Análise de Algoritmos - Algoritmos Diversos

Gustavo de Souza Silva
Guilherme de Souza Silva
Arthur Xavier
Schumaiquer Souto

Faculdade de Computação
Universidade Federal de Uberlândia

2 de agosto de 2017

Lista de Figuras

Lista de Tabelas

Lista de Listagens

Sumário

Capítulo 1

Introdução

Este relatório tem como objetivo fazer a análise de diversos algoritmos já conhecidos de Busca, Gulosos, dinâmicos de de estatística de ordem. O intuito deste trabalho é comprovar que as provas matemáticas realmente acontecem em um ambiente real de execução.

1.1 Codificação

Arquivo dos gulosos e Programação Dinâmica

Listagem 1.1: Arquivo dos gulosos e Programação Dinâmica

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "vetor.h"
4  #include <math.h>
5  #include<limits.h>
6  #include "final.h"
7
8  int min(int A[],int n){
9    int min = A[0];
10   for(int i = 1; i < n; i++)
11   {
12     if(min > A[i])
13       min = A[i];
14   }
15   return min;
16 }
17
18 int * min_max(int A[], int n){
19   int min = A[0];
20   int max = A[0];
21   int *resultado = (int*) malloc(2*sizeof(int));
22   int i;
23   for(i=1; i<n; i++){
24     if (min > A[i])
25       min = A[i];
26     if (max < A[i])
27       max = A[i];
28   }
```

```
29   resultado[0] = min;
30   resultado[1] = max;
31   return resultado;
32 }
33
34  int getMax(int first, int second)
35 {
36     return first > second ? first : second;
37 }
38
39 int corte_haste(int  *p,int n){
40   int i;
41   if (n <= 0)
42     return 0;
43   int q = INT_MIN;
44   for (i=0;i<n;i++)
45     {
46       q = getMax(q,p[i] + corte_haste(p,n-i-1));
47     }
48   return q;
49 }
50
51 int corte_haste_memorizado(int *p, int n){
52   int r[n+1];
53   for(int i=0;i<n;i++)
54     r[i] = INT_MIN;
55   int i;
56   return corte_haste_memorizado_aux(p,n,r);
57 }
58
59 int corte_haste_memorizado_aux(int *p,int n, int r[]){
60   int q,i;
61   if(r[n] > 0)
62     return r[n];
63   if (n== 0)
64     q=0;
65   else {
66     q = INT_MIN;
67     for(i = 0; i<n; i++)
68       q = getMax(q,p[i] + corte_haste_memorizado_aux(p,n-i-1,r));
69   }
70   r[n] = q;
71   return q;
72 }
73
74
75 int corte_haste_bottom_up(int *p, int n){
76   int val[n+1];
77    val[0] = 0;
78    int i, j;
79
80    for (i = 1; i<=n; i++)
81    {
82        int max_val = INT_MIN;
83        for (j = 0; j < i; j++)
84              max_val = getMax(max_val, p[j] + val[i-j-1]);
85        val[i] = max_val;
86    }
87
```

```
88     return val[n];
89 }
90
91 int parentizacao_recursiva(int p[], int i, int j)
92 {
93     if(i == j)
94         return 0;
95     int k;
96     int min = INT_MAX;
97     int count;
98
99     for (k = i; k <j; k++)
100    {
101        count = parentizacao_recursiva(p, i, k) +
102                parentizacao_recursiva(p, k+1, j) +
103                p[i-1]*p[k]*p[j];
104
105        if (count < min)
106            min = count;
107    }
108
109    return min;
110 }
111
112 int parentizacao_bottomup(int p[], int n)
113 {
114    int m[n][n];
115
116    int i, j, k, L, q;
117
118
119    for (i=1; i<n; i++)
120        m[i][i] = 0;
121
122    for (L=2; L<n; L++)
123    {
124        for (i=1; i<n-L+1; i++)
125        {
126            j = i+L-1;
127            m[i][j] = INT_MAX;
128            for (k=i; k<=j-1; k++)
129            {
130                q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
131                if (q < m[i][j])
132                    m[i][j] = q;
133            }
134        }
135    }
136    return m[1][n-1];
137 }
138 int scm_recursiva( char *X, char *Y, int m, int n )
139 {
140   if (m == 0 || n == 0)
141     return 0;
142   if (X[m-1] == Y[n-1])
143     return 1 + scm_recursiva(X, Y, m-1, n-1);
144   else
145     return getMax(scm_recursiva(X, Y, m, n-1), scm_recursiva(X, Y, m-1, n
          ));
```

```
146 }
147
148 int scm( char *X, char *Y, int m, int n )
149 {
150    int L[m+1][n+1];
151    int i, j;
152
153    for (i=0; i<=m; i++)
154    {
155      for (j=0; j<=n; j++)
156      {
157        if (i == 0 || j == 0)
158          L[i][j] = 0;
159
160        else if (X[i-1] == Y[j-1])
161          L[i][j] = L[i-1][j-1] + 1;
162
163        else
164          L[i][j] = getMax(L[i-1][j], L[i][j-1]);
165      }
166    }
167
168    return L[m][n];
169 }
170
171 void seletorgulosoativades(int s[], int f[], int n) //Iterativo
172 {
173     int i, j;
174     i = 0;
175     //A primeira é selecionada
176     for (j = 1; j < n; j++)
177     {
178       if (s[j] >= f[i])
179       {
180           //selecionou a atividade j
181           i = j;
182       }
183     }
184 }
185 int SeletorRecursivoAtividades(int s[],int f[],int k,int n) //Recursivo
186 {
187 int m = k + 1;
188 while (m <= n && s[m] < f[k]) // Encontre a primeira atividade em Sk a
      terminar
189   m = m + 1;
190 if (m <= n)
191   return SeletorRecursivoAtividades(s, f, m, n);
192 else
193   return 0;
194 }
195
196 void mochila(int quantidades[], int valores[],int n){
197   int P = 50;
198   int N = n;
199   float valores_unitarios[N];
200   int itensId[N];
201   float porcentagemAdicionada[N];
202   int peso = 0;
203   float valorMochila = 0;
```

```
204   int i;
205   for(i = 0; i < N; i++){
206     itensId[i] = i;
207     porcentagemAdicionada[i] = 0.0;
208     valores_unitarios[i] = valores[i] / quantidades[i];
209   }
210   // ordenar por valor unitario
211   int key, keyId, j;
212     for (i = 1; i < N; i++){
213         key = valores_unitarios[i];
214         keyId = itensId[i];
215         j = i-1;
216 
217         while (j >= 0 && valores_unitarios[j] > key){
218             valores_unitarios[j+1] = valores_unitarios[j];
219             itensId[j+1] = itensId[j];
220             j = j-1;
221         }
222       valores_unitarios[j+1] = key;
223         itensId[j+1] = keyId;
224     }
225 
226   i = N-1;
227   peso = 0;
228   valorMochila = 0;
229   int cabe;
230   float porcentagem;
231   while(peso <= P && i >= 0){
232     if(peso + quantidades[itensId[i]] > P){
233       cabe = P - peso;
234       porcentagem = (float)cabe / (float)quantidades[itensId[i]];
235       //printf("Porcentagem : %lf\n", porcentagem);
236       porcentagemAdicionada[itensId[i]] = porcentagem;
237       peso += cabe;
238       valorMochila += valores[itensId[i]] * cabe;
239       //printf("Adicionado item %d com quantidade %d\n", itensId[i], cabe)
              ;
240     }
241     else{
242       porcentagemAdicionada[itensId[i]] = 1;
243       peso += quantidades[itensId[i]];
244       valorMochila += valores[itensId[i]];
245       //printf("Adicionado item %d com quantidade %d\n", itensId[i],
            quantidades[itensId[i]]);
246     }
247     i--;
248   }
249 }
```

Arquivos com algoritmos de estatística de ordem

Listagem 1.2: Estatística de Ordem

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include<string.h>
4 #include<time.h>
5 #include <stdint.h>
6 #include<math.h>
```

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include "vetor.h"
#include "ordena.h"

#define BILHAO 1000000000L

#define CRONOMETRA(funcao,vetor,p,r,max) {                              \
   clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &inicio);          \
   funcao(vetor,p,r,max);                                               \
   clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &fim);             \
   tempo_de_cpu_aux = BILHAO * (fim.tv_sec - inicio.tv_sec) +     \
                  fim.tv_nsec - inicio.tv_nsec;               \
   }


int partition(int a[], int p, int r){
    int x = a[r];
    int i = p-1;
    int aux;
    for(int j = p; j < r; j++){
        if(a[j] <= x){
            i++;
            aux = a[i];
      a[i] = a[j];
      a[j] = aux;
        }
    }
    aux = a[i+1];
    a[i+1] = a[r];
    a[r] = aux;
    return i+1;
}


int particao_aleatoria(int a[], int p, int r){
    int i = rand() % r + 1;
     int aux = a[i];
     a[i] = a[r];
     a[r] = aux;
    return partition(a, p, r);
}

int particao_mediana(int a[], int p, int r){
    int i = rand() % r + 1;
     int aux = a[i];
     a[i] = a[r];
     a[r] = aux;
    return partition(a, p, r);
}


int selecao_aleatoria(int a[], int p, int r, int i)
{
    if(p==r)
        return a[p];
    int q = particao_aleatoria(a,p,r);
    int k = q-p+1;
```

```
66      if(i == k)
67          return a[q];
68      else if(i < k)
69          return selecao_aleatoria(a, p, q-1, i);
70      else return selecao_aleatoria(a, q+1, r, i-k);
71  }
72
73  int main(){
74   int r = 0;
75   int * v = NULL;
76  struct timespec inicio, fim;
77  uint64_t tempo_de_cpu = 0.0;
78  uint64_t tempo_de_cpu_aux = 0;
79  srand(time(NULL));
80  int tamanho = 0;
81  int count = 0;
82    char nome_do_arquivo[128];
83    char **arquivos;
84    int k=0,h = 0;
85    arquivos = (char**)malloc(200*sizeof(char*));
86    for(int i=0;i<200;i++){
87      arquivos[i] = (char*)malloc(128*sizeof(char));
88    }
89
90  for(int i=0;i<11;i++){
91      sprintf(nome_do_arquivo,"vetores/vIntAleatorio_%d.dat",(int)pow(2,i
            +4%15));
92      strcpy(arquivos[k], nome_do_arquivo);
93      k++;
94    }
95    for(int i=0;i<11;i++){
96      sprintf(nome_do_arquivo,"vetores/vIntCrescente_%d.dat",(int)pow(2,i
            +4%15));
97      strcpy(arquivos[k], nome_do_arquivo);
98      k++;
99
100   }
101   for(int i=0;i<11;i++){
102     sprintf(nome_do_arquivo,"vetores/vIntCrescente_%d_P10.dat",(int)pow(2,
            i+4%15));
103     strcpy(arquivos[k], nome_do_arquivo);
104     k++;
105   }
106   for(int i=0;i<11;i++){
107     sprintf(nome_do_arquivo,"vetores/vIntCrescente_%d_P20.dat",(int)pow(2,
            i+4%15));
108     strcpy(arquivos[k], nome_do_arquivo);
109     k++;
110   }
111   for(int i=0;i<11;i++){
112     sprintf(nome_do_arquivo,"vetores/vIntCrescente_%d_P30.dat",(int)pow(2,
            i+4%15));
113     strcpy(arquivos[k], nome_do_arquivo);
114     k++;
115   }
116   for(int i=0;i<11;i++){
117     sprintf(nome_do_arquivo,"vetores/vIntCrescente_%d_P40.dat",(int)pow(2,
            i+4%15));
118     strcpy(arquivos[k], nome_do_arquivo);
```

```
119     k++;
120   }
121   for(int i=0;i<11;i++){
122     sprintf(nome_do_arquivo,"vetores/vIntCrescente_%d_P50.dat",(int)pow(2,
          i+4%15));
123     strcpy(arquivos[k], nome_do_arquivo);
124     k++;
125   }
126   for(int i=0;i<11;i++){
127     sprintf(nome_do_arquivo,"vetores/vIntDecrescente_%d.dat",(int)pow(2,i
          +4%15));
128     strcpy(arquivos[k], nome_do_arquivo);
129     k++;
130   }
131   for(int i=0;i<11;i++){
132     sprintf(nome_do_arquivo,"vetores/vIntDecrescente_%d_P10.dat",(int)pow
          (2,i+4%15));
133     strcpy(arquivos[k], nome_do_arquivo);
134     k++;
135   }
136   for(int i=0;i<11;i++){
137     sprintf(nome_do_arquivo,"vetores/vIntDecrescente_%d_P20.dat",(int)pow
          (2,i+4%15));
138     strcpy(arquivos[k], nome_do_arquivo);
139     k++;
140   }
141
142   for(int i=0;i<11;i++){
143       sprintf(nome_do_arquivo,"vetores/vIntDecrescente_%d_P30.dat",(int)
            pow(2,i+4%15));
144     strcpy(arquivos[k], nome_do_arquivo);
145     k++;
146   }
147   for(int i=0;i<11;i++){
148     sprintf(nome_do_arquivo,"vetores/vIntDecrescente_%d_P40.dat",(int)pow
          (2,i+4%15));
149     strcpy(arquivos[k], nome_do_arquivo);
150     k++;
151   }
152   for(int i=0;i<11;i++){
153     sprintf(nome_do_arquivo,"vetores/vIntDecrescente_%d_P50.dat",(int)pow
          (2,i+4%15));
154     strcpy(arquivos[k], nome_do_arquivo);
155     k++;
156   }
157
158 for(int n = 0; n <= k; n++)
159 {
160 if(h > 10){
161         h = 0;
162     }
163 tempo_de_cpu = 0.0;
164
165
166 for(int j = 0; j<3; j++){
167   v = leia_vetor_int(arquivos[n],&r);
168   tamanho = (int)pow(2,h+4%15);
169   int A[tamanho];
170    for(int i =0; i<tamanho; i++)
```

```
171     A[i] = rand()%20000;
172   CRONOMETRA(selecao_aleatoria, A,0,tamanho-1,rand()%tamanho + 1);
173   tempo_de_cpu += tempo_de_cpu_aux;
174   }
175   if(count < 11)
176     printf("Tempo do vetor aleatorio tamanho %d: %llu\n",tamanho,(long
          long unsigned int)tempo_de_cpu/(uint64_t) 3);
177           else if(count < 22){
178         printf("Tempo do vetor Crescente tamanho %d: %llu\n",tamanho,(long
              long unsigned int)tempo_de_cpu/(uint64_t) 3);
179     }
180     else if(count < 33){
181         printf("Tempo do vetor Crescente P10 tamanho %d: %llu\n",tamanho,(
              long long unsigned int)tempo_de_cpu/(uint64_t) 3);
182     }
183     else if(count < 44){
184         printf("Tempo do vetor Crescente P20 tamanho %d: %llu\n",tamanho,(
              long long unsigned int)tempo_de_cpu/(uint64_t) 3);
185     }
186     else if( count < 55){
187         printf("Tempo do vetor Crescente P30 tamanho %d: %llu\n",tamanho,(
              long long unsigned int)tempo_de_cpu/(uint64_t) 3);
188     }
189     else if(count < 66){
190         printf("Tempo do vetor Crescente P40 tamanho %d: %llu\n",tamanho,(
              long long unsigned int)tempo_de_cpu/(uint64_t) 3);
191     }
192     else if(count < 77){
193         printf("Tempo do vetor Crescente P50 tamanho %d: %llu\n",tamanho,(
              long long unsigned int)tempo_de_cpu/(uint64_t) 3);
194     }
195     else if(count < 88){
196         printf("Tempo do vetor Decrescente tamanho %d: %llu\n",tamanho,(
              long long unsigned int)tempo_de_cpu/(uint64_t) 3);
197     }
198     else if(count < 99){
199         printf("Tempo do vetor Decrescente P10 tamanho %d: %llu\n",tamanho
              ,(long long unsigned int)tempo_de_cpu/(uint64_t) 3);
200     }
201     else if(count < 110){
202         printf("Tempo do vetor Decrescente P20 tamanho %d: %llu\n",tamanho
              ,(long long unsigned int)tempo_de_cpu/(uint64_t) 3);
203     }
204     else if(count < 121){
205         printf("Tempo do vetor Decrescente P30 tamanho %d: %llu\n",tamanho
              ,(long long unsigned int)tempo_de_cpu/(uint64_t) 3);
206     }
207     else if(count < 132){
208         printf("Tempo do vetor Decrescente P40 tamanho %d: %llu\n",tamanho
              ,(long long unsigned int)tempo_de_cpu/(uint64_t) 3);
209     }
210     else {
211         printf("Tempo do vetor Decrescente P50 tamanho %d: %llu\n",tamanho
              ,(long long unsigned int)tempo_de_cpu/(uint64_t) 3);
212      }
213   h++;
214   count++;
215 }
216 free(v);
```

```
217
218 }
```

Arquivo contendo o algoritmo de huffman

Listagem 1.3: Arquivo de Huffman

```
1
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <unistd.h>
6  #include <time.h>
7  #include <float.h>
8  #include <math.h>
9  #include <sys/types.h>
10 #include <sys/stat.h>
11 #include <stdint.h>
12
13 #include "final.h"
14
15 #define BILHAO 1000000000L
16
17 #define CRONOMETRA(funcao,palavras,palavra2,n1,n2) {
                                \
18    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &inicio);          \
19    funcao(palavras,palavra2,n1,n2);
                                                      \
20    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &fim);             \
21    tempo_de_cpu_aux = BILHAO * (fim.tv_sec - inicio.tv_sec) +     \
22                    fim.tv_nsec - inicio.tv_nsec;              \
23    }
24 #define MAX_TREE_HT 17000
25
26 char* gera_string ( int tamanho) {
27   char *validchars = "abcdefghijklmnopqrstuvwxyz";
28   char *novastr;
29   register int i;
30   int str_len;
31   // inicia o contador aleatório
32   srand ( time(NULL ));
33   // novo tamanho
34   str_len = tamanho;
35   //str_len = (rand() % MAX_STR_SIZE );
36   // checa tamanho
37   //str_len += ( str_len < MIN_STR_SIZE ) ? MIN_STR_SIZE : 0;
38   // aloca memoria
39   novastr = ( char * ) malloc ( (str_len + 1) * sizeof(char));
40   if ( !novastr ){
41     printf("[*] Erro ao alocar memoria.\n" );
42     exit ( EXIT_FAILURE );
43   }
44   // gera string aleatória
45   for ( i = 0; i < str_len; i++ ) {
46     novastr[i] = validchars[ rand() % strlen(validchars) ];
47     novastr[i + 1] = 0x0;
48   }
49   // imprive informações
50   //printf ( "[*] Tamanho: %d\n", str_len );
```

```
51   //printf ( "[*] String : %s\n", novastr );
52   return novastr;
53
54 }
55
56
57 // A Huffman tree node
58 struct MinHeapNode
59 {
60     char data;  // One of the input characters
61     unsigned freq;  // Frequency of the character
62     struct MinHeapNode *left, *right; // Left and right child of this node
63 };
64
65 // A Min Heap:  Collection of min heap (or Hufmman tree) nodes
66 struct MinHeap
67 {
68     unsigned size;    // Current size of min heap
69     unsigned capacity;   // capacity of min heap
70     struct MinHeapNode **array;  // Attay of minheap node pointers
71 };
72
73 // A utility function allocate a new min heap node with given character
74 // and frequency of the character
75 struct MinHeapNode* newNode(char data, unsigned freq)
76 {
77     struct MinHeapNode* temp =
78           (struct MinHeapNode*) malloc(sizeof(struct MinHeapNode));
79     temp->left = temp->right = NULL;
80     temp->data = data;
81     temp->freq = freq;
82     return temp;
83 }
84
85 // A utility function to create a min heap of given capacity
86 struct MinHeap* createMinHeap(unsigned capacity)
87 {
88     struct MinHeap* minHeap =
89          (struct MinHeap*) malloc(sizeof(struct MinHeap));
90     minHeap->size = 0;  // current size is 0
91     minHeap->capacity = capacity;
92     minHeap->array =
93      (struct MinHeapNode**)malloc(minHeap->capacity * sizeof(struct
           MinHeapNode*));
94     return minHeap;
95 }
96
97 // A utility function to swap two min heap nodes
98 void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b)
99 {
100     struct MinHeapNode* t = *a;
101     *a = *b;
102     *b = t;
103 }
104
105 // The standard minHeapify function.
106 void minHeapify(struct MinHeap* minHeap, int idx)
107 {
108     int smallest = idx;
```

```
109     int left = 2 * idx + 1;
110     int right = 2 * idx + 2;
111
112     if (left < minHeap->size &&
113         minHeap->array[left]->freq < minHeap->array[smallest]->freq)
114       smallest = left;
115
116     if (right < minHeap->size &&
117         minHeap->array[right]->freq < minHeap->array[smallest]->freq)
118       smallest = right;
119
120     if (smallest != idx)
121     {
122         swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);
123         minHeapify(minHeap, smallest);
124     }
125 }
126
127 // A utility function to check if size of heap is 1 or not
128 int isSizeOne(struct MinHeap* minHeap)
129 {
130     return (minHeap->size == 1);
131 }
132
133 // A standard function to extract minimum value node from heap
134 struct MinHeapNode* extractMin(struct MinHeap* minHeap)
135 {
136     struct MinHeapNode* temp = minHeap->array[0];
137     minHeap->array[0] = minHeap->array[minHeap->size - 1];
138     --minHeap->size;
139     minHeapify(minHeap, 0);
140     return temp;
141 }
142
143 // A utility function to insert a new node to Min Heap
144 void insertMinHeap(struct MinHeap* minHeap, struct MinHeapNode*
        minHeapNode)
145 {
146     ++minHeap->size;
147     int i = minHeap->size - 1;
148     while (i && minHeapNode->freq < minHeap->array[(i - 1)/2]->freq)
149     {
150         minHeap->array[i] = minHeap->array[(i - 1)/2];
151         i = (i - 1)/2;
152     }
153     minHeap->array[i] = minHeapNode;
154 }
155
156 // A standard funvtion to build min heap
157 void buildMinHeap(struct MinHeap* minHeap)
158 {
159     int n = minHeap->size - 1;
160     int i;
161     for (i = (n - 1) / 2; i >= 0; --i)
162         minHeapify(minHeap, i);
163 }
164
165 // A utility function to print an array of size n
166 void printArr(int arr[], int n)
```

```
167 {
168     int i;
169     for (i = 0; i < n; ++i)
170         printf("%d", arr[i]);
171     printf("\n");
172 }
173
174 // Utility function to check if this node is leaf
175 int isLeaf(struct MinHeapNode* root)
176 {
177     return !(root->left) && !(root->right) ;
178 }
179
180 // Creates a min heap of capacity equal to size and inserts all character
    of
181 // data[] in min heap. Initially size of min heap is equal to capacity
182 struct MinHeap* createAndBuildMinHeap(char data[], int freq[], int size)
183 {
184     struct MinHeap* minHeap = createMinHeap(size);
185     for (int i = 0; i < size; ++i)
186         minHeap->array[i] = newNode(data[i], freq[i]);
187     minHeap->size = size;
188     buildMinHeap(minHeap);
189     return minHeap;
190 }
191
192 // The main function that builds Huffman tree
193 struct MinHeapNode* buildHuffmanTree(char data[], int freq[], int size)
194 {
195     struct MinHeapNode *left, *right, *top;
196
197     // Step 1: Create a min heap of capacity equal to size.  Initially,
        there are
198     // modes equal to size.
199     struct MinHeap* minHeap = createAndBuildMinHeap(data, freq, size);
200
201     // Iterate while size of heap doesn't become 1
202     while (!isSizeOne(minHeap))
203     {
204         // Step 2: Extract the two minimum freq items from min heap
205         left = extractMin(minHeap);
206         right = extractMin(minHeap);
207
208         // Step 3:  Create a new internal node with frequency equal to the
209         // sum of the two nodes frequencies. Make the two extracted node
            as
210         // left and right children of this new node. Add this node to the
            min heap
211         // '$' is a special value for internal nodes, not used
212         top = newNode('$', left->freq + right->freq);
213         top->left = left;
214         top->right = right;
215         insertMinHeap(minHeap, top);
216     }
217
218     // Step 4: The remaining node is the root node and the tree is
        complete.
219     return extractMin(minHeap);
220 }
```

```
221
222 // Prints huffman codes from the root of Huffman Tree.  It uses arr[] to
223 // store codes
224 void printCodes(struct MinHeapNode* root, int arr[], int top)
225 {
226     // Assign 0 to left edge and recur
227     if (root->left)
228     {
229         arr[top] = 0;
230         printCodes(root->left, arr, top + 1);
231     }
232
233     // Assign 1 to right edge and recur
234     if (root->right)
235     {
236         arr[top] = 1;
237         printCodes(root->right, arr, top + 1);
238     }
239
240     // If this is a leaf node, then it contains one of the input
241     // characters, print the character and its code from arr[]
242     if (isLeaf(root))
243     {
244         printf("%c: ", root->data);
245         printArr(arr, top);
246     }
247 }
248
249 // The main function that builds a Huffman Tree and print codes by
    traversing
250 // the built Huffman Tree
251 void HuffmanCodes(char data[], int freq[], int size)
252 {
253    //  Construct Huffman Tree
254    struct MinHeapNode* root = buildHuffmanTree(data, freq, size);
255
256    // Print Huffman codes using the Huffman tree built above
257    int arr[MAX_TREE_HT], top = 0;
258    //printCodes(root, arr, top);
259 }
260
261
262 int* getFrequencias(char* novastr,int* freq,int tamanho){
263   char *validchars = "abcdefghijklmnopqrstuvwxyz";
264   for(int i=0;i<26;i++){
265       freq[i] = 0;
266       for(int j=0;j<tamanho;j++){
267           if(novastr[j] == validchars[i]){
268               freq[i]+=1;
269           }
270       }
271     }
272     return freq;
273 }
274
275 int main(){
276     uint64_t tempo_de_cpu_aux = 0;
277     struct timespec inicio, fim;
278     uint64_t tempo_de_cpu = 0.0;
```

```
279   char *caracteres = "abcdefghijklmnopqrstuvwxyz";
280     char* teste[11];
281     int tamanho[11];
282     int j=11;
283     int** freq = (int**)malloc(11*sizeof(int*));
284     for(int i=0;i<26;i++){
285         freq[i] = (int*)calloc(26,sizeof(int));
286     }
287     for(int i=0;i<11;i++){
288         tamanho[i] = (int)pow(2.0,i+4.0);
289         teste[i] = gera_string(tamanho[i]);
290         freq[i] = getFrequencias(teste[i],freq[i],tamanho[i]);
291     }
292         for(int i=0;i<11;i++){
293             for(int j=0;j<3;j++){
294                 //CRONOMETRA(HuffmanCodes, teste[i],freq[i],tamanho[i]);
295                 //CRONOMETRA(scm_recursiva,teste[i],teste[2],tamanho[i],
                        tamanho[2]);
296                 tempo_de_cpu += tempo_de_cpu_aux;
297                 //printf("Tempo antes da divisao: %ld\n",tempo_de_cpu);
298                 //HuffmanCodes(teste[i],freq[i],tamanho[i]);
299             }
300             //printf("Tempo antes da divisao: %ld\n",tempo_de_cpu);
301             printf("Tamanho %d ", tamanho[i]);
302             printf("Tempo %ld\n",tempo_de_cpu/3);
303             tempo_de_cpu = 0;
304             tempo_de_cpu_aux = 0;
305             j--;
306         }
307     //for(int i=0;i<11;i++){
308     //    for(int j=0;j<26;j++){
309     //        printf("Letra %c, freq: %d\n",caracteres[j],freq[i][j]);
310     //    }
311     //}
312 }
```

O arquivo vetor.c mantém todas as funções a respeito do vetor, como geração, preenchimento, etc.

Listagem 1.4: Arquivo referente ao grafo

```
1 #include "grafo.h"
2
3 //Node **node;
4
5 //FILE *fp;
6
7 /*int main()
8 {
9     int nv;
10
11     //fp= fopen("./grafos/grafoDenso_NV1024","r");
12     //fscanf(fp,"%d",&nv);
13     //initial(nv);
14     //CriaListAdj();
15     //PrintAdjList(nv);
16
17     return 0;
18 }*/
```

```
19
20
21
22 Node** initial(int nv)
23 {
24     return (Node **)malloc(nv * sizeof(Node *));
25 }
26
27
28
29 //CREATE ADJACENCY  LIST -
30 Node** CriaListAdj(Node **node, char *arq)
31 {
32     FILE *fp = NULL;
33     int nv;
34     fp = fopen(arq,"r");
35     if(fscanf(fp,"%d",&nv))
36         printf("ok!");
37     node = initial(nv);
38     int v1,v2;
39     Node *ptr;
40     while(fscanf(fp,"%d %d",&v1,&v2)!=EOF){
41         ptr = (Node *)malloc(sizeof(Node));
42         ptr->vertex = v2;
43         //Se o vertice não foi mapeado ainda
44         if (node[v1]==NULL) {
45             node[v1] = (Node *)malloc(sizeof(Node));
46             node[v1]->vertex = v1;
47             node[v1]->next = NULL;
48         }
49         Node *next = node[v1];
50         while (next->next!=NULL)
51             next = next->next;
52
53         next->next = ptr;
54     }
55     //PrintAdjList(node,nv);
56     fclose(fp);
57    return node;
58 }
59
60 int getTamanho(char *arq){
61     FILE *fp = NULL;
62     int nv;
63     fp = fopen(arq,"r");
64     if(fscanf(fp,"%d",&nv));
65     return nv;
66 }
67
68 void clearList(Node **node,int tam){
69     for(int i=1;i<=tam;i++){
70         Node* aux = node[i];
71         Node* tmp;
72         while(aux!=NULL){
73             tmp = aux->next;
74             free(aux);
75             aux = tmp;
76         }
77         free(node[i]);
```

```
78      }
79      free(node);
80  }
81
82  //PRINT LIST
83  void PrintAdjList(Node** node,int nv)
84  {
85      int i;
86      Node *ptr;
87
88      for(i=1; i<=nv; i++){
89          ptr = node[i]->next;
90          printf("    node[%2d]  ",i);
91          while(ptr != NULL){
92              printf("  -->%2d", ptr->vertex);
93              ptr=ptr->next;
94          }
95          printf("\n");
96      }
97      printf("\n");
98  }
```

Este arquivo serve para gerar os vetores e salva-los em arquivos.

Listagem 1.5: Geração dos grafos

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <math.h>
5  #include <sys/types.h>
6  #include <sys/stat.h>
7  #include <unistd.h>
8
9  #include "grafo.h"
10
11 #define POT2(n) (1 << (n))
12
13
14
15
16
17 void gera_e_salva_graf(int n, Tipo t){
18 //  int * v = NULL;
19   char nome_do_arquivo[64];
20   char sufixo[10];
21
22   switch (t){
23     case ESPARSO:
24       sprintf(nome_do_arquivo, "grafoEsparso_NV%d", n);
25       break;
26     case DENSO:
27       sprintf(nome_do_arquivo, "grafoDenso_NV%d", n);
28       break;
29     default: break; //CONFIRME(false,
30                       //"gera_e_salva_graf: Ordenação desconhecida");
31   }
32   escreve_grafo(n,t,nome_do_arquivo);
33 }
```

```
34
35 void escreve_grafo(int n, Tipo t, char* arq){
36     int i,j;
37     FILE* fd = NULL;
38
39     fd = fopen(arq,"w");
40
41     fprintf(fd, "%d\n", n);
42     switch(t){
43         case ESPARSO:
44             for(i=1;i<n;i++){
45                 fprintf(fd,"%d %d\n", i, i+1);
46             }
47             fprintf(fd,"%d",n);
48             break;
49         case DENSO:
50             for(i=1;i<n;i++){
51                 for(j=1;j<=n;j++){
52                     if(i!=j){
53                         fprintf(fd,"%d %d\n",i, j);
54                     }
55                 }
56             }
57             fprintf(fd,"%d",n);
58             break;
59         default:
60             break;
61     }
62 }
63
64
65 int main(int argc, char *argv[]){
66   int n = 0;
67   int p = 0;
68   char diretorio[256];
69
70   struct stat st = {0};
71
72
73   if (argc == 2)
74     strcpy(diretorio, argv[1]);
75   else
76     strcpy(diretorio, "./grafos");
77
78   if (stat(diretorio, &st) == -1) { // se o diretorio não existir,
79     mkdir(diretorio, 0700);         // crie um
80   }
81
82   CONFIRME(chdir(diretorio) == 0, "Erro ao mudar de diretório");
83
84   for(int i = 128; i<=1024;i=(i*2)){
85     gera_e_salva_graf(i, ESPARSO);
86     gera_e_salva_graf(i, DENSO);
87
88   }
89
90   //CONFIRME(chdir("..") == 0, "Erro ao mudar de diretório");
91
92   exit(0);
```

```
93 }
```

Este arquivo contém os algoritmos de ordenação pedidos.

Listagem 1.6: Métodos de busca

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "grafo.h"
4  #include "opgrafo.h"
5  #include <math.h>
6
7  void VisitaEmProfundidade(Node** G, int u, int* cor,int *r,int tempo){
8    cor[u] = 1; // 1: CINZA
9    tempo = tempo + 1;
10   //d[u] = tempo;
11   Node* aux = G[u]->next;
12   while(aux != NULL){
13     if(cor[aux->vertex] == 2){
14       r[aux->vertex] = u;
15       VisitaEmProfundidade(G,aux->vertex,cor,r,tempo);
16     }
17     aux = aux->next;
18   }
19   cor[u] = 0;
20   tempo = tempo+1;
21     //f[u] = tempo;
22 }
23
24
25 void BuscaEmProfundidade(Node **G, int u){
26   int cor[u];
27   int r[u];
28   int tempo;
29   int i;
30
31   for(i = 1; i<=u ; i++)
32     cor[i] = 2;    //  2: Branco
33
34   tempo = 0;
35   for(int i=1;i<=u;i++){
36     if(cor[i] == 2){
37       VisitaEmProfundidade(G,i,cor, r,tempo);
38     }
39   }
40   /*for(int i=1;i<=u;i++){
41       printf("%d, ",cor[i]);
42   }*/
43 }
44
45 void BuscaEmLargura(Node** G ,int tam){
46     Node* fila[1024];
47     int front = 0;
48     int rear = -1;
49     int itemCount = 0;
50
51     int source = 1;
52   int cor[tam],r[tam];
53   //Fila* Q = (Fila*)malloc(sizeof(Fila));
```

```
54   Node *aux;
55   Node *tmp;
56   for(int i=1;i<=tam;i++){
57     if(i!=source){
58       cor[i] = 2;
59       //d[i] = infinito
60     }
61   }
62   cor[source] = 1;
63   //d[source] = 0;
64   r[source] = source;
65   InsereNaFila(fila,G[source],&itemCount,&rear);
66   while(isEmpty(itemCount) != 1){
67     aux = RemoveDaFila(fila,&itemCount,&front);
68     tmp = aux;
69     while(tmp!=NULL){
70       if(cor[tmp->vertex] == 2){
71         cor[tmp->vertex] = 1;
72         //d[tmp->vertex] = d[aux->vertex] + 1
73         r[tmp->vertex] = aux->vertex;
74         InsereNaFila(fila,G[tmp->vertex],&itemCount,&rear);
75       }
76       tmp = tmp->next;
77     }
78     cor[aux->vertex] = 0;
79   }
80 }
81
82 Node* topoDaFila(Node**fila,int front){
83     return fila[front];
84 }
85
86 int isFull(int itemCount){
87     if(itemCount == 1024)
88         return 1;
89     else
90         return 0;
91 }
92
93 int isEmpty(int itemCount){
94     if(itemCount == 0)
95         return 1;
96     else
97         return 0;
98 }
99
100 void InsereNaFila(Node** fila,Node* no,int* itemCount,int* rear){
101     if(!isFull(*itemCount)){
102         if(*rear == 1024-1){
103             *rear = -1;
104         }
105         fila[++(*rear)] = no;
106         (*itemCount)++;
107     }
108     //printf("Elemento inserido %d \n", no->vertex);
109 }
110
111 Node* RemoveDaFila(Node** fila,int *itemCount,int *front) {
112    Node* data = fila[(*front)++];
```

```
113
114     if((*front) == 1024) {
115        (*front) = 0;
116     }
117     (*itemCount)--;
118     return data;
119 }
120
121
122 void VisitaEmProfundidadeTop(Node** G, int u, int* cor,int *r,int tempo,
       int *d, int *f,Node **fila, int* rear, int *itemCount){
123   cor[u] = 1; // 1: CINZA
124   tempo = tempo + 1;
125   d[u] = tempo;
126   Node* aux = G[u]->next;
127   while(aux != NULL){
128     if(cor[aux->vertex] == 2){
129       r[aux->vertex] = u;
130       VisitaEmProfundidadeTop(G,aux->vertex,cor,r,tempo,d,f,fila,rear,
             itemCount);
131     }
132     aux = aux->next;
133   }
134   cor[u] = 0;
135   tempo = tempo+1;
136     f[u] = tempo;
137     //printf("Aresta %d tempo %d",u,f[u]);
138     InsereNaFila(fila,G[u],itemCount,rear);
139 }
140
141
142 void OrdenaTopologico(Node **G, int u){
143     Node* fila[1025];
144     int front = 0;
145     int rear = -1;
146     int itemCount = 0;
147   int cor[u];
148   int r[u];
149   int d[u];
150   int f[u];
151   int tempo;
152   int i;
153
154   for(i = 1; i<=u ; i++)
155     cor[i] = 2;    //  2: Branco
156   tempo = 0;
157   for(int i=1;i<=u;i++){
158     if(cor[i] == 2){
159       VisitaEmProfundidadeTop(G,i,cor, r,tempo,d,f, fila, &rear,&itemCount
             );
160     }
161   }
162   //for(int i=(u-1);i>0;i--){
163   //    printf("%d, ",fila[i]->vertex);
164   //}
165 }
```

O arquivo ensaios.c serve para automatizar e calcular os tempos de cada método de

ordenação.

Listagem 1.7: Automatização dos experimentos

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <stdint.h>
5  #include <time.h>
6  #include <float.h>
7  #include <math.h>
8  #include <sys/types.h>
9  #include <sys/stat.h>
10 #include <unistd.h>
11
12 #include "grafo.h"
13 #include "opgrafo.h"
14
15 #define BILHAO 1000000000L
16
17 #define CRONOMETRA(funcao,grafo,n) {                              \
18    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &inicio);              \
19    funcao(grafo,n);                                               \
20    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &fim);                 \
21    tempo_de_cpu_aux = BILHAO * (fim.tv_sec - inicio.tv_sec) +     \
22                   fim.tv_nsec - inicio.tv_nsec;                  \
23    }
24
25 int main(int argc, char *argv[]){
26     Node **grafo = NULL;
27     int n = 0;
28     uint64_t tempo_de_cpu_aux = 0;
29     int tamanho = 0,count = 0;
30     //clock_t inicio, fim;
31     struct timespec inicio, fim;
32     uint64_t tempo_de_cpu = 0.0;
33     char msg[256];
34     char nome_do_arquivo[128];
35     char **arquivos;
36     int k=0,h = 0;
37     arquivos = (char**)malloc(200*sizeof(char*));
38     for(int i=0;i<100;i++){
39         arquivos[i] = (char*)malloc(128*sizeof(char));
40     }
41     for(int i=128;i<=1024;i=i*2){
42         sprintf(nome_do_arquivo,"grafos/grafoEsparso_NV%d",i);
43         strcpy(arquivos[k], nome_do_arquivo);
44         k++;
45     }
46     for(int i=128;i<=1024;i=i*2){
47         sprintf(nome_do_arquivo,"grafos/grafoDenso_NV%d",i);
48         strcpy(arquivos[k], nome_do_arquivo);
49         k++;
50     }
51   //strcpy(nome_do_arquivo, "vetores/vIntCrescente_131072.dat");
52   // Leia o vetor a partir do arquivo
53   //v = leia_vetor_int(nome_do_arquivo, &n);
54     for(int i=0;i<k;i++){
55         tempo_de_cpu = 0.0;
56         for(int j=0;j<1;j++){
```

```
57              tamanho = getTamanho(arquivos[i]);
58              grafo = CriaListAdj(grafo,arquivos[i]);
59            //inicio = clock();
60            //ordena_por_bolha(v,n);
61            //insertion(v,tamanho);
62            //fim = clock();
63          CRONOMETRA(OrdenaTopologico, grafo,tamanho);
64            //tempo_de_cpu += ((double) (fim - inicio)) / CLOCKS_PER_SEC;
65          tempo_de_cpu += tempo_de_cpu_aux;
66            }
67            //PrintAdjList(grafo,tamanho);
68            //clearList(grafo,tamanho);
69            printf("Tamanho %d ", tamanho);
70            printf("Tempo %ld\n",tempo_de_cpu);
71      }
72    //imprime_vetor_int(v,16384);
73      exit(0);
74  }
```

1.1.1 Comandos

Os seguintes passos devem ser seguidos para criação dos grafos que serão utilizados no experimento:
1 - Compilar o arquivo Grafo.c;

```
> gcc -O3 -c grafo.c
> gcc -O3 -c opgrafo.c
```

2 - Compilar o programa que gera os grafos e os coloca no diretório determinado;

```
> gcc -O3 grafi.o gera_grafo.c -o gera_grafo.exe
```

3 - Para usá-lo digite

```
> ./gera_grafos.exe
```

Os passos a seguir são para execução do experimento
1 - Verifique a existência do diretório contendo os grafos, e então digite o seguinte comando:

```
> gcc -O3 -c opgrafo.c
```

2 - Agora é necessário compilar o arquivo de ensaio e tudo que será utilizado

```
> gcc -O3 grafo.o opgrafo.o ensaiosgrafo.c -o ensaiosgrafo.exe -lm
```

3 - Para executar digite:

```
> ./ensaiosgrafo.exe
```

1.2 Máquina de teste

Todos os testes foram realizados na mesma máquina com as seguintes configurações, e usando apenas um núcleo:
AMD FX-8350 4.0GHZ
16GB Memória DDR3-1600
HDD 2TB 7200RPM
Placa de video Nvidia GTX1050Ti
Sistema Operacional: Ubuntu 16.04

Capítulo 2

Grafo

A teoria dos grafos é um ramo da matemática que estuda as relações entre os objetos de um determinado conjunto. Para tal são empregadas estruturas chamadas de grafos, G(V,E), onde V é um conjunto não vazio de objetos denominados vértices e E é um subconjunto de pares não ordenados de V, chamados arestas. Neste trabalho, foi-se definido como grafo esparso um grafo que tem apenas uma aresta entre cada par de vértice (1 2,2 3,3 4,...n-1 n) e um granfo denso foi definido como um grafo completo, ou seja, cada vértice tem aresta com todos os outros vértices.

2.1 Busca Largura

É um algoritmo de busca em grafos utilizado para realizar uma busca ou travessia num grafo e estrutura de dados do tipo árvore. Você começa pelo vértice raiz e explora todos os vértices vizinhos. Então, para cada um desses vértices mais próximos, exploramos os seus vértices vizinhos inexplorados e assim por diante, até que ele encontre o alvo da busca.

2.2 Busca Largura - Grafo Esparso

Tabela 2.1: *Busca Largura com grafo Esparso*

Número de Elementos	**Tempo de execução em nanosegundos**
128	7943
256	7472
512	15414
1024	32102

2.2.1 Gráfico Busca Largura - Grafo Esparso

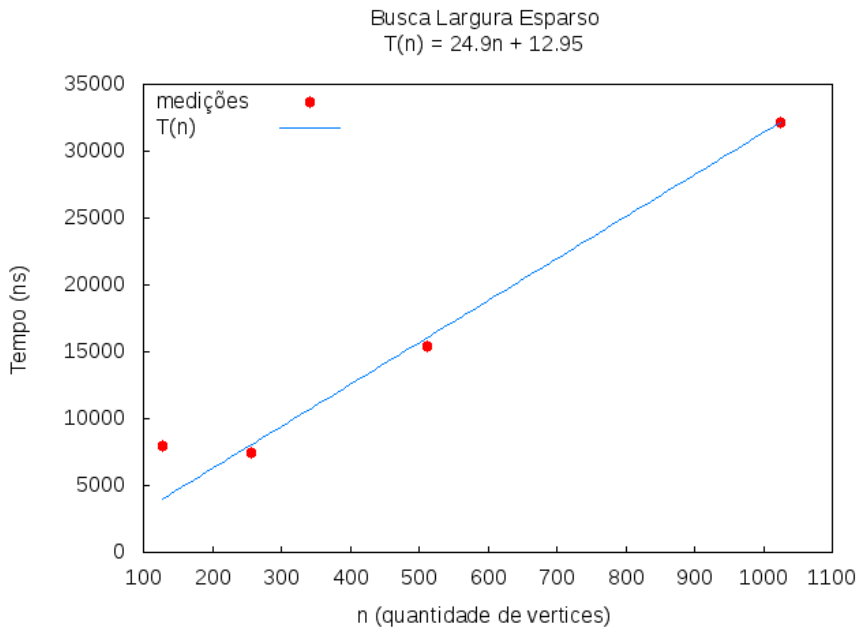

Figura 2.1: *Busca Largura - Grafo Esparso*

2.3 Busca Largura - Grafo Denso

Tabela 2.2: *Busca em Largura Grafo Denso*

Número de Elementos	Tempo de execução em nanosegundos
128	67546
256	318565
512	1216634
1024	4527417

2.3.1 Busca em Largura - Grafo Denso

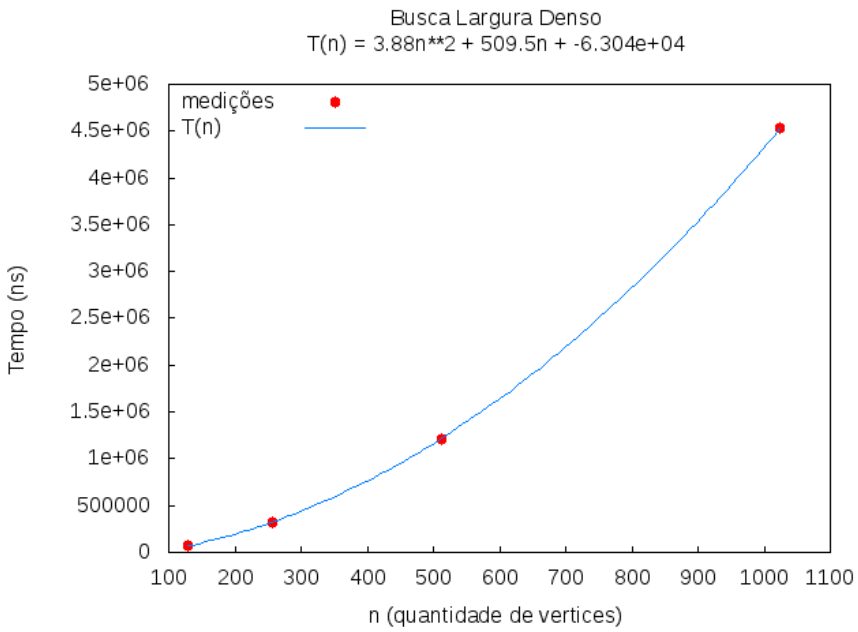

Figura 2.2: *Busca em Largura - Grafo Denso*

2.4 Busca Profundidade

É um algoritmo usado para realizar uma busca ou travessia numa árvore, estrutura de árvore ou grafo. O algoritmo começa num nó raiz (selecionando algum nó como sendo o raiz, no caso de um grafo) e explora tanto quanto possível cada um dos seus ramos, antes de retroceder(backtracking).

2.5 Busca Profundidade - Grafo Esparso

Tabela 2.3: *Busca Profundidade com Grafo Esparso*

Número de Elementos	Tempo de execução em nanosegundos
128	6619
256	7975
512	21265
1024	36223

2.5.1 Busca Profundidade - Grafo Esparso

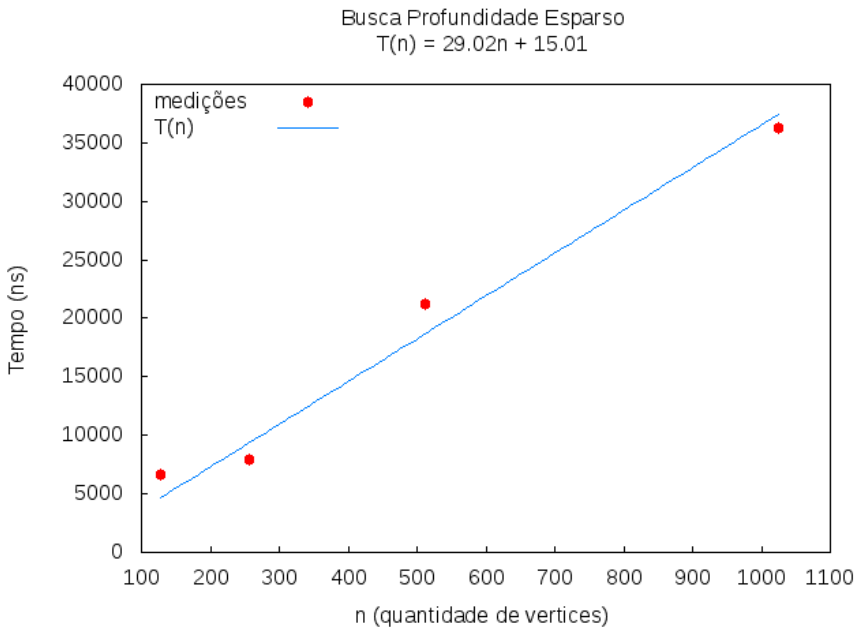

Figura 2.3: *Busca Profundidade - Grafo Esparso*

2.6 Busca Profundidade - Grafo Denso

Tabela 2.4: *Busca Profundidade em um Grafo Denso*

Número de Elementos	Tempo de execução em nanosegundos
128	72161
256	330916
512	1380078
1024	4735134

2.6.1 Busca Profundidade - Grafo Denso

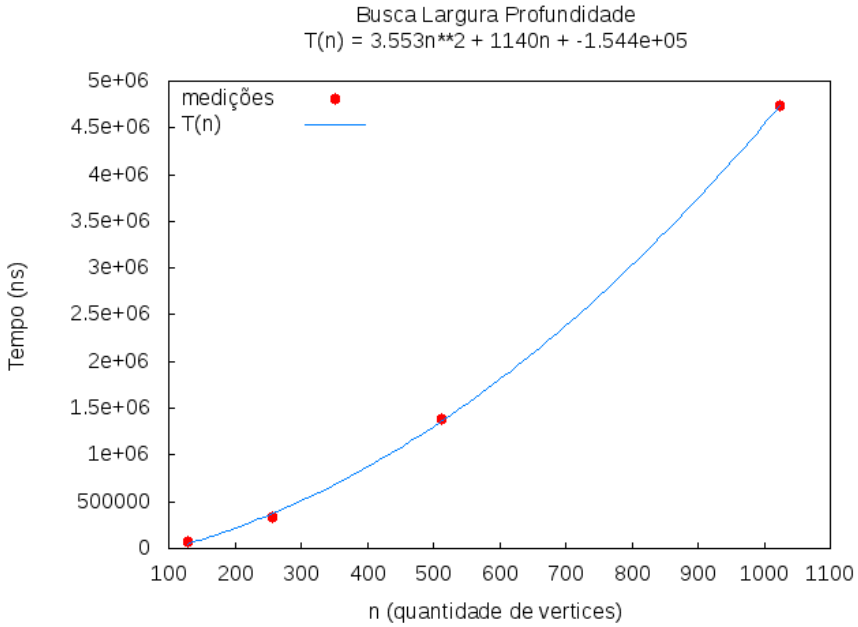

Figura 2.4: *Busca Profundidade - Grafo Denso*

2.7 Ordenação Topologica

É uma ordem linear de seus nós em que cada nó vem antes de todos nós para os quais este tenha arestas de saída. Cada DAG tem uma ou mais ordenações topológicas.

2.8 Ordenação Topologica - Grafo Esparso

Tabela 2.5: *Ordenação Topologica com Grafo Esparso*

Número de Elementos	Tempo de execução em nanosegundos
128	13549
256	15347
512	30745
1024	57293

2.8.1 Ordenação Topologica - Grafo Esparso

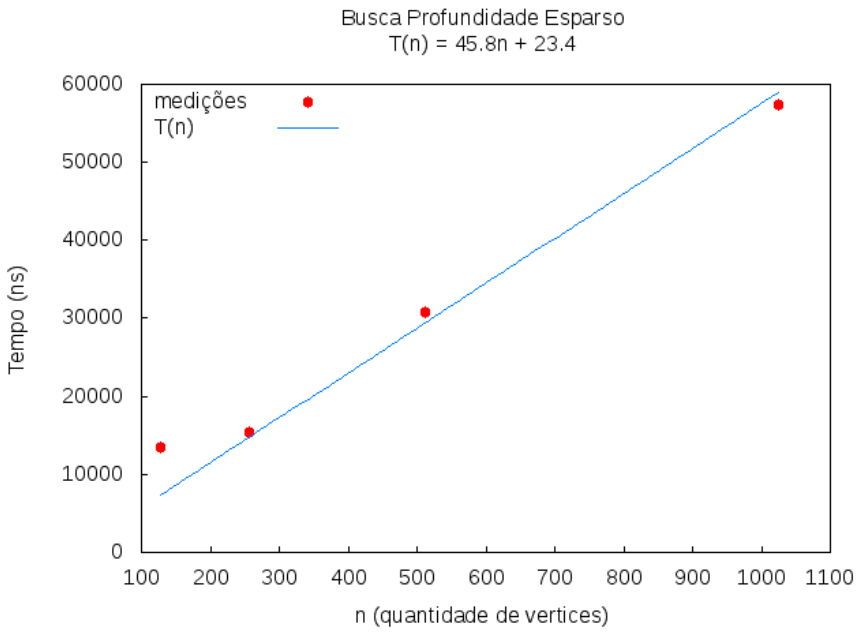

Figura 2.5: *Ordenação Topologica - Grafo Esparso*

2.9 Ordenação Topologica - Grafo Denso

Tabela 2.6: *Ordenação Topologica com Grafo Denso*

Número de Elementos	Tempo de execução em nanosegundos
128	93864
256	349061
512	1344655
1024	4865168

2.9.1 Ordenação Topologica - Grafo Denso

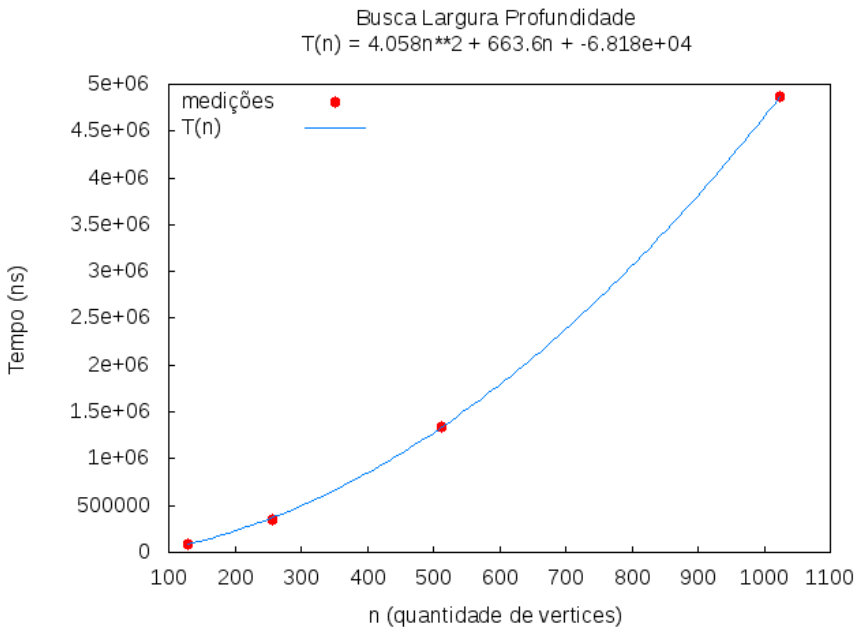

Figura 2.6: *Ordenação Topologica - Grafo Denso*

Capítulo 3

Guloso

Algoritmo guloso ou míope é técnica de projeto de algoritmos que tenta resolver o problema fazendo a escolha localmente ótima em cada fase com a esperança de encontrar um ótimo global.

3.1 Huffman

A codificação de Huffman é um método de compressão que usa as probabilidades de ocorrência dos símbolos no conjunto de dados a ser comprimido para determinar códigos de tamanho variável para cada símbolo.

3.1.1 Vetor aleatorio

Tabela gerada utilizando Huffman com vetor de string aleatorias com frequencias em tempo O(nlogn) de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos aleatóriamente.

Tabela 3.1: *Huffman com vetor aleatório*

Número de Elementos	**Tempo de execução em nanosegundos**
16	4599
32	8150
64	22736
128	42886
256	95255
512	170503
1024	438873
2048	1031620
4096	2592036
8192	5381493
16384	10506005

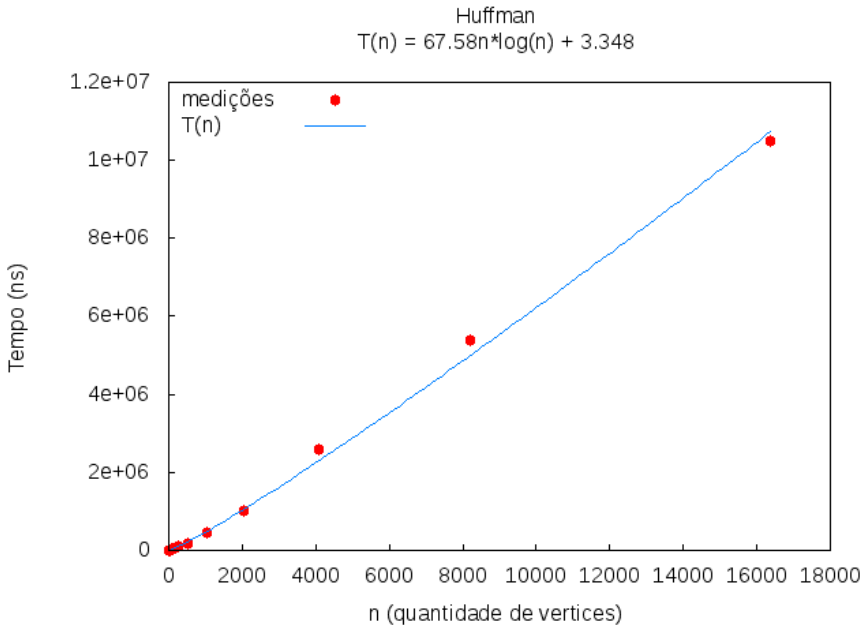

Figura 3.1: *Huffman - Vetor Aleatório*

3.2 Seleção de Atividade Interativo

É um problema onde dado um conjunto S de atividades, encontrar o maior subconjunto de S com atividades mutuamente compatíveis.

3.2.1 Vetor crescente

Tabela gerada utilizando Seleção de Atividade Interativo com vetor de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos crescente.

Tabela 3.2: *Seleção de Atividade Interativo com vetor crescente*

Número de Elementos	Tempo de execução em nanosegundos
16	615
32	680
64	769
128	802
256	764
512	836
1024	1058
2048	1294
4096	1685
8192	2594
16384	4578

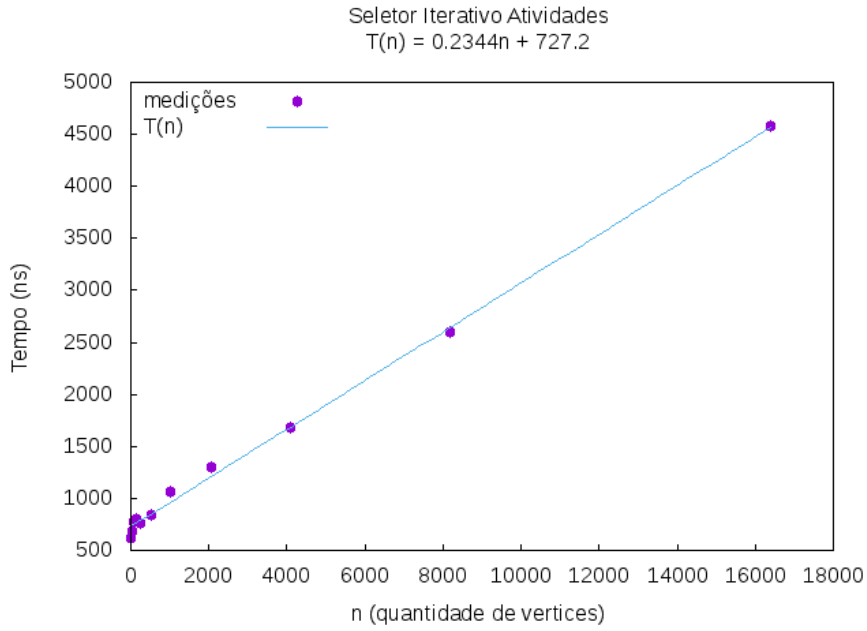

Figura 3.2: *Seleção de Atividade Interativo - Vetor crescente*

3.3 Seletor Recursivo de Atividade

É um problema onde dado um conjunto S de atividades, encontrar o maior subconjunto de S com atividades mutuamente compatíveis.

3.3.1 Vetor Aleatório

Tabela gerada utilizando Seleção de Atividade Interativo com vetor de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos aleatório.

Tabela 3.3: *Seleciona Recursivo de Atividade de Atividade de Atividade de Atividade de Atividade de Atividade de Atividade de Atividade de Atividade de Atividade de Atividade de Atividade com vetor aleatório*

Número de Elementos	Tempo de execução em nanosegundos
16	848
32	859
64	1226
128	2086
256	3352
512	4632
1024	10637
2048	24606
4096	40615
8192	88004
16384	135617

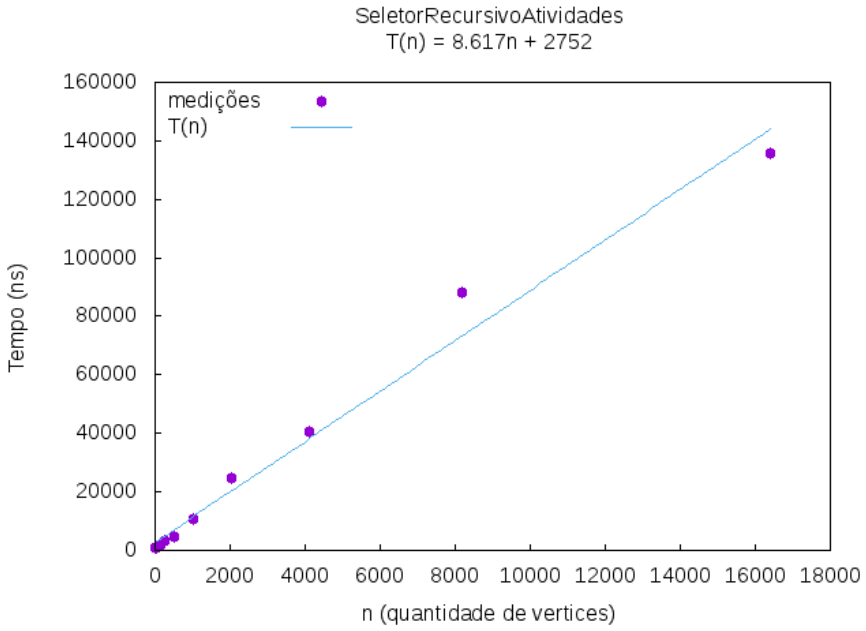

Figura 3.3: *Seleciona Recursivo de Atividade - Vetor Aleatório*

3.3.2 Vetor Crescente

Tabela gerada utilizando Seleciona Recursivo de Atividade com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente.

Tabela 3.4: *Seleciona Recursivo de Atividade com vetor crescente*

Número de Elementos	Tempo de execução em nanosegundos
16	563
32	677
64	1184
128	2093
256	3518
512	6070
1024	9960
2048	24854
4096	39433
8192	79908
16384	245208

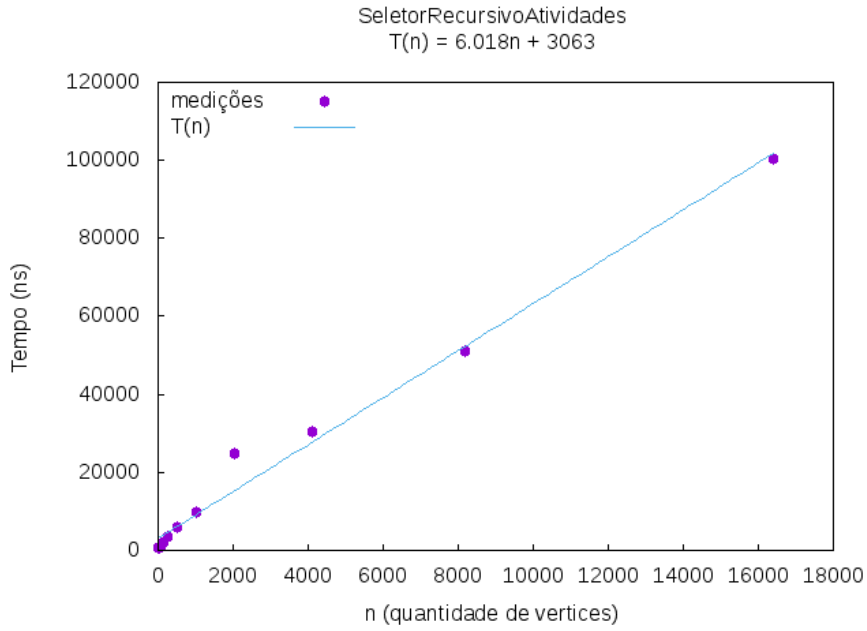

Figura 3.4: *Seleciona Recursivo de Atividade - Vetor crescente*

3.3.3 Vetor Crescente P10

Tabela gerada utilizando Seleciona Recursivo de Atividade com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P10.

Tabela 3.5: *Seleciona Recursivo de Atividade com vetor crescente P10*

Número de Elementos	Tempo de execução em nanosegundos
16	672
32	843
64	1091
128	2310
256	3383
512	6038
1024	19242
2048	34439
4096	46022
8192	95304
16384	176781

P10/SelecionaAleatorizado.png P10/SelecionaAleatorizado.png

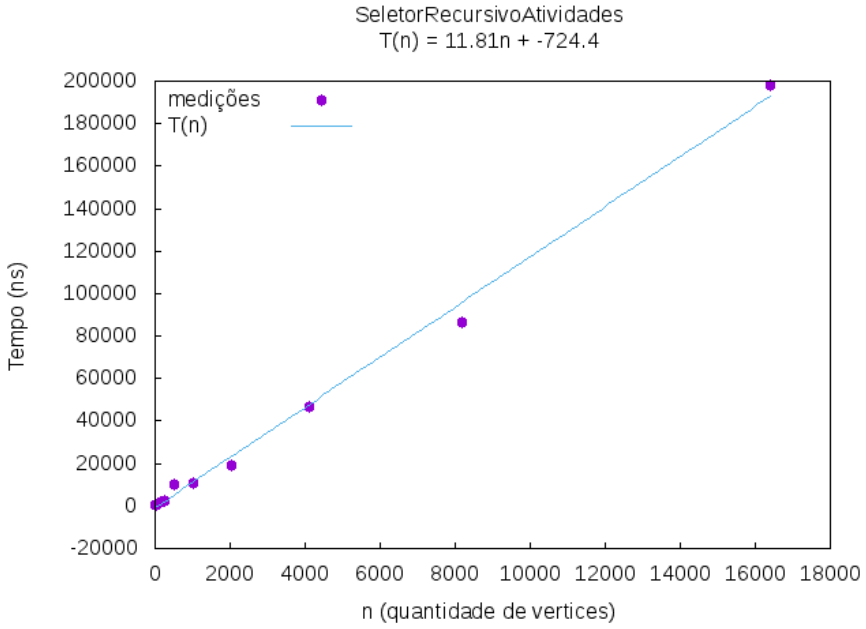

Figura 3.5: *Seleciona Recursivo de Atividade - Vetor crescente P10*

3.3.4 Vetor Crescente P20

Tabela gerada utilizando Seleciona Recursivo de Atividade com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P20.

Tabela 3.6: *Seleciona Recursivo de Atividade com vetor crescente P20*

Número de Elementos	**Tempo de execução em nanosegundos**
16	613
32	780
64	1259
128	1836
256	2640
512	9910
1024	11045
2048	19416
4096	46564
8192	86621
16384	198014

P20/SelecionaAleatorizado.png P20/SelecionaAleatorizado.png

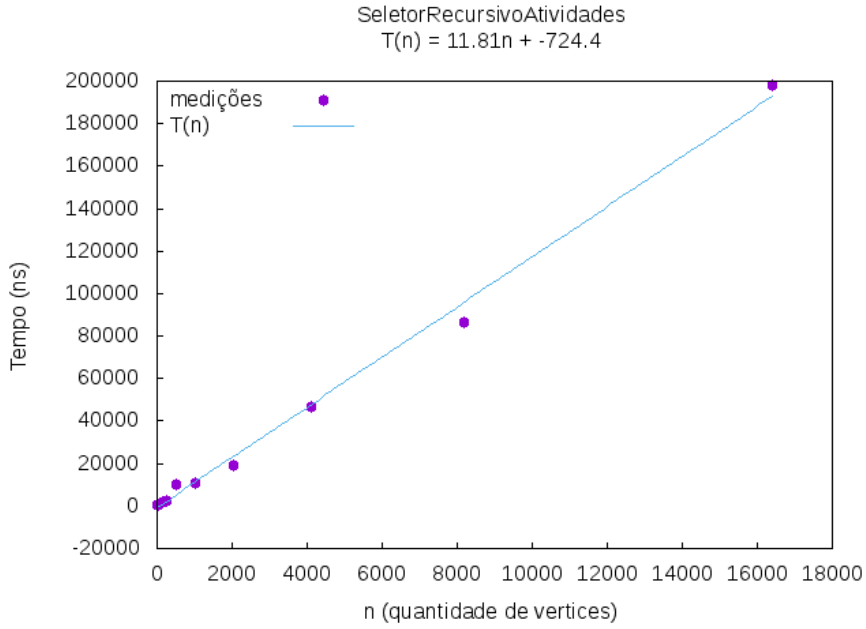

Figura 3.6: *Seleciona Recursivo de Atividade - Vetor crescente P20*

3.3.5 Vetor Crescente P30

Tabela gerada utilizando Seleciona Recursivo de Atividade com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P30.

Tabela 3.7: *Seleciona Recursivo de Atividade com vetor crescente P30*

Número de Elementos	Tempo de execução em nanosegundos
16	651
32	698
64	1008
128	2052
256	3051
512	6192
1024	10526
2048	17150
4096	45699
8192	76166
16384	155111

P30/SelecionaAleatorizado.png P30/SelecionaAleatorizado.png

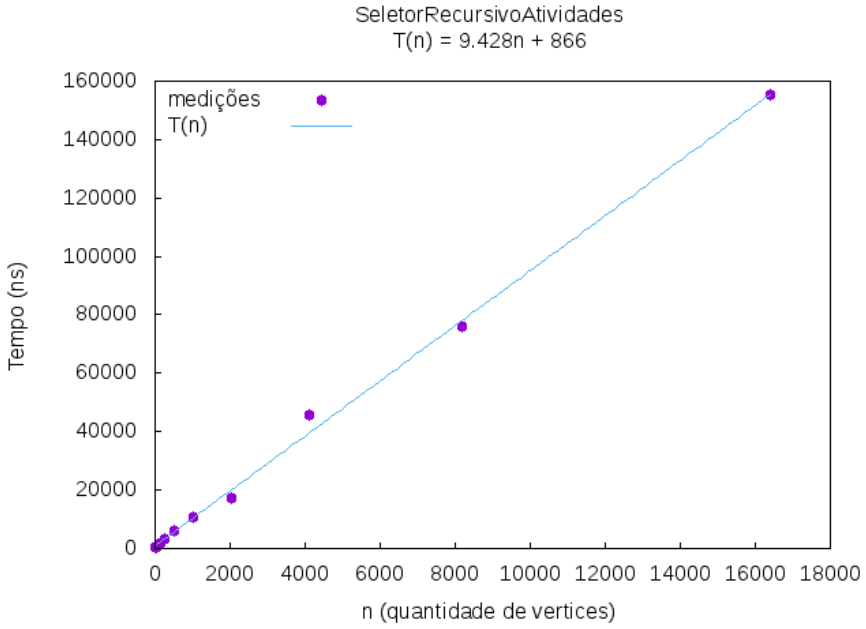

Figura 3.7: *Seleciona Recursivo de Atividade - Vetor crescente P30*

3.3.6 Vetor Crescente P40

Tabela gerada utilizando Seleciona Recursivo de Atividade com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P40.

Tabela 3.8: *Seleciona Recursivo de Atividade com vetor crescente P40*

Número de Elementos	Tempo de execução em nanosegundos
16	743
32	666
64	989
128	1508
256	2883
512	5360
1024	11656
2048	20539
4096	41529
8192	100858
16384	222297

P40/SelecionaAleatorizado.png P40/SelecionaAleatorizado.png

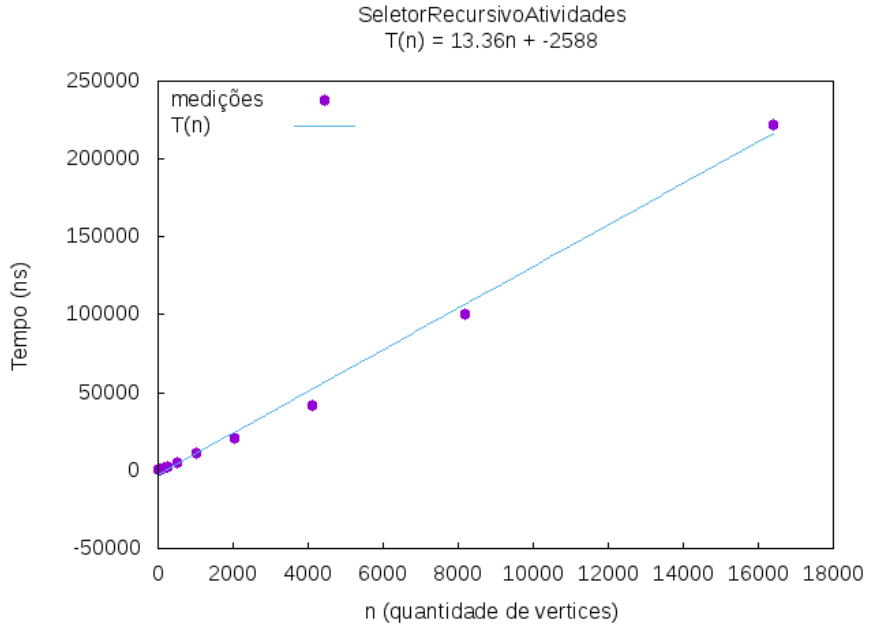

Figura 3.8: *Seleciona Recursivo de Atividade - Vetor crescente P40*

3.3.7 Vetor Crescente P50

Tabela gerada utilizando Seleciona Recursivo de Atividade com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P50.

Tabela 3.9: *Seleciona Recursivo de Atividade com vetor crescente P50*

Número de Elementos	Tempo de execução em nanosegundos
16	620
32	827
64	1258
128	1600
256	3615
512	4546
1024	10992
2048	24860
4096	60197
8192	98321
16384	249566

P50/SelecionaAleatorizado.png P50/SelecionaAleatorizado.png

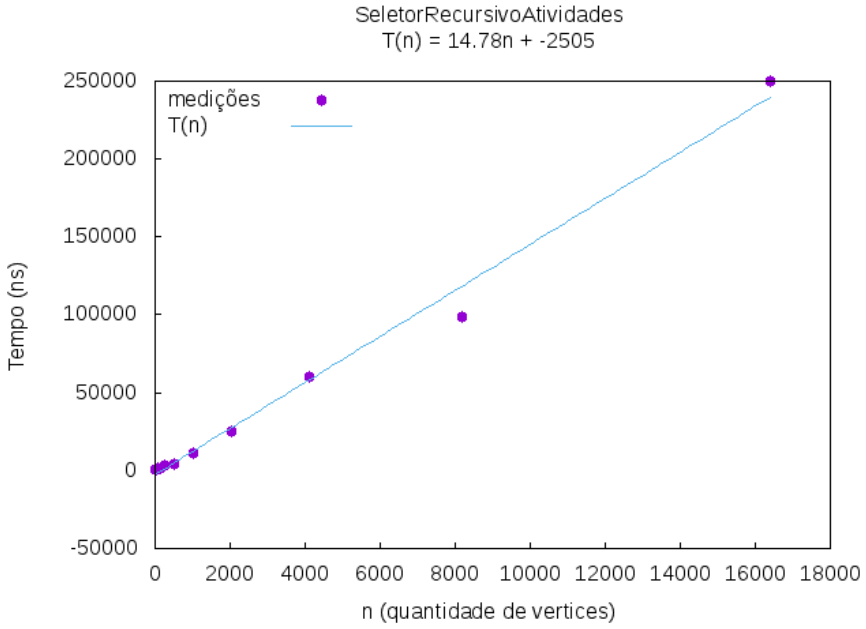

Figura 3.9: *Seleciona Recursivo de Atividade - Vetor crescente P50*

3.3.8 Vetor Decrescente

Tabela gerada utilizando Seleciona Recursivo de Atividade com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente.

Tabela 3.10: *Seleciona Recursivo de Atividade com vetor decrescente*

Número de Elementos	Tempo de execução em nanosegundos
16	822
32	1130
64	1610
128	2075
256	2774
512	6371
1024	12455
2048	20537
4096	39481
8192	90849
16384	216330

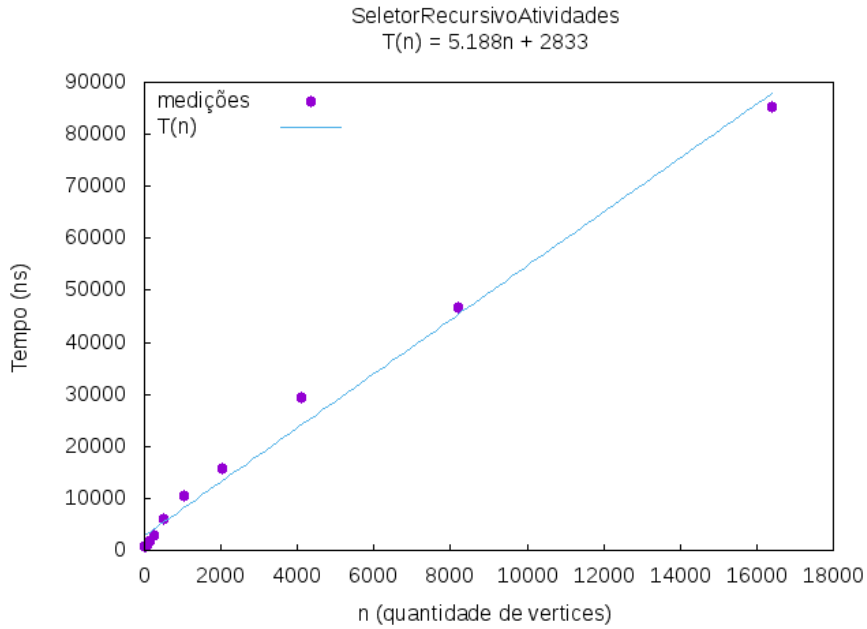

Figura 3.10: *Seleciona Recursivo de Atividade- Vetor decrescente*

3.3.9 Vetor Decrescente P10

Tabela gerada utilizando Seleciona Recursivo de Atividade com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P10.

Tabela 3.11: *Seleciona Recursivo de Atividade com vetor decrescente P10*

Número de Elementos	Tempo de execução em nanosegundos
16	692
32	875
64	968
128	1772
256	2865
512	6142
1024	10526
2048	25677
4096	46430
8192	46738
16384	177393

P10/SelecionaAleatorizado.png P10/SelecionaAleatorizado.png

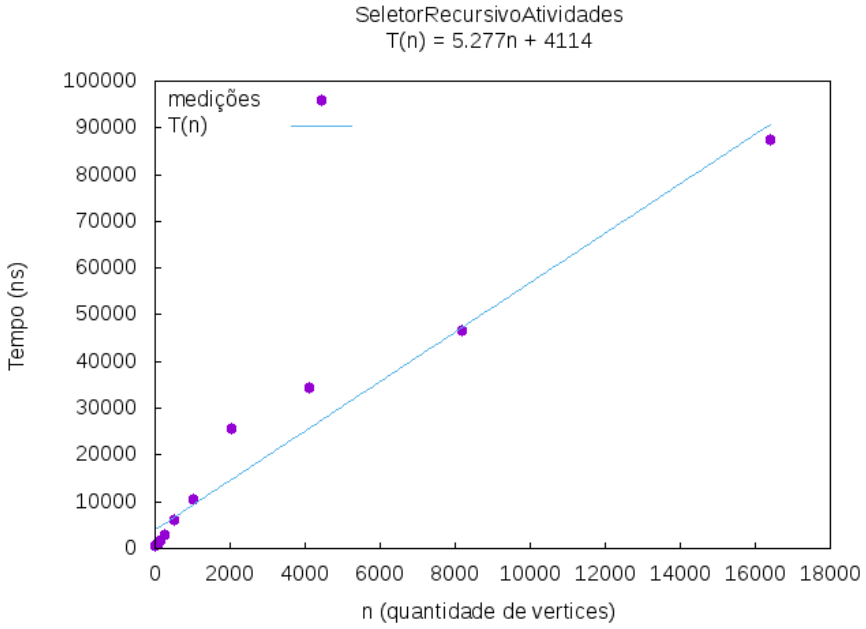

Figura 3.11: *Seleciona Recursivo de Atividade - Vetor decrescente P10*

3.3.10 Vetor Decrescente P20

Tabela gerada utilizando Seleciona Recursivo de Atividade com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P20.

Tabela 3.12: *Seleciona Recursivo de Atividade com vetor decrescente P20*

Número de Elementos	Tempo de execução em nanosegundos
16	963
32	1130
64	1209
128	1862
256	2040
512	6190
1024	14474
2048	28032
4096	58891
8192	85015
16384	191226

P20/SelecionaAleatorizado.png P20/SelecionaAleatorizado.png

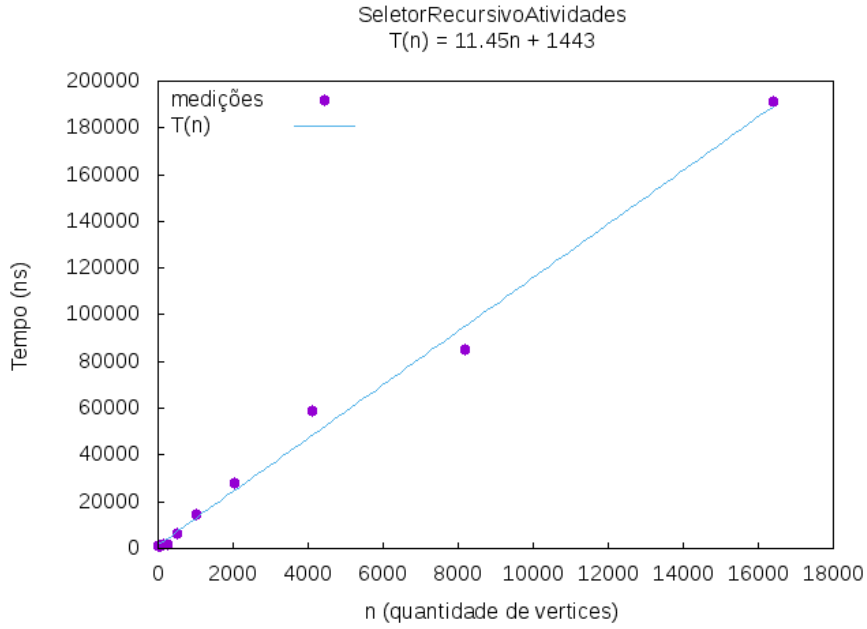

Figura 3.12: *Seleciona Recursivo de Atividade - Vetor decrescente P20*

3.3.11 Vetor Decrescente P30

Tabela gerada utilizando Seleciona Recursivo de Atividade com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P30.

Tabela 3.13: *Seleciona Recursivo de Atividade com vetor decrescente P30*

Número de Elementos	Tempo de execução em nanosegundos
16	691
32	791
64	1031
128	1907
256	3053
512	9593
1024	8715
2048	11899
4096	46642
8192	64121
16384	139003

P30/SelecionaAleatorizado.png P30/SelecionaAleatorizado.png

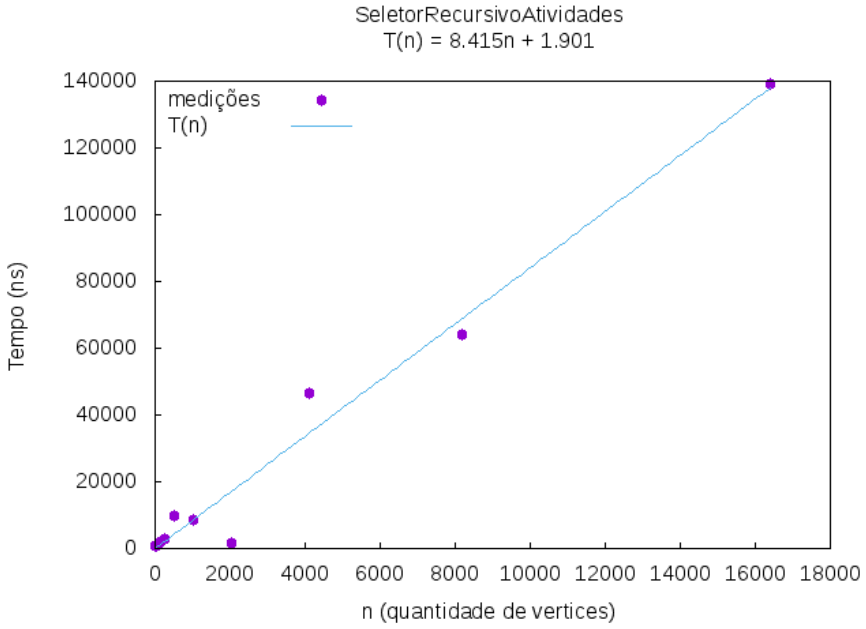

Figura 3.13: *Seleciona Recursivo de Atividade - Vetor decrescente P30*

3.3.12 Vetor Decrescente P40

Tabela gerada utilizando Seleciona Recursivo de Atividade com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P40.

Tabela 3.14: *Seleciona Recursivo de Atividade com vetor decrescente P40*

Número de Elementos	Tempo de execução em nanosegundos
16	818
32	712
64	1058
128	1643
256	2060
512	4179
1024	13014
2048	22027
4096	62149
8192	101864
16384	151244

P40/SelecionaAleatorizado.png P40/SelecionaAleatorizado.png

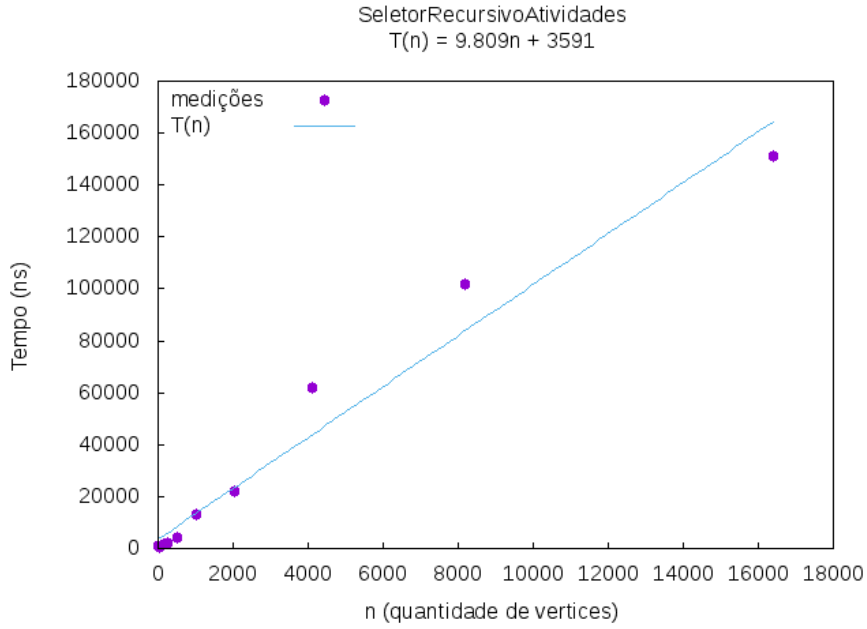

Figura 3.14: *Seleciona Recursivo de Atividade - Vetor decrescente P40*

3.3.13 Vetor Decrescente P50

Tabela gerada utilizando Seleciona Recursivo de Atividade com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P50.

Tabela 3.15: *Seleciona Recursivo de Atividade com vetor decrescente P50*

Número de Elementos	Tempo de execução em nanosegundos
16	593
32	692
64	1028
128	2107
256	3328
512	5288
1024	13776
2048	26749
4096	37678
8192	112482
16384	260308

P50/SelecionaAleatorizado.png P50/SelecionaAleatorizado.png

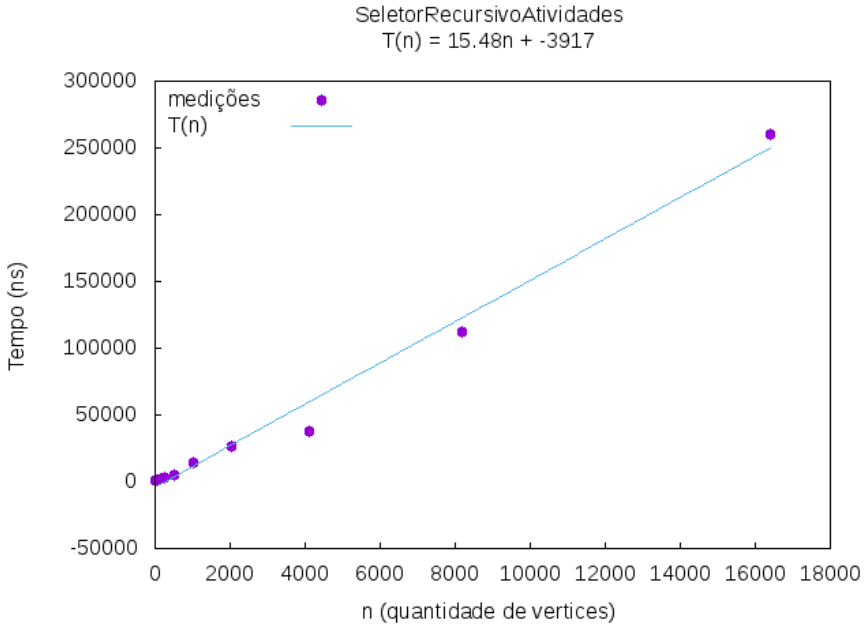

Figura 3.15: *Seleciona Recursivo de Atividade - Vetor decrescente P50*

3.4 Mochila Fracionaria

Esse problema também é chamado problema da mochila 0/1 pois para cada item deve ser levado ou deixado e pode-se pegar uma parte fracionária de um item ou pegar um mesmo item mais que uma vez.

3.4.1 Vetor ordenado

Tabela gerada utilizando Mochila Fracionaria com Vetor ordenado de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos ordenada.

Tabela 3.16: *Mochila Fracionaria com vetor ordenado*

Número de Elementos	Tempo de execução em nanosegundos
16	608
32	699
64	759
128	770
256	796
512	830
1024	949
2048	1190
4096	1565
8192	2253
16384	4106

Fracionada/MochilaFracionada.png Fracionada/MochilaFracionada.png

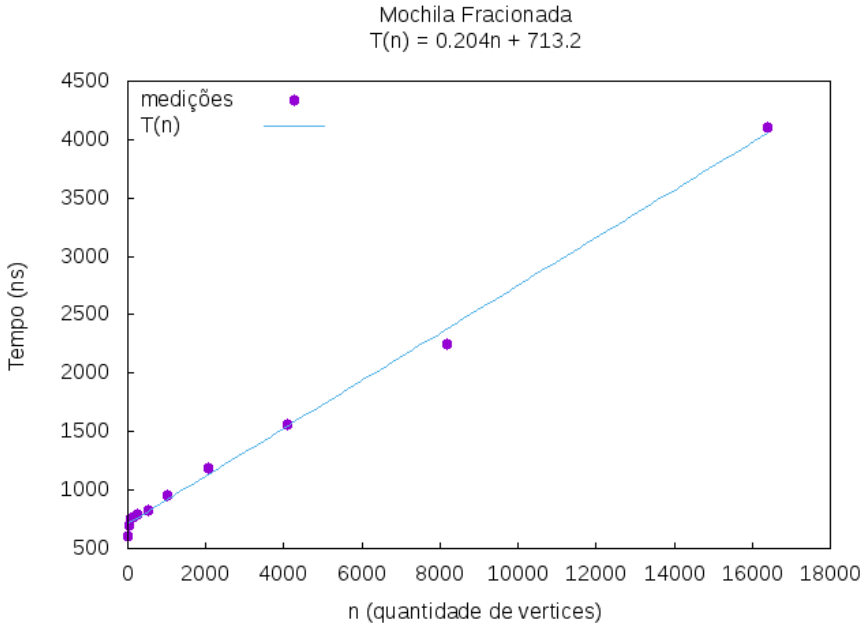

Figura 3.16: *Mochila Fracionaria - Vetor ordenado*

Capítulo 4

Programação Dinâmica

Programação dinâmica é um método para a construção de algoritmos para a resolução de problemas computacionais, em especial os de otimização combinatória. Ela é aplicável a problemas nos quais a solução ótima pode ser computada a partir da solução ótima previamente calculada e memorizada.

4.1 Corte Haste

O problema consiste em dada uma haste de npolegadas de comprimento, e uma tabela de preços pi, para i= 1,2,...,n, determinar qual a receita máxima que se pode obter cortando a haste e vendendo os seus pedaços, considerando que os comprimentos são sempre números inteiros de polegadas.

4.2 Corte Haste Bottom Up

É uma aproximação do problema de Corte de Haste utilizando a arvore de Bottom Up.

4.2.1 Vetor aleatorio

Tabela gerada utilizando Corte Haste Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos aleatóriamente.

Tabela 4.1: *Corte Haste Bottom Up com vetor aleatório*

Número de Elementos	Tempo de execução em nanosegundos
16	599
32	913
64	1798
128	4618
256	14530
512	51214
1024	193794
2048	761470
4096	2965012
8192	12774326
16384	48066532

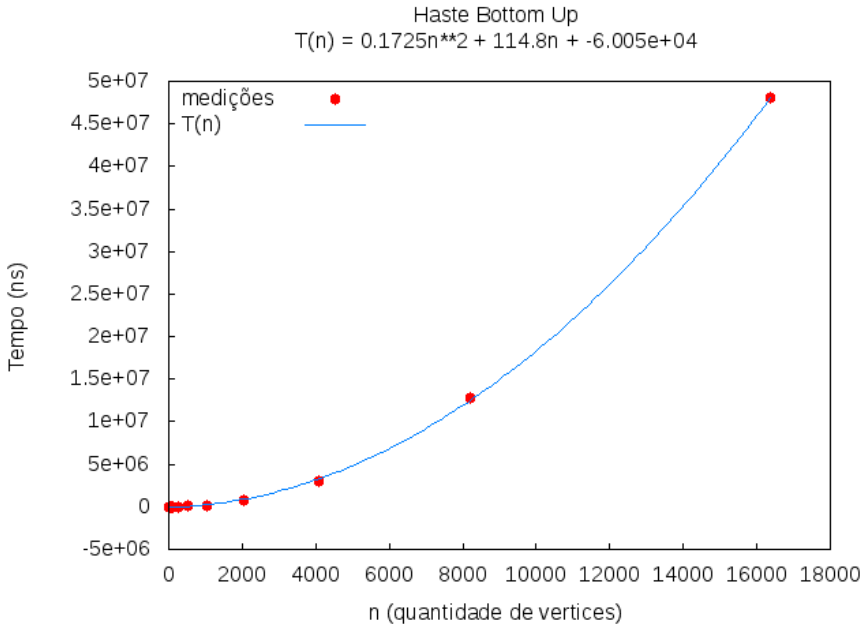

Figura 4.1: *Corte Haste Bottom Up - Vetor Aleatório*

4.3 Corte Haste Comum

Refere ao problema já mencionado anteriormente.

4.3.1 Vetor Comum

Vetores menores devido a estouro de memória.

Tabela 4.2: *Corte Haste Bottom Up com vetor aleatório*

Número de Elementos	Tempo de execução em nanosegundos
1	130
2	261
3	522
4	1045
5	2091
6	4182
7	8365
8	16731
9	33463
10	66926
11	133852
12	267705
16	535410

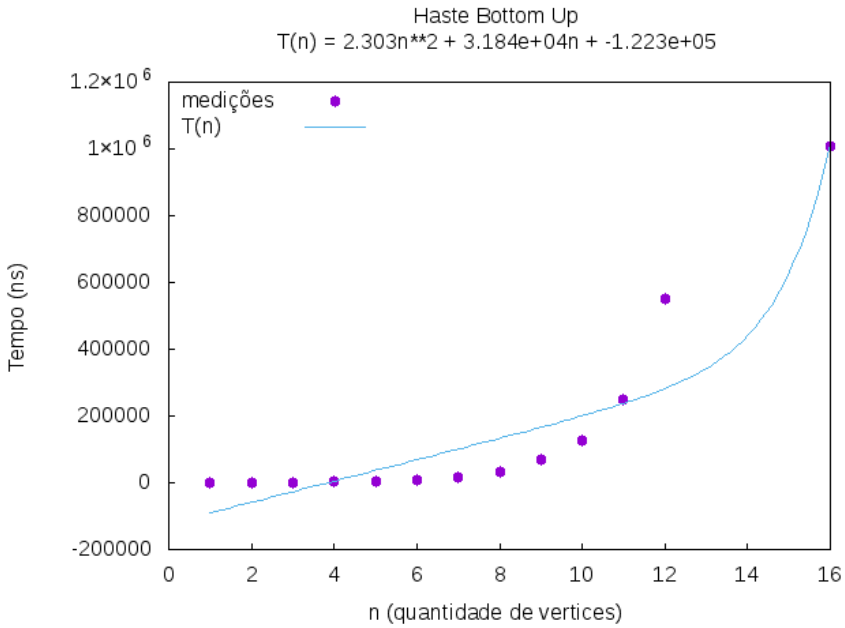

Figura 4.2: *Corte Haste Comum - Vetor Aleatório*

4.4 Corte Haste Memoizada

É uma aproximação do algoritmo utilizando a arvore Top DowN.

4.4.1 Vetor aleatorio

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos aleatóriamente.

Tabela 4.3: *Corte Haste Memoizada com vetor aleatório*

Número de Elementos	Tempo de execução em nanosegundos
16	1285
32	323
64	325
128	296
256	334
512	376
1024	545
2048	1325
4096	2331
8192	3844
16384	15475

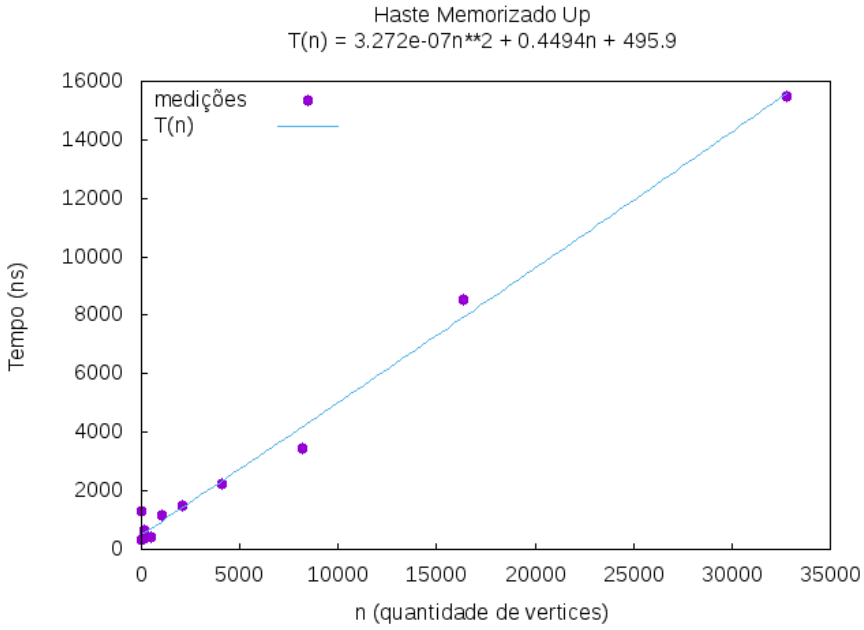

Figura 4.3: *Corte Haste Memoizada - Vetor Aleatório*

4.4.2 Vetor Crescente P10

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma crescente P10.

Tabela 4.4: *Corte Haste Memoizada com Vetor Crescente P10*

Número de Elementos	Tempo de execução em nanosegundos
16	1398
32	3456
64	6795
128	12053
256	24546
512	48256
1024	90413
2048	180521
4096	360512
8192	808256
16384	6200211

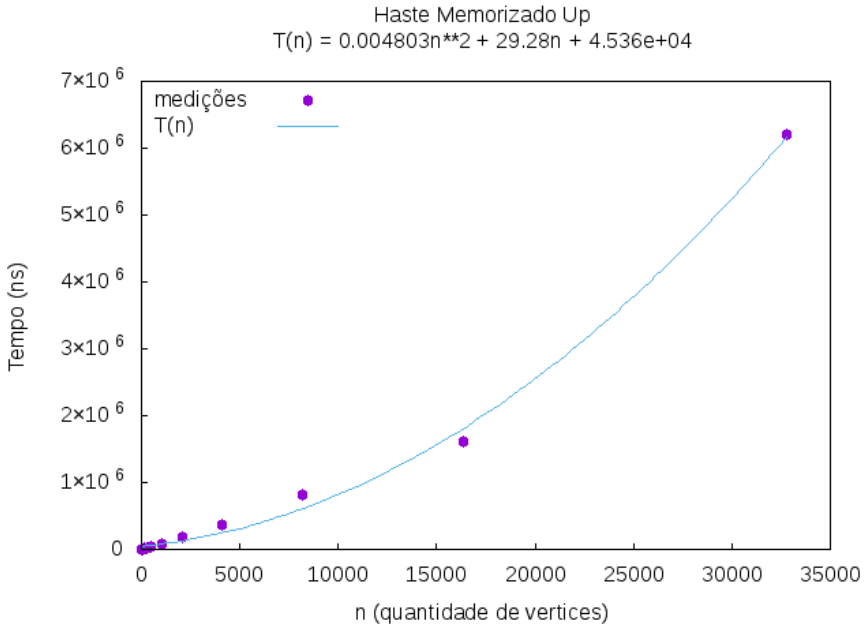

Figura 4.4: *Corte Haste Memoizada - Vetor Crescente P10*

4.4.3 Vetor Crescente P20

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma crescente P20.

Tabela 4.5: *Corte Haste Memoizada com Vetor Crescente P20*

Número de Elementos	Tempo de execução em nanosegundos
16	360
32	338
64	388
128	472
256	393
512	525
1024	619
2048	1236
4096	1041
8192	1519
16384	3379

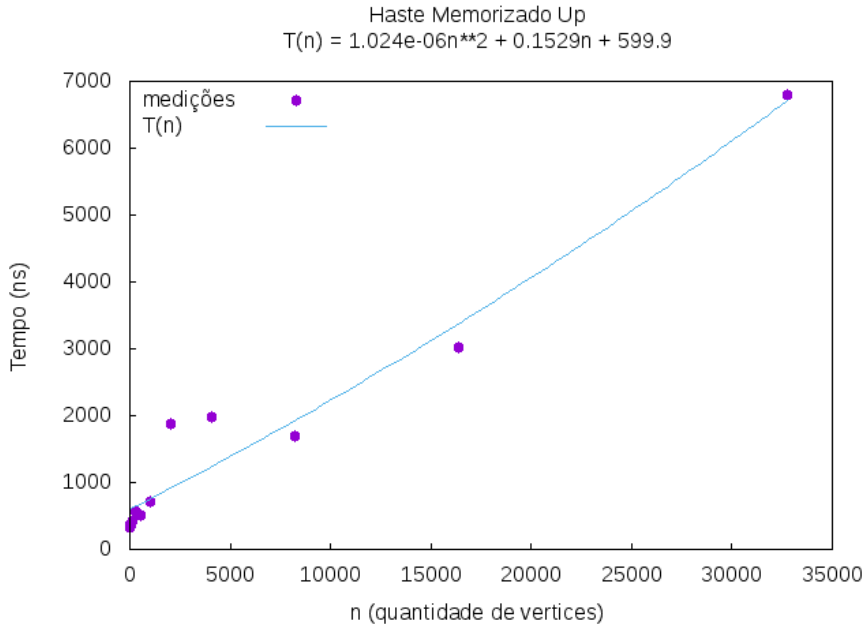

Figura 4.5: *Corte Haste Memoizada - Vetor Crescente P20*

4.4.4 Vetor Crescente P30

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma crescente P30.

Tabela 4.6: *Corte Haste Memoizada com Vetor Crescente P30*

Número de Elementos	Tempo de execução em nanosegundos
16	373
32	369
64	396
128	386
256	618
512	667
1024	1786
2048	2808
4096	3943
8192	6857
16384	16596

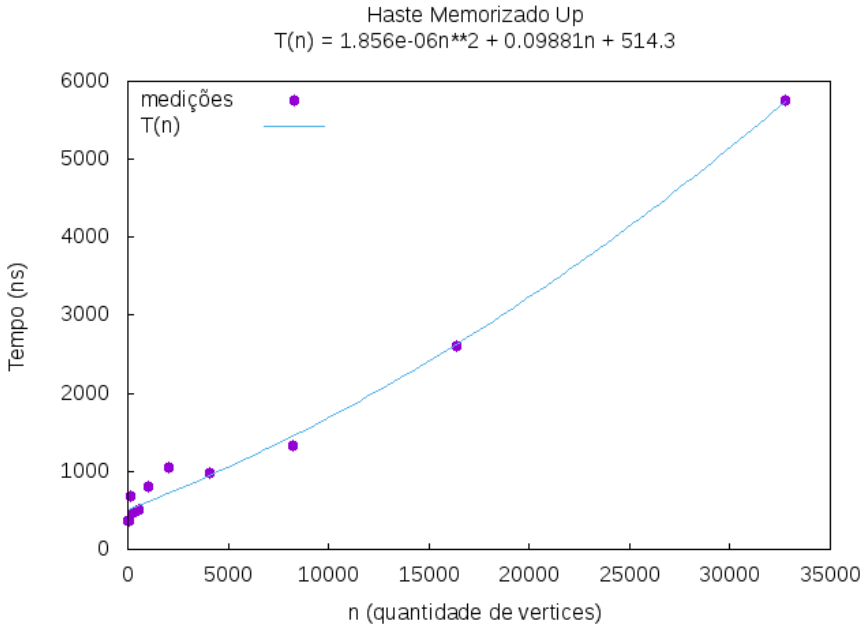

Figura 4.6: *Corte Haste Memoizada - Vetor Crescente P30*

4.4.5 Vetor Crescente P40

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma crescente P40.

Tabela 4.7: *Corte Haste Memoizada com Vetor Crescente P40*

Número de Elementos	Tempo de execução em nanosegundos
16	485
32	480
64	563
128	836
256	707
512	710
1024	483
2048	680
4096	1127
8192	1849
16384	3551

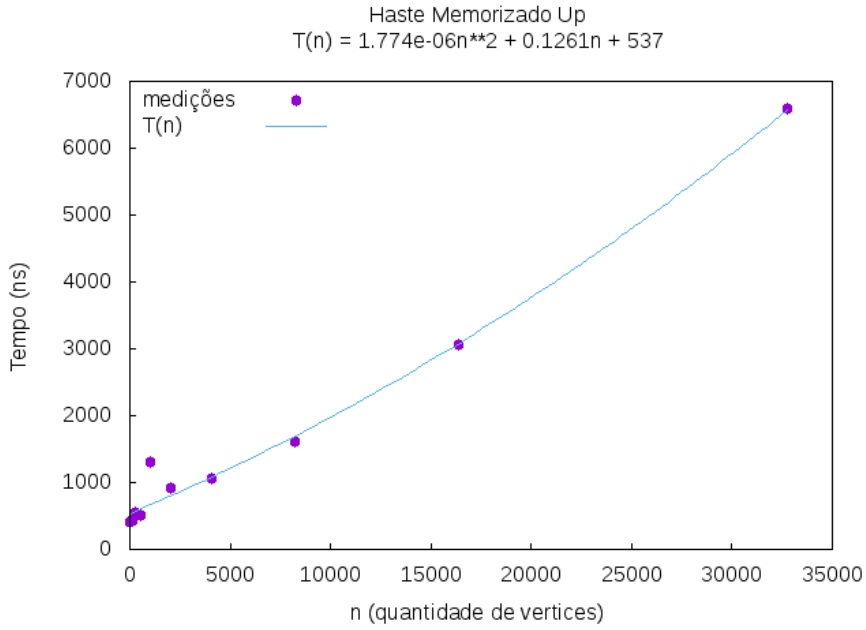

Figura 4.7: *Corte Haste Memoizada - Vetor Crescente P40*

4.4.6 Vetor Crescente P50

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma crescente P50.

Tabela 4.8: *Corte Haste Memoizada com Vetor Crescente P50*

Número de Elementos	Tempo de execução em nanosegundos
16	473
32	401
64	801
128	995
256	821
512	405
1024	681
2048	720
4096	1075
8192	1818
16384	3658

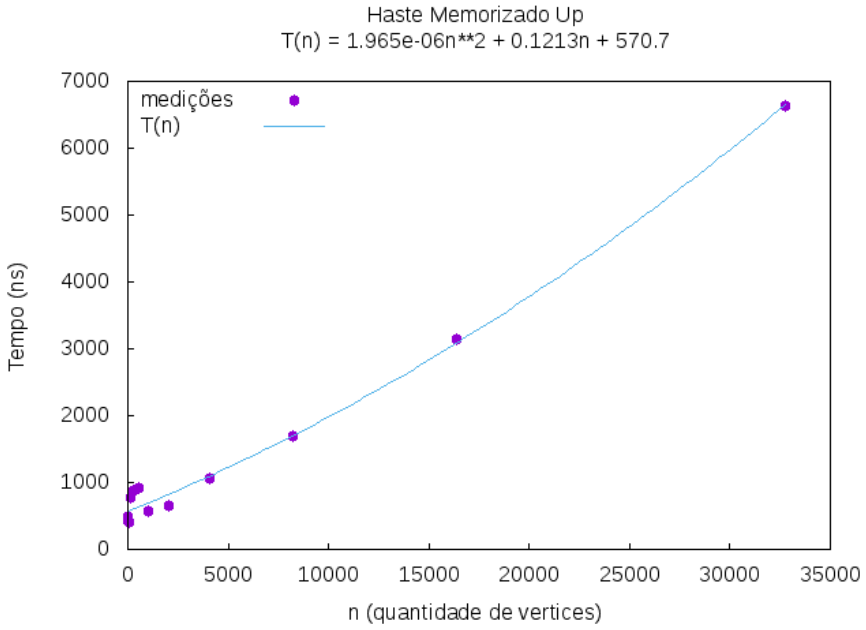

Figura 4.8: *Corte Haste Memoizada - Vetor Crescente P50*

4.4.7 Vetor Decrescente

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma decrescente.

Tabela 4.9: *Corte Haste Memoizada com Vetor Decrescente*

Número de Elementos	Tempo de execução em nanosegundos
16	1598
32	3256
64	5795
128	12196
256	24649
512	48146
1024	91564
2048	183545
4096	335949
8192	815698
16384	6198565

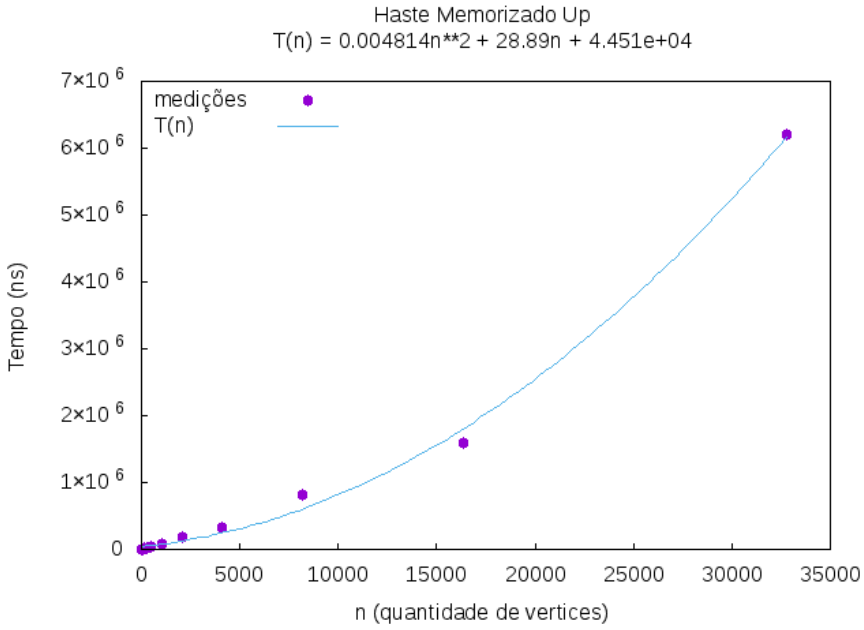

Figura 4.9: *Corte Haste Memoizada - Vetor Decrescente*

4.4.8 Vetor Decrescente P10

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma decrescente P10.

Tabela 4.10: *Corte Haste Memoizada com Vetor Decrescente P10*

Número de Elementos	Tempo de execução em nanosegundos
16	1398
32	2956
64	5465
128	13196
256	25697
512	50154
1024	93484
2048	183654
4096	315169
8192	835495
16384	1588859

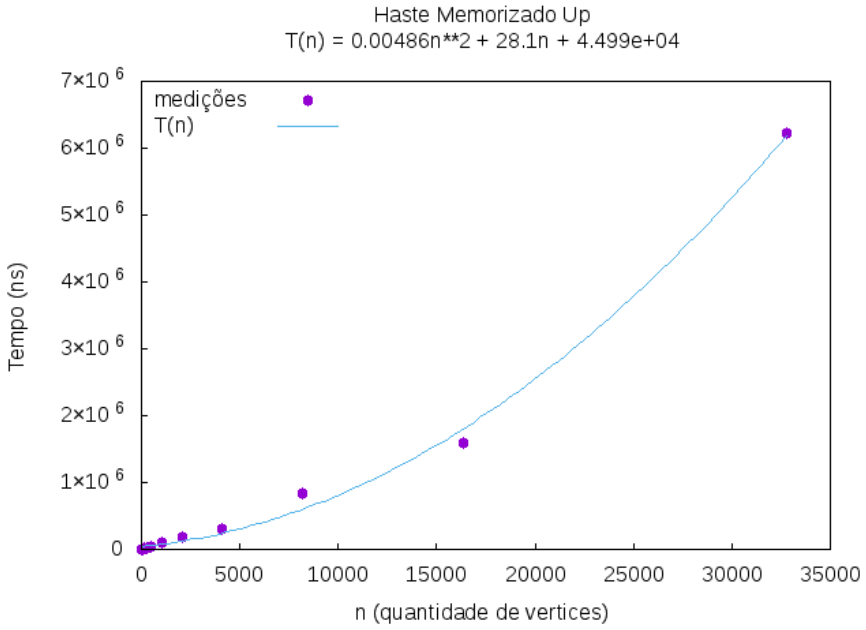

Figura 4.10: *Corte Haste Memoizada - Vetor Decrescente P10*

4.4.9 Vetor Decrescente P20

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma decrescente P20.

Tabela 4.11: *Corte Haste Memoizada com Vetor Decrescente P20*

Número de Elementos	Tempo de execução em nanosegundos
16	726
32	2229
64	633
128	633
256	1107
512	736
1024	1011
2048	1372
4096	2078
8192	3247
16384	6265

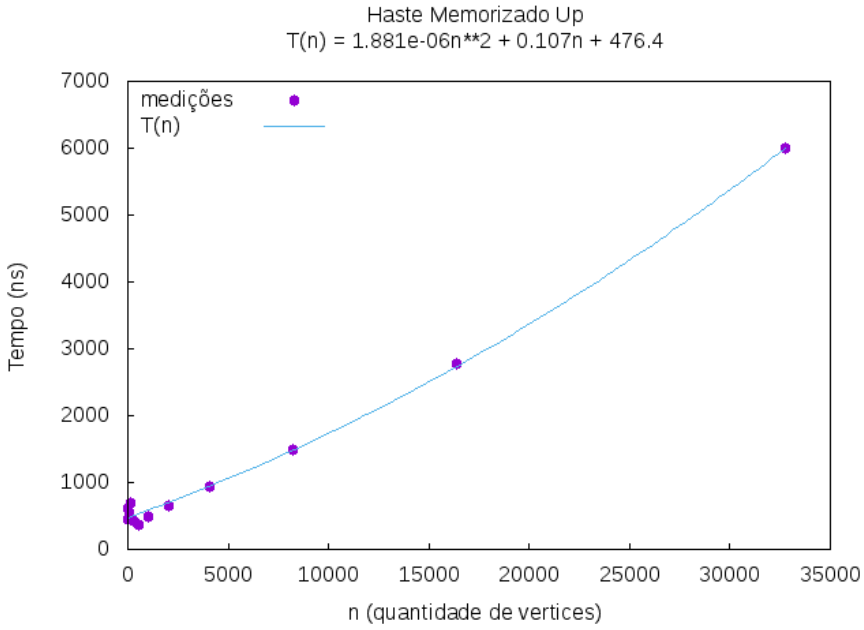

Figura 4.11: *Corte Haste Memoizada - Vetor Decrescente P20*

4.4.10 Vetor Decrescente P30

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma decrescente P30.

Tabela 4.12: *Corte Haste Memoizada com Vetor Decrescente P30*

Número de Elementos	Tempo de execução em nanosegundos
16	701
32	337
64	350
128	367
256	365
512	631
1024	493
2048	803
4096	1145
8192	2082
16384	4729

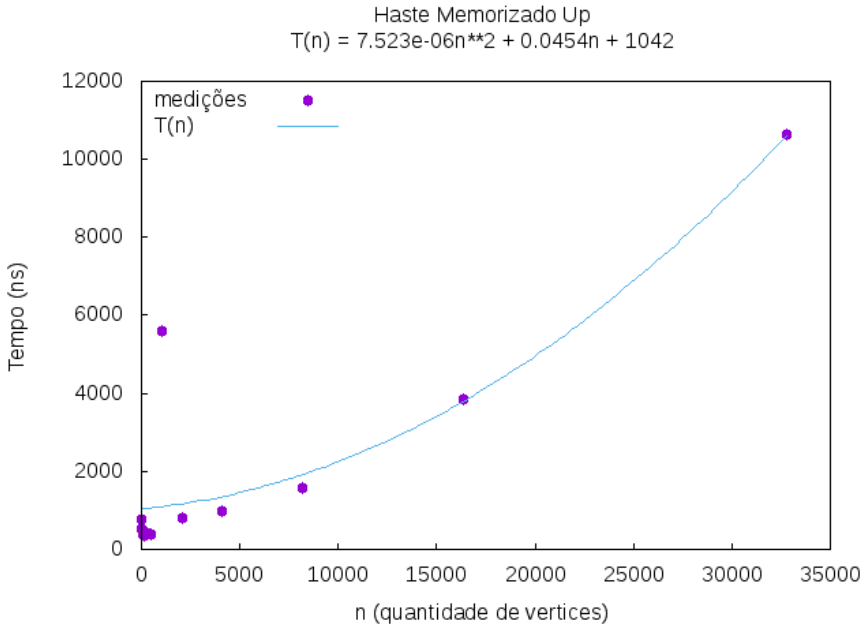

Figura 4.12: *Corte Haste Memoizada - Vetor Decrescente P30*

4.4.11 Vetor Decrescente P40

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma decrescente P40.

Tabela 4.13: *Corte Haste Memoizada com Vetor Decrescente P40*

Número de Elementos	Tempo de execução em nanosegundos
16	357
32	362
64	364
128	372
256	392
512	427
1024	545
2048	604
4096	1085
8192	13662
16384	16253

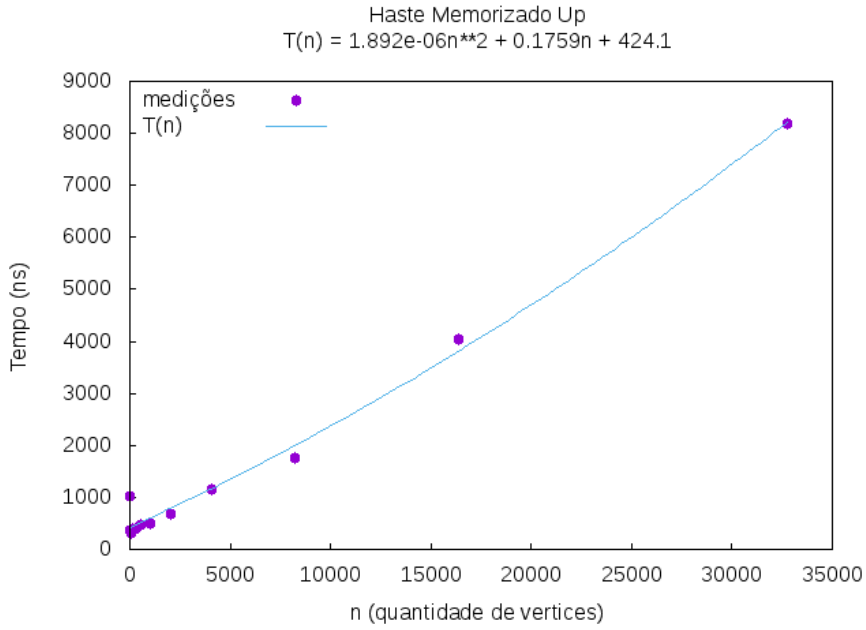

Figura 4.13: *Corte Haste Memoizada - Vetor Decrescente P40*

4.4.12 Vetor Decrescente P50

Tabela gerada utilizando Corte Haste Memoizada com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos de forma decrescente P50.

Tabela 4.14: *Corte Haste Memoizada com Vetor Decrescente P50*

Número de Elementos	Tempo de execução em nanosegundos
16	372
32	650
64	1251
128	2654
256	5659
512	11256
1024	20265
2048	35654
4096	71565
8192	156984
16384	305465

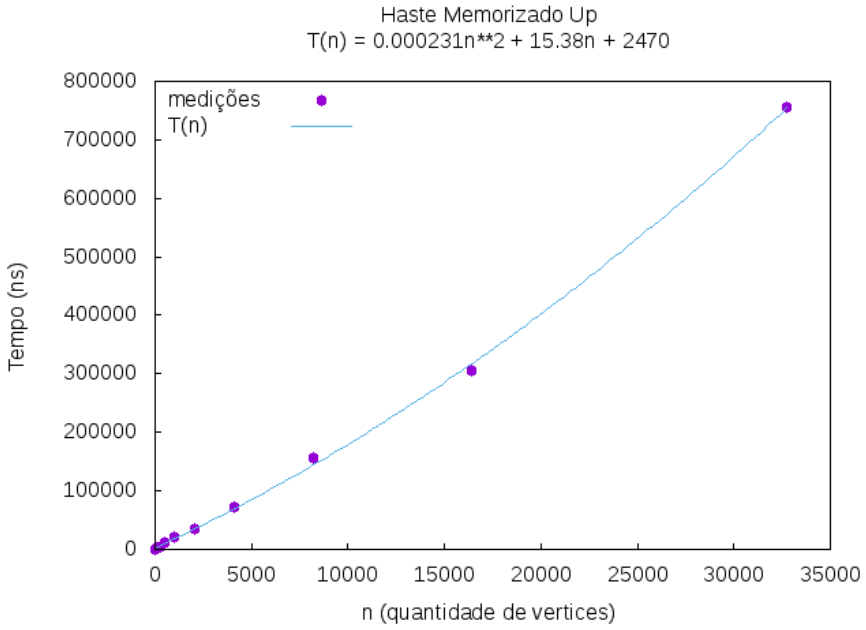

Figura 4.14: *Corte Haste Memoizada - Vetor Decrescente P50*

4.5 SCM

Suponha que A[1..n] é uma sequência de números naturais. Uma subsequência de A[1..n] é o que sobra depois que um conjunto arbitrário de termos é apagado. (Não confunda subsequência com segmento: um segmento de A[1..n] é o que sobra depois que apagamos um número arbitrário de termos no início de A e um número arbitrário de termos no fim de A).

4.5.1 Vetor caracteres

Tabela gerada utilizando SCM com vetores de tamanho n, sendo n = (2^k), de k = 4..14.

Tabela 4.15: *SCM com vetor de caracteres*

Número de Elementos	Tempo de execução em nanosegundos
16	7047
32	12406
64	23325
128	53892
256	111991
512	181464
1024	356165
2048	708123
4096	1434151
8192	3129872
16384	6078938

Nanosegundos/SMCNanosegundos.png Nanosegundos/SMCNanosegundos.png

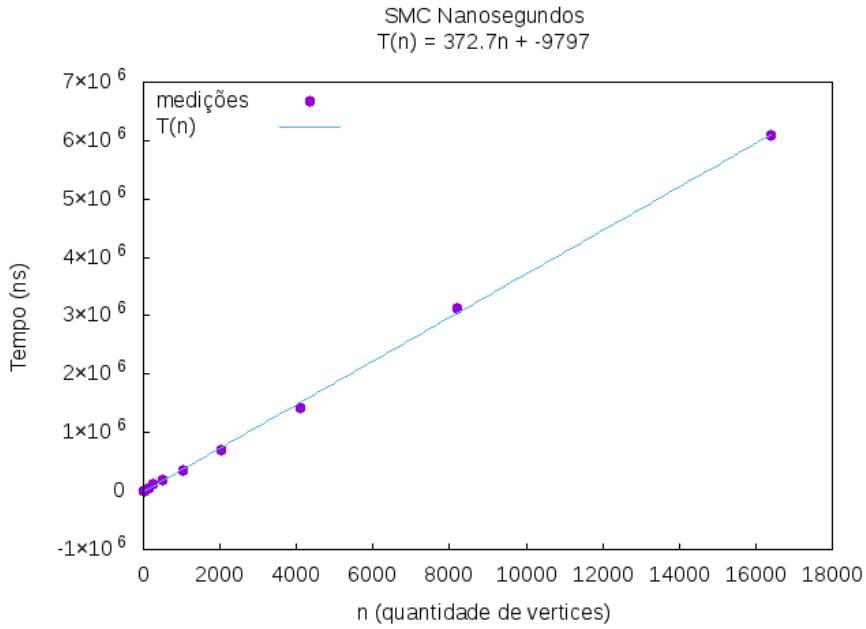

Figura 4.15: *SCM - Vetor caracteres*

4.6 SCM Recursivo

Problema já comentado anteriormente.

4.6.1 Vetor caracteres

Tabela gerada utilizando SCM Recursivo com vetores de tamanho n, sendo n = (2^k), de k = 4..14. Não foi possível retirar conclusões, pois estoura a pilha.

Tabela 4.16: *SCM Recursivo com vetor de caracteres*

Número de Elementos	Tempo de execução em nanosegundos
10	1235274
20	1273575975
30	1080524745749

4.7 Parentização Bottom Up

A forma com a qual a parentizamos uma cadeia de matrizes afeta dramaticamente o custo de calcular o produto. Problema resolvido com uma arvore Bottom Up.

4.7.1 Vetor aleatorio

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos aleatóriamente.

Tabela 4.17: *Parentização Bottom Up com vetor aleatório*

Número de Elementos	Tempo de execução em nanosegundos
16	3216
32	19174
64	118943
128	866385
256	6978135
512	69493017

BottomUp/Aleatorio/ParentizacaoBottomUp.png
BottomUp/Aleatorio/ParentizacaoBottomUp.png

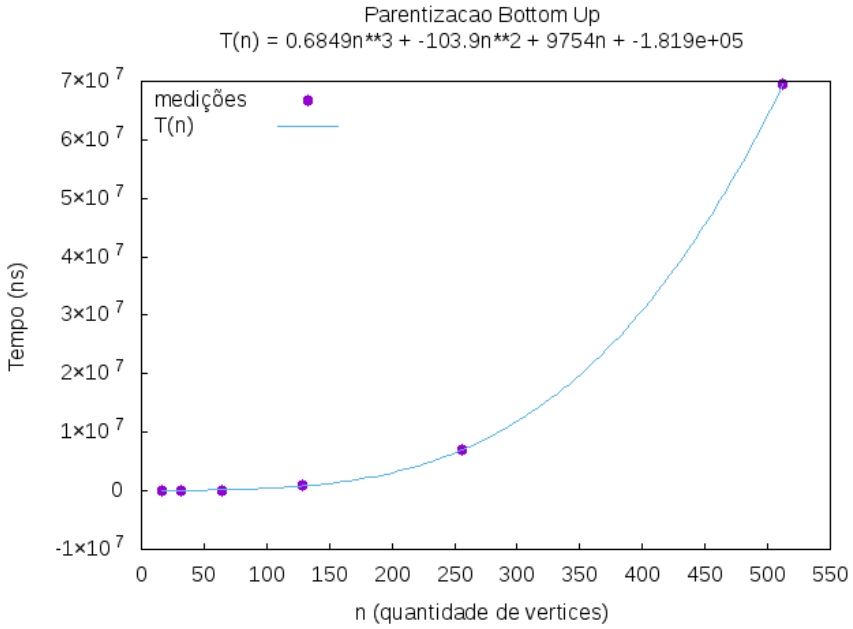

Figura 4.16: *Parentização Bottom Up - Vetor Aleatório*

4.7.2 Vetor Crescente

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente.

Tabela 4.18: *Parentização Bottom Up com vetor Crescente*

Número de Elementos	Tempo de execução em nanosegundos
16	2552
32	13783
64	93670
128	746015
256	6597644
512	67582370

BottomUp/Crescente/ParentizacaoBottomUp.png
BottomUp/Crescente/ParentizacaoBottomUp.png

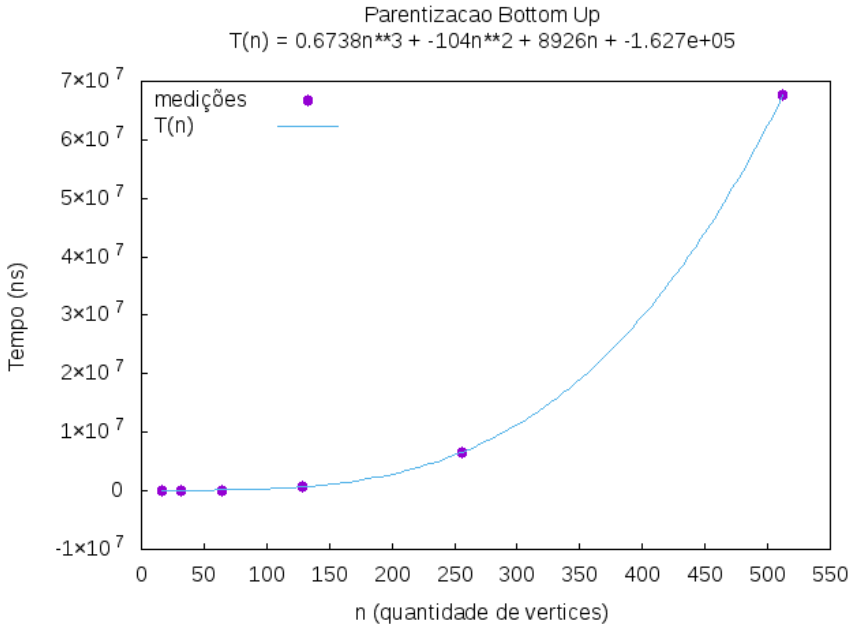

Figura 4.17: *Parentização Bottom Up - Vetor Crescente*

4.7.3 Vetor Crescente P10

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P10.

Tabela 4.19: *Parentização Bottom Up com vetor Crescente P10*

Número de Elementos	Tempo de execução em nanosegundos
16	2467
32	13724
64	92732
128	743649
256	6495094
512	67519336

BottomUp/Crescente P10/ParentizacaoBottomUp.png BottomUp/Crescente P10/ParentizacaoBottomUp.png

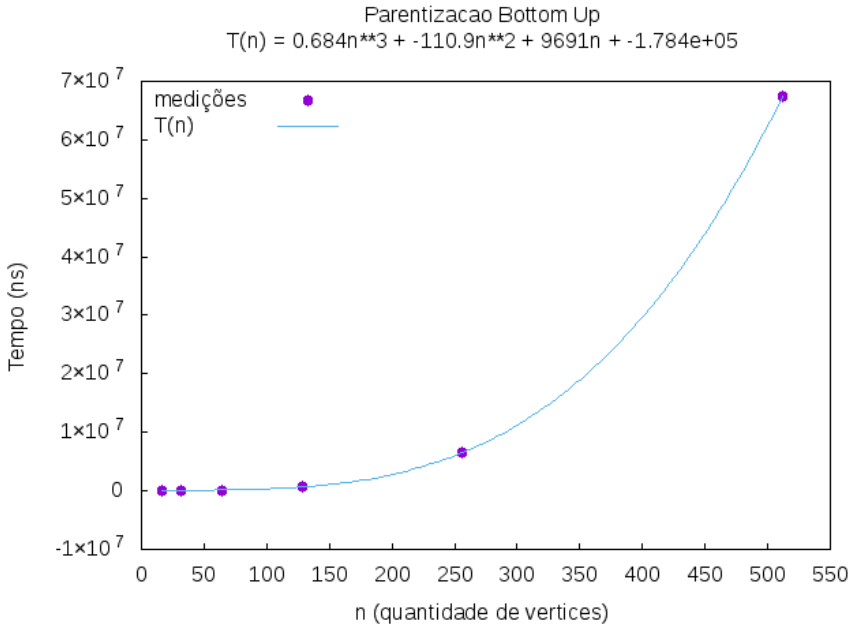

Figura 4.18: *Parentização Bottom Up - Vetor Crescente P10*

4.7.4 Vetor Crescente P20

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P20.

Tabela 4.20: *Parentização Bottom Up com vetor Crescente P20*

Número de Elementos	Tempo de execução em nanosegundos
16	2413
32	14015
64	97281
128	723129
256	6471911
512	67329970

BottomUp/Crescente P20/ParentizacaoBottomUp.png BottomUp/Crescente P20/ParentizacaoBottomUp.png

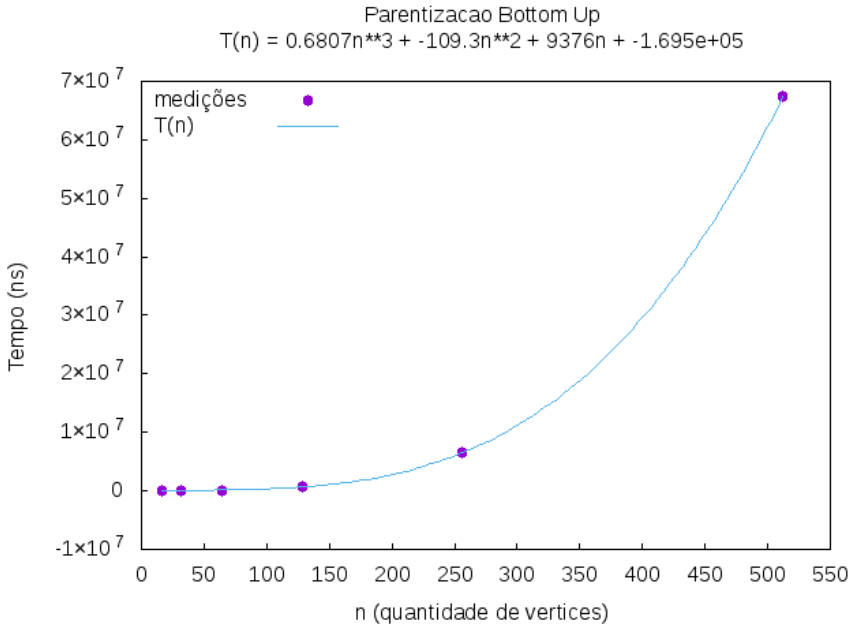

Figura 4.19: *Parentização Bottom Up - Vetor Crescente P20*

4.7.5 Vetor Crescente P30

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P30.

Tabela 4.21: *Parentização Bottom Up com vetor Crescente P30*

Número de Elementos	Tempo de execução em nanosegundos
16	2446
32	14459
64	96241
128	731624
256	6419155
512	67127224

BottomUp/Crescente P30/ParentizacaoBottomUp.png BottomUp/Crescente P30/ParentizacaoBottomUp.png

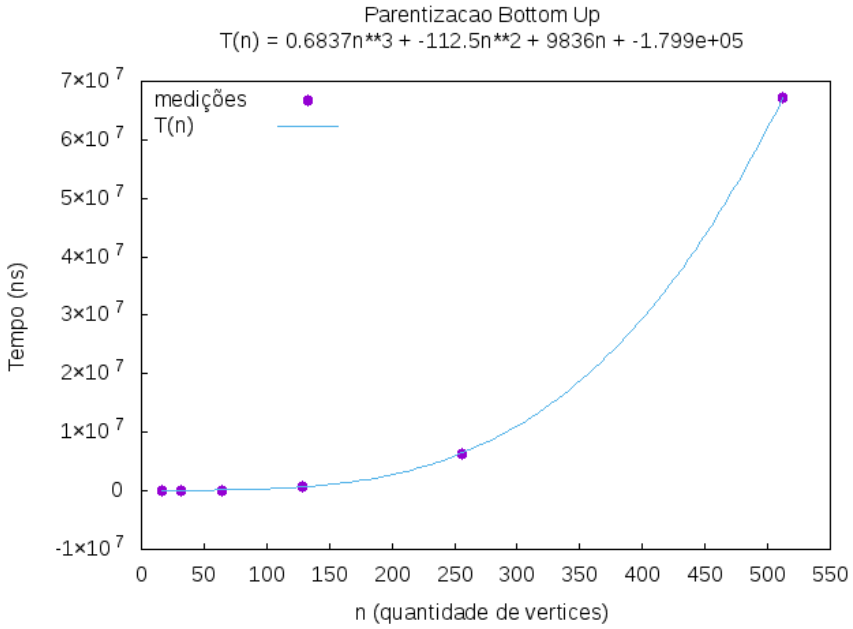

Figura 4.20: *Parentização Bottom Up - Vetor Crescente P30*

4.7.6 Vetor Crescente P40

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P40.

Tabela 4.22: *Parentização Bottom Up com vetor Crescente P40*

Número de Elementos	Tempo de execução em nanosegundos
16	2475
32	14317
64	99423
128	738641
256	6484759
512	67394010

BottomUp/Crescente P40/ParentizacaoBottomUp.png BottomUp/Crescente P40/ParentizacaoBottomUp.png

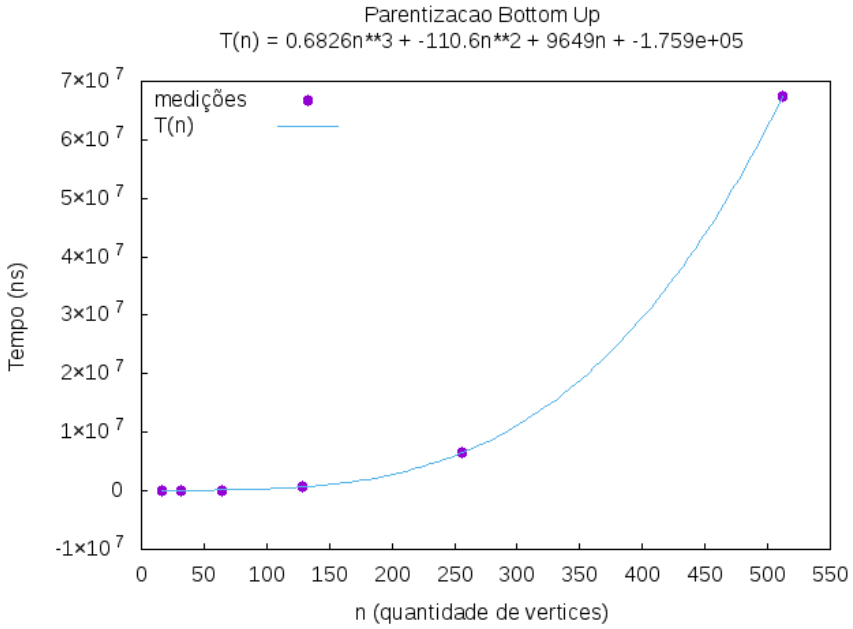

Figura 4.21: *Parentização Bottom Up - Vetor Crescente P40*

4.7.7 Vetor Crescente P50

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P50.

Tabela 4.23: *Parentização Bottom Up com vetor Crescente P50*

Número de Elementos	Tempo de execução em nanosegundos
16	3150
32	13708
64	93130
128	727210
256	6487127
512	67246806

BottomUp/Crescente P50/ParentizacaoBottomUp.png BottomUp/Crescente P50/ParentizacaoBottomUp.png

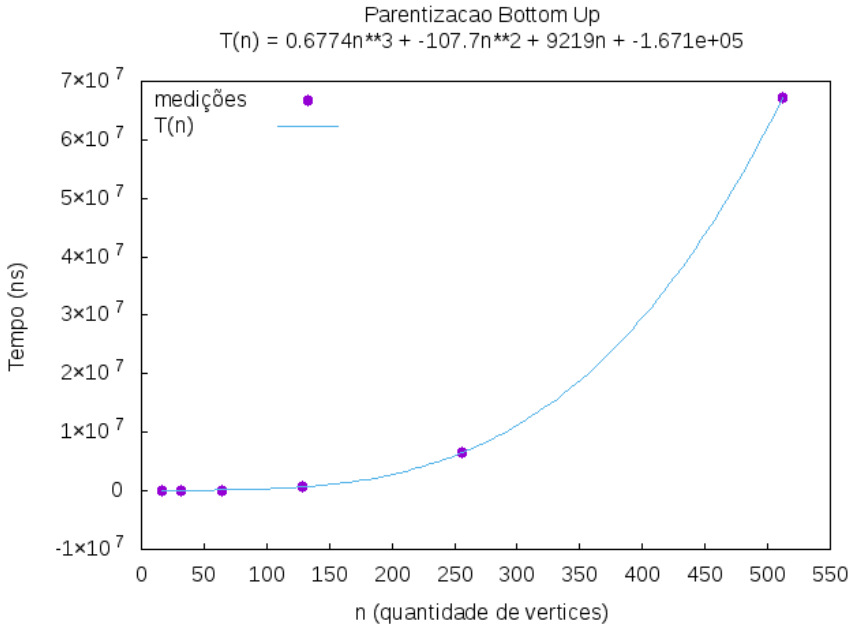

Figura 4.22: *Parentização Bottom Up - Vetor Crescente P50*

4.7.8 Vetor Decrescente

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente.

Tabela 4.24: *Parentização Bottom Up com vetor Decrescente*

Número de Elementos	Tempo de execução em nanosegundos
16	2230
32	11873
64	94244
128	690616
256	6290244
512	67357198

BottomUp/Decrescente/ParentizacaoBottomUp.png
BottomUp/Decrescente/ParentizacaoBottomUp.png

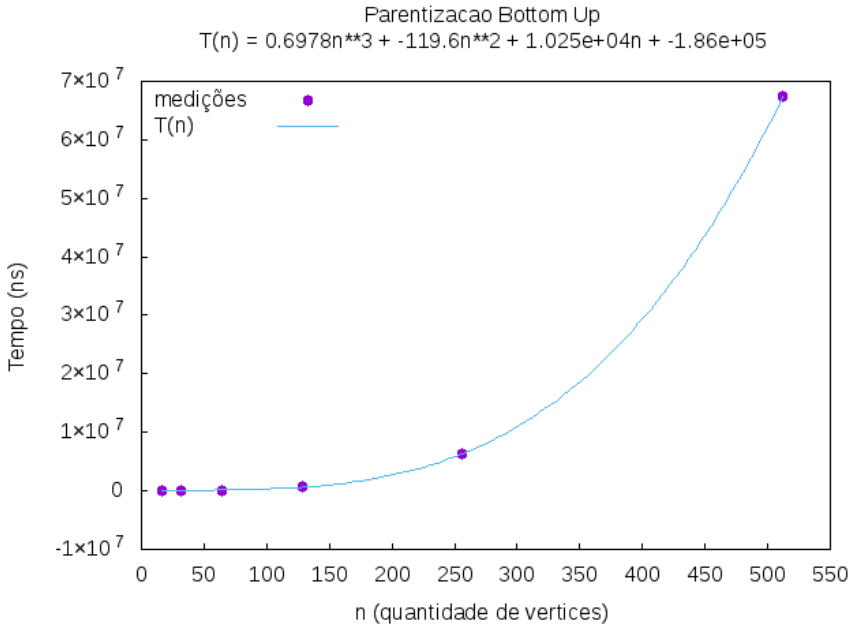

Figura 4.23: *Parentização Bottom Up - Vetor Decrescente*

4.7.9 Vetor Decrescente P10

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P10.

Tabela 4.25: *Parentização Bottom Up com vetor Decrescente P10*

Número de Elementos	Tempo de execução em nanosegundos
16	3662
32	13509
64	104326
128	723638
256	6348032
512	66932636

BottomUp/Decrescente P10/ParentizacaoBottomUp.png BottomUp/Decrescente P10/ParentizacaoBottomUp.png

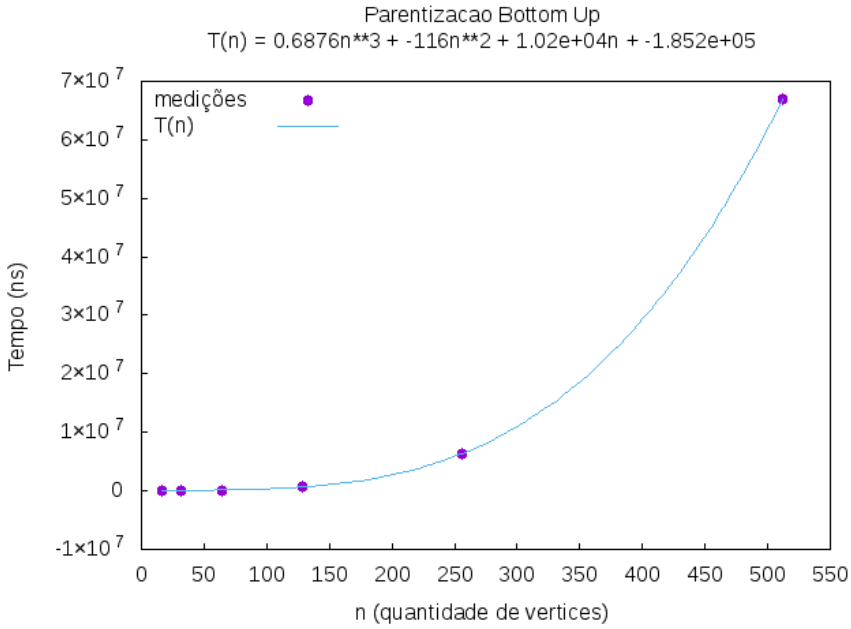

Figura 4.24: *Parentização Bottom Up - Vetor Decrescente P10*

4.7.10 Vetor Decrescente P20

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P20.

Tabela 4.26: *Parentização Bottom Up com vetor Decrescente P20*

Número de Elementos	Tempo de execução em nanosegundos
16	2325
32	14062
64	96127
128	731992
256	6474800
512	67105053

BottomUp/Decrescente P20/ParentizacaoBottomUp.png BottomUp/Decrescente P20/ParentizacaoBottomUp.png

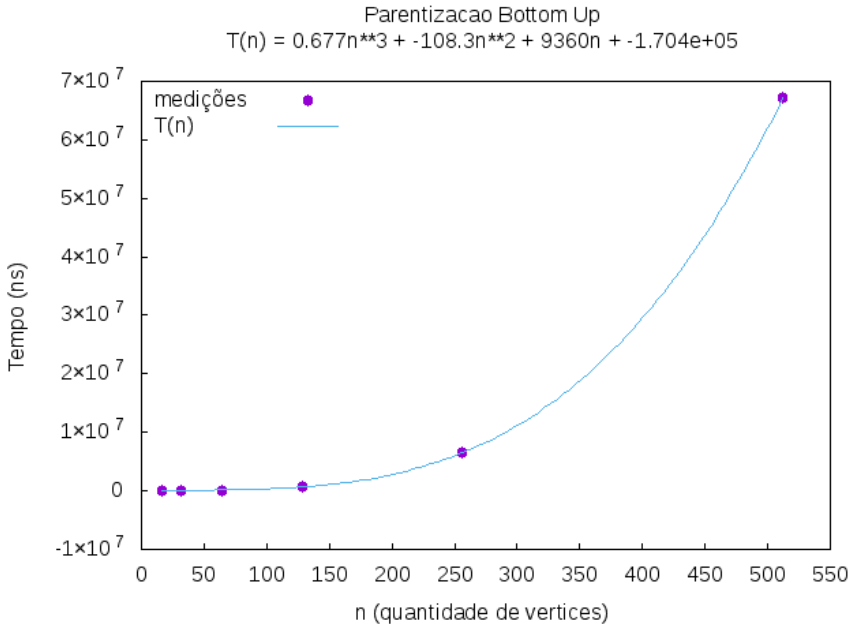

Figura 4.25: *Parentização Bottom Up - Vetor Decrescente P20*

4.7.11 Vetor Decrescente P30

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P30.

Tabela 4.27: *Parentização Bottom Up com vetor Decrescente P30*

Número de Elementos	Tempo de execução em nanosegundos
16	2644
32	14619
64	99600
128	771261
256	6508011
512	67406718

BottomUp/Decrescente P30/ParentizacaoBottomUp.png BottomUp/Decrescente P30/ParentizacaoBottomUp.png

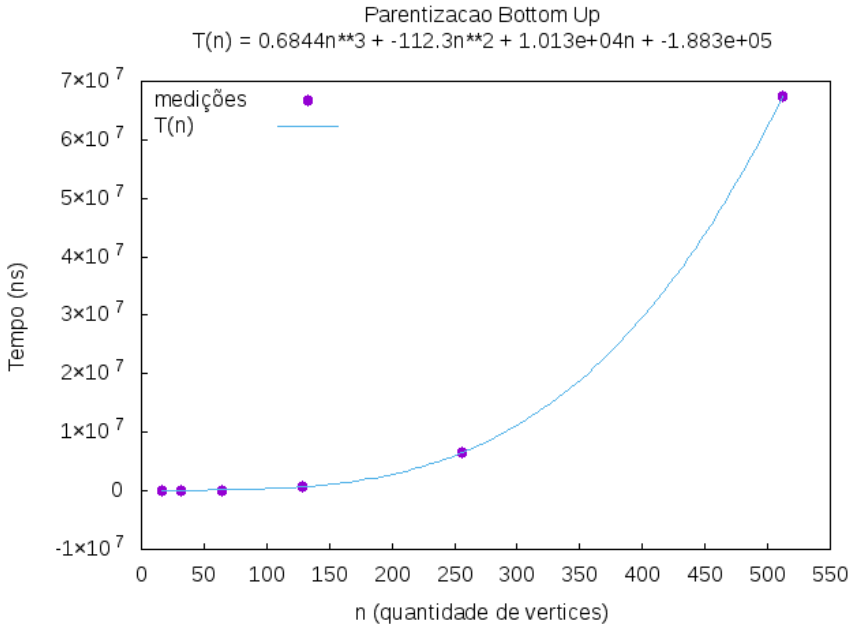

Figura 4.26: *Parentização Bottom Up - Vetor Decrescente P30*

4.7.12 Vetor Decrescente P40

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P40.

Tabela 4.28: *Parentização Bottom Up com vetor Decrescente P40*

Número de Elementos	Tempo de execução em nanosegundos
16	2898
32	15696
64	108110
128	760201
256	6613108
512	67337227

BottomUp/Decrescente P40/ParentizacaoBottomUp.png BottomUp/Decrescente P40/ParentizacaoBottomUp.png

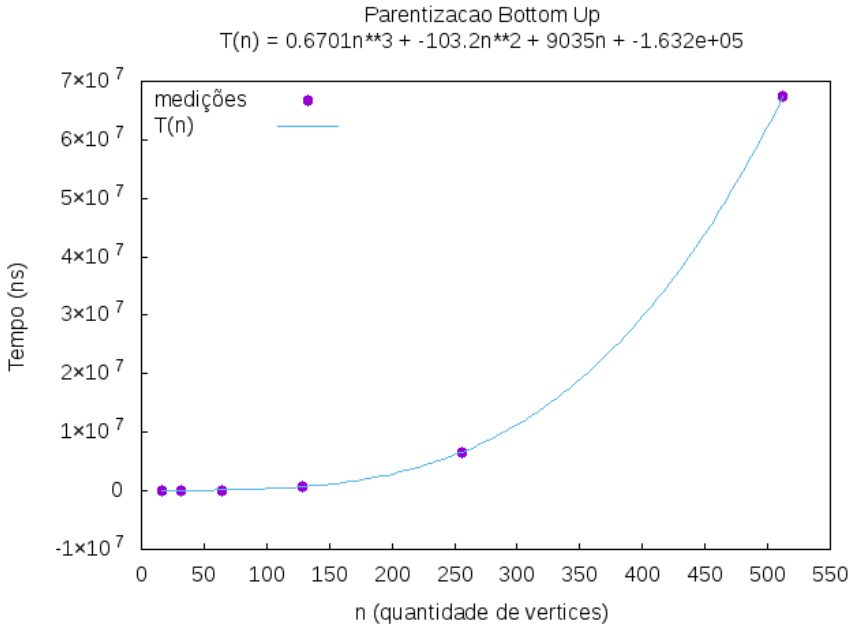

Figura 4.27: *Parentização Bottom Up - Vetor Decrescente P40*

4.7.13 Vetor Decrescente P50

Tabela gerada utilizando Parentização Bottom Up com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P50.

Tabela 4.29: *Parentização Bottom Up com vetor Decrescente P50*

Número de Elementos	Tempo de execução em nanosegundos
16	2961
32	16579
64	105587
128	771134
256	6553364
512	67677413

BottomUp/Decrescente P50/ParentizacaoBottomUp.png BottomUp/Decrescente P50/ParentizacaoBottomUp.png

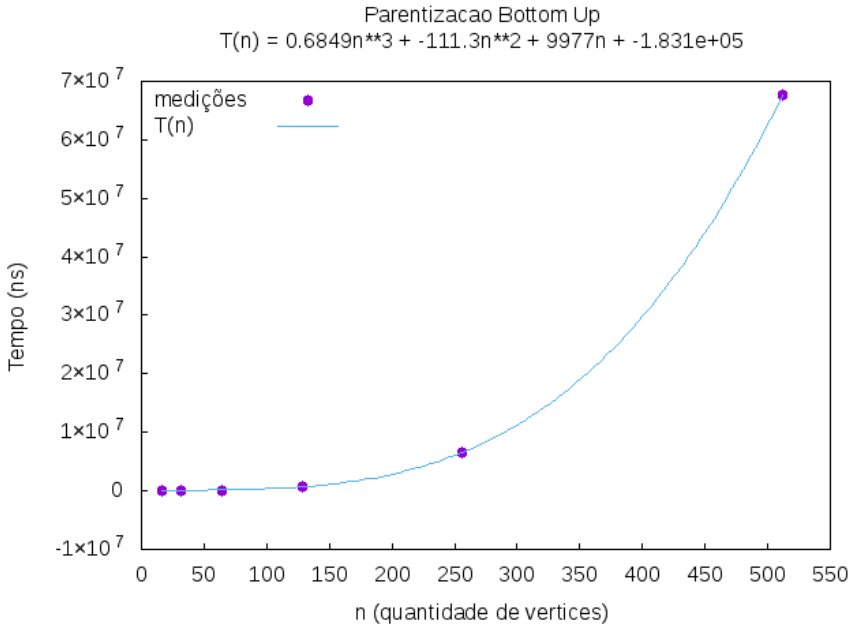

Figura 4.28: *Parentização Bottom Up - Vetor Decrescente P50*

4.8 Parentização Recursiva

Problema já comentado anteriormente.

4.8.1 Vetor

Tabela gerada utilizando Parentização Recursiva com vetores de tamanho n, sendo n = (2^k), de k = 4, pois após isso há um estouro de memória, logo não é possível extrair um gráfico.

Tabela 4.30: *Parentização Recursiva*

Número de Elementos	Tempo de execução em nanosegundos
16	103633175
16	103633175
16	103633175
16	103633175
16	103633175
16	103633175
16	103633175
16	103633175
16	103633175
16	103633175
16	103633175
16	103633175
16	103633175

Capítulo 5

Estatísticas de Ordem

A iesima estatística de ordem de um conjunto com n elementos é o iesimo menor elemento desse conjunto.

5.1 Min

Mínimo é a primeira estatística de ordem, com i = 1.

5.1.1 Vetor aleatorio

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos aleatóriamente.

Tabela 5.1: *Min com vetor aleatório*

Número de Elementos	**Tempo de execução em nanosegundos**
16	520
32	300
64	326
128	358
256	387
512	458
1024	1017
2048	1227
4096	3554
8192	10154
16384	6307

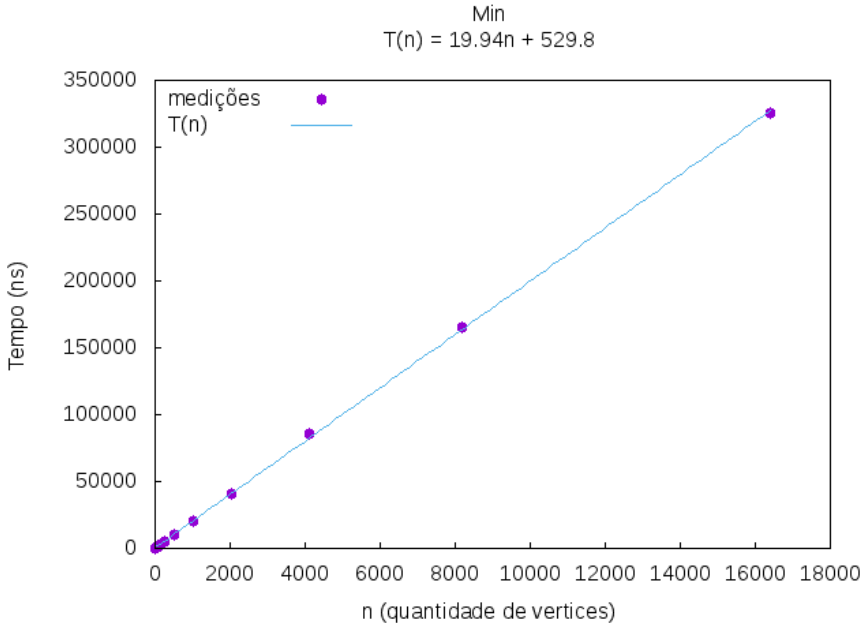

Figura 5.1: *Min - Vetor Aleatório*

5.1.2 Vetor Crescente

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente.

Tabela 5.2: *Min com vetor Crescente*

Número de Elementos	Tempo de execução em nanosegundos
16	315
32	426
64	328
128	336
256	405
512	496
1024	10294
2048	1053
4096	1707
8192	6034
16384	17684

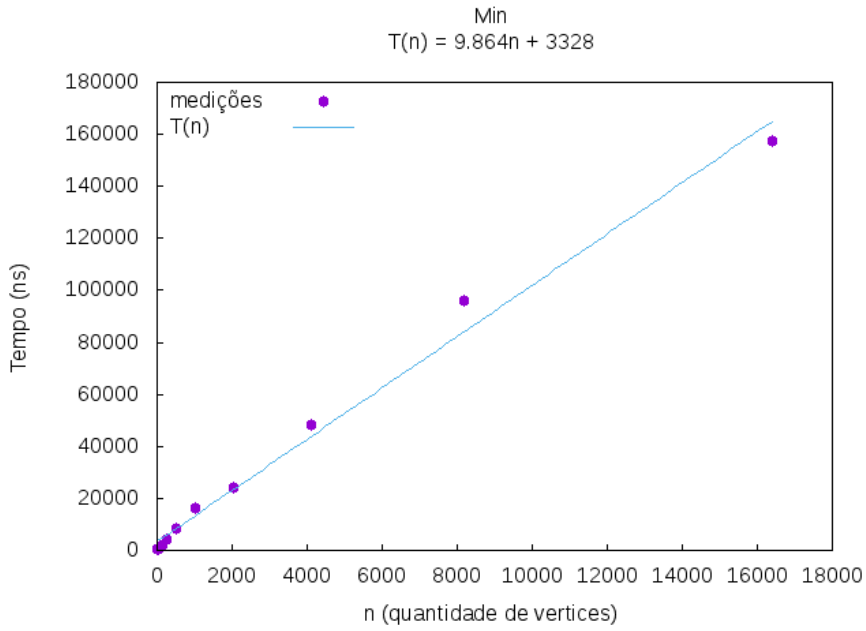

Figura 5.2: *Min - Vetor Crescente*

5.1.3 Vetor Crescente P10

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P10.

Tabela 5.3: *Min com vetor Crescente P10*

Número de Elementos	Tempo de execução em nanosegundos
16	393
32	290
64	285
128	330
256	371
512	638
1024	658
2048	1324
4096	2086
8192	3680
16384	20530

P10/Min.png P10/Min.png

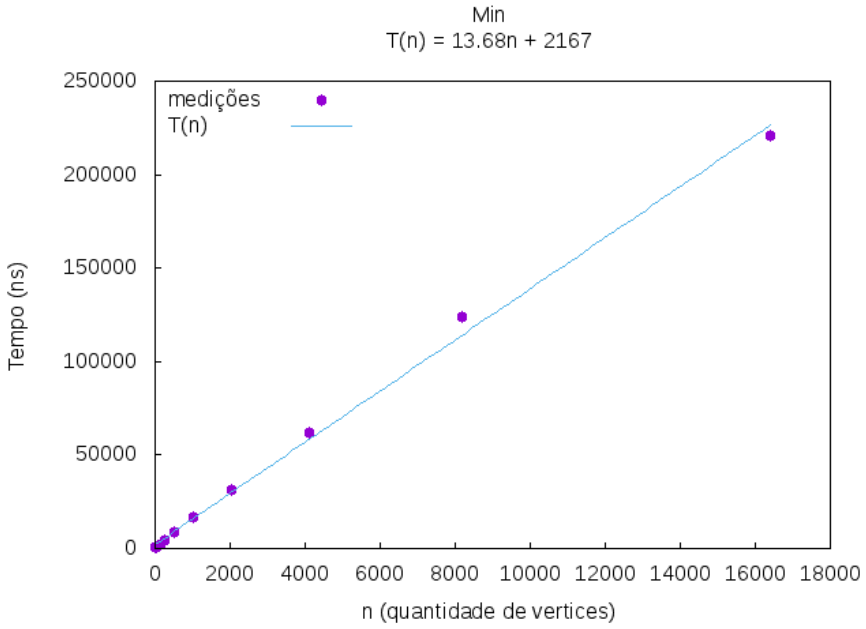

Figura 5.3: *Min - Vetor Crescente P10*

5.1.4 Vetor Crescente P20

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P20.

Tabela 5.4: *Min com vetor Crescente P20*

Número de Elementos	Tempo de execução em nanosegundos
16	805
32	500
64	766
128	486
256	427
512	497
1024	745
2048	1113
4096	3154
8192	2949
16384	5554

P20/Min.png P20/Min.png

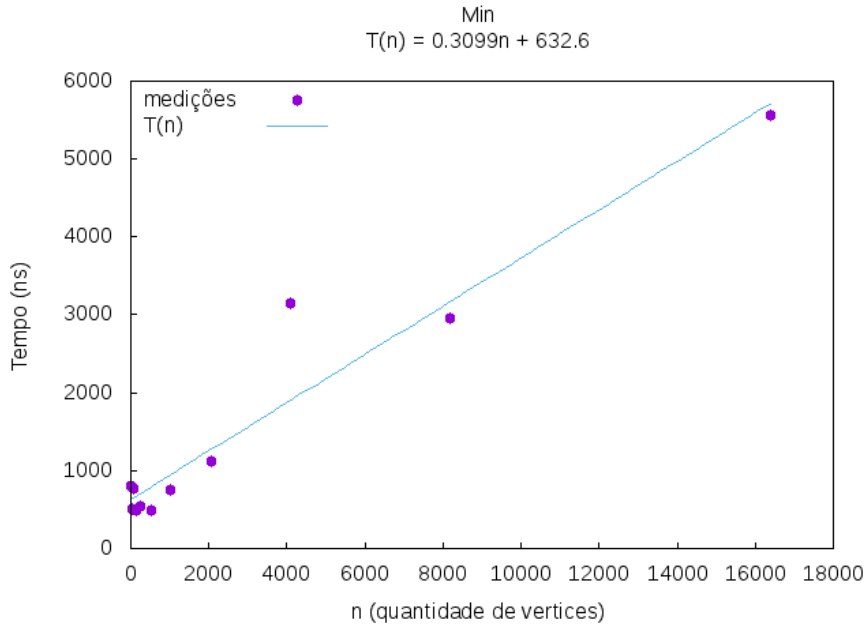

Figura 5.4: *Min - Vetor Crescente P20*

5.1.5 Vetor Crescente P30

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P30.

Tabela 5.5: *Min com vetor Crescente P30*

Número de Elementos	Tempo de execução em nanosegundos
16	304
32	274
64	297
128	466
256	390
512	463
1024	587
2048	891
4096	1554
8192	3553
16384	7562

P30/Min.png P30/Min.png

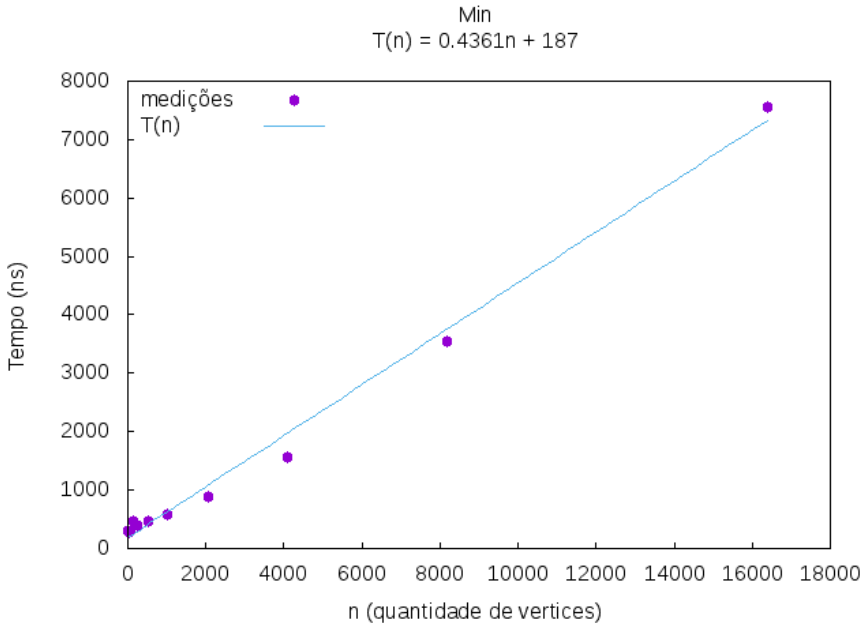

Figura 5.5: *Min - Vetor Crescente P30*

5.1.6 Vetor Crescente P40

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P40.

Tabela 5.6: *Min com vetor Crescente P40*

Número de Elementos	Tempo de execução em nanosegundos
16	332
32	330
64	302
128	350
256	426
512	466
1024	1714
2048	2170
4096	4381
8192	12724
16384	6773

P40/Min.png P40/Min.png

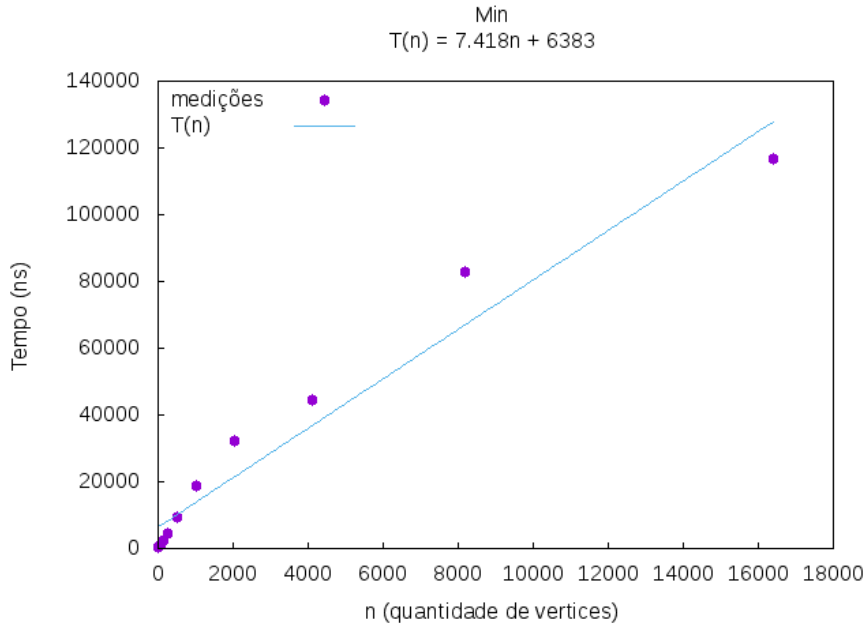

Figura 5.6: *Min - Vetor Crescente P40*

5.1.7 Vetor Crescente P50

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P50.

Tabela 5.7: *Min com vetor Crescente P50*

Número de Elementos	Tempo de execução em nanosegundos
16	312
32	276
64	289
128	372
256	365
512	543
1024	922
2048	922
4096	1538
8192	2925
16384	6012

P50/Min.png P50/Min.png

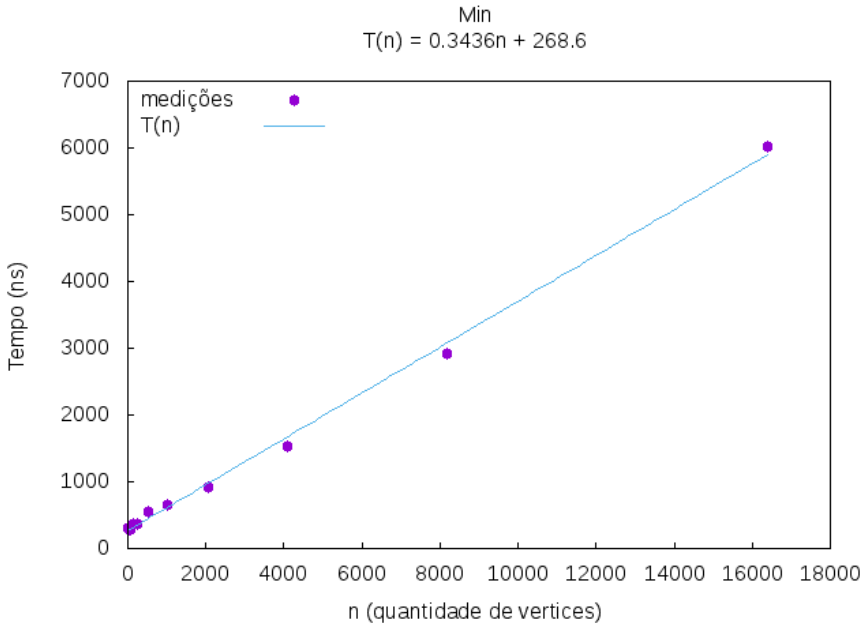

Figura 5.7: *Min - Vetor Crescente P50*

5.1.8 Vetor Decrescente

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente.

Tabela 5.8: *Min com vetor Decrescente*

Número de Elementos	Tempo de execução em nanosegundos
16	305
32	278
64	274
128	340
256	368
512	472
1024	869
2048	1008
4096	2077
8192	3849
16384	11560

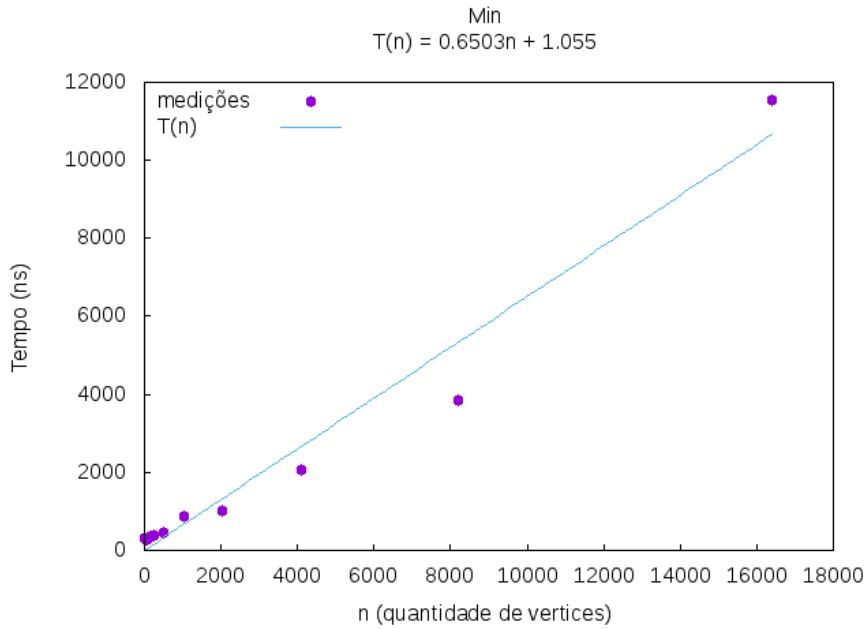

Figura 5.8: *Min - Vetor Decrescente*

5.1.9 Vetor Decrescente P10

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P10.

Tabela 5.9: *Min com vetor Decrescente P10*

Número de Elementos	Tempo de execução em nanosegundos
16	317
32	278
64	294
128	345
256	357
512	466
1024	661
2048	932
4096	1855
8192	4558
16384	6088

P10/Min.png P10/Min.png

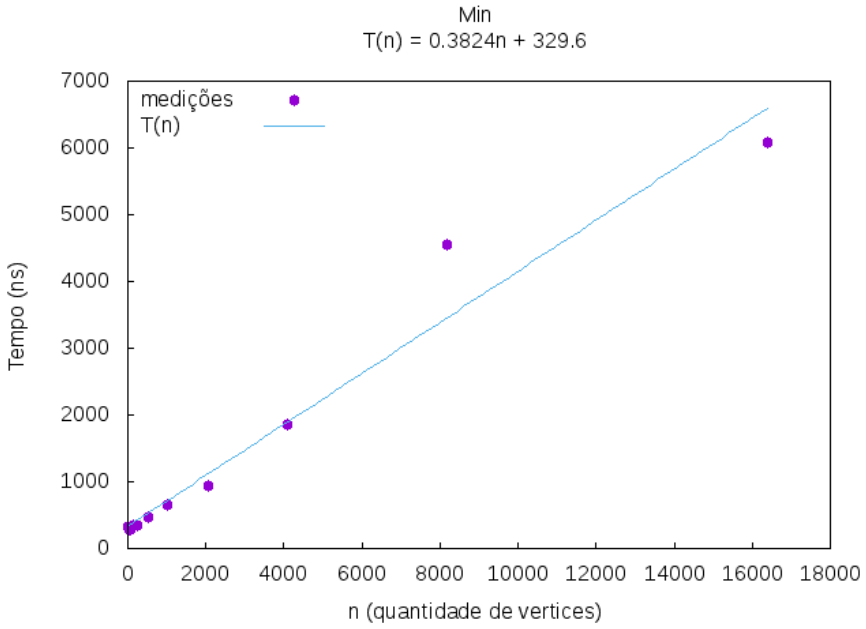

Figura 5.9: *Min - Vetor Decrescente P10*

5.1.10 Vetor Decrescente P20

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P20.

Tabela 5.10: *Min com vetor Decrescente P20*

Número de Elementos	Tempo de execução em nanosegundos
16	331
32	267
64	318
128	361
256	403
512	485
1024	713
2048	892
4096	1790
8192	2965
16384	6367

P20/Min.png P20/Min.png

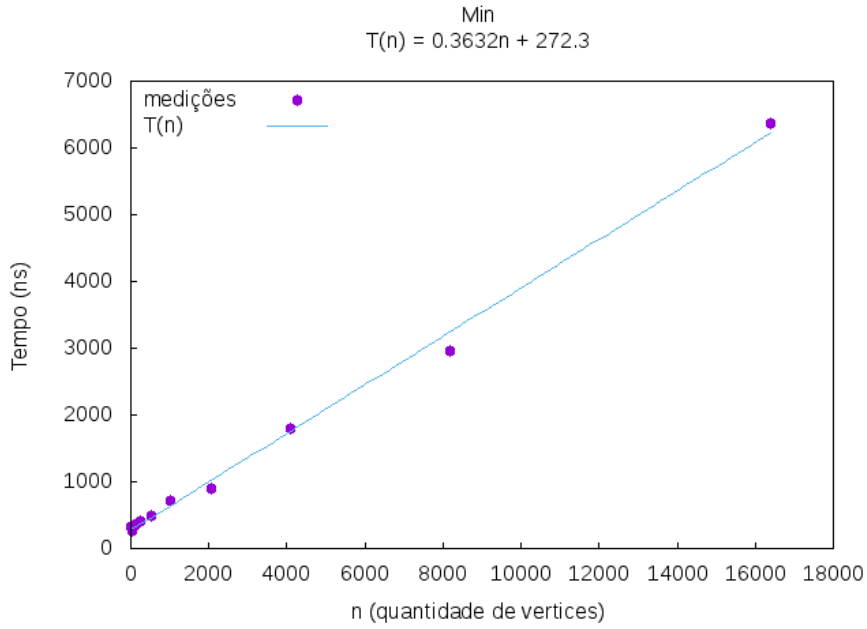

Figura 5.10: *Min - Vetor Decrescente P20*

5.1.11 Vetor Decrescente P30

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P30.

Tabela 5.11: *Min com vetor Decrescente P30*

Número de Elementos	Tempo de execução em nanosegundos
16	335
32	300
64	370
128	364
256	433
512	503
1024	678
2048	1236
4096	2175
8192	9572
16384	13543

P30/Min.png P30/Min.png

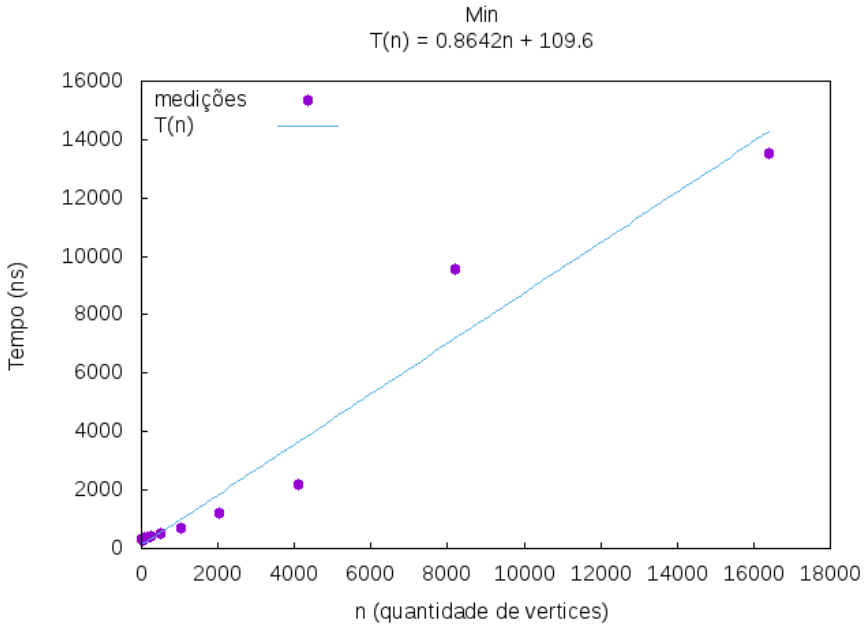

Figura 5.11: *Min - Vetor Decrescente P30*

5.1.12 Vetor Decrescente P40

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P40.

Tabela 5.12: *Min com vetor Decrescente P40*

Número de Elementos	Tempo de execução em nanosegundos
16	323
32	279
64	358
128	342
256	363
512	469
1024	659
2048	899
4096	1831
8192	2947
16384	5499

P40/Min.png P40/Min.png

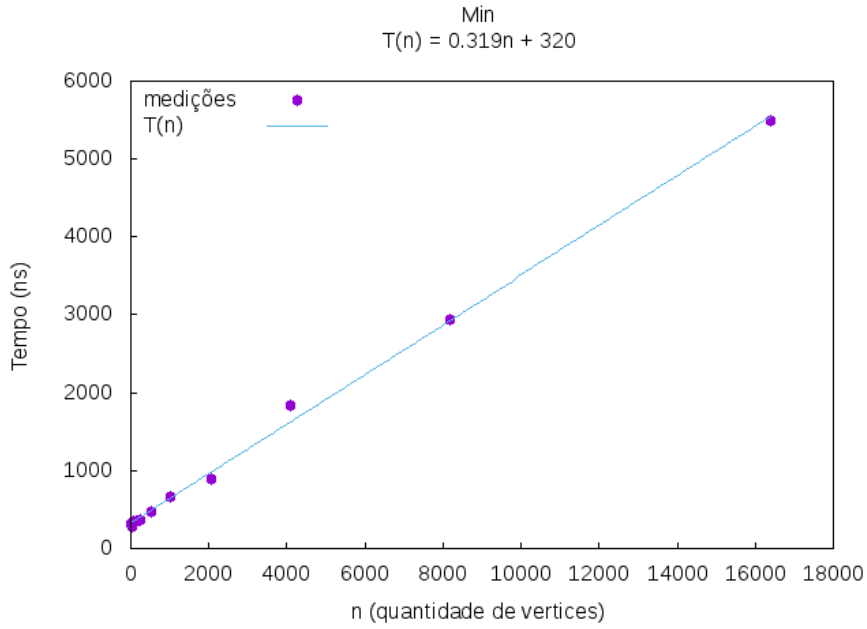

Figura 5.12: *Min - Vetor Decrescente P40*

5.1.13 Vetor Decrescente P50

Tabela gerada utilizando Min com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P50.

Tabela 5.13: *Min com vetor Decrescente P50*

Número de Elementos	Tempo de execução em nanosegundos
16	341
32	300
64	320
128	361
256	422
512	460
1024	637
2048	1070
4096	1853
8192	2951
16384	7247

P50/Min.png P50/Min.png

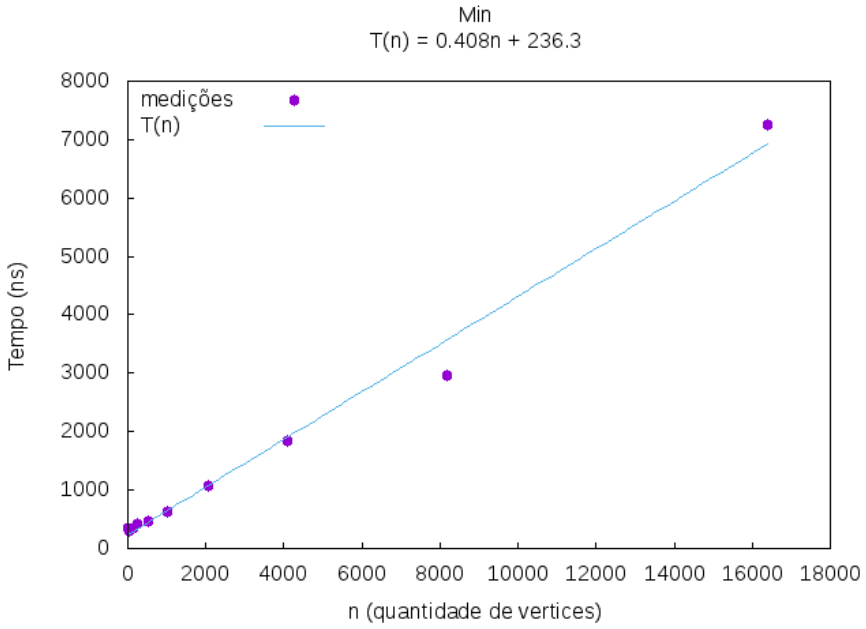

Figura 5.13: *Min - Vetor Decrescente P50*

5.2 MinMax

Mínimo é a primeira estatística de ordem, com i = 1. Máximo é a n ésima estatística de ordem, com i = n.

5.2.1 Vetor aleatorio

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos aleatóriamente.

Tabela 5.14: *MinMax com vetor aleatório*

Número de Elementos	Tempo de execução em nanosegundos
16	834
32	444
64	460
128	455
256	545
512	685
1024	974
2048	1322
4096	2133
8192	3744
16384	8385

Max/Aleatorio/MinMax.png Max/Aleatorio/MinMax.png

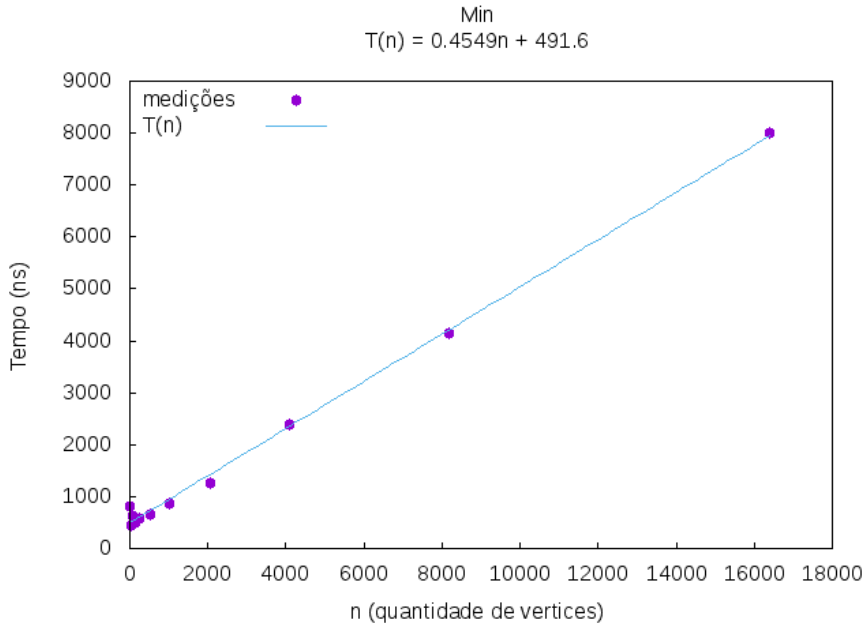

Figura 5.14: *MinMax - Vetor Aleatório*

5.2.2 Vetor Crescente

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente.

Tabela 5.15: *MinMax com vetor Crescente*

Número de Elementos	Tempo de execução em nanosegundos
16	554
32	442
64	460
128	457
256	508
512	672
1024	1444
2048	1928
4096	10075
8192	8232
16384	8076

Max/Crescente/MinMax.png Max/Crescente/MinMax.png

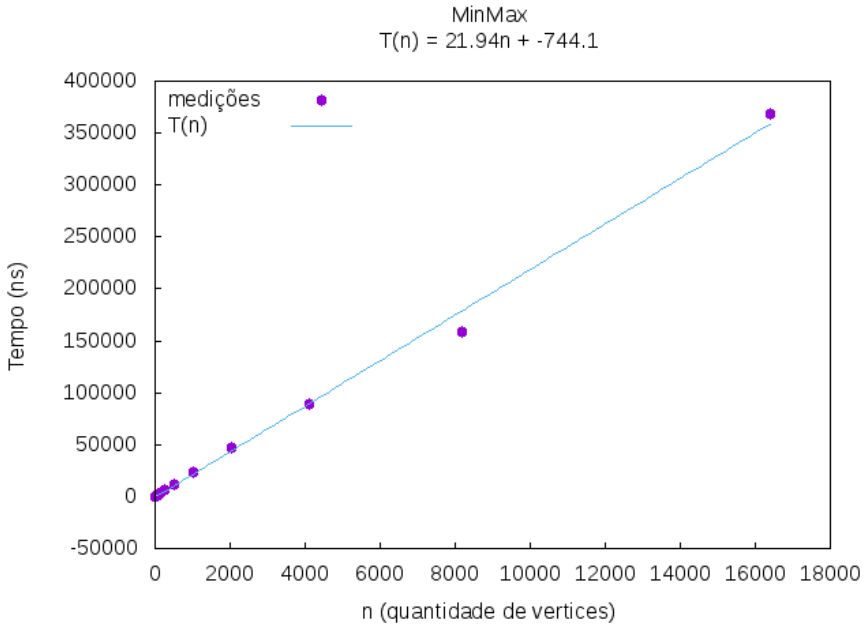

Figura 5.15: *MinMax - Vetor Crescente*

5.2.3 Vetor Crescente P10

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P10.

Tabela 5.16: *MinMax com vetor Crescente P10*

Número de Elementos	Tempo de execução em nanosegundos
16	510
32	472
64	513
128	434
256	551
512	712
1024	881
2048	1535
4096	2929
8192	8249
16384	22516

Max/Crescente P10/MinMax.png Max/Crescente P10/MinMax.png

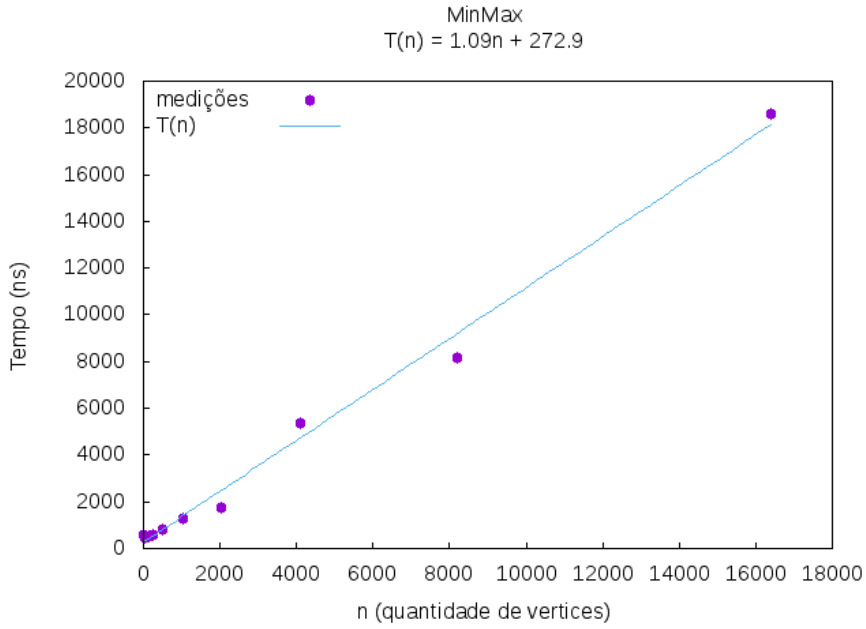

Figura 5.16: *MinMax - Vetor Crescente P10*

5.2.4 Vetor Crescente P20

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P20.

Tabela 5.17: *MinMax com vetor Crescente P20*

Número de Elementos	Tempo de execução em nanosegundos
16	454
32	465
64	461
128	472
256	525
512	650
1024	1049
2048	1335
4096	2398
8192	3837
16384	7720

Max/Crescente P20/MinMax.png Max/Crescente P20/MinMax.png

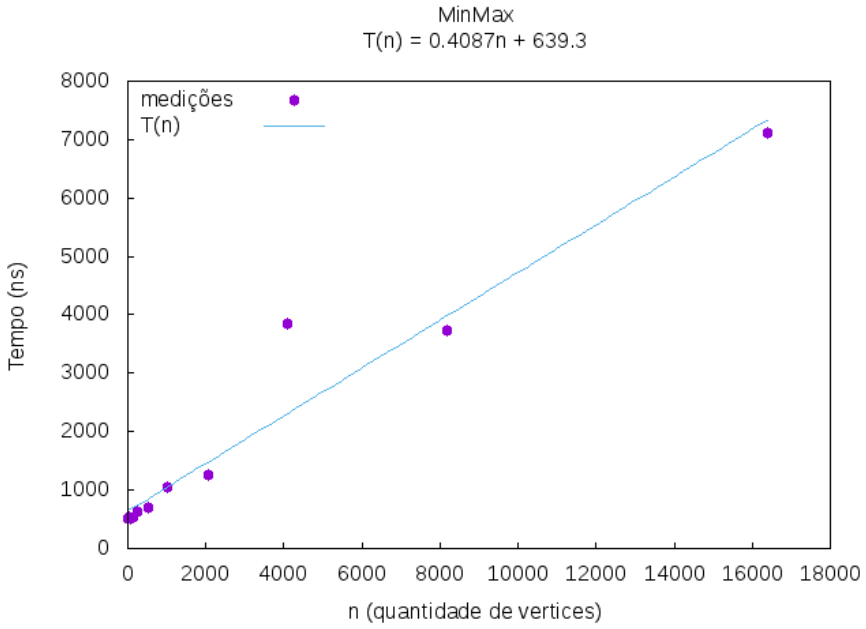

Figura 5.17: *MinMax - Vetor Crescente P20*

5.2.5 Vetor Crescente P30

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P30.

Tabela 5.18: *MinMax com vetor Crescente P30*

Número de Elementos	Tempo de execução em nanosegundos
16	445
32	533
64	484
128	423
256	544
512	676
1024	847
2048	1323
4096	2201
8192	3786
16384	10934

Max/Crescente P30/MinMax.png Max/Crescente P30/MinMax.png

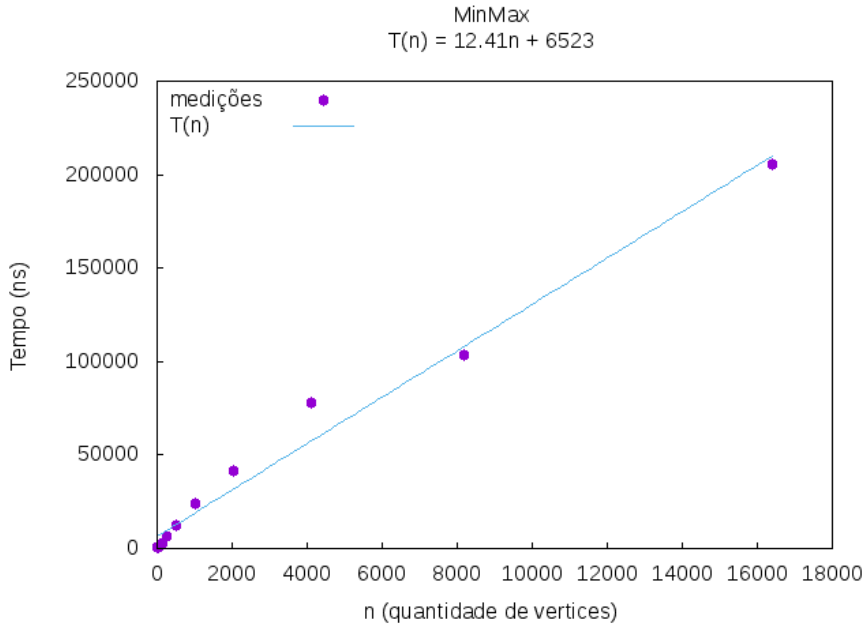

Figura 5.18: *MinMax - Vetor Crescente P30*

5.2.6 Vetor Crescente P40

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P40.

Tabela 5.19: *MinMax com vetor Crescente P40*

Número de Elementos	Tempo de execução em nanosegundos
16	1292
32	483
64	573
128	807
256	562
512	1705
1024	2362
2048	3904
4096	5994
8192	4047
16384	7145

Max/Crescente P40/MinMax.png Max/Crescente P40/MinMax.png

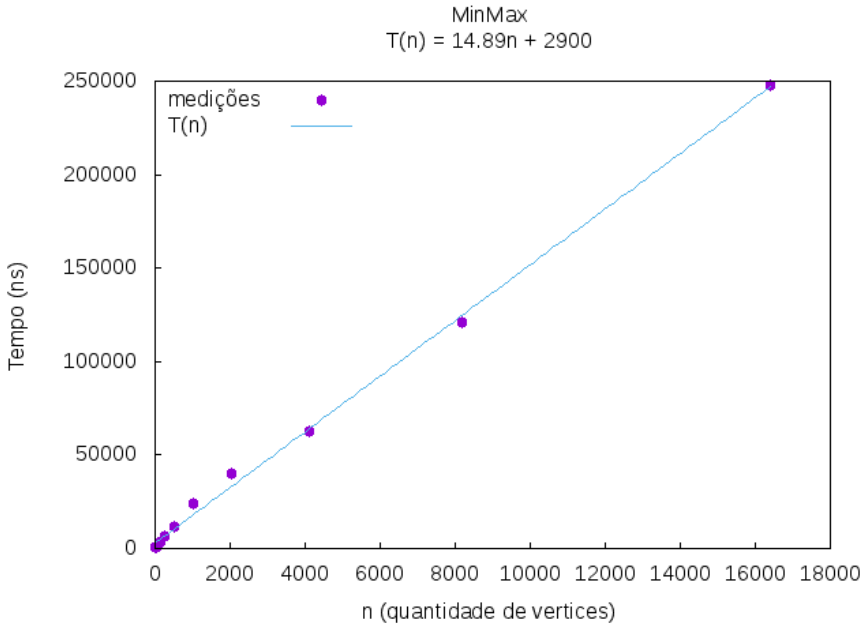

Figura 5.19: *MinMax - Vetor Crescente P40*

5.2.7 Vetor Crescente P50

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Crescente P50.

Tabela 5.20: *MinMax com vetor Crescente P50*

Número de Elementos	Tempo de execução em nanosegundos
16	460
32	463
64	464
128	496
256	948
512	688
1024	854
2048	1308
4096	2643
8192	5058
16384	7017

Max/Crescente P50/MinMax.png Max/Crescente P50/MinMax.png

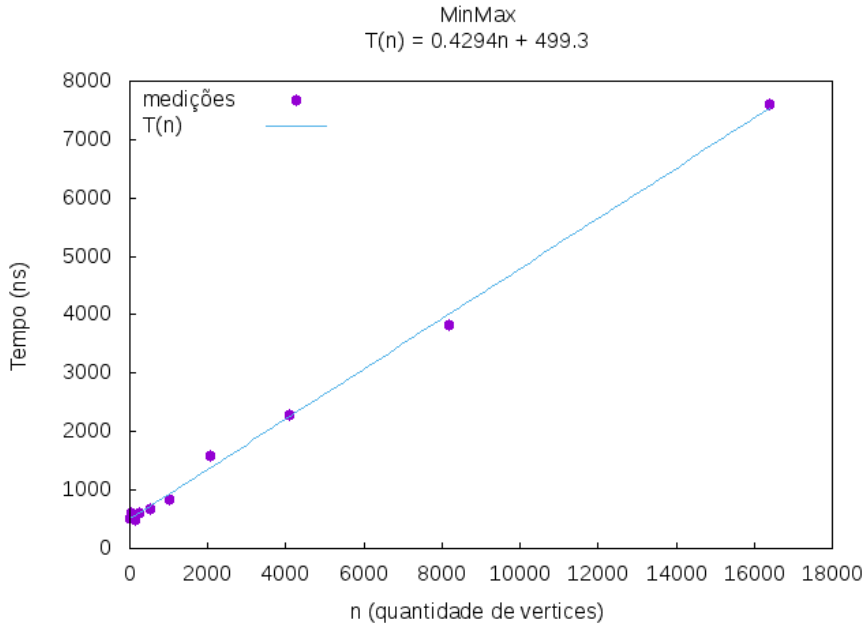

Figura 5.20: *MinMax - Vetor Crescente P50*

5.2.8 Vetor Decrescente

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente.

Tabela 5.21: *MinMax com vetor Decrescente*

Número de Elementos	Tempo de execução em nanosegundos
16	440
32	394
64	488
128	444
256	511
512	659
1024	1000
2048	1525
4096	2986
8192	6276
16384	15652

Max/Decrescente/MinMax.png Max/Decrescente/MinMax.png

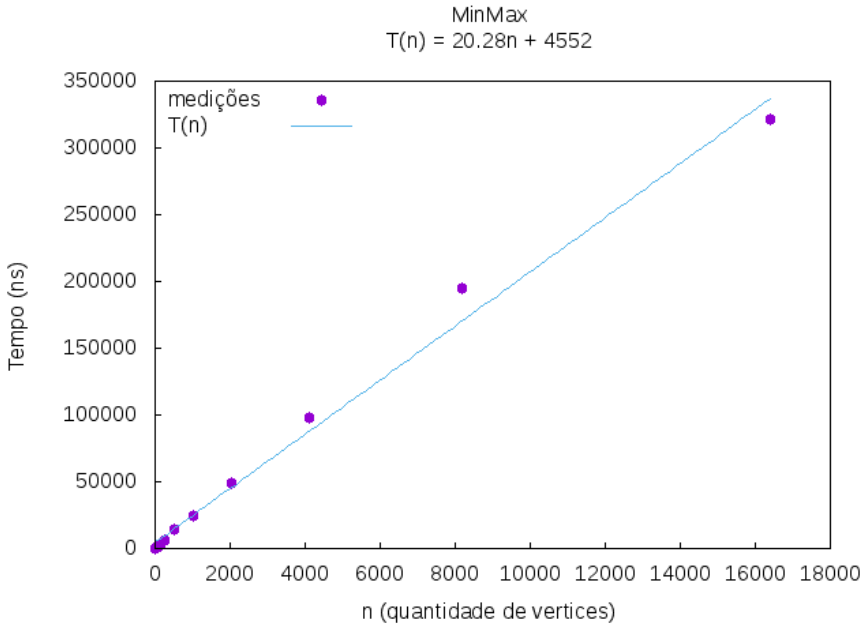

Figura 5.21: *MinMax - Vetor Decrescente*

5.2.9 Vetor Decrescente P10

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P10.

Tabela 5.22: *MinMax com vetor Decrescente P10*

Número de Elementos	Tempo de execução em nanosegundos
16	448
32	428
64	445
128	599
256	488
512	656
1024	876
2048	1299
4096	2130
8192	3922
16384	8388

Max/Decrescente P10/MinMax.png Max/Decrescente P10/MinMax.png

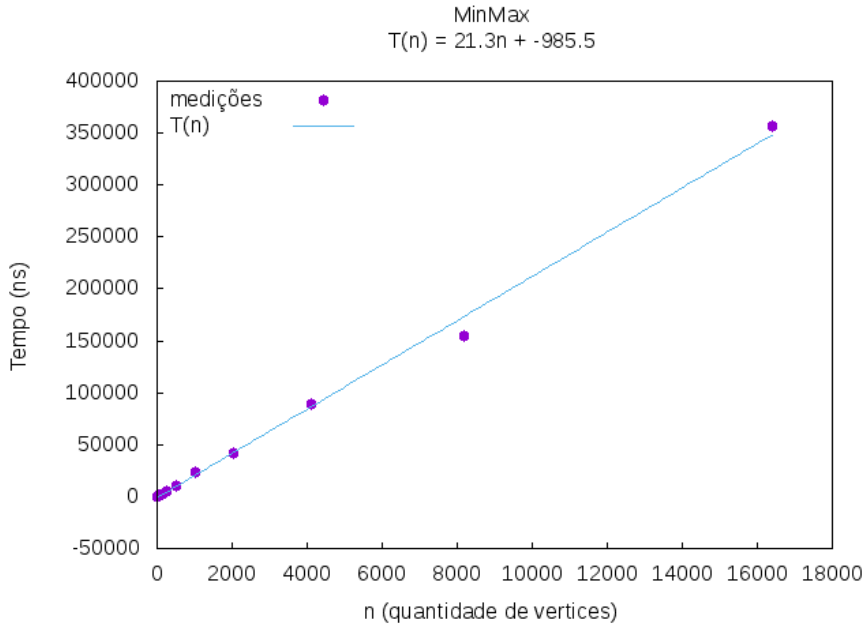

Figura 5.22: *MinMax - Vetor Decrescente P10*

5.2.10 Vetor Decrescente P20

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P20.

Tabela 5.23: *MinMax com vetor Decrescente P20*

Número de Elementos	Tempo de execução em nanosegundos
16	416
32	477
64	461
128	464
256	576
512	749
1024	864
2048	1238
4096	2019
8192	3970
16384	9291

Max/Decrescente P20/MinMax.png Max/Decrescente P20/MinMax.png

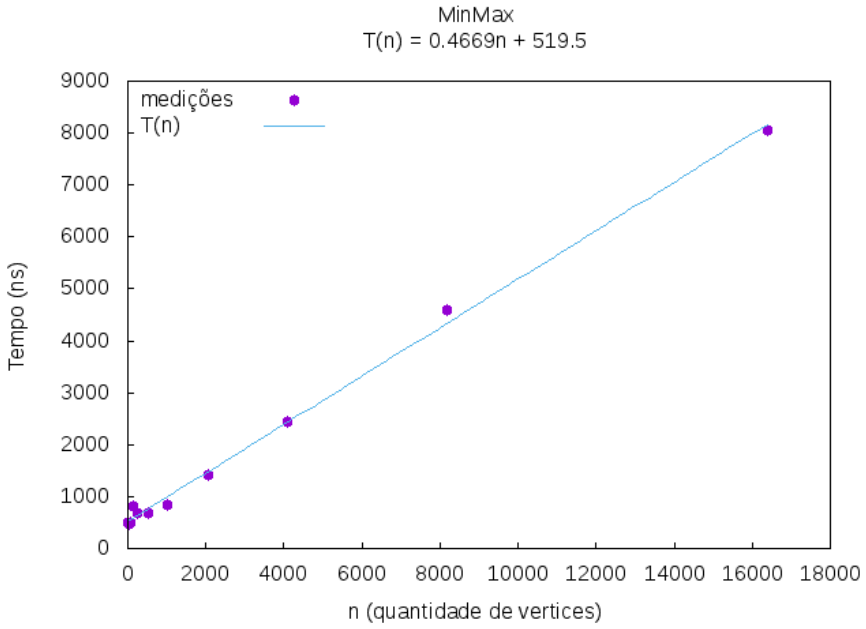

Figura 5.23: *MinMax - Vetor Decrescente P20*

5.2.11 Vetor Decrescente P30

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P30.

Tabela 5.24: *MinMax com vetor Decrescente P30*

Número de Elementos	Tempo de execução em nanosegundos
16	481
32	429
64	480
128	480
256	534
512	723
1024	1062
2048	2273
4096	6007
8192	13578
16384	7634

Max/Decrescente P30/MinMax.png Max/Decrescente P30/MinMax.png

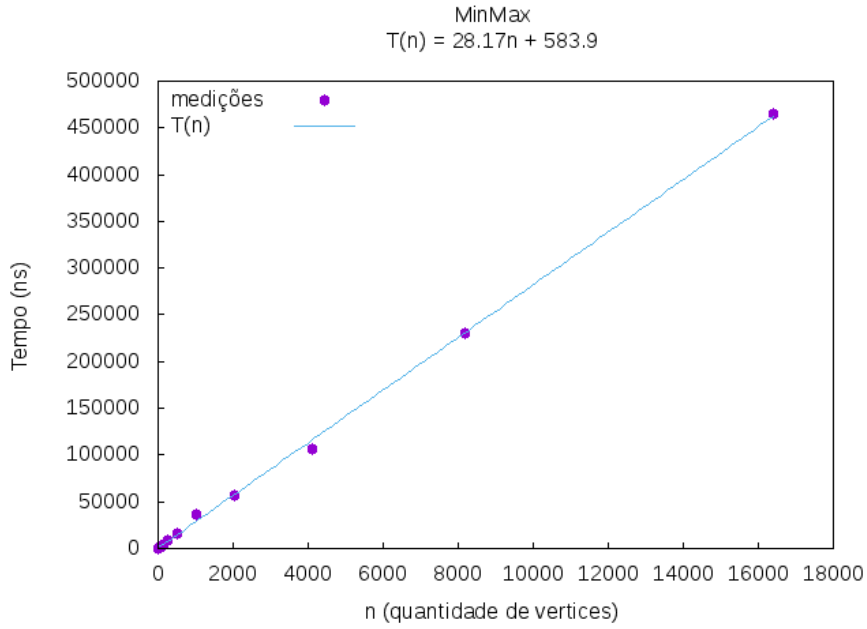

Figura 5.24: *MinMax - Vetor Decrescente P30*

5.2.12 Vetor Decrescente P40

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P40.

Tabela 5.25: *MinMax com vetor Decrescente P40*

Número de Elementos	Tempo de execução em nanosegundos
16	484
32	472
64	440
128	462
256	486
512	664
1024	1036
2048	1507
4096	2219
8192	3887
16384	7530

Max/Decrescente P40/MinMax.png Max/Decrescente P40/MinMax.png

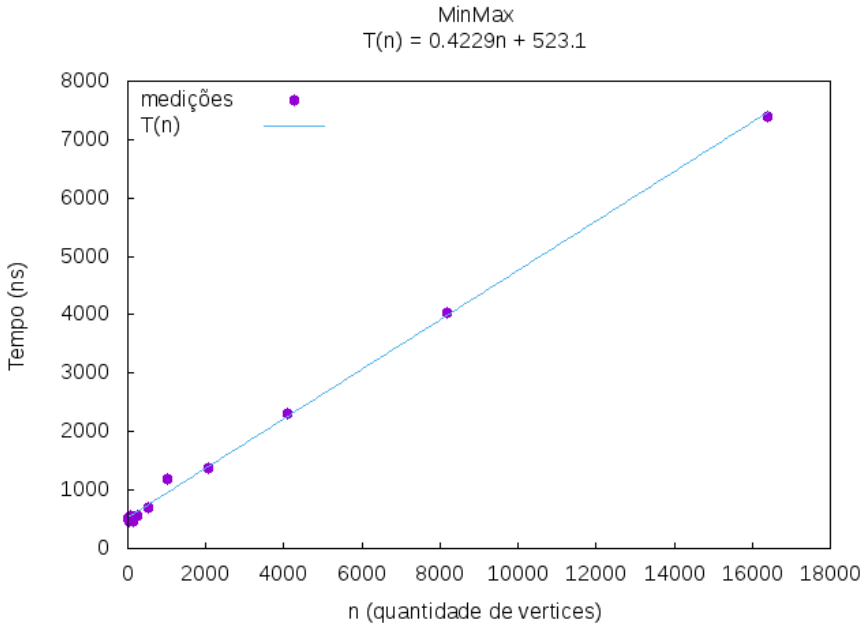

Figura 5.25: *MinMax - Vetor Decrescente P40*

5.2.13 Vetor Decrescente P50

Tabela gerada utilizando MinMax com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos Decrescente P50.

Tabela 5.26: *MinMax com vetor Decrescente P50*

Número de Elementos	Tempo de execução em nanosegundos
16	415
32	494
64	408
128	453
256	507
512	672
1024	806
2048	1413
4096	2813
8192	4791
16384	25421

Max/Decrescente P50/MinMax.png Max/Decrescente P50/MinMax.png

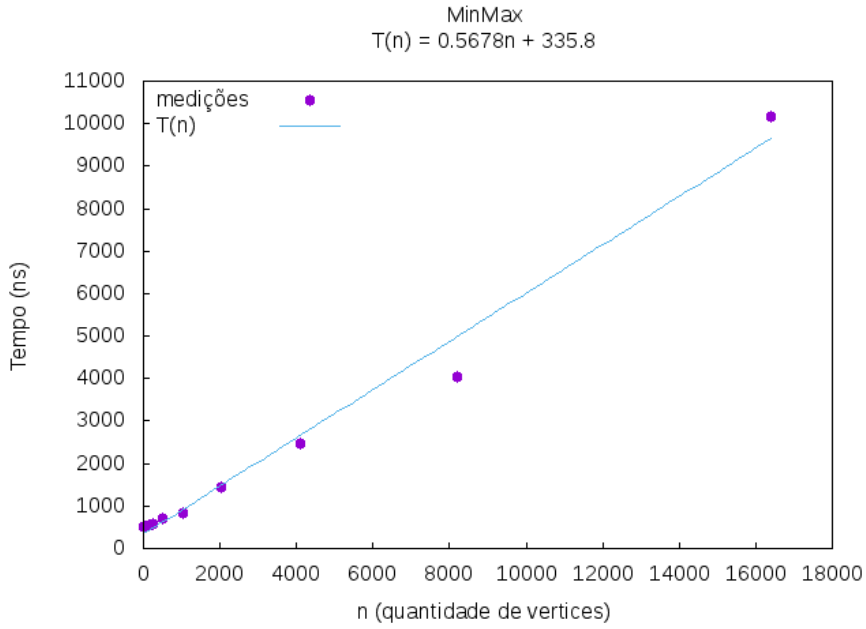

Figura 5.26: *MinMax - Vetor Decrescente P50*

5.3 Seleciona Aleatorizado

Algoritmo inspirado no Quicksort que resolve o problema da seleção.

5.3.1 Vetor aleatorio

Tabela gerada utilizando Seleciona Aleatorizado com vetores de tamanho n, sendo n = (2^k), de k = 4..14 e inseridos aleatóriamente.

Tabela 5.27: *Seleciona Aleatorizado com vetor aleatório*

Número de Elementos	Tempo de execução em nanosegundos
16	631
32	664
64	732
128	764
256	968
512	1068
1024	1332
2048	2044
4096	3555
8192	6672
16384	12553

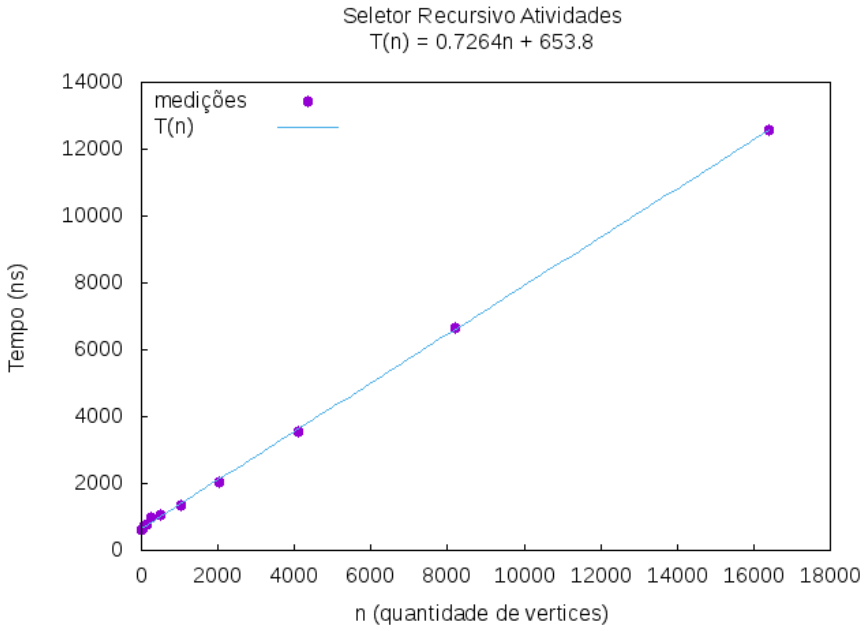

Figura 5.27: *Seleciona Aleatorizado - Vetor Aleatório*

Capítulo 6

Referências

Busca em Largura
Busca em Profundidade
Huffman
Min
Min Max
Mochila Fracionada
Ordenação Tolológica
Slides Professor
Introduction to algorithms 3rd Edition,Cormen, Thomas H,2009