Pontifícia Universidade Católica
do Rio de Janeiro

Projeto Final de Programação

Busca Ortogonal em Kd-Tree de 2 Dimensões

André Ricardo Ducca Fernandes

Orientador: Prof. Hélio Côrtes Vieira Lopes

29 de maio de 2020

Conteúdo

1 - Introdução

A princípio quando pensamos em banco de dados, geometria não é o primeiro campo que nos vem a mente. Contudo um grande número de perguntas (queries) sobre os dados, podem ser respondidas de forma geométrica. Para tanto, basta pensarmos nos dados como pontos no espaço. Vamos considerar como exemplo a base de dados de uma empresa que contenha os registros de cada funcionário, tais como nome, endereço, data de aniversário, salário entre outras coisas. Uma pergunta que podemos querer responder seria, quem são os funcionários nascidos entre 1950 e 1955 que ganham entre $3000 e $4000 por mês?

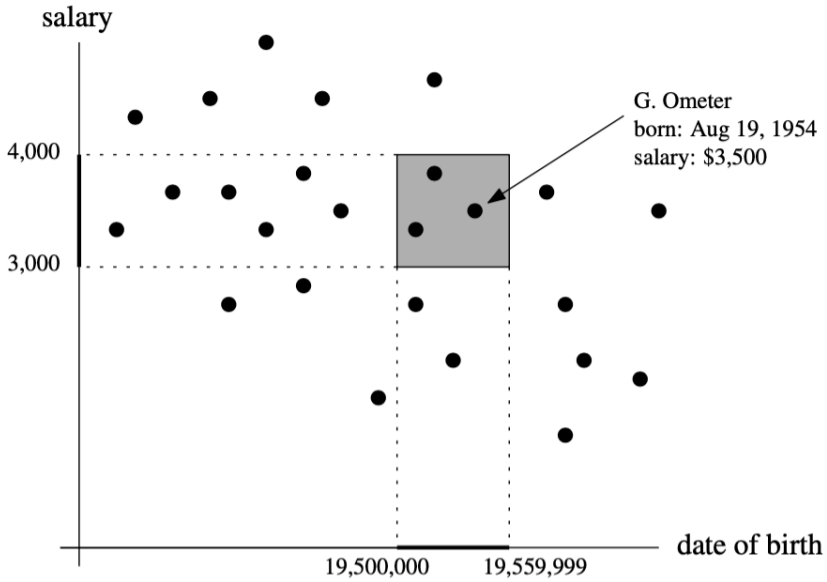

Figura 1: Interpretando base de dados geometricamente. Figura extraída de Berg et al., 2008.

Para respondermos essa pergunta de uma forma geométrica, representamos cada funcionário como um ponto no plano. A primeira coordenada deste ponto seria a data de nascimento, que será representada por um inteiro de forma que seja 10.000 x anos + 100 x meses + dias, e a segunda o salário mensal. A partir disto, podemos considerar que queremos todos os pontos cuja primeira coordenada esteja entre 19.500,000 e 19,559,999, e cuja segunda coordenada enteja entre 3000 e 4000. Em outras palavras, queremos todos os pontos que estão dentro de um retângulo paralelo aos eixos considerados e limitados pela query.

Neste exemplo, o nosso problema pode ser mapeado em 2 dimensões. Contudo, podemos acrescentar quantas restrições quisermos à query, isso apenas aumentaria o número de dimensões com as quais trabalharíamos.

Este tipo de query é chamada de Busca Ortogonal e neste trabalho iremos nos ater a queries com 2 dimensões. Para tanto, utilizaremos uma estrutura de dados chamada Kd-tree para nos auxiliar.

2 – Descrição da Técnica

Dado um conjunto de pontos $P=\{p1, p2, ..., pn\}$, onde cada ponto possui uma coordenada x e uma coordenada y, e uma query com duas restrições $[x, x']$ e $[y, y']$, formando

um retângulo, precisamos encontrar um conjunto de pontos tal que px pertença ao intervalo *[x, x']* e py pertença ao intervalo *[y, y']*.

Em primeiro lugar, precisamos descrever a estrutura de dados que iremos utilizar para fazer a busca ortogonal em 2 dimensões. A estrutura que será implementada se baseia em uma árvore binária balanceada. Para tanto, as folhas da árvore guardarão os pontos *P*, enquanto os nós internos guardarão apenas os valores utilizados para dividir os dados, que chamaremos de linha de corte e serão responsáveis por guiar a busca. Assumimos que o valor da linha de corte divide os dados de tal forma que a sub árvore à esquerda possua apenas valores menores ou iguais a ele, enquanto a sub árvore da direita possua apenas valores maiores. Como é uma árvore balanceada, escolhemos a mediana para fazer essa divisão. Dado que cada ponto possui uma coordenada x e y, uma convenção escolhida é que sempre quando estamos em um nível par da árvore utilizamos a mediana das coordenadas x para fazer o corte, enquanto se estivermos em um nível ímpar da árvore, escolhemos a mediana das coordenadas y.

A partir do nó raiz, quando estamos no nível zero da árvore, dividimos o conjunto de pontos *P* com uma linha de corte vertical (mediana da coordenada *x* dos pontos), em dois subconjuntos de tamanhos iguais. Neste momento, os pontos que possuem um *x* menor ou igual à linha de corte são passados para a sub árvore da esquerda, enquanto os nós que possuem um *x* maior que a linha de corte vão para a sub árvore da direita. O nó raiz guarda apenas o valor da linha de corte utilizada para dividir os dados. Espacialmente, todos os nós à esquerda ou sobre a linha de corte pertencem à sub árvore da esquerda, enquanto todos os nós a direita pertencem a sub árvore da direita.

Agora se descermos para, por exemplo, o filho da esquerda do nó raiz, estaremos no nível um da árvore. Neste caso, devemos dividir o conjunto de pontos deste nó utilizando uma linha de corte horizontal, que é a mediana da coordenada *y* dos pontos que pertencem a este nó. Desta forma, todos os pontos abaixo ou sobre a linha de corte pertencem a sub árvore da esquerda e, desta forma, têm um *y* menor ou igual a mediana, enquanto os pontos que estão acima da linha de corte pertencem a sub árvore da direita e possuem um *y* maior que a mediana.

Esse processo deve ser feito até ficarmos com apenas um ponto. Neste caso, este nó será um nó folha e irá guardar o ponto. Todos os nós internos da árvore guardam apenas os valores da linha de corte, enquanto os nós folha guardam os pontos. A essa estrutura, dá-se o nome de Kd-tree, e no nosso caso é uma Kd-tree de duas dimensões, mas ela pode ser utilizada para quantas dimensões forem necessárias. Abaixo apresentamos a Figura 2 que ilustra a construção de uma Kd-tree.

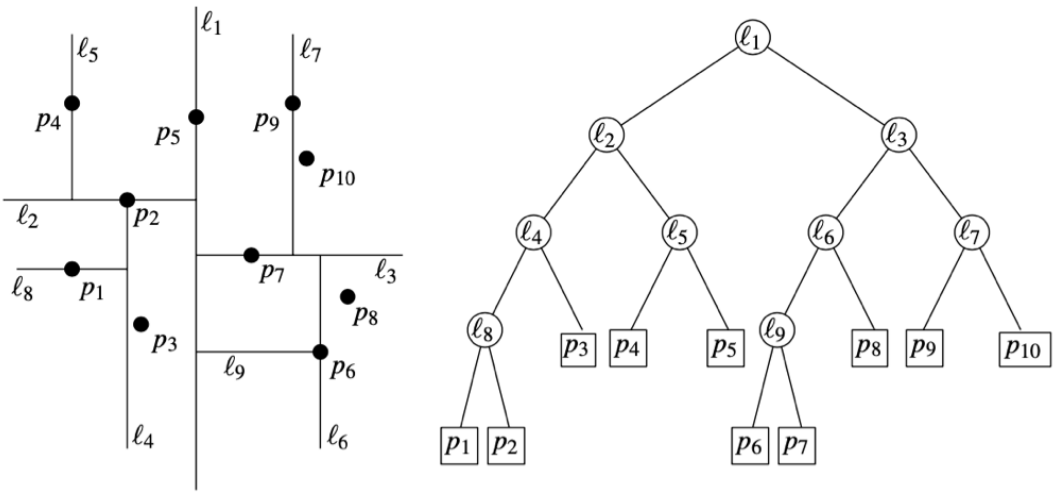

Figura 2: Ilustração da construção de uma Kd-tree. Figura extraída de Berg et al., 2008.

O algoritmo para construir a Kd-tree recebe dois parâmetros, o conjunto de pontos *P* e o nível da recursão "*depth"* que no primeiro caso é zero. O nível da recursão é passado para sabermos em que ponto precisamos fazer um corte utilizando uma linha vertical (em *x*), ou horizontal (em *y*). O algoritmo retorna o nó raiz da Kd-tree.

A complexidade de construção de uma Kd-tree é *O(nlogn)* e utiliza O(n) de espaço. Na Figura 3, mostramos o pseudo-código do algoritmo recursivo utilizado para construir a Kd-tree.

Agora podemos falar sobre a busca ortogonal em uma Kd-tree de duas dimensões. A linha de corte guardada no nó raiz da Kd-tree divide o plano ao meio. Os pontos do lado esquerdo do plano e sobre a linha de corte ficam guardados do lado esquerdo da sub árvore e os pontos a direita do plano ficam na sub árvore da direita. Da mesma forma, quando cortamos o plano no eixo y, os pontos que estão abaixo e sobre a linha pertencem a sub árvore da esquerda, enquanto os pontos acima pertencem a sub árvore da direita.

Algorithm BUILDKDTREE(P, *depth*)
Input. A set of points P and the current depth *depth*.
Output. The root of a kd-tree storing P.
1. **if** P contains only one point
2. **then return** a leaf storing this point
3. **else if** *depth* is even
4. **then** Split P into two subsets with a vertical line ℓ through the median x-coordinate of the points in P. Let P_1 be the set of points to the left of ℓ or on ℓ, and let P_2 be the set of points to the right of ℓ.
5. **else** Split P into two subsets with a horizontal line ℓ through the median y-coordinate of the points in P. Let P_1 be the set of points below ℓ or on ℓ, and let P_2 be the set of points above ℓ.
6. $v_{\text{left}} \leftarrow$ BUILDKDTREE(P_1, *depth* $+ 1$)
7. $v_{\text{right}} \leftarrow$ BUILDKDTREE(P_2, *depth* $+ 1$)
8. Create a node v storing ℓ, make v_{left} the left child of v, and make v_{right} the right child of v.
9. **return** v

Figura 3: Pseudo-código do algoritmo recursivo de construção da Kd-tree. Figura extraída de Berg et al., 2008.

Para fazermos uma busca ortogonal em uma Kd-tree, primeiramente precisamos saber se uma linha de corte armazenada em um nó interno intercepta a query [x, x'] ou [y, y'] de tal forma que se estamos em um nível par da árvore, testamos se a linha intercepta a região [x, x'] e se estivermos em um nível ímpar, testamos se a linha intercepta a região [y, y']. Vale ressaltar que a Kd-tree foi construída de tal forma que um ponto sobre a linha pertence à sub árvore da esquerda. Dito isto, temos três casos para direcionar a busca. No primeiro caso, se a linha de corte armazenada no nó interno for menor que o valor do limite esquerdo da query, no caso x ou y (dependendo do nível que estamos), nossa busca deve ser direcionada toda para a sub árvore da direita, visto que se formos para a sub árvore da esquerda, encontraremos apenas valores (de x ou y de acordo com o nível em que estamos) menores do que os que estamos procurando.

De forma análoga temos o segundo caso, onde, se o valor da linha de corte for maior ou igual ao do limite direito da query, ou seja, x' ou y' (dependendo de que nível estamos), devemos direcionar a busca para a sub árvore da esquerda. Aqui vale ressaltar que se o valor da linha de corte for igual vamos para a esquerda, pois pela construção da Kd-tree, os valores sobre a linha de corte pertencem à sub árvore da esquerda.

O último caso é quando a linha de corte intercepta a nossa query em [x, x'] ou [y, y'] (dependendo do nível em que nos encontramos na árvore). Neste caso, a busca deve ser direcionada tanto para o filho da esquerda quanto para o filho da direita. A única exceção é se a linha de corte for igual ao limite superior da nossa query, x' ou y', a busca vai apenas para a sub árvore da esquerda, como tratado acima. É importante dizer que quando chegamos às folhas devemos avaliar se elas respeitam os limites da query. Caso respeitem, devemos reportar estes pontos.

Um outro caso que deve ser verificado quando estamos fazendo uma busca ortogonal é se uma região está totalmente contida dentro da query. Para sabermos se uma região está totalmente contida, basta olharmos para as linhas de corte armazenadas nos ancestrais do nó atual até quatro gerações. Se elas estiverem contidas no intervalo da query, dizemos que essa região é totalmente contida e desta forma descemos diretamente até as folhas e as reportamos. É importante salientar que só podemos ter uma região contida a partir do quarto nível da árvore, pois antes disso não temos uma região fechada.

Apesar de atravessarmos a Kd-tree durante a busca, visitamos apenas nós que fazem interseção com a query pedida. Na Figura 4, apresentamos o pseudo-código do algoritmo recursivo utilizado para fazer a busca na Kd-tree.

O algoritmo recebe como argumento o nó raiz *v* da Kd-tree e a query *R*. Utilizamos uma subrotina chamada ReportSubtree(v) que atravessa uma sub árvore a partir de um nó v e reporta todos os pontos guardados em suas folhas. Os termos *lc(v)* e *rc(v)* denotam o filho da esquerda e o filho da direita a partir de um nó *v*, respectivamente.

A complexidade de uma busca ortogonal em uma Kd-tree de *n* pontos é $O(\sqrt{n} + k)$, onde *k* é o número de nós reportados.

Algorithm SEARCHKDTREE(v, R)
Input. The root of (a subtree of) a kd-tree, and a range R.
Output. All points at leaves below v that lie in the range.

```
1.  if v is a leaf
2.     then Report the point stored at v if it lies in R.
3.     else  if region(lc(v)) is fully contained in R
4.              then REPORTSUBTREE(lc(v))
5.              else  if region(lc(v)) intersects R
6.                       then SEARCHKDTREE(lc(v),R)
7.           if region(rc(v)) is fully contained in R
8.              then REPORTSUBTREE(rc(v))
9.              else  if region(rc(v)) intersects R
10.                      then SEARCHKDTREE(rc(v),R)
```

Figura 4: Pseudo-código do algoritmo recursivo da busca na Kd-tree. Figura extraída de Berg et al., 2008.

3 – Tecnologias

3.1 – Python

A linguagem de programação escolhida para este projeto foi o Python, versão 3.7, pelo fato de ser uma linguagem de programação de alto nível, interpretada e que permite o paradigma programação orientado a objetos. Ela também já possui todo um ferramental para projetos na área de ciência de dados e uma comunidade que oferece suporte e é altamente ativa,

além de ter farta documentação e ser uma linguagem interpretada, rodando assim em diferentes plataformas.

4 – Documentação

4.1 – Diagrama de Classes

Na Figura 5, temos o diagrama de classes do sistema.

- **KdTree**: essa classe é responsável por criar kd-tree e fazer a busca ortogonal
- **Node**: essa classe é responsável por representar cada nó da kd-tree

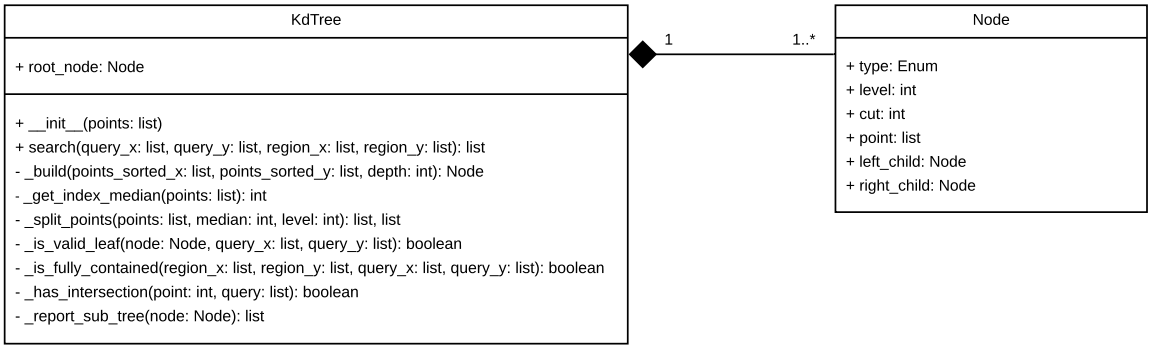

Figura 5: Diagrama de Classes

4.2 – Diagrama de Casos de Uso

Na Figura 6, apresentamos o diagrama de casos de uso do sistema, com o cenário, seus atores e comunicações. Em nosso sistema, possuímos dois atores, (1) Usuário e (2) Sistema.

1. **Usuário**: Esse ator é responsável por interagir com o sistema digitando a query de busca para que o sistema retorne os pontos que se encontram nessa região.
2. **Sistema**: Esse ator é responsável por gerar os pontos aleatoriamente e construir a kd-tree utilizando estes pontos.

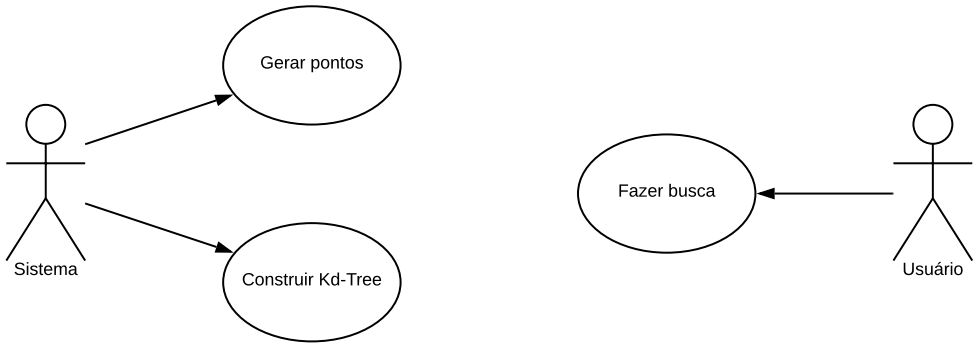

Figura 6: Diagrama de casos de uso

As especificações de caso de uso para os atores Sistema e Usuário estão descritos respectivamente nas Tabelas 1 e 2.

Tabela 1: Especificações de caso de uso para o ator Sistema

Nome:	Gerar pontos
Atores:	Sistema
Objetivo:	Gerar pontos para construção da kd-tree
Fluxo Principal:	1. O sistema gera os pontos

Nome:	Construir kd-tree
Atores:	Sistema
Objetivo:	Construir a kd-tree
Fluxo Principal:	1. O sistema carrega os pontos 2. O sistema constrói a kd-tree recursivamente

Tabela 2: Especificações de caso de uso para o ator Usuário

Nome:	Fazer Busca
Atores:	Usuário
Objetivo:	Fazer uma busca ortogonal e retornar os pontos encontrados
Fluxo Principal:	1. O usuário envia uma query 2. É feita uma busca na kd-tree pelos pontos que tenham alguma interseção com a query 3. Os pontos encontrados são retornados

4.3 – Diagrama de Sequência

As Figuras 7, 8 e 9 apresentam os diagramas de sequência das atividades relatadas no diagrama de casos de uso.

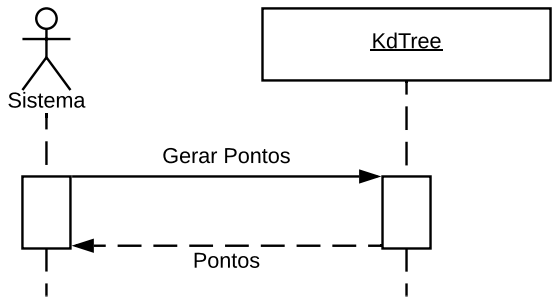

Figura 7: Diagrama de sequência da atividade gerar pontos

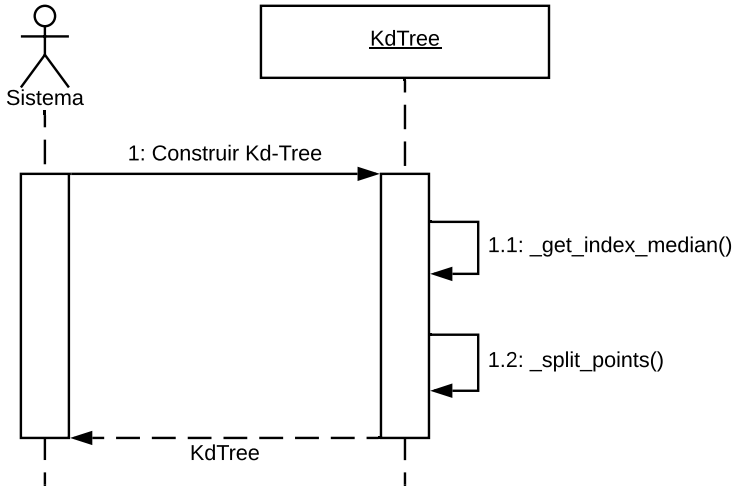

Figura 8: Diagrama de sequência da atividade construir kd-tree

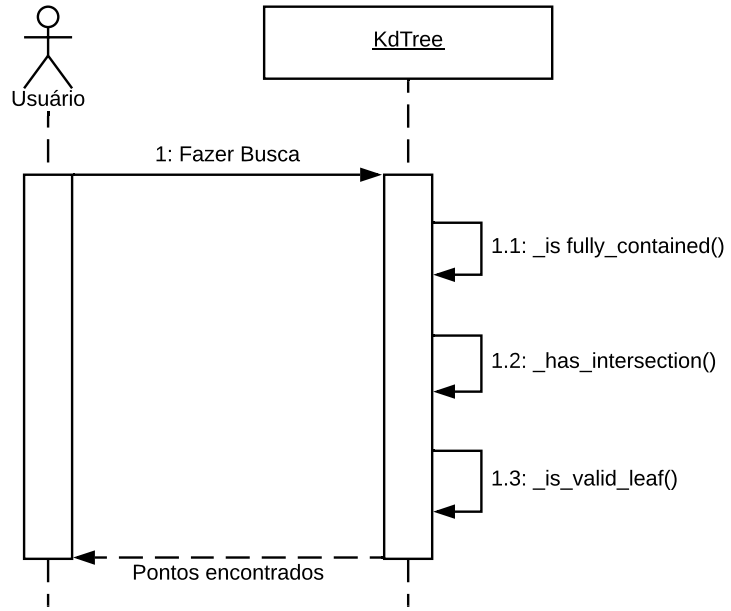

Figura 9: Diagrama de sequência da atividade fazer busca

5 – Teste

Para realizar os testes unitários do sistema, foi utilizado o Unit testing framework[1], que é o framework padrão da linguagem Python para tanto. Com o UnitTest, foram criados 9 casos de teste diferentes. Todas as funções do nosso programa foram testadas. Na figura 10, temos um trecho do código de teste. Para executar o script de teste, basta utilizar o comando *python3 test_kdtree.py*.

[1] https://docs.python.org/3/library/unittest.html

```python
import unittest
from kdtree import KdTree
from node import Node, NodeType


class TestKdTree(unittest.TestCase):

    def setUp(self):

        """This is called immediately before calling every test method"""

        # ODD CASE
        self.points_odd = [[2, 4], [-1, 5], [3, -3], [6, 1], [1, 2]]
        self.points_odd_sorted_x = [[-1, 5], [1, 2], [2, 4], [3, -3], [6, 1]]
        self.points_odd_sorted_y = [[3, -3], [6, 1], [1, 2], [2, 4], [-1, 5]]
        self.kd_tree_odd = KdTree(self.points_odd)

        # EVEN CASE
        self.points_even = [[2, 4], [-1, 5], [3, -3], [1, 2]]
        self.points_even_sorted_x = [[-1, 5], [1, 2], [2, 4], [3, -3]]
        self.points_even_sorted_y = [[3, -3], [1, 2], [2, 4], [-1, 5]]
        self.kd_tree_even = KdTree(self.points_even)

        # QUERY TO SEARCH IN THE KD-TREE
        self.query_x = [1, 3]
        self.query_y = [2, 4]

```

Figura 10: Trecho de código utilizando o framework UnitTest

6 – Resultados

Foi implementada uma função que resolve este mesmo problema utilizando o algoritmo de busca binária. Na Tabela 3, mostramos um comparativo do tempo de execução da busca na kd-tree e da busca, utilizando busca binária.

Tabela 3: Performance da Kd-tree comparada a busca binária.

Nº Pontos	Query (% da área total)	Kd-tree tempo (seg)	Binary Search tempo (seg)
1000	10%	0,0003	0,0020
1000	30%	0,0015	0,0017
10.000	10%	0,0019	0,0292
10.000	30%	0,0091	0,0290
100.000	10%	0,0089	0,2348
100.000	30%	0,0516	0,2617
1.000.000	10%	0,0671	3,3713
1.000.000	30%	1,4547	3,4528

Os intervalos utilizados para 1000 pontos correspondente a 10% de área foram [300,400] [400,500]. Já os referentes a 30% foram [300,600] [400,700]. À Medida que os pontos iam aumentando, multiplicávamos os intervalos por 10 para manter a proporção. Os

pontos foram gerados de modo que sempre fosse um range do tamanho máximo da lista, desta forma não haviam pontos repetidos. Depois de gerados eles eram embaralhados. Os testes foram feitos em um Macbook Pro i5 de 2,3GHz com 8GB de memória.

7 – Conclusão

A busca ortogonal utilizando a estrutura de dados Kd-tree se mostrou muito eficiente, mesmo frente a um algoritmo utilizando busca binária. A Kd-tree não é uma estrutura de dados difícil de ser implementada e tem a vantagem de poder ser elevada a qualquer dimensão.

8 – Bibliografia

De Berg, M., Cheong, O., Van Kreveld, M., & Overmars, M. Computational Geometry: Algorithms and Applications, volume 17. 2008. doi, 10, 3620533.