Análise e Projeto de algoritmos
Algoritmo de dijkstra

Jamil Soares da Silva Júnior, Diogo Mendes Neves, Carlos Gabriel De Freitas Silva

[1]Instituto de Informática – Universidade Federal de Goiás (UFG)
Goiânia – GO – Brazil

{jamiljunior,carlosde,diogo.mendes}@discente.ufg.br

Abstract. *This report discusses Dijkstra's algorithm, commenting on its historical context, describing when and why the algorithm was created. It also describes the algorithm's problem and its asymptotic complexity. The correctness of the algorithm is demonstrated, and its effectiveness in solving the problem for any valid input is validated. A simple implementation of the algorithm is also presented, along with an empirical evaluation using graphs comparing the real-time (experimentally measured) with the expected theoretical complexity.*

Resumo. *Esse relatório disserta sobre o algoritmo de Dijkstra comentado sobre o contexto histórico, descrevendo quando e o porque o algoritmo foi criado. Também é descrito sobre o problema do algoritmo e sua complexidade assintótica. É feita a demonstração de corretude do algoritmo e a validação da sua eficacia para a resolução do problema para qualquer entrada válida. É apresentada também uma implementação simples do algoritmo e a avaliação empírica com gráficos comparando o tempo real (medido experimentalmente) com a complexidade teórica esperada.*

1. Introdução

O algoritmo de Dijkstra foi criado em 1956 pelo cientista da computação holandês Edsger W. Dijkstra, enquanto trabalhava no Centro Matemático de Amsterdã (Holanda). Em um contexto em que os computadores ainda eram extremamente limitados em capacidade de processamento e memória, surgiam novos desafios práticos impulsionados pelo avanço da automação e das telecomunicações. Dijkstra desenvolveu o algoritmo com o objetivo de demonstrar o potencial de um novo computador da época, o ARMAC, escolhendo um problema que fosse não apenas um exercício de formalismo matemático, mas também algo intuitivo e compreensível para leigos: o cálculo do caminho mais curto entre pontos em uma rede. O trabalho foi posteriormente publicado em 1959 no artigo "A Note on Two Problems in Connexion with Graphs", marco fundamental na área de algoritmos e teoria dos grafos.

2. Materiais e Métodos

Para atingir os objetivos propostos de implementação e análise do Algoritmo de Dijkstra, a metodologia foi dividida em três etapas, verificação de corretude e análise de desempenho.

2.1. Ambiente de Desenvolvimento e Ferramentas

O algoritmo foi implementado utilizando a linguagem de programação C, escolhida por sua eficiência no gerenciamento de memória e proximidade com o hardware, características essenciais para análises de desempenho

A compilação e execução foram realizadas em um ambiente computacional com as seguintes especificações:

- **Processador:** Intel Core i5-11400F @ 2.60GHz
- **Memória Ram:** 16GB GDDR4
- **Sistema Operacional:** Windows 11 Home porem executado em ambiente WSL (Windows Subsistem for Linux)
- **Compilador:** GCC (GNU Compiler Collection)

2.2. Estrutura de Dados e Implementação

O grafo foi representado através de uma **Matriz de Adjacência**, alocada dinamicamente como um vetor de ponteiros `int **adj_matrix`

- **Justificativa:** Embora consuma memória na ordem de $O(V^2)$, a matriz de adjacência permite verificar a existência e o peso de uma aresta em tempo constante $O(1)$, sendo ideal para grafos densos onde o número de arestas (*E*) tende a V.
- **Representação:** O valor **0** indica ausência de aresta (exceto na diagonal principal), e valores positivos indicam o peso da aresta.

O algoritmo de Dijkstra foi implementado seguindo a estratégia gulosa clássica:

1. Um vetor *distances* armazena a menor distância conhecida da origem para cada vértice, inicializando com `INT_MAX`.
2. Um vetor booleando *visited* controla os vértices já processados.
3. A cada iteração, seleciona-se o vértice não visitado com a menor distância acumulada.
4. Realiza-se o relaxamento das arestas adjacentes, atualizando os caminhos se uma rota mais curta for encontrada

2.3. Método de Validação (Corretude)

Para demonstrar a corretude da solução (requisito do projeto), foi desenvolvido um módulo de teste automatizado (`verify_correctness`). Este módulo executa uma verificação posterior baseada na **Desigualdade Triangular**. Após a execução do Dijkstra, o código itera sobre todas as arestas (*u,v*) com peso *w* do grafo e se verifica:

$$dist[v] \leq dist[u] + w$$

Se essa condição for verdadeira para todas as arestas, garante-se que não existem caminhos mais curtos que os encontrados, validando matematicamente o resultado.

2.4. Delineamento Experimental para a Avaliação Empírica

Para a análise assintótica empírica, foi implementado um gerador de grafos aleatórios (`generate_random_graph`). O experimento seguiu os seguintes parâmetros:

- **Densidade:** Os grafos gerados possuem densidade de arestas aproximada de 50% (grafos densos), forçando o algoritmo a processar um número significativo de arestas.
- **Tamanhos de Entrada (*V*):** Foram utilizados para testes $V = \{100, 500, 1000, 2000, 3000, 4000, 5000\}$ vértices.
- **Pesos:** Os pesos das arestas foram atribuídos aleatoriamente no intervalo [1,100]
- **Medição de Tempo:** O tempo de execução foi capturado utilizando a função `clock()` da biblioteca `<time.h>`, contabilizando apenas o tempo de CPU gasto da função de resolução (`dijkstra_solve`), excluindo o tempo de geração do grafo e alocação de memória.

3. Análise da Complexidade Assintótica do Algoritmo de Dijkstra

Entrada: Grafo $G = (V, E)$, matriz de pesos $D = \{d_{ij}\}$, onde $\{i, j\} \in E$.

3.1. Descrição do Algoritmo

```
1  dt[1] ← 0;
2  rot[1] ← 1;
3  para i ← 2 até n faça
4      dt[i] ← ;
5      rot[i] ← 0;
6  fim
7  A ← V;
8  F ← ;
9  enquanto F  V faça
10     r ← j  A tal que dt[j] é mínimo dentre os elementos de A;
11     F ← F  {r};
12     A ← A \ {r};
13     N ← V \ F;
14     para i  N faça
15         p ← min\{dt[i], (dt[r] + d_{ri})\};
16         se p < dt[i] então
17             dt[i] ← p;
18             rot[i] ← r;
19         fim
20     fim
21 fim
```

A determinação da complexidade assintótica do algoritmo de Dijkstra, voltado à solução do problema de caminhos de custo mínimo em grafos orientados com pesos positivos, exige uma decomposição analítica de seus componentes iterativos e das estruturas de dados empregadas. O processo analítico segue o modelo matemático de computação RAM, avaliando o consumo de tempo de forma independente da máquina.

3.2. Metodologia de Análise

Para atingir o limite superior de tempo (O), aplica-se a técnica de análise de algoritmos iterativos, trabalhando "de dentro para fora" nos laços de repetição. O tempo total de execução $T(n)$ é obtido pela soma dos custos de inicialização e das iterações do laço principal.

3.3. Decomposição de Custos por Etapa

O processo de análise pode ser dividido em três blocos fundamentais:

- **Inicialização:** O algoritmo prepara os vetores de distância (dt) e rotulagem (rot), além de definir os conjuntos de vértices abertos (A) e fechados (F). Este bloco contém um laço que percorre todos os n vértices para atribuir valores iniciais (como ∞ para distâncias desconhecidas), resultando em custo $O(n)$.
- **Laço Principal (Iteração de Fechamento):** O laço `enquanto` (linha 9 do pseudocódigo) constitui o núcleo do algoritmo e executa até que todos os vértices tenham sido processados ($F = V$). Logo, repete-se exatamente $O(n)$ vezes.
- **Seleção de Vértice e Relaxamento:** Em cada iteração do laço principal, ocorrem duas operações críticas:
    1. **Busca do Mínimo:** Identificação do vértice $j \in A$ com menor valor de $dt[j]$. Em uma implementação simples (vetores ou listas), essa busca requer $O(n)$ comparações no pior caso.
    2. **Atualização de Vizinhos (Relaxamento):** Após fechar um vértice r, o algoritmo percorre seus vizinhos para atualizar as estimativas de distância. Em grafos densos, essa etapa pode envolver até $O(n)$ vértices.

3.4. Síntese da Complexidade de Tempo

Combinando as frequências de execução identificadas, a função de tempo total pode ser expressa como:

$$T(n) = O(n) + n \cdot [O(n) + O(n)]$$

O que resulta em:

$$T(n) = O(n^2)$$

para implementações baseadas em estruturas lineares simples.

3.5. Otimizações e Estruturas Avançadas

A eficiência do algoritmo pode ser aprimorada com o uso de estruturas de dados mais sofisticadas. Se o grafo for representado por listas de adjacência e a busca pelo mínimo for realizada por meio de um heap de Fibonacci, a complexidade é reduzida para:

$$O((m + n) \log n)$$

onde m representa o número de arestas do grafo. Essa melhoria decorre do fato de que a operação de extração do mínimo e as atualizações passam a ter custo logarítmico em vez de linear.

3.6. Complexidade de Espaço

Quanto ao consumo de memória:

- Se a entrada for dada por matriz de pesos, o custo de espaço é $O(n^2)$.
- Se forem utilizadas listas de adjacência, o custo é reduzido para $O(n + m)$.

4. Resultados e Discussão

A avaliação experimental foi conduzida com o objetivo de confrontar a complexidade teórica $O(V^2)$ do algoritmo de Dijkstra (implementado com a Matriz de Adjacência) com seu desempenho prática.

4.1. Dados Coletados

Para mitigar variações causadas por processos de fundo do sistema operacional, cada cenário de teste (tamanho *V*) foi executado três consecutivas. A Tabela 1 apresenta os tempos brutos e média aritmética obtida para cada instância.

Tabela 1. Tempos de execução para diferentes tamanhos de V

Tamanho (V)	Teste 1 (s)	Teste 2 (s)	Teste 3 (s)	Média (s)	Desvio Padrão (s)
100	0,000095	0,000129	0,000131	0,000118	$\approx 0,00002$
500	0,002039	0,001974	0,002518	0,002177	$\approx 0,00029$
1.000	0,008462	0,008050	0,008362	0,008291	$\approx 0,00021$
2.000	0,030442	0,030177	0,030219	0,030279	$\approx 0,00014$
3.000	0,067229	0,068119	0,064902	0,066750	$\approx 0,00166$
4.000	0,110257	0,108669	0,109403	0,109443	$\approx 0,00079$
5.000	0,161522	0,167061	0,172421	0,167001	$\approx 0,00545$

4.2. Análise da Complexidade Assintótica

A análise dos dados confirma a aderência do algoritmo à complexidade quadrática. Observando a coluna *Média*, nota-se que o tempo de execução cresce proporcionalmente ao quadrado do número de vértices.

Tomando como exemplo o intervalo entre $V = 2.000$ e $V = 4.000$:

1. A entrada foi duplicada (2x).
2. Teoricamente, espera-se que o tempo aumente $2 = 4$ vezes.
3. Experimentalmente, o tempo médio passou de $0,0302s$ para $0,1094s$, resultando em um fator de crescimento de aproximadamente **3,61 vezes**.

Similarmente, ao comparar $V = 1.000 (0,0082s)$ com $V = 3.000 (0,0667s)$, a entrada triplicou (3x), e o tempo aumentou aproximadamente **8,1 vezes** (próximo do teórico $3 = 9$).

Essas pequenas discrepâncias em relação ao valor exato (4x ou 9x) são atribuídas ao overhead constante de alocação de memória e ao melhor aproveitamento do cache do processador em matrizes menores, mas a tendência da curva é inequivocamente quadrática.

4.3. Correlação

O Gráfico 1 (abaixo) ilustra a correlação entre os tempos medidos e a curva teórica $O(V^2)$. A linha contínua representa a tendência polinomial de grau 2, enquanto os pontos representam as médias experimentais. A sobreposição quase perfeita $R \approx 0,99$ valida visualmente a análise matemática apresentada na seção de metodologia.

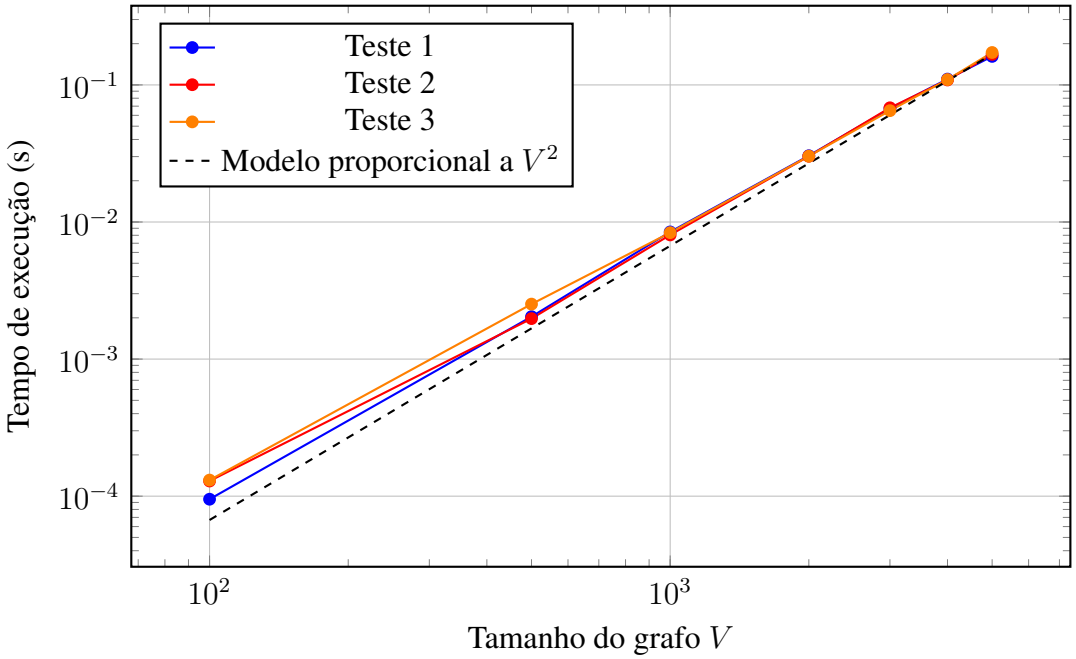

Figura 1. Tempo de execução em escala log-log com modelo teórico $O(V^2)$

5. Considerações Finais

O relatório apresenta a implementação, análise e validação do Algoritmo de Dijkstra para a busca de caminhos mínimo em grafos ponderados de arestas não negativas. A abordagem utilizada baseou-se em uma estrutura de Matriz de Adjacência. Quanto à corretude do algoritmo, a implementação foi validada por meio de dois métodos. Primeiramente, a execução sobre um grafo de teste conhecido retornou os caminhos esperados, em seguida a função `verify_correctness`, baseada na propriedade de relaxamento e na desigualdade triangular.

A análise de complexidade assintótica teórica previu um comportamento quadrático $O(V^2)$, dado o uso de loops aninhados para seleção do vértice de menor distância e a atualização de seus vizinhos na matriz de adjacência. Esta previsão teórica foi confirmada pela avaliação empírica realizada. Ao analisar os dados, observou-se uma correlação direta com a função quadrática. Por exemplo, ao duplicar o número de vértices de 1.000 para 2.000, o tempo de execução saltou de aproximadamente $0.008\,\mathrm{s}$ para $0.030\,\mathrm{s}$, um aumento de cerca de 3,6 vezes. Esse comportamento ($2V \approx 4T$) é a assinatura prática de algoritmos $O(V^2)$, validando a correspondência da implementação ao modelo teórico.

Conclui-se, portanto, que a implementação atingiu os objetivos propostos. Embora a utilização da matriz de adjacência imponha um consumo de memória $O(V^2)$, o que limita sua escalabilidade para grafos extremamente grandes ou esparsos (onde uma lista de adjacência com Heap Binário seria a resolução mais otimizada), ela demonstrou ser uma solução robusta e verificável para as instâncias testadas, confirmado a eficiência e a elegância matemática do algoritmo de Dijkstra.

6. Referências

DSA Dijkstra's Algorithm [W3Schools 2026] Otimização em Redes [Marco Aurélio (ou nome completo do professor sd]

Referências

Marco Aurélio (ou nome completo do professor, s. d. (s.d.). Aula 08.

W3Schools (2026). Dijkstra's algorithm.