Universidade Federal
de São João del-Rei

A Busca por Caminhos Mágicos

André Chagas Lima e Davi Greco Resende Freitas

Projeto e Análise de Algoritmos
Ciência da Computação

São João Del Rei

2024.1

1 Introdução

No vasto e enigmático reino de Xulambis, situado entre as florestas ancestrais de Mysthollow e os prados encantados de Luminae, existe uma rede de caminhos mágicos repleta de maravilhas e perigos. Estes caminhos não apenas conectam os dois reinos, mas também são permeados por mistérios antigos e guardados por encantamentos poderosos. O desafio de desvendar esses caminhos é reservado apenas aos aventureiros mais valentes e astutos, os quais buscam tanto a glória quanto o conhecimento esquecido.

O objetivo deste projeto é identificar os k caminhos mais curtos que ligam Mysthollow a Luminae. Para modelar o problema, utilizamos um grafo para representar o reino e seus respectivos caminhos mágicos. A figura 1 é exemplo de um grafo com vértice inicial em 1 (Mysthollow) e final em 5 (Luminae) com 7 arestas (Voos).

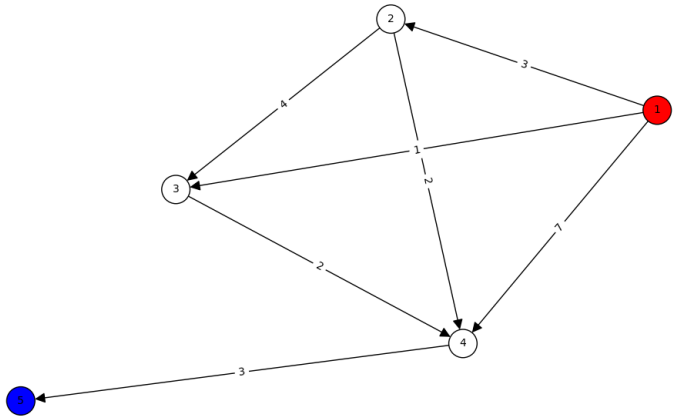

Figura 1: Exemplo de grafo

2 Estrutura do Código

Levando em consideração o problema apresentado, é de grande importância uma boa estruturação de código. Por isso, é imprescindível definir como será feita a estruturação antes de começar a programar definitivamente.

Como é visto no fluxograma da Figura 2, o código inicia a execução e recebe os arquivo de entrada e de saída, logo após lê a entrada e cria o grafo. Depois disso, executa o algoritmo principal do programa, encontrando os K menores caminhos e por fim escreve os resultados de K e o tempo de execução no arquivo de saida.

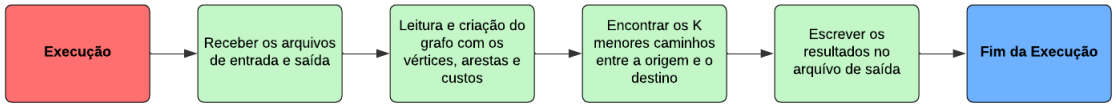

Figura 2: Fluxograma de linha do tempo do código

2.1 Leitura de Arquivos

Para receber as informações do reino foi utilizado como base um arquivo padrão que contém na primeira linha as informações das constantes n, numero de vértices, m, número de arestas e a constante k, já no restante das linhas são representados os voos disponíveis com as informações de origem, destino e custo respectivamente.

A compilação do código é feita a partir de um makefile que define os passos de compilação dos arquivos e cria um executável final automaticamente ao inserir o comando **make** no diretório do projeto. Na execução basta utilizar o "./executavel"que foi gerado pelo makefile seguido da definição dos arquivos de entrada(grafo) e de saída(resultado) a partir da primitiva getopt que utiliza as tags -i para entrada e -o para saída na linha de comando como representado na Figura 3.

```
./executável -i entrada.txt -o saida.txt
```

Figura 3: Exemplo da linha de comando para a execução

2.2 Criação do Grafo

Logo após receber os arquivos de entrada e saída o grafo é lido com a função **fscanf()** e gerado baseado na primeira linha, que diz a quantidade de vértices N e arestas M, além do parametro K.

Em seguida lê os N vértices e M arestas, adicionando-os no grafo por meio da função **adicionarAresta()**, dessa forma o grafo está iniciado e pronto para resolver o problema.

2.3 Solução Proposta

Na tentativa de encontrar os k menores caminhos, escolhemos o algoritmo de Dijkstra como base, que encontra o menor caminho no grafo com os padrões especificados no problema, como representado na Figura 4.

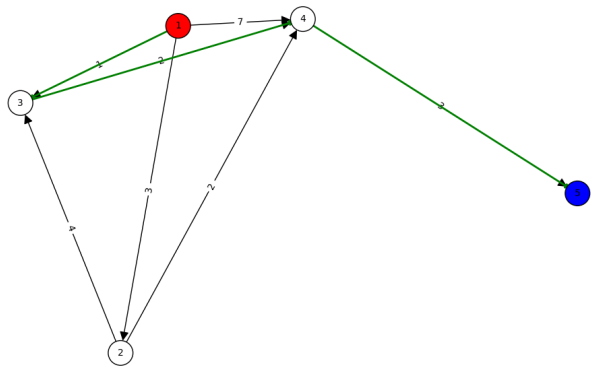

Figura 4: Exemplo de caminho mínimo utilizando Dijkstra

Nas tentativas e pesquisas realizadas para adptar o algoritmo ou manipulá-lo para que encontre os k menores caminhos encontramos e implementamos o algoritmo de Yen, que manipula o grafo a fim de que o algoritmo de Dijkstra seja induzido a encontrar novos caminhos, porém, a partir de testes e análise mais aprofundada do algoritmo, percebemos que não são consideradas as cidades que já foram visitadas, ou seja, os vértices não se repetem em um mesmo caminho, o que vai contra as especificações.

Ao encontrar o problema conseguimos uma solução no algoritmo de Eppstein que usa os princípios de Dijkstra para encontrar os k menores caminhos e permitir a repetição de uma cidade em um caminho até k vezes.

2.4 Escrever os resultados

Com a resolução do problema concluida, após encontrar os K caminhos, o resultado é guardado no arquivo recebido na hora da execução, sendo essa a única saída do programa. Com a utilização da função **fopen()**, caso o arquivo não exista ele é criado e aberto para escrita, caso contrário, ele é aberto para escrita.

A escrita foi realizada com a função **fprintf()**.O arquivo de saída é dividido por três partes, sendo a primeira o tempo de execução de usuário, a segundo o tempo de sistema e a terceira, os K resultados obtidos.

3 Estruturas de Dados

3.1 Representação do Grafo

Em termos de programação optamos por representar nosso grafo através de uma lista de adjacência, exemplificada na Figura 5, tendo em vista que frequentemente é mais comum na representação de caminhos gerar um grafo esparso, assim otimizando a utilização de memória em relação à matriz de adjacência e reduzindo as iterações para encontrar os vizinhos de um vértice.

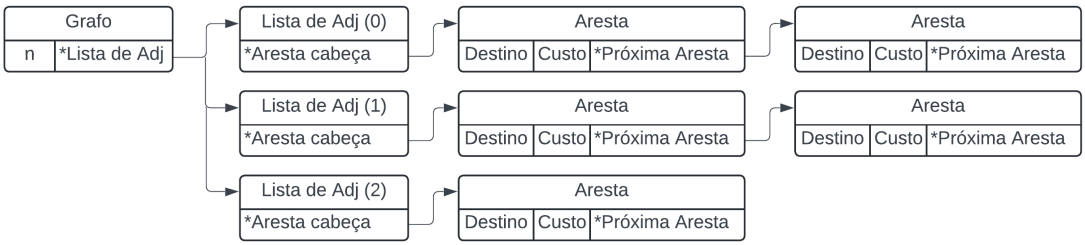

Figura 5: Representação do grafo por lista de adjacência

3.2 Implementação da MinHeap

A fim de reduzir a complexidade de encontrar os menores valores nos algoritmos de busca de caminhos, optamos por utilizar uma estrutura MinHeap representada

por um vetor de nós, onde cada nó armazena o vértice e o custo do caminho atual encontrado até o mesmo, mantendo os custos de cada vértice ordenados a partir do menor custo.

A Heap foi implementada por um array onde, i sendo o a posição do pai no array os filhos estão na posição i*2+1 e i*2+2, como representado na Figura 6. Escolhemos essa implementação de Heap por sua otimização de memória em relação à implementação por árvore binária, reduzindo a necessidade de dois ponteiros em cada nó, e pela facilidade em acessar os nós e realizar suas operações de reordenação.

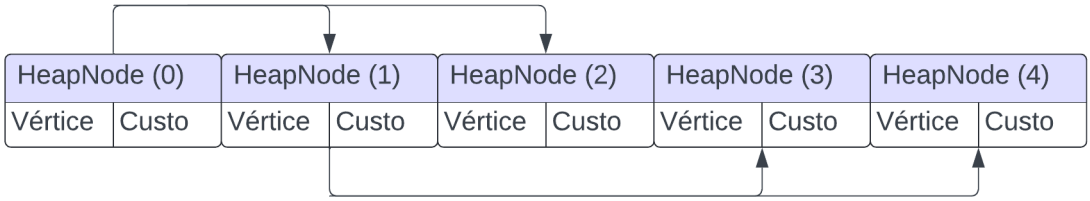

Figura 6: Representação da MinHeap binária por array

4 Algoritmo de Resolução

O problema pode ser resolvido, por meio de um algoritmo de força bruta, que consiste em testar todas as possibilidades de caminhos, porém, o custo para testar todos os caminhos é muito alto. Por esse motivo, é importante arquitetar outra solução que demore menos tempo para resolver o problema.

4.1 Algoritmo Eppstein

O algoritmo de Eppstein, que se baseia em princípios do algoritmo de Dijkstra, armazena o menor custo de caminho para cada vértice em uma estrutura MinHeap. A MinHeap é utilizada para manter os vértices organizados de forma que o vértice com o menor custo acumulado até ele seja facilmente acessível. O algoritmo inicia no vértice de origem, inserindo-o na Heap com um custo inicial de zero.

A partir daí, o processo se desdobra de maneira iterativa. Em cada iteração do algoritmo, o vértice com o menor custo acumulado é extraído da MinHeap. Após sua extração, todos os seus vizinhos são explorados. Cada vizinho é então inserido na MinHeap com um custo atualizado, que é a soma do custo acumulado até o vértice atual mais o custo da aresta que conecta esse vértice ao seu vizinho. Este procedimento garante que a Heap sempre contenha os próximos vértices a serem explorados ordenados pelo custo do caminho até eles.

O algoritmo continua a extrair o próximo vértice com o menor custo da MinHeap e a inserir seus vizinhos até que o vértice de destino seja alcançado e extraído k vezes, indicando que k caminhos mínimos foram encontrados para o destino. Cada vez que o vértice de destino é extraído da Heap, um dos k menores caminhos até ele é identificado, conforme ilustrado no fluxograma da Figura 7.

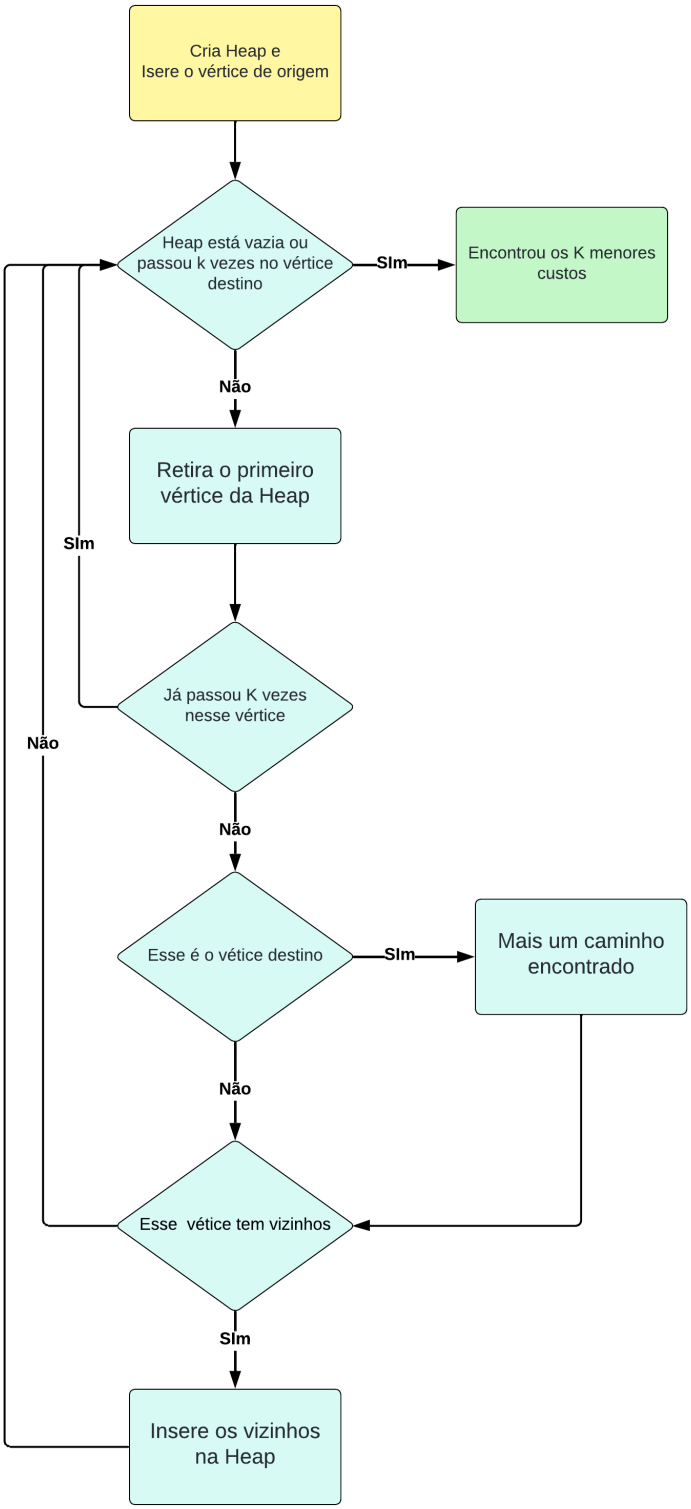

Figura 7: Fluxograma de funcionamento do algoritmo de Eppstein

5 Análise Matemática

A implementação do algoritmo de Eppstein utilizando uma estrutura de MinHeap binária nos permite alcançar uma complexidade de

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

onde k é o número de caminhos mais curtos desejados e m é o número total de arestas no grafo. Esta complexidade é derivada de uma análise detalhada das funções e operações fundamentais envolvidas no algoritmo. Vamos aprofundar a explicação para entender cada componente desta complexidade.

A MinHeap é utilizada para gerenciar os caminhos potenciais de forma eficiente, permitindo a rápida recuperação do caminho com o menor custo acumulado. As operações principais na MinHeap são inserções e retiradas (operações de pop), cada uma com uma complexidade de O(logn), onde n representa o número de elementos na Heap. Em particular, no contexto do algoritmo de Eppstein, um vértice pode ser inserido na heap até k vezes, refletindo a busca por k caminhos distintos. Assim, o valor de n, o número de elementos na Heap, pode chegar a ser tão grande quanto km, resultando numa complexidade por operação de O(log(km)).

Durante a execução do algoritmo, cada vértice processado leva à inserção de todos os seus vizinhos na Heap. Considerando que um vértice pode ter até m vizinhos (no caso de um grafo completamente conectado ou em cenários onde cada vértice está conectado a muitos outros), a complexidade de inserir todos os vizinhos de cada vértice na heap é proporcional ao número de arestas, m. Portanto, a cada extração de um vértice da Heap, potencialmente todas as suas conexões (arestas) são revisitadas para inserção na Heap. Dado que cada um dos k caminhos desejados pode exigir processar cada aresta no pior caso, a complexidade total de todas as inserções no loop principal do algoritmo é de km inserções. Como cada inserção tem uma complexidade de O(log(km)), a complexidade do loop principal pode ser expressa como O(kmlog(km)).

Somando todas essas contribuições, a complexidade total do algoritmo de Eppstein torna-se O(kmlog(km)). Esta análise reflete não apenas o número de operações realizadas, mas também a eficiência com que a estrutura de dados da MinHeap gerencia essas operações, equilibrando a necessidade de rápida recuperação e inserção de novos caminhos com a manutenção da ordem de custo mínimo.

6 Teste de Software

Tendo o algoritmo pronto, ele deve ser devidamente testado para avaliar seu desempenho para solucionar o problema.

Foram realizados diversos testes, consideram tempo de execução, além de vários tamanhos de entradas para as arestas, vértices e para o parametro K, com o intuito de tirar algumas conclusões sobre a análise matemática.

6.1 Verificação do Tempo

Para verificiar o tempo foram propostos duas funções, a**getrusage()** e a **gettimeofday()**, ambas pegam o início e fim de execução e por meio de uma pequena função, é encontrado o tempo em segundos.

Porém, as duas foram utilizadas de formas diferentes, a **getrusage()** foi usada para calcular o tempo de CPU, tendo mais precisão no tempo de execução. Já a **gettimeofday()** foi usada para calcular o tempo de Usuario.

6.2 Tempo do Algoritmo de Eppstein

Como citada na análise matemática, a complexidade do Algoritmo de Eppstein é O(kmlog(km)). Foram realizados testes para diferentes valores do parametro K, em um mesmo grafo. Com os resultados obtidos, foi gerado um gráfico, representado na Figura 8 com base na variação do tempo em segundos.

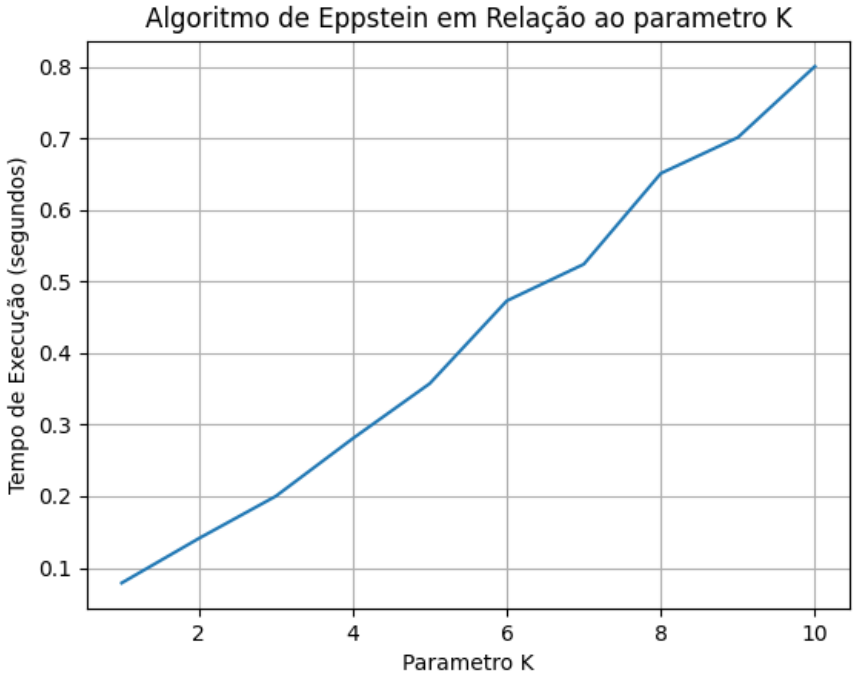

Figura 8: Grafico de variação do algoritmo de Eppstein em relação a K

Da mesma forma, foram realizados testes para diferentes valores de arestas em grafos com as mesma quantidade de vértices e o mesmo parametro K. Com os resultados, foi traçado um grafico representado na Figura 9 com base na variação do tempo em segundos.

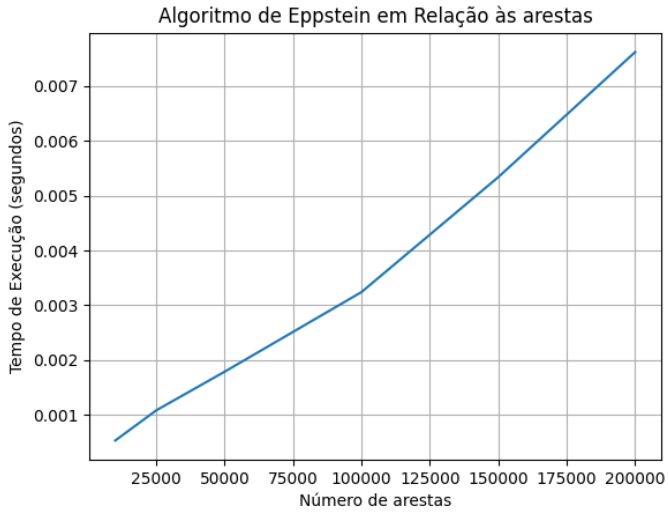

Figura 9: Grafico da função m log(m)

Da mesma maneira, foi traçado também um gráfico de uma função m log (m). Representado na Figura 10.

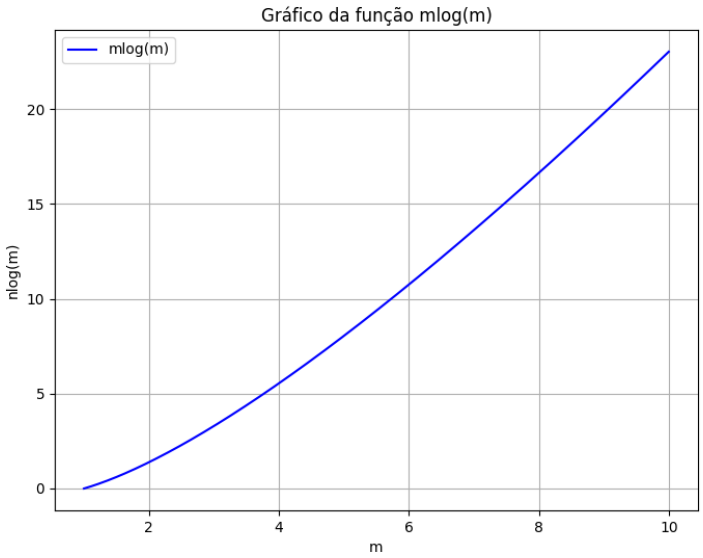

Figura 10: Grafico da função m log(m)

O resultado obtido comprova a análise matematica do Algoritmo já que, quanto maior o parametro K e o número de arestas, o tempo de execução fica maior. Com os Gráficos traçados, é possível ver uma Progressão seguindo a Progressão de uma função mlog(m).

7 Conclusão

O Algoritmo de Eppstein demonstrou ser uma escolha eficaz e adequada para resolver o problema de encontrar os caminhos mais curtos entre Mysthollow e Luminae. A eficiência desse método deriva principalmente da utilização da estrutura de dados MinHeap e da representação do grafo por meio de listas de adjacência. Ao armazenar e acessar os vértices com base no menor custo acumulado, a MinHeap garante que o processamento dos caminhos seja feito de maneira prioritária e ordenada, reduzindo significativamente o tempo de busca e a complexidade operacional, especialmente útil em grafos onde o número de caminhos k não é excessivamente grande.

A representação do grafo por listas de adjacência complementa perfeitamente a utilização da MinHeap ao otimizar o espaço de armazenamento e a velocidade de acesso aos vizinhos de cada vértice. Esta escolha é particularmente benéfica em aplicações como a nossa, onde o grafo pode ser extenso, mas não necessariamente denso. As listas de adjacência permitem um rápido percurso pelos vizinhos diretos de um vértice, facilitando a inserção subsequente desses caminhos na MinHeap com os custos atualizados após cada iteração. Esta combinação maximiza a eficiência ao lidar com a dinâmica dos caminhos mágicos, que exigem reavaliações frequentes das rotas conforme novas conexões são exploradas.

Portanto, a aplicação do algoritmo de Eppstein, ancorada na sólida implementação utilizando MinHeap e listas de adjacência, provou ser uma solução robusta para o problema proposto. Ela não apenas atendeu às exigências de encontrar múltiplos caminhos mínimos de forma eficiente, mas também se mostrou adaptável às restrições específicas do contexto mágico entre Mysthollow e Luminae. Essa abordagem metodológica garante que mesmo com a limitação na quantidade de caminhos explorados, definida pela constante k, a solução permanece prática e eficaz, revelando caminhos que são tanto viáveis quanto ótimos dentro do vasto reino encantado.

Referências

[1] David Eppstein, “Finding the k Shortest Paths”, 1997.

[2] Codeforces, "K shortest paths and Eppstein’s algorithm"https://codeforces.com/blog/entry/102085