Relatório 2º Projeto **Análise e Síntese de Algoritmos**

Grupo: 15 2º Semestre 2016/2017

Membros: Catarina Coelho, nº84706
Pedro Dias, nº84752

Introdução

Este projeto consiste no desenvolvimento de um programa, que tem como objetivo decidir para um dado conjunto de cidades, quais as estradas e aeroportos a construir entre as mesmas de forma a que todas as cidades fiquem ligadas em rede e minimizando o custo total das obras. No caso de haver soluções com o mesmo custo total, o programa deve optar pela solução que minimiza o número de aeroportos a construir.

Como input recebe o número total de cidades; o número de potenciais aeroportos a serem construídos, e o custo de construção desses aeroportos nas respetivas cidades; e o número de potenciais estradas entre duas cidades, e o custo das mesmas.

O output deverá retornar o custo total bem como o número de aeroportos e estradas a construir, caso não seja possível construir a rede o output deve consistir em mandar a mensagem “Insuficiente” para o stdout.

Como referência foram utilizados os slides da cadeira, bem como slides da cadeira de IAED, informação disponibilizada nas aulas teóricas e práticas, e os seguintes links:

- https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

Descrição da Solução

Para implementação da solução foi escolhida como representação do problema um grafo não dirigido, em que os vértices correspondem às cidades, e os arcos entre eles correspondem às possíveis ligações entre cidades, que seja através de estradas ou aeroportos. A cada ligação é atribuída um custo que representa o custo de construir uma estrada ou aeroporto entre duas cidades, de forma a liga-las.

Como representação do grafo escolhemos um vetor de ponteiros para estruturas, as estruturas representam a ligação entre duas cidades e o custo da respetiva ligação. À medida que o programa lê a informação das ligações do stdin, a informação é guardada numa estrutura e inserida num vetor de tamanho (Nº de potenciais aeroportos + Nº de potenciais estradas).

Cada ligação via terrestre, isto é, através de estrada, é guardada na estrutura de dados utilizando os índices que identificam as cidades ligadas pela estrada e o respetivo custo. No caso de a ligação ser feita via aérea, isto é, através de aeroportos, é guardada numa estrutura a possível ligação desse aeroporto a um vértice adicional fictício, em que o custo associado a esta ligação corresponde ao custo de construção desse aeroporto na respetiva cidade.

Adicionalmente foi criado um novo vetor apenas com as ligações através de estradas. Uma vez que as estruturas de dados referentes às ligações via terrestre e/ou aérea se mantêm inalteradas durante toda a execução do programa, só há necessidade de as criar apenas uma vez, de maneira a que apenas as referências para as estruturas referentes a ligações através de estradas foram copiadas para este novo vetor.

Após armazenar toda a informação o programa executa o algoritmo de Kruskal sobre o vetor das ligações via terrestre. Se o número de potenciais aeroportos

for inferior a dois, a informação relativa aos aeroportos é desprezada, e após executar o algoritmo, o programa envia para o stdout o número total e o custo total de todas as estradas que foram necessárias construir.

Neste caso o número de aeroportos a construir será, por definição, zero. No caso de o número de potenciais aeroportos ser igual ou superior a dois, significa que há a possibilidade de haver caminhos em que o custo total seja inferior ao verificado na primeira execução do algoritmo. Pelo que, executamos o algoritmo de Kruskal uma segunda vez, no vetor que contém todas as ligações via aérea e terrestre e comparamos os resultados obtidos entre as duas execuções do algoritmo. Se o custo total for igual nas duas execuções damos prioridade à solução que irá minimizar o número de aeroportos a construir, pelo que iremos enviar para o stdout o custo total, nº estradas e nº aeroportos referentes à primeira execução do algoritmo. Caso contrário envia para o stdout os dados referentes à segunda execução do algoritmo.

Em ambas as execuções do algoritmo é verificada a possibilidade/impossibilidade de construção da rede, com um auxílio a uma flag. Isto é, no caso de no final da execução do algoritmo de Kruskal haver algum vértice que não tenha ligação para nenhum dos outros, esta flag é ativada. Desta forma podemos garantir a existência de ligação entre todos os vértices. Nos casos em que esta condição não seja satisfeita, apenas é enviado para o stdout a palavra “Insuficiente”.

A execução do programa termina com a libertação de toda a memória alocada durante a execução do programa.

Análise Teórica

Vetor de ponteiros para estruturas

Considerando um grafo não dirigido G = (V, E), como se verifica na implementação escolhida, vetor das ligações terá um tamanho de |E|, sendo E o número total de ligações existentes no vetor. No caso do vetor que armazena todas as ligações por via terrestre e aérea terá um tamanho de nº de potenciais aeroportos + nº de potenciais estradas a construir. Já o vetor que apenas contém as ligações por via terrestre terá um tamanho de nº de potenciais estradas a construir.

Em relação à libertação de memória alocada durante a execução do programa, verificamos que, a complexidade de desalocar todas as estruturas de dados criadas para as ligações existentes no grafo é de O(E), sendo E o nº total de potenciais aeroportos + potenciais estradas a construir, e a complexidade de desalocar cada um dos vetores criados é O(1). Contabilizando uma complexidade total de O(E).

Algoritmo de ordenação QuickSort

O algoritmo QuickSort é um algoritmo de ordenação que dado um vetor, divide-o em várias partes e ordena as várias partes independentes de forma recursiva. Ou seja, é uma técnica de “dividir para conquistar”, os dados menores são divididos para um lado e os maiores para outro, de seguida o algoritmo é aplicado recursivamente a cada uma das partes separadas por uma posição, o pivot, de forma a que no fim da execução do algoritmo o vetor esteja ordenado.

Para tal dispomos de dois iteradores, i e j, por exemplo, o i à esquerda e o j à direita (antes do pivot). O i avança até encontrar um elemento maior que o pivot,

ou seja, que deve estar do lado direito do vetor, e o j avança até encontrar um elemento menor que o pivot, ou seja, que deve estar do lado esquerdo do vetor. De seguida trocamos o conteúdo na posição i com o conteúdo da posição j, e repetimos estes passos até que i e j se cruzem. Nessa altura trocamos o pivot com o conteúdo na posição i do vetor, posicionando a divisão entre o "lado esquerdo" e o "lado direito" do vetor.

A escolha deste algoritmo foi baseada na sua eficiência e pela sua facilidade e simplicidade de implementação, dado que a biblioteca C fornece uma concretização para este algoritmo com nome qsort(). A comparação feita para ordenar o vetor consiste em comparar os custos das ligações existentes entre os vértices, de forma a ordenar o vetor com base nos custos das ligações.

Em relação à sua complexidade:

- No pior caso: vetor já ordenado, $O(N^2)$:
  - Cerca de $N^2/2$ comparações;
  - Se o vetor já estiver ordenado, todas as partições degeneram e a função chama-se a si própria N vezes;
  - O número de comparações é $(N + 1)N / 2$;
  - O mesmo acontece quando o vetor está ordenado pela ordem inversa;
- No melhor caso: quando cada partição divide o vetor de entrada em duas metades iguais, O(N lg N), em média, para ordenar N objetos:
  - Nº de comparações usadas por QuickSort satisfaz a recursão de dividir para conquistar: $C_N \approx N \lg N$

Algoritmo Kruskal

O algoritmo de Kruskal é um algoritmo que tem como objetivo encontrar árvores abrangentes de menor custo para um grafo com pesos. Este algoritmo consiste em encontrar um subconjunto de arcos que formam uma árvore que inclui todos os vértices do grafo, onde o peso total, dado pela soma do peso dos arcos da árvore, é mínimo.

A sua execução processa-se da seguinte forma: o algoritmo mantém uma floresta A (conjunto de árvores), onde cada conjunto representa uma sub-árvore de uma MST. Aquando da criação dessa floresta para cada vértice é criado uma árvore separada, utilizando a função Make-Set(); de seguida é ordenado o vetor, dado como input, que contém todos os arcos do grafo por ordem crescente de peso e posteriormente percorrido. À medida que este vetor é percorrido até ao final, são removidos todos os arcos de menor peso, caso sejam arcos que liguem árvores diferentes (para verificar esta condição é utilizada a função Find-Set()), são adicionados ao conjunto A (função Union()), juntando duas árvores numa única árvore; caso contrário desprezam-se esses arcos.

No fim da execução do algoritmo obtemos um conjunto com apenas uma componente que forma uma árvore abrangente de menor custo.

A escolha deste algoritmo foi feita com base na sua eficiência e com base na sua utilidade para encontrar árvores abrangentes de menor custo de um grafo, que no caso do problema dado nos iria fornecer a solução do problema.

Em relação à sua complexidade, para um grafo G = (V, E):

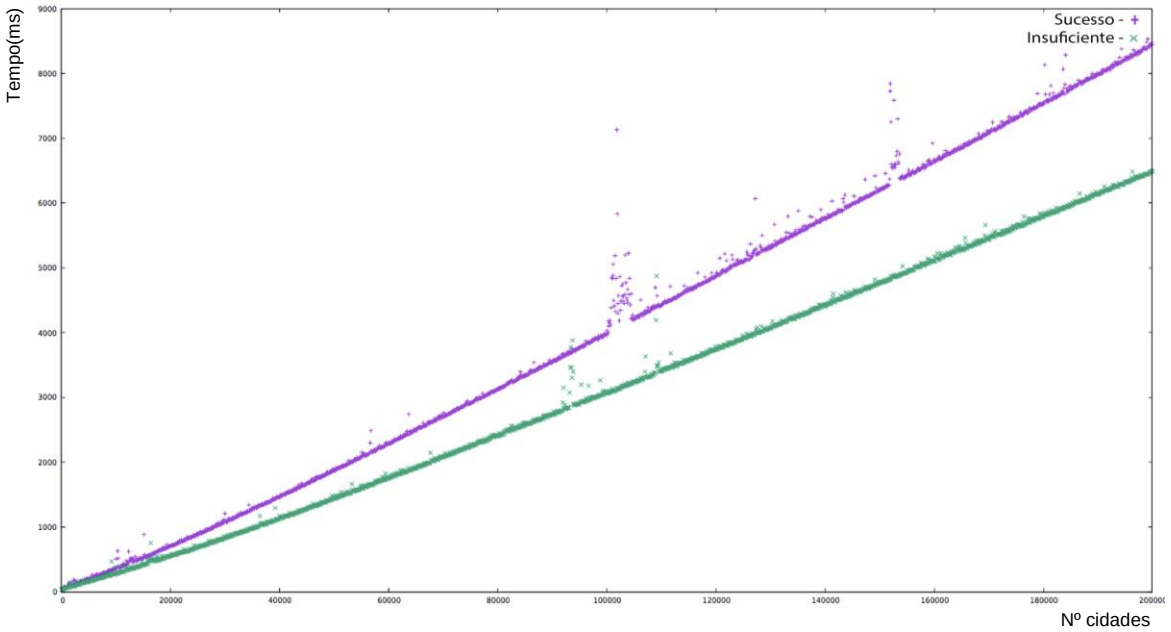
- Inicialização: O(E lg E) devido à ordenação dos arcos;
- Operações sobre os conjuntos disjuntos:

- O(V) operações de Make-Set;
- O(E) operações de Find-Set e Union;
- Com estruturas de dados adequadas (árvores com compressão de caminhos e união por categorias) para conjuntos disjuntos é possível estabelecer que O((V+E)α(E,V));
- Como |E| ≥V−1 porque o grafo é ligado, então temos O(Eα(E,V));

- Logo, é possível assegurar que O(E lg E);
- Dado que E < V^2, obtém-se também O (V lg E);

Avaliação Experimental dos Resultados

De forma a obter uma análise experimental com base na resolução apresentada, foram gerados cerca de 2000 testes para cada um dos casos em estudo (Sucesso e Insuficiente) com uma quantidade crescente de input (E).



Através do gráfico obtido é facilmente observável que à medida que o input aumenta, o tempo que é necessário para tratar qualquer um dos casos irá aumentar de forma tendencialmente linear. Podemos constatar que existem alguns pontos que se encontram um pouco dispersos, em relação à maioria dos restantes, contudo, estes podem ser desprezados tendo em conta a quantidade de dados analisados.

Em relação à disparidade entre os casos de “Sucesso” e “Insuficiente”, podemos concluir que o caso que tem uma execução mais eficiente é o “Insuficiente”. Isto verifica-se porque o programa apenas necessita de verificar se está a visitar um vértice que não tem nenhuma ligação para nenhum dos outros vértices, nesse caso pode logo concluir que é um caso de insuficiência e o programa termina a sua execução. Enquanto que na hipótese de ser “Sucesso” o programa tem de analisar todos os vértices e só depois faz as suas conclusões.