Universidade Federal de São João del-Rei

Ciência da Computação
Projeto e Análise de Algoritmos

Documentação Trabalho Prático 1
Caminhos Mágicos

Lucas Eduardo Bernardes de Paula

Messias Feres Curi Melo

2024.1

1 Introdução

Um aventureiro deseja se deslocar de Mysthollow para Luminae, porém para isso ele terá que enfrentar uma rede de caminhos mágicos, guardados por encantamentos e cheios de desafios.

Cada caminho está repleto de desafios, eles se torcem e viram através das florestas, além de que um caminho pode fazê-lo revisitar a mesma cidade várias vezes. A missão desse aventureiro consiste em encontrar os k caminhos mais curtos entre as duas cidades, desbloqueando segredos e ganhando a admiração das criaturas mágicas que habitam essas terras. A Figura 1 exemplifica uma possível disposição dessas cidades representadas em um grafo.

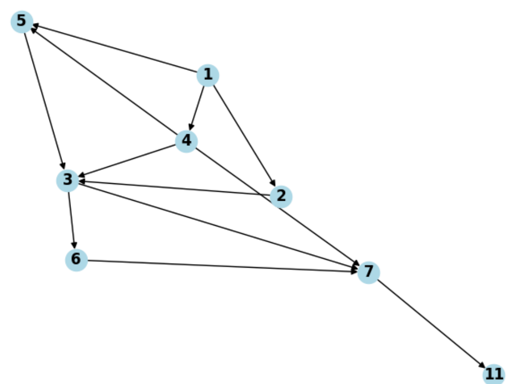

Figure 1: Exemplo da disposição das cidades em um grafo

Descobrir os k caminhos é um desafio adicional, pois os algoritmos tradicionais de caminhos mínimos não são capazes de solucionar esse problema de forma isolada, visto que repetiriam o mesmo caminho em todas as iterações. Portanto, é necessário introduzir uma estrutura de *Heap* para armazenar e selecionar caminhos mais curtos de forma eficiente, fazendo com que os algoritmos de caminho mínimo consigam identificar os k menores.

Faz-se necessário, portanto, desenvolver um programa que encontre a solução de maneira otimizada ao mesmo tempo que garante a facilidade de manutenção do código, o que pode ser alcançado por meio de boas práticas de programação e documentação adequada, assim como demonstrado adiante.

2 Arquitetando a solução

Antes de iniciar a programação, é fundamental realizar um planejamento detalhado do funcionamento do programa. Esse planejamento é essencial para garantir a qualidade do software ao longo de todo o projeto, facilitando processos como depuração e manutenção.

O processo foi divido em 4 etapas ilustradas pelos fluxograma da Figura 2. Essa divisão proporciona uma visão clara e estruturada do processo, permitindo uma abordagem mais organizada e eficiente na implementação do programa.

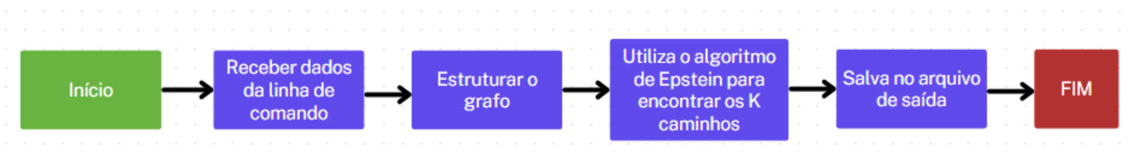

Figure 2: Fluxograma de funcionamento

2.1 Receber dados da linha de comando

Primeiramente, é necessário receber as informações do grafo pela linha de comando, como o número de vértices, arestas, suas conexões e seus pesos, contidos no arquivo de entrada. Esta operação requer o uso dos argumentos da linha de comando assim como no exemplo abaixo.

./tp1 -i entrada.txt -o saida.txt

Para receber esses dados é necessário utilizar os parâmetros $argc$ e $argv$, recebidos pela função $main()$, verificando se as entradas são válidas para o funcionamento correto do programa.

Após obter o endereço do arquivo de entrada, é efetuada a leitura completa do mesmo utilizando a função $fscanf()$, pertencente à biblioteca *stdlib.h*.

A primeira linha contém o número de vértices, o número de arestas e a quantidade de caminhos a serem encontrados, as linhas seguintes representam cada uma das arestas do grafo, sendo o vértice de origem, o vértice de destino e o peso da aresta

2.2 Estruturar o grafo

A estruturação do grafo é feita a partir da leitura do arquivo, onde é incumbida de gerar um novo grafo, especificamente com o número de vértices fornecido, estabelecendo uma base pronta para acolher estruturas de elementos relacionados em algoritmos de grafos

2.3 Utiliza o algoritmo de Eppstein para encontrar os K caminhos

O algoritmo mais conhecido para encontrar o menor caminho em um grafo é o $dijkstra$, este método se destaca especialmente em grafos com pesos não negativos, cenário frequente em diversas situações práticas de redes e sistemas. Porém, para conseguir encontrar os K caminhos, como pode ser visto na Figura 3, é necessária uma modificação no algoritmo, introduzindo uma estrutura de $Heap$ para armazenar e selecionar caminhos mais curtos de forma eficiente, permitindo a busca pelos k caminhos mais curtos em um grafo ponderado.

O funcionamento dessa estratégia para resolver o problema é descrito com detalhes na seçao 4 da documentação.

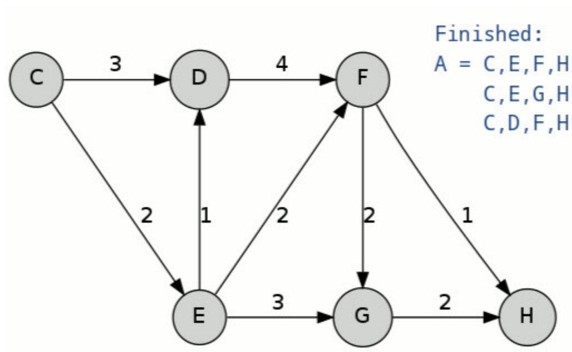

Figure 3: Menores caminhos para k = 3

2.4 Salvar os resultados no arquivo de saída

Os k caminhos encontrados são salvos em um arquivo de saída, o qual o nome foi especificado ao receber os dados da linha de comando, usando a função $fprintf$, também da biblioteca *stdio.h*.

Após a conclusão do programa, todas as estruturas de dados dinamicamente alocadas são liberadas para não ocorrer vazamentos de memória indesejados.

3 Estrutura de dados

Para o desenvolvimento eficiente do algoritmo, foi necessário implementarmos algumas estruturas de dados, entre elas iremos começar falando do grafo. Essa estrutura é fundamental, pois facilita tanto o armazenamento quanto a manipulação das informações relativas aos caminhos e às operações que podem ser realizadas sobre eles.

A seguir temos a implementação dessa estrutura:

```c
// graph.h

typedef struct Edge {
    int destino;
    unsigned long int peso;
    struct Edge* proxima;
} Edge;

typedef struct Vertex {
    int id;
    Edge* proxima;
    unsigned long int custo;
    int noAnterior;
} Vertex;

typedef struct Graph {
    int numVertices;
    Vertex* vertices;
} Graph;
```

Entendendo melhor o código acima, podemos começar falando sobre o $Graph$ ser responsável por mapear o conjunto de cidades(vértices) e seus caminhos(arestas), onde cada $Vertex$ representa uma cidade especifica, e cada $Edge$, uma rota entre duas cidades. Este modelo foi utilizado para otimizar tanto o armazenamento das informações vitais dos caminhos quanto a realização de operações de busca e otimização de rotas.

Para reforçar a eficiência na busca pelos caminhos mais curtos, incorporamos uma estrutura de dados complementar chamada $MinHeap$. Essa estrutura age como uma fila de prioridades, onde sempre temos acesso imediato ao elemento de menor valor, o que, neste contexto, corresponde ao caminho com o menor custo.

A organização da estrutura do $MinHeap$ é detalhadas logo a seguir:

```c
// Heap.h

typedef struct {
    unsigned long int distancia;
    int vertice;
} HeapNode;

typedef struct {
    HeapNode* elementos;
    int tamanho;
    int capacidade;
} MinHeap;
```

A $MinHeap$, constituída por $HeapNode$, é crucial para o algoritmo, pois cada nó contém o índice de uma cidade e a distância acumulada desde o ponto de partida, facilitando a localização do caminho mais eficiente entre duas cidades.

Essas estruturas, *Graph* e $MinHeap$, formam a base da nossa abordagem algorítmica, sendo fundamentais para o sucesso do projeto. Além disso, foram desenvolvidas técnicas específicas para a criação e liberação dessas estruturas, assegurando uma gestão eficaz dos recursos.

4 Estratégia de resolução

Conforme visto em seções anteriores, foi elaborada uma estratégia para encontrar os K caminhos, tal estratégia consiste em modificar o algoritmo de $dijkstra$, introduzindo uma estrutura de *Heap* para armazenar e selecionar caminhos mais curtos de forma eficiente, permitindo a busca pelos caminhos, tal modificação consiste na implementação do algoritmo de *Eppstein*.

O algoritmo de *Eppstein* é usado para encontrar os k caminhos mais curtos entre um vértice de origem e um vértice de destino em um grafo. Ele começa inicializando uma $MinHeap$, que é uma estrutura de dados que mantém o menor elemento sempre na frente, como dito anteriormente. Também são criados dois arrays: um para contar quantas vezes cada nó foi visitado e outro para armazenar os k caminhos mais curtos encontrados.

Aqui está o pseudocódigo do algoritmo de *Eppstein* implementado:

Algorithm 1 Função Eppstein

```
function EPPSTEIN(grafo, origem, destino, k, numeroVertices)
  Inicializa uma MinHeap q
  Inicializa um array count com tamanho numeroVertices, todos os elementos são 0
  Inicializa um array caminhos com tamanho k, todos os elementos são LONGMAX
  caminhosEncontrados ← 0
  Insere origem na MinHeap q
  while houver caminhos a serem encontrados e a MinHeap não estiver vazia do
    Extrai o nó mínimo da MinHeap q
    Incrementa o contador para o nó atual
    if nó atual for o destino e ainda não tivermos k caminhos encontrados then
      Adiciona o caminho atual ao array caminhos
    end if
    for cada aresta saindo do nó atual do
      if ainda não tivermos k caminhos para o destino then
        Insere o caminho até o próximo nó na MinHeap q
      end if
    end for
  end while
  Imprime os caminhos encontrados
  Libera a MinHeap q
  Retorna os caminhos encontrados
end function=0
```

Durante a exploração, o algoritmo atualiza um contador para cada nó visitado e verifica se o nó atual é o destino e se ainda não foram encontrados k caminhos até ele. Se essa condição for atendida, o caminho atual é adicionado ao array de caminhos mais curtos.

Para cada aresta saindo do nó atual, o algoritmo verifica se ainda não foram encontrados k caminhos até o destino. Se essa condição for verdadeira, o caminho até o próximo nó é inserido na $MinHeap$ para continuar a busca.

Quando todos os caminhos mais curtos foram encontrados ou não há mais nós para explorar na $MinHeap$, o algoritmo imprime os k caminhos mais curtos encontrados e libera a memória ocupada pela $MinHeap$. Finalmente, ele retorna os k caminhos mais curtos encontrados como resultado da função.

5 Análise matemática

O algoritmo de *Eppstein* é notável por sua complexidade que pode ser expressa como

$$O(km \log(km))$$

onde k é o número de caminhos mais curtos que estamos buscando e m é o número total de arestas no grafo. Esta complexidade deriva de uma análise detalhada das operações essenciais realizadas pelo algoritmo.

Primeiramente, a inserção e a extração de elementos na $Heap$ são operações cruciais para o funcionamento do algoritmo. A $Heap$ é uma estrutura de dados que mantém seus elementos organizados de forma a permitir um acesso rápido ao elemento de menor valor (ou maior, dependendo da implementação). No caso do algoritmo de *Eppstein*, a $Heap$ é utilizada para manter os caminhos mais curtos encontrados até o momento de forma ordenada, de modo que o caminho mais curto seja sempre acessível de forma eficiente.

A complexidade de inserção e extração em uma Heap depende do número de elementos na $Heap$, representado por n. Em uma Heap binária, por exemplo, a altura da árvore é aproximadamente $\log n$, o que significa que tanto a inserção quanto a extração de um elemento têm uma complexidade de $O(\log n)$. Isso ocorre porque, ao inserir ou extrair um elemento, a Heap precisa reorganizar seus elementos para manter a propriedade da $Heap$ (por exemplo, o menor elemento no topo em uma $min - Heap$).

No contexto do algoritmo de *Eppstein*, essas operações de inserção e extração ocorrem para cada um dos k caminhos mais curtos encontrados até o momento. Portanto, a complexidade total das operações de inserção e extração na Heap ao longo do algoritmo é $O(k \log n)$, onde n representa o número de elementos na $Heap$ em um determinado momento.

Além disso, o loop principal do algoritmo percorre as arestas do grafo, contribuindo significativamente para a complexidade total. Em um grafo densamente conectado, onde o número de arestas é considerável (m), a complexidade pode alcançar $O(m)$. Durante cada iteração desse loop, há a possibilidade de inserção na $Heap$, acrescentando mais $O(\log n)$ à complexidade.

Ao somar todas essas operações ao longo do algoritmo, a complexidade total do algoritmo de *Eppstein* pode ser descrita como $O(km \log(km))$. É interessante notar como o número de caminhos desejados k e a densidade do grafo m podem influenciar diretamente a eficiência do algoritmo. Quando k é pequeno em comparação com m, a complexidade tende a se aproximar de $O(m \log(m))$, especialmente em grafos densos, onde o número de arestas é considerável.

Na prática, se o algoritmo for aplicado na sua forma mais básica, pode acabar sendo ineficiente e lento, especialmente para valores grandes de k em grafos densos. Isso ocorre porque o número de operações e iterações começa a crescer exponencialmente. Por isso, quando for aplicar o algoritmo de *Eppstein* em problemas reais, é necessário considerar bem o número de vértices e arestas no grafo para garantir que o desempenho não seja comprometido em cenários de grande escala.

6 Testes

Os testes estiveram presentes em todas as etapas do desenvolvimento do software, desde o funcionamento das funções mais elementares até o tempo de execução da função para encontrar a solução do problema, garantindo uma aplicação performática e de fácil manutenção.

6.1 Tempo de execução

Para monitorar o tempo de execução total do algoritmo na busca dos caminhos, foi utilizada a biblioteca *getrusage.h*, que permite acompanhar especificamente o tempo de CPU, evitando inconsistências na medição decorrentes do uso do tempo físico como parâmetro

6.2 Tempo em função de k

Os resultados dos testes do algoritmo mostram como o tempo total de execução aumenta à medida que o valor de k aumenta. Isso reflete diretamente na eficiência do algoritmo, conforme já foi comprovado na análise de complexidade, e pode ser visto na Figura 4.

Figure 4: Variação de tempo em função de K

Ao analisar os tempos de execução para diferentes valores de k, podemos observar que, conforme k aumenta, o tempo total de execução também aumenta significativamente. Isso ocorre porque o algoritmo precisa encontrar k caminhos mais curtos, o que demanda mais operações à medida que k cresce.

Quando k é pequeno, como nos casos em que $k = 10$ e $k = 100$, o tempo total de execução é relativamente baixo. Porém, à medida que k aumenta para valores como $k = 1000$ e $k = 4000$, o tempo total de execução aumenta de forma considerável. Isso evidencia que a eficiência do algoritmo está diretamente ligada ao valor de k.

Além disso, podemos observar que o tempo de usuário e o tempo de sistema também aumentam com o aumento de k. Isso indica que o algoritmo está consumindo mais recursos do sistema à medida que é exigido encontrar mais caminhos mais curtos.

Portanto, esses resultados reforçam a importância de considerar o valor de k ao aplicar o algoritmo em cenários práticos, pois o desempenho do algoritmo pode ser comprometido significativamente para valores elevados de k, especialmente em grafos densos.

7 Conclusão

Durante o desenvolvimento do projeto, foram enfrentados alguns desafios, como na escolha inicial do algoritmo de Yen. Onde, após sua implementação e testes, foi descoberto que não se encaixaria no objetivo proposto. Assim, foi necessário substituilo por um algoritmo mais eficiente e que aceitasse loops, no caso retornos a mesma cidade, chegando a encontrar finalmente o algoritmo de *Eppstein*. Utilizando o *Eppstein* junto com uma estrutura $MinHeap$, possibilitou a busca eficaz dos K caminhos mágicos mais curtos, com uma complexidade de tempo total descrita como $O(km \cdot \log(km))$.

Ao realizar testes e análises matemáticas como descrito na seção 5 e 6, torna-se evidente a importância de considerar vários aspectos ao lidar com problemas algorítmicos. Pois, especificamente no caso do *Eppstein*, foi encontrado um problema no valor de k, onde para valores pequenos, o algoritmo apresente um tempo de execução eficiente e rápido, mas quando seu valor aumenta, a execução cresce consideravelmente, não sendo mais um algoritmo eficaz após k maior que 4000. Porém, como o objetivo proposto é um k limitado até 10, essa solução se adapta especialmente no contexto, se tornando a solução mais eficiente encontrada.

Portanto, ressaltamos a boa prática de verificar as condições de diferentes abordagens antes de colocar seu código em pratica. E, através dos resultados obtidos, mostramos que a necessidade de garantir uma rapidez e eficiência aos algoritmos é uma tarefa crucial de todo programador. Já que quanto mais eficiente é um algoritmo, menor é a quantidade de recursos necessários para executa-lo, reduzindo significativamente os custos de operação e sua complexidade.

8 Referências

1. Thomas H. Cormen. *Algoritmos: Teoria e prática.* LTC, 2012.
2. Nivio Ziviani. Projetos de Algoritmos em C e Pascal. 2. ed. São Paulo: Cengage Learning, 2006.
3. Diogo Haruki Kykuta. Comparação de algoritmos para o Problema dos K Menores Caminhos. Dissertação de Mestrado, Instituto de Matemática e Estatística, Universidade de São Paulo, 2018.
4. Jiménez, Víctor M. and Marzal, Andrés. "A Lazy Version of Eppstein's K Shortest Paths Algorithm."