1, indicando, respectivamente, se uma célula assume um estado “morto” ou “vivo”.

A vizinhança adotada foi a vizinhança de Moore. A informação de vizinhança é usada na aplicação das regras de sobrevivência. Duas regras foram usadas para alterar o estado das células da matriz. A primeira regra define que toda célula com 4 ou mais vizinhos vivos é marcada como morta. Já a segunda regra diz que toda célula com menos de 4 vizinhos vivos é marcada como viva.

Como entrada, o algoritmo recebe o tamanho e a altura da matriz a ser criada, assim como uma porcentagem de ocupação inicial de células, que define a chance de uma célula ser marcada como viva na distribuição inicial. Além desses parâmetros, também é fornecido um número que indica quantas vezes as regras de sobrevivência serão aplicadas. Há ainda a possibilidade de se usar uma semente pré-definida ou habilitar a geração de uma semente aleatória.

Para o tamanho da matriz usada neste trabalho, os valores escolhidos foram 100

tiles de largura por 100 *tiles* de altura. A porcentagem de ocupação inicial usada foi fixada em 3 diferentes valores para realização dos testes, sendo esses valores 40%, 45% e 50% de preenchimento inicial. A semente utilizada foi obtida aleatoriamente em cada execução através da função `System.Random()` presente no ambiente de desenvolvimento. Quanto ao número de execuções das regras de sobrevivência, todas as execuções usaram o valor padrão 5.

Uma vez definidos os parâmetros de entrada, o algoritmo cria a matriz com o tamanho definido e demarca todas as bordas com células mortas para criar a parede externa do mapa. A parte interna da matriz é então populada aleatoriamente com a chance de preenchimento inicial escolhida. As regras de sobrevivência que modificam os estados das células são então aplicadas de acordo com o número de vezes escolhido. Como podemos ver na Figura 3.1, as saídas podem variar em formato, e sua estrutura geral pode ser levemente manipulada pelos parâmetros de entrada. A implementação do autômato pode ser visto no Algoritmo 3.1.

```
1  void AutomatoCelular(int[,] map, int width, int height, string
       seed, bool useRandomSeed, int randomFillPercent, int repetition
        = 5)
2  {
3      System.Random randNum;
4
5      if(useRandomSeed){
6          System.Random randNum = new System.Random();
7          seed = randNum.Next().ToString();
8      }
9      randomSeed = new System.Random(seed);
10
11     // prepara a matriz
12     map = new int[width, height];
13     RandomFillMap(map, randomSeed, randomFillPercent);
14
15     // aplica as regras de sobrevivencia
16     for (int i = 0; i < repetition; i++)
17         SmoothMap(map);
18 }
19
20 void RandomFillMap(int[,] map, System.Random seed, int
       randomFillPercent)
21 {
22     for (int x = 0; x < width; x++)
23         for (int y = 0; y < height; y++) {
24             // se for uma borda
25             if (x == 0 || x == width - 1 ||
```

```
                y == 0 || y == height - 1)
            {
                map[x, y] = 1;
            }
            else
            {
                map[x, y] = (pseudoRandom.Next(0, 100) <
                    randomFillPercent) ? 1 : 0;
            }
        }
}

void SmoothMap(int[,] map)
{
    for (int x = 0; x < width; x++)
        for (int y = 0; y < height; y++) {
            // verifica os vizinhos
            int neighbourWallTiles=GetSurroundingWallCount(x,y);
            // se tem mais de 4 vizinhos vivos, tile fica vivo
            if (neighbourWallTiles > 4)
                map[x, y] = 1;
            // se tem menos de 4 vizinhos vivos, tile morre
            else if (neighbourWallTiles < 4)
                map[x, y] = 0;
        }
}
```

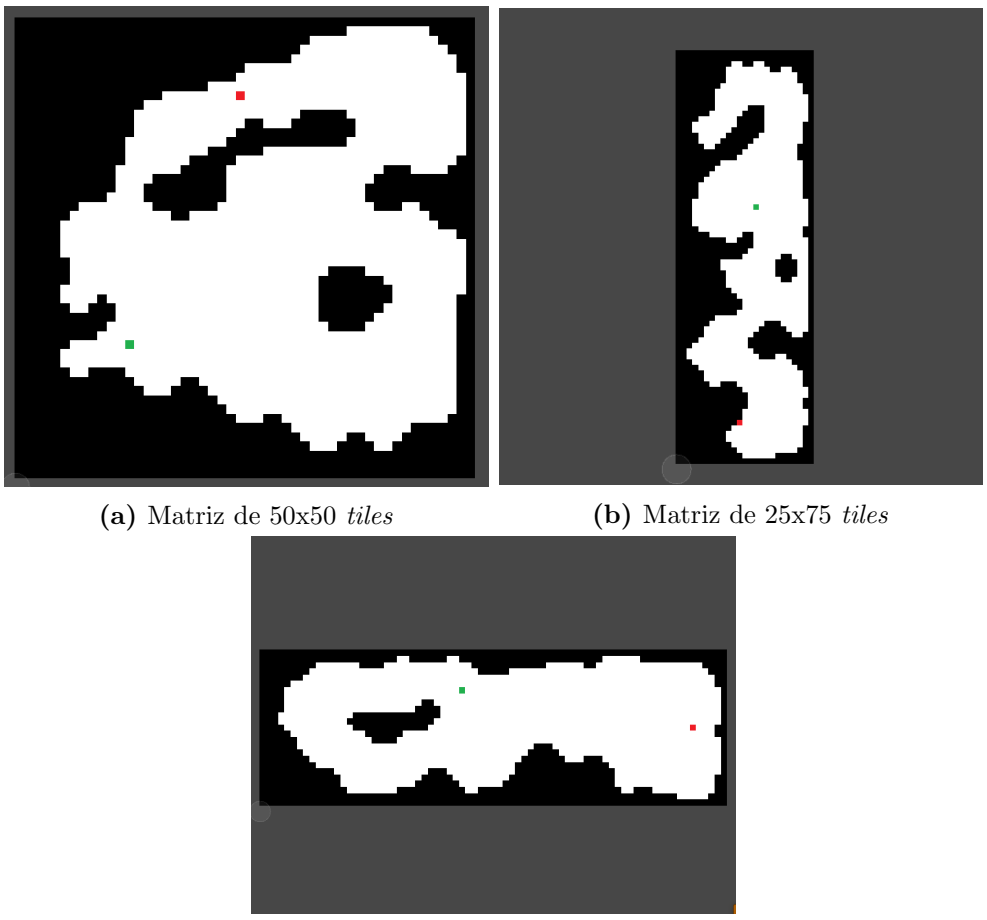
Algoritmo 3.1: Implementação do autômato celular .

Compatibilidade

Como pode ser visto na Figura 3.1, as áreas criadas são amplas e abertas (Fig.3.1a) e podem assumir formas diferentes de acordo com o espaço inicialmente disponível (Figs. 3.1b e 3.1c). Uma vez que o *roguelike* comumente se destaca por uma exploração ágil das áreas, o leiaute gerado se encaixa por contar com menos obstáculos para interferir nas distância do início ao fim, resultando em caminhos mais diretos para conclusão do nível.

3.2.2 Wang Tiles

Este algoritmo trabalha com a organização de *tiles* com encaixes pré-definidos. Para a estrutura principal, uma matriz de inteiros foi utilizada para guardar a informação sobre a localização e o tipo de cada *tile*. Os *tiles*, por sua vez, foram criados utilizando uma classe própria contendo um identificador em cada tipo de peça, identificador esse que é armazenado na matriz para demonstrar sua existência naquela posição. Além do



(a) Matriz de 50x50 *tiles*

(b) Matriz de 25x75 *tiles*

(c) Matriz de 75x25 *tiles*

Figura 3.1: Exemplos de saída para diferentes dimensões da matriz, onde os pontos verde e vermelho são, respectivamente, o inicio e o final do nível.

identificador, cada *tile* foi idealizado como um quadrilátero com informações sobre os possíveis encaixes em cada uma de suas arestas. Esses *tiles* foram adicionados a um vetor, que serve como uma paleta (Fig. 3.2) para o sorteio e a organização na matriz, seguindo as diretrizes de encaixe.

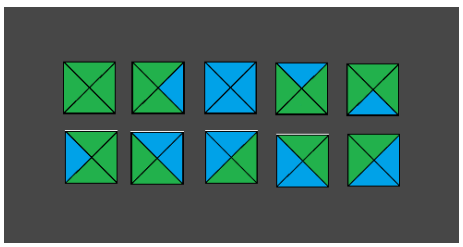


Figura 3.2: Conjunto de *tiles* utilizados. Cada *tile* tem 4 segmentos alinhados às arestas, e cada segmento pode assumir a cor azul ou verde.

Como entrada deste algoritmo, temos o tamanho e a altura da matriz a ser criada para abrigar o mapa, além da informação de cada *tile* disponível para sorteio. Nesta etapa, partindo do ponto inicial da matriz, uma peça aleatória é selecionada na paleta e

seu número identificador é inserido na matriz nas coordenadas iniciais. O algoritmo então avança para a próxima posição, onde, após fazer uma leitura das peças ortogonalmente adjacentes (caso existam), selecionará na paleta um subconjunto de possíveis peças para encaixe. Com o subconjunto selecionado uma das peças é sorteada e seu identificador acrescentado nas coordenadas atuais na matriz. O processo se repete até que toda a matriz seja preenchida com as peças e o leiaute seja gerado (Fig. 3.3a). O Algoritmo 3.2 apresenta a implementação utilizada neste trabalho.

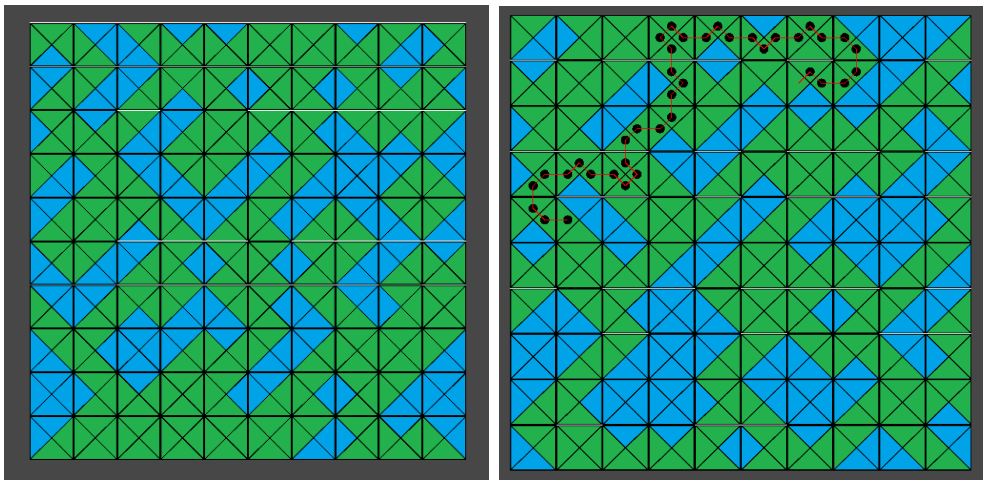
```
1  void WangTiles(int[,] map, int width, int height, List<Tiles>
       paleta)
2  {
3      System.Random randNum;
4
5      map = new int[width, height];
6      for (int x = 0; x < width; x++)
7          for (int y = 0; y < height; y++)
8              map[x, y] = SorteiaTile(x,y,paleta);
9  }
10
11 int SorteiaTile(int x, int y, List<Tiles> paleta)
12 {
13     // se primeiro tile, seleciona um aleatorio
14     if (x == 0 & y == 0)
15         int newTile = randNum.Next(numTilesPaleta);
16         return newTile;
17     // se primeira fila, procura alguem que se encaixa na
         esquerda
18     if (y == 0 & x >= 0)
19         int newTile = ProcuraTileEncaixe(encaixeEsquerda);
20         return newTile;
21     // se primeira coluna, procura alguem que se encaixa abaixo
22     if (y > 0 & x == 0)
23         int newTile = ProcuraTileEncaixe(encaixeAbaixo);
24         return newTile;
25     // procura alguem que se encaixa na lateral esquerda e abaixo
26     if (x > 0 & y > 0)
27         int newTile = ProcuraTileEncaixe(encaixeEsquerda,
             encaixeAbaixo);
28         return newTile;
29     else
30         return 0;
31 }
```

Algoritmo 3.2: Implementação do algoritmo Wang Tiles.

Devem ser criados todos os *tiles* necessários para garantir todos os possíveis encaixes, evitando situações onde nenhuma peça consiga se adequar. Para este trabalho foi

criado um conjunto de 10 *tiles*, com cada *tile* dividido em 4 segmentos e cada divisão alinhada a uma aresta. Cada um desses segmentos e suas arestas pode assumir 2 tipos. O encaixe é definido a partir do tipo assumido em cada um dos seus 4 segmentos. Como pode ser visto na Figura 3.2, cada *tile* conta com 4 segmentos. Neste trabalho, os segmentos de cor verde representam caminhos livres e os de cor azul identificam paredes ou áreas sem acesso.

Como podemos ver na Figura 3.3b, é possível obter uma estrutura de grafo a partir do nosso leiaute. Para cada segmento caminhável do *tile* (representado pela cor verde), um novo vértice é gerado. Por sua vez, cada aresta adjacente se torna uma conexão.



(a) Leiaute gerado pelo algoritmo. **(b)** Caminho computado para o leiaute.

Figura 3.3: Exemplo utilizando duas possibilidades de encaixe (cores) no algoritmo Wang Tiles. Mesmo com apenas duas possibilidades, é possível ver a variedade de resultado com múltiplos encaixes.

Compatibilidade

Este método atende bem a ambos os subgêneros. Usando o resultado como um leiaute final, é possível utilizá-lo como nível em um jogo *roguelike*, deixando o jogador livre, ou *metroidvania*, explorando sua capacidade de se tornar mais complexo de acordo com a paleta inicial usada.

3.2.3 Passeio aleatório

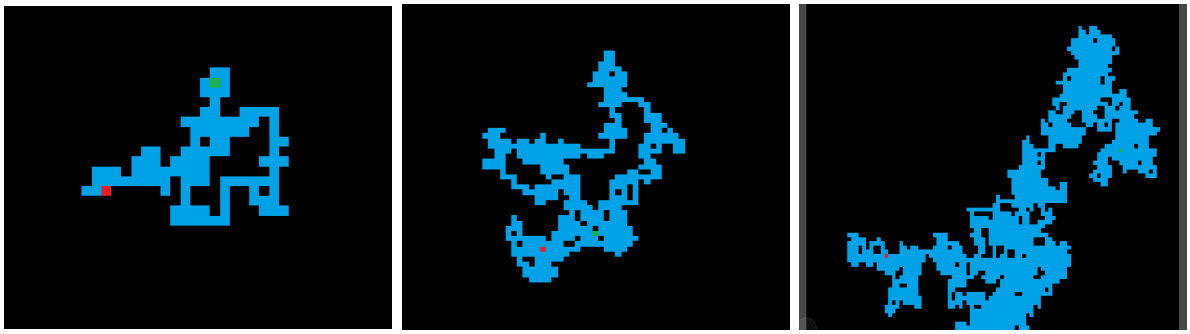
Para este algoritmo, uma matriz de inteiros de altura e largura indicados na entrada é criada e um número de passos N é previamente escolhido e inserido pelo usuário. O

algoritmo marca todos os campos com o valor indicativo de ocupado, definido como o número 0. Um par de coordenadas é sorteado como ponto inicial na matriz. A partir do *tile* inicial, o algoritmo realiza N iterações. A cada iteração, o algoritmo escolhe aleatoriamente uma das quatro direções da matriz e verifica a possibilidade de avanço. Paredes não são consideradas avanços válidos. Assim, se o avanço sorteado é na direção de uma parede, uma nova direção é sorteada sem que um passo seja contado. Caso o avanço seja possível, o algoritmo avança, marcando a matriz na nova posição com o número 1 (caminho disponível) e incrementando em um o número de passos efetuados, mesmo que o passo seja em um *tile* já visitado. No fim do processo, o último campo visitado é marcado como ponto final. A matriz é então percorrida e *tiles* são distribuídos nas posições marcadas com valor 1. Como resultado temos construções semelhantes às variações presentes na Figura 3.4. A implementação do passeio aleatório pode ser vista no Algoritmo 3.3.

```
1  void PasseioAleatorio(int[,] map, int width, int height, int
       nPassos)
2  {
3      System.Random randNum;
4
5      // prepara a matriz
6      map = new int[width, height];
7      for (int i = 0; i < width; ++i)
8          for (int j = 0; j < height; ++j)
9              setTile(i, j, 0);
10
11     while(contPassos < nPassos) {
12         // sorteia uma direcao
13         int dir = UnityEngine.Random.Range(0, 4);
14         // verifica se direcao de dir e valida
15         if (scanAround())
16             switch (dir)
17                 case 0:
18                     setTile(map, avancaParaDireita(), 1);
19                     break;
20                 case 1:
21                     setTile(map, avancaParaEsquerda(), 1);
22                     break;
23                 case 2:
24                     setTile(map, avancaParaCima(), 1);
25                     break;
26                 case 3:
27                     setTile(map, avancaParaBaixo(), 1);
28                     break;
```

```
29                    default:
30                        break;
31        }
32    }
```

Algoritmo 3.3: Implementação do passeio aleatório.



(a) **(b)** **(c)**

Figura 3.4: Caminhos gerados pelo algoritmo para (a) 100, (b) 500 e (c) 1000 passos.

Compatibilidade

Apesar da aleatoriedade e do difícil controle em sua forma básica, é possível encaixar os resultados gerados para ambos os subgêneros, filtrando os resultados com base em algumas características presentes no leiaute. Definir um valor mínimo para a solução, para evitar caminhos muito curtos, ou, no caso do *roguelike*, um valor máximo para não perder a dinâmica de agilidade do subgênero. Também é possível estabelecer um número mínimo de caminhos para finalização do nível, além da presença de áreas-chave, no caso do *metroidvania*, todos esse filtros podem ser aplicados no período de produção para escolher o melhor leiaute a ser usado, ou durante a execução do jogo, porem neste ultimo e necessário verificar o custo e tempo para gerar tal saída que se enquadre nos requerimentos.

3.3 Estruturas usadas para análise

Para armazenar o resultado de cada execução e poder analisar os *layouts*, foi utilizada uma estrutura de lista de adjacência para representar um grafo, em que cada *tile* do leiaute foi mapeado como um vértice e cada ligação entre eles como uma aresta. O resultado gerado

por cada algoritmo é convertido na estrutura em grafo, que é usada para computar dados sobre o leiaute.

Para o cômputo desses dados, foram escolhidos alguns algoritmos de busca não-informada. O algoritmo de Dijkstra foi usado para uma análise de caminho mínimo, bem como a distância de cada vértice até todos os demais vértices. Também foi calculado o grau de conectividade entre todas as áreas disponíveis. Já o algoritmo de busca em profundidade foi usado para determinar uma das métricas de distância. Essa métrica visa aproximar a distância que um jogador humano percorreria ao tentar encontrar o caminho até o destino. A busca em profundidade, devido à sua característica de busca não informada, pode, em alguns aspectos, se assemelhar ao comportamento do jogador que desconhece o mapa.

3.4 Métricas propostas

Para realizar a análise de cada resultado gerado, foram estabelecidas métricas com base em características comuns presentes em todos os *layouts* gerados. Essas métricas visam medir e analisar o leiaute base quanto à estrutura e à organização e uso do espaço gerado. Cada métrica é descrita nesta Seções a seguir.

Distância mínima

Esta métrica é calculada diretamente a partir da saída obtida pelo algoritmo de Dijkstra, que computa o caminho mínimo do ponto inicial ao ponto final do leiaute.

Distância provável

A distância provável é o comprimento de um caminho provável qualquer existente entre o ponto inicial e o ponto final do leiaute. Obtido pelo percurso em profundidade, a ideia deste caminho é se aproximar da forma como o jogador exploraria um cenário sem qualquer conhecimento prévio a respeito do mesmo. Neste trabalho, foi utilizado o algoritmo de percurso em profundidade padrão, com uma visita sistemática dos nós do grafo que representa o mapa gerado, sempre na mesma ordem. No entanto, o percurso também

poderia ser adaptado para aleatorizar a ordem de escolha das direções a serem exploradas pelo algoritmo, se aproximando mais do comportamento do jogador.

Conectividade geral

A conectividade geral pode ser obtida com qualquer um dos dois métodos de busca. Seu papel é indicar se o leiaute gerado é conexo ou desconexo. Seja V o conjunto de vértices do grafo que representa o mapa e C o conjunto contendo todos os caminhos partindo do ponto inicial. A conectividade geral pode ser determinada pela seguinte condição:

$$\forall v_i, v_j \in V, \exists c \in C | v_i \neq v_j, (v_i, v_j) \in c.$$

Conectividade média

Esta métrica representa o número médio de conexões dos vértices. É calculada a partir do total de vértices (*tiles*) do mapa gerado e da soma de todas as conexões ou arestas criadas entre estes vértices. Seja V o conjunto de vértices do grafo e $a(v)$ a função que retorna o número de arestas partindo de um vértice $v \in V$. A conectividade média Γ pode ser calculada por:

$$\Gamma = \frac{1}{|V|} \sum_{v \in V} a(v).$$

Altura e largura do leiaute

A largura geral do layout representa a distância entre a maior e a menor coordenada válida no eixo x da matriz. A altura é definida da mesma forma, porém considerando o eixo y.

Relação de uso do espaço

Esta métrica representa a porcentagem do total de vértices (*tiles* válidos do mapa) do grafo que o caminho mínimo ou provável utiliza. Esta relação pode ser definida como:

$$P = \frac{|c| \times 100}{|V|},$$

onde c é o caminho e V é o conjunto de vértices do grafo.

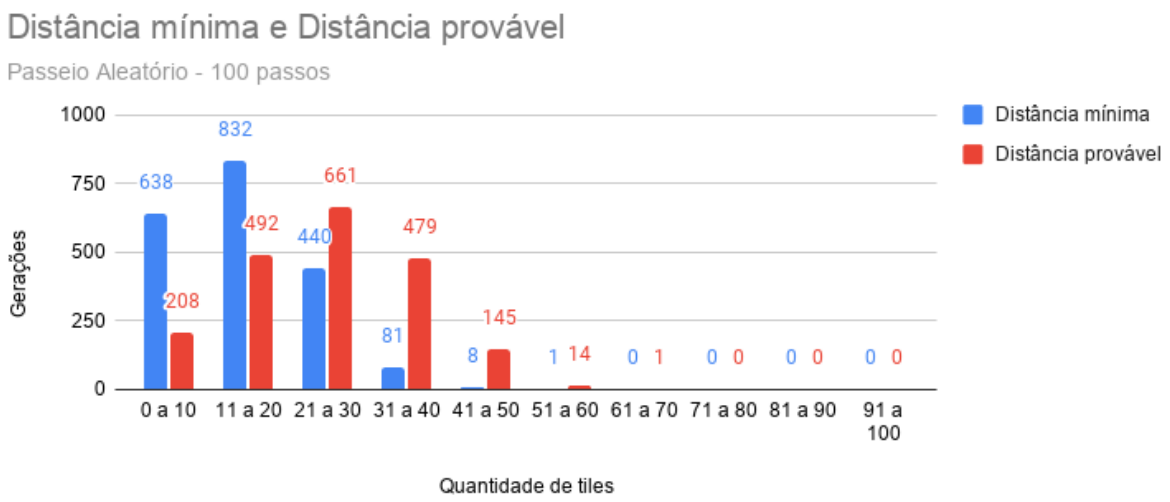
4 Análises e resultados

As seções abaixo apresentam os dados das métricas propostas, gerados após 2000 execuções de cada algoritmo. Para cada execução, foi utilizado o ambiente de desenvolvimento Unity em um sistema Windows 10 64bits com 8GB de memoria RAM DDR4, processador AMD Ryzen 5 3600 de 3.60Ghz e placa gráfica GForce GTX750Ti 2Gb. A partir dos resultados obtidos para cada métrica, é feito um paralelo entre múltiplas execuções de cada solução, com diferentes variações de entrada. Após a comparação dos algoritmos, serão apresentadas também propostas e hipóteses de uso e manipulações do leiaute baseadas na análise feita em cima dos dados apresentados neste capítulo.

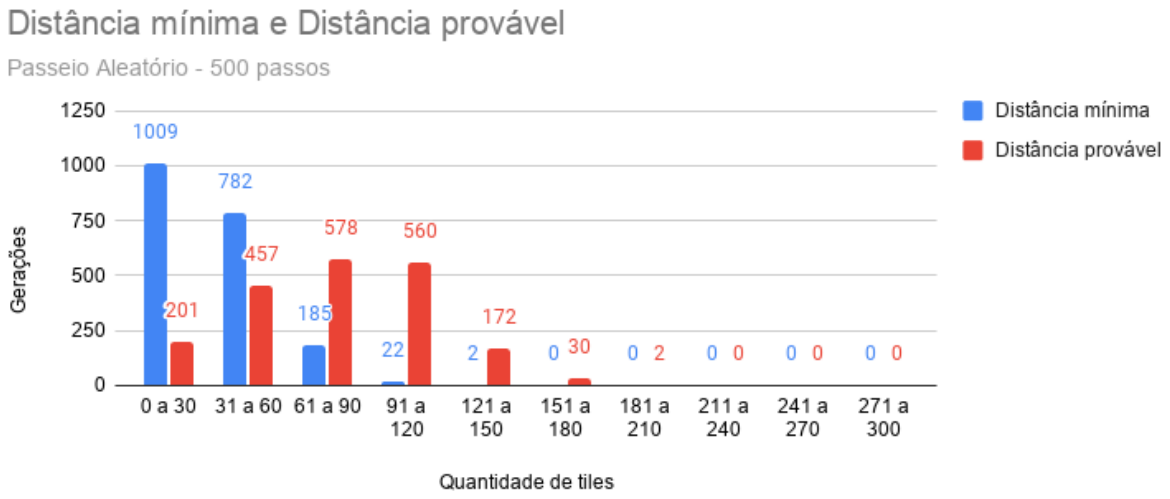
4.1 Distância mínima e Distância provável

Os gráficos desta seção evidenciam como se dá a distribuição das métricas de distância obtidas em *layouts* gerados. Para o algoritmo de passeio aleatório descrito na Seção 3.2.3, considere n como sendo o número de passos fornecido como entrada. Foram realizados experimentos com $n = 100$, $n = 500$ e $n = 1000$, compilados nas Figuras 4.1. É possível perceber que a distribuição de ocorrências da **Distância mínima**, quando $n = 100$ (Fig. 4.1a), é maior em caminhos de tamanhos de 0 a 30 *tiles*. Para os outros valores de n, as ocorrências estão predominantemente nos caminhos de tamanhos 0 a 90 (Fig. 4.1b e 4.1c). A incidência de comprimentos maiores para a **Distância mínima** tende a decair gradativamente à medida que o tamanho dos caminhos aumenta. Para $n = 100$ (Fig. 4.1a) e $n = 1000$(Fig. 4.1c), há um pequeno aumento inicial nesses valores, mas a tendência de queda se confirma na sequência. Já para a **Distância provável**, a distribuição dos comprimentos dos caminhos é um pouco mais dispersa nas faixas de tamanhos, tendo um comportamento similar a uma distribuição normal.

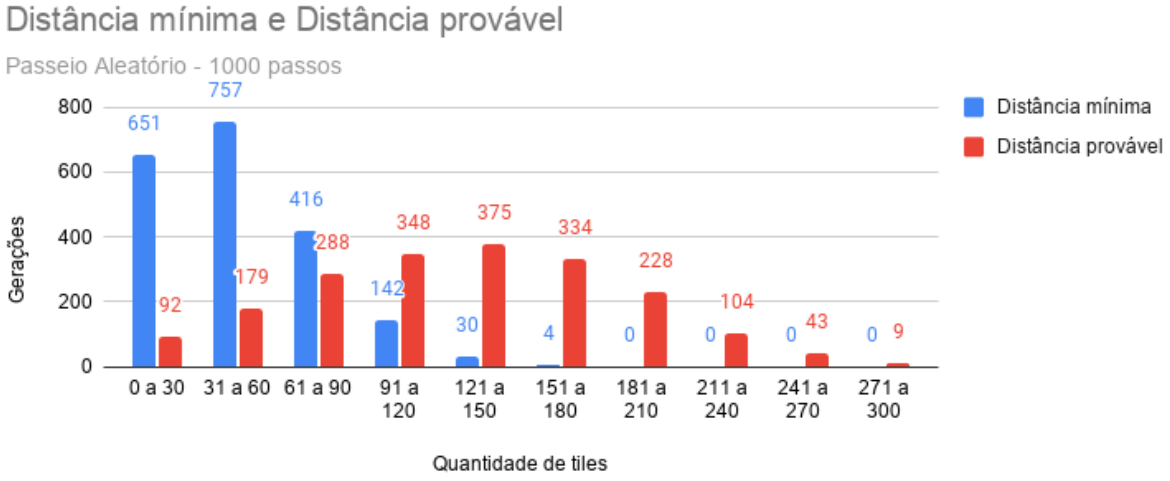
O padrão perceptível no algoritmo de Wang Tiles e no autômato celular é similar para o **Distância mínima**, demonstrando uma maior incidência em faixas menores de tamanho (Figs. 4.3 e 4.3). Já a distribuição da **Distância provável** se comporta de modo



(a) $n = 100$

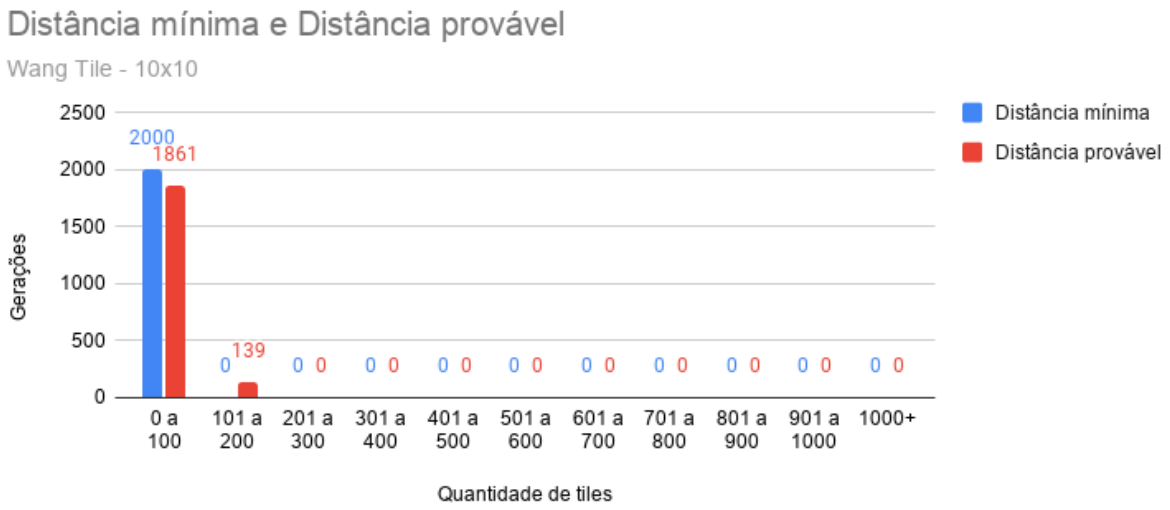


(b) $n = 500$

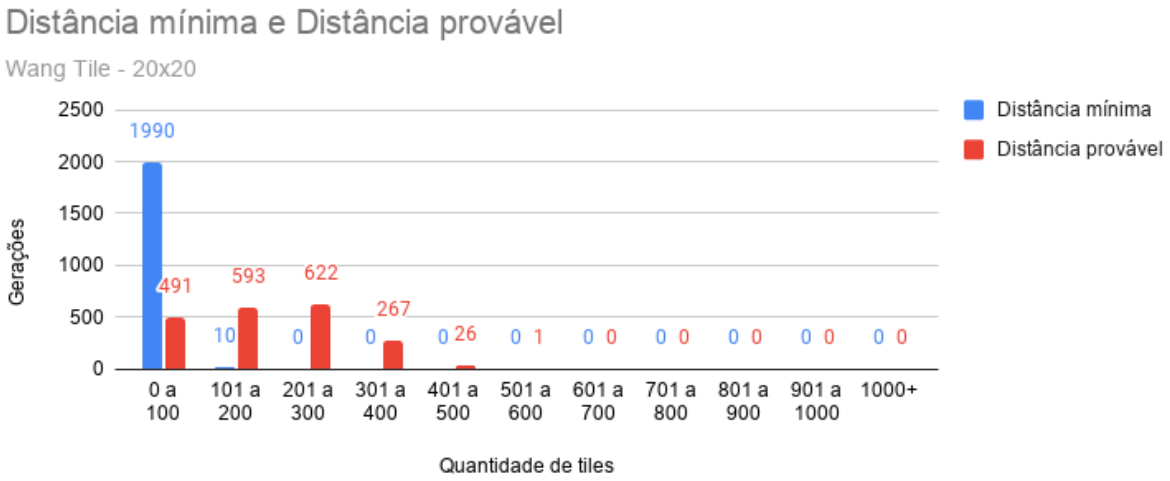


(c) $n = 1000$

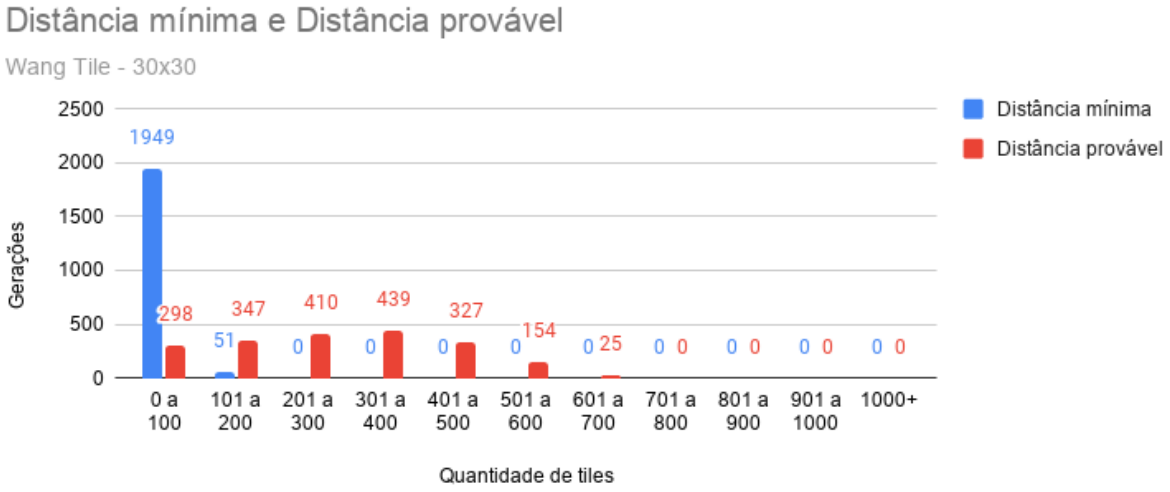
Figura 4.1: Métricas de distância mínima e distância provável para o algoritmo do passeio aleatório, utilizando valores de n iguais a 100, 500 e 1000.



(a) Matriz 10x10

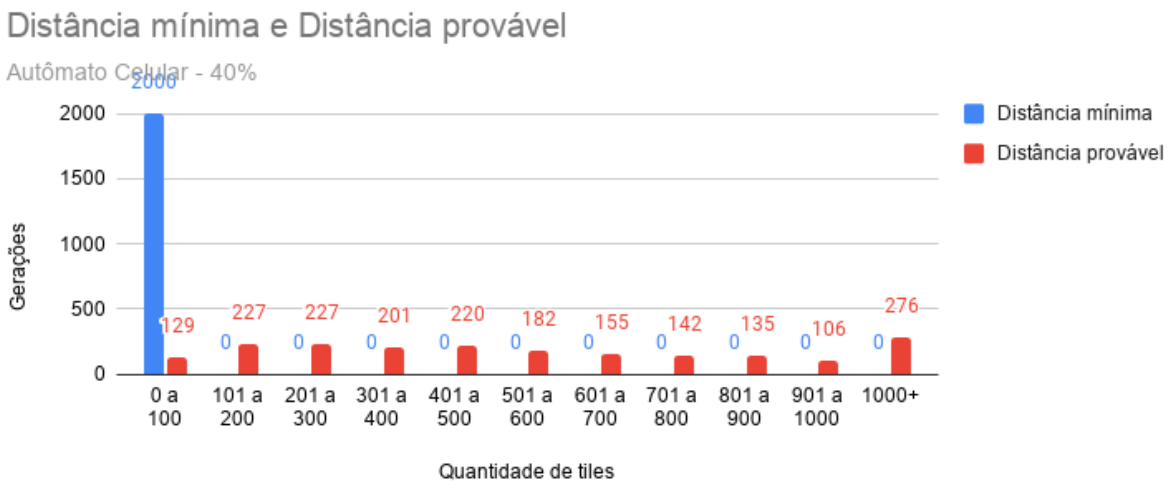


(b) Matriz 20x20

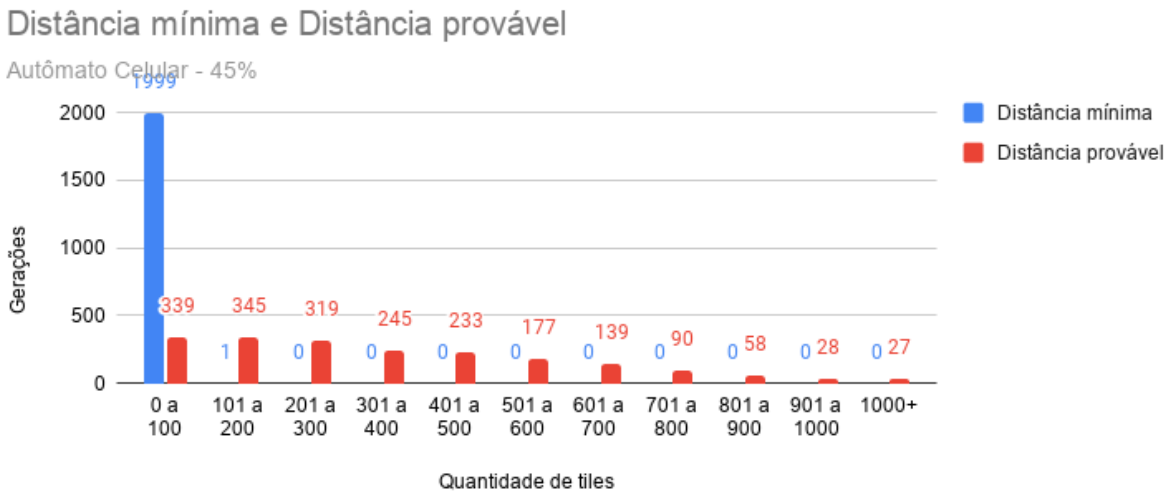


(c) Matriz 30x30

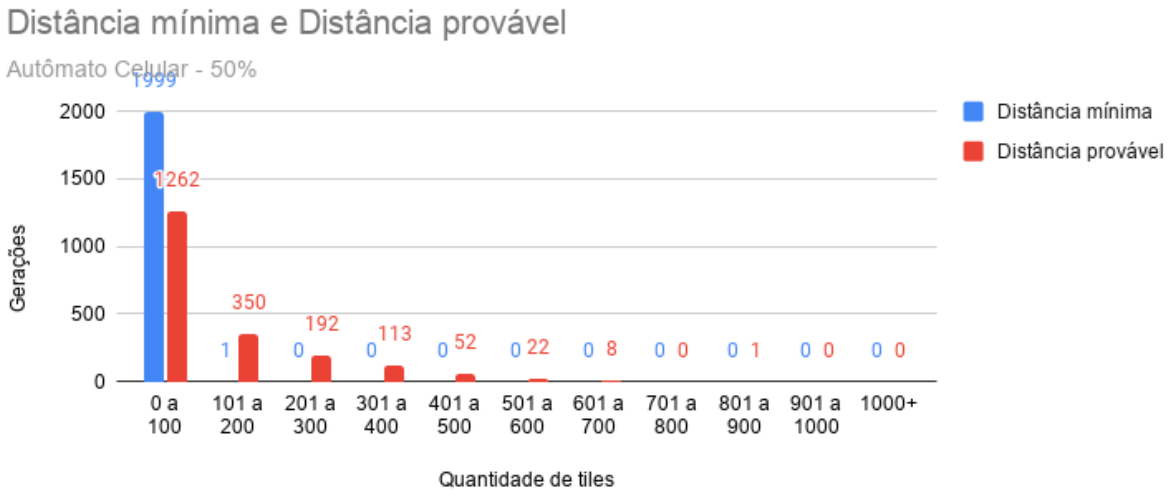
Figura 4.2: Métricas de distância mínima e distância provável para o algoritmo Wang Tiles, usando matrizes de dimensões totais 10x10, 20x20 e 30x30.



(a) Preenchimento inicial de 40%.



(b) preenchimentos inicial de 45%.



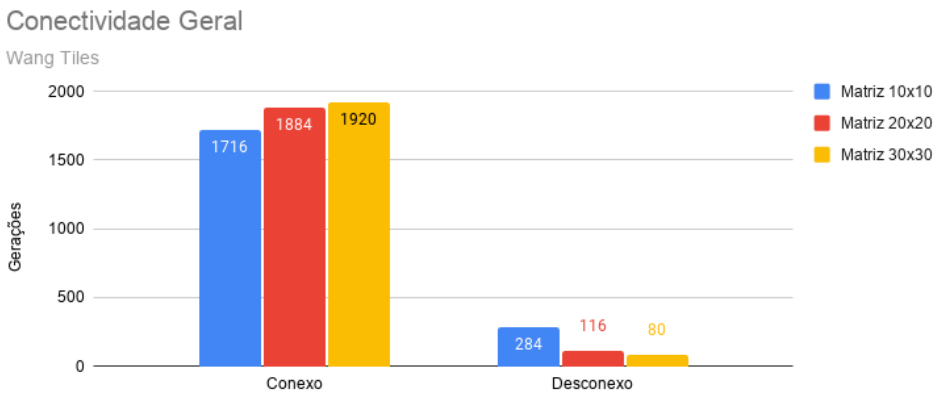
(c) preenchimentos inicial de 50%.

Figura 4.3: Métricas de distância mínima e distância provável para o autômato celular, com preenchimentos iniciais de 40%, 45% e 50%.

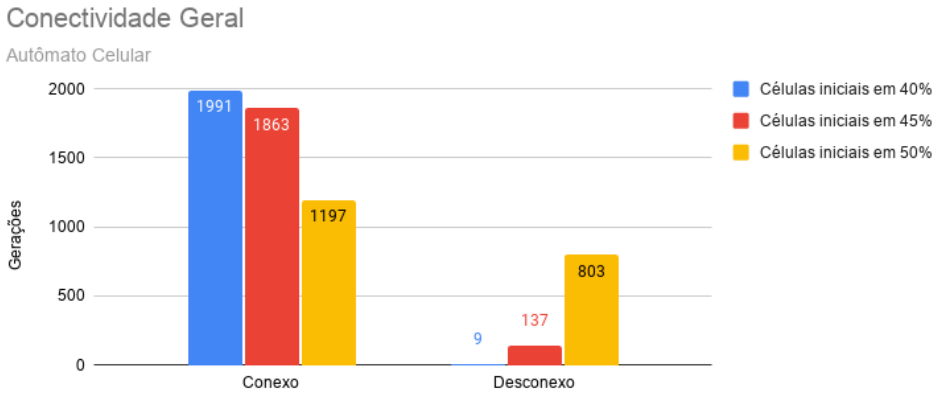
quase uniforme, com exceção da Figura 4.3c, que representa o caso em que o autômato celular é executado com preenchimento inicial de 50%. Isso se explica pelo fato de que o autômato, para esse parâmetro de entrada, em virtude das regras de sobrevivência, gera áreas muito abertas, que resultam em caminhos muito diretos até o objetivo.

4.2 Conectividade geral

Devido à forma de funcionamento do algoritmo de passeio aleatório, não há possibilidade de se obter um ambiente fragmentado, por isso neste caso pode-se sempre considerar que o produto gerado será um grafo conexo. Para os demais algoritmos, as distribuições de ocorrência de um leiaute conexo ou desconexo podem ser vistas na Figura 4.4.



(a) Wang Tiles



(b) Autômato celular

Figura 4.4: Métrica de conectividade geral para os algoritmos Wang Tiles e autômato celular

No algoritmo Wang Tiles (Fig.4.4a), pode-se notar que quanto menor a matriz usada para gerar o leiaut, mais resultados desconexos ocorrem. Já no autômato celular, ilustrado no gráfico da Figura 4.4b, a maior incidência de resultados desconexos acontece

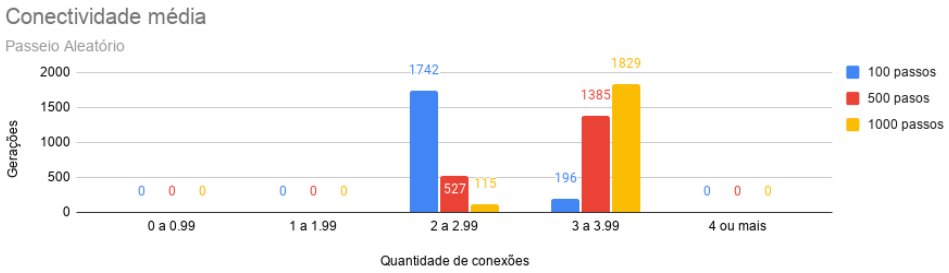
à medida que a porcentagem de preenchimento inicial aumenta. Isso está relacionado ao fato de que quanto mais células iniciais, mais essas células são convertidas em células mortas, como pode ser visto pela regra de sobrevivência usada e descrita na Seção 3.2.1.

4.3 Conectividade média

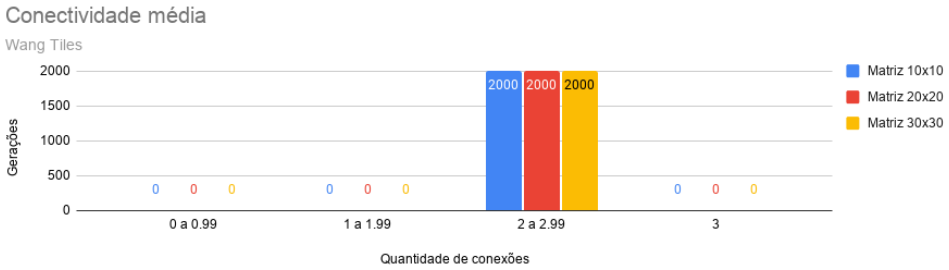
Nos gráficos da Figura 4.5, podemos observar a variação na média de conexões que cada *tile* gera em cada tipo de algoritmo. Os resultados do passeio aleatório (Fig.4.5a) demonstram o maior número de conexões à medida que mais passos são efetuados, com **Conectividade média** de 2 a 2.99 para *layouts* de 100 a 500 passos e 3 a 3.99 para *layouts* de 1000 passos. Por outro lado, o autômato celular (Fig.4.5c) e o algoritmo Wang Tiles (Fig.4.5b) se concentram em um único intervalo, mesmo quando variados os parâmetros de entrada. No caso do autômato, isso ocorre devido às regras de sobrevivência adotadas para o autômato, que foram definidas de modo a preservar *tiles* com estado vivo, aumentando o número da **Conectividade médias**. Para o Wang Tiles, o resultado se deve à paleta inicial do algoritmo, que foi planejada para favorecer a escolha de *tiles* com maior capacidade de formar conexões, como pode ser visto na Figura 3.2, em que a paleta utilizada apresenta 5 peças com 3 conexões, 4 peças com 2 conexões e apenas uma peça com nenhuma conexão.

4.4 Largura e altura do leiaute

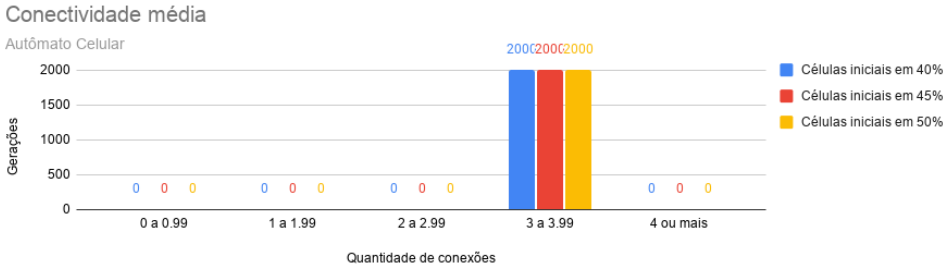
Os resultados para o autômato celular (Fig.4.7) indicam que a **Altura** e a **Largura** do leiaute gerado apresenta um comportamento contrário ao valor definido para o preenchimento inicial, onde essas métricas assumem valores menores ao ponto que a preenchimento inicial aumenta, o que novamente pode ser explicado pelo fato das regras de sobrevivência da Seção 3.2.1. No passeio aleatório (Fig.4.8), em sua implementação mais básica, com uma probabilidade igual para todas as direções, a **Largura** e a **Altura** do leiaute tendem a crescer com o número de passos, além de não variarem muito entre si, o que pode indicar que o crescimento em ambos os eixos se comporta de forma proporcional. Já para o Wang Tiles (Fig.4.6), para todos os experimentos executados, os valores obtidos para **Largura** e **Altura** foram exatamente iguais ao tamanho da matriz.



(a) Passeio aleatório



(b) Wang Tiles

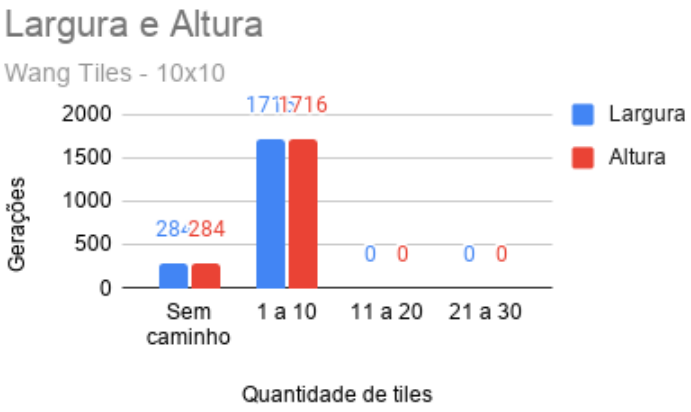


(c) Autômato celular

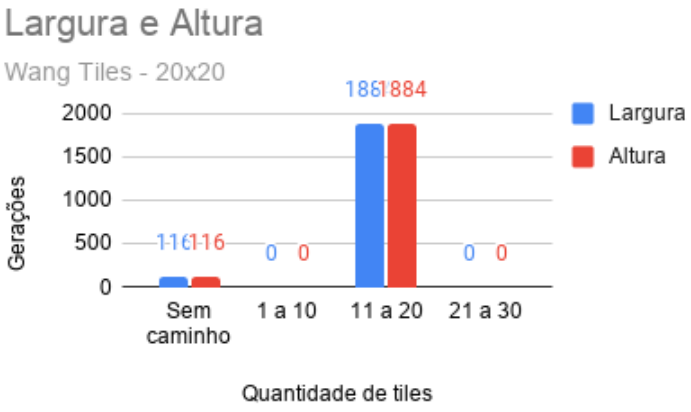
Figura 4.5: Métrica de conectividade média para os três algoritmos analisados. Note que, para este algoritmo, há no máximo 3 conexões devido às características do conjunto de peças escolhido como explicado na Seção 4.3 onde e visto conexões que cada peça pode formar.

4.5 Relação de uso do espaço

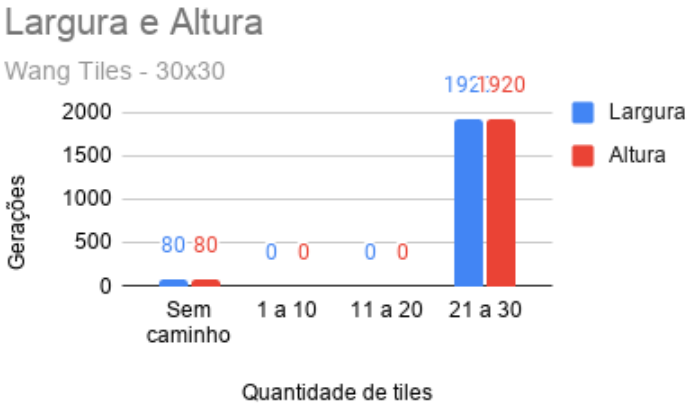
Esta métrica estabelece uma relação entre o total de *tiles* válidos gerados em cada layout e o espaço visitado nos percursos usando o **Caminho provável** ou o **Caminho mínimo**. O algoritmo de passeio aleatório (Fig.4.9) apresenta uma boa variedade de distribuição dos **Caminhos prováveis** para diferentes execuções, sendo possível gerar *layouts* com taxas de uso total do mapa entre 25% e 90% de aproveitamento, para mapas gerados com até 1000 passos. Para o **Caminho mínimo**, os resultados são mais concentrados na faixa de até 25% nas execuções com 500 (Fig.4.9b) e 1000 (Fig.4.9c) passos. Para execuções de até 100 passos, a concentração alcança valores expressivos na faixa de 51% a 75% oque pode indicar que o caminho mínimo e provável possam ter tamanhos iguais para *layouts* de ate 100 passos (Fig.4.9a).



(a) Matriz 10x10



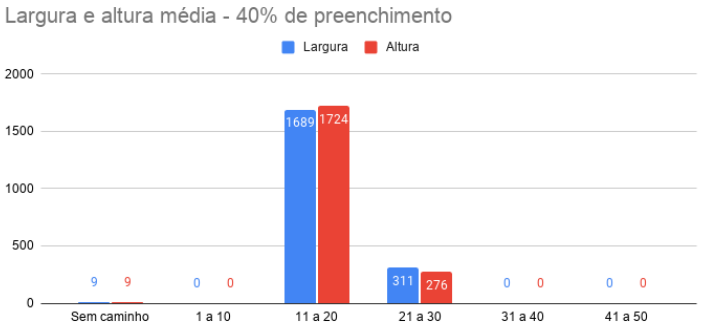
(b) Matriz 20x20



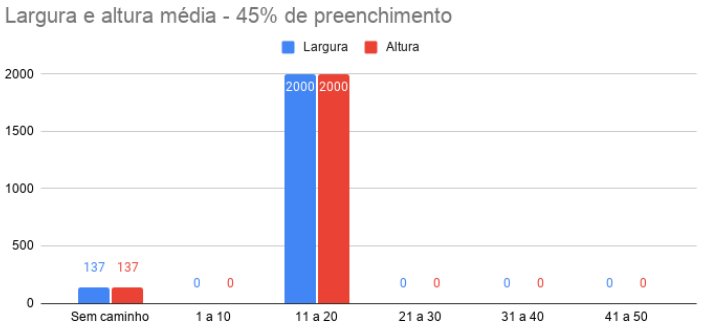
(c) Matriz 30x30

Figura 4.6: Métrica de largura e altura para o algoritmo Wang Tiles.

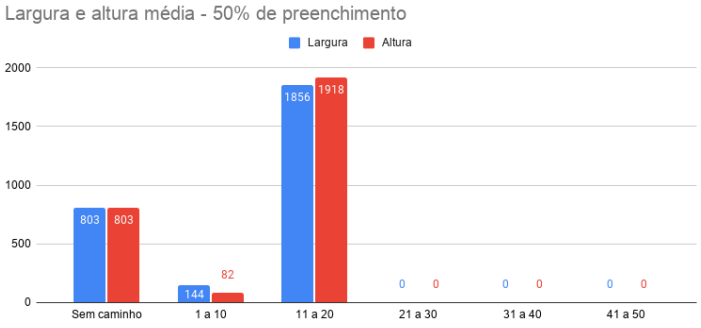
O algoritmo de passeio aleatório apresenta uma distribuição bem mais equilibrada que os demais. Baseado na métrica de conectividade média do passeio aleatório, pode-se traçar um paralelo entre elas. Tendo uma conectividade média menor que os demais, o algoritmo de passeio aleatório apresenta estruturas mais lineares e, portanto, a **Distância mínima** e **Distância provável** cobrem uma maior parte do mapa, o que pode explicar



(a) Preenchimento em 40%



(b) Preenchimento em 45%

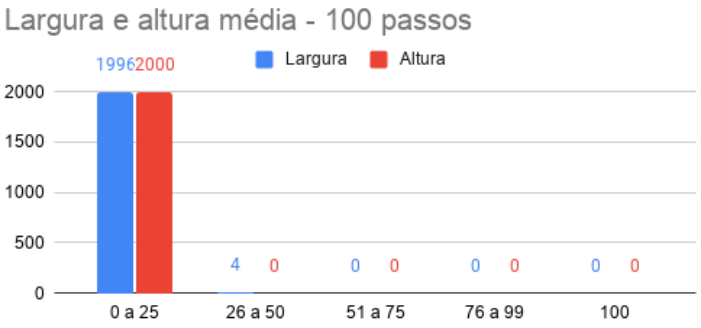


(c) Preenchimento em 50%

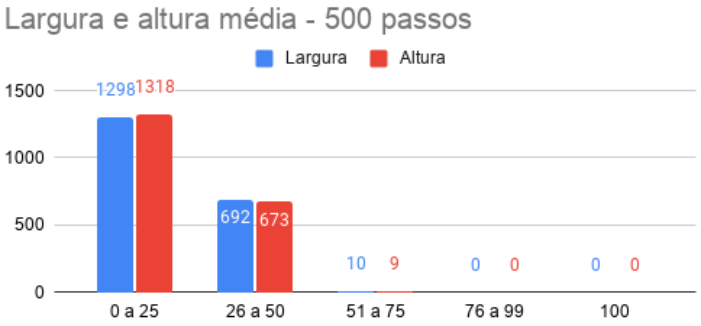
Figura 4.7: Métrica de largura e altura para o autômato celular.

a maior taxa de uso de espaço.

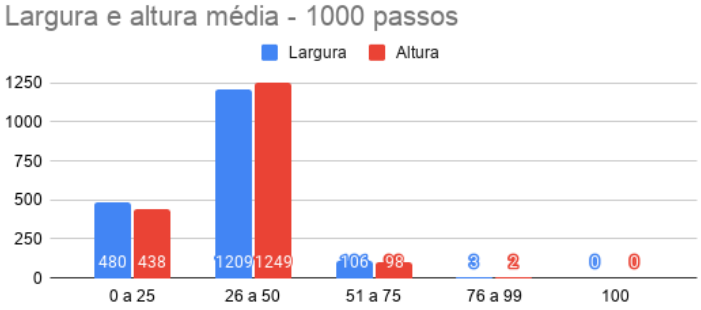
O algoritmo Wang Tiles (Fig. 4.10) apresenta poucas variações com modificações dos parâmetros de entrada, tendo como valor predominante do **Caminho miníma** o uso de até 25% do espaço produzido, enquanto o **Caminho provável** sofre uma pequena variação, com algumas gerações alcançando valores entre 26% e 50% do espaço. Os resultados do autômato celular (Fig. 4.11) são, em geral, bem similares aos do Wang Tiles. Como ambos apresentam uma valor elevado de **Conectividade média**, conforme demonstrado nas Figuras 4.5b e 4.5c, é provável que essa media alta indique uma maior quantidade de caminhos possíveis para se movimentar e, portanto, alcançar o objetivo



(a) Caminhada de 100 passos.



(b) Caminhada de 500 passos.



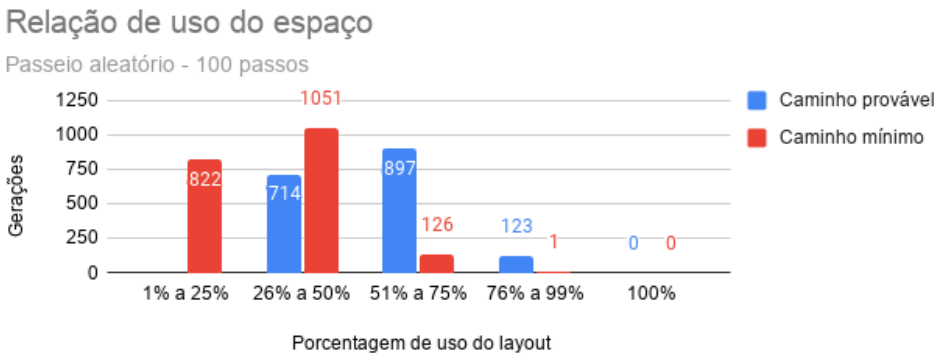
(c) Caminhada de 1000 passos.

Figura 4.8: Métrica de largura e altura para o passeio aleatório.

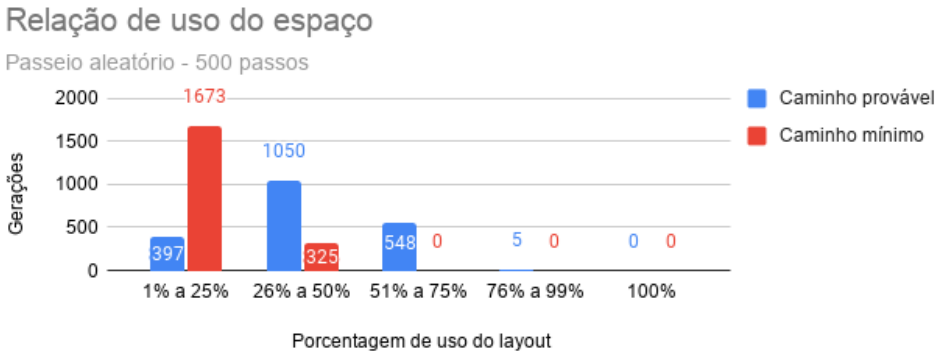
sem realizar tantos movimentos.

4.6 Uso das métricas na adaptação dos algoritmos

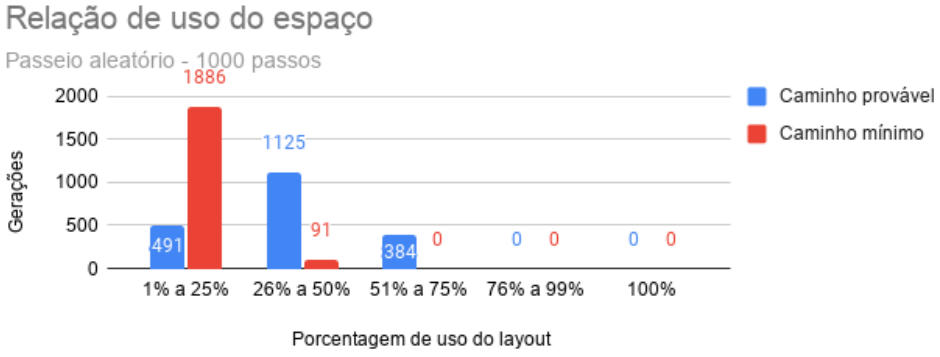
Nesta seção, são discutidas algumas possibilidades de uso das informações obtidas através das métricas apresentadas. O objetivo é, a partir do significado desses resultados, levantar hipóteses para adaptação dos algoritmos analisados, de forma que os *layouts* gerados possam melhor atender às características do subgêneros considerados.



(a) Caminhada de 100 passos.



(b) Caminhada de 500 passos.

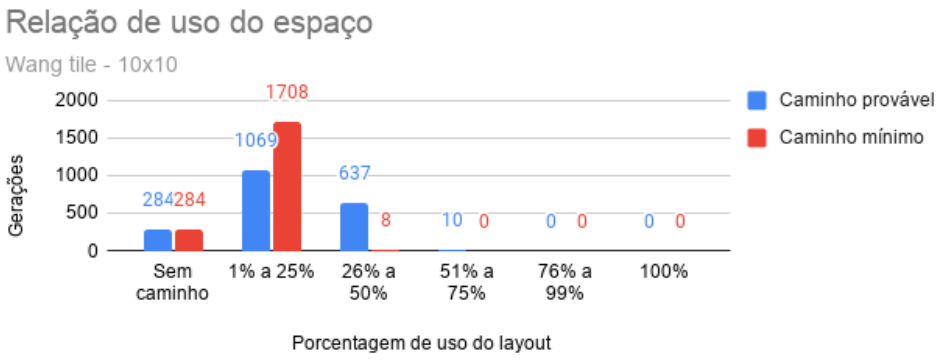


(c) Caminhada de 1000 passos.

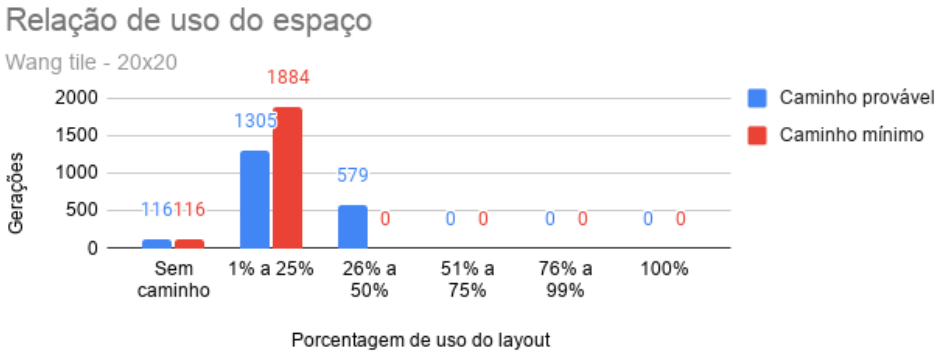
Figura 4.9: Métrica de relação de uso do espaço para o algoritmo de passeio aleatório.

4.6.1 Distância mínima e Distância provável

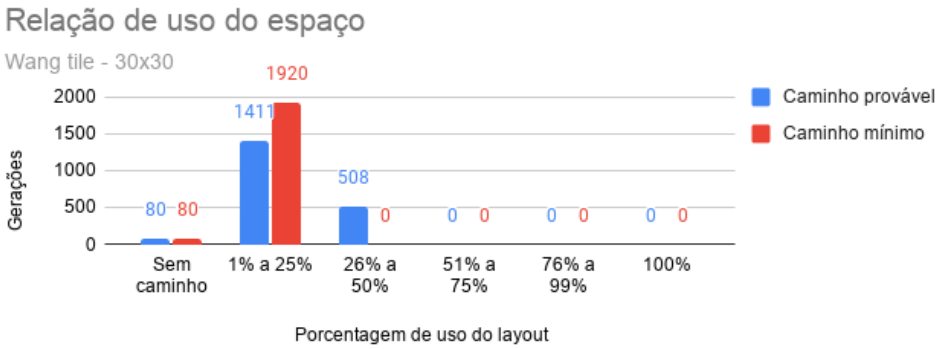
Com acesso às métricas de **Distância mínima** e **Distância provável**, é possível ter a percepção do quão simples é o leiaute gerado. Em casos onde a **Distância provável** é próxima ou igual à Distância mínima, o percurso provável adotado pelo jogador será muito próximo do caminho mínimo necessário para alcançar o objetivo, o que pode resultar em um mapa com baixo grau de desafio. Conhecendo cada *tile* dos caminhos calculados, é possível bloquear algum ponto da trajetória, permitindo gerenciar melhor o percurso do jogador(Fig.4.12). Esse controle sobre o percurso pode ser especialmente útil



(a) Matriz 10x10



(b) Matriz 20x20



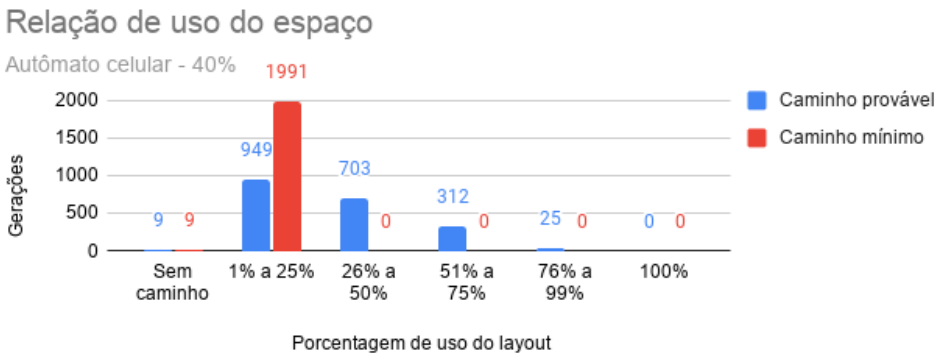
(c) Matriz 30x30

Figura 4.10: Métrica de relação de uso do espaço para o algoritmo Wang Tiles.

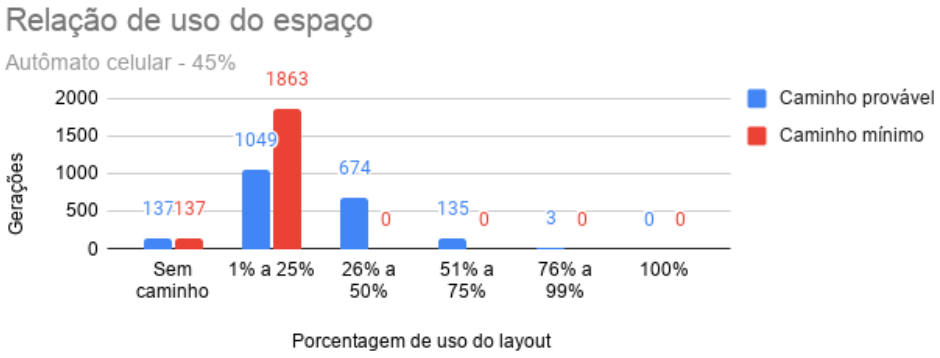
no subgênero *metroidvania*, em que áreas já exploradas são frequentemente revisitadas e atalhos e desvios são comumente implementados.

4.6.2 Conectividade média

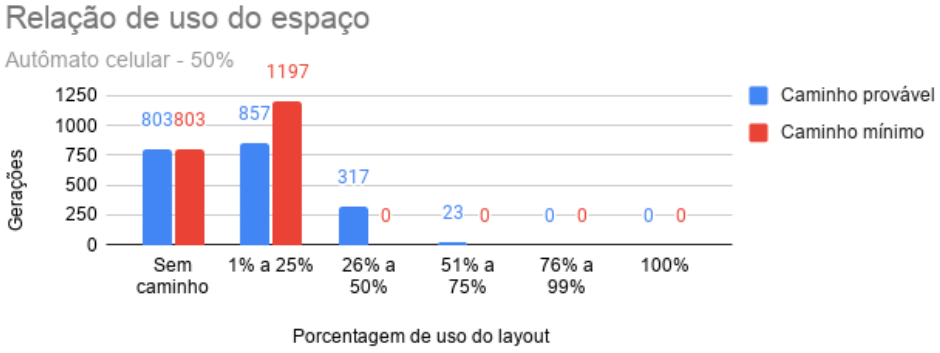
A possibilidade de obter a **Conectividade média** do leiaute ou de parte dele permite analisar uma série de possibilidades para adaptação do mapa gerado. Abaixo, são discutidas algumas dessas possibilidades.



(a) Preenchimento em 40%.



(b) Preenchimento em 45%.



(c) Preenchimento em 50%.

Figura 4.11: Métrica de relação de uso do espaço para o autômato celular.

Localizações de possíveis *hubs*

Um *hub* é um ambiente que funciona como ponto de encontro de vários caminhos. Analisando de forma individual cada vértice ou blocos de vértices interligados, é possível observar o número de conexões do vértice ou bloco e, dessa forma, identificar se ele se ramifica para múltiplos caminhos diferentes com menor conectividade média. A Figura 4.13 ilustra um exemplo em que foram definidos como *hubs* somente vértices ou blocos com o máximo possível de conexões, que neste caso são 4.

Alguns candidatos a *hub* ilustrados na Figura 4.13 não parecem muito interessan-

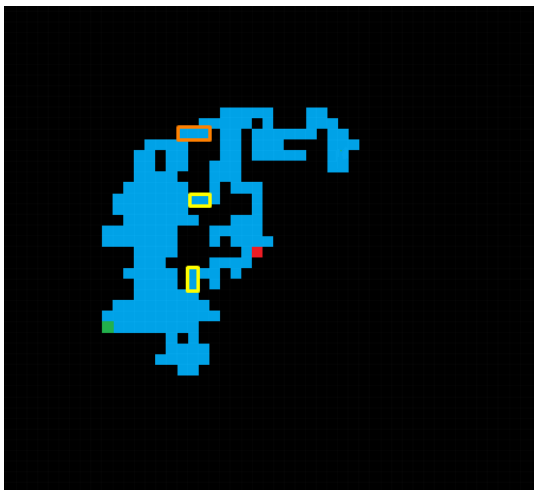


Figura 4.12: Ao bloquear os caminhos em amarelo, o jogador é forçado a usar o caminho laranja. Os caminhos destacados em amarelo e laranja são meramente ilustrativos, não sendo gerados pelo algoritmo.

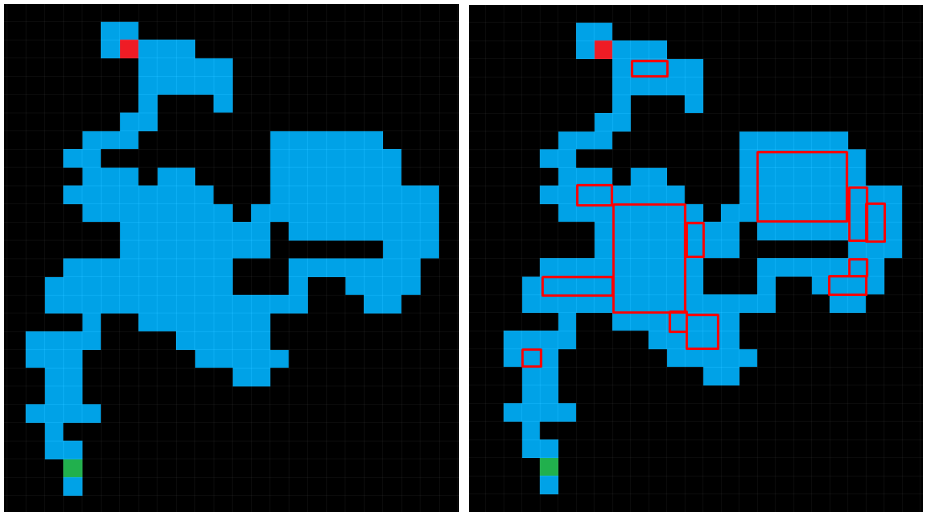


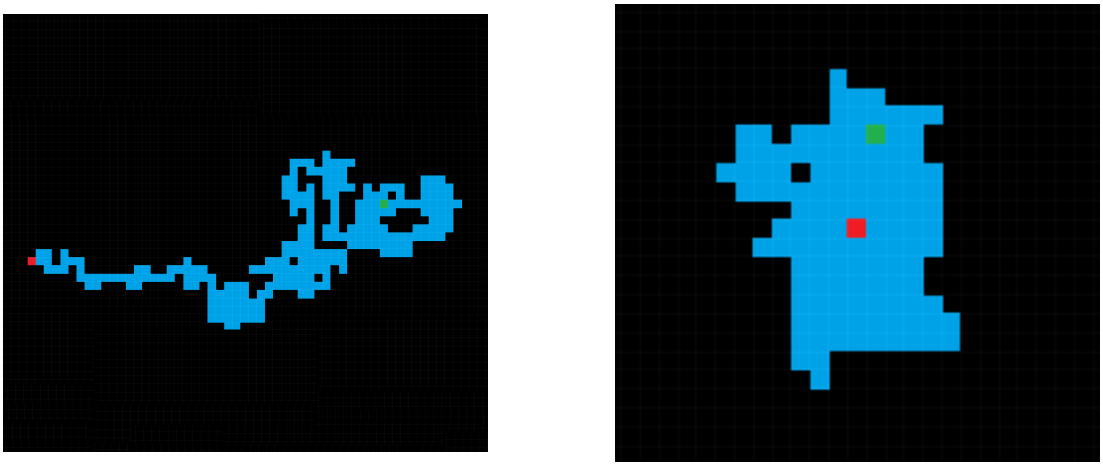
Figura 4.13: À esquerda, um exemplo de mapa gerado pelo algoritmo de passeio aleatório. À direita, o mesmo mapa com regiões destacadas manualmente em vermelho para indicar conjuntos de *tiles* candidatos a serem definidos como *hubs*. Os pontos verde e vermelho no mapa indicam, respectivamente, o início e o fim.

tes, sejam por terem um aspecto de corredor ou por serem muito pequenos. Porém, basta definir um tamanho mínimo para o *hub*, além de definir um limite máximo de conectividade para os tiles ao redor. Assim, são excluídos *tiles* únicos ou regiões muito pequenas para serem consideradas *hubs*. A partir dessas características, é possível também identificar quando um leiaute produzido pode suportar um ou mais *hubs* ou áreas mais abertas e densas que possam ser aproveitadas pelo enredo ou por algum evento específico do jogo.

Determinar geometria do cenário

Como pode ser visto na Figura 4.14, a formação de algumas estruturas pode estar diretamente ligadas ao valor de **Conectividade média**. Um alto índice de **Conectividade**

média indica uma possível aglomeração de conexões de um conjunto de *tiles*. Esta aglomeração pode estar contida em uma área muito pequena, indicando que o leiaute está muito provavelmente concentrado (Fig. 4.14b). É possível visualizar isso como uma estrutura parecida com uma grande sala ou um conjunto de pequenas salas conectadas. Em contraponto, uma **Conectividade média** baixa serve como indicador de áreas possivelmente mais dispersas e lineares (Fig. 4.14a). Setorizar o nível gerado também pode ajudar a dar mais precisão a este tipo de análise, verificando o grau de concentração em partes específicas do mapa.



(a) Conectividade média de 2.8.

(b) Conectividade média de 3.5

Figura 4.14: Dois resultados obtidos com o algoritmo do passeio aleatório, executado com 300 passos.

4.6.3 Conectividade geral

Além de demonstrar se todas as áreas do nível são acessíveis a partir de qualquer ponto, saber se o nível gerado é conexo ou desconexo permite compreender se o leiaute funciona bem ou não para os subgêneros. Identificar quantos fragmentos foram gerados e suas características permite considerar a união ou exclusão de alguns fragmentos para o encaixe no subgênero pretendido.

Para o *roguelike*, a escolha do maior fragmento pode fornecer um ambiente com dimensões adequadas, enquanto as subáreas podem ser desconsideradas e descartadas. Em contraponto, para o *metroidvania*, conectar os fragmentos pode fornecer uma estrutura mais adequada às características do subgênero. O autômato celular, por exemplo, quando usado com uma alta porcentagem de preenchimento inicial, permite a criação de

vários pequenos espaços que podem ser usados para o propósito descrito acima. A Figura 4.15 mostra uma execução do algoritmo que leva à formação de um conjunto de espaços isolados.

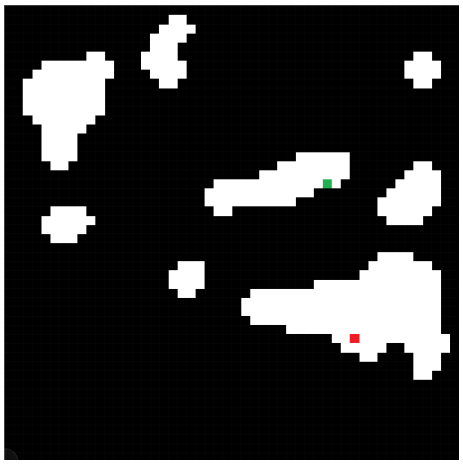
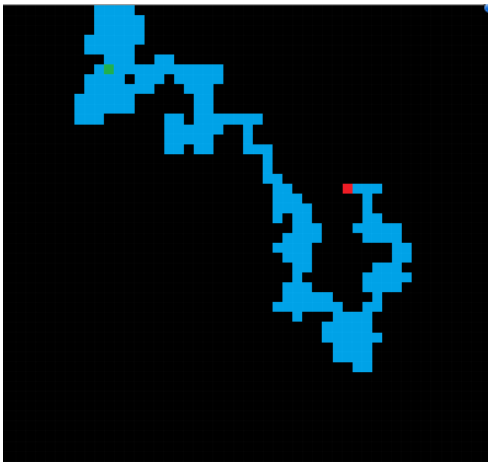


Figura 4.15: Resultado da execução do algoritmo de autômato celular com uma matriz 50x50 e 55% de preenchimento inicial. É possível ver a formação de ilhas.

4.6.4 Altura e largura do leiaute

Quando a diferença entre a **Altura** e a **Largura** é significativa, o leiaute gerado se aproxima de um corredor, como pode ser visto no mapa de baixa conectividade média da Figura 4.14a. Entretanto, em situações onde essa diferença é pequena, apesar de imaginarmos o resultado como uma grande sala com uma área centralizada, nem sempre é isso que vemos. Como visto na Figura 4.16a, podemos ter um leiaute heterogêneo contendo traços pertencentes a salas e a corredores, que nem sempre ocupam o espaço total do ambiente. Para isso, podemos usar em conjunto outras métricas para termos uma visão mais clara da estrutura criada.

Para identificar essa falta de equilíbrio e ter uma visão melhor daquilo que foi gerado e se atende às expectativas, podemos utilizar informações de outras métricas em conjunto. Vamos tomar como exemplo a Figura 4.16b. Tendo a quantidade total de *tiles* contidos no retângulo envolvente do mapa (área em vermelho) obtida através da métrica de **Altura** e **Largura**, e a quantidade total de *tiles* válidos gerados(espaços azuis), é possível criar uma relação de uso total de espaço que permita evidenciar que há uma grande parte não aproveitada (áreas amarelas). Poderíamos ainda usar a métrica de **Conectividade média** já discutida para determinar algumas estruturas em partes específicas da área útil



(a) O leiaute gerado tem zonas lineares, algumas pequenas salas e áreas não utilizadas.

(b) Para ilustrar a ideia as áreas foram demarcadas a mão para identificar a composição do mapa gerado.

Figura 4.16: Exemplo de leiaute gerado pelo algoritmo de passeio aleatório. É possível ver que apesar de ter altura e largura semelhantes, o espaço não é ocupado de forma homogênea.

do mapa (áreas verdes).

4.6.5 Relação de uso do espaço dos caminhos

A métrica de **Relação do uso de espaço** em cada algoritmo dá uma ideia geral da porcentagem de áreas que são críticas para a finalização do nível. Seu valor pode ser usado para indicar o mínimo que o jogador deve explorar do ambiente para chegar no objetivo. Uma baixa porcentagem indica um uso muito pobre do leiaute. Algumas medidas podem ser adotadas para aumentar esse valor e garantir que o jogador tenha que explorar uma área maior do ambiente antes de concluí-lo. A criação de obstáculos nos trajetos que levam ao objetivo, ou a inclusão de mecânicas como portais que transportam o jogador para áreas que naturalmente não fossem exploradas, são exemplos de medidas para aumentar a relação do uso de espaço do leiaute.

4.7 Compatibilidade dos algoritmos nos subgêneros

As análise feitas, evidenciam alguns resultados satisfatórios que podem ser utilizados por ambos os subgêneros considerados neste trabalho. Porém, nem todas os mapas gerados apresentaram as características desejadas, necessitando de adaptações para se encaixarem nos requisitos de cada subgênero. Alguns resultados gerados são bem variados, contendo tanto áreas lineares quanto grandes salões, o que poderia resultar em mapas interessantes.

Outros, contudo, são muito pequenos ou apresentam caminhos muito simples. Nesses casos, algumas métricas, como já descrito anteriormente, podem permitir filtrar os melhores resultados ou até mesmo manipular a geração.

Uma boa parcela dos *layouts* gerados não consegue em sua forma básica atender as necessidades do subgênero *metroidvania*. No algoritmo de passeio aleatório, algumas das saídas são muito lineares, contando com poucas ou apenas uma rota até o objetivo, o que não condiz com a característica necessária. Para contornar isso, uma possível solução é forçar a criação de alguns *hubs* e múltiplos caminhos. Usando alguma ferramenta que analisa o grau de **Conectividade média** passo a passo, é possível, em determinado momento da caminhada, forçar o algoritmo a escolher direções com maior potencial de conectividade para então estabelecer um novo hub. A partir desse *hub*, outras caminhadas podem ser iniciadas em diferentes direções, com novas possibilidades de formação de *hubs*. Desta forma, é possível a criação de mapas com varias áreas principais interligadas que podem ser analisadas para verificação de número de caminhos e quantidade de *hubs* gerados.

Para o subgênero *roguelike*, as restrições são baixas e não é difícil encaixar a maioria dos *layouts* gerados. Como já dito anteriormente, sua característica principal é o avanço rápido por conjuntos de salas de *layouts* suficientemente variados, fazendo bom uso da área disponível. Entretanto, alguns resultados podem não ser favoráveis para o subgênero. Por exemplo, o algoritmo Wang Tiles, devido à sua característica de trabalhar com paletas, é possível definir uma paleta que gere conjuntos de salas variados com a garantia de que o objetivo poderá ser atingido. Utilizando a métrica de relação do uso de espaço, é possível redefinir caminhos somente trocando um *tile* por outro com um padrão de encaixe que corte o caminho, aumentando assim a **Distancias mínimas** e **Distância provável** e, consequentemente, o aproveitamento do leiaute criado.

5 Conclusão

Neste trabalho foi apresentado e analisado um pequeno conjunto de algoritmos para geração procedural de *layouts* de mapas para jogos, com foco específico em jogos dos subgêneros *roguelike* e *metroidvania*. Foram estudados três algoritmos de preenchimento do espaço: passeio aleatório, autômato celular e Wang Tiles. Cada algoritmo foi implementado em sua forma original, buscando verificar a sua adequação à geração de layouts no contexto de cada subgênero. Um conjunto de métricas foi proposto para permitir a análise desses algoritmos. Um conjunto de experimentos foi conduzido, e os resultados comparativos dos três algoritmos estudados foram apresentados.

Ao fim do trabalho, foi possível perceber como cada métrica apresentada varia de acordo com as características de cada algoritmo. Também foram discutidos alguns exemplos de como cada métrica pode ser usada de forma criativa para, através de medições e regras de restrições, obter dados sobre um leiaute e adaptá-lo. Como discutido no capítulo anterior, considerando os subgêneros *roguelike* e *metroidvania* como nossos objetivos, alguns dos algoritmos analisados em suas versões mais básicas podem não atender a todos os requisitos necessários para cada geração do leiaute. Nesse caso, as alterações discutidas na Seção 4.7 são algumas possibilidades para adaptação dos algoritmos as características presentes nos subgêneros estudados.

Como trabalhos futuros, pode-se citar a criação de uma ferramenta que permita avaliar a saída do algoritmo durante a geração, usando as métricas propostas. Isso permitiria validar o leiaute a cada geração com base em requisitos definidos inicialmente. Também seria interessante utilizar a métricas de **Conectividade média** para durante a geração do mapa forçar o algoritmo a formar um novo *hub* e partir desse *hub*, iniciar outras caminhadas em diferentes direções, cada uma podendo formar novos *hubs* como descrito na Seção 4.7. Desta forma, é possível a criação de mapas com varias áreas principais interligadas que podem ser analisadas para verificação de número de rotas e quantidade de *hubs* gerados. Além disso, outras métricas podem ser propostas para fornecer um melhor panorama dos resultados desses algoritmos no contexto dos subgêneros abordados neste

trabalho. Da mesma forma, outros algoritmos podem ser avaliados utilizando as métricas aqui apresentadas. Por fim, outros elementos poderiam ser considerados na avaliação dos algoritmos, além do leiaute do mapa. Isso permitiria mensurar outros parâmetros como grau de dificuldade, distribuição de conteúdo, entre outros.

Bibliografia

BARON, J. R. Procedural dungeon generation analysis and adaptation. In: *Proceedings of the SouthEast Conference*. New York, NY, USA: ACM, 2017. (ACM SE '17), p. 168–171. ISBN 978-1-4503-5024-2. Disponível em: ⟨http://doi.acm.org/10.1145/3077286.3077566⟩.

BITKID, I. Chasm. 2018. Disponível em: ⟨http://www.chasmgame.com/⟩.

CAMPOS, H. Controlando o caos. 2016. Disponível em: ⟨https://www.indiegame.com.br/geracao-procedural/⟩.

CRUZ, G. J. Estimativa da qualidade de mapas procedurais para jogos do gênero roguelike. 2014.

DUARTE, P. M. *Geração procedural de cenários orientada a objetivos*. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Norte, 2012.

FORSYTH, W. Globalized random procedural content for dungeon generation. *J. Comput. Sci. Coll.*, Consortium for Computing Sciences in Colleges, USA, v. 32, n. 2, p. 192–201, dez. 2016. ISSN 1937-4771. Disponível em: ⟨http://dl.acm.org/citation.cfm?id=3015063.3015093⟩.

INFERNO, J.; CLAY, M.; ELAARAG, H. Hierarchical dungeon procedural generation and optimal path finding based on user input. *J. Comput. Sci. Coll.*, Consortium for Computing Sciences in Colleges, USA, v. 33, n. 1, p. 175–183, out. 2017. ISSN 1937-4771. Disponível em: ⟨http://dl.acm.org/citation.cfm?id=3144605.3144639⟩.

JANSSEN, F.; ARAUJO, R.; BAIÃO, F. Uma proposta de ontologia de gêneros e narrativas em jogos digitais para a game ontology project (gop). *RelaTe-DIA*, v. 12, n. 1, 2019.

KONAMI. Castlevania: Symphony of the night. 1997.

LUCCHESE, F.; RIBEIRO, B. Conceituação de jogos digitais. *Sao Paulo*, 2009.

MCMILLEN, F. H. E. The binding of isaac. 2011.

SHORT, T. X.; ADAMS, T. *Procedural Generation in Game Design*. [S.l.]: AK Peters/-CRC Press, 2017.

SMITH, A. M.; MATEAS, M. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games*, v. 3, n. 3, p. 187–200, Sep. 2011. ISSN 1943-068X.

SMITH, T.; PADGET, J.; VIDLER, A. Graph-based generation of action-adventure dungeon levels using answer set programming. In: *Proceedings of the 13th International Conference on the Foundations of Digital Games*. New York, NY, USA: ACM, 2018. (FDG '18), p. 52:1–52:10. ISBN 978-1-4503-6571-0. Disponível em: ⟨http://doi.acm.org/10.1145/3235765.3235817⟩.

STALNAKER, T. W. Procedural generation of metroidvania style levels (thesis). 2020.

TEAMCHERRY. Hollow knight. 2017. Disponível em: ⟨https://hollowknight.com/⟩.

TOY, G. W. M. Rogue: Exploring the dungeons of doom. 1980.

VALTCHANOV, V.; BROWN, J. A. Evolving dungeon crawler levels with relative placement. In: *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*. New York, NY, USA: ACM, 2012. (C3S2E '12), p. 27–35. ISBN 978-1-4503-1084-0. Disponível em: ⟨http://doi.acm.org/10.1145/2347583.2347587⟩.

YU, D. Spelunky. 2008. Disponível em: ⟨https://spelunkyworld.com/⟩.