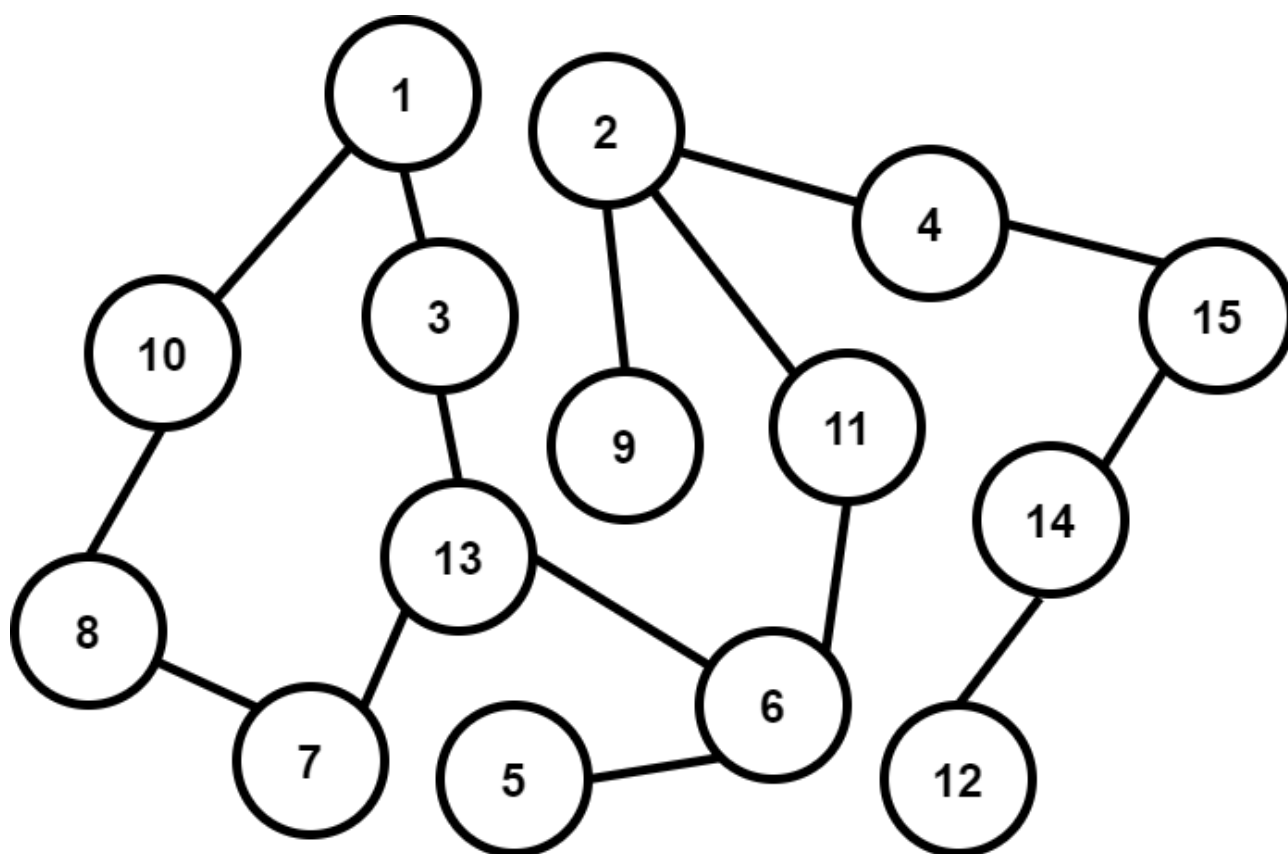


Projeto AirRoutes Grupo 32



1 Descrição do Problema

Este projeto, criado no âmbito da Unidade Curricular de Algoritmos e Estruturas de Dados, aborda o problema de encontrar um conjunto mínimo de rotas que garantam a existência de um caminho entre cada par de aeroportos, sem qualquer redundância. O Programa que este projeto visa produzir deve não só produzir uma rede mínima de rotas, mas sim a rede que garante todos os destinos já existentes, com o menor custo. Uma ferramenta como esta pode ser necessária, em termos práticos, quando uma companhia aérea quer ou precisa de otimizar de custos. Numa situação desta natureza, poderá ser necessário reduzir o seu conjunto de rotas à rede que garante que todos os aeroportos nela podem ser utilizados, com o menor custo.

O Projeto que este relatório descreve foi desenvolvido na linguagem C, com recurso a Makefile e e, para o seu correto funcionamento deve receber as informações de todas as rotas existentes através de um ficheiro de texto. Este deve conter uma ou mais redes de aeroportos, cada uma devidamente identificada com um cabeçalho contendo o número de aeroportos, o número total de rotas existentes e a variante que se pretende obter. Terminada a execução, o programa produz um segundo ficheiro de texto que contém o conjunto mínimo de menor custo para cada rede fornecida, também identificados pelo cabeçalho fornecido, com a adição de algumas informações úteis, como o número de rotas mantidas e o custo total da rede

2 Abordagem do Problema

O Estudo de grafos assenta em alguns conceitos chaves, como o número de vértices (V), o número de arestas que ligam esses vértices (E), se as arestas são ponderadas (Custo) ou direcionadas, e a sua densidade (esparso ou denso). Para resolver este problema, foi necessário criar uma estrutura capaz de guardar uma representação do grafo formado pelo conjunto de rotas fornecido. A escolha desta estrutura requer uma análise cuidada ao tipo de grafo (denso ou esparso) e aos algoritmos que se pretende utilizar. Uma má escolha poderia levar a tempos de execução muito maiores que os pretendidos e maior utilização de memória, devido ao aumento da complexidade.

3 Arquitetura do Projeto

A estrutura geral do programa é apresentada na Figura 3. Após verificar e inicializar os ficheiros necessários, procede-se à leitura dos Argumentos. Dependendo da leitura dos argumentos, é decidida a melhor forma de representação do grafo para o problema em questão. As formas de representação do grafo são definidas com maior detalhe na secção??.

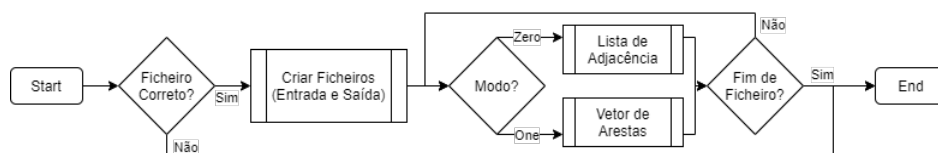


Figura 1: Fluxograma Representativo da execução da função main

4 Estruturas e Tipos de Dados

4.1 Argumentos do Problema

A estrutura do tipo `PBArg` é usada para guardar os argumentos do problema. É declarada no ficheiro *Graph.h*. Esta Estrutura é composta por:

- `v` : Integer que guarda o número de vértices no grafo;
- `e` : Integer que guarda o número de arestas no grafo;
- `vi` : Integer que guarda o 1º vértice da aresta a eliminar, nos modos aplicáveis;
- `vj` : Integer que guarda o 2º vértice da aresta a eliminar, nos modos aplicáveis;
- `var` : String que guarda a variante do problema pretendida;
- `err` : Boolean que é ativada quando ocorrem erros durante a execução;

4.2 Aresta

A estrutura do tipo `Aresta` é usada para guardar uma aresta, quer durante a sua leitura em ambos os modos, quer na sua manipulação no modo 1, onde está inserida na estrutura `graph`, para formar o vetor de arestas. Apesar de relativamente simples, esta estrutura é a base sobre a qual a maioria das funções deste programa operam. Por esse motivo foi propositamente tornada mais simples e eficiente. Esta Estrutura é composta por:

- `vi` : Integer que guarda o 1º Vértice da Aresta
- `vj` : Integer que guarda o 2º Vértice da Aresta
- `cost` : double que guarda o custo da aresta

4.3 Lista de Adjacências

A estrutura do tipo `Lista de Adjacências` é utilizada pelas funções do modo 0. Esta é a base para a representação do grafo sob a forma de vetor de listas de adjacências. As razões pelas quais esta representação do grafo foi escolhida para o modo 0 será discutida n??. Esta Estrutura é composta por:

- `v` : Integer que guarda o vértice base da lista, a partir do qual se encontram os adjacentes;
- `cost` : Double que guarda o custo da aresta entra o vértice e o 1º vértice adjacente;
- `next` : Apontador para list que guarda o próximo elemento da lista;

4.4 Grafo sob a forma de Lista de Adjacência

A estrutura do tipo `graph0` é a estrutura mãe para a representação do grafo na forma de listas de adjacências. Decidiu-se anexar à representação propriamente dita um apontador para `PBArg` para facilitar grande parte dos parâmetros de entrada das funções. Assim, os 4.1 são enviados em conjunto com a representação do grafo. Esta Estrutura é composta por:

- `Arg` : Estrutura do tipo `PBArg` que guarda os argumentos referentes ao grafo representado no outro membro desta estrutura;
- `data` : Vetor de estruturas do tipo `list` que guarda a lista de adjacências de todos os vértices do grafo;

4.5 Grafo sob a forma de Vetor de Arestas

A estrutura do tipo `graph` é a estrutura mãe da representação do grafo na forma de vetor de arestas. Como a estrutura `graph0`, a estrutura `graph` inclui um apontador para uma estrutura do tipo `PBArg` para facilitar os parâmetros de entrada de grande parte das funções que operam sobre o grafo. Esta Estrutura é composta por:

- `Arg` : Estrutura do tipo `PBArg` que guarda os argumentos referentes ao grafo apontado pelo vetor de arestas;
- `data` : Vetor de estruturas do tipo `edge` que guarda todas as arestas lidas do ficheiro de entrada para o grafo em estudo;

4.6 Formas de Representar o Grafo

Para a resolução do Problema anteriormente discutido, decidiu-se representar o grafo sob a forma de Vetor de Arestas. Nesta decisão, foram tidos em conta os seguintes fatores:

- O Ficheiro de Entrada é sempre um vetor de arestas;
- A leitura do ficheiro de entrada tem uma complexidade inerente mínima de E (Precorrer o grafo);
- O Algoritmo de *Kruskal* recebe dados sob a forma de vetor de arestas. A escolha do Algoritmo de *Kruskal* será discutida na secção 5

Qualquer outra forma de ler os dados do ficheiro de entrada teria complexidade igual ou superior, devido à necessidade de 'tradução' dos dados. Na utilização do Algoritmo de Krukak seria necessário 'traduzir' de novo os dados para a forma de vetor de arestas pelo que estaríamos a 'gastar' tempo em conversões desnecessárias.

5 Algoritmos

5.1 Algoritmos para encontrar a Árvore Mínima de Suporte

Para encontrar a árvore mínima de suporte de um grafo, foram estudados dois algoritmos nas aulas teóricas de AED.

- Algoritmo de *Prim*;
- Algoritmo de *Kruskal*;

O Algoritmo de *Prim* assenta numa representação do grafo sob a forma de matriz de adjacências que consome mais memória que a representação em vetor de arestas para grafos, exceto na remota hipótese de cada vértice estar ligado a todos os outros. A sua otimização só é possível recorrendo a uma fila prioritária cuja prioridade é definida pelo custo das arestas.

O Algoritmo de *Kruskal* assenta numa representação do grafo sob a forma de Vetor de Arestas. A sua otimização depende da implementação do algoritmo de Conectividade (Compress Weighted Quick Union) e do algoritmo de ordenação (Quick Sort).

5.1.1 Problema dos Sub-Grafos não conectados

Com o algoritmo de *Prim*, o programa cria uma árvore e vai adicionando os vértices cuja aresta tem menor custo (Na implementação com fila prioritária, os vértices cuja aresta for mais prioritária). Isto cria o problema de, quando se analisam grafos constituídos por vários sub-grafos não conectados, o programa não conseguir desenvolver o backbone com apenas uma leitura de todas as arestas. Torna-se necessário correr o algoritmo para todos os vértices que não pertençam a uma árvore mínima de suporte. O aumento da complexidade nesta situação é discutido na secção 5.1.2.

Por outro lado, com o algoritmo de *Kruskal*, o programa cria várias árvores que se vão ligando com a análise das arestas do grafo. Ao criar várias árvores mínimas de suporte em simultâneo, resolvemos este problema porque todas as arestas são consideradas para a criação de uma ou mais árvores.

5.1.2 Análise da Complexidade dos Algoritmos para encontrar a MST

O limiar teórico para a complexidade deste problema é definido pelo Algoritmo de *Kruskal*, cuja complexidade é $O(E * \log E)$. Devido ao problema discutido na secção 5.1.1, o Algoritmo de *Prim* tem complexidade difícil de determinar, e difícil também de aproximar do limiar teórico da complexidade estabelecido pelo algoritmo de *Kruskal*. Por isso, complexidade do algoritmo de *Prim* é, no melhor caso, $O(E * \log V)$.

5.1.3 Prós e Contras

Partindo da teoria, o Algoritmo de *Kruskal* é o caminho a seguir quando se quer analisar grafos esparsos (como os grafos do Projeto AirRoutes) devido às implementações mais eficientes, tanto em tempo como em memória. A representação em vetor de arestas não só reduz o tempo dispendido na leitura e escrita como simplifica o código fonte. Este é ainda mais simplificado com a não implementação da fila prioritária do Algoritmo de *Prim*.

5.2 Algoritmos para o Problema da Conectividade

O Problema da Conectividade, necessário à implementação do algoritmo de *Kruskal*, foi largamente discutido nas aulas teóricas de AED. Nesta discussão foram abordados quatro algoritmos que podem ser implementados:

- Quick Find
- Quick Union
- Weighted Quick Union
- Compressed Weighted Quick Union

Dessa discussão, concluiu-se que o algoritmo mais otimizado para resolver o problema da conectividade é a Compressed Weighted Quick Union, devido à compressão de caminho. O custo de execução deste algoritmo está apenas a um fator constante do custo (inevitável) de leitura de dados. Realisticamente, esse custo de execução é negligenciável quando comparado com o tempo de execução de todo o programa.

Operação	Complexidade
Ler o Grafo	$O(E)$
<i>Kruskal</i>	$E * \log(E)$
Ordenação	$E * \log(E)$
CWQU	$E * \log(E)$
Repor a Conectividade Inicializar	$O((E - X) * \log(E - X))$

5.3 Análise da Complexidade Geral do Programa

A complexidade do Programa depende da variante selecionada.No entanto há operações que são realizadas em várias as variantes:

Podem ser selecionadas as seguintes variantes:

- A1,