

Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Mateus Barros Rodrigues

Implementação de algoritmos para consultas de segmentos em janelas

São Paulo
Setembro de 2016

Implementação de algoritmos para consultas de segmentos em janelas

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. Carlos Eduardo Ferreira

São Paulo
Setembro de 2016

Resumo

Este trabalho de conclusão de curso fundamentou-se na compreensão e implementação em linguagem *python* de um algoritmo para consultas de intersecções de segmentos de retas com janelas retangulares no espaço, um subproblema de geometria computacional conhecido por: buscas em regiões ortogonais. Este algoritmo foi o foco da tese de mestrado de Álvaro Junio Pereira Franco. Além da implementação, foi feita também a adaptação do visualizador de algoritmos geométricos feito por Alexis Sakurai Landgraf para exposição dos resultados obtidos.

Palavras-chave: Geometria, janelas, segmentos, buscas.

Sumário

1	Introdução	1
2	Definições e Primitivas	3
2.1	Pontos e segmentos	3
2.2	Comparações entre pontos	3
2.3	Posição relativa entre ponto e segmento	3
3	Consultas sobre pontos em janelas	5
3.1	Janela limitada - Caso unidimensional	5
3.1.1	Pré-processamento	5
3.1.2	Realizando a consulta	6
3.1.3	Análise	8
3.2	Janela limitada - Caso bidimensional	9
3.2.1	Pré-processamento	9
3.2.2	Realizando a consulta	10
3.2.3	Análise	11
3.3	Cascadeamento fracionário	12
3.3.1	Pré-processamento	12
3.3.2	Realizando a consulta	14
3.3.3	Análise	16
3.4	Janelas ilimitadas - caso unidimensional	17
3.4.1	Realizando a consulta	17
3.4.2	Análise	17
3.5	Janelas ilimitadas - caso bidimensional	17
3.5.1	Pré-processamento	18
3.5.2	Realizando a consulta	19
3.5.3	Análise	22
4	Consultas sobre segmentos em janelas	23
4.1	Intervalos na reta	23
4.1.1	Pré-processamento	23
4.1.2	Realizando a consulta	24

4.1.3	Análise	25
4.2	Consultas sobre segmentos horizontais e verticais	26
4.2.1	Pré-processamento	26
4.2.2	Realizando a consulta	27
4.2.3	Análise	28
Referências Bibliográficas		29

Capítulo 1

Introdução

Neste trabalho de conclusão de curso foi abordado o problema de *consultas de segmentos em janelas*, um problema de *buscas em intervalos ortogonais*, que é um dos tópicos fundamentais da área de geometria computacional.

Dado um conjunto S de segmentos no espaço (Seja no \mathbb{R} , \mathbb{R}^2 , etc.) e uma janela W de lados paralelos, queremos responder rapidamente a seguinte pergunta: *quais segmentos de S estão contidos na ou intersectam a janela W ?*

Este trabalho foi baseado em *Consultas de segmentos em janelas: algoritmos e estruturas de dados* de [Álvaro Junio \(2009\)](#), portanto seguiremos a mesma divisão do problema que foi proposta nessa dissertação: Encontrar pontos contidos em janelas e achar todos os segmentos que intersectam com um dado segmento (Horizontal ou vertical). Seguiremos também a mesma divisão de capítulos: Primeiramente apresentaremos definições e primitivas geométricas, dedicaremos um capítulo para falar de consultas de pontos em janelas, um para falar de encontrar intersecção de segmentos e finalmente um onde agregaremos esses algoritmos para resolver o problema proposto. Todo o código desenvolvido foi escrito em linguagem *python* e está disponível no [gitHub](#).

Capítulo 2

Definições e Primitivas

Explicaremos a seguir algumas das noções fundamentais que serão utilizadas ao longo do trabalho:

2.1 Pontos e segmentos

Neste trabalho trataremos basicamente com pontos (no \mathbb{R} e \mathbb{R}^2) e segmentos de reta (restritos ao \mathbb{R}^2). Sejam $x, y \in \mathbb{R}$ definimos um **ponto** no \mathbb{R}^2 como um par $p = (x, y)$. Um **segmento** (definido pelos pontos $u, v \in \mathbb{R}^2$) é o conjunto $\{p \in \mathbb{R}^2 : p = u + t * v \text{ para algum } t \in [0, 1]\} \subseteq \mathbb{R}^2$. Seremos um pouco relaxados quanto a isso e os representaremos como um par de pontos e um reta por cima para dar destaque: $s := \overline{(x_1, y_1)(x_2, y_2)}$, onde $u = (x_1, y_1)$ e $v = (x_2, y_2)$ são pontos chamados de **pontos extremos** de s . Seja p um ponto, diremos que $p \in \overline{p_1, p_2}$ caso p seja uma combinação afim de p_1 e p_2 .

2.2 Comparações entre pontos

Uma outra definição que será usada repetidamente ao longo desta monografia é a relação de desigualdade associada a uma dada coordenada. Sejam u, v pontos, dizemos que $u \leq_x v$ caso $x(u) < x(v)$ ou $x(u) = x(v)$ e $y(u) \leq y(v)$, ou seja, sempre comparamos primeiro a coordenada de maior interesse e desempatamos pela segunda coordenada nas comparações. Quando tivermos pontos ordenados pela ordem \leq_x diremos que estes pontos estão ordenados **sobre a coordenada x** . Todas essas definições são simétricas para a ordem \leq_y .

2.3 Posição relativa entre ponto e segmento

Usaremos também bastante a noção de posição relativa entre pontos e segmentos, isto é, dado um ponto p e um segmento s , queremos saber se p se encontra à esquerda, à direita ou sobre o segmento s .

Sejam $p := (x_1, y_1) \in \mathbb{R}^2$, $s := \overline{(x_2, y_2)(x_3, y_3)}$ e $d := \det \begin{pmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{pmatrix}$

Dizemos que p está **à esquerda** de s caso $d > 0$, que está **sobre** s caso $d = 0$ e que está **à direita** de s caso contrário. Seguem os trechos de código que foram usados no trabalho para realizarmos essas verificações:

Algoritmo 1 Retorna **TRUE** caso p esteja à esquerda de s .

```
1 def left(p,s):
2     b = s.beg
3     c = s.end
4     if b.x == c.x and p.x == b.x: return p.y > c.y
5     if b.y == c.y and p.y == b.y: return p.x < c.x
6     return (b.x-p.x)*(c.y-p.y) - (b.y-p.y)*(c.x-p.x) > 0
```

Algoritmo 2 Retorna **TRUE** caso p esteja à direita de s .

```
1 def right(p,s):
2     b = s.beg
3     c = s.end
4     if b.x == c.x and p.x == b.x: return p.y < b.y
5     if b.y == c.y and p.y == b.y: return p.x > c.x
6     return not(left_on(p,s))
```

Algumas ressalvas sobre essas funções:

- A única diferença da função *left_on* em relação à função *left* é que ela também retorna *true* caso o ponto esteja sobre o segmento dado.
- As modificações presentes nas linhas 4 e 5 foram adicionadas apenas para resolverem os casos degenerados apresentados no capítulo x .

Capítulo 3

Consultas sobre pontos em janelas

Nesse capítulo mostraremos os algoritmos implementados para localizarmos todos os pontos numa dada janela e algumas variações desse problema. Todas as provas de corretude e de eficiência dos algoritmos expostos, tanto deste capítulo quanto dos próximos, poderão ser encontradas na dissertação de [Álvaro Junio \(2009\)](#).

3.1 Janela limitada - Caso unidimensional

Analisaremos primeiramente o problema no espaço \mathbb{R} , ou seja, nossos pontos estarão todos contidos na reta. Sejam u, v pontos na reta tais que $u \leq v$, definimos uma **janela** como sendo um *intervalo fechado* com extremos u e v .

3.1.1 Pré-processamento

Para resolvermos rapidamente sucessivas consultas sobre um dado conjunto de pontos, precisaremos armazenar esses dados em uma estrutura de dados apropriada. A estrutura que usaremos será um tipo de árvore de busca binária balanceada (ABBB) chamada de **árvore limite**, onde cada nó terá 3 campos: um ponteiro para um ponto associado, um ponteiro para o filho esquerdo e um ponteiro para o filho direito. O balanceamento da árvore virá da sua construção, onde subdividimos o vetor de pontos ordenados e colocamos um ponteiro para o ponto central na raiz, conseguimos assim altura $\mathcal{O}(\log n)$. A seguir está o trecho de código referente à construção dessa árvore:

Algoritmo 3 Retorna uma raiz v de uma árvore limite 1D construída sobre um conjunto de pontos ordenados.

```

1 def buildTree(self, points):
2     v = Node(None)
3     l = points[:len(points)//2]
4     r = points[len(points)//2:]
5
6     v.point = points[len(points)//2-1]
7
8     if len(points) == 1:
9         v.l = v.r = None
10    else:
11        v.l = self.buildTree(l)
12        v.r = self.buildTree(r)
13    return v

```

3.1.2 Realizando a consulta

Seja P um conjunto de pontos e seja $W = [w_1, w_2]$ uma janela. Podemos consultar todos os pontos em $P \cap W$ da seguinte forma:

1. Construimos a ABBB sobre o conjunto P .
2. Achamos o **ponto divisor** de P , este é o ponto que se encontra na raiz da subárvore que contém os pontos $S := (v : w_1 \leq v \leq w_2)$, chamaremos esse ponto de v_{div} .
3. Percorremos a subárvore esquerda de v_{div} verificando se o ponto r da raiz é tal que $w_1 \leq r$, caso seja, adicionamos todos os pontos dessa subárvore na resposta e nos movemos para a subárvore esquerda, caso contrário vamos para a subárvore direita. Ao chegar na folha apenas verificamos se $w_1 \leq r \leq w_2$ e adicionamos na resposta caso seja verdade.
4. Percorremos a subárvore direita de v_{div} de forma simétrica ao item 3.

Segue a implementação das rotinas supracitadas juntamente com suas funções auxiliares:

Algoritmo 4 Retorna *true* caso $w_1 \leq p \leq w_2$

```

1 def inRange(self, rng, p):
2     w1, w2 = rng
3     return w1 <= p and p <= w2

```

Algoritmo 5 Retorna o ponto divisor v_{div} de uma ABBB referente a uma dada janela rng .

```

1 def findDividingNode(self, rng):
2     w1, w2 = rng
3     div = self.root
4
5     while(not div.isLeaf() and
6           (w1 > div.point or w2 <= div.point)):
7         if w2 <= div.point:
8             div = div.l
9         else:
10            div = div.r
11    return div

```

Algoritmo 6 Devolve uma lista com as folhas de uma dada árvore.

```

1 def listSubTree(self):
2     l = []
3     self.findLeaves(l)
4     return l
5
6 def findLeaves(self, l):
7     if self.isLeaf():
8         l.append(self.point)
9
10    if self.l is not None: self.l.findLeaves(l)
11    if self.r is not None: self.r.findLeaves(l)

```

Algoritmo 7 Retorna uma lista com todos os pontos contidos numa dada janela rng .

```

1 def query(self, rng):
2     w1, w2 = rng
3     div = self.findDividingNode(rng)
4     p = []
5
6     if div.isLeaf():
7         if self.inRange(rng, div.point):
8             p.append(div.point)
9     else:
10        v = div.l
11        while(not v.isLeaf()):
12            if w1 <= v.point:
13                subtree = v.r.listSubTree()
14                p += subtree
15                v = v.l
16            else:
17                v = v.r

```

Algoritmo 7 Continuação do algoritmo 7.

```

1         if self.inRange(rng, v.point):
2             p.append(v.point)
3
4         v = div.r
5
6         while(not v.isLeaf()):
7             if w2 > v.point:
8                 subtree = v.l.listSubTree()
9                 p += subtree
10                v = v.r
11            else:
12                v = v.l
13        if self.inRange(rng, v.point):
14            p.append(v.point)
15
16    return p

```

3.1.3 Análise

- O pré-processamento requer que seja feita uma ordenação sobre o conjunto de pontos de entrada, portanto tem complexidade $\Theta(n \log n)$.
- A árvore terá altura $\mathcal{O}(\log n)$ e visitaremos $\mathcal{O}(\log n)$ pontos em cada subárvore de v_{div} , além disso, consumiremos tempo $\mathcal{O}(k)$ para visitar os k pontos das folhas que estão contidos no intervalo e devem aparecer na resposta final. Portanto a complexidade final da consulta é da ordem $\mathcal{O}(n \log n + k)$.

3.2 Janela limitada - Caso bidimensional

Analisaremos agora o problema no espaço do \mathbb{R}^2 . Sejam $w_1 = (x_1, y_1)$ e $w_2 = (x_2, y_2)$ pontos no \mathbb{R}^2 , os segmentos de reta que formam um retângulo de lados paralelos ao eixos e que passam pelos pontos w_1 e w_2 são: $s_1 := \overline{(x_1, y_1)(x_1, y_2)}$, $s_2 := \overline{(x_1, y_2)(x_2, y_2)}$, $s_3 := \overline{(x_2, y_2)(x_2, y_1)}$ e $s_4 := \overline{(x_2, y_1)(x_1, y_1)}$, uma **janela** será definida como a união desses 4 segmentos e sua região interna, porém, usaremos uma representação compacta representando a janela pelo segmento $s := \overline{w_1, w_2}$. Mostraremos primeiro o algoritmo mais simples que estende a ideia apresentada no algoritmo anterior e no tópico seguinte uma estrutura de dado diferente que pode ser usada neste algoritmo para diminuir o consumo de tempo.

3.2.1 Pré-processamento

Precisaremos de uma estrutura de dados que consiga particionar o espaço de tal forma que consigamos saber a ordem entre os pontos em cada semiplano. Uma estrutura que nos fornece isso é a chamada **árvore limite de 2 níveis**. A árvore limite é uma ABBB cuja ordem dos elementos é feita sobre a coordenada x e cada nó terá 4 elementos: um ponteiro para uma raiz de uma ABBB cujos elementos são os mesmos da subárvore do nó com elementos ordenados pela coordenada y (que seria o “segundo nível” da árvore), um ponteiro para um ponto associado, um ponteiro para o filho esquerdo e um ponteiro para o filho direito.

Segue o algoritmo de construção dessa árvore. Omitiremos a implementação da estrutura auxiliar que utilizamos nesse trabalho com o nome de *VerticalTree* cuja descrição está presente no trabalho de [Álvaro Junio \(2009\)](#), essa estrutura é uma ABBB construída sobre um *heap* e tem tempo de construção $\mathcal{O}(n)$. Ela será utilizada para fazermos consultas unidimensionais sobre a coordenada y .

Algoritmo 8 Retorna um ponteiro para uma raiz v de uma ABBB ordenada pela coordenada x a partir de um vetor de pontos ordenados por x e um vetor de pontos ordenados por y .

```

1 def buildTree(self, vx, vy):
2     v = Node(None)
3     v.tree = VerticalTree(vy)
4     lx = vx[:len(vx)//2]
5     rx = vx[len(vx)//2:]
6     n = len(vx)
7     ly = []
8     ry = []
9
10    for i in range(n):
11        if vy[i].x < vx[n//2-1].x or
12            (vy[i].x == vx[n//2-1].x and
13             vy[i].y <= vx[n//2-1].y):
14            ly.append(vy[i])
15        else: ry.append(vy[i])

```

Algoritmo 8 Continuação do algoritmo 8.

```

1      v.point = vx[n//2-1]
2
3      if len(vx) == 1:
4          v.l = v.r = None
5      else:
6          v.l = self.buildTree(lx, ly)
7          v.r = self.buildTree(rx, ry)
8
9      return v

```

3.2.2 Realizando a consulta

Seja P um conjunto de pontos e seja $W = \overline{(x_1, y_1)(x_2, y_2)}$ uma janela. Podemos consultar todos os pontos em $P \cap W$ da seguinte forma:

1. Construimos a árvore limite sobre o conjunto P .
2. Achamos o **ponto divisor** no primeiro nível da árvore limite de forma similar ao algoritmo anterior.
3. Percorremos a subárvore esquerda de v_{div} verificando se o ponto r da raiz é tal que $w_1 \leq_x r$, caso seja, realizamos a consulta unidimensional na árvore associada ao nó. Caso contrário, caso contrário vamos para a subárvore direita. Ao chegar na folha apenas verificamos se $w_1 \leq_x r \leq_x w_2$ e adicionamos na resposta caso seja verdade.
4. Percorremos a subárvore direita de v_{div} de forma simétrica ao item 3.

Segue a implementação das rotinas supracitadas juntamente com suas funções auxiliares:

Algoritmo 9 Verifica se o ponto p está contido na janela rng .

```

1 def inRange(self, rng, p):
2     w1, w2 = rng
3     a = w1.x < p.x or (w1.x == p.x and w1.y <= p.y)
4     b = p.x < w2.x or (p.x == w2.x and p.y <= w2.y)
5     c = w1.y < p.y or (w1.y == p.y and w1.x <= p.x)
6     d = p.y < w2.y or (p.y == w2.y and p.x <= w2.x)
7     return a and b and c and d

```

Algoritmo 10 Retorna uma lista com todos os pontos contidos numa dada janela *rng*.

```

1 def query(self, rng):
2     p = []
3     w1, w2 = rng
4     div = self.findDividingNode(rng)
5
6     if div.isLeaf():
7         if self.inRange(rng, div.point):
8             p.append(div.point)
9     else:
10        v = div.l
11
12        while not v.isLeaf():
13            if w1.x < v.point.x or
14                (w1.x == v.point.x and w1.y <= v.point.y):
15                p += v.r.tree.oneDimQuery(rng)
16                v = v.l
17            else:
18                v = v.r
19
20        if self.inRange(rng, v.point): p.append(v.point)
21
22        v = div.r
23
24        while not v.isLeaf():
25            if w2.x > v.point.x:
26                p += v.l.tree.oneDimQuery(rng)
27                v = v.r
28            else:
29                v = v.l
30
31        if self.inRange(rng, v.point): p.append(v.point)
32
33    return p

```

3.2.3 Análise

- No pré-processamento ordenamos 2 vezes o conjunto de pontos, levando tempo $\mathcal{O}(n \log n)$. Como a construção da estrutura auxiliar leva tempo $\mathcal{O}(n)$, a construção da árvore levará também tempo $\mathcal{O}(n)$. O que nos leva à complexidade total de $\mathcal{O}(n \log n)$.
- Os caminhos esquerdo e direito a partir de v_{div} têm $\mathcal{O}(\log n)$ nós, e possivelmente chamamos o algoritmo anterior para cada um deles, o que consome tempo $\mathcal{O}(\log n + k)$. O que nos leva ao consumo total de tempo de $\mathcal{O}(\log^2 n + k)$.

3.3 Cascadeamento fracionário

Apresentaremos uma estrutura chamada **árvore limite com camadas** que utilizaremos no segundo nível do algoritmo acima para conseguirmos complexidade total $\mathcal{O}(\log n + k)$, juntamente com a consulta modificada associada. A intuição dessa técnica vem da seguinte característica das estruturas que vínhamos utilizando: Sempre ao acessarmos o filho de um dado nó passamos a lidar com um subconjunto do conjunto que tínhamos na subárvore anterior e cujos elementos mantêm a mesma ordem relativa entre si.

3.3.1 Pré-processamento

O primeiro nível da árvore limite com camadas será exatamente como mostrado anteriormente, a diferença estará presente no segundo nível onde teremos uma estrutura que definimos como **árvore de camadas**. Os “nós” dessa árvore são na verdade vetores de nós auxiliares ordenados pelos pontos associados.

Seja P o conjunto de pontos associados a um dado vetor da árvore de camadas, sejam V^x e V^y vetores com os pontos de P ordenados por x e y respectivamente, particionamos V^y em 2 vetores: V_e^y e V_d^y . Essa partição é feita da seguinte forma: Seja v_{max} o maior ponto de V^x , seja $q \in V^y$, se $q \leq_x v_{max}$, $q \in V_e^y$, caso contrário $q \in V_d^y$.

Portanto, seja P o conjunto de pontos associado ao vetor, seja $p \in P$ o ponto associado ao nó do vetor V^y , e sejam V_e^y e V_d^y como definidos anteriormente, cada elemento dos nós auxiliares terão os seguintes campos: Um ponteiro para o ponto p , um ponteiro $pt_e(q)$ para o menor ponto q em V_e^y tal que $q \geq_y p$, um ponteiro $pt_d(u)$ para o menor ponto u em V_d^y tal que $u \geq_y p$, uma variável booleana que indica se o vetor ao qual o nó pertence é V_e^y ou V_d^y e finalmente um ponteiro para o próximo elemento do vetor. Esse último ponteiro foi uma adaptação ao fato da linguagem *python* não apresentar aritmética de ponteiros, que foi utilizada na descrição desse algoritmo na dissertação de [Álvaro Junio \(2009\)](#).

Algoritmo 11 Retorna um ponteiro para um vetor ordenado de nós verticais a partir de um vetor de pontos ordenados por x e um vetor de pontos ordenados por y.

```
1 def buildTree(self, vx, vy):
2     v = Node(None)
3     lx = vx[:len(vx)//2]
4     rx = vx[len(vx)//2:]
5     n = len(vx)
6
7     ly = []
8     ry = []
9
10    for i in range(n):
11        if vy[i].point.x < vx[n//2-1].x or
12            ((vy[i].point.x == vx[n//2-1].x) and
13             vy[i].point.y <= vx[n//2-1].y):
14            ly.append(LayerNode(vy[i].point))
15        else:
16            ry.append(LayerNode(vy[i].point))
17
18    v.tree = self.createPointers(vy, ly, ry)
19    v.point = vx[n//2-1]
20
21    if n == 1:
22        v.l = v.r = None
23    else:
24        for k in range(len(ly)-1): ly[k].nxt = ly[k+1]
25
26        for k in range(len(ry)-1): ry[k].nxt = ry[k+1]
27
28        v.l = self.buildTree(lx, ly)
29        v.r = self.buildTree(rx, ry)
30
31    return v
```

Algoritmo 12 Preenche os ponteiros de um vetor v de uma árvore de camadas a partir dos dois subvetores l e r .

```

1 def createPointers(self, v, l, r):
2     il = 0
3     ir = 0
4     i = 0
5     n = len(v)
6     nl = len(l)
7     nr = len(r)
8
9     if n == 1:
10        v[0].pl = v[0].pr = None
11        return v
12
13    while i < n:
14        if il < nl:
15            v[i].pl = l[il]
16            l[il].side = False
17        else:
18            v[i].pl = None
19
20        if ir < nr:
21            v[i].pr = r[ir]
22            r[ir].side = True
23        else:
24            v[i].pr = None
25
26        if il < nl and v[i].point == l[il].point:
27            il += 1
28        else:
29            ir += 1
30
31        i += 1
32
33    return v

```

3.3.2 Realizando a consulta

Seja P um conjunto de pontos e seja $W = \overline{(x_1, y_1)(x_2, y_2)}$ uma janela. Podemos consultar todos os pontos em $P \cap W$ da seguinte forma:

1. Construímos a árvore limite com camadas sobre o conjunto P .
2. Achamos o **ponto divisor** no primeiro nível da árvore limite com camadas de forma similar ao algoritmo anterior.
3. Na árvore de camadas associada ao nó v_{div} procuramos com uma busca binária o menor ponto $v'_{div} : v'_{div} \geq_y w_1$, conseguiremos pontos com a mesma característica nas subárvores de v_{div} em tempo constante apenas utilizando os ponteiros auxiliares.

4. Percorremos a subárvore esquerda de v_{div} , seja v um nó dessa subárvore e v' o nó cujo ponto é o menor tal que $\geq_y w_1$ nessa subárvore. Caso $w_1 >_x p(v)$, continuamos a busca na subárvore direita de v e acessamos o nó apontado por $pt_d(v')$ na árvore de camadas de $d(v)$. Se $w_1 \leq_x p(v)$, listamos todos os pontos $p: p \leq_y w_2$ da árvore de camadas de $d(v)$ a partir do nó apontado por $pt_d(v')$. Retomamos a busca na subárvore esquerda de v e acessamos o nó apontado por $pt_e(v's)$ na árvore de camadas de $e(v)$.
5. Simetricamente ao item 4, percorremos a subárvore direita de v_{div} .

Segue a consulta modificada referente à rotina acima:

Algoritmo 13 Retorna uma lista para todos os pontos contidos numa dada janela rng .

```

1 def query(self, rng):
2     p = []
3     w1, w2 = rng
4     div = self.findDividingNode(rng)
5
6     if div.isLeaf():
7         if self.inRange(rng, div.point):
8             p.append(div.point)
9     else:
10        div2 = self.binarySearch(div.tree, w1) #menor ponto
            em div.tree  $\geq_y$  que w1
11        if div2 is not None:
12            v = div.l
13            v2 = div2.pl
14
15            while not v.isLeaf() and v2 is not None:
16                if w1.x < v.point.x or
17                ( w1.x == v.point.x and w1.y <= v.point.y ):
18                    u = v2.pr
19                    while u and u.side and
20                    (u.point.y < w2.y or
21                    ( u.point.y == w2.y and
22                    u.point.x <= w2.x)):
23                        p.append(u.point)
24                        u = u.nxt
25                        if u is None: break
26
27                    v = v.l
28                    v2 = v2.pl
29            else:
30                v = v.r
31                v2 = v2.pr
32
33            if v2 is not None and self.inRange(rng, v.point):
34                p.append(v.point)

```

Algoritmo 13 Continuação do algoritmo 13.

```

1      if div2 is not None:
2          v = div.r
3          v2 = div2.pr
4
5          while not v.isLeaf() and v2 is not None:
6              if w2.x > v.point.x or
7              (w2.x == v.point.x and
8              w2.y >= v.point.y):
9                  u = v2.pl
10
11                 while not u.side and
12                 (u.point.y < w2.y or
13                 ( u.point.y == w2.y and
14                 u.point.x <= w2.x)):
15                     p.append(u.point)
16                     u = u.nxt
17                     if u is None: break
18
19                 v = v.r
20                 v2 = v2.pr
21             else:
22                 v = v.l
23                 v2 = v2.pl
24
25             if v2 is not None and self.inRange(rng,v.point):
26                 p.append(v.point)
27
28     return p

```

3.3.3 Análise

- Precisamos inicialmente ordenar os pontos, o que leva $\mathcal{O}(n \log n)$. Criamos os ponteiros da árvore de camadas em $\mathcal{O}(n)$, portanto o algoritmo de construção leva $\mathcal{O}(n)$. Chegamos então no consumo total de $\mathcal{O}(n \log n)$.
- Achamos v_{div} e v'_{div} realizando buscas binárias, o que consome tempo $\mathcal{O}(\log n)$. Nas subárvores de v_{div} levamos tempo proporcional ao número de pontos que se encontram na janela, nos dando complexidade $\mathcal{O}(k)$. Portanto a complexidade final de tempo é $\mathcal{O}(\log n + k)$.

3.4 Janelas ilimitadas - caso unidimensional

Seja p um ponto na reta, definiremos uma janela ilimitada W^- como o intervalo $(-\infty : p]$, definimos similarmente uma janela ilimitada W^+ como o intervalo $[p : \infty)$. A implementação desta seção resolve uma consulta sobre todos os pontos contidos numa janela W^- , mas a implementação para uma janela W^+ é simétrica. Omitiremos a explicação da construção da estrutura que utilizaremos, pois trata-se de um *minheap* simples construído sobre o conjunto de pontos.

3.4.1 Realizando a consulta

Seja P um conjunto de pontos e $W^- := (-\infty : w]$ uma janela ilimitada, podemos listar todos os pontos em $P \cap W^-$ da seguinte forma:

- Olhamos para a raiz do heap, caso o ponto associado esteja à esquerda de w , listamos o ponto e repetimos a verificação para seus filhos esquerdo e direito.

Algoritmo 14 Retorna uma lista para todos os pontos em um *minheap* v contidos numa dada janela ilimitada rng .

```

1 def query(self, v, rng):
2     l = []
3     w = rng[1]
4     if p is not None:
5         if v.point <= w:
6             l += v.point
7             l += self.query(v.l, rng)
8             l += self.query(v.r, rng)
9     return l

```

3.4.2 Análise

- Consumimos tempo $\mathcal{O}(1)$ por nó visitado que está na resposta. Portanto, chegamos à complexidade total de $\mathcal{O}(k)$.

3.5 Janelas ilimitadas - caso bidimensional

Sejam $w_1 = (x_1, y_1)$ e $w_2 = (x_2, y_2)$ pontos no \mathbb{R}^2 , similarmente à seção 3.2 iremos definir 4 segmentos de reta que farão parte da janela a ser consultada. Porém agora teremos uma pequena modificação: Seja x_{min} o menor e seja x_{max} o maior valor de x do conjunto de pontos, definimos arbitrariamente que ou $x_1 = x_{min} - 1$ ou $x_2 = x_{max} + 1$. Chamamos a janela construída com $x_1 = x_{min} - 1$ de W^- e a janela construída com $x_2 = x_{max} + 1$ de W^+ . O algoritmo a seguir resolve uma consulta sobre pontos numa janela W^- , mas é simétrico para uma janela W^+ ou mesmo para janelas verticais. Na implementação faremos um certo abuso de linguagem permitido pela linguagem de programação escolhida: definiremos $x_1 = -\infty$ e falaremos que $w_1 = (-\infty, y_1)$ é um ponto.

3.5.1 Pré-processamento

A estrutura de dados que utilizaremos para essa consulta é chamada de **árvore de busca em prioridade**, uma árvore de busca balanceada sobre a coordenada y . Os nós da estrutura terão 4 campos: Um ponteiro para o ponto associado ao nó, um ponteiro para o filho esquerdo, um ponteiro para o filho direito e um ponteiro para um ponto denominado p_{min} , através desse último ponteiro manteremos a propriedade de *minheap*. Essa estrutura será balanceada por construção, pois em cada nó pegamos o x-menor ponto v_{min} do conjunto de pontos, em seguida atribuímos ao nó um ponto v tal que ao retirarmos v , os tamanhos das partições que serão usadas para construção dos filhos difiram em no máximo 1. Uma definição adicional que será usada no algoritmo é: Dado um nó de uma árvore de busca em prioridade, caso o ponteiro para o filho direito desse nó seja nulo e o ponteiro para o filho esquerdo seja não-nulo, chamaremos esse nó de **semi-folha**. Segue o código referente à essa implementação:

Algoritmo 15 Retorna um ponteiro para uma raiz de uma árvore de busca em prioridade a partir de 2 vetores ordenados.

```

1 def buildTree(self, vx, vy):
2     v = minPrioritySearchNode(None)
3     n = len(vx)
4
5     if n > 0:
6         ly = []; lx = []
7         ry = []; rx = []
8         d = 0
9         v.pmin = vx[0]
10
11         for i in range(ceil((n-1)/2)+d):
12             if vy[i] != v.pmin: ly.append(vy[i])
13         else: d+=1
14
15         for i in range(ceil((n-1)/2)+d, n):
16             if vy[i] != v.pmin: ry.append(vy[i])
17
18         if n != 1: v.point = vy[ceil((n-1)/2)+d-1]
19
20         for i in range(1, n):
21             if vx[i].y < v.point.y or
22                (vx[i].y == v.point.y and
23                 (vx[i].x <= v.point.x)):
24                 lx.append(vx[i])
25             else:
26                 rx.append(vx[i])

```

Algoritmo 15 Continuação do algoritmo 15.

```

1
2     v.l = self.buildTree(lx, ly)
3     v.r = self.buildTree(rx, ry)
4     else:
5         v = None
6
7     return v

```

3.5.2 Realizando a consulta

Seja P um conjunto de pontos e W^- uma janela ilimitada, podemos consultar todos os pontos em $P \cap W^-$ da seguinte forma:

- Começamos achando o nó divisor dessa árvore, a estrutura básica é similar à implementação anterior, porém agora verificamos se o nó checado não é um semi-folha e já adicionamos na resposta todos os $pmin$ dos nós acessados que estão dentro da janela na resposta final.
- Percorremos a subárvore esquerda de v_{div} enquanto o nó atual não é uma folha ou semi-folha. Seja v o nó que estamos verificando, se o ponto $pmin(v) <_y w_1$ adicionamos todas os pontos do *minheap* da subárvore direita de v na resposta e seguimos para a subárvore esquerda de v , caso contrário apenas seguimos para a subárvore direita de v .
- Seja u o último nó verificado, caso $pmin(u)$ esteja na resposta adicionamos esse ponto na resposta. Caso u seja uma semi-folha, verificamos se o $pmin(u.l)$ está na janela e o adicionamos na resposta.
- Repetimos o processo simetricamente para a subárvore direita de v_{div} .

Seguem os códigos que explicitam essa rotina:

Algoritmo 16 Retorna uma lista com todos os pontos de um *minheap* v que estão contidos numa dada janela rng .

```

1 def pointsMinHeap(v, rng):
2     p = []
3
4     if v is not None:
5         if self.inRange(rng, v.point):
6             p.append(v.point)
7             p += self.pointsMinHeap(v.l, rng)
8             p += self.pointsMinHeap(v.r, rng)
9     return p

```

Algoritmo 17 Devolve um ponteiro para o nó divisor de uma árvore de busca em prioridade e uma lista com pontos que estão dentro da janela dada *rng*.

```

1 def findDividingNode(self, rng):
2     p = []
3     w1, w2 = rng
4     div = self.root
5
6     while (not div.isLeaf()) and
7           (not div.isSemiLeaf()) and
8           (w1.y > div.point.y and
9            w2.y < div.point.y or
10            (w2.y == div.point.y and
11             (w2.x <= div.point.x))) :
12         if self.inRange(rng, div.pmin):
13             p.append(div.pmin)
14             if w2.y < div.point.y or
15               (w2.y == div.point.y and
16                (w2.x <= div.point.x)):
17                 div = div.l
18             else:
19                 div = div.r
20
21     return p, div

```

Algoritmo 18 .

```

1 def query(self, rng):
2     w1, w2 = rng
3
4     p, div = self.findDividingNode(rng)
5
6     if not div.isLeaf() and not div.isSemiLeaf():
7         if self.inRange(rng, div.pmin):
8             p.append(div.pmin)
9
10        u = div.l
11
12        while not u.isLeaf() and not u.isSemiLeaf():
13            if self.inRange(rng, u.pmin):
14                p.append(u.pmin)
15
16            if w1.y < u.pmin.y or (w1.y == u.pmin.y and
17                                 (w1.x <= u.pmin.x)):
18                p += self.pointsMinHeap(u.r, rng)
19                u = u.l
20            else:
21                u = u.r

```

Algoritmo 18 Continuação do algoritmo 17.

```

1      if self.inRange(rng,u.pmin):
2          p.append(u.pmin)
3
4      if u.isSemiLeaf():
5          if self.inRange(rng,u.l.pmin):
6              p.append(u.l.pmin)
7
8      u = div.r
9
10     while not u.isLeaf() and not u.isSemiLeaf():
11         if self.inRange(rng,u.pmin):
12             p.append(u.pmin)
13
14         if u.pmin.y < w2.y or (u.pmin.y == w2.y and
15             (u.pmin.x <= w2.x)):
16             p += self.pointsMinHeap(u.l,rng)
17             u = u.r
18         else:
19             u = u.l
20
21     if self.inRange(rng,u.pmin):
22         p.append(u.pmin)
23
24     if u.isSemiLeaf():
25         if self.inRange(rng,u.l.pmin):
26             p.append(u.l.pmin)
27
28     else:
29         if self.inRange(rng,div.pmin):
30             p.append(div.pmin)
31
32         if div.isSemiLeaf():
33             if self.inRange(rng,div.l.pmin):
34                 p.append(div.l.pmin)
35
36     return p

```

3.5.3 Análise

- Na construção, começamos ordenando os pontos da entrada, o que consome tempo $\mathcal{O}(n \log n)$. A construção em si é composta por partes $\theta(n)$ junto com duas chamadas recursivas para metade do tamanho, que consumirá por recorrência tempo $\mathcal{O}(n \log n)$. Chegamos portanto em consumo de tempo total de $\mathcal{O}(n \log n)$.
- A consulta seguirá por dois caminhos na árvore de tamanho $\log n$ cada, onde verificar se um dado p_{\min} pertence à janela W^- consome tempo $\mathcal{O}(1)$ e cada chamada de *pointsMinHeap* consome tempo equivalente ao número de pontos da resposta contidos no heap, portando todas as chamadas totalizarão tempo $\mathcal{O}(k)$. Chegamos no total ao consumo de tempo de $\mathcal{O}(\log n + k)$.

Capítulo 4

Consultas sobre segmentos em janelas

Analisaremos nesse capítulo os algoritmos associados às consultas sobre conjuntos de segmentos no espaço.

4.1 Intervalos na reta

Primeiramente explicaremos um algoritmo que resolve consultas no espaço \mathbb{R} . Definiremos a “janela” como um dado ponto no espaço e encontraremos todos os intervalos que contém esse ponto.

Na implementação usaremos intervalos como segmentos de reta, onde seus limites serão dados pelos campos p_e e p_d que denotam o ponto extremo esquerdo e direito do segmento, respectivamente. Diremos que um dado conjunto $S = [s_1, s_2, \dots, s_n]$ de segmentos está p_e -ordenado caso $p_e(s_1) \leq p_e(s_2) \leq \dots \leq p_e(s_n)$.

4.1.1 Pré-processamento

Armazenaremos os intervalos num tipo de árvore binária que chamaremos de **árvore de intervalos**. Cada nó dessa estrutura terá os seguintes campos: Um ponteiro para um ponto associado, um ponteiro para o nó esquerdo, um ponteiro para o nó direito, um ponteiro para um *minheap* de segmentos p_e -ordenados (que chamaremos de L_1) e um ponteiro para um *maxheap* de segmentos p_d -ordenados (que chamaremos de L_2).

Segue o código referente à construção dessa estrutura:

Algoritmo 19 Devolve um ponteiro para uma raiz v de uma árvore de intervalos a partir de um vetor de intervalos ordenado.

```
1 def buildTree(self, s):
2     n = len(s)
3     if n > 0:
4         v = IntervalNode()
5         l = []
6         r = []
7         v.point = s[n//2].beg
8         l1 = []
9         l2 = []
10
11         i = 0
```

Algoritmo 19 Continuação do algoritmo 18.

```

1      while i < n and s[i].beg <= v.point:
2          if s[i].end < v.point:
3              l.append(s[i])
4          else:
5              l1.append(Point(s[i].beg.x,
6                             s[i].beg.y,
7                             s=s[i])
8              l2.append(Point(s[i].end.x,
9                             s[i].end.y,
10                             s=s[i])
11          i+=1
12
13      v.L1 = buildMinHeap(l1)
14      v.L2 = buildMaxHeap(l2)
15
16      while i < n:
17          r.append(s[i])
18          i+=1
19
20      v.l = self.buildTree(l)
21      v.r = self.buildTree(r)
22
23  else:
24      v = None
25
26  return v

```

4.1.2 Realizando a consulta

Dado um ponto w e um conjunto S de segmentos, podemos consultar todos os segmentos de $S' = \{s \in S: s \ni w\}$ da seguinte forma: Verificamos se o ponto associado ao nó está à esquerda do ponto w , caso esteja, adicionamos todos os segmentos que têm $p_e \leq w$, o que é equivalente à fazer uma consulta de janela ilimitada da forma $W^- = (-\infty, w)$. Caso contrário, adicionamos todos os segmentos que têm $p_e \geq w$, o que é equivalente à fazer uma consulta de janela ilimitada da forma $W^+ = (w, \infty)$.

Segue o algoritmo referente a esta rotina:

Algoritmo 20 Devolve uma lista de intervalos que contenham um dado ponto p .

```

1 def query(self, p):
2     return self.query_r(self.root, p)
3
4 def query_r(self, v, p):
5     l = []
6
7     if v is not None:
8         if p > v.point:
9             aux = []
10            rng = (p, Point(math.inf, 0))
11            aux = v.L2.maxheap_query(rng)
12            for pnt in aux:
13                l.append(pnt.seg)
14            l += self.query_r(v.r, p)
15        else:
16            aux = []
17            rng = (Point(-math.inf, 0), p)
18            aux = v.L1.minheap_query(rng)
19            for pnt in aux:
20                l.append(pnt.seg)
21            l += self.query_r(v.l, p)
22
23     return l

```

(As chamadas das linhas 11 e 18 referem-se ao algoritmo descrito na seção 3.4.1)

4.1.3 Análise

- Na construção da árvore, gastamos tempo inicial $\mathcal{O}(n \log n)$ para ordenar o conjunto de segmentos. Separar o conjunto de pontos em dois menores e construir os *heaps* auxiliares leva tempo $\mathcal{O}(n)$. Chegaremos portanto no consumo total de tempo de $\mathcal{O}(n \log n)$.
- Pelo algoritmo de consulta, visitaremos $\mathcal{O}(\log n)$ nós. Em cada nó realizamos algumas operações $\mathcal{O}(1)$, e seja k' o número de pontos do heap que está contido na janela, uma consulta de tempo $\mathcal{O}(k')$ (O somatório de todos esses k 's será k). Chegamos ao consumo total de tempo na consulta de $\mathcal{O}(\log n + k)$.

4.2 Consultas sobre segmentos horizontais e verticais

O tipo de consulta que resolveremos nessa seção é o seguinte: Seja S um conjunto de segmentos horizontais(ou verticais) não-intersectantes, e seja w um segmento vertical(ou horizontal), queremos achar todos os segmentos de S que intersectam w .

4.2.1 Pré-processamento

Utilizaremos uma estrutura que chamaremos de **árvore de intervalos horizontal**. Ela será idêntica à estrutura da seção bla, com modificações nos ponteiros L_1 , que agora aponta para uma árvore de busca em prioridade mínima, e L_2 , que agora aponta para uma árvore de busca em prioridade máxima.

Segue o seu algoritmo de construção:

Algoritmo 21 Devolve um ponteiro v para uma raiz de uma árvore de intervalos horizontal a partir de um vetor p_e -ordenado s .

```

1 def buildTree(self, s):
2     n = len(s)
3
4     if n > 0:
5         v = HorizontalIntervalNode()
6         l = []
7         r = []
8         l1 = []
9         l2 = []
10
11         v.point = s[n//2].beg
12
13         i = 0
14
15         while i < n and s[i].beg <= v.point:
16             if s[i].end < v.point:
17                 l.append(s[i])
18             else:
19                 l1.append(s[i])
20                 l2.append(s[i])
21             i+=1
22
23         while i < n:
24             r.append(s[i])
25             i+=1
26
27         aux = []
28         for s in l1:
29             aux.append(Point(s.beg.x, s.beg.y, s))

```

Algoritmo 21 Continuação do algoritmo 21.

```

1      v.L1 = minPrioritySearchTree(aux)
2
3      aux = []
4
5      for s in l2:
6          aux.append(Point(s.end.x, s.end.y, s))
7
8      v.L2 = maxPrioritySearchTree(aux)
9
10     v.l = self.buildTree(l)
11     v.r = self.buildTree(r)
12
13     else:
14         v = None
15
16     return v

```

4.2.2 Realizando a consulta

Seja S um conjunto de segmentos horizontais não-intersectantes, e seja $w = \overline{(x, y), (x, y')}$ um segmento vertical, podemos encontrar todos os segmentos $S' := \{s \in S: s \cap w \neq \emptyset\}$ da seguinte forma: Seja v o nó que estamos olhando atualmente, caso $x > x(p(v))$ nenhum segmento que esteja armazenado à esquerda de v pode interceptar w , por isso seguiremos para $d(v)$. Mas antes disso fazemos uma consulta por todos os pontos finais de segmentos que se encontram à direita do segmento w , que é equivalente a realizar uma consulta na estrutura L_2 com uma janela $\overline{(x, y), (\infty, y')}$. Caso $x < x(p(v))$, fazemos uma busca em L_1 com janela $\overline{(-\infty, y), (x, y')}$ e seguimos para $e(v)$, simetricamente ao que foi feito no outro caso.

Segue o algoritmo referente a essa rotina:

Algoritmo 22 Retorna uma lista de segmentos horizontais não-intersectantes que intersectam um dado segmento *seg*.

```

1 def query(self, seg):
2     return self.query_r(self.root, seg)
3
4 def query_r(self, v, seg):
5     l = []
6     w1, w2 = seg
7     x = w1.x
8     y = w1.y
9     y2 = w2.y
10
11     if v is not None:
12         if x > v.point.x:
13             rng = (Point(x, y), Point(inf, y2))
14             l = v.L2.query(rng)
15             l += self.query_r(v.r, seg)
16         else:
17             rng = (Point(-inf, y), Point(x, y2))
18             l = v.L1.query(rng)
19             l += self.query_r(v.l, seg)
20
21     return l

```

4.2.3 Análise

- Na construção, inicialmente p_e -ordenamos o vetor de segmentos, consumindo tempo $\mathcal{O}(n \log n)$. Na construção em si, particionamos um vetor em 2 e preenchemos 2 vetores auxiliares, todas operações $\mathcal{O}(n)$. Além disso, construímos 2 árvores de busca em prioridade, que como vimos anteriormente tem complexidade $\mathcal{O}(n_v \log n_v)$, somando-se todas as chamadas que serão feitas à essas funções, teremos também complexidade $\mathcal{O}(n \log n)$. As chamadas para os filhos esquerdo e direito com vetores aproximadamente com a metade de elementos de v resulta num recorrência cuja resolução nos mostra que a complexidade total da construção da árvore será $\mathcal{O}(n \log n)$.
- Visitaremos um nó por nível da árvore, portanto visitaremos $\mathcal{O}(\log n)$ nós. Em cada nó realizamos algumas operações $\mathcal{O}(1)$ e uma busca numa árvore de busca em prioridade, consumindo tempo $\mathcal{O}(\log n' + k') = \mathcal{O}(\log n + k')$, onde n' é o número de elementos armazenados na árvore e k' o número de elementos da árvore que intersectam o segmento w ($\sum k' = k$). Assim, chegamos ao consumo total de tempo de consulta de $\mathcal{O}(\log^2 n + k)$.

Referências Bibliográficas

Álvaro Junio(2009) Álvaro Junio. Consultas de segmentos em janelas: algoritmos e estruturas de dados. Dissertação de Mestrado, Instituto de Matemática e Estatística, Universidade de São Paulo, Brasil. Citado na pág. [1](#), [5](#), [9](#), [12](#)