Trabalho Prático 1

Projeto e Análise de Algoritmos

Davi Gomes e Ian Nunes

6 de abril de 2024

1 Introdução

Esse relatório apresenta uma das possíveis soluções para o problema de encontrar os k menores caminhos dentro de um grafo. O problema foi modelado como uma rede de caminhos que conectam dois reinos de uma terra mágica. No mundo real, esse problema tem diversas aplicabilidades, principalmente em contextos em que se quer reduzir os custos em relação a transporte, como sistemas de navegação, rotas de entregas, entre outros.

Partindo dessa premissa, o objetivo principal deste trabalho consiste em avaliar a implementação de um algoritmo capaz de encontrar esses k menores caminhos entre dois vértices de um grafo ponderado. Para isso, implementamos o algoritmo de Eppstein. No decorrer desse documento, iremos abordar como foi realizada essa implementação e a complexidade das suas rotinas.

2 O Problema

O problema consiste em encontrar os k menores caminhos mágicos entre as cidades Mysthollow e Luminae dentro do reino de Xulambis. Para encontrar esses menores caminhos representamos o mapa entre as cidades em um grafo e utilizamos o algoritmo proposto por David Eppstein em 1997.

Esse algoritmo é uma adaptação do algoritmo de Dijkstra. O algoritmo de Eppstein funciona por meio de uma fila de prioridade, essa fila irá armazenar os vértices e os seus respectivos pesos em relação ao nó de origem. Para isso, o algoritmo deve começar com a fila de prioridade vazia e em seguida inserir nela o primeiro vértice no caminho, ou seja, o vértice de origem. Em seguida é iniciado um laço de repetição que terá fim quando o vértice de destino for retirado da fila k vezes. Para termos um controle de quantas vezes cada nó foi retirado da fila, devemos ter uma estrutura que armazena um contador para eles. Dentro da estrutura de repetição principal devemos verificar se a fila está vazia, para o caso das entradas pedirem um k maior do que existem de caminhos, nessa situação, devemos sair do loop. Depois disso, extraímos o vértice com menor peso e verificamos se ele já foi retirado k vezes por meio da análise do seu contador, se essa análise confirmar que já foi retirado k vezes, o algoritmo voltará para o início do laço. Após a verificação anterior, iremos incrementar do vértice que saiu da fila e verificar se ele é igual ao nó de destino, se for iremos ter descoberto um caminho e devemos o armazenar, em seguida vamos aumentar o contador do vértice de destino e voltar para o início do loop. No caso de não ser igual iremos inserir os nós adjacentes ao vértice, que estamos analisando, na fila de prioridade e voltaremos ao início da estrutura de repetição. Ao sairmos do laço, teremos então, o valor dos k menores caminhos armazenados, dessa forma resolvendo o problema inicial.

3 Implementação

A estrutura de dados escolhida para representar o grafo nesse problema é uma lista de adjacência encadeada. Essa escolha se dá pela restrição de entrada definida:

$$2 \leq \mathbf{n} \leq 10^5$$
$$1 \leq \mathbf{m} \leq 2 \cdot 10^5$$

No qual **n** é o número de vertíces e **m** o número de arestas. Desta forma, trata-se de um grafo esparso, uma vez que o valor de |m| é muito menor que $|n^2|$.

Além disso, também optamos por implementar o algoritmo de Eppstein, uma vez que ele é capaz de resolver o problema dos k menores caminhos para grafos com ciclos e loops, com uma complexidade de $f(ka(log(ka)))$.

Para a implementação do algoritmo de Eppstein optamos por utilizar uma fila de prioridade, implementada por meio de uma *Min-Heap*, essa estrutura foi utilizada para termos um menor custo com relação a encontrar o menor elemento, uma atividade recorrente no algoritmo que encontra os k menores caminhos. Essa complexidade é $O(1)$, porém, para se ter o elemento de menor peso, devemos sempre organizar a fila depois de cada inserção e remoção e as complexidades dessas funções são $O(log(v))$, sendo v o número de vértices.

3.1 Lista de Adjacência Encadeada

A lista de adjacência é a estrutura base para o desenvolvimento da solução do problema. Por meio dela, é possível representar de forma eficiente a rede de conexões entre os vértices e armazenar os respectivos pesos das arestas.

A representação do *struct* grafo se dá por um vetor de listas de adjacência associada a cada vértice individual com seus vizinhos, acessados através de ponteiros para os próximos elementos àquele vértice.

Sua implementação contempla métodos de inserção e remoção de arestas, validação dos elementos, pesquisa e leitura. Funções fundamentais para toda a execução do programa.

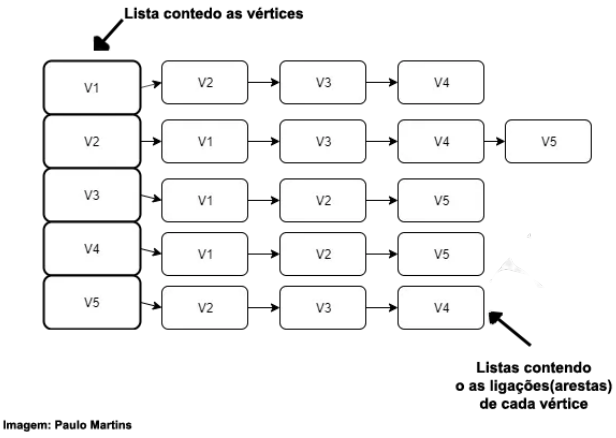

Figura 1: Lista de adjacência encadeada.

Função	Complexidade
inicializaGrafo	O(v)
insereAresta	O(1)
removeAresta	O(v)
leGrafo	O(a)
liberaGrafo	O(v+a)

Figura 2: Ordens de Complexidade da Lista

Durante a inicialização do grafo, para alocar seu espaço na memória, é realizada a chamada da função *calloc()* para reservar espaço para a lista de adjacência de cada vértice, dominando assintóticamente com ordem $O(v)$.

A instânciação do grafo é dada pelo arquivo de entrada. Através dele, a função **preencheGrafo(char *)** o inicializa e insere todas as listas de adjacência, suas arestas e peso. Além disso, também armazena a quantidade de caminhos que se deseja encontrar.

Por se tratar de uma lista encadeada, a inserção é constante, já que não é necessário mudar elementos de lugar.

Para selecionar o arquivo de entrada, basta executar o programa no seguinte formato:

./programa -i entrada.txt -o saida.txt

3.2 *Min-Heap*

A estrutura *Min-Heap* foi implementada para armazenar vértices, ou seja, uma estrutura que representa os vértices. A implementação se trata de uma fila de prioridade em que seu primeiro elemento vai ser sempre o elemento de menor peso dentro da heap. Desse modo, podemos utilizá-la no nosso algoritmo para termos acesso fácil ao menor elemento da heap.

As funções mais custosas dentro do funcionamento da *Min-Heap* são aquelas relacionadas a manutenção das suas propriedades. Portanto, quando inserimos um elemento ou removemos o primeiro elemento dessa fila de prioridade devemos organizar a fila de modo que o primeiro elemento continue sendo o menor da fila.

Os custos para manter a fila tem ordem de complexidade $O(log(v))$, sendo v o número de elementos na heap, nesse caso vértices. Uma vez que, as funções utilizadas para a inserção de elementos são void **insere(Heap *h, Vertice data)** e void **ajudaInserir(Heap *h, int index)**, a primeira função faz $O(1)$ operações e chama a segunda que por sua vez faz $O(log(v))$ operações, portanto para inserir temos custo da maior entre elas no caso, $O(log(v))$. O mesmo acontece na extração do elemento de maior prioridade, utilizando a função **extraiMenor(Heap *h)** que faz $O(1)$ operações e chama a função **minHeapify(Heap *h, int index)** que realiza $O(log(v))$ operações, logo para extrair o menor teremos uma complexidade $O(log(v))$.

Função	Complexidade
criarHeapVazia	O(1)
insere	O(log v)
minHeapify	O(log v)
extraiMenor	O(log v)
destroiHeap	O(1)

Figura 3: Ordens de Complexidade da *Heap*

3.3 Algoritmo de Eppstein

No nosso trabalho, o algoritmo de Eppstein foi implementado utilizando uma heap, especificamente uma Min-Heap. Essa implementação tem a premissa de colocar todos os adjacentes de um vértice que foi retirado da heap dentro dela. Além disso, o mesmo vértice pode aparecer várias vezes dentro da fila com distâncias até ele diferentes, já que existem diferentes caminhos até ele. Então podemos inferir que todas essas combinações de pesos de arestas vão estar na fila de prioridade. Desse modo, quando um vértice sai da heap é porque o caminho total até ele é o menor dentre os demais. Logo, se um nó sai da heap k vezes teremos as k menores distâncias até ele. Utilizamos esse conceito para encontrarmos os k menores caminhos entre dois vértices.

Ao analisarmos o comportamento do algoritmo de Eppstein no pior caso podemos estabelecer uma função que domina assintoticamente a complexidade do nosso algoritmo. Portanto vamos começar nossa análise pelas funções mais internas e ir diminuindo o nível de profundidade dentro do algoritmo para obtermos essa função de complexidade.

As funções mais intrínsecas do algoritmo que implementamos são a remoção e a inserção de nós dentro da Min-Heap, essas por sua vez tem complexidade $O(logv)$, sendo v o número de vértices. Logo, se todos os vértices que representam as a arestas vão ser incluídos na Min-Heap, por meio da função **insereAdjacentes(Grafo *grafo, Vertice pai, Heap *heap)**, teremos uma função de complexidade $f(aloga)$, consequentemente se pensarmos que cada vértice é visitado no máximo k vezes teremos uma função $f(ka(log(ka)))$. Portanto é possível definir que o algoritmo é $O(ka(log(ka)))$, já que existe uma constante c que se multiplicada por essa função vai ser sempre maior ou igual a $f(ka(log(ka)))$, após um determinado m, no caso, $c = 1$ e $m = 0$.

Função	Complexidade
eppstein	O(ka log ka)
insereAdjacentes	O(a log a)

Figura 4: Ordens de Complexidade de Eppstein

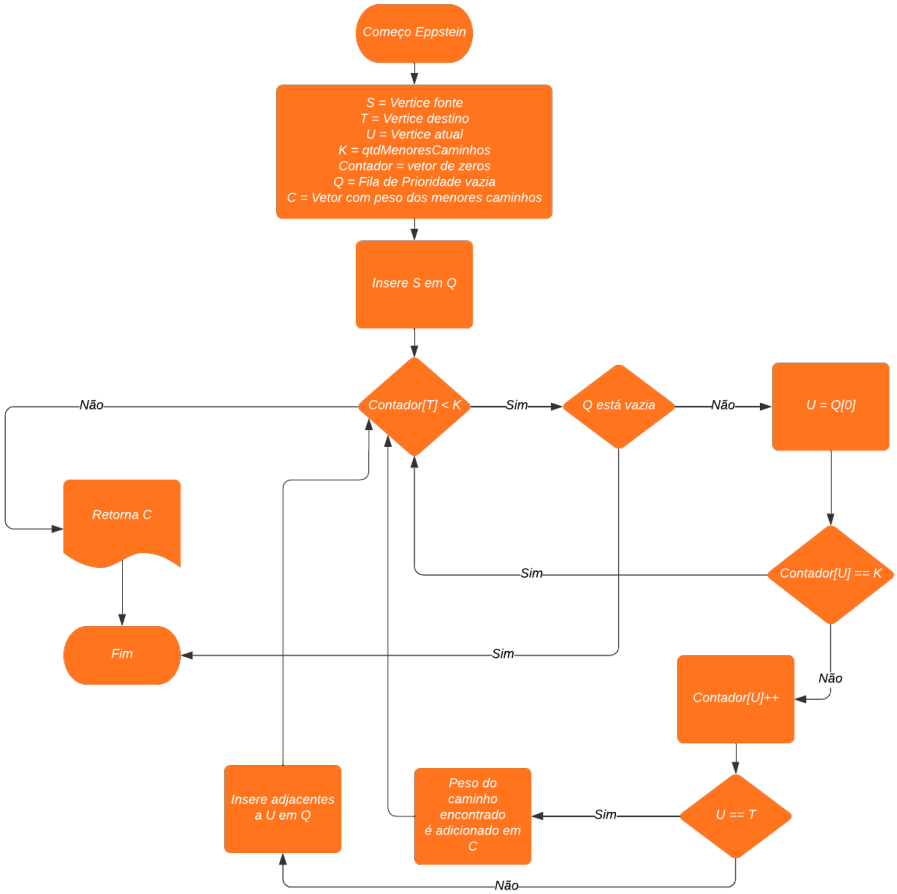

Figura 5: Fluxograma Eppstein

4 Resultados e Discussões

Inicialmente tentamos resolver o problema pela abordagem do algoritmo de Yen, no entanto, essa solução se mostrou inviável para o problema, pois esse algoritmo não é capaz de identificar os menores caminhos de um grafo com ciclos. Dessa forma, optamos pela abordagem do algoritmo de Eppstein, esse algoritmo se mostrou mais adequado para resolver o problema e ainda apresenta uma complexidade menor que a do algoritmo de Yen que implementamos. A seguir iremos apresentar os gráficos e suas análises baseadas nos resultados que obtivemos por meio da nossa implementação do algoritmo proposto por Eppstein.

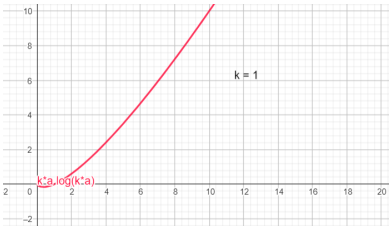

Figura 6: Comportamento quando k = 1

Figura 7: Comportamento quando k = 5

Figura 8: Comportamento quando k = 10

4.1 Testes

Para demonstração da eficiência da solução proposta para resolver o problema inicial, foram realizados diversos testes para comparação entre diferentes tipos de entrada. Variamos a quantidade de vértices, arestas, pesos e quantos menores caminhos deveriam ser encontrados.

A fim de sumarizar, iremos tratar de 3 casos de teste em particular:

Caso 1: O número de vértices e arestas é pequeno. V = 7 e A = 11

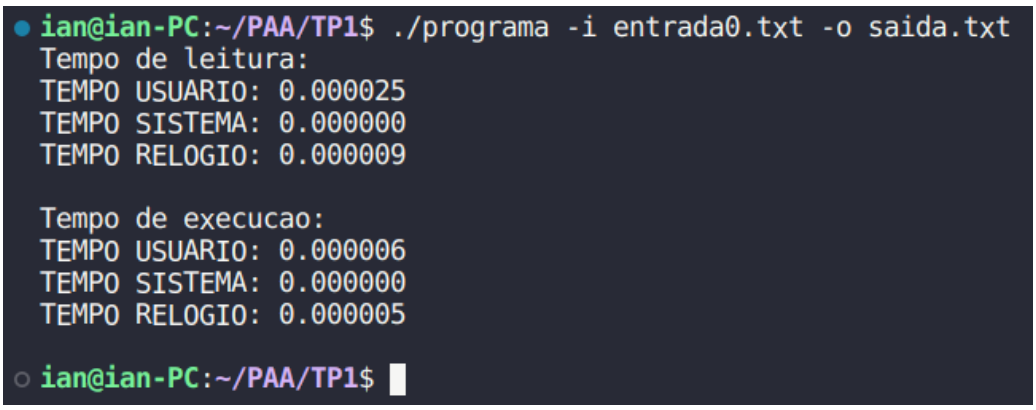

Figura 9: Caso de teste 1

Nesse ambiente, o algoritmo é executado quase instantâneamente como é possível perceber pelo tempo de relógio medido na execução. Esse tempo é dado pela captura do momento exato em que a função é chamada até retornar os menores caminhos.

Caso 2: O grafo é denso. V = 20.000 e A = 200.000

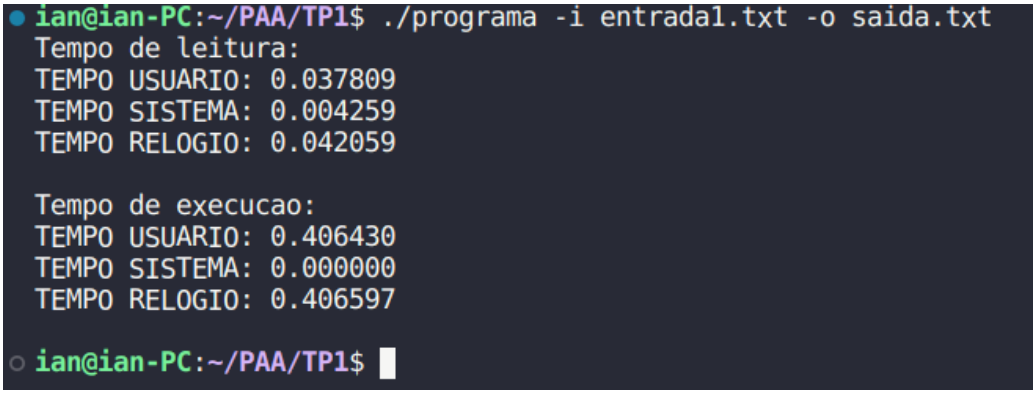

Figura 10: Caso de teste 2

Mesmo ao testar o grafo com um número de arestas muito maior que a quantidade de vértices, nota-se que o desempenho do algoritmo de Eppstein se prova muito eficiente ao lidar com vértices de grau elevado e a alta possibilidade de ciclos. Fator este que seria uma limitação do algoritmo de Yen anteriormente implementado.

Caso 3: O grafo está em seu caso máximo delimitado. V = 100.000 e A = 200.000

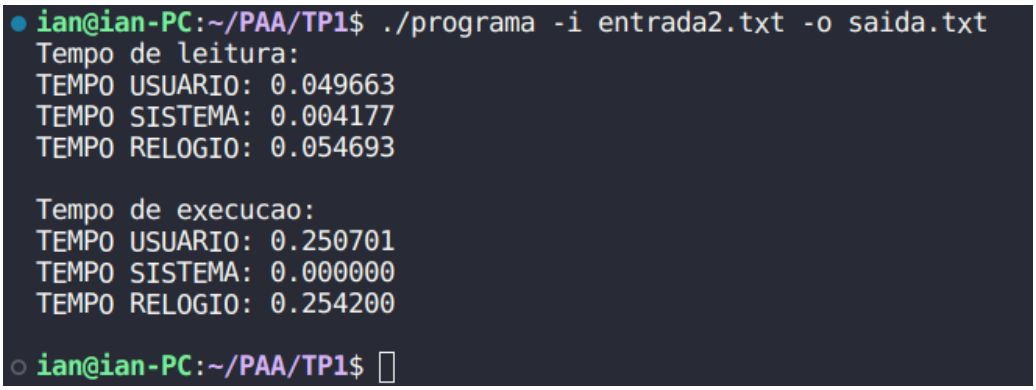

Figura 11: Caso de teste 3

Ao analisar o grafo em seu pior caso possível, ou seja, com o número máximo de vértices e arestas, é possível perceber a eficiência do algoritmo proposto por Eppstein em 1997 para lidar com um problema atemporal. Nesse caso de teste, mesmo com um grande número de vértices, a solução se provou satisfatória, uma vez que apresentou um resultado ainda melhor que o caso anterior. Isso ocorreu pela limitação da lista de adjacência, já que não é a melhor opção para representar grafos muito densos.

Com os testes realizados foi possível entender melhor o comportamento do algoritmo em diferentes cenários. Nesse sentido, sua aplicação trata-se de uma solução eficaz e confiável para encontrar os menores caminhos em uma modelagem baseada em grafos.

5 Conclusão

A pesquisa e a implementação do trabalho permitiram que pudéssemos aprofundar nossos conhecimentos num tema muito importante no nosso cotidiano, encontrar os menores caminhos entre dois pontos. É por meio da resolução desse problema, que nos foi apresentado de forma lúdica, que pudemos entender o funcionamento de diversas ferramentas utilizadas no cotidiano, como aplicativos de mapeamento e GPS, logística e redes de computadores.

Além disso, colocamos em prática os conhecimentos sobre criação de algoritmos que havíamos adquirido no decorrer do curso, principalmente no que diz respeito a modularização. Pudemos observar que a resolução que apresentamos tem uma complexidade bem menor do que outras possíveis soluções, como por exemplo o algoritmo de Yen, com complexidade $O(n^3)$, além do fato de considerar caminhos com ciclos.

Ademais, por meio do trabalho, conseguimos compreender ainda melhor os conceitos de grafos e suas aplicabilidades na hora de modelar problemas e aumentamos nossa perícia sobre as filas de prioridade. Por fim, entregamos uma implementação adequada, de forma que solucionar o problema estando de acordo com todos os requisitos definidos por ele.

6 Bibliografia

[1] LIMA, Ariane. Implementação de Grafos por Listas de Adjacência. 2024.

[2] ROCHA, Leonardo. Grafos: Algoritmo de Dijkstra. 2024.

[3] EPPSTEIN, David. Finding the k-Shortest Paths. SIAM Journal on computing, 1997.