Tópicos Especiais em Computação II - 2023.2

Trabalho Prático 1

Vanessa Carvalho do Nascimento (Mat.: 471584)
Raniery Alves Vasconcelos (Mat.: 473532)
Francilândio Lima Serafim (Mat.: 472644)
Iago Magalhães de Mesquita (Mat.: 995573)
Sávio Araújo Gomes (Mat.: 474201)

Conteúdo

1 Problema de Programação Hiperbólica

A base utilizada para executar os algoritmos apresentados a seguir contém 4 instâncias para cada quantidade de pares. As quantidades de pares são: 100, 500, 1000, 5000, 10000, 50000, 100000, 500000 e 1000000. Caso o tempo de execução utilizando alguma instância fosse menor que 5 segundos, a execução era repetida até que o tempo total fosse superior aos 5 segundos. Para a análise de dados, foi utilizada as médias de tempo e espaço calculadas para cada tamanho.

1.1 Versão $O(N^2)$

A implementação do algoritmo do Problema de Programação Hiperbólica (PPH) com complexidade $O(N^2)$ consiste em iterar sobre os índices de 1 a n, sendo n o tamanho do vetor de pares candidatos $[(a_1, b_1), (a_2, b_2), \cdots, (a_n, b_n)]$, e em cada iteração obter o maior quociente $\frac{a_i}{b_i}$ presente no vetor quo $= [\frac{a_1}{b_1}, \frac{a_2}{b_2}, \cdots, \frac{a_n}{b_n}]$. Com o par obrigatório (a_0, b_0) calcula-se o primeiro valor de $R = \frac{a_0}{b_0}$. Durante uma iteração, ao encontrar o maior quociente $\frac{a_i}{b_i}$, verifica-se se $\frac{a_i}{b_i} > R$ atual. Caso seja verdade, i é inserido em S e R é atualizado. Além disso, quo$[i]$ passa a ser $-\infty$ e, assim, as iterações continuam. A Figura 1 mostra o consumo de tempo de acordo com o tamanho da instâncias, já a Figura 2 apresenta o código da implementação.

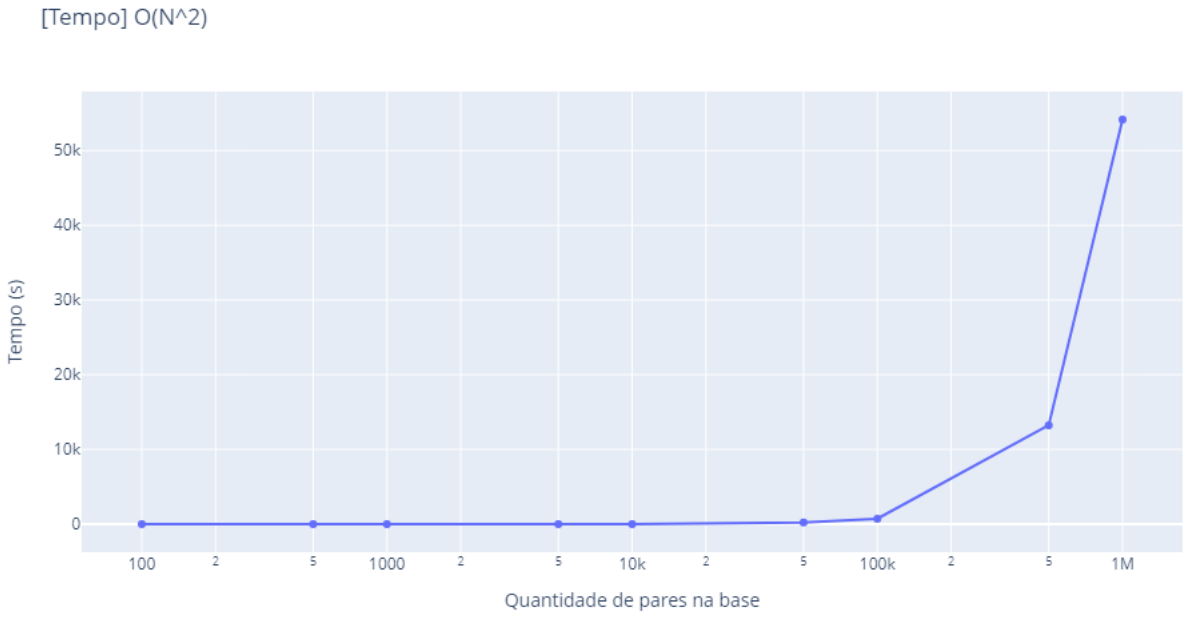


Figura 1: Tempo de execução para o algoritmo $O(N^2)$.

```python
def PPH(num):  #Versão O(N^2)
    tracemalloc.start() #início da medição de memória
    S = []  # índices das tuplas selecionadas
    num_s = [] # tuplas que possuem seus índices em S


    #xxxxxxxxxxxxxxxxxxxxxxx


    prim = num[0][0]        # numerador considerando apenas a0
    sec = num[0][1]         # denominador considerando apenas b0
    R_ = prim/sec           # valor de R considerando apenas (a0,b0)

    quo = []
    for e in num[1:]:
        if e[1]==0:    #caso em que o denominador é 0 não é possível fazer a divisão
            quo.append(inf)
        else:
            quo.append(e[0]/e[1])
    elem = num[1:]




    #XXXXXXXXXXXXXXXXXXX Inserções O(N^2) XXXXXXXXXXXXXXXXXXX
    for i in range(len(elem)):
#         print(f"Iteração {i} --- R_: {R_}")

        ind = np.argmax(quo)

        q = quo[ind]

        if q>R_:         # caso a razão ai/bi seja maior que R atual, S deve representar (ai,bi)
            S.append(ind)  # adiciona o índice desse par à R
            num_s.append(elem[ind])  # adiciona o par à num_s
            prim += elem[ind][0]     # atualiza o numerador
            sec += elem[ind][1]      # atualiza o denominador
            R_ = prim/sec         # atualiza o R após a inserção



        quo[ind] = -inf

    first_size, first_peak = tracemalloc.get_traced_memory()  #término da medição de memória

    tracemalloc.reset_peak()

    print("\nXXXXXX-----FIM-----XXXXXXX")
    print(f"\n==============\n Par obrigatório: {num[0]}")
    print(f"\n==============\nS: {S}")
    print(f"\n==============\n Pares selecionados: {num_s}")
    print(f"\n==============\nR_: {R_}")
    return R_,first_size
```

Figura 2: Algoritmo $O(N^2)$.

1.2 Usando ordenação

A ordenação dos elementos de quo $= [\frac{a_1}{b_1}, \frac{a_2}{b_2}, \cdots, \frac{a_n}{b_n}]$ pode ser utilizada como um mecanismo auxiliar para o PPH. Isso se dá, uma vez que basta iterar sobre o novo vetor ordenado de forma decrescente, e ir adicionando o índice dos elementos enquanto eles forem maior que o R atual, sendo que a cada nova adição o R é atualizado. Ao encontrar um elemento que seja menor ou igual a R, o loop é cessado. Todos os pares que S está representando após o loop deverão ser escolhidos, uma vez que o último índice j adicionado a S representa um par (a_j, b_j) com o menor quociente $\frac{a_j}{b_j}$ dentre os quocientes calculados dos pares com índices em S. Como j foi adicionado a S, então $\frac{a_j}{b_j}$ é maior que o R atual e como todos os outros pares possuem o quociente $\frac{a_k}{b_k}$ maior que $\frac{a_j}{b_j}$, então todos terão o quociente maior que o R atual. A Figura 3 mostra um esquema que representa as iterações a e a Figura 4 apresenta a implementação.

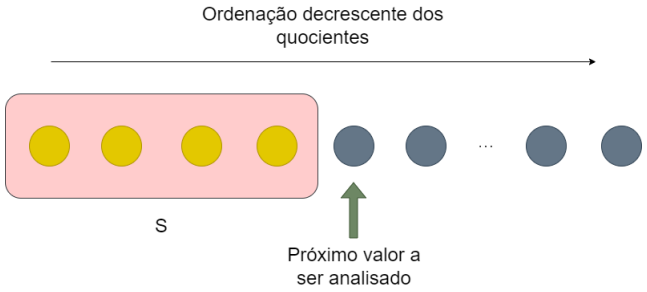


Figura 3: Esquematização das iterações.

Nessa implementação, a complexidade do algoritmo de ordenação definirá a complexidade do algoritmo, pois a ordenação será a etapa que mais consome tempo de processamento. Assim, o algoritmo possui dois parâmetros: o vetor com os pares que serão analisados e o algoritmo de ordenação. Para tais algoritmos foram usadas 6 opções:

- $O(N^2)$
  - bubbleSort
  - insertionSort
  - selectionSort
- $O(NlogN)$
  - quickSort
  - mergeSort
  - heapSort

A Figura 5 mostra o tempo consumido usando o mergeSort, que em qualquer caso tem complexidade $O(N \log N)$.

Comparando as implementações com cada algoritmo de ordenação, é possível gerar o gráfico da Figura 6.

Percebe-se que são gerados dois grupos de curvas, cada grupo representando o algoritmo com uma das complexidades. As três curvas com maior tempo sendo as representantes de $O(N^2)$ e as outras três sendo as representantes de $O(N \log N)$.

```python
def PPH_ord(num,algo):
    tracemalloc.start()
    S = [] # índices das tuplas selecionadas
    num_s = [] # tuplas que possuem seus índices em S

    #xxxxxxxxxxxxxxxxxxxxxxx
    prim = num[0][0]    # numerador considerando apenas a0
    sec = num[0][1]    # denominador considerando apenas b0
    R_ = prim/sec      # valor de R considerando apenas (a0,b0)

    vet = []   # vetor que armazenará as razões ai/bi

    for i in range(1,len(num)):
        if num[i][1]==0:
            vet.append(inf)
        else:
            vet.append(num[i][0]/num[i][1])


    vet,ind = algo(vet) #aplica o algoritmo de ordenação
    #xxxxxxxxxxxxxxxxxxxxxx

    for i in range(len(vet)):

        if vet[i]>R_: #verifica se o quociente analisado é maior que o R atual

            S.append(ind[i]+1)
            num_s.append(num[ind[i]+1])
            prim += num[ind[i]+1][0] #atualiza o numerador
            sec += num[ind[i]+1][1]  #atualiza o denominador
            R_ = prim/sec   # atualiza R


        else: #caso o quociente analisado seja menor ou igual ao R atual, o loop é cessado
            break


    first_size, first_peak = tracemalloc.get_traced_memory()
    tracemalloc.reset_peak()

    print(f"R_: {R_}")
    return R_,first_size
```

Figura 4: Algoritmo 2.

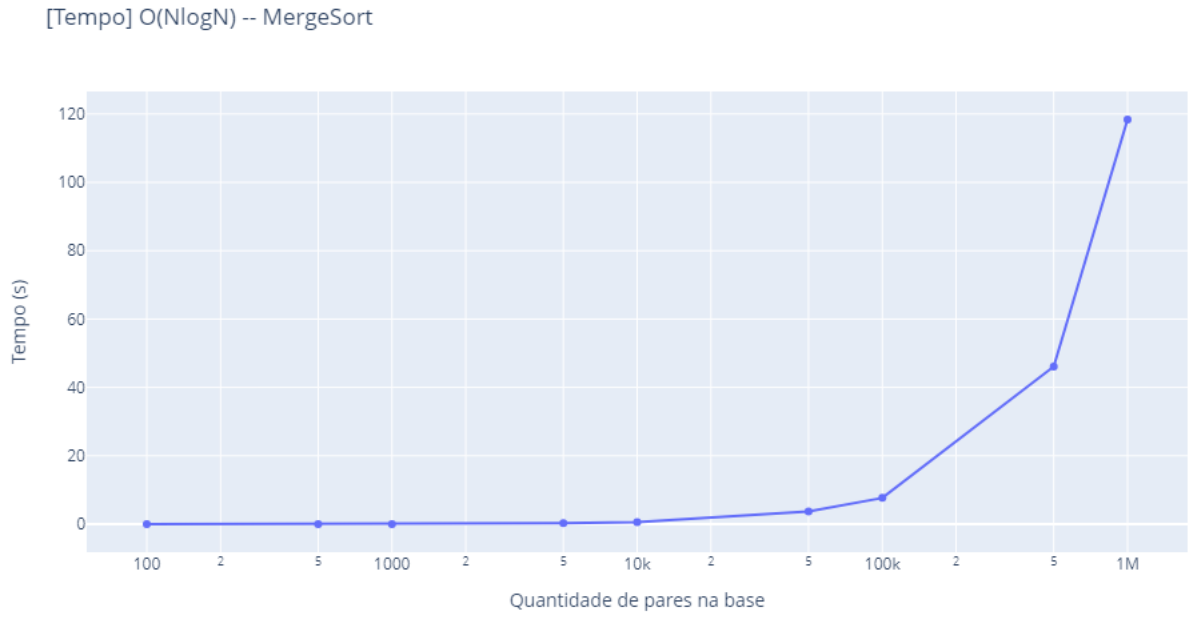


Figura 5: Tempo de execução do PPH usando mergeSort.

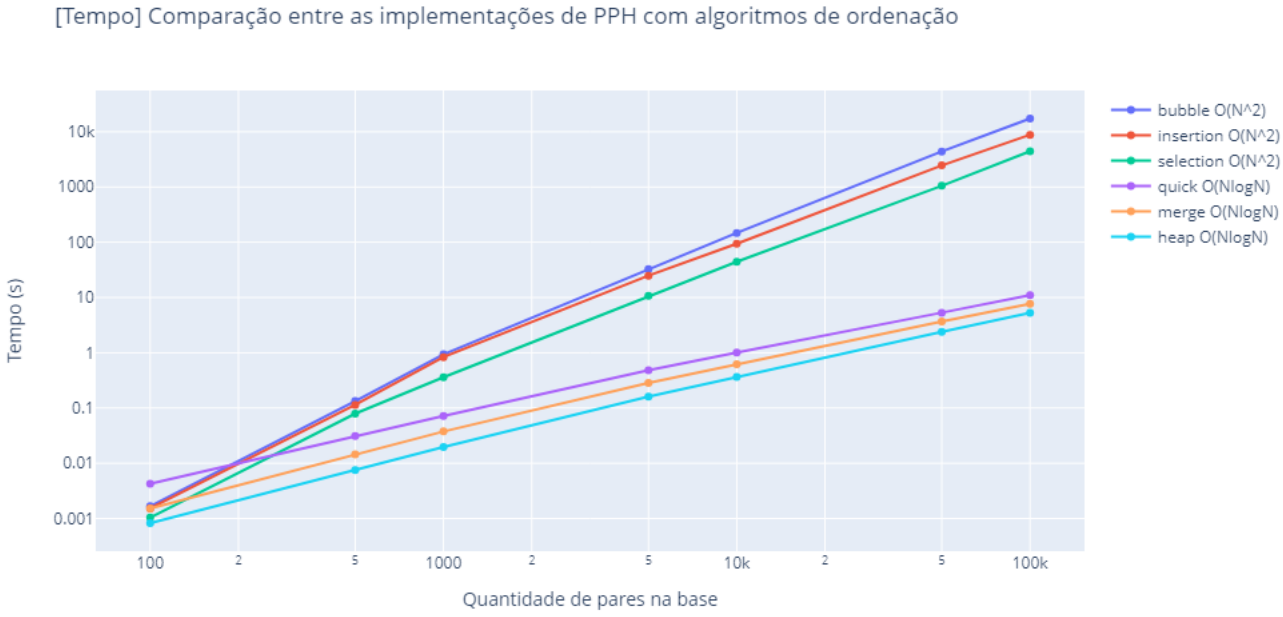


Figura 6: Comparação entre implementações de PPH com algoritmos de ordenação.

1.3 Versão $O(N)$

Esta versão utiliza dois algoritmos auxiliares com complexidade $O(N)$: mediana das medianas e partition. Inicialmente, J é um vetor com todos os índices $1, 2 \cdots, n$. A cada iteração J é analisado e separado em 3 partes, utilizando o algoritmo partition:

- J_0: índices dos pares cujo quociente é menor que o R atual.
- J_1: índices dos pares cujo quociente é maior que o R atual.
- J_2: índices dos pares cujo quociente é igual que o R atual.

Para a execução do partition, o pivot escolhido é a mediana das medianas dos quocientes dos pares cujos índices estão em J. A cada iteração é realizado o teste de optimalidade. Caso ele falhe, pelo menos metade do restante variáveis são eliminadas, já que $J \leftarrow J_0$ ou $J \leftarrow J_1$, e a busca prossegue para a próxima iteração. Portanto, a complexidade do algoritmo é $O(N) + O(\frac{N}{2}) + O(\frac{N}{4}) + \cdots = O(N)$. A Figura 7 mostra o gráfico de consumo de tempo do algoritmo e a Figura 8 apresenta sua implementação.

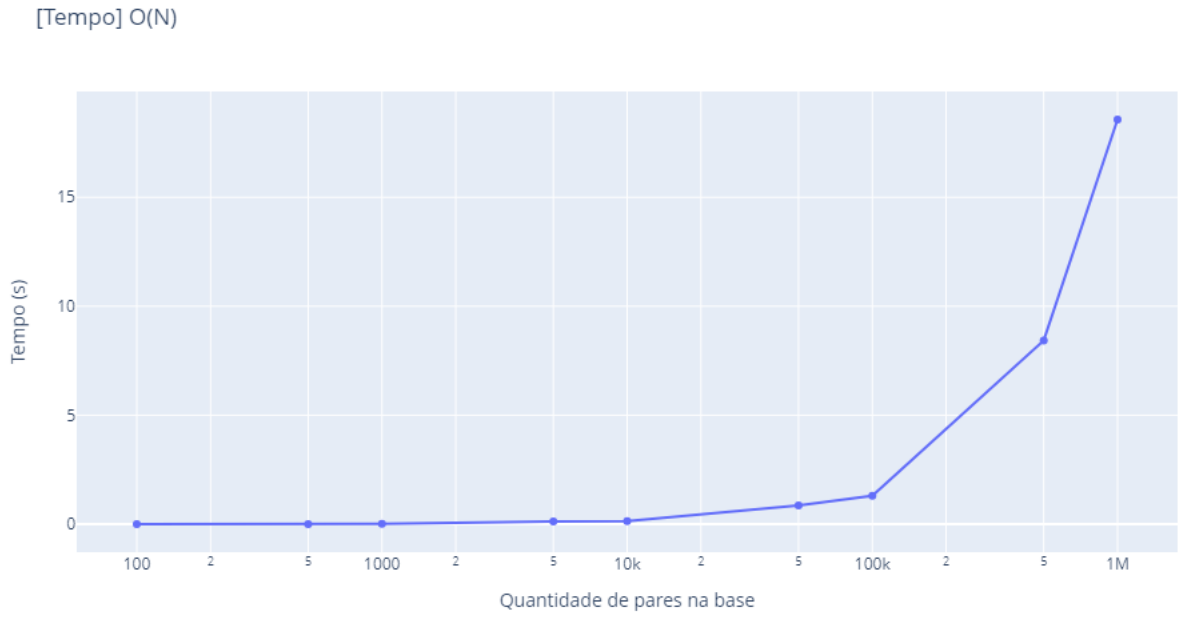


Figura 7: Tempo de execução para o algoritmo O(N).

```python
def PPH_N(num):
    tracemalloc.start()
    c0 = num[0][0]   # elemento a0 do par obrigatório
    d0 = num[0][1]   # elemento b0 do par obrigatório
    prim = c0        # numerador considerando apenas a0
    sec = d0         # denominador considerando apenas b0
    elem = num[1:]   # elem recebe todos os pares não obrigatórios
    x = [0 for i in range(len(elem))] # x é um vetor que contem 1's e 0's, mostrando se o par i foi
selecionados ou não, respectivamente
    J = [i for i in range(len(elem))] # J conterá os índices dos pares selecionados
    stop = False  # variável que controla o loop a seguir

    m=0
    while(not stop):

        print(f"XXXXXXXXXXXXXX ITERAÇÃO {m} XXXXXXXXXXXXXX\n")
        t_iter = time_ns()
        elem = [elem[i] for i in J] # pares representados em J
        quo = []  # armazena os quocientes ai/bi dos pares em elem

        for e in elem:
            if e[1]==0:  # verifica se o segundo elemento do par é 0
                quo.append(inf)
            else:
                quo.append(e[0]/e[1])

        if len(quo)==0:   # se quo não tiver nenhum elemento, deve-se parar o loop
            break

        if len(quo)==1:   # se quo tiver apenas um elemento, median recebe esse elemento
            median = quo[0]

        else:
            t_ini = time_ns()
            median = median_of_medians(quo)   # calcula a mediana das medianas em quo

        t_ini = time_ns()
        elemJ0,J0,elemJ2,J2,elemJ1,J1= partition(quo,median)  # aplica o algoritmo partition usando
median como pivot

        # Jun02 deve ser a união de J0 com J2
        Jun02 = J0[:]
        Jun02.extend(J2)

        # Jun12 deve ser a união de J1 com J2
        Jun12 = J1[:]
        Jun12.extend(J2)
        t_ini = time_ns()
        prim = c0 +sum([elem[j][0] for j in Jun12])
        sec = d0 +sum([elem[j][1] for j in Jun12])
        R_ = prim/sec   # atualiza R

        if R_>median:

            for j in Jun02:
                x[j]=0  # nenhum dos pares representados por J02 é selecionado

            J = J1[:]   # J recebe J1

        else:

            for j in Jun12:
                x[j]=1   # todos os pares representados por J12 são selecionados
            stop = True   # ajusta a variável stop para que o loop cesse

            if len(J0)!=0:
                maior = max(elemJ0)
                if R_ < maior:
                    J=J0[:]    # J recebe J0
                    c0 = prim
                    d0 = sec
                    stop = False   # garante mais uma iteração do loop

                else:
                    for i in J0:
                        x[i]=0   # nenhum dos pares representados por J0 é selecionado

        m+=1

    print(R_)
    first_size, first_peak = tracemalloc.get_traced_memory()

    tracemalloc.reset_peak()
    return R_, first_size
```

Figura 8: Algoritmo 3.

1.4 Versão O(N) alterando o pivot

Esta implementação é uma adaptação da implementação anterior, alterando o modo como o pivot é selecionado. Ele passa a ser dado por:

$$\frac{a_0 + \sum_{t \in K} a_t}{b_0 + \sum_{t \in K} b_t}$$

A Figura 9 mostra um comparativo de tempo de execução das duas versões.

Figura 9: Comparação entre diferentes escolhas para o pivot.

Percebe-se que a performance do algoritmo é melhor do que o anterior que seleciona o pivot como sendo a mediana das medianas. Analisando as instâncias, pode se ter uma estimativa de que este algoritmo tem complexidade $O(\log N)$.

1.5 Comparação

Selecionando um algoritmo de cada complexidade e analisando os tempos de execução e possível compará-los conforme é mostrado na Figura 10.

Figura 10: Comparação entre diferentes complexidades.

1.6 Tabelas

As tabelas com os dados de tempo de todos os algoritmos são mostradas a seguir.

	nome das instâncias	complexidade teórica	tempo de CPU	razão	valores de tempo mais altos	valores mais baixos
0	pph_100	6.643856e+02	0.003753	177024.488581	0.013016	0.000907
1	pph_500	4.482892e+03	0.037186	120552.494277	0.096965	0.020711
2	pph_1000	9.965784e+03	0.116630	85447.787022	0.188976	0.070187
3	pph_5000	6.143856e+04	2.327486	26396.963598	2.717208	1.879808
4	pph_10000	1.328771e+05	8.222180	16160.814506	8.535072	7.969117
5	pph_50000	7.804820e+05	211.962921	3682.163000	221.083324	207.610358
6	pph_100000	1.660964e+06	730.175508	2274.746316	842.598978	546.169080
7	pph_500000	9.465784e+06	13236.910806	715.105240	13520.545085	12997.922596
8	pph_1000000	1.993157e+07	54152.809927	368.061576	54152.809927	54152.809927

Figura 11: Tabela com dados de tempo do Algoritmo 1.

	nome das instâncias	complexidade teórica	tempo de CPU usado	razão	valores de tempo mais altos	valores mais baixos
0	pph_100	6.643856e+02	0.001673	397065.971960	0.247936	0.000000
1	pph_500	4.482892e+03	0.133663	33538.808462	0.194268	0.087997
2	pph_1000	9.965784e+03	0.937475	10630.453099	1.038457	0.829442
3	pph_5000	6.143856e+04	32.486800	1891.185378	33.186411	31.864412
4	pph_10000	1.328771e+05	147.188047	902.771157	155.856245	142.578707
5	pph_50000	7.804820e+05	4394.999676	177.584091	4483.253215	4263.904182
6	pph_100000	1.660964e+06	17343.573648	95.768270	18208.291551	16724.224589

Figura 12: Tabela com dados de tempo do Algoritmo 2 usando bubbleSort.

	nome das instâncias	complexidade teórica	tempo de CPU usado	razão	valores de tempo mais altos	valores mais baixos
0	pph_100	6.643856e+02	0.001526	435345.856542	0.011091	0.000000
1	pph_500	4.482892e+03	0.114284	39225.768224	0.170146	0.075095
2	pph_1000	9.965784e+03	0.832763	11967.136089	1.535410	0.684594
3	pph_5000	6.143856e+04	24.854458	2471.933233	26.202078	23.120994
4	pph_10000	1.328771e+05	93.900117	1415.090069	105.591585	83.677448
5	pph_50000	7.804820e+05	2462.197601	316.985941	2654.300658	2261.564339
6	pph_100000	1.660964e+06	8810.267815	188.525943	9281.748903	8539.037134

Figura 13: Tabela com dados de tempo do Algoritmo 2 usando insertionSort.

	nome das instâncias	complexidade teórica	tempo de CPU usado	razão	valores de tempo mais altos	valores mais baixos
0	pph_100	6.643856e+02	0.001042	637594.454875	0.034300	0.000000
1	pph_500	4.482892e+03	0.078948	56782.812121	0.130327	0.052204
2	pph_1000	9.965784e+03	0.360999	27606.111271	0.453299	0.288968
3	pph_5000	6.143856e+04	10.745153	5717.792974	11.990215	9.772939
4	pph_10000	1.328771e+05	44.355538	2995.727945	46.434199	42.704623
5	pph_50000	7.804820e+05	1054.477317	740.160088	1081.418355	1037.056726
6	pph_100000	1.660964e+06	4434.427395	374.561110	4612.383332	4271.777545

Figura 14: Tabela com dados de tempo do Algoritmo 2 usando selectionSort.

	nome das instâncias	complexidade teórica	tempo de CPU usado	razão	valores de tempo mais altos	valores mais baixos
0	pph_100	6.643856e+02	0.004254	156163.956354	0.242410	0.000506
1	pph_500	4.482892e+03	0.031259	143410.230633	0.067686	0.018793
2	pph_1000	9.965784e+03	0.071521	139339.860534	0.128090	0.041998
3	pph_5000	6.143856e+04	0.480549	127850.804831	0.630643	0.380541
4	pph_10000	1.328771e+05	1.006214	132056.581642	1.110429	0.825705
5	pph_50000	7.804820e+05	5.313607	146883.669321	5.537093	5.118325
6	pph_100000	1.660964e+06	11.090159	149769.181844	12.115880	10.075878
7	pph_500000	9.465784e+06	50.048619	189131.776602	54.878935	47.648910
8	pph_1000000	1.993157e+07	101.141560	197066.058689	104.999140	94.767607

Figura 15: Tabela com dados de tempo do Algoritmo 2 usando quickSort.

	nome das instâncias	complexidade teórica	tempo de CPU usado	razão	valores de tempo mais altos	valores mais baixos
0	pph_100	6.643856e+02	0.001526	435328.318990	0.005027	0.000000
1	pph_500	4.482892e+03	0.014562	307839.517097	0.037084	0.009645
2	pph_1000	9.965784e+03	0.037613	264956.580123	0.073690	0.028067
3	pph_5000	6.143856e+04	0.282845	217216.523231	0.403809	0.233046
4	pph_10000	1.328771e+05	0.619152	214611.626985	0.709085	0.531099
5	pph_50000	7.804820e+05	3.682599	211937.823918	4.002770	3.363956
6	pph_100000	1.660964e+06	7.670532	216538.317127	7.886114	7.563856
7	pph_500000	9.465784e+06	46.108782	205292.439320	47.064392	45.433691
8	pph_1000000	1.993157e+07	118.376031	168375.036994	122.353111	113.557653

Figura 16: Tabela com dados de tempo do Algoritmo 2 usando mergeSort.

	nome das instâncias	complexidade teórica	tempo de CPU usado	razão	valores de tempo mais altos	valores mais baixos
0	pph_100	6.643856e+02	0.000825	805584.896155	0.168412	0.000000
1	pph_500	4.482892e+03	0.007580	591407.164081	0.022053	0.003694
2	pph_1000	9.965784e+03	0.019762	504283.803905	0.050287	0.011741
3	pph_5000	6.143856e+04	0.161172	381199.299922	0.214760	0.108240
4	pph_10000	1.328771e+05	0.366934	362128.576849	0.476263	0.308091
5	pph_50000	7.804820e+05	2.387527	326899.759999	2.515765	2.262610
6	pph_100000	1.660964e+06	5.267426	315327.467842	5.399400	5.069822
7	pph_500000	9.465784e+06	34.762258	272300.616118	36.709313	33.050697
8	pph_1000000	1.993157e+07	71.870465	277326.279528	75.636779	69.155720

Figura 17: Tabela com dados de tempo do Algoritmo 2 heapSort.

	nome das instâncias	complexidade teórica	tempo de CPU	razão	valores de tempo mais altos	valores mais baixos
0	pph_100	6.643856e+02	0.001492	4.453394e+05	0.008774	0.000000
1	pph_500	4.482892e+03	0.008576	5.227232e+05	0.052934	0.002517
2	pph_1000	9.965784e+03	0.013881	7.179515e+05	0.032084	0.007254
3	pph_5000	6.143856e+04	0.114629	5.359782e+05	0.245339	0.058853
4	pph_10000	1.328771e+05	0.163549	8.124623e+05	0.370414	0.071529
5	pph_50000	7.804820e+05	1.063225	7.340707e+05	1.288739	0.611755
6	pph_100000	1.660964e+06	1.618823	1.026032e+06	2.575671	1.140499
7	pph_500000	9.465784e+06	9.110620	1.038984e+06	10.499479	6.595667
8	pph_1000000	1.993157e+07	20.876952	9.547164e+05	24.521739	16.149204

Figura 18: Tabela com dados de tempo do Algoritmo 3.

	nome das instâncias	complexidade teórica	tempo de CPU	razão	valores de tempo mais altos	valores mais baixos
0	pph_100	6.643856e+02	0.000500	1.327696e+06	0.028989	0.000000
1	pph_500	4.482892e+03	0.002899	1.546587e+06	0.008471	0.000587
2	pph_1000	9.965784e+03	0.006065	1.643242e+06	0.014667	0.001985
3	pph_5000	6.143856e+04	0.028013	2.193212e+06	0.071176	0.010667
4	pph_10000	1.328771e+05	0.063020	2.108505e+06	0.107364	0.028821
5	pph_50000	7.804820e+05	0.296100	2.635869e+06	0.443319	0.177859
6	pph_100000	1.660964e+06	0.551422	3.012149e+06	0.753534	0.367822
7	pph_500000	9.465784e+06	2.666951	3.549290e+06	3.853700	1.803583
8	pph_1000000	1.993157e+07	5.830701	3.418383e+06	6.805167	4.884969

Figura 19: Tabela com dados de tempo do Algoritmo 3 modificado.

2 Problema da Mochila Fracionária

O problema da mochila é um dos clássicos em computação. Nesta seção iremos abordar sobre este tópico, em especial, a mochila fracionária e analisar algoritmos com diferentes graus de complexidade para sua resolução.

O problema da mochila fracionária pode ser descrita como:

> Dado um conjunto $I = \{1, 2, \ldots, n\}$ de n itens onde cada $i \in I$ tem um peso w_i e um valor v_i associados e dada uma mochila com capacidade de peso W, selecionar frações $f_i \in [0, 1]$ dos itens tal que $\sum_{i=1}^{n} f_i w_i \leq W$ e $\sum_{i=1}^{n} f_i v_i$ é máximo.

Figura 20: Problema da Mochila Fracionária.

Logo, dados os pesos e lucros de N itens, na forma de lucro, peso coloque esses itens em uma mochila de capacidade W para obter o lucro total máximo na mochila. Na Mochila Fracionária , podemos quebrar itens para maximizar o valor total da mochila.

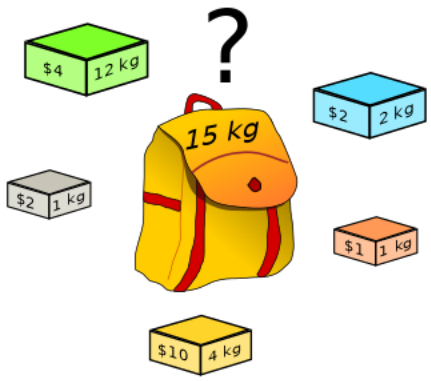


Figura 21: Problema da mochila: Como maximizar o valor com um peso máximo?

A base utilizada para executar os algoritmos apresentados a seguir contém 12 instâncias que aumentam exponencialmente.

2.1 Versão $O(N \cdot \log N)$

Uma estratégia para solucionar o problema da mochila fracionária é a gulosa. Onde sempre escolhemos o item de maior que ainda cabe na mochila. Porém, ao utilizar esta forma de resolução não teremos uma solução ótima, mais sim uma solução viável para o problema.

A utilização de uma estratégia gulosa nos retorna um algoritmo de complexidade $O(N \cdot \log N)$, onde o gargalo do algoritmo se encontra na ordenação dos valores de entrada. Seguindo o pseudocódigo da figura 22, onde é utilizado para ordenação o MergeSort de tempo $O(N \cdot \log N)$.

```
1  Ordene os itens pela razão valor/peso e os renomeie de forma que
     v_1/w_1 ≥ v_2/w_2 ≥ ··· ≥ v_n/w_n
2  capacidade = W
3  Seja f[1..n] um vetor
4  i = 1
5  enquanto i ≤ n e capacidade ≥ w_i faça
6  |  f[i] = 1
7  |  capacidade = capacidade − w_i
8  |  i = i + 1
9  se i ≤ n então
10 |  f[i] = capacidade/w_i
11 para j = i + 1 até n, incrementando faça
12 |  f[j] = 0
13 devolve f
```

Figura 22: Pseudocódigo - Mochila Fracionária $O(N \cdot \log N)$.

O funcionamento do algoritmo pode ser compreendido através das seguintes etapas:

1. **Calculo da relação**: Calcula a relação lucro/peso para cada valor de entrada.
2. **Ordenação da entrada**: Ordena os n elementos, através do algoritmo MergeSort, organizando em ordem decrescente.
3. **Interação**: Percorre os itens ordenados
   (a) **Condição**: Verifica se o peso do item atual for menor ou igual à capacidade restante, adicione o valor desse item ao resultado.
   (b) **Condição**: Caso contrário, adicione o item atual o máximo que pudermos e saia do loop.

A implementação do algoritmo pode ser visualizada na figura 23:

```python
def fractionalKnapsack(W, arr):
  arr.sort(key=lambda x: (x.profit/x.weight),
reverse=True)
  finalvalue = 0.0

  for item in arr:
    if item.weight <= W:
      W -= item.weight
      finalvalue += item.profit
    else:
      finalvalue += item.profit * W / item.weight
      break

  return finalvalue
```

Figura 23: Implementação do algoritmo que seleciona o pivô.

O resultado de consumo de memória obtido para cada tamanho de instância pode ser visualizado na figura 24.

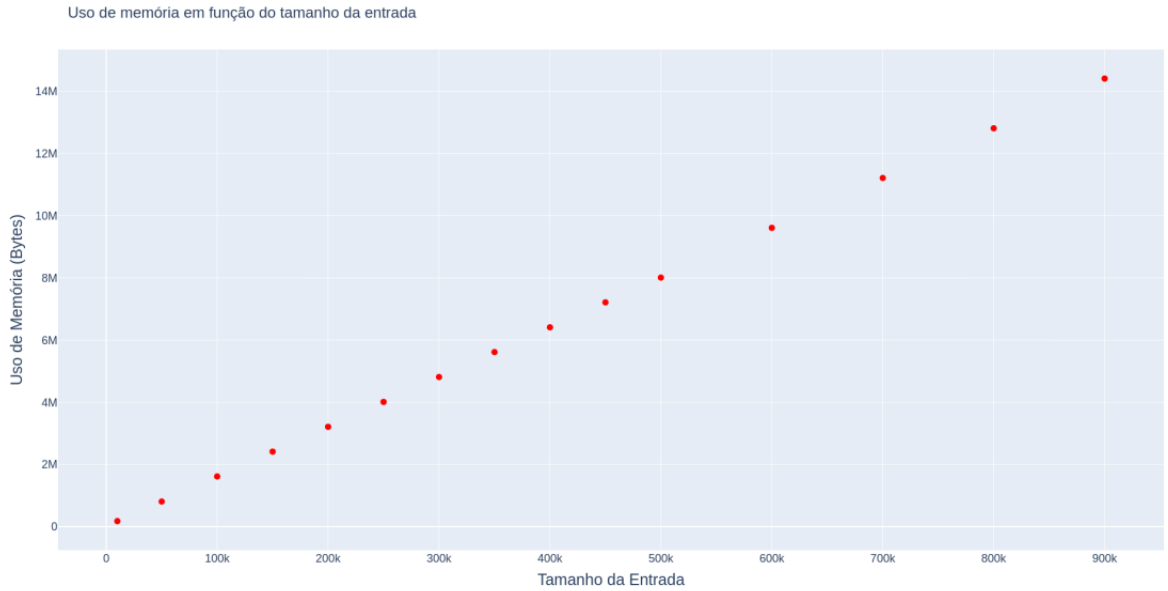


Figura 24: Consumo de memória para cada instância utilizada.

O resultado do tempo de execução para cada tamanho de instância pode ser visualizado na figura 25.

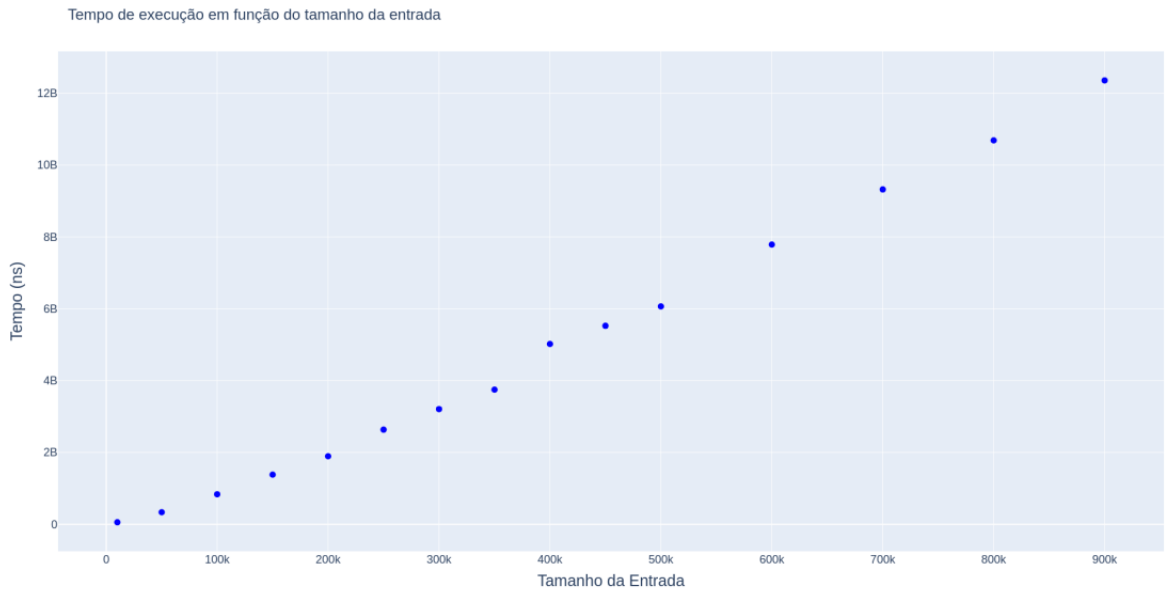


Figura 25: Tempo de execução para cada instância utilizada.

A tabela 1 apresenta os dados de tempo, memoria e complexidade.

Instância	Tmp medio	Menor tmp	Maior tmp	N° de repetições	Mem. média
10000	53694915.39325843	52873111	68448857	89	178003.42696629214
50000	333247944.8666667	311515109	390949186	15	809430.8666666667
100000	833994083.5	791211130	994744959	6	1613944.3333333333
150000	1379051709.5	1269596009	1654673369	4	2414196.0
200000	1891951694.33	1741124456	2132356345	3	3211033.0
250000	2631839275.5	2368729626	2894948925	2	4013161.0
300000	3204603787.5	2888227937	3520979638	2	4812629.0
350000	3744835675.5	3356336982	4133334369	2	5614701.0
400000	5018115589.0	5018115589	5018115589	1	6412309.0
450000	5523472316.0	5523472316	5523472316	1	7214397.0
500000	6064581388.0	6064581388	6064581388	1	8010813.0
600000	7785959656.0	7785959656	7785959656	1	9611785.0
700000	9319398201.0	9319398201	9319398201	1	11213569.0
800000	10685000127.0	10685000127	10685000127	1	12812013.0
900000	12356098414.0	12356098414	12356098414	1	14412873.0

Tabela 1: Tabela com dados de tempo do algoritmo da mochila $O(N \cdot \log N)$

2.2 Versão $O(N)$

O fator que nos impede de alcançar uma complexidade mais eficiente,no caso $O(N \cdot \log N)$, está relacionado à etapa de ordenação. Na abordagem $O(N)$, decidimos abandonar a ordenação e adotar uma estratégia semelhante àquela utilizada pelo algoritmo de seleção do i-ésimo menor ou maior elemento em um arranjo.

Um aspecto fundamental desse algoritmo é a seleção do pivô usando a técnica da mediana das medianas, garantindo assim que a complexidade permaneça em $O(N)$. A Figura 26 exemplifica o comportamento dos elementos em torno da mediana das medianas.

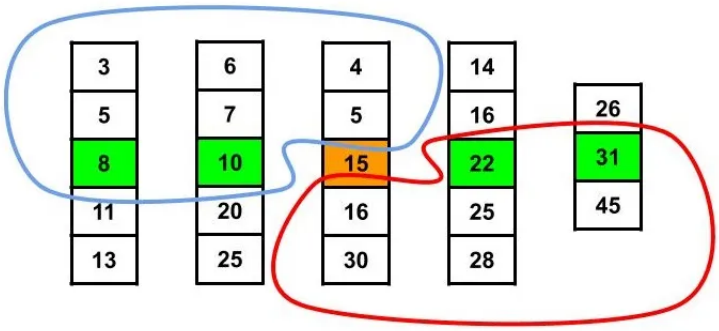


Figura 26: Tendência dos Elementos em Relação à Mediana das Medianas.

O funcionamento do algoritmo pode ser compreendido através das seguintes etapas:

1. **Divisão em Grupos**: Os n elementos do arranjo de entrada são divididos em grupos de cinco elementos cada, resultando em n/5 grupos.
2. **Determinação das Medianas Locais**: Para cada um dos n/5 grupos, a mediana é determinada. Isso é feito ordenando os elementos do grupo e selecionando a mediana da lista ordenada.
3. **Seleção da Mediana Global**: A mediana x é selecionada a partir das n/5 medianas locais calculadas na etapa 2.
4. **Particionamento em Torno do Pivô**: O arranjo de entrada é particionado em torno da mediana global, que agora atua como o pivô.
5. **Verificando se é um pivô ideal**: Na última etapa, verifica-se se a soma dos pesos até o pivô é igual ou superior à capacidade máxima da mochila. Se for, esse pivô é considerado ideal e é retornado. No entanto, se a soma for menor que a capacidade máxima, avaliamos o lado inferior ao pivô. Se esse lado tiver uma soma maior que a capacidade máxima, então o lado superior ao pivô é avaliado.

A parte fundamental que assegura a complexidade de $O(N)$ deste algoritmo reside na seleção do pivô utilizado durante o processo de particionamento. A implementação desse método de seleção de pivô pode ser observada na Figura 27.

```python
def ratioCmp(item):
  return item[0] / item[1]

def select_pivot_v1(obj, ini, fim):
  # Armazenamos o número de candidatos para ser o pivô
  ncand = (fim - ini ) + 1
  # Preenchemos uma lista com os indices dos candidados
  cand = list(range(ini, fim+1))

  while ncand > 1:
    storeind = 0

    # Aqui dividimos a lista de candidatos em grupos de 5
    for i in range(0, ncand, 5):
      posini, posfim = i, min(i + 4, ncand)

      # Ordenamos este grupo de 5 elementos baseando na razão valor/peso
      cand[posini:posfim] = sorted(cand[posini:posfim], key=lambda x: ratioCmp(obj[x]))

      # Pegamos a mediana deste grupo
      cand[storeind] = cand[(posini + posfim) // 2]

      storeind += 1

    # Aqui atualizamos o número de candidatos depois que pegamos a mediana de cada grupo
    ncand = storeind

  return cand[0]
```

Figura 27: Implementação do algoritmo que seleciona o pivô.

Os resultados obtidos para cada tamanho das instâncias podem ser visualizados na Figura 30.

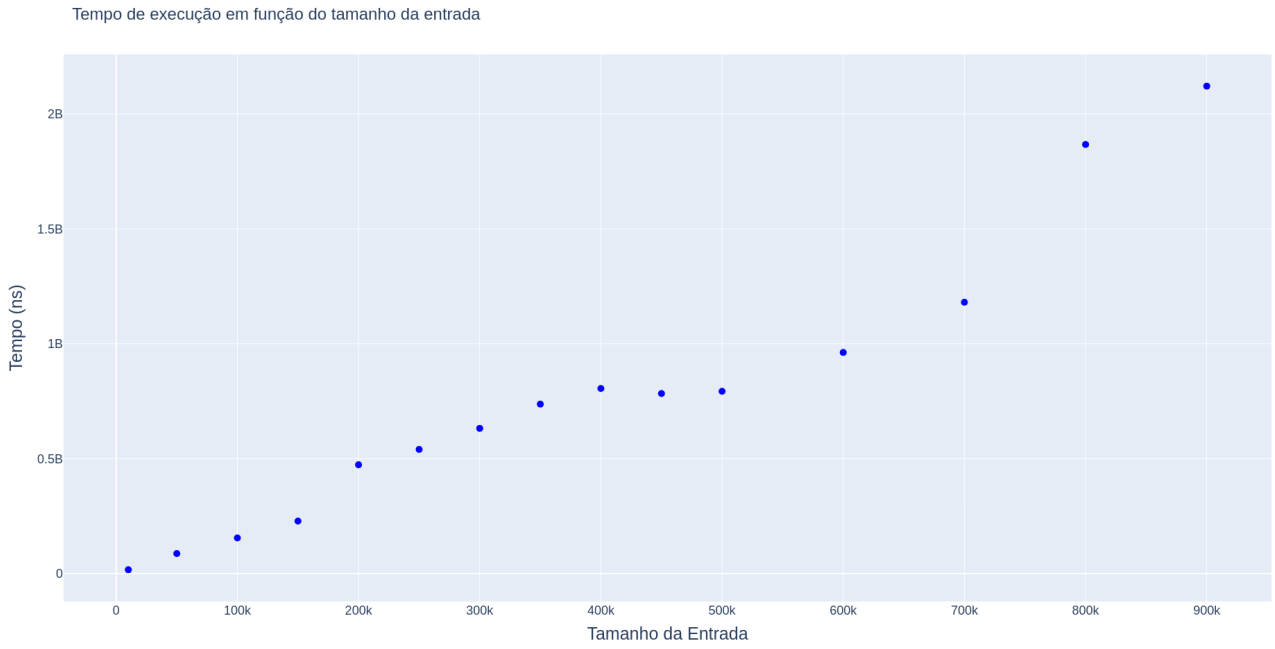


Figura 28: Tempo de execução em função do tamanho da entrada.

2.3 Versão $O(N)$ alterando o pivô

Esta implementação representa uma variação da anterior, com a principal diferença sendo o cálculo do pivô, agora determinado pela média conforme apresentado na equação a seguir:

$$pivot = \frac{1}{|K|} \cdot \sum_{j \in K} \frac{v_j}{w_j} \tag{1}$$

Na imagem a seguir, é possível visualizar o código utilizado para calcular o pivô com base na equação mencionada anteriormente:

```python
def select_pivot_v2(obj, start, end):
  # Caso em que só existe um elemento no array maior que pivô
  if(start > end):
    return obj[start][0]/obj[start][1]

  # Armazena a soma total da razões
  totalSum = 0
  # Armazena o tamnho do arranjo
  size = (end - start) + 1

  for index in range(start, end+1):
    totalSum += obj[index][0]/obj[index][1]

  # retorna a media
  return totalSum/size
```

Figura 29: Implementação do algoritmo que seleciona o pivô.

Os resultados obtidos após a execução deste algoritmo revelam uma tendência linear no tempo em relação ao tamanho da entrada, como pode ser observado a seguir:

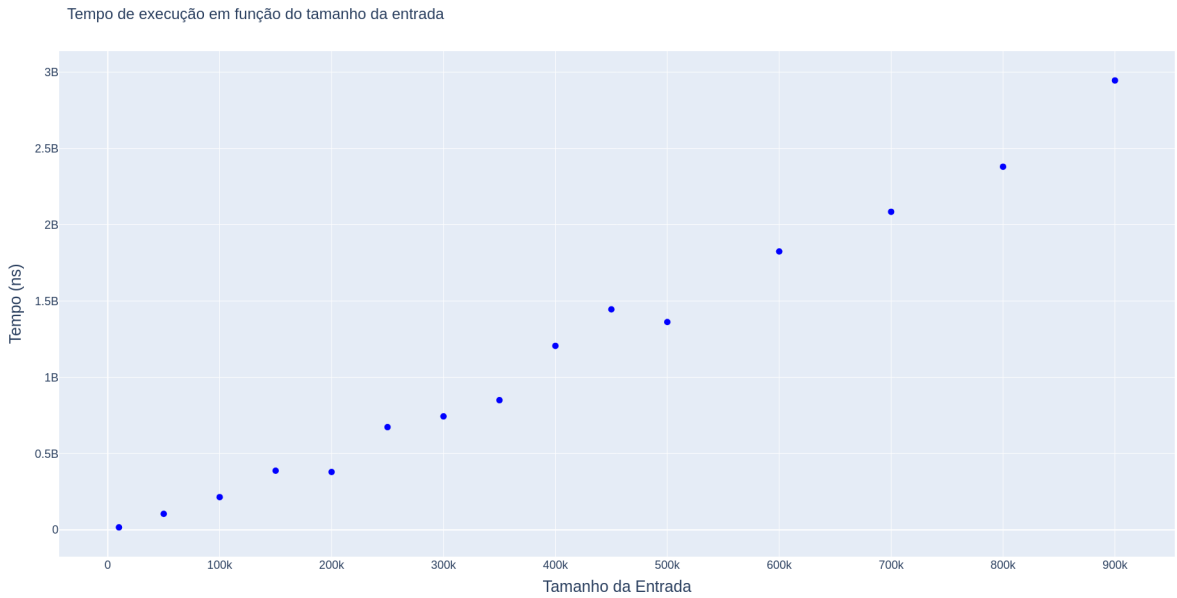


Figura 30: Tempo de execução em função do tamanho da entrada.

2.4 Comparações

Os tempos de execução em relação ao tamanho das entradas para os algoritmos com complexidades $O(N \cdot \log N)$, $O(N)$ usando a técnica da mediana das medianas e $O(N)$ utilizando a média das razões podem ser observados na Figura 31.

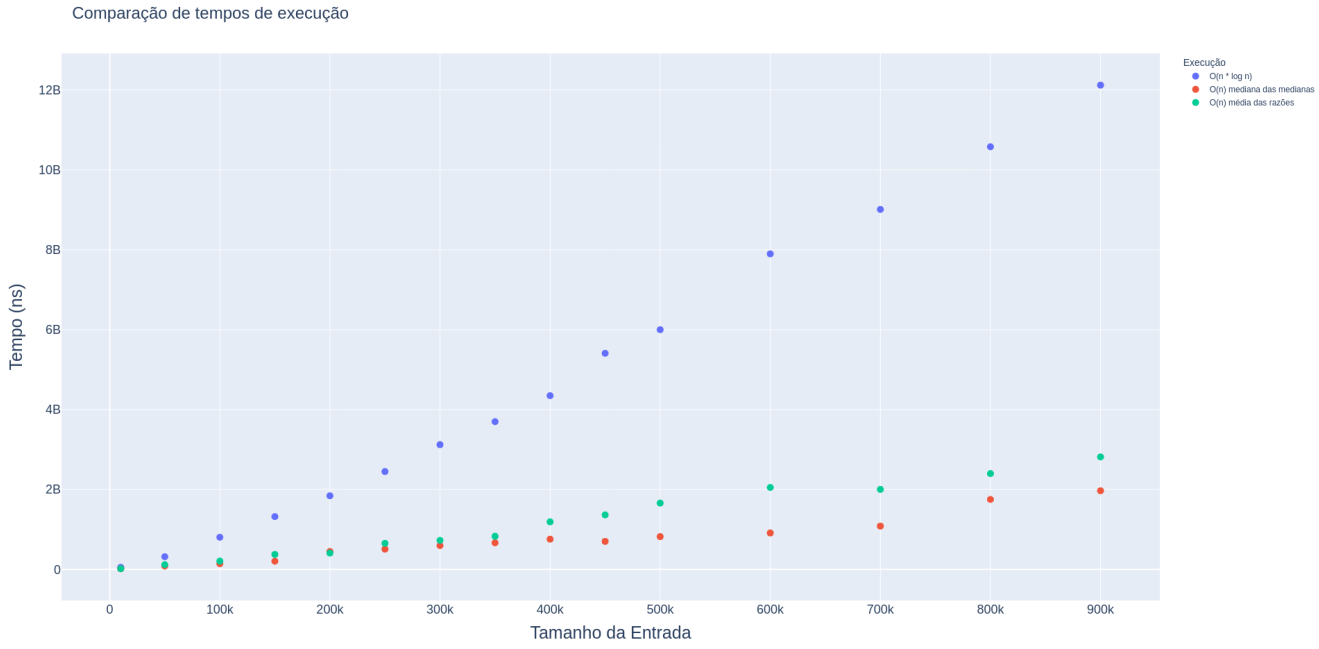


Figura 31: Comparação de desempenho entre os algoritmos

3 Problema da Árvore Geradora Mínima

3.1 Algoritmo de Prim

O algoritmo de Prim é usado para obter a Árvore Geradora Mínima (MST) dado um grafo de entrada $G(E, V)$ ponderado, não direcionado e conexo. Ele começa com um vértice arbitrário e adiciona a aresta de menor peso que conecta o vértice atual a um vértice fora da árvore geradora parcial. Esse processo é repetido até que todos os vértices estejam na árvore geradora. O pseudocódigo do algoritmo de Prim é exibido em **Algorithm 1**.

Algorithm 1 Algoritmo de Prim

```
1: procedure PRIM(G, w, r)
2:     Q ← fila de prioridade contendo todos os vértices de G
3:     key[v] ← ∞ para todo v ∈ G
4:     parent[v] ← indefinido para todo v ∈ G
5:     key[r] ← 0
6:     parent[r] ← nulo
7:     while Q não está vazia do
8:         u ← vértice em Q com o menor valor de chave
9:         remove u de Q
10:        for all vértices v adjacentes a u do
11:            if v está em Q e w(u, v) < key[v] then
12:                key[v] ← w(u, v)
13:                parent[v] ← u
14:            end if
15:        end for
16:    end while
17: end procedure
```

Onde:

- G: grafo não direcionado e ponderado.
- w: função de peso que atribui um peso a cada aresta do grafo.
- r: vértice raiz a partir do qual o algoritmo começa a construir a árvore geradora mínima.
- Q: fila de prioridade que contém todos os vértices do grafo.
- $key[v]$: menor chave conhecida para o vértice v.
- $parent[v]$: pai do vértice v na árvore geradora mínima.
- u: vértice selecionado da fila de prioridade com o menor valor de chave.
- v: é um vértice adjacente a um vértice u no grafo.

Geralmente a complexidade do algoritmo de Prim é dominada pela estrutura de dados que implementa a fila de prioridades Q. Portanto, é de grande importância a observação cautelosa da estrutura de dados a ser escolhida para a implementação do algoritmo. Nas subseções a seguir serão mostradas as implementações desse algoritmo e suas diferentes complexidades ao implementar a fila de prioridades Q usando **árvore binária**, **árvore AVL** e **Heap de Fibonacci**.

3.1.1 Árvore Binária (de busca)

É de conhecimento comum na Computação que a árvore binária é uma estrutura de dados que pode ter: zero elementos (árvore vazia) ou ter um elemento diferente dos demais - chamado de raiz - que possui dois ponteiros (que armazenam um endereço de memória) para outros dois nodos (subárvore esquerda e subárvore direita). No contexto da Teoria dos Grafos, diz-se que a árvore binária consiste em um grafo conexo (só há um caminho entre dois nodos diferentes), acíclico, dirigido e onde o máximo grau possível para cada nó é dois (2).

Então, a partir da estrutura acima explicada, foi criada a Árvore Binária de Busca (ABB) por P.F. Windley, A.D. Booth, A.J.T. Colin, e T.N. Hibbard. Há uma característica chave aqui: o valor (em número) da subárvore esquerda é menor que o valor contido pelo nó raiz, enquanto a subárvore direita tem valor maior que o nó raiz. Mas, neste momento, ainda não existe cálculo do fator de balanceamento (que aparece na AVL). Tal relaxamento de restrições implicou em uma implementação diferenciada para o código-fonte do algoritmo de Prim, visto abaixo:

```python
def prim(graph):
    visited = set()  # Set to keep track of visited nodes
    mst = defaultdict(dict)  # Dictionary to store the MST
    start_node = list(graph.keys())[0]  # Start from any node

    # Priority queue to store edges with weights
    priority_queue = [(start_node, None, 0)]

    while priority_queue:
        node, parent, weight = priority_queue.pop(0)  # Get the edge with the lowest weight
        if node not in visited:
            visited.add(node)  # Mark the current node as visited
            if parent is not None:
                mst[parent][node] = weight  # Add the edge to the MST
                mst[node][parent] = weight
            for neighbor, edge_weight in graph[node]:
                if neighbor not in visited:
                    priority_queue.append((neighbor, node, edge_weight))  # Add neighboring
edges to the priority queue
            priority_queue.sort(key=lambda x: x[2])  # Sort the priority queue based on edge
weights

    return mst
```

Figura 32: Implementação do algoritmo de Prim para a árvore binária de busca.

Por outro lado, sua complexidade depende diretamente da altura da árvore (isto é, a maior distância possível de um dado nó até o nó raiz), onde uma distribuição uniforme dos elementos gera altura O(log n) - para dividir o vetor linear original de elementos. Porém, se os elementos não estiverem ordenados da forma supracitada, a ABB apresenta pouca utilidade prática, pois acaba indesejadamente comportando-se com O(n).

Com tudo isso em mente, seguiu-se para a rodada de testes do algoritmo apresentado neste subtópico com os três grupos de entradas de grafos disponibilizadas pelo professor ("ALUE", "ALUT"e "DMXA"). Abaixo, em múltiplos gráficos, pode-se observar o comportamento de consumo de tempo (em nanossegundos) para execução relacionado com o número de vértices (V) e arestas (E) dos grafos tomados como *input*:

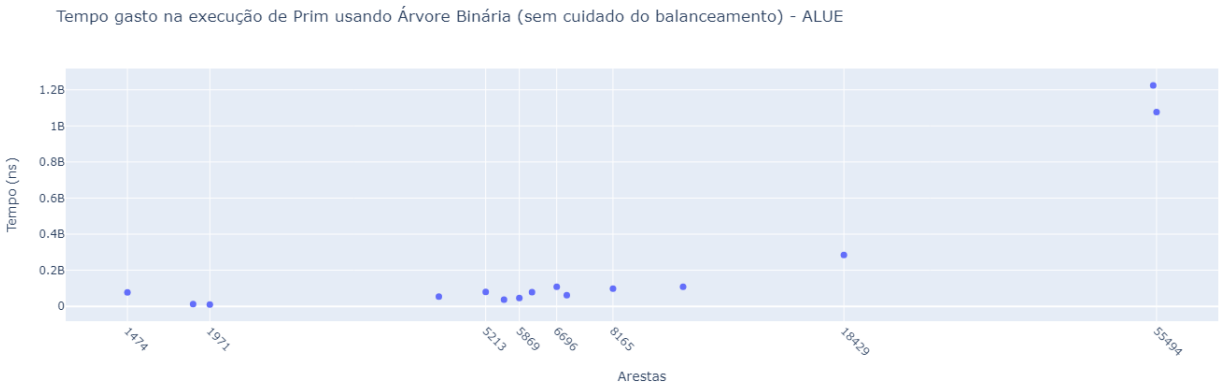


Figura 33: Arestas - Tempo na execução de Prim com Árvore Binária (sem cuidado do balanceamento) - ALUE

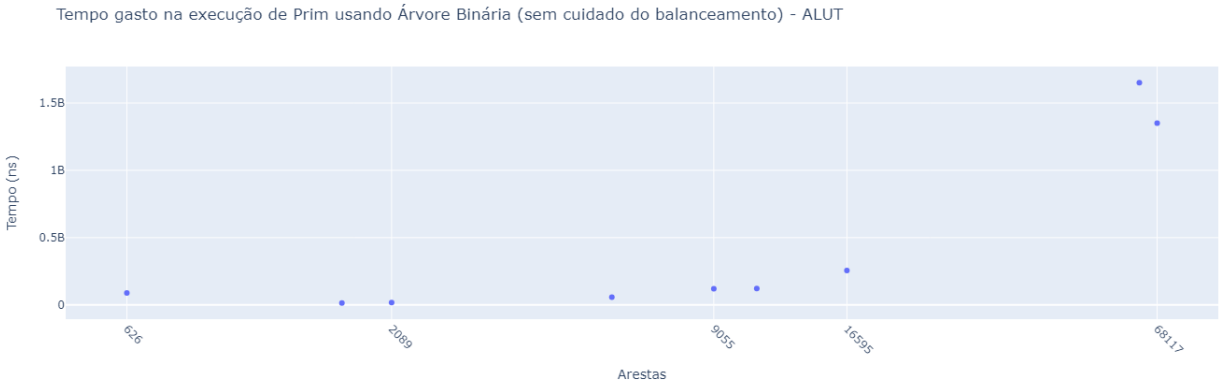


Figura 34: Arestas - Tempo na execução de Prim com Árvore Binária (sem cuidado do balanceamento) - ALUT

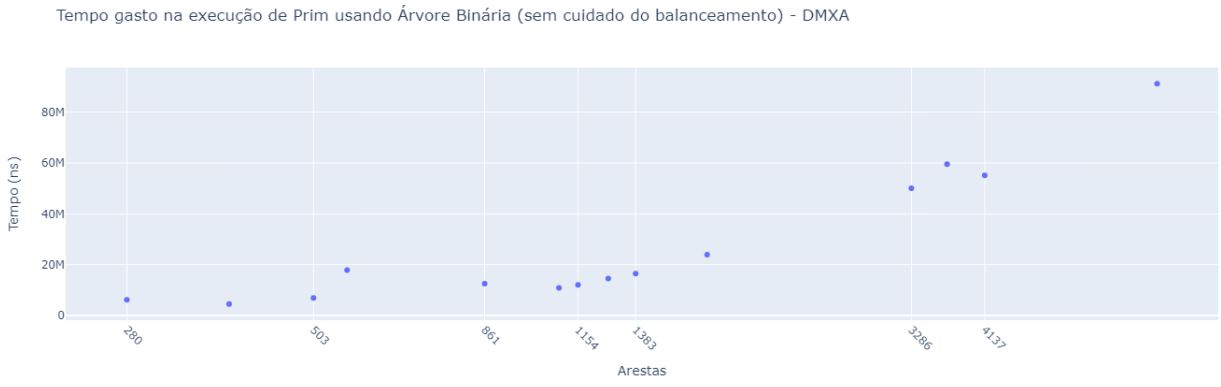


Figura 35: Arestas - Tempo na execução de Prim com Árvore Binária (sem cuidado do balanceamento) - DMXA

Vale notar que para verificar a consistência entre as execuções dos *inputs* "ALUE"e "ALUT"(que só executaram uma vez no período de 5 segundos), cada gráfico destes grupos foi plotado a partir de execuções diferentes (e os resultados de tempo mostraram-se consistentes). Nos gráficos acima, viu-se um aumento mais ou menos

linear, O(n), do consumo de tempo com o aumento do número de n arestas (e n vértices, vide gráficos abaixo).

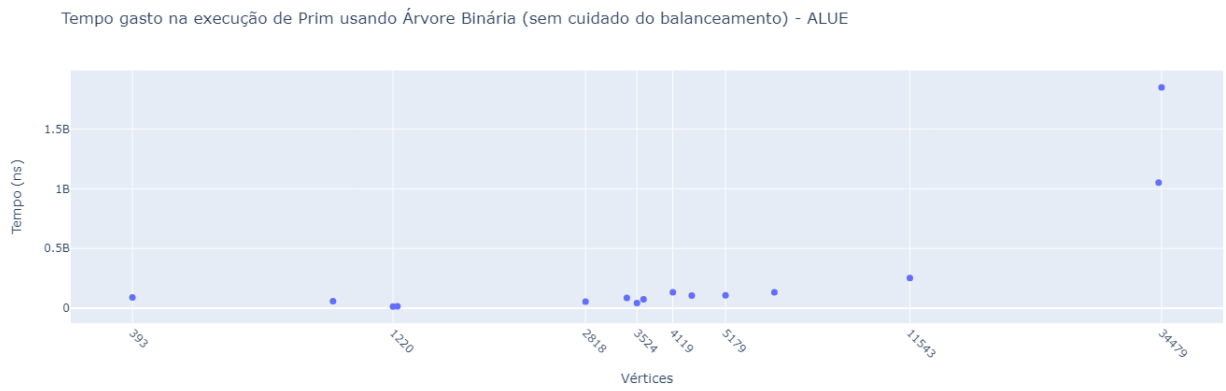


Figura 36: Vértices - Tempo na execução de Prim com Árvore Binária (sem cuidado do balanceamento) - ALUE

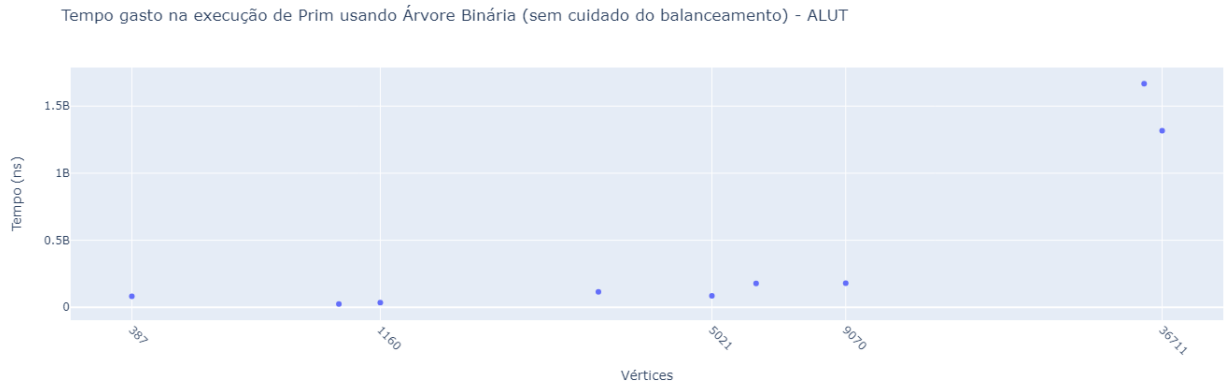


Figura 37: Vértices - Tempo na execução de Prim com Árvore Binária (sem cuidado do balanceamento) - ALUT

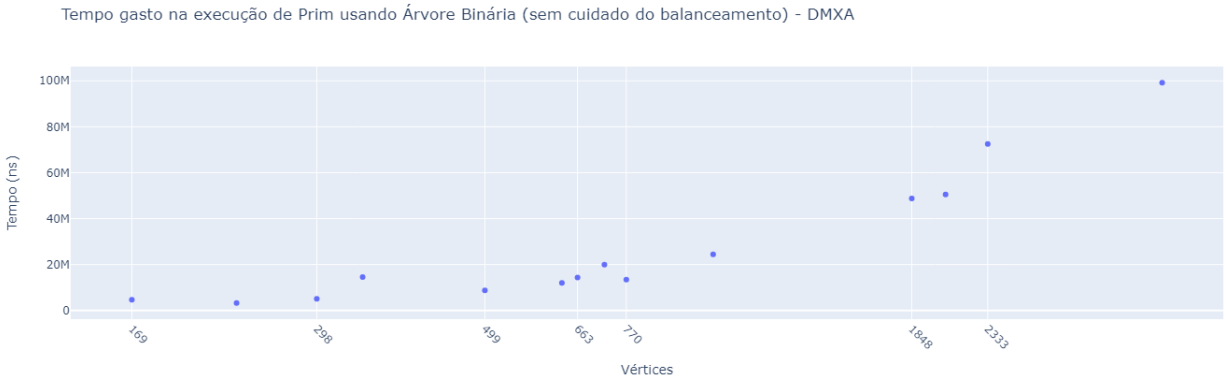


Figura 38: Vértices - Tempo na execução de Prim com Árvore Binária (sem cuidado do balanceamento) - DMXA

3.1.2 Árvore AVL

A árvore AVL é um tipo de árvore binária de busca. Ela foi nomeada em homenagem aos seus criadores, Adelson-Velsky e Landis. As árvores AVL possuem a propriedade de autobalanceamento dinâmico, além de todas as outras propriedades mostradas pelas árvores binárias de busca. Algumas das propriedades dessa estrutura de dados, são:

1. **Autobalanceamento dinâmico**: A diferença entre as profundidades das camadas do lado direito e esquerdo não pode ser maior que um. Essa diferença é chamada de fator de balanço.

2. **Inserção**: A inserção em uma árvore AVL é similar à inserção em uma árvore binária de busca. Depois de inserir um elemento, no entanto, é preciso ajustar as propriedades da AVL utilizando rotações para esquerda ou para a direita.

3. **Rotações**: Nas árvores AVL, após cada operação, como inserção e exclusão, o fator de balanceamento de cada nó precisa ser verificado. Se cada nó satisfizer a condição do fator de balanceamento, a operação pode ser concluída. Caso contrário, a árvore precisa ser rebalanceada utilizando as operações de rotação.

4. **Complexidade**: As árvores AVL têm uma complexidade de tempo de busca, inserção e exclusão, no pior dos casos, de $O(\log V)$, onde V é o número de nós na árvore. A complexidade de espaço de pior caso é $O(V)$.

Tendo como base a complexidade das operações feitas na árvore AVL é possível deduzir a complexidade do algoritmo de Prim construído com uso de uma fila de prioridade Q implementada usando uma árvore AVL. Essa análise é feita a seguir:

1. **Inserção na Fila de Prioridade**: Inicialmente, todos os vértices são inseridos na fila de prioridade. Como sabemos, a inserção em uma árvore AVL tem uma complexidade de tempo de $O(\log V)$. Portanto, para V vértices, a complexidade de tempo seria $O(V \log V)$.

2. **Extração do Mínimo da Fila de Prioridade**: Em cada iteração do algoritmo, extraímos o vértice com o menor custo da fila de prioridade. A extração do mínimo em uma árvore AVL também tem uma complexidade de tempo de $O(\log V)$. Como fazemos isso para cada aresta no grafo, a complexidade total para essa operação seria $O(E \log V)$.

3. **Atualização da Fila de Prioridade**: Após a extração do vértice mínimo, atualizamos os custos dos vértices adjacentes na fila de prioridade. Novamente, como cada atualização pode ser feita em tempo $O(\log V)$ e fazemos isso para cada aresta no grafo, a complexidade total para essa operação seria $O(E \log V)$.

Portanto, a complexidade total do algoritmo seria dominada pela maior das três operações acima, que é $O(E \log V)$.

Na Figura 39 é mostrada a implementação do algoritmo de Prim em Python através de uma função em que a estrutura da fila de prioridades é uma árvore AVL. A árvore AVL é implementada a partir da classe AVL_Tree() da biblioteca pyavl3.

A função que implementa o algoritmo de Prim foi executada tomando como entradas os grafos das bases "ALUE", "ALUT"e "DMXA"respeitando o tempo mínimo de 5 segundos de execução por base. Ao fim da execução considerando todos os grafos de cada base, foram colhidas as medições do tempo de CPU e memória gastos, tirada a média desses valores considerando o número de iterações decorridos e fixados os resultados para posterior análises a fim de comparar o resultado teórico esperado com o resultado obtido.

A Tabela da Figura 40 exibe os tempos medidos em segundos para a implementação do algoritmo de Prim usando AVL, sendo usada a base "ALUE". Na Tabela os grafos de entrada foram ordenados de modo crescente segundo sua quantidade de vértices, é possível assim notar que quanto maior a quantidade de vértices, mais tempo para o algoritmo terminar sua execução.

De modo análogo, a Tabela da Figura 41 mostra os dados de tempo colhidos na execução do algoritmo de Prim usando árvore AVL tendo como entradas os gráficos da base "ALUT". A organização quanto aos grafos

```python
def prim(graph, start_node):  # Define a função prim
    mst = []  # Inicializa a árvore geradora mínima como uma lista vazia
    visited = set([start_node])  # Inicializa o conjunto de nós visitados com o nó inicial
    avl_tree = pyavl3.AVLTree()  # Cria uma árvore AVL vazia

    # Cria uma lista de arestas do nó inicial e seus custos
    edges = [
        (cost, start_node, to)
        for to, cost in graph[start_node].items()
    ]

    # Insere todas as arestas na árvore AVL com o custo como chave
    for edge in edges:
        avl_tree[edge] = edge[0]

    # Continua enquanto houver arestas na árvore AVL
    while len(avl_tree) > 0:
        # Obtém e remove a aresta de menor custo da árvore AVL
        cost, frm, to = avl_tree._get_min(avl_tree.root).key
        avl_tree.pop((cost, frm, to))

        # Se o nó 'to' ainda não foi visitado
        if to not in visited:
            visited.add(to)  # Adiciona o nó 'to' ao conjunto de nós visitados
            mst.append((frm, to, cost))  # Adiciona a aresta à árvore geradora mínima

            # Para cada nó adjacente ao nó 'to'
            for to_next, cost2 in graph[to].items():
                # Se o nó adjacente ainda não foi visitado
                if to_next not in visited:
                    # Adiciona a aresta à árvore AVL
                    avl_tree[(cost2, to, to_next)] = cost2

    return mst # Retorna a árvore geradora mínima
```

Figura 39: Algoritmo de Prim implementado com AVL

de entrada também se dá de forma crescente considerando a respectiva quantidade de vértices no grafo. Após uma análise desses dados é notório que o tempo de execução cresce conforme o número de vértices do grafo.

Por fim, na Tabela da Figura 42 encontram-se os dados de tempos medidos a partir das execuções do algoritmo de Prim com AVL em que as entradas usadas foram os grafos da base "DMXA". Assim como nas tabelas anteriores, as instâncias nomeadas dessa tabela estão organizadas de modo crescente relativo ao número de vértices do grafo e de modo similar é observado o comportamento do tempo de execução que cresce proporcional ao número de vértices do respectivo grafo.

nome das instâncias	complexidade teórica	tempo(s)	razão	tempos maiores	tempos menores
alue6457	52891.03182736107	0.489804105	0.0006878475798401095	0.97960821	0.2449020525
alue7229	14557.98597923681	0.127618695	0.0008551073296241588	0.25523739	0.0638093475
alue2105	19049.45237349256	0.134520762	0.0007423015966115537	0.269041524	0.067260381
alue2087	20263.399187927418	0.147872221	0.0007713041133288142	0.295744442	0.0739361105
alue6951	50643.75460579228	0.335085499	0.0008690275136416399	0.670170998	0.1675427495
alue6179	61093.17392357108	0.434915282	0.0009777366763720839	0.869830564	0.217457641
alue5067	65513.4700415975	0.464403122	0.0009841836609250088	0.928806244	0.232201561
alue3146	69396.01387798847	0.528671785	0.0010651050383756642	1.05734357	0.2643358925
alue6735	80406.09303917384	0.560512313	0.0010051785849404318	1.121024626	0.2802561565
alue5623	84135.07563258606	1.126807815	0.0019695107192555725	2.25361563	0.5634039075
alue5345	100743.50828916338	1.063885425	0.0016076797876648926	2.12777085	0.5319427125
alue7066	132190.65076856405	1.822354667	0.0022042896040139247	3.644709334	0.9111773335
alue5901	248694.39032866867	1.94379828	0.0014233563481482814	3.88759656	0.97189914
alue7065	825642.0964206395	6.104014983	0.0016757016148689745	12.208029966	3.0520074915
alue7080	836484.9417267024	5.846744721	0.0015881086081966527	11.693489442	2.9233723605

Figura 40: Tabela com dados de tempo e complexidade para a base ALUE

nome das instâncias	complexidade teórica	tempo(s)	razão	tempos maiores	tempos menores
alut2764	5381.214787346401	0.088876498	0.0012204460729546153	0.177752996	0.044438249
alut0805	16519.8550451404	0.149011582	0.0008869032852109821	0.298023164	0.074505791
alut0787	21265.8300890412	0.21947464	0.0010695222033335354	0.43894928	0.10973732
alut1181	65869.88926516833	0.6006093	0.0012206653544263196	1.2012186	0.30030465
alut2566	111319.98783254251	1.582749239	0.002148861150414269	3.165498478	0.7913746195
alut2010	138469.26264689502	1.267798381	0.0014479382705784014	2.535596762	0.6338991905
alut2288	218172.5870337266	1.897966637	0.001503606664298515	3.795933274	0.9489833185
alut2610	945320.5098099541	7.860923753	0.0018832679197013872	15.721847506	3.9304618765
alut2625	1032921.0651407773	9.342545866	0.0020797989178820415	18.685091732	4.671272933

Figura 41: Tabela com dados de tempo e complexidade para a base ALUT

nome das instâncias	complexidade teórica	tempo(s)	razão	tempos maiores	tempos menores
dmxa0628	2072.2462421590117	1.717385484	0.04539343897394688	3.434770968	0.858692742
dmxa0296	3035.5758518365524	0.088022059	0.0017933208725692787	0.176044118	0.0440110295
dmxa1109	4707.934204290603	0.045483192	0.0006852636651096121	0.090966384	0.022741596
dmxa0848	7717.053460595381	0.526618416	0.005482027987344293	1.053236832	0.263309208
dmxa0903	10113.20967326851	12.080104444	0.10339525589885544	24.160208888	6.040052222
dmxa0734	10816.286279369926	1.457326236	0.01183650100397729	2.914652472	0.728663118
dmxa0454	35658.847350093165	0.399319859	0.0011187215146444448	0.798639718	0.1996599295
dmxa0368	40441.176522587964	0.038559363	0.0001153991545372053 5	0.077118726	0.0192796815
dmxa1010	85009.11944510063	13.667343998	0.02299613263248804	27.334687996	6.833671999

Figura 42: Tabela com dados de tempo e complexidade para a base DMXA

3.1.3 Heap de Fibonacci

Um heap de Fibonacci é uma estrutura de dados que consiste em uma coleção de árvores que seguem a propriedade de heap mínimo ou máximo. Algumas das principais propriedades dessa estrutura, são:

1. **Estrutura das Árvores**: Em um heap de Fibonacci, as árvores podem ter qualquer forma.
2. **Propriedades do Heap**: As duas propriedades principais são a propriedade de heap mínimo e a propriedade de heap máximo.
3. **Lista Circular Duplamente Ligada**: Todas as raízes das árvores são conectadas usando uma lista circular duplamente ligada.
4. **Operações Eficientes**: O heap de Fibonacci tem operações mais eficientes do que aquelas suportadas pelos heaps binomial e binário. A ideia principal é executar as operações de forma "preguiçosa".
5. **Nós Marcados**: Consiste em um conjunto de nós marcados (operação Decrease key).
6. **Representação na Memória**: A representação na memória dos nós em um heap de Fibonacci é tal que os nós filhos de um nó pai estão conectados entre si através de uma lista circular duplamente ligada.
7. **Complexidade das Operações**:
   - **MAKE-HEAP**: $\Theta(1)$
   - **INSERT**: $\Theta(1)$
   - **MINIMUM**: $\Theta(1)$
   - **UNION**: $\Theta(1)$
   - **DECREASE-KEY**: $\Theta(1)$
   - **EXTRACT-MIN**: $O(\log n)$
   - **DELETE**: $O(\log n)$

Com base nas propriedades enumeradas acima com destaque às complexidades das operações na Heap de Fibonacci, tem-se a seguir a análise da complexidade total do algoritmo de Prim usando fila de prioridade Q implementada usando Heap de Fibonacci:

1. **Inserção na Fila de Prioridade**: Inicialmente, todos os vértices são inseridos na fila de prioridade. A inserção em um heap de Fibonacci pode ser feita em tempo constante, ou seja, $O(1)$. Portanto, para V vértices, a complexidade de tempo seria $O(V)$.
2. **Extração do Mínimo da Fila de Prioridade**: Em cada iteração do algoritmo, extraímos o vértice com o menor custo da fila de prioridade. A extração do mínimo em um heap de Fibonacci tem uma complexidade de tempo amortizado de $O(\log V)$. Como fazemos isso para cada vértice no grafo, a complexidade total para essa operação seria $O(V \log V)$.
3. **Atualização da Fila de Prioridade**: Após a extração do vértice mínimo, atualizamos os custos dos vértices adjacentes na fila de prioridade. A operação **DECREASE-KEY**, que é usada para atualizar a chave de um vértice específico em um heap de Fibonacci, pode ser feita em tempo constante amortizado, ou seja, $O(1)$. Como fazemos isso para cada aresta no grafo, a complexidade total para essa operação seria $O(E)$.

Portanto, a complexidade total do algoritmo seria a soma das três operações acima, que é $O(E + V \log V)$.

Na Figura 43 é mostrada a implementação do algoritmo de Prim em Python através de uma função em que a estrutura da fila de prioridades é uma Heap de Fibonacci. A Heap de Fibonacci é implementada a partir da classe makefheap() da biblioteca fibheap.

A função que implementa o algoritmo de Prim foi executada tomando como entradas os grafos das bases "ALUE", "ALUT"e "DMXA"respeitando o tempo mínimo de 5 segundos de execução por base. Ao fim da

```python
def prim(graph, start_node):
    mst = []  # Inicializa a árvore geradora mínima como uma lista vazia
    visited = set([start_node])  # Inicializa o conjunto de nós visitados com o nó inicial
    fib_heap = makefheap()  # Cria um heap de Fibonacci vazio

    # Cria uma lista de arestas do nó inicial e seus custos
    edges = [
        (cost, start_node, to)
        for to, cost in graph[start_node].items()
    ]

    # Insere todas as arestas no heap de Fibonacci com o custo como chave
    for edge in edges:
        fheappush(fib_heap, edge)

    # Continua enquanto houver arestas no heap de Fibonacci
    while fib_heap.min is not None:
        # Obtém e remove a aresta de menor custo do heap de Fibonacci
        cost, frm, to = fheappop(fib_heap)

        # Se o nó 'to' ainda não foi visitado
        if to not in visited:
            visited.add(to)  # Adiciona o nó 'to' ao conjunto de nós visitados
            mst.append((frm, to, cost))  # Adiciona a aresta à árvore geradora mínima

            # Para cada nó adjacente ao nó 'to'
            for to_next, cost2 in graph[to].items():
                # Se o nó adjacente ainda não foi visitado
                if to_next not in visited:
                    # Adiciona a aresta ao heap de Fibonacci
                    fheappush(fib_heap, (cost2, to, to_next))

    return mst # Retorna a árvore geradora mínima
```

Figura 43: Algoritmo de Prim implementado com Heap de Fibonacci

execução considerando todos os grafos de cada base, foram colhidas as medições do tempo de CPU e memória gastos, tirada a média desses valores considerando o número de iterações decorridos e fixados os resultados para posterior análises a fim de comparar o resultado teórico esperado com o resultado obtido.

A Tabela da Figura 44 exibe os dados de tempo de execução do algoritmo de Prim usando Heap de Fibonacci para implementar a fila de prioridades, nela os grafos descritos são da base "ALUE"e estão dispostos em ordem crescente de acordo com a quantidade de vértices de cada grafo. De uma análise da Tabela é possível concluir que a medida que a quantidade de vértices aumenta, assim também ocorre com o tempo de execução do algoritmo.

Na Figura 45 encontra-se a Tabela que contém as medições de tempo das execuções do algoritmo de Prim usando Heap de Fibonacci em que foram usados como entradas os grafos da base "ALUT". As instâncias foram organizadas na tabela de acordo com a ordem crescente das quantidades de vértices dos grafos, e, assim como observado nos resultados obtidos na base "ALUE", nota-se que a quantidade de tempo acompanha a quantidade de vértices do grafo.

Por fim, a Tabela exibida pela Figura 46 contém as medições de tempo e também uma análise segundo a complexidade teórica esperada para o algoritmo de Prim usando Heap de Fibonacci. Ao comparar os resultados de todas as Tabelas considerando a fila de prioridades implementada a partir de Heap de Fibonacci com as Tabelas desse algoritmo com uso de árvores AVL é notória a maior eficiência dado o uso de Heap de Fibonacci,

nome das instâncias	complexidade teórica	tempo(s)	razão	tempos maiores	tempos menores
alue6457	9524.025502387633	0.107903667	1.1329628104518305e-05	0.215807334	0.0539518335
alue7229	10757.9259297711	0.049110011	4.56500735556235e-06	0.098220022	0.0245550055
alue2105	14366.251827589304	0.027907863	1.9425987609659644e-06	0.055815726	0.0139539315
alue2087	14760.27883804247	0.02963842	2.0079851014474936e-06	0.05927684	0.01481921
alue6951	36714.56471580055	0.071884711	1.9579344479893443e-06	0.143769422	0.0359423555
alue6179	44730.7791042167	0.091443322	2.0443042538326763e-06	0.182886644	0.045721661
alue5067	47083.28568823554	0.093711697	1.990338941519862e-06	0.187423394	0.0468558485
alue3146	48743.41579853232	0.100351722	2.0587749207970285e-06	0.200703444	0.050175861
alue6735	56157.274974366344	0.127684979	2.273703256760292e-06	0.255369958	0.0638424895
alue5623	61168.62240255475	0.136140757	2.22566328376743e-06	0.272281514	0.0680703785
alue5345	72065.87316957467	0.158560288	2.200213235839099e-06	0.317120576	0.079280144
alue7066	91445.1151877418	0.257564483	2.816601876122154e-06	0.515128966	0.1287822415
alue5901	174198.67537922962	0.460631186	2.644286387351731e-06	0.921262372	0.230315593
alue7065	567410.2604937381	1.355177361	2.3883554023516916e-06	2.710354722	0.6775886805
alue7080	575210.8037228344	1.365983508	2.374752871745781e-06	2.731967016	0.682991754

Figura 44: Tabela com dados de tempo e complexidade para a base ALUE

nome das instâncias	complexidade teórica	tempo(s)	razão	tempos maiores	tempos menores
alut2764	3952.725435627887	0.05603304	1.4175798676767743e-05	0.11206608	0.02801652
alut0805	11244.739479955357	0.050916437	4.52802282265077e-06	0.101832874	0.0254582185
alut0787	13897.694544417323	0.097225638	6.995810541760349e-06	0.194451276	0.048612819
alut1181	40878.373837234656	0.215079143	5.261440776885602e-06	0.430158286	0.1075395715
alut2566	70781.96398754235	0.192289444	2.716644653062227e-06	0.384578888	0.096144722
alut2010	87772.09156267798	0.548045833	6.2439646047245086e-06	1.096091666	0.2740229165
alut2288	135837.26359722207	0.540964795	3.9824476780100786e-06	1.08192959	0.2704823975
alut2610	572993.5121476576	2.233185105	3.897400332910784e-06	4.46637021	1.1165925525
alut2625	624799.8430844441	2.415203045	3.865562822610339e-06	4.83040609	1.2076015225

Figura 45: Tabela com dados de tempo e complexidade para a base ALUT

corroborando o que se esperava na teoria.

nome das instâncias	complexidade teórica	tempo(s)	razão	tempos maiores	tempos menores
dmxa0628	1530.7486247316892	0.006705215833333333	4.3803507153296505e-06	0.013410431666666667	0.0033526079166666667
dmxa0296	2218.3553717044474	0.0091335785	4.117274723653641e-06	0.018267157	0.00456678925
dmxa1109	3447.768214797275	0.014600569166666666	4.234788494192659e-06	0.02920113833333333	0.007300284583333333
dmxa0848	5333.485106663293	0.0187491635	3.515368117664015e-06	0.037498327	0.00937458175
dmxa0903	6966.989432847929	0.0222174575	3.188960987259323e-06	0.044434915	0.01110872875
dmxa0734	7368.209534854645	0.024126001	3.274336985922312e-06	0.048252002	0.0120630005
dmxa0454	23340.032228536875	0.07609032416666667	3.260077939122733e-06	0.15218064833333333	0.038045162083333334
dmxa0368	26228.88679850526	0.08720704633333333	3.324847409776579e-06	0.17441409266666666	0.043603523166666665
dmxa1010	54743.24518146255	0.1647174585	3.008909281026282 4e-06	0.329434917	0.08235872925

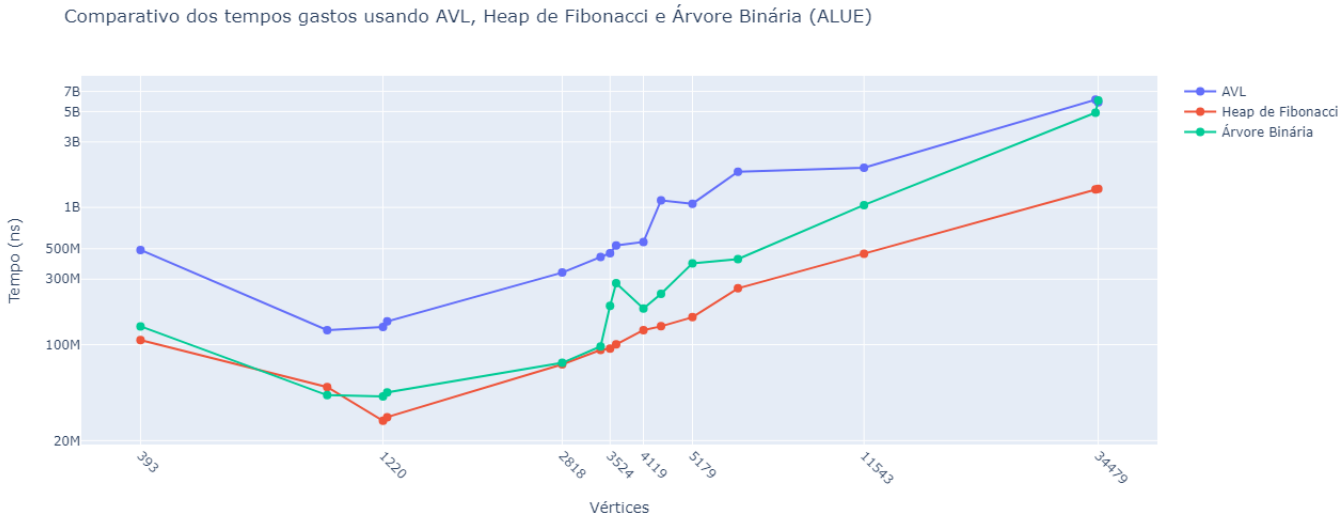
Figura 46: Tabela com dados de tempo e complexidade para a base DMXA

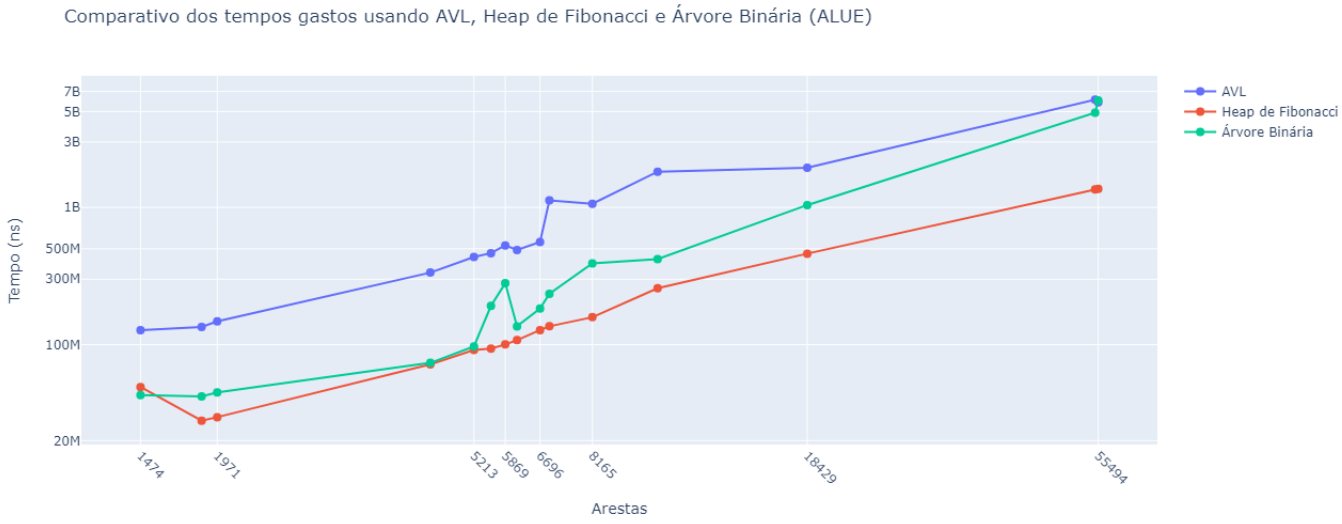
3.1.4 Comparação

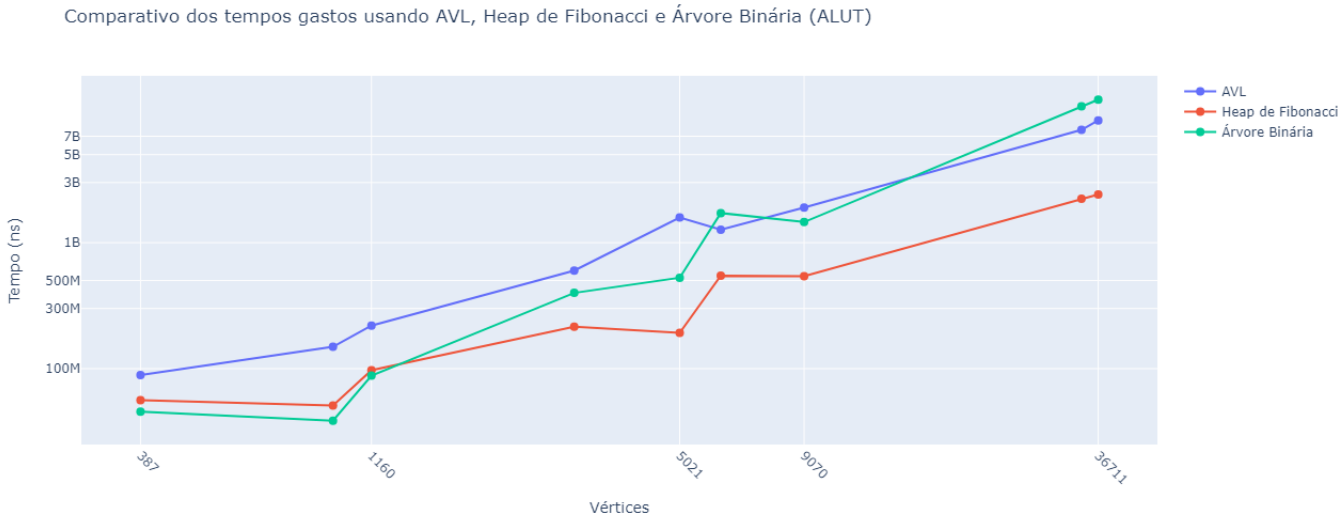
A seguir serão mostrados alguns gráficos que permitem uma avaliação facilitada em termos de comparação do algoritmo de Prim utilizando as diferentes estruturas de dados mostradas nesta secão na implementação da fila de prioridades. A partir da análise dos gráficos é possível concluir que a escolha de uma estrutura de dados adequada para a execução de um algoritmo é muito importante para melhorar sua eficiência, no presente caso de estudo a estrutura usada para armazenar as arestas a serem alocadas para a MST teria que se mostrar eficiente na operação de busca do mínimo valor, visto que essa era uma das operações mais usadas pelo algoritmo.

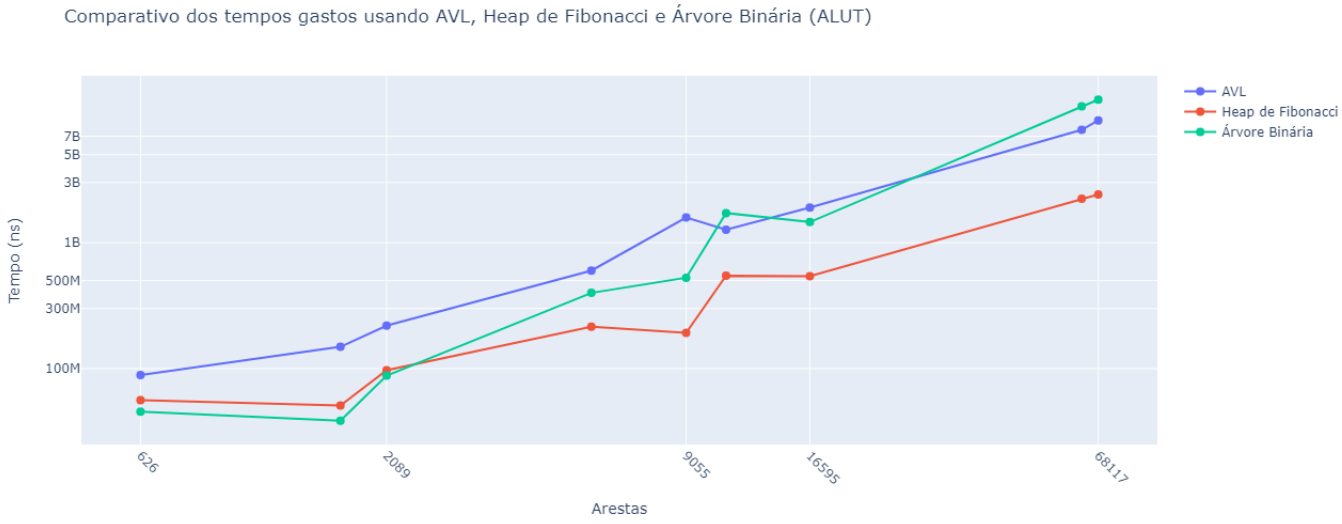
Os gráficos foram plotados levando em conta as três diferentes abordagens discutidas nesta seção considerando as correlações "número de vértices vs tempo"e "número de arestas vs tempo"cada uma para as três bases usadas "ALUE", "ALUT"e "DMXA"sendo que no eixo x encontram-se as quantidades de vértice ou arestas, de forma crescente.

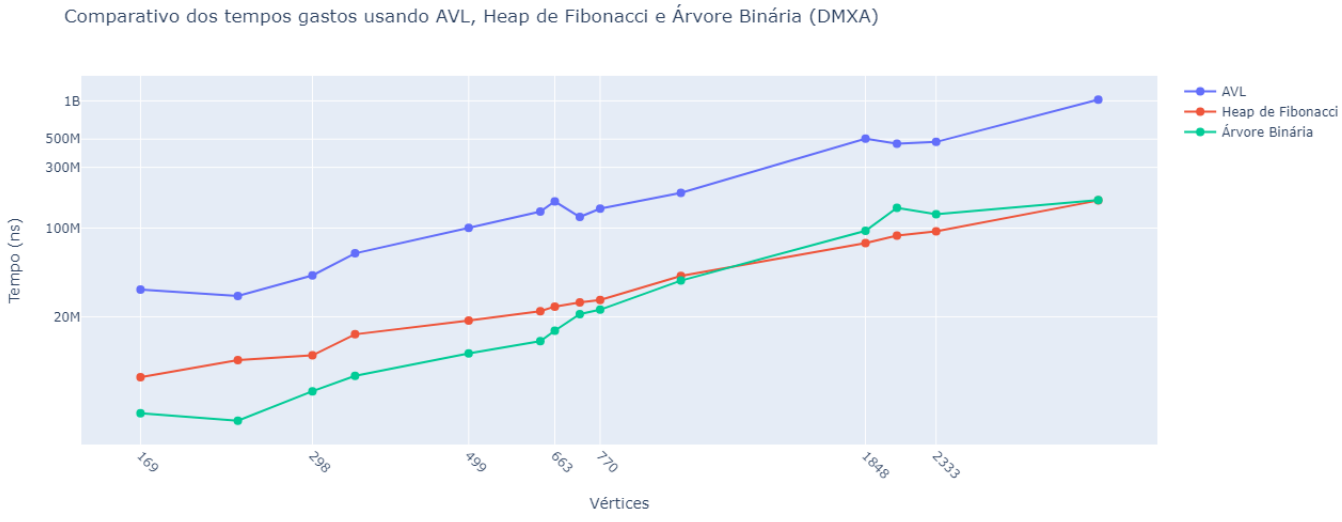
Analisando todos os gráficos de forma ponderada é possível inferir que a abordagem que usa Heap de Fibonacci é a mais eficiente, o que condiz com o esperado. Já entre as abordagens que usam árvore binária de busca e árvore AVL, a maior eficiência observada é na abordagem com árvore binária de busca.

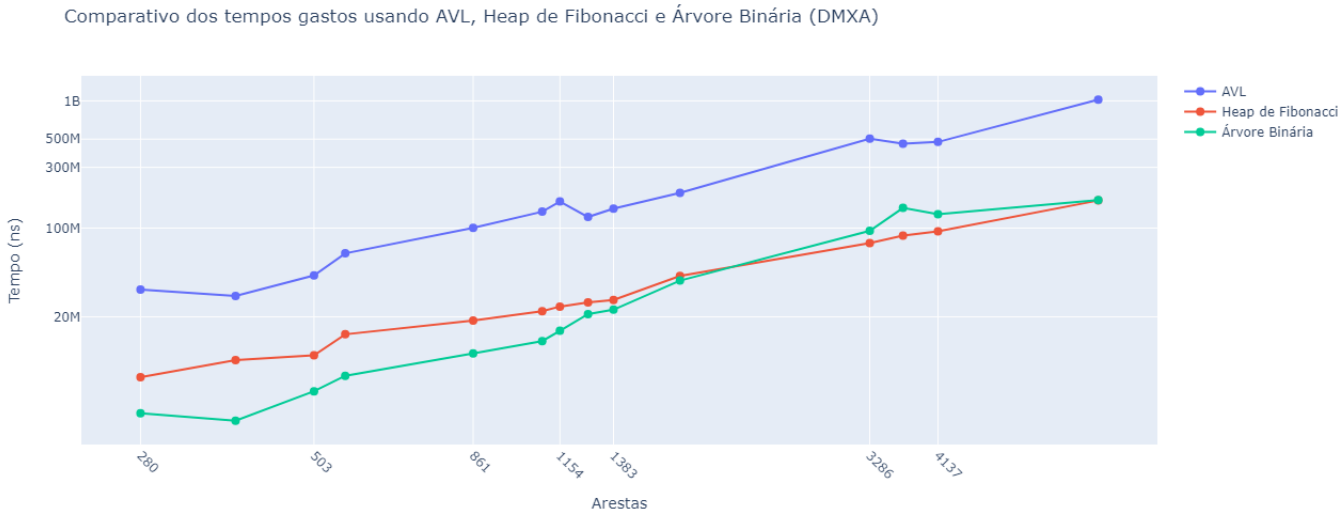



Comparativo dos tempos gastos usando AVL, Heap de Fibonacci e Árvore Binária (ALUE)
Tempo (ns)
7B
5B
3B
1B
500M
300M
100M
20M
1474
1971
5213
5869
6696
8165
18429
55494
Arestas
AVL
Heap de Fibonacci
Árvore Binária


Comparativo dos tempos gastos usando AVL, Heap de Fibonacci e Árvore Binária (ALUT)
Tempo (ns)
7B
5B
3B
1B
500M
300M
100M
387
1160
5021
9070
36711
Vértices
AVL
Heap de Fibonacci
Árvore Binária


Comparativo dos tempos gastos usando AVL, Heap de Fibonacci e Árvore Binária (ALUT)
AVL
Heap de Fibonacci
Árvore Binária
7B
5B
3B
1B
500M
300M
100M
Tempo (ns)
626
2089
9055
16595
68117
Arestas


Comparativo dos tempos gastos usando AVL, Heap de Fibonacci e Árvore Binária (DMXA)
AVL
Heap de Fibonacci
Árvore Binária
1B
500M
300M
100M
20M
Tempo (ns)
169
298
499
663
770
1848
2333
Vértices


Comparativo dos tempos gastos usando AVL, Heap de Fibonacci e Árvore Binária (DMXA)
1B
500M
300M
100M
20M
Tempo (ns)
280
503
861
1154
1383
3286
4137
Arestas
AVL
Heap de Fibonacci
Árvore Binária