

EXERCÍCIO PROGRAMA II

Luca Marinho Nasser Valadares Paiva

Número USP: 13691375

Introdução:

Neste EP, implementei dois algoritmos baseados em estruturas de heap: o *HeapSort* para ordenação de vetores e o algoritmo de Dijkstra para encontrar caminhos mínimos em grafos. Para o Dijkstra, desenvolvi duas implementações de fila de prioridade: uma baseada em *heap* binário mínimo e outra baseada em lista encadeada ordenada, permitindo comparação direta de desempenho entre os dois. Além disso, construí estruturas auxiliares como lista encadeada e grafo com suporte para matriz de adjacência.

Minha abordagem priorizou três aspectos principais: modularidade, reutilizando a estrutura de heap tanto para HeapSort quanto para Dijkstra; eficiência nas operações críticas (ordenação, inserção, remoção, atualização de prioridade); e clareza. A decisão de implementar o heap de forma genérica, com suporte tanto para max-heap (HeapSort) quanto min-heap (Dijkstra), demonstra como podemos usar uma mesma estrutura de dados para resolver problemas distintos.

Através dos testes realizados, validei empiricamente as previsões teóricas sobre complexidade, $\mathcal{O}(V+E)\log V$, para Dijkstra com heap. A comparação entre heap e lista encadeada no Dijkstra revelou as vantagens da estrutura de heap. As próximas seções detalham os componentes construídos, as decisões de implementação e a análise dos resultados.

Decisões de Implementação:

Esta seção explica e justifica as escolhas que fiz durante o desenvolvimento do EP.

Lista Encadeada:

Implementei a lista ligada como estrutura base em `list.c` e `list.h`. Cada nó da lista é do tipo `List`, contendo uma estrutura `Edge` (que armazena um vértice e seu valor/peso) e um ponteiro `next` para o próximo elemento. A estrutura `LinkedList` mantém ponteiros para `head` (cabeça) e `tail` (cauda), permitindo inserções eficientes em ambas as pontas.

A decisão mais importante foi criar a função `list_insert_sorted`, que insere elementos mantendo a lista ordenada crescente por valor (prioridade). Essa função percorre a lista desde o início até encontrar a posição correta onde o novo elemento deve ser inserido. Isso é essencial para a fila de prioridade, pois garante que o elemento de menor prioridade sempre esteja no início. Com isso, `list_remove_min` pode simplesmente remover o primeiro elemento em $\mathcal{O}(1)$, sem precisar buscar o mínimo.

Porém, essa escolha tem um custo, a inserção ordenada precisa percorrer a lista no pior caso até

o final, resultando em complexidade $\mathcal{O}(n)$. Para grafos com muitas arestas, isso deixa a implementação mais lenta, como veremos nos testes de desempenho.

Outra função crítica é `list_decrease_priority`, usada no Dijkstra quando encontramos um caminho melhor para um vértice. Essa função primeiro busca o vértice na lista (usando `list_remove_with_vert`), remove-o, atualiza sua prioridade e reinsere na posição ordenada correta. Como envolve busca linear, remoção e inserção ordenada, tem custo total $\mathcal{O}(n)$.

Para facilitar a implementação, criei funções auxiliares privadas (marcadas com `static`):

- `add_node`: cria um novo nó com `malloc` e inicializa seus campos;
- `remove_with`: função genérica que remove um elemento buscando por vértice ou por valor, evitando duplicação de código;
- `search_with`: função genérica de busca, também reutilizada por diferentes funções públicas.

As operações básicas e suas complexidades são:

- `list_insert_sorted`: inserção mantendo ordem em $\mathcal{O}(n)$;
- `list_remove_min`: remoção do primeiro elemento em $\mathcal{O}(1)$;
- `list_decrease_priority`: atualização de prioridade em $\mathcal{O}(n)$;
- `list_search_with_vert`: busca linear por vértice em $\mathcal{O}(n)$.

Uma observação importante: escolhi retornar `(Edge){-1, -1}` como valor de erro em funções que retornam `Edge`, já que -1 é um índice inválido de vértice.

Heap Binário:

O heap foi implementado em `heap.c` e `heap.h` como uma estrutura genérica que suporta dois tipos: *max-heap* (para HeapSort) e *min-heap* (para Dijkstra). O heap usa representação por vetor, onde o pai do elemento na posição i está em $(i - 1)/2$, o filho esquerdo em $2i + 1$ e o filho direito em $2i + 2$.

Uma decisão foi usar `union` para os dados. Para max-heap, armazeno diretamente um vetor de inteiros (`int *data`), para min-heap, armazeno um vetor de `Edge` (`Edge *nodes`), onde cada edge contém um vértice e sua distância. Essa abordagem economiza memória e permite reutilizar as funções básicas do heap em ambos os contextos, apenas mudando o campo `type`.

As funções auxiliares privadas `return_parent`, `return_left_child` e `return_right_child` encapsulam o cálculo de índices, tornando o código mais legível e evitando erros de cálculo manual.

A função, que diria mais importante, é a `heapify_down`, que restaura a propriedade do heap fazendo um elemento "descer" na árvore. Ela compara o elemento na posição i com seus filhos e o troca com o maior (max-heap) ou menor (min-heap) entre os três. Esse processo continua recursivamente até o elemento estar na posição correta. Como o heap tem altura $\mathcal{O}(\log n)$, essa função custa $\mathcal{O}(\log n)$ no pior caso.

Para facilitar as comparações sem duplicar código, criei a função `compare_priority` que retorna verdadeiro se o elemento i tem prioridade maior (max-heap) ou menor (min-heap) que o elemento j . Isso mantém o código limpo e evita condicionais espalhadas.

A função `swap_positions` não apenas troca dois elementos no vetor do heap, mas também atualiza o vetor auxiliar `vert_index` quando estamos usando min-heap. Esse vetor é para manter a eficiência do Dijkstra.

Para o min-heap usado no Dijkstra, mantive um vetor adicional `vert_index` que mapeia cada vértice para sua posição atual no heap. Por exemplo, se o vértice 5 está na posição 3 do heap, então `vert_index[5] = 3`. Isso transforma a operação de buscar um vértice no heap de $\mathcal{O}(n)$ (busca linear) para $\mathcal{O}(1)$ (acesso direto por índice). Toda vez que elementos são trocados no heap, atualizo esse vetor para manter a consistência.

Implementei também `heapify_up`, que faz o caminho inverso, "sobe" um elemento comparando-o com seu pai. Essa função é usada em `heap_insert_sorted` (quando inserimos um novo elemento no final e ele precisa "subir") e em `heap_decrease_priority` (quando diminuímos a prioridade de um elemento e ele pode precisar subir).

A função `heap_decrease_priority` usa o vetor `vert_index` para encontrar instantaneamente a posição do vértice, atualiza sua prioridade e chama `heapify_up` para restaurar a propriedade do heap. Sem o `vert_index`, precisaríamos buscar linearmente o vértice a cada atualização, levando a complexidade total do Dijkstra de $\mathcal{O}((V+E)\log V)$ para $\mathcal{O}((V+E)V)$.

As complexidades das operações principais são:

- `heapify_down`: restauração do heap em $\mathcal{O}(\log n)$;
- `heapify_up`: percola elemento para cima em $\mathcal{O}(\log n)$;
- `heap_insert_sorted`: inserção em $\mathcal{O}(\log n)$;
- `heap_remove_min`: remoção do mínimo em $\mathcal{O}(\log n)$;
- `heap_decrease_priority`: atualização de prioridade em $\mathcal{O}(\log n)$ graças ao `vert_index`;

- `swap_positions`: troca com atualização de índices em $\mathcal{O}(1)$.

Essa estrutura permite tanto ordenação (HeapSort) quanto gerenciamento de prioridades (Dijkstra).

HeapSort:

O HeapSort foi implementado em `algorithms.c` seguindo o algoritmo em duas fases. A primeira fase, implementada em `create_max_heap`, constrói um max-heap a partir do vetor desordenado. A ideia é chamar `heapify_down` para cada nó não-folha, começando do último nó não-folha (posição $n/2 - 1$) e indo até a raiz (posição 0). Fazer de baixo para cima garante que quando processamos um nó, seus filhos já formam heaps válidos.

A segunda fase, implementada em `heap_sort`, extrai repetidamente o máximo. O algoritmo funciona assim:

1. Troca a raiz (maior elemento) com o último elemento do heap;
2. Reduz o tamanho do heap em 1 (excluindo o último elemento, que agora está ordenado);
3. Chama `heapify_down` na raiz para restaurar a propriedade do max-heap;
4. Repete até o heap ficar vazio.

São $n - 1$ iterações, cada uma custando $\mathcal{O}(\log n)$ para o `heapify`, resultando em complexidade total $\mathcal{O}(n \log n)$ para essa fase. Somando com a construção do heap ($\mathcal{O}(n)$), a complexidade total do HeapSort é $\mathcal{O}(n \log n)$.

Eu decidi por fazer todas as modificações *in-place*. O vetor original é reordenado sem necessidade de vetor auxiliar. À medida que o heap diminui, os elementos ordenados se acumulam no final do vetor. Isso economiza memória e é uma vantagem clássica do HeapSort sobre MergeSort, que precisa de $\mathcal{O}(n)$ de espaço extra.

Note que em `create_max_heap`, não aloco um novo vetor para o heap. Em vez disso, faço o campo `data` do heap apontar diretamente para o vetor de entrada. Isso é seguro porque o HeapSort modifica o vetor original, não cria uma cópia. No final, apenas libero a estrutura `Heap` (com `free(heap)`), mas não o vetor de dados, pois ele pertence ao "usuário".

Fila de Prioridade:

A estrutura `Priority_Queue` contém um campo `type` (que pode ser `heap` ou `linked_list`) e um `union` que armazena ou um ponteiro para `Heap` ou para `LinkedList` (novamente uso `union` para economizar memória).

As funções públicas da fila de prioridade (`pq_insert_sorted`, `pq_remove_min`, `pq_decrease_priority`, `pq_is_empty`) são wrappers que verificam o tipo e delegam para as funções correspondentes da estrutura ativa. Por exemplo:

```

Status pq_insert_sorted(Priority_Queue *pq,
                       Edge data) {
    if(pq->type == heap) {
        return heap_insert_sorted(
            pq->heap, data);
    } else {
        return list_insert_sorted(
            pq->list, data);
    }
}

```

Essa abstração mantém o código do Dijkstra completamente independente da implementação escolhida. Para testar ambas as versões, basta mudar um parâmetro na criação da fila de prioridade. Isso torna o código mais modular e facilita a comparação de desempenho, já que o algoritmo de Dijkstra é exatamente o mesmo nos dois casos.

Grafo e Algoritmo de Dijkstra:

Implementei o grafo em `graph.h` e `graph.c` com suporte para duas representações: matriz de adjacência e lista de adjacência. O campo `type` indica qual representação está ativa, e um `union` armazena ou `adj_matrix` (matriz $n \times n$) ou `adj_list` (vetor de n listas encadeadas).

Para os testes de desempenho, criei grafos completos onde cada par de vértices tem uma aresta com peso aleatório entre 1 e 100. Isso gera $E = \frac{V(V - 1)}{2}$ arestas. Por exemplo, para $V = 100$, temos 4.950 arestas. Eu utilizei a implementação de grafos com matriz de adjacência para os testes, pois no EP passado mostramos que essa implementação é mais eficiente para grafos densos, que era o caso dos grafos testados.

O algoritmo de Dijkstra foi implementado em `dijkstra_aux` dentro de `algorithms.c`, que recebe o tipo de fila de prioridade como parâmetro. Isso permite testar ambas implementações com exatamente o mesmo código. As funções `dijkstra_heap` e `dijkstra_list` são apenas wrappers que chamam `dijkstra_aux` com o tipo apropriado.

A implementação segue o algoritmo clássico de Dijkstra:

1. Inicializo todas as distâncias como INF (42424242, a resposta para tudo), exceto a origem que recebe distância 0;
2. Crio um vetor `visited` inicializado com zeros (nenhum vértice visitado);
3. Insiro a origem na fila de prioridade com prioridade 0;
4. Enquanto a fila não está vazia:
 - Removo o vértice u de menor distância da fila;
 - Se u já foi visitado (`visited[u] == 1`), pula para o próximo (isso evita processar o mesmo vértice múltiplas vezes);

- Marco u como visitado (`visited[u] = 1`);
- Para cada vizinho v de u não visitado:
 - Calculo a distância $new_dist = dist[u] + peso(u, v)$;
 - Se $new_dist < dist[v]$ (encontrei um caminho melhor):
 - * Se $dist[v]$ era INF, insiro v na fila pela primeira vez;
 - * Caso contrário, atualizo a prioridade de v na fila com `decrease_priority`;
 - * Atualizo $dist[v] = new_dist$.

Uma decisão importante foi usar o vetor `visited` para marcar vértices já processados. Isso é necessário porque quando fazemos `decrease_priority`, não removemos a entrada antiga da fila (seria muito custoso buscar e remover). Então o mesmo vértice pode aparecer múltiplas vezes na fila com prioridades diferentes. O `visited` garante que processamos cada vértice apenas uma vez, na primeira vez que ele é removido (com a menor distância possível).

Separrei o processamento de vizinhos em duas funções privadas: `process_matrix_neighbors` (que percorre a linha da matriz de adjacência) e `process_list_neighbors` (que percorre a lista de adjacência).

Uma observação sobre memória, o Dijkstra aloca dinamicamente os vetores `dist` e `visited` com `malloc` e `calloc`. No final, libera apenas `visited` e a fila de prioridade, mas retorna o vetor `dist` para o "usuário". É responsabilidade dele de liberar esse vetor depois de usá-lo.

Considerações sobre Complexidade:

Para o **HeapSort**, a complexidade é $\mathcal{O}(n \log n)$ no pior caso, melhor caso e caso médio. Isso contrasta com QuickSort, que tem $\mathcal{O}(n^2)$ no pior caso (quando o pivô é sempre o menor ou maior elemento), e o coloca junto com MergeSort. A vantagem do HeapSort é ser *in-place* ($\mathcal{O}(1)$ de espaço extra além do vetor), enquanto MergeSort precisa de $\mathcal{O}(n)$ para os vetores auxiliares usados.

Para o **Dijkstra com heap**, a complexidade é $\mathcal{O}((V+E)\log V)$. Vamos entender de onde vem essa fórmula:

- Cada vértice é inserido na fila uma vez: V inserções $\times \mathcal{O}(\log V) = \mathcal{O}(V \log V)$;
- Cada vértice é removido da fila uma vez: V remoções $\times \mathcal{O}(\log V) = \mathcal{O}(V \log V)$;
- Cada aresta pode causar um `decrease_priority`: até E operações $\times \mathcal{O}(\log V) = \mathcal{O}(E \log V)$.

Somando tudo, temos $\mathcal{O}(V \log V + V \log V + E \log V) = \mathcal{O}((V+E)\log V)$.

Para o **Dijkstra com lista encadeada** temos:

- Cada `remove_min` é $\mathcal{O}(1)$ porque o mínimo está sempre no início: V remoções $\times \mathcal{O}(1) = \mathcal{O}(V)$;

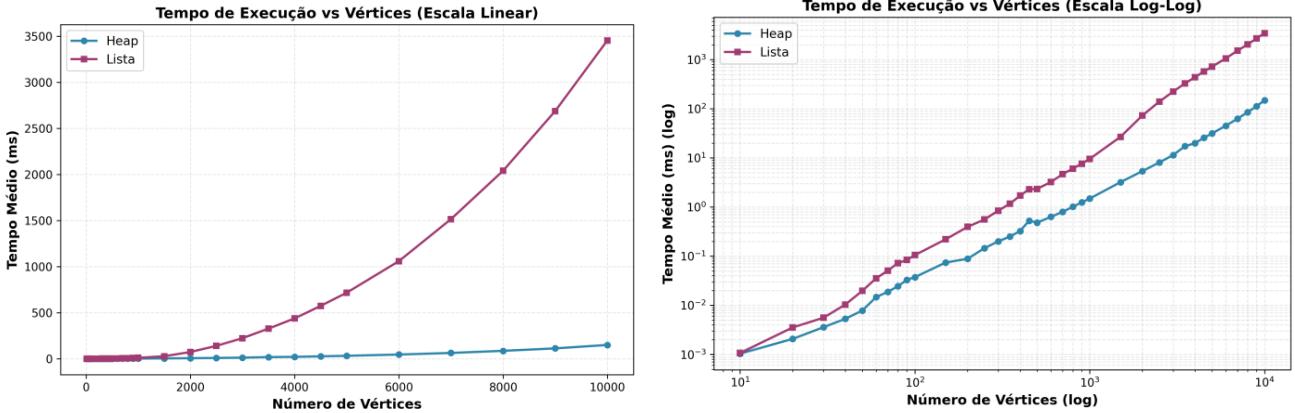


Figura 1: Comparação de tempo de execução em escala linear e log-log (1 a 10.000 vértices)

- Cada inserção ordenada custa até $\mathcal{O}(V)$ no pior caso: até V inserções $\times \mathcal{O}(V) = \mathcal{O}(V^2)$;
- Cada `decrease_priority` precisa buscar o vértice ($\mathcal{O}(V)$), remover ($\mathcal{O}(1)$ depois de encontrado) e reinserir ordenado ($\mathcal{O}(V)$): até E operações $\times \mathcal{O}(V) = \mathcal{O}(EV)$.

Somando, temos $\mathcal{O}(V + V^2 + EV) = \mathcal{O}(V^2 + EV)$.

Testes e Análise de Desempenho:

Realizei testes para validar ambas implementações: Heap vs Lista. Fiz um teste seguindo a especificação do enunciado, usando um grafo completo de $n = 100$ vértices (totalizando 4.950 arestas), executando cada implementação 30 vezes e calculando a média dos tempos. Usei matriz de adjacência para ambas as versões (heap e lista), mantendo o mesmo grafo em todas as execuções (fixando a seed do gerador aleatório) para garantir uma comparação justa.

Resultados para n=100 Vértices:

A tabela apresenta os resultados obtidos para o grafo completo com 100 vértices:

Podemos ver que para grafos de até ≈ 1.000 vértices, a diferença entre as implementações não é muito perceptível. Porém, a partir daí, a diferença começa a crescer rapidamente, e a curva da lista encadeada dispara, confirmando que o comportamento assintótico passa a dominar para tamanhos maiores.

A em escala log-log é particularmente boa para análise de complexidade. Observando os resultados:

- A linha do heap tem inclinação menor, correspondendo ao termo V^2 que domina em $(V+E)\log V$ quando $E \approx V^2$. O termo lo-

Tabela 1: Comparação de Desempenho - Dijkstra com $n=100$ (tempos em ms)

Implementação	Tempo Médio
Heap	0,0373
Lista ligada	0,1050

Os resultados mostraram que o heap é **2,82 vezes mais rápido** que a lista encadeada, ou seja, o heap reduz o tempo de execução em 64,5%. A diferença absoluta de tempo é de 0,0677 ms, o que pode parecer pequeno em valor absoluto, mas representa quase 3 vezes o tempo de execução do heap.

Análise Extra:

Para entender melhor o comportamento das implementações e verificar como a diferença cresce com o tamanho do problema, realizei testes adicionais com diferentes tamanhos de grafo (de 10 até 10.000 vértices), sempre mantendo o grafo completo. Meu objetivo era montar uma curva para poder visualizar a diferença entre as implementações. As figuras mostram a comparação em escala linear e log-log. Eu realizei as plotagens utilizando python, apenas copiei os resultados obtidos nas simulações para um csv e a partir daí preparei as visualizações.

garítmico adiciona um fator multiplicativo pequeno que não muda a inclinação;

- A linha da lista tem inclinação maior (próxima de 3), correspondendo ao crescimento cúbico V^3 da complexidade EV quando $E \approx V^2$;

Para $n = 1.000$ vértices, o ganho de velocidade foi de aproximadamente 12×. Para $n = 5.000$ vértices cerca de 20×. E para $n = 10.000$ vértices de aproximadamente 23×. Isso confirma que para grafos grandes, a escolha da estrutura de dados para fila de prioridade é muito importante para o desempenho do Dijkstra.

Conclusão:

Implementei os dois algoritmos propostos no EP: HeapSort para ordenação e Dijkstra para encontrar caminhos mínimos em grafos. Para o Dijkstra, desenvolvi duas implementações completas de fila de prioridade (heap binário e lista ligada ordenada).

As decisões foram pensadas para equilibrar eficiência e clareza. Criar uma estrutura de heap genérica (suportando tanto max-heap quanto min-heap com um campo `type`) permitiu reutilizar o mesmo código para contextos diferentes. Manter o vetor `vert_index` no min-heap foi importante para otimizar `decrease_priority` de $\mathcal{O}(n)$ para $\mathcal{O}(\log n)$. Reutilizar funções como `heapify_down`, `swap_positions` e as funções de navegação (pai/filhos) em múltiplos contextos contribuiu para um código mais modular.

Os testes de desempenho do Dijkstra confirmaram as previsões teóricas. Para o grafo completo com $n = 100$ vértices especificado no enunciado, o heap foi 2,82 vezes mais rápido que a lista encadeada, executando em média 0,0373 ms contra 0,1050 ms da lista. Essa diferença de 64,5% de redução no tempo valida quantitativamente a análise de que $\mathcal{O}((V+E)\log V)$ é significativamente melhor que $\mathcal{O}(V^2+EV)$.

A análise extra que facilitou a observação do padrão de ganho não constante, aumentando exponencialmente conforme o tamanho do grafo cresce. De 2,82× para 100 vértices, chegamos a aproximadamente 23× para 10.000 vértices. Os gráficos em escala log-log confirmam visualmente as diferenças de crescimento assintótico, aproximadamente quadrático (com fator logarítmico) para o heap versus cúbico para a lista em grafos completos.