**UNIVERSIDADE FEDERAL DO CEARÁ - CAMPUS SOBRAL**

**CURSO DE ENGENHARIA DE COMPUTAÇÃO**

**DISCIPLINA: TÓPICOS ESPECIAIS EM COMPUTAÇÃO II E ESTUDOS ESPECIAIS**

**PROFESSOR: FISCHER JÔNATAS FERREIRA**

# TRABALHO PRÁTICO 01

**ALUNO - MATRÍCULA**

**ALEX DE SOUSA RAMOS - 561822**

**AKYLA DE AQUINO PINTO - 412723**

**FRANCISCO CASSIANO DE VASCONCELOS SOUZA - 413067**

**IZAIAS MACHADO PESSOA NETO - 497372**

**JOÃO PAULO DE ABREU MILITAO - 494959**

**MARCOS VINICIUS ANDRADE DE SOUSA - 496788**

**RAFAEL BENVINDO HOLANDA MENDES - 415546**

**SAMYLE DE SOUSA ARAUJO - 418797**

**SOBRAL**

**2023**

## LISTA DE ILUSTRAÇÕES

Figura 1 – Representação da Árvore Binária . . . . . 14
Figura 2 – Representação da Árvore AVL . . . . . 15
Figura 3 – Representação da uma Heap de Fibonacci . . . . . 15
Figura 4 – Gráfico do tempo de execução para a Mochila Fracionária . . . . . 31
Figura 5 – Gráfico do consumo de memória da Mochila Fracionária . . . . . 31

## LISTA DE TABELAS

Tabela 1 – Complexidade teórica . . . . . 12
Tabela 2 – Complexidade teórica . . . . . 15
Tabela 3 – Resultados PPH . . . . . 28
Tabela 4 – Resultados por ordenação . . . . . 29
Tabela 5 – Resultados usando a medina das medianas . . . . . 29
Tabela 6 – Resultados por partições sucessivas pela média . . . . . 30
Tabela 7 – Resultado árvore binária . . . . . 32
Tabela 8 – Resultado árvore AVL . . . . . 33
Tabela 9 – Resultado árvore binária . . . . . 34

**SUMÁRIO**


**1 INTRODUÇÃO** . . . . . 5

**2 FUNDAMENTAÇÃO TEÓRICA** . . . . . 6

**2.1 Problema de Programação Hiperbólica (PPH)** . . . . . 6

***2.1.1 Análise da Complexidade e Lógica do Código*** . . . . . 6

***2.1.2 Complexidade do Código*** . . . . . 6

***2.1.3 Lógica do Código*** . . . . . 6

**2.2 Relação entre o Problema de Programação Hiperbólica (PPH) e Algoritmos de Ordenação** . . . . . 7

***2.2.1 Observações do PPH na Projeção de Algoritmos de Ordenação*** . . . . . 7

***2.2.2 Aplicação Prática nos Algoritmos de Ordenação*** . . . . . 8

***2.2.3 Significado da Relação*** . . . . . 8

**2.3 Caracterização do Conjunto $S^*$ no Contexto do PPH** . . . . . 8

**2.4 Análise de Complexidade dos Algoritmos de Ordenação** . . . . . 9

***2.4.1 Complexidade das instâncias testadas*** . . . . . 9

***2.4.2 Realização de Experimentos Computacionais Comparativos*** . . . . . 9

**2.5 Problema da Mochila Fracionária** . . . . . 11

***2.5.1 O problema da Mochila Fracionária*** . . . . . 11

***2.5.2 Mochila Fracionária por Ordenação*** . . . . . 11

***2.5.3 Mochila Fracionária utilizando a Mediana das Medianas*** . . . . . 12

***2.5.4 Mochila Fracionária por partições sucessivas utilizando a média*** . . . . . 12

***2.5.5 Complexidade teórica*** . . . . . 12

**2.6 Algoritmo de Prim** . . . . . 13

***2.6.1 Árvore Binária*** . . . . . 13

***2.6.2 AVL (Árvore de Busca Balanceada)*** . . . . . 14

***2.6.3 Heap de Fibonacci*** . . . . . 14

***2.6.4 Complexidade teórica*** . . . . . 15

**3 METODOLOGIA** . . . . . 16

**3.1 Problema da Mochila Fracionária** . . . . . 16

***3.1.1 Geração de entradas*** . . . . . 16

***3.1.2 Implementação e execução*** . . . . . 16

*3.1.3* ***Coleta e processamento dos dados*** . . . . . 18
**3.2** **Algoritmo de Prim** . . . . . 19
*3.2.1* ***Árvore Binária*** . . . . . 19
*3.2.1.1* *BinaryTree* . . . . . *19*
*3.2.1.2* *Node* . . . . . *20*
*3.2.1.3* *Prim_AB* . . . . . *20*
*3.2.1.4* *Dificuldades relacionadas* . . . . . *22*
*3.2.2* ***AVL (Árvore de Busca Balanceada)*** . . . . . 23
*3.2.2.1* *AVLNode* . . . . . *23*
*3.2.2.2* *AVLTree* . . . . . *23*
*3.2.2.3* *Dificuldades relacionadas* . . . . . *25*
*3.2.3* ***Heap Fibonacci*** . . . . . 25
*3.2.3.1* *FibonacciHeapNode* . . . . . *26*
*3.2.3.2* *FibonacciHeap* . . . . . *26*
*3.2.3.3* *Dificuldades relacionadas* . . . . . *27*
**4** **RESULTADOS** . . . . . 28
**4.1** **Problema de Programação Hiperbólica (PPH)** . . . . . 28
**4.2** **Problema da Mochila Fracionária** . . . . . 28
*4.2.1* ***Mochila Fracionária por Ordenação*** . . . . . 28
*4.2.2* ***Mochila Fracionária utilizando a Mediana das Medianas*** . . . . . 29
*4.2.3* ***Mochila Fracionária por partições sucessivas utilizando a média*** . . . . . 29
*4.2.4* ***Comparativo dos resultados*** . . . . . 30
*4.2.5* ***Analise gráfica dos testes*** . . . . . 30
**4.3** **Algoritmo de Prim** . . . . . 31
*4.3.1* ***Ávore binária*** . . . . . 32
*4.3.1.1* *Árvore AVL* . . . . . *33*
*4.3.1.2* *Heap Fibonacci* . . . . . *34*
**5** **CONCLUSÕES** . . . . . 35
**5.1** **Execução do Problema de Programação Hiperbólica (PPH)** . . . . . 35
*5.1.1* ***Algoritmos de Ordenação e sua Relação com o PPH*** . . . . . 35
*5.1.2* ***Experimentos Computacionais e Insights*** . . . . . 36
**5.2** **Problema da Mochila Fracionária** . . . . . 36

**5.3 Algoritmo de Prim** . . . . . 37

# 1 INTRODUÇÃO

Este trabalho visa de forma prática realizar experimentos com algoritmos para avaliar seu comportamento dado um conjunto de entradas. Para isso, o trabalho foi dividido em uma fundamentação teórica (Capítulo 2) para a explicação dos problemas que os algoritmos implementados se propõe a solucionar. Os experimentos seguem uma metodologia de estudo (Capítulo 3) que foi seguida a fim de produzir e analisar os resultados (Capítulo 3). Por fim, posterior a análise dos resultados, podem ser tiradas conclusões (Capítulo 5) que relacionam o comportamento esperado na fundamentação teórica e o que foi encontrado nos experimentos.

# 2 FUNDAMENTAÇÃO TEÓRICA

## 2.1 Problema de Programação Hiperbólica (PPH)

O Problema da Programação Hiperbólica (PPH) é uma classe de problemas de otimização não linear que tem aplicações em diversas áreas, como engenharia, economia, física e ciência da computação. Esse problema é caracterizado por ser não convexo, o que significa que ele pode ter múltiplos mínimos locais e pode ser difícil de resolver devido à presença de curvas hiperbólicas em sua função objetivo.

O objetivo do PPH é encontrar um conjunto de valores para as variáveis de decisão $x$ que minimizem a função objetivo $f(x)$, enquanto satisfazem todas as restrições. Esses valores são chamados de solução ótima.

1. Função objetivo:

$$f(x) = \frac{a_0 + \sum_{i \in S} a_i}{b_0 + \sum_{i \in S} b_i}$$

### *2.1.1 Análise da Complexidade e Lógica do Código*

### *2.1.2 Complexidade do Código*

O código apresenta uma complexidade de $O(n \cdot m)$, onde $n$ é o número de iterações e $m$ é o número total de arquivos (99, neste caso). Isso se deve ao aninhamento de dois loops: um loop principal que executa o código enquanto o tempo decorrido é menor que o tempo desejado e um loop interno que itera sobre os arquivos de 1 a 99.

A função `funcao_formatacao` e a função `pph` são chamadas dentro do loop interno, contribuindo para a complexidade total.

### *2.1.3 Lógica do Código*

A lógica do código é projetada para executar o Problema de Programação Hiperbólica (PPH) em múltiplos arquivos de entrada para uma instância específica. Aqui está a lógica detalhada:

1. **Importação de Módulos:** Importam-se os módulos necessários, incluindo funções para formatação de arquivos e resolução do PPH.

2. **Definição de Função:** Define-se a função `executa_pph_o2`, que recebe uma instância como parâmetro.
3. **Configuração do Tempo:** Define-se o tempo desejado para a execução (5 segundos) e registra-se o tempo inicial.
4. **Loop Principal:** Um loop principal é iniciado para executar o código enquanto o tempo decorrido é menor que o tempo desejado.
5. **Loop Interno:** Dentro do loop principal, há um loop interno que itera sobre os arquivos de 1 a 99. Cada iteração envolve:
    - Construção do caminho do arquivo.
    - Formatação do arquivo usando `funcao_formatacao`.
    - Resolução do PPH usando `pph`.
    - Impressão de resultados parciais.
    - Contagem do número total de iterações.
6. **Verificação de Tempo:** Verifica-se o tempo atual e interrompe-se o loop principal se o tempo decorrido for maior ou igual ao tempo desejado.
7. **Retorno:** Retorna o número total de iterações realizadas.

A lógica busca explorar diversas iterações do PPH dentro do tempo especificado, fornecendo resultados parciais a cada iteração.

## 2.2 Relação entre o Problema de Programação Hiperbólica (PPH) e Algoritmos de Ordenação

A relação entre o Problema de Programação Hiperbólica (PPH) e o problema de ordenação reside na utilização de observações sobre o PPH para projetar algoritmos de ordenação com diferentes complexidades.

### *2.2.1 Observações do PPH na Projeção de Algoritmos de Ordenação*

A observação sobre o PPH é fundamental na concepção de algoritmos de ordenação. Em particular, o PPH é empregado como base para criar algoritmos de ordenação com complexidades específicas, abrangendo tanto $O(n^2)$ quanto $O(n \log n)$. O lema do PPH é destacado como uma ferramenta essencial para a elaboração desses algoritmos de ordenação.

### *2.2.2 Aplicação Prática nos Algoritmos de Ordenação*

Os algoritmos de ordenação apresentados no documento, tais como Bubble sort, Insertion sort, Selection sort, Merge-sort, Quick-sort e Heap-sort, são meticulosamente projetados com base nas observações relacionadas ao PPH.

### *2.2.3 Significado da Relação*

## 2.3 Caracterização do Conjunto $S^*$ no Contexto do PPH

No contexto do Problema de Programação Hiperbólica (PPH), o conjunto $S^*$ é caracterizado como o conjunto que maximiza a função $R(S)$. A função $R(S)$ é definida como a soma dos elementos $a_i$ para $i$ pertencente a $S$, adicionado ao elemento $a_0$, dividido pelo somatório dos elementos $b_i$ para $i$ pertencente a $S$, adicionado ao elemento $b_0$.

Matematicamente, a função $R(S)$ é expressa como:

$$R(S) = \frac{\sum_{i \in S} a_i + a_0}{\sum_{i \in S} b_i + b_0}$$

Essa função $R(S)$ representa uma razão que envolve a soma ponderada dos elementos de dois conjuntos, sendo $a$ e $b$ os conjuntos de elementos $a_i$ e $b_i$ respectivamente, com um elemento adicional $a_0$ no numerador e $b_0$ no denominador.

A caracterização de $S^*$ como o conjunto que maximiza essa razão é crucial para o projeto de algoritmos com complexidade $O(n)$. A maximização dessa função, conforme descrita na caracterização do conjunto $S^*$, geralmente implica somatórios ponderados de elementos de conjuntos, tornando o problema intrinsecamente mais complexo e desafiador de ser resolvido em complexidade linear.

Portanto, a relação intrínseca reside na aplicação criativa dos conceitos e observações relacionados ao PPH para a criação de algoritmos de ordenação eficientes com diferentes complexidades. Essa abordagem não apenas destaca a versatilidade do PPH, mas também demonstra como os princípios de otimização não linear podem ser extrapolados e aplicados de maneira inovadora em outros contextos computacionais, como a ordenação de dados.

## 2.4 Análise de Complexidade dos Algoritmos de Ordenação

### *2.4.1 Complexidade das instâncias testadas*

Os algoritmos de ordenação implementados no código apresentam diferentes complexidades de tempo. Vamos analisar cada um deles:

1. **Bubble Sort:** $O(n^2)$ - O pior caso ocorre quando o array está completamente desordenado, levando a $n \times (n-1)/2$ comparações e trocas.
2. **Insertion Sort:** $O(n^2)$ - O pior caso também ocorre com um array completamente desordenado, resultando em $n \times (n-1)/2$ comparações e movimentos.
3. **Selection Sort:** $O(n^2)$ - Independentemente da ordem do array de entrada, há $n \times (n-1)/2$ comparações e trocas no pior caso.
4. **Merge Sort:** $O(n \log n)$ - Este algoritmo divide o array pela metade recursivamente e depois mescla as partes ordenadas, resultando em uma complexidade de $O(n \log n)$.
5. **Quick Sort:** $O(n \log n)$ em média, $O(n^2)$ no pior caso - O pivô é escolhido e o array é particionado, mas o desempenho depende da escolha do pivô. Em média, o desempenho é $O(n \log n)$, mas no pior caso pode ser $O(n^2)$.
6. **Heap Sort:** $O(n \log n)$ - O array é transformado em uma heap e então é ordenado. A fase de criação da heap tem complexidade $O(n)$, e a fase de ordenação tem complexidade $O(n \log n)$.

Portanto, os algoritmos de ordenação apresentados têm complexidades diferentes, variando entre $O(n^2)$ e $O(n \log n)$. O Heap Sort destaca-se com complexidade $O(n \log n)$ em ambos os casos.

### *2.4.2 Realização de Experimentos Computacionais Comparativos*

Para realizar experimentos computacionais comparativos, você pode executar os algoritmos de ordenação em diferentes conjuntos de dados e comparar o tempo de execução. No seu código, há uma estrutura para medir o tempo de execução de cada algoritmo em uma lista de números aleatórios. Vamos discutir um exemplo de como conduzir essas experiências:

```
1 import time
2 import random
3
4 #cdigo de algoritmos e funes aqui...
```

```
5
6  def run_experiment(algorithm, data_size):
7      random_list = random.sample(range(1, 100), data_size)
8
9      start_time = time.time()
10     algorithm(random_list.copy())
11     end_time = time.time()
12
13     return end_time - start_time
14
15 # Defina tamanhos diferentes de conjuntos de dados
16 data_sizes = [10, 100, 500, 1000, 5000]
17
18 # Execute cada algoritmo para cada tamanho de conjunto de dados
19 for algorithm_name, algorithm_function in [
20     ("Bubble Sort", bubble_sort),
21     ("Insertion Sort", insertion_sort),
22     ("Selection Sort", selection_sort),
23     ("Merge Sort", merge_sort),
24     ("Quick Sort", lambda x: quick_sort(x, sequential_pivot_partition)),
25     ("Heap Sort", heap_sort)
26 ]:
27     print(f"Experincias para {algorithm_name}:")
28     for size in data_sizes:
29         time_taken = run_experiment(algorithm_function, size)
30         print(f"Tamanho {size}: {time_taken} segundos")
31     print()
```

Neste exemplo, cada algoritmo é executado em diferentes tamanhos de conjuntos de dados, e o tempo de execução é medido. Os resultados podem ser utilizados para comparar o desempenho relativo dos algoritmos para diferentes tamanhos de entrada.

## 2.5 Problema da Mochila Fracionária

O Problema da Mochila Fracionária, também conhecido como Problema da Mochila Contínua ou *Fractional Knapsack Problem* em inglês, é um importante problema de otimização combinatória. Neste problema, você é apresentado a um conjunto de objetos, cada um com um peso e um valor associados, e o objetivo é determinar a melhor maneira de preencher uma mochila com capacidade limitada de forma a maximizar o valor total, levando em consideração que você pode escolher frações dos objetos.

### *2.5.1 O problema da Mochila Fracionária*

Nesse contexto, dado um conjunto de $n$ objetos, onde cada objeto $i$ tem um peso $p_i$ e um valor $v_i$, e dada uma capacidade máxima da mochila $c$, é retornado a soma dos valores dos itens que foram escolhidos para ocupar a mochila.

1. Restrição de capacidade:

$$\sum_{i=1}^{n} x_i \cdot p_i \leq c$$

2. Função objetivo a ser maximizada:

$$\max \sum_{i=1}^{n} x_i \cdot v_i$$

O desafio aqui é decidir qual fração de cada objeto levar na mochila de forma a atender às restrições de capacidade e maximizar o valor total.

A principal característica do Problema da Mochila Fracionária é a flexibilidade para escolher frações de objetos, o que o torna mais simples de resolver do que o Problema da Mochila Booleana (onde os objetos devem ser escolhidos na íntegra ou não). A solução pode ser obtida com um algoritmo guloso eficiente, como é descrito na Subseção 2.5.2.

### *2.5.2 Mochila Fracionária por Ordenação*

Uma solução ótima para o Problema da Mochila Fracionária envolve a escolha de frações dos objetos de acordo com a relação valor-peso ($v_i/p_i$). Geralmente, os objetos são ordenados de forma decrescente de acordo com essa relação e, em seguida, o algoritmo escolhe os objetos com as maiores relações valor-peso primeiro, preenchendo a mochila até que a capacidade seja atingida.

### *2.5.3 Mochila Fracionária utilizando a Mediana das Medianas*

Para uma solução não ordenada, o algoritmo da mochila fracionária é implementado recursivamente, tendo como entrada índices de início e fim do vetor de itens (**??**). Por conseguinte, é chamada o algoritmo `Select-BFPRT` (**??**), que divide ajuda a separar o vetor em duas partes, uma em que todos os elementos são menores que a media do vetor e a outra parte tem o restante dos elementos.

Nesse sentido, se for possível colocar na mochila todos os elementos maiores que a mediana dentro da mochila, a soma de seus valores é adicionada no conjunto solução e a capacidade da mochila é reduzida. Após isso, o algoritmo é chamado para o outro lado do vetor. Já se não for possível colocar todos os itens maiores que a mediana na mochila, o algoritmo é chamado somente para o lado direito (maior). Com isso, a capacidade não é alterada e na chamada recursiva, é feita novamente mediana, mas somente para uma parte do vetor, que é maior que a primeira mediana.

### *2.5.4 Mochila Fracionária por partições sucessivas utilizando a média*

Assim como o algoritmo descrito na subseção 2.5.3, este algoritmo faz partições sucessivas. Entretanto, nesse caso, ao invés de utilizar o algoritmo de seleção da mediana das medianas, é implementado um algoritmo de partição que tira a média da razão elementos do conjunto e coloca todos os itens com valores de razão maiores que a média para um lado e o restante para outro lado.

### *2.5.5 Complexidade teórica*

Tabela 1 – Complexidade teórica

| Algoritmo | Complexidade Temporal | Complexidade Espacial |
|---|---|---|
| Ordenada | $O(nlog(n))$ | $O(nlog(n))$ |
| Mediana das Medianas | $O(n)$ | $O(n)$ |
| Partições usando a média | $O(n)$ | $O(n)$ |

## 2.6 Algoritmo de Prim

O algoritmo de Prim, assim como o algoritmo de Kruskal, é um exemplo de um algoritmo ganancioso. A abordagem do algoritmo de Prim começa com a escolha de um único nó inicial e, a partir desse ponto, gradualmente explora os nós adjacentes, incorporando as arestas de menor peso ao longo do caminho. Esse processo é repetido até que todos os nós do grafo estejam incluídos na árvore geradora mínima. Basicamente, o algoritmo de Prim busca construir a árvore de maneira incremental, selecionando sempre a aresta de menor custo disponível, garantindo que a árvore resultante seja mínima em termos de peso total.

O algoritmo de Prim inicia com uma árvore geradora vazia e segue uma abordagem que envolve a gestão de dois conjuntos de vértices. O primeiro conjunto é composto pelos vértices já incorporados na Árvore de Abrangência Mínima (Minimum Spanning Tree - MST), enquanto o segundo conjunto abriga os vértices que ainda não fazem parte dela.

A cada etapa do processo, o algoritmo analisa todas as arestas que conectam esses dois conjuntos e, entre elas, escolhe a aresta com o menor peso. Posteriormente, ele transfere a extremidade oposta dessa aresta para o conjunto que já contém a MST.

Essencialmente, o algoritmo de Prim funciona selecionando as arestas de menor custo que conectam o MST em crescimento aos vértices não incluídos, garantindo assim que a árvore geradora mínima seja construída passo a passo, com atenção constante para minimizar o peso total da árvore.

O objetivo deste trabalho é apresentar a implementação do Algoritmo de Prim, utilizando três estruturas de dados diferentes para selecionar o vértice mais próximo da árvore corrente.

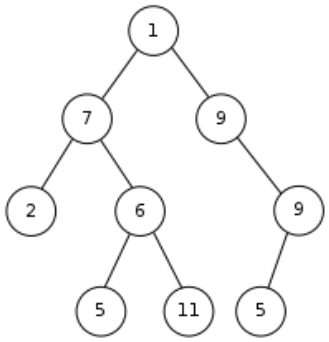
1. Árvore Binária (sem se preocupar com o balanceamento)
2. AVL (Árvore de Busca Balanceada)
3. Heap de Fibonacci

### *2.6.1 Árvore Binária*

O termo ”raiz” se refere ao nó mais elevado em uma árvore binária, enquanto o termo ”folhas” é utilizado para descrever os nós situados na parte mais baixa da árvore. Pode-se conceitualizar uma árvore binária como uma estrutura hierárquica, onde a raiz ocupa a posição superior e as folhas se encontram na base da hierarquia.

Árvores binárias desempenham um papel significativo na área da ciência da computação, sendo aplicadas em diversas áreas, como armazenamento e recuperação de informações, avaliação de expressões matemáticas, roteamento de redes e desenvolvimento de algoritmos de inteligência artificial em jogos. Além disso, essas estruturas de dados podem ser empregadas na implementação de algoritmos de busca, ordenação e construção de estruturas de gráficos. Na Figura 1 pode-se ver a representação de uma árvore binária.

Figura 1 – Representação da Árvore Binária




Fonte: https://en.wikipedia.org/wiki/Binary_tree

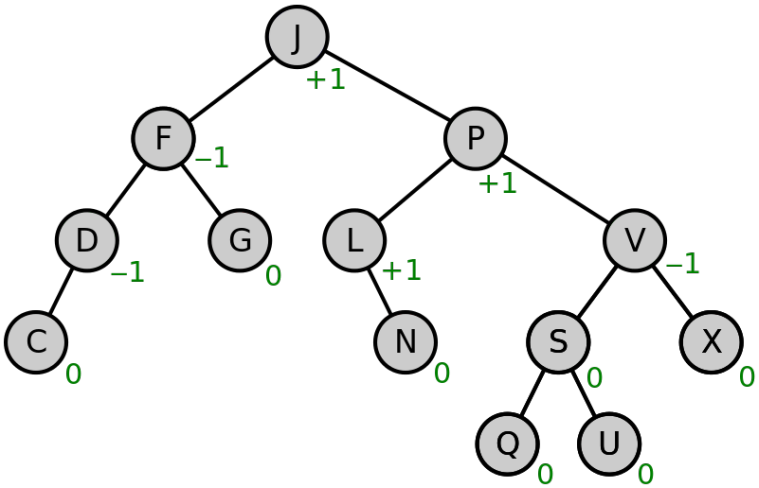
### *2.6.2 AVL (Árvore de Busca Balanceada)*

A Árvore AVL é uma forma especial de Árvore de Pesquisa Binária (BST) que se mantém automaticamente equilibrada. Isso significa que a diferença entre as alturas das subárvores esquerda e direita de qualquer nó, chamada de fator de equilíbrio, não pode exceder um. Em outras palavras, a AVL garante que a árvore permaneça balanceada, o que resulta em operações eficientes de busca, inserção e exclusão de elementos, tornando-a uma estrutura valiosa para implementações eficazes de árvores binárias. Na Figura 2 pode-se ver um exemplo de avl.

### *2.6.3 Heap de Fibonacci*

Um heap de Fibonacci é uma estrutura de dados eficiente usada em algoritmos de grafos, especialmente em operações que envolvem uniões e diminuições de chave. Ele é uma variação de um heap binomial que utiliza árvores de Fibonacci em vez de árvores binomiais. Cada nó no heap de Fibonacci mantém uma chave e um ponteiro para um dos seus filhos, facilitando a manipulação eficiente da estrutura. Uma característica distintiva é que os nós

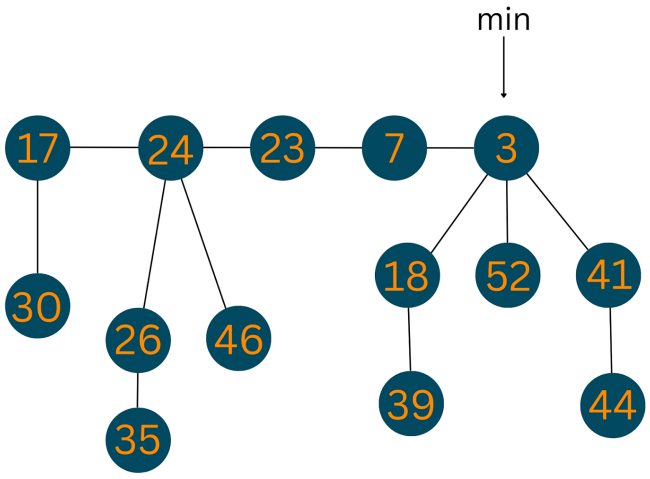
Figura 2 – Representação da Árvore AVL




Fonte: https://en.wikipedia.org/wiki/File:AVL-tree-wBalance_K.svg

podem ser removidos e reorganizados durante as operações, como extrair o mínimo ou realizar uma diminuição de chave, o que contribui para o desempenho otimizado em certos contextos algorítmicos. Na figura 3 pode-se ver um exemplo de Heap Fibonacci.

Figura 3 – Representação da uma Heap de Fibonacci




Fonte: https://favtutor.com/blogs/fibonacci-heap-algorithm-operations-cpp-java-python

### *2.6.4 Complexidade teórica*

Tabela 2 – Complexidade teórica

| Algoritmo | Complexidade Temporal | Complexidade Espacial |
|---|---|---|
| Árvore Binária | $O(n\log(n))$ | $O(n)$ |
| AVL | $O(n\log(n))$ | $O(n)$ |
| Heap Fibonacci | $O(n\log(n))$ | $O(n)$ |

# 3 METODOLOGIA

## 3.1 Problema da Mochila Fracionária

Nesta seção, estão dispostos os passos metodológicos utilizados desde a geração de entradas (Subseção 3.1.1), implementação e execução dos algoritmos (Subseção 3.1.2) e também coleta e processamento de dados (Subseção 3.1.3).

### *3.1.1 Geração de entradas*

De acordo com os testes preliminares realizados, foram selecionados os tamanhos $(100, 1000, 10000, 100000, 1000000)$ para realizar os experimentos. Dado um tamanho $n$, para gerar uma entrada para o algoritmo da mochila, é preenchido o vetor com valores de 1 até $n$ e após isso, os valores são embaralhados. Uma entrada aleatória, aliada com várias execuções permitem eliminar a variabilidade dos melhores e piores casos, por isso, permite encontrar tempos e consumos de memória que refletem um caso médio do algoritmo.

### *3.1.2 Implementação e execução*

Foram implementados os algoritmos da Mochila Fracionária por Ordenação (Listagem 3.1.1), utilizando a Mediana das Medianas (Listagem 3.1.2) e também partições sucessivas utilizando a média (Listagem 3.1.3). Cada um dos algoritmos mencionados é testado para uma entrada gerada, com isso, pode ser comparado os tempos de execução dos algoritmos.

```python
def sorted_fractional_knapsack(items, W):
    v1 = 0
    w1 = 0

    items = sorted(items, key=lambda item: item.ratio, reverse=True)

    for item in items:
        if item.peso + w1 <= W:
            w1 += item.peso
            v1 += item.valor
        else:
            v1 += item.valor * ((W - w1) / item.peso)
            break

    return v1
```

Listing 3.1.1 – Mochila Fracionária por Ordenação

```python
def median_of_medians_fractional_knapsack(items, W):
    if W == 0 or len(items)==0:
        return 0

    if len(items) == 1 and items[0].peso > W:
        return W * items[0].ratio

    k = len(items) // 2

    mid = select_bfprt(items, k)
    items_right = items[mid:]

    w1 = 0
    v1 = 0

    for item in items_right:
        w1 += item.peso
        v1 += item.valor

    if(w1 > W):
        return median_of_medians_fractional_knapsack(items_right, W)

    items_left = items[:mid]
    return v1 + median_of_medians_fractional_knapsack(items_left, W -
    ↪ w1)
```

Listing 3.1.2 – Mochila Fracionária utilizando a Mediana das Medianas

```python
def mean_partition_fractional_knapsack(items, W):
    if W == 0 or len(items) == 0:
        return 0

    if len(items) == 1 and items[0].peso > W:
        return W * items[0].ratio


    mid = partition_by_mean(items, 0, len(items) - 1)
    items_right = items[: mid]

    w1 = 0
    v1 = 0

    for item in items_right:
        w1 += item.peso
        v1 += item.valor

    if(w1 > W):
        return mean_partition_fractional_knapsack(items_right, W)

    items_left = items[mid : ]
    return v1 + mean_partition_fractional_knapsack(items_left, W - w1)
```

Listing 3.1.3 – Mochila Fracionária por partição utilizando a média

### *3.1.3 Coleta e processamento dos dados*

São feitos vários ciclos completos de geração de instâncias e execução para cada um dos algoritmos, as métricas coletadas são salvas por meio de arquivos *csv*, onde são salvos dados de cada uma das execuções, tais como o tempo de processador, uso de memória e tamanho da entrada. Com os dados processados, uma tabela é montada com as colunas referentes ao tamanho da entrada $n$, complexidade temporal teórica $c$, tempo médio de execução $t_{med}$, tempo máximo de execução $t_{max}$, tempo mínimo de execução $t_{min}$, uso médio de memória $m_{med}$ e uma razão $r$ entre a complexidade teórica e o tempo médio de execução.

Por fim, por meio da ferramenta Jupyter Notebook e da biblioteca Pandas do Python, os dados são processados e formatados. Após isso, com o auxílio da biblioteca Matplotlib, são gerados os gráficos que estão dispostos na Seção 4.2, correspondente aos resultados das instâncias de execução deste problema.

## 3.2 Algoritmo de Prim

Nesta seção, exploraremos as implementações dos algoritmos de Prim, empregando três distintas estruturas de dados: árvore binária, AVL e heap Fibonacci. Cada implementação será acompanhada de uma ênfase em funções cruciais e dificuldades relacionadas. Essas estruturas de dados desempenham um papel fundamental na eficiência do algoritmo de Prim, cujo objetivo é encontrar a Árvore Geradora Mínima (AGM) em um grafo ponderado e conectado. A escolha da estrutura de dados impacta diretamente no desempenho e na complexidade do algoritmo, oferecendo uma oportunidade para análises mais detalhadas e comparações significativas

### *3.2.1 Árvore Binária*

Inicialmente, foi desenvolvida a implementação de uma Árvore Binária, que serviria como a estrutura de dados fundamental para o subsequente Algoritmo de Prim. A implementação da Árvore Binária é uma estrutura de dados que permite armazenar elementos em uma estrutura hierárquica de forma que cada elemento tenha, no máximo, dois filhos: um à esquerda e outro à direita. A estrutura é composta por duas classes principais: **BinaryTree** e **Node**.

#### *3.2.1.1 BinaryTree*

Esta classe representa a árvore binária e possui um atributo *root* que aponta para o nó raiz da árvore. O método insert permite inserir um novo elemento na árvore, mantendo a propriedade de que valores menores são colocados à esquerda e valores maiores à direita.

O método extract_min extrai e remove o nó com o menor valor da árvore, útil em algoritmos como o de Prim. Os métodos _find_min e _delete são auxiliares para encontrar o nó mínimo e deletar um nó específico da árvore, mantendo a estrutura da árvore binária. Abaixo é mostrado parte da estrutura utilizada para a classe **BinaryTree**.

```
1 class BinaryTree:
2     def __init__(self):
3         self.root = None
4 
5     def insert(self, key):
6         if not self.root:
7             self.root = Node(key)
```

```
        else:
            self._insert_rec(self.root, key)

    def _insert_rec(self, node, key):
        # implementao da funo _insert_rec

    def extract_min(self):
        # implementao da funo extract_min

    def _find_min(self, node):
        # Implementao da funo _find_min

    def _delete(self, node, key):
        # Implementao da funo _delete
```

#### *3.2.1.2 Node*

Esta classe representa os nós individuais da árvore binária, contendo um valor (key), um ponteiro para o filho esquerdo (left) e um ponteiro para o filho direito (right).

Abaixo é mostrado parte da estrutura utilizada para a classe **Node**.

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
```

#### *3.2.1.3 Prim_AB*

O Algoritmo de Prim é um algoritmo para encontrar a Árvore Geradora Mínima (AGM) de um grafo ponderado, ou seja, uma subárvore que inclui todos os vértices do grafo original com o menor custo possível. A implementação utiliza a estrutura de árvore binária para manter as chaves (pesos) dos vértices. A estrutura implementada é formada por uma função principal **Prim_AB**.

Esta função recebe um grafo representado como um dicionário, onde as chaves são vértices e os valores são listas de tuplas representando as arestas e seus pesos. Inicializa-se os conjuntos para armazenar as arestas da AGM X, os vértices não visitados Q e um dicionário para rastrear os pais dos vértices na AGMpi.

Em seguida, cria-se uma árvore binária utilizando a estrutura implementada anteriormente, para manter as chaves (pesos) dos vértices, inicializando todas as chaves como infinito, exceto a chave do vértice raiz, que é definida como 0. O algoritmo itera enquanto houver vértices não visitados em Q. Em cada iteração, extrai o vértice com a menor chave da árvore binária, adiciona a aresta correspondente a X, atualiza o peso total da AGM e os valores em pi. O algoritmo seleciona a próxima chave mínima para adicionar à árvore binária, caso essa chave seja menor que a já existente.

Pode ser observado abaixo a implementação do **Algoritmo Prim** com utilização da **Árvore Binária**.

```
1  def Prim_AB(graph, root):
2      X = set()
3      Q = set(graph.keys())
4      pi = {u: None for u in graph}
5
6      # Inicialize a rvore binria com chaves infinitas
7      key_binary_tree = BinaryTree()
8      for u in graph:
9          key_binary_tree.insert((float('inf'), u))
10
11 key_binary_tree.insert((0, root)) # Defina a chave do vrtice raiz como 0
12
13 total_weight = 0 # Varivel para armazenar o peso total da AGM
14 weights = {} # Dicionrio para armazenar os pesos das arestas selecionadas
15
16 while Q:
17     min_node = key_binary_tree.extract_min()
18     if min_node is None:
19         break
20
```

```
    u = min_node.key[1]

    if u in Q:
        Q.remove(u)

        if pi[u] is not None:
            X.add((u, pi[u]))
            total_weight += min_node.key[0]
            weights[(u, pi[u])] = min_node.key[0] # Armazene o peso da aresta

        for v, weight in graph[u]:
            if v in Q and weight < key_binary_tree.root.key[0]:
                key_binary_tree.insert((weight, v))
                pi[v] = u

return X, total_weight, weights
```

#### *3.2.1.4 Dificuldades relacionadas*

A implementação da Árvore Binária como parte do Algoritmo de Prim apresentou desafios e complexidades específicos, que são cruciais para entender a estrutura e funcionamento dessa parte do código.

Inicialmente, compreender como a Árvore Binária se encaixaria no contexto do Algoritmo de Prim foi desafiador. A conexão entre a estrutura da árvore e a lógica do algoritmo não era imediatamente intuitiva, exigindo um estudo cuidadoso da teoria por trás do Algoritmo de Prim.

Outro desafio foi determinar quais métodos seriam essenciais na classe da Árvore Binária para suportar as operações específicas do Algoritmo de Prim foi um passo crítico. Isso incluiu a necessidade de métodos para inserção, extração do mínimo e exclusão de nós, bem como a capacidade de localizar o mínimo na árvore.

### *3.2.2 AVL (Árvore de Busca Balanceada)*

Foi desenvolvida a implementação da Árvore AVL, em que posteriormente seria utilizada como estrutura de dados pelo Algoritmo de Prim. A Árvore AVL é uma árvore binária balanceada que mantém seu equilíbrio automaticamente após inserções e exclusões de elementos. Cada nó da árvore possui informações, incluindo sua chave, valor associado e a altura da subárvore enraizada naquele nó. A implementação é composta por duas principais classes, **AVLNode** e **AVLTree**.

#### *3.2.2.1 AVLNode*

Esta classe representa um nó individual na Árvore AVL. Cada nó contém uma chave, um valor, uma altura e apontadores para os filhos esquerdo e direito. A altura é usada para determinar o equilíbrio da árvore. Segue abaixo parte da estrutura utilizada para a classe **AVLNode**.

```
1 class AVLNode:
2     def __init__(self, key, value):
3         self.key = key
4         self.value = value
5         self.height = 1
6         self.left = None
7         self.right = None
```

#### *3.2.2.2 AVLTree*

Esta classe representa a Árvore AVL em si. Ela possui métodos para realizar operações como inserção, exclusão, verificação da existência de chaves e outras operações associadas a uma árvore binária balanceada. As operações de inserção e exclusão incluem ajustes de altura e rotações de nós para manter o equilíbrio da árvore. Segue abaixo parte da estrutura utilizada para a classe **AVLTree**.

```
1 class AVLTree:
2     def __init__(self):
3         self.root = None
```

```
4
5     def height(self, node):
6         if node is None:
7             return 0
8         return node.height
9
10    def balance_factor(self, node):
11        if node is None:
12            return 0
13        return self.height(node.left) - self.height(node.right)
14
15    def insert(self, key, value):
16        self.root = self._insert(self.root, key, value)
17
18    def _insert(self, node, key, value):
19        # Implementao da funo _insert
20
21    def delete(self, key):
22        self.root = self._delete(self.root, key)
23
24    def _delete(self, node, key):
25        # Implementao da funo _delete
26
27    def contains(self, key):
28        return self._contains(self.root, key)
29
30    def _contains(self, node, key):
31        # Implementao da funo _contains
32
33    def _min_value_node(self, node):
34        # Implementao da funo _min_value_node
35
36    def _rotate_left(self, node):
37        # Implementao da funo _rotate_left
```

```
38
39      def _rotate_right(self, node):
40          # Implementao da funo _rotate_right
41
42      def __len__(self):
43          return self._count_nodes(self.root)
44
45      def _count_nodes(self, node):
46          # Implementao da funo _count_nodes
```

#### *3.2.2.3 Dificuldades relacionadas*

Durante o processo de implementação da Árvore AVL para ser utilizada no Algoritmo de Prim, várias dificuldades surgiram, exigindo um entendimento aprofundado da estrutura de dados e sua adaptação para o problema específico.

Uma das principais dificuldades foi compreender como incorporar a Árvore AVL no Algoritmo de Prim. Isso envolveu a identificação dos métodos necessários na Árvore AVL para suportar as operações específicas do Algoritmo de Prim, como a inserção, exclusão e busca de vértices com seus respectivos pesos. Outro ponto importante foi compreender o funcionamento das rotações e garantir que elas fossem aplicadas corretamente, especialmente durante as operações de inserção e exclusão, exigindo um esforço adicional. Contudo, a realização de testes e a depuração foram etapas críticas para garantir que a implementação funcionasse conforme o esperado e produzisse resultados corretos e eficientes.

### *3.2.3 Heap Fibonacci*

A implementação do Heap Fibonacci é uma peça crucial para o funcionamento eficiente do Algoritmo de Prim. O Heap Fibonacci é uma estrutura de dados versátil que permite operações de inserção, extração do mínimo e consolidação eficiente, tornando-o uma escolha ideal para o algoritmo.

#### *3.2.3.1 FibonacciHeapNode*

A classe FibonacciHeapNode representa um nó individual no Heap Fibonacci. Cada nó contém informações essenciais, incluindo chave, vértice associado, grau do nó, marcação, apontadores para os filhos, e apontadores para os nós vizinhos na lista duplamente encadeada. Aqui está parte da estrutura utilizada para a classe:

```
class FibonacciHeapNode:
    def __init__(self, key, vertex):
        self.key = key
        self.vertex = vertex
        self.degree = 0
        self.marked = False
        self.parent = None
        self.child = None
        self.next = self
        self.prev = self
```

#### *3.2.3.2 FibonacciHeap*

A classe **FibonacciHeap** gerencia a estrutura do Heap Fibonacci. Possui métodos para inserção, extração do mínimo e consolidação da heap. A consolidação é fundamental para garantir a eficiência do algoritmo. Aqui está parte da estrutura utilizada para a classe:

```
class FibonacciHeap:
    def __init__(self):
        self.min_node = None
        self.node_count = 0

    def insert(self, key, vertex):
        # Implementao da insero no Heap Fibonacci
        # ...

    def extract_min(self):
        # Implementao da extrao do mnimo no Heap Fibonacci
```

```
12          # ...
13
14      def _link(self, node1, node2):
15          # Implementao do link entre dois ns no Heap Fibonacci
16          # ...
17
18      def _consolidate(self):
19          # Implementao da consolidao no Heap Fibonacci
20          # ...
```

#### *3.2.3.3 Dificuldades relacionadas*

Durante a implementação, foram enfrentados desafios ao integrar o Heap Fibonacci no Algoritmo de Prim. Foi necessário compreender como as operações no Heap Fibonacci, como a consolidação, se encaixam no contexto específico do algoritmo. Testes extensivos foram essenciais para garantir que a implementação funcionasse corretamente, produzindo resultados precisos e eficientes.

O uso do Heap Fibonacci no Algoritmo de Prim oferece uma abordagem otimizada para encontrar a Árvore Geradora Mínima (AGM) em grafos ponderados, proporcionando eficiência em termos de complexidade temporal e permitindo a manipulação eficaz de arestas com diferentes pesos.

# 4 RESULTADOS

## 4.1 Problema de Programação Hiperbólica (PPH)

Tabela 3 – Resultados PPH

| Tamanho da Entrada ($n$) | Média Tempo de Execução (seg) | Média Uso de Memória (bytes) |
|---|---|---|
| 100 | 0.00052 | 0.47 |
| 1000 | 0.00375 | 2008.73 |
| 10000 | 0.04506 | 14596.35 |
| 100000 | 0.54713 | 29687.60 |
| 500000 | 2.78210 | 30116.27 |

Na Tabela 3, apresentamos os resultados obtidos com o Problema de Programação Hiperbólica. Com um tempo de execução médio de 0,00052 segundos para 100 itens. No entanto, à medida que o número de itens aumenta, o tempo de execução cresce consideravelmente, atingindo 2,78210 segundos para 500000 de itens. O consumo médio de memória também aumenta substancialmente, de 0,47 KB para 30116.27 bytes.

## 4.2 Problema da Mochila Fracionária

Esta seção do trabalho se propõe a mostrar o resultados obtidos após a experimentação com as três soluções proposta para o Problema da Mochila Fracionária. Estas são, a Mochila Fracionária por Ordenação (Subseção 4.2.1), por partições sucessivas utilizando a Mediana das Medianas (Subseção 4.2.2) e por partições sucessivas utilizando a média (Subseção 4.2.3). Por fim, são apresentados os resultados comparativos (Subseção 4.2.4) e também a análise dos gráficos (Subseção 4.2.5).

### *4.2.1 Mochila Fracionária por Ordenação*

Na Tabela 4, apresentamos os resultados obtidos com o método de ordenação. Observamos que este método é altamente eficiente para problemas de tamanho reduzido, com um tempo de execução médio de apenas 0,000058 segundos para 100 itens. No entanto, à medida que o número de itens aumenta, o tempo de execução cresce consideravelmente, atingindo 0,635050 segundos para 1.000.000 de itens. O consumo médio de memória também aumenta substancialmente, de 1,178 KB para 24 MB. Portanto, embora seja uma escolha viável para problemas pequenos, pode não ser escalável para instâncias maiores do problema.

Tabela 4 – Resultados por ordenação

| $n$ | $c$ | $t_{med}$ | $t_{max}$ | $t_{min}$ | $m_{med}$ | $r$ |
|---|---|---|---|---|---|---|
| 100 | 664.38 | 0,000058 | 0,000116 | 0,000039 | 1178,46 | 11359754,13 |
| 1000 | 9965.78 | 0,000409 | 0,000644 | 0,000267 | 25104,31 | 24375226,61 |
| 10000 | 132877.12 | 0,004280 | 0,017579 | 0,003080 | 248251,54 | 31046382,83 |
| 100000 | 1660964.04 | 0,047565 | 0,075554 | 0,043883 | 2497395,54 | 34919778,50 |
| 1000000 | 19931568.56 | 0,635050 | 0,937983 | 0,558995 | 24873825,85 | 31385850,05 |

### *4.2.2 Mochila Fracionária utilizando a Mediana das Medianas*

Na Tabela 5, apresentamos os resultados do método de partições sucessivas com a mediana das medianas. Este método demonstra um desempenho menos eficiente em comparação com o método de ordenação, mesmo para instâncias pequenas. Para 100 itens, o tempo de execução médio é de 0,003301 segundos, e o uso de memória é de 2,507 KB. À medida que o tamanho do problema aumenta, o tempo de execução e o uso de memória crescem de forma acentuada, atingindo 78 segundos e 8 MB, respectivamente, para 1.000.000 de itens. Esses resultados indicam que este método pode não ser adequado para problemas de grande escala.

Tabela 5 – Resultados usando a medina das medianas

| $n$ | $c$ | $t_{med}$ | $t_{max}$ | $t_{min}$ | $m_{med}$ | $r$ |
|---|---|---|---|---|---|---|
| 100 | 100 | 0,003301 | 0,006405 | 0,002125 | 2507,96 | 30291,01 |
| 1000 | 1000 | 0,052523 | 0,253658 | 0,033775 | 9440,65 | 19039,17 |
| 10000 | 10000 | 0,677082 | 1,152225 | 0,551563 | 82521,85 | 14769,25 |
| 100000 | 100000 | 7,598661 | 8,587149 | 5,401850 | 809005,96 | 13160,21 |
| 1000000 | 1000000 | 78,026148 | 80,935381 | 75,118907 | 8025082,27 | 12816,21 |

### *4.2.3 Mochila Fracionária por partições sucessivas utilizando a média*

Na Tabela 6, apresentamos os resultados do método de partições sucessivas com a média. Este método demonstra um desempenho intermediário em relação aos outros dois métodos avaliados. Para 100 itens, o tempo de execução médio é de 0,000145 segundos, e o uso de memória é de 1,405 KB. À medida que o tamanho do problema aumenta, tanto o tempo de execução quanto o uso de memória aumentam, mas o crescimento não é tão acentuado quanto no método de mediana das medianas. Para 1.000.000 de itens, o tempo de execução é de 0,839843 segundos, e o uso de memória é de 8,697 MB. Embora este método seja uma escolha intermediária em termos de eficiência, ele pode ser considerado uma opção viável para problemas de tamanho moderado.

Tabela 6 – Resultados por partições sucessivas pela média

| $n$ | $c$ | $t_{med}$ | $t_{max}$ | $t_{min}$ | $m_{med}$ | $r$ |
|---|---|---|---|---|---|---|
| 100 | 100 | 0,000145 | 0,001477 | 0,000082 | 1405,55 | 690070,91 |
| 1000 | 1000 | 0,000894 | 0,001712 | 0,000567 | 9200,00 | 1118380,70 |
| 10000 | 10000 | 0,008778 | 0,048281 | 0,006371 | 87800,00 | 1139197,51 |
| 100000 | 100000 | 0,080350 | 0,133142 | 0,070029 | 824640,00 | 1244561,21 |
| 1000000 | 1000000 | 0,839843 | 1,079476 | 0,724088 | 8697640,00 | 1190698,86 |

### *4.2.4 Comparativo dos resultados*

Os resultados destacam a importância de escolher o método apropriado para resolver o problema da mochila fracionária, levando em consideração o tamanho da instância do problema. O método de ordenação é eficaz para problemas pequenos, mas não escalável para instâncias maiores. Os métodos de partições sucessivas, tanto com a mediana das medianas quanto com a média, mostram desempenho inferior em relação ao método de ordenação. Portanto, a escolha do método deve ser cuidadosamente ponderada de acordo com as características do problema em questão.

### *4.2.5 Analise gráfica dos testes*

Nota-se na Figura 4 que dois dos métodos utilizados não se distanciam muito do eixo das abscisas somente o, método “Mediana das medianas” assume a característica linear. Como era de se esperar. É importante ressaltar que os outros métodos não estão constantes, só não variam muito devido a escala do gráfico.

Já na análise de memória da Figura 5, nota-se que todos seguem uma característica linear. Somente o Ordenado que vai gastar mais memória que os outros dois a medida que o tamanho da instância aumenta.

Figura 4 – Gráfico do tempo de execução para a Mochila Fracionária

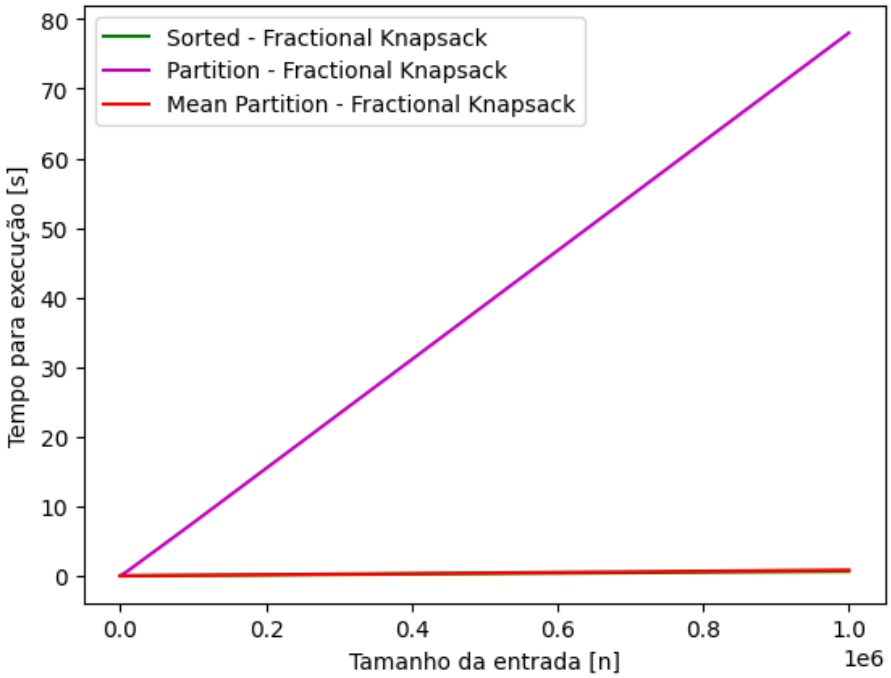
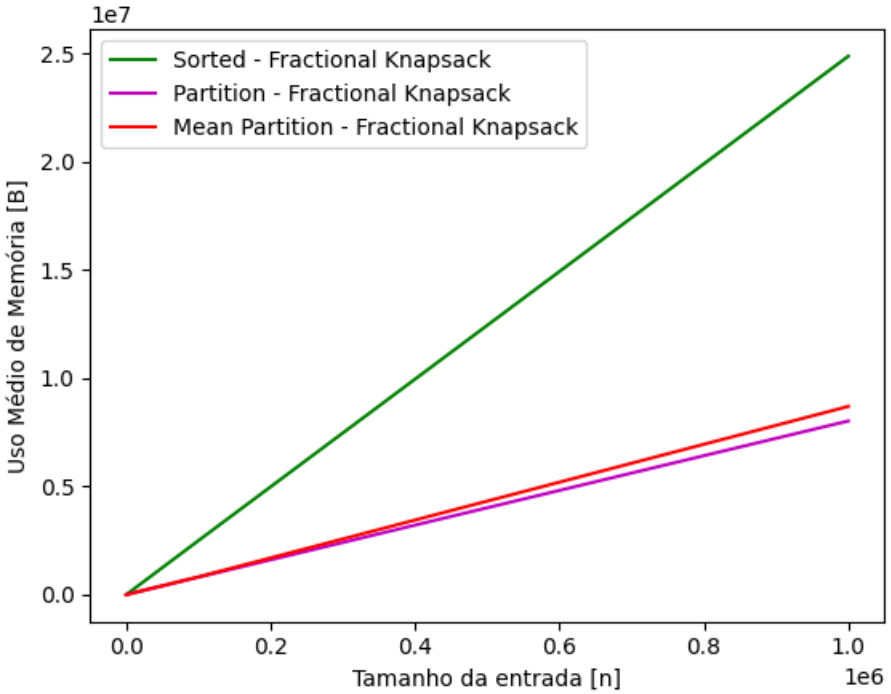



Figura 5 – Gráfico do consumo de memória da Mochila Fracionária




## 4.3 Algoritmo de Prim

Os experimentos foram meticulosamente planejados para avaliar o desempenho do Algoritmo de Prim em três distintas estruturas de dados: Árvore Binária desbalanceada, Árvore AVL e Heap de Fibonacci. A abrangência das instâncias de teste incluiu três grafos diversos

(ALUE, ALUT e DMXA). Cada experimento foi repetido não apenas 10 vezes, mas também ajustado em alguns casos para garantir um tempo mínimo de execução de 5 segundos. Essa abordagem visa assegurar médias confiáveis de tempo de execução e consumo de memória.A condução dos testes ocorreu em um ambiente controlado, utilizando uma máquina equipada com um processador Intel Core i5-1145G7 e 8 GB de RAM.

### *4.3.1 Ávore binária*

Na Tabela 7 pode-se ver os resultados de tempo em segundos e memória em bytes para testes com árvore binária relacionada a Prim.

Tabela 7 – Resultado árvore binária

| Nome do arquivo | Tempo médio | Memória média | Complexidade Teórica | C/T |
|---|---|---|---|---|
| alue2087 | 0.02224 | 116016.0 | 3849.956553 | 173109.56 |
| alue2105 | 0.02322 | 3128547.9 | 3765.358993 | 162160.16 |
| alue3146 | 0.07678 | 4222505.4 | 12906.485202 | 168096.97 |
| alue5067 | 0.07421 | 4271148.1 | 12499.754511 | 168437.60 |
| alue5345 | 0.13042 | 4982436.1 | 19236.079573 | 147493.33 |
| alue5623 | 0.14492 | 4686334.5 | 16325.044027 | 112648.66 |
| alue5901 | 0.44637 | 8146713.2 | 46891.344704 | 105050.39 |
| alue6179 | 0.09056 | 4274462.0 | 11896.036872 | 131360.83 |
| alue6457 | 0.11254 | 4519286.3 | 14134.020318 | 125591.08 |
| alue6735 | 0.12297 | 4669180.8 | 14889.327391 | 121080.97 |
| alue6951 | 0.08148 | 4117791.0 | 9721.933706 | 119316.81 |
| alut0787 | 0.03304 | 116016.0 | 3554.771268 | 107589.93 |
| alut0805 | 0.01714 | 3083603.7 | 2883.487904 | 168231.50 |
| alut1181 | 0.09313 | 4165099.5 | 10591.852934 | 113731.91 |
| alut2010 | 0.24713 | 5688780.8 | 23107.391060 | 93502.98 |
| alut2288 | 0.46307 | 7154939.3 | 35895.498094 | 77516.35 |
| alut2764 | 0.00982 | 2820403.8 | 1001.444143 | 101980.06 |
| dmxa0296 | 0.00462 | 44168.0 | 551.593930 | 119392.63 |
| dmxa0368 | 0.06978 | 3565178.7 | 6789.095415 | 97292.85 |
| dmxa0454 | 0.04236 | 3564373.2 | 6036.865235 | 142513.34 |
| dmxa0628 | 0.00334 | 2663023.7 | 376.512853 | 112728.40 |
| dmxa0734 | 0.01014 | 2959649.7 | 1870.663469 | 184483.58 |
| dmxa0848 | 0.00971 | 2863686.6 | 1346.352172 | 138656.25 |
| dmxa0903 | 0.01474 | 2942494.2 | 1770.053193 | 120085.02 |
| dmxa1010 | 0.12607 | 4688250.2 | 14339.637650 | 113743.46 |
| dmxa1109 | 0.00788 | 2871826.9 | 869.605883 | 110356.08 |
| dmxa1200 | 0.02033 | 3110617.0 | 2222.597858 | 109326.01 |

A análise dos resultados na Tabela 7 revela nuances importantes sobre o desempenho do Algoritmo de Prim em conjunto com a Árvore Binária. Os tempos médios de execução e o consumo de memória exibem variações notáveis entre diferentes instâncias de grafos, indicando uma sensibilidade significativa às características específicas de cada grafo. O arquivo alue7065 se destaca com um tempo de execução notavelmente mais elevado, pois tem um grafo mais

complexo. Além disso, a variação nos consumos de memória, notadamente em instâncias como **alue5901** e alut2010, ressalta a necessidade de considerar não apenas a eficiência média, mas também a variabilidade em diferentes cenários.

#### *4.3.1.1 Árvore AVL*

Na Tabela 8 pode-se ver os resultados de tempo em segundos e memória em bytes para testes com àrvore avl relacionada a Prim.

Tabela 8 – Resultado árvore AVL

| Nome do arquivo | Tempo médio | Memória média | Complexidade Teórica | C/T |
|---|---|---|---|---|
| alue2087 | 12.72788 | 71476.8 | 3849.956553 | 836.04 |
| alue2105 | 14.46124 | 3055517.3 | 3765.358993 | 687.36 |
| alue3146 | 146.51563 | 4303159.9 | 12906.485202 | 204.36 |
| alue5067 | 137.83600 | 4329625.7 | 12499.754511 | 201.90 |
| alue5345 | 268.57094 | 5099215.0 | 19236.079573 | 149.78 |
| alue5623 | 103.57566 | 4744302.2 | 16325.044027 | 173.84 |
| alue5901 | 671.04280 | 8212647.9 | 46891.344704 | 69.88 |
| alue6179 | 54.63335 | 4366031.8 | 11896.036872 | 217.74 |
| alue6457 | 70.72272 | 4673538.2 | 14134.020318 | 199.85 |
| alue6735 | 87.22341 | 4785902.3 | 14889.327391 | 170.70 |
| alue6951 | 56.73688 | 4181302.8 | 9721.933706 | 171.35 |
| alut0787 | 3.89556 | 66772.8 | 3554.771268 | 912.52 |
| alut0805 | 3.02433 | 3045711.6 | 2883.487904 | 953.43 |
| alut1181 | 42.80601 | 4247917.2 | 10591.852934 | 247.44 |
| alut2010 | 187.76857 | 5879703.9 | 23107.391060 | 123.06 |
| alut2288 | 471.61470 | 7408031.8 | 35895.498094 | 76.11 |
| alut2764 | 1.31831 | 2903575.1 | 1001.444143 | 759.64 |
| dmxa0296 | 0.10922 | 14860.8 | 551.593930 | 5050.30 |
| dmxa0368 | 14.21790 | 3560562.1 | 6789.095415 | 477.50 |
| dmxa0454 | 12.50322 | 3538240.6 | 6036.865235 | 482.82 |
| dmxa0628 | 0.08469 | 2666797.9 | 376.512853 | 4445.78 |
| dmxa0734 | 1.57821 | 2867512.2 | 1870.663469 | 1185.31 |
| dmxa0848 | 0.77088 | 2813494.1 | 1346.352172 | 1746.51 |
| dmxa0903 | 1.20360 | 2870280.2 | 1770.053193 | 1470.63 |
| dmxa1010 | 61.51632 | 4750118.8 | 14339.637650 | 233.10 |
| dmxa1109 | 0.42857 | 2895463.0 | 869.605883 | 2029.09 |
| dmxa1200 | 1.95279 | 3064313.8 | 2222.597858 | 1138.17 |

A Tabela 8 apresenta os resultados de tempo médio de execução e consumo médio de memória para testes utilizando a Árvore AVL no contexto do Algoritmo de Prim. Ao analisar esses dados, é possível observar variações significativas nos tempos de execução e nos consumos de memória entre diferentes instâncias de grafos. Notavelmente, o arquivo alue5901 destaca-se com um tempo médio de execução consideravelmente alto e um consumo de memória significativo, sugerindo um grafo que pode apresentar desafios específicos para a estrutura da Árvore AVL. Por outro lado, algumas instâncias, como dmxa0296, demonstram tempos de

execução e consumos de memória mais modestos.

#### *4.3.1.2 Heap Fibonacci*

Na Tabela 9 pode-se ver os resultados de tempo em segundos e memória em bytes para testes com heap fibonacci relacionada a Prim.

Tabela 9 – Resultado árvore binária

| Nome do arquivo | Memória média | Tempo médio | Complexidade Teórica | C/T |
|---|---|---|---|---|
| alut2288 | 7056576.2 | 3.934794 | 35895.498094 | 9122.59 |
| alut0787 | 712167.5 | 0.475000 | 3554.771268 | 7483.72 |
| alut0805 | 582157.8 | 0.393659 | 2883.487904 | 7324.83 |
| alut2764 | 227403.7 | 0.165151 | 1001.444143 | 6063.81 |
| alut1181 | 2303122.1 | 1.297900 | 10591.852934 | 8160.76 |
| alut2010 | 4354144.3 | 2.629490 | 23107.391060 | 8787.78 |
| alue6179 | 2070858.6 | 1.395533 | 11896.036872 | 8524.37 |
| alue3146 | 2253300.7 | 1.527768 | 12906.485202 | 8447.94 |
| alue2087 | 668222.5 | 0.514206 | 3849.956553 | 7487.18 |
| alue5067 | 2164353.9 | 1.460431 | 12499.754511 | 8558.95 |
| alue6457 | 2326071.2 | 1.668003 | 14134.020318 | 8473.62 |
| alue2105 | 654019.4 | 0.527386 | 3765.358993 | 7139.67 |
| alue5345 | 2922348.0 | 2.249644 | 19236.079573 | 8550.72 |
| alue6735 | 2467952.0 | 1.701280 | 14889.327391 | 8751.84 |
| alue7065 | 21576995.5 | 15.055321 | 154298.722264 | 10248.78 |
| alue5623 | 2584715.4 | 1.911023 | 16325.044027 | 8542.57 |
| alue5901 | 7860154.6 | 4.913895 | 46891.344704 | 9542.60 |
| alue6951 | 1837819.8 | 1.170626 | 9721.933706 | 8304.90 |
| dmxa0734 | 390483.9 | 0.277819 | 1870.663469 | 6733.39 |
| dmxa0454 | 1209445.0 | 0.758394 | 6036.865235 | 7960.06 |
| dmxa1109 | 212163.2 | 0.139205 | 869.605883 | 6246.94 |
| dmxa0628 | 133457.5 | 0.063734 | 376.512853 | 5907.58 |
| dmxa1010 | 2559634.9 | 1.628949 | 14339.637650 | 8803.00 |
| dmxa0848 | 299207.9 | 0.198288 | 1346.352172 | 6789.89 |
| dmxa0903 | 362710.5 | 0.253363 | 1770.053193 | 6986.23 |
| dmxa1200 | 487639.5 | 0.334420 | 2222.597858 | 6646.12 |
| dmxa0296 | 155347.7 | 0.091546 | 551.593930 | 6025.31 |
| dmxa0368 | 1325118.6 | 0.867685 | 6789.095415 | 7824.38 |

Na Tabela 9, são apresentados os resultados de tempo médio de execução e consumo médio de memória para os testes utilizando a estrutura de dados Heap Fibonacci no contexto do Algoritmo de Prim. Analisando esses destaca-se que o arquivo **alue5901** apresenta um tempo médio de execução e um consumo de memória notavelmente elevados, indicando que essa instância específica pode apresentar desafios específicos para a eficiência da Heap Fibonacci. Em contraste, algumas instâncias, como **dmxa0628** e **dmxa1109**, demonstram tempos de execução e consumos de memória mais modestos.

# 5 CONCLUSÕES

## 5.1 Execução do Problema de Programação Hiperbólica (PPH)

O Problema da Programação Hiperbólica (PPH) emerge como uma fascinante área de pesquisa no âmbito da otimização não linear. Sua aplicabilidade transcende diversas disciplinas, incluindo engenharia, economia, física e ciência da computação. A complexidade inerente ao PPH, fundamentada em sua não convexidade, introduz desafios significativos na busca por soluções ótimas. A presença de curvas hiperbólicas na função objetivo adiciona uma camada de complexidade, criando um cenário no qual múltiplos mínimos locais podem coexistir.

O cerne do PPH reside na busca por um conjunto de valores para as variáveis de decisão $x$ que não apenas minimizem a função objetivo $f(x)$ mas também satisfaçam um conjunto específico de restrições. Estas soluções ótimas, ao serem encontradas, fornecem insights valiosos e podem ser aplicadas para otimizar processos e tomadas de decisão.

A função objetivo do PPH, expressa como uma razão ponderada, reflete a delicada balança entre diferentes elementos. A maximização do conjunto $S^*$ nesse contexto, caracterizado como aquele que maximiza essa razão, torna-se uma tarefa de grande relevância. No entanto, a complexidade linear inerente à maximização do conjunto $S^*$ ressalta a intrincada natureza computacional do PPH.

### *5.1.1 Algoritmos de Ordenação e sua Relação com o PPH*

A relação inovadora entre o PPH e algoritmos de ordenação destaca a aplicação criativa de conceitos e observações sobre otimização não linear. A utilização do lema do PPH como fundamento para o projeto de algoritmos eficientes de ordenação é um testemunho da versatilidade desses princípios.

A diversidade de complexidades apresentadas pelos algoritmos de ordenação é notável. Algoritmos como Bubble Sort, Insertion Sort e Selection Sort, com complexidade $O(n^2)$ no pior caso, contrastam com abordagens mais eficientes como Merge Sort e Quick Sort, operando em $O(n \log n)$. O Heap Sort, com sua complexidade $O(n \log n)$ em todos os casos, destaca-se como uma ferramenta versátil.

A aplicação prática desses algoritmos em experimentos computacionais oferece uma visão tangível de seu desempenho relativo em diferentes cenários. A estrutura do código permite uma análise detalhada dos tempos de execução, proporcionando uma compreensão mais

profunda da eficiência desses algoritmos em conjuntos de dados de tamanhos variados.

### *5.1.2 Experimentos Computacionais e Insights*

A condução de experimentos computacionais comparativos se revela fundamental para extrair insights valiosos sobre o desempenho prático dos algoritmos de ordenação. A estrutura do código, que permite a medição precisa do tempo de execução em conjuntos de dados diversos, é uma ferramenta valiosa para pesquisadores e profissionais da área de algoritmos.

A análise combinada do PPH, algoritmos de ordenação e experimentos computacionais fornece uma perspectiva abrangente sobre otimização não linear e estratégias eficientes de ordenação. A interconexão desses tópicos destaca não apenas a teoria subjacente, mas também sua aplicação tangível em cenários do mundo real.

Em síntese, este estudo contribui para a compreensão aprofundada do PPH e sua relação com algoritmos de ordenação, enriquecendo o campo da otimização computacional e fornecendo um guia prático para a escolha e implementação de estratégias de ordenação eficazes.

## 5.2 Problema da Mochila Fracionária

Com base nos resultados obtidos na experimentação com as três abordagens propostas para o Problema da Mochila Fracionária, fica evidente a importância de escolher o método apropriado de acordo com o tamanho da instância do problema. Cada método apresenta vantagens e desvantagens específicas que devem ser consideradas ao abordar esse desafio.

O método de ordenação se destaca pela sua eficiência em problemas de tamanho reduzido, com um tempo de execução muito baixo e um consumo de memória aceitável. No entanto, ele não é escalável para instâncias maiores do problema, uma vez que o tempo de execução e o consumo de memória aumentam consideravelmente com o aumento do tamanho da mochila.

Por outro lado, o método de partições sucessivas utilizando a Mediana das Medianas e o método de partições sucessivas utilizando a média apresentam desempenhos inferiores em relação ao método de ordenação, mesmo para instâncias pequenas. Ambos os métodos mostram um crescimento acentuado no tempo de execução e no consumo de memória à medida que o problema se torna mais complexo. No entanto, o método de Mediana das Medianas demonstra um desempenho um pouco pior em comparação com o método da média.

Portanto, a escolha do método deve ser cuidadosamente ponderada de acordo com as características do problema em questão. Para problemas pequenos, o método de ordenação pode ser uma escolha viável devido à sua eficiência. No entanto, para problemas de maior escala, os métodos de partições sucessivas podem ser mais adequados, com uma ligeira vantagem para o método da média em relação à Mediana das Medianas.

Além disso, a análise gráfica dos testes confirma a tendência dos métodos em relação ao tempo de execução e ao consumo de memória. O método de ordenação permanece eficiente para instâncias menores, enquanto os métodos de partições sucessivas mostram um aumento linear no consumo de recursos, com o método de Mediana das Medianas sendo o menos eficiente nesse aspecto.

## 5.3 Algoritmo de Prim

Nos experimentos conduzidos, foram analisadas três estruturas de dados distintas - Árvore Binária desbalanceada, Árvore AVL e Heap de Fibonacci - no contexto do Algoritmo de Prim. Observaram-se diferentes desempenhos em termos de tempo de execução e consumo de memória.

A Árvore Binária desbalanceada revelou tempos de execução relativamente reduzidos, sugerindo eficiência para grafos de menor complexidade. Entretanto, a ausência de balanceamento pode tornar os tempos de execução menos previsíveis em grafos maiores e mais intrincados.

A Árvore AVL, por outro lado, apresentou tempos de execução mais estáveis, mas com uma tendência a consumir mais memória, especialmente em grafos de grande escala. A estrutura balanceada proporciona consistência no desempenho, contudo, com um custo adicional em termos de recursos.

Por fim, o Heap de Fibonacci demonstrou resultados diversos, destacando-se por tempos de execução extremamente rápidos em alguns casos, como **dmxa0628.stp**, enquanto revelava consumos elevados de memória em instâncias específicas, como alue5901.stp. A eficiência teórica da Heap de Fibonacci parece ser sensível às características individuais de cada grafo.

Resumidamente, a escolha entre essas estruturas de dados depende das peculiaridades do grafo em questão e das prioridades do aplicativo em termos de tempo de execução versus consumo de memória. A Árvore Binária desbalanceada pode ser uma opção adequada para

grafos menores, a Árvore AVL oferece um equilíbrio entre desempenho e consumo de memória, e a Heap de Fibonacci, embora eficiente em alguns casos, requer uma consideração cuidadosa devido à sua variabilidade nos resultados.

# REFERÊNCIAS

BLUM, M.; FLOYD, R. W.; PRATT, V. R.; RIVEST, R. L.; TARJAN, R. E. *et al.* Time bounds for selection. **J. Comput. Syst. Sci.**, v. 7, n. 4, p. 448–461, 1973. Disponível em: <http://people.csail.mit.edu/rivest/pubs/BFPRT73.pdf>.

CORMEN, T.; LEISERSON, C.; RIVEST, R.; STEIN, C. **Introduction To Algorithms**. MIT Press, 2001. (Mit Electrical Engineering and Computer Science). ISBN 9780262032933. Disponível em: <https://books.google.com.br/books?id=NLngYyWFl_YC>.