UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LETÍCIA FREIRE CARVALHO DE SOUSA

UMA IMPLEMENTAÇÃO DO ALGORITMO DE *PATCH FITTING* PARA SÍNTESE DE TEXTURAS USANDO O S-T-CORTE MÍNIMO EM REDES PLANARES

RIO DE JANEIRO
2024

LETÍCIA FREIRE CARVALHO DE SOUSA

UMA IMPLEMENTAÇÃO DO ALGORITMO DE *PATCH FITTING* PARA SÍNTESE DE TEXTURAS USANDO O S-T-CORTE MÍNIMO EM REDES PLANARES

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Profa. Dra. Márcia Rosana Cerioli

RIO DE JANEIRO

2024

CIP - Catalogação na Publicação

S725i

Sousa, Letícia Freire Carvalho de
Uma Implementação do Algoritmo de Patch Fitting Para Síntese de Texturas Usando o s-t-Corte Mínimo em Redes Planares / Letícia Freire Carvalho de Sousa. - Rio de Janeiro, 2024.
32 f.

Orientador: Cerioli Márcia Rosana.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2024.

1. algoritmo. 2. corte mínimo em grafos. 3. grafos planares. 4. síntese de texturas. I. Márcia Rosana, Cerioli, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

LETÍCIA FREIRE CARVALHO DE SOUSA

UMA IMPLEMENTAÇÃO DO ALGORITMO DE *PATCH FITTING* PARA SÍNTESE DE TEXTURAS USANDO O S-T-CORTE MÍNIMO EM REDES PLANARES

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 19 de dezembro de 2024

BANCA EXAMINADORA:

Documento assinado digitalmente
gov.br
MARCIA ROSANA CERIOLI
Data: 24/02/2025 14:54:24-0300
Verifique em https://validar.iti.gov.br

Márcia Rosana Cerioli
D.Sc. (UFRJ)

Documento assinado digitalmente
gov.br
CLAUDSON FERREIRA BORNSTEIN
Data: 24/02/2025 16:56:09-0300
Verifique em https://validar.iti.gov.br

Claudson Ferreira Bornstein
PhD. (UFRJ)

Documento assinado digitalmente
gov.br
MITRE COSTA DOURADO
Data: 24/02/2025 15:18:38-0300
Verifique em https://validar.iti.gov.br

Mitre Costa Dourado
D.Sc. (UFRJ)

Dedico este trabalho aos meus pais e ao meu irmão pelo apoio durante toda a minha vida, e aos meus amigos, que me acompanharam durante a minha graduação. Em especial, aos que fizeram parte comigo da Maratona SBC de Programação, que foi essencial para a escolha do tema deste trabalho.

AGRADECIMENTOS

Gostaria de expressar meus agradecimentos ao Conselho Nacional do Desenvolvimento Científico e Tecnológico - CNPq pelo apoio fornecido a minha Iniciação Científica cujo tema evoluiu para o presente trabalho.

Agradeço também à minha orientadora Márcia Cerioli pelo apoio durante toda a graduação e por coordenar o Projeto Competições de Algoritmos e Programação, pelo qual conheci as competições de programação que me levaram a estudar cortes em grafos planares.

RESUMO

Síntese de texturas é o processo de construir de forma algorítmica imagens digitais grandes a partir de imagens digitais pequenas. Uma das formas de realizar tal síntese é o algoritmo de *patch fitting* em que, iterativamente, são copiados pedaços de uma imagem original na imagem destino. A cópia de cada um dos demais pedaços é dividida em duas etapas: *matching* e *blending*.

Na etapa de *matching* um pedaço retangular da imagem original é posicionado na imagem destino de forma que ele se sobreponha com pedaços já posicionados anteriormente. Na etapa de *blending*, um pedaço irregular desse retângulo é escolhido, de forma que ele se misture bem com os pedaços copiados anteriormente, e somente este pedaço é copiado na imagem destino.

Neste trabalho, estudamos detalhadamente o algoritmo, proposto em (REIF, 1983), mais aplicado atualmente para $s-t-$corte mínimo planar e como realizar a etapa de *blending* utilizando o $s-t-$corte mínimo em redes planares, conforme proposto em (KWATRA et al., 2003). Também apresentamos e implementamos sua aplicação na etapa de *blending* do algoritmo de *patch fitting* para síntese de texturas.

Por fim, analisamos as texturas geradas pela nossa implementação no caso de texturas com padrões regulares e no caso de texturas sem tais padrões.

Palavras-chave: algoritmo; corte mínimo em grafos; grafos planares; síntese de texturas.

ABSTRACT

Texture synthesis is the process of algorithmically constructing large digital images from small digital images. One approach to perform such synthesis is the *patch fitting* algorithm, where patches from the original image are iteratively copied onto the target image. The copying of each subsequent patch is divided into two stages: *matching* and *blending*. In the *matching* stage, a rectangular patch from the original image is positioned onto the target image such that it overlaps with previously positioned patches. In the *blending* stage, an irregular subregion of this rectangle is selected to blend well with the previously copied patches, and only this subregion is copied onto the output image.

In this work, we study in detail the most widely applied algorithm for $s - t$ minimum planar cut, proposed in (REIF, 1983), and how to perform the *blending* stage using $s - t$ minimum cuts in planar networks, as proposed in (KWATRA et al., 2003). We also present and implement its application in the *blending* step of the *patch fitting* algorithm for texture synthesis.

Finally, we analyze the textures generated by our implementation in the case of textures with regular patterns and textures without such patterns.

Keywords: algorithm; minimum cut on graphs; planar graphs; texture synthesis.

SUMÁRIO

1 INTRODUÇÃO 8
1.1 MOTIVAÇÃO 8
1.2 ALGORITMO DE *PATCH FITTING* 8
1.3 *S*-*T*-CORTE MÍNIMO EM REDES PLANARES 9
1.4 OBJETIVOS 9

2 TEORIA 11
2.1 CORTE MÍNIMO E FLUXO MÁXIMO 11
2.2 OUTROS ALGORITMOS PARA ENCONTRAR O FLUXO MÁXIMO 12
2.3 REDES PLANARES 13
2.4 CICLO SEPARADOR 13
2.5 DUALIDADE ENTRE CICLO SEPARADOR E *S*-*T*-CORTE 14
2.6 CICLO SEPARADOR *F*-MÍNIMO 15
2.7 CASOS ESTUDADOS 15
2.7.1 Caso 1 15
2.7.2 Caso 2 17

3 APLICAÇÃO NO ALGORITMO DE *PATCH FITTING* . . . 22

4 IMPLEMENTAÇÃO 23
4.1 RESULTADOS 26
4.1.1 Texturas Sem Padrões Regulares 26
4.1.1.1 Grãos de Areia 26
4.1.1.2 Grãos de Café 27
4.1.1.3 Tecido Jeans 27
4.1.2 Texturas Com Padrões Regulares 27
4.1.2.1 Muro de Tijolos 28
4.1.2.2 Mosaico de Burle Marx 29

5 CONCLUSÃO E TRABALHOS FUTUROS 31

REFERÊNCIAS 32

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

A repetição de padrões gera os mais lindos cenários, tanto naturais quanto feitos pelo ser humano. Percebemos isso na praia de Copacabana, com seus milhares de grãos de areia formando a faixa de areia na beira do mar, e também no mosaico das ondas do calçadão projetado por *Burle Marx*.

Figura 1 – Calçadão de Copacabana

Em visão computacional, textura é uma imagem composta por padrões repetidos. Porém, assim como dois grãos de areia não são iguais e pequenas imperfeições tornam cada onda do mosaico de *Burle Marx* única, as repetições dos padrões não precisam ser idênticas na definição de textura.

A síntese de texturas é o processo de gerar de forma algorítmica imagens digitais grandes a partir de imagens digitais pequenas. Uma linha de pesquisa bem-sucedida para manipulação de imagens com base nos objetivos do usuário é a síntese de texturas baseada em *patches* (BARNES; ZHANG, 2017). Um exemplo de uso da síntese de texturas são os "pincéis" de programas de *design* digital (BARNES; ZHANG, 2017). Nesses programas, os usuários podem escolher texturas para seus pincéis a partir de imagens de um banco de dados em vez de desenharem utilizando apenas cores sólidas.

1.2 ALGORITMO DE *PATCH FITTING*

O algoritmo de *patch fitting* para síntese de texturas consiste em copiar, iterativamente, pedaços de formato irregular de uma imagem original para uma imagem destino. Uma analogia para o algoritmo é a confecção de uma colcha de retalhos, em que cada pedaço copiado é um retalho e queremos costurá-los de forma com que as costuras fiquem o menos aparentes possível, mesclando os padrões.

No algoritmo, um primeiro pedaço retangular é copiado da imagem origem para a imagem destino. Após, a cópia de cada um dos demais pedaços é dividida em duas etapas. Na primeira, chamada de *matching*, um pedaço retangular da imagem original é posicionado na imagem destino de forma que ele se sobreponha com pedaços já posicionados anteriormente. Na segunda etapa, o *blending*, um pedaço irregular desse retângulo é escolhido, de forma que ele se misture bem com os pedaços copiados anteriormente, e somente este pedaço é copiado na imagem destino.

1.3 *S*-*T*-CORTE MÍNIMO EM REDES PLANARES

Um *grafo* $G = (V, E)$ é dado por um conjunto não vazio V, de *vértices*, e um conjunto E de *arestas*, onde cada aresta é um par não ordenado de vértices distintos. Uma *rede* $N = (G, c)$ é um grafo conexo $G = (V, E)$ com uma função c de E nos reais positivos, com dois vértices especiais e distintos s e t.

Uma rede $N = (G, c)$ é planar se o grafo subjacente a G é planar, isto é, pode ser desenhado no plano sem cruzamento de arestas.

Um *s-t-corte* em uma rede N é um subconjunto $S \subseteq E$ tal que não há caminho entre os vértices s e t no grafo $H = (V, E \setminus S)$. O *custo* de S é dado por $\sum c(e), \forall e \in S$. Um s-t-corte é *mínimo* se não há em N algum s-t-corte com custo menor que ele (CORMEN et al., 2009).

O problema do s-t-corte mínimo em uma rede é dual do problema do fluxo máximo. Assim, em redes gerais podemos resolvê-lo com algoritmos que encontram um fluxo máximo e seu valor (FORD; FULKERSON, 1956). Existem, na literatura, vários algoritmos eficientes que resolvem o problema do fluxo máximo, e consequentemente, do s-t-corte mínimo, cada um com suas propriedades específicas, e consequentemente, com suas utilidades específicas. Por serem, em geral, utilizados com muita frequência, detalhes na complexidade de tempo podem ser de relevância prática e, com isto, a área tem tido um grande desenvolvimento e impacto nas últimas décadas.

Em redes planares, um s-t-corte mínimo pode ser encontrado de forma ainda mais eficiente, utilizando outra dualidade: um s-t-corte mínimo corresponde a um ciclo de corte no grafo dual.

1.4 OBJETIVOS

Apresentar os algoritmos mais utilizados na prática para o problema do s-t-corte mínimo. Apresentamos os algoritmos que encontram um corte mínimo através da dualidade corte mínimo – fluxo máximo e suas complexidades e consideramos suas aplicações ao problema do s-t-corte mínimo em redes planares. Depois, o caso de redes planares é considerado, de forma a elucidar as vantagens que uma rede particular deste tipo favorece os algoritmos já existentes de fluxo máximo. A aplicação da dualidade do ciclo separador

com corte mínimo da rede planar trouxe um desafio a ser transposto no caso de grafos planares, dada pela situação causada pela representação planar dada e a localização dos vértices s e t em relação a ela.

Além disto, o problema da síntese de texturas e sua solução computacional é considerado, com implementação das melhores soluções algorítmicas encontradas para a etapa de *blending* do *patch fitting*, sendo que esta última utiliza o s-t-corte mínimo em redes planares. A implementação por nos desenvolvida está disponibilizada no formato de código aberto para que todos possam lê-la e utilizá-la. Ela está disponível no seguinte endereço: https://github.com/LeticiaFCS/ GraphCutTextureSynthesis.

2 TEORIA

2.1 CORTE MÍNIMO E FLUXO MÁXIMO

Um *fluxo* em uma rede $N = (G, c)$, em que G é direcionado, com dois vértices especiais s e t é uma função f de E nos reais não negativos, de forma que $f(e) \leq c(e),\ \forall e \in E$. Além disso, seja $f_+(v)$ a soma do fluxo nas arestas que saem de v e seja $f_-(v)$ a soma do fluxo nas arestas que entram em v, a função f deve ser tal que $f_+(v) = f_-(v)$, se $v \neq s$ e $v \neq t$, e $f_-(s) = f_+(t) = 0$. O *valor do fluxo* na rede N será o valor de $f_+(s) = f_-(t)$ e denotaremos tal valor por $f(N)$. Dizemos que f é *máximo* se não há fluxo em N de valor maior que $f(N)$.

Dado um fluxo f em uma rede N, uma aresta e é *saturada* se $f(e) = c(e)$. Um caminho de s a t é chamado de *caminho aumentante* quando nenhuma de suas arestas está saturada. Uma rede residual $N'(f)$ é criada a partir de $N = (G, c)$, $G = (V, E)$ e de um fluxo f da seguinte forma: O conjunto de vértices de $N'(f)$ é V e para cada aresta $(u, v) \in E$, criamos em $N'(f)$ a aresta (u, v) (aresta direta) com capacidade $c(u, v) - f(u, v)$ se ela não está saturada, e a aresta (aresta contrária) (v, u) com capacidade $f(u, v)$, no caso de haver algum fluxo positivo em (u, v).

Ford e Fulkerson propuseram o seguinte algoritmo para encontrar um fluxo máximo (FORD; FULKERSON, 1956):

Algoritmo 1 Algoritmo de Ford-Fulkerson para Encontrar Fluxo Máximo

Entrada: Rede $N = (G, c)$, com capacidade $c : E \longrightarrow \mathbb{R}^+$ e vértices especiais s e t.
Saída: f, um Fluxo Máximo em N, e F ser valor

```
Seja f função de E nos reais não negativos, com f(e) = 0, ∀e ∈ E(G).
Seja R = N'(f)
Seja F = 0, valor do fluxo que passa por R
enquanto há caminho aumentante P em R faça
    Seja m = min{c(e) | e ∈ P}
    F = F + m
    para todo e ∈ P faça
        se e é direta então
            f(e) = f(e) + m
        senão
            f(e) = f(e) − m
        fim se
    fim para
    R = N'(f)
fim enquanto
```

Seja F o valor do fluxo máximo na rede, no caso em que as capacidades são números inteiros, o algoritmo de Ford-Fulkerson tem complexidade de tempo $O(|V| \times F)$, no en-

tanto, há casos em que com capacidades não inteiras, o procedimento pode não levar ao valor correto do fluxo máximo (FORD; FULKERSON, 1956).

A corretude do algoritmo de Ford-Fulkerson é dado pela prova do célebre Teorema de Ford e Fulkerson (FORD; FULKERSON, 1956) que estabelece:

Teorema [Ford e Fulkerson, 1956] Em toda rede, o valor do fluxo máximo é igual a capacidade do s-t-corte mínimo.

Pela dualidade dos problemas de corte mínimo e fluxo máximo estabelecidas pelo teorema acima, podemos resolver o problema de encontrar um corte mínimo da seguinte forma: encontre um fluxo máximo e faça uma busca em grafos visitando vértices a partir de s na última rede residual R construída pelo algoritmo. As arestas no corte mínimo serão as arestas de G que saem de vértices visitados e entram em vértices não visitados na busca.

2.2 OUTROS ALGORITMOS PARA ENCONTRAR O FLUXO MÁXIMO

O algoritmo de Ford-Fulkerson para encontrar o fluxo máximo em uma rede não especifica como deve ser feita a busca pelo caminho aumentante a cada iteração. Uma melhoria simples para o algoritmo, proposta por Edmonds e Karp, em 1972, (EDMONDS; KARP, 1972), consiste em realizar a busca pelo caminho aumentante utilizando o procedimento de **busca em largura**. Dessa forma, cada caminho aumentante encontrado terá pelo menos o mesmo número de arestas do caminho encontrado na iteração anterior e garantimos que cada aresta é a aresta de menor capacidade em um caminho aumentante escolhido no máximo $O(|E||V|)$ vezes. Logo, o algoritmo encontra o fluxo máximo na complexidade de tempo $O(|E|^2|V|)$. Vale ressaltar que o algoritmo sempre termina, mesmo com as arestas tendo números reais como capacidades.

De forma independente, Dinitz também propôs utilizar a busca em largura para encontrar caminhos aumentantes na ordem crescente de número de arestas (DINIC, 1970). Porém, a grande diferença do algoritmo de Dinitz é que, ao realizar a busca em largura a partir de s ele armazena em uma estrutura chamada rede em camadas, todos os caminhos de s a t de um mesmo tamanho. para encontrar um caminho aumentante P, encontrando todos os caminhos aumentantes de tamanho $|P|$. Desta forma, o algoritmo de Dinitz roda com a complexidade de tempo $O(|E||V|^2)$.

Como os algoritmos de Edmonds-Karp e de Dinitz são basicamente o mesmo algoritmo de Ford-Fulkerson, sua corretude é obtida pelo mesmo Teorema de Ford-Fulkerson.

Outros algoritmos para fluxo máximo são também conhecidos (CORMEN et al., 2009) e foram também estudados neste período porém não foram considerados de relevância para a implementação da resolução do problema em grafos planares e, consequentemente, para o problema da síntese de texturas.

2.3 REDES PLANARES

Dada uma representação planar de uma rede não direcionada planar $N = (G, c)$, $G = (V, E)$, o *dual* de N, $D(N) = (D(G), D(c))$, onde $D(G) = (V', E')$ é construído da seguinte forma:

Associe um vértice a cada face de G, isto é, V' é o conjunto das faces da representação planar de G e, para cada aresta e de custo k que está em duas faces f_i e f_j da representação planar de G associamos uma aresta de custo k em E' entre os vértices correspondentes a f_i e f_j. Na figura a seguir temos uma ilustração de como este processo é aplicado para a criação de uma única aresta,

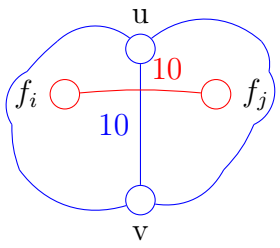


Figura 2 – Em vermelho, exemplo de aresta em $D(N)$, e sua capacidade

Abaixo temos a rede original N em azul e em vermelho $D(N)$. Omitimos os pesos das arestas em $D(N)$ para melhorar a sua visualização.

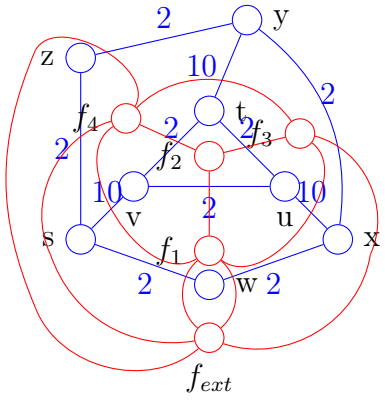


Figura 3 – Redes N em azul e $D(N)$ em vermelho

2.4 CICLO SEPARADOR

Um ciclo C em $D(N)$ divide N em duas partes, N_{int}, a subrede de N induzida pelos vértices que estão na parte interna de C e N_{ext}, a subrede de N induzida pelos vértices que estão na parte externa de C. Dizemos que C é *ciclo separador* se s está em N_{int} e t está em N_{ext} ou vice-versa. O custo de C é dado pela soma dos custos das arestas em C.

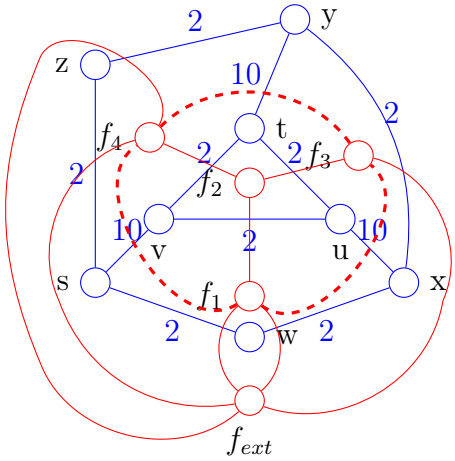


Figura 4 – Em pontilhado um ciclo separador de custo 30 em $D(N)$

2.5 DUALIDADE ENTRE CICLO SEPARADOR E S-T-CORTE

Há, dada diretamente pela construção da rede dual, uma correspondência um entre os ciclos separadores e os s-t-cortes e esta correspondência mantém os pesos das arestas. Logo um ciclo separador mínimo é dual de um s-t-corte mínimo.

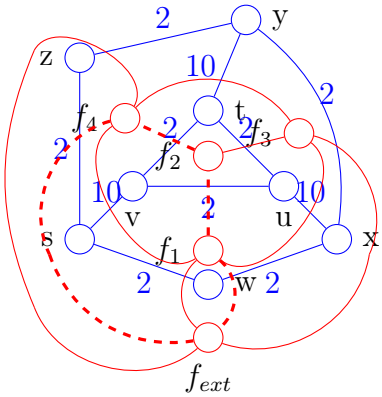


Figura 5 – Em pontilhado um ciclo separador mínimo em $D(N)$

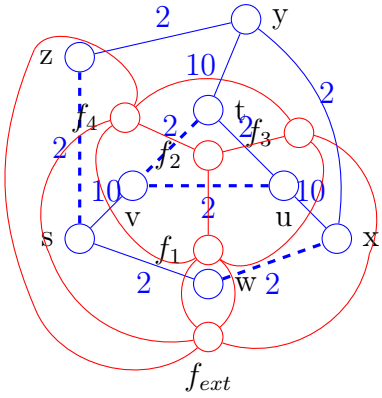


Figura 6 – Em pontilhado um s-t- corte mínimo em N

2.6 CICLO SEPARADOR F-MÍNIMO

Seja f uma face de N. Um ciclo separador C é dito *f-mínimo* se é o ciclo separador de menor custo que contém o vértice correspondente a f de $D(N)$.

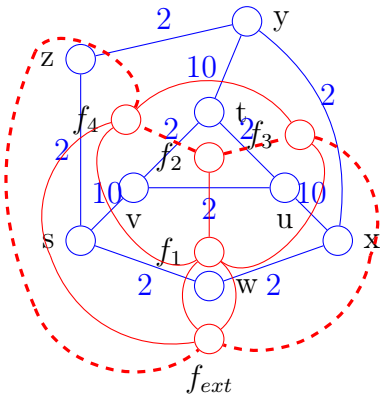


Figura 7 – Em pontilhado um ciclo separador f_3-mínimo.

2.7 CASOS ESTUDADOS

O problema do corte mínimo pode ser resolvido de forma mais eficiente em grafos planares utilizando a dualidade entre o problema do s-t-corte mínimo e o problema de encontrar o ciclo separador mínimo.

Estudamos dois casos para o s-t-corte mínimo em redes planares: o caso 1 em que s e t estão na mesma face de uma representação planar de N e o caso 2, em que eles estão em faces diferentes.

2.7.1 Caso 1

O caso em que s e t estão em uma mesma face f_i em N é mais simples.

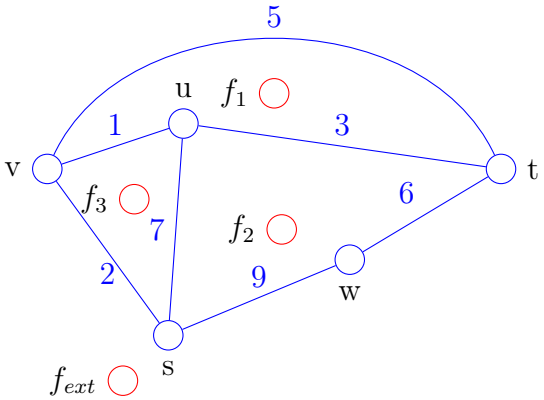


Figura 8 – Rede N, com s e t na mesma face f_2

Seja $H = N + \{e\}$, onde e é aresta entre s e t com custo infinito, isto é, um valor grande o suficiente para ser maior que todos os outros valores. Sabemos que a aresta e divide f_i em duas faces f_i' e f_i''.

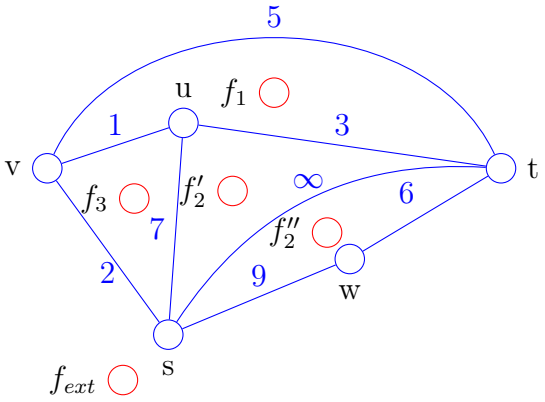


Figura 9 – Rede $H = N + \{e\}$

Vamos mostrar que um caminho mínimo entre os vértices correspondentes a f_i' e f_i'' em $D(H)$ corresponde a um s-t-corte mínimo em N.

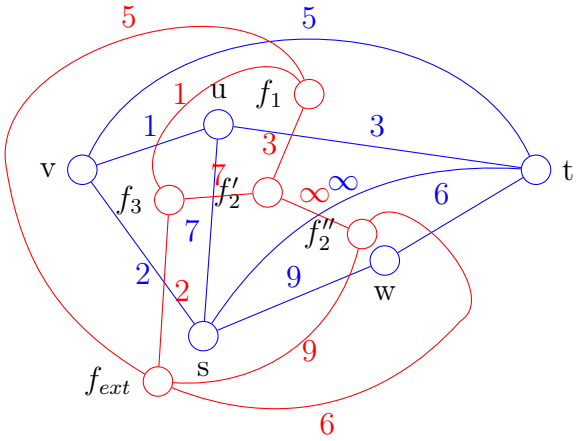


Figura 10 – Redes H em azul e $D(H)$ em vermelho

Um caminho mínimo entre os vértices correspondentes a f_i' e f_i'' em $D(H)$ não contém a aresta e pois ela tem peso infinito.

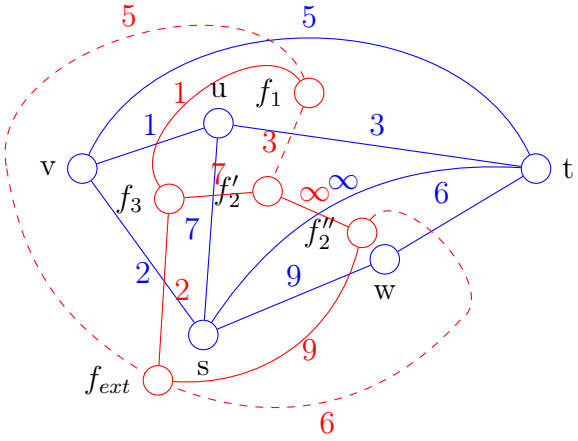


Figura 11 – Em pontilhado, caminho mínimo P entre f_2' e f_2'' em $D(H)$

Contraindo os vértices correspondentes a f_i' e f_i'' em $D(H)$ fica evidente que P corresponde a um ciclo mínimo em $D(N)$ que contém s em N_{int} e t em N_{ext} (ou que contém t em N_{int} e s em N_{ext}), caso contrário P não seria mínimo.

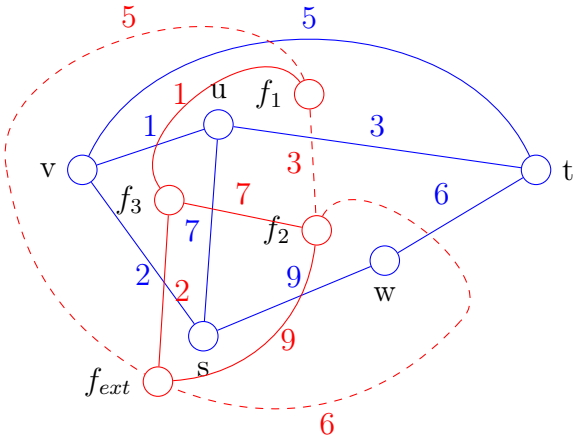


Figura 12 – Em pontilhado, ciclo separador mínimo corresponde a P em $D(N)$.

Logo, P corresponde a um s-t-corte mínimo em N.

Seja n o número de vértices em uma rede planar, o algoritmo de Dijkstra (DIJKSTRA, 1959) para encontrar caminhos mínimos possui complexidade de tempo $O(n \log(n))$.

Logo, no caso 1, é possível achar o s-t- corte mínimo com complexidade de tempo $O(n \times \log(n))$.

2.7.2 Caso 2

Seja S o conjunto de vértices em $D(N)$ correspondentes as faces que contém s e T o conjunto de vértices em $D(N)$ correspondentes as faces que contém t.

Seja P um caminho mínimo entre um vértice de S e um vértice de T.

Sejam $(f_0, f_1, f_2, \ldots, f_k)$ as faces que P atravessa, nesta ordem.

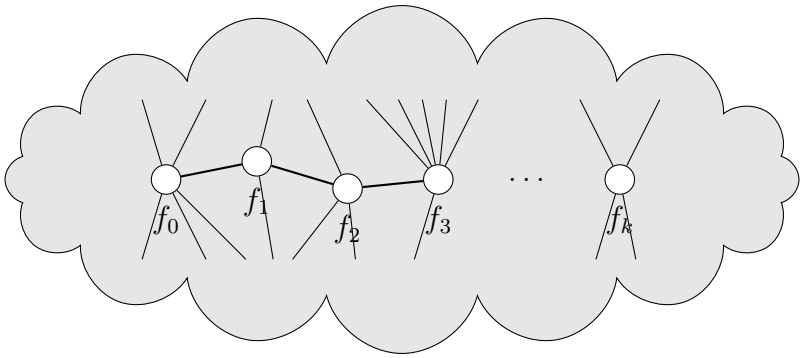


Figura 13 – Caminho P em uma rede $D(N)$

Podemos ver P como uma linha horizontal com s na esquerda e t na direita. Criamos uma nova rede $D'(N)$ dividindo cada vértice f_i em P em dois vértices fc_i e fb_i.

As arestas incidentes a f_i por cima da linha serão incidentes a fc_i e as arestas incidentes por baixo serão incidentes a fb_i.

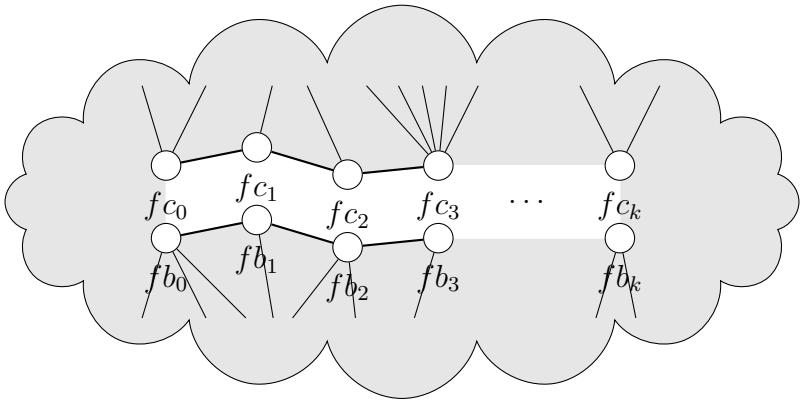


Figura 14 – Caminho P dividido em $D'(N)$

Um caminho **mínimo** entre fc_i e fb_i corresponde a um ciclo separador f_i-mínimo.

Se houvesse um ciclo separador menor que contém fb_i e fc_i haveria uma contradição com o fato do caminho ser mínimo.

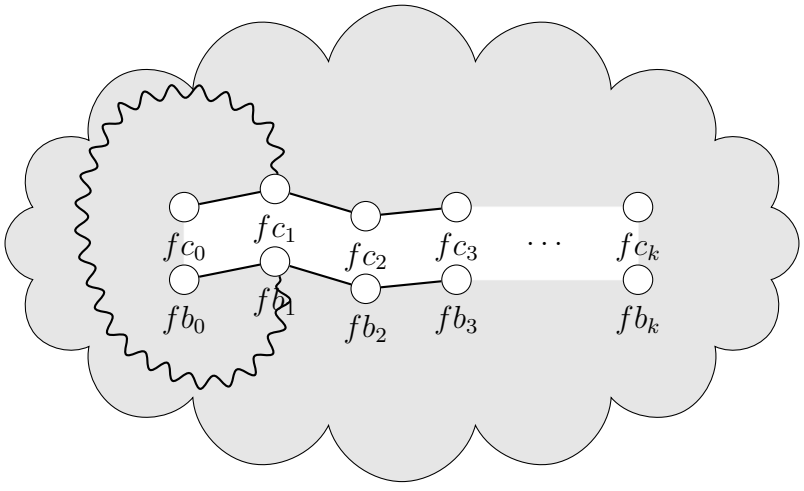


Figura 15 – Caminho P dividido em $D'(N)$. Em destaque ciclo separador f_1-mínimo

Um primeiro algoritmo para achar um s-t- corte mínimo é:

- Achar caminho mínimo P entre uma face que s está e uma face que t está.
- Criar rede $D'(N)$.
- Achar um caminho mínimo entre fb_i e fc_i para toda f_i pela qual P passa e selecionar o menor entre eles.

Tal algoritmo pode ser implementado em $O(n^2 \log(n))$, onde n é o número de vértices em N.

Sejam f_i e f_j faces em P tal que $i < j$. Se há ciclo separador f_j-mínimo C que contém s então há ciclo separador f_i-mínimo contido na região delimitada por C.

Da mesma forma, se há ciclo separador f_i-mínimo C' que contém t então há ciclo separador f_j-mínimo contido na região delimitada por C'.

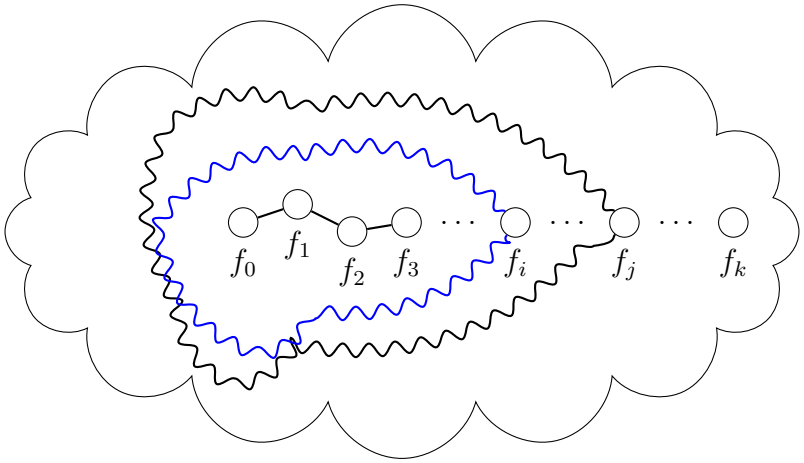


Figura 16 – Caminho P dividido em $D'(N)$. Em destaque ciclos de corte f_i-mínimo e f_j-mínimo

Dado um s-t-corte C em N podemos remover as arestas de C e dividir a rede resultante em duas redes N_s e N_t em que nenhum vértice em N_s tem caminho para t e nenhum vértice em N_t tem caminho para s.

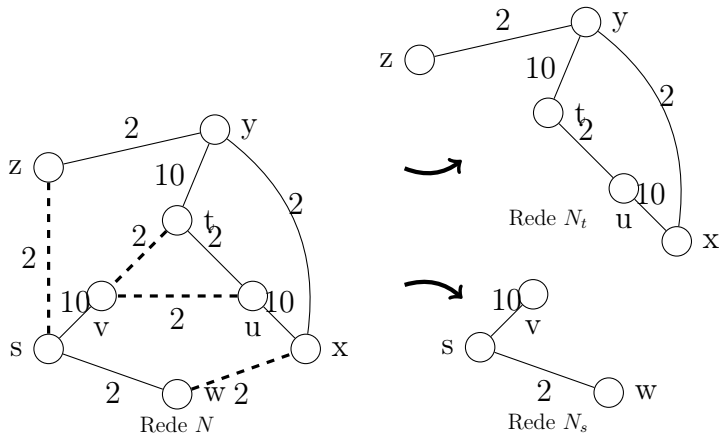


Figura 17 – Rede N e sua divisão em N_s e N_t

Criamos a rede N'_s acrescentando em N_s um novo vértice t', e para cada aresta e de custo k em c, acrescentamos uma aresta em N'_s entre a ponta de e em N_s e o novo vértice t'.

Caso N'_s tenha arestas múltiplas entre dois vértices, junte todas em uma só aresta cujo custo é a soma dos custos das demais. Dessa forma, N'_s também será uma rede não direcionada planar.

De forma análoga, criamos a nova rede N'_t.

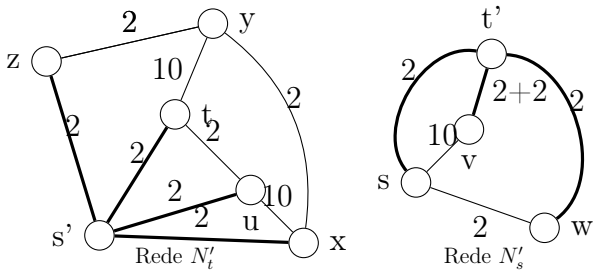


Figura 18 – Redes N'_s e N'_t

O s-t-corte mínimo corresponde a um ciclo separador f mínimo para alguma face f em N'_s ou N'_t.

Consideramos que temos implementados os seguintes algoritmos:

- $CORTE\_MINIMO\_CASO\_1(N, s, t)$: Computa um s-t-corte mínimo para uma rede em que s e t estão na mesma face.
- $CAMINHO\_MINIMO(N, u, v)$: Computa um caminho mínimo entre u e v.
- $CICLO\_SEPARADOR\_F\_MINIMO(N, f)$: Computa um ciclo separador f mínimo em N.
- $DIVIDE(N, c)$: Dado o ciclo separador c retorna as redes N'_s e N'_t conforme especificadas anteriormente.

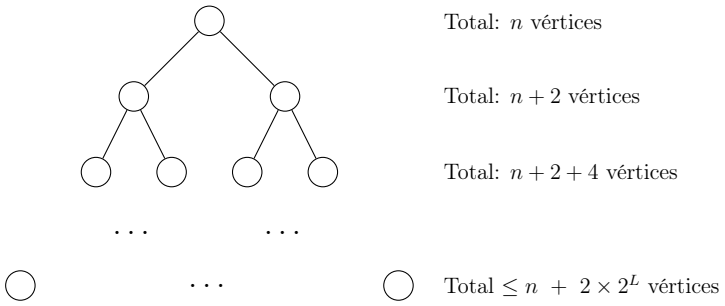
O $CORTE\_MINIMO\_CASO\_2(N, s, t)$ é implementado da seguinte forma:

Algoritmo 2 $CORTE\_MINIMO\_CASO\_2(N, s, t)$

Entrada: Rede N, P caminho mínimo em $D(N)$ entre uma face que contém s e uma face que contém t
Saída: Corte s-t-mínimo

```
1: função CORTE_MINIMO_CASO_2(N, s, t)
2:     Sejam (f_0, f_1, f_2, ..., f_k) as faces que P atravessa, nesta ordem.
3:     se k = 1 então
4:         retorna CORTE_MINIMO_CASO_1(N, s, t)
5:     senão
6:         c ← CICLO_SEPARADOR_F_MINIMO(N, f_⌊k/2⌋)
7:         N'_s, N'_t = DIVIDE(N, c)
8:         Sejam P_s e P_t os subcaminhos de P em N'_s e N'_t respectivamente.
9:         c_1 ← CORTE_MINIMO_CASO_2(N'_s, s, t')
10:        c_2 ← CORTE_MINIMO_CASO_2(N'_t, s', t)
11:        se o custo de c_1 for menor que o custo de c_2 então
12:            retorna c_1
13:        senão
14:            retorna c_2
15:        fim se
16:    fim se
17: fim função
```

Seja n o número de vértices em N e seja k o número de vértices no caminho mínimo entre uma face que s está e uma face que t está. Analisando a árvore das chamadas recursivas de $CORTE\_MINIMO\_CASO\_2$ nível a nível, temos:



Seja L o número de níveis da árvore, sabemos que $L = \lceil \log(k) \rceil$, logo $L = O(\log(n))$.

Há $O(n + 2 \times 2^{\log(n)}) = O(n)$ vértices em cada nível.

Logo, é possível implementar o algoritmo $CORTE\_MINIMO\_CASO\_2(N, s, t, c)$ com complexidade tempo $O(n \log^2(n))$, utilizando o algoritmo de Dijkstra (DIJKSTRA, 1959) para encontrar os caminhos mínimos.

3 APLICAÇÃO NO ALGORITMO DE *PATCH FITTING*

O problema de encontrar um s-t-corte mínimo em uma rede planar possui aplicações em visão computacional como a segmentação de imagens e a síntese de texturas. Neste trabalho estudamos sua aplicação no algoritmo de *patch fitting* para síntese de texturas, conforme apresentado em (KWATRA et al., 2003).

No algoritmo, um primeiro pedaço retangular é copiado da imagem origem para a imagem destino. Após, a cópia de cada um dos demais pedaços é dividida em duas etapas.

Na primeira, chamada de *matching*, um pedaço retangular da imagem original é posicionado na imagem destino de forma que ele se sobreponha com pedaços já posicionados anteriormente. A forma mais rápida de realizar esta etapa, e que nós escolhemos seguir, é escolher o pedaço retangular como a imagem original toda e escolher de forma aleatória sua posição na imagem destino.

Na segunda, o *blending*, um pedaço irregular desse retângulo é escolhido, de forma que ele se misture bem com os pedaços copiados anteriormente, e somente este pedaço é copiado na imagem destino. Para escolher o formato do pedaço, o algoritmo analisa a interseção do pedaço selecionado com os pixels já preenchidos na imagem da saída.

Seja p o número de pixels na interseção. É criado um grafo planar G com $p+2$ vértices. Cada pixel na interseção corresponde a um vértice em G e, além desses, são criados dois vértices especiais s e t que correspondem aos pixels que não estão na interseção, sendo que s corresponde aos pixels que já existiam na imagem da saída na iteração anterior do algoritmo e t corresponde aos pixels do retângulo novo, que não estão na interseção.

Vértices que correspondem a pixels adjacentes são ligados por uma aresta. As arestas em que um de seus extremos é s ou t terão custo infinito. As demais arestas uv terão um custo $f(uv)$ que representa o quanto os pixels adjacentes são "parecidos" na imagem anterior e no retângulo copiado.

Utilizamos o sistema RGB em que os pixels são representados como vetores tridimensionais e cada posição representa quanto de vermelho, verde e azul temos no pixel.

Seja $A(u)$ o vetor do pixel u na imagem de saída na iteração antiga e $B(u)$ seu vetor no retângulo novo, uma função de custo simples e rápida de ser calculada é $f(uv) = |A(u) - B(u)| + |A(v) - B(v)|$ (KWATRA et al., 2003).

Seja C um s-t-corte mínimo em G, podemos visualizar C como uma curva em que s está à sua esquerda e t à sua direita. O pedaço do retângulo copiado para a imagem de saída conterá apenas os pixels à direita de C.

Como G é planar, podemos utilizar o algoritmo estudado para encontrar um s-t-corte mínimo em G na complexidade de tempo $O(p\log(p))$ ou $O(p\log^2(p))$, a depender de onde o retângulo foi encaixado.

4 IMPLEMENTAÇÃO

A implementação foi realizada na linguagem `C++` devido a nossa familiaridade com a linguagem e também devido a ela ser uma das linguagens com tempo de execução mais rápido.

Escolhemos trabalhar com imagens no formato PNG devido a esse formato não ser comprimido, como o JPEG por exemplo, e isso facilita a manipulação da imagem pixel a pixel. Utilizamos a biblioteca de software livre png++ para realizar tal manipulação.

Optamos por criar uma classe em `C++`, a `ImageTexture`, sem interface gráfica, porque nossos usuários finais serão desenvolvedores que podem utilizar nossa classe para gerar texturas de forma dinâmica para o uso em outras aplicações.

Realizamos duas implementações da classe, a primeira `imagetexture.cpp` possui a complexidade de tempo esperada e é a que deve ser usada em aplicações que precisam de alta performance. A segunda, `visualimagetexture.cpp` é uma implementação visual, em que após cada passo do algoritmo, renderiza o formato do novo *patch* encontrado na cor vermelha, se o algoritmo detectou o caso 1 ou em verde se o algoritmo detectou o caso 2, há uma pausa de 8 segundos e, após, os pixels do patch são renderizados corretamente. A implementação visual exerce o papel de facilitar testes da implementação da classe e facilitar o entendimento do algoritmo por pessoas que queiram estudá-lo. Para facilitar a mudança entre as implementações, disponibilizamos um arquivo de `makefile` no repositório do github, além de instruções de como usá-lo.

As implementações foram realizadas no formato de código aberto, sua documentação detalhada pode ser acessada em: `https://leticiafcs.github.io/GraphCutTexture Synthesis/html/index.html`.

Dessa forma, qualquer pessoa pode criar sua própria implementação da classe, em particular, como trabalho futuro, outras implementações da função de *matching* podem ser feitas para melhorar os resultados obtidos em aplicações cuja performance desta etapa não seja crucial.

Utilizamos a sobrecarga de métodos na nossa implementação. Em particular, nos métodos que tem como um dos atributos uma imagem de entrada optamos por declará-los tanto recebendo tal imagem como um objeto do tipo `png::image<png::rgb_pixel>` quanto como um objeto do tipo `std::string` com o nome do arquivo que será usado como a imagem de entrada. Fizemos isso para aumentar a flexibilidade do uso da classe. Em todos os métodos cuja imagem de entrada é dada como um objeto do tipo `std::string` é realizada uma verificação: caso o arquivo não exista ou não seja uma imagem PNG a função retorna imediatamente e uma mensagem de erro é escrita na `stream std::cerr`.

Abaixo apresentamos os métodos públicos da declaração da classe `ImageTexture`:

Apresentamos também os principais métodos privados da classe `ImageTexture`:

```
class ImageTexture{
public:
    /**...
    ImageTexture(const png::image<png::rgb_pixel> & _img);

    /**...
    ImageTexture(int width, int height);

    /**...
    void patchFittingIteration(const png::image<png::rgb_pixel> &inputImg);

    /**...
    void patchFittingIteration(const std::string &file_name);

    /**...
    void patchFitting(const png::image<png::rgb_pixel> &inputImg, int CntIterations = 100000);

    /**...
    void patchFitting(const std::string &file_name, int CntIterations = 100000);

    /**...
    void blending(int heightOffset, int widthOffset, const png::image<png::rgb_pixel> &inputImg);

    /**...
    void blending(int heightOffset, int widthOffset, const std::string &file_name);

    /**...
    void render(const std::string &file_name);
```

Figura 19 – Métodos Públicos da classe `ImageTexture`

```
template<typename Pixel>
static long double calcCost(const Pixel &as, const Pixel &bs, const Pixel &at, const Pixel &bt);

std::pair<int, int> matching(const png::image<png::rgb_pixel> &inputImg);
bool isFirstPatch(int heightOffset, int widthOffset, const png::image<png::rgb_pixel> &inputImg);
bool stPlanarGraph(int heightOffset, int widthOffset, const png::image<png::rgb_pixel> &inputImg);
void blendingCase1(int heightOffset, int widthOffset, const png::image<png::rgb_pixel> &inputImg);
void blendingCase2(int heightOffset, int widthOffset, const png::image<png::rgb_pixel> &inputImg);
```

Figura 20 – Principais Métodos Privados da classe `ImageTexture`

Além desses, a classe possui outros membros privados auxiliares para facilitar sua implementação.

O construtor da classe possui complexidade de tempo linear no número de pixels da imagem de saída. Foram realizadas duas declarações do construtor, que tem como parâmetros, respectivamente:

(a) Um objeto do tipo `png::image<png::rgb_pixel>`, uma cópia desse objeto será usada como a imagem de saída na classe.

(b) As dimensões da imagem de saída da classe.

Nos métodos descritos a seguir, denotaremos por p o número de pixels na imagem de entrada.

O método `patchFittingIteration` realiza uma iteração de `patch fitting`. Primeiramente ele realiza uma chamada do método privado `matching` que retorna um posicio-

namento da imagem de entrada na imagem de saída. Após, ele chama o método privado `isFirstPatch` que verifica se a imagem de entrada nessa posição possui interseção com pixels já copiados anteriormente. Caso não possua, a imagem de entrada inteira é copiada nesta posição. Caso contrário, é chamado o método `blending`. Na implementação `imagetexture.cpp`, a complexidade de tempo do `patchFittingIteration` é $O(p \log^2 p)$ pois é limitada pela complexidade de tempo do método `blending`.

O método `patchFitting` realiza *cntIterations* chamadas de `patchFittingIteration`, todas com a mesma imagem de entrada. Por padrão, *cntIterations* = *1000* porque este valor para o parâmetro foi o suficiente para gerar imagens visualmente bonitas nos nossos testes.

O método `blending`, primeiramente realiza uma chamada do método `stPlanarGraph` para verificar se s e t estão em uma mesma face do grafo criado. Caso estejam, o método `blendingCase1` é chamado pois ele é uma implementação de 2.7.1, caso contrário o método `blendingCase2` que é uma implementação de 2.7.2 é chamado. Escolhemos declarar o método `blending` como público e não privado para possibilitar ao usuário posicionar uma imagem em uma posição específica, isto é particularmente interessante para gerarmos texturas com padrões regulares como mostraremos em 4.1.2. O método `blending` tem complexidade de tempo $O(p \log p)$ se s e t estão em uma mesma face ou $O(p \log^2 p)$ caso contrário.

O método `render` cria um arquivo PNG com o nome recebido com a imagem de saída gerada. Os pixels da imagem de saída em que não foram copiados novos pixels em nenhuma iteração do patch fitting realizada anteriormente são pretos (possuem código *rgb* $(255, 255, 255)$). Ter essa função separada foi muito útil para a implementação `visualimagetexture.cpp`.

No nosso repositório há um exemplo simples de uso da classe `ImageTexture`:

```cpp
#include "imagetexture.hpp"

int main(int argc, char *argv[]){
    // creates the image texture with dimensions 500x500
    ImageTexture texture(500, 500);

    // runs 500 iterations of the patch fitting using
    // "../input/areia_da_praia.png" as the input image
    texture.patchFitting("../input/areia_da_praia.png", 500);

    // renders the texture to the file "../output/output.png"
    texture.render("../output/output.png");
}
```

Figura 21 – Exemplo de uso da classe `ImageTexture`

4.1 RESULTADOS

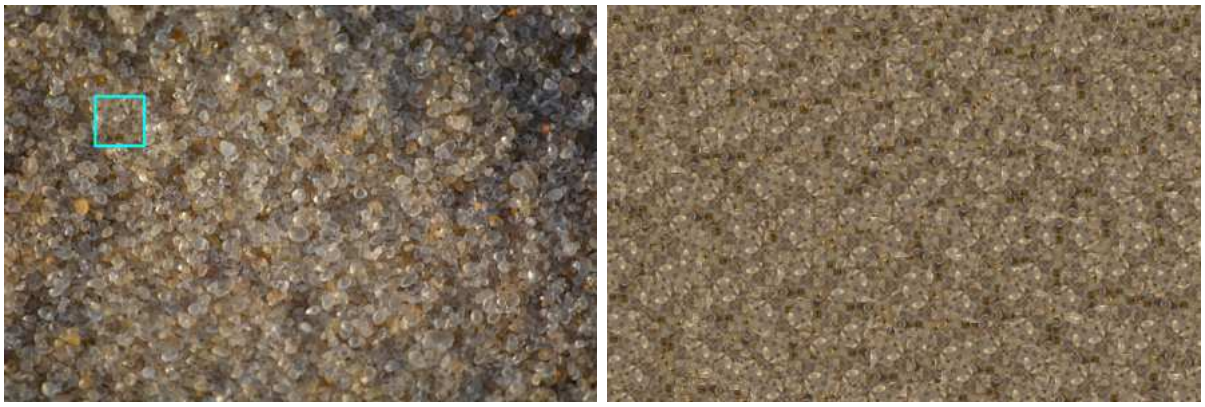
Separamos os resultados em duas categorias: a de texturas que possuem padrões regulares e as texturas que não possuem padrões regulares.

4.1.1 Texturas Sem Padrões Regulares

Para texturas sem padrões regulares, nosso algoritmo gerou texturas visualmente boas. Em alguns casos, optamos por usar mais de uma rotação ou reflexão da imagem de entrada para que não seja tão perceptível que a imagem de saída é formada por repetições da pedaços de uma mesma imagem.

Apresentamos a seguir as imagens de entrada e de saída do nosso algoritmo para esse caso.

4.1.1.1 Grãos de Areia



(a) Imagem de entrada 50×50 recortada da foto 640 × 427 da areia da praia de Itaipuaçu. Adaptada de (MIPORTO, 2014)

(b) Imagem 640×427 gerada por 2000 iterações do nosso programa a partir de 4 rotações da imagem à esquerda.

Figura 22

4.1.1.2 Grãos de Café

(a) Imagem de entrada 50×50 recortada de foto 240 × 240 de pote de café. Adaptada de (BELIAL, 2018)

(b) Imagem 600 × 600 gerada por 500 iterações do nosso programa a partir de 4 rotações da imagem à esquerda

Figura 23

4.1.1.3 Tecido Jeans

(a) Imagem de entrada 50 × 50 recortada de foto 640 × 427 de tecido jeans. Adaptada de (HERNANDEZ, 2011)

(b) Imagem 640 × 600 gerada por 427 iterações do nosso programa a partir de 2 reflexões da imagem à esquerda

Figura 24

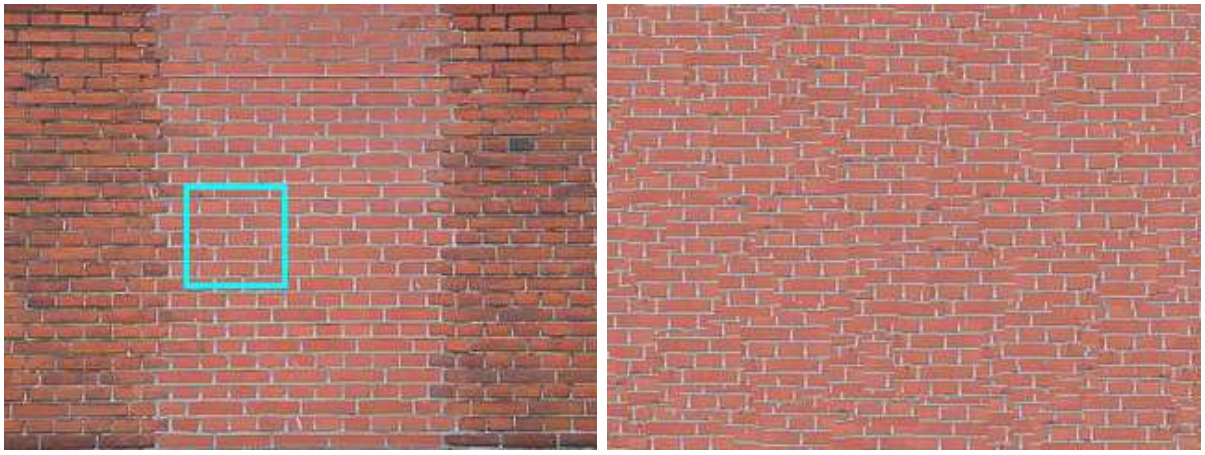
4.1.2 Texturas Com Padrões Regulares

Nas texturas com padrões regulares percebemos que escolher posições aleatórias de forma uniforme na etapa de *matching* não gera resultados visualmente bons. Nesses casos, optamos por escolher manualmente as posições dos pedaços retangulares e executamos a etapa de *blending* da mesma forma que para texturas sem padrões regulares. Por esse

motivo, na nossa implementação optamos por deixar os métodos de *blending* públicos, assim, o usuário pode escolher entre usar o método *PatchFittingIteration* para executar uma iteração do algoritmo ou pode escolher manualmente a posição da imagem de entrada na imagem de saída e usar o método *blending* passando tal posição como parâmetro do método.

Apresentamos a seguir as imagens de entrada e de saída do nosso algoritmo para esse caso, mostrando um comparativo entre a imagem gerada pelo *matching* aleatório, pelo *matching* manual, e pela cópia das imagens de entrada completas nas mesmas posições que no *matching* manual.

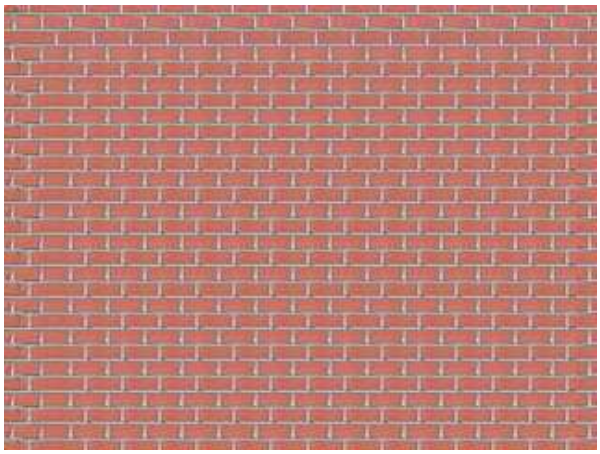
4.1.2.1 Muro de Tijolos



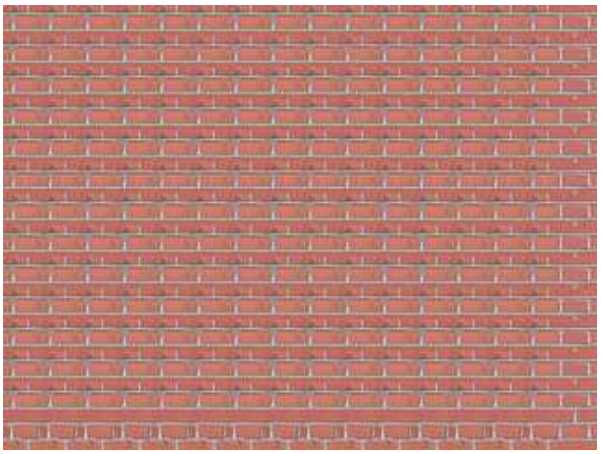
(a) Imagem de entrada 50 × 50 recortada da foto de muro de tijolos. Adaptada de (ATOBAR, 2009)

(b) Imagem 320 × 240 gerada por 2000 iterações do nosso programa, com *blending* aleatório, a partir de 2 reflexões da imagem à esquerda.

Figura 25



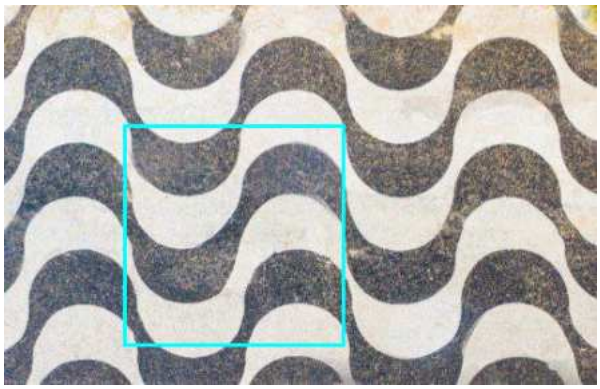
(a) Imagem 320×240 gerada pelo nosso algoritmo usando o *matching* manual.



(b) Imagem 320×240 gerada copiando a imagem de entrada nas mesmas posições escolhidos pelo *matching* manual.

Figura 26

4.1.2.2 Mosaico de Burle Marx



(a) Imagem de entrada 175×175 recortada de foto do mosaico de Burle Marx. Adaptada de (DABRAVOLSKAS, 2021).

(b) Imagem 483×309 gerada por 100 iterações do nosso programa, com *blending* aleatório, a partir da imagem à esquerda.

Figura 27

(a) Imagem 483×309 gerada pelo nosso algoritmo usando o *matching* manual.

(b) Imagem 483×309 gerada copiando a imagem de entrada nas mesmas posições escolhidos pelo *matching* manual.

Figura 28

5 CONCLUSÃO E TRABALHOS FUTUROS

Exploramos rotações e reflexões das imagens originais para melhorar as texturas geradas e nossa implementação apresentou resultados visualmente melhores para texturas que não apresentam padrões regulares.

Para texturas que possuem padrões regulares, em especial no Mosaico de Burle Marx (4.1.2.2), nossa implementação com etapa de *matching* aleatória não gerou resultados visualmente bons, e por isso foi necessário executar tal etapa manualmente.

Como trabalhos futuros podem ser explorados e implementados outros algoritmos para a etapa de *matching*, em particular escolhendo as posições aleatoriamente com probabilidade maior para posições em que a imagem de entrada se mescla melhor com os pedaços copiados anteriormente (KWATRA et al., 2003).

REFERÊNCIAS

ATOBAR. **File:Muro de ladrillos.JPG — Wikimedia Commons, the free media repository**. 2009. Acesso em 15 maio 2024. Disponível em: https://commons.wikimedia.org/wiki/File:Muro_de_ladrillos.JPG.

BARNES, C.; ZHANG, F.-L. A survey of the state-of-the-art in patch-based synthesis. **Computational Visual Media**, v. 3, p. 3–20, 2017.

BELIAL, A. **File:Roasted coffee beans in white bowl.png — Wikimedia Commons, the free media repository**. 2018. Acesso em 15 maio 2024. Disponível em: https://commons.wikimedia.org/wiki/File:Roasted_coffee_beans_in_white_bowl.png.

CORMEN, T. H. et al. **Introduction to Algorithms**. 3. ed. Massachusetts: MIT, 2009.

DABRAVOLSKAS, D. **File:Top Down View of Copacabana Mosaic and Palm Trees 3.jpg — Wikimedia Commons, the free media repository**. 2021. Acesso em 15 maio 2024. Disponível em: https://commons.wikimedia.org/wiki/File:Top_Down_View_of_Copacabana_Mosaic_and_Palm_Trees_3.jpg.

DIJKSTRA, E. W. A note on two problems in connexion with graphs. **Numerische Mathematik**, v. 1, p. 269–271, 1959.

DINIC, E. A. Algorithm for solution of a problem of maximum flow in networks with power estimation. **Soviet Math. Doklady**, v. 11, p. 1277–1280, 1970.

EDMONDS, J.; KARP, R. M. Theoretical improvements in algorithmic efficiency for network flow problems. **Journal of the ACM**, v. 19, n. 2, p. 248–264, 1972.

FORD, L. R.; FULKERSON, D. R. Maximal flow through a network. **Canadian Journal of Mathematics**, v. 8, p. 399–404, 1956.

HERNANDEZ, W. R. R. **File:Blue Denim Fabric Texture Free Creative Commons (6816223272).jpg — Wikimedia Commons, the free media repository**. 2011. Acesso em 15 maio 2024. Disponível em: https://commons.wikimedia.org/wiki/File:Blue_Denim_Fabric_Texture_Free_Creative_Commons_(6816223272).jpg.

KWATRA, V. et al. Graphcut textures: Image and video synthesis using graph cuts. **ACM Transactions on Graphics (tog)**, v. 22, n. 3, p. 277–286, 2003.

MIPORTO. **File:Areia da Praia.JPG — Wikimedia Commons, the free media repository**. 2014. Acesso em 15 maio 2024. Disponível em: https://commons.wikimedia.org/wiki/File:Areia_da_Praia.JPG.

REIF, J. H. Minimum s-t cut of a planar undirected network in $O(n \log^2(n))$ time. **SIAM Journal on Computing**, v. 12, n. 1, p. 71–81, 1983.