INSTITUTO SUPERIOR TÉCNICO

Relatório do Projeto

AirRoutes

1º SEMESTRE 2020/2021

Grupo: 66

Joao Paulo Patrício da Silva nº 85210

joao.p.p.silva@tecnico.ulisboa.pt

João Pedro Andrade Gonçalves nº 85211

jpedrogoncalves@tecnico.ulisboa.pt

Docente: Carlos Bispo

Índice

1. Descrição do problema

Pretende-se analisar um ficheiro de entrada contendo informação sobre rotas (partida e destino) ligando aeroportos, e o custo de cada uma. No cabeçalho encontra-se a informação relativa ao problema a resolver. Este tem 5 variantes:

- A1: cria um ficheiro contendo o conjunto mínimo de rotas que garante a ligação de todos os vértices, com o menor custo (backbone).
- B1: retira uma rota ao backbone verifica se existe alguma rota alternativa que reponha a sua ligação, representando a de menor custo.
- C1: retira uma rota e cria um novo backbone.
- D1: retira um aeroporto e todas as rotas a este associadas. Escreve todas as rotas de menor custo que reponham ligações quebradas pela remoção do aeroporto.
- E1: calcula rotas alternativas de menor custo para todas as rotas do backbone.

2. Descrição da arquitetura do programa

Para resolver os problemas acima descritos foram utilizadas duas tabelas ***st*** e ***val*** (de dimensão igual ao número de vertices do problema em análise) com a informação das ligações no backbone e seu custo, e as duas estruturas seguintes.

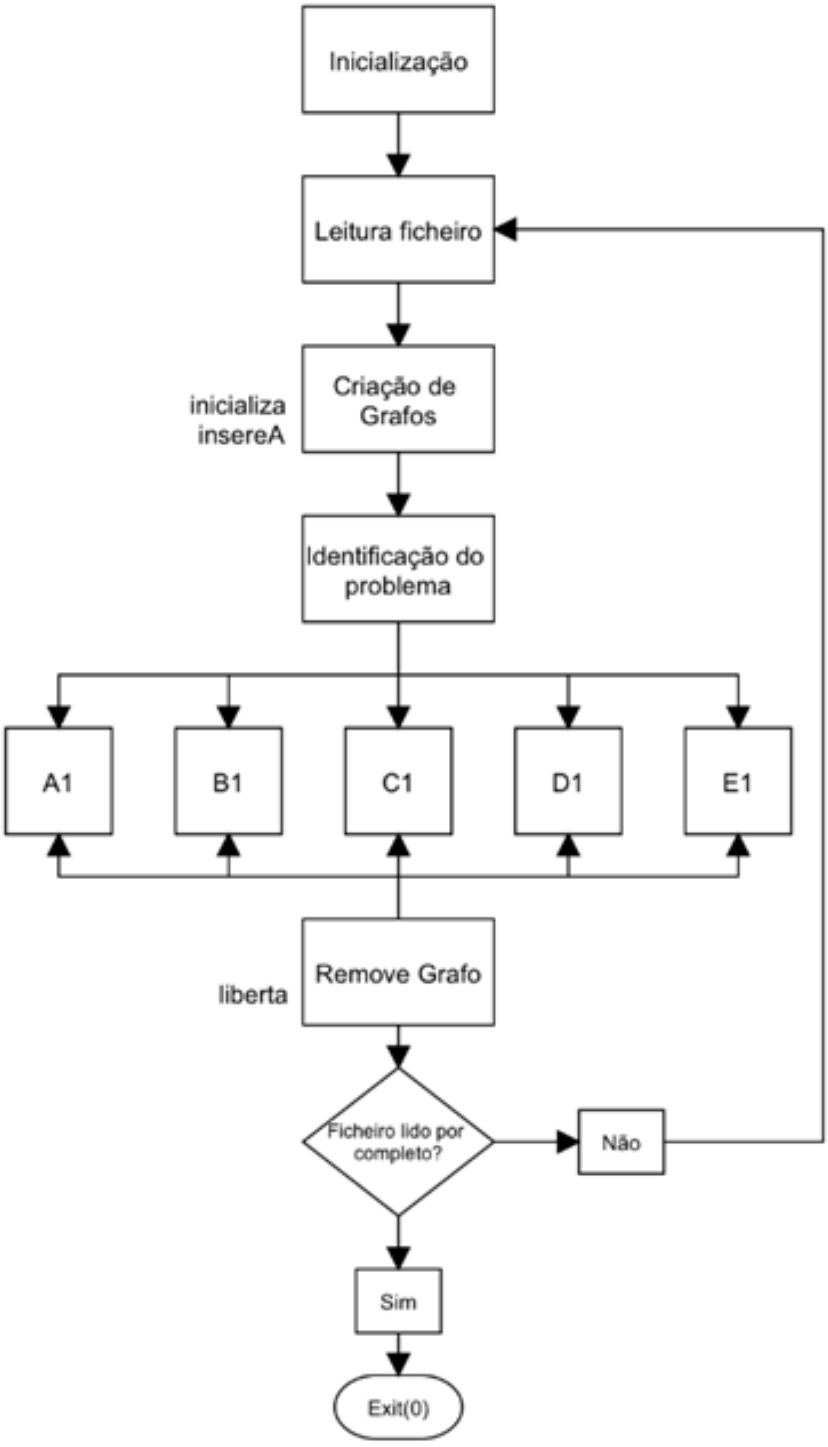
```
struct graph{
    int V; /*número de vértices*/
    int A; /*número de arestas*/
    double **m; /*matriz adjacências*/
};

typedef struct rotas{
    int p1;
    int p2;
    double c;
}Rotas;
```

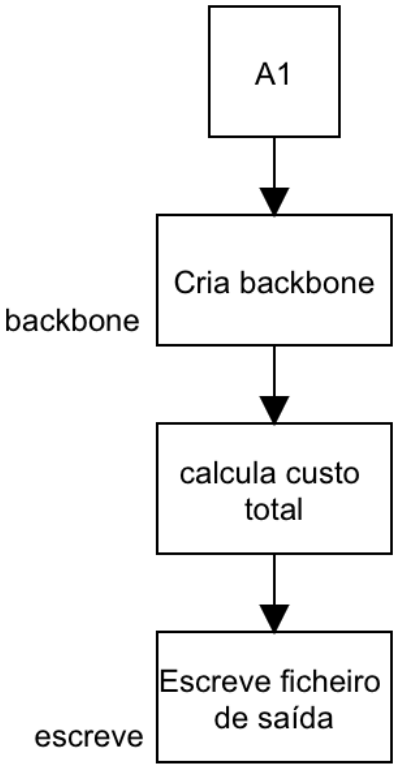
Na estrutura ***Graph*** guardam-se a informações do grafo sendo as ligações e seu respectivo custo guardadas na matriz de adjacências. A estrutura ***Rotas*** é usada na forma de vector (dimensão igual ao numero de rotas alternativas) para guardar as rotas não pertencentes ao backbone principal, para facilitar a análise das rotas alternativas.

Podem ser usadas pontualmente as seguintes tabelas para guardar informações relativas a backbones secundários ou ao backbone original (para ser escrito depois de sofrer alterações): ***stbb***, ***valbb***, ***st2***, ***val2***, ***bbv1***, ***bbv2***, ***bbc***.

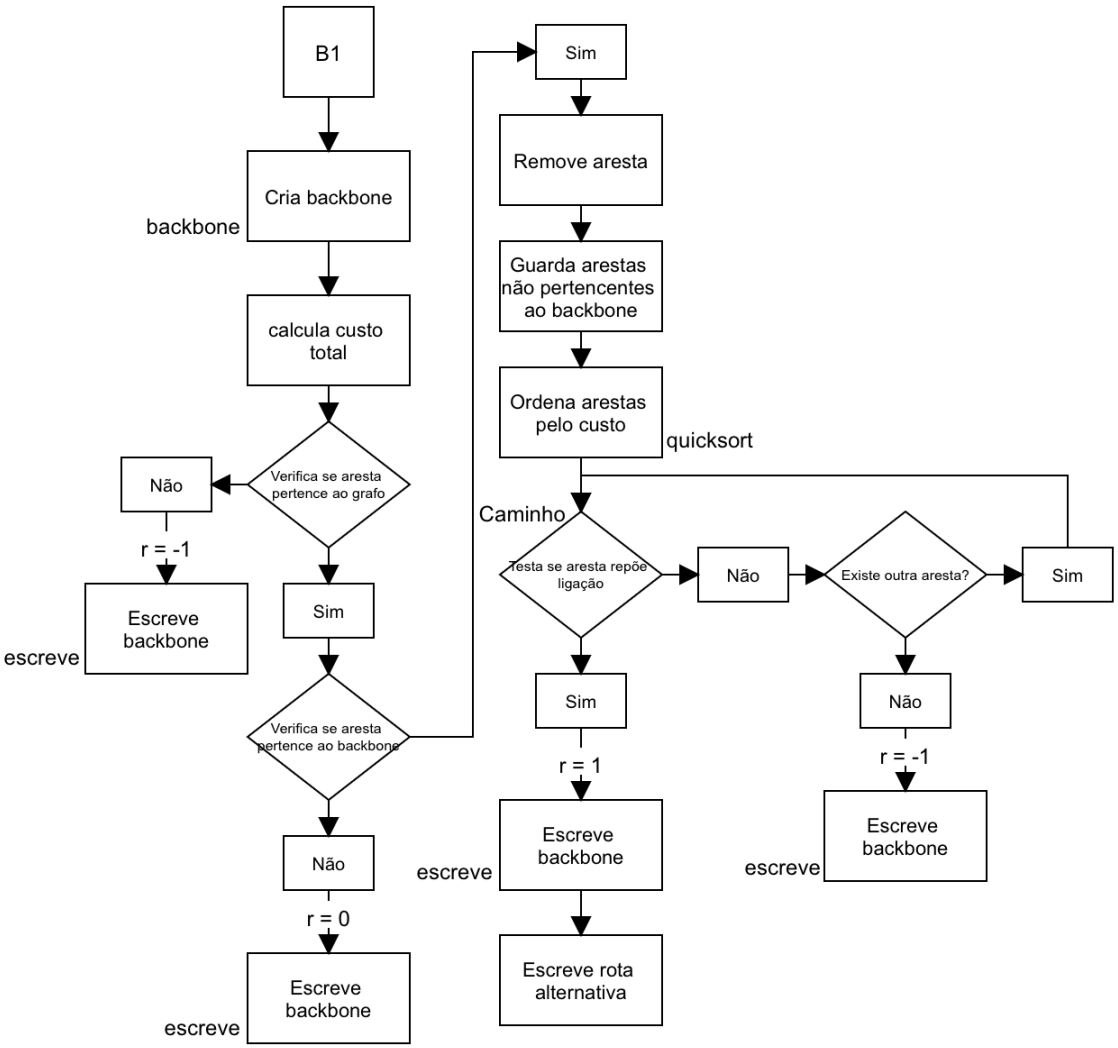
De seguida representa-se o fluxograma principal do programa sendo depois analisado com maior detalhe o fluxograma de cada uma das funções (A1, B1, C1, D1 e E1) nele representadas. Nesta secção o programa testa se o ficheiro de entrada foi escrito corretamente, aloca memoria para a struct ***Graph*** e preenche os seu valores, selecciona o problema a analisar (garantindo a análise de vários problemas caso existam) e liberta a memória alocada após leitura integral do ficheiro de entrada.


Inicialização
Leitura ficheiro
inicializa
insereA
Criação de Grafos
Identificação do problema
A1
B1
C1
D1
E1
liberta
Remove Grafo
Ficheiro lido por completo?
Não
Sim
Exit(0)

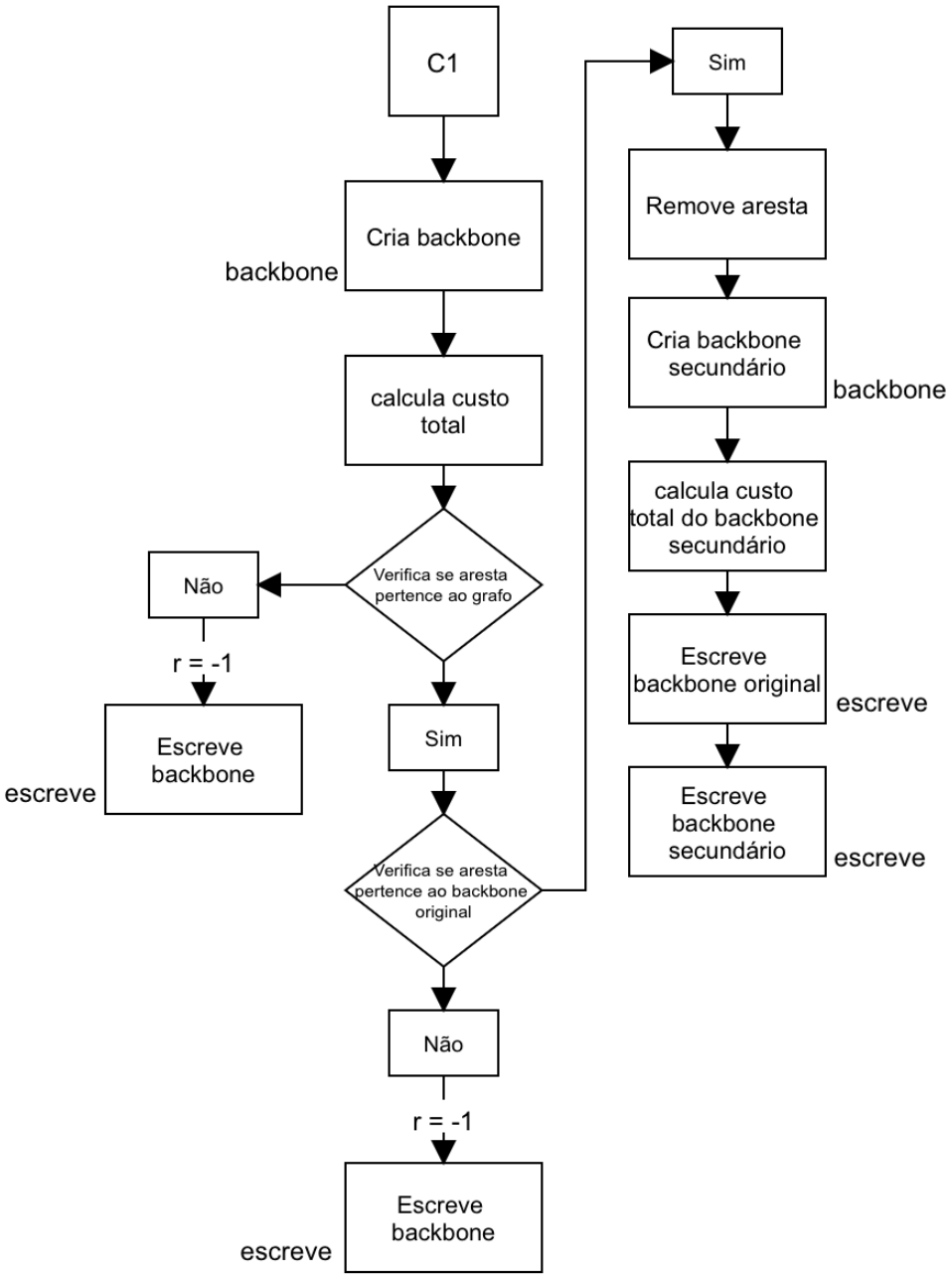
Na secção A1 do nosso problema usamos o Algoritmo de Prim para matriz de adjacências (função ***backbone***) para o cálculo do backbone sendo criadas as tabelas ***st*** e ***val*** do backbone original. Percorremos a tabela ***val*** somando todos os custos para saber o custo total do backbone e o número de arestas nele presente. Escrevemos no cabeçalho os dados do problema e o número de rotas e custo total do backbone (Para as restantes funções o cabeçalho do ficheiro de saída começa sempre desta maneira podendo ser acrescentados valores em seguida). De seguida a função ***escreve*** imprime no ficheiro de saída o backbone original com as arestas organizadas. Está comentado no código de que forma se garante a ordenação do ficheiro na função escreve.



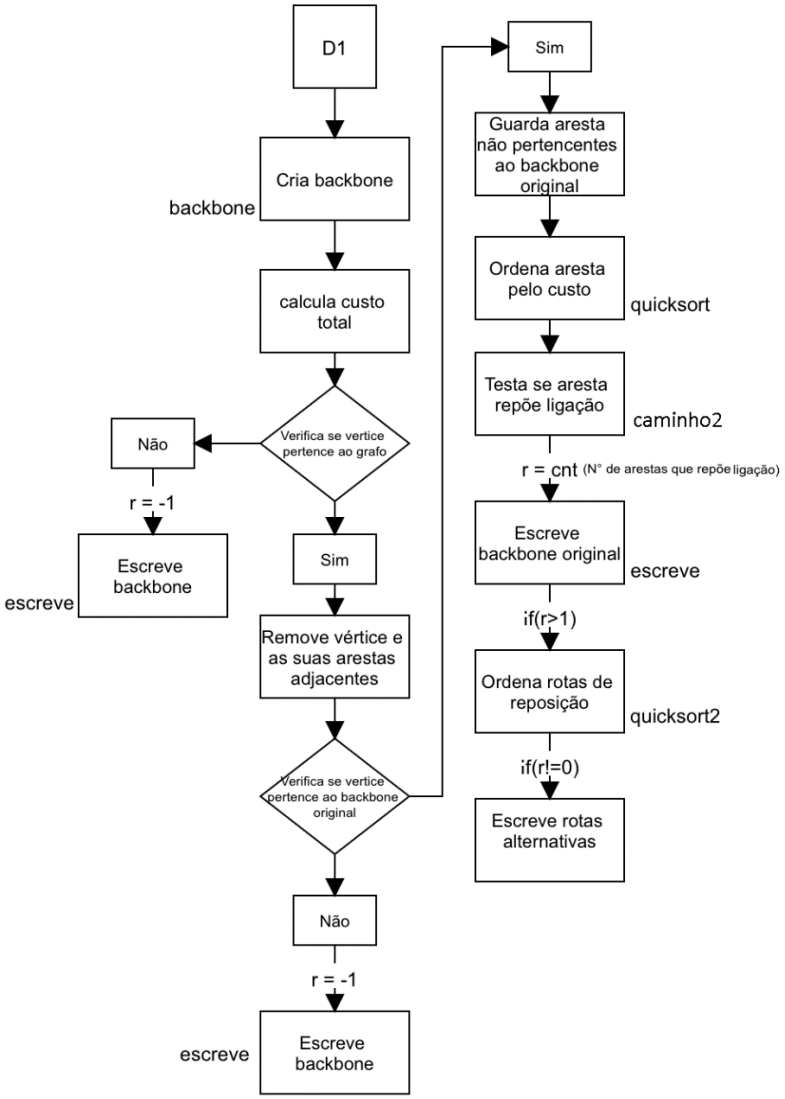
Na função B1 criamos de novo o backbone recorrendo ao Algoritmo de Prim, sendo calculado o seu custo. Se a aresta não pertencer ao grafo escrevemos o backbone através da função ***escreve*** com o valor **-1** no cabeçalho. Caso não pertença ao backbone original procedemos da mesma forma mas com o valor **0** no cabeçalho. Caso a aresta a retirar pertença ao backbone removemos a aresta e percorremos a matriz de adjacências de forma a encontrar todas arestas que não pertencem ao backbone, guardando-as na estrutura ***Rotas***. De seguida ordenamos o vetor da estrutra ***Rotas*** por ordem ascendente de custo atraves da função ***quicksort*** que implementa o algoritmo quicksort melhorado. Percorremos agora o vetor da estrutra ***Rotas*** enviando o valor de cada rota num ciclo para a função ***caminho***. Esta função testa se a aresta pertence ao backbone através do algoritmo ***Quick Union***, retornando **1** no caso em que pertence e **-1** quando não pertence. Se a função caminho retornar **1** escrevemos o backbone seguido da rota alternativa com o valor **1** no cabeçalho e terminamos o ciclo. Caso o ciclo termine sem que a função caminho retorne **1** escrevemos o backbone com o valor **-1** no cabeçalho.



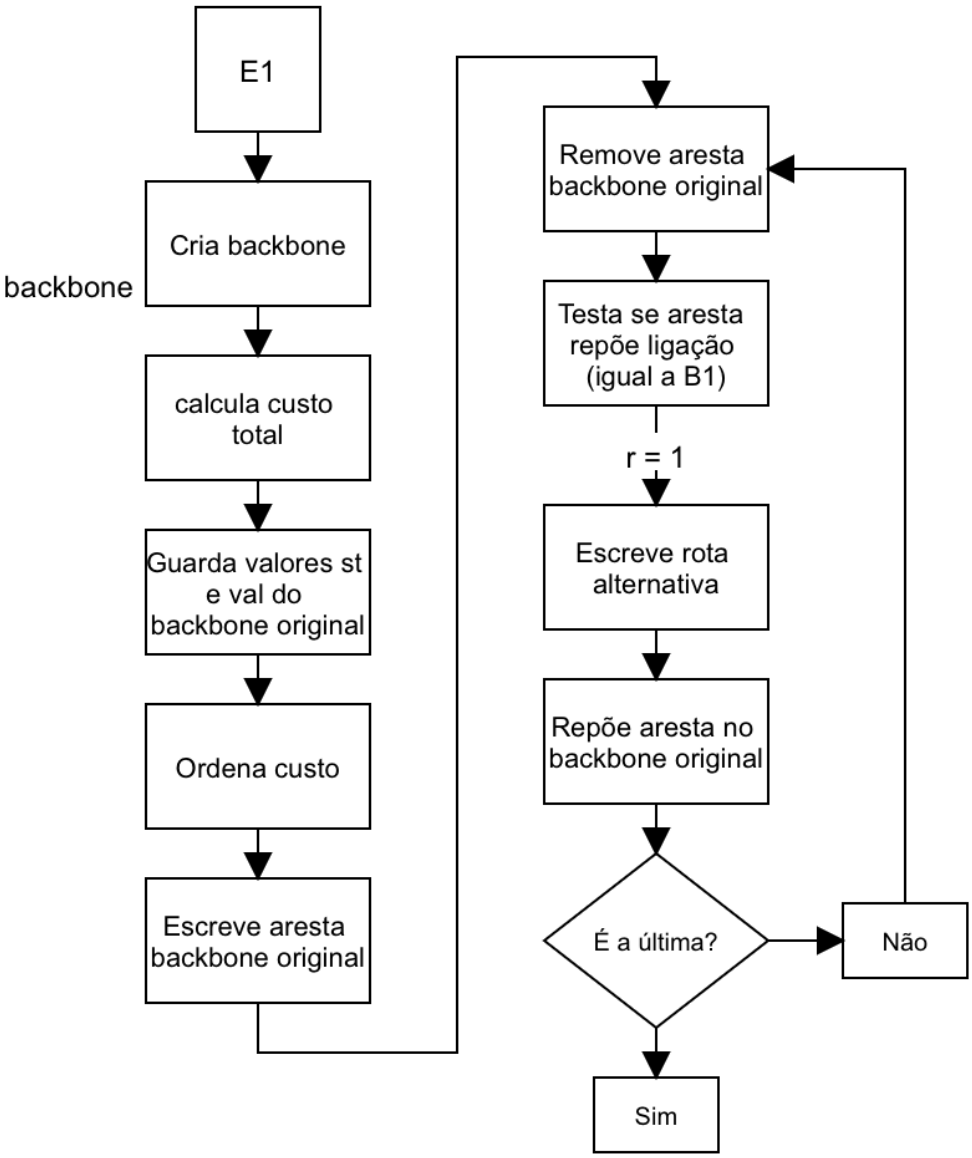
Na secção C1 criamos o backbone e calculamos o seu custo. De seguida verificamos se a aresta escolhida pertence ao grafo e ao backbone escrevendo o backbone no ficheiro de saída com o valor **-1** no cabeçalho se alguma das condições não for verificada. Caso ambas condições se verifiquem removemos a aresta e criamos um segundo backbone usando de novo a função ***backbone*** e calculamos o seu custo. Para finalizar escrevemos no ficheiro de saída ambos os backbones indicando no cabeçalho o custo e número de rotas do backbone secundário.



Na secção D1 calculamos o backbone e seu custo total. De seguida verificamos se o vértice fornecido pelo problema pertence ao grafo e ao backbone e caso não pertença escrevemos o backbone original com **-1** no cabeçalho. Caso pertença guardamos as arestas que não pertencem ao backbone e ordenamos pelo seu custo como explicado anteriormente para B1. Depois enviamos o vetor da estrutura ***Rotas*** para a função ***caminho2*** que analisa o vetor inteiro contando o número de rotas que repõe ligações através do algoritmo ***Quick Union*** e tornando o custo das rotas que não o fazem infinito (para facilitar a sua análise posteriormente). Depois escreve o backbone original seguido do número de rotas alternativas encontradas. Caso o número de rotas retornado pela função for **0** o problema acaba, caso for maior que um ordenamos o vetor ***Rotas*** (pois como é óbvio, uma única rota não necessita de ser ordenada) com o auxílio da função ***quicksort2*** (igual a ***quicksort*** mas ordena por ordem crescente do vértice) e do algoritmo Selection Sort. Caso tenham sido encontradas rotas escrevemos as rotas no ficheiro de saída.



Para a secção E1, criamos e calculamos o custo do backbone. Depois guardamos o valor de ***st*** e ***val*** para em tabelas para o escrevermos mais tarde, dados que estes valores vão sofrer alterações. Guarda também as arestas que não pertencem ao backbone. De seguida analisamos cada aresta do backbone. Para cada uma começamos por escreve-la no ficheiro de saída e depois removemos essa aresta, testamos se existe uma aresta que repõe a sua ligação (como em B1) e caso exista escrevemos os seus vértices e custo no ficheiro de saída à frente da aresta do backbone. Caso não exista escrevemos **-1**. Depois voltamos a adicionar a aresta e repetimos o ciclo até a última aresta ser analisada.



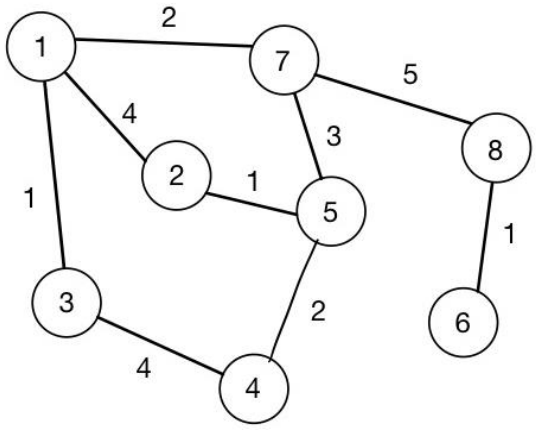
3. Análise dos requisitos computacionais

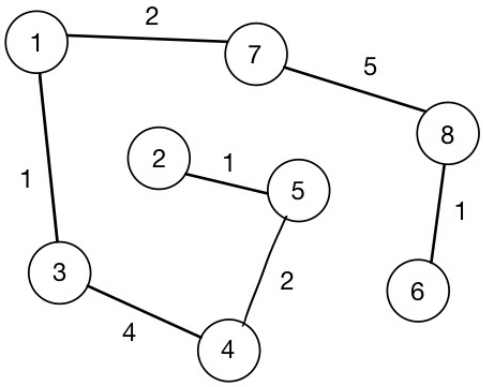
Para o nosso programa usamos o Algoritmo de Prim para matrizes de adjacências pois este permite aceder às arestas do grafo com grande velocidade. No entando para grafos com muitos vértices a memória usada será muito grande pois temos de alocar memória para uma matriz de *doubles* com dimensão $\mathbf{V^2}$. O nosso programa é então ideal para problemas de densidade elevada pois é necessário aceder a muitos valores da tabela, mas ocupa muita memória em problemas muito esparsos. A restante ocupação de memória ocorre por vetores de dimensão nunca superior a **V**.

A nível de complexidade o Algoritmo de Prim tem um tempo de execução igual a $\mathbf{V^2}$. O Quick Union **N**, sendo este o número de pontos analisados. Selection Sort tem uma complexidade de **NlogN**, sendo o quicksort um pouco inferior. Dado que temos os valores sempre guardados em matriz ou tabelas a velocidade do nosso programa é bastante alta, sendo a complexidade quando queremos aceder a um elemento igual a **O(1)**.

4. Exemplo de aplicação

Nas figuras seguintes apresentamos uma rede de aeroportos (esquerda) com o seu backbone (direita). Depois encontram-se exemplos para cada uma das variantes dos problemas.





- A1:

8	9	A1
1	2	4
1	3	1
1	7	2
2	5	1
3	4	4
4	5	2
5	7	3
6	8	1
7	8	5

8	9	A1	7	16
1	3	1		
1	7	2		
2	5	1		
3	4	4		
4	5	2		
6	8	1		
7	8	5		

- B1:

8	9	B1	2	5
1	2	4		
1	3	1		
1	7	2		
2	5	1		
3	4	4		
4	5	2		
5	7	3		
6	8	1		
7	8	5		

8	9	B1	2	5	7	16	1
1	3	1					
1	7	2					
2	5	1					
3	4	4					
4	5	2					
6	8	1					
7	8	5					
1	2	4					

- C1:

```
8  9  C1  4  5
1  2  4
1  3  1
1  7  2
2  5  1
3  4  4
4  5  2
5  7  3
6  8  1
7  8  5
```

```
8  9  C1  4  5  7  16  7  17
1  3  1
1  7  2
2  5  1
3  4  4
4  5  2
6  8  1
7  8  5
1  3  1
1  7  2
2  5  1
3  4  4
5  7  3
6  8  1
7  8  5
```

- D1:

```
8  9  D1  5
1  2  4
1  3  1
1  7  2
2  5  1
3  4  4
4  5  2
5  7  3
6  8  1
7  8  5
```

```
8  9  D1  5  7  16  1
1  3  1
1  7  2
2  5  1
3  4  4
4  5  2
6  8  1
7  8  5
1  2  4
```

- E1:

8	9	E1
1	2	4
1	3	1
1	7	2
2	5	1
3	4	4
4	5	2
5	7	3
6	8	1
7	8	5

8	9	E1	7	16	
1	3	1	1	2	4
1	7	2	5	7	3
2	5	1	1	2	4
3	4	4	5	7	3
4	5	2	5	7	3
6	8	1	-1		
7	8	5	-1		