Geração procedural de mapas dungeon crawl baseada em gramática de grafos para uso em jogos roguelike*

Renã Ferreira de Souza[1], Braz Araujo da Silva Junior[1], Luciana Foss[1], Gerson Geraldo Homrich Cavalheiro[1], Simone André da Costa Cavalheiro[1]

[1]Programa de Pós-Graduação em Computação
Universidade Federal de Pelotas
Pelotas – RS – Brazil

{rfsouza,badsjunior,lfoss,gerson.cavalheiro,simone.costa}@inf.ufpel.edu.br

Abstract. *This paper features the conception and the implementation of a procedural level generation engine, based in graph grammars, for roguelike games. The featured implementation explores the multithreaded programming over multiprocessed architectures, enabling the use of this tool in execution time, considering the high computational cost due to the use of graph grammars. The obtained results were very positive, both regarding performance gain and success of map generation, enabling the use of this dungeon crawl level generation technique in execution time. These results are illustrated by the introduction of the instance of a game and the discussion about the performance of the concurrent execution in a computer with gaming specifications.*

Resumo. *Este artigo apresenta a concepção e a implementação de um motor de geração procedural de mapas, baseado em gramáticas de grafos, para jogos do tipo* roguelike*. A implementação realizada explora a programação multitarefa sobre arquiteturas multiprocessadas, viabilizando o uso desta ferramenta em tempo de execução, tendo em vista o alto custo computacional decorrente do uso de gramática de grafos. Os resultados obtidos foram muito positivos, tanto no que se refere ao ganho de desempenho quanto no sucesso em geração de mapas, viabilizando o uso dessa técnica de geração de mapas* dungeon crawl *em tempo de execução. Estes resultados são ilustrados com a apresentação da instanciação de um jogo e de uma discussão sobre o desempenho da execução paralela em um computador com configuração* gamer.

1. Introdução

Geração procedural de conteúdo é uma técnica algorítmica que vem se popularizando em razão do aumento da demanda por criação de novos jogos (Shaker et al. 2016) para atendimento do público que cresce em número e em nível de exigência. No contexto de produção de jogos, as aplicações da geração procedural são diversas, desde a disposição e distribuição de elementos dentro de um jogo até a criação de fases e cenários inteiros. Deve ser ressaltado que o interesse no uso desta técnica não está restrito ao seu uso como

*O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001

ferramenta para substituir o trabalho manual, seu uso permite também criar experiências únicas para cada rodada, aumentando o valor de entretenimento e o interesse do jogador em repetir a experiência de jogo. Nesse artigo, esse valor será referido como **rejogabilidade**, que é o potencial que um jogo tem de ser jogado repetidamente, oferecendo uma experiência nova, após ser previamente completado.

Um estilo de jogo que explora de geração procedural para aumentar rejogabilidade é o *roguelike*. Esse estilo se caracteriza pela geração de mapas aleatórios a cada nível, mapas estes representando cenários do tipo *dungeon crawl*, os quais são formados por masmorras (*dungeons*) e/ou labirintos infestados por monstros e armadilhas no qual o personagem deve percorrer. Muitos *roguelikes*, como os títulos Spelunky e Darkest Dungeon, foram criados para terem rejogabilidade infinita, o que significa que tentam entregar novas experiências independente de quantas vezes forem jogados.

Para que os geradores procedurais serem aplicados em jogos duas características são desejáveis. A primeira é que o gerador ofereça recursos de controle sobre a aleatoriedade do que é gerado, permitindo, por exemplo, explicitar níveis de dificuldades em diferentes fases. A segunda é relacionada ao desempenho, sendo desejável obter resultados complexos em um intervalo de tempo que não afete a qualidade de experiência do jogador.

Em relação aos requisitos de controle, trabalhos já foram publicados validando a abordagem de geração procedural baseada em gramática de grafos para a geração de níveis *dungeon crawl* (Adams 2002)(Dormans and Bakkes 2011). Demonstrando a capacidade desta abordagem para geração de níveis interessantes. Esse tipo de gramática é apropriado para a geração de cenários *dungeon crawl*, pois estes, são essencialme, onde cada nó representa uma sala e cada aresta representa uma adjacencia entre duas salas. A GG também permite descrever a geração procedural empregando **regras de transformação**, possibilitando criar cenários complexos de forma intuitiva e com grande poder de expressão.

Porém, preocupações foram levantadas em relação ao desempenho dessa abordagem devido ao custo computacional elevado da avaliação das gramáticas de grafos. O objetivo desse trabalho é desenvolver um gerador procedural baseado em gramática de grafos que atenda essas questões de desempenho pelo uso de estratégias de programação concorrente.

O custo das gramáticas de grafos reside, sobretudo, no processo em que deve se encontrar a ocorrência de um **grafo padrão** em um **grafo hospedeiro** (Adams 2002). Esse problema, denominado Problema do Isomorfismo de Subgrafos (PIS) (Zampelli et al. 2007), é reconhecidamente NP-Completo e as soluções existentes para realizar essa avaliação de grafos possuem complexidade exponencial. Embora isso ofereça um obstáculo, o impacto da complexidade desse algoritmo pode ser amenizado ao se utilizar do fato de que o PIS, e a avaliação das GG em si, possui uma natureza concorrente. A ferramenta proposta nesse artigo explora essa concorrência realizando uma implementação com recursos *multithreading* (OpenMP) sobre arquiteturas multiprocessadas.

O uso de programação concorrente vem com outra vantagem, dada pela inserção de indeterminismo ao processo de geração. Indeterminismo está altamente conectado com

programação concorrente (Karp and Miller 1969), pois o entrelaçamento da execução das instruções que compõem os diferentes fluxos de execução não pode ser definido *a priori*. Em GG, a sequência em que regras são avaliadas e aplicadas pode influenciar diretamente no grafo resultante. Sendo a sequência de avaliação indeterminada, cada avaliação de uma mesma GG gera um grafo diferente. Esse indeterminismo é desejado, pois o esperado é que o resultado de uma geração procedural seja algo aleatório.

Neste artigo é apresentado um esquema da implementação do gerador criado integrado a um jogo, bem como características do algoritmo utilizado e sua implementação concorrente. A validação é apresentada na forma de estudo de casos e uma discussão sobre o desempenho da versão concorrente.

O restante deste artigo está organizado como segue. Na Seção 2 são apresentados trabalhos relacionados ao proposto. Na Seção 3 são apresentados conceitos de GG úteis a compreensão da proposta, bem como ferramentas que realizam sua avaliação. Na Seção 4 é descrita a realização do trabalho, apresentando a técnica utilizada de geração procedural concorrente baseada em GG. Na Seção 5 são relatados os resultados obtidos e na Seção 6 é apresentada uma conclusão.

2. Trabalhos relacionados

A primeira utilização de GG para geração procedural de mapas *dungeon crawl* encontrada tem registro em 2002 no trabalho de Adams (Adams 2002). Em seu trabalho, Adams apresenta um gerador procedural de mapas *dungeon crawl* construído sobre uma ferramenta de GG genérica, não exclusiva para geração de mapas. Esta ferramenta proporciona controle sobre os mapas gerados a partir de parâmetros de geração e regras de produção que atendam os parâmetros.

Uma das questões levantadas por Adams está relacionada à eficiência do método de geração procedural: o trabalho alega que o alto custo computacional é um problema a ser considerado. Devido a isso, a implementação proposta não prevê geração dos conteúdos durante o jogo, mas sim limitada a etapa de projeto do jogo, o que inviabiliza o uso da ferramenta em jogos *roguelike*, que necessitam de geração em tempo de execução.

O trabalho de Adams é também limitado pela natureza *ad-hoc* e *hard-coded* das regras e principalmente parâmetros do gerador. Significando que para a generalização da ferramenta para jogos além do apresentado seria necessário a criação de novos parâmetros e regras que resolvem tais parâmetros, o que exige muito tempo e esforço.

Outro trabalho (Dormans and Bakkes 2011), publicado por Dormans, apresenta o uso de gramática de grafos para a criação de **missões**, que são ações que o jogador deve executar em uma ordem específica representadas como grafos direcionados. E a partir do grafo gerado, gera um mapa utilizando **gramáticas geométricas**, que são outro tipo de gramática generativa utilizadas para gerar formas geométricas. Dormans não utiliza parâmetros e proporciona controle sobre o que é gerado apenas a partir das regras de transformações das gramáticas. Embora Dormans não utilize parâmetros, o custo de generalização da técnica utilizada também é muito alto.

O motor desenvolvido nesse trabalho é similar ao apresentado por Adams, e busca resolver apenas uma das questões levantadas, deixando a questão da generalização da

ferramenta para um trabalho futuro. Para atender a questão de desempenho, a ferramenta desenvolvida nesse trabalho explora a concorrência natural do problema em uma implementação paralela para obter benefícios tanto do poder de processamento paralelo de arquiteturas multiprocessadas quanto do potencial de indeterminismo inerente a sua execução. Como resultado, pretende-se reduzir o tempo necessário para a geração de cada mapa, permitindo a geração de mapas em tempo de execução do jogo e viabilizando o uso do método utilizado para o uso em jogos *roguelike*.

3. Gramática de grafos

GG é uma generalização das gramáticas de Chomsky, substituindo palavras por *grafos*. Uma GG utiliza regras de produção para reescrever um grafo, assim, criando um novo grafo. O glossário a seguir introduz os componentes e funcionalidades básicas de uma GG.

Derivação $G_0 \overset{U}{\Longrightarrow} G_k$ Expressa a geração do grafo G_k a partir de múltiplas aplicações de regras pertencentes a um conjunto de regras U sobre o grafo G_0.

Grafo G Um conjunto de vértices e arestas podendo ou não conter $labels$. Uma aresta possui um vértice origem e um vértice destino. Uma $label$ é um elemento pertencente a um conjunto de símbolos que é atribuído aos vértices e arestas de um grafo para que possam ser distinguidos.

Match m Expressa o mapeamento m do grafo G_l de uma regra r em um subgrafo G'(isomorfo a G_l) de um grafo G.

Passo de derivação $G_i \overset{r}{\longrightarrow} G_{i+1}$ Expressa a geração do grafo G_{i+1} a partir de uma única aplicação da regra r sobre o grafo G_i.

Regra de produção Expressa uma regra de produção. Uma regra de produção é composta por dois grafos, o lado esquerdo e o lado direito onde G_l é o lado esquerdo e G_r é o lado direito, e um mapeamento entre os grafos G_l e G_r (homomorfismo de grafos). O lado esquerdo de uma regra de produção é a condição de aplicação da regra, ou seja, é o grafo que deve ser encontrado no **grafo hospedeiro** para que a regra possa ser aplicada. O lado direito de uma regra de produção é o grafo que será **embutido** no lugar do sub-grafo do **grafo hospedeiro** isomorfo a G_l.

4. Projeto e Implementação

Esta seção apresenta o projeto e a implementação de um motor de execução para jogos baseados em geração procedural de conteúdo. A primeira seção apresenta o esquema geral de operação da ferramenta proposta. As seções seguintes detalham a implementação realizada.

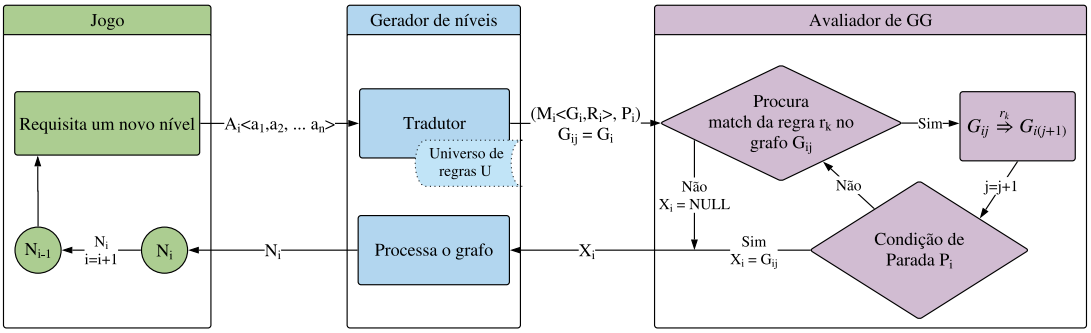


Figura 1. Motor de execução para a ferramenta de geração procedural

4.1. Visão Geral

A Figura 1 representa o esquema de execução de um jogo construído sobre este motor. O sistema representado é composto por três módulos, o **Jogo**, o **Gerador de Níveis** e o **Avaliador de GG**. Os dois últimos compõem o chamado **Motor de Execução**.

O módulo **Jogo** é composto por uma sequência de níveis N_i, com $1 \leq i \leq n$, proceduralmente gerados. A geração destes níveis inicia pela requisição de um novo nível à interface de geração de níveis. Esta requisição é parametrizada por uma tupla $A_i < a_1, a_2, \ldots a_n >$. Cada elemento desta tupla descreve um parâmetro a ser considerado na geração do nível, como dificuldade, número de salas, número de inimigos. Para cada nível i a tupla A_i é pré-concebida no momento do projeto do jogo.

Ao acionar o motor de execução do jogo, o módulo Gerador de Níveis é responsável pela integração com o módulo de Avaliação de GG. Este módulo é pré-configurado com o conjunto de regras de transformação de grafos responsável pela lógica do jogo. Este conjunto de regras, na figura, é representado pelo **Universo de Regras** U. A saber, a alteração do Universo de Regras implica em alterar o jogo implementado. A ativação do módulo Gerador de Níveis resulta na geração do nível N_i. A execução nesse módulo inicia pela tradução dos parâmetros A_i em uma condição de parada P_i e uma GG M_i, descrita por uma tupla $M_i =< G_i, R_i >$, onde G_i é o **grafo hospedeiro** e R_i é um conjunto de regras, tal que $R_i \subseteq U$.

A GG M_i é então submetida, juntamente a condição de parada P_i ao módulo Avaliador de GG. Este módulo tem por objetivo retornar um grafo X_i representado o nível gerado. A execução nesse módulo inicia com a busca de uma match da regra r_k no **grafo hospedeiro** $G_{i,j}$, tal que $r_k \in R_i$ e $G_i \overset{r_k}{\Longrightarrow} G_{i,j}$. Para cada match encontrada, então $j = j + 1$, tal que $G_{i,j-1} \overset{r_k}{\longrightarrow} G_{i,j}$, ou seja, o grafo $G_{i,j}$ é o grafo produzido pela derivação da regra r_k. No caso, indesejado, de não ser encontrada uma match para uma regra qualquer, isto significa que U foi mal projetado e que o jogo, ou nível, não pode ser construído.

O procedimento executado pelo módulo Avaliador de GG é iterativo, enquanto os parâmetros descritos em A_i não forem todos atendidos.

O retorno X_i produzido pelo Avaliador de GG é traduzido, de sua forma de grafo, pelo módulo Gerador de Níveis para uma estrutura de dados utilizada pelo módulo Jogo. Neste ponto da execução é que os cenários pré-concebidos de cada sala são utilizados para montar o mapa. O nível N_i é então entregue ao módulo Jogo.

4.2. Avaliador de GG

Para a implementação do avaliador de GG é necessário a implementação de dois algoritmos principais. Sendo estes o algoritmo de isomorfismo de subgrafos e o algoritmo de aplicação de regras. O algoritmo de isomorfismo de subgrafos é o mais custoso, com complexidade exponencial, e opera apenas fazendo leituras na memória, sendo este o ponto onde a concorrência é explorada. O algoritmo de aplicação de regras possui menor custo e realiza escritas à estrutura de dados que descreve o grafo, sendo implementado de forma sequencial.

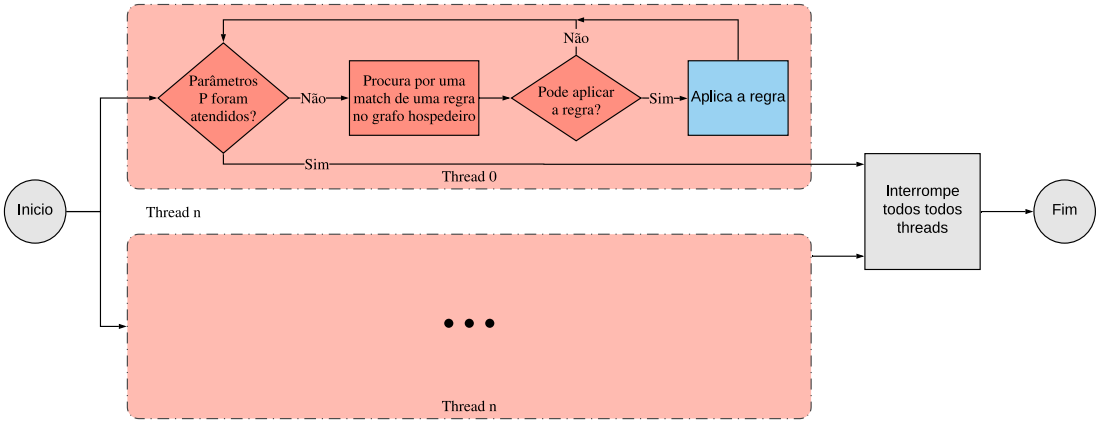


Figura 2. Esquema do paralelismo

A Figura 2 apresenta um esquema ilustrando a maneira com que o gerador explora a programação paralela. Na figura, os processos em vermelho são executados em paralelo e o processo em azul representa um processo executado em um fluxo de execução paralelo mas protegido por um *mutex*. O primeiro passo é decidir se regras precisam ser aplicadas testando se os parâmetros de geração já foram atendidos. Se os parâmetros foram atendidos, todos os threads são interrompidos o processo encerra. Se não, diversas buscas por $matches$ são iniciadas em paralelo. Conforme as buscas terminam, cada fluxo de execução decide se a regra pode ser aplicada. Esse teste leva em consideração primeiramente se uma $match$ foi encontrada e também se nenhum elemento do lado esquerdo da regra já foi excluído. Se possível, a regra é aplicada, adicionando elementos novos ao grafo hospedeiro e marcando elementos a serem excluídos pelo coletor de lixo. O programa então retrocede ao primeiro passo.

A busca por $matches$ é um algoritmo que soluciona o PIS. Existem diversos algoritmos para este propósito. A ideia comum por trás da maioria destes algoritmos é similar ao algoritmo básico proposto por Ullmann (Ullmann 1976) que utiliza matrizes de permutação e é baseado em um procedimento de *back-tracking* com característica *look-ahead* para reduzir o espaço de busca. Em 2008, uma solução para o PIS explorando programação paralela que apenas faz uma implementação paralela da solução de Ullmann foi apresentada por Sharma em (Sharma et al. 2008). A solução apresentada neste trabalho também se baseia no procedimento *back-tracking* e é diferente no sentido

que utiliza de classes, do paradigma de programação orientada a objetos, para representar os grafos ao contrário de matrizes de adjacência.

A técnica utilizada nesse trabalho é também baseada no procedimento *back-tracking* e se assemelha ao trabalho de Sharma ao utilizar classes para representar os grafos. A técnica utilizada formula o mapeamento incrementalmente utilizando uma busca por profundidade, mapeando os elementos conforme os encontra e quando um elemento não se encaixa no mapeamento, a busca por profundidade **retrocede**(*back-tracking*).

Esta técnica foi implementada prevendo a eficiência da execução e as necessidades da geração de níveis. Levando em consideração que mapas de jogos podem ser representados sempre por grafos conexos, o método para solucionar o PIS implementado nesse artigo prevê encontrar apenas ocorrências de grafos conexos, isso significa que o lado esquerdo das regras de produção precisa ser sempre um grafo conexo. Sendo assim, o algoritmo implementado se resume a uma busca por profundidade recursiva que ocorre em ambos grafos simultaneamente.

4.3. Gerador de níveis

O gerador de níveis implementado recebe como parâmetros a quantidade de cada tipo de sala a ser gerada, que são traduzidos em condições de aplicações para cada regra. Por exemplo, se uma regra cria salas, e o número máximo de salas é de X salas, quando o número de salas for maior que X, essa regra não será mais aplicada.

A discussão sobre a tradução do grafo gerado em um nível do jogo foge ao escopo deste artigo, pois refere-se a implementação de um jogo utilizando a ferramenta apresentada.

5. Resultados

Nesta seção é apresentado, como caso de estudo, a construção do jogo **Carcere Exire**. Este jogo, concebido para demonstração das facilidades da ferramenta proposta, tem um aventureiro capturado em um calabouço de um castelo como personagem controlado pelo jogador. O castelo é comandado por um **chefão**, o qual representa uma entidade multidimensional que pode possuir várias instâncias em cada nível, e possui salas, as quais podem conter **inimigos** a serem combatidos ou **baús** que contém tesouros que devem ser coletados. O sucesso em uma fase é conseguir atingir a saída do castelo, sendo a pontuação dada pelo número de inimigos abatidos e baús coletados.

5.1. Regras para construção dos níveis

Antes de gerar um mapa, é necessário definir quais as regras que serão utilizadas para GG e também definir um grafo inicial a ser utilizado. Esta etapa corresponde a descrição do jogo. A linguagem de descrição destas regras não é apresentada neste artigo, sendo apenas descritas as regras definidas.

Para o grafo inicial foi utilizado um grafo com um único vértice com $label$ igual a “sala inicial”. A Figura 3 apresenta as regras definidas para o jogo, em número de nove. Nessa figura, os círculos representam vértices e as linhas representam arestas. Os vértices preenchidos em branco expressam vértices que estão no lado esquerdo da regra de produção e também estão presentes no lado direito. Os vértices preenchidos em verde

representam vértices que estão no lado direito da regra mas não estão presente no lado esquerdo. O mesmo vale para as arestas. Completando o padrão de cores que será utilizado na sequência, um vértice ou aresta em vermelho é um elemento presente somente no lado esquerdo de uma regra. As $labels$ dos elementos do grafo também são expressas na figura, sendo elas o texto dentro dos vértices e sobre as arestas.

As regras descrevem a construção do mapa e do cenário das salas. As regras 1 e 2 permitem criar salas adjacentes a uma outra sala. A regra 1 insere uma sala vazia adjacente a sala inicial e a regra 2 insere uma sala vazia adjacente a qualquer outra sala. A aresta com $label$ "vazia" é uma aresta de controle, sendo utilizada para representar o conteúdo da sala, conforme apresentado nas regras descritas na sequência nas regras 7, 8 e 9.

A regra 3 cria uma sala com $label$ "saída", que representa o objetivo final do jogador no mapa.

As regras 4 e 5 apresentam duas variações de salas, sendo elas o corredor e a cela. O corredor representa uma sala que interliga duas salas comuns. A cela representa uma sala sempre adjacente a um corredor, correspondendo a um ponto sem saída. A regra 6 introduz ciclos ao mapa, ou seja, permite que existam mais de uma rota de fuga.

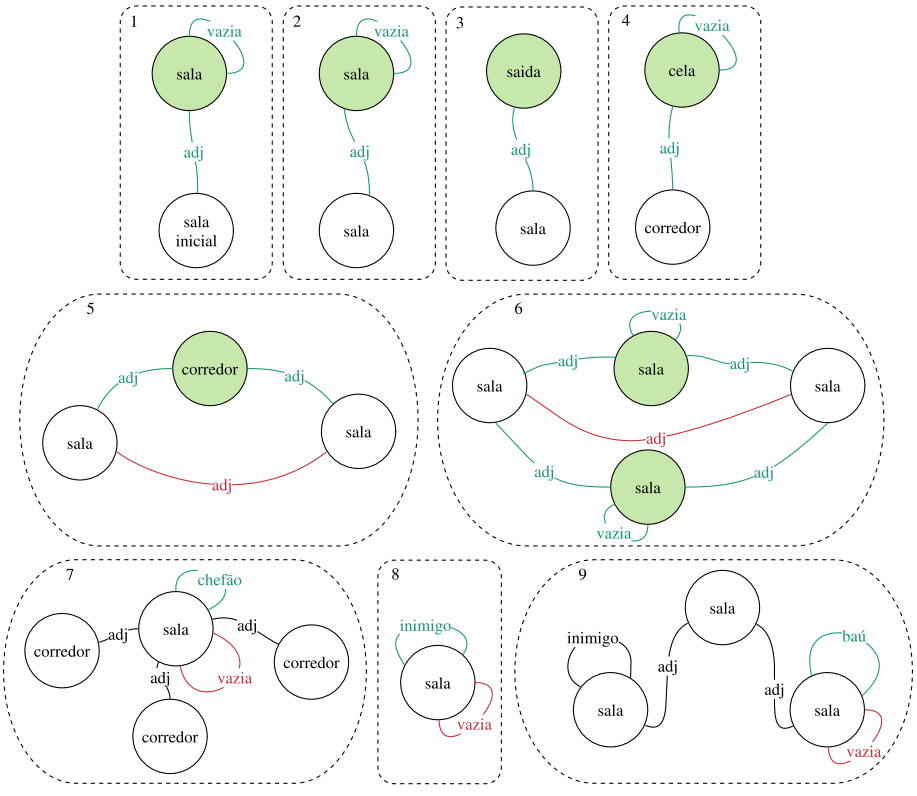


Figura 3. Regras de produção para a geração de um mapa

Por fim, as regras 7, 8 e 9 atribuem o conteúdo de cada sala. As arestas de controle são aplicadas neste ponto. Para fazer a atribuição do conteúdo de uma sala é necessário haver uma aresta com $label$ "vazia" incidente no vértice, e quando é atribuído o conteúdo a sala, essa aresta é removida. A primeira regra adiciona um personagem, denominado **chefão**, em uma sala adjacente a três corredores. A segunda regra acrescenta um **inimigo**

em uma sala vazia. A terceira regra acrescenta um **baú** em uma sala vazia que esteja adjacente a uma sala adjacente a outra sala que contenha um **inimigo**, isso faz com que as recompensas sejam sempre encontradas a uma distância pequena de um **inimigo**.

5.2. Geração de Mapas

De posse das regras, o próximo passo é lançar o jogo, ativando o motor de geração especificado para criação dos mapas para cada nível. As figuras 4 e 5 ilustram dois mapas produzidos pelo gerador de mapas utilizando as regras apresentadas. Nestes dois mapas, os parâmetros de entrada informam a quantidade desejada de salas, baús, inimigos corredores, celas e chefões a serem instanciado no nível. Nos mapas gerados, o vértice preenchido de azul representa a sala inicial. Os vértices rosa representam os corredores. O vértice cinza representa uma cela. O vértice amarelo representa a saída. As arestas representam as adjacências entre as salas. Os ícones presentes nos vértices representam o conteúdo de cada sala (**baús**, **inimigos** e **chefão**).

Os exemplos de mapas apresentados, embora simples, permitem visualizar o potencial de geração de mapas da ferramenta proposta. Observe-se que a parametrização da geração destes dois mapas diferiu apenas por informar um diferente valor para o número de salas, inimigos e baús. Este aspecto realça o poder da ferramenta em quesito de criação de cenários distintos. Ainda assim, é possível, a exemplo do que foi apresentado em (Adams 2002), realizar uma avaliação da qualidade dos níveis gerados. No entanto, a exemplo da linguagem de descrição das regras, a discussão deste aspecto foge ao escopo deste trabalho.

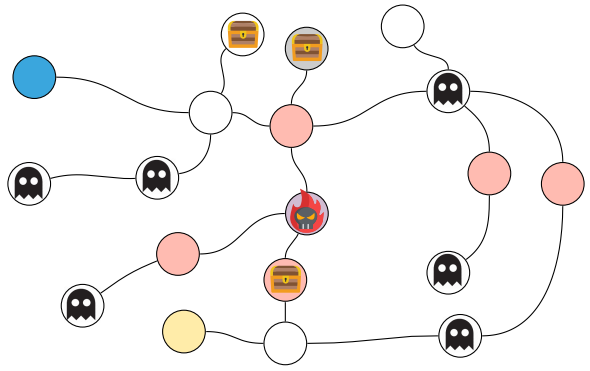


Figura 4. Mapa gerado pelo gerador de mapas

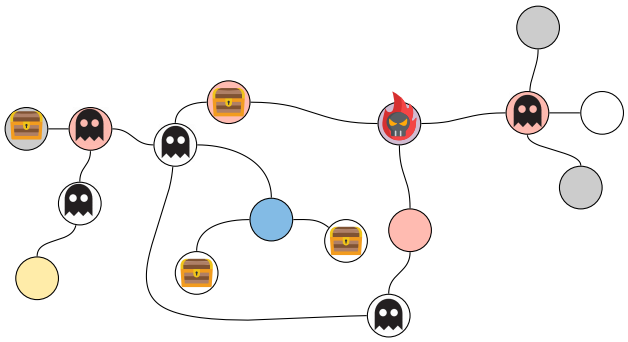


Figura 5. Outro mapa gerado pelo gerador de mapas

Tabela 1. Variação do tempo de execução na geração de mapas em função do suporte paralelo

	sequencial	4 threads	8 threads	16 threads
50 salas				
Tempo médio (s)	18,27	6,67	3,48	2,32
Desvio padrão	13,40	5,27	2,73	1,26
Speedup	-	2,73	5,25	7,875
100 salas				
Tempo médio (s)	36,22	17,92	10,89	6,75
Desvio padrão	26,64	15,05	9,39	5,82
Speedup	-	2,01	3,32	6.22

5.3. Comparativo de desempenho

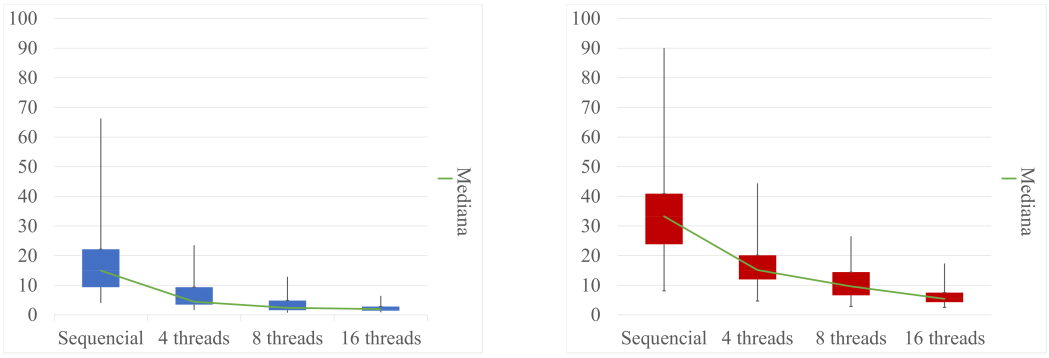
Para obter a informação sobre desempenho da implementação paralela realizada, foram executados estudos de caso em uma máquina utilizando o sistema operacional *Ubuntu 14.04* e equipada com o processador *AMD Opteron(TM) Processor 6276* dotado de 16 núcleos de processamento, sendo oito físicos e oito lógicos e 16 GB. Na realidade de mercado atual, tal configuração está presente em boa parte dos computadores classificados como *gamer*, sendo estes populares entre os jogadores assíduos. Deve-se atentar que, em função da natureza indeterminista da aplicação, o tempo de execução varia também em função dos caminhos percorridos pela aplicação das regras. Ou seja, duas execuções podem ter tempo de execução completamente diferentes.

Para coleta de tempos de execução, foram utilizadas as mesmas regras definidas anteriormente para o jogo **Carcere Exire**. Cada caso de teste foi executado 30 vezes para obtenção dos resultados. A Tabela 1 apresenta o tempo médio, apresentado em segundos) e o desvio padrão observado para os tempos de execução considerando a geração de um mapa com 50 e 100 salas. O mapa de 50 salas é complementado por 15 baús, 20 inimigos, 10 corredores, 25 celas e 5 salas de chefões e o mapa de 100 salas por 30 baús, 40 inimigos, 20 corredores, 50 celas e 10 salas de chefões. Na Figura 6 são apresentados os gráficos de caixa detalhando a distribuição das amostras coletadas.

Observando os resultados dos testes, é possível verificar que uma implementação sequencial do gerador é viável apenas para geração de mapas com pequeno número de salas ou para avaliação de regras muito simples. Ainda assim, mesmo com pequeno número de salas (50 salas), a implementação sequencial se demonstrou instável. Isto fica claro pelo elevado valor apresentado pelo desvio padrão, caracterizando a não aderência dos tempos coletados a uma distribuição normal. A execução paralela da avaliação de GG permite a geração de mapas muito maiores, com regras mais complexas e com um tempo de execução com menor variação.

6. Conclusão

Neste trabalho foi apresentado uma ferramenta para geração procedural de mapas do tipo *dungeon crawl* para jogos *roguelike* baseada em GG que explora paralelismo. Os resultados produzidos se mostraram promissores. A implementação paralela da ferramenta apresentou um ganho de desempenho grande em relação a implementação sequencial,



(a) 50 salas **(b) 100 salas**

Figura 6. Análise da variação do tempo de execução na geração de mapas

suficiente para permitir a geração de mapas em tempo de execução do jogo, requisito necessário para os jogos *roguelike*. Como trabalho futuro, um novo artigo deverá apresentar a linguagem definida para especificação das regras e parâmetros de construção dos mapas. O objetivo final a ser atingido é a construção de uma ferramenta de geração de mapas *dungeon crawl* que possa ser facilmente generalizada para diferentes jogos.

Referências

[Adams 2002] Adams, D. (2002). Automatic generation of dungeons for computer games. Bachelor thesis, University of Sheffield, UK.

[Dormans and Bakkes 2011] Dormans, J. and Bakkes, S. (2011). Generating missions and spaces for adaptable play experiences. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):216–228.

[Karp and Miller 1969] Karp, R. M. and Miller, R. E. (1969). Parallel program schemata. *J. Comput. Syst. Sci.*, 3:147–195.

[Shaker et al. 2016] Shaker, N., Togelius, J., and Nelson, M. J. (2016). Procedural content generation in games. In *Computational Synthesis and Creative Systems*.

[Sharma et al. 2008] Sharma, A., Bahir, S., Narsale, S., and Tambe, U. (2008). A parallel algorithm for finding subgraph isomorphism.

[Ullmann 1976] Ullmann, J. R. (1976). An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42.

[Zampelli et al. 2007] Zampelli, S., Deville, Y., Solnon, C., Sorlin, S., and Dupont, P. (2007). Filtering for subgraph isomorphism. In *CP*.