Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

(MEEC)

Relatório

Air Routes

Grupo 28

- Manuel Graça- nº 96271
  manuel.s.graca@tecnico.ulisboa.pt

- Tiago Lourinho - nº 96327
  tiagolourinho@tecnico.ulisboa.pt

Unidade Curricular de Algoritmos e Estruturas de Dados

2020/2021

1º Semestre

Índice

1. Visão Geral ---------- 3

2. Estruturas de dados utilizadas ---------- 3

3. Algoritmos utilizados ---------- 4

4. Descrição da implementação ---------- 5

4.1. Modo -A1 ---------- 6

4.2. Modo -B1 ---------- 7

4.3. Modo -C1 ---------- 9

4.4. Modo -D1 ---------- 11

4.5. Modo -E1 ---------- 13

5. Análise dos requisitos computacionais ---------- 14

5.1. Análise Teórica ---------- 14

5.2. Análise Experimental ---------- 15

6. Exemplo Prático ---------- 19

7. Bibliografia ---------- 20

1. Visão Geral

O projeto *Air Routes* tem como base uma rede de aeroportos e as ligações entre os mesmos, sendo que o problema a resolver é representar a rede de aeroportos de tal forma que una todos os aeroportos, sem redundâncias (isto é, 2 rotas possíveis diferentes entre aeroportos) e com as ligações de menor custo possível (denominada de *backbone*). Para além disso, pode ser também necessário calcular ajustes quando por exemplo um aeroporto se encontra indisponível ou uma rota interditada.

Para resolver o problema a rede de aeroportos foi representada como um grafo ponderado e não direcionado, tendo a resolução do mesmo tido por base a conectividade que os aeroportos possuíam. Cada aeroporto é representado como um vértice e a ligação entre os mesmos como uma aresta, sendo que estes aspetos serão mencionados diversas vezes *V* representa o número total de aeroportos e *E* o número total de ligações entre os mesmos na rede inicial.

2. Estruturas de dados utilizadas

Dependo do modo foram utilizadas 3 estruturas de dados diferentes:

- Grafo

  Esta estrutura representa um grafo geral através de uma lista de adjacências, sendo que neste projeto acaba por representar o *backbone*. Desta forma a tabela de listas tem tamanho V, contudo, contrariamente ao habitual as arestas apenas são representadas na lista do vértice menor sendo que por esta razão apenas existem E nós no total (ao invés de 2E). Para além disso, a estrutura guarda também outras informações acerca do grafo que representa, como por exemplo número total de vértices, de arestas ou custo total das mesmas.

  A inserção de uma aresta nesta estrutura é feita de forma ordenada (comparando os vértices) de forma a facilitar o *print* das mesmas no final.

  A escolha desta forma de representação do grafo quando comparado à matriz de adjacências deve-se ao facto de ocupar significativamente menos memória e de não ter sido necessário aceder rapidamente a uma aresta, pois esta estrutura apenas serviu para ir guardando as arestas do *backbone* e para ser possível no final dar *print* das mesmas.

- Arestas (*Edge*)

  Esta estrutura é uma tabela de tamanho E que possui todas as arestas do grafo inicial. Cada elemento da tabela tem a informação dos dois vértices incidentes da aresta e do seu custo.

  A escolha desta forma de representação do grafo inicial deve-se ao facto de o algoritmo de *Kruskal* necessitar de uma tabela de arestas para descobrir a *MST*.

- *Backbone*

  Esta estrutura é composta por 2 tabelas de inteiros de tamanho V cada uma (a *id* e a *sz*) e serve para representar a conectividade do *backbone* final quando excluindo uma aresta ou um vértice. Serve, portanto, como auxílio na descoberta das rotas substitutas.

  Esta estrutura foi criada para simplificar o processo de descobrir se existe uma ligação entre 2 vértices ou não, pois em vez de procurarmos diretamente se existe um caminho entre eles (por exemplo, com algoritmos *DFS* ou *BFS*) apenas temos de verificar se estão conectados nesta estrutura.

3. Algoritmos Utilizados

Para a resolução do problema proposto foram utilizados algoritmos já estudados nas aulas teóricas, sendo eles:

- Algoritmo de *Kruskal*

  Sendo este um algoritmo que serve para descobrir a *MST* de um grafo, foi escolhido em vez do Algoritmo de Prim pois apresenta também uma eficiência equivalente e para alem disso faz uso do vetor de arestas ordenado que foi posteriormente utilizado para descobrir por exemplo a aresta mais barata que substitui uma que ficou indisponível. Para além disso este algoritmo possui uma vantagem em relação ao de Prim no que diz respeito à descoberta do *backbone* em grafos não ligados.

- *Quicksort*

  Para ordenar o vetor de arestas foi escolhido este algoritmo por apresentar uma eficiência média próximo do limite teórico para o problema da ordenação (O(NlogN)).

  Foi usada a versão básica deste algoritmo visto que no contexto deste problema e no método como o pretendemos resolver não se considerou necessário adicionar melhoramentos ao mesmo.

- *Compressed Weighted Quick Union*

  Para gerir o problema da conectividade foi escolhido este algoritmo uma vez que é o que apresenta melhor eficiência quando comparado aos outros algoritmos estudados, especialmente porque é necessário escalar a arvore da conectividade várias vezes para descobrir se 2 vértices estão conectados.

4. Descrição da implementação

O main do projeto verifica erros de invocação e redireciona a resposta consoante a extensão do ficheiro passado como argumento seja .routes0 ou .routes (1ª ou 2ª parte do projeto), depois lê o cabeçalho do problema e as arestas do grafo inicial, evocando um modo diferente consoante o cabeçalho.

O fluxograma do mesmo pode ser visto na figura 1.

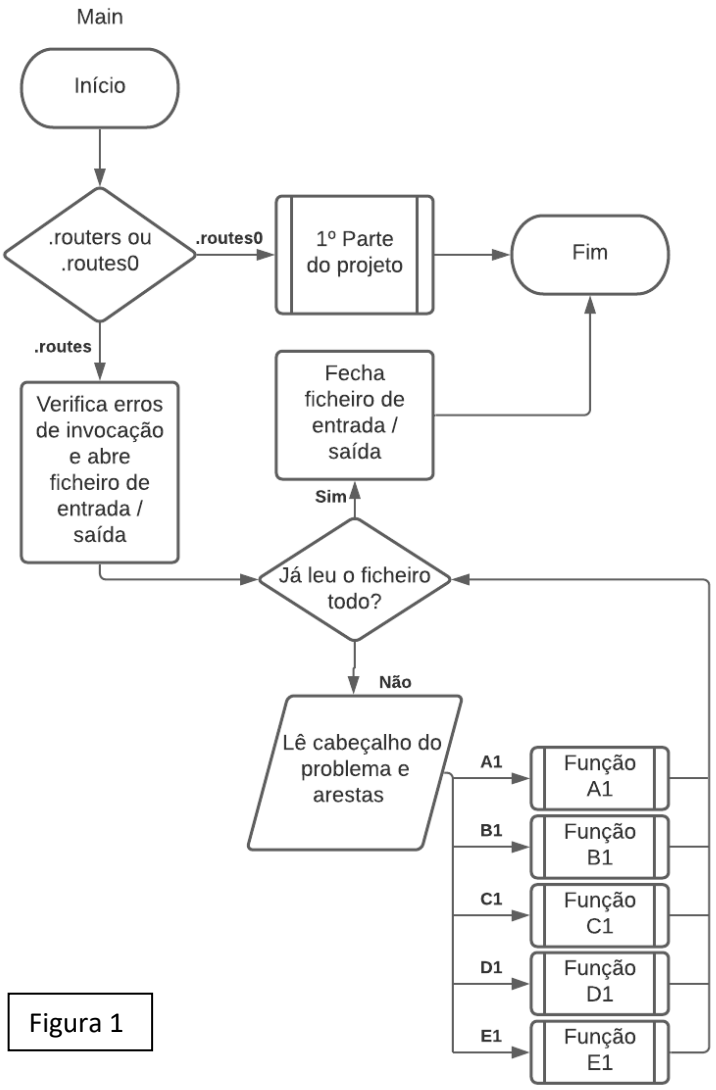

Figura 1

4.1. Modo -A1

Como já foi referido neste modo foi utilizado o algoritmo de *Kruskal* para calcular o *backbone*, ou seja, ordena-se a tabela de arestas e depois percorre-se a mesma adicionando as arestas que unem 2 vértices que não estão conectados. O processo termina quando se acabarem as arestas disponíveis (no caso de o grafo não ser ligado) ou quando tiverem sido adicionadas o número máximo de arestas ao *backbone* (V-1), pois quando são adicionadas estas arestas, e não havendo ciclos, todos os vértices já se encontram necessariamente conectados.

O parâmetro de entrada *flag* indica se no fim da função é suposto dar *print* de *-1* no cabeçalho da resposta, pois esta função tanto pode ser chamada diretamente no *main* ou então posteriormente através de outro modo (quando a aresta ou o vértice a remover é inválida/o ou quando não pertence ao grafo original).

Na figura 2 pode se analisar o fluxograma da *função_A1*.

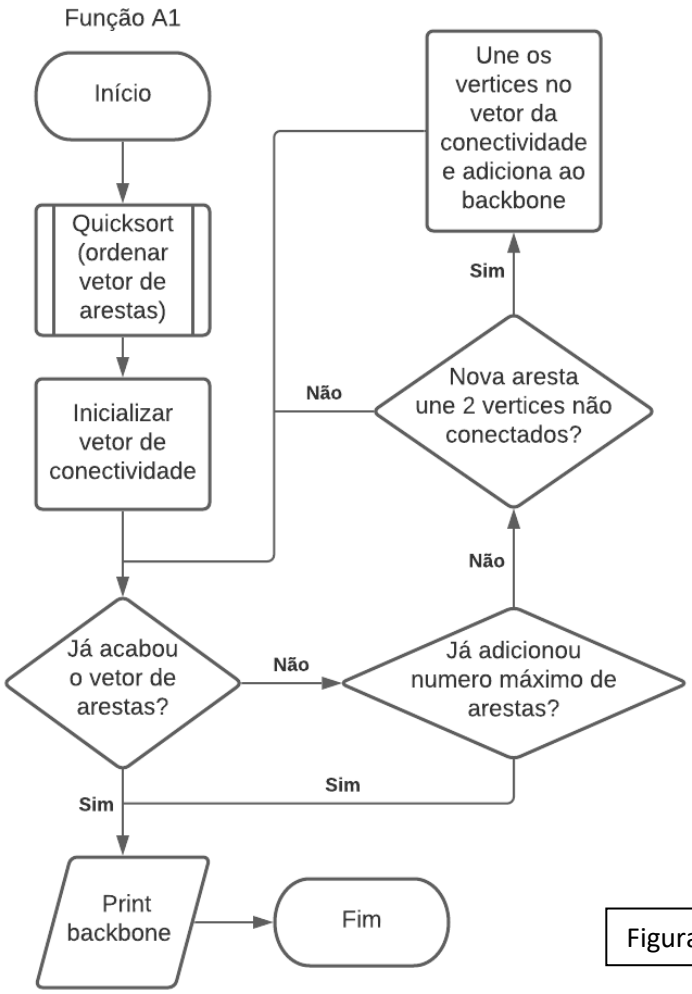

Figura 2

4.2. Modo -B1

Este modo começa por verificar se a aresta a remover existe e possui vértices válidos, em caso afirmativo continua, caso contrário chama a *função_A1*.

O resto do modo funciona de forma semelhante ao A1, isto é, calcula o *backbone* inicial da mesma forma, contudo vai registando a conectividade do mesmo quando removida uma aresta (ou seja, estamos a trabalhar com 2 vetores de conectividade em simultâneo, um faz parte do algoritmo de *Kruskal* e o outro representa o *backbone* variante). Para além disso, temos uma *flag* que indica se a aresta a remover foi introduzida no backbone ou não. Caso a aresta a remover não faça parte do mesmo, então não é necessário ir à procura de uma aresta substituta, caso contrário vamos à procura dessa aresta. Para tal, fazemos uso do vetor da conectividade do *backbone* variante, isto é, voltamos a percorrer o vetor ordenado de arestas (descartando todas as arestas que foram introduzidas no *backbone)* à procura de uma que una 2 vértices não conectados. Se a encontrarmos sabemos que é a aresta que repõe a conectividade original de menor custo, uma vez que o vetor estava ordenado. Caso percorramos o vetor todo é porque não existe substituta.

Por exemplo, supondo que temos um grafo inicial ligado como o da figura 3a (e com *backbone* original da figura 3b) a tabela da conectividade do *backbone* quando se ignora a aresta retirada (figura 3b para a 3c) vai possuir 2 raízes (uma raiz associada aos vértices 2 e 7, e a outra aos restantes), desta forma a aresta substituta é a que una 2 vértices que estejam associados a raízes diferentes (neste caso seria a aresta a verde na figura 3d por ser a alternativa de menor custo).

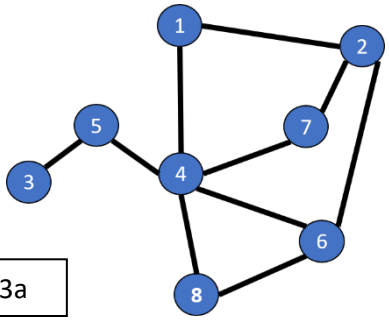

Figura 3a

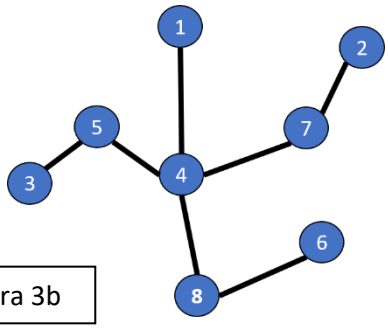

Figura 3b

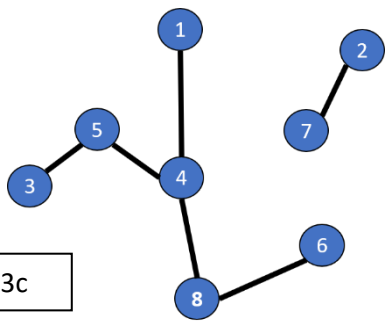

Figura 3c

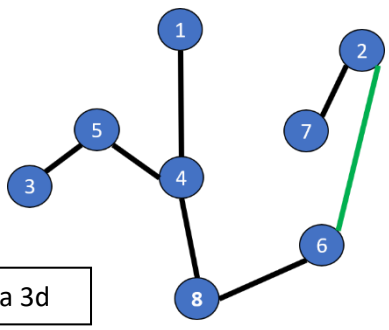

Figura 3d

Na figura 4 pode se analisar o fluxograma da *função_B1*.

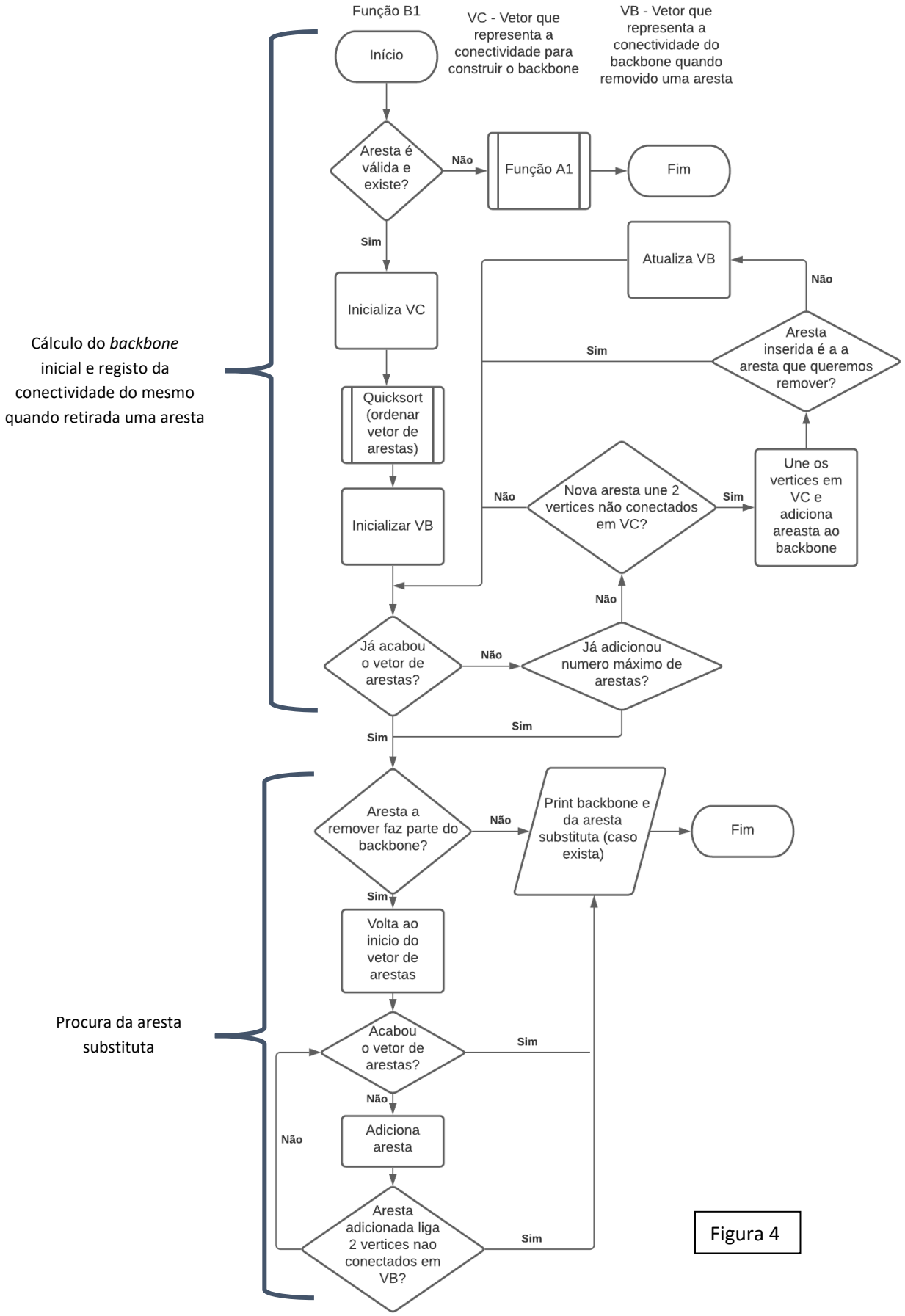
Função B1
VC - Vetor que representa a conectividade para construir o backbone
VB - Vetor que representa a conectividade do backbone quando removido uma aresta
Início
Aresta é válida e existe?
Não
Função A1
Fim
Sim
Atualiza VB
Não
Inicializa VC
Aresta inserida é a a aresta que queremos remover?
Sim
Quicksort (ordenar vetor de arestas)
Une os vertices em VC e adiciona areasta ao backbone
Não
Nova aresta une 2 vertices não conectados em VC?
Sim
Inicializar VB
Não
Já acabou o vetor de arestas?
Não
Já adicionou numero máximo de arestas?
Sim
Sim
Aresta a remover faz parte do backbone?
Não
Print backbone e da aresta substituta (caso exista)
Fim
Sim
Volta ao inicio do vetor de arestas
Acabou o vetor de arestas?
Sim
Não
Não
Adiciona aresta
Aresta adicionada liga 2 vertices nao conectados em VB?
Sim
Cálculo do backbone inicial e registo da conectividade do mesmo quando retirada uma aresta
Procura da aresta substituta

Figura 4

4.3. Modo -C1

Este modo começa por verificar se a aresta a remover existe e possui vértices válidos, em caso afirmativo continua, caso contrário chama a *função_A1*.

Efetivamente neste modo é realizado o algoritmo de *Kruskal* 2 vezes no mesmo varrimento do vetor ordenado de arestas (ou seja, vamos trabalhar outra vez com 2 vetores distintos de conectividade). Isto é, a construção do *backbone* original e do seu variante (quando se elimina uma aresta) é idêntica até ao momento em que aparece a aresta a remover. Desta forma, enquanto não aparece a aresta, ao perguntar se uma aresta liga 2 vértices não conectados, em caso de resposta afirmativa, inserimos a aresta e registamos a mudança da conectividade em ambos os *backbones*. A partir do momento em que aparece a aresta a remover, já não se pode fazer apenas 1 pergunta pois as conectividades de cada um vão diferir, isto é, ao testar uma nova aresta perguntamos se liga 2 vértices não conectados no *backbone* original, se sim então inserimos a aresta e atualizamos a conectividade, de seguida repetimos a mesma pergunta e o mesmo procedimento para o *backbone* variante. Se a aresta a remover chegou a ser introduzida no *backbone* então damos print dos 2 *backbones* distintos, caso não tenha aparecido damos apenas print do *backbone* original.

A partir do momento em que o *backbone* original fica totalmente preenchido (isto é, com V-1 arestas) passamos apenas a executar o algoritmo de *Kruskal* para o *backbone* variante (evitando assim procurar se 2 vértices estão conectados desnecessariamente).

Na figura 5 pode se analisar o fluxograma da *função_C1*.

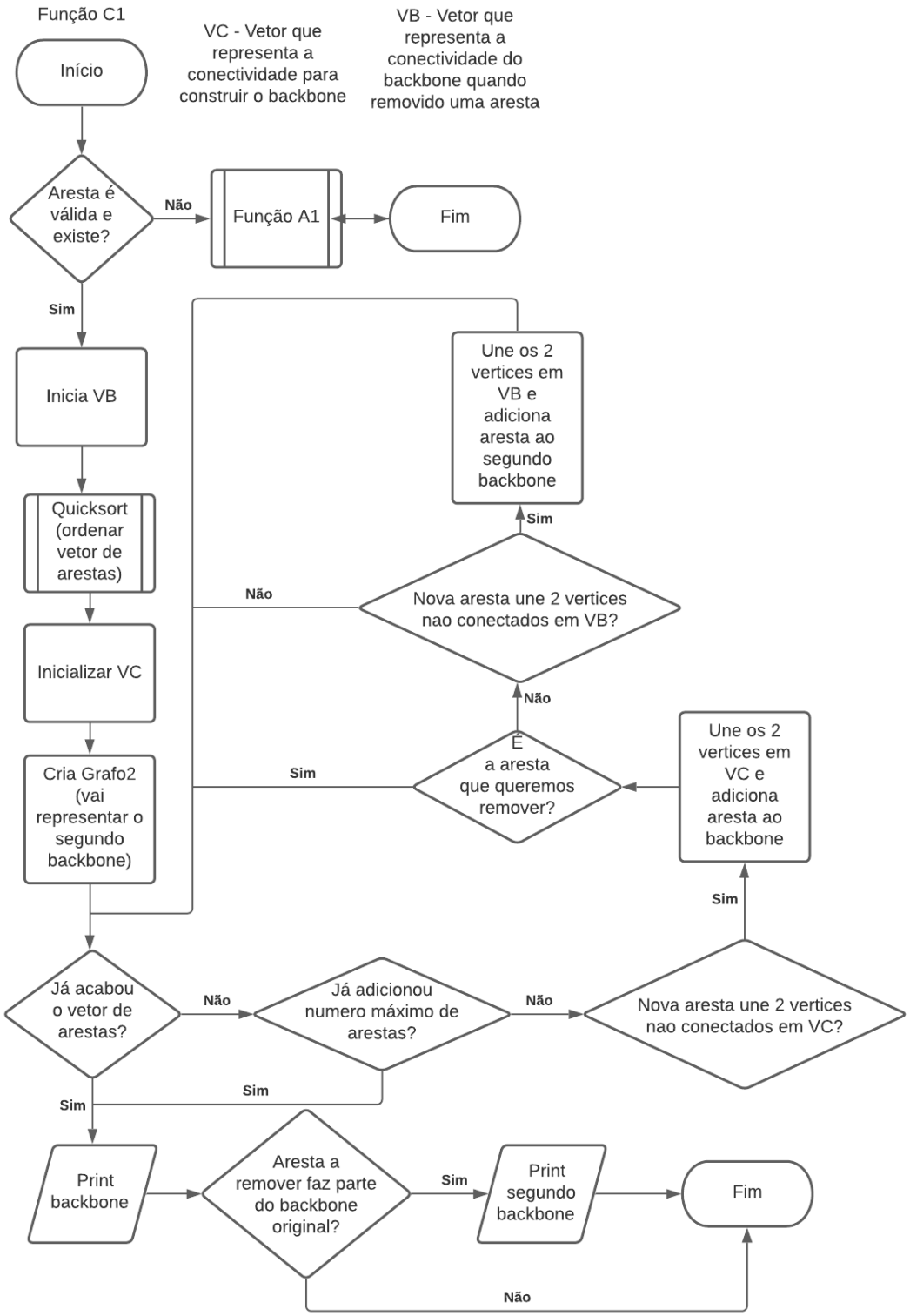
Função C1
Início
VC - Vetor que representa a conectividade para construir o backbone
VB - Vetor que representa a conectividade do backbone quando removido uma aresta
Aresta é válida e existe?
Não
Função A1
Fim
Sim
Inicia VB
Quicksort (ordenar vetor de arestas)
Inicializar VC
Cria Grafo2 (vai representar o segundo backbone)
Une os 2 vertices em VB e adiciona aresta ao segundo backbone
Sim
Não
Nova aresta une 2 vertices nao conectados em VB?
Não
É a aresta que queremos remover?
Sim
Une os 2 vertices em VC e adiciona aresta ao backbone
Sim
Já acabou o vetor de arestas?
Não
Já adicionou numero máximo de arestas?
Não
Nova aresta une 2 vertices nao conectados em VC?
Sim
Sim
Print backbone
Aresta a remover faz parte do backbone original?
Sim
Print segundo backbone
Fim
Não

Figura 5

4.4. Modo -D1

Este modo é em tudo semelhante ao modo -B1 no que diz respeito ao raciocínio utilizado.

Começa-se por verificar se o vértice a remover é valido, se não for evoca-se a *função_A1*. Caso contrário vamos então construir o *backbone* e registando a conectividade do mesmo quando removido o vértice (ou seja, ignorando todas as arestas que o possuem como vértice incidente). Para além disso, regista-se o número de arestas com o vértice que foram inseridas no *backbone* original (*count*).

Por fim, vamos percorrer de novo o vetor de arestas ordenado (desta vez, ignorando as arestas que já foram inseridas no *backbone* e as arestas que contêm o vértice) à procura de uma aresta que una dois vértices não conectados, em caso positivo registamos essa alteração na conectividade do backbone, guardamos a aresta num vetor alocado de tamanho *count* e continuamos o processo até esgotar o vetor de arestas ou até adicionar o número máximo de arestas substitutas (*count*). Finalmente, ordena-se o vetor de arestas substitutas utilizando o *quicksort* (no entanto, desta vez a ordenação não é feita pelo custo das mesmas, mas sim pelo valor dos vértices), dá-se print do *backbone* original encontrado e das arestas.

Por exemplo, vamos considerar o grafo da fig. 6a. À medida que se vai construindo o *backbone* original (fig. 6b), para além do registo da conectividade do algoritmo de *Kruskal,* vai se também registando a conectividade do *backbone* quando se retira um vértice (fig. 6c). Neste caso, como se pode observar, vão existir 4 raízes, desta forma, as arestas que podem restabelecer a conectividade perdida são as que unam 2 vértices associados a raízes distintas, sempre que tal ocorre atualiza-se a conectividade e continua-se o processo. Neste caso em particular as arestas substitutas seriam as arestas a verde na fig. 6d (não tendo sido possível restabelecer totalmente a conectividade original).

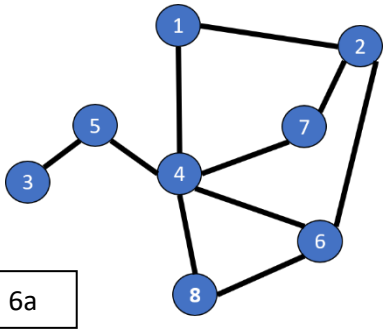

Figura 6a

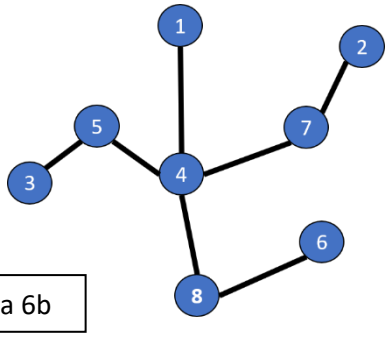

Figura 6b

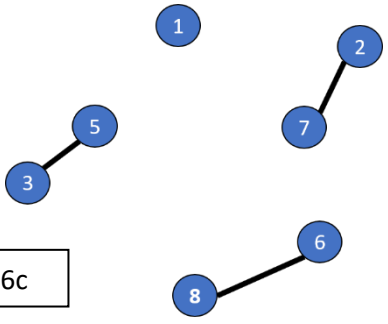

Figura 6c

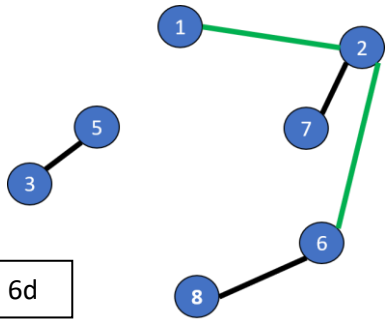

Figura 6d

Na figura 7 pode se analisar o fluxograma da *função_D1*.

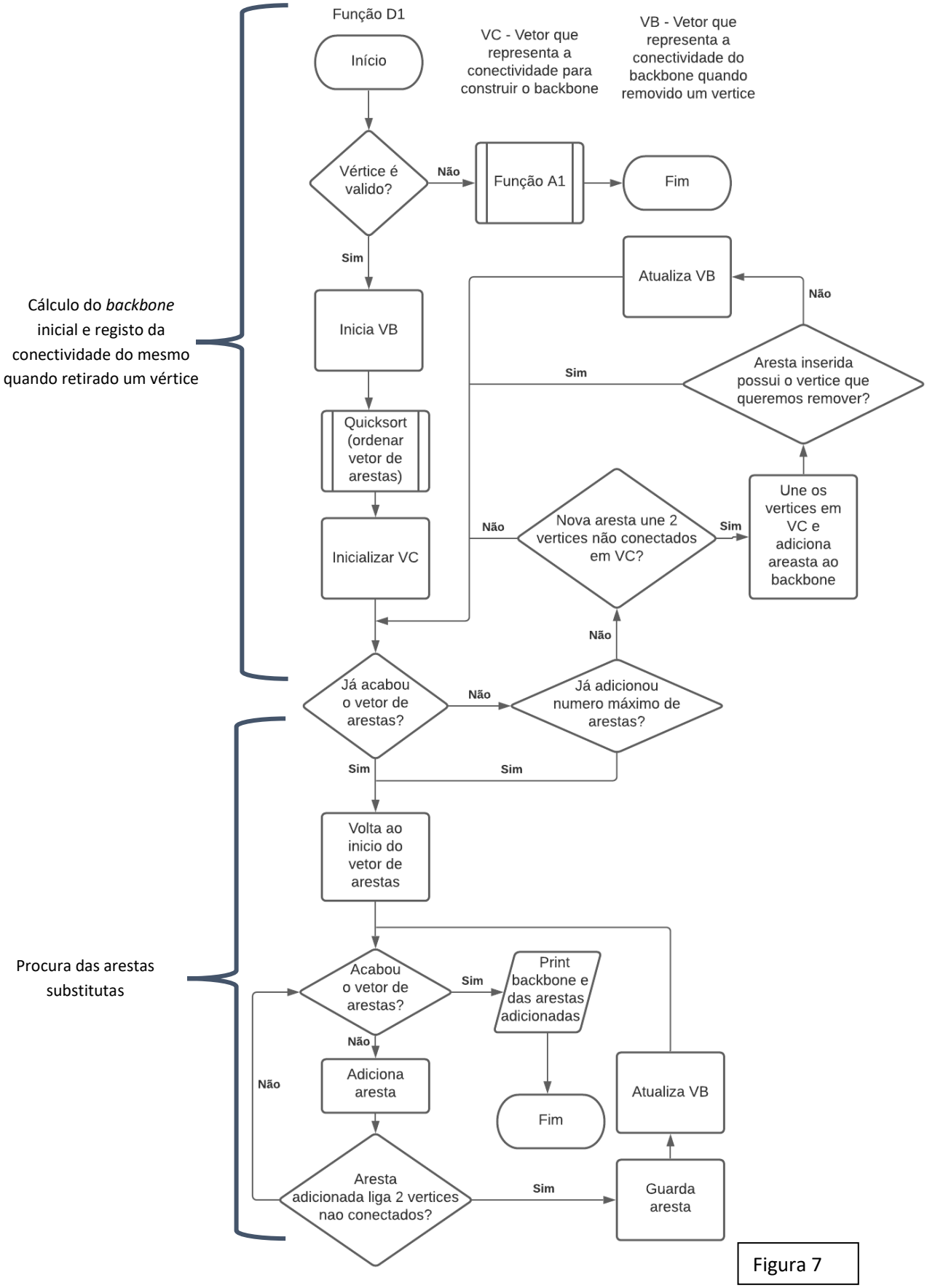

Figura 7

4.5. Modo -E1

Neste modo, começa-se por determinar o *backbone* do grafo original tal como em modo -A1.

Posteriormente, para cada aresta do *backbone* criamos um vetor que represente a conectividade do *backbone* quando removida essa aresta, sendo que o processo a seguir é em tudo semelhante ao modo -B1, ou seja, adiciona-se as arestas que não foram inseridas inicialmente no *backbone* a ver se alguma aresta une 2 vértices não conectados, em caso positivo sabemos que é essa a aresta substituta da removida.

Para conseguirmos representar a conectividade do *backbone* quando retirada uma aresta temos uma função *ResetConnect* que percorre o vetor da conectividade e inicializa de novo os valores do mesmo (id[i]=i e sz[i]=1) que é seguida da função *InitConnect* que percorre todas as arestas do *backbone* original, excluindo a aresta a remover, atualizando para cada uma a conectividade do *backbone* variante.

Na figura 8 pode se analisar o fluxograma da *função_E1.*

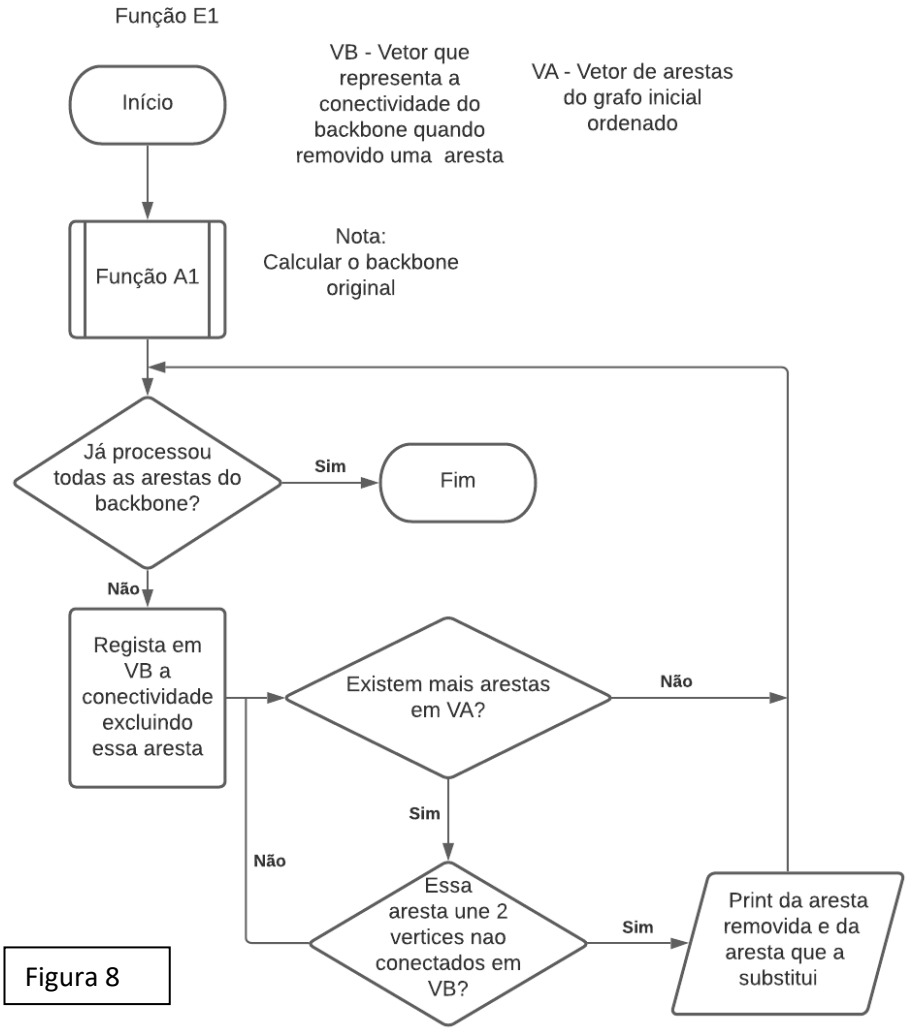

Figura 8

5. Análise dos requisitos computacionais

5.1. Análise Teórica

A seguinte tabela apresenta a complexidade teórica de cada função utilizada. Note-se que V simboliza o número de vértices do grafo original e E o número de arestas do grafo original. Pelo facto de o *backbone* de um grafo ligado ter V-1 arestas (e de um grafo não ligado ter V-x ≈ V), considera-se que o número médio de arestas do mesmo é da ordem de V.

Função	**Complexidade**	**Justificação**
validateHeader	O(1)	Apenas verifica o cabeçalho
openFile	O(1)	Apenas muda o nome e abre o ficheiro de output
Verify	O(1)	Verifica erros de invocação
Le_arestas	O(E)	Lê as arestas do grafo e guarda no vetor de arestas
InitConnect	O(V)	Percorre as arestas do backbone
ResetConnect	O(V)	Percorre vetor da conectividade
NewSorted	O(x) *	Insere vértice ordenado numa lista
GraphInit	O(V)	Aloca a memória para o grafo e inicializa tabela
GraphInsertE	O(x)	Apenas escolhe qual o vértice onde insere a aresta e depois chama *NewSorted*
freegraph	O(V)	Liberta a memória alocada para cada aresta do *backbone*
verifyEdge	O(E)	Verifica se a aresta é válida e existe no grafo inicial
BackboneInit	O(V)	Aloca as tabelas da conectividade e inicializa
freeBackbone	O(1)	Faz-se *free* das tabelas da conectividade
freeArestas	O(E)	Faz-se *free* de cada aresta e do vetor e arestas
printBackbone	O(V)	Faz-se *print* de cada aresta do backbone
LessCusto	O(1)	Compara 2 arestas relativamente ao custo
LessAresta	O(1)	Compara 2 arestas relativamente aos vértices
partition	O(ElogE)	Valor teórico de ordenação
Quicksort		
CWQUinit	O(V)	Inicializa tabelas da conectividade
CWQUfind	O(1)	Altura da árvore tem um valor máximo realístico de 5
CWQUunion	O(1)	Apenas junta as raízes (a compressão de caminho tem custo máximo realístico de 5)

* Nesta função vamos ter de percorrer a lista dos vértices adjacentes a um determinado vértice, onde x é o comprimento da lista (sendo que apenas estariam presentes os vértices maiores, tendo em conta a estrutura de dados utilizada). No entanto apenas é realizada esta função sobre *backbones* ou seja, em média, o número de elementos na lista de cada vértice vai ser significativamente inferior a V. O pior caso acontece quando todos os vértices do *backbone* estiverem ligados ao vértice 1, neste caso a função teria complexidade O(V).

De seguida é analisa a complexidade dos modos principais do projeto, tendo por base a tabela apresentada anteriormente:

- **Modo -A1**

  Por ser o algoritmo de *Kruskal* sem ordenação parcial apresenta uma complexidade **O(ElgE)**.

- **Modo -B1**

  Começa-se por executar o *verifyEdge*, *BackboneInit* e de seguida executa-se o algoritmo de *Kruskal* com o registo da conectividade do *backbone* ignorando uma aresta. Por fim, caso a aresta faça parte do *bacbkone*, percorre-se de novo o vetor de arestas à procura da substituta e dá-se print do *backbone* original, desta forma é expectável a seguinte complexidade:

  O(E) + O(V) + O(ElgE) + O(E) + O(V) ≈ **O(V + ElgE)**

- **Modo -C1**

  Começa-se por executar o *verifyEdge*, *BackboneInit* e de seguida executa-se o algoritmo de *Kruskal* para os 2 *backbones* simultaneamente. Por fim, caso a aresta a remover faça parte do *backbone* dá-se print do *backbone* original e da variante, desta forma é expectável a seguinte complexidade:

  O(E) + O(V) + O(ElgE) + O(V) ≈ **O(V + ElgE)**

- **Modo -D1**

  Começa-se por executar o *BackboneInit* e de seguida executa-se o algoritmo de *Kruskal* com o registo da conectividade do *backbone* ignorando um vértice. Por fim percorre-se de novo o vetor de arestas à procura das arestas substitutas, ordena-se as arestas encontradas (o vetor das arestas substitutas é de modo geral bastante reduzido) e dá-se print do *backbone* original, desta forma é expectável a seguinte complexidade:

  O(V) + O(ElgE) + O(E) + O(V) ≈ **O(V + ElgE)**

- **Modo -E1**

  Começa-se por descobrir o *backbone* com recurso ao algoritmo de *Kruskal* de seguida, para cada aresta do *backbone*, cria um vetor de conectividade para auxiliar na descoberta da sua substituta desta forma é expectável a seguinte complexidade:

  $O(ElgE) + O(V^2) \approx \mathbf{O(V^2 + ElgE)}$

5.2. Análise Experimental

Com recurso a uma variável global que conta os acessos a estruturas de dados (como por exemplo arestas[i] ou id[i]) obteve-se a seguinte tabela de complexidade:

Notação utilizada:

E – Valores experimentais

V – Número de vértices do grafo A – Número de arestas do grafo

Complexidades Teóricas:

A1 – O(ElgE) B1 – O(V + ElgE) C1 – O(V + ElgE)

D1 – O(V + ElgE) E1 - $O(V^2 + ElgE)$

	A1 (E)	B1 (E)	C1 (E)	D1 (E)	E1 (E)
V: 100 A: 100	6 312	8 971	9 246	8 715	186 638
V: 100 A: 500	25 519	28 194	28 422	32 714	279 841
V: 100 A: 1500	80 193	85 443	85 667	99 457	331 846
V: 100 A:2500	134 935	140 677	140 925	165 915	373 825
V: 100 A: 3750	205 488	209 675	209 916	251 316	448 915
V: 500 A: 2500	1 656 318	1 678 459	1 679 569	1 980 015	8 151 896
V: 1000 A: 2500	1 708 969	1 736 552	1 738 796	2 037 913	28 227 917
V: 2000 A: 2500	1 783 471	1 894 154	1 898 476	2 120 589	108 617 239
V: 4000 A: 2500	2 070 642	2 184 732	2 193 524	2 436 660	423 144 677
V: 8000 A: 2500	2 057 181	2 276 283	2 293 426	2 479 892	1 715 092 420

Para verificar a validade dos valores experimentais obtidos, foram feitas regressões lineares em que o eixo das abcissas representa o valor teórico calculado. Por exemplo, para um **O(ElgE)** calculamos **E * log(E).** Ou seja, se o resultado for uma reta linear com um erro desprezável podemos concluir que os modos têm a complexidade teorizada.

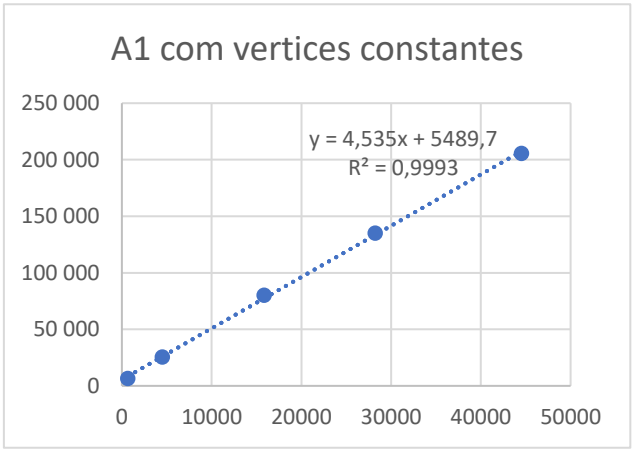

Figura 9

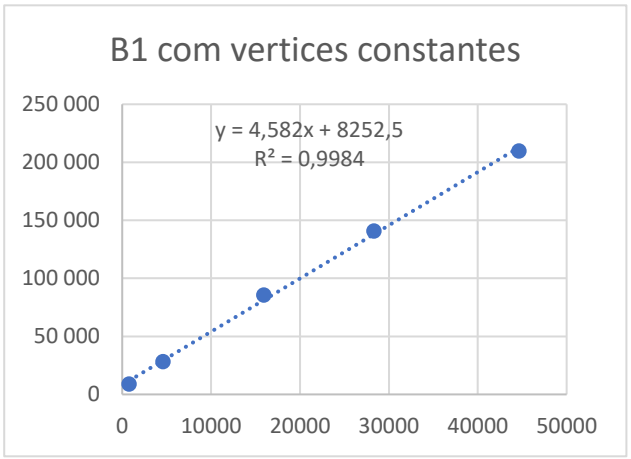

Figura 10

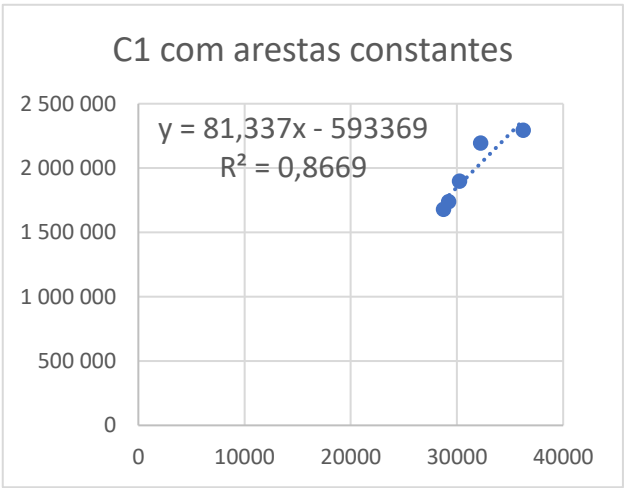

Figura 11

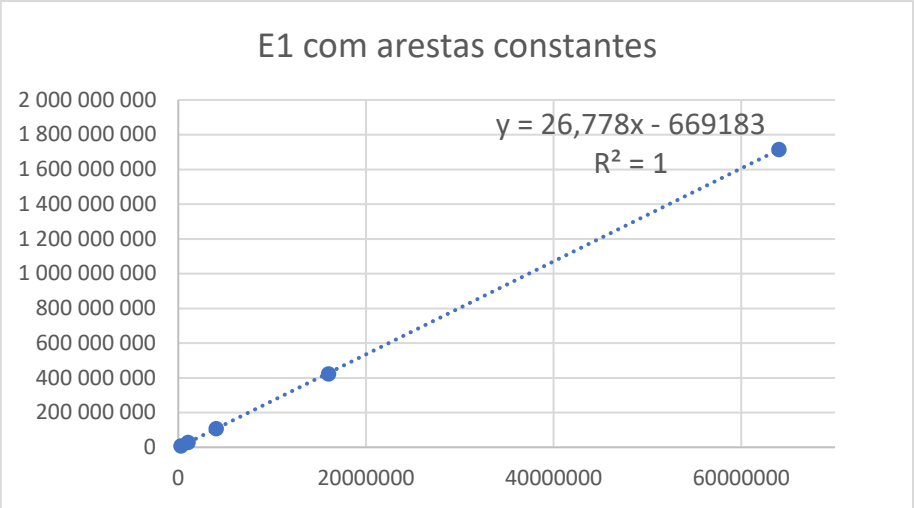

Figura 12

Apenas foram retratados estes gráficos uma vez que para os modos B1, C1 e D1 apresentam todos um gráfico semelhante devido à sua igual complexidade teórica. Para além disso, ao observar-mos os dados experimentais do modo A1 a arestas constantes verificamos que apresentam valores relativamente próximos entre eles, o que seria expectável pois a complexidade teórica não depende de V (O(ElgE)).

Podemos visualizar também que o modo E1 foi o que apresentou dados com ordem de grandeza bastante superior aos restantes modos, especialmente quando se aumentou o número de vértices, o que seria expectável tendo em conta a sua complexidade teórica de $O(V^2 + ElgE)$.

Em suma, pela análise dos dados experimentais e dos gráficos podemos concluir que a complexidade teórica deduzida se encontra correta.

6. Exemplo Prático

Tomando como exemplo o e ficheiro de entrada da fig.13 o *backbone* gerado pelo algoritmo de *Kruskal* seria o da fig 14:

```
8 10 B1 4 7
1 2 20.5
1 4 9.22
2 6 11.7
2 7 3.68
3 5 4.0
4 5 3.96
4 6 5.3
4 7 5.2
4 8 4.8
6 8 5.1
```

Figura 13

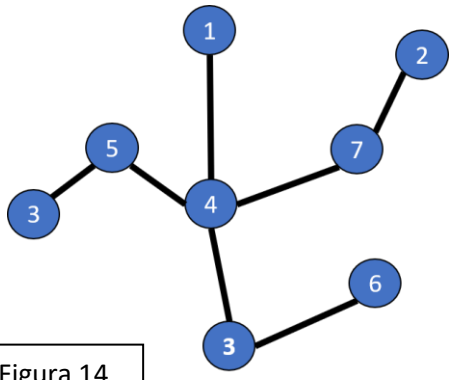

Figura 14

Para este exemplo, o vetor da conectividade do *backbone* quando ignorada a aresta 4-7 tem o aspeto da fig. 15:

	0	1	2	3	4	5	6	7
Id	3	1	3	3	3	3	1	3
Size	1	2	1	6	1	1	1	1

Figura 15

Ou seja, tendo em conta que os vértices no vetor estão numerados de 0 a V-1, podemos observar no vetor de id que existem 2 raízes distintas, a raiz 2 que tem também o vértice 7 e a raiz 4 que tem os restantes vértices. A informação está coerente com o vetor de *Size* uma vez que o vértice 4 apresenta 6 vértices ligados a ele (incluindo ele mesmo), o vértice 2 possui 2 vértices e os restantes apenas possuem a eles mesmos.

Portanto as arestas da fig.16 são as candidatas a arestas substitutas, pois foram as que não foram inseridas no *backbone* estando também ordenadas por custo devido ao algoritmo de *Kruskal*. Sendo assim o processo de procura seria realizado da seguinte forma:

```
4 6 5.3
2 6 11.7
1 2 20.5
```

Figura 16

1º - Testa aresta 4-6: Rejeitada porque os vértices 4 e 6 têm ambos a mesma raiz (3)

2º - Testa aresta 2-6: Aceite porque os vértices 2 e 6 têm raízes diferentes (1 e 3) logo é a aresta que repõe a conectividade original

7. Bibliografia

- Acetatos da Unidade Curricular de Algoritmos e Estruturas de Dados