

Universidade Federal de São João del-Rei

Implementação do *Seam Carving* para o redimensionamento de imagens

Algoritmos e Estrutura de Dados III

Isabella Vieira Ferreira
Mônica Neli de Resende

1 Introdução

Neste trabalho trataremos sobre o redimensionamento de imagens, uma ferramenta fundamental em diversos aspectos de processamento de imagens digitais. Com a diversidade e a quantidade de dispositivos de exibição digital, o redimensionamento de imagens é bastante utilizado, tendo em vista que uma imagem pode ser ampliada ou reduzida a diferentes tamanhos adequando-se às capacidades de cada dispositivo. Porém abordaremos somente a redução de imagens e para tal propomos como forma de redimensionamento o *Seam Carving*, um algoritmo desenvolvido por Shai Avidan, da Mitsubishi Electric Research Labs, e Ariel Shamir, do The Interdisciplinary Center & MERL.

O objetivo do algoritmo é possibilitar a exibição de imagens sem distorção em vários meios de comunicação (telefones celulares, PDAs), utilizando padrões de documentos, como HTML, que já suportam mudanças dinâmicas no layout de página e texto, mas não nas imagens. (*Shai Avidan e Ariel Shamir*)

O *Seam carving* utiliza uma função de energia definindo a importância dos pixels para que sejam feitos “cortes” nos caminhos menos importantes, ou seja, os pixels menos significativos são eliminados a fim de preservar a estrutura e a característica principal da imagem. Utilizaremos o Operador de Sobel cuja finalidade é realçar linhas verticais e horizontais mais escuras que o fundo, detectando as bordas, ou seja, sem realçar pontos isolados.

Propomos duas soluções para o problema: a primeira é desenvolvida utilizando programação dinâmica, que é um paradigma de programação que armazena os resultados de sub-cálculos, possibilitando a utilização dos resultados em cálculos posteriores. A segunda solução é modelada em grafos, que são estruturas comumente utilizadas em modelagem de problemas computacionais. A partir dessas estratégias encontraremos o caminho de menor energia.

A motivação do nosso trabalho deve a importância do tema, tendo em vista que é uma técnica muito utilizada. Devido a isso a procura por formas de redimensionamento da imagens eficientes torna-se cada vez maiores, levando em consideração a diversidade de dispositivos de visualização. Esperamos aplicar a teoria apresentada em sala de aula reforçando e acrescendo nosso conhecimento a respeito. O desafio é desenvolver um algoritmo eficiente em termos de tempo e espaço, de modo a obter resultados em um tempo considerável, além de analisar os resultados obtidos apontando as características presentes nas imagens que resultam em bons ou maus resultados.

2 Formato de entrada e saída

A entrada de dados é uma imagem no formato PPM (“Portable Pixmap Map”), um formato simples para imagens que possibilita sua leitura como arquivo de texto. Uma imagem em formato PPM possui: uma string mágica denominada “P3”; a largura (número de colunas) e a altura (número de linhas) da imagem (necessariamente nessa ordem); o máximo valor de cor, definido como 255 e para cada pixel da imagem deverá ser especificado 3 inteiros contendo as intensidades RGB do pixel. O modelo RGB representa o sistema de cores formado por Vermelho (Red), Verde (Green) e Azul (Blue). Os pixels são inseridos por linha no arquivo que contém no máximo setenta caracteres em cada linha.

Vale ressaltar que a imagem poderá conter comentários, onde os mesmos começarão com o símbolo #, e poderá conter também um número arbitrário de *whitespaces*, sendo estes espaços, tabs, linhas em branco etc.

O arquivo de saída será a imagem reduzida, segundo as especificações informadas pelo usuário, também no formato PPM e conterá os mesmos campos do arquivo de entrada. A imagem final terá o nome de “*saida.ppm*”.

3 Programação Dinâmica

3.1 Solução apresentada

A programação dinâmica é um paradigma de programação aplicável a problemas que possuem subestrutura ótima e superposição de subproblemas. O que significa dizer que a solução ótima para o problema pode ser computada a partir da combinação de soluções

ótimas de subproblemas. Tais soluções são previamente calculadas e memorizadas, para que estas possam ser utilizadas no cálculo de soluções posteriores, de modo que não necessitam ser recalculadas.

3.1.1 Estrutura de Dados

Para a representação da imagem no algoritmo, utilizamos uma matriz alocada dinamicamente com dimensões $h \times w$, onde h é o número de linhas e w o número de colunas. Em cada célula da matriz armazenamos as informações conforme a Figura 1.

	->	R	G	B	IL	Grad	coluna	distancia
	->	R	G	B	IL	Grad	coluna	distancia
	->	R	G	B	IL	Grad	coluna	distancia

Figura 1: Ilustração da estrutura de dados

3.1.2 Remoção de colunas

De acordo com o algoritmo, a remoção de colunas será executada na seguinte ordem:

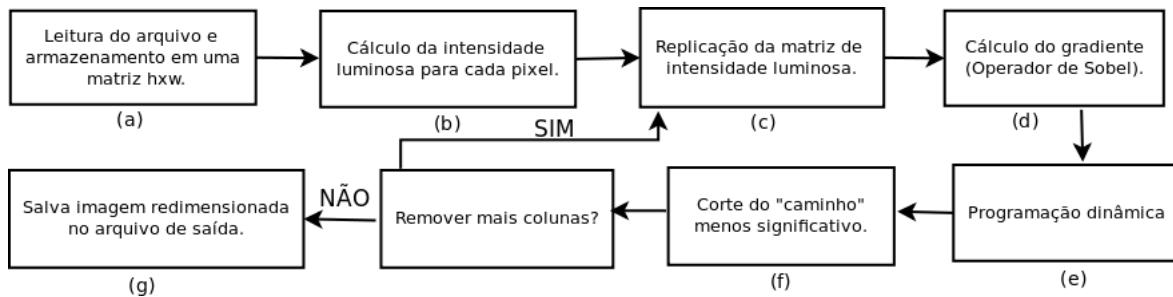


Figura 2: Remoção de colunas na programação dinâmica

(a): é feita a leitura da imagem e seus pixels como arquivo de texto , ou seja, suas cores em RGB são armazenadas na matriz cuja forma está representada na Figura 1. As dimensões da matriz são dois números a mais que as especificadas no arquivo ($h+2 \times w+2$) e o seu preenchimento é feito a partir da posição (1, 1) até a posição (h, w), de modo a simplificar o cálculo do vetor gradiente. Durante a leitura do arquivo são desprezados comentários, espaços e linhas em branco.

(b): a intensidade luminosa é uma medida de quão claro ou escuro o pixel é. Para cada célula (pixel) da matriz calculamos a sua intensidade luminosa e armazenamos na mesma célula, no campo “IL”. O cálculo é definido como uma média ponderada das intensidades de cada cor, onde os pesos levam em consideração a sensibilidade do olho humano:

$$IL(R, G, B) = 0,30 * R + 0,59 * G + 0,11 * B$$

(c): A “replicação”da matriz foi um artifício utilizado para que cada pixel tenha sempre oito pixels em sua volta (“vizinhos”), sendo necessário para realizar os cálculos do Operador de Sobel que será descrito mais adiante. Conforme foi mencionado em (a), a matriz “replicada”é de ordem ($h+2, w+2$), como mostra a Figura 3.

1	2	3	4	5
1				
2				
3				
4				
5				

Figura 3: Matriz de ordem 3x3 replicada

(d): o Operador de Sobel estima a energia de um pixel calculando o vetor gradiente da intensidade da imagem em cada ponto, dando a direção da maior variação de claro para escuro levando em consideração os oito pixels à sua volta. Como as variações claro-escuro intensas correspondem a fronteiras entre objetos, consegue-se fazer a detecção de contornos, ou seja, pixels de maior energia. Assim, podemos obter os caminhos de menor energia (lugares na imagem que não possuem contornos), preservando a estrutura principal da imagem durante o redimensionamento (corte dos caminhos de menor energia). A Figura 4 apresenta um exemplo da aplicação do Operador de Sobel em uma imagem arbitrária.

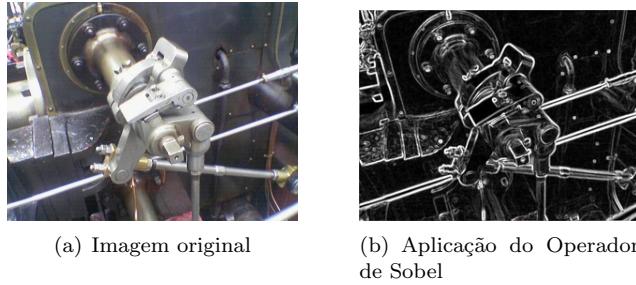


Figura 4: Exemplo do Operador de Sobel

Para encontrar o vetor gradiente, o Operador de Sobel calcula as derivadas parciais em relação às duas direções horizontal e vertical. Na Figura 5, temos que \mathbf{A} é uma matriz 3×3 que contém os “vizinhos” de um determinado pixel, \mathbf{Gx} e \mathbf{Gy} é a matriz de pesos do Operador de Sobel.

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{e} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

Figura 5: Cálculo do Operador de Sobel

Os respectivos elementos da matriz \mathbf{A} são multiplicados pelos respectivos elementos da matriz \mathbf{Gx} e somados, em seguida, fazemos o mesmo para \mathbf{Gy} . Depois dos cálculos feitos, obtemos a norma do vetor gradiente pela seguinte fórmula: $e(x,y) = \sqrt{Gx^2 + Gy^2}$.

Para melhor entendimento do cálculo das energias dos pixels, a Figura 6 mostra como a energia é calculada para o pixel de intensidade luminosa igual a 10.

	1	2	3	4	5
1	10	10	11	12	12
2	10	10	11	12	12
3	13	13	14	15	15
4	16	16	17	18	18
5	16	16	17	18	18

(a) Matriz replicada

(b) Cálculo de energia

█ Pixels sendo avaliados
█ Comum ao 10 e ao 16
█ “Replica” do número 10

$\mathbf{G}_x: 10(-1) + 10(0) + 11(1) + 10(-2) + 10(0) + 11(2) + 13(-1) + 13(0) + 14(1)$
 $\mathbf{G}_y: 10(1) + 10(2) + 11(1) + 10(0) + 10(0) + 11(0) + 13(-1) + 13(-2) + 14(-1)$
 $e(x,y) = (\sqrt{4^2 + (-12)^2}) = 12,64$

Figura 6: Cálculo da energia de um pixel

Esse processo é feito para todos os pixels da matriz, sendo que a cada cálculo os valores são armazenados no campo “Grad” da célula em questão.

(e): para a remoção de colunas devemos encontrar o caminho de menor energia. Portanto para cada pixel da última linha da imagem, calculamos o caminho de menor energia que termina nele. Para isso, utilizando a programação dinâmica encontramos o menor caminho até cada uma das células da matriz e pegamos na última linha o menor valor obtido. O procedimento inicia-se a partir da segunda linha da imagem, pois a primeira não possuiá caminhos até elas, dessa forma guardam os próprios valores de energia. O modo como são feitas as somas das energias para determinar o menor caminho é apresentado na Figura 7.



Figura 7: Estratégia para programação dinâmica

Conforme apresentado na Figura 7, o valor no canto à esquerda (em vermelho) de cada célula é o resultado da norma do gradiente (energia) e o valor em preto é a soma resultante das energias (menor caminho) até a referida célula.

Exemplo: para a primeira célula da segunda linha, cuja energia é três, há dois caminhos possíveis (marcados por setas na cor preta e verde). Compara-se as somas $3+1$ e $3+4$ e substitui a energia da célula analisada com a menor soma ($3 + 1 = 4$) e por fim armazenamos o número da coluna proveniente (coluna 1). Em seguida o procedimento é repetido para cada célula da matriz. Ao final, teremos os valores das somas de energias (menores caminhos) até cada uma das células da última linha. Os caminhos de corte estão destacados com o fundo branco na Figura 7.

É importante ressaltar que é possível ter mais de uma solução, ou seja, mais de um caminho de menor energia, neste caso definimos que o primeiro caminho encontrado será a solução.

(f): ao determinar o menor valor da última linha, passamos a cortar o caminho traçado a partir do número da coluna de onde a soma procedeu. Para cada pixel a ser cortado armazenamos a coluna de onde a soma se originou e deslocamos o restante dos itens da linha em questão para a esquerda, substituindo-o. Por fim diminui-se a ordem da matriz.

(g): caso seja necessário retirar várias colunas, o processo de (c) a (f) é executado até que seja completado o número de colunas a serem retiradas. Após isso, a matriz final é salva no arquivo de saída.

3.1.3 Remoção de linhas

Para fazer a remoção de linhas, basta fazer a transposta da matriz e executar a remoção de colunas, conforme descrito na seção 3.1.2. A ordem de execução é basicamente a mesma da Figura 2, porém após a leitura do arquivo fazemos a transposta da matriz e executamos os passos de (b) a (g) na mesma sequência. Caso seja necessário retirar várias linhas, o processo de (c) a (f) é executado até que seja completado o número de linhas que o usuário deseja remover. Após isso, fazemos a transposta novamente e imprimimos o resultado no arquivo de saída.

3.1.4 Porque utilizamos programação dinâmica?

A grande vantagem oferecida por esse paradigma está no reaproveitamento de cálculos já realizados, o que implica em um ganho em tempo e processamento. Como o objetivo é extamente desenvolver um algoritmo eficiente em tempo essa é uma estratégia ideal.

Conforme apresentamos, o paradigma da programação dinâmica é aplicável a problemas que possuem subestrutura ótima e superposição de subproblemas. Através da técnica de projeto de algoritmos, indução matemática, mostraremos a seguir que o paradigma é aplicável ao nosso problema.

Problema original: para cada pixel da última linha da imagem, calcule o caminho de menor energia que termina nele.

Sejam os subproblemas o menor caminho de energia até cada célula de uma linha.

Passo base: como a primeira linha não há caminho que cheguem à ela, começamos da segunda linha. Para cada célula dessa linha substitui-se a energia pelo resultado da menor soma, ou seja, o caminho de menor energia, que termina nela.

Passo indutivo: utilizando os resultados anteriores, encontramos o caminho de menor energia até cada célula da próxima linha. Repetindo o procedimento para cada célula até a última linha dividimos o problema em subproblemas e combinando soluções ótimas obtemos uma solução ótima para o problema original.

Através dessa análise temos que a função de decomposição que representa o *Seam Carving* na implementação utilizando programação dinâmica será: $M(i,j) = e(i,j) + \min(M(i-1, j-1), M(i-1, j), M(i-1, j+1))$.

Dessa forma, concluímos que a natureza do problema possui superposição de problemas (divisão em subproblemas) e substrutura ótima (combinação de soluções ótimas) características para a aplicabilidade do paradigma da programação dinâmica, como deveria ser provado. Contudo a ordem na qual as soluções devem ser calculadas é a partir da segunda linha até a última para que assim em cada linha possamos utilizar os resultados anteriores, como foi mostrado.

4 Grafos

4.1 Solução apresentada

Grafos é um conjunto não vazio de pontos (vértices ou nós) e um conjunto de arestas (ligação) unindo todos ou alguns desses pontos. As arestas podem ser direcionadas ou não, por exemplo, uma aresta (u, v) significa que u alcança v , porém o contrário não acontece por essa aresta. As arestas também podem ser ponderadas ou não, podemos ponderá-la com a distância entre os vértices, por exemplo. Quando um nó possui aresta a outros, dizemos que os nós os quais ele alcança são seus adjacentes.

Para este problema utilizaremos um grafo direcionado e ponderado com a energia dos pixels. No exemplo abaixo, temos a representação do grafo do nosso problema, onde os vértices representam os pixels e cada aresta possui o peso referente a energia do pixel adjacente, representados por (x) (y) e (z) .

A solução do problema em grafos pode ser transformada em uma solução para o problema original através do cálculo da menor distância para cada vértice.

4.1.1 Estrutura de Dados

Para representar o grafo computacionalmente utilizamos a mesma matriz de energia utilizada para a programação dinâmica. Poderíamos representá-lo de outras formas como matriz de adjacência ou lista de adjacência, mas percebemos que utilizando somente uma matriz seria mais conveniente devido a economia de memória. Caso não representássemos dessa maneira, a melhor forma que poderíamos ter utilizado seria: no momento de obtermos o caminho mais curto, colocarmos os valores de energia em uma lista de adjacência que guardaria os possíveis caminhos e a distância daquele pixel a outros adjacentes. Vale ressaltar que a lista de adjacência seria mais adequada para este caso, pois ela conteria no máximo três células em cada lista, conforme apresentamos

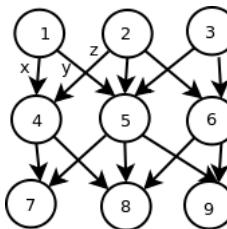


Figura 8: Grafo direcionado e ponderado

na Figura 7. A matriz de adjacência não seria uma boa opção, pois como o grafo é esparço (sem muitas conexões), esta alocaria uma memória que não seria totalmente utilizada, tendo assim um custo maior de memória.

Através desta análise percebemos que se modelássemos nosso algoritmo utilizando alguma das duas representações de grafos apresentadas, alocaríamos na memória a matriz contendo os pixels da imagem e a lista de adjacência contendo os possíveis caminhos de corte, o que seria contraditório de acordo com objetivo do trabalho que diz que o algoritmo deverá ser eficiente em termos de espaço e tempo.

4.1.2 Remoção de colunas

De acordo com o algoritmo, a remoção de colunas será executada na seguinte ordem:

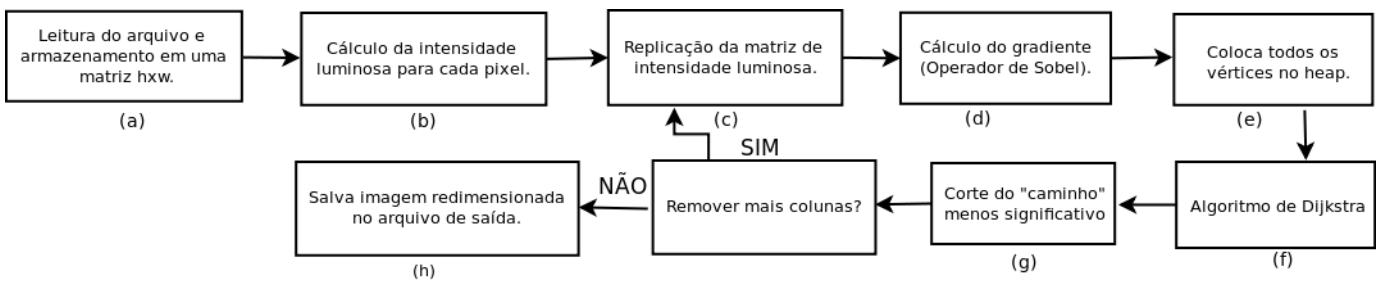


Figura 9: Remoção de colunas no grafo

A descrição da solução da sequência de (a) a (d) é a mesma descrita na subseção 3.1.2. Portanto, descreveremos a seguir a partir do passo (e).

(e): a função `heap_adjacentes` é responsável por guardar no heap todos os vértices do grafo, ou seja, o valor de energia de todos os pixels na matriz.

(f): utilizamos o algoritmo de Dijkstra para solucionarmos o problema do menor caminho em um grafo dirigido com arestas de peso não negativo. Iremos mostrar a seguir o funcionamento do algoritmo:

1º passo: no pseudo-código abaixo, $V[G]$ é o conjunto de vértices(v) que formam o Grafo G e $d[v]$ é a distância de v . Primeiramente fazemos a inicialização das variáveis admitindo-se a pior estimativa possível, o caminho infinito. $[v]$ identifica o vértice de onde se origina uma conexão até v (inicializado com nulo), ou seja, o nó “pai” com menor distância, de maneira a formar um caminho mínimo.

```

para todo  $v \in V[G]$ 
     $d[v] \leftarrow \infty$ 
     $n[v] \leftarrow \text{nulo}$ 

```

2º passo: no código abaixo extraímos o primeiro valor do heap. Para cada adjacente verificamos: se $d[v] > d[u] + w(u,v)$, então $d[v] = d[u] + w(u,v)$ (Relaxamento de aresta), onde $w(u, v)$ é o peso da aresta que vai de u a v , e por fim atualizamos o antecessor (nó “pai”). Logo realizamos uma série de relaxamentos até que não tenha mais elementos no heap.

```

enquanto  $Q \neq \emptyset$ 
     $u \leftarrow \text{extrair}(Q)$ 

    para cada  $v$  adjacente a  $u$ 
        se  $d[v] > d[u] + w(u, v)$  //relaxe ( $u, v$ )
            então  $d[v] \leftarrow d[u] + w(u, v)$ 
             $n[v] \leftarrow u$ 

```

No final do algoritmo teremos o grafo totalmente preenchido com o menor caminho entre os vértice de G.

(g): primeiramente, vamos na última linha do grafo e verificamos qual é a menor distância armazenada. A partir disso, vamos “subindo” no grafo a partir da coluna do antecessor (ou nó “pai”), de modo que o menor caminho é detectado. Então, deslocamos o restante dos itens da linha, a partir da posição em questão, para a esquerda substituindo o pixel que deve ser eliminado. Por fim, diminui-se a ordem da matriz.

(h): caso seja necessário retirar várias colunas, o processo de (c) a (g) é executado até que seja completado o número de colunas a ser retirado. Após isso, a matriz final é salva no arquivo de saída.

4.1.3 Remoção de linhas

Para fazer a remoção de linhas, basta fazer a transposta da matriz e executar a remoção de colunas, conforme descrito na subseção 4.1.2. A ordem da execução é basicamente a mesma da Figura 9, porém após a leitura do arquivo fazemos a transposta da matriz e executamos os passos de (b) a (h) na mesma sequência. Caso seja necessário retirar várias linhas, o processo de (c) a (g) é executado até que seja completado o número de linhas que o usuário deseja remover. Após isso, fazemos a transposta novamente e imprimimos o resultado no arquivo de saída.

4.1.4 Por que utilizamos grafos?

A vantagem de modelar um problema utilizando grafos está na facilidade da visualização dos vértices adjacentes, tal como adicionar ou remover arestas, tendo em vista que é muito útil para algoritmos em que necessitamos saber com rapidez se existe uma aresta ligando dois vértices. Sua desvantagem está em requerer muito espaço de armazenamento, por isso optamos pela representação da própria matriz como o grafo.

Grafos é uma ótima modelagem para nosso problema uma vez que queremos saber o caminho de menor energia em uma imagem. Utilizando os algoritmos e técnicas apropriadas, temos com facilidade o menor caminho de energia.

5 Análise de Complexidade de Tempo e Espaço das Rotinas

Nesta seção faremos uma análise da complexidade de tempo e espaço das funções de redimensionamento de imagem. Cada função será analisada individualmente e em seguida calcularemos a complexidade total a partir da análise da função principal. Ressaltamos que “ h ” é o número de linhas e “ w ” o número de colunas e as complexidades serão calculadas para o pior caso. Cálculos e atribuições serão consideradas constantes, dessa forma, complexidade ($O(1)$).

1. void leitura_arquivo (FILE *entrada, matriz *imagem)

A função leitura do arquivo é responsável por preencher a matriz. E para isso é necessário um *loop* aninhado, $O(h.w)$, e como procedimento interno do *loop* temos chamadas a três funções:

- a função *cria_matriz*, que é responsável por criar a matriz e possui complexidade de tempo $O(h)$, pois para cada linha alocamos um vetor de colunas. Como a matriz tem dimensões $(h+2,w+2)$ sua complexidade de espaço será $O(h+2.w+2)$.

- a função *ignora_comentario*, que é responsável por desprezar os comentários lidos no arquivo. Esta função tem complexidade de tempo $O(m)$, onde “ m ” é o número de caracteres lidos, uma vez que a leitura é feita caracter por caracter. Como não há alocação dinâmica de memória, não há complexidade de espaço.

- a função *le_palavra* é responsável por ler os números do arquivo caracter por caracter e convertê-los para inteiro. Sua complexidade de tempo será $O(m')$, onde “ m' ” é o número de caracteres lidos.

A complexidade total dos procedimentos internos do *loop* aninhado será portanto $O(\max(h, m, m'))$. De posse dessas informações, temos que a leitura do arquivo terá complexidade: $O(\max(h, m, m')).(h.w) = O(h.w)$.

Complexidade de tempo: $O(h.w)$ e **Complexidade de espaço:** $O(h+2.w+2)$

2. matriz transposta (matriz *original)

Esta função é responsável por criar a transposta da matriz original, resultando em $O(h.W)$. Nela temos a função *cria_matriz* que tem complexidade $O(h)$, como vimos anteriormente. Temos uma chamada à função *destruir_matriz*, onde a matriz “original” é desalocada - complexidade $O(h)$ - pois para cada posição do vetor de linhas libera o vetor de colunas. Como o restante são apenas operações básicas a complexidade total da função será: $O(h) + O(h.w) + O(h) = O(\max((h.w),h)) = O(h.w)$.

Complexidade de tempo: $O(h.w)$ e **Complexidade de espaço:** $O(w+2.h+2)$

3. void intensidade_luminosa (matriz *imagem)

Esta função é responsável por calcular a intensidade luminosa de cada pixel da matriz. Como possui apenas operações básicas sua complexidade é $O(h.w)$.

Complexidade de tempo: $O(h.w)$ e **Complexidade de espaço:** não há alocação de memória

4. void replica (matriz *mat_transposta)

Esta função é responsável por replicar a matriz. Portanto temos dois *loops*: o primeiro percorrendo uma linha inteira e o segundo percorrendo uma coluna inteira. Logo sua complexidade será $O(h) + O(w) = O(\max(h, w))$.

Complexidade de tempo: $O(\max(h, w))$ e **Complexidade de espaço:** não possui alocação de memória

5. void auxiliar_gradiente (matriz *mat_transposta)

Essa função percorre toda a matriz, portanto $O(h.w)$. Em seu interior faz chamadas a função vizinhos responsável por criar a matriz $3x3$ com os “vizinhos” do pixel - complexidade $O(1)$, e também a função operador_sobel também com complexidade $9.O(1)$. Por fim chama a função *destruir_float*, responsável por destruir a matriz de vizinhos - complexidade $O(3)$. Resumindo, temos que o *loop* interno terá complexidade $9.O(1)$ e complexidade total da função será: $9.O(1).O(h.w) + O(3) = O(h.w)$.

Complexidade de tempo: $O(h.w)$ e **Complexidade de espaço:** $O(9)$

6. void melhor_caminho (matriz *mat)

Essa função é responsável por fazer a programação dinâmica. Para isso, como foi explicado, é necessário percorrer toda a matriz com exceção da primeira linha. Logo, a complexidade será $O((h-1).w)$.

Complexidade de tempo: $O((h-1).w)$ e **Complexidade de espaço:** não há alocação de memória

7. void corta_imagem (matriz *imagem)

Essa função é responsável por determinar o caminho de menor energia. Primeiramente, é necessário buscar o menor elemento da última linha, essa pesquisa tem complexidade $O(1).(w-1) = O(w-1)$. Para a remoção da coluna é necessário um laço percorrendo o número de linhas e em cada iteração temos o deslocamento que no pior caso é $O(w)$, ou seja, quando deslocamos todas as células da linha. O loop terá então complexidade $O(h.w)$. Logo, a complexidade total da função será $O(w-1) + O(h.w) = O(\max(w-1, h.w)) = O(h.w)$.

Complexidade de tempo: $O(h.w)$ e **Complexidade de espaço:** não há alocação de memória

8. void heap_adjacentes (matriz *imagem, celula **heap)

Essa função é responsável por colocar todas os vértices no heap, percorrendo toda a matriz, complexidade $O(h.w)$. Como será somente a operação de inserção no heap, temos que seu laço interno será $O(1)$, portanto $O(h.w).O(1) = O(h.w)$. Nessa função será alocado um heap (vetor) de tamanho $((w.h)+1)$. Dessa forma:

Complexidade de tempo: $O(h.w)$ e **Complexidade de Espaço:** $O((w.h)+1)$

9. void Dijkstra(matriz *imagem, celula *heap)

Nessa função temos o algoritmo de Dijkstra, descrito em 4.1.2(f). Nele temos a chamada da função adjacentes que é responsável por verificar quais são os adjacentes de cada pixel. Nesta função teremos no máximo três atribuições (os três possíveis adjacentes). Dessa forma, temos que a complexidade será $3.O(1) = O(3)$

A função possui um loop que tem o número de iterações igual ao tamanho do heap, logo, a complexidade será $O((h.w)+1)$. Dessa forma, o Dijkstra terá como complexidade de tempo final: $O(3) . O(h.w)+1 = O((h.w)+1)$ **Complexidade de tempo:** $O((h.w)+1)$ e **Complexidade de espaço:** não há alocação de memória

10. void corta_caminho_grafos (matriz *imagem)

Essa função é semelhante a função “corta_imagem”, porém foi adequada para a estrutura de grafos. Dessa forma:

Complexidade de tempo: $O(h.w)$ e **Complexidade de espaço:** não há alocação de memória

11. void imprimir (matriz imagem)

Essa função é responsável por imprimir no arquivo de saída cada pixel da matriz após o redimensionamento. Logo, a complexidade da função será $O(h-n.w-n')$, onde n e n' é o número de remoções realizadas.

Complexidade de tempo: $O(h-n.w-n')$ e **Complexidade de espaço:** não há alocação de memória

12. int main (int argc, char *argv[])

• Programação Dinâmica

Caso o usuário escolha remoção de colunas, temos chamadas às funções leitura_arquivo $O(h.w)$ e intensidade_luminosa $O(h.w)$, resultando em $O(h.w)+(h.w) = O(h.w)$. Temos também um laço de repetição com N chamadas às funções replica $O(\max(h, w))$, auxiliar_gradiente $O(h.w)$, melhor_caminho $O((h-1).w)$ e corta_imagem $O(h.w)$, onde N é o número de remoções a serem realizadas. A complexidade do laço interno portanto é $N.O(\max(\max(h, w), (h.w), ((h-1).w, (h.w))) = O(N.(h.w))$. Somando a complexidade do laço interno e a complexidade das operações externas temos $O(h.w) + O(N.(h.w)) = O(N.(h.w))$.

Complexidade de tempo final para a remoção de colunas: $O(N.(h.w))$.

Caso o usuário queira remover linhas, temos que a complexidade será a anterior acrescida do custo de calcular a transposta $O(h.w)$. Logo, $O(h.w) + O(N.(h.w)) + O(h,w) = O(N.(h.w))$. Lembrando que N é o número de remoções a serem realizadas, pois o algoritmo faz remoções em linhas ou colunas, mas não simultaneamente.

Complexidade de tempo final para a remoção de linhas: $O(N.(h.w))$.

Em ambos os casos, a complexidade final de tempo: $O(N.(h.w))$.

Considerando as funções que possuem alocação de memória, temos a leitura de arquivo, complexidade $O(h.w)$, transposta, complexidade $O(h+2,w+2)$ e aux_gradiente, complexidade $O(9)$. Dessa forma, temos: $O(h.w) + O(h+2.w+2) + O(9) = O(\max(h.w, h+2.w+2, 9)) = O(h+2.w+2)$.

Complexidade de espaço final: $O(h+2.w+2)$.

• Grafos

Caso o usuário escolha remoção de colunas, primeiramente temos a chamada das funções: leitura_arquivo $O(h.w)$ e intensidade_luminosa $O(h.w)$, resultando em $O(h.w)+(h.w) = O(h.w)$. Temos também um laço de repetição com N iterações que contém a inicialização do grafo - complexidade $O(h.w)$, chamada às funções replica $O(\max(h, w))$, auxiliar_gradiente $O(h.w)$, heap_adjacentes $O(h.w)$, dijkstra $O((h.w)+1)$ e corta_caminho_grafos $O(h.w)$, onde N é o número de remoções a serem realizadas. O laço interno terá complexidade: $O(h.w) + O(\max(h, w)) + O(h.w) + O(h.w) + O((h.w)+1).O(h.w) = O((h.w)+1)$. Dessa forma, a complexidade total será: $O(h.w) + N.O((h.w)+1) = N.O((h.w)+1)$.

Complexidade de tempo final para a remoção de colunas: $O(N.((h.w)+1))$.

Caso o usuário queira remover linhas, temos que a complexidade será a anterior acrescida do custo de calcular a transposta $O(h.w)$. Logo, $O(h.w) + O(N.((h.w)+1)) + O(h,w) = O(N.((h.w)+1))$. Lembrando que N é o número de remoções a serem realizadas, pois o algoritmo faz remoções em linhas ou colunas, mas não simultaneamente.

Complexidade de tempo final para a remoção de linhas: $O(N.((h.w)+1))$.

Em ambos os casos, a complexidade final de tempo: $O(N.((h.w)+1))$.

Considerando as funções que possuem alocação de memória, temos a leitura de arquivo, complexidade $O(h+2.w+2)$, porém somente ao calcular a transposta temos também a matriz transposta de complexidade $O(w+2.h+2)$. Após isso a matriz original é destruída, restando somente a transposta. Dessa forma, consideramos a complexidade de espaço $O(h+2.w+2)$ para essa matriz. Temos também a função aux_gradiente, complexidade $O(9)$ e a função heap_adjacente - complexidade de espaço $O((w.h)+1)$. Logo, temos: $O(h+2.w+2) + O(9) + O((w.h)+1) = O(\max(O((w.h)+1), h+2.w+2, 9)) = O(h+2.w+2)$.

Complexidade de espaço final: $O(h+2.w+2)$.

6 Testes e resultados

6.1 Análise qualitativa

Abordaremos aqui a análise tanto para a implementação em grafos quanto em programação dinâmica, pelo fato das duas sempre possuirem a mesma solução, logo, produzirão os mesmos resultados.

6.1.1 Imagens que produziram bons resultados



(a) anders.ppm

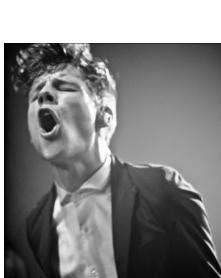


(b) circusperformer.ppm



(c) furnas.ppm

Figura 10: Imagens originais que produziram bons resultados



(a) Remoção de 400 colunas



(b) Remoção de 500 colunas



(c) Remoção de 50 linhas



(d) Remoção de 100 colunas

Figura 11: Resultado do redimensionamento

A imagem “circusperformer.ppm” possui dimensões 1024x720 e tamanho 7,8MB; a imagem “anders.ppm” possui dimensões 1024x681 e tamanho 6,4MB e a imagem “furnas.ppm” possui dimensões 1024x576 e tamanho 5,7MB.

Na Figura 10(a), pelo fato da imagem estar em escala de cinza e ter uma grande área vazia ela é ideal para a remoção de colunas sem deformações no cantor. Porém como o cantor preenche todas as linhas, em uma remoção horizontal é esperado uma deformação caso seja drástica. Ao retirar cem linhas, a remoção ocorre no terno e no pescoço e ao reduzir drásticamente na vertical a deformação ocorre nos ombros.

Na Figura 10(b), o algoritmo produz bons resultados na redução de colunas devido à grande área azul, ou seja, uma parte homogênea que não faz diferença ao ser cortada. Essa é a principal característica em uma imagem para que os resultados sejam

bons. Porém como o rosto da mulher cobre todas as linhas uma remoção horizontal muito grande provoca sérias deformações. Por isso é conveniente uma remoção leve, de até cinquenta linhas, obtendo o resultado com pequenas deformações próximas aos lábios e contorno do rosto.

Na Figura 10(c), a imagem produz bons resultados em reduções de colunas devido ao fato de conter superfícies mal delineadas e com cores mais escuras, de modo que os cortes são despercebidos. Porém as reduções horizontais devem ser leves com reduções até 50 linhas, do contrário a deformação ocorre fazendo ondas na água.

6.1.2 Imagens que produziram resultados ruins



Figura 12: Imagens originais que produziram resultados ruins



Figura 13: Resultado do redimensionamento

A imagem “papa.ppm” possui dimensões 640x480 e tamanho 3,4MB; a imagem “pracasete.ppm” possui dimensões 750x562 e tamanho 3,7MB; a imagem “london.ppm” possui dimensões 1024x768 e tamanho 7,8MB.

Na Figura 12(a), o algoritmo produz bons resultados para remoção em até 50 linhas. A partir disso, a imagem começa a deformar devido a grande variedade de cores presente na imagem. O mesmo acontece para a remoção de colunas, que mesmo com um pequeno valor de cortes a serem feitos percebemos nitidamente a deformação da imagem. Neste caso, a deformação ocorre na cruz pois é onde possui uniformidade de cores.

Na Figura 12(b), a imagem é indicada para remoções verticais e horizontais leves, inferior a cinquenta, pelo fato da imagem conter muitos elementos. As deformações acontecem no contorno dos prédios e nas janelas, causando um aspecto de tombamento e entortamento, na torre central a ponta é cortada e os postes de luz também sofrem deformação.

Na Figura 12(c), com um número pequeno de remoção de colunas o algoritmo produz bons resultados. Em contrapartida, para remoções mais significativas percebemos nitidamente a deformação no cabide e nas roupas. Isso se deve ao fato de a imagem conter espaços totalmente preenchidos, e no caso do cabide que é onde contém menos variação de cores.

6.1.3 Conclusão da análise qualitativa

Imagens com partes vazias ou mais uniformes, poucos elementos e presença de superfícies com cores escuras e sem padrões (como foi o exemplo da imagem “furnas.ppm” devido ao barranco) obtemos um bom resultado. Em contrapartida imagens muito cheias, com superfícies bem delineadas (como foi o exemplo dos prédios na imagem “pracasete.ppm”), ou que não possuem partes vazias, que contêm muitos elementos importantes e ainda onde as partes vazias são áreas claras, as deformações são evidentes. Isso ocorre devido ao fato de que haverá poucos lugares que não fará diferença em serem cortados, o que implica que apenas remoções leves devem ser realizadas para que não haja deformação.

Para contornar as deformações dos resultados ruins, sugerimos a utilização de outras funções de energia. O método de filtragem digital possui os operadores que consistem na aplicação de técnicas de transformação, ou seja, aplicação de filtros ou máscaras com o objetivo de corrigir, suavizar ou realçar determinadas características de uma imagem dentro de uma aplicação específica. Podemos citar para o melhoramento das imagens ruins, o operador de energia Laplaciano que é insensível a direção da borda e,

portanto capaz de realçar as bordas em qualquer direção, porém seu uso é restrito devido a sua grande sensibilidade a ruído. Ele utiliza para cálculo a derivada parcial segunda em relação a x e y e tende a realçar descontinuidades nos tons de cinza, desrealçando regiões com nível de cinza constante. Neste trabalho, implementamos como extra para correção das deformações o Operador de Scharr que está descrito na seção 7.

Dessa forma, os operadores baseados na primeira derivada (no caso do Sobel) determinam o presença da borda quando o resultado é superior a um certo limiar. Em geral, essa operação resulta em bordas com muitos pixels ou espessas. Já os operadores baseados na derivada segunda as máscaras podem ser usadas para a detecção de linhas horizontais, verticais e diagonais.

6.2 Análise quantitativa

6.2.1 Imagens que produziram bons resultados

Nos gráficos abaixo observamos que a imagem “furnas.ppm” foi a que obteve resultado em maior tempo na remoção de colunas. Podemos justificar esse fato com base nas características da imagem, pois possui maior variação de cores na vertical, diferente das demais que possuem espaços com cores mais homogêneas. Já na remoção de linhas a situação se inverte, pois nas imagens “circusperformer.ppm” e “anders.ppm” não há locais onde possamos cortar sem passar pelos personagens, logo a energia dos caminhos serão altas. Na imagem “furnas.ppm” a retirada de linhas é mais rápida, pois o céu favorece o corte. O consumo de memória é cerca de 2,5 vezes o tamanho da imagem tanto para remoção de linhas quanto para a remoção de colunas.

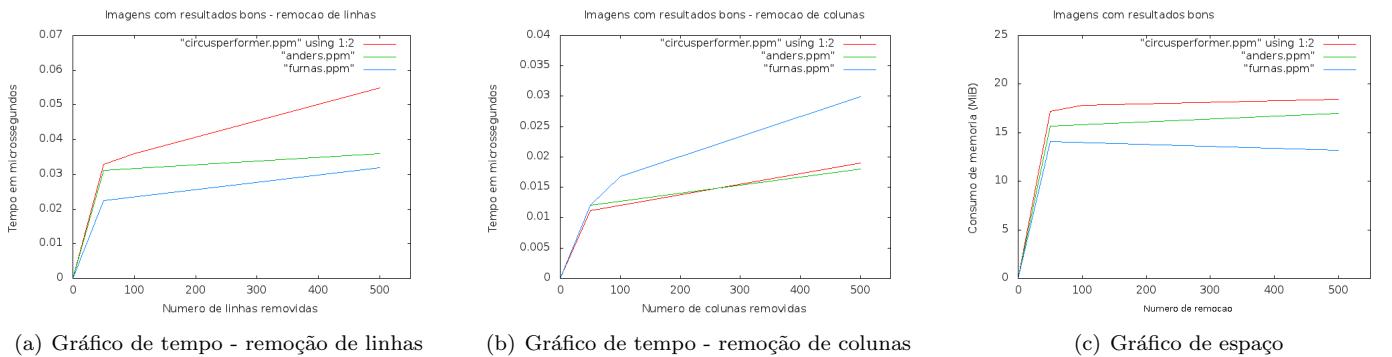


Figura 14: Análise quantitativa - Programação Dinâmica - Resultados bons

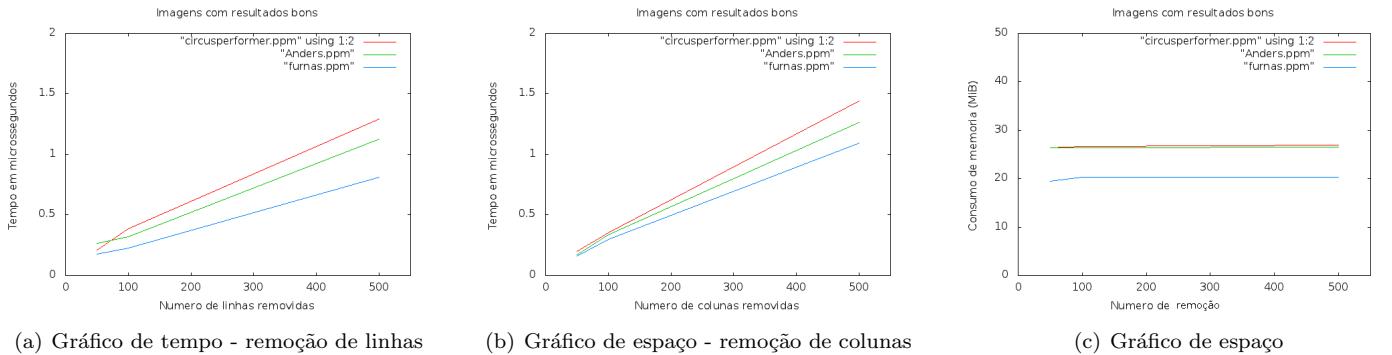


Figura 15: Análise quantitativa - Grafos - Resultados bons

6.2.2 Imagens que produziram resultados ruins

Nos gráficos abaixo observamos que a imagem “london.ppm” é onde possui um consumo maior de tempo em relação ao corte de linhas e colunas. Isso se deve ao fato de a imagem possuir uma grande heterogeneidade de cores e não possuir espaços vazios, fazendo com que ocorra deformações na imagem durante o redimensionamento. Observamos que o consumo de memória é 2,5 vezes o tamanho da imagem, tanto para remoção de linhas, quanto para a remoção de colunas.

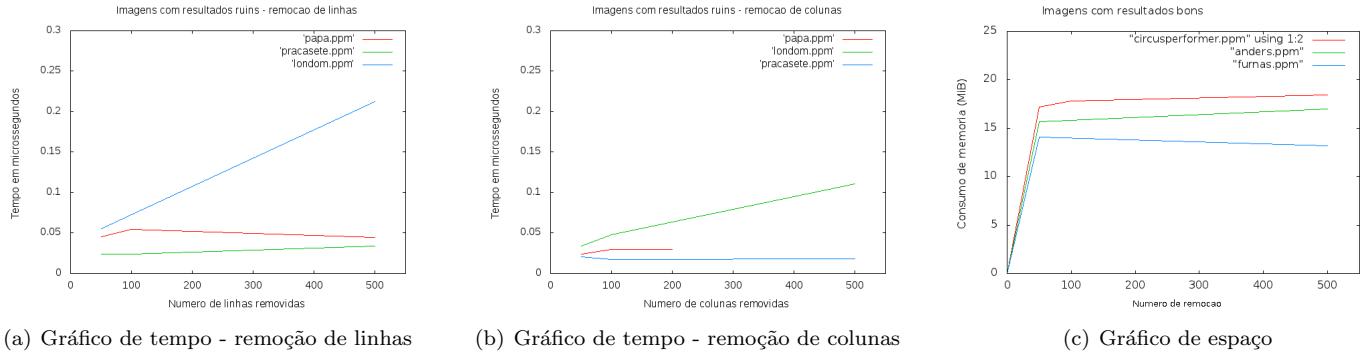


Figura 16: Análise quantitativa - Programação dinâmica - Resultados ruins

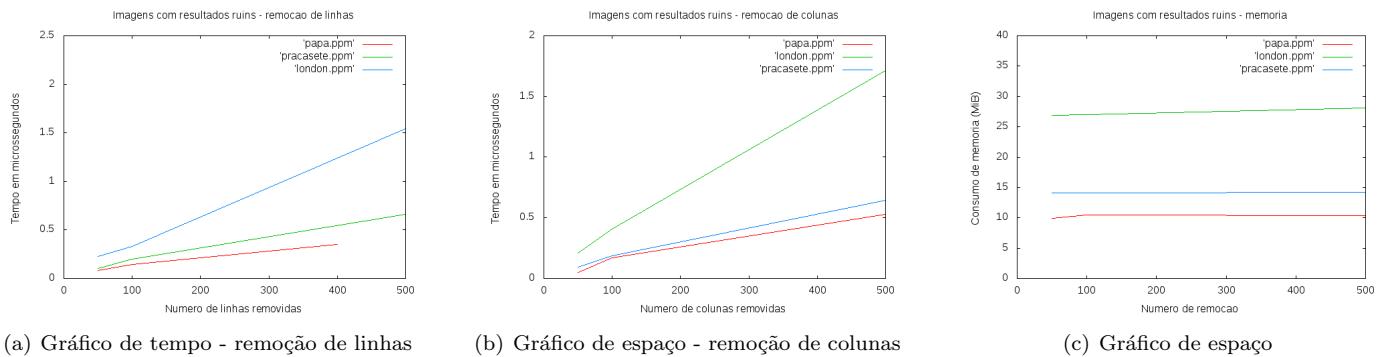


Figura 17: Análise quantitativa - Grafos - Resultados ruins

7 Extra

Como extra propomos a implementação de outra função de energia. Para tal, utilizamos o Operador de Scharr.

Enquanto o operador de Sobel reduz artefatos associados com a diferença central dos operadores, ele não possui um eixo de simetria perfeito. Scharr observou essa propriedade e tentou aperfeiçoá-la desenvolvendo uma nova função de energia.

$$\begin{bmatrix} +3 & +10 & +3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix} \quad \begin{bmatrix} +3 & 0 & -3 \\ +10 & 0 & -10 \\ +3 & 0 & -3 \end{bmatrix}$$

Figura 18: Máscara de energia com o Operador de Scharr

O operador de Scharr é resultado de uma otimização de Sobel minimizando o erro da raiz quadrada da expressão na transformada de Fourier. O resultado é um filtro feito sobre a condição onde o resultado é consistente com a realidade. Portanto o filtro calcula realmente o vetor gradiente mais perfeito possível, ao invés de simplesmente se manter a simetria da máscara.

A vantagem deste filtro em relação ao filtro de Sobel é que ele é menos radical com relação às bordas dos objetos, produzindo um efeito menos distorcido nas imagens. A Figura 19 apresenta um exemplo de uma imagem arbitrária com a aplicação do Operador de Sobel e Operador de Scharr.

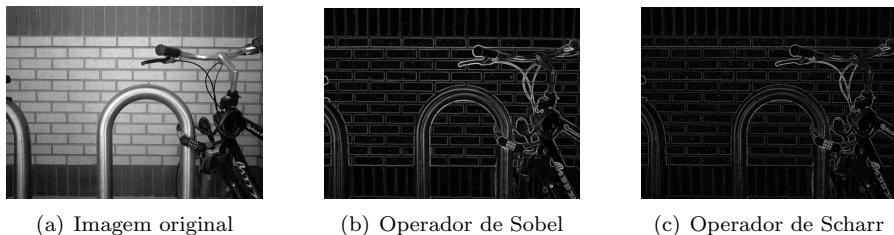


Figura 19: Comparação entre os operadores de energia

8 Conclusão

O planejamento e desenvolvimento deste trabalho permitiu o conhecimento de um tema muito utilizado, assim como permitiu aplicar conceitos importantes de desenvolvimento, como a utilização do paradigma da programação dinâmica e a modelagem através de grafos.

Observamos que o resultado da implementação utilizando programação dinâmica é o mesmo da solução utilizando grafos, o que garante que o algoritmo possui solução ótima, garantindo sempre o menor caminho.

Como trabalhos futuros, sugerimos a aplicação de outros operadores de energia, pois apesar de o Operador de Sobel ser bastante simples e possuir várias vantagens, ele possui inúmeros casos com resultados ruins. Características como muita variação de cor ou imagens sem partes vazias ou uniformes causam a deformação da imagem. Propomos também a implementação da remoção nos dois sentidos simultaneamente e a ampliação de imagens. Pode-se implementar também outros paradigmas ou técnicas de programação que visem otimizar ainda mais o tempo gasto e o consumo de memória.

A principal dificuldade enfrentada neste trabalho foi a implementação e representação da modelagem por grafos.

Conseguimos atingir os objetivos deste trabalho, que eram desenvolver um algoritmo eficiente em termo de tempo e espaço, bem como comparar os resultados obtidos.

9 Referências Bibliográficas

- [1] Ziviani, Nívio. Projetos de Algoritmos com implementações em Pascal e C, 2011.
- [2] Kernighan, Brian W., Ritchie, Dennis M. C - A linguagem de programação padrão ANSI, 1989.
- [3] Cormen, Thomas H. [et al] - Algoritmos - tradução da 2^a edição americana, 2002.
- [4] <http://netpbm.sourceforge.net/doc/ppm.html>. Acessado em
- [5] Avidan, Shai, Shamir, Ariel - Seam Carving for Content-Aware Image Resizing
- [6] Liu,Jingwei - Introduction to Digital Image Processing - University of Southern California
- [7] Rocha, Leonardo Chaves Dutra da - Algoritmos em Grafos e Paradigmas de Projeto de Algoritmo - 2012