EXERCÍCIO PROGRAMA I

Luca Marinho Nasser Valadares Paiva ,
Número USP: 13691375

Introdução:

Neste EP, implementei e comparei duas representações de grafos não direcionados, lista de adjacência e matriz de adjacência. Assim, desenvolvi operações como criação do grafo, inserção e remoção de arestas, busca em largura (BFS), busca em profundidade (DFS) e verificação de conectividade. Para isso, criei estruturas as auxiliares: lista ligada, fila e pilha.

Minha abordagem priorizou três aspectos principais, modularidade, evitando criações de funções redundantes ou desnecessárias; custo das operações (por exemplo, `enqueue`, `dequeue`, `push`, `pop` em tempo $\mathcal{O}(1)$); e simplicidade estrutural do código.

Através dos testes de desempenho realizados, pude validar empiricamente as previsões teóricas sobre complexidade computacional e identificar os cenários onde cada representação oferece melhor desempenho. As seções seguintes detalham os componentes construídos, as decisões tomadas e a análise dos resultados obtidos.

Decisões de Implementação:

Esta seção explica e justifica as escolhas que fiz durante o desenvolvimento do EP. Meu critério principal, como já dito, foi equilibrar simplicidade de código, custo assintótico e reutilização das funções. Abaixo explico como cada decisão impacta o desempenho, uso de memória e facilidade de desenvolvimento.

Lista ligada:

Implementei a lista ligada usando nós do tipo `List`, onde cada nó contém um valor inteiro e um ponteiro para o próximo elemento. A principal decisão foi criar uma estrutura `LinkedList` que mantém dois ponteiros, `head` (início) e `tail` (final da lista). Essa escolha permitiu inserções no final em tempo constante $\mathcal{O}(1)$ através da função `append_node`, sem precisar de um loop percorrer toda a lista.

Escolhi separar a criação de nós na função `add_node` para centralizar a alocação de memória e tornar o código mais legível. As operações básicas ficaram assim:

- `insert_node`: inserção no início em $\mathcal{O}(1)$;
- `delete_first_node`: remoção do primeiro elemento em $\mathcal{O}(1)$;
- `search_node_with_value`: busca por valor em $\mathcal{O}$(n);
- `delete_node_with_value`: remoção por valor também em $\mathcal{O}$(n).

A decisão de manter o ponteiro `tail` foi importante para o desempenho das estruturas que dependem da lista (fila especialmente), garantindo operações eficientes mesmo com muitos elementos.

Fila e pilha:

Implementei tanto a fila quanto a pilha reutilizando a lista ligada como estrutura base.

Para a fila, uso `append_node` para inserir elementos no final (operação `enqueue`) e `delete_first_node` para remover do início (`dequeue`). Ambas as operações executam em $\mathcal{O}(1)$ graças ao ponteiro `tail` da lista ligada. Sem esse ponteiro, a inserção no final seria $\mathcal{O}$(n), tornando a fila menos eficiente.

Para a pilha, uso `insert_node` para empilhar elementos no início (`push`) e `delete_first_node` para desempilhar (`pop`). Novamente, ambas as operações são $\mathcal{O}(1)$.

As funções de acesso (`front` para fila, `top` para pilha) e verificação de vazio simplesmente delegam para as operações correspondentes da lista.

Essa abordagem modular facilitou muito a implementação e os testes, além de garantir consistência no comportamento das estruturas.

Grafo:

Implementei o grafo com suporte para as duas representações pedidas no EP: lista de adjacência (tipo 0) e matriz de adjacência (tipo 1). Para economizar memória, usei um `union`, assim armazenando apenas a estrutura ativa.

Para a lista de adjacência (tipo 0), criei um vetor de ponteiros para `LinkedList`, uma lista para cada vértice. Para inserir uma aresta (u,v), adiciono v na lista de u e u na lista de v, mas primeiro verifico se a aresta já existe usando busca linear. Essa verificação custa $\mathcal{O}$(n^{o}vizinhos(u)). Escolhi essa abordagem porque evita arestas duplicadas. As complexidades das funções básicas são:

- inserção: $\mathcal{O}$(n^{o}vizinhos(u));
- remoção: $\mathcal{O}$(n^{o}vizinhos(u));
- BFS e DFS: $\mathcal{O}$(V+E).

Para a matriz de adjacência (tipo 1), uso uma matriz $n \times n$ de inteiros alocada com `calloc`, garantindo inicialização em zero. As operações de inserção, remoção e teste de existência de aresta são todas $\mathcal{O}(1)$, o que é uma grande vantagem. Porém, BFS e DFS precisam percorrer toda a linha de cada vértice visitado ($\mathcal{O}$(V) por vértice), resultando em $\mathcal{O}(V^2)$.

Para evitar duplicação de código, criei a função auxiliar `create_neg1_array` que aloca um vetor de inteiros de tamanho especificado e inicializa

todos os valores com -1. Esta função é chamada internamente pelas implementações (`bfs_0`, `bfs_1`, `dfs_0`, `dfs_1`). Sem essa função, eu teria que repetir o mesmo loop de inicialização em cada implementação, violando a decisão de modularização que foi definida no início desta seção.

Para validação, criei a função auxiliar `check` que verifica se os vértices estão dentro dos limites antes de qualquer operação e o `Graph*` $\neq$ NULL, prevenindo acessos inválidos. Essa também é uma função global criada para evitar repetições em cada implementação.

Para verificar conectividade, executo BFS a partir do vértice 0 (escolha arbitrária) e verifico se todos os vértices foram alcançados (distância $\neq -1$). Se algum vértice permanecer com distância -1, o grafo é desconexo.

Considerações sobre Gerenciamento de Memória: O enunciado do trabalho impôs restrições ao arquivo `.h`, impedindo que eu incluisse funções de desalocação como `free_graph()`, `free_list()` ou equivalentes para fila e pilha. Dessa forma, embora cada alocação seja necessária (listas de adjacência, matriz, nós, vetores de distâncias/predecessores), não há uma forma para liberar toda essa memória. O que poderia levar à vazamentos de memória. Conceitualmente, a liberação seria simples, para lista de adjacência, percorrer cada lista liberando nós e depois o vetor; para matriz, liberar cada linha e depois o vetor de ponteiros; liberar vetores auxiliares retornados por BFS/DFS após uso.

Testes e Análise de Desempenho:

Realizei os testes pedidos no EP comparando as duas representações de grafo (lista de adjacência vs. matriz de adjacência) em três cenários de tamanho (pequeno, médio, grande) e três níveis de densidade (esparsa $\approx 5\%$, média $\approx 50\%$, densa $\approx 90\%$), usei três densidades pois fiz uma análise adicional além da pedida no EP (para esta utilizei apenas a densidade média). Cada medição foi repetida 15 vezes e calculei a média e desvio padrão dos tempos em microssegundos.

Justificativa dos Parâmetros de Teste:

Escolhi os tamanhos de vértices (10, 100, 500) para testar em diferentes escalas e ver como as complexidades que estudamos aparecem na prática. Grafos pequenos (10 vértices) servem como ponto de partida e ajudam a entender os custos básicos das estruturas. Grafos médios (100 vértices) tenta representar tamanhos que encontramos em problemas reais, onde as diferenças de desempenho começam a ficar maiores. Grafos grandes (500 vértices) aumentam essas diferenças, deixando bem visível qual implementação funciona melhor em cada caso. Tentei usar 1000 vértices, mas para as densidades maiores levou tempos acima dos 10 minutos. Assim, preferi limitar a análise para no máximo 500 vértices

Escolhi as densidades 5%, 50% e 90% para testar os três cenários principais que a teoria prevê. Com 5% (esparsa), espero que a lista de adjacência seja mais eficiente, já que ela só precisa percorrer as poucas arestas que existem. Com 90% (densa), a matriz deveria ser superior porque suas operações $\mathcal{O}(1)$ compensam o fato de precisar examinar muitas posições.

O caso de 50% (média) é o melhor para a análise pedida no EP porque é onde teoricamente é um ponto mais justo de análise onde as duas representações "deveriam"ter desempenho similar. Logo, queria ver na prática se realmente existe esse ponto de equilíbrio e como outros fatores (como acesso à memória) influenciam o resultado.

Análise dos Grafos de Densidade Média: A Tabela 1 apresenta os resultados para grafos com densidade média (≈50% das arestas possíveis):

Tabela 1: *Resultados para Grafos de Densidade Média (os tempos são em μs)*

Tamanho	V	E	Tipo	Construção	BFS	DFS
Pequeno	10	22	Lista	2.85 ± 0.37	0.70 ± 0.10	0.73 ± 0.05
Pequeno	10	22	Matriz	1.86 ± 0.08	1.29 ± 0.07	1.42 ± 0.10
Médio	100	2475	Lista	656.65 ± 50.28	28.20 ± 3.05	27.87 ± 3.59
Médio	100	2475	Matriz	242.10 ± 33.52	71.43 ± 3.53	72.03 ± 3.82
Grande	500	62375	Lista	99880.56 ± 4784.13	3303.43 ± 983.39	1970.97 ± 535.66
Grande	500	62375	Matriz	9665.57 ± 2710.15	1790.51 ± 145.26	1770.46 ± 134.30

Os resultados estão coerentes com a teoria. Para a construção do grafo, a matriz de adjacência é consistentemente mais rápida porque cada inserção de aresta é $\mathcal{O}(1)$, enquanto na lista preciso verificar duplicatas ($\mathcal{O}(n^o vizinhos(u))$. Conforme o grafo cresce, essa diferença se acentua dramaticamente, no grafo grande, a lista leva ≈10 vezes mais tempo que a matriz.

Para BFS e DFS, observo um comportamento interessante, em grafos pequenos, a lista é mais eficiente, mas conforme o tamanho aumenta, a matriz se torna cada vez melhor e até superior. Isso acontece porque na lista percorro apenas as arestas existentes $\mathcal{O}$(V+E), enquanto na matriz percorro toda a linha do vértice $\mathcal{O}(V^2)$. Com densidade média, $E \approx \frac{V^2}{2}$, então as complexidades se aproxi-

mam.

Comparação por Densidade: Esparsa vs. Densa: Além da análise pedida pelo EP fiz uma comparação entre diferentes densidades, a Tabela 2 mostra esses extremos de densidade para grafos de tamanho médio:

Tabela 2: *Impacto da Densidade em Grafos Médios (V=100)*

Densidade	E	Tipo	Construção	BFS	DFS
Esparsa (5%)	247	Lista	397.55 ± 24.34	10.85 ± 0.36	10.39 ± 0.21
Esparsa (5%)	247	Matriz	368.46 ± 56.64	40.03 ± 4.07	38.04 ± 3.70
Densa (90%)	4455	Lista	2091.75 ± 302.21	67.21 ± 6.47	66.44 ± 8.74
Densa (90%)	4455	Matriz	419.06 ± 51.74	47.36 ± 3.42	45.85 ± 1.78

A diferença de densidade revela claramente as forças e fraquezas de cada representação.

Em grafos esparsos, a lista de adjacência é superior para BFS/DFS, sendo cerca de 4 vezes mais rápida que a matriz. Isso confirma o que era esperado, com poucas arestas, percorrer apenas as existentes $\mathcal{O}$(V+E) é muito mais eficiente que examinar toda a matriz $\mathcal{O}(V^2)$.

Por outro lado, em grafos densos, a matriz de adjacência é bem melhor. Para construção, é cerca de 5 vezes mais rápida, e para BFS/DFS, é cerca de 30% mais eficiente. Com muitas arestas, o custo extra de gerenciar listas ligadas supera os benefícios, e a estrutura compacta da matriz oferece melhor desempenho.

Os dados sugerem que o ponto de equilíbrio está na densidade média. Para densidades baixas ($<$ 30%), prefiro lista de adjacência. Para altas densidades ($>$ 70%), a matriz é claramente superior.

Esses resultados vão de acordo com a complexidade esperado, lista de adjacência é ideal para grafos esparsos ($E \ll V^2$), enquanto matriz de adjacência é melhor para grafos densos.

Conclusão:

Implementei duas representações de grafo e um conjunto de estruturas de suporte (lista ligada, fila e pilha) que permitiram executar BFS e DFS. As decisões de manter o ponteiro `tail` na lista, reutilizar a mesma estrutura base para fila e pilha, e separar a validação em função auxiliar contribuíram para um código modular e de fácil entendimento, além de com um custo menor.

Os testes de desempenho confirmaram as previsões sobre complexidade computacional. A lista de adjacência mostrou-se superior para grafos esparsos, especialmente em operações de busca (BFS/DFS), onde seu custo $\mathcal{O}$(V+E) é significativamente menor que o $\mathcal{O}(V^2)$ da matriz quando $E \ll V^2$. Por outro lado, a matriz de adjacência dominou em grafos densos, onde sua estrutura compacta e operações $\mathcal{O}(1)$ para inserção/remoção compensam o maior custo das buscas.

O ponto de inflexão experimental observado (densidade $\approx$ 50%) não significa uma igualdade assintótica exata entre $\mathcal{O}(V+E)$ (lista) e $\mathcal{O}(V^2)$ (matriz). Usando a definição de densidade para grafos não direcionados $E = p\,\dfrac{V(V-1)}{2}$, se igualarmos os custos dominantes $V + E \approx V^2$ obtemos $p = 2$, valor fora do intervalo válido $[0, 1]$. Ao ignorar fatores constantes e escrever $E \approx pV^2$, a igualdade $V + pV^2 \approx V^2$ leva a $p \approx 1 - 1/V \to 1$. A forma simplificada que gera a referência informal $p \sim 2/V$ surge ao comparar apenas o termo linear V com o termo denso $pV^2/2$ (isto é, $V \approx pV^2/2 \Rightarrow p \approx 2/V$). Esses arranjos mostram que, teoricamente, a matriz só "empata"quando o grafo é muito denso. O cruzamento prático perto de $p \approx 0.5$ decorre de constantes e efeitos de implementação. Assim, a densidade de equilíbrio empírico desloca-se para valores intermediários, mesmo que a análise assintótica favoreça a lista até densidades muito altas.

Como aprendizado, destaco que a escolha da representação de grafo deve sempre considerar o padrão de uso esperado, para aplicações com grafos esparsos e operações frequentes, a lista de adjacência é claramente superior. Para grafos densos ou aplicações que requerem muitas consultas diretas de adjacência, a matriz oferece melhor desempenho.