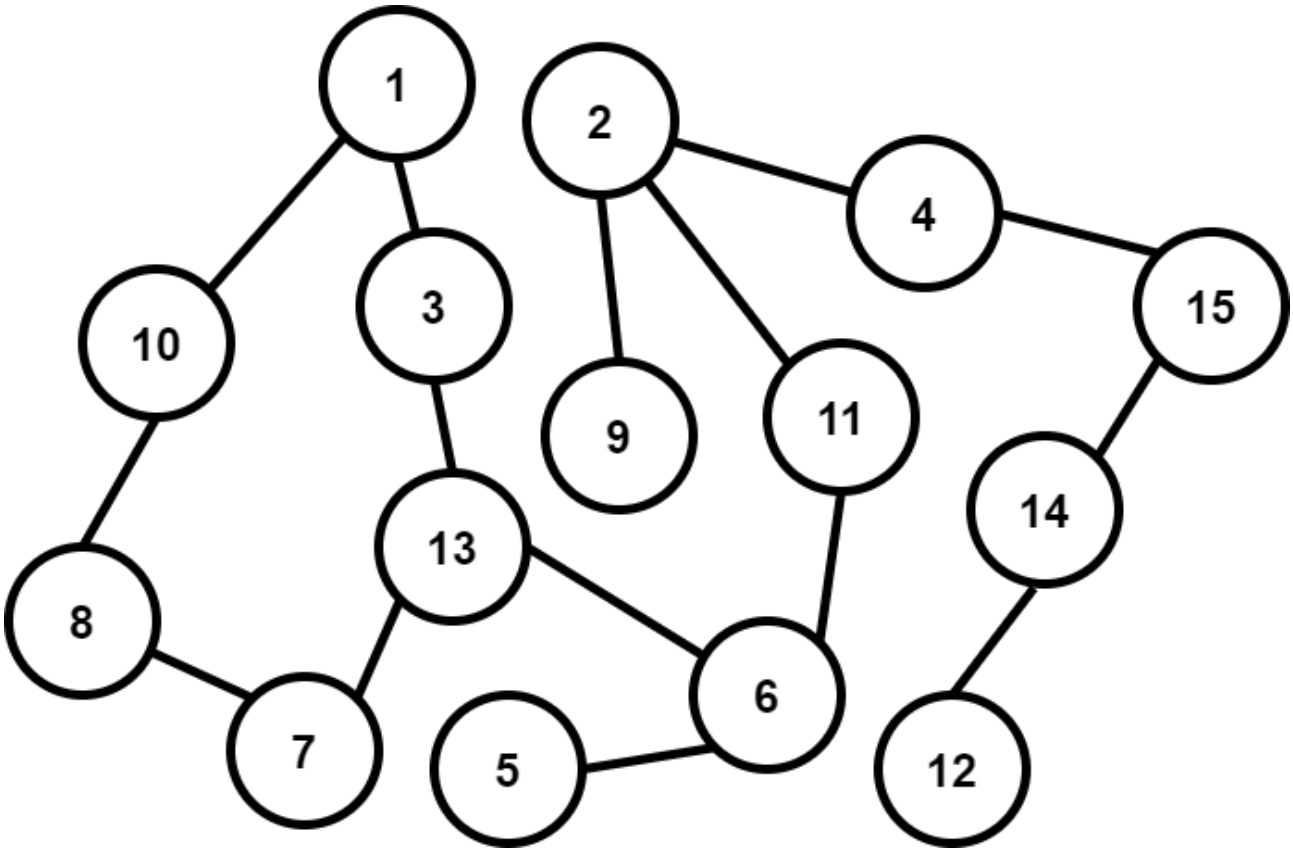
Mestrado Integrado em Engenharia Eletrótecnica e de Computadores
Instituto Superior Técnico
1ºSemestre 2020
Algoritmos e Estruturas de Dados

Projeto AirRoutes Grupo 32



António Vidais (96162) e Tiago Leite (96332)
Professor Carlos Bispo

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 pertende obter. Terminada a execução, o programa produz um segundo ficheiro de texto que contêm 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 esparço)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 questâo. As formas de representação do grafo são definidas com maior detalhe na secção4.6.

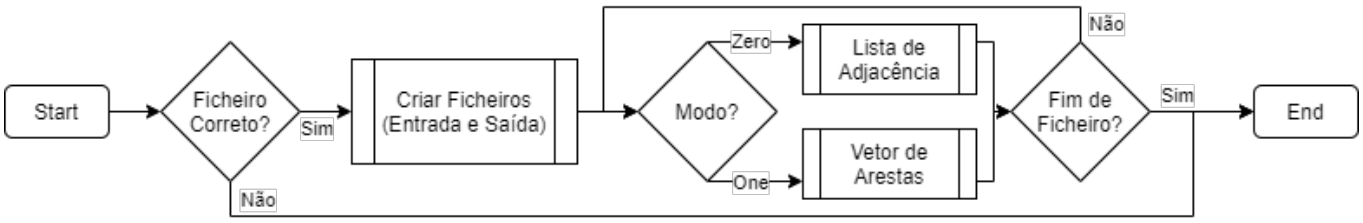


Figura 1: Fluxograma Representativo da execução da função main

4 Estruturas e Tipos de Dados

4.1 Argumentos do Problema

A estrutura do tipo PBArg é usada para guardar os argumentos do problema. È 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 Krukal seria necessário 'traduzir' de novo os dados para a forma de vetor de arestas pelo que estariamos a 'gastar' tempo em conversões desnecessárias.

5 Algoritmos

5.1 Algoritmos para encontrar a Árvore Minima de Suporte

Para encontrar a árvore mínima de suporte de uma 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 hipotése de cada vértice estar ligado a todos os outros. A sua otimizição 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 Connectividade (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 constituidos 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 téorico para a complexidade deste problema é definido pelo Algoritmo de *Kruskal*, cuja complexidade é $O(E * logE)$. Devido ao problema discutido na secção 5.1.1, o Algoritmo de *Prim* tem complexidade díficil de determinar, e díficil também de aproximar do limiar teórico da complexidade establecido pelo algoritmo de *Kruskal*. Por isso, complexidade do algoritmo de *Prim* é, no melhor caso, $O(E * logV)$.

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 á 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.

5.3 Análise da Complexidade Geral do Programa

A complexidade do Programa depende da variante selecionada.No entanto as operações que são as mesmas em todas as variantes:

Operação	Complexidade
Ler o Grafo	$O(E)$
Kruskal	$O(E * log(E))$
Ordenação	$O(E * log(E))$
CWQU	$O(E * log(E))$

Repor Conectividade	Complexidade
Inicializar	$O(E)$
Repor	$O(E * log(E))$
Total	$O(E * log(E))$

Podem ser selecionadas as seguintes variantes e a sua complexidade:

Variante	Complexidade
A1	$O(E) + 2 * O(E * log(E))$
B1	$O(E * log(E))$
C1	$O(E * log(E))$
D1	$O(E * log(E))$
E1	$O((E - X) * E * log(E)) \approx O((E^2) * E * log(E))$

Máximo	$O(E * log(E))$

Esta complexidade é o resultado da realização das seguintes operações:

- A1 : Ler o grafo,Algoritmo de *Kruskal* e ordenar;
- B1 e C1 : Ler o grafo, Algoritmo de *Kruskal*, repor a conectividade e ordenar;
- D1 : Ler o grafo, Algoritmo de *Kruskal*, repor a conectividade para as arestas perdidas e ordenar;
- E1 : Ler o grafo, Algoritmo de *Kruskal*, repor a conectividade para todas as arestas e ordenar;

6 Funções Principais

Todas as variantes do programa backbone operam sobre um conjunto de funções comuns, pelo que são estas as funções mais utilizadas em todo o programa:

6.1 Algoritmo de *kruskal*

A função Kruskal implementa o Algoritmo de *Kruskal* para encontrar a árvore mínima de suporte na sua totalidade. Discutir esta função é na verdade discutir o algoritmo de *Kruskal*. A função tem como parâmetros de entrada :

- g : Grafo ao qual aplicar o algoritmo;
- cost : Custo total, passado por referência para poder ser alterado na scope desta função;

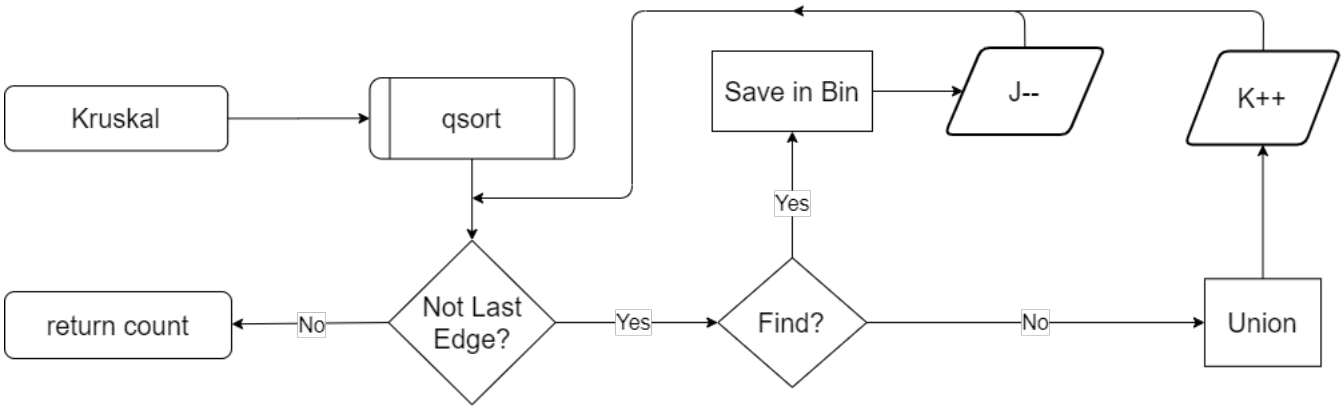


Figura 2: Representação do funcionamento da função Kruskal

A função começa por inicializar os vetores auxiliares à CWQU e ordenar o grafo segundo o critério definido na função lessCost. De seguida corre todas as arestas, e verifica se existe conectividade entre os vértices da mesma. Como o grafo está neste momento organizado por custo, a primeira aresta que encontrar que ligue 2 vértices é a mais barata. Para se obter o tamanho da árvore de suporte, o programa conta o número de uniões realizadas, i. e. quantas arestas com vértices ainda não conectados são encontrados. Quando o programa encontra arestas que não pertençam à árvore mínima de suporte, ele guarda-as num vetor auxiliar, ao qual chamou-se 'data'. Como não se sabe à partida o tamanho quantas arestas pretencem à MST, as que pertencem são guardadas no ínicio do vetor 'data' e as que não pretencem são guardadas a partir do final. Como o vetor 'data' têm tamanho E, no final todas as suas posições estão preenchidas e não há nenhuma livre, como se mostra na figura 3. Quando este vetor estiver terminado

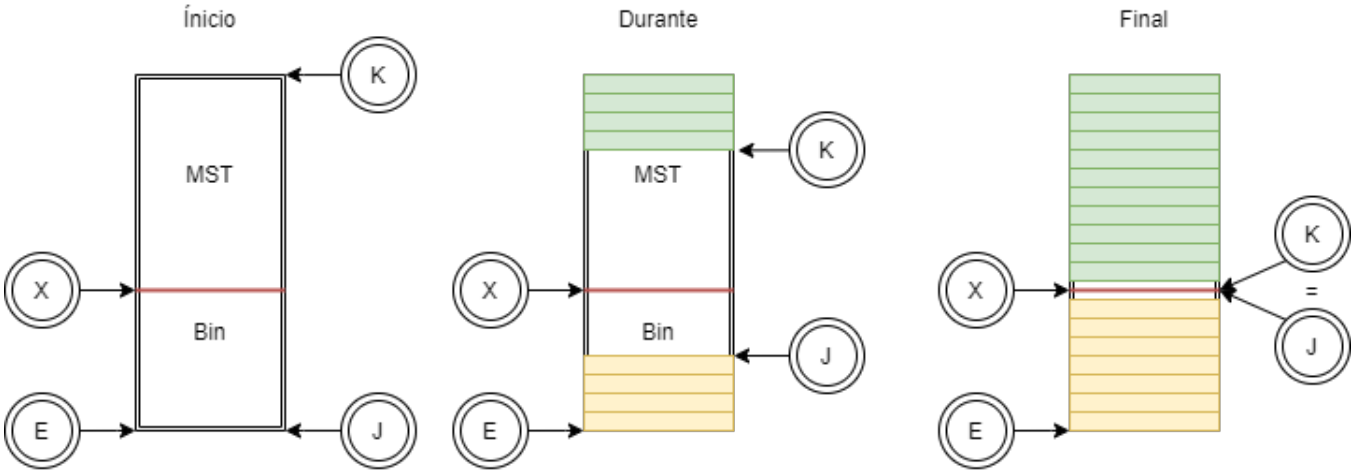


Figura 3: Representação do preenchimento do vetor auxiliar 'data'

No nosso programa existe uma versão do algoritmo de *Kruskal* chamada CWQU, cuja função é a mesma que a função Kruskal, mas a primeira sai assim que encontra uma união. Esta função é utilizada para repor a conectividade.

6.2 Weighted Quick Union com compressão de caminho

Este capítulo fala sobre as funções que implementam o algoritmo de CWQU. Estas não devem ser confundidas com a função CWQU, que implementa uma versão modificada do algoritmo de *Kruskal*. As funções que implementam a CWQU são :

6.2.1 Incialização

A função UFinit inicializa os vetores auxiliares à CWQU. Cada posição do vetor id deve ser inicializada com o valor da posição, e cada posição do vetor sz deve ser inicializada com o índice 1 Esta função recebe como parâmetros de entrada os 2 vetores que vai inicializar e o número de elementos que neles existem

6.2.2 Encontrar uma ligação já existente

A função UFfind determina se já existe conectividade entre 2 vértices. Isto é conseguido através da análise da raiz de cada conjunto. Os Parâmetros de entrada desta função são os 2 vértices e o vetor id. A função retorna 1 se os vértices estiverem conectados, e 0 se não estiverem.

6.2.3 Criar uma nova ligação

A função UFunion cria uma nova ligação entre 2 vértices. No contexto de encontrar a árvore mínima de suporte, esta função só deve ser chamada depois de verificado que não existem já ligações entre este 2 vértices. A nova ligação é conseguida através da alteração da raíz do vértice cujo peso (sz) é mais pequeno, alterando também todos os vértices (filhos) que pretençam ao mesmo conjunto. Após isto, é realizada a compressão de caminho.

6.3 Marcação de arestas como removidas

A função SearchDelete é muito simples, mas o seu papel é fundamental na reposição da conectividade do grafo. Esta função marca a(s) aresta(s) que não podem ser utilizadas na nova árvore, através do processo demonstrado na figura 6.3. A marcação é feita passando o custo da aresta para o seu simétrico. Como todas as arestas são verificadas quando são lidas do ficheiro de entrada, não há risco de uma aresta ser negativa à partida. Com as arestas marcadas, o programa apenas precisa de repor a conectividade sem contar com arestas cujo custo seja negativo. Esta função tem como parâmetros de entrada:

- g : Grafo onde marcar as arestas;
- start end : Inicio e Fim da pesquisa. Em algumas situações não há necessidade de procurar as arestas em todo o grafo;
- (*Delete) : Apontador para função que define o critério através do qual uma aresta é eliminada ou não. Depende da Variante em que a função é chamada;

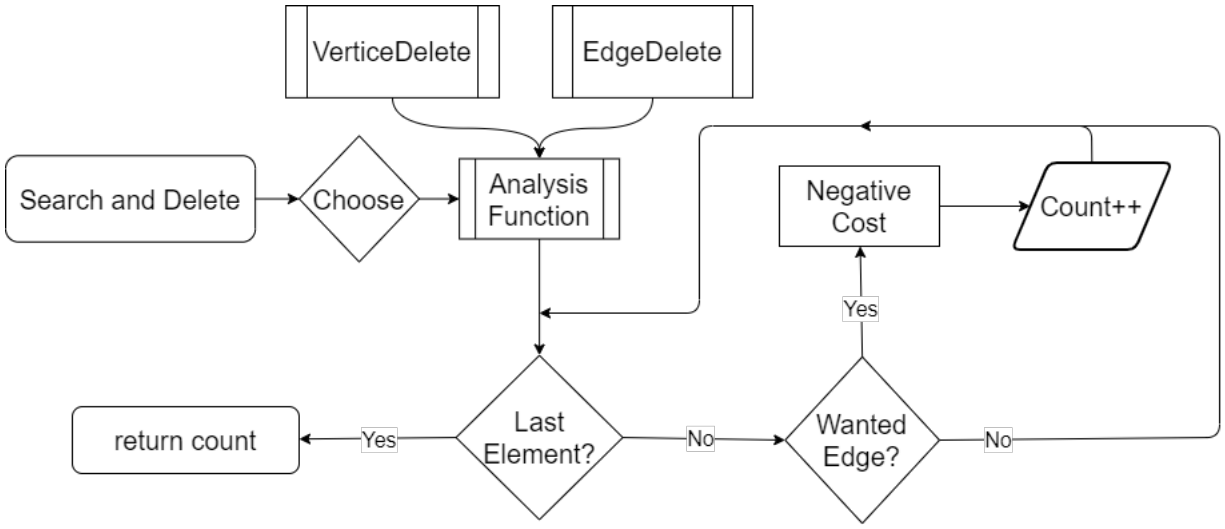


Figura 4: Fluxograma Representativo da execução da função SearchDelete

6.4 Reposição da Conectividade

A função binsearch corre o grafo em busca de uma aresta que reponha a conectividade do grafo. Para ser corretamente implementada, deve-se começar por ter os vetores auxiliares de CWQU (id e sz) no estado a partir do qual queremos procurar a nova aresta. Esses vetores são passados como argumentos à função binsearch, que corre a CWQU até encontrar uma Union, i. e. encontrar uma aresta que repõe a conectividade. Esta função pode ter que ser chamada várias vezes, uma por cada aresta que tenha sido eliminada. No entanto, saber quantas arestas foram eliminadas é muito simples, uma vez que é o return da função SearchDelete Os parâmetros de entrada desta função são:

- data : Vetor de arestas que contem a representação do grafo;
- id e sz : Vetores auxiliares para a CWQU;

- start e end : Posições de inicio e fim da procura;

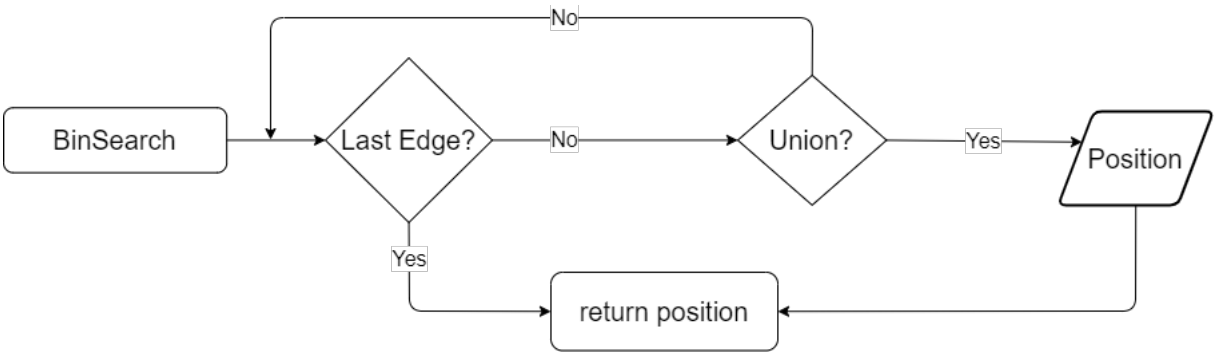


Figura 5: Representação do funcionamento da função binsearch