Relatório 2º Projecto de ASA

Grupo 154
João Daniel Silva 86416 & Francisco Sousa 86416

2 de Maio de 2018

1 Introdução

No âmbito do segundo projeto da cadeira de Análise e Síntese e Algoritmos, foi-nos proposto um problema de segmentação de imagens, no contexto da análise de imagens obtidas de cameras de carros autómatos.

Considerando que uma imagem é um conjunto de pixeis, pretende-se que cada um seja classificado como sendo de primeiro plano ou de cenário. No enunciado foi definido que o algoritmo a utilizar consistiria num processo de minimizar uma soma de peso, ou seja, o problema pode ser representado como um problema de fluxos.

Cada vértice tem um peso de primeiro plano *lp*, cenário *cp* e de vizinhança *fv*. Todos estes pesos contribuem para a soma de peso total. Cada vértice tem 4 vizinhos, o de cima, o de baixo, o da esquerda e o da direita. O peso total de uma segmentação é calculada com a seguinte fórmula:

$$\sum_{p \in L} lp + \sum_{p \in C} cp + \sum_{v \in V} fv$$

Para modelar este problema recorremos à teoria de grafos para representar **fluxos**. É criado um grafo dirigido e pesado, em que cada vértice corresponde a um pixel, cada arco entre dois vértices tem uma capacidade e pode receber um fluxo. Ao conjunto de vértices que representam os pixeis são adicionados dois que representam a **source** e a **sink**.

A source conecta-se a todos os pixeis e cada aresta tem a capacidade correspondente ao peso de primeiro plano de cada vértice; analogamente, todos os vértices conectam-se à sink com a capacidade correspondente ao peso de cenário; cada aresta entre pixeis vizinhos tem a capacidade da vizinhança definida no input.

Implementámos a nossa solução na linguagem de programação C++ para facilitar a implementação de estruturas de dados. O algoritmo escolhido para

a resolução do problema de fluxos foi o **Algoritmo de Edmonds-Karp**, pois tem uma implementação relativamente simples em relação a outros, como o **Algoritmo de Dinic**.

2 Descrição da Solução

2.1 Estruturas

Cada vértice é representado por uma **struct vértice** que armazena a capacidade da rede residual das ligações aos seus 4 vizinhos (*north*, *south*, *east*, *west*). Como a source e a sink se podem ligar a todos os vértices, optámos por representar em cada **struct vértice** também a capacidade da rede residual da ligação de cada vérice a estes dois, evitando criar uma estrutura separada que armazenasse as V-ligações que a source e a sink têm.

Como apenas armazenamos a capacidade da aresta, para sabermos a que vértice a aresta aponta definimos as funções **relatdPos** e **getVertixInDir** que devolvem, respetivamente, a direção entre dois vértices e o vértice apontado dada uma direção. Estas funções serão chamadas quando for necessário percorrer os vértices adjacentes de um vértice.

2.2 Algoritmo

Ao ler o input, é criado o grafo de $N * M$ vértices e de seguida a rede residual de cada aresta e das suas inversas é atualizada.

Para otimizar o tempo de execução, antes de aplicar o **Edmonds-Karp**, atualizamos para cada vértice a rede residual entre a *source-vértice-sink* para o mínimo dos flows que é possível passar nas ligações *s-vértice* e *vértice-sink*. Assim, reduzimos número de arcos que a **BFS** irá percorrer para procurar *augmenting paths* durante o **Edmonds-Karp**.

Para facilitar a implementação da **BFS**, usamos a queue do **std::queue**; como não armazenamos os vértices adjacentes à source, temos de percorrer todos os vértices e verificar se ainda é possível receber flow da source, e posteriormente adicioná-los à queue.

De seguida, corremos a **BFS** de forma convencional, atualizando o **array pi**, que representa o predecessor de cada vértice, que terá a informação do caminho mais curto. Após ser verificado quanto flow pode ser enviado, as ligações são atualizadas. Este processo é repetido enquanto houver um *augmenting path*. Ao finalizar o algoritmo, é devolvido o *max-flow* do fluxo.

O teorema *max-flow min-cut* afirma que o valor máximo do fluxo desde a source até à sink é igual ao peso das arestas no *min-cut*. Com isto em

mente, percorremos uma **DFS** desde a source e verificamos quais vértices são possíveis de descobrir. Esses serão os vértices que farão parte do cenário e os restantes serão os de primeiro plano.

3 Análise

3.1 Teórica

O algoritmo **Edmonds-Karp** tem complexidade $O(VE^2)$. A cada iteração, o tamanho do caminho mais curto desde a source até aos outros vértices varia de forma monótona crescente, pois os mais curtos são sucessivamente esgotados, tendo cada caminho no máximo O(E) arestas. O número máximo de iterações é V*E, pois há O(E) pares de vértices que para cada dois vértices u e v podem ficar saturados O(V) vezes.

Na separação dos pixeis em dois grupos de vértices na deteção do *min-cut* a **DFS** demora $O(V + E)$, no entanto é majorada pelo **Edmonds-Karp**.

Assim, podemos concluir que a complexidade final do algoritmo é $O(VE^2)$.

3.2 Experimental

Para esta análise utilizámos o gerador de inputs disponibilizados pelos docentes. Apoiados pelo gráfico 1, podemos verificar que a execução do algoritmo é polinomial de grau 2, em relação ao número de vértices.

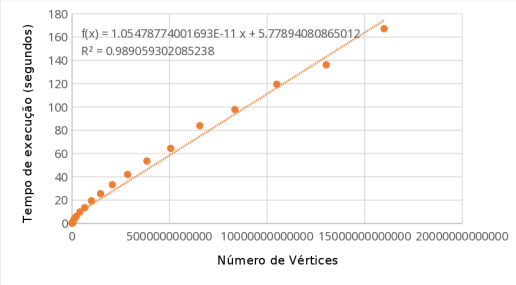

Figura 1: Variação do número de vértices V^2 em função do tempo de execução do programa

A diferença em relação à análise teórica pode-se explicar pelo facto de que, para os inputs gerados, a procura de caminhos mais curtos na **BFS** do **Edmonds-Karp** não precisa de percorrer muitas arestas, se as ligações que saiem da source ficam saturadas rapidamente. Esta saturação também

é acelerada pelo processo de saturação das ligações *s-vértice* e *vértice-sink* explicado acima, antes da execução do **Edmonds-Karp**.

O pequeno desvio do gráfico é explicado pelos processos a correr em paralelo no computador, que interferem na execução.

4 Referências

- https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for...
- https://brilliant.org/wiki/edmonds-karp-algorithm/