UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Henry Bernardo Kochenborger de Avila

Marcos Samuel Winkel Landi

MINERAÇÃO DE OPINIÃO

Relatório do projeto

Porto Alegre, Rio Grande do Sul

2018

SUMÁRIO

1. INTRODUÇÃO

Como trabalho final da cadeira de Estrutura de Dados do Instituto de Informática da Universidade Federal do Rio Grande do Sul (UFRGS), este relatório terá um enfoque pragmático ante as diferenças entre o uso das Árvores Binárias de Pesquisa (ABP) e das Árvores AVL para calcular o valor das opiniões do público em relação a um certo filme.

1.1 Apresentação do tema e do problema

Existem inúmeras formas de analisar uma crítica a um filme, música ou pintura. Uma delas é conhecida como mineração de opinião, que, essencialmente, utiliza uma base seleta de dialetos e frases mais comuns que possuem um valor inteiro definindo seu significado. Ou seja, uma palavra que condiz um sentimento bom em relação a algo possui valor inteiro positivo – de forma análoga, uma palavra negativa possui valor inteiro negativo e palavras irrelevantes possuem valor nulo. Dessa forma, é possível calcular se uma crítica foi positiva ou negativa somando todos os valores presentes na frase.

Assim sendo, este trabalho utilizará esse conceito como tema para analisar as diferenças entre as duas árvores já citadas.

1.1.1 Objetivos

Utilizando um programa escrito na linguagem C, serão feitas afirmações quanto as comparações ocorridas na execução desse programa. Dessa forma, será possível analisar, por meio de contadores de comparações – que ficarão mais claros futuramente –, qual dos dois tipos de dados são mais custosos computacionalmente para cada caso.

2. TIPOS ABSTRATOS DE DADOS

Como citado anteriormente, esse trabalho tratará de dois tipos abstratos de dados para verificar suas diferenças quanto a eficiência. Por consequência, é importante esclarecer o conceito de tipo abstrato de dados: um TAD (tipo abstrato de dado) é uma estrutura de dados que suporta algumas operações específicas que facilitam a manipulação da informação – árvore binária é, pelo menos na computação, o tipo mais conhecido.

Árvores binárias são estruturas recursivas baseadas em grafos. Nesse tipo de grafo, cada vértice tem grau no máximo três e possui um pai e no máximo dois filhos.

2.1 Árvore Binária de Pesquisa

Sendo uma categoria das árvores binárias, as ABP's possuem nodos com relações de ordem entre si, sendo os nodos a esquerda com valores menores (quando tratando-se de números) e os nodos a direita com valores maiores. No caso deste trabalho, os nodos serão organizados quanto a ordem lexicográfica.

Dentro do programa, o tipo referente a essa árvore é o tipo BSTNode, possuindo os seguintes componentes:

- char info[50]: é a string que será guardada nesse vértice.
- int value: trata-se do valor do vértice.
- BSTNode *left: é a subárvore esquerda do vértice.
- BSTNode *right: é a subárvore direita do vértice.

2.1.1 Funções

Para o tratamento da informação organizada na estrutura de ABP, foram usadas essas funções:

- BSTNode* BST_initialize(): retorna um ponteiro nulo com o intuito de inicializar a árvore.
- BSTNode* BST_insert(BSTNode*, char*, int): insere um novo vértice na árvore informada com o valor também dado na chamada da função, devolvendo a árvore com essa inserção.

- BSTNode* BST_search(BSTNode*, char*): dada uma árvore e uma string, essa função retorna o vértice que contém essa string. Caso não exista esse vértice, retorna um ponteiro nulo.
- BSTNode* BST_maxNode(BSTNode*): retorna o vértice com maior valor da árvore quanto a ordem já citada.
- int BST_height(BSTNode*): retorna o tamanho da árvore informada.
- BSTNode* BST_destroy(BSTNode*): dada uma árvore, essa função desaloca a memória utilizada por essa árvore e retorna um ponteiro nulo.

Observação: o arquivo referente a esse tipo presente no programa possui outras funções para manipulação, mas que não foram usadas durante a execução da análise.

2.2 Árvore AVL

As árvores AVL tratam-se de um subconjunto das árvores binárias de pesquisa com alguns critérios que as tornam mais eficientes em alguns casos.

Com o intuito principal de aumentar a eficiência, Adelson Velsky e Landis criaram uma estrutura que tenta manter a árvore cheia e, por esta razão, diminuir a sua altura – dessa forma, o número de comparações tende a diminuir.

Dentro do programa, foi criado um tipo que faz referência às árvores AVL, sendo ele o tipo AVLNode, possuindo as seguintes características:

- char info[50]: é a string que será guardada nesse vértice.
- int value: trata-se do valor do vértice.
- int balanceFactor: trata-se do valor referente ao balanceamento deste nodo. Essencialmente, é a diferença da altura da sua subárvore esquerda da direita.
- AVLNode *left: é a subárvore esquerda do vértice.
- AVLNode *right: é a subárvore direita do vértice.

2.2.1 Funções

Como forma de manipulação dos dados armazenados na estrutura apresentada acima, foram utilizadas essas funções:

- AVLNode* AVL_initialize(): retorna um ponteiro nulo com o intuito de inicializar a árvore.

- AVLNode* AVL_insert(AVLNode*, char*, int*, int): insere um novo vértice na árvore informada com o valor também dado na chamada da função, devolvendo a árvore com essa inserção e alterando o balanceamento dado pelo ponteiro de inteiro.
  Essa função utiliza as funções de rotação descritas abaixo.
- AVLNode* AVL_search(AVLNode*, char*): dada uma árvore e uma string, essa função retorna o vértice que contém essa string. Caso não exista esse vértice, retorna um ponteiro nulo.
- AVLNode* AVL_maxNode(AVLNode*): retorna o vértice com maior valor da árvore quanto a ordem já citada.
- int AVL_height(AVLNode*): retorna o tamanho da árvore informada.
- AVLNode* AVL_destroy(AVLNode*): dada uma árvore, essa função desaloca a memória utilizada por essa árvore e retorna um ponteiro nulo.
- AVLNode* AVL_rotateLeft(AVLNode*): dada uma árvore, essa função rotaciona a árvore para esquerda – normalmente para balanceá-la.
- AVLNode* AVL_rotateRight(AVLNode*): dada uma árvore, essa função rotaciona a árvore para direita – normalmente para balanceá-la.
- AVLNode* AVL_double_rotateLeft(AVLNode*): dada uma árvore, essa função faz uma dupla-rotação para esquerda – normalmente para balanceá-la – que, essencialmente, é uma rotação à direita e outra à esquerda (nessa sequência).
- AVLNode* AVL_double_rotateRight(AVLNode*): dada uma árvore, essa função faz uma dupla-rotação para direita – normalmente para balanceá-la – que, essencialmente, é uma rotação à esquerda e outra à direita (nessa sequência).

3. DESENVOLVIMENTO

3.1 Comparações

De modo a efetuar a análise quanto a eficiência de cada árvore, foram criadas duas variáveis para cada tipo que se referem a quantidade de comparações (if's, while's e do-while's) que ocorrem nas funções de busca e de inserção. Sendo assim, as variáveis são:

- comp_insert_BST: trata-se do número de comparações que ocorreram durante uma execução do programa ao inserir algum nodo numa ABP.
- comp_search_BST: trata-se do número de comparações que ocorreram durante uma execução do programa ao buscar por algum nodo em uma ABP.
- comp_insert_AVL: trata-se do número de comparações que ocorreram durante uma execução do programa ao inserir algum nodo numa árvore AVL.
- comp_search_AVL: trata-se do número de comparações que ocorreram durante uma execução do programa ao buscar por algum nodo em uma árvore AVL.

3.1.1 Gráficos

Utilizando os contadores citados acima e os casos de teste disponibilizados pelas professoras Viviane P. Moreira e Renata Galante, chegamos as seguintes conclusões:

- Caso 1:

  É interessante reparar o comportamento da inserção e da busca em cada árvore. Com o gráfico abaixo, é fácil notar que as rotações presentes nas inserções da AVL aumentam as comparações efetuadas tornando-a mais custosa quando é necessário pôr algum nodo na árvore.

Na etapa de busca, as rotações se compensam fazendo com que a diferença no número de comparações nesta fase chegue a quase 90 mil.

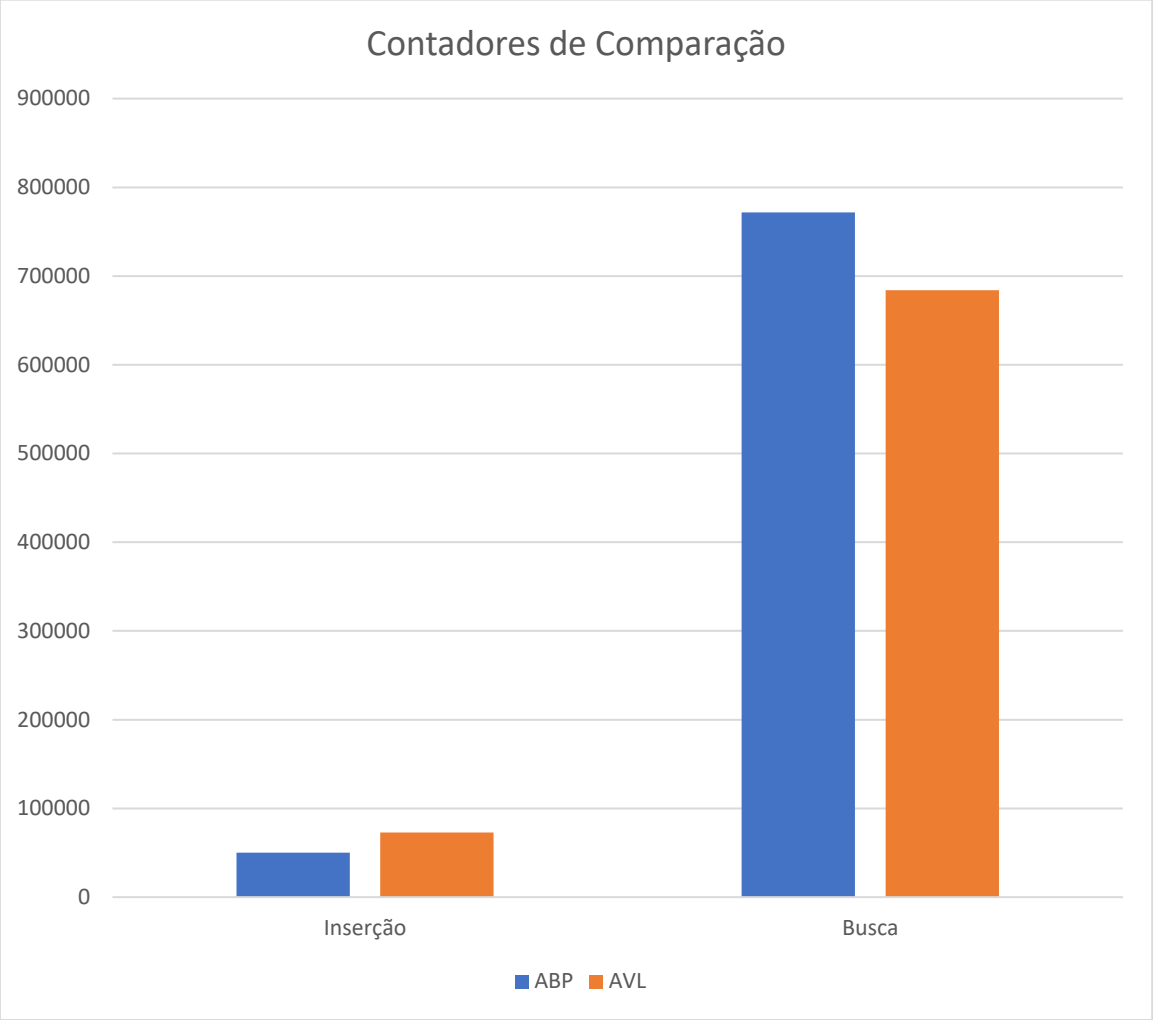

Figura 1: valor atribuído aos contadores no fim da execução.

Já em relação à altura, o gráfico mostra que a maneira em que foram inseridos os vértices não foi das piores, fazendo com que a ABP, mesmo com dois mil nodos, possua 26 de altura, e a AVL, 11 – esta que não depende da maneira que foram inseridos os elementos, visto que, com as rotações, a árvore torna-se balanceada.

Sobre o tempo de execução, a diferença é irrisória e se dá pelo fato da forma em que os elementos foram inseridos ter sido interessante para a ABP, visando a eficiência.

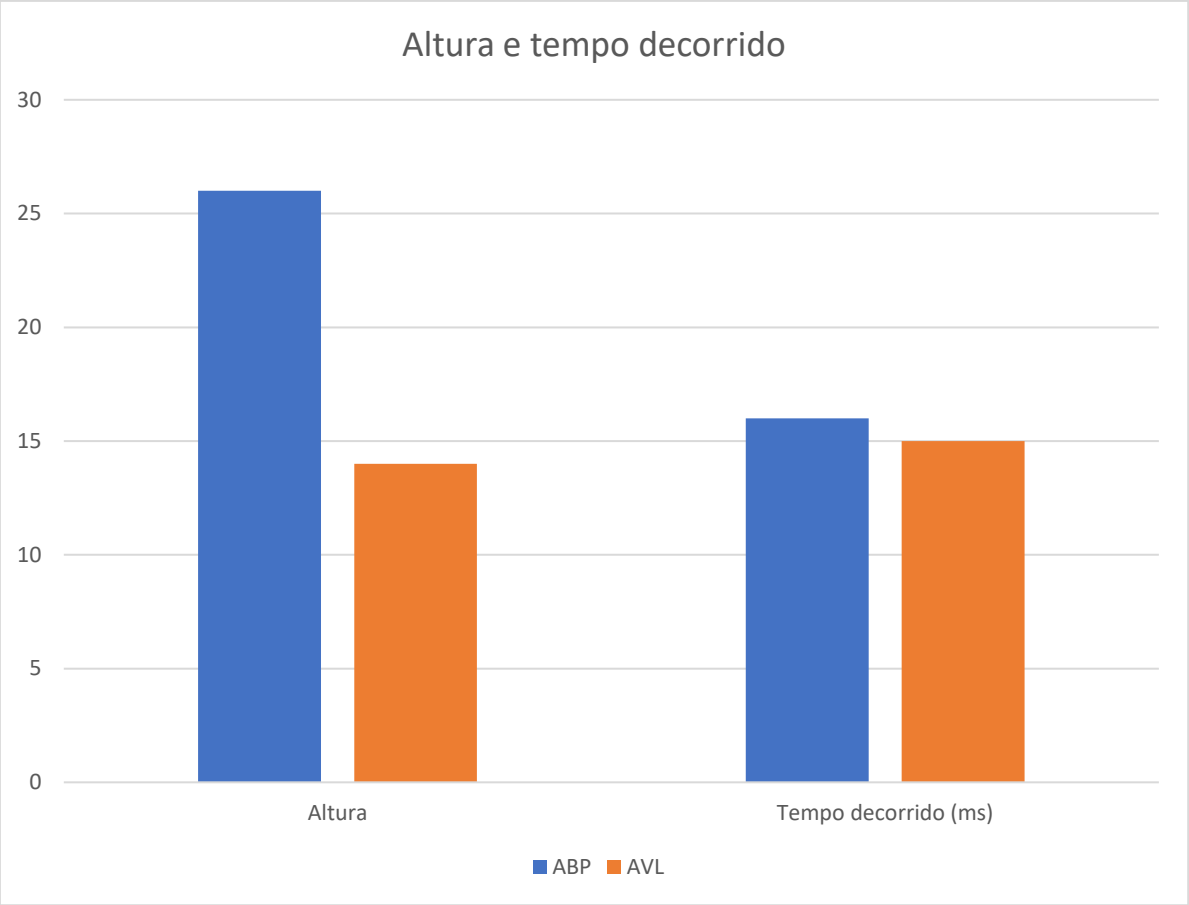

Figura 2: altura das árvores e o valor atribuído às variáveis relacionadas ao tempo de execução.

- Caso 2:

Neste caso, a maneira em que os elementos foram inseridos nas árvores demonstra o quão ineficiente uma ABP pode ser se não balanceada.

A inserção dos elementos foi feita em ordem crescente (lexicográfica), fazendo com que a ABP virasse uma lista ordenada, perdendo sua utilidade quanto árvore. Dessa forma, é possível ver nos gráficos a seguir a gigante disparidade entre elas, chegando a incríveis 390 mil comparações a mais na ABP durante a etapa de inserção e absurdos 253 milhões de comparações a mais para a ABP durante as buscas.

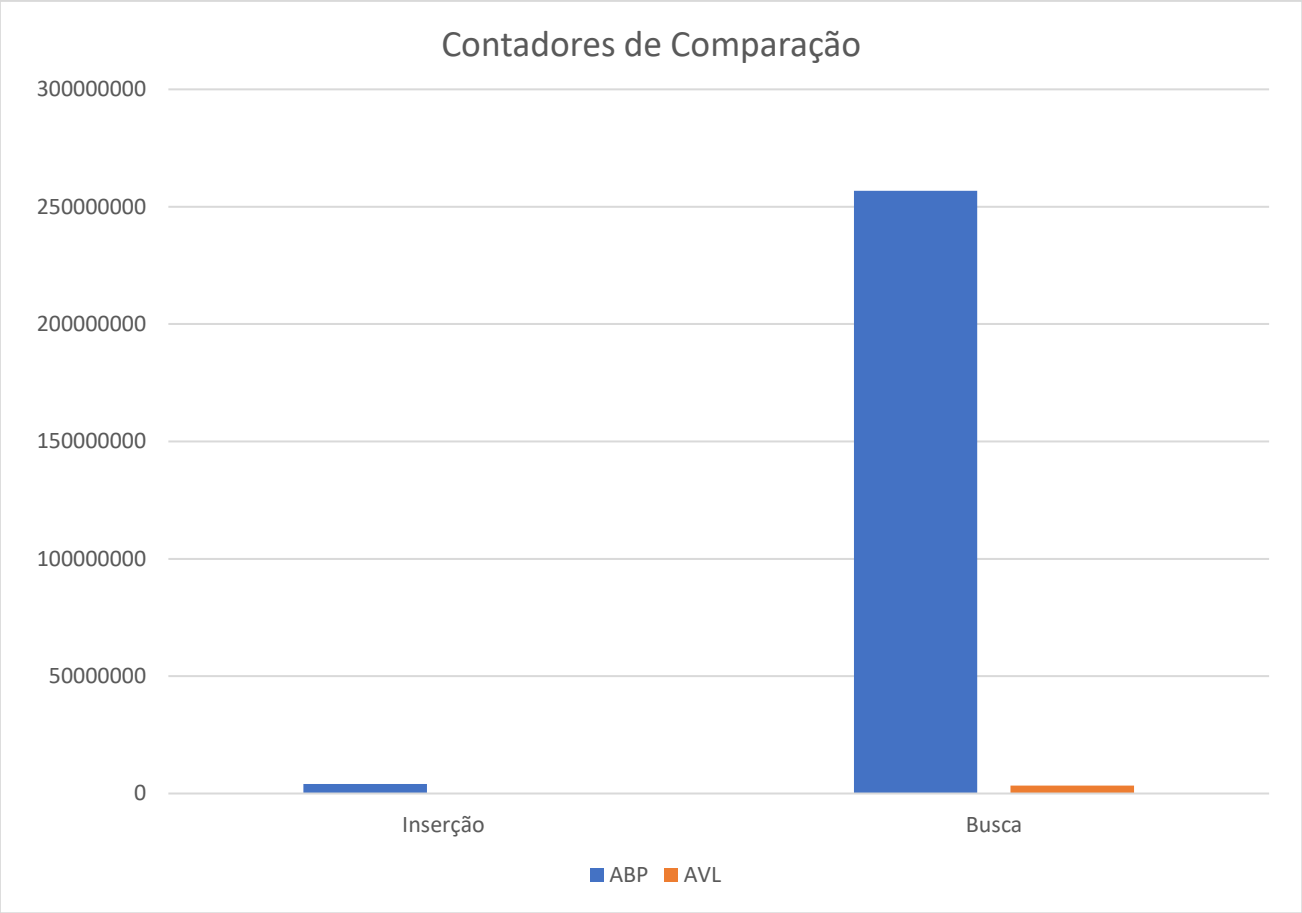

Figura 3: valor atribuído aos contadores no fim da execução.

Quando se trata de altura, a disparidade continua grande. Como dito anteriormente, a inserção em ordem lexicográfica fez com que a ABP virasse uma lista, logo sua altura é o número de elementos inseridos, diferentemente da AVL, que durante a inserção faz rotações para melhorar o balanceamento da árvore. Dessa forma, é possível perceber uma diferença na casas de mil e novecentos.

O tempo de execução só confirma as grandiosas divergências. Para efetuar as inserções e as buscas na ABP, foram decorridos 1122 milissegundos, ou, 1,122 segundos – um tempo muito relevante tratando-se de manipulação de dados. Já para a AVL, devido ao seu balanceamento incrível, decorreram-se 62 milissegundos, um valor de dar inveja a qualquer ABP.

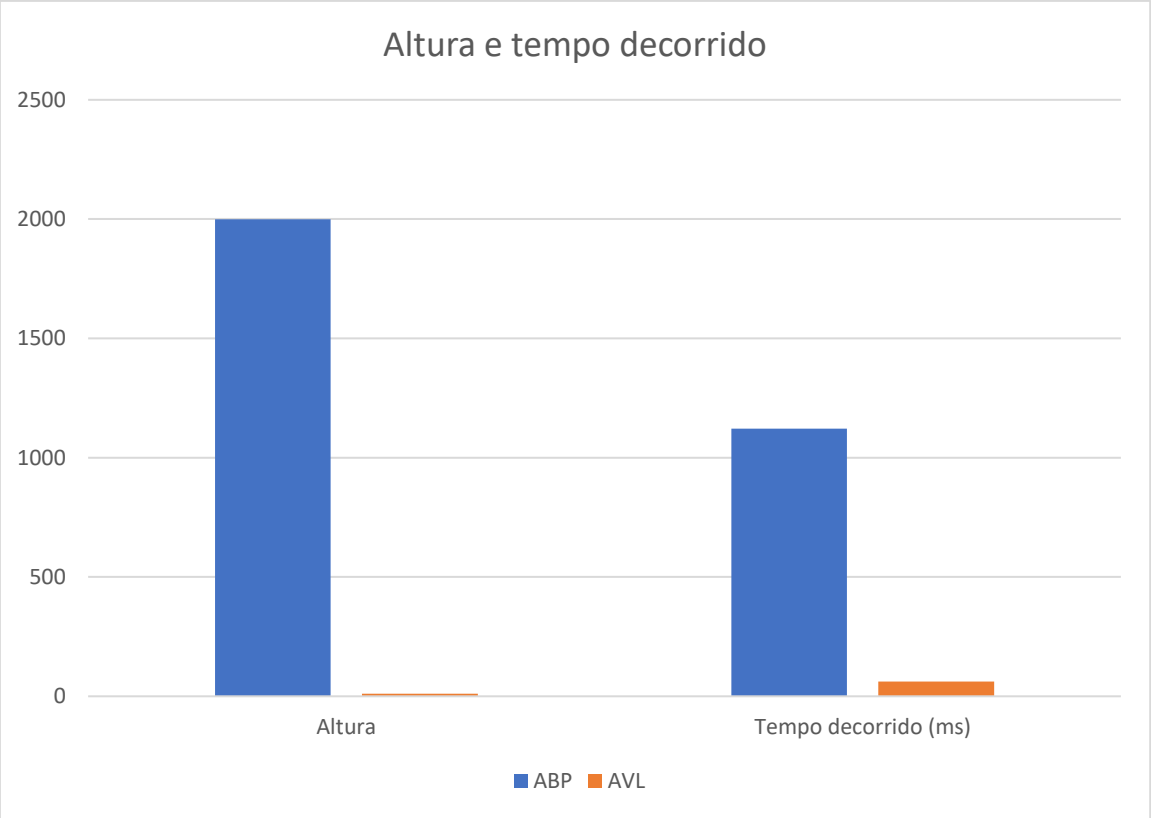

Figura 4: altura das árvores e o valor atribuído às variáveis relacionadas ao tempo de execução.

3.2 Bônus – Gráficos em "Tempo Real"

Para uma visualização mais imediata, o número de comparações de inserção e busca, os tempos e as alturas são registradas em um arquivo "python_input.txt", que mais tarde pode ser utilizado por um script que foi desenvolvido em Python: plot_data.

O programa plot_data utiliza a biblioteca matplotlib para exibir 4 gráficos em uma mesma janela: Heights (alturas), Time (tempo), Insert (insere) e Search (busca).

Para fácil execução e compatibilidade, o script foi transformado em um executável, usando a ferramenta py2exe. Basta clicar duas vezes que o programa lê os dados de python_input.txt, cria uma imagem comparison_plot.png e exibe-a.

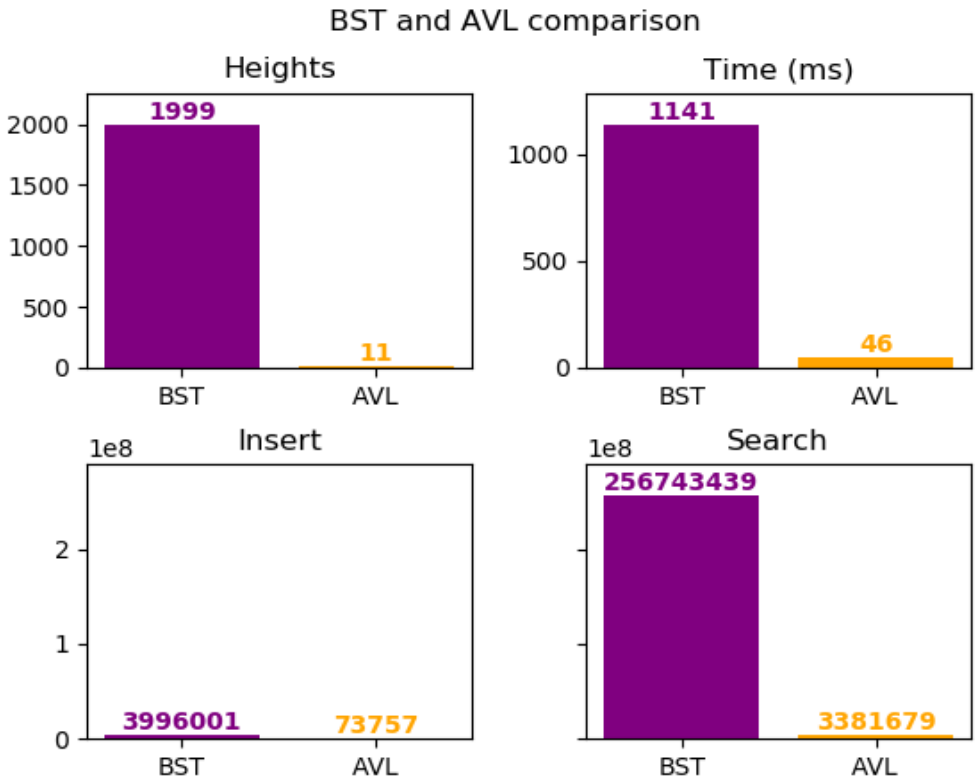

Figura 5: Gráfico gerado por plot_data, utilizando o "caso 2" da seção anterior.