Utilização de Grafos e Hash para soluções de desafios cotidianos

Daniel Rodrigues de Sousa

1 Resumo

O documento aborda quatro problemas de computação relacionados a estruturas de dados e algoritmos. O primeiro problema envolve a modelagem do grafo para a Torre de Hanói, utilizando algoritmos de Dijkstra e Bellman-Ford-Moore para encontrar o menor caminho. O segundo desafio é a implementação de um programa para determinar o caminho mais confiável em grafos orientados. O terceiro trata da criação e análise de tabelas de hashing para organizar dados de funcionários, com diferentes funções de dispersão e resolução de colisões. Cada problema exige a implementação eficiente e a comparação de desempenho entre soluções.

2 Introdução

Estruturas de dados e algoritmos são componentes centrais da computação, fornecendo a base para resolver problemas complexos de maneira eficiente. Este trabalho explora esses conceitos através de desafios práticos que destacam sua relevância em diversas aplicações. Desde problemas clássicos como a Torre de Hanói até a organização de bases de dados usando hashing, o documento reflete a importância de modelar e implementar soluções robustas. Esses problemas testam não apenas o entendimento teórico, mas também a habilidade de aplicar algoritmos em situações práticas.

O primeiro problema, a Torre de Hanói, é abordado com uma abordagem gráfica, modelando configurações como vértices e movimentos como arestas. Algoritmos como Dijkstra e Bellman-Ford-Moore são utilizados para encontrar o caminho mais eficiente, analisando o tempo de execução de cada um. Outro desafio apresentado é o cálculo do caminho mais confiável em um grafo orientado, destacando a necessidade de algoritmos eficientes para redes de comunicação e confiabilidade.

O trabalho também inclui o estudo de tabelas de hashing para gerenciar bases de dados de funcionários. Diferentes funções de dispersão e estratégias de resolução de colisões são implementadas e comparadas. Este exercício reforça a importância do desempenho em sistemas que manipulam grandes volumes de dados. Assim, o documento oferece uma visão abrangente de problemas computacionais, preparando o aluno para enfrentar desafios práticos e avançar no estudo das estruturas de dados e algoritmos.

3 Seções Especificas

Nesta seção, serão detalhadas as funções implementadas para cada questão desenvolvida, incluindo as estruturas criadas, quando necessário. Além disso, serão apresentadas informações técnicas sobre a máquina utilizada para realizar os testes.

3.1 Grafos

Grafos são estruturas matemáticas compostas por um conjunto de vértices (ou nós) conectados por arestas (ou arcos). Eles são amplamente utilizados para modelar relações ou conexões entre diferentes entidades em uma ampla gama de contextos, como redes de transporte, sistemas de comunicação, estruturas de dados, e até

mesmo em problemas biológicos ou sociais. Cada aresta pode conter atributos adicionais, como peso, direção, ou rótulos, que fornecem informações sobre a relação entre os vértices conectados. A representação de grafos pode ser feita de várias formas, como listas de adjacência, matrizes de adjacência ou representações baseadas em objetos.

A versatilidade dos grafos permite sua aplicação em diversos algoritmos para resolver problemas complexos, como encontrar o caminho mais curto, determinar a conectividade, identificar ciclos e calcular fluxos máximos em redes. Exemplos populares incluem o algoritmo de Dijkstra para o menor caminho e o algoritmo de Kruskal para árvores geradoras mínimas. Graças à sua capacidade de modelar relações de forma clara e eficiente, os grafos se tornaram uma ferramenta indispensável em áreas como ciência da computação, engenharia, inteligência artificial e ciência de dados.

3.2 Hash

As funções de hash são algoritmos que transformam dados de entrada de tamanho variável em uma saída de comprimento fixo, conhecida como valor hash ou resumo hash. Elas desempenham um papel crucial em diversas áreas da computação, como segurança da informação, estruturas de dados e sistemas de armazenamento. Uma boa função de hash distribui uniformemente os valores de saída, minimizando conflitos, ou colisões, em que dois dados de entrada diferentes produzem o mesmo valor hash. A eficiência de uma função de hash é avaliada com base na uniformidade da distribuição, velocidade de cálculo e resistência a ataques em contextos de segurança.

Na prática, as funções de hash são amplamente utilizadas em tabelas hash, que permitem acesso rápido a dados por meio de chaves, e em algoritmos de criptografia, onde asseguram a integridade e autenticidade de informações. Exemplos conhecidos incluem os algoritmos MD5, SHA-256 e CRC32. Apesar de sua utilidade, o uso inadequado de funções de hash, como a escolha de uma função ineficiente ou com alta probabilidade de colisões, pode comprometer o desempenho de sistemas e até mesmo sua segurança. Portanto, selecionar a função de hash apropriada é essencial para atender às necessidades específicas de cada aplicação.

3.3 Informações Técnicas

Abaixo, segue as informações do hardware utilizados para os testes em ambas as árvores:

Componente	Descrição
Processador	10th Gen Intel(R) Core(TM) i5-10300H @ 2.50GHz Até 4.50 GHz
Memória RAM	8GB DDR4
Sistema Operacional	Windows 11 23H2

Table 1: Especificações de hardware e software

3.4 Funções

Nesta subseção, são descritos os escopos das funções implementadas para as diferentes questões do projeto, com breves explicações sobre sua funcionalidade. As funções estão organizadas em subseções conforme a questão correspondente.

3.4.1 Questão 1

As funções desenvolvidas nesta questão implementam um algoritmo para resolver o problema da Torre de Hanói utilizando o método de Dijkstra.

A função abaixo executa o algoritmo de Dijkstra para encontrar o menor caminho entre duas configurações em uma matriz de adjacência:

```
void runDijkstra(int matriz[][N_CONFIGS], int start, int end);
```

Para coletar os tempos de execução para o relatório, a seguinte função executa o algoritmo de Dijkstra 100 vezes:

```
void measureTime(int matrix[][N_CONFIGS]);
```

3.4.2 Questão 2

Nesta questão, foi implementado um algoritmo baseado no método de Bellman-Ford para resolver o problema da Torre de Hanói. As principais funções são:

A função que executa o algoritmo de Bellman-Ford-Moore para encontrar o menor caminho entre duas configurações:

```
void bellmanFord(int matrix[][N_CONFIGS], int startVertex);
```

Assim como na questão anterior, esta função coleta os tempos de execução para o relatório, executando o algoritmo 100 vezes:

```
void measureExecutionTime(int matrix[][N_CONFIGS]);
```

3.4.3 Questão 3

O objetivo desta questão foi desenvolver um algoritmo para calcular a confiabilidade entre diferentes pontos de um grafo. As principais funções implementadas foram:

Esta função inicializa o grafo, atribuindo IDs aos vértices e configurando todas as arestas com peso 0.0:

```
void initializeGraph(Graph *graph);
```

A função abaixo gera o grafo, atribuindo pesos aleatórios entre 10% e 100% às arestas:

```
void generateGraph(Graph *graph);
```

Para exibir o grafo na forma de uma tabela de conexões, foi implementada a seguinte função:

```
void displayGraph(Graph graph);
```

A função a seguir aplica o algoritmo de Dijkstra para encontrar os caminhos mais prováveis a partir de um vértice inicial:

```
Distance *runDijkstra(int start, Graph graph);
```

Por fim, esta função reconstrói e exibe o caminho de maior probabilidade entre dois vértices, ou indica a inexistência de um caminho:

```
void findShortestPath(int start, int end, Distance *distances);
```

3.4.4 Questão 4

Nesta questão, foi implementado um algoritmo para calcular a quantidade de colisões em um sistema de controle de funcionários utilizando tabelas Hash. Foram criadas duas variantes: uma com 101 espaços e outra com 150. As principais funções são:

A função `functionHash1` calcula a posição inicial de armazenamento na tabela Hash:

```
int functionHash1(char registration[]);
```

Para resolver colisões geradas por `functionHash1`, utiliza-se a seguinte função:

```
int resolveCollision1(int hash, char registration[]);
```

Como alternativa, a função `functionHash2` utiliza o método de ”folding shift” para calcular posições:

```
int functionHash2(char registration[]);
```

A função abaixo resolve colisões geradas por `functionHash2`, aplicando um incremento fixo de 7 ao índice:

```
int resolveCollision2(int hash, char registration[]);
```

Por fim, a função `testHashFunction` combina uma função de Hash com sua respectiva estratégia de resolução de colisões, avaliando a eficiência do sistema:

```
void testHashFunction(int hashFunction(char[]), int resolveCollision(int, char[]));
```

4 Metodologia

Neste trabalho, foi implementada uma metodologia de caráter essencialmente prático, priorizando a validação de cada questão por meio da execução e análise de testes relacionados aos códigos desenvolvidos. Essa abordagem garantiu a verificação contínua da funcionalidade e eficiência das soluções propostas, permitindo a identificação e correção de eventuais inconsistências ao longo do processo. O projeto foi estruturado em torno do desenvolvimento de quatro questões principais, cada uma abordando diferentes desafios e exigindo estratégias específicas para sua resolução.

O primeiro desafio abordado foi a resolução do problema da Torre de Hanói, com o objetivo principal de desenvolver dois códigos distintos capazes de solucionar a mesma questão. Para isso, foram utilizados os algoritmos de Dijkstra e Bellman-Ford. Embora ambos resolvam o problema, o foco deste projeto foi validar a hipótese: ”Entre os dois algoritmos, há algum que apresenta melhor desempenho em relação ao outro”. Para verificar essa hipótese, realizou-se uma análise de desempenho cronometrando os tempos de execução de ambos. A fim de evitar valores muito baixos, os algoritmos foram executados 100 vezes consecutivas dentro de um laço de repetição for. Além disso, para assegurar maior precisão nos resultados, cada teste foi repetido 30 vezes. Os resultados obtidos estão detalhados na seção de Resultados.

O segundo desafio consistiu em calcular o número de caminhos possíveis entre dois vértices e avaliar a confiabilidade de sucesso entre dois caminhos. Para isso, foram realizados cálculos repetidos diverdas vezes utilizando valores aleatórios e distintos. Essa abordagem permitiu uma análise detalhada das possibilidades de conexão entre os vértices, além de verificar a probabilidade de sucesso associada a cada caminho avaliado. Os resultados obtidos nessa etapa também são apresentados na seção de Resultados.

Por fim, foi resolvida mais uma questão, Nesse cenário, não se trata mais de um problema de grafos, mas sim da aplicação de diferentes métodos de hashing. O objetivo nesta etapa do desenvolvimento foi comparar duas funções distintas de hash e analisar qual gerava o menor número de colisões. Para isso, foram implementados códigos com tabelas de 100 e 150 espaços, e duas funções de hash distintas foram desenvolvidas, permitindo uma comparação direta para determinar qual delas era mais eficiente no cenário proposto.

5 Resultados

Nesta seção, são apresentados os resultados obtidos com a execução das questões 1, 2, 3 e 4, organizados em subseções que refletem as temáticas abordadas. Os resultados incluem a modelagem do grafo da Torre de Hanói e a aplicação dos algoritmos de Dijkstra e Bellman-Ford para encontrar o menor caminho entre configurações, destacando o desempenho de cada abordagem. Também são exibidos os resultados do cálculo do caminho

mais confiável em um grafo orientado, evidenciando a eficiência do programa desenvolvido. Além disso, os experimentos com tabelas de hashing apresentam uma análise comparativa entre diferentes funções de dispersão e estratégias de resolução de colisões, identificando o impacto dessas escolhas no desempenho e na ocorrência de colisões.

5.1 Torre de Hanói

O clássico desafio da Torre de Hanói foi implementado em dois cenários distintos, permitindo a comparação dos tempos de execução entre os algoritmos de Dijkstra e Bellman-Ford. Os resultados obtidos são apresentados e analisados nas seções seguintes.

A Figura 1 apresenta uma comparação detalhada entre os dois algoritmos, executados conforme a metodologia descrita na seção anterior, utilizando a máquina cujas especificações estão descritas na Tabela 1. Além disso, os tempos de execução serão analisados de forma mais aprofundada em suas respectivas seções, proporcionando uma compreensão clara dos resultados obtidos.

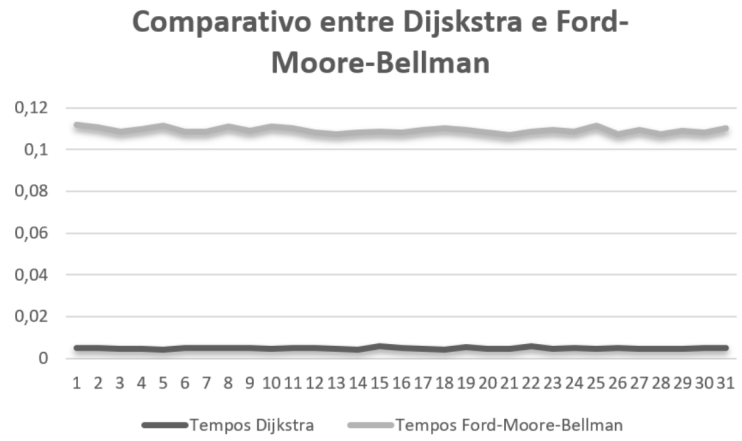

Figure 1: Comparativo entre os algoritmos Dijkstra e Bellman-Ford

5.1.1 Algoritmo de Dijkstra

A Torre de Hanói foi implementada e executada de acordo com a metodologia descrita. Após 30 execuções, o algoritmo baseado em Dijkstra apresentou um tempo médio de execução de 0,005052 segundos, com desvios mínimos, como ilustrado na Figura 2. Esses resultados demonstram a eficiência e a consistência do algoritmo nesse cenário.

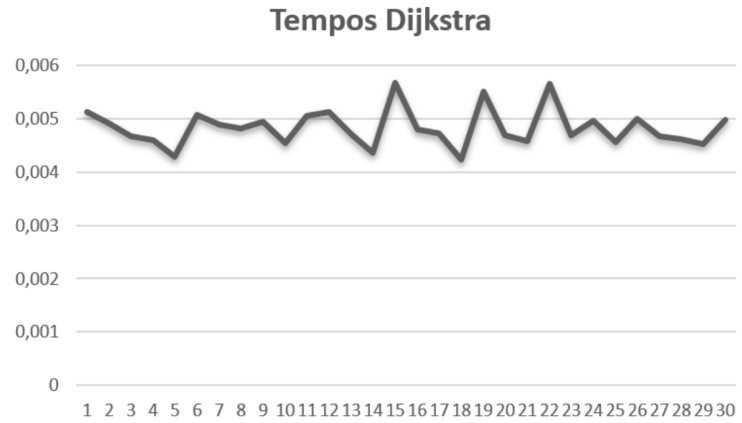

Figure 2: Tempos em segundos das execuções do algoritmo Dijkstra

5.1.2 Algoritmo de Bellman-Ford

Por outro lado, um algoritmo alternativo foi implementado para a Torre de Hanói utilizando Bellman-Ford. Nesse caso, os tempos de execução foram significativamente mais elevados, apresentando uma média de 0,1101625 segundos. Esse desempenho contrasta com o algoritmo baseado em Dijkstra, evidenciando diferenças consideráveis no custo computacional. A Figura 3 contém os tempos individuais da execução do algoritmo.

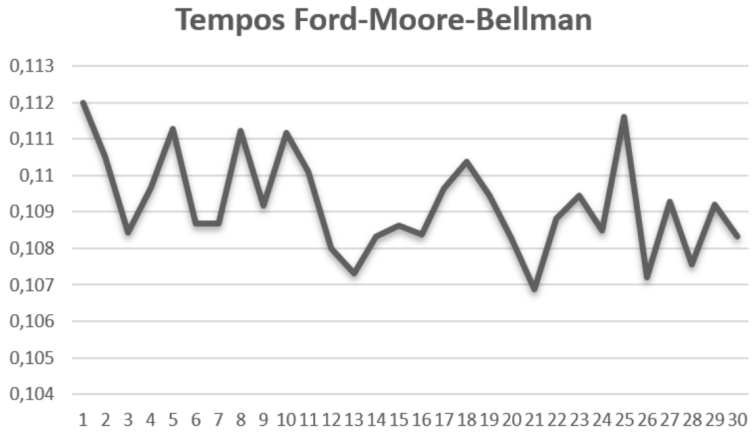

Figure 3: Tempos em segundos das execuções do algoritmo Bellman-Ford

5.2 Confiabilidade Caminhos do Grafo

Para responder à terceira questão da atividade, foi desenvolvido um código que cria e gera um grafo, permitindo testar a confiabilidade entre os caminhos dos seus nós. A partir dessas análises, o código determina e retorna o caminho mais confiável entre os vértices do grafo. Abaixo sera apresentada uma figura que foi gera com a coleção de resultados obtidos ao executar o código, a Figura 4 é referente aos caminhos possíveis a partir de cada vértice.

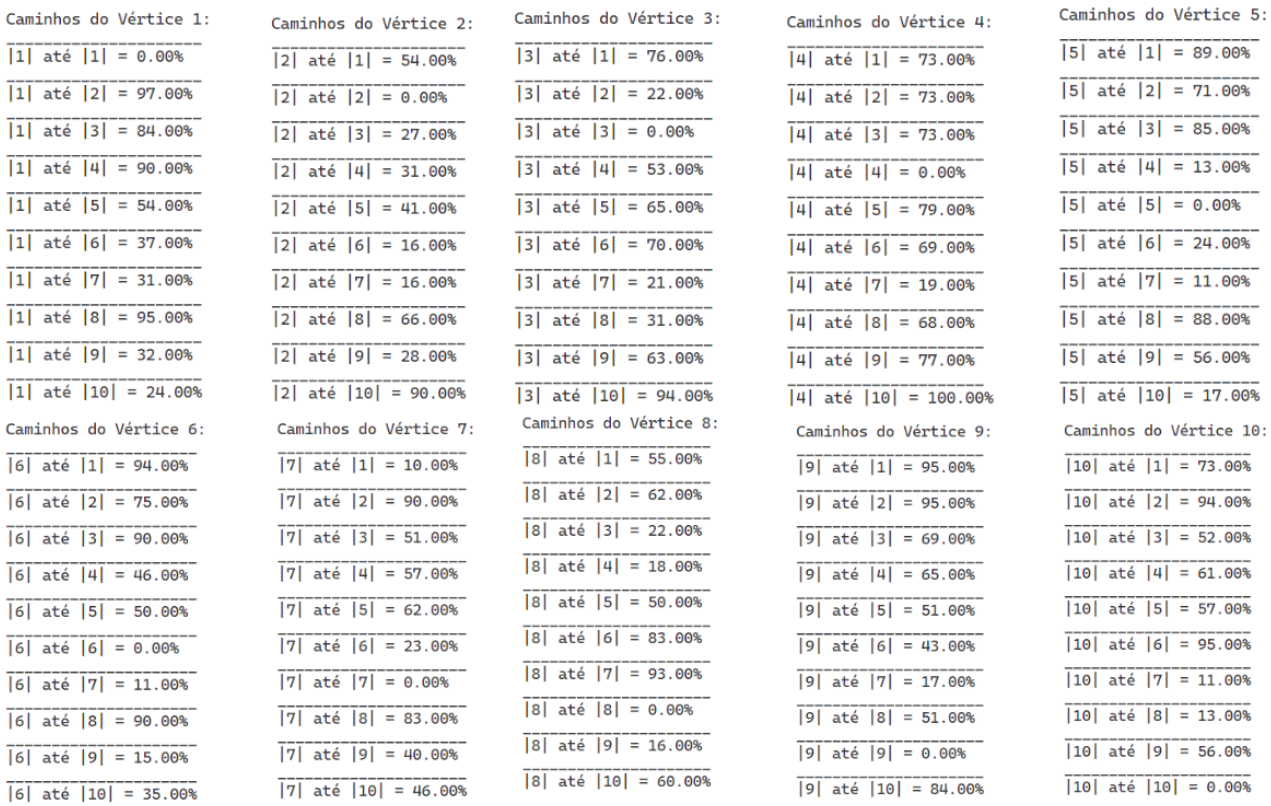

Figure 4: Confiabilidade a partir de cada vértice

É importante observar que alguns nós possuem probabilidades mais altas que outros, enquanto alguns podem ser inacessíveis diretamente a partir de um nó específico. Com base nisso, é possível calcular o menor caminho entre o nó com a menor probabilidade e o nó com a maior probabilidade. Vale ressaltar que, como o cenário é gerado de forma aleatória a cada execução do código, os resultados podem variar em cada execução. Assim, você pode obter uma saída diferente da apresentada neste estudo de caso.

```
Caminho curto entre 1 e 10: 1 -> 9 -> 6 -> 10
```

5.3 Sistema de Gerenciamento de Funcionários

Por fim, foi abordado um desafio adicional na quarta questão, que consistiu no desenvolvimento de um sistema de cadastro para mil funcionários, utilizando diferentes tipos de funções de Hash e diversos tamanhos de tabelas. Esta seção será dividida em duas partes: Função de Hash 1 e Função de Hash 2, apresentando os tempos de execução de cada uma delas.

5.3.1 Hash 1

Função de Hash 1 foi testada em dois cenários distintos: o primeiro com uma tabela de 101 posições (gerando em média 35.000 colisões) disponíveis e o segundo com 150 posições (gerando em média 53.000 colisões). A seguir, serão apresentados os tempos médios de execução para cada cenário, acompanhados de uma tabela comparativa e gráficos que visam proporcionar uma melhor compreensão do desempenho da função em cada caso.

Abaixo, segue abaixo a Tabela 2 com tempo médio de execução com 101 espaços livres

Abaixo, segue abaixo a Tabela 3 com tempo médio de execução com 150 espaços livres

Média de Tempo
0,000625

Table 2: Tabela de Média de Tempo

Média de Tempo
0,000744

Table 3: Tabela de Média de Tempo

A Figura 5 apresenta os tempos das 30 execuções, com os dados coletados durante esse período. Nela, os valores de Hash 1, referentes a 101 e 150 espaços, são exibidos lado a lado, facilitando a comparação entre os dois conjuntos de dados.

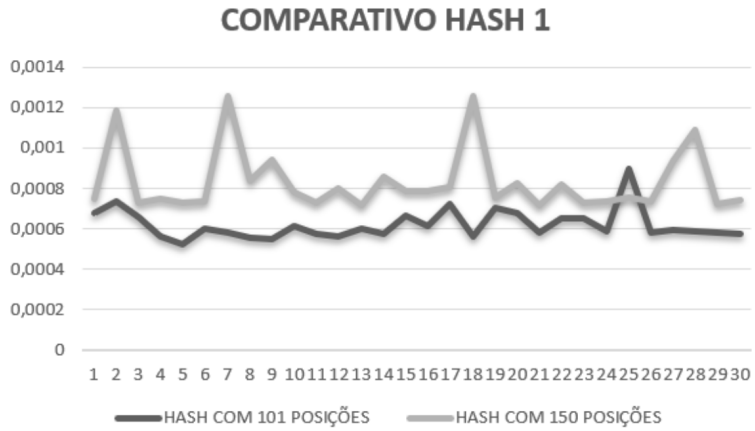

Figure 5: Testes realizados no Hash1

5.3.2 Hash 2

Esta seção apresenta uma nova função Hash. A metodologia aplicada para a coleta dos resultados permanece idêntica à descrita na seção anterior. As diferenças nos resultados observados decorrem exclusivamente das escolhas realizadas para o método Hash, que divergem em relação à abordagem utilizada anteriormente.

Abaixo, segue abaixo a Tabela 4 com tempo médio de execução com 101 espaços livres, com 7.000 colisões.

Média de Tempo
0,000307

Table 4: Tabela de Média de Tempo

Abaixo, segue abaixo a Tabela 5 com tempo médio de execução com 150 espaços livres, com 13.000 colisões.

A Figura 6 apresenta os tempos das 30 execuções, com os dados coletados durante esse período. Nela, os valores de Hash 2, referentes a 101 e 150 espaços, são exibidos lado a lado, facilitando a comparação entre os dois conjuntos de dados.

Média de Tempo
0,000232

Table 5: Tabela de Média de Tempo

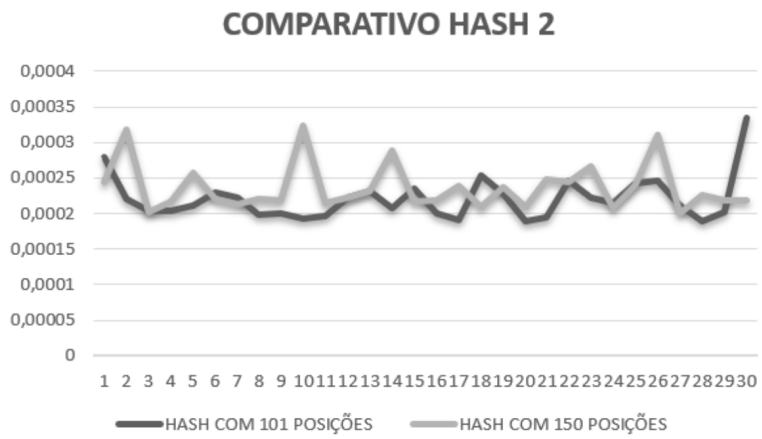

Figure 6: Testes realizados no Hash 2

6 Conclusão

O estudo de caso apresentado neste artigo explorou diferentes cenários e casos de teste, sendo apropriado elaborar conclusões específicas para cada tópico abordado.

Inicialmente, foi analisada a Torre de Hanói, implementada com os algoritmos de Dijkstra e Bellman-Ford-Moore. Os testes demonstraram que ambos os algoritmos são eficazes, alcançando os mesmos resultados. No entanto, o algoritmo de Dijkstra mostrou-se significativamente mais eficiente em comparação ao Bellman-Ford-Moore, evidenciando uma vantagem em termos de desempenho.

O segundo desafio consistiu na implementação de um algoritmo capaz de percorrer um grafo arbitrário. Para este propósito, manteve-se a utilização do algoritmo de Dijkstra, que já havia se provado eficaz e eficiente. Sua aplicação permitiu a realização dos cálculos de confiabilidade e sua exibição ao usuário final, destacando sua versatilidade e adequação ao problema.

Por fim, no desafio relacionado às funções Hash, o objetivo foi implementar um programa que avaliasse diferentes métodos de hashing e determinasse o mais eficiente. Os resultados dos testes indicaram que, no cenário proposto, o Hash 2, que utiliza a técnica Fole Shift, apresentou desempenho superior, gerando menos conflitos em ambos os casos testados, com 101 e 150 espaços de armazenamento.

Com base nesses resultados, conclui-se que grafos são ferramentas fundamentais na resolução de problemas cotidianos, e que o algoritmo de Dijkstra representa uma escolha robusta em aplicações que demandam eficiência e confiabilidade. No gerenciamento de memória e outras operações que envolvem alocação de recursos, as técnicas de hashing demonstraram ser promissoras. No entanto, a escolha da técnica adequada é crucial, uma vez que decisões inadequadas podem comprometer o desempenho do sistema, resultando em maior risco de colisões e menor eficiência.