ANÁLISE E SÍNTESE DE ALGORITMOS

2º PROJETO – PIXEL SEGMENTATION

INTRODUÇÃO:

Para o segundo projeto de Análise e Síntese de Algoritmos do ano 2017/2018 foi proposto aos alunos apresentar uma solução para fazer segmentação de pixéis numa dada imagem, ou seja, classificá-los como sendo de primeiro plano **(P)** ou cenário **(C)**, não omitindo nenhum e de forma a que cada um destes tenha uma e uma só classificação. Foi ainda pedido para usar um modelo simples, que procura minimizar o peso total, ou seja, encontrar uma solução que descreva, num conjunto de pixéis, a segmentação que maximiza o número de pixéis classificados em primeiro plano. O input consiste em várias matrizes que representam pesos de primeiro plano, pesos de cenário, pesos das relações de vizinhança horizontais e pesos das relações de vizinhança verticais, respetivamente.

DESCRIÇÃO DA SOLUÇÃO:

A solução apresentada pelo grupo 44 encontra-se num único ficheiro, main.cpp, em C++. Esta linguagem foi escolhida pela facilidade em manusear conteúdo dentro de bibliotecas e estruturas de dados.
Foi aplicado o algoritmo de Edmonds Karp pela capacidade de calcular o fluxo máximo numa rede de fluxos, usando uma BFS que auxilia na determinação do corte mínimo e classificação de pixéis. Também se demonstrou mais rápido que outros algoritmos, tal como, Relabel-To-Front (tendo em conta que estamos a trabalhar com grafos esparsos e este adapta-se melhor a grafos densos. É ainda mais difícil de implementar), e Ford-Fulkerson (pois, poderia não encontrar caminhos de aumento tão depressa, devido a fazer uso de uma DFS).
Para melhor conhecimento do algoritmo: https://pt.wikipedia.org/wiki/Algoritmo_de_Edmonds-Karp.

Neste ficheiro existem várias estruturas (**edge, pixel, matrix**) que simulam o grafo. Uma matriz é composta por um vetor de pixéis que representam os vértices do grafo e um vetor de ligações entre estes, ou seja, edges. Cada píxel é composto por um vetor de índices que correspondem à posição de uma dada ligação (edge) no vetor de ligações referido, que se encontra na matriz. É nestas estruturas onde se vai adicionando o input para a segmentação requerida. Cada píxel guarda os pesos de primeiro plano e de cenário. As ligações, por sua vez, guardam os pesos das relações de vizinhança entre pixéis. Estas ligações são partilhadas entre dois pixéis, podendo ser percorridas nos dois sentidos.
Foram implementadas funções auxiliares para otimização do algoritmo tais como **saturateDirectEdges** e **printMatrix**. A primeiro consiste em saturar os arcos diretos, ou seja, se tivermos um píxel em que é possível sair da source e chegar diretamente ao target a partir dele, temos um caminho praticamente direto logo é possível esgotar a aresta com menos peso entre esses três vizinhos. A segunda trata de imprimir os pixéis já classificados de P ou C no ecrã. Para este efeito são criados dois pixéis adicionais, **source** e **target** que estão

ligados a todos os outros pixéis da matriz. Estas ligações à source e ao target têm o peso correspondente ao peso de primeiro plano e ao de cenário, respetivamente. Como referido acima, cada píxel guarda os pesos de primeiro plano e de cenário, ou seja, os valores guardados vão corresponder ao peso das ligações para a source e para o target.
Quanto às funções principais, **BFS** e **edmondsKarp**, tem-se que a primeira trata de calcular um caminho possível dentro dos pixéis e ligações destes que ainda não foram esgotadas, isto é, se ainda existe uma relação pesada entre pixéis vizinhos, e se suceder retorna o valor do fluxo desse mesmo. A segunda fica encarregue de chamar sucessivamente a BFS para o cálculo de todos os caminhos possíveis, ou seja, todas as possibilidades que se podem obter da source percorrendo vários pixéis e suas ligações não esgotadas até chegar ao target. Sempre que a BFS retorna o valor mínimo do fluxo do caminho que encontrou, a variável **maxFlow** (impressa no fim, como resultado de output) que apresenta o fluxo máximo é somada ao valor de retorno da BFS e são ainda atualizadas as capacidades de cada ligação pelo valor referido. No fim de todas as iterações da BFS, caso um píxel possa ser acedido, ainda, é classificado como Cenário (C), os restantes são classificados como Primeiro Plano (P).
Se no fim destes processos, os pixéis forem todos de Cenário ou todos de Primeiro Plano, então não existe corte mínimo, pelo que o valor retornado é apenas a soma de todos os píxeis, isto é, a variável maxFlow não é adicionada aos pesos. Para concretizar este último procedimento usaram-se flags **counterBlack** e **counterWhite**.

ANÁLISE TEÓRICA:
Sabendo que a inicialização é dada por $O(V)$ (V representa m*n+2, sendo os últimos 2 o source e o target), que a complexidade das funções auxiliares, saturateDirectEdges e printMatrix, é $O(V)$ e ainda que a chamada contínua da função BFS custa $O(V+E)$ (E representa (m-1)*n + (n-1)*m+2V, isto é, as ligações de vizinhança entre pixéis. 2V é o número de ligações de cada píxel ao source e ao target) temos que a complexidade do programa descrito acima é dada pela expressão: $O(V^3)$. Em relação ao espaço ocupado temos que $O(V+E)$.

Descrição das várias funções implementadas para a realização da solução:

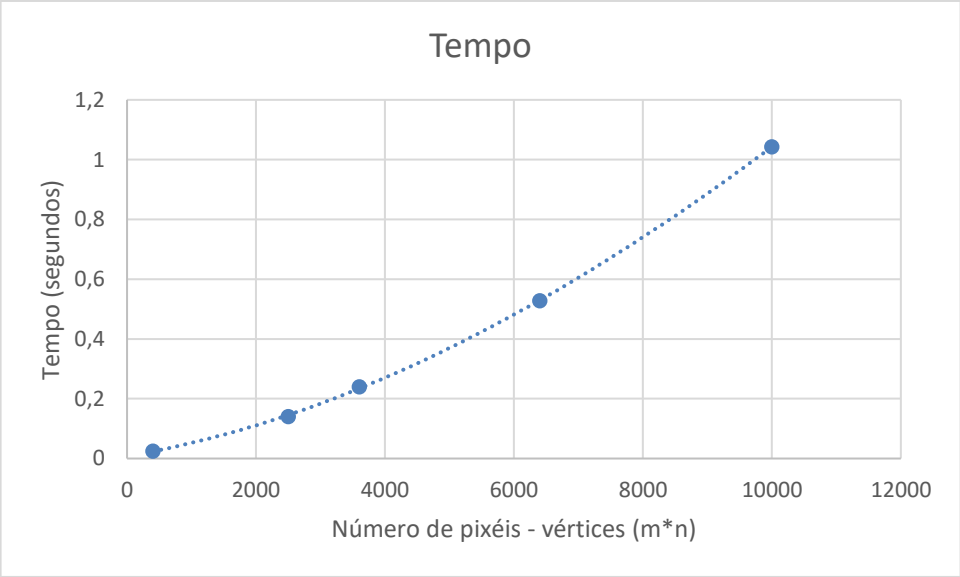
- **void printMatrix(struct matrix *matrix)**: Imprime a matriz matrix de acordo com a classificação dos pixéis.
- **void saturateDirectEdges(struct matrix* matrix)**: função que esgota os arcos dos caminhos mais curtos possíveis, isto é, percursos que começam na source vão para um dado píxel e que depois vão diretamente para o target. Trata-se de uma otimização em termos de tempo, pois a BFS não precisa de se preocupar com este processo.
- **int BFS(struct matrix *matrix, int startNode)**: Encontra possíveis caminhos dentro da matrix, indo da source, passando por vários vértices (pixéis) até ao target, a partir de três vetores, **flows** (responsável por guardar a informação dos índices das ligações usadas por cada píxel para percorrer o caminho), **visitedPixels** (informa se um dado píxel já foi visitado) e **currentPathCapacity** (tem o valor atual do fluxo de um dos caminhos até agora encontrados) que são atualizados. Quando encontra o caminho mais curto retorna o fluxo desse para a seguinte função (edmondsKarp).
- **int edmondsKarp(struct matrix *matrix, int startNode)**: Chama continuamente a BFS e após receber um fluxo da mesma, atualiza os valores das ligações, podendo

estas ficar esgotadas ou não. Atualiza, adicionando, ainda, o valor do fluxo máximo (variável maxFlow) pelo retorno da BFS referido. Os caminhos encontrados são percorridos inversamente com auxílio da informação guardada pelos três vetores referidos acima. Este ciclo termina quando a BFS retorna 0 de fluxo e após esta iteração é feita a classificação dos pixéis e a soma dos pesos.

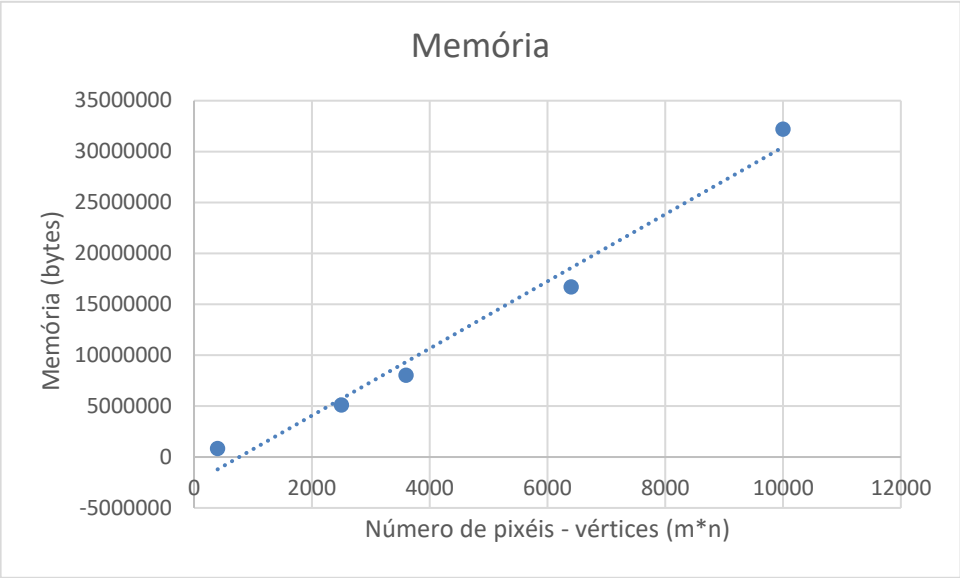
ANÁLISE EXPERIMENTAL:

De maneira a fazer uma análise experimental **o ficheiro foi submetido a vários testes aleatórios criados pelo gerador**, fornecido pelos docentes da cadeira. A partir do uso das ferramentas **Valgrind** e **Time**, foi possível obter a informação apresentada nos gráficos seguintes (Gráfico1 e Gráfico2).

Conclui-se que, em termos de **tempo**, existe um **crescimento potencial cúbico**, e que em termos de **memória** alocada existe um **crescimento linear**.



(Gráfico 1 – Avaliação experimental de tempo)



(Gráfico 2 – Avaliação experimental de memória)

REFERÊNCIAS:

Os sites usados para consulta auxiliar no processo de implementação do projeto foram os seguintes:

- https://www.sanfoundry.com/cpp-program-implement-edmonds-karp-algorithm/
- https://en.wikipedia.org/wiki/Edmonds%E2%80%93Karp_algorithm
- https://en.wikipedia.org/wiki/Max-flow_min-cut_theorem#Image_segmentation_problem