Instituto Federal de Educação Ciência e Tecnologia
do Norte de Minas Gerais - IFNMG
Bacharelado em Ciência da Computação

Disciplina de Laboratório de PAA

Lab 5: Caixeiro Viajante.

Thiago Emanuel Silva Antunes Lopes

Sumário

1 Introdução **1**

2 Algoritmos Caixeiro Viajante **2**

2.1 Implementação 2

2.1.1 Algoritmo sem programação dinâmica 2

2.1.2 Algoritmo com programação dinâmica 5

2.2 Gráficos da execução dos algoritmos 8

2.2.1 Algoritmo sem programação dinâmica 8

2.2.2 Algoritmo com programação dinâmica 9

2.3 Análise de complexidade 10

2.3.1 Solução sem programação dinâmica 10

2.3.2 Solução com programação dinâmica 11

2.4 Estimativa de tamanho do vetor para ordenação em 24 horas 12

2.4.1 Solução sem programação dinâmica 12

2.4.2 Solução com programação dinâmica 13

2.4.3 Gráfico para estimativas 14

2.4.4 Gráfico das funções 16

Capítulo 1

Introdução

Este relatório tem o objetivo de detalhar o funcionamento de como obter um ciclo que passa precisamente uma vez em cada vértice e que tenha o menor peso possível, dado um grafo não orientado com pesos positivos nas arestas, analisando também complexidade e tempo de execução com gráficos. A atividade foi proposta pelo Professor Alberto Miranda na disciplina Laboratório de Projeto de analise de algoritmos e realizado pelo discente Thiago Emanuel Silva Antunes Lopes.

Capítulo 2

Algoritmos Caixeiro Viajante

Na atividade foi pedido a implementação de 2 algoritmos para se achar o menor caminho. O primeiro é um algoritmo recursivo de busca exaustiva sem memorização nem programação dinâmica para resolver este problema, e o segundo é um algoritmo com programação dinâmica.

A implementação abaixo foi feita para os dois algoritmos de ordenação, é necessário comentar a chamada de uma função para analisar a outra. Linhas: 123 ou 124.

2.1 Implementação

2.1.1 Algoritmo sem programação dinâmica

```
1  // LAB 5 PAA - THIAGO EMANUEL SILVA ANTUNES LOPES
2  // CAIXEIRO VIAJANTE SEM PROGRAMACAO DINAMICA
3  #define MAX 9999
4
5  int caminhomin(int no, int mask, int n, int** grafo,int maskvisitados)
6  {
7          if(mask==maskvisitados)
8      {
9                  return grafo[no][0];
10         }
11         int customin = MAX;
12
13         for(int i=0; i<n; i++)
14     {
```

```
15                  if((mask&(1<<i))==0)
16          {
17                          int c1 = grafo[no][i] + caminhomin(i, mask
                                |(1<<i),n,grafo,maskvisitados);
18                          customin = min(c1, customin);
19                  }
20          }
21          return customin;
22  }
23
24  // FUNCAO TEMPO //
25  long double getNow()
26  {
27      struct timeval now;
28      long double valor = 1000000;
29
30      gettimeofday(&now,NULL);
31
32      return ((long double)(now.tv_sec*valor)+(long double)(now.tv_usec))
            ;
33  }
34
35  void temp()
36  {
37       long double t=0, ti, tf;
38       int n = 4;
39
40       while(t<60)
41       {
42           int maskvisitados = (1<<n) -1;
43           int **grafo = (int**)malloc(n * sizeof(int*));
44           for(int i = 0; i < n; i++)
45           {
46               grafo[i] = (int*) malloc(n * sizeof(int));
47               for(int j = 0; j < n; j++)
48               {
```

```
49                  if(i==j)
50                  {
51                      grafo[i][j]=0;
52                  }
53                  else if(i>j)
54                  {
55                      grafo[i][j]=grafo[j][i];
56                  }
57                  else
58                  {
59                      grafo[i][j]=rand()%1000+1;
60                  }
61
62              }
63          }
64          ti = getNow();
65
66          caminhomin(0,1,n,grafo,maskvisitados);
67
68          tf = getNow();
69          t=(tf-ti)/1000000;
70              printf("\n(%d, %Lf)\n", n, t);
71
72          for (int i=0; i < n;i++)
73          {
74              free(grafo[i]);
75          }
76          free(grafo);
77          ++n;
78      }
79  }
80  int main()
81  {
82      srand(time(NULL));
83          temp();
84          return 0;
```

```
}
```

2.1.2 Algoritmo com programação dinâmica

```
// LAB 5 PAA - THIAGO EMANUEL SILVA ANTUNES LOPES
// CAIXEIRO VIAJANTE COM PROGRAMACAO DINAMICA
#define MAX 9999

int caminhomin(int no, int mask, int n, int** grafo,int maskvisitados,
    int** memo)
{
        if(mask==maskvisitados)
    {
                return grafo[no][0];
        }
    if(memo[mask][no]!=-1)
    {
           return memo[mask][no];
        }
        int customin = MAX;

        for(int i=0; i<n; i++)
    {
                if((mask&(1<<i))==0)
        {
                        int c1 = grafo[no][i] + caminhomin(i, mask
                           |(1<<i),n,grafo,maskvisitados,memo);
                        customin = min(c1, customin);
                }
        }
        return memo[mask][no]=customin;
}


// FUNCAO TEMPO //
long double getNow()
{
```

```
32      struct timeval now;
33      long double valor = 1000000;
34
35      gettimeofday(&now,NULL);
36
37      return ((long double)(now.tv_sec*valor)+(long double)(now.tv_usec))
           ;
38  }
39
40  void temp()
41  {
42       long double t=0, ti, tf;
43       int n = 4;
44
45       while(t<60)
46       {
47            int maskvisitados = (1<<n) -1;
48            int **grafo = (int**)malloc(n * sizeof(int*));
49            int **memo = (int**)malloc((1<<n) * sizeof(int*));
50
51            for(int i=0; i<n; i++)
52            {
53                 grafo[i] = (int*) malloc(n * sizeof(int));
54                 for(int j = 0; j < n; j++)
55                 {
56                      if(i==j)
57                           grafo[i][j]=0;
58                      else if(i>j)
59                           grafo[i][j]=grafo[j][i];
60                      else
61                           grafo[i][j]=rand()%1000+1;
62                 }
63            }
64            for(int i=0; i<(1<<n); i++)
65            {
66                 memo[i] = (int*) malloc(n * sizeof(int));
```

```
67              for(int j = 0; j < n; j++)
68              {
69                  memo[i][j]=-1;
70              }
71          }
72
73          ti = getNow();
74
75          caminhomin(0,1,n,grafo,maskvisitados,memo);
76
77          tf = getNow();
78          t=(tf-ti)/1000000;
79              printf("\n(%d, %Lf)\n", n, t);
80
81
82          for (int i=0; i < n;i++)
83          {
84              free(grafo[i]);
85          }
86          free(grafo);
87
88          for (int i=0; i < (1<<n);i++)
89          {
90              free(memo[i]);
91          }
92          free(memo);
93          n++;
94      }
95  }
96
97  int main()
98  {
99      srand(time(NULL));
100         temp();
101         return 0;
102 }
```

2.2 Gráficos da execução dos algoritmos

2.2.1 Algoritmo sem programação dinâmica

Tabela Merge:

Quantidade de elementos	Tempo (s)
4	0.000001
5	0.000003
6	0.000013
7	0.000062
8	0.000416
9	0.003052
10	0.020322
11	0.220502
12	2.474771
13	31.492111
14	430.226985

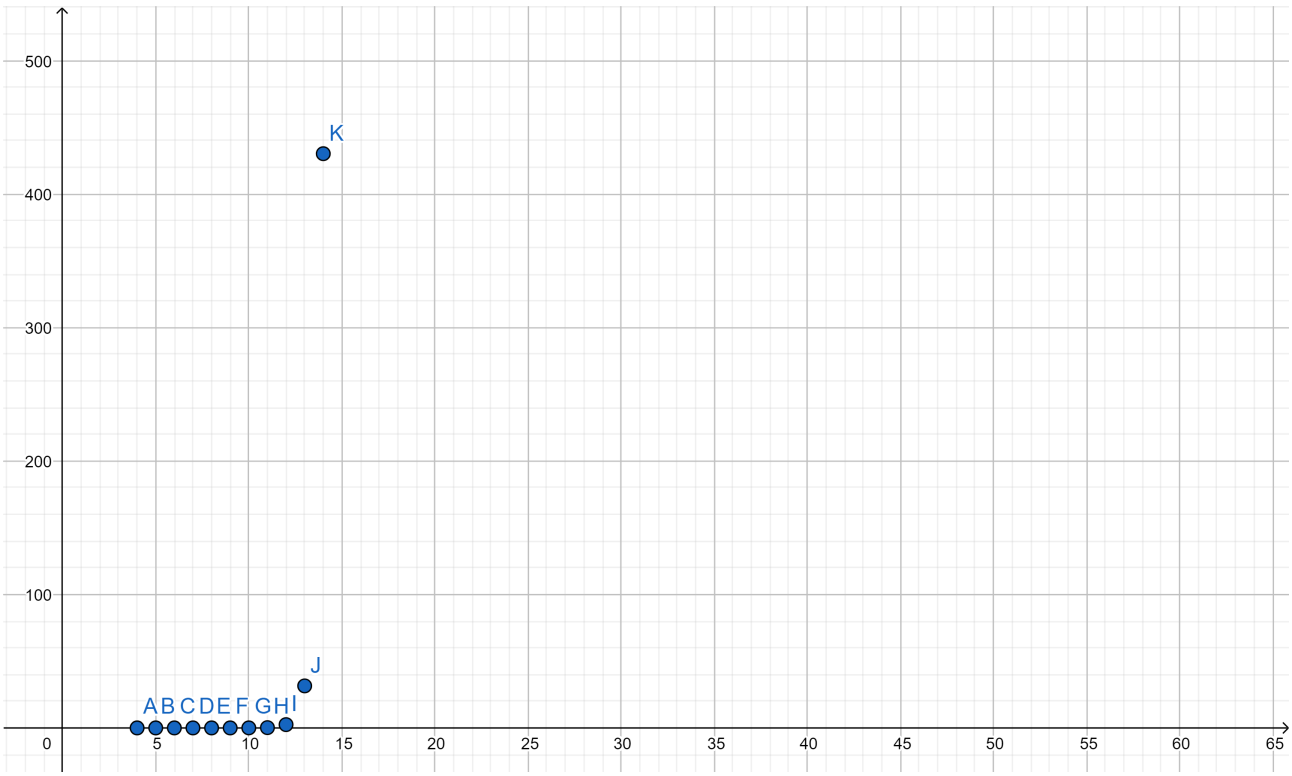

Figura 2.1: (Tamanho, Tempo(s))

2.2.2 Algoritmo com programação dinâmica

Tabela Heap:

Quantidade de elementos	Tempo (s)
4	0.000002
5	0.000002
6	0.000007
7	0.000019
8	0.000054
9	0.000131
10	0.000326
11	0.000784
12	0.001826
13	0.003342
14	0.008855
15	0.017329
16	0.063524
17	0.104944
18	0.269487
19	0.737855
20	1.669040
21	3.911876
22	9.146617
23	21.895304
24	52.333952
25	119.050925

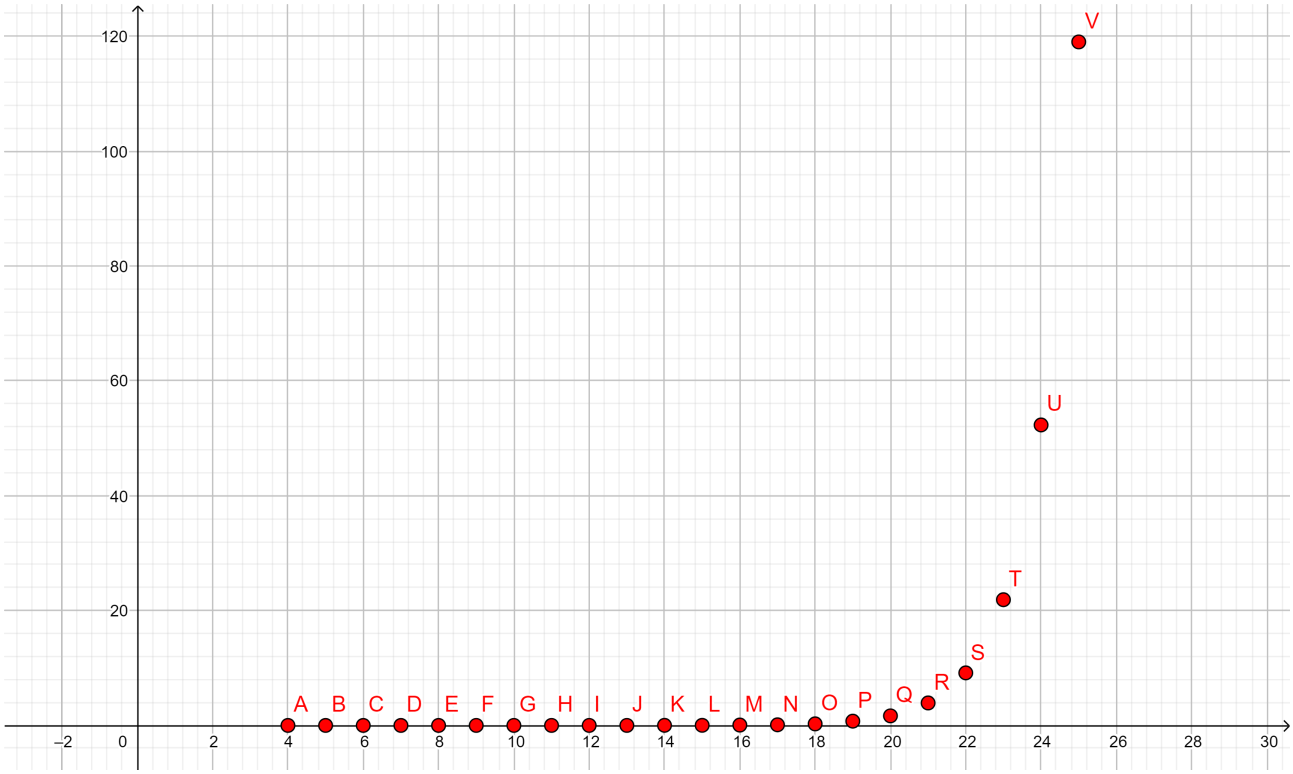

Figura 2.2: (Tamanho, Tempo(s))

2.3 Análise de complexidade

No problema do caixeiro viajante é dado um conjunto de cidades e distância entre os pares de cidades, o objetivo é encontrar a rota mínima que visita cada cidade exatamente uma vez e retorna ao ponto de partida.

No problema do ciclo hamiltoniano é descobrir se existe um caminho que visita todas as cidades exatamente uma vez. No problema do caixeiro já sabemos que o caminho hamiltoniano existe e existem muitos caminhos desse tipo, o problema é encontrar um caminho hamiltoniano que tenho peso mínimo.

2.3.1 Solução sem programação dinâmica

Nessa solução como todas as cidades são interligadas, e considerando que independente do ponto de saída o caminho será o mesmo, vai ser gerado todas as possibilidades em (n-1)!, depois de gerar todos caminhos é calculado o custo de cada permutação e retornar a de custo mínimo. Portanto a sua complexidade é fatorial com a analise assintótica, sendo $\Theta(n!)$.

Provando:

T(n) = n*T(n-1) + Θ(n)

Com os pesos dos filhos, temos:

R(n) = n*R(n-1) + C

Para explicar isso vamos pensar em uma arvore, que a partir do segundo nível, temos n elementos, sendo q cada filho tem n*(n-1).

Temos:

$\sum_{i=1}^{n-1} \frac{n!}{i!}$

$= n! \sum_{i=1}^{n-1} \frac{1}{i!}$

Como a razão entre um fatorial e outro é menor do que 1, o resultado é uma constante.

Ou seja, por fim a complexidade é Θ(n!).

2.3.2 Solução com programação dinâmica

Na programação dinâmica utilizamos a memorização de cálculos já feitos de subproblemas, com o objetivo de reduzir a utilização de memoria e cálculos de problemas já resolvidos antes, fazendo com que o algoritmo seja mais eficiente.

Dado um conjunto de vértice ser 1, 2, 3, ... n. Considerando que o ponto inicial é o vértice um para o destino i. Para cada outro vértice i, diferente de 1, encontramos o custo mínimo da origem até o destino, passando por todos os vértices exatamente uma vez, o custo desse caminho é o custo (i), e o custo do ciclo será custo(i) + dist(i,1). E por fim retornamos, o mínimo de todos valores (custo(i) + dist(i,1)).

Para calcular o custo(i) com a programação dinâmica, é necessário identificar os subproblemas, e a partir da solução desses subproblemas resolver o problema. Vamos definir como o termo Custo(S,i) como o custo do caminho de custo mínimo visitado em cada vértice no conjunto S exatamente uma vez e terminando em i. Começamos com todos os subconjuntos de tamanho 2 e calculamos Custo(S,i) para todos os subconjuntos de S de tamanho 3 e assim sucessivamente. Para o conjunto de tamanho n, temos n possibilidades de um vértice de começo e 2^n possibilidades para um novo subgrafo. Dessa forma existe no máximo $O(n * 2^n)$ subproblemas, e cada um desses subproblemas é resolvido em tempo linear. Portanto,a sua execução total é igual a $O(n^2 * 2^n)$.

2.4 Estimativa de tamanho do vetor para ordenação em 24 horas

Para fazer a estimativa de tempo de execução do algoritmo para qualquer tamanho é necessário se calcular uma constante que é multiplicada pela função do algoritmo. O exercício proposto foi para fazer a estimativa do tamanho do grafo que o menor caminho seja encontrado em 24 horas para os dois algoritmos.

2.4.1 Solução sem programação dinâmica

A partir dos dados tabelados, vamos calcular sua constante. Para isso, vamos utilizar o tempo para ordenar 3 vetores, e fazer a média das constantes.
Para vetor de tamanho 13:

$$Const * 13! = 31.492111$$
$$Const = 5,057331 * 10^{-9}$$

Para vetor de tamanho 12:

$$Const * 12! = 2.474771$$
$$Const = 5,166519 * 10^{-9}$$

Para vetor de tamanho 11:

$$Const * 11! = 0.220502$$
$$Const = 5,524040 * 10^{-9}$$

Fazendo a média:

$$Const = \frac{5,057331 * 10^{-9} + 5,166519 * 10^{-9} + 5,524040 * 10^{-9}}{3}$$
$$Const = 5,249296 * 10^{-9}$$

Com a constante calculada, agora vamos fazer a estimativa do tamanho do vetor que essa operação leve 24 horas.
Como foi utilizado o tempo em segundos, vamos transformar as 24 horas em segundos. Isso dá um total de 86400 segundos.
A equação fica:

$$5,249296 * 10^{-9} * n! = 86400$$
$$n! = 1,645934 * 10^{13}$$

Para encontrar o n que substituindo na equação dê $1,645934$, é feita uma busca binária.
O resultado dessa busca é aproximadamente: 16

2.4.2 Solução com programação dinâmica

Agora a partir dos dados tabelados do heap, vamos calcular sua constante. Para isso, vamos utilizar o tempo, e fazer a média das constantes.
Para vetor de tamanho 22:

$$Const * 2^{22} * 22^2 = 9.146617$$
$$Const = 4,505626 * 10^{-9}$$

Para vetor de tamanho 23 :

$$Const * 2^{23} * 23^2 = 21.895304$$
$$Const = 4,934071 * 10^{-9}$$

Para vetor de tamanho 24:

$$Const * 2^{24} * 24^2 = 53.781858$$
$$Const = 5,565362 * 10^{-9}$$

Fazendo a média:

$$Const = \frac{4,505626 * 10^{-9} + 4,934071 * 10^{-9} + 5,565362 * 10^{-9}}{3}$$
$$Const = 5,001686 * 10^{-9}$$

Com a constante calculada, agora vamos fazer a estimativa do tamanho do vetor que essa operação leve 24 horas.

Como foi utilizado o tempo em segundos, vamos transformar as 24 horas em segundos. Isso dá um total de 86400 segundos.

A equação fica:

$$5,001686 * 10^{-9} * 2^n * n^2 = 86400$$
$$2^n * n^2 = 1,727417 * 10^{13}$$

Para encontrar o n que substituindo na equação dê $1,727417 * 10^{13}$, é feita uma busca binária. O resultado dessa busca é aproximadamente: 34

2.4.3 Gráfico para estimativas

Gráfico de execução dos algorítimos em 24 horas. Algoritmo com programação dinâmica em vermelho e algoritmo sem programação dinâmica em azul.

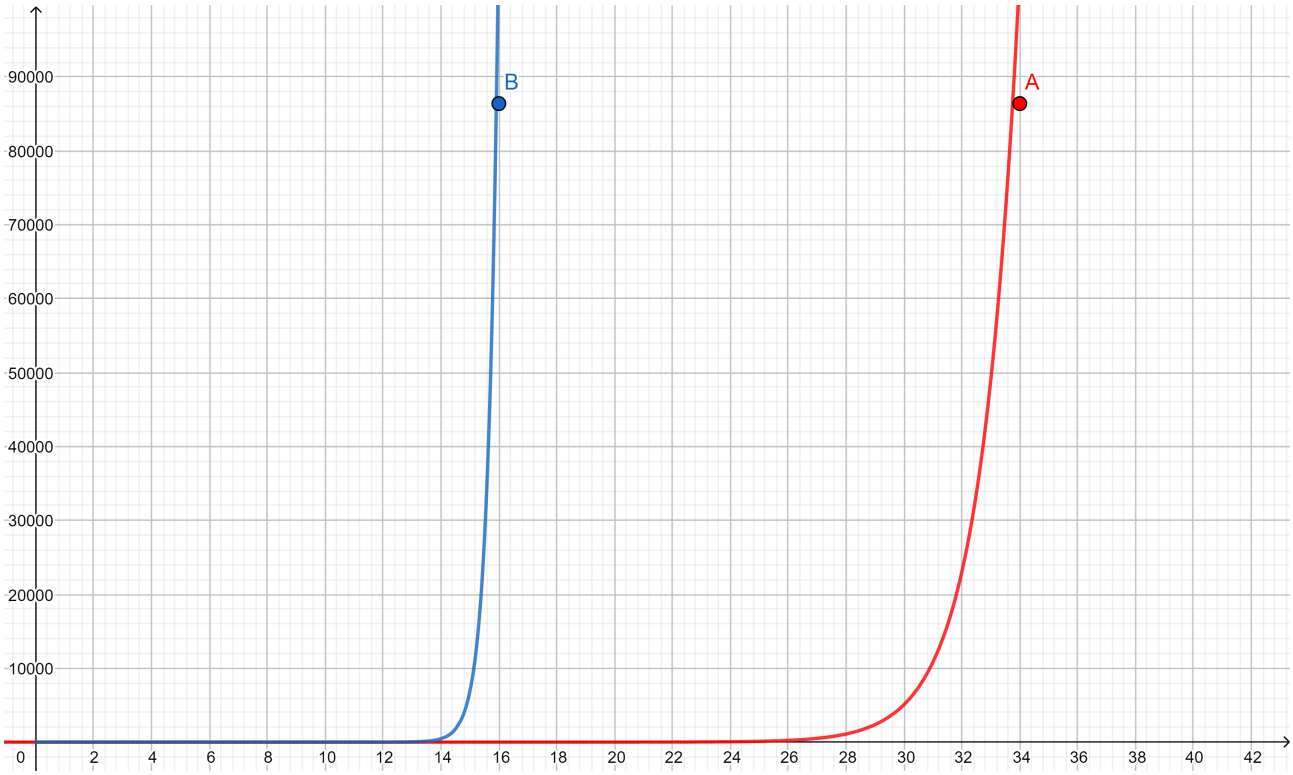

Figura 2.3: (Tamanho, Tempo(s))

2.4.4 Gráfico das funções

Gráfico das funções multiplicadas pelas constantes. Algoritmo com programação dinâmica em vermelho e algoritmo sem programação dinâmica em azul.

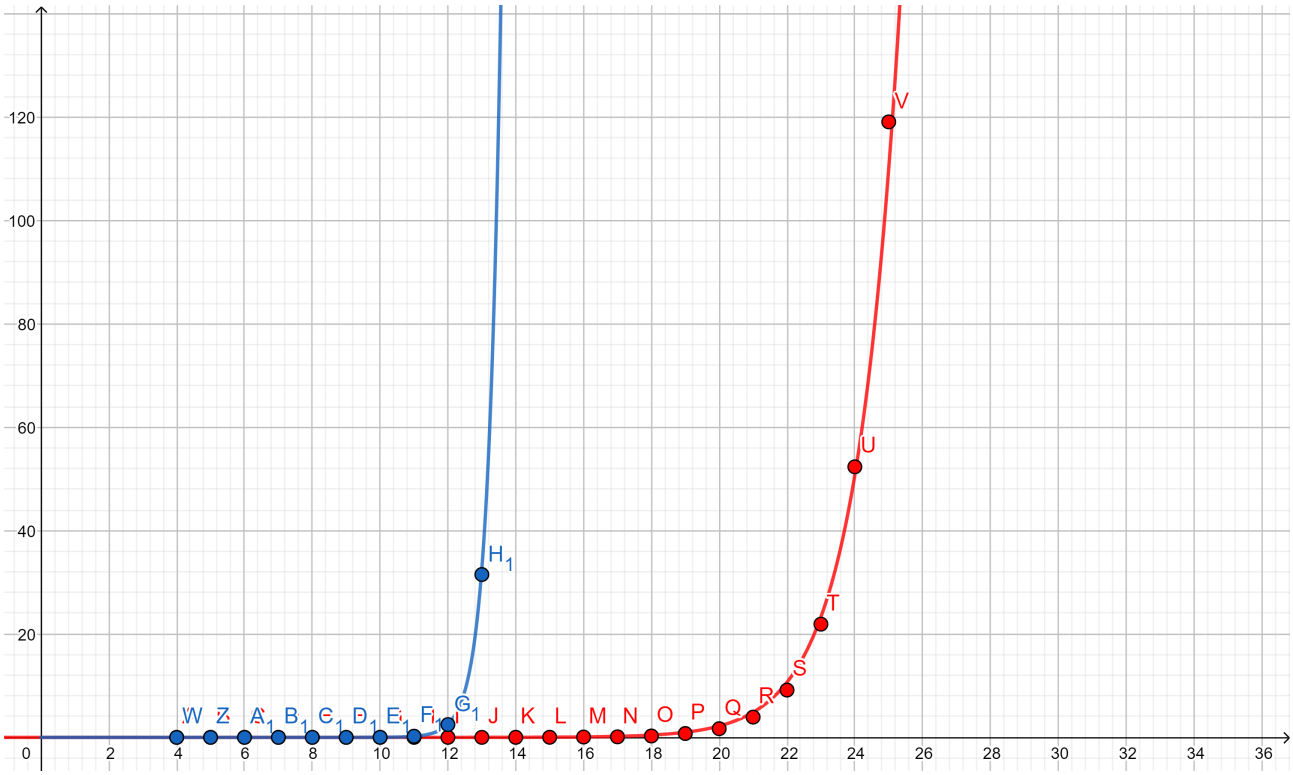

Figura 2.4: (Tamanho, Tempo(s))