UNIVERSIDADE FEDERAL DO PIAUÍ - UFPI
CAMPUS SENADOR HELVÍDIO NUNES DE BARROS
Curso Bacharelado em Sistemas de Informação
Disciplina: Estrutura de Dados II
Docente: Juliana Oliveira de Carvalho
Discente: Erlanny Rodrigues da Silva Rêgo

Relatório Trabalho III

Resumo

Este trabalho tem como objetivo aplicar, na prática, conceitos avançados de estruturas de dados, com ênfase em grafos e tabelas de hash, utilizando a linguagem C como ferramenta de implementação. Foram desenvolvidas soluções para problemas clássicos, como a Torre de Hanói, representada como grafo é resolvida com os algoritmos de Dijkstra e Ford-Moore-Bellman, permitindo análise de desempenho entre eles. Também foi abordado o problema de encontrar o caminho mais confiável em um grafo orientado com pesos probabilísticos nas arestas, reforçando o uso de grafos em contextos reais de decisão. Na segunda parte do trabalho, foi implementado um sistema de hashing para cadastro de 10.000 funcionários, utilizando duas estratégias distintas de funções hash (rotação e fold shift). Os testes foram realizados com vetores de tamanhos diferentes (121 e 180 posições), permitindo avaliar o número de colisões e a eficiência de cada função. Os resultados demonstram a importância da escolha adequada de algoritmos e estruturas de dados na resolução de problemas computacionais, evidenciando como decisões na modelagem e na implementação impactam diretamente o desempenho e a eficiência das soluções desenvolvidas.

1 Introdução

Na ciência da computação, as estruturas de dados desempenham um papel fundamental na organização e manipulação eficiente de informações. Dentre os diversos modelos existentes, os grafos se destacam como uma das estruturas mais versáteis e amplamente aplicáveis. Grafos são modelos matemáticos que representam relações entre objetos discretos, sendo compostos por vértices (ou nós), que simbolizam entidades, e arestas, que definem as conexões entre essas entidades. Essa abstração permite modelar desde sistemas simples, como redes de amizade em mídias sociais, até problemas complexos, como rotas de logística, circuitos elétricos e até mesmo relações semânticas em bancos de dados.

A teoria dos grafos tem raízes profundas tanto na matemática quanto na computação, oferecendo ferramentas para resolver problemas de caminhamento, conectividade, fluxo em redes e muito mais. Algoritmos como Busca em Largura (BFS), Busca em Profundidade (DFS), Dijkstra e Prim são exemplos clássicos que demonstram o poder dos grafos na resolução de desafios computacionais. Além disso, a representação eficiente de grafos, seja por matrizes de adjacência ou listas de adjacência.

Paralelamente, os índices hash (ou tabelas hash) são estruturas de alta eficiência para armazenamento e recuperação de dados. Eles utilizam funções hash para mapear chaves

diretamente a posições em uma tabela, reduzindo a complexidade de acesso de O(n) para O(1) em casos ideais. Essa característica os torna indispensáveis em aplicações que demandam buscas rápidas, como bancos de dados, caches e compiladores. No entanto, desafios como colisões de hash e a necessidade de funções hash bem distribuídas exigem atenção na implementação para garantir desempenho ótimo.

Neste trabalho, exploraremos a implementação prática de soluções que combinam grafos e índices hash, utilizando a linguagem C. Através deste estudo, espera-se não apenas consolidar os conceitos teóricos, mas também demonstrar como a integração dessas estruturas pode resolver problemas computacionais de forma elegante e eficiente.

2 Seções Específicas

Esta seção dedica-se à exploração detalhada dos componentes fundamentais do trabalho, dividida em três partes essenciais: os algoritmos de grafos selecionados, os princípios do hashing, a estrutura do trabalho e a aplicação prática dessas estruturas na resolução de problemas específicos.

2.1 Grafos

Um grafo dirigido é um par (V, E), onde V é um conjunto finito de elementos chamados vértices e E é um conjunto de pares ordenados de vértices, chamados arestas. Da mesma forma, um grafo não dirigido é um conjunto $G = (V, E)$ sendo que E consiste de pares não ordenados de vértices. Dizemos que um determinado vértice é adjacente a outro se houver uma aresta que os una. O grau de um vértice é o número de arestas incidentes no vértice.

Existem várias maneiras de representar grafos, temos a lista de adjacências e matriz de adjacências. Considere o grafo:

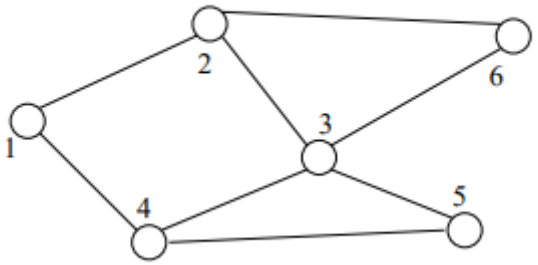

Figura 1 - Grafo.

A representação com lista de adjacências de um grafo $G = (V, E)$ é um vetor de $|V|$ listas, uma para cada vértice em V de modo que, para cada $u \in V$, a lista ligada a u contém os vértices u tais que $(u, v) \in V$. Representada abaixo:

1	→	2	→	4		
2	→	1	→	3	→	6

3	→	2	→	4	→	5	→	6
4	→	1	→	3	→	5		
5	→	3	→	4				
6	→	2	→	3				

A representação com matrizes, por sua vez, de um grafo $G = (V, E)$ e uma matriz $|V| \times |V|$ tal que $a_{i,j} = 1$ se existir aresta entre v_i e v_j, e 0 se não existir tal aresta. A matriz é:

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	1	0	1	0	0	1
3	0	1	0	1	1	1
4	1	0	1	0	1	0
5	0	0	1	1	0	0
6	0	1	1	0	0	0

2.1.1 Algoritmo de Dijskstra

O Algoritmo de Dijkstra, desenvolvido pelo cientista da computação Edsger Dijkstra em 1956, é um dos métodos mais conhecidos para encontrar o caminho de custo mínimo a partir de um vértice de origem em um grafo ponderado sem pesos negativos. Ele é amplamente utilizado em aplicações como roteamento em redes, sistemas de navegação (GPS) e otimização de trajetos.

Em grafos, o algoritmo de Dijkstra é um dos algoritmos que calcula o caminho de custo mínimo entre vértices de um grafo, este algoritmo calcula a distância mínima deste vértice para todos os demais vértices do grafo, ou seja, as restantes cidades. Este algoritmo parte de uma estimativa inicial para a distância mínima, que é considerada infinita (∞), e vai sucessivamente ajustando esta distância. Ele considera que uma cidade estará "fechada" quando já tiver sido obtido um caminho de distância mínima da cidade tomada como origem da busca até ela.

2.1.2 Algoritmo Ford-Moore-Bellman

O Algoritmo de Ford-Moore-Bellman é outro método utilizado para encontrar os caminhos de custo mínimo a partir de um vértice de origem em um grafo. Diferentemente do Algoritmo de Dijkstra, que só funciona corretamente em grafos com pesos não negativos, o Algoritmo de

Bellman-Ford é capaz de lidar com arestas de peso negativo, além de detectar a existência de ciclos negativos no grafo.

O algoritmo opera realizando sucessivas relaxações das arestas, ajustando iterativamente as estimativas de distância mínima. Inicialmente, a distância do vértice para si mesmo é definida como zero, enquanto as distâncias para os demais vértices são consideradas infinitas (∞). A cada iteração, o algoritmo percorre todas as arestas do grafo, verificando se é possível melhorar a distância de um vértice a outro através da relaxação.

Uma das características importantes do Algoritmo de Bellman-Ford é que ele requer $|V| - 1$ iterações para garantir que todas as distâncias mínimas tenham sido encontradas. Caso após essas iterações ainda seja possível realizar uma relaxação, isso indica a presença de um ciclo negativo no grafo, o que inviabiliza a determinação de caminhos mínimos, pois a distância poderia ser indefinidamente reduzida percorrendo esse ciclo repetidamente.

2.2 Hashing

O hashing é uma técnica fundamental em computação que permite o armazenamento e recuperação eficiente de dados por meio de tabelas de hash (também conhecidas como tabelas de dispersão). Essas estruturas são amplamente utilizadas devido à sua capacidade de realizar operações como buscas, inserções e remoções em tempo constante médio, O(1), desde que bem implementadas.

Uma tabela de hash funciona como um vetor onde cada posição, chamada de bucket, pode armazenar uma ou mais entradas, cada uma contendo uma chave e seu respectivo valor associado. A chave é processada por uma função de hash, que calcula um índice no vetor, determinando onde o dado será armazenado ou buscado. O objetivo principal dessa função é distribuir as chaves de maneira uniforme, minimizando colisões, situações em que duas chaves diferentes geram o mesmo índice.

Quando ocorre uma colisão, existem estratégias para resolvê-la. Uma abordagem comum é o encadeamento (chaining), onde cada bucket contém uma lista encadeada de entradas. Assim, se duas chaves forem mapeadas para a mesma posição, elas são armazenadas em uma lista naquele bucket. Outra estratégia é o endereçamento aberto (open addressing), que busca posições alternativas no vetor quando uma colisão acontece, utilizando métodos como sondagem linear (linear probing) ou sondagem quadrática (quadratic probing).

A eficiência de uma tabela de hash depende criticamente da qualidade da função de hash. Uma função mal projetada pode levar a muitas colisões, degradando o desempenho para O(n) no pior caso, onde n é o número de elementos. Por outro lado, uma boa função garante uma distribuição uniforme, mantendo operações rápidas. Além disso, o tamanho da tabela também influencia seu desempenho; tabelas muito pequenas aumentam a chance de colisões, enquanto tabelas muito grandes podem desperdiçar memória.

2.3 Estrutura do trabalho

A estrutura do trabalho está organizada em uma pasta principal denominada “trabalho-III”, conforme ilustrado na Figura X. Dentro dessa pasta, encontra-se os arquivos de código-fonte desenvolvidos para cada questão proposta no trabalho. Essa organização tem como objetivo facilitar o acesso e a identificação das soluções para cada exercício.

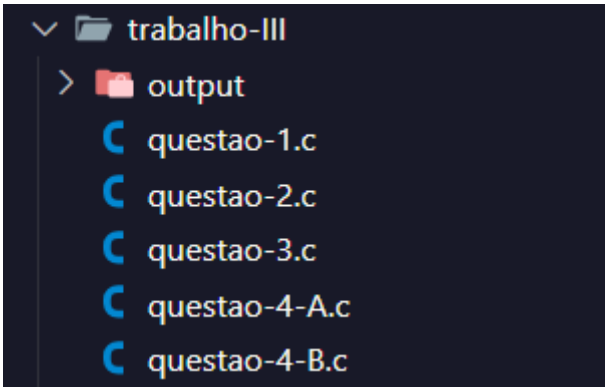

Figura 2 - Estrutura do trabalho.

2.4 Problema

O problema clássico da Torre de Hanói pede para mover n discos entre 3 pinos com o menor número de movimentos, respeitando duas regras: só mover um disco por vez e nunca colocar um disco maior sobre um menor. O problema pode ser representado como um grafo, onde cada configuração dos discos é um vértice e cada movimento válido é uma aresta (com peso 1). Para 4 discos, modele esse grafo e representa-o como uma matriz de adjacência. Use o algoritmo de Dijkstra para encontrar o menor caminho entre uma configuração inicial e a final, medindo o tempo gasto. Repita o processo com o algoritmo de Bellman-Ford e compare os tempos com Dijkstra.

Além disso, dado um grafo com probabilidades nas arestas (representando confiabilidade de comunicação), encontre o caminho mais confiável (aquele que maximiza o produto das probabilidades). Implemente um sistema de hashing para armazenar dados de 10.000 funcionários (matrícula, nome, função, salário), testando duas funções de hash diferentes e comparando desempenho e colisões em tabelas de 121 e 180 posições.

2.4.1 Questão 1 e 2

Neste tópico, descrevemos a funcionalidade de cada função relacionada à questão 1 e 2, implementada em “questao-1.c” e “questao-2.c”.

- criarGrafo: Cria um grafo com matriz de adjacência (arestas) e, se necessário, uma matriz de pesos.
- possibilidadeDeMovimentos: Gera todas as configurações possíveis da Torre de Hanoi com 4 discos e 3 pinos ($3^4 = 81$ configurações). Cada configuração é um vetor de tamanho 4.

- inserirAresta: Adiciona uma aresta entre dois vértices do grafo, representando uma configuração válida de movimento entre duas torres.
- quantidadeMovimentos: Conta quantos discos mudaram de posição entre duas configurações.
- qualDiscoFoiMovido: Identifica qual disco foi movido, se apenas um foi alterado entre as duas torres.
- discoMenorOrigem: Verifica se o disco movido é o menor no pino de origem, ou seja, está no topo.
- movimentoValido: Determina se um movimento entre duas configurações é válido segundo as regras da Torre de Hanoi.
- criarArestas: Adiciona arestas no grafo conectando as configurações válidas com base na regra da torre.
- criarArestasMatriz: Mesma lógica da anterior, mas gera uma matriz de adjacência binária (0 ou 1), usada no Dijkstra.
- dijkstra: Executa o algoritmo de Dijkstra para encontrar o menor número de movimentos (passos) do estado inicial ao final.

2.4.2 Questão 3

Neste tópico, descrevemos a funcionalidade de cada função relacionada à questão 3, implementada em “questao-3.c”.

- criarGrafo: Cria e aloca dinamicamente um grafo com V vértices e E arestas.
- bellmanFord: Executa o algoritmo de Bellman-Ford com base na confiabilidade das arestas.
- imprimirCaminho: Imprime o caminho da origem até o destino, usando o vetor de predecessores (pred) calculado pelo Bellman-Ford.
- main: Cria um grafo com 5 vértices e 6 arestas, define a confiabilidade de cada aresta, roda o Bellman-Ford para encontrar o caminho mais confiável de um vértice origem para outro destino. Por fim, imprime o caminho e a confiabilidade total e libera a memória alocada.

2.4.3 Questão 4

Neste tópico, descrevemos a funcionalidade de cada função relacionada à questão 4, implementada em “questao-4-A.c” a resolução da letra A e “questao-4-B.c” a resolução para a letra B.

- tabelaHash: Calcula a posição da matrícula na tabela hash. Usa partes da string para gerar um número inteiro e aplica o operador módulo (%) com o tamanho da tabela para determinar o índice.
- cadastrarFuncionario: Insere um funcionário na tabela hash.
- buscarFuncionario: Procura um funcionário na tabela hash.
- mostrarTodos: Percorre toda a tabela e imprime os dados dos funcionários presentes.

3 Resultados da Execução do Programa

Esta seção apresenta exemplos práticos da execução do programa, demonstrando suas funcionalidades, além de uma análise comparativa em aspectos específicos. Inicialmente, foram realizados testes com os algoritmos de Dijkstra e de Ford-Moore-Bellman (Questão 1 e 2). Para isso, os quatro discos foram posicionados no primeiro pino da Torre de Hanói, e cada algoritmo foi executado para encontrar a solução, transferindo todos os discos para o último pino. A tabela a seguir exibe o tempo médio para a resolução do problema, calculado com base em 20 execuções de cada algoritmo.

Tabela 1 - Tempo médio dos algoritmos.

Algoritmo	Dijskstra	Ford-Moore-Bellman
Tempo gasto (ms)	1,15	2,05

No próximo problema, dado um grafo orientado com arestas ponderadas por valores reais representando probabilidades, o objetivo é encontrar o caminho mais confiável entre dois vértices. A confiabilidade de um caminho é definida como o produto das probabilidades associadas às arestas que o compõem. Foram inseridas 6 arestas no grafo, e o caminho mais confiável, assim como sua confiabilidade total, estão ilustrados na Figura 3.

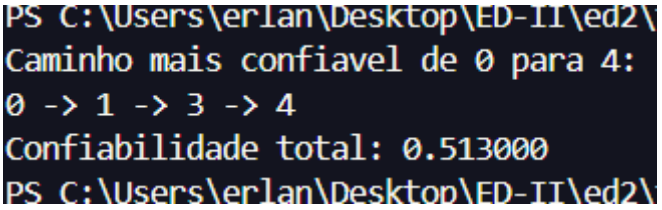

Figura 3 - Resultado 3° questão.

Foi fornecida uma matrícula composta por seis dígitos, sobre a qual foram aplicadas duas funções de hashing distintas, uma para a alternativa A e outra para a alternativa B. Em ambas as alternativas, os testes foram realizados utilizando vetores de destino com 121 e 180 posições, com o objetivo de identificar o número de colisões geradas em função do tamanho do vetor.

Na alternativa A, a função de hashing consiste em realizar uma rotação de dois dígitos para a esquerda na matrícula original. Após a rotação, são extraídos o segundo, o quarto e o quinto dígitos da nova sequência. O valor obtido a partir desses dígitos é utilizado para calcular o resto da divisão pelo tamanho do vetor de destino, determinando a posição de inserção. Caso ocorra colisão, ela é tratada somando-se ao valor obtido o primeiro dígito da matrícula original, buscando assim uma nova posição disponível, como mostra a Tabela 2.

Tabela 2 - Letra A da 4° questão.

Tamanho	121	180
Quantidade de colisões	4	2

Na alternativa B, a função de hashing segue a técnica de fold shift, onde a matrícula é dividida em dois grupos de três dígitos. O primeiro grupo é formado pelos primeiro, segundo e sexto dígitos, enquanto o segundo grupo é composto pelo terceiro, quarto e quinto dígitos. Cada grupo é interpretado como um número, e a soma dos dois resulta no valor base do hash. O índice no vetor é então obtido pelo resto da divisão desse valor pelo tamanho do vetor de destino. Em caso de colisão, o tratamento consiste em somar ao valor do hash o número formado pela concatenação do primeiro e do sexto dígito da matrícula, buscando uma nova posição disponível.

Tabela 3 - Letra B da 4° questão.

Tamanho	121	180
Quantidade de colisões	6	4

Esses testes permitem avaliar como o comportamento das funções de dispersão varia conforme o tamanho do vetor e o método de manipulação da matrícula, permitindo analisar a eficiência de cada abordagem em termos de colisões geradas.

4 Conclusão

Este trabalho proporcionou uma análise prática e aprofundada sobre a aplicação de estruturas de dados fundamentais na ciência da computação: grafos e tabelas de hash. Através da implementação de algoritmos clássicos, como Dijkstra e Ford-Moore-Bellman, foi possível compreender na prática como os grafos podem representar e resolver problemas reais, como a Torre de Hanói e a busca pelo caminho mais confiável em uma rede de comunicações. Os resultados obtidos permitiram comparar os algoritmos quanto à eficiência, destacando o melhor desempenho do Dijkstra em grafos com pesos positivos.

Além disso, a implementação de tabelas de hash com diferentes funções de dispersão evidenciou a importância da escolha adequada da função de hashing e do tamanho da tabela. A análise comparativa entre as alternativas A e B demonstrou que variações simples na forma como os dados são processados podem impactar diretamente o número de colisões, influenciando o desempenho final da estrutura.

Referências

SANTOS, P. R. dos. Os grafos e os algoritmos. Medium – Programadores Ajudando Programadores, 2019. Disponível em: https://medium.com/programadores-ajudando-programadores/os-grafos-e-os-algoritmos-697c1fd4a416.

ASSUNÇÃO, N. R. de. Algoritmos em grafos. Instituto de Computação, Universidade Estadual de Campinas, Campinas, SP, 2012. Disponível em: https://www.ic.unicamp.br/~norton/arquivos/alg_grafos.pdf.

DE CARVALHO, B. M. P. S. Algoritmo de dijkstra. Universidade de Coimbra, Coimbra, Portugal, 2008.