PROJETO E ANÁLISE DE ALGORITMOS

ALGORITMO DE DIJKSTRA

(2025)

DANIEL SOARES OLIVEIRA SANTOS
GUSTAVO MENDES LIMA

ALGORITMO DE DIJKSTRA

Projeto de extensão apresentado à Faculdade Independente do Nordeste - FAINOR como produto final do componente curricular Projeto e Análise de Algoritmos do curso de bacharelado em Engenharia da Computação, orientado pelo Prof. Adalberto Igor.

VITÓRIA DA CONQUISTA - BA

2025

SUMÁRIO

1. Introdução

O presente relatório documenta a implementação do algoritmo de Dijkstra para resolução de problemas de caminho mínimo em grafos ponderados. O algoritmo de Dijkstra, proposto por Edsger W. Dijkstra em 1959, constitui uma das soluções mais eficientes para encontrar o caminho de menor custo entre dois vértices quando os pesos das arestas são não negativos.

Nossa implementação foi desenvolvida em linguagem C++ e contempla os três requisitos principais solicitados para esse projeto, sendo eles: capacidade de leitura de arquivos contendo a estrutura do grafo, interação com o usuário para definição dos pontos de origem e destino, e garantia de retorno do caminho ótimo. O trabalho permitiu compreender na prática os conceitos de estruturas de dados avançadas e algoritmos gulosos estudados ao longo da disciplina.

2. Fundamentação Teórica e Metodologia

2.1. Estrutura do Sistema

Organizamos a implementação em quatro módulos distintos, cada um responsável por uma etapa específica do processamento. O primeiro módulo realiza a leitura e interpretação do arquivo de entrada. O segundo valida os dados fornecidos pelo usuário. O terceiro executa o algoritmo de Dijkstra propriamente dito. O quarto módulo reconstrói e apresenta o resultado final.

Esta divisão modular facilitou tanto o desenvolvimento quanto os testes, permitindo validar cada componente de forma independente antes da integração completa.

2.2. Escolha das Estruturas de Dados

Para representação do grafo, optamos por utilizar um mapa de adjacências implementado através de estruturas map do C++. Esta escolha se justifica pela eficiência nas operações de busca e inserção, além da facilidade de representar grafos esparsos.

As principais estruturas utilizadas foram:

- **Representação do grafo:** Mapa bidimensional onde cada vértice aponta para seus vizinhos e respectivos custos
- **Vetor de distâncias:** Armazena o menor custo conhecido da origem até cada vértice
- **Mapa de predecessores:** Registra o vértice anterior no caminho ótimo, possibilitando a reconstrução do trajeto

- **Fila de prioridade mínima:** Mantém os vértices ordenados por distância, extraindo sempre o mais próximo

A fila de prioridade baseada em min-heap foi fundamental para alcançar a complexidade esperada, pois garante extração do mínimo em tempo logarítmico.

2.3. Processamento do Arquivo de Entrada

Desenvolvemos um parser específico para processar arquivos no formato JSON simplificado. O algoritmo percorre o arquivo linha a linha, identificando padrões que indicam a declaração de novos vértices ou a especificação de arestas.

Pseudocódigo do Paser:

```
FUNÇÃO carregarGrafo(nomeArquivo):
    grafo ← mapa vazio
    arquivo ← abrir("grafos/" + nomeArquivo)
    noAtual ← string vazia

    ENQUANTO existirem linhas NO arquivo:
        linha ← proximaLinha(arquivo)
        linha ← limpar(linha)  // remove espaços das pontas

        SE linha vazia OU linha = "{" OU linha = "}" ENTÃO:
            CONTINUE

        SE linha contém "\":" E linha contém "{" ENTÃO:
            // Detectou novo vértice (ex: "A": {)
            noAtual ← extrair nome entre as aspas
            grafo[noAtual] ← mapa vazio

        SENÃO SE linha contém "\":" E noAtual não vazio ENTÃO:
            // Detectou aresta (ex: "B": 4)
            destino ← extrair nome entre primeiras aspas
            custo ← extrair número após os dois pontos
            grafo[noAtual][destino] ← custo

    fechar(arquivo)
    RETORNAR grafo
```

Durante a implementação, enfrentamos desafios relacionados ao tratamento de caracteres especiais e formatação. Resolvemos estas questões através de funções auxiliares para remoção de espaços e validação de formato.

2.4. Implementação do Algoritmo de Dijkstra

Optamos por utilizar o valor constante 999999 para representar distâncias infinitas ao invés de std::numeric_limits, por questões de simplicidade e clareza no código. Este valor é suficientemente grande para os casos práticos que o sistema se propõe a resolver, mantendo o código mais legível.

Pseudocódigo principal:

```
FUNÇÃO menorCaminho(grafo, origem, destino):
    distancias ← mapa vazio
    predecessores ← mapa vazio
    fila ← nova FilaPrioridade(min-heap)

    // Inicializa todas distâncias como infinito
    PARA CADA vertice EM grafo:
        distancias[vertice] ← INF  // constante 999999

    distancias[origem] ← 0
    fila.inserir(par(0, origem))

    ENQUANTO NOT fila.vazia():
        distAtual ← fila.topo().primeiro
        u ← fila.topo().segundo
        fila.removerTopo()

        // Otimização: para se achou o destino
        SE u = destino ENTÃO:
            BREAK

        // Ignora se já processou caminho melhor
        SE distAtual > distancias[u] ENTÃO:
            CONTINUE

        // Relaxamento das arestas
        PARA CADA vizinho v DE u:
            peso ← grafo[u][v]
            nova ← distancias[u] + peso

            SE nova < distancias[v] ENTÃO:
                distancias[v] ← nova
                predecessores[v] ← u
                fila.inserir(par(nova, v))

    // Verifica se encontrou caminho
    SE distancias[destino] = INF ENTÃO:
```

```
        RETORNAR par(vetor_vazio, -1)

    // Reconstrói o caminho de trás para frente
    caminho ← vetor vazio
    atual ← destino
    ENQUANTO atual ≠ origem:
        caminho.adicionar(atual)
        atual ← predecessores[atual]
    caminho.adicionar(origem)
    reverter(caminho)

    RETORNAR par(caminho, distancias[destino])
```

Um aspecto importante da implementação foi a verificação de distância antes de processar cada vértice extraído da fila. Como o mesmo vértice pode ser inserido múltiplas vezes com distâncias diferentes, esta verificação evita processamento desnecessário.

2.5. Análise de Complexidade

A complexidade temporal da implementação é O(E log V), onde E representa o número de arestas e V o número de vértices do grafo. Este desempenho é alcançado através da fila de prioridade, que realiza inserções e extrações em tempo logarítmico.

Na prática, devido à possibilidade de inserções duplicadas na fila de prioridade (quando um vértice é atualizado múltiplas vezes), o número de operações pode chegar a O(E) inserções, resultando em O(E log E). No entanto, como $E \leq V^2$, temos que $\log E \leq 2 \log V$, mantendo a complexidade como O(E log V)

3. Caso de Teste

Para validação da implementação, utilizamos o grafo de exemplo fornecido como referência:

Arquivo: grafo.json

```
{
  "A": {
    "B": 4,
    "C": 2
  },
  "B": {
    "A": 4,
```

```
    "C": 5,
    "D": 10
  },
  "C": {
    "A": 2,
    "B": 5,
    "D": 3
  },
  "D": {
    "B": 10,
    "C": 3,
    "E": 7
  },
  "E": {
    "D": 7,
    "F": 1
  },
  "F": {
    "E": 1
  }
}
```

Este grafo apresenta 6 vértices e 10 arestas bidirecionais. Trata-se de um grafo conexo onde existe pelo menos um caminho entre quaisquer dois vértices, com pesos variando entre 1 e 10 unidades.

4. Resultados Experimentais

Realizamos uma bateria de testes para verificar a corretude da implementação em diferentes cenários.

Experimento 1: Caminho Longo (A → F)

- **Parâmetros:**
  - Vértice de origem: A
  - Vértice de destino: F
- **Resultado obtido:**
  - Caminho encontrado: A -> C -> D -> E -> F
  - Custo total do caminho: 13
- **Análise:** O algoritmo identificou corretamente a rota ótima, evitando o caminho mais custoso através de B. A decomposição dos custos parciais (A→C: 2, C→D: 3, D→E: 7, E→F: 1) confirma o resultado total de 13 unidades.

Experimento 2: Caminho Intermediário (A → D)

- **Parâmetros:**
  - Vértice de origem: A
  - Vértice de destino: D
- **Resultado obtido:**
  - Caminho encontrado: A -> C -> D
  - Custo total do caminho: 5
- **Análise:** Observamos que o algoritmo descartou adequadamente a rota A→B→D (custo 14) em favor da rota ótima A→C→D (custo 5), demonstrando a capacidade de comparar alternativas.

Experimento 3: Simetria do Grafo (F → A)

- **Parâmetros:**
  - Vértice de origem: F
  - Vértice de destino: A
- **Resultado obtido:**
  - Caminho encontrado: F -> E -> D -> C -> A
  - Custo total do caminho: 13
- **Análise:** A simetria dos resultados confirma a propriedade de reversibilidade esperada para grafos não direcionados. O custo idêntico em ambas as direções valida a consistência da implementação.

Experimento 4: Adjacência Direta (E → F)

- **Parâmetros:**
  - Vértice de origem: E
  - Vértice de destino: F
- **Resultado obtido:**
  - Caminho encontrado: E -> F
  - Custo total do caminho: 1
- **Análise:** Para vértices adjacentes, o sistema retorna apropriadamente o caminho direto, sem processamento desnecessário de rotas alternativas.

Experimento 5: Validação de Entrada

Testamos a robustez do sistema fornecendo vértices inexistentes. O programa detectou corretamente a inconsistência e solicitou nova entrada, prevenindo erros de execução.

5. Discussão

5.1. Avaliação da Corretude

A bateria de testes executada confirma que a implementação produz resultados corretos para todos os casos analisados. Realizamos verificações manuais dos custos em cada cenário, e os valores retornados pelo programa corresponderam exatamente aos cálculos teóricos.

A exploração sistemática de vizinhos em ordem crescente de distância e a atualização apropriada das distâncias mínimas demonstram que o algoritmo opera conforme especificado na literatura.

5.2. Desempenho Computacional

A escolha da fila de prioridade como estrutura central foi determinante para o desempenho. Embora não tenhamos realizado testes com grafos de grande escala, a complexidade teórica O(E log V) garante escalabilidade adequada para aplicações práticas.

A representação por mapa de adjacências também contribuiu positivamente, permitindo acesso eficiente aos vizinhos de cada vértice sem percorrer toda a estrutura.

5.3. Robustez da Solução

Implementamos mecanismos de validação em múltiplos pontos do sistema. A verificação da existência do arquivo, a validação dos vértices informados e a detecção de caminhos inexistentes demonstram preocupação com situações adversas.

Descobrimos e corrigimos diversas instâncias extremas durante o desenvolvimento, incluindo consultas com a mesma origem e destino e grafos com um único vértice.

5.4. Limitações Observadas

Apesar dos resultados satisfatórios, reconhecemos algumas limitações:

- **Restrição de pesos:** O algoritmo de Dijkstra não opera corretamente na presença de arestas com peso negativo. Para tais casos, algoritmos como Bellman-Ford seriam mais apropriados.
- **Formato de entrada:** O parser desenvolvido é específico para o formato JSON apresentado. Variações significativas na estrutura do arquivo podem causar falhas na interpretação.
- **Grafos direcionados:** A implementação atual assume grafos não direcionados. Adaptações seriam necessárias para processar adequadamente grafos direcionados.

5.5. Aplicações Práticas

Ao longo da investigação, descobrimos diversas aplicações práticas para o algoritmo desenvolvido. Variantes dessa técnica são utilizadas por sistemas de navegação para determinar rotas. Ideias semelhantes são utilizadas por protocolos de roteamento de redes de computadores para identificar as rotas de menor latência. O método é utilizado em sistemas de busca de caminhos para movimentação inteligente de agentes em jogos digitais.

6. Conclusão

Os experimentos comprovaram a precisão da solução, mostrando que ela pode lidar com diversas topologias de grafos e retornar de forma confiável o caminho de menor custo. A análise de complexidade verifica se a implementação funciona bem o suficiente para usos no mundo real.

O desenvolvimento deste trabalho proporcionou compreensão aprofundada sobre greedy algorithms, estruturas de dados avançadas e técnicas de otimização. A experiência de transformar conceitos teóricos em código funcional consolidou o aprendizado adquirido na disciplina.

7. REFERÊNCIAS BIBLIOGRÁFICAS

RUSSELL, Stuart J.; NORVIG, Peter. **Inteligência Artificial: Uma Abordagem Moderna.** 4. ed. Rio de Janeiro: GEN LTC, 2022. E-book. ISBN 9788595159495.

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. **Algoritmos: Teoria e Prática.** 3. ed. Rio de Janeiro: Elsevier, 2012..