Alexandre Marangoni Costa
André Luiz de Brandão Damasceno
Antonio José Grandson Busson
Beatriz Marques Santiago

Problema de Caminho-mais-Curto

Rio de Janeiro - RJ
2017

Alexandre Marangoni Costa
André Luiz de Brandão Damasceno
Antonio José Grandson Busson
Beatriz Marques Santiago

Problema de Caminho-mais-Curto

Relatório técnico apresentado como requisito parcial para obtenção de aprovação na disciplina Projeto e Análise de Algoritmos.

Pontíficia Universidade Católica do Rio de Janeiro
Programa de Pós-Graduação em Informática

Rio de Janeiro - RJ
2017

Sumário

Introdução

O problema do caminho mais curto consiste em determinar o menor custo de travessia entre dois ou mais pontos. Segundo Kleinberg e Tardos (2005) o problema pode ser modelado utilizando teoria de grafos da seguinte forma: dado um grafo $G = (V,E)$, tal que V e E consistem respectivamente de uma coleção de vértices e arestas. Assumindo que para todo vértice s existe um caminho para qualquer outro vértice em G, e que para cada aresta e existe um valor $c_e \geq 0$ indicando o tempo (ou distância, ou custo) associado à sua travessia. Um caminho P é definido pela soma dos valores de todas as arestas em P. O objetivo então é determinar o menor caminho de s para qualquer outro vértice no grafo.

O dijkstra é um dos algoritmos mais eficientes para encontrar o menor caminho em um grafo sem arestas com valores negativos. Sendo desenvolvido por Edsger W. Dijkstra em 1956 (MISA; FRANA, 2010) e publicado em 1959 (DIJKSTRA, 1959), o primeiro programa que utilizou o algoritmo tinha o objetivo de encontrar o caminho mais curto entre duas cidades na Holanda (MISA; FRANA, 2010). Na época foi utilizado um mapa reduzida da Holanda com apenas 64 cidades, onde cada cidade foi codificada utilizando apenas seis bits.

Conforme pode ser visto no pseudocódigo abaixo, a estrutura do algoritmo de Dijskstra se baseia em um conjunto S de vértices cujos os valores de caminhos mais curto $d(u)$ desde a origem s já foram determinados. Inicialmente $S = \{s\}$ e $d(s) = 0$. Para cada vértice $v \in V\text{-}S$ é determinado o caminho mais curto que pode ser feito passando pelas arestas exploradas de S. Assim, os passos são repetidos até que todos os vértices que possuem caminho a partir de s estejam em S.

```
Dijkstra (G,c)
Seja S um conjunto de vértices a serem explorados
    Para cada vértice u ∈ S, armazene uma distância d(u)
Inicie S={S} e d(s)=0
Enquanto S ≠ V
    Selecione um vértice v ∉ S com ao menos uma aresta de S para
        d'(v) = min_{e=(u,v):u∈S} d(u) + c_e ser a menor possível
    Adicione v em S e defina d(v) = d'(v)
```

O dijkstra tem a característica de ser um algoritmo guloso, ou seja, toma a decisão do caminho mais curto que parece ser a melhor no momento de cada iteração. A corretude de que para todo vértice $u \in S$, o caminho P_u é o mais curto, pode ser feita por indução do tamanho de S. Supondo que a afirmação seja mantida quando $|S| = k$ para qualquer

valor de $k \geq 1$. Aumentando S para o tamanho de $k+1$ através da adição do vértice v, a aresta (u,v) passa a ser a aresta final do caminho P_v do vértice s ao v. Baseado na hipótese de indução, P_u é o caminho mais curto para cada vértice $u \in S$. Agora considerando um outro caminho P de s até v, deseja-se mostrar que ele possui pelo menos o mesmo tamanho de P_v. Para alcançar v, esse caminho P deve sair pelo conjunto S através de y, o primeiro vértice em P que não está em S, e $x \in S$ o vértice anterior ao y. Essa situação pode ser vista na Figura 1.

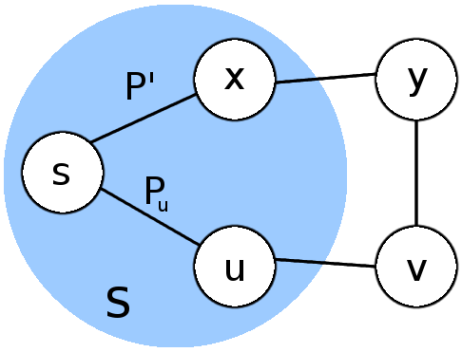

Figura 1: Caminho mais curto P_v e um caminho P alternativo do vértice s ao v através do vértice y.

Entretanto, P não pode ser menor do que P_v porque ele já é pelo menos do tamanho de P_v. Na iteração $k+1$, o algoritmo de Dijkstra deve ter considerado adicionar o vértice y para o conjunto S via aresta (x,y), e rejeitou essa opção em favor da adição de v. Isso significa que não existe caminho de s a y por meio de x que seja mais curto do que P_v. A prova desse argumento pode ser feita utilizando inequações. Sabendo que $x \in S$, P' é um subgrafo de P e o caminho de s à x está contido em P', sabe-se por hipótese de indução que P_x é o caminho mais curto de s à x (de tamanho $d(x)$), logo $c(P') \geq c(P_x) = d(x)$. Assim o subgrafo de P até y tem tamanho $c(P') + c(x,y) \geq d(x) + c(x,y) \geq d'(y)$, e o caminho completo de P é pelo menos do mesmo tamanho desse subgrafo. Desde que o dijkstra selecione v nessa iteração, sabe-se que $d'(y) \geq d'(v) = c(P_v)$. Combinando essas inequações pode-se mostrar que $c(P) \geq c(P') + c(x,y) \geq c(P_v)$.

A complexidade em tempo de execução do Dijkstra é altamente relacionada com a estrutura de dados usada nesta etapa:

```
Selecione um vértice v ∉ S com ao menos uma aresta de S para
    d'(v) = min_{e=(u,v):u∈S} d(u) + c_e ser a menor possível
```

A fim de analisar diferentes cenários, este relatório apresenta detalhes sobre as complexidades usando 5 tipos de estruturas de dados diferentes: vetor, árvore balanceada de busca, heap de Fibonacci, buckets e árvore α. Cada estrutura foi dividida em um capítulo, e em cada capítulo será apresentado um código diferente do original, modificando principalmente o trecho acima para ilustrar a estrutura de dados analisada.

1 Estruturas de dados

A Listagem 1.1 ilustra a estrutura utilizada para armazenar os dados dos vértice. O atributo *index* informa o índice do vértice; O atributo *distance* informa a distancia mínima até o vértice de partida, durante a inicialização do algoritmo, este atributo recebe o valor da macro INT_MAX da biblioteca limits.h, o qual simboliza o valor ∞; O atributo *adjs* faz referencia aos vértices adjacentes; E por último, o atributo *weights* faz referencia aos pesos das arestas.

Listing 1.1: Estrutura de um vértice

```
struct vertice{
  guint32 index;
  guint32 distance;
  struct vertice** adjs;
  guint32* weights;
};
```

1.1 Vetor

Conforme visto na introdução, o algoritmo do Dijkstra é um algoritmo guloso, que usa uma estrutura de dados para armazenar as menores distâncias conhecidas a cada iteração. Portanto, inserir, remover e atualizar os elementos dessa estrutura são as operações que mais influenciam a complexidade final. Ilustraremos isso com o seguinte pseudo-código (DASGUPTA; PAPADIMITRIOU; VAZIRANI, 2006), onde G representa o grafo, l guarda o custo de cada aresta, e s é o nó inicial de onde dijkstra partirá.

```
Dijkstra (G, l, s)
para todo u em G.vertices
    dist[u] = ∞

dist[s] = 0
vetor = cria_vetor(dist)

enquanto vetor != ∅
    u = remove_min(vetor)
    para todo (u, v) em G.arestas
        se dist[v] > dist[u] + l(u, v)
            dist[v] = dist[u] + l (u, v)
            atualiza_chave(vetor, v, dist[v])
```

Inicialmente, o algoritmo percorre os vértices para inicializar o vetor de distâncias (resultado final) e cria a estrutura de dados que armazena as distâncias conhecidas a cada passo guloso. Neste capítulo, essa estrutura também é um vetor, então para acessar um elemento através de um índice, gasta-se $O(1)$. Para inicializar n vértices gasta-se n operações. Portanto a complexidade destes primeiros passos é n x $O(1) = O(n)$.

Em seguida, o algoritmo deve remover o menor elemento do vetor, o vértice u. O vetor não está ordenado, então a busca de u ocorre em $O(n)$ operações. Além disso, esse passo remove cada vértice do vetor um a um. Como temos n vértices, temos $O(n)$ operações que custam $O(n)$, resultando em $O(n^2)$ operações.

Por fim, caso encontremos um vértice v com aresta para u (do passo anterior) através de um caminho melhor do que conhecíamos previamente, atualizamos a distância que conhecemos até o vértice v. Atualizar o elemento de um vetor custa O(1). Como isso é feito uma vez para cada aresta do grafo, em um grafo de m arestas, temos $O(m)$ operações. Além disso, assumindo que um grafo possa conter no máximo $O(n^2)$ arestas, a complexidade do pior caso é $O(n^2)$.

Conclui-se que a complexidade total do Dijkstra com vetor é $O(n^2)$.

Abaixo temos esse trecho implementado em C. No código abaixo, *vertices* é o *G.vertices* do pseudo código, e *v->distance* (veja 1.1 acima) substitui o vetor *dist* do pseudo código. Ambos já estão inicializados, e o resto do código pode ser facilmente mapeado ao pseudo-código linha a linha:

Listing 1.2: Corpo do algortimo Dijkstra com uso do vetor

```
for(guint32 count = 0; count < numVertices; count++){
    vertice* v = get_min_distance(vertices); /* 1.1 */

    for(guint32 i=0;i< v->adjCount;i++){  /* 1.3 */
        vertice * w = vertices[v->adjs[i]->index];
        if(v->distance + v->weights[i] < w->distance)
           w->distance = v->distance + v->weights[i];
    }
}
```

1.2 Árvore Balanceada de Busca

Em uma árvore balanceada de busca, a altura da árvore é constantemente balanceada, possuindo sempre um valor proporcional a $O(log(n))$. Para buscar elementos, bem como a encontrar a posição correta de um elemento nessa estrutura, percorre-se as alturas, uma a uma. Portanto, temos que a complexidade de se inserir/remover/atualizar

um elemento é $O(log(n))$.

Assim sendo, usando o mesmo pseudo código acima, mas trocando vetor por essa nova estrutura, podemos ver que inicialmente o algoritmo inicializa os n vértices em $O(n.log(n))$ operações.

Em seguida, o algoritmo remove cada vértice, também em $O(n.log(n))$ operações.

Por último, para cada nova aresta conhecida (m arestas), o algoritmo guloso atualiza o vértice destino na árvore balanceada, de acordo com a nova distância revelada por essa nova aresta, em $O(log(n))$. Temos aqui $O(m.log(n))$ operações.

Portanto, a complexidade do Dijkstra usando uma árvore balanceada de busca é O((n+m).log(n)).

A implementação em C apresentada neste trabalho usa uma árvore balanceada binária, a variável *tree*, que corresponde à variável *vetor* do pseudo código. Ao invés de inicializar essa árvore com todos os vértices, eles são inseridos apenas quando o algoritmo guloso chega até eles. Para buscar, remover e inserir elementos na árvore, são usadas as funções g_tree_search, g_tree_remove e g_tree_insert respectivamente, e o restante do algoritmo pode ser facilmente compreendido.

Listing 1.3: Corpo do algoritmo Dijkstra com uso da árvore balanceada de busca

```
while(g_tree_height(tree)>0){
   vertice* v = g_tree_search(tree,&query); /* 1.1 */
   g_tree_remove(tree,v);

   for(guint32 i=0;i<v->adjCount;i++){ /* 1.3 */
       vertice * w = v->adjs[i];
       if(v->distance + v->weights[i] < w->distance){
          w->distance = v->distance + v->weights[i];
          g_tree_insert(tree,w);
       }
   }
}
```

1.3 Heap de Fibonacci

Uma análise análoga a anterior pode ser feita para o Heap de Fibonacci. Remover o menor elemento do heap (função heap_extract_min) custa O(log(n)) para um grafo de n vértices. Já inserir no heap de fibonacci (heap_insert) custa O(1) (amortizado) (DASGUPTA; PAPADIMITRIOU; VAZIRANI, 2006).

A implementação em C do algoritmo insere os vértices no heap a medida em que

eles vão sendo descobertos. Além disso, todos os vértices já são inicializados com distância ∞. Portanto, o trecho do pseudo código que interessa analizar é o seguinte:

Listing 1.4: parte do Pseudo-código do algoritmo de Dijkstra

```
enquanto heap != ∅
    u = remove_min(heap)
    para todo (u, v) em G.arestas
        se dist[v] > dist[u] + l(u, v)
            dist[v] = dist[u] + l (u, v)
            atualiza_chave(heap, v, dist[v])
```

Inicialmente os n vértices são removidos do heap um a um, quando sua distância mínima é encontrada. Como remover do heap de Fibonacci custa $O(log(n))$, este passo custa $O(n.log(n))$ no total.

Além disso, para cada aresta de um grafo com m arestas, são feitas algumas operações em $O(1)$ e, ocasionalmente, atualizações de chave, em $O(1)$ amortizado. O que nos dá uma complexidade amortizada de $O(m)$

Portanto, a complexidade amortizada total é O(n.log(n) + m).

Na implementação em C apresentada, ao invés de atualizarmos uma chave, é feita a inserção do nó corrente, que também é $O(1)$ amortizado. Portanto a complexidade final é a mesma.

Listing 1.5: Corpo do algortimo Dijkstra com uso da heap de fibonacci

```
while (!is_empty(heap)){
   vertice * v = heap_extract_min(heap); /* 1.1 */

   for(guint32 i=0;i< v->adjCount;i++){ /* 1.3 */
      vertice * w = v->adjs[i];
      if(v->distance + v->weights[i] < w->distance){
         w->distance = v->distance + v->weights[i];
         heap_insert(heap,w);
      }
   }
}
```

1.4 Buckets

A implementação do algoritmo de Dijkstra usando Buckets é identica a usando o Heap de Fibonacci, e a sua complexidade pode ser analisada de forma análoga, conforme veremos adiante.

Veja o código abaixo:

Listing 1.6: Corpo do algortimo Dijkstra com uso de buckets

```
while(!bucket_is_empty(buc)){
   vertice* v = bucket_remove_min(buc); /* 1.1 */

   for(guint32 i=0;i< v->adjCount;i++){ /* 1.3 */
     vertice * w = v->adjs[i];
     if(v->distance + v->weights[i] < w->distance){
        w->distance = v->distance + v->weights[i];
        bucket_add(buc,w);
     }
   }
}
```

Inicialmente, o algoritmo remove o vértice com menor distância dos buckets, uma vez para cada vértice. Remover o mínimo elemento dos buckets custa O(C) amortizado, onde C é o valor máximo do custo de uma aresta. Assim sendo, a complexidade amortizada deste passo é O(nC).

Em seguida, para cada aresta o algoritmo faz algumas operações em O(1) e, eventualmente, insere um elemento nos buckets. Inserir nos buckets custa O(1). Portanto este passo custa O(m).

No total, podemos ver que o algoritmo de Dijkstra implementado com buckets tem complexidade O(nC + m).

1.5 Árvore α

BB[α] tree, ou simplesmente Árvore α, é um tipo de árvore binária de busca com auto-balancemanto, introduzida pela primeira vez em (NIEVERGELT; REINGOLD, 1973). Na árvore α o tamanho de um nó interno é a soma do tamanho de seus dois filhos mais um (size[n] = size[n.left] + size[n.right] + 1). Um nó da árvore α é dito balanceado se: size[n.left] $\leq \alpha$.size[n] e size[n.right] $\leq \alpha$.size[n].

1. Caso seja necessário realizar o balanceamento da árvore α, pega-se o v mais alto que não é α-balanceado, então a sub-árvore de v é trocada por uma sub-árvore α-balanceada que contém os mesmos elementos. Para realizar o balanceamento checa-se cada elemento da sub-árvore, obtendo assim O(size[x]) operações.

2. Para a operação de busca, no pior caso (quando o nó é uma folha) é preciso percorrer apenas um nó por nível, até chegar na folha. Quando α é próximo de 1/2 a árvore α

tem um comportamento semelhante a uma árvore binária de busca, que no pior caso tem uma complexidade O(log(n)). Já quando α é próximo a 1 a árvore α tem um comportamento semelhante a uma lista, que no pior caso tem complexidade O(n).

3. A partir daqui considere alpha estritamente maior que 1/2.

4. Dependendo como os nós de uma árvore binária de busca estão distribuídos, o potencial pode ser maior que zero, por exemplo quando uma árvore degenera para uma lista crescente, tem-se que para todo nó n, size[n.left] = 0 e size[n.right] = 1. Já para uma árvore 1/2-balanceada, tem-se que o tamanhos dos filhos esquerdo e direito são os mesmos (size[n.left] = size[n.right]), pois ambos devem ter um tamanho maior que a metade de size[n]. Caso contrário, a condição de balanceamento α não esta satisfeita. Logo, o potencial é igual a ZERO.

5. Quando a árvore degenera para uma lista (com α próximo a 1), o balanceamento pode ser feito com apenas 1 troca de seus filhos, resultando em O(1) operações para balancear uma sub-árvore.

6. Para a operação de inserção na árvore α é necessário percorrer um nó por nível até que a folha seja alcançada (vide item ii), o que resulta em O(log(n)) operações. Caso seja necessário fazer o balanceamento da árvore (vide item i) é gasto O(size[v]) operações, onde v é o nó raiz da sub-arvore balanceada. O que resulta em uma complexidade que varia de O(log(n) + size[v])) até O(n), dependendo do valor de α Como a árvore α dificilmente degenera para uma lista, obtém-se uma complexidade amortizada de O(log(n)). Para a operação de deleção é feito um processo semelhante, com a diferença que quando o nó a ser deletado é encontrado, deve-se exclui-lo, colocando em seu lugar um de seus filhos de forma que o balanceamento seja mantido. Como na operação de inserção, a deleção também tem complexidade amortizada de O(log(n)).

Portanto, a complexidade do Dijkstra usando uma árvore α pode variar de O((n+m).log(n)) até O(n^2), dependendo do valor de α.

A implementação em C apresentada neste trabalho usa uma árvore α, que ao invés de inicializar essa árvore com todos os vértices, eles são inseridos apenas quando o algoritmo guloso chega até eles. Para remover o menor e inserir elementos na árvore, são usadas as funções alpha_tree_smallest e alpha_tree_insert respectivamente, e o restante do algoritmo pode ser facilmente compreendido.

Listing 1.7: Corpo do algortimo Dijkstra com uso da árvore α

```
while(alpha_tree_height(tree)>0){
   vertice* v = alpha_tree_smallest(tree); /* 1.1 */

   for(guint32 i=0;i< v->adjCount;i++){ /* 1.3 */
     vertice * w = v->adjs[i];
     if(v->distance + v->weights[i] < w->distance){
        w->distance = v->distance + v->weights[i];
        alpha_tree_insert(tree,w);
     }
   }
}
```

2 Resultados

Para realizar o experimento de execução do dijkstra foram utilizados máquinas de mesma configuração com processador Intel Core i7-6700 3,4 GHz x 8, memória de 15,6 GB e sistema operacional ubuntu 16.04 LTS na versão 64 bits. Os algoritmos foram implementados utilizando a linguagem C e compilados através do gcc 5.4.0.

Os dados dos grafos utilizados estavam em arquivos no formato STP divididos em pastas chamadas ALUE, ALUT e DMXA. Cada arquivo apresentava a quantidade de vértices e arestas. Cada aresta era especificada pelo valor do vértice de origem, destino e custo de travessia. O vértice 1 de cada um dos arquivos foi definido como o vértice de origem do dijkstra. Para a análise foi assumido que n = V (número de vértices) e m = E (número de arestas).

Os resultados das execuções do dijkstra em cada uma das estrutura de dados (Vetor, Árvore Balanceada de Busca, Heap de Fibonacci, Buckets e Árvore α), podem ser observados nas tabelas e gráficos abaixo. Em geral todas as tabelas estão com seus resultados ordenados pelo número de vértices dos grafos. Em todas as tabelas, tem suas colunas organizadas pelo nome do arquivo, numero de vértices e arestas, quantidade de vértices processados, tempo de execução e complexidade teórica. A tabela de Árvore α, além de apresentar as mesmas colunas das outras estruturas de dados, com exceção da coluna de complexidade teórica, mostra os valores do tempo de execução com diferentes valores para α.

Vetor					**Complexidade Teórica**
Arquivo	Vértices	Arestas	V. Processados	Tempo	$\|V\|^2$
alue7229	940	1474	940	0,002559	883600
alue2105	1220	1858	1858	0,005142	1488400
alue2087	1244	1971	1971	0,005306	1547536
alue6951	2818	4419	4419	0,045333	7941124
alue6179	3372	5213	5213	0,068000	11370384
alue5067	3524	5560	5560	0,076727	12418576
alue3146	3626	5869	5869	0,078092	13147876
alue6457	3932	6137	6137	0,112444	15460624
alue6735	4119	6696	6696	0,114364	16966161
alue5623	4472	6938	6938	0,138595	19998784
alue5345	5179	8165	8165	0,206080	26822041
alue7066	6405	10454	10454	0,348000	41024025
alue5901	11543	18429	18429	1,233600	133240849
alue7065	34046	54841	54841	2,874000	1159130116
alue7080	34479	55494	55494	2,996000	1188801441

Tabela 1: Tabela de resultados do ALUE utilizando Vetor.

Árvore Balanceada de Busca					**Complexidade Teórica**
Arquivo	Vértices	Arestas	V. Processados	Tempo	$(\|V\|+\|E\|)\log\|V\|$
alue7229	940	1474	809	0,001171	23841,91
alue2105	1220	1858	896	0,001403	31557,70
alue2087	1244	1971	933	0,001410	33052,68
alue6951	2818	4419	2035	0,003392	82939,32
alue6179	3372	5213	2748	0,004204	100610,95
alue5067	3524	5560	2975	0,005062	107036,76
alue3146	3626	5869	2852	0,004501	112270,43
alue6457	3932	6137	3578	0,005741	120234,41
alue6735	4119	6696	2970	0,004689	129867,37
alue5623	4472	6938	2750	0,004470	138365,70
alue5345	5179	8165	4215	0,007176	164644,38
alue7066	6405	10454	4767	0,008124	213181,77
alue5901	11543	18429	8579	0,014905	404464,07
alue7065	34046	54841	31374	0,075032	1338211,36
alue7080	34479	55494	34479	0,082340	1356201,75

Tabela 2: Tabela de resultados do ALUE utilizando Árvore Balanceada de Busca.

Heap de Fibonacci					**Complexidade Teórica**
Arquivo	Vértices	Arestas	V. Processados	Tempo	|V|log|V| + E
alue7229	940	1474	809	0,000558	10757,93
alue2105	1220	1858	896	0,000610	14366,25
alue2087	1244	1971	933	0,000641	14760,28
alue6951	2818	4419	2035	0,001475	36714,56
alue6179	3372	5213	2748	0,002002	44730,78
alue5067	3524	5560	2975	0,002233	47083,29
alue3146	3626	5869	2852	0,002129	48743,42
alue6457	3932	6137	3578	0,002734	53089,20
alue6735	4119	6696	2970	0,002201	56157,27
alue5623	4472	6938	2750	0,002046	61168,62
alue5345	5179	8165	4215	0,003262	72065,87
alue7066	6405	10454	4767	0,003693	91445,12
alue5901	11543	18429	8579	0,006775	174198,68
alue7065	34046	54841	31374	0,032623	567410,26
alue7080	34479	55494	34479	0,035800	575210,80

Tabela 3: Tabela de resultados do ALUE utilizando Heap de Fibonacci.

Buckets						**Complexidade Teórica**
Arquivo	Vértices	Arestas	V. Processados	C	Tempo	|V|C + |E|
alue7229	940	1474	809	13	0,000078	13694
alue2105	1220	1858	896	13	0,000094	17718
alue2087	1244	1971	933	13	0,000095	18143
alue6951	2818	4419	2035	13	0,000213	41053
alue6179	3372	5213	2748	13	0,000305	49049
alue5067	3524	5560	2975	13	0,000334	51372
alue3146	3626	5869	2852	13	0,000320	53007
alue6457	3932	6137	3578	13	0,000417	57253
alue6735	4119	6696	2970	13	0,000319	60243
alue5623	4472	6938	2750	13	0,000315	65074
alue5345	5179	8165	4215	13	0,000490	75492
alue7066	6405	10454	4767	13	0,000554	93719
alue5901	11543	18429	8579	13	0,001064	168488
alue7065	34046	54841	31374	13	0,010160	497439
alue7080	34479	55494	34479	13	0,011111	503721

Tabela 4: Tabela de resultados do ALUE utilizando Buckets.

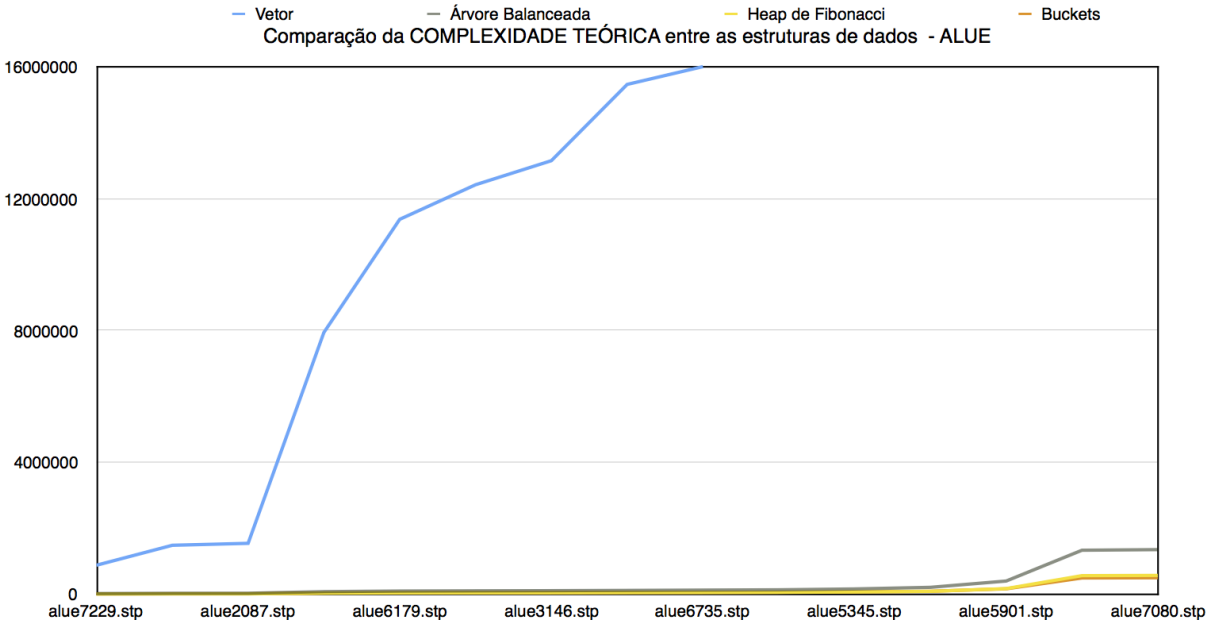

Figura 2: Gráfico comparativo das COMPLEXIDADES TEÓRICAS das estruturas de dados com as instâncias ALUE.

Árvore α				$\alpha = 0.6$	$\alpha = 0.7$	$\alpha = 0.8$	$\alpha = 0.9$
Arquivo	Vértices	Arestas	V. Processados	Tempo	Tempo	Tempo	Tempo
alue7229	940	1474	809	0,001394	0,001439	0,001874	0,002653
alue2105	1220	1858	896	0,001562	0,001633	0,002159	0,002995
alue2087	1244	1971	933	0,001591	0,001657	0,002241	0,003251
alue6951	2818	4419	2035	0,005416	0,005554	0,006596	0,008015
alue6179	3372	5213	2748	0,006164	0,006590	0,008869	0,009958
alue5067	3524	5560	2975	0,007102	0,007628	0,011092	0,011959
alue3146	3626	5869	2852	0,006712	0,007080	0,010059	0,010731
alue6457	3932	6137	3578	0,007822	0,008427	0,012173	0,014627
alue6735	4119	6696	2970	0,007019	0,007576	0,010976	0,011811
alue5623	4472	6938	2750	0,006452	0,006828	0,009103	0,010173
alue5345	5179	8165	4215	0,008266	0,008907	0,012673	0,016955
alue7066	6405	10454	4767	0,010225	0,010404	0,021761	0,023841
alue5901	11543	18429	8579	0,016952	0,018853	0,115619	0,141570
alue7065	34046	54841	31374	0,077132	0,083448	0,306071	0,357562
alue7080	34479	55494	34479	0,085292	0,087687	0,486298	0,586772

Tabela 5: Tabela de resultados do ALUE utilizando Árvore α.

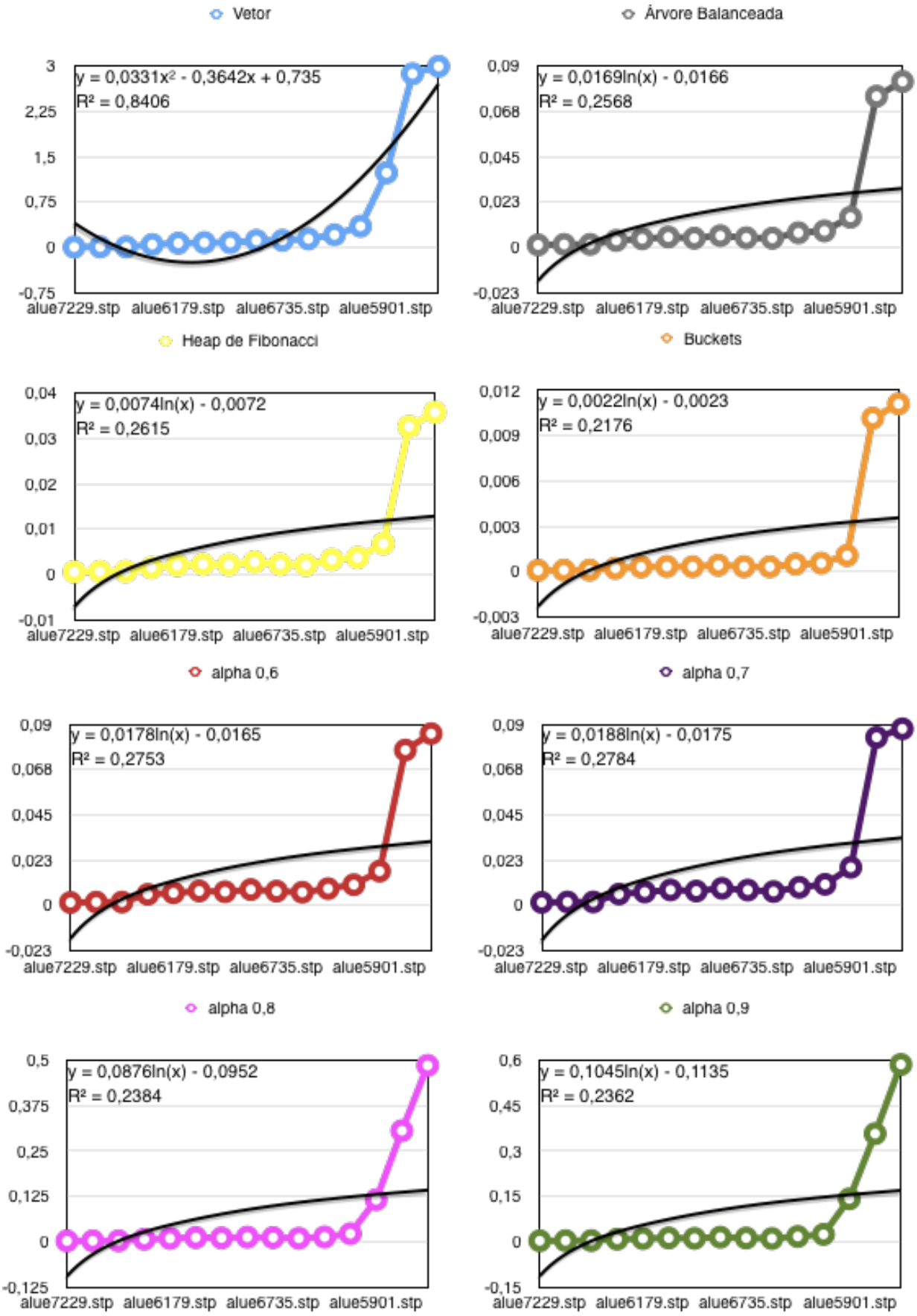

Figura 3: TEMPO DE CPU das estruturas de dados executadas com as instâncias ALUE.

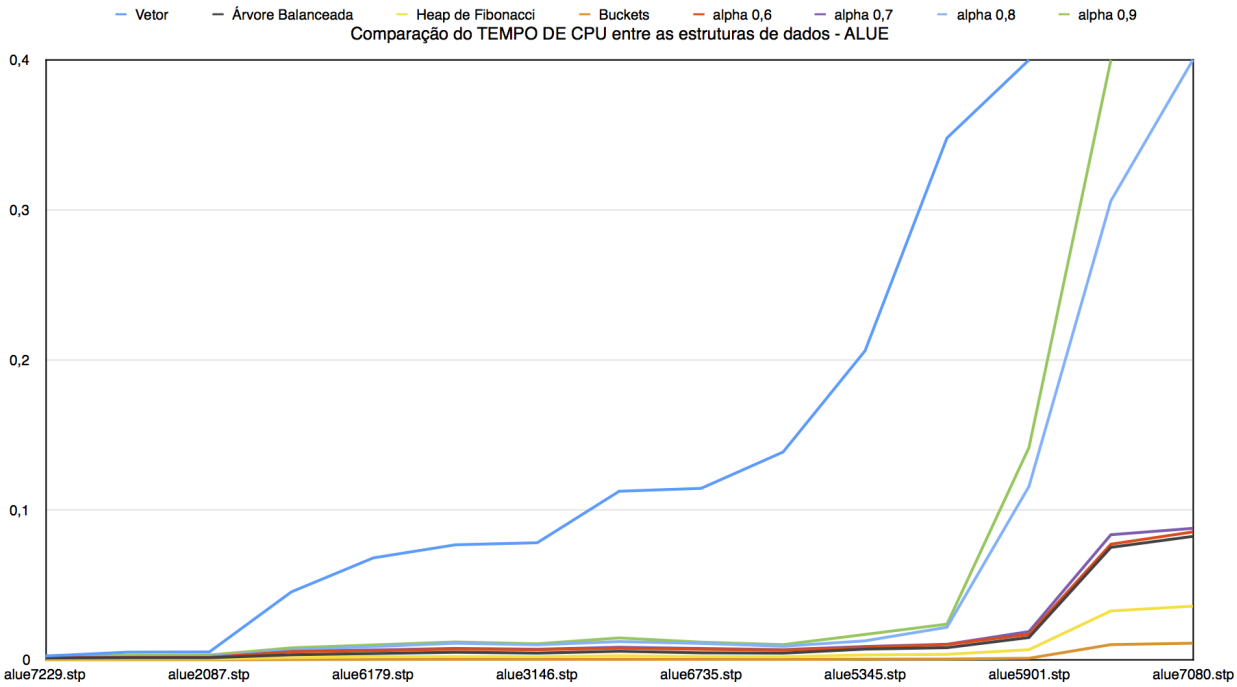

Figura 4: Gráfico comparativo de TEMPO DE CPU das estruturas de dados executadas com as instâncias ALUE.

Vetor					**Complexidade Teórica**
Arquivo	Vértices	Arestas	V. Processados	Tempo	$\|V\|^2$
alut2764	387	626	387	0,000367	149769
alut0805	966	1666	966	0,002558	933156
alut0787	1160	2089	1160	0,003765	1345600
alut1181	3041	5693	3041	0,050949	9247681
alut2566	5021	9055	5021	0,178345	25210441
alut2010	6104	11011	6104	0.306118	37258816
alut2288	9070	16595	9070	0.696500	82264900
alut2610	33901	62816	33901	2,794000	1149277801
alut2625	36711	68117	36711	4,026000	1347697521

Tabela 6: Tabela de resultados do ALUT utilizando Vetor.

Árvore Balanceada de Busca					**Complexidade Teórica**
Arquivo	Vértices	Arestas	V. Processados	Tempo	(|V|+|E|)log|V|
alut2764	387	626	387	0,000552	8707,94
alut0805	966	1666	956	0,001447	26098,59
alut0787	1160	2089	1135	0,001702	33074,52
alut1181	3041	5693	2465	0,003865	101055,26
alut2566	5021	9055	3836	0,005875	173046,95
alut2010	6104	11011	4729	0,007546	215230,35
alut2288	9070	16595	8456	0,014194	337414,85
alut2610	33901	62816	32746	0,070478	1455498,02
alut2625	36711	68117	36702	0,079989	1589603,91

Tabela 7: Tabela de resultados do ALUT utilizando Árvore Balanceada de Busca.

Heap de Fibonacci					**Complexidade Teórica**
Arquivo	Vértices	Arestas	V. Processados	Tempo	|V|log|V| + E
alut2764	387	626	387	0,000240	3952,73
alut0805	966	1666	956	0,000658	11244,74
alut0787	1160	2089	1135	0,000774	13897,69
alut1181	3041	5693	2465	0,001757	40878,37
alut2566	5021	9055	3836	0,002798	70781,96
alut2010	6104	11011	4729	0,003430	87772,09
alut2288	9070	16595	8456	0,006452	135837,26
alut2610	33901	62816	32746	0,030643	572993,51
alut2625	36711	68117	36702	0,034778	624799,84

Tabela 8: Tabela de resultados do ALUT utilizando Heap de Fibonacci.

Buckets						**Complexidade Teórica**
Arquivo	Vértices	Arestas	V. Processados	C	Tempo	|V|C + |E|
alut2764	387	626	387	13	0,000035	5657,00
alut0805	966	1666	956	13	0,000078	14224,00
alut0787	1160	2089	1135	13	0,000102	17169,00
alut1181	3041	5693	2465	13	0,000219	45226,00
alut2566	5021	9055	3836	13	0,000373	74328,00
alut2010	6104	11011	4729	13	0,000461	90363,00
alut2288	9070	16595	8456	13	0,000875	134505,00
alut2610	33901	62816	32746	13	0,007657	503529,00
alut2625	36711	68117	36702	13	0,009155	545360,00

Tabela 9: Tabela de resultados do ALUT utilizando Buckets.

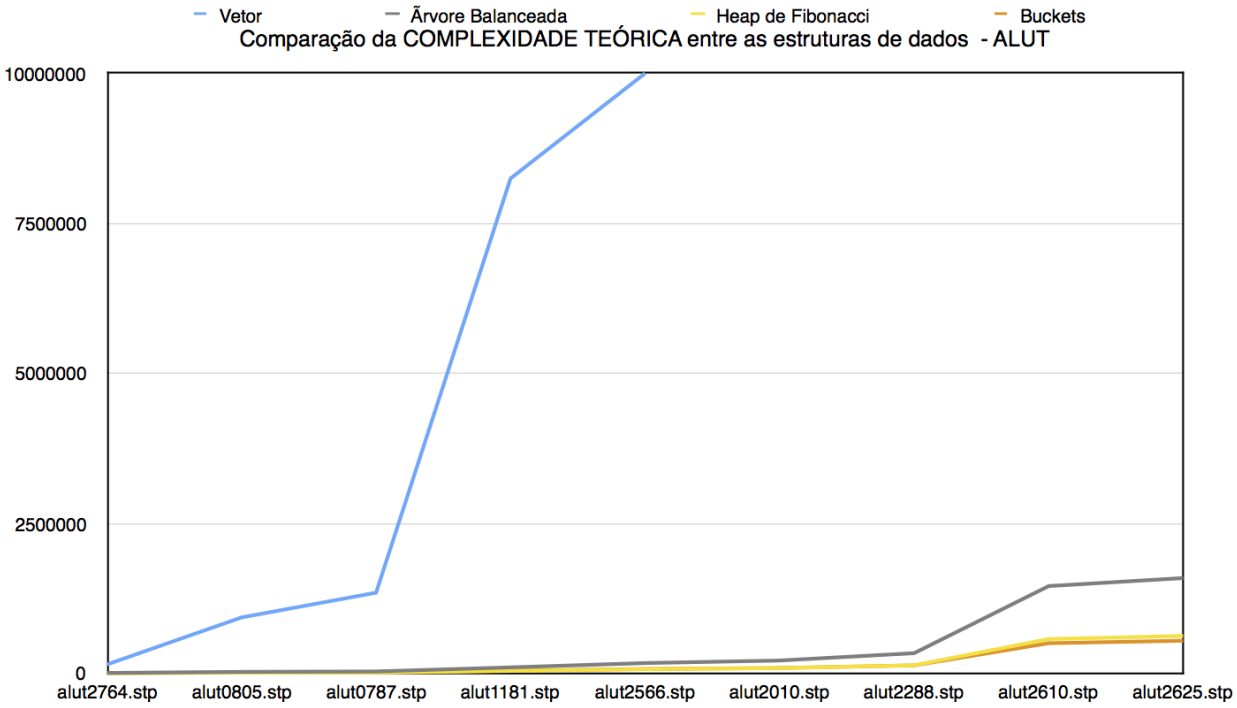

Figura 5: Gráfico comparativo das COMPLEXIDADES TEÓRICAS das estruturas de dados com as instâncias ALUT.

Árvore α				$\alpha = 0.6$	$\alpha = 0.7$	$\alpha = 0.8$	$\alpha = 0.9$
Arquivo	Vértices	Arestas	V. Processados	Tempo	Tempo	Tempo	Tempo
alut2764	387	626	387	0,000875	0,000908	0,001107	0,001187
alut0805	966	1666	956	0,001812	0,001893	0,002965	0,003833
alut0787	1160	2089	1135	0,002563	0,002663	0,002849	0,003428
alut1181	3041	5693	2465	0,005778	0,005999	0,008133	0,012162
alut2566	5021	9055	3836	0,007988	0,008059	0,015542	0,016673
alut2010	6104	11011	4729	0,008669	0,008797	0,018705	0,026794
alut2288	9070	16595	8456	0,016335	0,016554	0,028892	0,035797
alut2610	33901	62816	32746	0,072391	0,074272	0,162042	0,235736
alut2625	36711	68117	36702	0,081993	0,083734	0,180950	0,303431

Tabela 10: Tabela de resultados do ALUT utilizando Árvore α (com $\alpha = 0.6$).

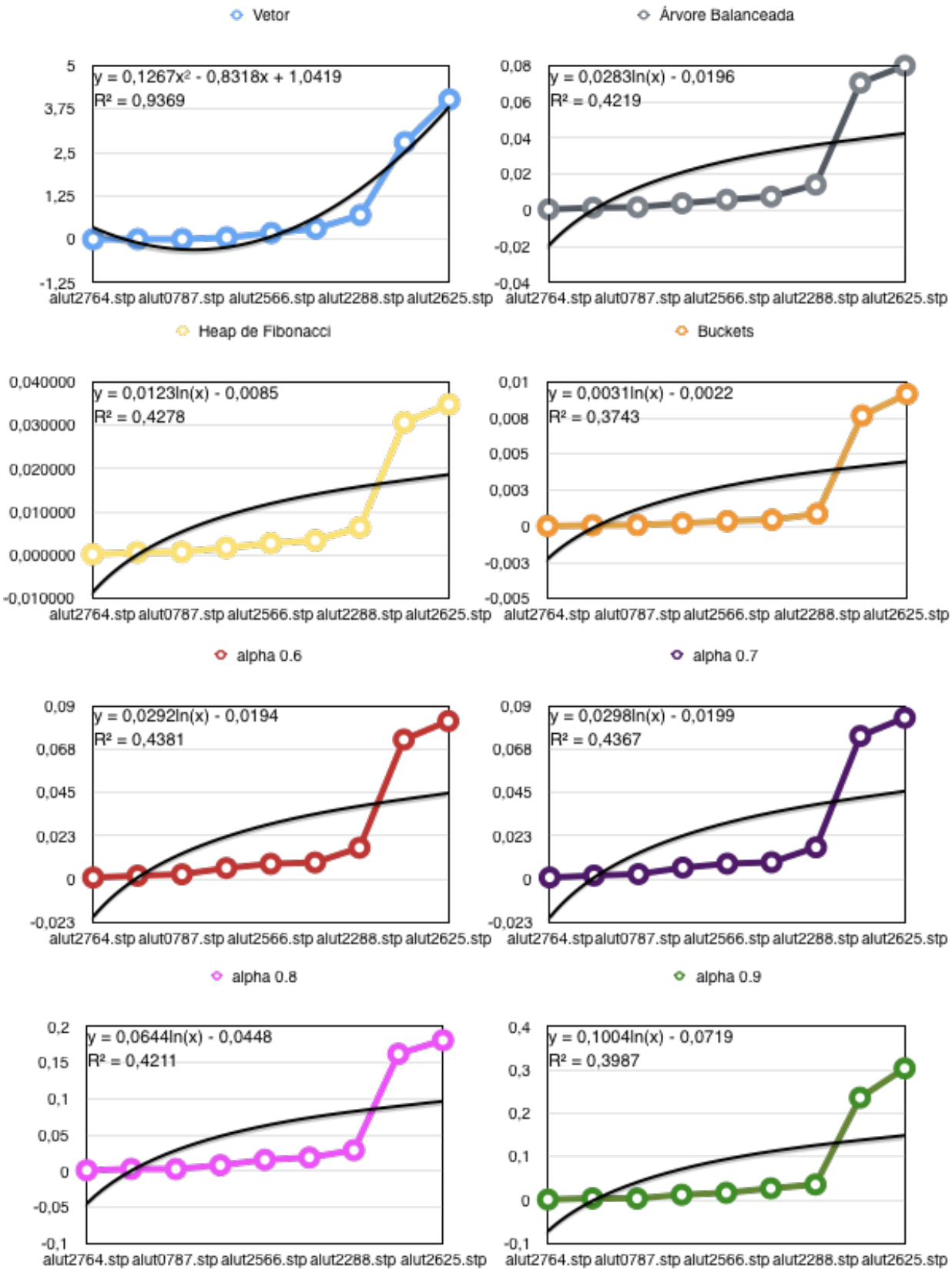

Figura 6: TEMPO DE CPU das estruturas de dados executadas com as instâncias ALUT.

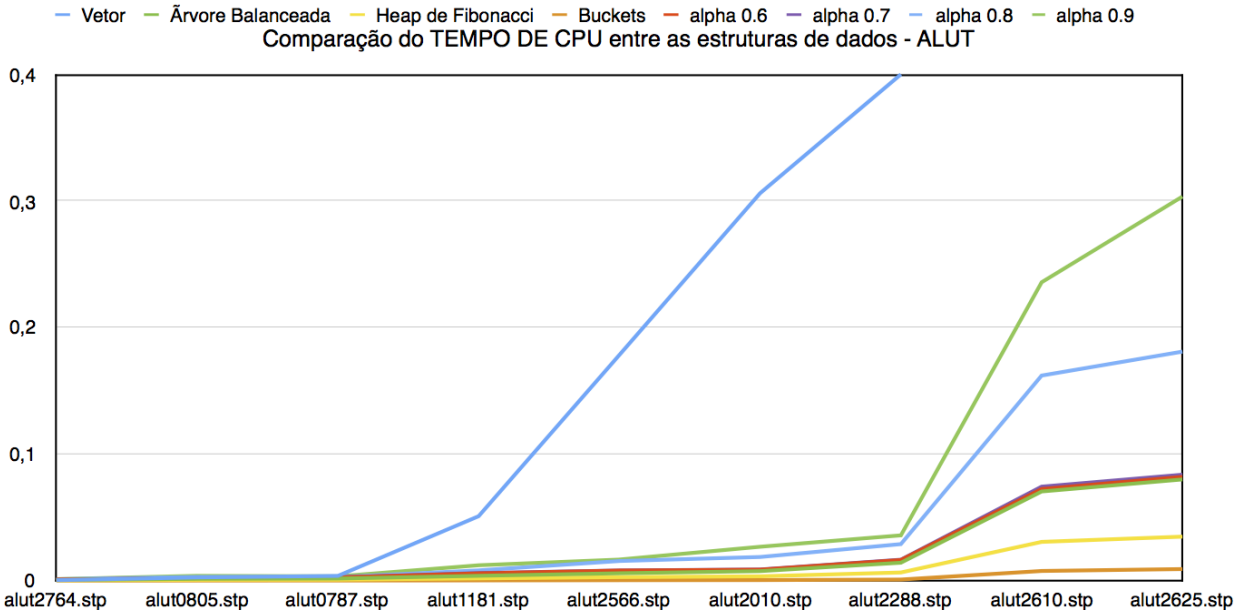

Figura 7: Gráfico comparativo de TEMPO DE CPU das estruturas de dados executadas com as instâncias ALUT.

Vetor					**Complexidade Teórica**
Arquivo	Vértices	Arestas	V. Processados	Tempo	$\|V\|^2$
dmxa0628	169	280	169	0,000091	28561
dmxa0296	233	386	233	0,000147	54289
dmxa1304	298	503	298	0,000268	88804
dmxa1109	343	559	343	0,000316	117649
dmxa0848	499	861	499	0,000749	249001
dmxa0903	632	1087	632	0,001125	399424
dmxa0734	663	1154	663	0,001228	439569
dmxa1516	720	1269	720	0,001468	518400
dmxa1200	770	1383	770	0,001749	592900
dmxa1721	1005	1731	1005	0,002817	1010025
dmxa0454	1848	3286	1848	0,017255	3415104
dmxa0368	2050	3676	2050	0,022631	4202500
dmxa1801	2333	4137	2333	0,031400	5442889
dmxa1010	3983	7108	3983	0,101520	15864289

Tabela 11: Tabela de resultados do DMXA utilizando Vetor.

Árvore Balanceada de Busca					**Complexidade Teórica**
Arquivo	Vértices	Arestas	V. Processados	Tempo	(|V|+|E|)log|V|
dmxa0628	169	280	169	0,000218	3322,99
dmxa0296	233	386	178	0,000228	4867,93
dmxa1304	298	503	298	0,000415	6583,55
dmxa1109	343	559	343	0,000506	7596,70
dmxa0848	499	861	455	0,000690	12189,54
dmxa0903	632	1087	588	0,000875	15993,20
dmxa0734	663	1154	663	0,000969	17030,50
dmxa1516	720	1269	646	0,001009	18879,30
dmxa1200	770	1383	713	0,001012	20644,50
dmxa1721	1005	1731	751	0,001207	27286,07
dmxa0454	1848	3286	911	0,001272	55712,88
dmxa0368	2050	3676	2033	0,003019	62994,06
dmxa1801	2333	4137	1821	0,002816	72386,17
dmxa1010	3983	7108	3778	0,005940	132644,36

Tabela 12: Tabela de resultados do DMXA utilizando Árvore Balanceada de Busca.

Heap de Fibonacci					**Complexidade Teórica**
Arquivo	Vértices	Arestas	V. Processados	Tempo	|V|log|V| + E
dmxa0628	169	280	169	0,000104	1530,75
dmxa0296	233	386	178	0,000109	2218,36
dmxa1304	298	503	298	0,000189	2952,31
dmxa1109	343	559	343	0,000221	3447,77
dmxa0848	499	861	455	0,000300	5333,49
dmxa0903	632	1087	588	0,000398	6966,99
dmxa0734	663	1154	663	0,000451	7368,21
dmxa1516	720	1269	646	0,000439	8103,13
dmxa1200	770	1383	713	0,000482	8766,31
dmxa1721	1005	1731	751	0,000503	11753,84
dmxa0454	1848	3286	911	0,000606	23340,03
dmxa0368	2050	3676	2033	0,001438	26228,89
dmxa1801	2333	4137	1821	0,001280	30238,54
dmxa1010	3983	7108	3778	0,002829	54743,25

Tabela 13: Tabela de resultados do DMXA utilizando Heap de Fibonacci.

Buckets						Complexidade Teórica
Arquivo	Vértices	Arestas	V. Processados	C	Tempo	|V|C + |E|
dmxa0628	169	280	169	13	0,000015	2477
dmxa0296	233	386	178	13	0,000019	3415
dmxa1304	298	503	298	13	0,000025	4377
dmxa1109	343	559	343	13	0,000029	5018
dmxa0848	499	861	455	13	0,000037	7348
dmxa0903	632	1087	588	13	0,000050	9303
dmxa0734	663	1154	663	13	0,000056	9773
dmxa1516	720	1269	646	13	0,000058	10629
dmxa1200	770	1383	713	13	0,000062	11393
dmxa1721	1005	1731	751	13	0,000070	14796
dmxa0454	1848	3286	911	13	0,000086	27310
dmxa0368	2050	3676	2033	13	0,000194	30326
dmxa1801	2333	4137	1821	13	0,000169	34466
dmxa1010	3983	7108	3778	13	0,000363	58887

Tabela 14: Tabela de resultados do DMXA utilizando Buckets.

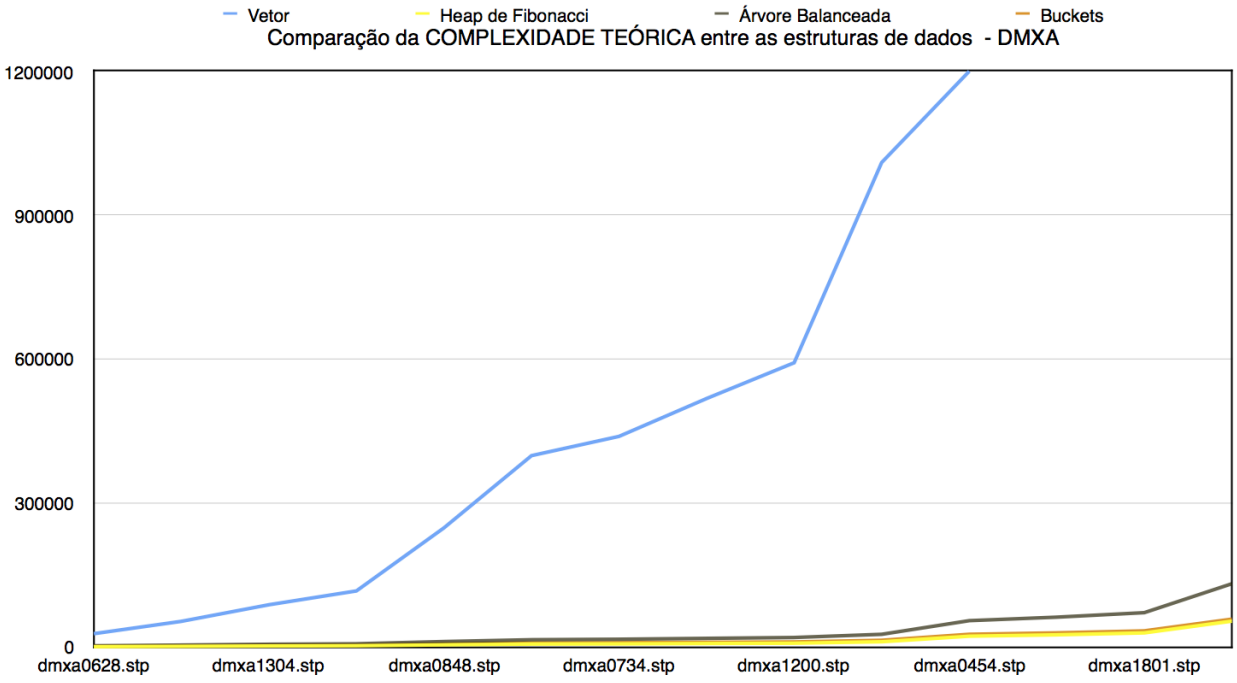

Figura 8: Gráfico comparativo das COMPLEXIDADES TEÓRICAS das estruturas de dados com as instâncias DMXA.

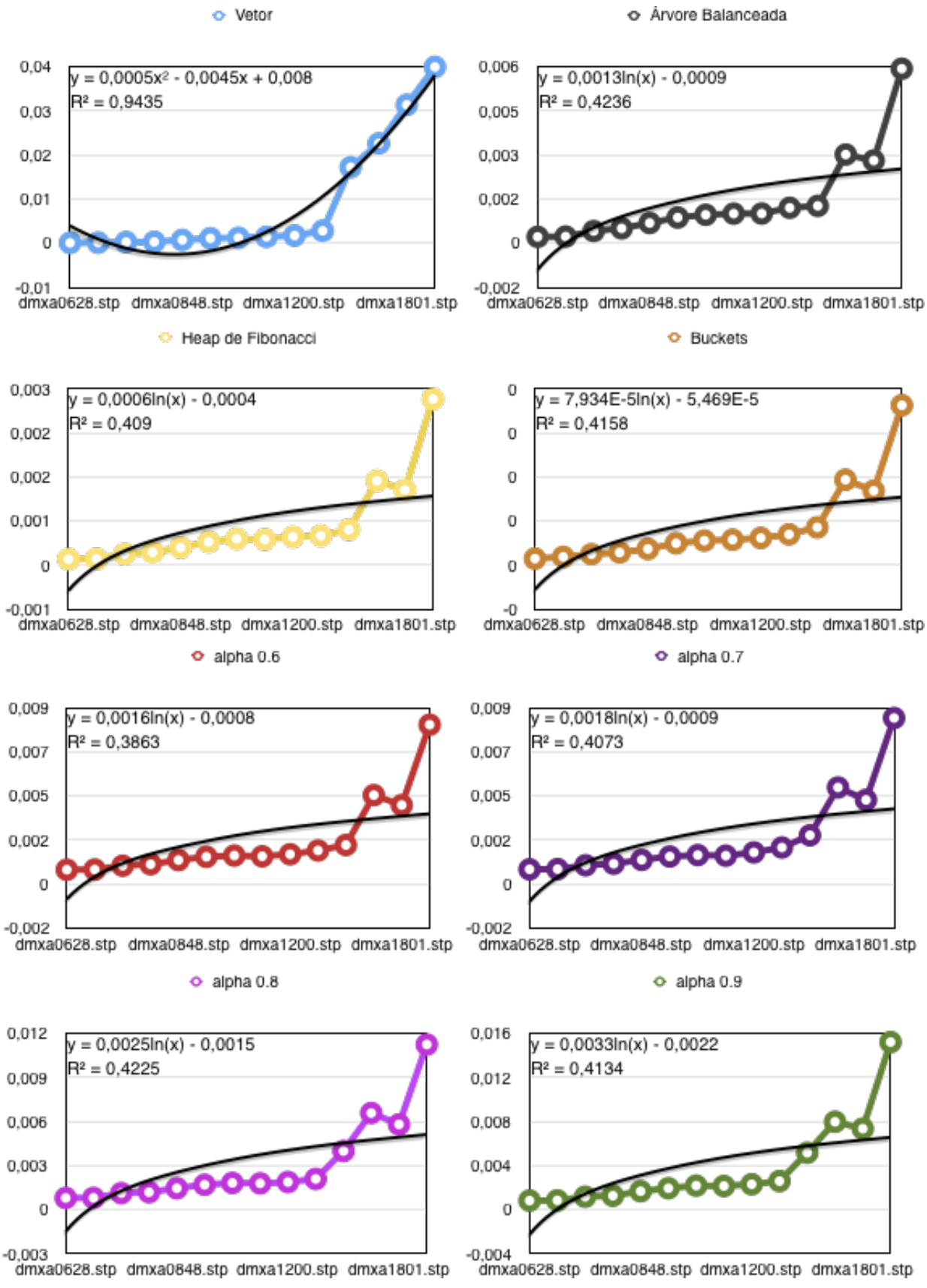

Figura 9: TEMPO DE CPU das estruturas de dados executadas com as instâncias DMXA.

Árvore α				$\alpha = 0.6$	$\alpha = 0.7$	$\alpha = 0.8$	$\alpha = 0.9$
Arquivo	Vértices	Arestas	V. Processados	Tempo	Tempo	Tempo	Tempo
dmxa0628	169	280	169	0,000723	0,000745	0,0008050	0,0008360
dmxa0296	233	386	178	0,000732	0,000754	0,0008090	0,0008550
dmxa1304	298	503	298	0,000927	0,000949	0,0011330	0,0011850
dmxa1109	343	559	343	0,001002	0,001029	0,0011970	0,0012760
dmxa0848	499	861	455	0,001215	0,001228	0,0014660	0,0016830
dmxa0903	632	1087	588	0,001382	0,001399	0,0017030	0,0019540
dmxa0734	663	1154	663	0,001447	0,001471	0,0018280	0,0021920
dmxa1516	720	1269	646	0,001409	0,001441	0,0017980	0,0021570
dmxa1200	770	1383	713	0,001512	0,001607	0,0018890	0,0023150
dmxa1721	1005	1731	751	0,001707	0,001857	0,0021000	0,0026040
dmxa0454	1848	3286	911	0,001985	0,002481	0,0040000	0,0051770
dmxa0368	2050	3676	2033	0,004529	0,004922	0,0065850	0,0079870
dmxa1801	2333	4137	1821	0,004023	0,004287	0,0058130	0,0073610
dmxa1010	3983	7108	3778	0,008125	0,008469	0,0112490	0,0152190

Tabela 15: Tabela de resultados do DMXA utilizando Árvore α (com $\alpha = 0.6$).

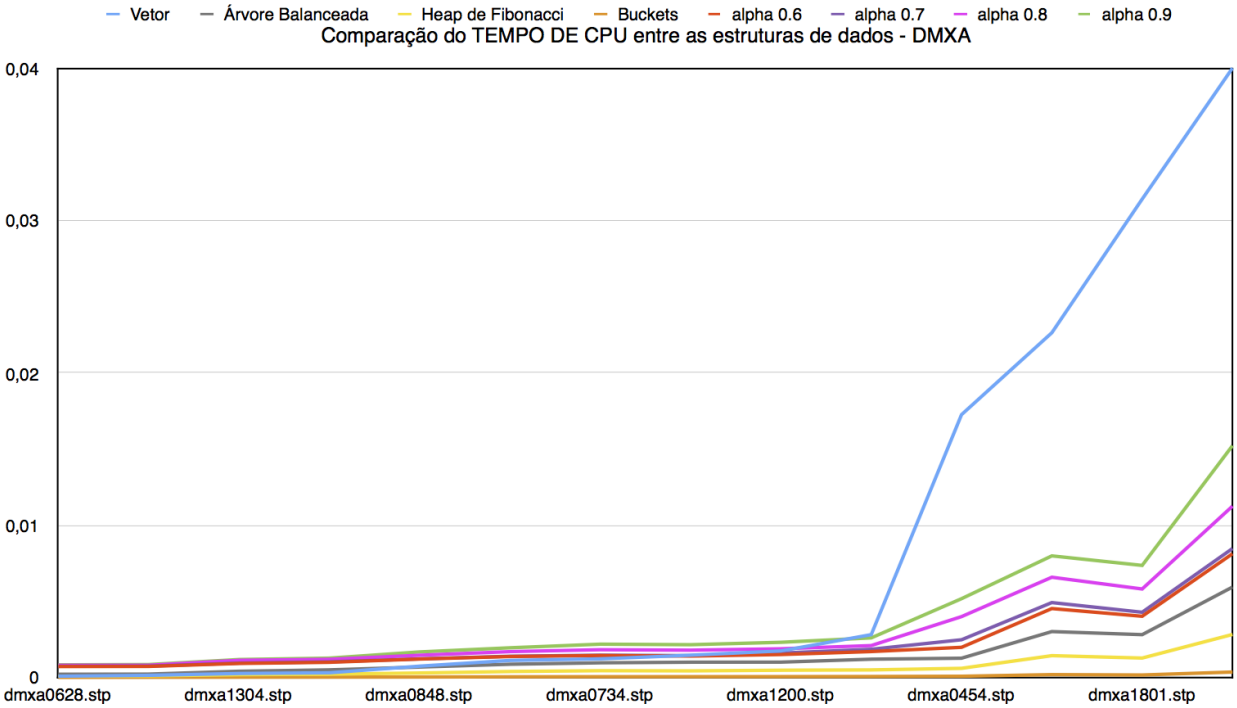

Figura 10: Gráfico comparativo de TEMPO DE CPU das estruturas de dados executadas com as instâncias DMXA.

As figuras 3, 6, 9, mostram como se comportou o algoritmo com diferentes estruturas de dados. Em preto, desenhamos uma linha de tendência de acordo com a complexidade teórica calculada. Em algoritmos polinomiais, desenhamos uma função polinomial, e nos demais desenhamos uma função envolvendo logaritmos em função do número de

vértices.

Nota-se uma elevação no tempo das última instâncias em relação a linha de tendência logarítmica apresentada. Isso deve-se ao fato de que a última instância apresenta um crescimento desproporcional no número de arestas, o que não é contabilizado pelas funções tendência apresentadas. Isso ocorre pois os gráficos foram ilustrados em 2 dimensões, o que impossibilita múltiplas variáveis.

Conclusão

Para todas as instâncias, obteve-se a complexidade teórica e desempenho esperado para cada operação utilizada com as estruturas de dados, como mostram as tabelas da Seção de Resultados. Ao comparar o tempo de CPU, é possivel notar a seguinte hierarquia: **Buckets < Heap de Fibonacci < Árvore Balanceada de Busca < Árvore α(0.6) < Árvore α(0.7) < Árvore α(0.8) < Árvore α(0.9) < Vetor**.

Em grafos densos, o número de arestas é proporcional a n^2. Essa hierarquia nos resultados, e as complexidades analisadas nos capítulos anteriores (tabela abaixo), mostram que o input usado neste trabalho era composto de grafos não tão densos.

Operações	Vetor	Árvore BB	H. de Fibonacci	Buckets
Inicializar	O(n)	O($n.log(n)$)	O(n)	O(n)
RemoverMin	O(n^2)	O($n.log(n)$)	O($n.log(n)$)	O(nC)
Atualizar	O(n^2)	O($m.log(n)$)	O(m)	O(m)
Total	O(n^2)	O($(n+m).log(n)$)	O($n.log(n)+m$)	O($nC+m$)

Tabela 16: Tabela com o resumo das complexidades teórica.

Observa-se que a inicialização do algoritmo tem menor complexidade que o seu corpo. Assim, essa inicialização não influencia a complexidade final. A complexidade depende predominantemente do custo de se remover o menor elemento e de se atualizar um elemento na estrutura, e do tamanho/densidade do grafo.

Os resultados mostraram que os desempenhos do algoritmo Dijkstra com o uso de Heap de Fibonacci e Buckets são próximos para quantidades pequenas de vértices. O desempenho do Buckets supera o Heap de Fibonacci para quantidades grandes de vértices, isso deve-se ao limite superior de nC iterações definidos pelo algoritmo, já que para todos os arquivos de entrada, a aresta com maior peso era sempre C=13.

É importante notar que, com o uso de Buckets, o algoritmo é pseudo-polinomial em relação a aresta com maior peso. Portanto, com C constante temos uma complexidade O($n+m$), que supera a complexidade de O($n.logn+m$) do Heap de Fibonacci. Caso não soubéssemos de antemão o comprimento máximo da representação do tamanho de uma aresta, seria inviável fixar a constante C, e o Buckets não teria performance melhor que as outras estruturas de dados.

Em seguida, nota-se que o desempenho da Árvore Balanceada de Busca é bem superior ao Vetor. Caso os grafos de entrada fossem densos, o número de arestas seria proporcional a n^2, o que faria a Árvore Balanceada de Busca custar O($n^2.log(n)$), superior a complexidade total do Vetor. Mais uma vez, podemos notar que os grafos de entrada

não eram muito densos.

Conforme analisado na seção 1.5, a complexidade do algoritmo Dijkstra com o uso da árvore-α varia entre O$((n + m).log(n))$ e O(n^2) para $1/2 < \alpha < l$ 1. Isso foi corroborado pelos resultados obtidos na execução do algoritmo para os valores 0.6, 0.7, 0.8 e 0.9. É visível o crescimento do tempo de CPU conforme o aumento do α, onde: α=0.6 é limite superior de O$((n + m).log(n))$ da Árvore Balanceada de Busca; E α=0.9, é limitado superiormente pelo O(n^2) do Vetor.

Referências

DASGUPTA, S.; PAPADIMITRIOU, C.; VAZIRANI, U. *Algorithms*. [S.l.]: McGraw-Hill Education, 2006. 119–127 p. Citado 2 vezes nas páginas 5 e 7.

DIJKSTRA, E. W. A note on two problems in connexion with graphs. *Numer. Math.*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 1, n. 1, p. 269–271, dez. 1959. ISSN 0029-599X. Disponível em: <http://dx.doi.org/10.1007/BF01386390>. Citado na página 3.

KLEINBERG, J.; TARDOS, E. *Algorithm design*. [S.l.]: Pearson, 2005. Citado na página 3.

MISA, T. J.; FRANA, P. L. An interview with edsger w. dijkstra. *Commun. ACM*, ACM, New York, NY, USA, v. 53, n. 8, p. 41–47, ago. 2010. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1787234.1787249>. Citado na página 3.

NIEVERGELT, J.; REINGOLD, E. M. Binary search trees of bounded balance. *SIAM journal on Computing*, SIAM, v. 2, n. 1, p. 33–43, 1973. Citado na página 9.