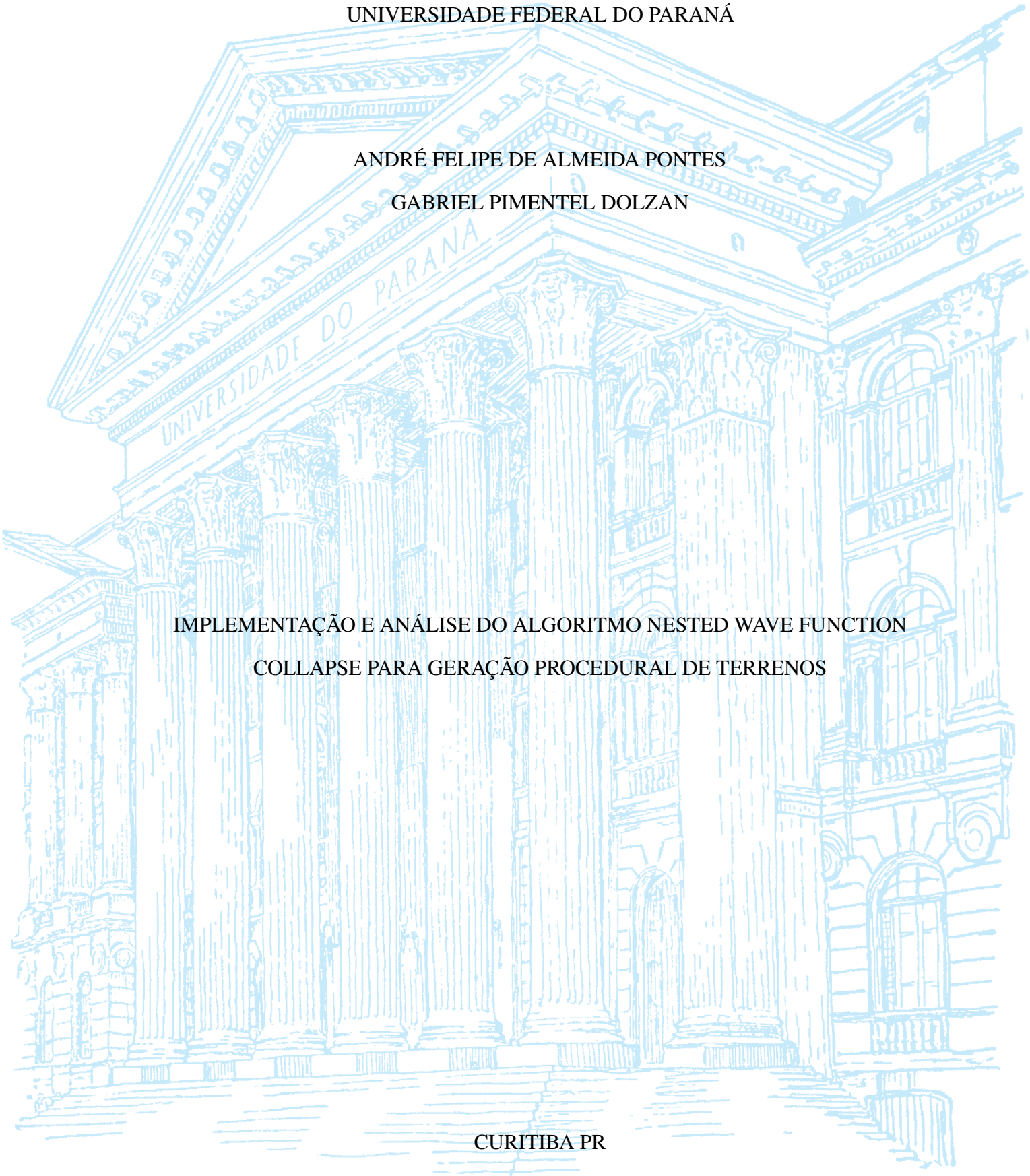
UNIVERSIDADE FEDERAL DO PARANÁ

ANDRÉ FELIPE DE ALMEIDA PONTES

GABRIEL PIMENTEL DOLZAN



IMPLEMENTAÇÃO E ANÁLISE DO ALGORITMO NESTED WAVE FUNCTION COLLAPSE PARA GERAÇÃO PROCEDURAL DE TERRENOS

CURITIBA PR

2025

ANDRÉ FELIPE DE ALMEIDA PONTES
GABRIEL PIMENTEL DOLZAN

IMPLEMENTAÇÃO E ANÁLISE DO ALGORITMO NESTED WAVE FUNCTION COLLAPSE PARA GERAÇÃO PROCEDURAL DE TERRENOS

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Computação*.

Orientador: Guilherme Alex Derenievicz.

CURITIBA PR

2025

AGRADECIMENTOS DE ANDRÉ FELIPE

Agradeço profundamente aos meus pais Reginaldo Rubens de Pontes Lima e Maria Regina de Almeida Pontes, pelo apoio incondicional em todos os momentos dessa jornada. Sair de uma cidade do interior de São Paulo para estudar em uma das maiores universidades federais do país é algo que, por muito tempo, pareceu um sonho distante e só foi possível graças à força, incentivo que sempre recebi deles. Agradeço aos meus amigos que estiveram ao meu lado ao longo dessa jornada. Foi uma experiência repleta de aprendizados, companheirismo e, acima de tudo, de muitos momentos de alegria. Compartilhar essa fase com vocês tornou tudo mais leve, significativo e inesquecível. Agradeço ao meu professor orientador Guilherme A. Derenievicz por compartilhar seus conhecimentos e por me guiar ao longo do desenvolvimento deste trabalho. Sua orientação não apenas contribuiu para a construção deste TCC, mas também me proporcionou uma valiosa perspectiva sobre a experiência de ser pesquisador e professor no Brasil, experiência que me fez ter vontade de também trilhar esse caminho. Agradeço à Universidade Federal do Paraná e a todos os professores do curso de Ciência da Computação por compartilharem seus conhecimentos e por proporcionarem experiências acadêmicas e pessoais tão enriquecedoras ao longo da minha formação.

AGRADECIMENTOS DE GABRIEL PIMENTEL DOLZAN

Primeiramente gostaria de agradecer a todos os meu familiares. Aos meus pais Adriane e Rafael por me apoiarem e me incentivaram incansávelmente durante toda essa jornada. Ao meu irmãozinho Samuel que sempre me faz rir com suas travessuras e sua alegria contagiante. Aos meus avós Arnaldo e Iraci que sempre me deram carinho e compartilharam as suas experiências de vida. Aos meus dindo e dinda Rodrigo e Alessandra que sempre cuidaram de mim e proporcionaram momentos de alegria e felicidade. Ao meu primo Gustavo, meu irmão do coração, que cresceu comigo, dividindo sonhos e experiências de vida. Eu sempre serei eternamente grato de ter vocês na minha vida, vocês me dão força para buscar os meus sonhos e me tornar uma pessoa melhor, um dia após o outro.

Aos meus amigos, que eu considero família, Vitor e Karol. Ao Vitor pelas conversas e ensinamentos sobre tecnologia e computadores e a Karol, junto com toda a equipe do Fisk, por me guiar na conquista da proficiência em inglês.

Aos meus amigos de Foz do Iguaçu, Enzo, Gustavo, Matheus, Bernardo, Leonardo, Lucas, Nicolas e Johann. Obrigado pelas madrugadas cheia de conversas filosóficas e muita diversão com nossos jogos preferidos. Sem vocês não seria a mesma coisa.

Aos meus queridos amigos de faculdade, que transformaram esta jornada em algo muito mais especial e significativo. Vocês tornaram os dias mais leves, e as conquistas mais emocionantes. Ao Anderson, meu primeiro amigo na faculdade, obrigado por me acompanhar nessa jornada do começo ao fim. Ao Túlio, Claudinei e Heric, por todas aquelas tardes que passamos conversando, rindo e compartilhando experiências que vão muito além das disciplinas que vimos juntos. Esses momentos foram fundamentais para manter o equilíbrio durante os períodos mais puxados. Ao Leonardo e Dante, amigos que sempre estiveram ao meu lado, oferecendo apoio em qualquer circunstância.

Ao André, meu parceiro de TCC. Sua dedicação incansável, comprometimento e sede pelo conhecimento foram inspiradoras durante todo o desenvolvimento deste projeto. Trabalhar ao seu lado foi um privilégio. Obrigado por dividir comigo esta conquista tão importante.

Ao professor Guilherme Derenievicz, nosso orientador, por acreditar e nos apoiar sempre. Você nos ensinou de maneira didática, além de compartilhar conhecimentos para tornar o nosso trabalho mais completo. Sua abordagem de ensino didático me inspirou para continuar trilhando esse caminho acadêmico, podendo pesquisar de maneira científica uma das áreas que eu mais gosto, desenvolvimento de jogos.

Não poderia deixar de agradecer à Universidade Federal do Paraná por me proporcionar um espaço de estudo de alta qualidade, estendo o meu agradecimento também ao Departamento de Informática e todos os professores com quem eu pude aprender e aprimorar meus conhecimentos.

Por fim, a todos que, direta ou indiretamente, participaram dessa minha jornada para o desenvolvimento desse trabalho.

Podem ter certeza que cada palavra e cada momento que tive o prazer de compartilhar com cada um de vocês foi fundamental para essa conquista. Muito obrigado a todos vocês!

RESUMO

A geração procedural de terrenos existe há mais de 30 anos e contribuiu significativamente para criação e simulação de ambientes virtuais variados. Nos últimos anos o algoritmo Wave Function Collapse (WFC) tem se destacado como uma técnica eficaz para a geração de terrenos bidimensionais. Diante dessa popularidade crescente surgem implementações que visam a otimização e eficiência, além da melhoria dos resultados obtidos, por isso, este estudo propõe a implementação e análise de desempenho e custo computacional do WFC e o Nested Wave Function Collapse (N-WFC).

Palavras-chave: Geração Procedural de Terrenos, Wave Function Collapse, Análise de Desempenho.

ABSTRACT

Procedural terrain generation has existed for more than 30 years and has contributed significantly to the creation and simulation of varied virtual environments. In recent years, the Wave Function Collapse (WFC) algorithm has stood out as an effective technique for generating two-dimensional terrains. Given this growing popularity, implementations aimed at optimization and efficiency, as well as improving the obtained results, therefore, this study proposes the implementation and analysis of performance and computational cost of WFC and Nested Wave Function Collapse (N-WFC).

Keywords: Procedural Terrain Generation, Wave Function Collapse, Performance Analysis.

LISTA DE FIGURAS

LISTA DE ACRÔNIMOS

WFC	Wave Function Collapse
I-WFC	Interior Wave Function Collapse
N-WFC	Nested Wave Function Collapse
PCG	Procedural Content Generation
PTG	Procedural Terrain Generation
CSP	Constraint Satisfaction Problem
DAC	Directional Arc Consistency

SUMÁRIO

1 INTRODUÇÃO

Na área de Inteligência Artificial, os algoritmos de geração procedural de conteúdo (PCG, do inglês *Procedural Content Generation*) são utilizados para diminuir tempo e custo de desenvolvimento de jogos algoritmicamente, criando recursos dinamicamente e tornando a experiência mais imersiva e personalizada para cada usuário (Mehta, 2025). Esse estudo aborda um tipo específico de geração procedural, voltado para a criação de terrenos bidimensionais gerados proceduralmente para jogos. Estudos de mercado também indicam a importância do segmento de PCGs, em 2023 esse segmento detinha mais de 30% da fatia do mercado de IA em jogos.(Market.US, 2024)

Em 2016, o algoritmo Wave Function Collapse (WFC) se destacou como uma técnica simples e eficaz para geração procedural de terrenos bidimensionais, sendo portado para diversas linguagens e utilizado em diversos jogos comerciais como Bad North, Caves of Qud e Townscaper (Gumin, 2016). Esse algoritmo utiliza princípios e ideias da mecânica quântica (como o colapso de onda e superposição) em conjunto com Constraint Satisfaction Problem (CSP) e Backtracking da área de Ciência da Computação. Devido à sua capacidade de gerar padrões complexos a partir de regras simples ou pré-definidas, em conjunto com a sua capacidade de gerar uma enorme variedade de conteúdo, tornou-se popular entre desenvolvedores de jogos.

O algoritmo Nested Wave Function Collapse foi proposto como uma variante do Wave Function Collapse. Esse algoritmo utiliza o conceito de *Tileset completo e subcompleto* para reduzir conflitos, removendo a necessidade de *backtracking*. Além disso, também quebra a matriz de geração em diversas matrizes menores, denominadas *subgrids*, acelerando o processo de geração de terrenos, além de garantir a geração de conteúdo infinito, aperiódico e determinístico (NIE et al., 2023).

Nesse contexto, este trabalho propõe a implementação do algoritmo e de sua variante N-WFC, além de uma análise comparativa de desempenho, custo computacional e resultados em relação ao WFC clássico em dois ambientes distintos, em Unreal Engine 5 e em C++. Além desses algoritmos, foi implementado uma versão simplificada do N-WFC onde se aplica a consistência de arco direcionada (DAC). O estudo buscou contribuir para o avanço da pesquisa em geração procedural de terrenos, através da análise de dados sobre a aplicabilidade, eficiência e utilização das técnicas abordadas para jogos e simulações de ambientes virtuais.

1.1 OBJETIVOS DO TRABALHO

O objetivo geral é realizar uma análise focada na comparação dos algoritmos, avaliando o desempenho, custo computacional e qualidade dos terrenos bidimensionais.

Os objetivos específicos são:

1. Implementar, em Unreal Engine 5 e em C++, os algoritmos WFC eN-WFC para geração procedural de terrenos bidimensionais.

2. Avaliar qualitativamente a diversidade, variedade, coerência e consistência dos resultados obtidos pelos algoritmos utilizados.

3. Investigar a capacidade do algoritmo N-WFC em ser mais otimizado do que o WFC, analisando quais são esses casos e o motivo dessa melhoria.

4. Investigar a capacidade de outras variantes do algoritmo em comparação com o WFC em serem mais otimizados e mais eficientes, analisando quais são esses casos e o motivo dessas melhorias.

5. Apresentar quais são as limitações encontradas nas variantes e possíveis áreas de melhoria.

6. Criar um entendimento estruturado e fundamentado do WFC e N-WFC que será utilizado para avançar o estudo sobre esses algoritmos na área de geração procedural de terrenos.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 GERAÇÃO PROCEDURAL DE CONTEÚDOS

A geração procedural de conteúdos consiste na definição automática de elementos do jogo que não estão completamente especificados durante o seu desenvolvimento, podendo se referir a níveis, mapas, regras, texturas, itens, missões, músicas, armas, veículos, personagens, histórias, entre outros (Shaker et al., 2016). A utilização de *PCGs* é um mecanismo que possibilita os desenvolvedores a criar e modificar grandes partes de uma fase, removendo o fardo de modificar cada detalhe de maneira manual. (Smith et al., 2011).

Segundo Togelius et al (Togelius et al., 2011), existem muitos argumentos para a utilização desse tipo de abordagem para criação de conteúdos para jogos. Um deles é a redução da memória utilizada, mantendo o mundo comprimido até que seja necessário expandí-lo. Outro argumento para a utilização desses algoritmos é a facilidade de não ter que criar cada aspecto desses conteúdos manualmente, o que poderia aumentar significativamente os recursos e o tempo gasto na produção desses softwares. Além disso, é a criação de jogos que utilizam como sua mecânica principal esse tipo de algoritmos, criando assim novos estilos de jogos onde a re-jogabilidade é muito mais alta.

A *Geração Procedural de Terrenos (PTG)* é um subproblema da geração procedural de conteúdos e refere-se a criação automática de terrenos e texturas (Rose e Bakaoukas, 2016). Esse trabalho vai abordar esse subproblema da geração procedural de conteúdos, a geração procedural de terrenos (PTG) em maior profundidade. Na Figura 2.1 é possível ver um exemplo de um terreno procedural bidimensional gerado pelo WFC.

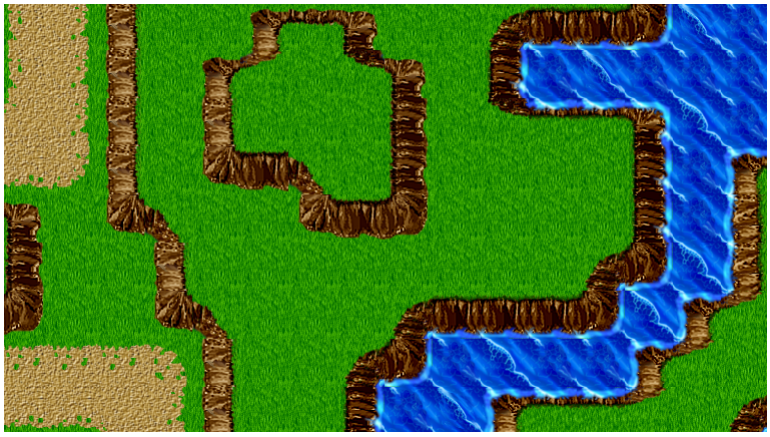


Figura 2.1: Exemplo de um terreno bidimensional gerado pelo algoritmo WFC (Gumin, 2016)

2.2 PROBLEMA DE SATISFAÇÃO DE RESTRIÇÕES

O Problema de Satisfação de Restrições (do inglês, *Constraint Satisfaction Problem*) pode ser utilizado para modelar e solucionar problemas no espaço de busca. Utilizando heurísticas específicas de domínios também é possível estimar o custo para atingir o objetivo a partir de um certo estado (Russell e Norvig, 2010). O objetivo do CSP é encontrar um conjunto de valores que satisfazem certas restrições estabelecidas.

O CSP pode ser representado por uma tripla (X, D, C), onde:

- $X = \{X_1, X_2, \ldots, X_n\}$ é um conjunto de **variáveis**;
- $D = \{D_1, D_2, \ldots, D_n\}$ é o conjunto dos **domínios** das variáveis, onde cada D_i é o conjunto de valores possíveis para a variável X_i;
- $C = \{C_1, C_2, \ldots, C_m\}$ é um conjunto de **restrições**, que especificam que valores são permitidos para as variáveis.

O objetivo do CSP é encontrar uma **atribuição** completa $A =< X_1 = v_1, X_2 = v_2, \ldots, X_n = v_n >$, com $v_i \in D_i$, isto é, todas as restrições em C sejam satisfeitas simultaneamente.

Formalmente, para toda restrição C_j envolvendo as variáveis $S_j = \{X_{j_1}, X_{j_2}, \ldots, X_{j_k}\}$, a tupla de valores $(v_{j_1}, v_{j_2}, \ldots, v_{j_k})$ atribuída às variáveis deve pertencer ao conjunto permitido definido pela restrição:

$$(v_{j_1}, v_{j_2}, \ldots, v_{j_k}) \in C_j$$

Backtracking é a técnica de busca em profundidade para CSPs. A cada nível, escolhemos uma variável não atribuída, tentamos um valor consistente e propagamos efeitos (com um algoritmo de propagação de restrições, como é o caso do Algoritmo 1). Se surgir inconsistência, a última atribuição é desfeita (*backtracking* é realizado) e outro valor será testado (Russell e Norvig, 2010). Esse processo é repetido até que uma atribuição completa, consistentes com as restrições, seja encontrada ou todas as opções sejam exploradas.

As heurísticas também são importantes para o CSP, com elas é possível determinar qual será a próxima variável a ser escolhida (Russell e Norvig, 2010), algumas heurísticas comuns para escolhas de variáveis são:

- ***Minimum Remaining Values* (MRV):** escolher a variável com menor número de valores remanescentes. Essa heurística é utilizada no WFC padrão (Gumin, 2016);
- ***Degree Heuristic*:** critério de desempate escolhendo a variável envolvida em mais restrições com outras não atribuídas;
- ***Least Constraining Value*:** testar primeiro o valor que elimina menos opções nas variáveis que compartilham alguma restrição;

O Algoritmo 1 apresenta o método AC-3 para aplicar uma forma de propagação por restrições (consistência de arco), que pode ser incorporado a um algoritmo de *Backtracking*. Em cada revisão, garante que todo valor em D_i tenha ao menos um suporte em D_j, removendo valores inconsistentes. Se D_i for reduzido, voltemos a verificar todos os arcos que apontam para X_i, até estabilizar todos os domínios. A complexidade no pior caso é $O(cd^3)$, onde c é o número de restrições e d o tamanho máximo de domínio (Russell e Norvig, 2010).

Também é possível utilizar métodos de *propagação de restrições* mais simples, dependendo do tipo do problema que é formulado. Por exemplo, *Directional Arc Consistency* ou DAC é utilizado para impor uma ordenação dentro de uma CSP e será consistente desde que seja seguido a direção dessa ordenação.(Rossi et al., 2006)

Algoritmo 1 AC-3

Require: CSP (X, D, C)
Ensure: Domínios D com consistência de arco
1: $Fila \leftarrow C$
2: **while** $Fila \neq \emptyset$ **do**
3: $C_j \leftarrow$ remover uma restrição de $Fila$
4: seja $S_j = \{X_{j_1}, X_{j_2}\}$ as variáveis da restrição C_j
5: **for all** $a \in D_{j_1}$ **do**
6: **if** não existe $b \in D_{j_2}$ tal que $(a, b) \in C_j$ **then**
7: remover a de D_{j_1}
8: **end if**
9: **end for**
10: **if** removeu algum valor de D_{j_1} **then**
11: **for all** restrição C_k sobre X_{j_1}, $k \neq j$ **do**
12: adicionar C_k em $Fila$
13: **end for**
14: **end if**
15: **end while**
16: **return** D

2.2.1 Sudoku e CSP

O jogo Sudoku tem um sistema muito semelhante ao funcionamento dos algoritmos estudados por esse trabalho, portanto, para compreender os algoritmos mencionados a explicação do jogo Sudoku, faz total relação com o método de como o WFC e suas variantes operam.

Sudoku é um jogo onde se deve preencher um grid 9×9 com dígitos de 1 a 9. Esse grid é subdividido em 9 subgrids (blocos) 3×3. Temos no total 81 variáveis, cada variável pode assumir um valor de 1 a 9. Para cada linha, todos os valores devem ser distintos, o mesmo ocorre para as colunas e para cada subgrid 3×3. Ou seja, o conjunto de dígitos $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ deve ser distribuído sem repetições em cada um dos casos que foram exemplificados.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figura 2.2: Exemplo de um jogo de Sudoku incompleto

Na Figura 2.2, é apresentado uma instância do jogo sudoku, na qual o usuário/jogador deve completar o tabuleiro. O jogo é finalizado quando todas as 81 variáveis são atribuídas de modo que as restrições de linha, coluna e subgrid sejam satisfeitas, como ilustrado na Figura 2.3.

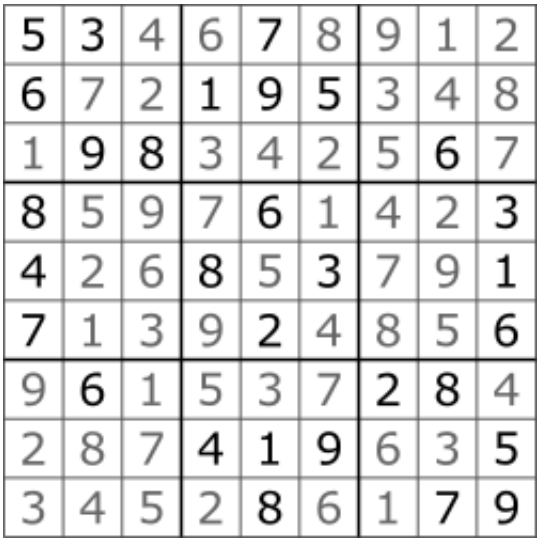


Figura 2.3: Exemplo de um jogo de Sudoku completo

O jogo Sudoku pode ser formalizado como um CSP, da seguinte forma:

- **Variáveis:** O jogo é composto por 81 variáveis, organizadas em uma grade 9×9. Cada variável pode assumir um valor em seu domínio. Denotamos cada variável como $X_{i,j}$, onde i e j indicam a linha e a coluna, respectivamente, com $1 \leq i, j \leq 9$.
- **Domínios:** O domínio inicial de uma variável $X_{i,j}$ é o conjunto dos números inteiros de 1 a 9:
$$D_{i,j} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$
- **Restrições:**
  - **Linha:** Cada número de 1 a 9 deve aparecer exatamente uma vez em cada linha. Formalmente, para toda linha i:
$$\forall j_1, j_2, \ (j_1 \neq j_2) \ \Rightarrow \ X_{i,j_1} \neq X_{i,j_2}.$$
  - **Coluna:** Cada número de 1 a 9 deve aparecer exatamente uma vez em cada coluna. Para toda coluna j:
$$\forall i_1, i_2, \ (i_1 \neq i_2) \ \Rightarrow \ X_{i_1,j} \neq X_{i_2,j}.$$
  - **Subgrid** 3×3**:** Cada número de 1 a 9 deve aparecer exatamente uma vez em cada subgrid. Para quaisquer células (i_1, j_1) e (i_2, j_2) dentro do mesmo subgrid:
$$(i_1, j_1) \neq (i_2, j_2) \ \Rightarrow \ X_{i_1,j_1} \neq X_{i_2,j_2}.$$
  - **Células Pré-Definidas:** Para as variáveis já preenchidas no estado inicial, o domínio é fixado para o valor dado. Se $X_{i,j}$ tem valor inicial v, então
$$D(X_{i,j}) = \{v\}, \quad 1 \leq v \leq 9.$$

Como foi citado na Seção 2.2, as heurísticas selecionam uma variáveis em CSP dependendo de um certo critério. Um dos motivo pelo Sudoku ser utilizado como exemplo para um funcionamento do WFC é a uma heurística utilizada para resolvê-lo: *Minimum Remaining Value* (MRV), isto é, a célula que possuir o menor domínio (menor quantidade de possibilidade de seleção) será a escolhida e definida, cada vez que isso é feito o objetivo de concluir o jogo fica mais próximo, assim como é possível ver na Figura 2.4.

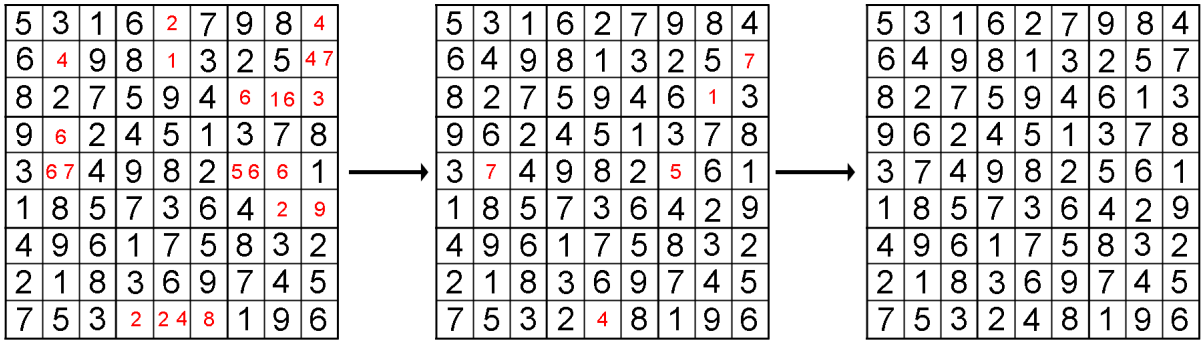


Figura 2.4: Exemplo de execução do MRV em um jogo de Sudoku, os números em vermelho representam os domínios das células

2.3 WAVE FUNCTION COLLAPSE

O algoritmo Wave Function Collapse, para geração procedural de terrenos, inspirado em ideias da mecânica quântica (como colapso de onda e superposição). Existem duas implementações que foram propostas (Gumin, 2016), o WFC Overlapping e WFC Simple-Tile.

A entrada do problema muda dependendo de qual WFC for utilizado. No caso do Overlapping a entrada é uma matriz de tamanho $n \times n$, as regras de adjacência são geradas a partir de uma imagem de input, rotacionando e espelhando as *Tiles*. No WFC Simple Tile um conjunto de padrões pré-definidos é utilizado, cada um com suas regras de adjacência.

De modo geral, o WFC consiste nos seguintes passos:

1. **inicialização:** todas as células da matriz começam não colapsadas (isto é, em um estado de sobreposição em todo o *tileset*). No caso da Figura 2.8, o tamanho do domínio inicial seria o tamamnho do tileset da Figura 2.7, ou seja, 16.

2. **seleção de célula:** Aqui é uma etapa que depende da heurística adotada pelo algoritmo; no caso do WFC original, ele escolhe a célula que possui a menor entropia, isto é, o menor domínio entre todas as células (o menor número efetivo de padrões possíveis de acordo com as regras estabelecidas), ou seja, MRV ou *Minimum Remaining Value*;

3. **colapso:** é a etapa na qual o algoritmo determina qual vai ser o tile daquela posição ao sortear um dos possíveis padrões para ser colocado naquela posição (colapsar a célula para tirá-la do estado de superposição);

4. **propagação:** após selecionar e colapsar uma célula, é necessário atualizar os domínios dos vizinhos, impedindo que padrões que não conectem com o *Tile* recém-colapsado sejam escolhidos. No caso do WFC (tanto Simple Tile quanto o Overlapping) é utilizado o AC-3. O Algoritmo 1 propaga as mudanças de domínio até que a matriz seja estabilizada.

5. ***backtracking*:** caso uma célula fique com seu domínio vazio, isto é, sem opções de padrões viáveis, é necessário realizar *backtracking*, desfazendo a etapa de colapso anterior e tentando outros padrões disponíveis. Caso contrário, se ainda houver células não colapsadas, voltar ao passo 2.

2.3.1 Wave Function Collapse Overlapping

Na implementação do overlapping, o input é uma imagem, a partir da qual se extraem todos os subpadrões de tamanho NxN; esses padrões são obtidos através da rotação e espelhamento dos tiles contidos na imagem original, as regras de adjacência são geradas e baseadas nesses padrões extraídos. Na Figura 2.5, as imagens à esquerda são as imagens de input, enquanto as da direita são as de saída.

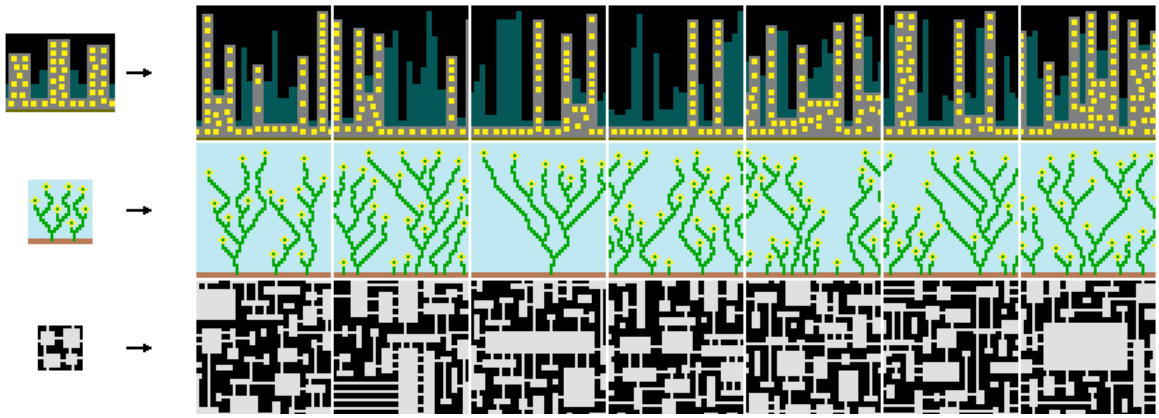


Figura 2.5: Exemplo de output usando WFC com Overlapping, retirado de (Gumin, 2016)

A Figura 2.6 mostra um exemplo de como os padrões são extraídos das imagens, rotacionando e espelhando as matriz, assim gerando novos padrões. As imagens a esquerda são as imagens a serem rotacionadas, as imagens a direita mostram todas as possibilidades geradas com a rotação e espelhamento.

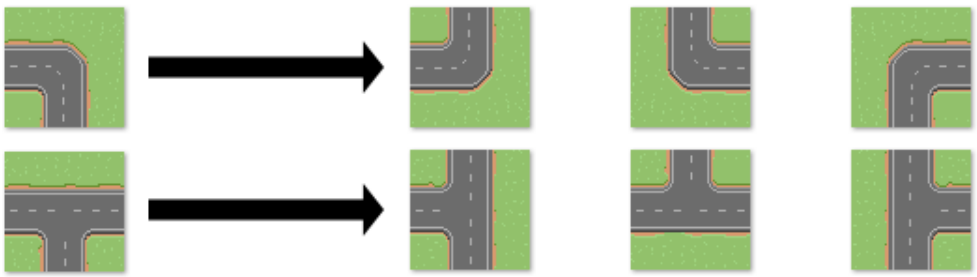


Figura 2.6: Exemplo de como os padrões são retirados do WFC Overlapping.

2.3.2 Wave Function Collapse Simple Tile

Na implementação do Simple Tile, o input é um *tileset* pré-definido, cada um com regras definidas com base em cada direção (norte, sul, leste, oeste), simplificando o processo de padrões possíveis (por isso o nome da implementação é *WFC Simple Tile*). Um exemplo de *tileset* para o Simple-Tile é apresentado na Figura 2.7 e um possível terreno de tamanho 10×10 é mostrado na Figura 2.8. As regras de validade, neste caso, é que bordas devem coincidir em estradas ou campo. o Algoritmo 2 mostra o funcionamento do WFC Simple-Tile.

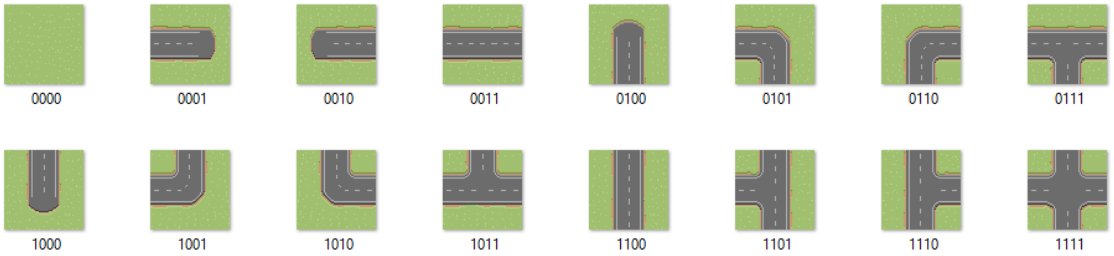


Figura 2.7: Exemplo de *tileset*, obtido de (Kubi, 2021) utilizado no WFC Simple-Tile.

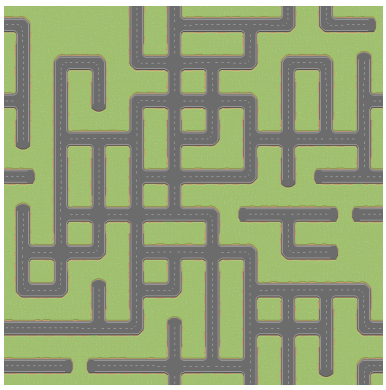


Figura 2.8: Exemplo de mapa gerado utilizando o WFC Simple-Tile a partir do *tileset* de (Kubi, 2021).

Algoritmo 2 Wave Function Collapse (WFC)

Require: CSP (X, D, C) representando um mapa $N \times N$, *tileset* τ
Ensure: Retorna uma solução *aceita*

Para cada célula $c \in G$, defina domínio $D(c) \leftarrow \tau$
while exista alguma célula não colapsada em G **do**
    Selecione a célula c não colapsada de menor entropia (domínio mínimo)
    Escolha aleatoriamente um padrão $p \in D(c)$
    Atribua $c \leftarrow p$ (colapse a célula)
    Chame AC-3(X, D, C) para propagar consistência de arco
    if existe alguma célula com $D(\cdot) = \emptyset$ **then**
        Backtrack:
            Desfaça o colapso de c e remova p de $D(c)$
            Opcionalmente, aplique AC-3 novamente
        continue para escolher outro padrão em c
    end if
end while
return G

2.3.3 N-WFC

O N-WFC é uma variante do WFC, proposta por (NIE et al., 2023), na qual a matriz é dividida em *subgrids* de tamanho fixo que se sobrepõem e o WFC Simple Tile (Seção 2.3.2) é executado separadamente em cada subgrid, chamado pelos autores de *Interior WFC* ou I-WFC. Em um laço externo as subgrids são percorridas em camadas diagonais, a Figura 2.10 mostra como funciona o processo de seleção de *subgrids* pelas diagonais utilizando uma matriz 3x3 com *subgrid* de tamanho 2x2 (totalizando 4 *subgrids* selecionadas). A sobreposição ocorre apenas

na primeira coluna e na primeira linha de cada subgrid, exigindo que haja coincidência com as escolhas feitas na camada diagonal anterior. Os seguintes passos detalham o algoritmo:

1. **inicialização:** fragmentar a matriz em subgrids de tamanho $C \times C$ que se sobrepõem na primeira linha e coluna, e separá-los em camadas diagonais, cada camada iniciando no lado superior direito e terminando no lado inferior esquerdo. Começar com a camada mais à esquerda;

2. **seleção de célula:** escolher o próximo subgrid seguindo a ordem diagonal estabelecida como mostra na imagem a direita da Figura 2.9;

3. **ajuste de restrições pré-estabelecidas:** Aplicar o Algoritmo 1 para as células já colapsadas no *subgrid*;

4. **I-WFC:** executar o WFC em $G_{a,b}$ para obter um novo *subgrid*;

5. **sobreposição:** inserir o *subgrid* na matriz, sobrepondo as bordas pré-colapsadas;

6. ***backtracking*:** *Backtracking* não ocorre se utilizar *Tilesets* completo e subcompletos, portanto, para o Algoritmo 3 a partir do momento que um *subgrid* é colocado, não é necessário retornar para ele. Então, o *backtracking* deve ser realizado apenas na execução do I-WFC e tentando os padrões disponíveis, assim como é descrito no Algoritmo 2.

A complexidade de tempo do N-WFC é polinomial se o *backtracking* do passo 6 do Algoritmo 3 nunca ocorrer, pois o I-WFC (passo 4) tem tamanho limitado em cada subgrid ($C \times C$) (NIE et al., 2023). Além disso, se os *Tilesets* forem **completos** ou **subcompletos**, o tempo polinomial é garantido. Tais *tilesets* garantem a existência de todas as permutações possíveis de bordas em cada lado dos padrões, evitando casos em que o *backtracking* seria necessário para corrigir inconsistências. O Algoritmo 3 detalha o funcionamento do N-WFC.

- ***tileset* completo:** é um conjunto que possui o mesmo número de *tiles* em cada uma das 4 direções (norte, sul, leste e oeste), assim se temos um *tileset* onde cada direção contém 2 *tiles* o número de *tiles* no conjunto será $|Norte/Sul|^2 * |Leste/Oeste|^2 = 2^2 * 2^2 = 16$.

- ***tileset* subcompleto:** é um conjunto que para todo *tile* existe pelo menos um *tile* em cada direção, isto é, devem existir combinações entre as direções N/S, L/O, N/O e S/L, de modo que permite a geração diagonal do N-WFC. Isso garante que apesar de possuir uma quantidade bem menor de *tiles*, sempre será possível montar um grid completo.

- ***tileset* incompleto** é um conjunto que para pelo menos um *tile* em uma das 4 direções não existe uma combinação possivel com nenhum outro tile de direção oposta. Por conta disso os algoritmos apresentados serão obrigados a realizar backtracking caso um *tile* incompleto seja escolhido. Foi observado que caso estejamos trabalhando com um *tileset* incompleto, pode-se realizar um pré-processamento para remover os *tiles* incompletos desse conjunto, transformando o conjunto em subcompleto ou completo

Algoritmo 3 Nested WFC (N-WFC)

Require: $Width$, $Height$, C
Ensure: Retorna uma solução *aceita*
1: $A \leftarrow (Width \cdot (C-1)+1)$
2: $B \leftarrow (Height \cdot (C-1)+1)$
3: $|G| \leftarrow A \times B$
4: Divida G em $A \times B$ subgrids, cada $|G^{sub}| = C \times C$
5: **for all** cada diagonal secundária **do**
6: **for all** $G^{sub}_{a,b}$ na diagonal secundária **do**
7: Inicialize a subgrid G^{sub}, $|G^{sub}| = C \times C$
8: **if** existe $G^{a,b-1}$ **then**
9: Copie os elementos da extrema direita para a extrema esquerda de G^{sub}
10: **end if**
11: **if** existe $G^{a-1,b}$ **then**
12: Copie os elementos inferiores para os superiores de G^{sub}
13: **end if**
14: Propague as mudanças para as outras posições vazias
15: $G^{sub} \leftarrow$ I-WFC(G^{sub})
16: **end for**
17: **end for**
18: **return** G

.

Propriedades garantidas pelo N-WFC Quando o N-WFC utiliza um tileset *completo* ou *subcompleto*, ele satisfaz as seguintes propriedades fundamentais:

Infinito A característica de infinidade garante que, não importa o quanto o processo de geração avance, o algoritmo sempre encontrará um tile compatível. (NIE et al., 2023).

Aperiódico O uso de tilesets completos ou subcompletos, garante que em cada célula ainda há pelo menos duas escolhas compatíveis, o que impede repetições e garante variabilidade. (NIE et al., 2023).

Determinístico O fato de que nenhum subgrid gerado pelo N-WFC sofrerá backtracking após ser aceito por um I-WFC, isto é, ele nunca será modificado ou revisitado. (NIE et al., 2023).

2.3.4 Consistência de Arco Direcionado (DAC)

Directional Arc Consistency ou DAC é uma consistência de arco (Seção 2.2), onde as variáveis são ordenadas e respeitam uma certa ordem (Rossi et al., 2006). Para esse estudo, uma nova variante do WFC foi proposta, essa variante realiza a consistência direcionada (diagonalmente ou iterativamente).

O DAC pode ser pensado como uma simplificação do N-WFC onde o *subgrid* é 1x1, desse modo, o DAC pode percorrer de maneira iterativa as células para serem colapsadas na ordem da matriz (esquerda para direita, de cima para baixo). Assim como podem ser percorridas na diagonal, isto é, segue a mesma ordem da matriz do N-WFC, mais especificamente a ordem na qual as *subgrids* são percorridas através das diagonais. A Figura 2.9 demonstra como os algoritmos percorrem a matriz de maneira diferente. Graças as definições de *tilesets completos e*

subcompletos (NIE et al., 2023), também foi possível garantir que nenhum *backtracking* ocorresse, devido ao fato que só precisamos verificar e modificar o donínio da célula a direita e a célula abaixo da recém colapsada, desde que os tilesets respeitem os limites estabelecido, o *backtracking* não ocorrerá. O Algoritmo 4 demonstra o funcionamento dessa variante.

Algoritmo 4 DAC

Require: CSP (X, D, C) representando um mapa $N \times N$
Ensure: Solução *aceita*

for $i = 1 \ldots N$ **do**
  for $j = 1 \ldots N$ **do**
    Escolher um *tile* t_1 em $D_{i,j}$ e atribuir a $X_{i,j}$
    for all *tile* t_2 em $D_{i,j+1}$ **do**
      if t_2.O $\neq t_1$.L **then**
        remover *tile* t_2 de $D_{i,j+1}$
      end if
    end for
    for all *tile* t_2 em $D_{i+1,j}$ **do**
      if t_2.N $\neq t_1$.S **then**
        remover *tile* t_2 de $D_{i+1,j}$
      end if
    end for
  end for
end for

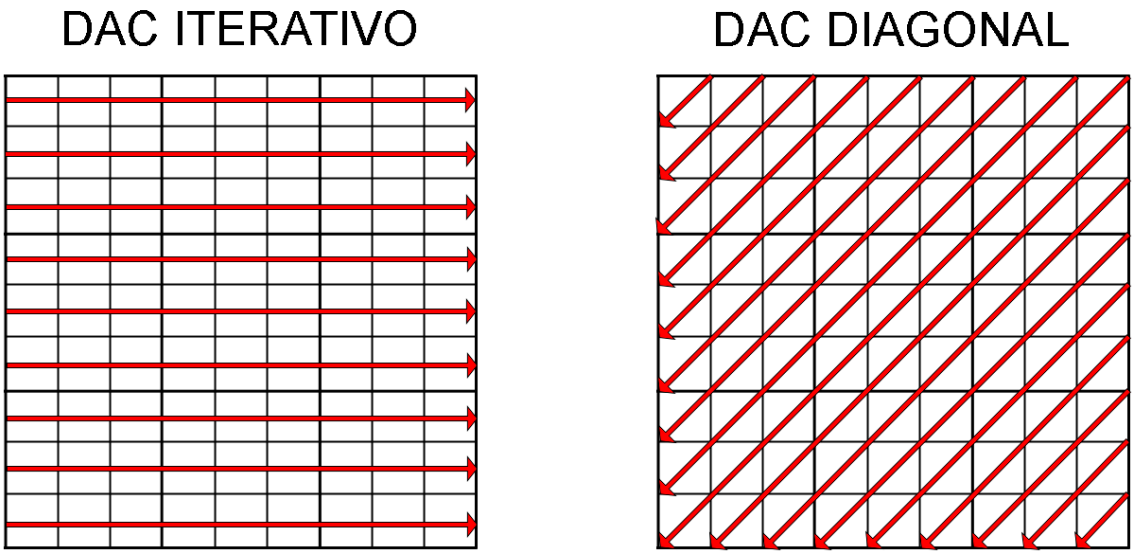


Figura 2.9: Imagem que demonstra como os diferentes DACs percorrem a matriz

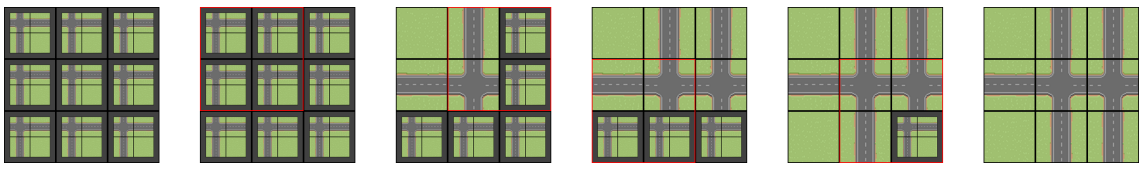


Figura 2.10: Imagem que demonstra como funciona a seleção de *subgrids* do N-WFC, da esquerda para a direita

3 REVISÃO BIBLIOGRÁFICA

No artigo (Efros e Leung, 1999) *Texture Synthesis* ou síntese de textura é um processo algorítmico de construir um output maior com base em um input menor (onde ambos input e output geralmente representam imagens), de modo a "expandir"esse input para fora pedaço por pedaço, no caso de imagens, pixel por pixel.

A ideia de dos autores foi criar clusters de informação para que distribuições sejam formadas a partir da imagem de entrada que nunca é modificada (estacionária) para serem selecionadas durante o processo de expansão da textura.

Nesse artigo a textura é modelada como sendo um *Markov Random Field* (MRF), isso significa que é assumido que uma determinada distribuição de valores e seus arredores (vizinhos) são independentes do resto da imagem. Além disso, a vizinhança de um pixel é descrita como sendo uma "janela"ao redor do pixel, essa *janela* é um parâmetro configurável, isso se da pelo fato que dependendo da imagem de entrada, uma janela maior ou menor podem impactar o resultado final.

Os pixels da imagem estendida são sintetizados um por um, onde cada pixel selecionado verifica seus vizinhos e gera um histograma de cada pixel central das vizinhanças no qual é possível selecionar ou estimar um valor para o novo pixel que é gerado, a Figura 3.1 mostra saídas do algoritmo, retirado de (Efros e Leung, 1999).

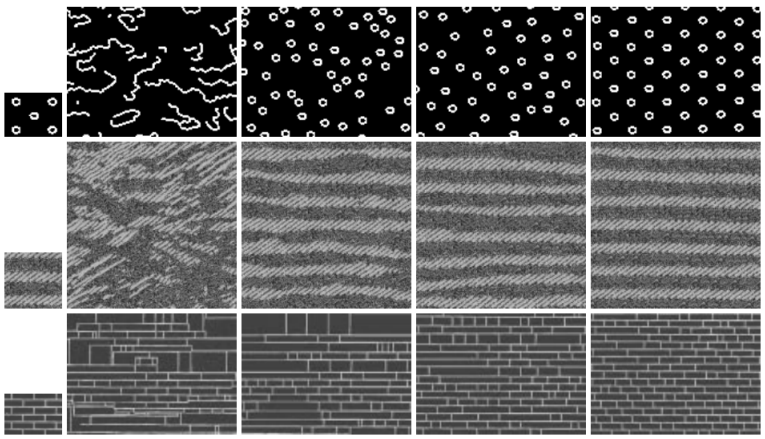


Figura 3.1: A imagem da esquerda mostra o input e as imagens da direita mostram outputs variados do algoritmo texture synthesis, retirado de (Efros e Leung, 1999)

O algoritmo é lento para imagens maiores, não trata imagens em perspectiva (apenas frontais) e o algoritmo apresentado nesse artigo pode "escorregar"e gerar algo que não condiz com uma "continuação"da imagem original, ou seja, copiar continuamente sem ampliar a imagem ou ainda gerar "lixo"em certas partes da imagem, como mostra a Figura 3.2 retirada de (Efros e Leung, 1999).

O WFC foi inspirado pelo processo criado por Efros e Leung, portanto o WFC também é classificado como um algoritmo de síntese de textura segundo o autor do WFC (Gumin, 2016).

No artigo feito por Paul Merrell (Merrell, 2021) é feita a comparação de um algoritmo de sua autoria, o Model Synthesis, com o algoritmo WFC de Maxim Gumin (Gumin, 2016). Em sua análise éele entende que ambos são similares em seu funcionamento, portanto é possível concluir que estão comparando o método de funcionamento do Model Synthesys com o método

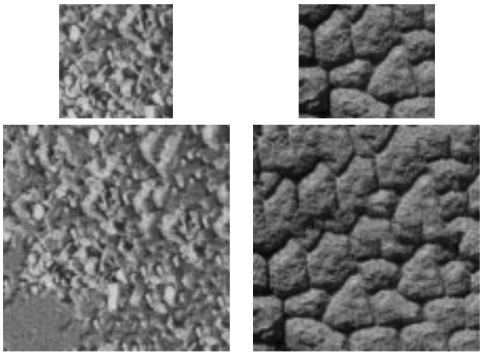


Figura 3.2: Os input estão localizados em cima, a parte inferior da imagem da esquerda mostra lixo sendo gerado, enquanto a imagem da direita mostra que o algoritmo ficou copiando a imagem ao invés de ampliá-la, retirada de (Efros e Leung, 1999)

de funcionamento do Model Synthesis do WFC, isto é, o WFC também é considerado um algoritmo de Model Synthesis.

A análise mostra que para entradas menores, ambos os algoritmos geram imagens similares e em tempo de execução similar, porém o WFC encontra grande dificuldade para gerar imagens maiores, falhando durante a execução. O autor também complementa falando que o Model Synthesis consegue gerar em segundos o que o WFC demora em mais de 20 minutos.(Merrell, 2021)

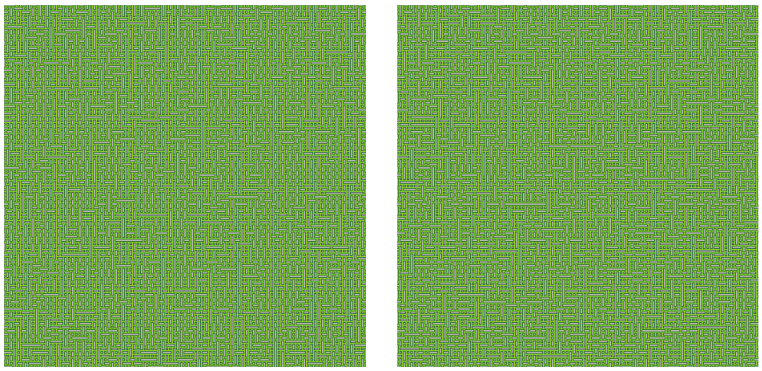


Figura 3.3: A imagem da esquerda mostra uma possível saída criada pelo Model Synthesis e a da direita mostra uma imagem criada pelo WFC, retirado de (Merrell, 2021)

Além disso, Merrell destaca duas diferenças principais entre o Model Synthesis e o WFC, que fazem com que o algoritmo que Maxim criou tenha problemas de geração de imagens. Uma delas é a ordem de escolha da célula a ser colapsada, o Model Synthesis segue a ordem de leitura, isto é, da esquerda para direita nas linhas e de cima para baixo até escolher cada célula sistematicamente. A ordem de escolha de células impacta a velocidade de execução do algoritmo, fazendo com que o WFC demore em certas instâncias testadas pelo autor mais do que 20 minutos de excução, enquanto o Model Synthesis demora alguns segundos. A outra grande diferença é o que o autor chama de "Modificar em Blocos", que é um processo que está ausente no WFC, onde uma área é modificada sem afetar a matriz toda, isso faz com que o WFC tente gerar a matriz toda de uma vez o que aumenta a dificuldade exponencialmente, isso também faz com que o Model Synthesis consiga gerar certos outputs que o WFC não consegue.

No artigo (NIE et al., 2023), os autores propuseram a variante N-WFC que resolve diversos problemas de complexidade do algoritmo e problemas de conflito (backtracking) estabelecendo restrições nos *tilesets*.

Os autores também introduziram uma ideia semelhante a do autor do Model Synthesis, Paul Merrell, em modificar a matriz em blocos separadas, denominados *subgrids*. Juntando esse fator com a utilização de *Tilesets completos e subcompletos*, foi possível remover backtracking, além de garantir conteúdo infinito, aperiódico e determinístico que segundo eles faz com que o N-WFC seja utilizado para conteúdos de larga escala como jogos digitais. (NIE et al., 2023)

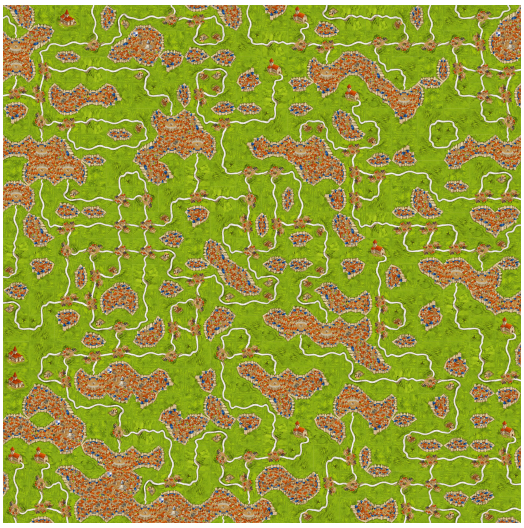


Figura 3.4: Um exemplo de imagem gerada utilizando o tileset do Carcassonne pela implementação em C++ do N-WFC

Nie et al também explica que existe uma grande limitação ao utilizar o N-WFC, pois não é possível garantir que a geração funcione com *Tilesets Incompletos*. Também explica que isso foi um dos motivos por não utilizarem o WFC Overlapping, devido a sua impossibilidade de garantir a geração de conjuntos subcompletos ou completos. Portanto, trabalhos futuros dos autores devem focar em modificar o Overlapping model para funcionar com o N-WFC e também trabalhar em métodos para pré-processar os conjuntos incompletos para torná-los subcompletos ou completos. O artigo também propõe uma variante do N-WFC que estende uma dimensão, ou seja, possibilita a criação de ambientes tri-dimensionais por meio de voxels. Isso é possível apenas adicionando duas direções cima e baixo, totalizando 6 direções. (NIE et al., 2023)

4 PROPOSTA

A principal proposta desse trabalho como foi discutido na Seção 1.1 é a implementação dos algoritmos WFC, N-WFC em C++ e em Unreal Engine 5. Isso se deve ao fato de que não existem implementações do N-WFC disponíveis para teste e verificação. Além disso, implementar todas as variantes no mesmo ambiente ou linguagem garante uma análise justa em consideração ao tempo de execução e consumo de memória.

Os testes serão realizados em 2 ambientes distintos de programação, o trabalho vai ser feito com a *Unreal Engine*, uma das engines mais versáteis e poderosas para o desenvolvimento de jogos e aplicações 3D em tempo real. Ela permite programar em C++ e, por meio de *Blueprints*, criar lógica de forma visual (“arrastar e soltar”). Além disso, dispõe de ferramentas avançadas para animação, física, áudio e iluminação. A escolha da Unreal Engine vem pela sua integração com C++ que possibilita alto desempenho, escalabilidade, controle em baixo nivel e suas ferramentas que proporcionam um trabalho mais ágil. Além disso, o trabalho será feito também em C++, que é um linguagem de alto desempenho, utilizada para criação de jogos e aplicações de alta performance. Essa linguagem foi escolhida para esse estudo para testar a capacidade da Unreal Engine, as formas de lidar com as estruturas, com a memória e performance do algoritmo medida pelo tempo. Todos os programas de teste vão ser realizados em Unreal Engine 5.4 e em C++ 17.

No Capítulo 5 será abordado os resultados das implementações do N-WFC, WFC, além de testar a versão simplificada do N-WFC (chamada de DAC), em C++ e na Unreal Engine. Os experimentos foram conduzidos em duas maquinas distintas. Os experimentos na Unreal Engine foram conduzidos em uma maquina AMD Ryzen 5600x, 4.6 GHz, com 32GB de memória RAM, executando Windows 11 v. 24H2 e Unreal 5.4, já os experimentos em C++ foram conduzidos em uma maquina Intel i7-13700K, 3.40 GHz, com 32GB de memória RAM, executando Windows 11 v. 24H2. As instancias foram executadas com *tileset* subcompleto apresentado na figura 2.7 e incompleto apresentados nas figuras 5.1 e 5.2. Cada instância foi executada 10 vezes e calculado a media de tempo, memória e backtracking.

5 AVALIAÇÃO DOS RESULTADOS

A avaliação dos resultados será feita utilizando os algoritmos WFC e N-WFC. Para os experimentos em C++ com tileset subcompleto, foi utilizado o tileset da figura 2.7 e o tileset incompleto 5.1. Na Unreal Engine os experimentos com Tileset subcompleto foram utilizados uma versão do tileset da figura 2.7 porém em Static Mesh (Modelo 3D) e o tileset incompleto 5.2.

A Unreal Engine, por ser uma ferramenta de desenvolvimento de jogos, contém vários subprocessos em execução junto dos algoritmos propostos, isso faz com que possa ter uma certa discrepância no uso de memória, tempo de execução e uso de memória cache, é mostrado na Seção 5.2.

Figura 5.1: Exemplo de Tileset Incompleto

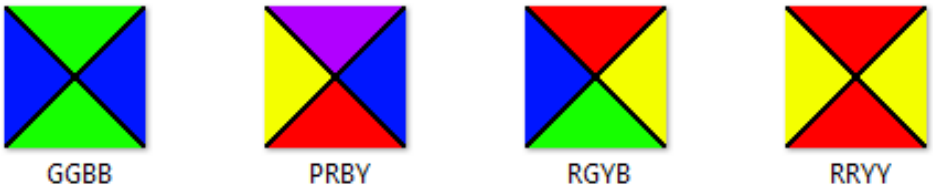


Figura 5.2: Exemplo de Tileset Incompleto Utilizado na Unreal Engine

5.1 EXPERIMENTOS EM C++

5.1.1 Tileset subcompleto

O gráfico da Figura 5.3 apresenta o tempo de execução dos algoritmos implementados em C++. O WFC com a heurística Diagonal apresenta melhor resultado que o N-WFC. O DAC (versão simplificada do N-WFC onde o *subgrid* é 1x1) por não utilizar o AC-3 e apenas comparar com no máximo 2 vizinhos escolhidos, possui uma grande vantagem em cima do N-WFC e WFC que fazem comparações com toda a vizinhança e vizinhos de vizinhos. O DAC iterativo é um pouco mais eficaz que o DAC Diagonal, pois quanto maior a diagonal maior será o número de cache miss.

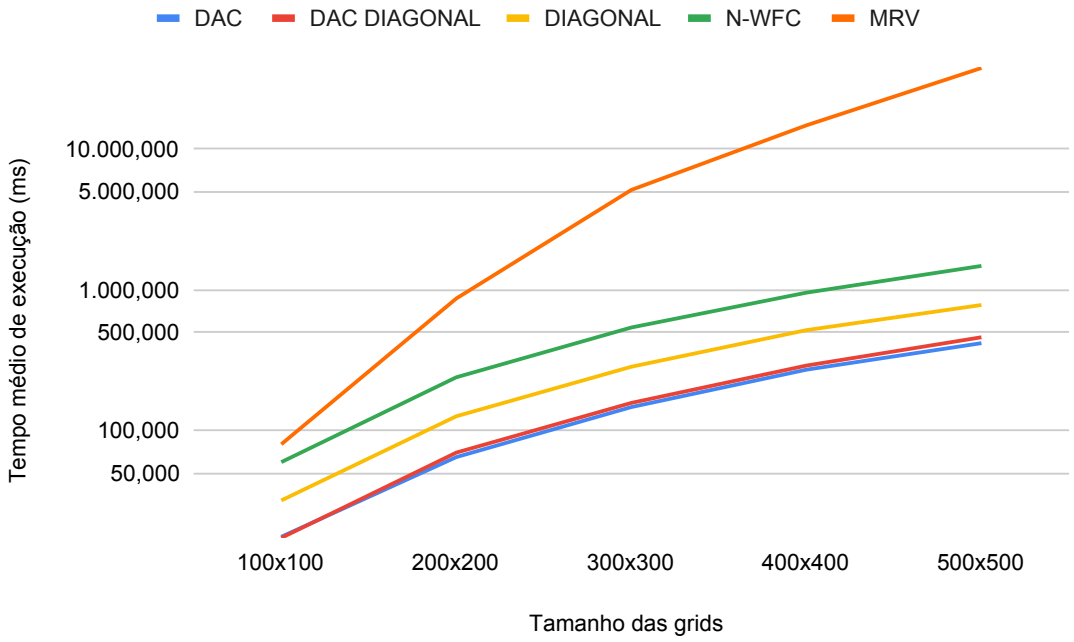


Figura 5.3: Tempo médio de execução (em ms) com base no tamanho da matriz, para o WFC variando o tamanho da subgrid em C++

O gráfico da Figura 5.4 apresenta o uso de memória dos algoritmos implementados em C++. Isso demonstra que o N-WFC tem o maior gasto de memória em todos os tamanhos de matrizes, isso se dá ao fato do N-WFC criar cópias para a subgrid CxC. O DAC apresenta melhor eficiência no uso de memória, pois contém menos estruturas de dados que o WFC, por conter uma simplicidade maior.

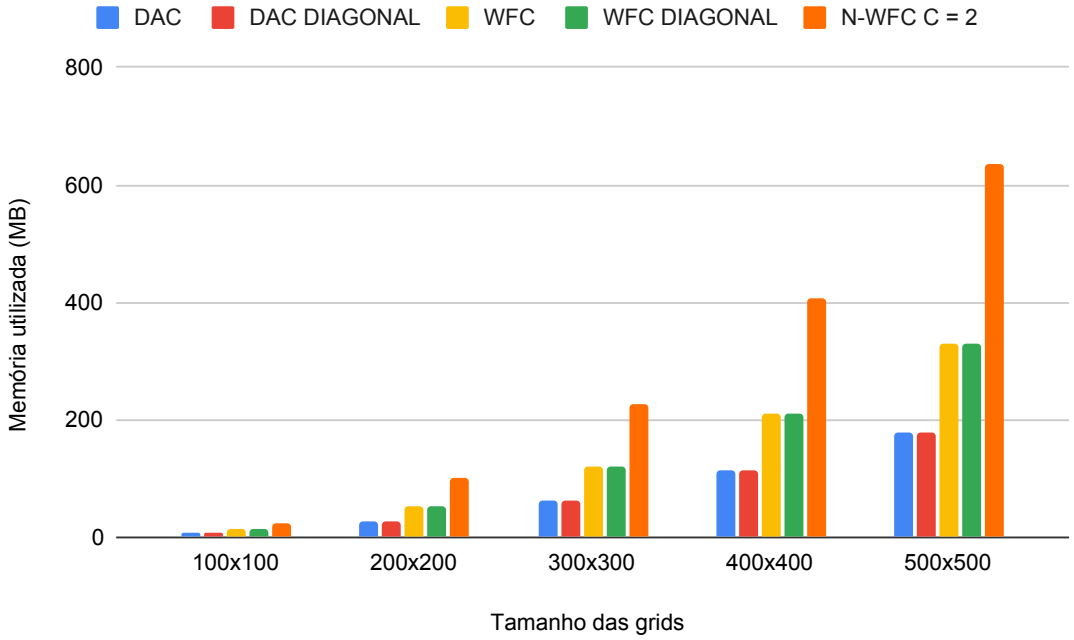


Figura 5.4: Uso de memória (em mb) com base no tamanho da matriz, para o WFC variando o tamanho da grid em C++

O gráfico da Figura 5.5 apresenta o tempo de execução de diferentes subgrids para o algoritmo N-WFC. É possível perceber que o N-WFC com subgrid de tamanho 5, possui o melhor tempo de execução. O gráfico da figura 5.6 apresenta o uso de memória de diferentes subgrids para o algoritmo N-WFC. O gasto de memória é sutilmente igual, isso se deve ao fato de as matrizes serem de mesmo tamanho e a diferença sutil vem do tamanho das subgrids alocadas.

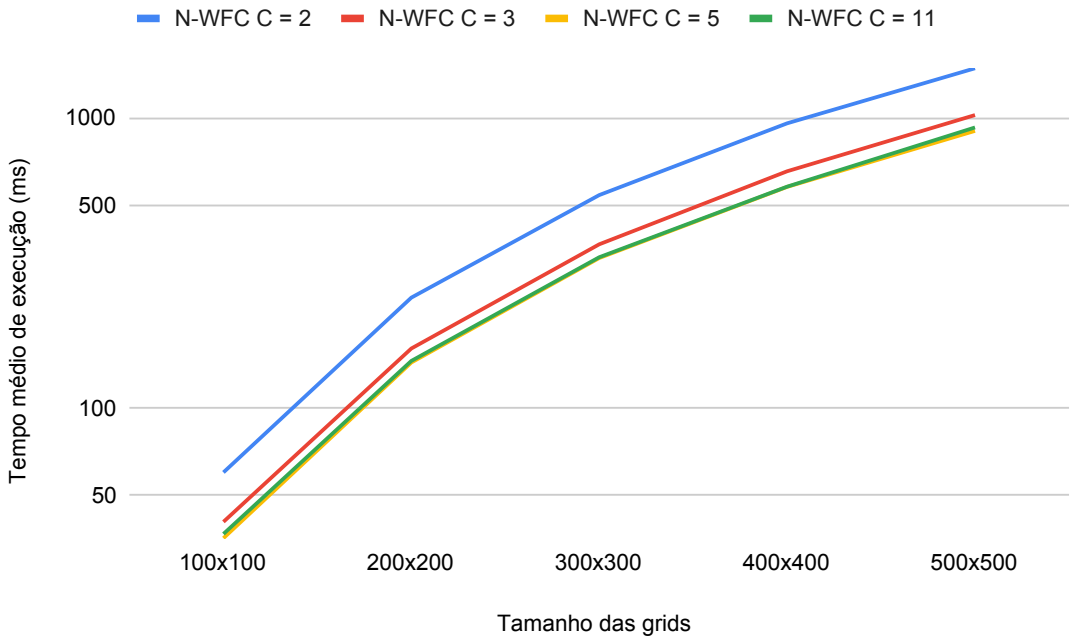


Figura 5.5: Tempo médio de execução (em ms) com base no tamanho da matriz, para o N-WFC variando o tamanho da subgrid em C++

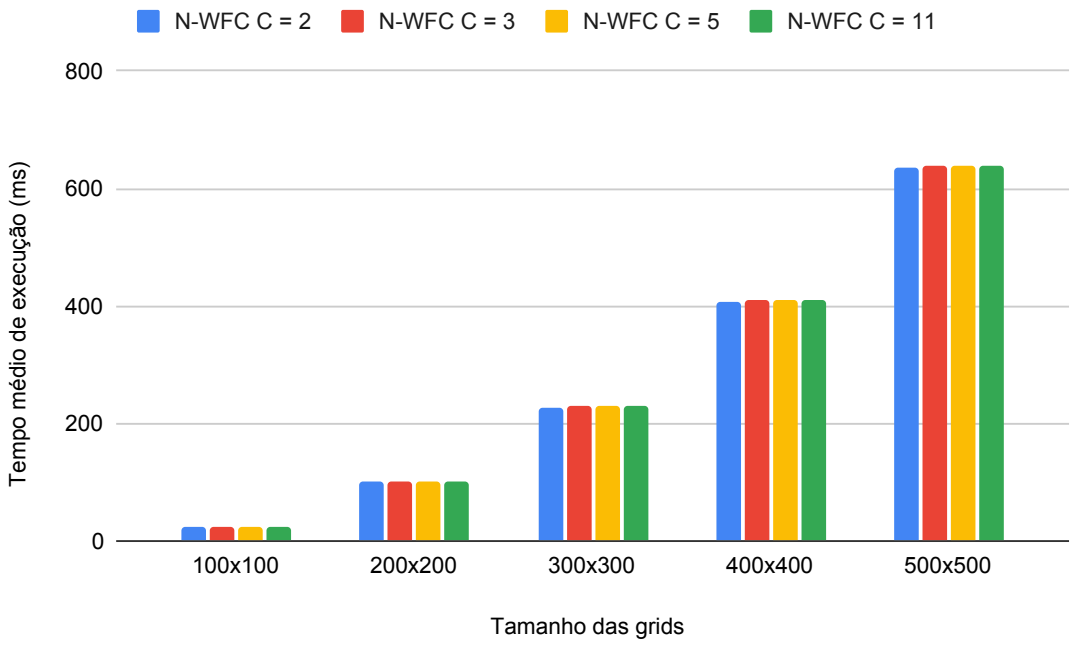


Figura 5.6: Uso de memória médio (em mb) com base no tamanho da matriz, para o N-WFC variando o tamanho da subgrid em C++

5.1.2 Tileset Incompleto

O algoritmo N-WFC para o *Tileset Incompleto* que foi utilizado, como mostra a Figura 5.1 tem um problema na hora de construir os *subgrids*, isto ocorre devido ao fato que quando um *subgrid* é colocado, ele nunca mais é visitado de novo, e no caso desse *Tileset* específico, os *subgrids* tendem a ter suas bordas preenchidas com tiles que para a matriz global são considerados incompatíveis, pois a sobreposição não pode ser concluída corretamente.

Portanto uma estratégia foi desenvolvida para fazer com que o processo de sobreposição do N-WFC consiga terminar corretamente. Para *Tilesets Incompletos* que possuem tiles que só podem ser colocados nas bordas da direita e inferior, o *subgrid* precisa ser aumentado, por exemplo, um *subgrid* 2x2 se tornará um subgrid 3x3, onde as extremidades da direita e de baixo são descartadas (se não estiver na borda global da matriz) após o I-WFC terminar de rodar. Essa estratégia garantiu que, apesar dos *backtrackings* realizados nos *subgrids*, não foi necessário realizar nenhum *backtracking* para refazer um subgrid existente, assim mantendo essa propriedade do N-WFC como é proposto no artigo original (NIE et al., 2023). A Figura 5.7 demonstra um subgrid 2x2 sendo expandido em 3x3 devido a inconsistência dos conjuntos incompletos. Os quadrados em azul claro serão descartados durante a etapa de sobreposição do subgrid sobre a matriz global.

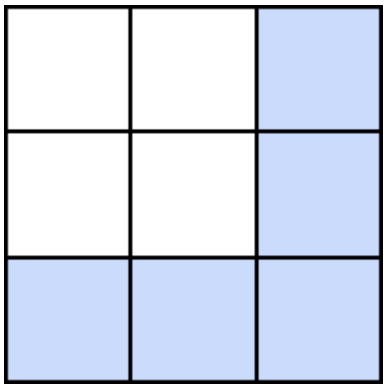


Figura 5.7: Um exemplo de um subgrid 2x2 que é expandido para 3x3.

O gráfico da figura 5.8 apresenta o tempo de execução dos algoritmos com backtracking implementados em C++. A análise do gráfico indica que o N-WFC contém o menor tempo de execução dos algoritmos, pois o WFC e o DAC, realizam backtracking na matriz de tamanho MxN, enquanto o N-WFC realiza backtracking apenas na subgrid de tamanho CxC.

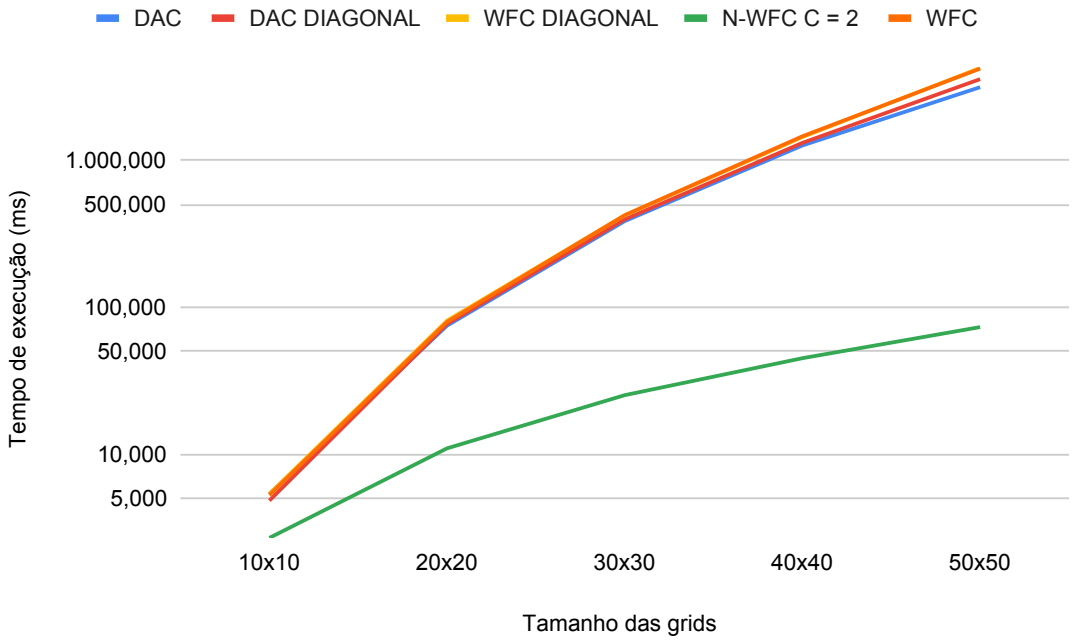


Figura 5.8: Tempo médio de execução (em ms) com base no tamanho da matriz, para o WFC variando o tamanho do grid em C++

O gráfico da figura 5.9 apresenta o uso de memória dos algoritmos DAC e WFC, implementados em C++. É indicado que o WFC contém o menor uso de memória, pois o WFC utiliza o AC-3 para diminuir o tamanho do domínio, logo as cópias dos domínios em memória são menores no WFC, consequentemente diminuindo o uso de memória.

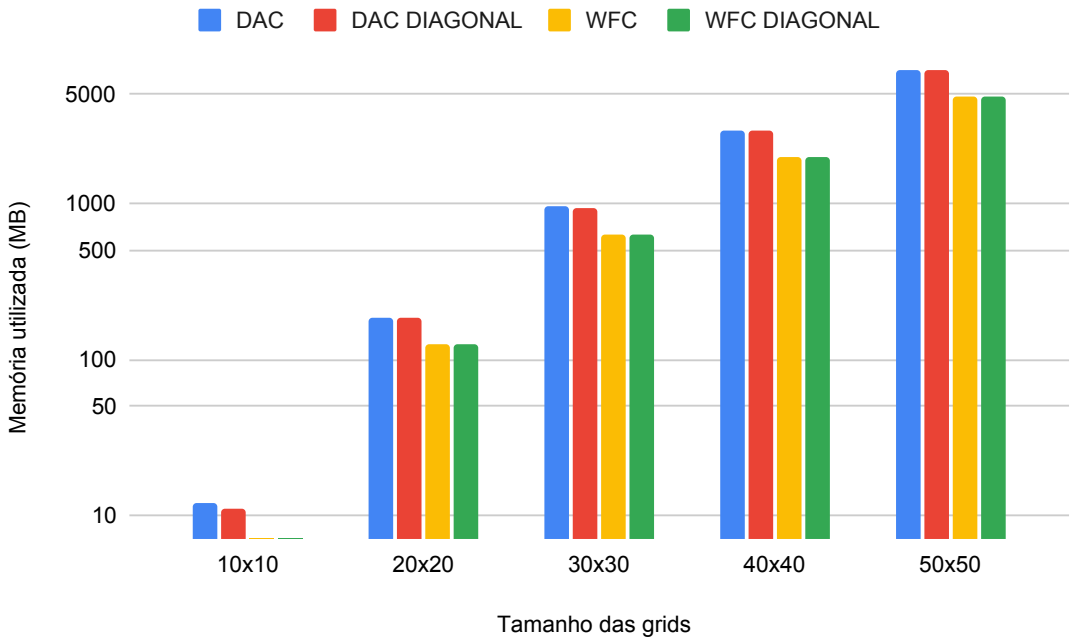


Figura 5.9: Uso de memória médio (em mb) com base no tamanho da matriz, para o WFC variando o tamanho do grid em C++

O gráfico da figura 5.10 apresenta o número de backtracking dos algoritmos implementados em C++. O N-WFC contém o maior quantia de backtrackings, pois o Tileset utilizado 5.1 faz com que as bordas do leste e sul da subgrid não sejam posições validas para os Tiles 1 (sul) e 2 (leste), consequentemente aumentando o número de backtrackings nas subgrids CxC.

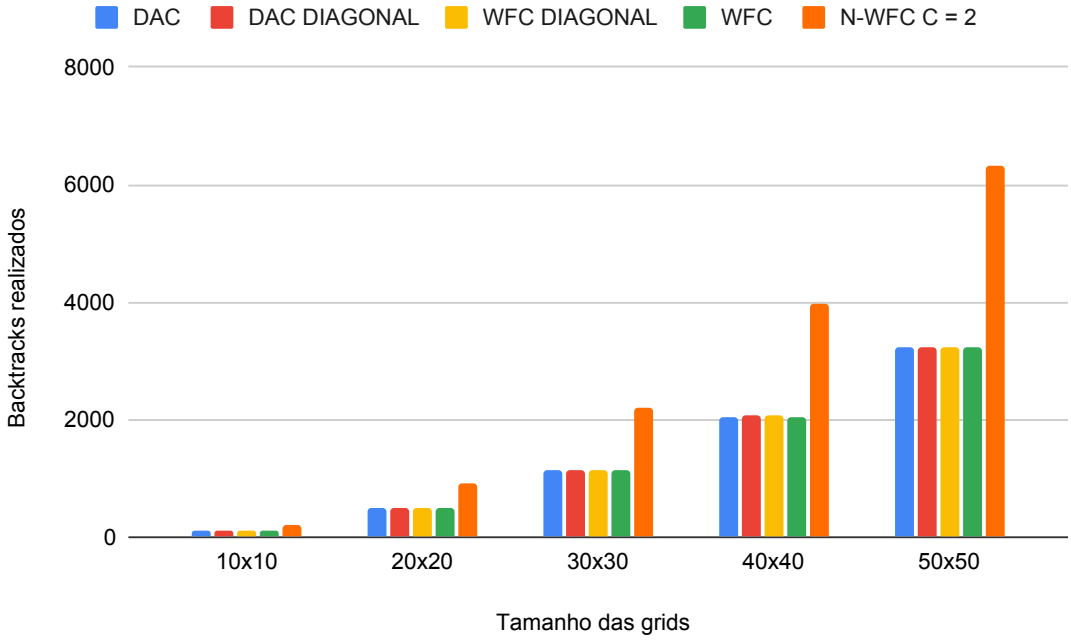


Figura 5.10: Número médio de backtrackings com base no tamanho da matriz, para o WFC variando o tamanho do grid em C++

Também foram feitas análises qualitativas com base nas saídas obtidas pelas implementações de cada variante em C++. Foram usados 2 *Tilesets* para testar as saídas, um deles é o tileset do jogo Carcassonne e o outro de estradas mostrado anteriormente.

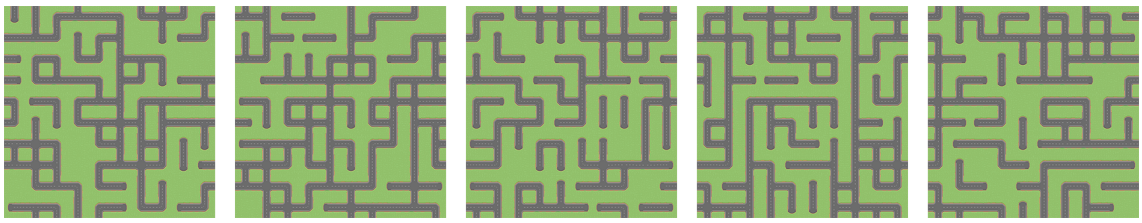


Figura 5.11: Imagens 10x10 com o tileset de estradas. Cada implementação em ordem da esquerda para a direita: DAC Diagonal, DAC, NWFC, WFC Diagonal e WFC

Figura 5.12: Imagens 10x10 com o tileset do Carcassonne. Cada implementação em ordem da esquerda para direita: DAC Diagonal, DAC, NWFC, WFC Diagonal e WFC

As análises qualitativas segundo a diversidade, variedade, coerência e consistência dos resultados indicam que não há mudança significativa na saída com base no algoritmo utilizado, em outras palavras, não foi possível perceber grandes mudanças visuais com base no algoritmo utilizado.

5.2 EXPERIMENTOS NA UNREAL ENGINE

5.2.1 Tileset subcompleto

O gráfico da Figura 5.13 apresenta o tempo de execução dos algoritmos na Unreal. O N-WFC apresenta um tempo de execução menor, vimos que em C++ o DAC apresentou o melhor tempo, como na Unreal Engine é um ambiente para desenvolvimento de jogos, existem muitos subprocessos além do cálculo dos algoritmos impactando a performance, com isso o uso da memória cache não é todo do algoritmo, como o N-WFC trabalha com subgrid ele consegue fazer o uso da memória cache de forma concisa, O DAC e WFC por estarem utilizando a Matrix toda MxN acabam obtendo um maior número de cache miss, impactando na performance. Porém não houve testes o suficiente em relação a memória cache, então é apenas uma hipótese do por que o N-WFC tem a melhor performance.

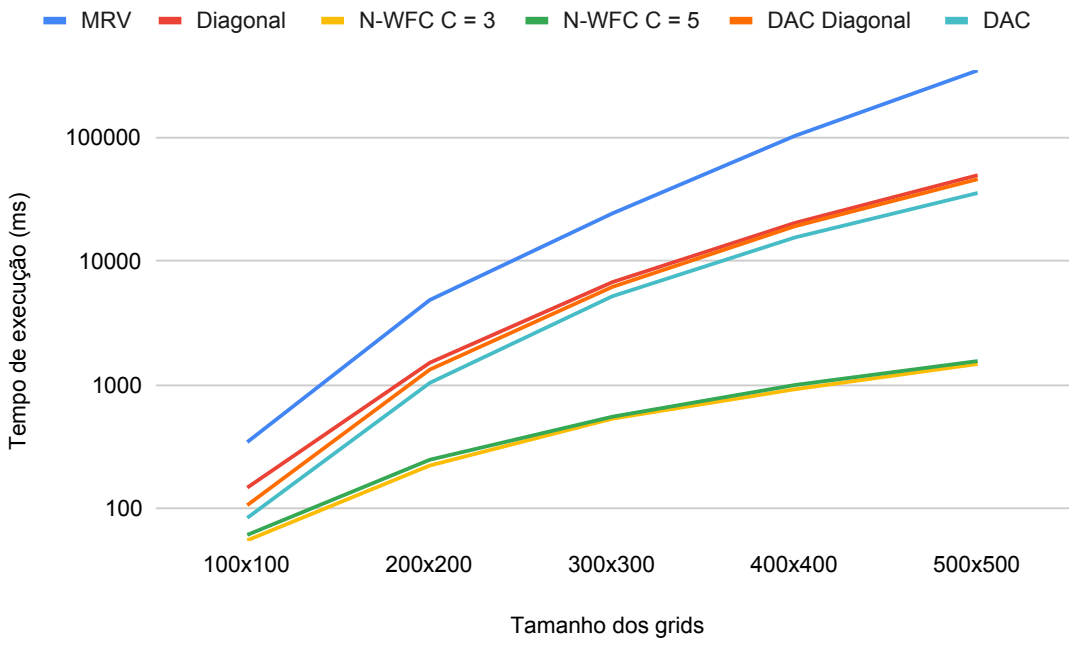


Figura 5.13: Tempo médio de execução (em ms) com base no tamanho da matriz, variando o tamanho do grid na Unreal Engine

O gráfico da figura 5.14 apresenta o uso de memória dos algoritmos na Unreal. O DAC faz uso mais eficiente de memória, pois não utiliza o AC-3, este que a cada iteração adiciona elementos na fila, para propagar as restrições.

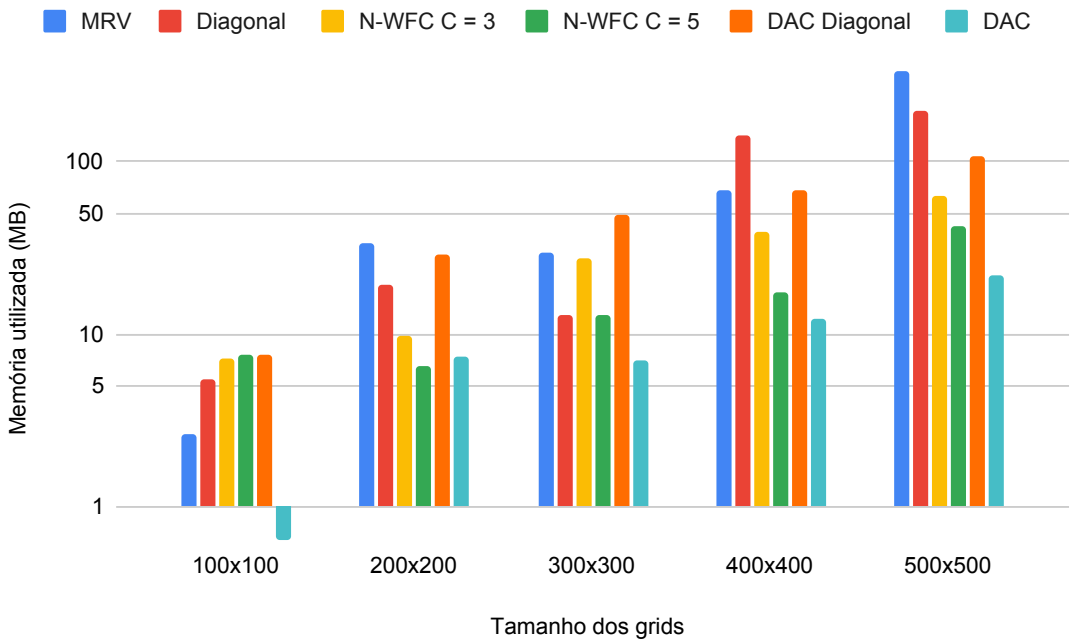


Figura 5.14: Uso de memória médio (em mb) com base no tamanho da matriz, variando o tamanho do grid na Unreal Engine

5.2.2 Tileset Incompleto

O gráfico da figura 5.15 apresenta o tempo de execução dos algoritmos DAC e WFC na Unreal. Todos os algoritmos apresentaram um tempo de execução muito próximo, pois com tileset incompleto os algoritmos ajustam os domínios até o tile incompleto encontrar o seu lugar na borda e caso esse tile incompleto esteja fora da borda o backtracking é acionado.

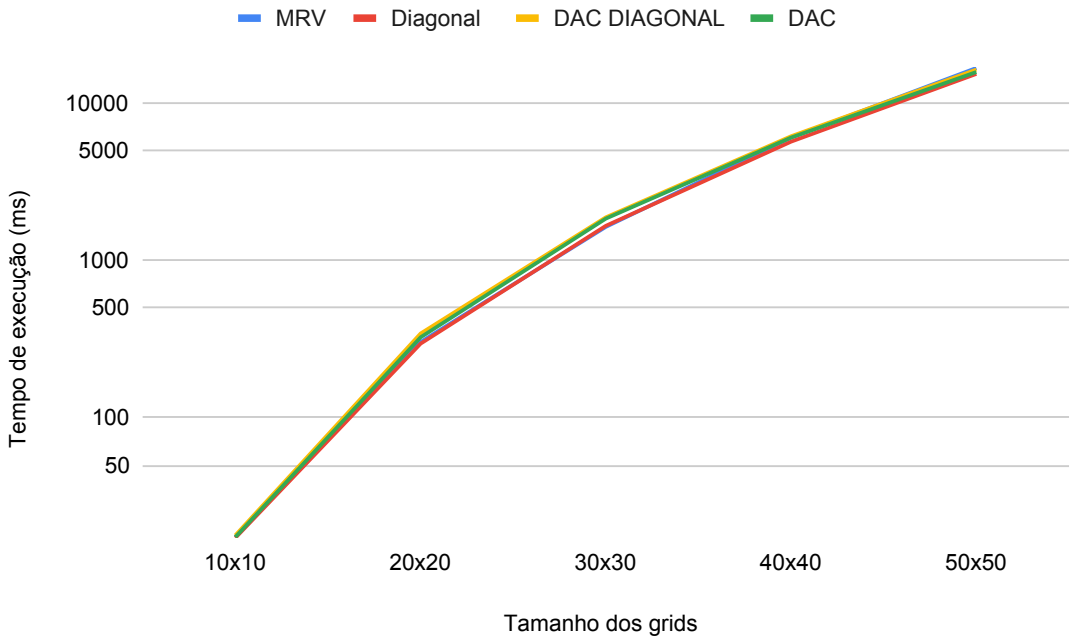


Figura 5.15: Tempo médio de execução (em ms) com base no tamanho da matriz, para o WFC variando o tamanho do grid na Unreal Engine

O gráfico da figura 5.16 apresenta o uso de memória dos algoritmos DAC e WFC na Unreal. É possível ver no gráfico que o WFC diagonal apresentou o menor uso de memória, pois utilizando a heurística diagonal o AC-3 retira os domínios inválidos e como está iterando na diagonal só faz duas comparações no AC-3, consumindo menos memória. Por causa do tileset utilizado não haverá backtracking.

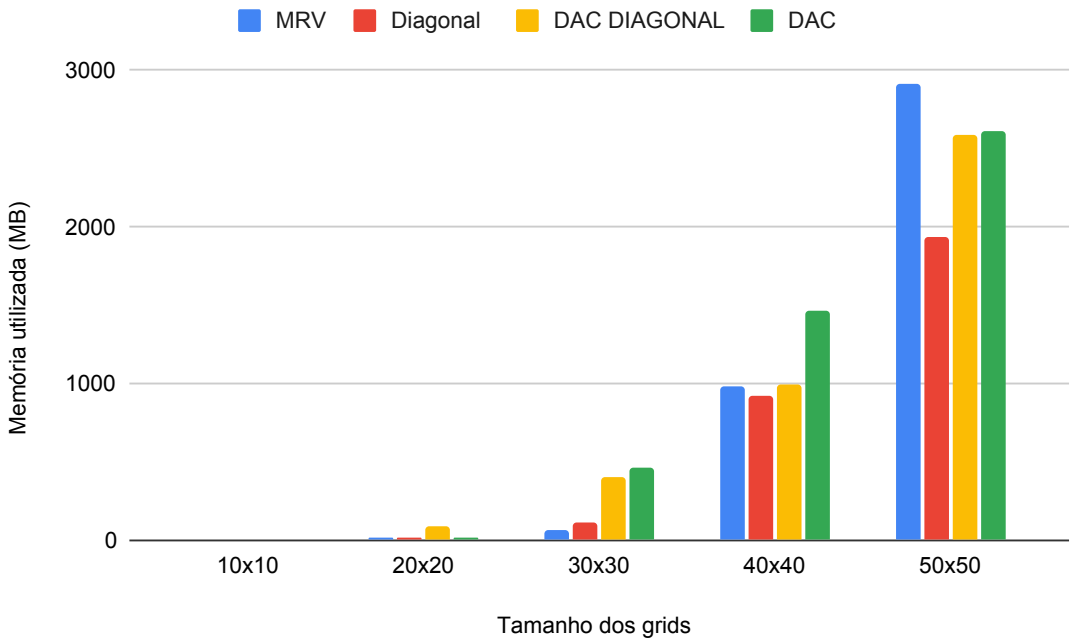


Figura 5.16: Uso de memória médio (em mb) com base no tamanho da matriz, para o WFC variando o tamanho do grid na Unreal Engine

O gráfico da figura 5.17 apresenta o tempo de execução para o N-WFC com diferentes subgrids. É possível perceber que, conforme o subgrid CxC aumenta, o custo do N-WFC também aumenta, assim como o custo de memória, como podemos ver no gráfico da figura 5.18, pois a cada iteração do algoritmo é feita uma cópia para o subgrid CxC.

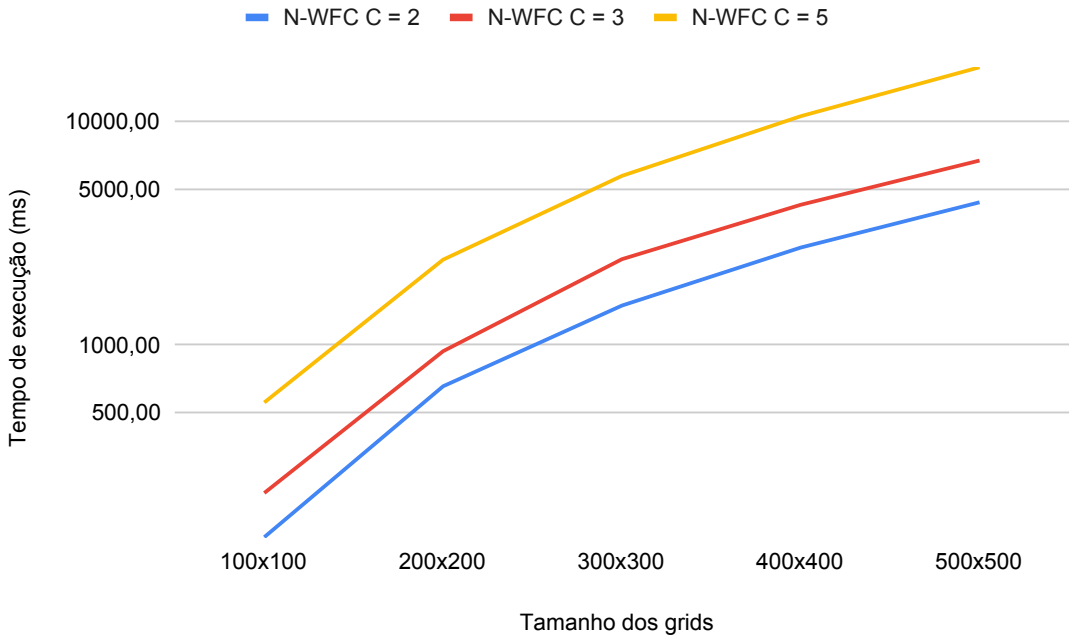


Figura 5.17: Tempo médio de execução (em ms) com base no tamanho da matriz, para o N-WFC variando o tamanho do grid na Unreal Engine

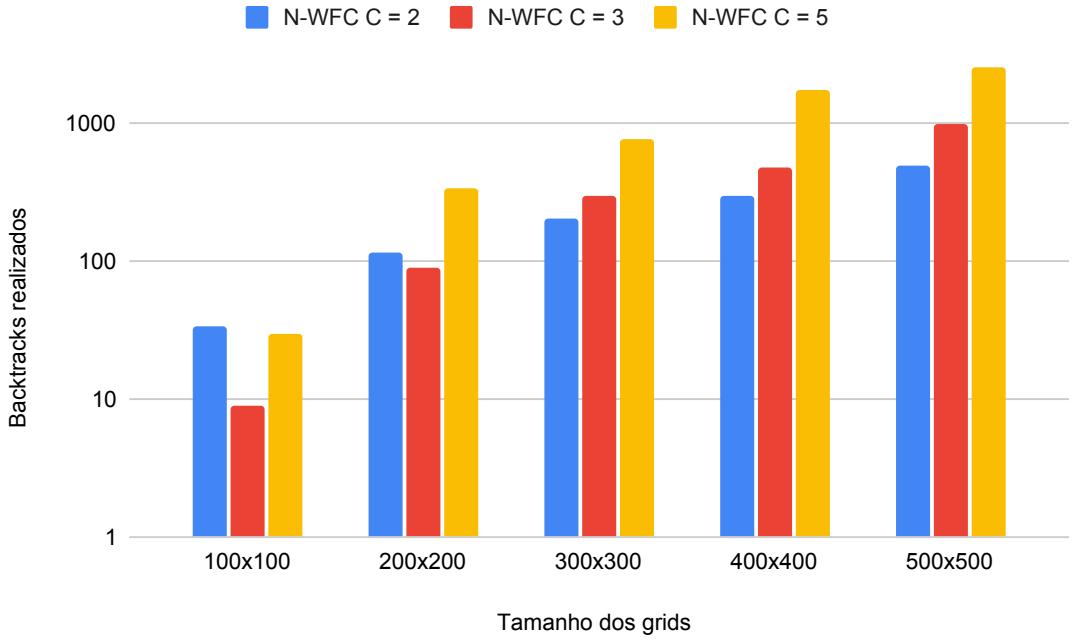


Figura 5.18: Uso de memória médio (em mb) com base no tamanho da matriz, para o N-WFC variando o tamanho do grid na Unreal Engine

O número de backtrackings do N-WFC para o tileset 5.2 será sempre 0, pois o AC-3 só permitirá que bloco 2(PRBY) sejá inserido nas bordas superiores do grid.

Exemplos de saídas geradas pela Unreal Engine:

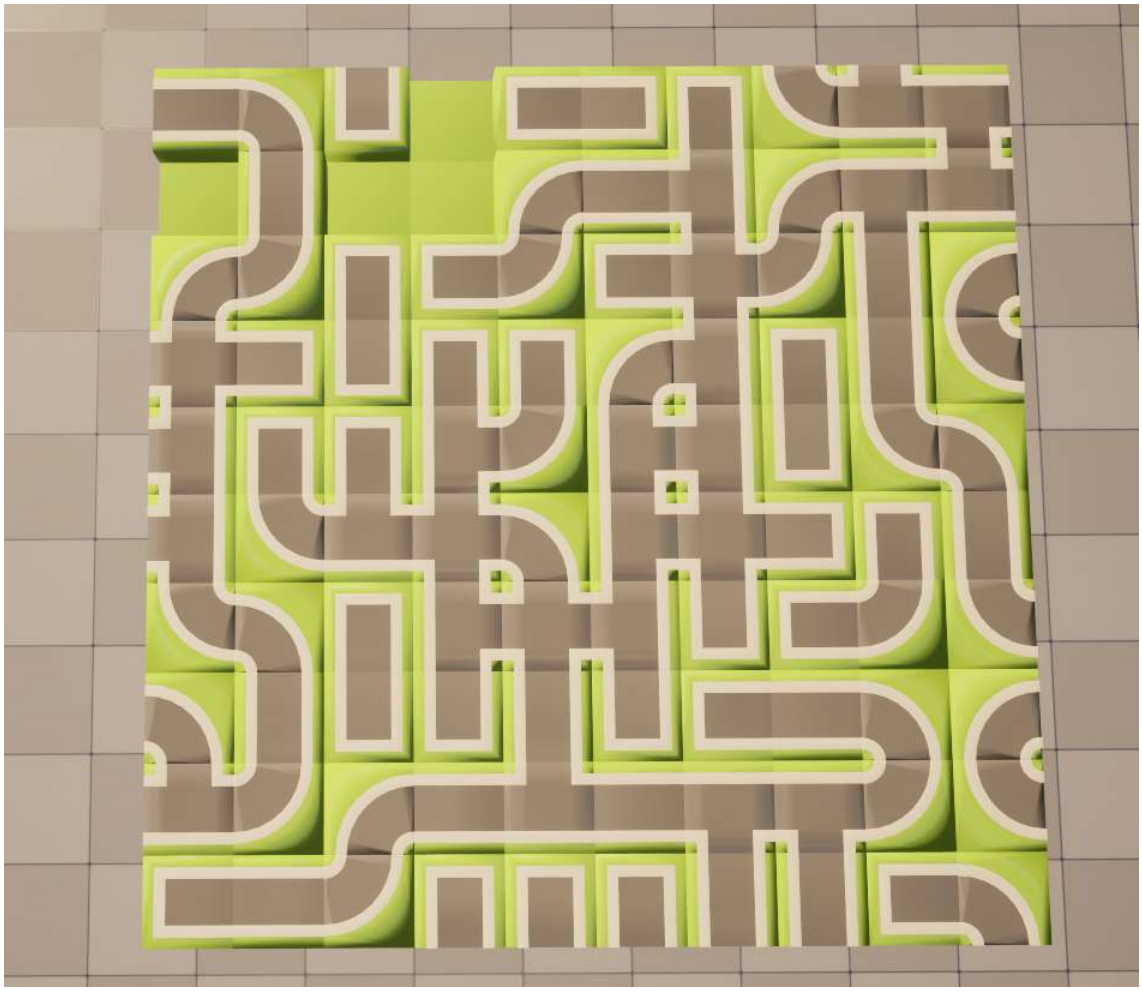


Figura 5.19: Exemplo de saída do DAC Diagonal

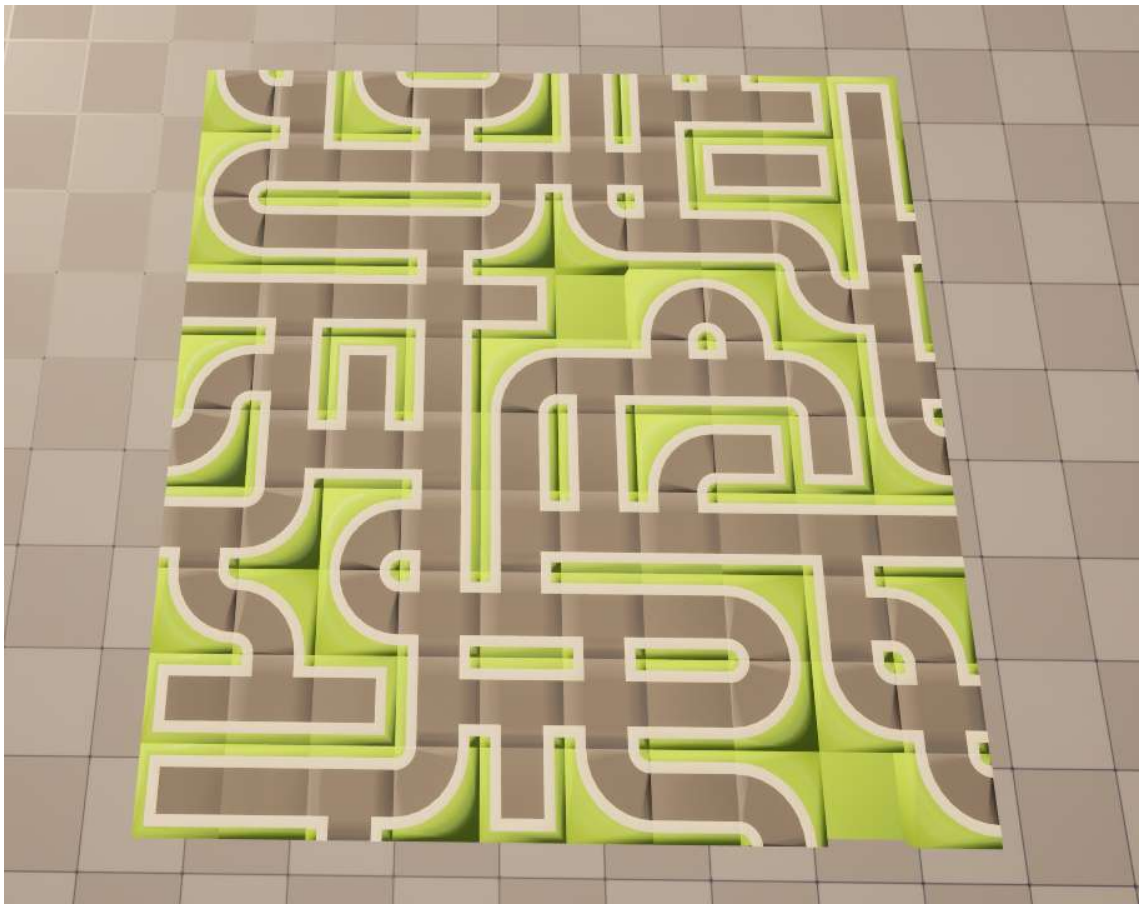


Figura 5.20: Exemplo de saída do DAC Iterativo

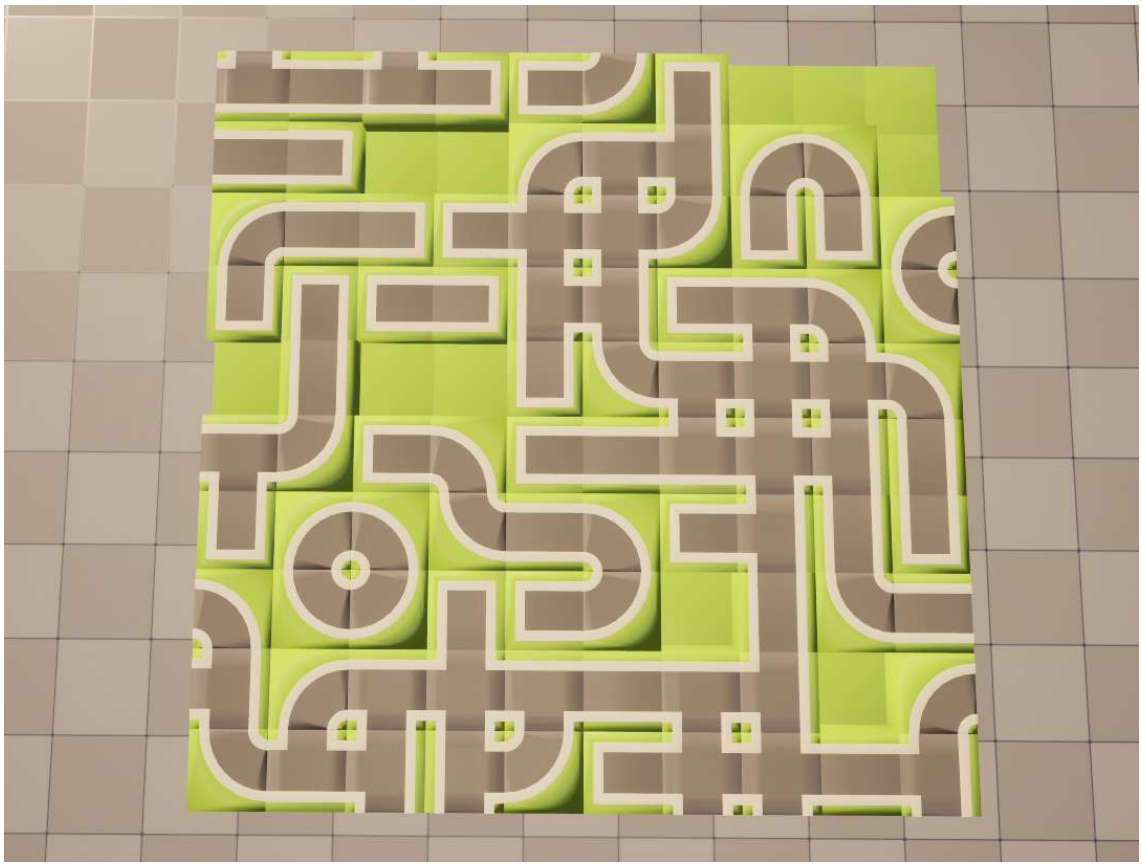


Figura 5.21: Exemplo de saída do WFC Diagonal

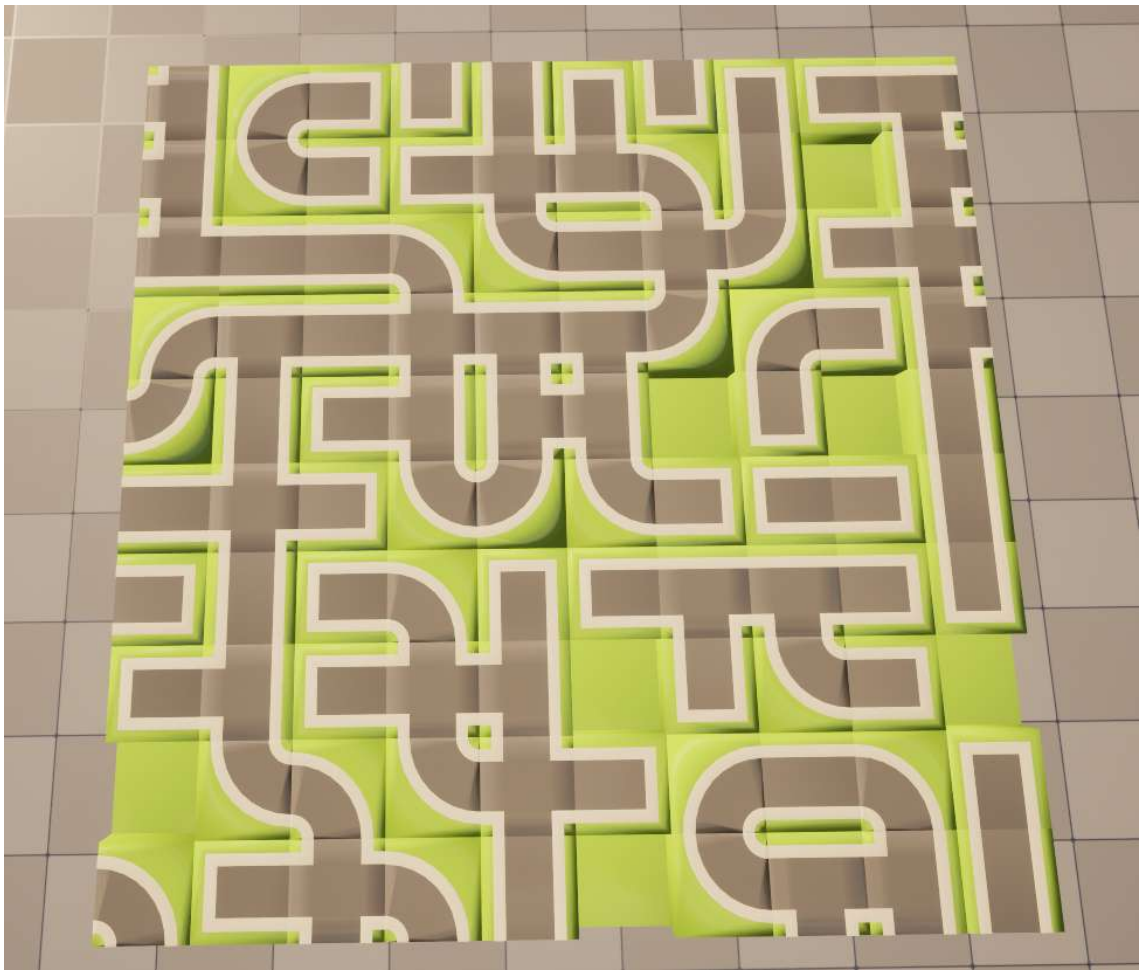


Figura 5.22: Exemplo de saída do WFC MRV

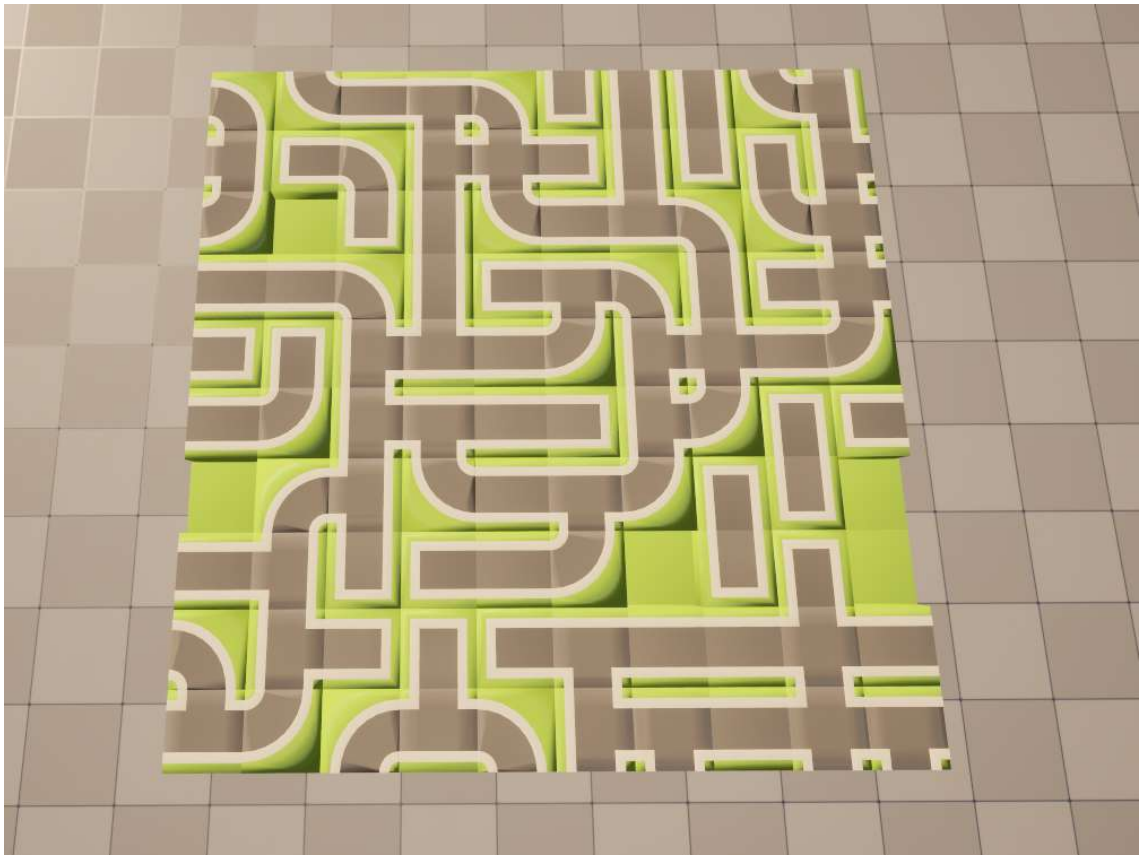


Figura 5.23: Exemplo de saída do NWFC

6 CONCLUSÃO

Neste trabalho, foi feita a implementação em Unreal Engine 5.4 e em C++ dos algoritmos WFC, N-WFC e DAC para a geração procedural de terrenos bidimensionais.

Devido aos testes realizados foi possível investigar e confirmar a capacidade do N-WFC e DAC de serem mais eficientes que o WFC Simple Tile original. Também foi possível identificar as limitações do DAC e do N-WFC, ambos relacionados a problemas na geração com *Tilesets* Incompletos.

Os resultados para Tilesets subcompletos o DAC implementado em C++ é mais eficaz, porém os testes na Unreal Engine indicam que o N-WFC é mais eficaz, pois o algoritmo esta disputando recursos com outros processos, como a memoria cache.

Em C++ também foram feitas análises qualitativas com base nas saídas geradas, não foi possível perceber nenhuma mudança visual com base no algoritmo utilizado, os resultados costumam gerar resultado semelhantes.

Para Tilesets Incompletos foi possível concluir que o N-WFC é mais eficaz, porém ainda é preciso de mais estudo, pois podem existir estrategias diferentes de como lidar com esses tiles Incompletos, como foi feito na implementação do N-WFC em C++, isto é, a estratégia de expandir o *subgrid* para descartar as extremidades que podem gerar erros na etapa de sobreposição do N-WFC.

Pesquisas futuras podem se concentrar em expandir e testar outras variantes, além de experimentar metodos para tornar tilesets incompletos em tilesets subcompletos e completos com pré-processamento e inicialização dos dominios, como o tileset da figura 5.1, com ele podemos inicializar a borda da matriz com todos os tiles e o meio da matriz mantémos apenas os tiles subcompletos, completos.

REFERÊNCIAS

Efros, A. A. e Leung, T. K. (1999). Texture synthesis by non-parametric sampling. *Proceedings IEEE International Conference on Computer Vision*.

Gumin, M. (2016). Wavefunctioncollapse. `https://github.com/mxgmn/WaveFunctionCollapse`. Acessado em 26/05/2025.

Kubi, G. (2021). Road tiles. `https://kubigames.itch.io/road-tiles`. Acessado em 18/06/2025.

Market.US (2024). Global generative ai in gaming market. Relatório Market.US. Disponível em: `https://market.us/report/generative-ai-in-gaming-market/`.

Mehta, N. (2025). The role of ai in game development and player experience. *SSRN Electronic Journal*.

Merrell, P. C. (2021). Comparing model synthesis and wave function collapse.

NIE, Y., ZHENG, S., ZHUANG, Z. e SONG, X. (2023). Extend wave function collapse algorithm to large-scale content generation. *arXiv preprint arXiv:2308.07307*.

Rose, T. e Bakaoukas, A. (2016). Algorithms and approaches for procedural terrain generation - a brief review of current techniques.

Rossi, F., van Beek, P. e Walsh, T., editores (2006). *Handbook of Constraint Programming*. Elsevier.

Russell, S. e Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*. Prentice Hall.

Shaker, N., Togelius, J. e Nelson, M. J. (2016). *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer.

Smith, G., Whitehead, J. e Mateas, M. (2011). Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3:201 – 215.

Togelius, J., Yannakakis, G. N., Stanley, K. O. e Browne, C. (2011). Search-based procedural content generation. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186.