Segundo trabalho de Algoritmo e Estrutura de Dados II
Relatório da Resolução

Diego F. Mello

[1]Prédio 32 – Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)

Abstract. *This work presents the implementation of a pathfinding algorithm based on Dijkstra's method to determine optimal routes between ports represented in a text-based map.*

Resumo. *Este trabalho descreve as escolhas de métodos implementados na solução do segundo trabalho da disciplina de Algoritmos e Estruturas de Dados II. De forma resumida, o problema proposto é resolvido por meio da utilização do algoritmo de Dijkstra e da estrutura de grafos.*

1. Informação Geral

O enunciado do trabalho foi disponibilizado em formato PDF por meio do repositório da disciplina (Moodle), acompanhado de uma pasta compactada contendo os casos de teste. Em linhas gerais, o problema proposto consiste em determinar o percurso de um grupo que precisa visitar uma série de pontos específicos e, ao final, retornar ao ponto de partida, calculando o total de combustível consumido durante o trajeto.

O consumo de combustível é proporcional à quantidade de movimentações realizadas no mapa, que é representado por uma matriz composta por pontos (.) e asteriscos (*). Os pontos indicam áreas transitáveis, enquanto os asteriscos representam obstáculos intransitáveis. Os locais que devem ser obrigatoriamente visitados estão marcados com números de 1 a 9. A Figura 1 mostra um exemplo de um dos casos de teste e a Figura 2 representa graficamente como o mapa em ASCII é representado na estrutura desenvolvida.

2. Escolhas Conscientes

Dentre os algoritmos apresentados ao longo da disciplina, optou-se pela utilização do algoritmo de Dijkstra, devido à sua complexidade computacional eficiente e à sua adequação para problemas que exigem a obtenção do caminho mínimo entre dois vértices, o que ocorre em diversas situações neste projeto. Quando implementado com fila de prioridade (heap), o algoritmo apresenta uma complexidade de $O((V + E) \log V)$, onde V é o número de vértices e E o número de arestas do grafo, o que o torna bastante eficiente para grafos esparsos — como é o caso deste problema.

A estrutura de grafo foi implementada por meio de uma lista de adjacência, acompanhada por uma lista de vértices e uma estrutura auxiliar denominada `hold`. Essa abordagem resultou, na prática, na planificação da matriz original em um vetor linear. Tal decisão foi motivada por afinidade pessoal com esse tipo de estrutura; no entanto, reconhece-se que, em uma eventual revisão ou otimização futura do projeto, essa escolha pode ser reavaliada.

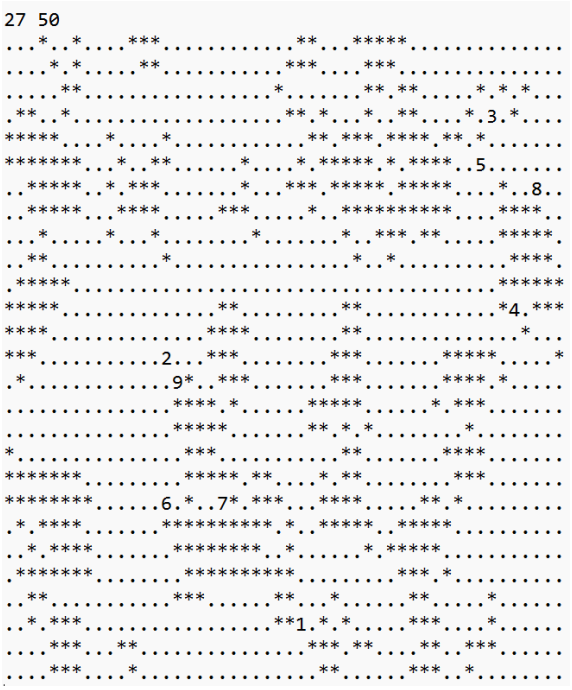


Figura 1. Exemplo de mapa representado em ASCII. Os pontos (‘.‘) indicam áreas transitáveis; os asteriscos (‘*‘) representam obstáculos. Os números (‘1‘ a ‘9‘) são os pontos obrigatórios a serem visitados.

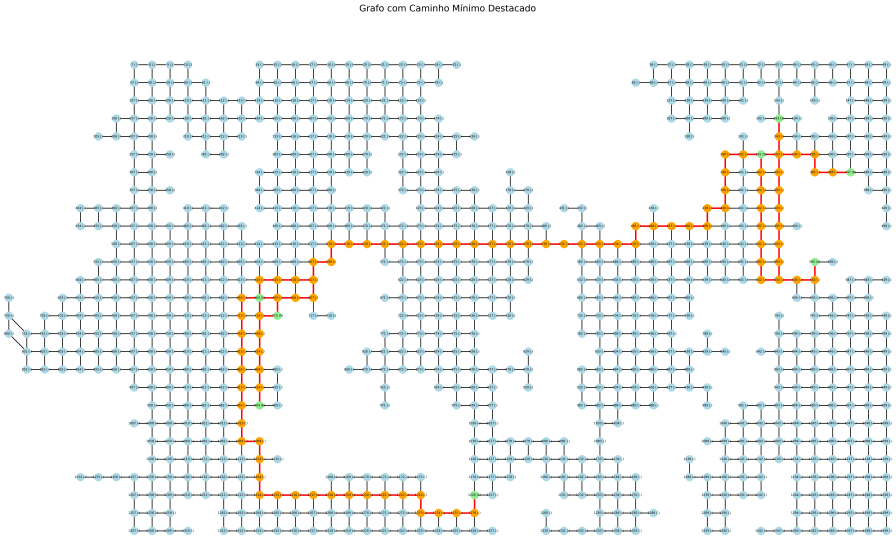


Figura 2. Grafo gerado a partir do mapa, com o caminho ótimo destacado. Os vértices representam os pontos acessíveis e as arestas indicam possíveis movimentações.

A classe que representa as arestas foi implementada de forma simples, tendo como principal função armazenar a conexão entre dois vértices. No contexto deste trabalho, não há necessidade de tratar as arestas como direcionais, tampouco de atribuir pesos variados a elas. Embora a estrutura da classe ainda comporte um atributo de peso — por questões de generalidade e flexibilidade — esse valor é fixado em 1, em conformidade com as características específicas da solução adotada.

Para a fila de prioridade utilizada no algoritmo de Dijkstra, foi implementada uma classe chamada `MinPriorityQueue`, construída sobre a biblioteca `heapq` da linguagem Python. Esta estrutura personalizada foi projetada para suportar a operação de atualização dinâmica de prioridade — fundamental para a eficiência do algoritmo — algo que uma *heap* tradicional não permite diretamente.

A `MinPriorityQueue` mantém um dicionário auxiliar chamado `entry_finder`, que permite acesso constante aos elementos para remoção ou atualização de suas prioridades, além de um contador interno para garantir a ordenação estável dos elementos com mesma prioridade. Os principais métodos implementados são:

- `push(item, priority)`: insere um item com a prioridade fornecida, substituindo entradas antigas, se existirem;
- `pop()`: remove e retorna o item com menor prioridade dentre os ativos;
- `decrease_key(item, new_priority)`: reinserção forçada de um item com nova prioridade;
- `contains(item)` e `is_empty()`: consultas auxiliares para verificação de existência e estado da fila.

Essa estrutura garantiu o comportamento esperado do algoritmo de Dijkstra, permitindo que a fila se mantivesse eficiente mesmo em casos de múltiplas atualizações de custo de caminho.

3. Procedimento e Descrição do Ambiente de Testes

3.1. Ambiente de Execução

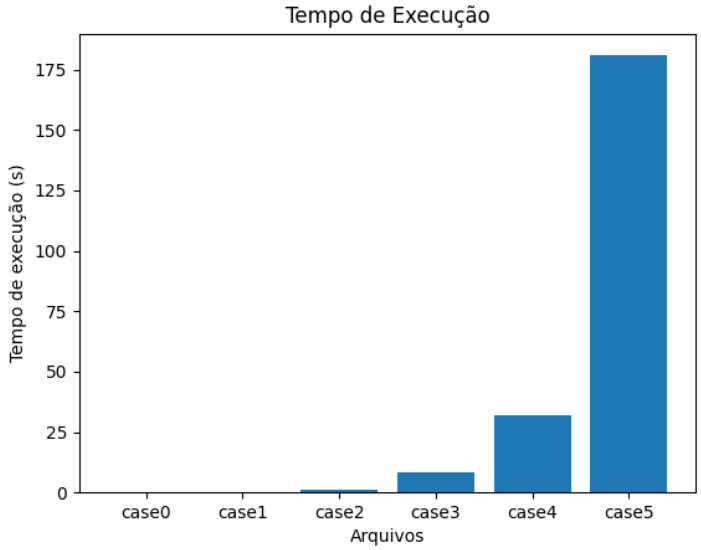
Os testes de desempenho foram realizados em um ambiente controlado com as seguintes especificações:

- **Sistema Operacional:** Windows 11 Education 24H2 (64 bits)
- **Processador:** 13th Gen Intel(R) Core(TM) i5-1345U 1.60 GHz
- **Memória RAM:** 16,0 GB (utilizável: 15,7 GB)
- **Versão do Python:** 3.10
- **Bibliotecas utilizadas:** `heapq`, `matplotlib`, `time`, `random`, entre outras.

3.2. Material de Testes

O material de teste foi fornecido juntamente com o enunciado, em formato de texto. Foram disponibilizados seis casos de teste. Cada um deles foi executado e teve o tempo de execução registrado, como mostrado na imagem 3, no formato de gráfico de barras.

Figura 3. Registro do tempo de cada caso de teste



3.3. Procedimento de Testes

Para realizar a análise empírica da complexidade do algoritmo, foram geradas entradas artificiais representando mapas de diferentes tamanhos, simulando a distribuição dos dados conforme o enunciado. Para cada valor de n, onde n representa a dimensão da matriz (com n^2 posições), o programa executou o algoritmo de descoberta do caminho mínimo e o tempo de execução foi medido utilizando a função `time.perf_counter()`.

Os tamanhos de entrada testados variaram entre 10^1 e 10^6 posições, cobrindo cenários desde pequenas matrizes até instâncias consideravelmente grandes. Cada teste foi repetido múltiplas vezes (tipicamente 5) e o tempo final registrado corresponde à média das execuções, visando minimizar o impacto de variações do sistema operacional ou do processador.

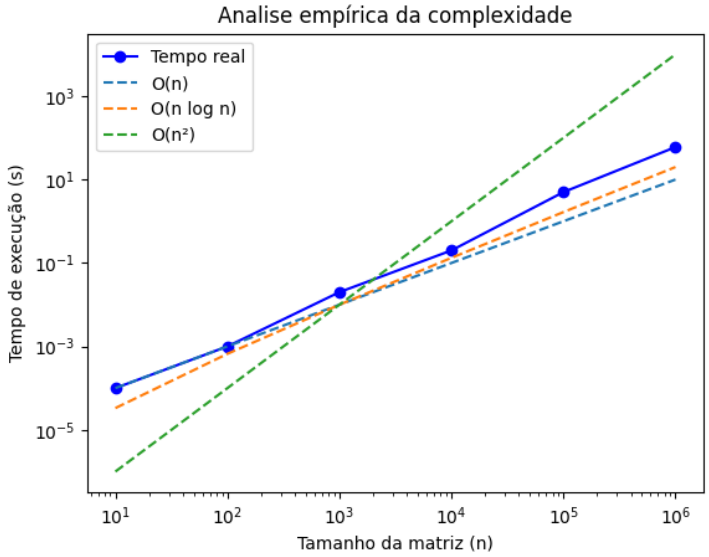
3.4. Objetivo dos Testes

O principal objetivo dos testes foi verificar se o comportamento empírico do tempo de execução estava coerente com a complexidade teórica esperada do algoritmo de Dijkstra utilizando fila de prioridade — isto é, $O((V + E) \log V)$. A comparação foi realizada com curvas teóricas de complexidade $O(n)$, $O(n \log n)$ e $O(n^2)$, conforme apresentado na Figura 4.

4. Conclusão

O desenvolvimento deste trabalho permitiu a aplicação prática de conceitos fundamentais de algoritmos e estruturas de dados na resolução de um problema inspirado em navegação por mapas. Por meio da utilização do algoritmo de Dijkstra, aliado à estrutura de grafos com lista de adjacência e a uma fila de prioridade personalizada, foi possível construir uma solução eficiente para a determinação do menor caminho entre pontos em uma matriz representando o terreno.

Figura 4. Gráfico de comparação entre a análise de complexidade e algumas curvas de complexidade conhecidas



A implementação mostrou-se compatível com a complexidade teórica esperada, conforme evidenciado pela análise empírica dos tempos de execução. O comportamento observado seguiu de forma bastante próxima à curva de $O(V \log V)$, onde $V = M \cdot N$ representa o número total de vértices da matriz planificada. Tal resultado reforça a eficácia da escolha do algoritmo e da estrutura de dados empregada, além de demonstrar a escalabilidade da solução proposta.

A fila de prioridade personalizada, construída sobre a biblioteca `heapq`, teve papel central na manutenção do desempenho, permitindo operações de atualização de prioridade de maneira eficiente — essencial para a dinâmica do algoritmo de Dijkstra.

Além disso, o trabalho proporcionou reflexões sobre decisões de modelagem, como a planificação da matriz em vetor e a organização das estruturas internas. Tais escolhas, ainda que funcionais, podem ser revistas e otimizadas em versões futuras do projeto.

Conclui-se, portanto, que o projeto atingiu seus objetivos com êxito, promovendo o aprofundamento do conhecimento em algoritmos de grafos, análise de complexidade e estruturação eficiente de dados, ao mesmo tempo em que demonstrou, por meio de testes e resultados, a validade da abordagem adotada.