TGC Implementação 4 - Segmentação de Imagens

André M. Rodrigues[1], **Arthur Martinho M. Oliveira**[2]

[1]Instituto de Ciências Exatas e Informática – PUC-MG

amrodrigues@sga.pucminas.br, arthur.martinho@sga.pucminas.br

Abstract. *This paper presents the implementation of "Implementação 4" for the course Teoria dos Grafos e Computabilidade, where two methods related to image segmentation using graph-based algorithms were developed. The methods were based on the works of Felzenszwalb and Huttenlocher in "Efficient Graph-Based Image Segmentation" and of Boykov and Funka-Lea in "Graph Cuts and Efficient N-D Image Segmentation". Both studies employ graph structures to identify segments in image data, despite some implementation differences.*

Resumo. *Este artigo aborda a realização da "Implementação 4" da disciplina de Teoria dos Grafos e Computabilidade, onde foram desenvolvidos dois métodos relacionados à segmentação de imagens utilizando algoritmos com base em grafos. Os métodos foram baseados nos trabalhos de Felzenszwalb e Huttenlocher em "Efficient Graph-Based Image Segmentation" e de Boykov e Funka-Lea em "Graph Cuts and Efficient N-D Image Segmentation". Ambos os estudos utilizam estruturas de grafos para identificar segmentos nos dados de imagem, apesar de algumas divergências na implementação.*

1. Introdução

A segmentação de imagens é um problema clássico da computação, com aplicações que vão desde a detecção de objetos e edição de imagens até o processamento de imagens médicas. Ao longo dos anos, diversos algoritmos foram desenvolvidos para resolver esse desafio, utilizando tanto técnicas heurísticas quanto de otimização matemática. Dentre esses, as abordagens baseadas em grafos se destacam como ferramentas importantes devido à sua capacidade de modelar as relações entre pixels e regiões.

Esta implementação foca em dois métodos originados da segmentação baseada em grafos. O primeiro, proposto por Felzenszwalb e Huttenlocher, utiliza um critério baseado em componentes e regiões que se ajusta dependendo da variabilidade entre vizinhanças locais. O segundo método, desenvolvido por Boykov e Funka-Lea, utiliza cortes em grafos para segmentação de imagens de diferentes dimensões. Nosso objetivo é observar essas metodologias e implementar os algoritmos, discorrendo sobre estrutura de dados para seu funcionamento.

2. Trabalhos relacionados

2.1. Efficient Graph-Based Image Segmentation

O método apresentado no primeiro artigo representa a imagem como um grafo, em que os vértices correspondem aos pixels e as arestas conectam pixels vizinhos. Os pesos das arestas são calculados com base nas diferenças de intensidade entre os pixels adjacentes, considerando separadamente os canais de cor R (vermelho), G (verde) e B (azul).

A segmentação é realizada através de uma abordagem inspirada no algoritmo de árvore geradora mínima (MST), que detecta regiões segmentadas ao unir componentes conexos. A união entre componentes ocorre conforme um predicado baseado nas diferenças internas de cada componente para os canais R, G e B. A união é permitida somente quando o peso da aresta não excede um valor limite, definido pela soma da diferença interna mínima e um parâmetro (K).

A diferença interna de um componente é atualizada a cada nova união e é calculada como o máximo entre as diferenças de intensidade observadas durante as uniões de pixels (arestas) nos canais R, G e B. Essa lógica permite uma segmentação com boa precisão, especialmente em imagens coloridas.

2.2. Graph Cuts and Efficient N-D Image Segmentation

O método do segundo artigo apresenta uma construção de grafo semelhante, porém com a introdução de vértices e arestas especiais. Os pixels são representados como vértices, e as relações entre pixels vizinhos são modeladas por arestas ponderadas. As arestas são classificadas em t-links e n-links, em que as t-links conectam os pixels a dois vértices auxiliares, o source (representando o objeto) e o sink (representando o fundo), e as n-links conectam pixels adjacentes e representam as fronteiras entre as regiões da imagem.

Os pesos das arestas t-links são calculados com base nas probabilidades de um pixel pertencer ao objeto ou ao fundo, enquanto os pesos das n-links são definidos por uma medida de similaridade de intensidade entre pixels vizinhos. A solução determina a classificação de cada pixel como pertencente ao objeto ou ao fundo, respeitando as informações de seeds manuais (restrições duras) e as relações entre vizinhos (restrições suaves). Esta metodologia destaca-se pela capacidade de lidar com imagens complexas.

3. Metodologia

Para a metodologia, adotamos uma implementação para cada um dos artigos em questão. Em ambas, separamos os códigos em bibliotecas, para fragmentar o problema em problemas menores e resolver um de cada vez. Apesar de alguns passos terem sido semelhantes, discutiremos os aspectos do desenvolvimento de ambas as soluções nessa sessão.

3.1. A Primeira Implementação

Trabalhamos com imagens no formato PPM, pois estas são mais fáceis de manipular. Inicialmente, a imagem é carregada em uma matriz de ”Pixel”, sendo ”Pixel” uma *struct* criada para os canais R, G e B do pixel.

```
1 struct Pixel {
2     int r, g, b; // Valores de cor (0-255)
3 };
```

Listing 1. struct Pixel

A *struct* ajuda a armazenar as intensidades das cores e facilita na atribuição dos pesos das arestas posteriormente. A matriz então é preenchida para cada pixel da imagem, sendo feito o tratamento para cada um dos formatos que a imagem pode assumir, com P3 sendo formato texto e P6 sendo formato binário. Essa manipulações foram feitas em *imageloader.hpp*.

Em seguida, a partir da representação da imagem, criamos o grafo (em *graph.hpp*) em que cada pixel corresponde a um vértice, e as arestas conectam pares de pixels vizinhos, com o peso de cada aresta sendo a dissimilaridade entre os pixels. Foi criada uma *struct* "Edge" para representar a aresta, e essa struct contém variáveis para os vértices de origem, de destino e o peso para cada um dos canais R, G e B. Após isso, foi implementado o construtor do grafo, que usa a imagem já como matriz de "Pixel", sua altura e sua largura. O construtor terá o trabalho de adicionar as arestas e seus respectivos pesos, e o método que faz esse cálculo é dado por:

```
vector<double> calculateWeights(const Pixel& p1, const Pixel& p2
    ) {
     double diffR = abs(p1.r - p2.r);
     double diffG = abs(p1.g - p2.g);
     double diffB = abs(p1.b - p2.b);
     return {diffR, diffG, diffB};
}
```

Listing 2. Método para calcular pesos

Como pode-se observar, *calculateWeights* calcula o peso para cada um dos canais R, G e B separadamente.

Com o grafo pronto, os próximos passos são construir os componentes e realizar a segmentação. No construtor da classe *GraphComponents* inicializamos um componente para cada vértice, atribuindo ele mesmo como seu "pai". Essa lógica foi implementada para facilitar a união de componentes, onde vértices que pertencem ao mesmo componente tem o mesmo "pai". No método *findMST*, utilizamos o algoritmo de Kruskal para gerar a árvore geradora mínima (MST) de cada componente. Em seguida, começamos a juntar componentes no método *segmentGraph*. A fórmula geral do predicado de comparação para juntar componentes é dada por:

$$D(C_1, C_2) = true \quad se \quad Dif(C_1, C_2) > MInt(C_1, C_2)$$

Onde:

- Dif(C_1, C_2): menor peso da aresta entre C_1 e C_2.
- MInt(C_1, C_2): mínimo entre $Int(C_1) + \tau(C_1)$ e $Int(C_2) + \tau(C_2)$.
- $\tau(C) = \frac{k}{|C|}$, onde k é uma constante e $|C|$ é o tamanho da região.
- Int(C) consiste em:

$$Int(C) = \max_{e \in MST(C,E)} w(e).$$

Essa etapa repete-se "e" vezes, sendo "e" o número total de arestas. Finalmente, teremos o menor número de componentes possível.

A classe *ImageColorizer* fornece funcionalidades para colorir e salvar a segmentação de uma imagem em componentes. O método principal, *colorComponents*, atribui uma cor aleatória a cada componente identificado no grafo, usando um mapeamento entre a "pai" do componente e uma cor única. Em seguida, cada pixel da imagem é colorido de acordo com o componente ao qual pertence. Por último, *savePPM* salva a imagem final no formato PPM.

3.2. A Segunda Implementação

Utilizamos uma estrutura de grafo que emprega o algoritmo de min-cut/max-flow para separar os pixels entre o objeto (source) e o fundo (sink). A estrutura do grafo nesse caso se difere um pouco do primeiro artigo, sendo que nesse as arestas e os vértices possuem um tratamento especial. O processo de construção do grafo e a segmentação da imagem serão detalhados nessa sessão.

O grafo é composto por dois tipos principais de arestas, sendo esses:

- Arestas N-links: Arestas entre pixels vizinhos que fazem a relação entre eles. O peso dessas arestas é calculado com base na diferença de intensidade de cor entre os pixels, utilizando uma função exponencial com o parâmetro σ. A fórmula para o peso das arestas é dada por:

$$w(u, v) = \exp\left(-\frac{diff^2}{2\sigma^2}\right)$$

  Onde $diff$ é a diferença de cor entre os pixels u e v.
- Arestas T-links: Arestas que conectam os pixels ao *source* ou ao *sink*, representando a probabilidade de um pixel pertencer ao objeto ou ao fundo, respectivamente. O peso dessas arestas é calculado usando o valor negativo do logaritmo das probabilidades fornecidas:

$$PesoSource = -\log(ProbObjeto), \quad PesoSink = -\log(ProbFundo)$$

  Quando um pixel é marcado como *seed* (semente) do objeto, a aresta ao *source* terá peso infinito, e o mesmo vale para a *seed* de fundo em relação ao *sink*.

O grafo foi construído com base nos pixels da imagem, adicionando os vértices auxiliares *source* e *sink*. O construtor do grafo recebe alguns parâmetros adicionais em relação ao da primeira implementação, como o valor de σ e as seeds para o objeto e o fundo. Essas seeds são atribuídas aos vértices *source* e *sink*, respectivamente.

Com o grafo construído, o próximo passo é aplicar o algoritmo de min-cut/max-flow para segmentar a imagem. Para isso, utilizamos o algoritmo *Edmonds-Karp*, que é uma implementação do algoritmo de Ford-Fulkerson para encontrar o máximo fluxo em redes.

```
double maxFlow() {
        vector<int> parent(numVertices);
        double flow = 0.0;
        while (bfs(parent)) {
            double pathFlow = numeric_limits<double>::max();
            for (int v = sink; v != source; v = parent[v]) {
                int u = parent[v];
                pathFlow = min(pathFlow, capacity[u][v]);
            }
            for (int v = sink; v != source; v = parent[v]) {
                int u = parent[v];
                capacity[u][v] -= pathFlow;
                capacity[v][u] += pathFlow;
```

```
            }

            flow += pathFlow;
        }
        return flow;
    }
```

Listing 3. algoritmo Edmonds-Karp utilizado

Após a execução do algoritmo de min-cut/max-flow, os pixels que permanecem conectados ao *source* são classificados como pertencentes ao objeto. Esses pixels então são extraídos e agrupados. A segmentação resulta em dois segmentos: o objeto e o fundo. Para exibir a segmentação, os pixels do fundo serão coloridos de azul, enquanto os pixels do objeto serão coloridos de vermelho. Para garantir que o código funcione corretamente com diferentes imagens, alguns parâmetros precisam ser ajustados manualmente na `main`:

- **Sigma**: Controla a suavização da diferença entre os pixels vizinhos.
- **Probabilidade Inicial**: Determina a probabilidade inicial de um pixel ser classificado como objeto ou fundo.
- **Seeds**: Define os pixels que servirão como semente para o objeto e para o fundo.

```
double sigma = 10.0;
vector<double> objProb(width * height, 0.5); // Probabilidade
    inicial de ser objeto
vector<double> bgProb(width * height, 0.5);  // Probabilidade
    inicial de ser fundo

vector<bool> isSeedObject(width * height, false);
vector<bool> isSeedBackground(width * height, false);

isSeedObject[1] = true;
objProb[1] = 0.99; // Alta probabilidade de objeto
bgProb[1] = 0.01;

isSeedBackground[15] = true;
objProb[15] = 0.01; // Alta probabilidade de fundo
bgProb[15] = 0.99;
// ...
```

Listing 4. configurações feitas na main

4. Resultados

4.1. Primeira Implementação

Os resultados obtidos são bastante similares aos apresentados no artigo de referência, com a principal diferença na suavização da imagem. No artigo base, os autores aplicam uma suavização com um valor de 0,8 no código deles, enquanto na nossa implementação não utilizamos nenhum tipo de suavização. Essa diferença pode ter impacto na qualidade da

segmentação, especialmente em imagens com maior ruído, onde a suavização ajudaria a melhorar a precisão na separação entre o objeto e o fundo.

Figure 1. Imagem Exemplo Original

Figure 2. Imagem após Aplicação do Algoritmo

4.2. Segunda Implementação

Devido à elevada complexidade do segundo artigo, os testes foram realizados em imagens menores, especificamente com dimensões de 5x5 pixels. Essa escolha visou simplificar o processamento e facilitar a análise dos resultados, garantindo que a implementação pudesse ser avaliada de forma mais prática e eficiente.

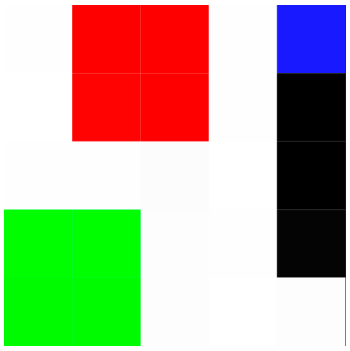

Figure 3. Imagem Exemplo original

Para o teste realizado, o código foi configurado com os seguintes parâmetros:

- $\sigma = 10$: valor que controla a suavização entre os pixels vizinhos.
- Probabilidade inicial de ser objeto: 0.5.
- Probabilidade inicial de ser fundo: 0.5.
- Seed Objeto: pixel 15 (cor verde).
- Seed Fundo: pixel 0 (cor branca).

Os resultados obtidos com essa configuração são apresentados a seguir:

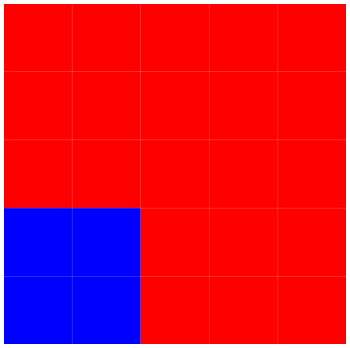

Figure 4. Imagem de saida do algoritmo

5. Conclusão

De maneira geral, conseguimos obter resultados satisfatórios após a implementação dos algoritmos descritos nos artigos. Contudo, devido ao alto custo computacional de nossas soluções, não obtivemos uma boa eficiência para imagens maiores, especialmente no caso da segunda implementação.

Apesar das tentativas, nossas implementações não ficaram totalmente em conformidade com a dos referidos artigos, principalmente em relação à otimização. Entretanto, conseguimos implementar as principais estruturas e lógicas dos estudos. Os principais desafios que encontramos foram na compreensão da lógica do segundo artigo, que utiliza um conceito diferentes de vértices auxiliares (*source* e *sink*) e na compreensão de alguns pontos do primeiro artigo, como a implementação da segmentação propriamente dita.

Em suma, o trabalho proporcionou uma melhor visão a respeito da complexidade de algoritmos de processamento de imagens, e nos ajudou a desenvolver nossas habilidades de implementação de soluções baseadas em grafos. Percebemos também a importância da eficiência computacional e da otimização no contexto de problemas reais, o que será valioso para nosso futuro.

References

[1] P. F. Felzenszwalb and D. P. Huttenlocher, *Efficient Graph-Based Image Segmentation*, International Journal of Computer Vision, vol. 59, no. 2, pp. 167-181, 2004.

[2] Y. Boykov and G. Funka-Lea, *Graph Cuts and Efficient N-D Image Segmentation*, International Journal of Computer Vision, vol. 70, no. 2, pp. 109-131, 2006.