Projeto 2 – Análise e Síntese de Algoritmos

LEIC-A - Instituto Superior Técnico 2017/2018

Grupo al058
Rui Alves 65284
João Coelho 86448

Introdução

Este relatório faz parte dos entregáveis e suporta o código produzido para a implementação do segundo projeto da cadeira de ASA, do curso de LEIC no Instituto Superior Técnico (Alameda), ano letivo 2017/2018.

O problema apresentado pelo corpo docente tem como objetivo a segmentação de imagens. Nos dias de hoje, há vários métodos e tecnologias para o fazer, e neste projeto vamos usar um método que consiste na abstração do problema como um problema de fluxo máximo (e corte mínimo), sendo cada pixel da imagem a segmentar um vértice do grafo. Segmentar uma imagem consiste em classificar cada vértice como sendo de cenário ou primeiro plano. Assim, sendo cada imagem um retângulo de *m* por *n* pixéis, cria-se um grafo de *m* * *n* + 2 pixéis, sendo os 2 vértices extra a *source* universal e o *sink* universal necessários para os algoritmos de fluxo máximo. Em termos de arestas, teremos *m* * *n* arestas que ligam todos os pixéis à *source, m* * *n* arestas que ligam todos os pixéis ao *sink*, e cada pixel terá ainda entre mais 2 a 4 arestas, uma para cada vértice adjacente (acima, abaixo, à esquerda e à direita): 2 arestas para os pixéis de canto, 3 arestas para os restantes pixéis nos limites da imagem e 4 arestas para os restantes (pixéis interiores).

O objetivo do programa é então calcular o fluxo máximo nesse grafo e um corte mínimo que faça a segmentação da imagem, imprimindo depois a classificação de cada pixel. Convém mencionar que, havendo a possibilidade de existir mais que um corte mínimo para o mesmo grafo, pretende-se escolher aquele que maximiza o número de pixéis classificados como sendo de primeiro plano.

O *input* que o programa recebe consiste em dois números (*m* e *n*), seguidos de 4 matrizes de valores numéricos: a primeira matriz representa a capacidade das arestas *source*->pixel, a segunda matriz a capacidade das arestas pixel->*sink*, a terceira a capacidade das arestas horizontais que ligam vértices adjacentes e a quarta a capacidade das arestas verticais que ligam vértices adjacentes.

Descrição da Solução

A linguagem de programação utilizada foi C. Esta decisão foi tomada tendo em conta não só a maior proficiência dos elementos do grupo nesta linguagem em comparação com as outras alternativas (C++ e Java), como também pelo facto de nos permitir gerir mais facilmente a alocação de memória durante a sua execução, sendo que esse recurso está limitado no ambiente de testes do projeto (*mooshak*).

Ao ler o input, modela-se o grafo com uma lista de adjacências, implementada como uma lista de listas ligadas, ou seja, uma lista de vértices em que, para cada vértice, a sua lista ligada são as arestas que o têm como origem. Sendo um problema de fluxo máximo, arestas entre pixel e *source* só nos interessam no sentido *source*->pixel, arestas entre pixel e *sink* só nos interessam no sentido pixel->*sink*, mas arestas entre pixéis não são direcionadas, o que corresponde a duas arestas na forma pixel_1->pixel_2 e pixel_2->pixel_1 (uma é o inverso da outra) na nossa modelação do problema.

Cada elemento da lista de vértices contém então o identificador do vértice em questão, um caracter que classifica o pixel correspondente a esse vértice como primeiro plano ou cenário (esta informação é irrelevante para o primeiro e o último elementos da lista, a *source* e o *sink*), e dois ponteiros, um para a *head* e um para a *tail* da sua lista ligada. Cada aresta contem identificadores do pixel de origem e do pixel de destino, a sua capacidade e fluxo, um ponteiro para a aresta seguinte (o típico *next* numa lista ligada) e um ponteiro para a aresta no sentido inverso.

Tendo a lista de adjacências, aplica-se então a nossa implementação do algoritmo de *Edmonds-Karp* para fluxo máximo. Este algoritmo calcula o fluxo máximo num grafo sistematicamente encontrando novos caminhos de aumento usando *BFS* (*Breadth First Search*). Usa-se a *BFS* para garantir que esses caminhos são descobertos por ordem crescente de comprimento (número de arestas que o constituem). Um caminho de aumento é um caminho do tipo (u_1, u_2, ..., u_k) na rede residual, onde u_1 é a *source* e u_k é o *sink*, tal que a capacidade residual em cada aresta desse caminho é maior que 0. A rede residual de um grafo é um grafo com a mesma configuração do grafo principal, mas que modela a capacidade residual para cada aresta (capacidade residual = capacidade total – fluxo). A cada iteração, o fluxo é atualizado para todas as arestas do caminho de aumento. Quando já não for possível encontrar um novo caminho de aumento, o algoritmo termina e devolve o somatório do fluxo de todos os caminhos de aumento calculados, o fluxo máximo. As *BFS*s foram implementadas de forma iterativa (e não recursiva) de forma a evitar ultrapassar o limite de memória estipulado.

Finalmente, é preciso encontrar o corte mínimo. Este consiste num corte que particiona o conjunto dos vértices em C = (S, T), tal que *source* ∈ S e *sink* ∈ T. O somatório do fluxo das arestas cortadas pelo corte mínimo no sentido S->T é igual ao fluxo máximo, mas, como já foi mencionado, para o mesmo grafo podem existir mais que um corte mínimo. Queremos encontrar o corte que maximiza o número de vértices classificados como primeiro plano, pelo que a nossa abordagem consistiu em inicializa-los todos como sendo de primeiro plano e, após aplicar o algoritmo de *Edmonds-Karp*, correr uma última *BFS* na rede residual começando na *source*. Classificando todos os vértices atingíveis a partir da *source* como cenário encontra-se assim o corte mínimo pretendido.

Convém também referir duas optimizações que foram implementadas de forma a tornar o programa mais rápido. Ao ler o input, imediatamente encontramos todos os caminhos de aumento do tipo *source*->pixel->*sink* (comprimento 2). Assim, pelo menos uma das arestas de cada um desses caminhos ficará logo saturada (com capacidade residual igual a 0). Isto é feito igualando o fluxo de ambas as arestas à menor capacidade das duas, quando o programa está a ler a segunda matriz. A segunda optimização está associada à anterior, pois ao criar as arestas *source*->pixel criámos também a aresta inversa pixel->*source* que, apesar de não serem necessárias no âmbito do problema como mencionado, permitem-nos aceder em O(1) a cada uma das arestas *source*->pixel (com o ponteiro para a inversa) ao invés de O(k), a complexidade para aceder ao termo k de uma lista ligada. Após encontrar cada um destes caminhos de aumento, a memória usada para as arestas *source*->pixel é libertada, otimizando assim também a memória utilizada pelo programa. Encontrar todos estes caminhos de aumento logo de início faz com que o algoritmo de *Edmonds-Karp* realize menos iterações, otimizando assim o tempo de execução do programa.

Análise Teórica

Antes de mais, é de notar que E=O(V), visto que cada pixel tem no máximo 6 arestas. Isto vai ser importante para a o cálculo das complexidades temporal e espacial. Teoricamente e analisando o código, o algoritmo tem uma complexidade polinomial (terceiro grau) em termos temporais e uma complexidade linear em termos espaciais.

- Complexidade temporal: $O(V^3)$

Ao ler o input, para cada matriz usa-se um duplo ciclo *for* para criar as várias arestas, tendo uma complexidade de O(E) para criar a lista de adjacências. O algoritmo de *Edmonds-Karp* tem uma complexidade de $O(VE^2)$ para grafos representados como listas de adjacência, sendo que o algoritmo *BFS* que utiliza tem uma complexidade de O(V+E), tanto para calcular os caminhos de aumento como para determinar o corte mínimo. Finalmente tem-se um ciclo para imprimir a classificação dos pixéis da imagem, que percorre todos os vértices e portanto tem complexidade O(V). Ou seja, a complexidade temporal do programa é $O(V^3)$.

$$O(E+VE^2+(V+E)+V) = O(VE^2) => O(V^3).$$

O algoritmo de *Edmonds-Karp* é um caso específico do algoritmo de *Ford-Fulkerson* para problemas de fluxo máximo/corte mínimo, na medida em que utiliza a *BFS* para encontrar os

caminhos de aumento. *Ford-Fulkerson* não tem um critério específico para encontrar os caminhos de aumento, tendo por isso uma complexidade de $O(Ef^*)$, sendo f^* o fluxo máximo do grafo. Assim, é preferível utilizar *Edmonds-Karp*, que garante que se encontram os caminhos de aumento mais curtos primeiro, e sendo por isso temporalmente mais eficiente que *Ford-Fulkerson*. Outra possibilidade seria implementar o algoritmo de *Dinic* que tem uma complexidade de $O(V^2E)$ e que é bastante semelhante ao algoritmo de *Edmonds-Karp*, na medida em que também usa *BFS*. No entanto, a *BFS* não é usada para iterativamente calcular novos caminhos de aumento mas sim para determinar se é possível enviar mais fluxo no grafo e construir um grafo de níveis em que esses níveis simbolizam a distância até à *source*. Tendo esse grafo, são enviados vários fluxos, razão pela qual este algoritmo é mais rápido que o *Edmonds-Karp*, que só envia um fluxo de cada vez.

Podíamos também ter implementado algoritmos de pré-fluxo, como *Push-Relabel* ou o seu caso específico *Relabel-to-Front*, que têm complexidades de $O(V^2E)$ e $O(V^3)$, respectivamente. No entanto, estes algoritmos são mais exigentes em termos de memória, razão pela qual não foi ponderada a sua implementação.

- Complexidade espacial: O(V)

A estrutura de dados mais complexa que utilizamos é a lista de adjacências, que tem complexidade espacial O(V+E). Para além desta, foi usada também uma fila de espera (*Queue*) para a implementação das funções de *BFS*. Esta *Queue* tem uma complexidade espacial O(V), pois utiliza uma lista com um tamanho máximo igual ao número de vértices do grafo. O algoritmo de *Edmonds-Karp* usa também duas listas auxiliares de comprimento igual ao número de vértices de grafo, ou seja, complexidade O(V). Logo, a complexidade espacial do programa é O(V)

$$O((V+E)+V+V+V) = O(V+E) \Rightarrow O(V).$$

Análise Experimental

Para a análise experimental do programa foram-nos disponibilizados alguns testes.

O Gráfico 1 foi desenhado com base na média de tempo decorrido ao executar cada teste 10 vezes, em função de V. Estes tempos foram obtidos com a função *time* do *Linux*.

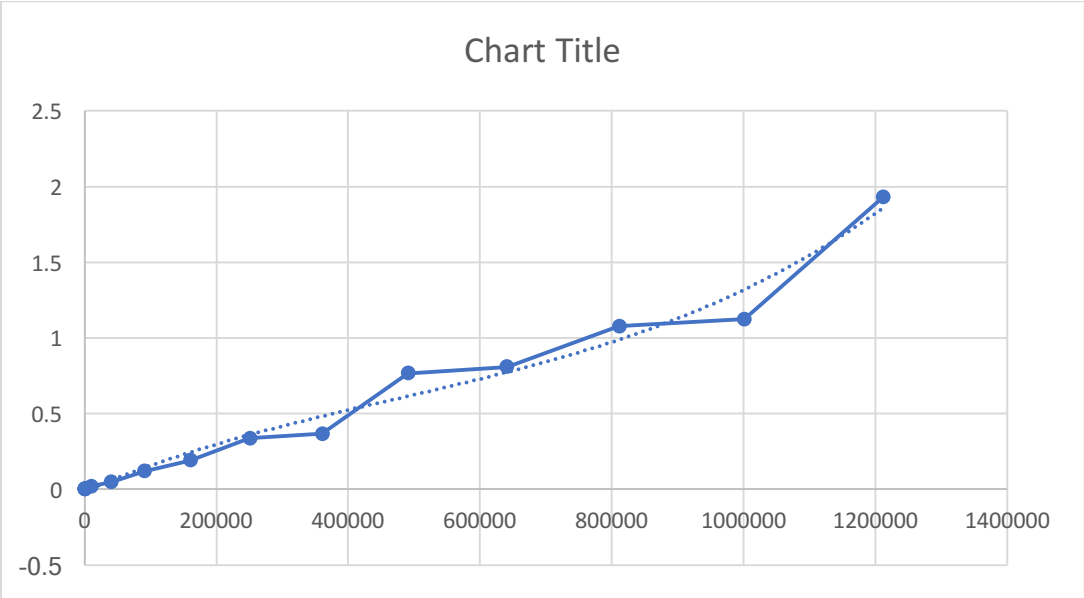

Gráfico 1 – Tempo de execução em função de V

O Gráfico 2 foi desenhado com o máximo de memória alocada em função de V. De mencionar que esse máximo foi obtido através da ferramenta *Valgrind*, para cada teste público.

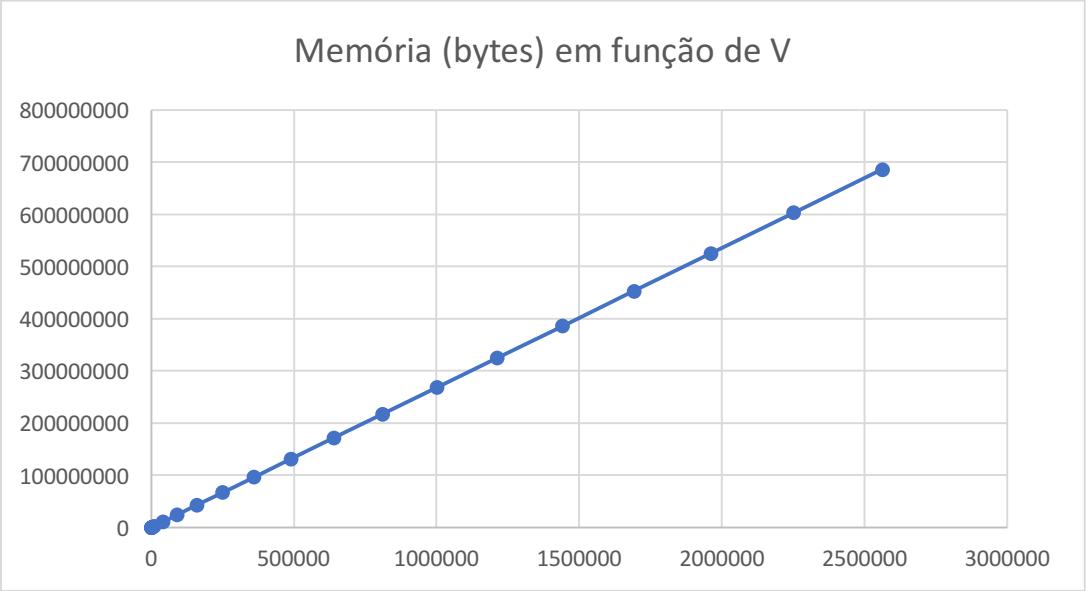

Gráfico 2 – Memória alocada (máximo) em função de V

Analisando o gráfico de tempo de execução, observa-se uma relação polinomial com V, com parte de uma concavidade que se pode considerar semelhante à de uma função de terceiro grau. Esta conclusão é retirada da comparação com a linha a tracejado no gráfico, que corresponde ao gráfico da função $f(x)=x^3$. Existe alguma irregularidade no gráfico mas acreditamos que tal se deve à influências de outros processos que pudessem estar a correr paralelamente na máquina. Assim, é $O(V^3)$ como era esperado através da análise teórica.

Analisando o gráfico da memória alocada, observa-se claramente uma linear com V, como era esperado através da análise teórica, O(V).

Referências

Bibliográficas:

- 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

Para além da obra anterior, foram consultados os seguintes websites:

- https://en.wikipedia.org/wiki/Edmonds%E2%80%93Karp_algorithm
- https://brilliant.org/wiki/edmonds-karp-algorithm/
- http://www.programmingworldtech.com/2014/03/bfs-and-dfs-on-graph-represented-using.html
- http://www.cs.upc.edu/~mjserna/docencia/grauA/T15/MaxFlow.pdf
- https://www.sanfoundry.com/c-program-implement-adjacency-list/