Titulo

Diego Augusto Lucas Costa

Maio de 2023

1 Introdução

O problema proposto para solucionar reside em um contexto de reinos encantados, onde deseja-se encontrar os k caminhos mínimos saindo do reino Mysthollow até o reino de Luminae. Entre esses reinos existem n cidades, e os caminhos entre as cidades contêm desafios. Este problema é bem conhecido na literatura e pode ser representado por um grafo. Portanto, aplicaremos uma técnica para encontrar o caminho mínimo (ou custo mínimo) em um grafo ponderado, onde os vértices são as cidades, tendo a cidade inicial como o reino de Mysthollow e a cidade final como o reino de Luminae. As arestas representam as ligações entre as cidades, e os desafios entre as cidades serão representados pelos pesos. Assim, resolver este problema significa determinar o caminho entre dois vértices com o custo mínimo, ou seja, com o menor tempo de viagem.

Os dados para a construção do grafo são fornecidos pelo usuário através de um arquivo de entrada, e a solução do problema será apresentada em um arquivo de saída. O principal foco deste trabalho é o desenvolvimento e análise de algoritmos que solucionam o problema, levando em consideração a eficiência deles nos diversos contextos em que o problema pode estar inserido.

Diante disso, é necessário desenvolver um programa que, além de encontrar a solução, preocupa-se com a otimização e facilite a manutenção por meio de boas práticas de programação e uma documentação adequada, como será demonstrado nas próximas seções.

2 Desenvolvimento do Projeto

Para desenvolver a solução, optamos pelo algoritmo de Eppstein. Fizemos essa escolha pensando em várias características do problema, como a quantidade de nós, arestas e o valor de k. Isso se deve ao fato de que os valores de k não excedem 10, tornando o algoritmo bem rápido. O motivo será explicado nas próximas seções.

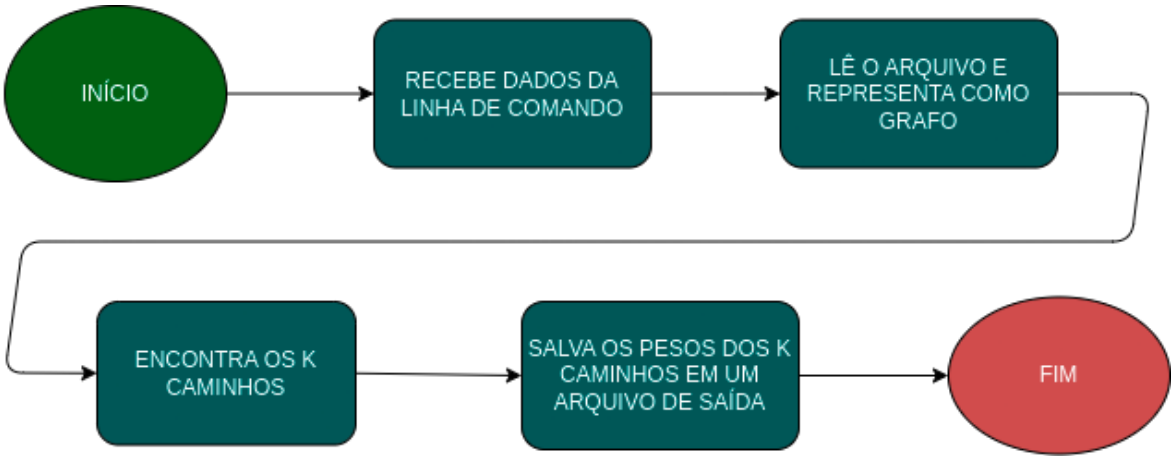

Figura 1: Fluxograma de execução

Como pode ser observado na figura **??**, algoritmo recebe da linha de comando o arquivo de entrada e saída. O arquivo de entrada contém a representação de um grafo de forma textual. A primeira contém a quantidade de vértices e arestas, além da quantidade de caminhos que se deseja encontrar; as demais linhas contém os arcos ponderados que formam o grafo. Então, ele lê o arquivo e cria uma representação em memória desse grafo através de uma lista de adjacência.

Depois disso, é executato a nossa implementação do algoritmo de Eppstein [citação], que percorre o gráfo e descobre os k menores caminhos. E então, guardamos os pesos dos menores caminhos no arquivo de saída.

2.1 Análise preliminar

Durante a execução do trabalho, ficou evidente a importância de uma boa organização para iniciar o desenvolvimento do algoritmo, desde uma boa análise do problema proposto até as escolhas para a sua resolução. Partindo disso, percebemos que seria crucial abordar algumas questões importantes no problema, como os limites de suas entradas e suas especificações. Durante os estudos das heurísticas que poderiam solucionar o problema, encontramos alguns algoritmos que pareciam ser boas opções, porém não eram adequados para o problema proposto. Um deles era o algoritmo de Yen, que, embora encontrasse o caminho mínimo em um grafo, não encontrava os k caminhos mínimos devido à sua incompatibilidade com loops, impossibilitando assim a correta

identificação dos k caminhos. A primeira impressão foi de que o algoritmo solucionaria o problema, mas como os testes estiveram presentes desde o início do desenvolvimento do software, foi possível detectar rapidamente o problema quando executamos um teste mais específico e não obtivemos o resultado esperado.

A partir disso, percebemos que o algoritmo de Yen não era uma solução viável, pois não tratava dos loops e resultava em caminhos mínimos incorretos devido às arestas compartilhadas. Sendo assim, buscamos um algoritmo que corrigisse as deficiências encontradas no Yen, baseando-se no mesmo princípio do Dijkstra, mas modificando sua lógica para lidar com as exceções. Com isso, optamos pelo algoritmo de Eppstein, o qual produziu excelentes resultados em diversos contextos, como demonstrado na seção de testes, proporcionando-nos uma solução eficiente para o problema.

2.2 Detalhes de implementação

Para representar o grafo, utilizamos a representação de lista de adjacência. Uma lista de adjacência consiste em uma lista de tamanho n, onde cada elemento guarda uma lista encadeada contendo as arestas do nó naquele índice. Ou seja: as arestas que saem do nó v ficam guardadas no índice $v - 1$ da lista de adjacência.

Fizemos essa escolha pois é uma representação relativamente simples de se implementar, além de performar bem em grafos esparsos e razoavelmente em grafos densos se comparado à uma matriz de adjacência [codeforce].

Imagem de uma lista de adjacencia

Para desenvolver a solução, optamos pelo algoritmo de Eppstein. Fizemos essa escolha pensando em várias características do problema, como a quantidade de nós, arestas e o valor de k. Isso se deve ao fato de que os valores de k não excedem 10, tornando o algoritmo bem rápido. O motivo será explicado nas próximas seções.

2.3 Lista Dinâmica

Para implementar várias estruturas de dados, criamos uma lista paramétrica homogênea, que funciona para vários tipos de elementos, todos do mesmo tipo, como as listas de linguagens de programação com suporte para tipos paramétricos (Rust, Java, C++, etc). Sua implementação encontra-se nos arquivos `list.h` e `list.c`.

A nossa lista consiste em um ponteiro para um arranjo alocado dinamicamente, o tamanho do elemento, o comprimento da lista e a capacidade de alocação de memória. Para tornar as inserções eficientes, reservamos mais espaço do que elementos contidos na lista, assim, quando temos espaço sobrando, apenas precisamos incrementar o contador de elementos. Isso faz com que as inserções não tenham um impacto significativo na execução do programa, pois as realocações são esporádicas.

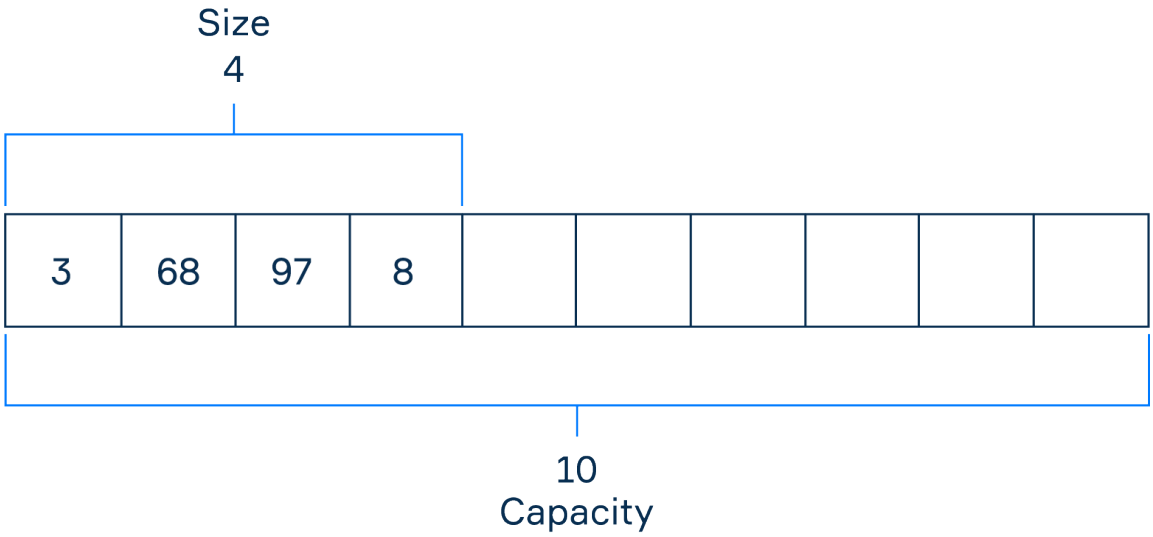

Figura 2: Lista com capacidade

2.4 Fila de prioridade

A fila de prioridade é

2.5 Eppstein

```
def eppstein(graph, s, t, k):
    q = Heap()
    visitados = [0] * len(graph)
    q.push(s, 0)
    while visitados[t] < k:
        (v, w) = q.pop()

        if visitados[v] == k:
            continue

        visitados[v] += 1

        if v == t:
            continue

        for (u, l) in g.sucessors(v):
            push(q, u, w + l)
```

Figura 3: Pseudocódigo do algoritmo de Eppstein

O algoritmo de Eppstein consiste em sucessivamente retirar um nó v da fila de prioridade. Em cada iteração, checamos se v já foi visitado o máximo de vezes, e se sim, vamos para o próximo nó da heap. Se não, incrementamos o contador de visitas do nó v. Caso esse nó seja o nó destino, encontramos um menor caminho, e vamos para a próxima iteração. Caso contrário, enumeramos as arestas que saem de v e adicionamos cada vértice u na heap, com a prioriade de $w + l$; onde w é o peso associado a v, e l é o peso associado à aresta $v - u$. Para cada vez que o t é visitado, é encontrado um caminho mínimo, até que t seja visitado no máximo k vezes. Quando o algoritmo retornar, teremos encontrado os pesos dos k caminhos.

3 Análises de complexidades

O algoritmo de Eppstein utiliza como base o algoritmo de Dijkstra, então antes de analisarmos o Eppstein, vamos analisar o Dijkstra. O Dijkstra, quando implementado com listas de adjacência e heap binária, tem complexidade de $O(n \log n + m)$, sendo n os vértices e m as arestas. Como o algoritmo inicializa as distâncias de cada vértice e cria uma fila de prioridade com o custo de $O(n)$, a cada passo do algoritmo, ele extrai o vértice com custo mínimo e aplica a técnica de relaxamento, que tem custo

$O(\log n)$, e todas as arestas são visitadas com custo $O(m)$. Portanto, chegamos na complexidade de $O(n \log n + m)$.

Como dito nas seções anteriores, o Eppstein assume que $n \leq m$, então a ordem de complexidade que domina assintoticamente é em função de m, por isso que no Eppstein analisamos em função de m, não mais de n como no Dijkstra. Além disso, o Eppstein permite que um vértice apareça várias vezes na heap, pois estamos analisando todas as arestas que incidem naquele vértice. Por isso, o pior caso aumenta para m, pois isso representa outros caminhos, no caso de arestas compartilhadas.

O algoritmo analisa cada aresta que sai do vértice atual e realiza uma inserção na heap cujo o custo no pior caso é $O(\log m)$. Como esse processo é feito no máximo m vezes, o custo total para inserir todas as arestas na heap é $O(m \log m)$. Porém, como queremos os k menores caminhos, essa lógica será executada k vezes, então conclui-se que a complexidade do algoritmo de Eppstein é $O(km \log(km))$.

4 Resultados e Análises

Como podemos ver na figura **??**, a análise do desempenho do algorítmo revelou que a nossa implementação desmonstrou consistentemente o comportamento esperado pela função de complexidade $km \log km$ (representada pela linha amarela), sendo k fixado em 10. Estes dados foram obtidos através da média de 5 execuções por teste.

Os testes estão localizados no diretório `tests/` e foram projetados para avaliar o desempenho do algoritmo em diferentes cenários. Cada teste varia em termos da quantidade de vértices e arestas do grafo, abrangendo uma ampla gama de casos de uso. A variedade de instâncias de teste inclui configurações com um número modesto de vértices e arestas, como $m = [20, 7, 6]$, até casos mais desafiadores com grafos massivos, representados por $m = [100202, 199945, 35299, 500, 149997, 200000]$.

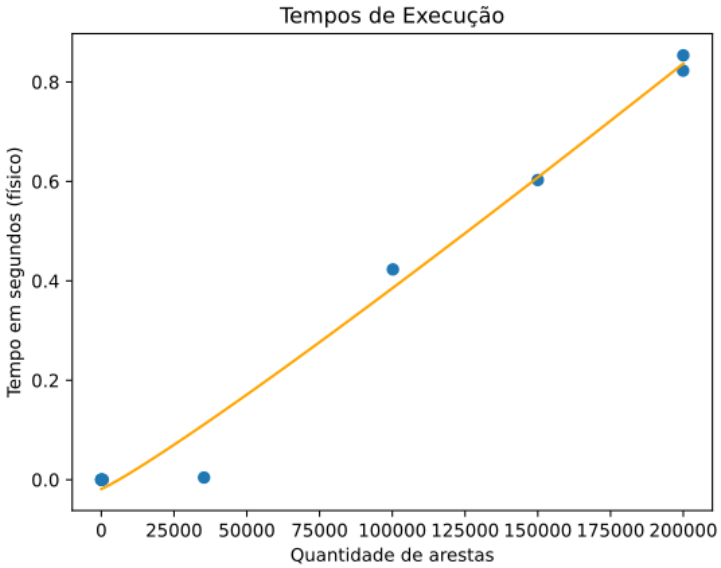

Figura 4: Gráfico dos tempos de execução do algoritmo Eppstein

4.1 Discussão sobre Tempos de Execução

Os tempos de usuário e sistema são duas diferentes formas de medir o tempo de execução. O tempo de sistema mede quanto tempo o sistema operacional executou tarefas para o programa, como ler arquivos, alocar memória, etc. Enquanto o tempo de usuário mede a quantidade de tempo que o programa utilizou a CPU.

Optamos pela biblioteca getrusage.h para avaliar o tempo de execução completo do nosso algoritmo, pois ela oferece uma variedade de informações úteis, incluindo o tempo total de processamento do programa. Isso representa uma vantagem signifi-

cativa em comparação com a biblioteca gettimeofday.h, que só registra o tempo físico decorrido, podendo resultar em medições inconsistentes.

É claro que, devido ao fato de que montar o grafo envolve muita leitura de arquivo, o tempo de sistema será predominante ao de usuário para essa parte, enquanto o tempo de usuário será predominante para a execução do algoritmo.

A soma destes valores equivale ao tempo físico. O tempo físico é a quantidade de tempo que nós seres humanos percebemos passar.

5 Conclusão