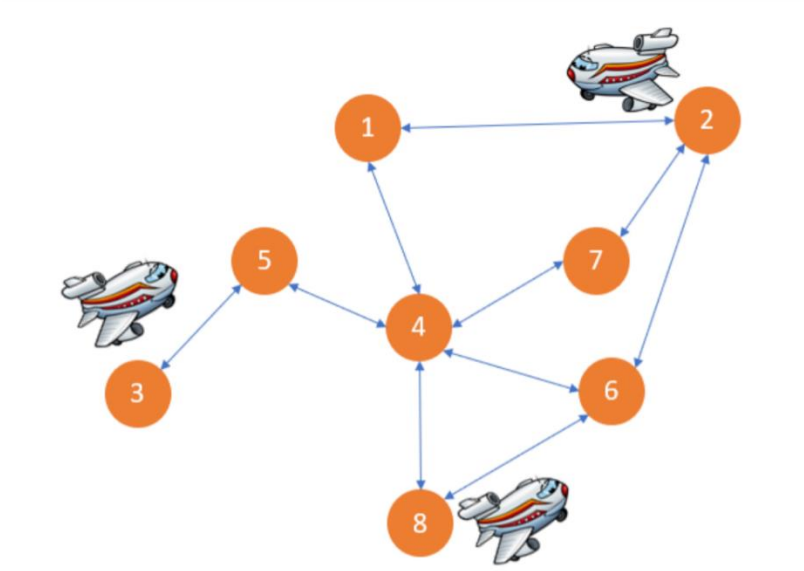
TÉCNICO LISBOA

AirRoutes:

Relatório de Projeto



Curso: Mestrado Integrado em Engenharia Electrotécnica e de Computadores

UC: Algoritmos e Estruturas de Dados

1º Semestre 2020/2021

Autores: (Grupo 11)

- Afonso Alemão 96135 afonso.alemao@tecnico.ulisboa.pt

- Rui Daniel 96317 ruipcdaniel@tecnico.ulisboa.pt

Data: 10/12/2020

TÉCNICO LISBOA

Índice

Problema proposto

No âmbito da UC de Algoritmos e Estruturas de Dados, foi-nos proposto como projeto um problema denominado *AirRoutes*, a desenvolver em linguagem C, tendo como objetivo ler e armazenar grafos, e criar uma série de modos que operem sobre estes. Cada grafo através dos seus vértices representa aeroportos, e das suas arestas as rotas que os ligam. Objetivo de solução para cada um dos modos:

A1 – O conjunto de rotas de custo total menor que assegure os mesmos serviços que o conjunto original – "backbone" do grafo original;

B1 – O "backbone" variacional em relação à solução obtida em A1 por exclusão da aresta entre os vértices v_i e v_j;

C1 – O "backbone" não variacional por exclusão da aresta entre os vértices v_i e v_j;

D1 – O "backbone" variacional por exclusão do vértice v_i;

E1 – O "backbone" e respetivo "backup".

O nosso programa deve receber como argumento de entrada um ficheiro de extensão ".routes", e produzir a solução para um ficheiro ".bbones".

Abordagem ao problema

Para resolver os problemas propostos, foi necessário criar estruturas do tipo Grafo e Aresta, que foram utilizadas para armazenar cada problema recebido. Tivemos de criar uma interface e implementar operações sobre estas, de modo a resolver os problemas que tínhamos como desafio.

A nossa abordagem geral consiste em resolver um dos problemas no ficheiro de entrada de cada vez, armazenando apenas 1 grafo em memória. Após detetado o modo e armazenado o grafo, apenas são guardadas em memória as estruturas de dados utilizadas no modo em processamento.

A abordagem a cada um dos modos encontra-se especificada mais à frente.

Arquitetura do projeto

O nosso projeto *AirRoutes* encontra-se dividido em vários ficheiros:

(1) **main.c**: programa principal;
(2) **airroutes.c airroutes.h**: processamento de ficheiros, codificações, gestão de memória e funções auxiliares;
(3) **grafo.c grafo.h**: funções que interagem com o grafo;
(4) **ordenacao.c ordenacao.h**: funções que executam algoritmos de ordenação;
(5) **conetividade.c conetividade.h**: funções que implementam, com algoritmos de conetividade, as operações procura e união;
(6) **Pilha.c Pilha.h**: funções que operam sobre a estrutura pilha;
(7) **heap.c heap.h**: funções que operam sobre filas prioritárias representadas por um acervo;
(8) **defs.h**: definição de tipos abstratos: Item, bool e macros;

e também contém uma Makefile.

A representação gráfica das dependências entre ficheiros é dada pela seguinte árvore:

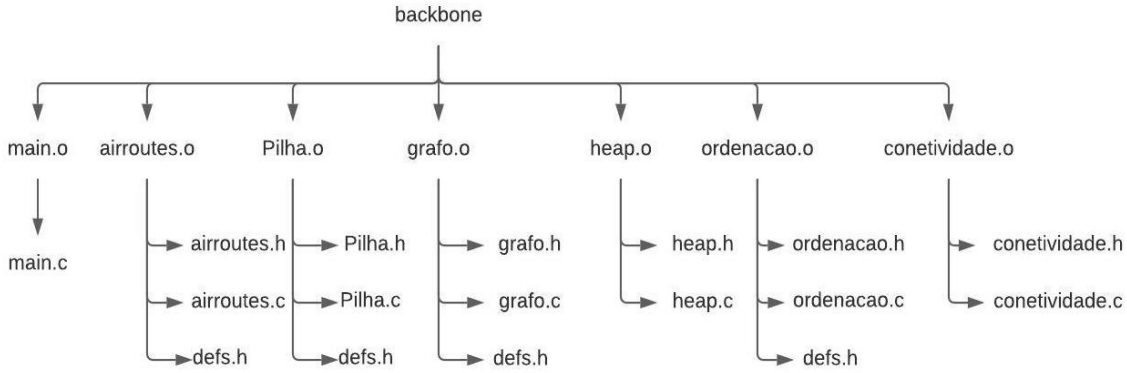


Figura 1: dependências do programa

Estruturas de Dados Utilizadas

Antes de detalharmos como resolvemos cada um dos problemas no projeto, iremos especificar as estruturas de dados utilizadas ao longo deste.

1-Bool

Tipo de dados que representa uma variável binária. Pode ter um de dois possíveis valores: 0 (falso) ou 1 (verdadeiro). Definição em "defs.h".

2-Grafo, Aresta e Link: Listas de adjacências

Definição de membros de cada uma destas 3 estruturas e funções que executam processamento sobre estas em "grafo.c". Protótipos em "grafo.h".

2.1-Aresta

Estrutura usada na representação das arestas no grafo. Composta pelos dois vértices adjacentes e pelo custo associado à aresta formada por estes. Representa uma rota.

2.2-Grafo

Estrutura utilizada para representação de um grafo. Contém como parâmetros o número de vértices e de arestas deste, e um ponteiro para uma tabela de listas de adjacências de cada um dos vértices no grafo.

2.3-Link

Também utilizamos esta estrutura tal como foi introduzida nas aulas teóricas. Representa um elemento "nodo", que se encontra ligado a outro "nodo" ou a NULL, através de um ponteiro "next". Desta forma implementamos listas simplesmente ligadas para representar as listas de adjacências de cada um dos vértices do grafo. Para além deste ponteiro de ligação, cada "nodo" também possui um indicador do vértice adjacente e do custo dessa ligação.

3-Heap

Estrutura utilizada para implementar operações num acervo (fila prioritária). É constituída pelo número de elementos na fila, pelo tamanho máximo que esta poderá atingir e por um vetor de inteiros (heapdata) que contém os vértices cumprindo a condição de acervo (sendo o vértice associado à aresta de menor custo o mais prioritário) .

A sua utilização tem como objetivo facilitar e tornar mais eficiente o processo de procura das arestas de menor custo, na construção da MST (de grande utilidade no modo A1: Utilizando um Algoritmo de Prim otimizado).

Definição, protótipos e implementações em "heap.h" e "heap.c".

4-Pilha

Estrutura utilizada para implementar operações numa Pilha, do tipo FIFO (First In, First Out), implementada através de uma lista simplesmente ligada com um ponteiro para o topo da pilha.

Foi utilizada para realizar uma implementação do QuickSort otimizada (utilização de uma pilha explícita para minimizar a utilização de memória), em que colocamos os índices das próximas sub-tabelas a ordenar na pilha: tabelas mais pequenas são inseridas por cima e tratadas em primeiro lugar, para que a pilha aumente o mínimo possível.

Definição, protótipos e implementações em "Pilha.h" e "Pilha.c".

5-Item

Estrutura do tipo abstrato, que nos permite generalizar algumas operações no nosso programa e permitir a fácil reutilização de código em outro contexto. Utilizado, por exemplo, em algoritmos de ordenação e em funções de comparação. Definição em "defs.h".

6-Vetores de Arestas

Tabela que contém todas as arestas (estrutura explicada na página anterior) do grafo. É extremamente útil na procura da melhor rota de substituição existente no grafo. Nos modos em que tivemos de calcular "backbone(s)", também estes foram representados por um vetor de arestas.

Main

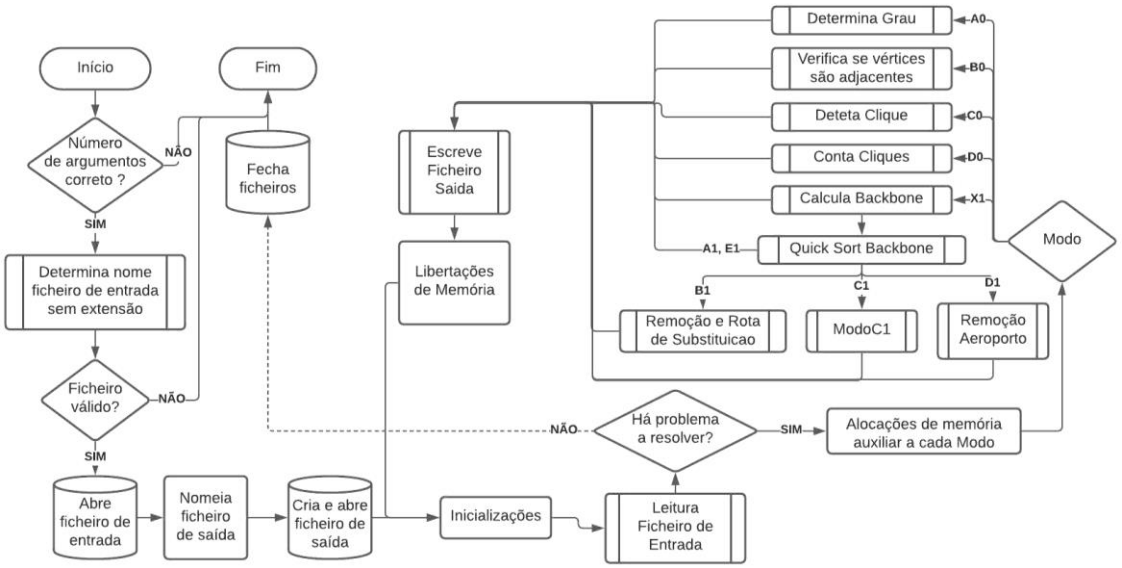


Figura 2: Fluxograma da função "main"

Através da função main, encaminhamos o fluxo do programa de forma a que, os modos pedidos no ficheiro de entrada sejam executados, sobre os respetivos grafos.

É validado o ficheiro de entrada e criado o ficheiro de saída com a extensão ".bbones".

Após isso entramos no nosso programa, que consiste num ciclo que faz 2 operações básicas: a leitura de um problema para um grafo no ficheiro de entrada, e a produção da sua solução. Em cada um destes ciclos, há sempre as alocações de memória necessárias e suficientes à produção de uma solução para o modo em questão (especificadas na secção "Estruturas Auxiliares a cada Modo"), e após resolvido o problema, há a libertação dessa memória.

Só lemos e processamos um grafo de cada vez, de modo a minimizar o pico de memória utilizado.

Quando já tiverem sido resolvidos todos os problemas presentes no ficheiro de entrada, damos o nosso programa como terminado.

Leitura e Armazenamento do Grafo

Através da função "LeituraFicheiroEntrada", o ficheiro de entrada é lido e armazenado no grafo representado por listas de adjacências. Caso o ficheiro já esteja totalmente lido retorna essa mesma informação (NULL): o que acontece quando uma das leituras falha (não representado no fluxograma abaixo por uma questão de simplicidade). Caso contrário, retorna um ponteiro para o grafo.

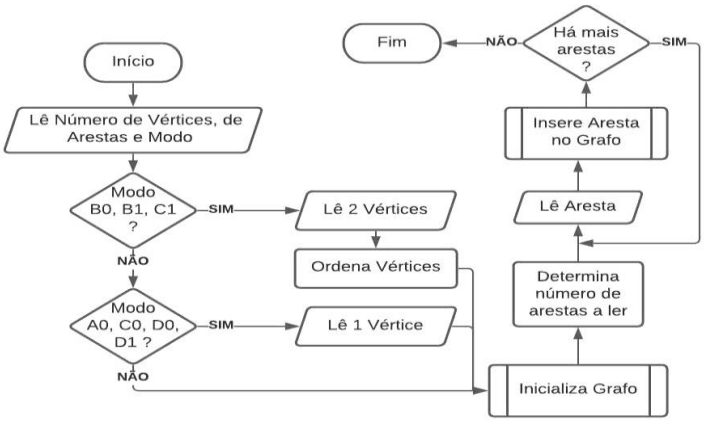


Figura 3: Fluxograma da função "LeituraFicheiroEntrada"

Estruturas Auxiliares a cada Modo

Aqui iremos enunciar as estruturas de dados alocadas na função main para cada um dos modos:

Aeroporto: Estrutura do tipo grafo, necessária em todos os modos, que guarda o grafo lido no ficheiro de entrada, numa tabela de listas de adjacências.

Flags: Estrutura do tipo vetor booleano, necessária aos modos C0 e D0.

Vetor de Arestas: Tabela de elementos do tipo Aresta, utilizada nos modos A1, B1, C1, D1, E1, para armazenar todas as arestas do grafo.

Backbone: Tabela de elementos do tipo Aresta que guarda todas as arestas de um "backbone". Utilizada nos modos A1, B1, C1, D1, E1.

Rota de Substituição: Tabela de elementos do tipo Aresta que guarda todas as arestas que representam rotas de substituição. Utilizada nos modos B1, C1, D1, E1.

Backbone 2: Tabela de elementos do tipo Aresta que guarda todas as arestas do "backbone" de substituição. Utilizada no modo C1.

Heap: Estrutura que representa o acervo, utilizada nos modos A1, B1, C1, D1 e E1. Permite operar sobre filas prioritárias, acedendo ao vértice associado à aresta de menor custo (maior prioridade) mais rapidamente.

Algoritmos Projeto Intermédio

A0: Para determinar o grau de um aeroporto, contamos os elementos da sua lista de adjacências.

B0: Para verificar se 2 vértices são adjacentes, percorremos a lista de adjacências do primeiro em busca do segundo.

C0, D0: Utilização de um vetor de flags que serve de marcador de vértices já visitados. Para detetar cliques, em primeiro lugar percorremos a lista de adjacências do vértice em análise marcando os seus adjacentes com a flag ativa. Após isto, percorremos a lista de adjacências de cada um destes em busca de elementos com a flag ativa (simultaneamente ligados a este e ao vértice em análise), e ao finalizar o varrimento da sua lista de adjacências desativamos a sua flag, de modo a não haver duplas contagens do mesmo clique.

Modo A1

Ideia:

Partimos do Algoritmo de Prim, estudado nas aulas teóricas, no qual se constrói a árvore mínima de suporte adicionando arestas, tais que, cada aresta adicionada é a de menor peso, de todas as que ligam um vértice que já está na árvore a outro que ainda não esteja.

Desenvolvemos um Algoritmo de Prim otimizado em que se percorre a lista de adjacências de cada vértice, sendo feita uma atualização da posição de cada um dos vértices que lhe são adjacentes na fila prioritária, se necessário (se o custo associado diminuir: aumenta a prioridade).

No fim do varrimento da lista desse vértice, retira-se da fila prioritária o vértice mais prioritário, se este estiver desconectado dos vértices anteriormente inseridos, modifica-se o seu valor e a sua lista de adjacências é percorrida. Caso contrário, cria-se uma aresta (contendo esse mesmo vértice e a franja deste elemento que já está contida na MST), sendo esta aresta inserida na MST. Este processo repete-se até a fila prioritária se encontrar vazia: MST totalmente calculada.

Otimizações:

Para tornar mais simples e eficiente, em termos temporais, o processo de procura do vértice associado à aresta de menor custo, optámos pela utilização de um acervo. A alternativa seria percorrer todos os vértices e verificar quais deles ainda não se encontravam no “backbone”, sendo que seria escolhido aquele que tivesse a menor distância a este (processo de complexidade O(V)) .

Dito isto, e dado que os processos de inserção, remoção e de modificação de prioridade têm custo O(lg N), achamos mais prudente a utilização desta estrutura.

Criámos ainda uma segunda tabela contendo as posições dos vértices no acervo, sendo deste modo mais fácil aceder aos seus elementos: O(1).

Após uma testagem e análise dos 3 algoritmos implementados: PrimOtimizado, Kruskal e KruskalOrdenacaoParcial, optámos pela utilização do PrimOtimizado como forma de determinar o "backbone", pois este é claramente o mais rápido (dados na Tabela 1).

Testagem:

Últimas 3 colunas – tempo (s)

Ficheiro (.routes)	Nº vértices	Nº arestas	Densidade	PrimOtim	Kruskal	KruskalOP
1200_004_02	1200	2400	4	0,008	0,011	0,013
1200_010_02	1200	6000	10	0,012	0,016	0,126
1200_030_02	1200	18000	30	0,029	0,046	1,039
1200_070_02	1200	42000	70	0,070	0,107	1,174
1200_100_02	1200	60000	100	0,127	0,173	3,287
1200_200_02	1200	120000	200	0,321	0,431	4,972
1200_300_02	1200	180000	300	0,478	0,624	6,099
1200_400_02	1200	240000	400	0,870	1,031	11,634
1200_500_02	1200	300000	500	1,006	1,286	7,173
1200_600_02	1200	360000	600	1,786	2,101	12,336
1200_700_02	1200	420000	700	1,678	2,109	14,260
1200_800_02	1200	480000	800	2,314	2,806	30,410

Tabela 1: Resultados experimentais de ficheiros gerados aleatoriamente para Modo A1

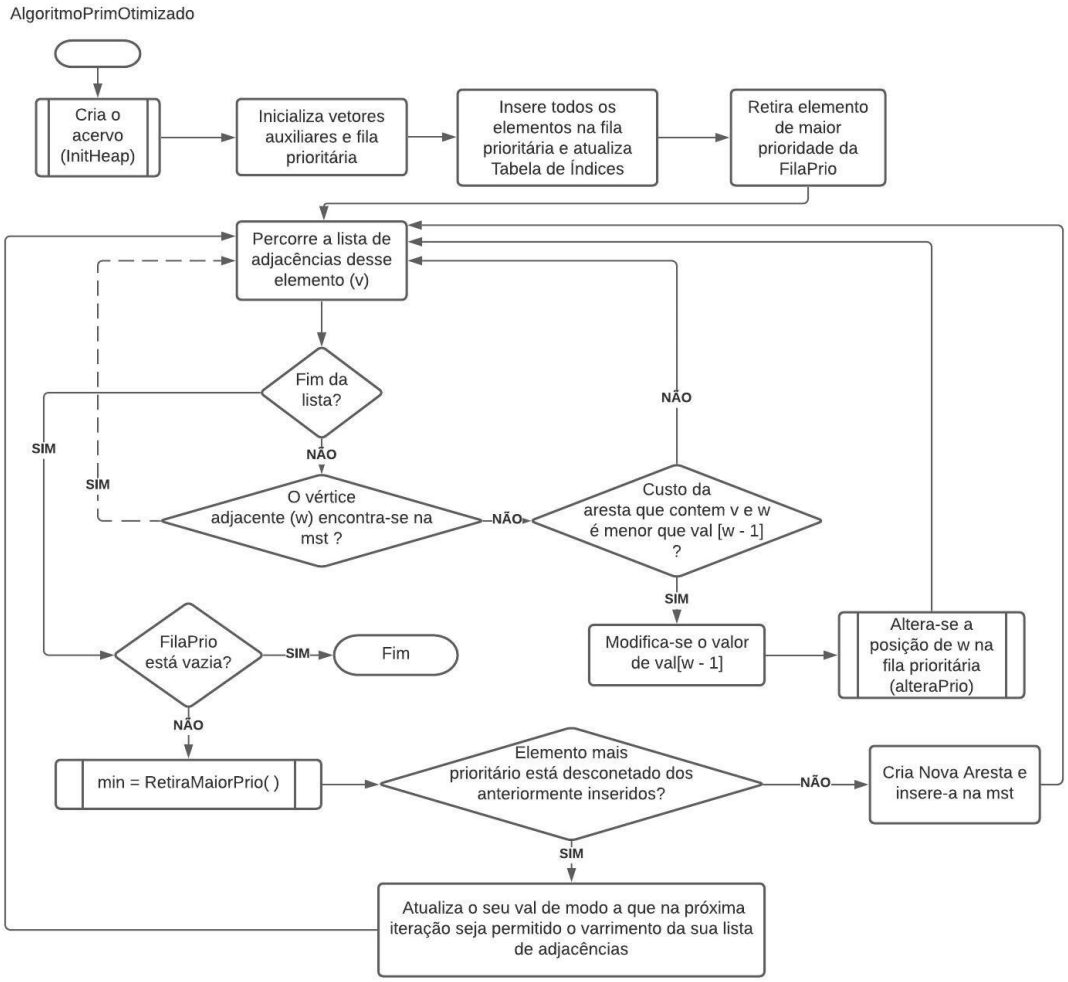


Figura 4: Fluxograma de "AlgoritmoPrimOtimizado"

*val refere-se à distância do vértice ao "backbone".

Modo B1

Ideia:

Em primeiro lugar calculamos o "backbone" do grafo.

Tendo como base o problema da conetividade, temos de resolver o seguinte: tendo um conjunto inicial C ("backbone"), ao interditar uma das suas rotas, ficamos com 2 conjuntos desconexos A e B, ambos contidos em C. O objetivo é reconectar o conjunto A, B com uma rota pertencente ao grafo, com o menor custo possível.

Método de procura da reconexão: após estabelecer a conetividade das arestas do "backbone" exceto a rota interditada, procurar (num vetor de arestas que representa o grafo) rotas que liguem os 2 conjuntos desconexos, guardando a de menor custo.

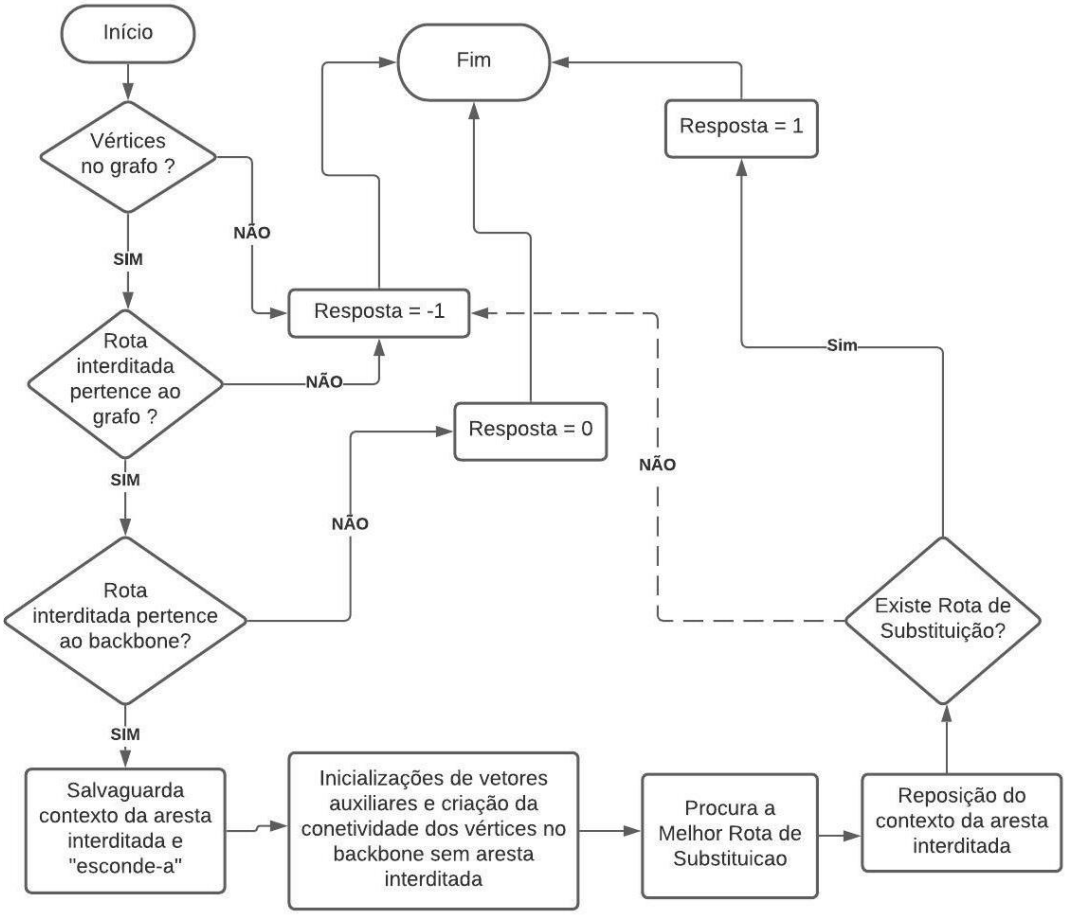


Figura 5: Fluxograma de "RemocaoERotaSubstituicao"

Iremos agora explicar detalhadamente como procuramos a melhor rota de substituição.

A procura da melhor rota de substituição, da rota interditada *a*-*b* (e sendo A o subconjunto de vértices do "backbone" onde o elemento *a* pertence, e B o subconjunto no qual o elemento *b* está contido), consiste em percorrer o vetor de arestas em busca de uma rota que cumpra os seguintes requisitos:

(1) Vértices dessa rota não pertencem ao mesmo conjunto (Procura = 0);
(2) Um dos vértices pertence ao conjunto A e o outro ao conjunto B, de modo a ser possível conectar os conjuntos com esta rota;
(3) De todas as rotas que cumpram (1) e (2), a mais barata.

Decidimos percorrer todo o vetor de arestas verificando as 3 condições enunciadas.

A outra opção consistia em ordená-lo por custo (caso custo fosse igual ordenar por vértice) e a primeira rota de substituição encontrada era a mais barata. Neste caso não compensa porque E lg E > E.

Mas em modos em que fazemos sucessivas (N) procuras da melhor rota de substituição, ou seja, complexidade (NE), compensa ordenar por custo antes de entrar no ciclo de chamadas a essa procura: assim que for encontrada a primeira rota de substituição esta é retornada: complexidade (E lg E + N E/2), sendo E/2 a média do local onde encontramos a primeira rota de substituição no vetor de arestas.

Modo C1

Ideia e otimização:

Para o cálculo do "backbone" não variacional por exclusão da aresta entre os vértices v_i e v_j basta executar o seguinte processo:

- Procurar a melhor rota de substituição (aresta de menor custo que reponha a conetividade original);
- Criar um "backbone" de substituição, semelhante ao "backbone" original, com a exceção da aresta em análise ser substituída pela rota de substituição encontrada.

Nos casos em que não haja rota de substituição, o novo "backbone" será constituída pelas arestas do "backbone" original exceto pela aresta interditada.

Chegámos a esta conclusão através do seguinte raciocínio:

Vamos supor que ao remover uma aresta x do backbone B, com vértices v1x e v2x, uma outra aresta y de vértices v1y e v2y com menor custo que a aresta mais cara de B, que não estava em B devido a fechar um ciclo, passa a estar disponível pois deixa de fechar esse ciclo.

- Então se temos um B, e se y fecha um ciclo entre v1x e v2x, ao remover x e fazer a conetividade de B/x, temos que v1y e v2y, tal como v1x e v2x ficam em conjuntos desconexos W e Q, por exemplo v1x e v1y pertencem a W e v2x e v2y pertencem a Q, ou v1x e v2y pertencem a W e v2x e v1y pertencem a Q.
- Ou seja, ao calcular a melhor rota de substituição que une os conjuntos W e Q, temos 2 situações: y é essa mesma rota (condição troca da aresta interditada pela rota de substituição verdadeira), ou há outra rota que resolve esse mesmo problema (que é rota de substituição - condição troca da aresta interditada pela rota de substituição verdadeira) e y volta a fechar um ciclo (voltará a não entrar no "backbone" de substituição pois não é a rota de substituição).

Fica assim demonstrado que só a rota de substituição da aresta interditada pode entrar no "backbone" de substituição. Esta condição foi uma importante otimização ao desempenho temporal deste modo.

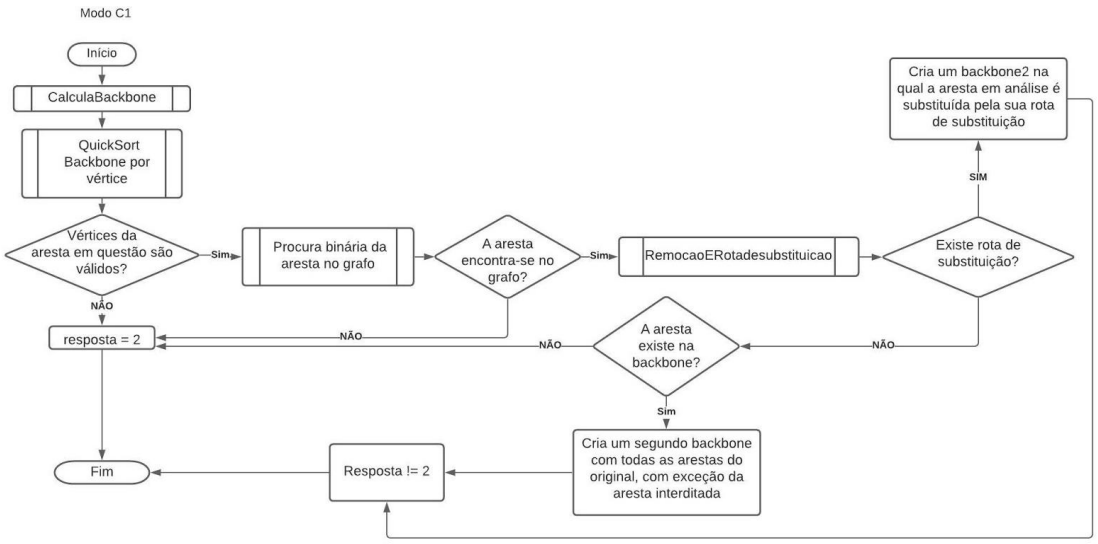


Figura 6: Fluxograma do funcionamento do modo C1

Quando a resposta é igual a 2 imprime o cabeçalho, o número de arestas do "backbone", o custo total deste, -1 e o "backbone" original (todas as suas arestas e respectivos custos) .

Quando a resposta é diferente de 2 imprime o cabeçalho, o número de arestas do "backbone", o custo total destas, o número de arestas do "backbone" de substituição, o custo total destas, as arestas do "backbone" original com o seu custo e o mesmo para as arestas do "backbone" de substituição.

Modo D1

Ideia:

Em primeiro lugar calculamos o "backbone" do grafo.

Tendo como base o problema da conetividade, temos de resolver o seguinte desafio: tendo um conjunto inicial B ("backbone"), ao interditar um aeroporto, ficamos com N conjuntos desconexos, ambos contidos em B. O objetivo é reconectar os conjuntos com as rotas pertencentes ao grafo com o menor custo possível, de modo a restabelecer um "backbone" do grafo sem aquele aeroporto.

Método de procura da reconexão: após estabelecer a conetividade das arestas do "backbone" exceto as rotas interditadas pela interdição do aeroporto, procurar (num vetor de arestas que representa o grafo) rotas que ligam os conjuntos desconexos. Temos assim algumas rotas candidatas a restabelecerem um "backbone". Basta então, para estas executar o Algoritmo de Kruskal, e inserir no novo "backbone" as rotas de substituição necessárias e suficientes, com o menor custo possível.

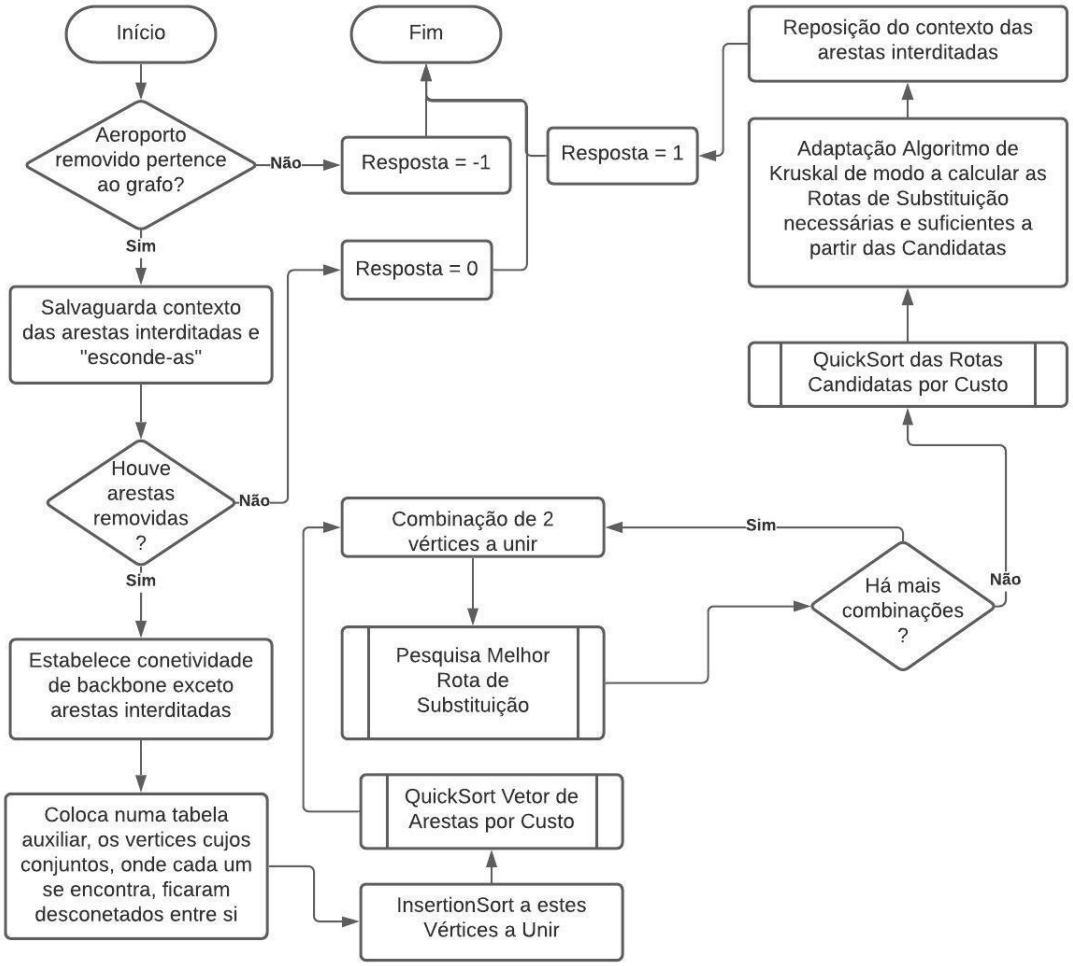


Figura 7: Fluxograma da função “RemoveAeroporto”

Otimização:

Ao fazer a pesquisa da melhor rota de substituição, decidimos ordenar o vetor de arestas por custo (quando o custo é igual, ordenação por vértice).

Assim a primeira rota de substituição encontrada é a rota de substituição de menor custo.

Ao fazer sucessivas (N) procuras da melhor rota de substituição, ou seja, complexidade (NE), compensa ordenar o vetor de arestas por custo antes de entrar no ciclo de chamadas a essa procura: com o objetivo de, ao fazer a procura, assim que encontrar a primeira rota de substituição, retorná-la, não sendo necessário percorrer o que resta do vetor. Complexidade ($E \lg E + N\, E/2$) < NE, para N e E elevados, sendo E/2 a média da posição do vetor de arestas onde encontramos a rota de substituição e paramos a procura.

Modo E1

Ideia:

Tomamos como ponto de partida o método de procura da rota de substituição de menor custo, detalhado no Modo B1, e decidimos aplicá-lo a todas as arestas do "backbone", inicialmente calculado pelo Algoritmo de Prim otimizado.

Nos casos em que esta rota existe, imprimem-se ambas as arestas e custos associados, todavia quando esta não existe o programa limita-se a imprimir a rota do "backbone" e o custo associado.

Otimização:

Dado o facto de que, para grafos com "backbone" de tamanho considerável, fazemos imensas procuras da melhor rota de substituição, o que tornava o programa lento, tivemos de fazer algumas otimizações: não é necessário verificar a existência dos vértices em análise no "backbone" (uma vez que estes pertencem a este de certeza) o que melhorou ligeiramente o desempenho da função.

No entanto, o que foi substancial na forma de tornar o programa mais eficiente foi a ordenação do vetor de arestas do grafo por ordem crescente de custo (e para custos iguais ordenar por vértice) - QuickSort com complexidade de O(E lg E). Isto possibilitou aceder à rota de substituição sem ser necessário (em maior parte dos casos) percorrer todo o vetor de arestas (a primeira aresta encontrada que fosse capaz de repor a conetividade original, corresponde à aresta de substituição de menor custo).

Ao fazer sucessivas (N) procuras da melhor rota de substituição, ou seja, complexidade (NE), compensou ordenar o vetor de arestas por custo antes de entrar no ciclo de chamadas a essa procura e assim que for encontrada a primeira rota de substituição parar a procura. Complexidade (E lg E + N E/2) < NE, para N e E elevados, sendo E/2 a média da posição do vetor de arestas onde encontramos a rota de substituição.

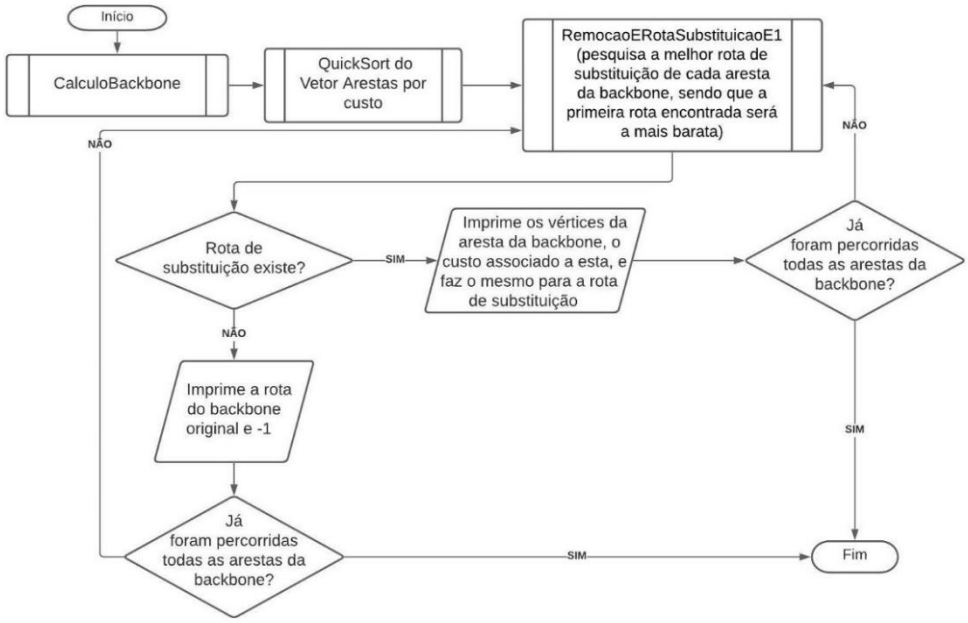


Figura 8: Fluxograma do funcionamento do modo E1: função mais importante neste modo "EscreveBackboneEBackup"

Decisões tomadas

Na secção "Estruturas de Dados utilizadas", enunciamos a nossa escolha das estruturas a que recorremos neste projeto. Iremos agora justificar essas decisões.

1-Utilização de uma tabela de lista de adjacências:

Após uma análise cuidada da complexidade de operações em grafos, com diferentes tipos de representações em memória, e da quantidade de memória que cada uma dessas representações utiliza, escolhemos representar os grafos processados em tabelas de listas de adjacências.

Em comparação com a representação do grafo numa matriz de adjacências, verificamos muitas vantagens: Espaço necessário = O(V + E), melhor que O(V^2), pois para grafos de tamanho considerável a diferença seria abismal: para grafos pouco densos e de elevado número de vértices a memória gasta por uma matriz de adjacências é enorme.

Para além disto, em termos de desempenho, a representação em lista de adjacências é bastante eficiente: por exemplo para fazer procura de um caminho de u a v, em listas de adjacências temos O(V+E), enquanto que em matriz de adjacências O(V^2), o que para grafos pouco densos e de enormes dimensões é muito pouco eficiente.

2-Utilização de um vetor de arestas:

Ao longo do programa são necessárias muitas operações de procura por uma aresta (exemplo procura da melhor rota de substituição). É muito mais rápido fazer esta procura binária num vetor, do que em listas de adjacências. Para além de que, já que necessitamos de ter em memória uma grande parte das arestas para representar o "backbone", mais vale ter todas as arestas e apenas termos de alocar um conjunto de ponteiros para estes elementos.

3-Escolha do Algoritmo de Prim Otimizado e Utilização de um Acervo:

De todos os algoritmos de construção de MST's decidimos escolher o Algoritmo de Prim como a base do programa para o cálculo do "backbone", sendo que tal decisão se deveu ao facto de termos detectado nos testes a supremacia (do ponto de vista de tempo) deste método em relação aos outros (Kruskal e Kruskal com ordenação parcial, ambos implementados em "grafo.c", mas não utilizados no programa).

De facto, a utilização de heaps foi essencial para atingir esta rapidez e eficiência, na medida em que estes facilitaram o processo de procura das arestas de menor custo, evitando o varrimento a partir de um ciclo que poderia ser enorme, dependendo do número de vértices.

Comparando as complexidades (O(E lg V) no caso do Prim, O(E lg E) e O(E + X lg V) para Kruskal e Kruskal com ordenação parcial, respetivamente) pouco podemos concluir acerca de qual o melhor algoritmo.

Todavia, reparámos (por via de testes realizados com ficheiros fornecidos pelos docentes e outros criados por nós) que o Prim para grafos mais densos consegue ser mais rápido, sendo que apenas para densidades muito baixas (abaixo de 2) os outros algoritmos conseguem ser mais vantajosos.

4-Escolha de Algoritmo de Conetividade:

Dos algoritmos estudados nesta UC que tratam o Problema da Conetividade, escolhemos utilizar o Compressed Weighted Quick Union. Este algoritmo tem complexidade O(N lg*N), e tem a grande vantagem de após estabelecida a conetividade do sistema, consegue fazer procuras com O(lg*N)~O(5)=O(1), ou seja, procura quase constante! Definição de protótipos e implementação de operações de procura e de união em “conetividade.c” e “conetividade.h”.

5-Escolha de Algoritmo de Ordenação:

Neste aspeto possuíamos conhecimento sobre 2 algoritmos com um bom desempenho e que não despendem muita memória: HeapSort e QuickSort otimizado.

Acabámos por optar por um **QuickSort otimizado** - em “ordenacao.c” - com complexidade O(N lg N): Utilização de uma pilha explícita de modo a não haver recursividade e a podermos minimizar a memória gasta, através da colocação de menores tabelas a ordenar por cima da pilha, para esta crescer o mínimo possível.

Também implementamos mediana de três, de modo a otimizar a escolha de um pivot que faça uma divisão mais eficiente da tabela; partição em três para melhorar a rapidez no caso de termos muitos elementos iguais; e ignorar tabelas de tamanho não maior que 10, realizando um InsertionSort no fim de modo a fazer estas ordenações (operação que apenas custa O(N) pois a tabela está quase totalmente ordenada).

6-Escolha de Algoritmo de Procura:

Utilizamos a Pesquisa Binária, implementada em "airroutes.c", devido a ter uma complexidade O(lg N) bastante melhor que a complexidade de O(N) da Procura Sequencial.

Variáveis:
V - Número de vértices no grafo;
E- Número de arestas no grafo;

Modo A1

- Inicialização de vetores auxiliares e da fila prioritária: O(V);
- Inserção de todos os vértices na heap, usando para isso FixUp, e atualizando a fila de índices: O(V lg V);
- Para cada aresta no "backbone":
    - Retira elemento de maior prioridade (troca a posição do elemento de maior prioridade com o de menor e retira este da fila) e restabelece a condição de acervo (fazendo FixDown a este): O(lg V);
    - Percorre a lista de adjacências do elemento anteriormente retirado da fila prioritária e altera a posição dos vértices que lhe são adjacentes no acervo (FixUp) caso o seu valor diminua: O(V lg V). Na realidade o número de elementos numa lista de adjacências << V, logo o caso médio é lg V;
    - Verifica se a fila está vazia: O(1);
    - Procura binária no vetor de arestas por uma aresta contendo o vértice mais prioritário e a sua franja, e insere essa aresta na MST: O(lg E);
    - Ou seja, no total deste ciclo temos aproximadamente O(E lg V),
- Caso o elemento mais prioritário esteja desconectado dos anteriormente inseridos, o seu val é atualizado de modo a que na próxima iteração seja permitido (pela condição no ciclo), o varrimento da sua lista de adjacências: caso exceção de um grafo constituído por 2 árvores não conectadas: O (V);
- Liberta o espaço alocado por todas as estruturas e vetores usados na função: O(1);
- QuickSort do "backbone" O(E lg E);
- Logo Modo A1: **O(E lg E)**, **caso médio E** (com V constante) pois número de arestas no "backbone" < número de arestas total.

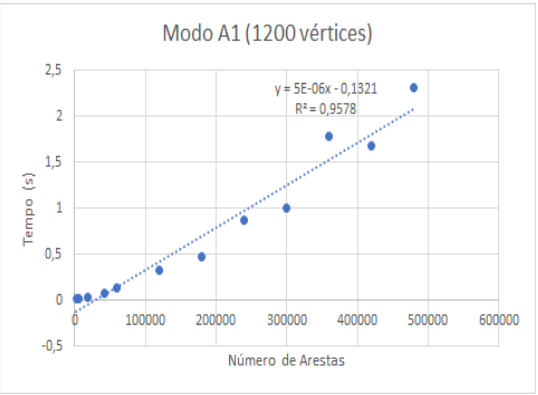


Figura 9: Gráfico da complexidade temporal do Modo A1 em função do número de arestas, com o número de vértices fixo (1200 vértices)

Ficheiro (.routes)	Nº vértices	Nº arestas	Densidade	Modo	tempo (s)
1200_004_02	1200	2400	4	A1	0,008
1200_010_02	1200	6000	10	A1	0,012
1200_030_02	1200	18000	30	A1	0,029
1200_070_02	1200	42000	70	A1	0,07
1200_100_02	1200	60000	100	A1	0,127
1200_200_02	1200	120000	200	A1	0,321
1200_300_02	1200	180000	300	A1	0,478
1200_400_02	1200	240000	400	A1	0,87
1200_500_02	1200	300000	500	A1	1,006
1200_600_02	1200	360000	600	A1	1,786
1200_700_02	1200	420000	700	A1	1,678
1200_800_02	1200	480000	800	A1	2,314

Tabela 2: Resultados experimentais de ficheiros gerados aleatoriamente Modo A1

Modo B1

- Cálculo e ordenação do "backbone" ~ E, com V constante (explicado no modo A1);
- Verificação se vértice é válido: O(1);
- Salvaguardas e reposições de contexto: O(E);
- Estabelece conetividade do "backbone" - CWQU: O(E lg*E);
- Como percorremos o vetor de E arestas, fazendo um máximo de 5 procuras, sendo que cada procura é O(lg*E): Pesquisa da melhor rota de substituição O(E lg*E);
- Criação da Aresta, alocações e libertações de memória: O(1);
- Logo Modo B1: **O(E lg*E) ~ O(E)**, pois lg*E é aproximadamente uma constante: Podemos observar este comportamento linear no seguinte gráfico, relativo aos pontos experimentais no Modo B1, apresentados na Tabela 3.

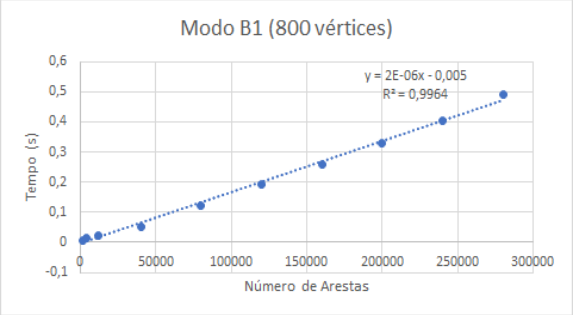


Figura 10: Gráfico da complexidade temporal do Modo B1 em função do número de arestas, com o número de vértices fixo (800 vértices)

Ficheiro (.routes)	Nº vértices	Nº arestas	Densidade	Modo	tempo (s)
800_004_01	800	1600	4	B1	0,007
800_010_01	800	4000	10	B1	0,013
800_030_01	800	12000	30	B1	0,022
800_100_01	800	40000	100	B1	0,053
800_200_01	800	80000	200	B1	0,122
800_300_01	800	120000	300	B1	0,193
800_400_02	800	160000	400	B1	0,257
800_500_01	800	200000	500	B1	0,329
800_600_01	800	240000	600	B1	0,405
800_700_01	800	280000	700	B1	0,491

Tabela 3: Resultados experimentais de ficheiros gerados aleatoriamente Modo B1

Modo C1

- Cálculo e ordenação do "backbone" ~ E (explicado no modo A1);
- Verificação se vértices são válidos: O (1);
- Faz procura binária para encontrar o índice da aresta (que contém os vértices em análise) no vetor de arestas: O (lg E);
- Executa a função "RemocaoERotadesubstituicao", utilizada para encontrar uma rota de substituição (modo B1): O (E lg*E);
- Procura o índice da aresta removida no "backbone" (procura binária): O (lg E);
- Copia "backbone" original para "backbone" final: O (E);
- Caso exista rota de substituição, troca a aresta em análise do "backbone" final pela rota de substituição: O (1);
- Ordena o segundo "backbone" por vértice (InsertionSort): O(E), pois este "backbone" já se encontra inicialmente ordenado com a exceção de apenas uma das suas arestas;
- Se não existir rota de substituição, o "backbone" original é copiado para o final e é retirada a aresta em análise: O (E);
- Calcula o custo do novo "backbone": O (E);
- Alocações e libertações: O (1);

- Logo <u>Modo C1</u>: **O(E lg*E) ~ O(E)**, pois lg*E é aproximadamente uma constante: Podemos observar este comportamento linear no seguinte gráfico, relativo aos pontos experimentais no Modo C1, apresentados na Tabela 4.

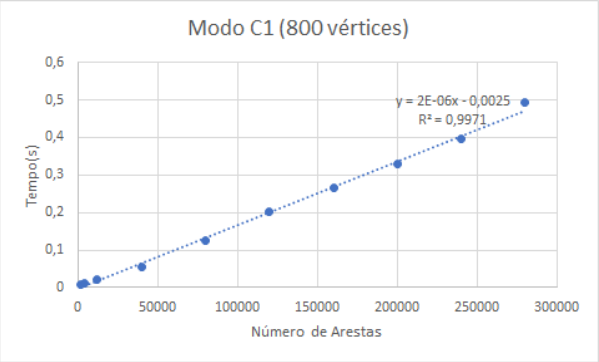


Figura 11: Gráfico da complexidade temporal do Modo C1 em função do número de arestas, com o número de vértices fixo (800 vértices)

Ficheiro (.routes)	Nº vértices	Nº arestas	Densidade	Modo	tempo (s)
800_004_02	800	1600	4	C1	0,008
800_010_02	800	4000	10	C1	0,011
800_030_02	800	12000	30	C1	0,022
800_100_02	800	40000	100	C1	0,056
800_200_02	800	80000	200	C1	0,125
800_300_02	800	120000	300	C1	0,203
800_400_01	800	160000	400	C1	0,266
800_500_02	800	200000	500	C1	0,329
800_1000_02	800	240000	600	C1	0,396
800_1300_02	800	280000	700	C1	0,492

Tabela 4: Resultados experimentais de ficheiros gerados aleatoriamente Modo C1

Modo D1

- Cálculo e ordenação do "backbone" ~ E, com V constante (explicado no modo A1);
- Verificação se vértice é válido: O(1);
- Salvaguardas e reposições de contexto: O(E);
- Estabelece conetividade do "backbone" sem as arestas removidas - CWQU: O(E lg*E);
- Verificação dos Vértices a Unir: O(E);
- Ordenação dos Vértices a Unir $O(E^2)$, na realidade número de vértices a unir << E, e estamos a fazer InsertionSort de uma tabela pequena e já quase ordenada $\Omega(E)$; Motivo(1);
- Ordenação Vetor Arestas: O(E lg E);
- Pesquisa das Candidatas a Rota de Substituição, sendo que a pesquisa da melhor rota de substituição é O(E), mas é a primeira encontrada que é retornada pois o vetor de arestas está ordenado por custo, caso médio E/2, logo: $O(E^3)$. Mas no caso médio, como o número de arestas removidas << E, podemos considerá-lo uma constante logo $\Omega(E)$. Motivo (2);
- Ordenação QuickSort a vetor Candidata Rota Substituição O(E lg E), na realidade este vetor é de tamanho << E. Motivo(3);
- Determinar Rotas de substituição que vão para a MST O(E lg*E), não realidade o número de iterações = número de candidatas << E. Motivo(4);
- Logo, <u>Modo D1</u>: **$O(E^3)$**->Só ocorre em caso extremo, em que todas as arestas são removidas (quando estão todas ligadas ao mesmo vértice).
- **No caso médio a complexidade é E lg E**, pelos motivos (1), (2), (3), (4).

- Curiosamente para os dados experimentais a complexidade cresce linearmente com o aumento do número de arestas (**E**): melhor que o caso médio.

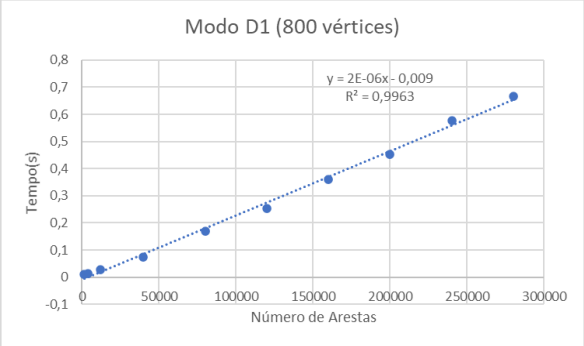


Figura 12: Gráfico da complexidade temporal do Modo D1 em função do número de arestas, com o número de vértices fixo (800 vértices)

Ficheiro (.routes)	Nº vértices	Nº arestas	Densidade	Modo	tempo (s)
800_004_03	800	1600	4	D1	0,01
800_010_03	800	4000	10	D1	0,014
800_030_03	800	12000	30	D1	0,028
800_100_03	800	40000	100	D1	0,073
800_200_03	800	80000	200	D1	0,169
800_300_03	800	120000	300	D1	0,252
800_400_03	800	160000	400	D1	0,361
800_500_03	800	200000	500	D1	0,454
800_1000_03	800	240000	600	D1	0,576
800_1300_03	800	280000	700	D1	0,667

Tabela 5: Resultados experimentais de ficheiros gerados aleatoriamente Modo D1

Modo E1

- Cálculo e ordenação do "backbone" ~ E, com V constante (explicado no modo A1);
- Ordena o vetor de arestas por ordem crescente de custo (utilização do algoritmo de ordenação QuickSort): O(E lg E);
- Chama a função RemocaoERotadesubstituicaoE1 um máximo de E vezes:
  - Salvaguardas e reposições de contexto: O(E);
  - Estabelece conetividade do "backbone" - CWQU: O(E lg*E);
  - Pesquisa das Candidatas a Rota de substituição, sendo que a pesquisa da melhor rota de substituição é O(E lg*E) ~ O(E), mas é a primeira encontrada que é retornada, pois o vetor de arestas está ordenado por custo, caso médio é (E/2 lg*E) ;
  - Criação da Aresta: O(1);
  - Logo E vezes este processo: $O(E^2 \lg *E)$;
- Imprime as arestas do "backbone" e a sua rota de substituição encontrada (ou "-1" no caso em que esta não existe): O(1);
- Logo, Modo E1: **$O(E^2 \lg *E) \sim O(E^2)$**, tal como se verifica no gráfico abaixo apresentado, dado que se percorrem todas as arestas do "backbone", sendo calculado para cada uma a rota de substituição. Devido à otimização, raramente é necessário iterar sobre todo o vetor de arestas para encontrar uma rota de substituição. Isto faz com que experimentalmente se verifique um termo de ordem 2 muito baixo, e uma complexidade quase linear.

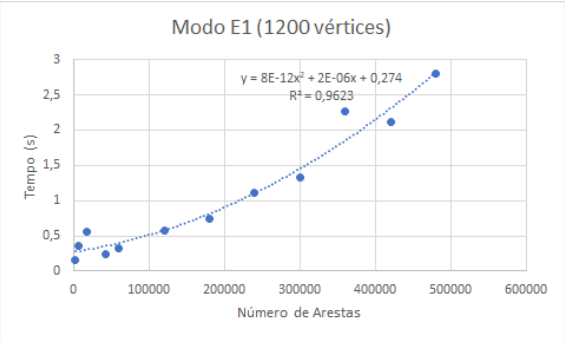


Figura 13: Gráfico da complexidade temporal do Modo E1 em função do número de arestas, com o número de vértices fixo (1200 vértices)

Ficheiro (.routes)	Nº vértices	Nº arestas	Densidade	Modo	tempo (s)
1200_004_01	1200	2400	4	E1	0,155
1200_010_01	1200	6000	10	E1	0,358
1200_030_01	1200	18000	30	E1	0,554
1200_070_01	1200	42000	70	E1	0,244
1200_100_01	1200	60000	100	E1	0,319
1200_200_01	1200	120000	200	E1	0,579
1200_300_01	1200	180000	300	E1	0,748
1200_400_01	1200	240000	400	E1	1,111
1200_500_01	1200	300000	500	E1	1,327
1200_600_01	1200	360000	600	E1	2,264
1200_700_01	1200	420000	700	E1	2,108
1200_800_01	1200	480000	800	E1	2,805

Tabela 6: Resultados experimentais de ficheiros gerados aleatoriamente Modo E1

Geral (N problemas a resolver)

O modo que apresenta uma pior complexidade temporal é o E1: $O(E^2)$.

Logo para N problemas a resolver a complexidade do nosso programa é $O(N\ E^2)$.

Num caso médio cada problema a resolver será da ordem de E: linear. Logo no total NE.

Análise da Complexidade de Memória

Vamos agora analisar o pico máximo de memória que o programa utiliza:

No geral

- Grafo representado por uma tabela de listas de adjacências: O(V + E);
- Vetor de Arestas com todas as arestas do grafo: O(E);
- Vetor "backbone" de ponteiros para o vetor de arestas: O(E);
- Outras variáveis alocadas no main: O(1);

Modo A1

- Fila de índices representado por um vetor de inteiros: O(V);
- Fila de prioridades que representa um acervo: O(V);
- Vetor de vértices da MST que se encontram mais próximos dos vértices do grafo (franja): O(V);
- Vetor val do tipo double, representativo da distância provisória do nó ao "backbone": O(V);
- Logo, modo A1: O(V + E).

Modo B1

- Uma rota de substituição e um ponteiro para esta: O(1);
- Vetores id e sz de suporte à conetividade: O(V);
- Aresta auxiliar: O(1);
- Logo, modo B1: O(V + E).

Modo C1

- Vetor "backbone" de substituição: O(E). Podíamos ter evitado esta alocação ao usar o "backbone" original com apenas mais uma aresta e o seu índice;
- Cria uma aresta para cada posição do vetor "backbone": O(E). Alocação desnecessária, pois, podíamos e devíamos ter reaproveitado as arestas já alocadas no vetor de arestas;
- Uma rota de substituição e um ponteiro para esta O(1);
- Logo, modo C1: O(V + E).

Modo D1

- Vetor de rotas de substituição: O(V);
- 4 Vetores de salvaguarda do contexto: índice, salvaguarda do custo e salvaguarda dos vértices e 1 vetor VerticesAUnir: O(V);
- Vetores id e sz de suporte à conetividade: O(V);
- Vetor CandidataRotaSubstituicao: $O(V^2)$ em casos muito extremos;
- Logo, modo D1: $O(V^2)$.

Modo E1

- Vetores id e sz de suporte à conetividade: O(V);
- Aresta interditada: O(1);
- Uma rota de substituição e um ponteiro para esta: O(1);
- Logo, modo E1: O(V + E).

Pior caso

$O(V^2)$: no modo D1.

Mas isto é um caso muito extremo em que um dos vértices se encontra ligado a todos os vértices do grafo. Em condições normais, ignorando esse caso especial, para qualquer um dos modos a memória gasta é da ordem de O(V + E).

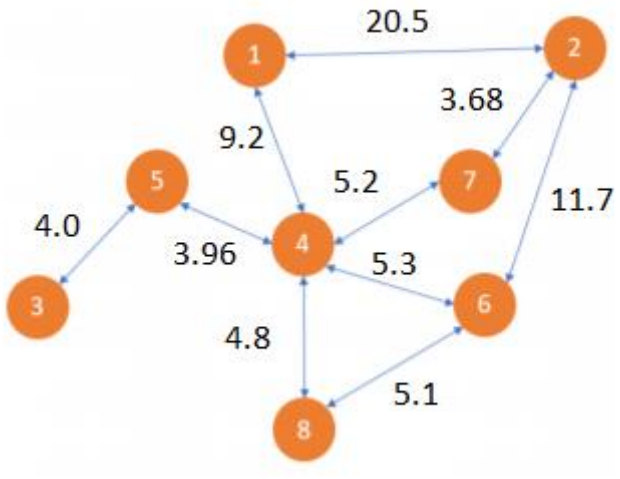
Análise Crítica e Avaliação do Desempenho

No geral, pensamos ter conseguido uma prestação de sucesso neste projeto pelos seguintes motivos. Conseguimos obter complexidades temporais de qualidade para todos os modos, e conseguimos passar 100% dos testes que tínhamos como desafio no mooshak, em ambas as fases. Para além de que planeamento, organização e trabalho árduo permitiram atingir estes objetivos atempadamente (em ambas das fases: cerca de 1 semana antes da data limite).

Na fase final apenas conseguimos atingir 100% dos testes com sucesso após várias submissões: após as primeiras submissões possuíamos alguns falhanços por tempo excedido e apostamos na otimização de complexidade nos Modos C1, D1 e E1. Um outro problema que estava a fazer com que o programa não passasse alguns dos testes à primeira tentativa, foi, por lapso, termos um float numa função que calcula o custo do "backbone", em vez de um double. Graças à publicação de testes pela equipa docente detetámos a gralha e corrigimos. Muito obrigado.

Exemplo de funcionamento

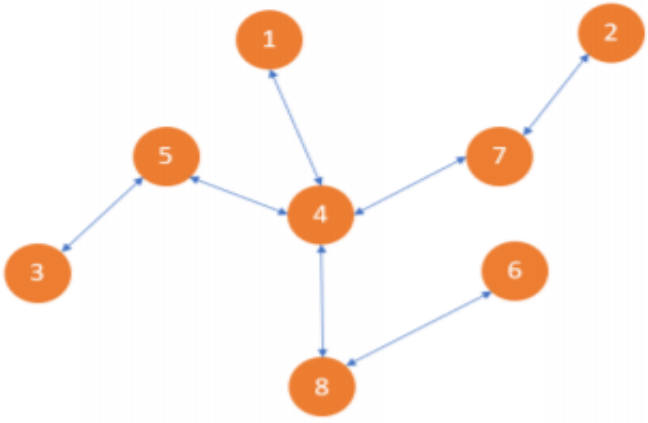
Para demonstrar o funcionamento do nosso programa decidimos dar um exemplo para o modo D1, utilizando o grafo dado no enunciado do projeto:



V1	V2	Custo
1	2	20,5
1	4	9,22
2	6	11,7
2	7	3,68
3	5	4
4	5	3,96
4	6	5,3
4	7	5,2
4	8	4,8
6	8	5,1

O grafo é lido e armazenado em listas de adjacências. É alocado também um vetor de arestas (a azul) com todas as arestas do grafo. E um conjunto de ponteiros para esse vetor de arestas que irá guardar o "backbone".

Como 1º passo, calculamos o "backbone" através do Algoritmo de Prim Otimizado, obtendo:



Após isto, vamos verificar o que acontece quando removemos um dos vértices no "backbone": vamos assumir que queremos interditar o vértice 4. Temos o nosso "backbone" original (a verde), com a seguinte conetividade: {1,2,3,4,5,6,7,8}.

V1	V2	Custo
1	4	9,22
2	7	3,68
3	5	4
4	5	3,96
4	7	5,2
4	8	4,8
6	8	5,1

Vamos “esconder” as arestas com o vértice 4 (a cinzento).

De seguida, executamos o problema da conetividade ao conjunto, obtemos os conjuntos A={2,7}, B={3,5} e C={6,8}.

Ora os conjuntos A, B, C passaram a estar desconexos.

V1	V2	Custo
0	0	inf
2	7	3,68
3	5	4
0	0	inf
0	0	inf
0	0	inf
6	8	5,1

Objetivo: Reconectá-los com o menor custo possível de rotas de substituição.

Criamos um vetor chamado vértices a unir (a laranja), contendo vértices cujos conjuntos onde se encontram são os que queremos unir. Ou seja, estes são os vértices que estavam ligados a 4 no “backbone” original. Também ordenamos este vetor por ordem crescente.

1	5	7	8

Vamos agora procurar candidatas a rotas de substituição, isto é, para cada combinação de 2 vértices a unir, vamos procurar a melhor rota de substituição no vetor de arestas do grafo original, e inseri-la neste novo vetor chamado Candidatas a Rota de substituição (CRS a branco). (NA = Não existe rota de substituição)

Combinações			
	1	5	7	8
1	X	(1,5)	(1,7)	(1,8)
5	X	X	(5,7)	(5,8)
7	X	X	X	(7,8)
8	X	X	X	X

Combinações		Candidatas RS	
1	5	NA	NA
1	7	1	2
1	8	NA	NA
5	7	NA	NA
5	8	NA	NA
7	9	2	6

Damos agora um exemplo de cálculo da Rota de substituição para a aresta (1,5), C1 = 1, C2 = 5, utilizando o vetor de arestas (exceto rotas escondidas) ordenado por custo. P(V1, V2) simboliza a procura se os vértices V1 e V2 estão no mesmo conjunto. (Utilização das 3 regras enunciadas na secção “Modo B1”):

x = irrelevante;

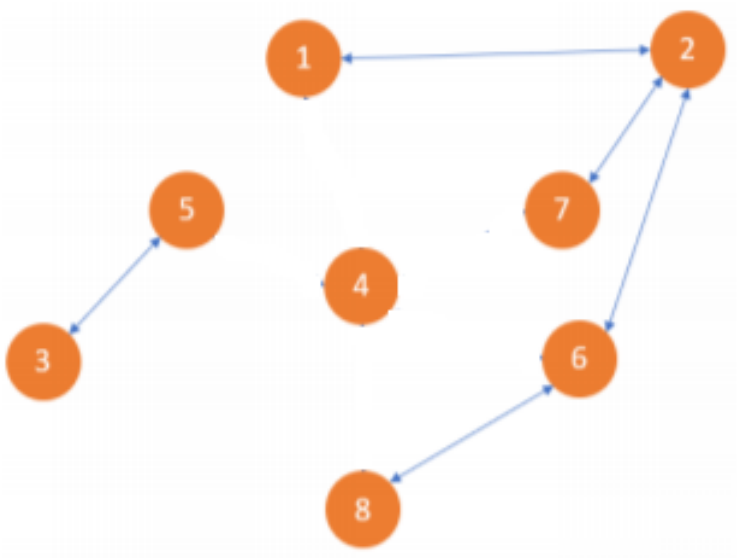
Há rota de substituição se !P(V1,V2) && ((P(V1,C1) && P(V2,C2)) || (P(V2,C1) && P(V1,C2)).

V1	V2	Custo	P(V1,V2)	P(V1,C1)	P(V2,C2)	P(V2,C1)	P(V1,C2)	Há rota?
2	7	3,68	1	x	x	x	x	0
3	5	4	1	x	x	x	x	0
6	8	5,1	1	x	x	x	x	0
4	6	5,3	0	0	x	x	0	0
2	6	11,7	1	0	0	0	0	0
1	2	20,5	0	1	0	0	0	0

Para finalizar temos de ordenar estas candidatas por ordem crescente de custo: [(2,6,11.70), (1,2,20.50)] e verificar quais destas podem entrar na MST (Algoritmo de Kruskal), obtendo o conjunto de rotas de substituição à interdição do vértice 4: [(2,6,11.70), (1,2,20.50)]

Antes de imprimirmos os resultados, seguindo as especificações do enunciado, temos de ordenar o "backbone" e as rotas de substituição por vértice.

Obtemos assim o resultado final:



Output:

8 10 D1 4 7 35.96 2

1 4 9.22

2 7 3.68

3 5 4.00

4 5 3.96

4 7 5.20

4 8 4.80

6 8 5.10

1 2 20.50

2 6 11.70

Bibliografia

[1] AED (IST/DEEC) Slides das Aulas Teóricas 2020/2021 1º Semestre;

[2] AED (IST/DEEC) Enunciado do Projeto “AirRoutes” 2020/2021 1º Semestre;

[3] Lucidchart: Website utilizado na produção de fluxogramas;

[4] Microsoft Office Excel: Produção de gráficos.