Relatório de Análise de Algoritmos

Daniel Marques, Jefferson Oliveira, Vinicius Gonzaga

August 2, 2017

Contents

1 Programação Dinâmica

1.1 Análise

Com o objetivo de evitar a computação repetida dos mesmos subproblemas, temos a solução resolvendo problemas pela combinação das soluções dos subproblemas. O exemplo estudado em sala visava obter o maior lucro vendendo uma haste, de modo que cada tamanho é vendido por um preço diferente. Foram calculados os tempos para o corte de hastes, de maneira top down, bottom up e bottom up extendido, com os resultados sendo mostrados abaixo.

1.2 Corte Haste

1.2.1 Análise

```
1  int corteHaste(int p[], int n){
2      if(n<=0) return 0;
3      int i;
4      int q = INT_MIN;
5
6      for(i=0;i<n;i++){
7          q = max(q, p[i]+corteHaste(p, n-i-1));
8      }
9      return q;
10 }
```

Listing 1: Código Corte Haste

1.3 Corte Haste Memoizado

1.3.1 Análise

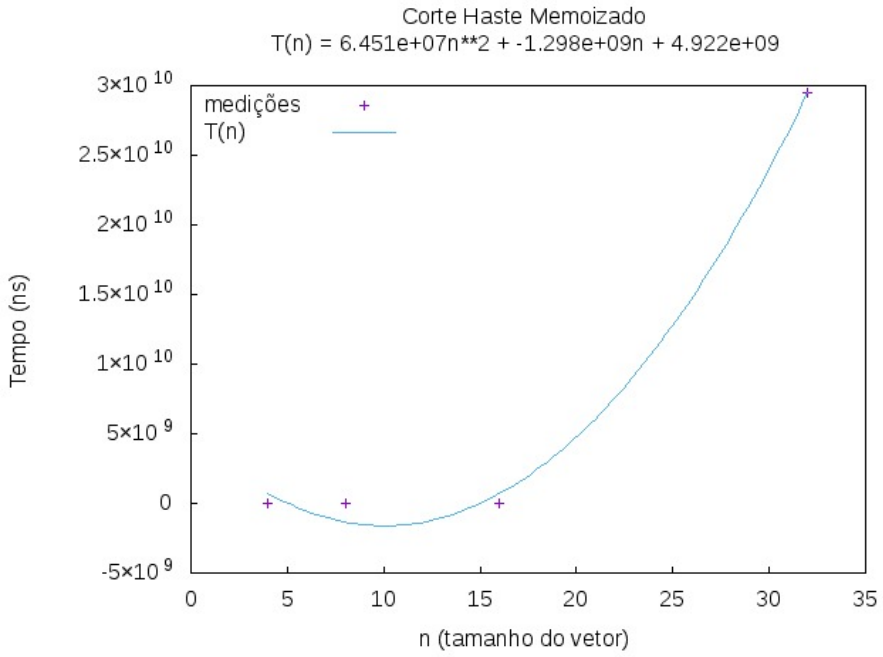
```
int corteHasteMemoizado(int p[], int n){
    int i, r[n];
    for(i=0;i<n;i++){
        r[i] = INT_MIN;
    }
    return corteHasteMemoizadoAux(p,n,r);
}

int corteHasteMemoizadoAux(int *p, int n, int *r){
    if(r[n-1]>=0) return r[n-1];
    int i;
    int q = INT_MIN;
    for(i=0;i<n;i++){
        q = max(q, p[i]+corteHaste(p, n-i-1));
        r[n] = q;
    }
    return q;
}
```

Listing 2: Código Corte Haste Memoizado

1.3.2 Resultados

Todas as comparações com a função T(x) = a*x^2 + b*x + c

Figure 1: corte Haste Memoizado



1.4 Corte Haste Bottom Up

1.4.1 Análise

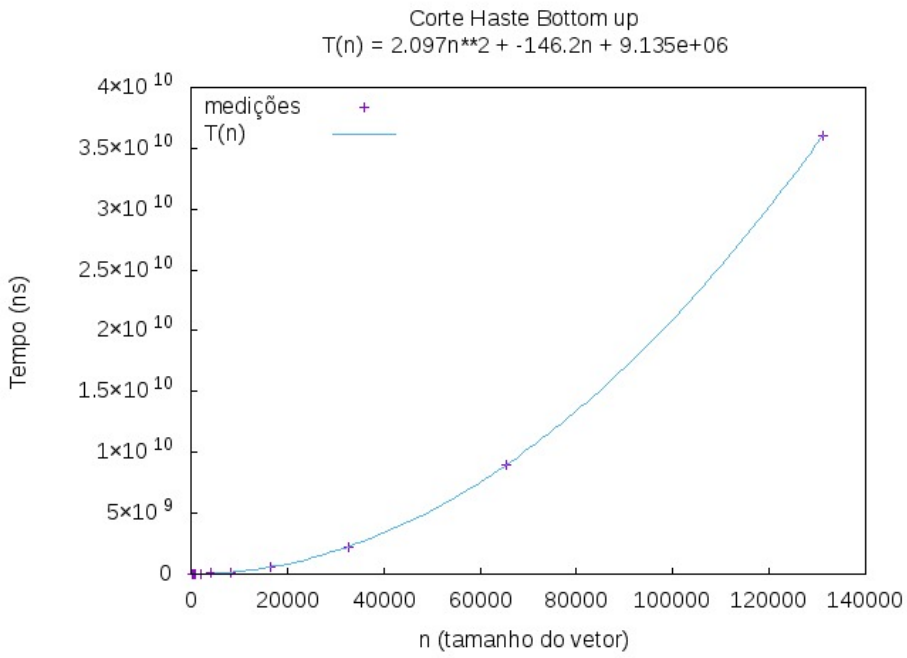
```
int corteBottomUp(int p[], int n){
    int r[n], i, j, q;
    r[0] = 0;
    for(j=1; j<=n; j++){
        q = INT_MIN;
        for(i=1;i<=j;i++)
            q = max(q, p[i-1]+r[j-i-1]);
        r[j-1]=q;
    }
    return r[n-1];
}
```

Listing 3: Código Corte Haste Bottom Up

1.4.2 Resultados

Todas as comparações com a função T(x) = a*x^2 + b*x + c

Figure 2: Corte Haste Bottom Up



1.5 Parentização Bottom Up

1.5.1 Análise

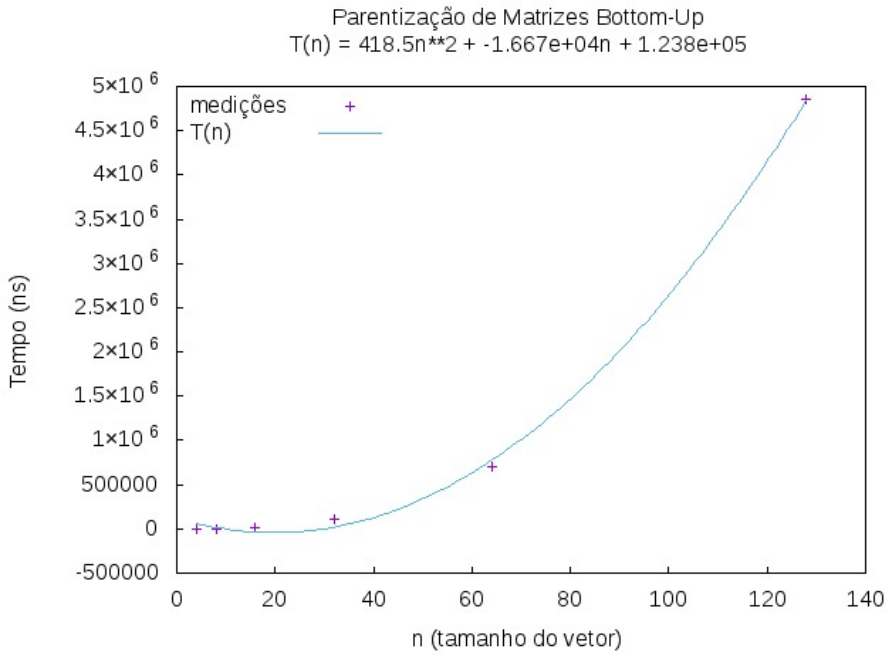
```
int MatrixChainOrder(int p[], int n){
    int **m, i, j, k, L, q;;
    m = (int *) malloc(n*sizeof(int));
    for(i=0;i<n;i++){
        m[i] = (int *) malloc(n*sizeof(int));
    }

    for (i=1; i<n; i++)
        m[i][i] = 0;

    for (L=2; L<n; L++){
        for (i=1; i<n-L+1; i++){
            j = i+L-1;
            m[i][j] = INT_MAX;
            for (k=i; k<=j-1; k++){
                q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }
    return m[1][n-1];
}
```

Listing 4: Código Parentização Bottom Up

1.5.2 Resultados

Todas as comparações com a função T(x) = a*x^2 + b*x + c

Figure 3: Parentizacao Bottom Up



1.6 Subsequencia Comum Maxima

1.6.1 Análise

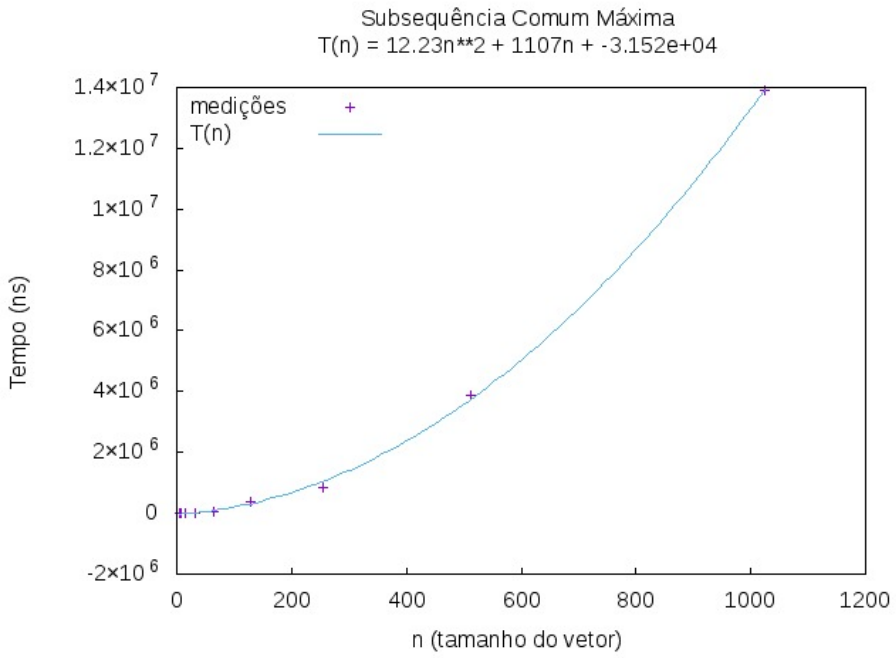
```
int scm( int *X, int *Y, int m, int n )
{
    if (m == 0 || n == 0)
        return 0;

    if (X[m-1] == Y[n-1])
        return 1 + scm(X, Y, m-1, n-1);

    else
        return max(scm(X, Y, m, n-1), scm(X, Y, m-1, n));
}
```

Listing 5: Código Subsequencia Comum Maxima

1.6.2 Resultados

Todas as comparações com a função T(x) = a*x^2 + b*x + c

Figure 4: Subsequencia Comum Maxima



2 Algoritmos Gulosos

2.1 Análise

Assim como os algoritmos de programação dinâmica, é utilizado para resolver problemas de otimização, com a diferença que ela faz a escolha que parecer melhor no momento. Isso implica que esse método não conseguirá sempre alcançar a escolha ótima, sendo papel do usuário determinar qual algoritmo utilizar para resolver o seu problema.

2.2 Seletor de Atividades Iterativo

2.2.1 Análise

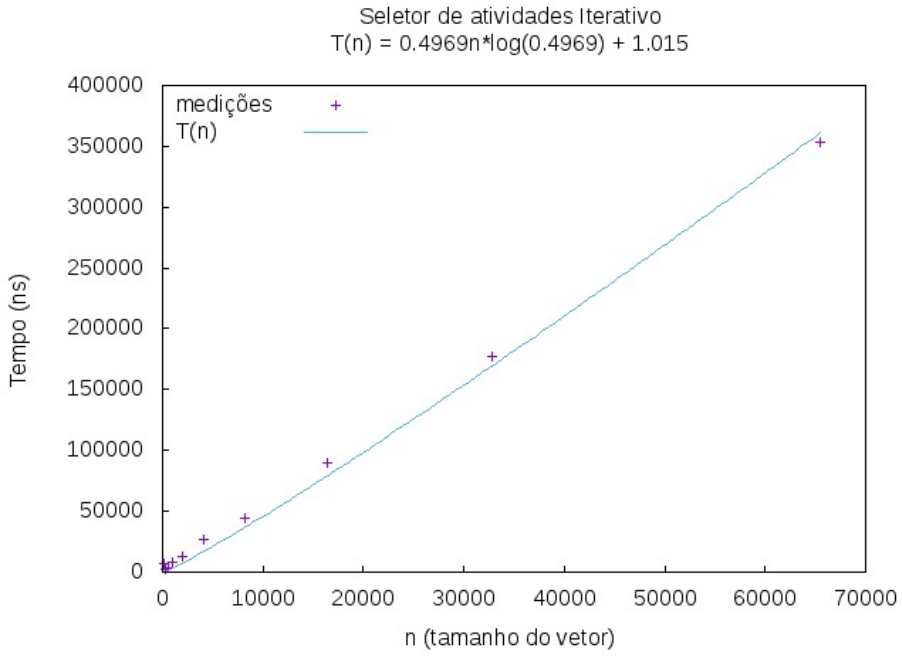
```
1  void seltorIterativo(int s[], int f[], int n){
2    int i, j;
3    //printf ("Selected Activities are:\n");
4    i = 1;
5    //printf("A%d ", i);
6    for (j = 1; j < n; j++){
7      if (s[j] >= f[i]){
8        //printf ("A%d ", j+1);
9        i = j;
10     }
11   }
12 }
```

Listing 6: Código Seletor de Atividades Iterativo

2.2.2 Resultados

Todas as comparações com a função T(x) = a*n*log(a) + b

Figure 5: seletor Atividades Iterativo



2.3 Seletor de Atividades Recursivo

2.3.1 Análise

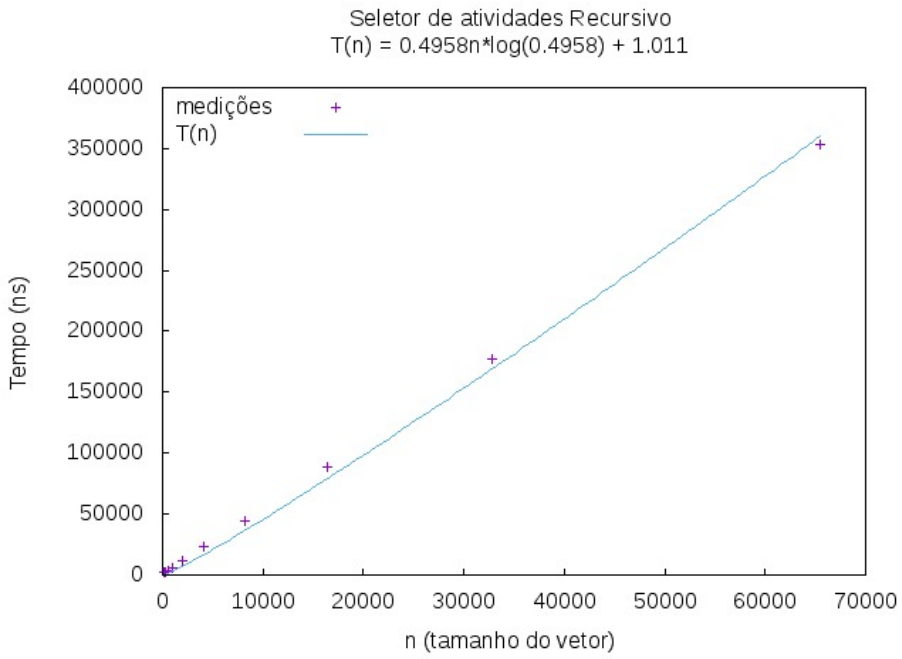
```
void seletorRecursivo(int s[], int f[], int i, int j, int a[]){
      int m = i + 1;
      while (m < j && s[m] < f[i]){
            m = m + 1;
      }
      if (m < j){
            a[m] = 1;
            seletorRecursivo(s,f,m,j,a);
      }
}
```

Listing 7: Código Seletor de Atividades Recursivo

2.3.2 Resultados

Todas as comparações com a função T(x) = a*n*log(a) + b

Figure 6: seletor Atividades Recursivo



2.4 Mochila Booleana

2.4.1 Análise

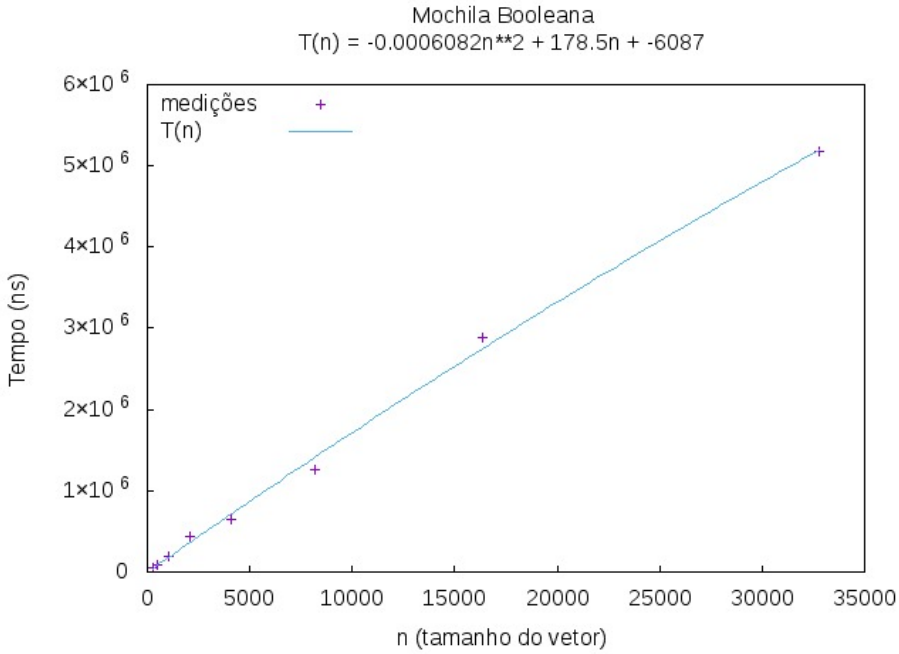
```
int *mochilaBooleana (item *items, int n, int w) {
    int i, j, a, b, *mm, **m, *s;
    mm = calloc((n + 1) * (w + 1), sizeof (int));
    m = malloc((n + 1) * sizeof (int *));
    m[0] = mm;
    for (i = 1; i <= n; i++) {
        m[i] = &mm[i * (w + 1)];
        for (j = 0; j <= w; j++) {
            if (items[i - 1].weight > j)
                m[i][j] = m[i - 1][j];

            else {
                a = m[i - 1][j];
                b = m[i - 1][j - items[i - 1].weight] + items[i - 1].value;
                m[i][j] = a > b ? a : b;
            }
        }
    }
    s = calloc(n, sizeof (int));
    for (i = n, j = w; i > 0; i--) {
        if (m[i][j] > m[i - 1][j]) {
            s[i - 1] = 1;
            j -= items[i - 1].weight;
        }
    }
    free(mm);
    free(m);
    return s;
}
```

Listing 8: Código Corte Haste

2.4.2 Resultados

Todas as comparações com a função T(x) = a*x^2 + b*x + c

Figure 7: Mochila Booleana



2.5 Mochila Fracionária

2.5.1 Análise

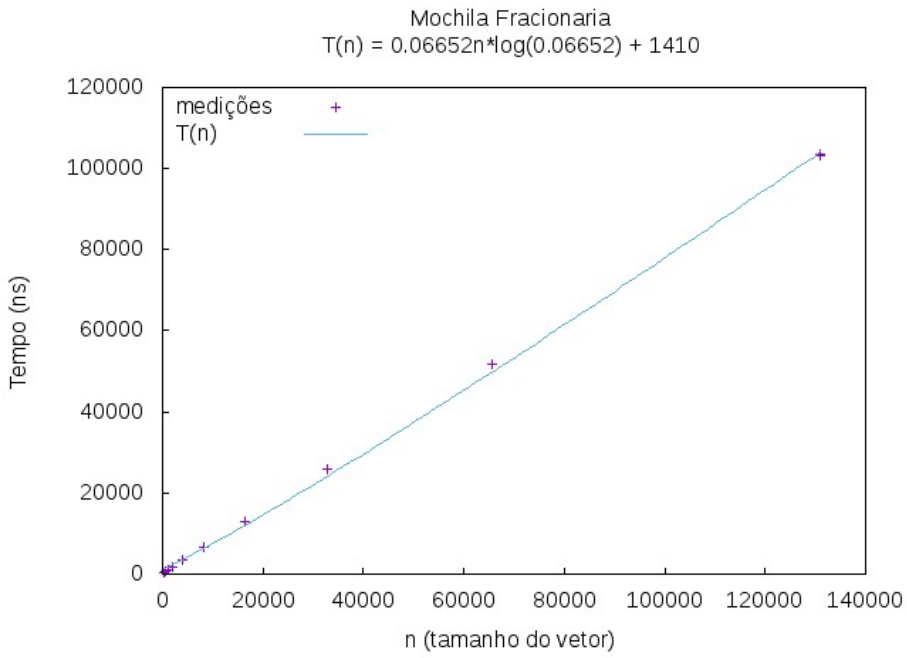
```
int MochilaFracionaria(int W, int wt[], int val[], int n)
{
   if (n == 0 || W == 0)
       return 0;

   if (wt[n-1] > W)
       return MochilaFracionaria(W, wt, val, n-1);

   else return max( val[n-1] + MochilaFracionaria(W-wt[n-1], wt, val, n-1),
                    MochilaFracionaria(W, wt, val, n-1 ));
}
```

Listing 9: Código Mochila Fracionaria

2.5.2 Resultados

Todas as comparações com a função T(x) = a*n*lg(a) + b

Figure 8: Mochila Fracionaria



3 Grafos

3.1 Análise

Utilizado para formulação de diversos problemas computacionais, além de armazenar e organizar dados de maneira eficiente, facilitando sua busca e modificação. Levando-se em consideração as maneiras possíveis de se realizar busca eficiente em um grafo, algumas delas foram selecionadas e suas performances calculadas. O resultado pode ser visto a seguir:

3.2 Busca em largura

3.2.1 Análise

A busca em largura é o tipo de busca que consiste em visitar todos os nós ao redor do vertice em questão, dito vértice raiz, ao terminar a busca ao redor ele busca ao redor dos outros vizinhos. Cada nó tem o valor da distância dele até o vértice inicial. A busca em largura gera uma árvore de busca em largura cuja raiz é o nó inicial a distância dessa raiz a qualquer nó da árvore corresponde a menor distância no grafo original. A complexidade de tempo de busca em largura é $\mathbf{O(|V|+|E|)}$.

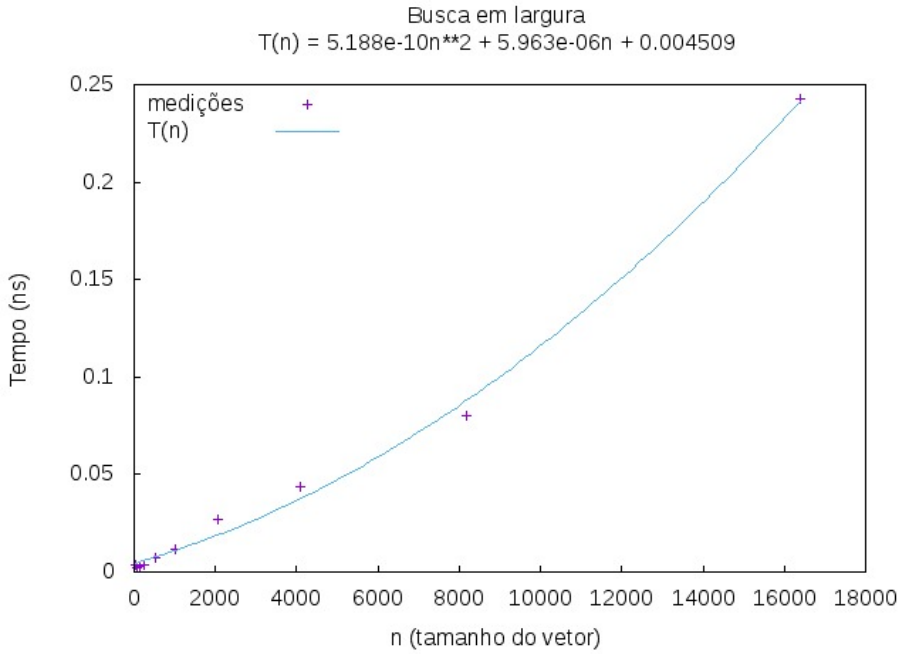
```
void busca_largura(grafo* g, int ini){
    int* visitados = (int *) malloc(g->qtd_vertice*sizeof(int));
    int i, vert, NV,cont=1, *fila, IF = 0, FF = 0;

    for(i=0;i<g->qtd_vertice;i++)
        visitados[i] = 0;
    NV = g->qtd_vertice;
    fila = (int *) malloc(NV*sizeof(int));
    FF++;
    fila[FF] = ini;
    visitados[ini] = cont;
    while(IF!=FF){
        IF = (IF+1)%NV;
        vert = fila[IF];
        for(i=0;i<g->grau[vert];i++){
            if(!visitados[g->aresta[vert][i]]){
                FF = (FF+1)%NV;
                fila[FF] = g->aresta[vert][i];
                visitados[g->aresta[vert][i]] = cont;
            }
        }
        cont++;
    }

    free(fila);
}
```

Listing 10: Busca em largura

3.2.2 Resultados

Todas as comparações com a função T(x) = a*x^2 + b*x + c

Figure 9: busca Em Largura



3.3 Busca em profundidade

3.3.1 Análise

```
1  void busca_profundidade(grafo* g, int V, int *visitado)
2  {
3      int i;
4      no* aux = g->aresta[V];
5      visitado[V]=1;
6      mostra_adjacentes(g,V);
7      while(aux!=NULL){
8          if(visitado[aux->vertice]==0)
9              busca_profundidade(g,aux->vertice, visitado);
10         aux=aux->prox;
11     }
12 }
13
14 void DFS(grafo* g, int V){
15     visitado = (int *) malloc(g->qtde_vertices*sizeof(int));
16     memset(visitado,(int)0, sizeof(int)*g->qtde_vertices);
17     busca_profundidade(g,V,visitado);
18 }
```

Listing 11: Código busca Em Profundidade

3.3.2 Resultados

Todas as comparações com a função T(x) = a*n + b

Figure 10: busca Em Profundidade