UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO DE CIÊNCIAS EXATAS, NATURAIS E DA SAÚDE
DEPARTAMENTO DE COMPUTAÇÃO

GUILHERME CONRADO DA SILVA RODRIGUES

ANÁLISE DE DESEMPENHO DO ALGORITMO DE DIJKSTRA COM DIFERENTES ESTRUTURAS DE DADOS

ALEGRE - ES
2023

GUILHERME CONRADO DA SILVA RODRIGUES

ANÁLISE DE DESEMPENHO DO ALGORITMO DE DIJKSTRA COM DIFERENTES ESTRUTURAS DE DADOS

Trabalho de conclusão de curso apresentado ao Departamento de Computação do Centro de Ciências Exatas, Naturais e da Saúde da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.
Orientador: Prof. Dr. Edmar Hell Kampke.

ALEGRE - ES
2023

GUILHERME CONRADO DA SILVA RODRIGUES

Análise de desempenho do algoritmo de Dijkstra com diferentes estruturas de dados

Trabalho de Conclusão de Curso apresentado ao Departamento de Computação do Centro de Ciências Exatas, Naturais e da Saúde da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 15 de dezembro de 2023.

COMISSÃO EXAMINADORA

Prof. Dr. Edmar Hell Kampke
Universidade Federal do Espírito Santo
Orientador

Prof. Dr. Dayan de Castro Bissoli
Universidade Federal do Espírito Santo

Prof. Dr. Rodrigo Freitas Silva
Universidade Federal do Espírito Santo

À minha querida mãe, Maria de Lourdes, cujo sonho de ver seu filho formado agora se torna realidade. Expresso minha profunda gratidão por sempre acreditar que a educação tem o poder de transformar o mundo e por incutir em mim a convicção de que seus filhos têm a capacidade de alcançar tudo aquilo que almejam.

AGRADECIMENTOS

Primeiramente, expresso minha profunda gratidão ao meu orientador, Prof. Dr. Edmar Hell Kampke, pela sua amizade, orientação astuta e preocupação constante. Seu apoio inestimável e encorajamento contínuo foram fundamentais em toda a minha jornada acadêmica. Tenho a certeza de que, sob sua tutela, todos os seus alunos se tornam seres humanos melhores.

À minha querida família, especialmente aos meus pais e minha irmã, meu mais sincero agradecimento por todo o amor incondicional e incentivo constante. Vocês sempre foram e sempre serão a base da minha vida.

Um agradecimento especial aos amigos da OPTSOL, cujos desafios constantes e provocações contribuíram significativamente para a conclusão desta etapa. A importância de cada um de vocês neste processo é inestimável.

À minha esposa, dedico um agradecimento muito especial. Seu apoio inabalável e a capacidade de mostrar que, apesar dos desafios da vida de pai, marido e empresário, é fundamental concluir os ciclos que iniciamos. Obrigado por compartilhar a vida comigo e por ser minha parceira constante.

Por fim, agradeço a todos que, direta ou indiretamente, participaram desta jornada e contribuíram para minha formação. Este trabalho marca o fim de um ciclo de 14 anos repleto de histórias e amizades valiosas. A cada um de vocês, meu mais sincero agradecimento.

RESUMO

Este trabalho de conclusão de curso aborda uma análise detalhada do algoritmo de Dijkstra, que é um método fundamental para resolver o problema do menor caminho em grafos. Focando em diferentes estruturas de dados, com ênfase nas variações *Bucket* e *Heap* Mínimo, o estudo compara a eficiência dessas estruturas na aplicação do algoritmo de Dijkstra. O trabalho inicia com uma revisão bibliográfica abrangente, explorando a origem e a importância da teoria dos grafos e ressaltando contribuições significativas para a área. Centralmente, o estudo envolve a implementação do algoritmo de Dijkstra e suas variações, seguida de uma análise comparativa de desempenho com base em instâncias pré-definidas. Os resultados alcançados revelam diferenças notáveis na eficiência entre as estruturas de dados, proporcionando *insights* importantes sobre a aplicabilidade do algoritmo. Contribuindo para a teoria dos grafos e a computação teórica, o trabalho oferece uma compreensão aprofundada sobre as implicações práticas das escolhas de estruturas de dados no desempenho do algoritmo de Dijkstra.

Palavras-chave: Algoritmo de Dijkstra, Teoria dos Grafos, Estruturas de Dados, Análise de Desempenho.

ABSTRACT

This final work addresses a detailed analysis of Dijkstra's algorithm, which is a fundamental method for solving the shortest path problem in graphs. Focusing on different data structures, with an emphasis on Bucket and Minimum Heap variations, the study compares the efficiency of these structures in applying Dijkstra's algorithm. The work begins with a comprehensive literature review, exploring the origin and importance of graph theory and highlighting significant contributions to the area. Centrally, the study involves the implementation of Dijkstra's algorithm and its variations, followed by a comparative performance analysis based on pre-defined instances. The results achieved reveal notable differences in efficiency between data structures, providing important information about the applicability of the algorithm. Contributing to graph theory and theoretical computing, the work offers a deep understanding of the practical implications of data structures choices in the performance of Dijkstra's algorithm.

Keywords: Dijkstra Algorithm, Graph Theory, Data Structures, Performance Analysis.

LISTA DE FIGURAS

Figura 1 Exemplo Dijkstra Normal 23
Figura 2 Exemplo Dijkstra Normal – Caminho Mínimo 25
Figura 3 Exemplo Dijkstra Heap Mínimo 27
Figura 4 Exemplo Dijkstra Heap Mínimo - Caminho Mínimo 28
Figura 5 Exemplo Dijkstra Bucket 31
Figura 6 Exemplo Dijkstra Bucket - Caminho Mínimo 32

LISTA DE TABELAS

Tabela 1 Configuração do Ambiente de Testes 33
Tabela 2 Instâncias Utilizadas 34
Tabela 3 Resultados 34

SUMÁRIO

1 INTRODUÇÃO 12

1.1 O problema e sua importância 12

1.2 Objetivos 13

1.2.1 Objetivo geral 13

1.2.2 Objetivos específicos 13

2 REVISÃO DA LITERATURA 14

2.1 Origem da Teoria dos Grafos 14

2.2 Definições 15

2.2.1 Autores que aplicaram Teoria dos Grafos 15

2.3 Problema do menor caminho 16

2.3.1 Algoritmo de Dijkstra 16

2.3.2 Estruturas de dados 16

2.3.3 Dijkstra Normal 17

2.3.4 Dijkstra Bucket 18

2.3.5 Dijkstra Heap Mínimo 18

3 METODOLOGIA 20

3.1 Dijkstra Normal 20

3.1.1 Pseudocódigo 21

3.1.2 Exemplo Prático 23

3.2 Dijkstra Heap Mínimo 25

3.2.1 Pseudocódigo 25

3.2.2 Exemplo Prático 27

3.3 Dijkstra Bucket 29

3.3.1 Pseudocódigo 29

3.3.2 Exemplo prático 31

4 RESULTADOS OBTIDOS 33

5 CONCLUSÕES.....36

6 REFERÊNCIAS BIBLIOGRÁFICAS.....37

1 INTRODUÇÃO

A Teoria dos Grafos (TG) é datada do século XVIII e foi desenvolvida no século XX, sendo uma teoria relativamente recente. Ainda no século XXI é amplamente difundida e utilizada para as mais diversas áreas da matemática. Um grafo é definida por uma conjuntura de pontos e linhas, denominados respectivamente de vértices e arestas, onde cada aresta liga um par de vértices. Quando essa conjuntura é finita e não vazia de vértices V, conectados por um conjunto de arestas A com valores, é denominada de grafo valorado G(V,A). Sendo o conjunto A formado por (u,v,w), em que u e v são vértices incidentes a aresta em questão e w é o peso/custo associado a esta aresta.

O Problema das Pontes de Königsberg é um enigma matemático clássico, solucionado por Leonhard Euler, que originou a TG. Esse enigma consistia na possibilidade de atravessar todas as sete pontes da cidade de Königsberg sem repetir nenhuma.

Um conceito importante da TG é o caminho, que consiste em uma sequência de vértices e arestas sem repetição, que une um vértice x a um vértice y do grafo. Dessa forma, a TG apresenta um conceito simples que viabiliza o uso nas mais diversas modelagens de problemas reais, tais como: logística, redes de computadores e inclusive a relação entre infectados em uma epidemia.

1.1 O problema e sua importância

Em 1959, Edsger Wybe Dijkstra apresentou o Algoritmo de Dijkstra, cujo objetivo era encontrar o menor caminho entre dois vértices de um grafo. O algoritmo de Dijkstra consiste em um processo iterativo em que, o vértice mais próximo do primeiro vértice é determinado pela primeira iteração, na segunda iteração determina-se o vértice mais próximo do último encontrado. Esse processo é repetido até que todos os vértices sejam alcançados.

As variações de estruturas de algoritmos são desenvolvidas e aplicadas com a finalidade de otimizar os algoritmos originais. Desta forma, no algoritmo de Dijkstra

as variações *Buckets* e *Heap* Mínimo, apresentam excelentes resultados quanto a eficiência e velocidade do algoritmo.

A estrutura de *Buckets*, proposta por Robert Dial torna o algoritmo muito mais rápido, quando o intervalo de pesos/valores nas arestas for pequeno. Já o *Heap* Mínimo é um tipo de estrutura em árvore que determina o menor caminho de um grafo, por meio de uma fila de prioridades. Uma fila de prioridades inicia-se na origem e posteriormente avalia-se o valor/peso de cada vértice, atribuindo a prioridade para o menor, esse processo é repetido para todos os vértices, até que o menor caminho seja encontrado.

1.2 Objetivos

A seguir são descritos o objetivo geral e os objetivos específicos relacionados a este trabalho.

1.2.1 Objetivo geral

O objetivo geral deste trabalho consiste na implementação do algoritmo de Dijkstra e suas variações de estruturas *Buckets* e *Heap* Mínimo, para a execução de instâncias complexas e pré-estabelecidas, com a finalidade de comparar a diferença que cada uma provoca na eficiência do algoritmo.

1.2.2 Objetivos específicos

a) Implementar o Algoritmo de Dijkstra;
b) Validar o Algoritmo de Dijkstra
c) Implementar o Algoritmo de Dijkstra com a variação de estrutura *Buckets*;
d) Validar o Algoritmo de Dijkstra com a variação de estrutura *Buckets*;
e) Implementar o Algoritmo de Dijkstra com a variação de estrutura *Heap* Mínimo;
f) Validar o Algoritmo de Dijkstra com a variação de estrutura *Heap* Mínimo;
g) Executar instancias pré-estabelecidas em cada algoritmo;
h) Comparar resultados;
i) Analisar os resultados.

2 REVISÃO DA LITERATURA

Em 1735, o problema das pontes de Königsberg foi formulado por Euler, o que é considerado um marco importante na história das pontes e da teoria dos grafos (BONDY; MURTY, 2008). Este problema levou Euler a desenvolver o conceito de caminho, que é considerado o marco inicial da teoria dos grafos (BOLLOBÁS, 1998). Desde então, a teoria dos grafos tem sido amplamente utilizada em várias áreas, incluindo ciência da computação, engenharia, física, biologia e química.

O objetivo desta revisão bibliográfica é apresentar a teoria dos grafos e o problema do menor caminho, um problema clássico em teoria dos grafos que consiste em encontrar o caminho mais curto entre dois vértices em um grafo ponderado. O algoritmo de Dijkstra, um dos algoritmos mais conhecidos para resolver o problema do menor caminho, bem como algumas das estruturas de dados utilizadas para implementar o algoritmo serão discutidos. Também serão abordados alguns dos autores mais conhecidos que aplicaram a teoria dos grafos em suas pesquisas, incluindo Euler, Dijkstra, Shannon e Erdős.

2.1 Origem da Teoria dos Grafos

As pontes sempre foram importantes obras de engenharia, permitindo a conexão de diferentes regiões e facilitando o transporte de pessoas e mercadorias. A história das pontes remonta à antiguidade, com a construção de pontes de pedra e madeira em diferentes partes do mundo. Com o tempo, a engenharia das pontes evoluiu, dando origem a diversos tipos de pontes, como pontes suspensas, pontes estaiadas e pontes em arco.

O problema das pontes de Königsberg, formulado em 1735 por Leonard Euler, foi um marco importante na história das pontes. O problema consistia em determinar se era possível atravessar todas as sete pontes da cidade de Königsberg, na Prússia (atual Kaliningrado, na Rússia), sem repetir uma ponte. Euler percebeu que o problema poderia ser resolvido analisando as propriedades topológicas do grafo formado pelas pontes e regiões da cidade (WATSON, 2012). Sua solução, baseada

em um conceito que ele chamou de caminho, é considerada o marco inicial da teoria dos grafos.

Desde então, a teoria dos grafos tem sido amplamente utilizada em diversas áreas, como em sistemas de roteamento de redes de computadores e em problemas de logística e transporte (DIJKSTRA, 1959).

2.2 Definições

Um grafo é uma estrutura matemática que consiste em um conjunto de vértices e um conjunto de arestas que conectam esses vértices. A teoria dos grafos, ramo da matemática que estuda as propriedades e aplicações de grafos, é utilizada em diversas áreas.

2.2.1 Autores que aplicaram Teoria dos Grafos

A teoria dos grafos tem sido aplicada em diversas áreas ao longo do tempo, e alguns dos autores mais conhecidos que utilizaram a teoria dos grafos incluem:

Leonhard Euler: é considerado o fundador da teoria dos grafos, tendo desenvolvido o conceito de caminho euleriano e o teorema de Euler, que relaciona o número de vértices, arestas e regiões de um grafo planar (HARARY, 1969).

Edsger W. Dijkstra: criador do algoritmo de Dijkstra, utilizado para encontrar o caminho mais curto em um grafo ponderado (DIJKSTRA, 1959).

Claude Shannon: pioneiro na teoria da informação e na criptografia, utilizou a teoria dos grafos para modelar problemas de comunicação e para desenvolver o conceito de entropia (SHANNON, 1948).

Paul Erdős: matemático húngaro conhecido por sua colaboração em diversos problemas matemáticos, sendo considerado o autor com o maior número de colaborações na história da matemática. Erdős trabalhou em diversos problemas da teoria dos grafos, incluindo o problema da coloração de grafos (BOLLOBÁS, 1998).

2.3 Problema do menor caminho

O problema do menor caminho é um problema clássico em teoria dos grafos que consiste em encontrar o caminho mais curto entre dois vértices em um grafo ponderado. Existem diversos algoritmos para resolver o problema do menor caminho, sendo um dos mais conhecidos o algoritmo de Dijkstra (HARARY, 1969).

2.3.1 Algoritmo de Dijkstra

O algoritmo de Dijkstra é um algoritmo utilizado para encontrar o caminho mais curto entre dois vértices em um grafo ponderado. O algoritmo utiliza a técnica de relaxação de arestas, em que a distância de um vértice é atualizada se um caminho mais curto para esse vértice é encontrado.

O algoritmo de Dijkstra tem complexidade $O((|A| + |V|) \log |V|)$, onde $|A|$ é o número de arestas do grafo e $|V|$ é o número de vértices. O algoritmo é considerado o mais eficiente para encontrar o caminho mais curto em grafos ponderados.

2.3.2 Estruturas de dados

O desempenho do algoritmo de Dijkstra pode ser significativamente afetado pela escolha da estrutura de dados utilizada para armazenar os vértices e suas distâncias. Algumas das estruturas mais comuns são:

Lista de adjacências: cada vértice do grafo é representado por uma lista que contém seus vizinhos e as arestas que os ligam. Essa estrutura é fácil de implementar, mas pode ser menos eficiente em grafos densos.

Matriz de adjacências: o grafo é representado por uma matriz booleana que indica se há ou não uma aresta ligando cada par de vértices. Essa estrutura é eficiente em grafos densos, mas pode ser menos eficiente em grafos esparsos.

Fila de prioridades: a fila de prioridades é uma estrutura de dados que armazena os vértices a serem visitados em ordem de prioridade, com base em sua distância

estimada. Essa estrutura é muito eficiente em grafos densos e esparsos, mas pode ser mais complexa de implementar.

Heap binário: é uma árvore binária em que cada nó tem um valor associado, e o nó pai sempre tem um valor menor ou igual aos seus filhos. É uma estrutura eficiente para encontrar o mínimo em uma lista de valores, e pode ser usada para implementar a fila de prioridades.

Bucket: é uma estrutura de dados que divide o intervalo das distâncias em vários intervalos menores, chamados de *buckets*. Cada *bucket* armazena os vértices cujas distâncias se encontram dentro do intervalo do *bucket*. Essa estrutura é particularmente eficiente em grafos com distâncias muito dispersas.

2.3.3 Dijkstra Normal

O algoritmo de Dijkstra é um algoritmo clássico de caminho mais curto, que utiliza uma fila de prioridades para selecionar o vértice mais próximo. Segundo Cormen *et al*. (2009), o algoritmo começa com um vértice origem e atribui a ele uma distância 0. Em seguida, atualiza as distâncias dos seus vizinhos e coloca esses vizinhos na fila de prioridades. O algoritmo repete esse processo para cada vértice da fila, selecionando sempre o vértice mais próximo. Quando o último vértice é selecionado, o algoritmo termina e retorna o caminho mais curto.

Entretanto, a versão convencional do algoritmo de Dijkstra, apesar de sua ampla aplicação, pode ser menos eficiente em grafos densos devido ao elevado número de operações necessárias para atualizar as distâncias dos vértices vizinhos. Além disso, a implementação do algoritmo com uma fila de prioridades baseada em lista ordenada pode resultar em tempos de execução mais longos em grafos de maior tamanho. Cormen *et al*. (2009) também discutem que variações na estrutura de dados utilizada para a fila de prioridades podem impactar significativamente a eficiência do algoritmo em diferentes contextos.

2.3.4 Dijkstra Bucket

O Dijkstra *Bucket* é uma variação do algoritmo de Dijkstra que utiliza uma estrutura de dados chamada de *bucket* para armazenar os vértices a serem visitados. Cormen *et al*. (2009) explicam que, nessa estrutura, os vértices são armazenados em “baldes’’ com distâncias pré-definidas, e quando um vértice é atualizado, ele é movido para o balde correspondente à sua nova distância.

Esta adaptação do algoritmo de Dijkstra é particularmente benéfica para grafos esparsos, isto é, grafos com um número reduzido de arestas. Isso se deve à capacidade do algoritmo de pular baldes vazios, otimizando o tempo de processamento. Kleinberg e Tardos (2006) observam que variações como o Dijkstra *Bucket* podem oferecer vantagens em termos de simplicidade de implementação quando comparadas a outras abordagens, como o Dijkstra com *Heap* Mínimo, especialmente em situações em que os grafos apresentam características específicas que se alinham com a metodologia do *bucket*.

2.3.5 Dijkstra Heap Mínimo

O algoritmo de Dijkstra com a estrutura de dados *Heap* Mínimo é uma variação do algoritmo de Dijkstra que utiliza uma fila de prioridades baseada em uma árvore binária, chamada *Heap*. Segundo Cormen *et al*. (2009), a principal vantagem do *heap* em relação à lista ordenada é que a inserção e a remoção de elementos são realizadas em tempo O(log n), em vez de O(n) na lista ordenada, sendo n o número de vértices do grafo.

Ao inserir um vértice na fila de prioridades, a distância desse vértice é armazenada no *heap,* que é reorganizado de forma a manter a propriedade de que o vértice com a menor distância esteja no topo da fila. Quando um vértice é removido do *heap*, os vértices adjacentes a ele são relaxados e a distância desses vértices é atualizada, se necessário. Se a distância de um vértice relaxado for atualizada, ele é reinserido na fila de prioridades, substituindo sua versão anterior.

De acordo com Sedgewick e Wayne (2011), o algoritmo de Dijkstra com *Heap* Mínimo é considerado o mais eficiente em termos de tempo de execução, pois permite inserções e remoções em tempo O(log n) e, portanto, é mais rápido do que a fila de prioridades baseada em lista ordenada ou o algoritmo com *Buckets*. No entanto, essa eficiência vem com um custo em termos de espaço de armazenamento, já que a árvore binária (*heap*) requer mais memória do que as outras estruturas de dados.

3 METODOLOGIA

O algoritmo de Dijkstra, desenvolvido por Edsger W. Dijkstra em 1959, é amplamente reconhecido e utilizado na teoria dos grafos e otimização de caminhos. Originalmente concebido para determinar o caminho mais curto em um grafo ponderado direcionado, sua aplicabilidade se estende a diversas áreas, como redes de computadores, logística e planejamento de rotas. A contribuição de Dijkstra vai além da mera criação de um algoritmo está entrelaçada com o desenvolvimento da teoria dos grafos e da computação teórica como um todo (DIJKSTRA, 1959).

Edsger W. Dijkstra, um matemático e cientista da computação holandês, foi um dos pioneiros no campo da teoria dos grafos. Além do algoritmo de Dijkstra, ele contribuiu significativamente em áreas como semântica de programação e algoritmos de exclusão mútua. Sua abordagem meticulosa e visão abrangente influenciaram gerações de pesquisadores na área de algoritmos e estruturas de dados.

A criação do algoritmo de Dijkstra ocorreu enquanto Dijkstra trabalhava na Eindhoven University of Technology, na Holanda. A motivação para desenvolvê-lo surgiu da necessidade de otimizar o roteamento em redes de computadores, uma área em constante crescimento na época. Desde então, o algoritmo de Dijkstra tornou-se uma ferramenta fundamental em algoritmos de caminho mais curto, encontrando aplicações em uma vasta gama de domínios, desde logística e transporte até sistemas de navegação e redes de comunicação.

A história do algoritmo de Dijkstra é inseparável do desenvolvimento da teoria dos grafos e da computação teórica. Sua contribuição não apenas revolucionou a forma como lidamos com problemas de otimização em grafos, mas também influenciou o pensamento e o desenvolvimento de algoritmos em geral (CORMEN *et al*., 2009).

3.1 Dijkstra Normal

Nesta seção, será abordado o algoritmo de Dijkstra na sua forma padrão.

O algoritmo de Dijkstra classifica-se como um algoritmo guloso, pois em cada passo, ele busca expandir o caminho mais promissor até o momento, sem reconsiderar esta decisão. Essa abordagem garante que, ao final do processo, o caminho mais curto de um vértice origem para todos os outros vértices do grafo será encontrado (SEDGEWICK e WAYNE, 2011).

A seguir, apresenta-se o pseudocódigo do algoritmo de Dijkstra, que serve como base para a implementação e análise subsequente.

3.1.1 Pseudocódigo

Antes de apresentar o pseudocódigo do algoritmo de Dijkstra, é essencial entender as notações utilizadas para uma interpretação correta. O pseudocódigo é uma ferramenta abstrata que ajuda na visualização do algoritmo em uma forma que é mais próxima da linguagem humana do que de uma linguagem de programação específica (CORMEN *et al*., 2012; SEDGEWICK e WAYNE, 2011).

Na implementação do algoritmo de Dijkstra, adotam-se as seguintes notações para facilitar o entendimento e a execução do algoritmo:

Dist[v]: Representa o custo estimado do menor caminho da origem s até o vértice v. Este valor é o melhor caminho encontrado até o momento para o vértice v.

P[v]: Denota o vértice predecessor do vértice v em um caminho de s até v. P[v] é crucial para reconstruir o caminho mínimo de s até v.

Durante a execução do algoritmo, os valores de P[v] podem não indicar necessariamente os caminhos mínimos. Contudo, ao final do processamento, P[] conterá efetivamente uma árvore de caminhos mínimos, e Dist[] os custos desses caminhos a partir da origem s.

O algoritmo começa com a inicialização dos predecessores e das estimativas dos custos dos caminhos mínimos:

Para todo v pertencente a V, P[v] = -1;
Dist[s] = 0;
Para todo v pertencente a V e diferente de s, Dist[v] = ∞.

O algoritmo de Dijkstra utiliza a técnica de relaxamento de aresta, que consiste em verificar se é possível melhorar o melhor caminho obtido até o vértice u passando por v (relaxamento da aresta (v,u)). Se isso for possível, então Dist[u] e P[u] devem ser atualizados. O passo de relaxamento pode, portanto, reduzir o valor de Dist[u] e atualizar o predecessor de u.

O primeiro vértice adicionado à árvore de caminhos mínimos é o vértice s. A cada passo, escolhe-se o vértice v com menor Dist[v] (estratégia gulosa), que é então adicionado à árvore de caminhos mínimos. A partir do vértice v com menor Dist[v], tenta-se relaxar as arestas (v,u), ou seja, reduzir o custo do caminho mínimo até u, passando por v.

O algoritmo termina quando todos os vértices forem inseridos na árvore de caminhos mínimos, isto é, quando não houver mais vértices com menor Dist[v] a serem escolhidos.

Com compreensão das notações e conforme apresentado por Cormen *et al*. (2009), o pseudocódigo para o algoritmo de Dijkstra é o seguinte:

```
DIJKSTRA(G, s)
  1. Inicializar Dist[] e P[] conforme as notações acima
  2. L ← conjunto de todos os vértices em G
  3. Enquanto L não está vazio
  4.    v ← vértice em L com o menor Dist[v]
  5.     Remover v de L
  6.    Para cada vértice u adjacente a v
  7.        Se u ainda estiver em L
  8.            Se Dist[u] > Dist[v] + peso(v, u)
  9.                Dist[u] ← Dist[v] + peso(v, u)
```

10. P[u] ← v
11. Retornar Dist[] e P[]

3.1.2 Exemplo Prático

Considere-se um grafo simples que possui cinco vértices (A, B, C, D, E) e as seguintes arestas com pesos: A-B(4), A-C(2), B-C(5), B-D(10), C-D(3), D-E(4), C-E(4), conforme ilustrado na Figura 1:

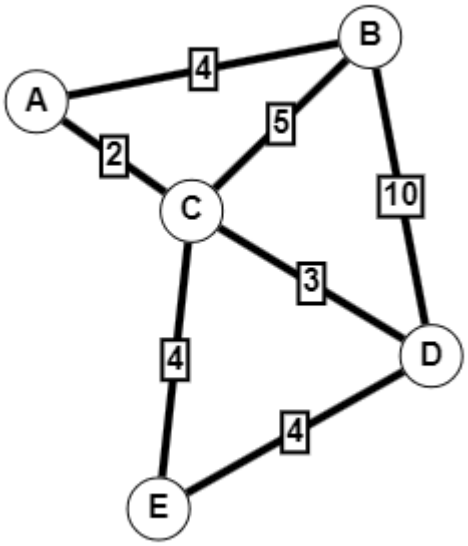

Figura 1 Exemplo Dijkstra Normal

Aplicando o algoritmo de Dijkstra começando pelo vértice A:

Passo 1: Inicialização

Dist[A] = 0, Dist[B] = ∞, Dist[C] = ∞, Dist[D] = ∞, Dist[E] = ∞

P[A] = -1, P[B] = -1, P[C] = -1, P[D] = -1, P[E] = -1

L = {A, B, C, D, E} (todos os vértices no grafo)

Passo 2: Processando A

Dist[A] = 0 (já está definido)

Relaxar A-B: Dist[B] = 4, P[B] = A

Relaxar A-C: Dist[C] = 2, P[C] = A

L = {B, C, D, E}

Passo 3: Processando C (menor Dist[] entre os vértices em L)

Relaxar C-D: Dist[D] = 5 (2 + 3), P[D] = C

Relaxar C-E: Dist[E] = 6 (2 + 4), P[E] = C

L = {B, D, E}

Passo 4: Processando B (próximo menor Dist[] entre os vértices em L)

Relaxar B-D: Dist[D] = 5 (já definido, não muda)

Relaxar B-C: Dist[C] = 2 (já definido, não muda)

L = {D, E}

Passo 5: Processando D (próximo menor Dist[] entre os vértices em L)

Relaxar D-E: Dist[E] = 6 (já definido, não muda)

L = {E}

Passo 6: Processando E (próximo menor Dist[] entre os vértices em L)

Como E é o último vértice em L e todas as arestas de saída já foram processadas, não há mais relaxamentos a serem feitos.

L = {}

Após o término do algoritmo, os vetores Dist[v] e P[v] ficariam:

Dist[v]: {A:0, B:4, C:2, D:5, E:6}

P[v]: {A: -1, B: A, C: A, D: C, E: C}

Esses vetores indicam as menores distâncias do vértice inicial A para todos os outros vértices e os predecessores correspondentes em cada caminho mínimo. Por exemplo, o caminho de A até E é determinado da seguinte maneira: E, P[E]=C, P[C]=A, ou seja, o menor caminho de A até E é: A C E.

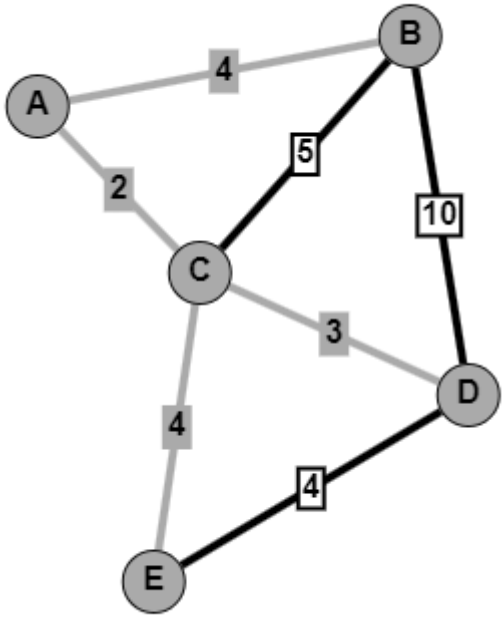

Figura 2 Exemplo Dijkstra Normal – Caminho Mínimo

3.2 Dijkstra Heap Mínimo

O algoritmo de Dijkstra pode ser otimizado através da utilização de um *heap* mínimo. Esta estrutura de dados permite a manutenção da fila de prioridade dos vértices a serem visitados, proporcionando uma maneira eficiente de extrair sempre o vértice com a menor distância estimada. O uso da *heap* mínimo reduz a complexidade de tempo do algoritmo de $O(V^2)$ para $O((V + A) \log V)$, onde V é o número de vértices e A é o número de arestas (TARJAN,1983).

3.2.1 Pseudocódigo

Na implementação do algoritmo de Dijkstra otimizado por um *heap* mínimo, são empregadas notações específicas que facilitam a compreensão e execução do algoritmo (CORMEN *et al*., 2012; SEDGEWICK e WAYNE, 2011).

G: Representa o grafo em questão, composto por um conjunto de vértices V e um conjunto de arestas A com pesos atribuídos.

s: É o vértice de origem, a partir do qual se desejam encontrar os caminhos mais curtos para todos os outros vértices no grafo.

Dist[v]: Um vetor que mantém o registro do custo mais baixo conhecido do vértice de origem s até cada vértice v no grafo.

P[v]: Denota o vértice predecessor ao vértice v em um caminho de s até v. P[v] é crucial para reconstruir o caminho mínimo de s até v.

Heap mínimo: Uma estrutura de dados prioritária que permite a remoção eficiente do elemento de menor valor, essencial para a seleção do próximo vértice a ser processado no algoritmo (SEDGEWICK e WAYNE, 2011).

Extrair-Mínimo (*Heap*): Refere-se à operação que remove e retorna o elemento com o valor mais baixo na *Heap*.

Atualizar-*Heap* (*Heap*, v, distância): Refere-se à operação que atualiza a distância de um vértice na *heap*, reorganizando-a para manter a ordem de prioridade.

A utilização dessas notações é essencial para o algoritmo, permitindo a representação abstrata dos passos computacionais que o compõem. A familiarização com essas notações é fundamental para a aplicação efetiva do algoritmo de Dijkstra com *Heap* Mínimo em problemas práticos que envolvem a determinação de caminhos mínimos (TARJAN, 1983).

O pseudocódigo para o algoritmo de Dijkstra *Heap* Mínimo é o seguinte:

DIJKSTRA_HEAP(G, s)

1. Inicializar distância de todos os vértices como ∞ e distância de s com 0
2. Inicializar todos os predecessores (P[v]) como -1
3. Inicializar a heap mínimo com todos os vértices, onde a chave de cada vértice é sua distância
4. Enquanto a heap não estiver vazia

```
v ← Extrair-Mínimo(heap)
Para cada vértice u adjacente a v
    Se Dist[u] > Dist[v] + peso(v, u)
        Dist[u] ← Dist[v] + peso(v, u)
        P[u] ← v
        Atualizar-Heap(heap, u, Dist[u])
Retornar Dist[], P[]
```

3.2.2 Exemplo Prático

Considere um grafo G com vértices A, B, C e D e as seguintes arestas e pesos: A-B(1), B-C(2), A-C(5), B-D(4), C-D(1).

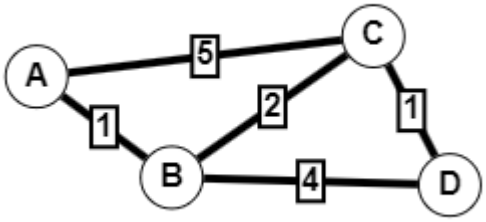

Figura 3 Exemplo Dijkstra Heap Mínimo

Passo 1: Inicialização

Dist[]: {A: 0, B: ∞, C: ∞, D: ∞}

P[]: {A: -1, B: -1, C: -1, D: -1}

Heap: {A(0), B(∞), C(∞), D(∞)}

Passo 2: Processamento do vértice A

Extrair A da heap.

Relaxar A-B: Dist[B] = 1, P[B] = A.

Relaxar A-C: Dist[C] = 5, P[C] = A.

Atualizar B e C na heap.

Heap: {B(1),C(5),D(∞)}

Passo 3: Processamento do vértice B

Extrair B da heap.

Relaxar B-C: Dist[C] = 3 (melhor que 5), P[C] = B.

Relaxar B-D: Dist[D] = 5, P[D] = B.

Atualizar C e D na heap.

Heap: {C(3),D(5)}

Passo 4: Processamento do vértice C

Extrair C da heap.

Relaxar C-D: Dist[D] = 4 (melhor que 5), P[D] = C.

Atualizar D na heap.

Heap: {D(4)}

Passo 5: Processamento do vértice D

Extrair D da heap.

Não há mais arestas para relaxar, heap vazia.

Após o término do algoritmo, os vetores Dist[v] e P[v] ficariam:

Dist[v]: {A: 0, B: 1, C: 3, D: 4}

P[v]: {A: -1, B: A, C: B, D: C}

Neste exemplo, o vetor Dist[v] exibe as menores distâncias do vértice A para todos os outros vértices, enquanto P[v] rastreia os predecessores em cada caminho mínimo. Por exemplo, o caminho mais curto do vértice A para o vértice D é identificado como A → B → C → D.

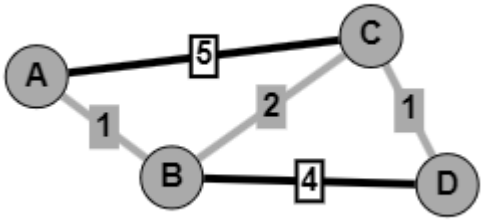

Figura 4 Exemplo Dijkstra Heap Mínimo - Caminho Mínimo

3.3 Dijkstra Bucket

A variante do algoritmo de Dijkstra que emprega *buckets* como estrutura de dados para armazenamento dos vértices foi desenvolvida para otimizar a eficiência do algoritmo tradicional. Conforme explicado por Cormen *et al*. (2009), nesta versão, os vértices são alocados em diferentes compartimentos, conhecidos como *buckets*, que correspondem a distâncias predefinidas. Quando a distância de um vértice é ajustada, ele é reposicionado para o *bucket* que reflete sua nova distância estimada.

Esta metodologia mostra-se particularmente eficaz em grafos com um número limitado de arestas, também conhecidos como grafos esparsos. Isso ocorre porque o algoritmo pode ignorar *buckets* vazios, economizando tempo no processamento. Kleinberg e Tardos (2006) destacam que, em comparação com a variação do Dijkstra que emprega *heap* mínimo, o Dijkstra *Bucket* é notável pela sua facilidade de implementação, especialmente em cenários onde a estrutura do grafo favorece essa abordagem.

3.3.1 Pseudocódigo

Antes de apresentar o pseudocódigo para o algoritmo de Dijkstra com *Bucket*, é importante compreender as notações específicas utilizadas:

G: Representa o grafo em questão, composto por um conjunto de vértices V e um conjunto de arestas A com pesos atribuídos.

s: É o vértice de origem, a partir do qual se desejam encontrar os caminhos mais curtos para todos os outros vértices no grafo.

Dist[v]: Este vetor armazena a menor distância conhecida de cada vértice a partir do vértice de origem. Na inicialização, a distância de todos os vértices é definida como infinita (representando que ainda não foram alcançados ou que a distância mínima para eles ainda não é conhecida), exceto para o vértice de origem (s), cuja distância é definida como 0, pois a distância de um vértice para ele mesmo é sempre zero.

P[v]: Este vetor mantém o registro do vértice predecessor para cada vértice no caminho de s até v. Inicialmente, todos os predecessores são definidos como "None" (ou um valor equivalente, como exemplo -a, que represente 'não definido' ou 'inexistente'), indicando que ainda não foram estabelecidos caminhos para esses vértices.

Buckets: São estruturas que agrupam os vértices de acordo com suas distâncias estimadas ao vértice de origem. Na inicialização, é criado um conjunto de *buckets*, cada um correspondendo a uma distância possível. Por exemplo, se as distâncias no grafo podem variar de 0 a um valor máximo x, são criados x+1 *buckets*, numerados de 0 a X, para cada uma dessas distâncias.

Inicialmente, todos os *buckets* estão vazios, exceto o *bucket* correspondente à distância 0, que contém o vértice de origem, já que a sua distância inicial é 0.

O pseudocódigo para o algoritmo de Dijkstra *Bucket* é o seguinte:

```
DIJKSTRA_BUCKET(G, s)
1. Inicializar distância de todos os vértices como ∞, exceto Dist[s] = 0
2. Inicializar todos os predecessores (P[v]) como "None"
3. Criar buckets para cada distância possível
4. Colocar s no bucket correspondente à sua distância inicial (bucket 0)
5. Para cada bucket
6.     Enquanto o bucket não estiver vazio
7.         Remover um vértice v do bucket
8.         Para cada vértice u adjacente a v
9.             Se Dist[u] > Dist[v] + peso(v, u)
10.                Dist[u] ← Dist[v] + peso(v, u)
11.                P[v] ← v
12.                Mover u para o bucket correspondente à sua nova distância
13. Retornar Dist[], P[]
```

3.3.2 Exemplo prático

Considere um grafo com vértices A, B, C, D e arestas com pesos inteiros: A-B(2), A-C(1), B-D(1), C-D(3).

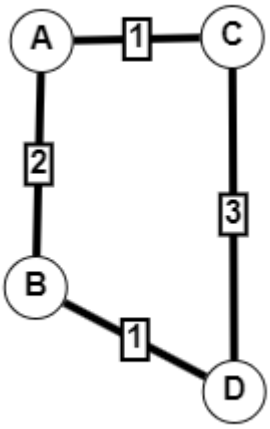

Figura 5 Exemplo Dijkstra Bucket

Passo 1: Inicialização

Dist[]: {A: 0, B: ∞, C: ∞, D: ∞}

P[]: {A: None, B: None, C: None, D: None}

Buckets: {0: [A], 1: [], 2: [], 3: []}

Passo 2: Processamento do vértice A

Remover A do bucket 0.

Relaxar A-B: Dist[B] = 2, P[B] = A, mover B para o bucket 2.

Relaxar A-C: Dist[C] = 1, P[C] = A, mover C para o bucket 1.

Buckets atualizados: {0: [], 1: [C], 2: [B], 3: []}

Passo 3: Processamento do vértice C

Remover C do bucket 1.

Relaxar C-D: Dist[D] = 4, P[D] = C, mover D para o bucket 0.

Buckets atualizados: {0: [4], 1: [], 2: [B], 3: []}

Passo 4: Processamento do vértice B

Remover B do bucket 2.

Relaxar B-D: Dist[D] = 3 (melhor que 4), P[D] = B, mover D para o bucket 3.

Buckets atualizados: {0: [], 1: [], 2: [], 3: [D]}

Passo 5: Processamento do vértice D

Remover D do bucket 3.

Não há mais arestas para relaxar.

Após o término do algoritmo, os vetores Dist[v] e P[v] ficariam:

Dist[]: {A: 0, B: 2, C: 1, D: 3}

P[]: {A: None, B: A, C: A, D: B}

Com Dist[] e P[], é possível determinar não apenas as distâncias mínimas do ponto de origem para todos os outros pontos, mas também traçar os caminhos mínimos específicos. O uso de *buckets* no algoritmo facilita o gerenciamento eficiente dos vértices e a otimização do processo de busca, especialmente em grafos onde os pesos das arestas são números inteiros relativamente pequenos.

O caminho mais curto do vértice A para o vértice D é identificado como A → B → D com custo 3.

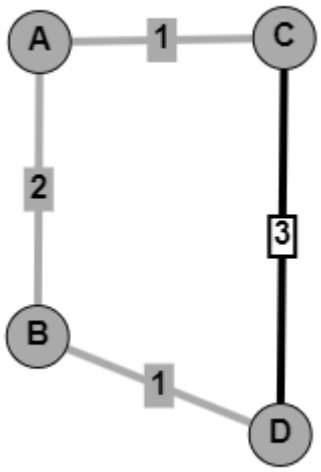

Figura 6 Exemplo Dijkstra Bucket - Caminho Mínimo

4 RESULTADOS OBTIDOS

Os algoritmos apresentados foram implementados na linguagem de programação C++ e compilados usando o compilador MinGW-W64 na versão 8.1.0.

Os testes computacionais foram realizados em um computador com as configurações apresentadas na Tabela 1.

Tabela 1 Configuração do Ambiente de Testes

Item	**Descrição**
Processador	AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz
Memória (RAM)	16,0 GB
Sistema Operacional	Windows 11 Home 22H2
Tipo de Sistema	64 bits

Neste trabalho, o objetivo era a implementação e validação do algoritmo de Dijkstra em suas formas tradicional e variadas, com especial atenção às variações de estrutura de *Buckets* e *Heap* Mínimo. Buscou-se não apenas implementar essas variantes do algoritmo, mas também submetê-las a um processo de análise e validação, com a execução de instâncias pré-estabelecidas, permitindo uma análise comparativa dos resultados obtidos.

As instâncias de teste foram obtidas no site do *9º DIMACS Implementation Challenge-Shortest Paths* (https://www.diag.uniroma1.it/challenge9/download.shtml), que possui informações sobre 12 redes de estradas dos EUA.

Foram selecionadas as quatro menores redes, sendo que para cada uma delas foram consideradas as informações de distância e tempo de viagem.

Essa comparação tem como finalidade principal determinar qual algoritmo apresenta a melhor performance em termos de velocidade. A Tabela 2 apresenta as instâncias utilizadas para testar os três algoritmos de Dijkstra e suas respectivas quantidades de vértices e arestas.

Tabela 2 Instâncias Utilizadas

Redes		
Nome	**Vértices**	**Arestas**
New York (NY)	264.346	733.846
São Francisco (BAY)	321.270	800.172
Colorado (COL)	435.666	1.057.066
Flórida (FLA)	1.070.376	2.712.798

Para cada instância foi realizada uma execução para cada versão do algoritmo de Dijkstra. A Tabela 3 apresenta o tempo de execução para cara instância e a média de tempo para obtenção dos resultados para cada versão do algoritmo.

Tabela 3 Resultados

Instâncias	**Tempo de Execução (em segundos)**		
	Dijkstra Normal	**Dijkstra H Mínimo**	**Dijkstra Buckets**
NY-d	75.857	**0.017**	0.054
BAY-d	114.212	**0.018**	0.082
COL-d	205.501	**0.025**	0.129
FLA-d	1232.133	**0.072**	0.363
NY-t	75.857	**0.016**	0.075
BAY-t	115.795	**0.017**	0.116
COL-t	204.964	**0.024**	0.210
FLA-t	1233.620	**0.070**	0.446
Média	407.242	**0.032**	0.184

Ao analisar as médias de tempo de execução, torna-se evidente que as versões do algoritmo de Dijkstra com variações nas estruturas de dados demonstram uma eficiência significativamente maior.

Na comparação individual entre as versões, observou-se que a implementação com a estrutura de *Buckets* superou em muito a eficiência do Dijkstra Normal. Além disso, ao avaliar os algoritmos com a estrutura de *Heap* Mínimo, percebeu-se que esta foi aproximadamente seis vezes mais eficiente que a estrutura de *Buckets*.

Dessa forma, o algoritmo que utiliza a estrutura de *Heap* Mínimo destaca-se como um exemplo claro do impacto positivo que uma escolha estratégica de estrutura de dados pode ter sobre o desempenho global de um algoritmo.

Este resultado reforça a ideia de que, no contexto de otimização de algoritmos, a seleção de estruturas de dados eficientes é fundamental na busca por soluções computacionais mais rápidas e eficazes.

5 CONCLUSÕES

Este estudo realizou uma análise comparativa detalhada do algoritmo de Dijkstra, utilizando diferentes estruturas de dados - especificamente as variações *Bucket* e *Heap* Mínimo - com o objetivo de avaliar a eficiência dessas estruturas na otimização do algoritmo. Através de implementações cuidadosas e uma série de testes em instâncias pré-definidas, foi possível observar diferenças significativas no desempenho das variações do algoritmo.

Os resultados obtidos demonstram que a utilização de estruturas de dados adequadas, como o *Heap* Mínimo, pode resultar em melhorias substanciais na eficiência do algoritmo de Dijkstra. A implementação com a estrutura de *Heap* Mínimo, em particular, mostrou-se notavelmente eficiente, superando a versão normal e a versão com *Buckets* em termos de velocidade nas instâncias avaliadas.

Por outro lado, a variação com *Buckets* mostrou ser mais eficiente do que a implementação tradicional do algoritmo. Isso indica que, para cenários específicos, a escolha de uma estrutura de dados mais adequada pode levar a uma melhora considerável no tempo de processamento.

Este trabalho reforça a importância da escolha estratégica de estruturas de dados na otimização de algoritmos computacionais. Ao considerar diferentes abordagens e testá-las em cenários variados, é possível identificar soluções que oferecem um bom equilíbrio entre eficiência e aplicabilidade.

Em conclusão, o estudo contribui para o campo da teoria dos grafos e da computação teórica, fornecendo *insights* valiosos sobre a aplicação do algoritmo de Dijkstra em diferentes contextos e destacando a relevância de escolhas informadas de estruturas de dados na busca por soluções computacionais eficientes.

Todo o código-fonte desenvolvido e instâncias utilizadas nesse trabalho encontram-se disponíveis no website: https://github.com/guiconrado/Djkstra

6 REFERÊNCIAS BIBLIOGRÁFICAS

BOLLOBÁS, B. Modern Graph Theory. Springer-Verlag, 1998.

BONDY, J. A.; MURTY, U. S. R. Graph theory. New York: Springer, 2008.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: Teoria e Prática. Rio de Janeiro: Elsevier, 2009.

DIJKSTRA, E. W. A note on two problems in connexion with graphs. Numerische Mathematik, v. 1, p. 269–271, 1959.

HARARY, F. Graph Theory. Addison-Wesley, 1969.

KLEINBERG, J.; TARDOS, É. Algorithm Design. Pearson, 2006.

SEDGEWICK, R.; WAYNE, K. Algorithms. Boston: Addison-Wesley Professional,2011.

SHANNON, C. E. A Mathematical Theory of Communication. The Bell System Technical Journal, v. 27, n. 3, p. 379–423, 1948.

TARJAN, R. Data Structures and Network Algorithms. Society for Industrial and Applied Mathematics, 1983.

WATSON, B. L. Euler's Konigsberg Bridge Problem: its history and modern relevance. Mathematics Magazine, v. 85, n. 2, p. 86-110, 2012.