RELATÓRIO TÉCNICO SOBRE IMPLEMENTAÇÕES EM GRAFOS E HASHING EM C

Sthefany Moura Godinho

RESUMO

Este relatório apresenta sete implementações em C envolvendo grafos, buscas e tabelas de *hashing*. Nas Questões 1 e 2, a Torre de Hanói com quatro discos foi modelada como grafo de 81 estados, e o algoritmo de Dijkstra encontrou, nas três configurações testadas, caminhos mínimos de 15, 7 e 15 movimentos. As listas de adjacência foram ligeiramente mais rápidas que a matriz (média de 0 , 000221 0,000221s contra 0 , 000291 0,000291s). Nas Questões 3 a 6, uma planilha com 160 células foi representada como grafo de dependências, permitindo referências e funções (@soma, @media etc.). Sobre esse grafo, BFS e DFS apresentaram tempos médios extremamente baixos — 0 , 000002 0,000002s e 0 , 000001 0,000001s — enquanto a reconstrução completa das dependências levou 0 , 000013 0,000013s. Na Questão 7, duas funções de *hashing* foram comparadas em uma base de 4000 alunos, resultando em tempos médios de inserção próximos de 0 , 022 0,022s e mais de 3,6 milhões de colisões em cada função devido ao alto fator de carga. Os resultados confirmam a eficiência de listas em grafos esparsos, a baixa complexidade das buscas em grafos pequenos e o impacto direto do fator de carga na performance de tabelas hash.

1 INTRODUÇÃO

Este relatório reúne a solução de sete questões propostas na disciplina de Estruturas de Dados II, com foco em três temas principais: (i) representação de problemas clássicos por grafos; (ii) uso de algoritmos de busca em grafos e medição de desempenho; e (iii) acesso eficiente a registros por meio de tabelas de *hashing*.

Nas Questões 1 e 2, a Torre de Hanói com quatro discos é modelada como um grafo de configurações, em que vértices representam estados possíveis do jogo e arestas representam movimentos legais. O menor número de movimentos entre uma configuração inicial e a configuração final é obtido pelo algoritmo de Dijkstra, e é feita uma comparação entre a implementação utilizando matriz de adjacência e aquela com listas de adjacência.

Nas Questões 3 a 6, uma planilha de cálculo simplificada (160 células) é vista como um grafo de dependências entre células. A partir dessa modelagem é possível implementar referências e funções agregadas, além de explorar buscas em largura (BFS) e profundidade (DFS) para atualizar ou inspecionar células relacionadas e medir tempos médios de inserção e de percursos sobre o grafo.

Na Questão 7, uma base de 4000 alunos é indexada em uma tabela de *hashing* usando duas funções de dispersão diferentes e dois tamanhos de tabela. São coletados tempos médios de inserção e colisões, permitindo discutir o fator de carga, a qualidade do espalhamento das chaves e o impacto prático dessas escolhas.

Nas seções seguintes, são descritas as questões implementadas, o ambiente de testes utilizado, os resultados obtidos e as principais conclusões.

2 DESCRIÇÃO DAS QUESTÕES

Questão 1 – Torre de Hanói com matriz de adjacência

A primeira questão modela a Torre de Hanói com 4 discos e 3 pinos como um grafo de estados finito. Cada estado é representado por um vetor $[d_0, d_1, d_2, d_3]$, em que cada componente indica o pino (0, 1 ou 2) em que o disco correspondente se encontra. Esse vetor é codificado em um índice inteiro em base 3, resultando em $3^4 = 81$ vértices numerados de 0 a 80.

- Cada movimento legal (mover um único disco, sem colocar disco maior sobre menor) gera uma aresta de peso 1 entre estados.
- O grafo é armazenado em uma matriz de adjacência 81×81 alocada dinamicamente com `malloc`, inicializada com zeros.
- Para uma configuração inicial digitada pelo usuário, aplica-se o algoritmo de Dijkstra para encontrar o menor caminho até o estado final $[2, 2, 2, 2]$.
- Ao final, são impressos o número mínimo de movimentos, o tempo de execução de Dijkstra (em segundos) e a sequência de estados intermediários.

Questão 2 – Torre de Hanói com lista de adjacência

A Questão 2 reaproveita a mesma codificação de estados e a mesma lógica de geração de movimentos da Questão 1, mas altera a estrutura de armazenamento do grafo para listas de adjacência, mais adequadas a grafos que não são completamente densos.

- Cada vértice possui uma lista encadeada de vizinhos, com um nó para cada estado alcançável em um único movimento.
- Dijkstra é adaptado para percorrer essas listas, relaxando apenas as arestas existentes, sem varrer colunas inteiras da matriz.
- Também são medidos o tempo de execução de Dijkstra e o número de movimentos, de modo a comparar diretamente com a versão em matriz.
- A implementação ilustra como a troca da estrutura de dados não altera a corretude do algoritmo, mas pode reduzir acessos desnecessários e o consumo de memória.

Questão 3 – Planilha com matriz de adjacência

Na Questão 3, é implementada uma planilha de cálculo rudimentar com 8 colunas (A..H) e 20 linhas (1..20), totalizando 160 células. Cada célula é identificada por uma coordenada (por exemplo, A1, B3) e por um índice linear interno.

- Cada célula pode conter: um número (por exemplo, `A1 10`), uma referência a outra célula (por exemplo, `B1 =A1`) ou uma função de intervalo (@soma, @max, @min, @media).
- O relacionamento de dependência entre células (por exemplo, C3 depender de A1..B2) é representado em uma matriz de adjacência 160×160: um valor 1 em (i, j) indica que a célula i depende da célula j.

- As expressões digitadas são analisadas com `sscanf` e funções auxiliares, que identificam se o texto é numérico, referência ou função, atualizando o tipo de conteúdo da célula.
- A avaliação dos valores é feita de forma recursiva: referências chamam a avaliação da célula referenciada e funções varrem um intervalo de células, combinando os valores conforme a operação desejada.

Questão 4 – Planilha com listas de adjacência

A Questão 4 mantém a mesma interface e semântica da Questão 3, mas substitui a matriz de adjacência por listas de adjacência, tornando a representação mais leve e mais próxima de implementações reais de planilhas.

- Para cada célula é mantida uma lista de dependências (`NoDependencia`), em que cada nó armazena o índice de uma célula da qual ela depende diretamente.
- Ao interpretar expressões, chamadas a `AdicionarDependencia` criam esses nós, evitando duplicatas.
- Na avaliação de funções, o programa percorre as listas, avaliando recursivamente as células dependidas; células vazias e funções são tratadas com cuidado para evitar ciclos simples.
- Com essa abordagem, a memória consumida passa a depender do número real de relações de dependência, e não de 160^2 entradas fixas.

Questão 5 – BFS e DFS na planilha

A Questão 5 adiciona ao modelo da planilha operações de **busca em largura** (BFS) e **busca em profundidade** (DFS) sobre o grafo de dependências. O usuário pode escolher uma célula de origem (por exemplo, C3) e emitir comandos específicos.

- O comando `BFS C3` executa uma busca em largura a partir da célula C3, usando uma fila interna, vetores de visitados e as listas de dependências para percorrer o grafo em camadas.
- O comando `DFS C3` executa uma busca em profundidade recursiva (ou equivalente iterativa com pilha), explorando um caminho até o fim antes de retroceder.
- Em ambos os casos, o programa imprime na tela a ordem em que as células são visitadas, usando a conversão índice $\leftrightarrow$ coordenada (por exemplo, $18 \rightarrow$ C3).
- Essa funcionalidade permite visualizar o impacto de alterações em uma célula sobre outras células da planilha, o que é útil para depuração e entendimento do fluxo de dependências.

Questão 6 – Tempos médios de inserção, BFS e DFS

Na Questão 6, o foco passa a ser o **desempenho** das operações sobre o grafo da planilha. Para uma célula escolhida (C3), implementa-se um experimento de medição de tempos médios.

- Um comando `TEMPOS C3` dispara três medições distintas: (i) reconstrução completa do grafo (reanalisando todas as expressões das 160 células), (ii) BFS sem impressão a partir de C3 e (iii) DFS sem impressão a partir de C3.

- Cada operação é repetida 30 vezes em sequência. O programa usa `clock()` antes e depois do laço de repetições, e divide o tempo total por 30 para obter o tempo médio.
- Os resultados numéricos são exibidos em segundos e depois discutidos na seção de Resultados e Discussão, comparando a ordem de grandeza das três operações.

Questão 7 – Tabela de hashing para 4000 alunos

A Questão 7 trata de uma base de 4000 alunos que deve ser organizada em uma tabela de *hashing* para acesso rápido por matrícula (string de 11 dígitos). Em vez de ordenar o vetor, opta-se por funções de hash e tratamento de colisão.

- A matrícula é composta por 4 dígitos de ano, 1 dígito de curso e 6 dígitos de número de aluno.
- Duas funções de dispersão são testadas: (a) rotação dos 5 primeiros dígitos seguida da extração de 3 dígitos e módulo do tamanho da tabela; (b) técnica de *fold shift* com blocos de 3 dígitos (6º,7º,11º) e (8º,9º,10º).
- Em ambos os casos, colisões são tratadas por endereçamento aberto com passo calculado a partir de dígitos da própria matrícula.
- São utilizados dois tamanhos de tabela: 1211 e 1280 posições. Para cada combinação (função, tamanho), as 4000 matrículas sintéticas são inseridas 30 vezes, acumulando tempo total e colisões.
- A partir disso, o programa imprime tempos médios e colisões médias, permitindo comparar as funções e discutir o impacto do fator de carga elevado.

3 TECNOLOGIAS E METODOLOGIA DE TESTE

Os programas foram desenvolvidos e executados em ambiente Linux, com compilador `gcc`. A Tabela 1 resume o ambiente de referência.

Tabela 1 – Especificações do ambiente de testes

Componente	**Especificação**
Sistema Operacional	Ubuntu 24.04.1 LTS (x86_64)
Processador	Intel Core i7 (8 núcleos lógicos)
Memória RAM	8 GiB
Armazenamento	SSD 256 GB
Compilador	`gcc` 13.x
Linguagem	C (ponteiros, structs, grafos, hashing)
Medição de tempo	`clock()` / `CLOCKS_PER_SEC` (`time.h`)

Para os testes quantitativos (Dijkstra, hashing, BFS/DFS), a medição de tempo é feita em torno apenas do trecho de cálculo, evitando `printf` dentro da janela cronometrada. Saídas em tela (caminhos, ordens de visita) são impressas após a medição.

4 RESULTADOS E DISCUSSÃO

4.1 Torre de Hanói (Questões 1 e 2)

Foram testadas três configurações iniciais para a Torre de Hanói com 4 discos, tanto na versão com matriz quanto na versão com lista de adjacência:

- Caso A: $[0, 0, 0, 0]$ (todos os discos no pino 0);
- Caso B: $[1, 0, 0, 2]$;
- Caso C: $[2, 1, 0, 0]$.

Em todos os casos, Dijkstra encontrou o mesmo número mínimo de movimentos:

- Casos A e C: 15 movimentos;
- Caso B: 7 movimentos.

Os tempos medidos foram, em segundos:

Tabela 2 – Dijkstra na Torre de Hanói: matriz vs. lista

Caso	**Configuração**	**Matriz (s)**	**Lista (s)**
A	[0,0,0,0]	0,000254	0,000211
B	[1,0,0,2]	0,000368	0,000198
C	[2,1,0,0]	0,000252	0,000255

As médias aproximadas foram:

$$\bar{t}_{\text{matriz}} \approx 0{,}000291\,\text{s}, \quad \bar{t}_{\text{lista}} \approx 0{,}000221\,\text{s}.$$

As diferenças são pequenas, mas a lista de adjacência foi ligeiramente mais rápida em média, além de mais econômica em memória. Isso confirma a vantagem de listas quando cada vértice possui poucas arestas em relação ao máximo possível. A sequência de estados calculada (caminho mínimo) foi idêntica entre as duas implementações, o que reforça que a mudança é apenas de representação e não de algoritmo.

4.2 Planilha de cálculo (Questões 3 e 4)

Na planilha, foram inseridas expressões que combinam valores simples, referências e funções, resultando em uma saída final como a Tabela 3, que mostra as três primeiras linhas da planilha após a edição.

Tabela 3 – Trecho da planilha após inserção das expressões

\C	A	B	C	D	E	F	G	H
01	10,00	5,00	7,00	20,00	62,00	0,00	200,00	222,00
02	20,00	20,00	0,00	0,00	0,00	0,00	0,00	0,00
03	10,00	55,00	10,33	0,00	0,00	0,00	0,00	0,00

Isso confirma, por exemplo, que:

- **B3** soma o intervalo `A1..B2` e obtém 55,00;
- **C3** calcula uma média aproximada de 10,33 sobre um conjunto de células que inclui A1, A2, A3, B1, B2 e B3;
- funções de máximo, mínimo e soma de intervalos se comportam conforme o esperado;
- a célula **H1**, com soma envolvendo a própria linha, é tratada sem entrar em ciclo infinito, graças às validações na análise de expressões.

As versões com matriz e com lista produzem os mesmos resultados numéricos. A diferença está na estrutura interna: a matriz consome 160^2 posições, enquanto as listas armazenam apenas as dependências reais. Em cenários maiores ou com muitas células vazias, essa diferença tende a crescer, favorecendo listas de adjacência.

4.3 Questões 5 e 6: buscas em largura/profundidade e tempos médios

As Questões 5 e 6 estendem o modelo da planilha (grafo de dependências entre células) com duas funcionalidades principais:

- **Q5** – implementação de *busca em largura* (BFS) e *busca em profundidade* (DFS) no grafo de dependências;
- **Q6** – medição do *tempo médio* de três operações: (a) reconstrução das dependências (inserção no grafo), (b) BFS e (c) DFS, repetindo cada uma 30 vezes.

A planilha utilizada é a mesma apresentada nas Questões 3 e 4, contendo, entre outras, as seguintes células relevantes:

- **A1** = 10, **B1** = 5, **A2** = 20, **B2** = 20;
- **B3** = @soma(A1..B2) = 55;
- **C3** = @media(A1..C2) $\approx$ 10,33, dependendo de A1, B1, A2, B2, C1, C2 e B3.

A partir dessa configuração, foi escolhido o vértice **C3** como ponto de partida das buscas no grafo de dependências.

BFS e DFS a partir de C3

Os comandos executados foram:

```
> BFS C3
> DFS C3
```

A BFS produziu a ordem:

C3, C2, B2, A2, C1, B1, A1, B3,

enquanto a DFS produziu:

C3, C2, B3, B2, A2, B1, A1, C1.

O conjunto de células visitadas é o mesmo em ambos os casos (8 células), mas em ordens diferentes, como esperado pelas estratégias de busca em largura e profundidade.

Medição de tempos médios (Q6)

Para a Questão 6, foi implementada uma rotina que mede o tempo médio das seguintes operações:

1. **Reconstrução do grafo de dependências**;
2. **BFS sem impressão** a partir de C3;
3. **DFS sem impressão** a partir de C3.

Cada operação foi repetida **30 vezes**. A saída obtida foi:

```
[TEMPOS] Resultados (media de 30 repeticoes) a partir de C3:
  Insercao/Reconstrucao do grafo: 0.000013 s
  BFS: 0.000002 s
  DFS: 0.000001 s
```

A Tabela 4 resume esses resultados.

Tabela 4 – Tempo médio das operações na planilha (30 repetições, origem C3)

Operação	**Tempo médio (s)**
Reconstrução do grafo (inserção das dependências)	0,000013
BFS (sem impressão)	0,000002
DFS (sem impressão)	0,000001

Como a planilha possui apenas 160 células e o subgrafo de dependências de C3 tem 8 vértices, os tempos absolutos são extremamente baixos (ordem de 10^{-6} a 10^{-5} segundos), ficando próximos do limite de resolução do `clock()`. Ainda assim, observa-se que:

- A reconstrução do grafo é a operação mais custosa, pois percorre todas as células;
- BFS e DFS apresentam tempos médios muito semelhantes, com pequena vantagem numérica para DFS neste cenário específico;
- Ambas as buscas têm custo linear no tamanho do subgrafo alcançado, compatível com o conjunto de 8 células visitadas.

4.4 Hashing de matrículas (Questão 7)

Para a base de 4000 alunos, foram testadas duas funções de *hashing* (a e b) e dois tamanhos de tabela (1211 e 1280). Os resultados médios (30 execuções) foram:

```
=== Tabela = 1211 ===
Hash (a): tempo medio = 0.022860 s, colisoes medias = 3605733.00
Hash (b): tempo medio = 0.022066 s, colisoes medias = 3628591.00

=== Tabela = 1280 ===
Hash (a): tempo medio = 0.022614 s, colisoes medias = 3736131.00
Hash (b): tempo medio = 0.022631 s, colisoes medias = 3762109.00
```

Esses dados são organizados nas Tabelas 5 e 6.

Tabela 5 – Hashing de matrículas: tabela com 1211 posições

Função	**Tempo médio (s)**	**Colisões médias**
Hash (a)	0,022860	3 605 733,00
Hash (b)	0,022066	3 628 591,00

Tabela 6 – Hashing de matrículas: tabela com 1280 posições

Função	**Tempo médio (s)**	**Colisões médias**
Hash (a)	0,022614	3 736 131,00
Hash (b)	0,022631	3 762 109,00

Principais observações:

- Os **tempos médios** são próximos (cerca de 0,022 s) em todos os cenários, o que indica que, para esse conjunto de testes, o custo de sondagem adicional devido às colisões não alterou muito o tempo global.
- O número de **colisões** é muito alto devido ao fator de carga superior a 3 (4000 elementos para $\approx$ 1200 posições).
- A função (a) gera ligeiramente menos colisões que a (b), mas sem grande impacto no tempo total.
- Aumentar a tabela de 1211 para 1280 posições não reduziu colisões; com essas funções de hash, a distribuição das chaves continua concentrada em determinadas regiões, ilustrando que apenas aumentar o tamanho da tabela não garante melhora se a função de dispersão não for bem escolhida.

Os testes mostram, na prática, a importância de equilibrar fator de carga, função de *hash* e estratégia de colisão para obter bom desempenho em tabelas de espalhamento.

5 CONCLUSÃO

Os experimentos tiveram como propósito compreender, na prática, o impacto da escolha de estruturas de dados na eficiência de soluções em C. Esse objetivo foi alcançado por meio da implementação e análise de grafos, buscas e tabelas hash.

Na Torre de Hanói, verificou-se que listas de adjacência são mais eficientes que matrizes em grafos esparsos, embora ambas produzam o mesmo caminho mínimo. Na planilha, confirmou-se que dependências entre células podem ser modeladas como grafos, e que BFS e DFS percorrem

o mesmo conjunto de células com tempos muito baixos. Já no hashing, observou-se que o fator de carga elevado aumenta drasticamente as colisões, independentemente da função de dispersão utilizada.

Os resultados obtidos permitiram descobrir como diferentes estruturas influenciam tempo e memória, e reforçaram a importância de medir desempenho repetidas vezes. Os experimentos demonstraram, de forma prática, que decisões de implementação afetam diretamente o comportamento do programa, consolidando o aprendizado sobre trade-offs entre matrizes, listas e tabelas hash.