Análise e Síntese de Algoritmos 2º Projeto

Grupo 91

João Bernardo – 86443 | Pedro Antunes – 86493

Introdução

O presente relatório aborda a solução encontrada ao problema proposto pelo corpo docente no segundo projeto do 2º Semestre do ano curricular 2017/2018.

O problema proposto aborda a segmentação dos pixéis de uma imagem como sendo de primeiro plano ou de cenário. É-nos pedido que realizemos essa segmentação, de modo a ser possível implementar um sistema de veículos sem motorista para uma empresa de distribuição de mercadorias.

Recebendo um input com as dimensões *m x n* dos pixéis da imagem, os pesos *lp* (primeiro plano) e *cp* (cenário) associados a cada píxel e os pesos associados às relações de vizinhança horizontais e verticais, o programa deve calcular o peso total da segmentação mínima para a figura em questão e classificar cada píxel como sendo de primeiro plano ou cenário.

Descrição da Solução

A implementação do programa foi elaborada em linguagem C.

O conjunto dos píxeis pode ser visto como um grafo, em que cada píxel é um vértice e as relações de vizinhança são os arcos do grafo.

Deste modo, após ser recebido o input descrito anteriormente, é alocado um array, *vertexes,* de estruturas *vertex* (um *vertex* tem um peso *lp* e um peso *cp*). Criámos também 2 vértices adicionais, a *source* e o *target*. Começamos por ligar a *source* a todos os vértices com arestas de capacidade *lp* e ligamos todos os vértices ao *target* com arestas de capacidade *cp.* Igualamos o fluxo destas arestas ao mínimo das duas capacidades, de forma a saturar os caminhos de aumento do tipo {(source, p),(p, target)} (caminhos mais curtos).

Alocamos também outro array chamado *edges* que guarda todas as arestas do nosso grafo. Cada aresta, *edge*, possui uma origem, destino, fluxo e capacidade. O array edges é ordenado pelos pontos de origem e de seguida pelos pontos de destino utilizando o quicksort. Desta forma será mais fácil aceder às arestas de um determinado vértice.

Para isso, é alocado um array de inteiros, *locator,* em que cada entrada indica o índice em *edges* na qual se iniciam as arestas de cada vértice. Por exemplo, se na terceira entrada do *locator* estiver o inteiro 6, significa que as arestas do vértice 3 se iniciam no índice 6 de *edges*.

No seguimento do programa, é alocado outro array de inteiros denominado *pred* que vai guardar o predecessor de cada vértice. Por exemplo, se depois de realizada uma BFS no índice 5 de *pred* estiver o inteiro 1, significa que o predecessor do vértice 5 é o vértice 1.

Criamos também outro array, denominado *pred_index* que vai guardar todas as arestas necessárias para chegar da source ao target em cada BFS.

Com o auxílio destas estruturas, deveremos determinar a segmentação que minimiza o peso total, i.e teremos de encontrar o corte mínimo do grafo, calcular a capacidade do corte e mostrar que vértices pertencem a que parte do corte. Como podem existir vários cortes mínimos e queremos o que maximiza o plano principal, consideramos que os vértices ligados ao target são de cenário. Ou seja, queremos o corte mínimo mais próximo do target.

Para calcular o fluxo máximo optámos por implementar o algoritmo *Edmonds-Karp* com uma lista de adjacência estática. Como já referimos anteriormente, para acedermos às arestas de um vértice acedemos ao índice de *locator* com o mesmo número do vértice, que por sua vez nos dá o índice em *edges* onde começam as arestas do vértice.

Depois de executado o algoritmo, onde é corrida uma BFS e saturados todos os caminhos possíveis, é devolvido o fluxo máximo do grafo. Falta então calcular o corte mínimo que maximize o número de píxeis do plano principal. Para isso, corremos de novo uma BFS. Se um vértice for visitado classificá-lo-emos como píxel de cenário; caso contrário, será um píxel de primeiro plano.

Para finalizar o programa, o output será uma linha com o peso total da segmentação mínima para a figura em questão e uma sequência de *m* linhas com *n* caracteres cada em que os caracteres são 'P' ou 'C', conforme o píxel em questão é classificado como primeiro plano ou cenário, respetivamente.

Análise Teórica

Sendo V o número de vértices (V = m x n) e E o número de arestas do grafo, começamos por notar que o número total de arestas que saem de um vértice é sempre igual ou inferior a 6 ($E < 6V$). Isto deve-se ao facto de, para um dado vértice, o maior número possível de vizinhanças serem 4 e o vértice se ligar sempre ao *source* e ao *target*.

A nossa solução, sendo baseada no algoritmo *Edmonds-Karp*, teria uma complexidade assimptótica de EV^2, mas uma vez que $E = O(V)$, podemos afirmar que a complexidade assimptótica deste algoritmo é V^3.

Temos também que, precisando este algoritmo apenas das estruturas auxiliares à BFS

- que têm uma complexidade temporal O(V + E) que, como já referido, equivale a O(V) na nossa solução - e das estruturas necessária para guardar os fluxos e capacidades - que têm uma complexidade O(E) = O(V), podemos afirmar que durante a execução do algoritmo *Edmonds-Karp* o espaço utilizado é O(V).

Em detalhe, temos as seguintes complexidades temporais para as diversas funções e passos do programa:

O(V) para a inicialização da array de vértices, *vertexes*, e O(E) para a inicialização da array de arestas, *edges*.

O(E) para a função *inputReader*, que efetua a leitura do input.

O(E logE) para a função *qsort* (quicksort).

O(V) para a inicialização da array *locator* e para a sua concretização deste array.

O(V) para a inicialização das três arrays de suporte à BFS, *pred*, *pred_index* e *bfsVertexes*.

$O(V^3)$, como já referido, para a função *Edmonds-Karp*.

O(V) para a função *minimumCutPrint*, que encontra o corte mínimo e classifica os píxeis da imagem.

Conclusão:

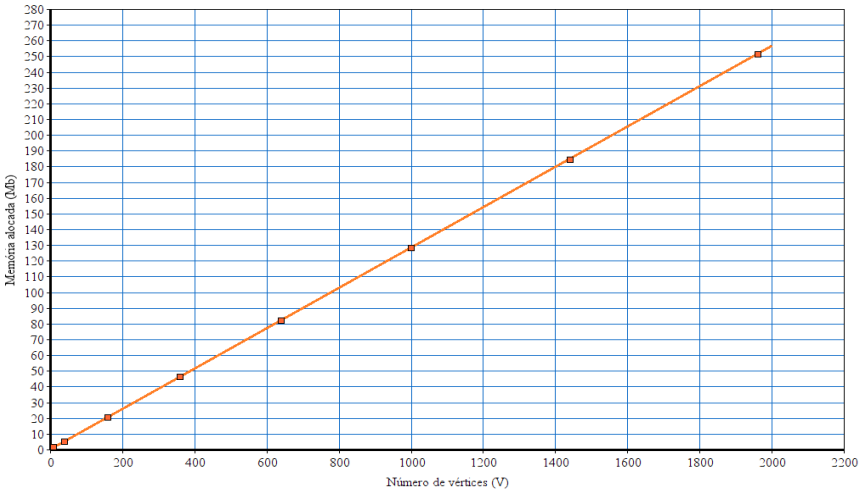
→ complexidade temporal: $O(V^3)$

→ complexidade espacial: O(V)

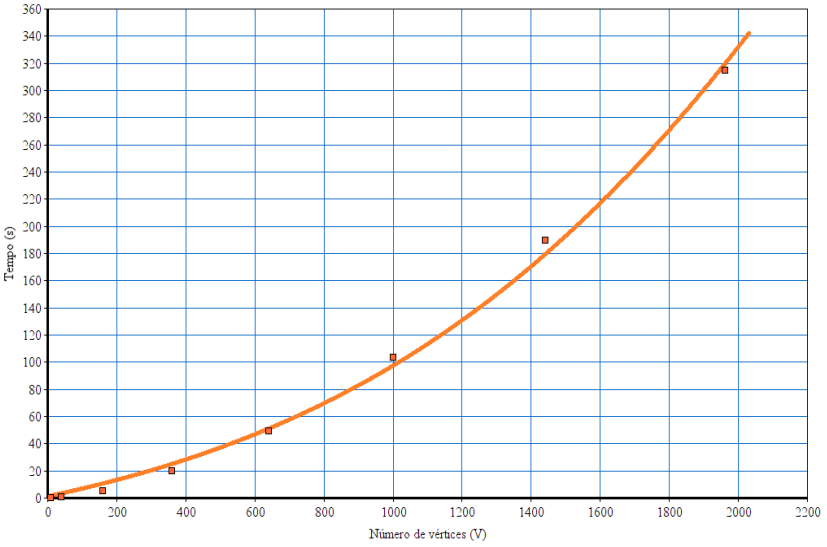
Avaliação Experimental dos Resultados

Para testar experimentalmente a eficiência do algoritmo desenvolvido, executámos o programa com alguns dos ficheiros de input disponibilizados na página da cadeira na pasta BigIO. A plataforma de testes foi um computador com processador Intel® Core™ i7-7700HQ CPU @ 2.80GHz com 16GB de memória a correr o Windows 10.

Para medir a memória utilizada (em bytes), foi utilizado o *Valgrind*, mais especificamente a ferramenta *Massif*; para medir tempos de execução (em segundos), foi utilizado o comando *time*, somando os tempos de execução em modo utilizador e sistema.



Pelo gráfico podemos concluir que a complexidade espacial do algoritmo desenvolvido é, de facto, linear, dado que a memória ocupada cresce linearmente com o número de vértices.



Com base no gráfico, constatamos que o tempo de execução cresce com o cubo do valor de V. Assim, é possível verificar que a complexidade do algoritmo é $O(V^3)$.

Valores experimentais usados na elaboração dos gráficos:

V (milhares)	Bytes (milhões)	Tempo (segundos)
10.1	1.29	0.111s
40.2	5.14	0.667s
160.4	20.51	5.333s

360.6	46.14	19.713s
640.8	81.99	49.221s
1001.0	128.09	103.228s
1441.2	184.40	189.630s
1961.4	251.02	314.422s

Referências

Os websites/obras consultados para a realização do projeto foram os seguintes:

Introduction to Algorithms, Third Edition: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein September 2009 ISBN-10: 0-262- 53305-7; ISBN-13: 978-0-262-53305-8

Edmonds-Karp algorithm - wikipedia

Efficient Graph Cuts for Multi-region Segmentation: Martin Rykfors - pdf