Relatório Referente ao trabalho II da disciplina de Estrutura de Dados II
Autor: Mauricio Benjamin da Rocha
Dupla: Lazaro Claubert Souza Rodrigues Oliveira

Resumo do Projeto
O presente trabalho tem como objetivo a implementação dos sistemas solicitados para a disciplina de Estrutura de dados II utilizando as estruturas de dados conhecidas por “Grafos” e “Tabela Hash” em suas diferentes variações. Em complemento ao desenvolvimento realizou-se a validação através do comparativo de desempenho s visando avaliar quais se sobressaem em diferentes cenários.

1. Introdução

A tabela de hash, uma estrutura de dados essencial à computação, demonstra sua eficiência por meio do uso de uma função “hash” que mapeia dados para índices na tabela. Tal mecanismo possibilita uma busca de tempo constante, propiciando acesso rápido aos dados armazenados. Além disso, a tabela hash proporciona uma utilização eficiente do espaço de armazenamento, visto que permite a compactação dos dados. Sua versatilidade é evidenciada em aplicações como dicionários, bancos de dados e caches, onde a busca e manipulação de dados são otimizadas. A agilidade nas operações de inserção e remoção de elementos também ressalta os benefícios práticos dessa estrutura.

Por outro lado, os grafos emergem como uma representação poderosa de relações complexas entre entidades. Compostos por nós (vértices) e arestas que conectam esses nós, os grafos podem ser direcionados ou não direcionados. Sua aplicabilidade destaca-se na modelagem de diversas interações, desde redes sociais até sistemas de transporte. Os algoritmos associados aos grafos, como os de caminho mínimo, desempenham um papel crucial em setores como logística, redes de computadores e planejamento de rotas. A análise de redes sociais, por meio de grafos, torna-se uma ferramenta valiosa para identificar influenciadores e compreender padrões de conexão. Além disso, os grafos são empregados em problemas que envolvem a otimização de recursos, como a distribuição eficiente de tarefas em projetos complexos. Em síntese, as tabelas hash e os grafos, embora distintos em suas características, são fundamentais na resolução de problemas computacionais complexos e na representação eficiente de relações e estruturas de dados.

O presente Projeto foi desenvolvido utilizando linguagem de programação C e estruturado em três pastas separadamente visando garantir uma maior organização do código fonte de forma a facilitar seu acesso, manutenção e atualização. A Figura 1 Apresenta visualmente como estão organizadas as pastas, onde a pasta Q1 guarda todos os arquivos da questão 01 do projeto, Q2 e Q3 guardam respectivamente as questões 02 e 03 do projeto.

Figura 1: Organização dos arquivos do Projeto.

O presente trabalho está organizado em seções de forma visando facilitar o acesso a informações específicas. As seções são seções específicas, resultado da execução dos programas e conclusão do projeto.

2. Seções Específicas

Esta seção visa apresentar de forma mais precisa como cada questão foi trabalhada, desde as estruturas de dados usadas até a interação com o usuário. A experimentação realizada visando analisar o desempenho foi realizada em uma máquina com a configuração descrita na Tabela 1.

Tabela 01: Hardware usada para os experimentos

Processador	**Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz 3.19 GHz**
Memória RAM	**8 GB DDR4**
Sistema Operacional	**Linux Ubuntu 22.04.3 LTS**

2.1. Q1 - Torre de Hanói

O problema clássico da Torre de Hanói consiste em mover uma quantidade específica de discos entre três pinos, buscando realizar essa tarefa com o menor número possível de movimentos. As condições estabelecidas são as seguintes: apenas um disco pode ser movido por vez, e um disco só pode ser colocado sobre outro de tamanho maior ou na base de algum pino. A representação de uma configuração de discos é feita através de um vetor, onde cada posição indica o pino onde o disco correspondente está posicionado.

Para modelar esse problema como um grafo, cada vértice do grafo representa uma configuração única dos discos. A interligação entre configurações é representada por arestas, indicando que é possível mover de uma configuração para outra por meio de um movimento legal de um disco. Dessa forma, o grafo encapsula todas as possíveis configurações alcançáveis a partir das regras estabelecidas.

Para solucionar o problema proposto começamos por ler os dados do arquivo “config_hanoi.csv” e preencher a estrutura VerticiesInfo para cada configuração de disco. Cada configuração é representada por um vetor com informações sobre o número de vértices (1,2,3 ... 81), identificação dos discos em relação aos pinos (ex: 1111, onde todos os discos estão no pino 1, 1112, onde todos os discos exceto o disco 4 estão no pino 1, e o quatro está no pino 2), e as arestas (movimentos válidos como 1111 para 2 ou 3, onde podemos mover o disco no topo tanto para a configuração 2 que é 1112 quanto para 3 que é 1113).

A matriz de adjacência semelhante a Figura 1 é então preenchida com base nessas informações, indicando as conexões entre as configurações. A posição (i, j) da matriz é 1 se há uma aresta direcionada de i para j, indicando que é possível mover de uma configuração para a outra.

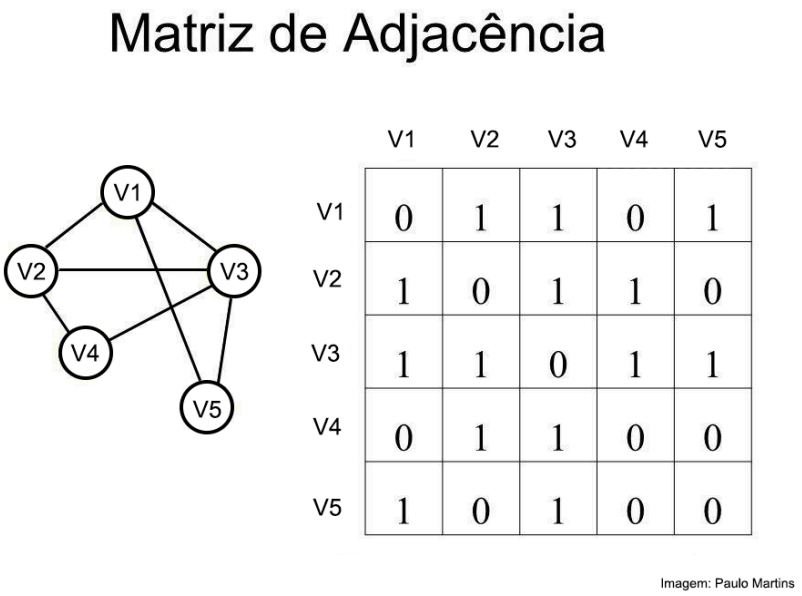

Figura 1. Exemplo de uma matriz de adjacências - Fonte: medium.com

Visando uma comparação de desempenho usaremos os algoritmos de Dijkstra e Bellman-Ford para encontrar o menor caminho a partir de uma configuração inicial (configuração de partida) para a configuração final (configuração com todos os discos no pino de destino). O tempo de execução é medido para ambos os algoritmos. A configuração inicial é representada pelo vértice 1, 1111, e a configuração final é representada pelo vértice 81, 3333. Ao final da experimentação obtemos os dados contidos na Tabela 1 onde podemos concluir que o menor caminho encontrado por ambos os algoritmos é o mesmo, entretanto Dijkstra garante um desempenho superior para atingir o mesmo resultado.

Algoritmo	Número de movimentos	Tempo gasto (ms)
Dijkstra	15	0.091000
Bellman-Ford	15	1.597000

2.2. Q2 - Caminho mais confiável

O enunciado descreve um problema em um grafo orientado G = (V, E), onde cada aresta (u, v) possui um valor associado r(u, v), sendo este um número real no intervalo $0 \leq r(u, v) \leq 1$. Esse valor representa a confiabilidade de um canal de comunicação do vértice u até o vértice v. A interpretação é que r(u, v) representa a probabilidade de que o canal de comunicação entre u e v não falhe, e supõe-se que essas probabilidades são independentes entre as arestas.

O desafio proposto é desenvolver um programa eficiente para encontrar o caminho mais confiável entre dois vértices específicos do grafo. Em outras palavras, o objetivo é determinar a rota entre dois vértices de forma a maximizar a confiabilidade total ao longo desse caminho, considerando as probabilidades associadas às arestas.

Para resolver esse problema, o programa precisa explorar diferentes caminhos possíveis no grafo, levando em consideração os valores de confiabilidade associados às arestas. A eficiência do programa será crucial, pois é necessário analisar todas as combinações de caminhos para encontrar o caminho mais confiável entre os dois vértices dados. Essa eficiência é especialmente importante

em grafos grandes, nos quais a busca exaustiva de todos os caminhos seria computacionalmente custosa.

O código começa solicitando o número de vértices e arestas. Em seguida, o grafo é inicializado com o número de vértices e as arestas fornecidas. As confiabilidades são inicializadas com -1, indicando que não há arestas entre os vértices. O código solicita as informações para cada aresta (vértice de origem, vértice de destino e confiabilidade). As informações fornecidas são então atribuídas à estrutura de dados do grafo.
O algoritmo de Dijkstra é usado para encontrar o caminho mais confiável do vértice inicial (0) para todos os outros vértices no grafo. O algoritmo mantém um array "dist" para armazenar as distâncias acumuladas até cada vértice e um array "verticeAnterior" para rastrear o vértice anterior no caminho mais curto.

Verifica-se então se há um caminho confiável do vértice inicial para o vértice final. Se houver, imprime o caminho e a confiabilidade associada, calculando a confiabilidade com "exp(dist[verticeFinal])" conforme a Figura 2. Caso contrário, informa que não há caminho confiável. Portanto, o sistema utiliza o algoritmo de Dijkstra para encontrar o caminho mais confiável em um grafo ponderado dirigido, onde as confiabilidades das arestas representam a probabilidade de que o canal não venha a falhar.

```
PS C:\Users\PC\Documents\GitHub\6_periodo\ED_II\TB_III> cd 'c:\Users\PC\Documents\GitHub\6_periodo\ED_II\TB_III\Q2\output'
PS C:\Users\PC\Documents\GitHub\6_periodo\ED_II\TB_III\Q2\output> & .\'Q2.exe'
Numero de vertices: 5
Numero de arestas: 7
aresta 1 (u v confiabilidade): 0 1 0.8
aresta 2 (u v confiabilidade): 0 2 0.5
aresta 3 (u v confiabilidade): 1 3 0.6
aresta 4 (u v confiabilidade): 2 3 0.7
aresta 5 (u v confiabilidade): 2 4 0.9
aresta 6 (u v confiabilidade): 3 4 0.6
aresta 7 (u v confiabilidade): 4 0 0.7
vertice inicial: 0
vertice final: 4
Caminho mais confiavel de 0 para 4: 0 -> 2 -> 4
Confiabilidade: 0.4500
PS C:\Users\PC\Documents\GitHub\6_periodo\ED_II\TB_III\Q2\output>
```

Figura 2. Exemplo de teste

2.3. Tabela Hash para uma base de dados de 1000 funcionários

O desafio proposto consiste em organizar uma base de dados com 1000 funcionários utilizando tabelas de hash para localizar informações por meio dos números de matrícula dos funcionários. Os dados relevantes incluem Matrícula (uma string de 6 dígitos), Nome, Função e Salário. Duas funções de hashing distintas são sugeridas para esta organização, ambas com a finalidade de distribuir eficientemente os dados em um vetor destino, inicialmente de 101 posições e, posteriormente, de 150 posições.

Na primeira abordagem (a), a função de hashing proposta realiza uma rotação de 2 dígitos para a esquerda, seguida pela extração dos 2º, 4º e 6º dígitos da matrícula. O resultado é obtido através do resto da divisão desse valor pelo tamanho do vetor destino. No caso de colisões, é estipulado o tratamento mediante a adição do primeiro dígito da matrícula ao resultado.

Na segunda abordagem (b), a função de hashing adota um fole shift com 3 dígitos, especificamente rearranjando os dígitos da matrícula. O resultado final é calculado através do resto da divisão desse rearranjo pelo tamanho do vetor destino. Em situações de colisão, é estabelecido o método de adicionar 7 ao valor resultante.

Os arquivos 101.c e 150.c solucionam o problema abordado acima atendendo aos critérios solicitados conforme descrito abaixo:

a. Estruturas e Tipos:
São definidas as estruturas "Funcionario", "Chave_hash", e "Tabela_Hash".

- "Funcionario" armazena os dados dos funcionários.
- "Chave_hash" contém o hash e o funcionário correspondente.

- “Tabela_Hash” é uma tabela com a configuração adequada para o uso proposto.

b. Funções de Hashing e Colisão (Letra A):
- “hash_a” Realiza a rotação de 2 dígitos para a esquerda, extrai os dígitos especificados e calcula o resto da divisão pelo tamanho do vetor destino.
- “colisao_a”: Trata colisões adicionando o primeiro dígito da matrícula ao resto da divisão.

c. Manipulação da Tabela Hash:
- “criar_tabela” Inicializa uma tabela hash com todas as posições marcadas como vazias.
- “insere_a” Insere uma chave na tabela, tratando colisões conforme a letra A do enunciado.

d. Geração de Matrículas e Testes:
- “gerar_matricula” Gera matrículas aleatórias para testes.
- O código preenche e insere dados na tabela utilizando a função de hashing da letra A.

e. Resultados e Saídas através da função “main.c”:
- Exibe o número de colisões e o tempo gasto para a função de hashing da letra A.
- Repete o processo para a função de hashing da letra B.

O código de 150.c é semelhante ao código 101.c, mas utiliza um vetor destino de 150 posições e implementa a função de hashing "fole shift com 3 dígitos" (letra B do enunciado). As estruturas, funções de hashing, manipulação da tabela hash e a geração de matrículas são equivalentes ao código 1, mas adaptadas para a função de hashing da letra B. A Tabela 2 mostra os resultados obtidos entre os casos solicitados onde podemos concluir que para este problema a letra B obteve um melhor resultado tanto em tempo quanto em menor número de colisões.

Tamanho do vetor	Total de colisões usando letra A	Tempo gasto(ms) letra A	Total de Colisões usando letra B	Tempo gasto(ms) letra B
101	18766	0.393	6799	0.259
150	24390	0.409	9123	0.328

4. Conclusão

Este projeto culmina com a implementação de três questões, cada uma requerendo uma abordagem distinta para sua resolução. Ao longo do desenvolvimento, enfrentamos diversos desafios lógicos e técnicos, demandando um tempo significativo para compreender completamente os requisitos e iniciar a implementação. Habilidades de desenho foram essenciais para visualizar as estruturas e apoiar a codificação.

5. Apêndice

Todo o material usado para o desenvolvimento e experimentação segue em conjunto com este relatório de forma a ficar melhor a visualização e acesso com o mesmo.