Geração de Mapas para Jogo em 2D Utilizando Problema de Satisfação de Restrições

Camile Reis de Sousa, Gabriel Vilas Novas, Gustavo Henrique Correia, João Henrique Pedrosa de Souza, Marcus Vinícius Araújo

[1]Universidade Federal de Ouro Preto (UFOP), Ouro Preto - MG

Abstract. *The creation of content in digital games is a fundamental aspect of their functionality and dynamics. However, generating random content that is both valid and playable presents significant challenges. This work addresses the problem of creating maps for a 2D game, ensuring that each generated map complies with a set of rules. To this end, the problem was modeled as a Constraint Satisfaction Problem (CSP) and solved using a recursive backtracking algorithm. The project was implemented in Python with the Pygame library, incorporating domain constraints to prevent element overlap, maintain consistent challenges, and ensure playability. The results demonstrate that the use of domain constraints is effective in generating valid maps while preserving the diversity and complexity of the levels.*

Resumo. *A criação de conteúdo em jogos digitais é um detalhe fundamental para seu funcionamento e dinâmica. Contudo, gerar conteúdo aleatório que seja simultaneamente válido e jogável apresenta desafios significativos. Este trabalho aborda o problema da criação de mapas para um jogo em 2D, garantindo que cada mapa gerado obedeça a um conjunto de regras. Para tal, o problema foi modelado como um Problema de Satisfação de Restrições (PSR) e solucionado com um algoritmo de backtracking recursivo. O projeto foi implementado em Python, com a biblioteca Pygame, incorporando restrições de domínio para evitar sobreposição de elementos, manter desafios consistentes e assegurar a jogabilidade. Os resultados obtidos demonstram que o uso de restrições de domínio é eficaz para gerar mapas válidos, preservando a diversidade e a complexidade das fases.*

1. Introdução

A Geração Procedural de Conteúdo (GPC) é uma área da computação focada na criação de dados de forma algorítmica, em vez de manual. Na indústria de jogos eletrônicos, a GPC é amplamente utilizada para gerar níveis, texturas, narrativas e outros elementos, oferecendo uma solução escalável para a criação de novo conteúdo e aumentando significativamente o fator de rejogabilidade dos jogos.

O principal desafio da GPC, no entanto, não é a geração de aleatoriedade pura, mas a criação de conteúdo coerente, estruturado e, no caso de mapas e níveis, funcional. Um mapa para um jogo de exploração, por exemplo, deve ser solucionável, garantindo que o jogador possa sempre acessar áreas críticas para progredir. A geração de mapas que atendam a esses critérios de validade é um problema que se alinha bem com as técnicas de Inteligência Artificial.

Neste contexto, o objetivo deste trabalho é explorar a geração procedural de mapas como um Problema de Satisfação de Restrições (PSR), utilizando, como técnica de I.A., um algoritmo de backtracking. O objetivo foi projetar um sistema capaz de distribuir elementos como jogador, saída, inimigos, baús, armadilhas, e itens especiais, respeitando condições que garantam equilíbrio e jogabilidade.

Este documento está estruturado da seguinte forma: a Seção 2 apresenta a fundamentação teórica sobre PSR e backtracking. A Seção 3 detalha a metodologia de aplicação desses conceitos ao problema. A Seção 4 apresenta e discute os resultados obtidos. Por fim, a Seção 5 sumariza as conclusões do trabalho.

O código-fonte do projeto está disponível para visualização no repositório do GitHub (https://github.com/vilas000/tp_IA).

2. Fundamentação Teórica

Para resolver problemas em Inteligência Artificial, é comum transformar o cenário estudado em um modelo formal, para facilitar a aplicação de algoritmos eficientes e confiáveis. Uma das formas mais usadas para isso é modelar o problema como um Problema de Satisfação de Restrições (PSR). Essa abordagem ajuda a organizar o problema em variáveis, possíveis valores para essas variáveis (domínios) e as restrições impostas sobre as váriaveis que precisam ser atendidas.

Neste projeto, por exemplo, isso significa trabalhar com os itens a serem colocados no mapa como variáveis, o conjunto de posições disponíveis no mapa como domínios, e as restrições que garantem uma distribuição válida.

Para encontrar uma solução que respeite todas essas regras, foi escolhido o método de backtracking. Ele é amplamente empregado para resolver PSRs devido à sua capacidade de lidar com restrições difíceis e complexas, além de ser simples e direto.

Dessa forma, a escolha da modelagem via PSR e do algoritmo de backtracking deve-se à maneira que o problema pode ser melhor resolvido. O desafio de posicionar vários elementos em um espaço, respeitando regras como distância e acessibilidade, encaixa muito bem nessa abordagem. Diferentemente de métodos de aprendizado de máquina, que precisariam de muitos dados de mapas já prontos para aprender, o PSR permite definir as regras do que é considerado um "bom" mapa de forma clara e direta, garantindo corretude na maior parte das tentativas.

2.1. Problemas de Satisfação de Restrições (PSR)

Um Problema de Satisfação de Restrições é um modelo matemático para descrever problemas a partir de suas características lógicas. Um PSR pode ser definido formalmente por um conjunto de três componentes:

X: Um conjunto de variáveis X1, X2, ..., Xn.

D: Um conjunto de domínios D1, D2, ..., Dm, onde cada D é o conjunto de valores possíveis para a variável X.

C: Um conjunto de restrições C1, C2, ..., Ck, sendo que cada restrição C especifica as combinações de valores permitidos para um subconjunto de variáveis.

Uma solução para um PSR é uma atribuição completa e consistente de valores para todas as variáveis, de modo que todas as restrições sejam satisfeitas simultaneamente.

2.2. Algoritmo de Backtracking

O backtracking é um algoritmo de busca sistemático e de propósito geral para resolver PSRs. Ele opera com base em uma busca em profundidade (DFS) no espaço de estados das atribuições parciais. O processo pode ser descrito em quatro passos fundamentais:

1. Selecionar: Escolhe-se uma variável ainda não atribuída.

2. Tentar: Itera-se sobre os valores no domínio da variável selecionada. Para cada valor, verifica-se se a atribuição é consistente com as restrições já aplicadas às variáveis atribuídas.

3. Recursão: Se a atribuição for consistente, o algoritmo é chamado recursivamente para a próxima variável.

4. Recuar (Backtrack): Se a chamada recursiva retornar uma falha (indicando que nenhuma solução pôde ser encontrada a partir daquele ponto), ou se todos os valores do domínio de uma variável foram testados sem sucesso, a atribuição atual é desfeita, e o algoritmo ”recua” para a variável anterior para tentar um valor diferente (volta ao passo 2).

Essa abordagem garante que todo o espaço de busca seja explorado de forma sistemática, encontrando uma solução, se ela existir.

3. Metodologia

Todo o código que modela e soluciona o problema da geração de mapas aleatórios foi desenvolvido na linguagem Python, com o auxílio da biblioteca Pygame, para criação de uma interface gráfica e visualização do resultado. A metodologia adotada parte da modelagem do problema como um Problema de Satisfação de Restrições (PSR), que é, então, resolvido por um algoritmo de backtracking customizado. As seções a seguir detalham cada etapa desse processo.

3.1. Modelagem do Problema como PSR

O primeiro passo para a solução foi traduzir as regras do mapa em um modelo formal de PSR, definindo variáveis, domínios e restrições.

Variáveis: As variáveis do problema são os próprios itens que precisam ser alocados no mapa. Em vez de tratar cada célula como uma variável, definimos uma lista contendo todos os objetos a serem posicionados, como jogador, saída, inimigos e tesouros. Isso simplifica a busca, focando nos elementos-chave. A lista depende de inputs do usuário.

Domínios: O domínio para cada variável é o conjunto de todas as coordenadas (linha, coluna) que estão disponíveis no mapa (inicialmente, as células com o valor ’V’, que são vazias). Para itens como o jogador e a saída, o domínio é restrito às células da borda do mapa.

Restrições: As restrições são as regras que definem um mapa válido. Elas foram implementadas como funções Python que retornam True - se a regra é satisfeita - e False - caso contrário. As restrições foram divididas em duas categorias:

Restrições de Distribuição: São regras locais que verificam o posicionamento relativo dos itens. Por exemplo, a função $eh\_distribuicao\_valida$ garante que o espaçamento entre os itens respeite o que foi solicitado pelo usuário.

Restrições de Jogabilidade: Essa restrição garante que o mapa seja solucionável. A função $verificar\_caminhos\_criticos$ é chamada ao final do processo e utiliza um algoritmo de Busca em Largura (BFS) para confirmar que o jogador pode alcançar o baú que contém a chave e, posteriormente, a saída.

3.2. Implementação do Algoritmo de Backtracking

O problema foi modelado e resolvido com um algoritmo de backtracking para posicionar todos os elementos do tabuleiro (jogador, saída, chave, espada, inimigos, armadilhas e baús) de forma que respeitem restrições de posicionamento e garantam que o mapa seja jogável e vencível.

Primeiro, o código cria uma lista ordenada com todos os itens que precisam ser colocados no mapa. Essa ordem é proposital, pois alguns elementos dependem da localização de outros (por exemplo, a chave precisa estar em posição viável em relação ao jogador e à saída).

```
# Cria a lista de todos os itens que precisam ser posicionados.
        itens_para_colocar = [
            ('J', 'JOGADOR'), ('S', 'SAIDA'),
            ('B', 'CHAVE'), ('B', 'ESPADA')
        ]
        itens_para_colocar.extend([('B', f'BAU_{i}')
            for i in range(num_baus_extras)])
        itens_para_colocar.extend([('I', f'INIMIGO_{i}')
            for i in range(num_inimigos)])
        itens_para_colocar.extend([('T', f'ARMADILHA_{i}')
            for i in range(2)])
```

A função para resolver o backtracking percorre essa lista e, para cada item, tenta posicioná-lo em diferentes locais do tabuleiro. Para cada posição candidata, o código verifica se a célula está livre para receber o item e se respeita as regras de distanciamento e dispersão definidas na função que verifica se uma distribuição é válida.

Se a posição for válida, o item é colocado no mapa e a função chama a si mesma para posicionar o próximo elemento. Quando todos os itens forem colocados, chega-se ao caso base da recursão.

Nesse momento, a função que verifica os caminhos críticos faz uma validação final, utilizando busca em largura (BFS) para garantir que existe caminho do jogador até a chave e que partindo da chave, seja possivel chegar até a saída.

```
def verificar_caminhos_criticos(mapa, contexto):
    pos_jogador = contexto.get('JOGADOR')
```

```
    pos_chave = contexto.get('CHAVE')
    pos_saida = contexto.get('SAIDA')

    # Garante que o jogador pode chegar ate o bau com a chave.
    if not existe_caminho(mapa, pos_jogador, pos_chave):
        return False

    # Garante que, da posicao da chave, eh possivel chegar ate a saida.
    # Isso impede que a chave fique presa em uma area sem saida para o
    # resto do mapa.
    if not existe_caminho(mapa, pos_chave, pos_saida):
        return False

    # Se todos os caminhos criticos existem, a solucao eh valida.
    return True
```

A mecânica de backtracking entra em ação quando não há posição válida para o item atual ou quando a validação final falha. Nesses casos, o algoritmo desfaz o último posicionamento (retrocede) e tenta uma nova posição para o item anterior. Esse processo de tentativa e erro continua até que se encontre um mapa que atenda a todas as restrições ou até que todas as possibilidades sejam esgotadas.

3.3. Otimizações e Heurísticas Aplicadas

A implementação pura do backtracking pode ser lenta para espaços de busca grandes. Portanto, algumas otimizações e heurísticas foram aplicadas para tornar o gerador mais eficiente e robusto.

A verificação de caminho *verificar_caminhos_criticos*, por ser computacionalmente cara, foi movida para fora do loop recursivo. Ela é executada apenas uma vez, após o backtracking encontrar uma solução que já satisfaz todas as restrições de distribuição.

```
if sucesso_posicionamento and verificar_caminhos_criticos(mapa_potencial,
contexto_potencial):
    print("Mapa␣valido␣gerado␣e␣validado␣com␣sucesso!")
    return mapa_potencial, jogador, ...
```

Foi implementado um contador de passos que aborta a busca se um limite é atingido, permitindo que o sistema tente gerar um novo mapa aleatório diferente. Isso evita que o algoritmo entre em buscas excessivamente longas em mapas muito densos ou com restrições impossíveis.

```
estado_busca['passos'] += 1
if estado_busca['passos'] > estado_busca['limite_passos']:
    return False, None, None # Aborta a busca
```

4. Resultados

O código desenvolvido demonstrou sucesso na geração de mapas 2D para jogos, ao garantir que todo mapa gerado seja comprovadamente solucionável e satisfaça as restrições impostas. Além disso, o algoritmo se mostrou eficiente ao lidar com cenários

de geração complexos, evitando cálculos infinitos. Alguns experimentos foram realizados para avaliar quantitativamente o desempenho do algoritmo de backtracking em diferentes cenários. Outros experimentos foram feitos para analisar qualitativamente a variedade dos mapas gerados. As métricas utilizadas estão listadas a seguir, juntamente com suas respectivas observações. Todos os testes foram realizados em tabuleiro 14x14 (12x12 espaços válidos), com máximo de 20 tentativas para geração do mapa.

4.1. Tempo

O tempo para gerar um mapa válido depende do tamanho do mapa e da quantidade de itens a serem posicionados. Quanto mais baús, mais o algoritmo irá demorar para encontrar um arranjo que satisfaça as restrições impostas. Caso o número limite de tentativas predefinido seja extrapolado, o programa é automaticamente encerrado. A tabela abaixo mostra o tempo necessário para gerar o tabuleiro.

Quantidade baús	Quantidade inimigos	Tempo
15	15	0.0023s
18	18	0.0024s
21	21	0.5676s
24	24	não gerado

4.2. Custo Computacional

Para medir o custo computacional do algoritmo, foram utilizados dois parâmetros: os passos e a quantidade de tentativas. Os passos consistem na quantidade de vezes em que o programa tenta alocar os itens até encontrar uma configuração que satisfaça as restrições de distância entre eles. Já a tentativa consiste na verificação que é feita no mapa encontrado para averiguar se ele possui ao menos uma solução. Dependendo do valor definido pelo usuário, o programa pode gerar um mapa válido ou falhar, deixando explícito o custo computacional. Tabuleiros menores e com poucos itens são facilmente resolvidos, enquanto tabuleiros mais complexos podem ter alto custo computacional e não encontrar uma solução.

Quantidade baús	Quantidade inimigos	Passos na busca	Tentativas
15	15	37	1
18	18	43	1
21	21	49	6
24	24	não gerado	não gerado

5. Conclusão

Em suma, este trabalho se propôs a resolver um desafio de Geração Procedural de Conteúdo: criar mapas que fossem não apenas aleatórios, mas garantidamente jogáveis e coesos. Através da modelagem do problema como um Problema de Satisfação de Restrições (PSR) e da aplicação de um algoritmo de backtracking, foi possível traduzir regras subjetivas de design de jogo em restrições lógicas e, assim, buscar sistematicamente por soluções válidas.

A metodologia demonstrou que a separação das restrições em categorias, como as de distribuição local e as de caminho global,, foi uma decisão de design crucial. A otimização de executar a verificação de caminho, que é custosa, apenas ao final da busca recursiva mostrou-se fundamental para alcançar tempos de geração eficientes, como foi validado na seção de Resultados. Isso confirma que, mesmo para algoritmos de busca exaustiva como o backtracking, otimizações específicas ao domínio do problema são essenciais para a viabilidade prática.

Com isso, o projeto alcançou seu objetivo principal, entregando um sistema capaz de produzir uma variedade numerosa de mapas diferentes e com solução garantida, unindo a imprevisibilidade da geração procedural à robustez da lógica da Inteligência Artificial. Como limitação, observa-se que em cenários de altíssima densidade de itens, o desempenho do algoritmo pode decair - uma característica inerente à complexidade exponencial de problemas de busca.

Para trabalhos futuros, diversos aprimoramentos poderiam ser apresentados:

1. Otimização da Busca: A eficiência do algoritmo poderia ser aprofundada com a implementação de heurísticas mais avançadas para a ordenação de variáveis, como a de Minimum Remaining Values (MRV), que prioriza as decisões mais restritas e pode podar a árvore de busca mais cedo.
2. Dificuldade Dinâmica: O conjunto de restrições poderia ser expandido para permitir um ajuste dinâmico dos parâmetros conforme a dificuldade desejada. Por exemplo, mapas no modo ”difícil” poderiam ter restrições que forçam uma menor distância entre inimigos ou posicionam a chave mais longe da saída.
3. Riqueza de Conteúdo: O sistema poderia evoluir para controlar aspectos de design mais sutis, como a criação de padrões arquitetônicos (ex: salas temáticas, corredores longos) ou a introdução de novas mecânicas e tipos de elementos, aumentando ainda mais a experiência do jogador.