Universidade de São Paulo
Instituto de Matemática e Estatística
Bachalerado 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 . . . 3

2.1.1 Comparações entre pontos . . . 3

2.2 Segmentos . . . 3

2.2.1 Intervalos . . . 3

2.3 Posição relativa entre ponto e segmento . . . 4

2.3.1 Posição relativa entre segmentos . . . 5

3 Consultas sobre pontos em janelas **7**

3.1 Janela limitada - Caso unidimensional . . . 7

3.1.1 Pré-processamento . . . 7

3.1.2 Realizando a consulta . . . 8

3.1.3 Análise . . . 10

3.2 Janela limitada - Caso bidimensional . . . 10

3.2.1 Pré-processamento . . . 10

3.2.2 Realizando a consulta . . . 11

3.2.3 Análise . . . 13

3.3 Cascateamento fracionário . . . 13

3.3.1 Pré-processamento . . . 13

3.3.2 Realizando a consulta . . . 15

3.3.3 Análise . . . 17

3.4 Janelas ilimitadas - caso unidimensional . . . 18

3.4.1 Realizando a consulta . . . 18

3.4.2 Análise . . . 18

3.5 Janelas ilimitadas - caso bidimensional . . . 18

3.5.1 Pré-processamento . . . 19

3.5.2 Realizando a consulta . . . 20

3.5.3 Análise . . . 23

4 Consultas sobre intersecções de segmentos **25**
4.1 Intervalos na reta 25
4.1.1 Pré-processamento 25
4.1.2 Realizando a consulta 26
4.1.3 Análise 27
4.2 Consultas sobre segmentos horizontais e verticais 28
4.2.1 Pré-processamento 28
4.2.2 Realizando a consulta 29
4.2.3 Análise 30
4.3 Uma outra abordagem para intervalos na reta. 30
4.3.1 Pré-processamento 30
4.3.2 Realizando a consulta 33
4.3.3 Análise 33
4.4 Consultas sobre segmentos com qualquer orientação 34
4.4.1 Pré-processamento 34
4.4.2 Realizando a consulta 34
4.4.3 Análise 35

5 Consultas sobre segmentos em janelas **37**
5.1 Pré-processamento 37
5.2 Realizando a consulta 37
5.3 Análise 39

6 Conclusão **41**

Referências Bibliográficas **43**

Capítulo 1

Introdução

Proveniente da área de análise de algoritmos, a geometria computacional é a área da computação que pode ser definida como o estudo sistemático de algoritmos e estruturas de dados para objetos geométricos, com foco em algoritmos exatos que são assintóticamente rápidos (1). A geometria computacional tem aplicações em diversas áreas como: computação gráfica, reconhecimento de padrões, processamento de imagens, robótica, metalurgia, manufatura e estatística (2). Tais problemas são tratados com o uso de objetos geométricos primitivos como: pontos, retas, segmentos de reta, polígonos.

Para exemplificar uma dessas aplicações: Imagine que temos um banco de dados com diversas informações como: Altura, idade, etc. Podemos resolver perguntas, ou **consultas**, interpretando o problema de forma geométrica (1). Caso queiramos saber todas as pessoas de altura entre $1,50m$ e $1,70m$ de altura que têm entre 15 e 20 anos, podemos representar essas pessoas como pontos no espaço Altura×Idade e a resposta seriam todos os pontos contidos na janela de lados paralelos que queremos (Como pode ser visto na figura a seguir), note que cada característica que adicionemos na busca aumentaria a dimensão do espaço de buscas.

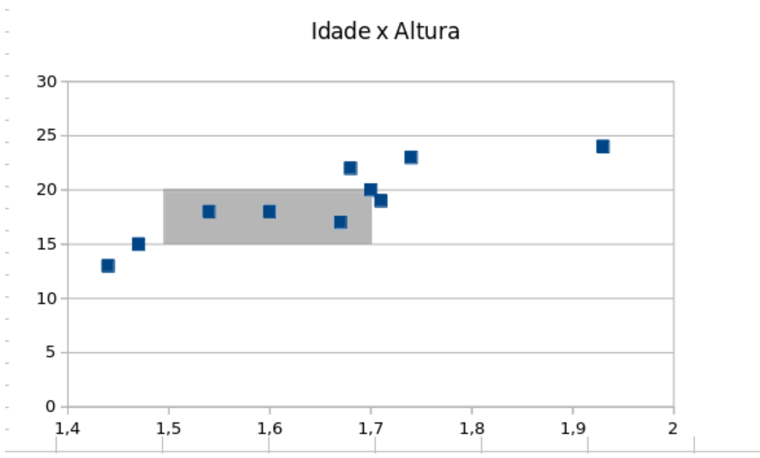

Figura 1.1: *Exemplo de uma consulta num banco de dados.*

A subárea da geometria computacional que trata de problemas como esse é chamada de *buscas em intervalos ortogonais*. Em geral, os algoritmos que veremos dessa área seguirão a mesma estrutura: Temos um conjunto conhecido que é dado, construímos uma estrutura de dados em cima desse conjunto (que de costume é a parte mais custosa computacionalmente) e a partir daí conseguimos responder rapidamente diversas consultas feitas sobre tal conjunto.

Neste trabalho de conclusão de curso foi abordado o problema de *consultas de segmentos em janelas*, um problema de buscas em intervalos ortogonais. Dado um conjunto S de segmentos no espaço (seja em $\mathbb{R}$ ou $\mathbb{R}^2$) queremos organizar os segmentos em estruturas de dados para que possamos responder eficientemente consultas do tipo: *dada uma janela W de lados paralelos, quais segmentos de S estão contidos ou intersectam a janela W?*.

Este trabalho foi baseado em *Consultas de segmentos em janelas: algoritmos e estruturas de dados* de Álvaro J. P. Franco (3), tendo o foco no estudo e implementação dos algoritmos e das estruturas de dados descritas. Ao longo desta monografia tomaremos uma postura mais intuitiva e didática nas descrições dos algoritmos e nas suas respectivas análises de complexidade. Caso o leitor sinta falta de alguma prova formal, estão todas disponíveis em (3).

Seguiremos a mesma divisão do problema que foi proposta na dissertação de Álvaro J. P. Franco: Primeiramente encontraremos pontos contidos em janelas e acharemos 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*. A escolha dessa linguagem se deu pela facilidade de escrita e pela existência de um visualizador de algoritmos geométricos criado por Alexis Sakurai Landgraf. Visto isso, na prática a implementação não foi otimizada para velocidade, apesar de usar os algoritmos mais eficientes dentre os que apresentaremos. Toda a implementação está disponível no gitHub juntamente com a adaptação do visualizador geométrico que foi feita.

Capítulo 2

Definições e Primitivas

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

2.1 Pontos

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 $p = (x, y)$.

2.1.1 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 $\leq_x$ (ou $\geq_x$) diremos que estes pontos estão ordenados **sobre a coordenada *x***. Além disso, seja $P = \{p_1, p_2, \ldots, p_n\}$ tal que $p_1 \leq_x p_2 \leq_x \cdots \leq_x p_n$, chamaremos p_1 de **o x-menor** e p_n de **o x-maior** pontos de P. Todas essas definições são simétricas para a ordem $\leq_y$.

2.2 Segmentos

Um **segmento** (definido pelos pontos $u, v \in \mathbb{R}^2$) é o conjunto $\{p \in \mathbb{R}^2 \colon p = (1 - t) * u + t * v$ 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 uma 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.

2.2.1 Intervalos

Ao longo deste trabalho usaremos segmentos horizontais (ou verticais) para representar intervalos ao longo de uma reta. Seguem as primitivas referentes a intervalos que utilizaremos na última seção do capítulo 4:

Algoritmo 1 Retorna **TRUE** caso $a \cap b \neq \emptyset$ e **FALSE** caso contrário.

```
def intersects(self,a,b):
    if self.contains(a,b) or
    self.belongsTo(a.beg,b) or
    self.belongsTo(a.end,b):
        return True
    else:
        return False
```

Algoritmo 2 Retorna **TRUE** caso $a \subseteq b$ e **FALSE** caso contrário.

```
def contains(self,a,b):
    if a.beg <= b.beg and a.end >= b.end:
        return True
    else:
        return False
```

Algoritmo 3 Retorna **TRUE** caso o ponto a pertença ao intervalo b e **FALSE** caso contrário.

```
 def belongsTo(self,a,b):
     if (b.beg < a and a < b.end) or
     (not b.open and b.beg == a) or
     (not b.open and b.end == a):
        return True
    else:
        return False
```

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 4 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 5 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 na seção 4.4.

2.3.1 Posição relativa entre segmentos

Uma noção que será usada na seção 4.4 é a de esquerda e direita entre segmentos. Sejam u e v segmentos, caso ambos os pontos extremos de u estejam à esquerda de v, ou caso um deles esteja à esquerda de v e o outro esteja sobre v, diremos que u **está à esquerda de** v. Simetricamente, caso ambos seus pontos extremos estejam à direita de v, ou caso um deles esteja à direita e o outro sobre v, diremos que u **está à direita de** v. Além disso, caso ambos os pontos extremos de u estejam em lados opostos em relação a v, diremos que u **pseudo-intercepta** v.

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 **(author?)** (3).

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. Consideramos os pontos ordenados e colocamos na raiz o elemento central, de forma que na subárvore esquerda e direita ficam aproximadamente metade dos elementos. As duas subárvores serão, portanto, construídas da mesma forma, obtendo altura $\mathcal{O}(\log n)$. A seguir está o trecho de código referente à construção dessa árvore:

Algoritmo 6 Retorna uma raíz v de uma árvore limite 1D construída sobre um conjunto de pontos ordenados.

```
def buildTree(self,points):
    v = Node(None)
    l = points[:len(points)//2]
    r = points[len(points)//2:]

    v.point = points[len(points)//2-1]

    if len(points) == 1:
        v.l = v.r = None
    else:
        v.l = self.buildTree(l)
        v.r = self.buildTree(r)
    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. Achamos o **ponto divisor** de P, este é o ponto que se encontra na raiz da subárvore que contém os pontos $S \coloneqq (v : w_1 \leq v \leq w_2)$, chamaremos esse ponto de v_{div}.

2. Percorremos a subárvore esquerda de v_{div}. Tome r como o ponto da raiz desta subárvore. Se $w_1 \leq r$, adicionamos todos os pontos da subárvore direita desta subárvore na resposta, e seguimos para sua subárvore esquerda. Caso contrário, ou seja $w_1 > r$, devemos seguir para a sua subárvore direita.

3. Percorremos a subárvore direita de v_{div} de forma simétrica ao item 2.

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

Algoritmo 7 Retorna *true* caso $w_1 \leq p \leq w_2$

```
def inRange(self,rng,p):
    w1,w2 = rng
    return w1 <= p and p <= w2
```

Algoritmo 8 Retorna o ponto divisor v_{div} de uma ABBB referente a uma dada janela *rng*.

```
def findDividingNode(self,rng):
    w1,w2 = rng
    div = self.root

    while(not div.isLeaf() and
    (w1 > div.point or w2 <= div.point)):
        if w2 <= div.point:
            div = div.l
        else:
            div = div.r
    return div
```

Algoritmo 9 Devolve uma lista com as folhas de uma dada árvore.

```
def listSubTree(self):
    leaves = []
    self.findLeaves(leaves)
    return leaves

def findLeaves(self,lvs):
    if self.isLeaf():
        lvs.append(self.point)

    if self.l is not None: self.l.findLeaves(lvs)
    if self.r is not None: self.r.findLeaves(lvs)
```

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

```
def query(self,rng):
    w1,w2 = rng
    div = self.findDividingNode(rng)
    p = []

    if div.isLeaf():
        if self.inRange(rng,div.point):
            p.append(div.point)
    else:
         v = div.l
         while(not v.isLeaf()):
             if w1 <= v.point:
                 subtree = v.r.listSubTree()
                 p += subtree
                 v = v.l
             else:
                 v = v.r
```

Algoritmo 10 Continuação do algoritmo 7.

```
        if self.inRange(rng,v.point):
            p.append(v.point)

        v = div.r

        while(not v.isLeaf()):
            if w2 > v.point:
                subtree = v.l.listSubTree()
                p += subtree
                v = v.r
            else:
                v = v.l
        if self.inRange(rng,v.point):
            p.append(v.point)

    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. Além disso, consumiremos tempo $\mathcal{O}(k)$ para visitar os k pontos das folhas em cada subàrvore de v_{div} que estão contidos no intervalo e devem aparecer na resposta final. Portanto a complexidade final da consulta é da ordem $\mathcal{O}(\log n + k)$.

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 dados 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 raíz 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 **(author?)** (3), 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 11 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.

```
def buildTree(self,vx,vy):
    v = Node(None)
    v.tree = VerticalTree(vy)
    lx = vx[:len(vx)//2]
    rx = vx[len(vx)//2:]
    n = len(vx)
    ly = []
    ry = []

    for i in range(n):
        if vy[i].x < vx[n//2-1].x or
        (vy[i].x == vx[n//2-1].x and
        vy[i].y <= vx[n//2-1].y):
            ly.append(vy[i])
        else: ry.append(vy[i])

    v.point = vx[n//2-1]

    if len(vx) == 1:
        v.l = v.r = None
    else:
        v.l = self.buildTree(lx,ly)
        v.r = self.buildTree(rx,ry)

    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. Achamos o **ponto divisor** no primeiro nível da árvore limite de forma similar ao algoritmo anterior.

2. 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, seguimos 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.

3. Percorremos a subárvore direita de v_{div} de forma simétrica ao item 2.

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

Algoritmo 12 Verifica se o ponto p está contido na janela *rng*.

```
def inRange(self,rng,p):
    w1,w2 = rng
    a = w1.x < p.x or (w1.x == p.x and w1.y <= p.y)
    b = p.x < w2.x or (p.x == w2.x and p.y <= w2.y)
    c = w1.y < p.y or (w1.y == p.y and w1.x <= p.x)
    d = p.y < w2.y or (p.y == w2.y and p.x <= w2.x)
    return a and b and c and d
```

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

```
def query(self,rng):
    p = []
    w1,w2 = rng
    div = self.findDividingNode(rng)

    if div.isLeaf():
        if self.inRange(rng,div.point):
            p.append(div.point)
    else:
        v = div.l

        while not v.isLeaf():
            if w1.x < v.point.x or
            (w1.x == v.point.x and w1.y <= v.point.y):
                p += v.r.tree.oneDimQuery(rng)
                v = v.l
            else:
                v = v.r

        if self.inRange(rng,v.point): p.append(v.point)

        v = div.r

        while not v.isLeaf():
            if w2.x > v.point.x:
                p += v.l.tree.oneDimQuery(rng)
                v = v.r
            else:
                v = v.l

        if self.inRange(rng,v.point): p.append(v.point)

    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 $\mathcal{O}(n \log n)$.

- Na consulta, 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 Cascateamento 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 $q \leq_y v_{max},\ q \in V_e^y,$ 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 **(author?)** (3).

Algoritmo 14 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.

```
def buildTree(self,vx,vy):
    v = Node(None)
    lx = vx[:len(vx)//2]
    rx = vx[len(vx)//2:]
    n = len(vx)

    ly = []
    ry = []

    for i in range(n):
        if vy[i].point.x < vx[n//2-1].x or
        ((vy[i].point.x == vx[n//2-1].x) and
        vy[i].point.y <= vx[n//2-1].y):
            ly.append(LayerNode(vy[i].point))
        else:
            ry.append(LayerNode(vy[i].point))

    v.tree = self.createPointers(vy,ly,ry)
    v.point = vx[n//2-1]

    if n == 1:
        v.l = v.r = None
    else:
        for k in range(len(ly)-1): ly[k].nxt = ly[k+1]

        for k in range(len(ry)-1): ry[k].nxt = ry[k+1]

        v.l = self.buildTree(lx,ly)
        v.r = self.buildTree(rx,ry)

    return v
```

Algoritmo 15 Preenche os ponteiros de um vetor v de uma árvore de camadas a partir dos dois subvetores l e r.

```
def createPointers(self,v,l,r):
    il = 0
    ir = 0
    i = 0
    n = len(v)
    nl = len(l)
    nr = len(r)

    if n == 1:
        v[0].pl = v[0].pr = None
        return v

    while i < n:
        if il < nl:
            v[i].pl = l[il]
            l[il].side = False
        else:
            v[i].pl = None

        if ir < nr:
            v[i].pr = r[ir]
            r[ir].side = True
        else:
            v[i].pr = None

        if il < nl and v[i].point == l[il].point:
            il += 1
        else:
            ir += 1

        i += 1

    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. Achamos o **ponto divisor** no primeiro nível da árvore limite com camadas de forma similar ao algoritmo anterior.

2. 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.

3. 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 $e(v)$.

4. Simetricamente ao item 3, percorremos a subárvore direita de v_{div}.

Segue a consulta modificada referente à rotina acima:

Algoritmo 16 Retorna uma lista para todos os pontos contidos numa dada janela *rng*.

```
def query(self,rng):
    p = []
    w1,w2 = rng
    div = self.findDividingNode(rng)

    if div.isLeaf():
        if self.inRange(rng,div.point):
            p.append(div.point)
    else:
        div2 = self.binarySearch(div.tree,w1) #menor ponto em div.tree >=_y que w1
        if div2 is not None:
            v = div.l
            v2 = div2.pl

            while not v.isLeaf() and v2 is not None:
                if w1.x < v.point.x or
                ( w1.x == v.point.x and w1.y <= v.point.y ):
                    u = v2.pr
                    while u and u.side and
                    (u.point.y < w2.y or
                    ( u.point.y == w2.y and
                    u.point.x <= w2.x)):
                        p.append(u.point)
                        u = u.nxt
                        if u is None: break

                    v = v.l
                    v2 = v2.pl
                else:
                    v = v.r
                    v2 = v2.pr

            if v2 is not None and self.inRange(rng,v.point):
                p.append(v.point)
```

Algoritmo 16 Continuação do algoritmo 16.

```
        if div2 is not None:
            v = div.r
            v2 = div2.pr

            while not v.isLeaf() and v2 is not None:
                if w2.x > v.point.x or
                (w2.x == v.point.x and
                w2.y >= v.point.y):
                    u = v2.pl

                    while not u.side and
                    (u.point.y < w2.y or
                    ( u.point.y == w2.y and
                    u.point.x <= w2.x)):
                        p.append(u.point)
                        u = u.nxt
                        if u is None: break

                    v = v.r
                    v2 = v2.pr
                else:
                    v = v.l
                    v2 = v2.pl

            if v2 is not None and self.inRange(rng,v.point):
                p.append(v.point)

    return p
```

3.3.3 Análise

- No pré-processamento, 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 $\mathcal{O}(n \log n)$.

- Na consulta, 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 $O(k)$. 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 17 Retorna uma lista para todos os pontos em um *minheap* v contidos numa dada janela ilimitada *rng*.

```
def query(self, v, rng):
    l = []
    if p is not None:
        w = rng[1]
        if v.point <= w:
            l += v.point
            l += self.query(v.l,rng)
            l += self.query(v.r,rng)
    return l
```

3.4.2 Análise

- Consumimos tempo $\mathcal{O}(1)$ por nó visitado e só continuamos a fazer chamadas da função quando $p \leq w_1$. Portanto, teremos feito $\mathcal{O}(k)$ chamadas paras os nós na resposta e $\mathcal{O}(k)$ chamadas para os nós que não estão na resposta, isto é, $p > w_1$. Logo, a complexidade total será $\mathcal{O}(k)$.

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 um 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 a essa implementação:

Algoritmo 18 Retorna um ponteiro para uma raiz de uma árvore de busca em prioridade a partir de 2 vetores ordenados.

```
def buildTree(self,vx,vy):
    v = minPrioritySearchNode(None)
    n = len(vx)

    if n > 0:
        ly = []; lx = []
        ry = []; rx = []
        d = 0
        v.pmin = vx[0]

        for i in range(ceil((n-1)/2)+d):
            if vy[i] != v.pmin: ly.append(vy[i])
        else: d+=1

        for i in range(ceil((n-1)/2)+d,n):
            if vy[i] != v.pmin: ry.append(vy[i])

        if n != 1: v.point = vy[ceil((n-1)/2)+d-1]

        for i in range(1,n):
            if vx[i].y < v.point.y or
            (vx[i].y == v.point.y and
            (vx[i].x <= v.point.x)):
                lx.append(vx[i])
            else:
                rx.append(vx[i])
```

Algoritmo 18 Continuação do algoritmo 18.

```

            v.l = self.buildTree(lx,ly)
            v.r = self.buildTree(rx,ry)
        else:
            v = None

        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:

1. 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.

2. 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.

3. 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.

4. Repetimos o processo simetricamente para a subárvore direita de v_{div}.

Seguem os códigos que explicitam essa rotina:

Algoritmo 19 Retorna uma lista com todos os pontos de um *minheap* v que estão contidos numa dada janela *rng*.

```
def pointsMinHeap(v,rng):
    p = []

    if v is not None:
        if self.inRange(rng,v.point):
            p.append(v.point)
            p += self.pointsMinHeap(v.l,rng)
            p += self.pointsMinHeap(v.r,rng)
    return p
```

Algoritmo 20 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*.

```
def findDividingNode(self,rng):
    p = []
    w1,w2 = rng
    div = self.root

    while (not div.isLeaf()) and
    (not div.isSemiLeaf()) and
    (w1.y > div.point.y and
     w2.y < div.point.y or
     (w2.y == div.point.y and
      ( w2.x <= div.point.x))) :
        if self.inRange(rng,div.pmin):
            p.append(div.pmin)
        if w2.y < div.point.y or
        (w2.y == div.point.y and
        (w2.x <= div.point.x)):
            div = div.l
        else:
            div = div.r

    return p, div
```

Algoritmo 21 Devolve uma lista de pontos contidos numa janela infinita *rng*.

```
def query(self,rng):
    w1,w2 = rng

    p,div = self.findDividingNode(rng)

    if not div.isLeaf() and not div.isSemiLeaf():
        if self.inRange(rng,div.pmin):
            p.append(div.pmin)

        u = div.l

        while not u.isLeaf() and not u.isSemiLeaf():
            if self.inRange(rng,u.pmin):
                p.append(u.pmin)

            if w1.y < u.pmin.y or ( w1.y == u.pmin.y and
            (w1.x <= u.pmin.x)):
                p += self.pointsMinHeap(u.r,rng)
                u = u.l
            else:
                u = u.r
```

Algoritmo 21 Continuação do algoritmo 21.

```
        if self.inRange(rng,u.pmin):
            p.append(u.pmin)

        if u.isSemiLeaf():
            if self.inRange(rng,u.l.pmin):
                p.append(u.l.pmin)

        u = div.r

        while not u.isLeaf() and not u.isSemiLeaf():
            if self.inRange(rng,u.pmin):
                p.append(u.pmin)

            if u.pmin.y < w2.y or (u.pmin.y == w2.y and
            (u.pmin.x <= w2.x)):
                p += self.pointsMinHeap(u.l,rng)
                u = u.r
            else:
                u = u.l

        if self.inRange(rng,u.pmin):
            p.append(u.pmin)

        if u.isSemiLeaf():
            if self.inRange(rng,u.l.pmin):
                p.append(u.l.pmin)

    else:
        if self.inRange(rng,div.pmin):
            p.append(div.pmin)

        if div.isSemiLeaf():
            if self.inRange(rng,div.l.pmin):
                p.append(div.l.pmin)

    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 pmin 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 intersecções de segmentos

Analisaremos nesse capítulo os algoritmos relacionados com achar intersecções de um dado segmento com um conjunto de segmentos no espaço, esses algoritmos serão posteriormente usados nas chamadas do algoritmo da seção 5.

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 contêm esse ponto. Veremos uma outra forma de resolver esse tipo de consulta numa futura seção usando uma ideia que será estendida para consultas sobre segmentos.

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 $p_e(s_1) \leq p_e(s_2) \leq \cdots \leq p_e(s_n)$.

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

```
def buildTree(self,s):
    n = len(s)
    if n > 0:
        v = IntervalNode()
        l = []
        r = []
        v.point = s[n//2].beg
        l1 = []
        l2 = []
        i = 0
        while i < n and s[i].beg <= v.point:
            if s[i].end < v.point:
                l.append(s[i])
            else:
                l1.append(Point(s[i].beg.x,
                                s[i].beg.y,
                                s=s[i]))
                l2.append(Point(s[i].end.x,
                                s[i].end.y,
                                s=s[i]))
            i+=1

        v.L1 = buildMinHeap(l1)
        v.L2 = buildMaxHeap(l2)

        while i < n:
            r.append(s[i])
            i+=1

        v.l = self.buildTree(l)
        v.r = self.buildTree(r)

    else:
        v = None

    return v
```

4.1.2 Realizando a consulta

Dado um ponto w e um conjunto S de segmentos, podemos consultar todos os segmentos de $S' = \{s \in S \colon s \ni w\}$ 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 23 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 $O(\log n)$ 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 $O(k')$ $(\sum k' = k)$. 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 4.1, 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 24 Devolve um ponteiro v para uma raiz de uma árvore de intervalos horizontal a partir de um vetor p_e-ordenado s.

```
def buildTree(self,s):
    n = len(s)

    if n > 0:
        v = HorizontalIntervalNode()
        l = []
        r = []
        l1 = []
        l2 = []

        v.point = s[n//2].beg

        i = 0

        while i < n and s[i].beg <= v.point:
            if s[i].end < v.point:
                l.append(s[i])
            else:
                l1.append(s[i])
                l2.append(s[i])
            i+=1

        while i < n:
            r.append(s[i])
            i+=1

        aux = []
        for s in l1:
            aux.append(Point(s.beg.x,s.beg.y,s))
```

Algoritmo 24 Continuação do algoritmo 24.

```
            v.L1 = minPrioritySearchTree(aux)

            aux = []

            for s in l2:
                aux.append(Point(s.end.x,s.end.y,s))

            v.L2 = maxPrioritySearchTree(aux)

            v.l = self.buildTree(l)
            v.r = self.buildTree(r)

        else:
            v = None

        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 $S' := \{s \in S \colon s \cap w \neq 0\}$ 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 25 Retorna um lista de segmentos horizontais não-intersectantes que intersectam um dado segmento *seg*.

```
def query(self,seg):
    return self.query_r(self.root,seg)

def query_r(self,v,seg):
    l = []
    w1,w2 = seg
    x = w1.x
    y = w1.y
    y2 = w2.y
```

Algoritmo 25 Continuação do algoritmo 25.

```
    if v is not None:
        if x > v.point.x:
            rng = (Point(x,y),Point(inf,y2))
            l = v.L2.query(rng)
            l += self.query_r(v.r,seg)
        else:
            rng = (Point(-inf,y),Point(x,y2))
            l = v.L1.query(rng)
            l += self.query_r(v.l,seg)

    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 a 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 $\mathcal{O}(\log^2 n + k)$.

4.3 Uma outra abordagem para intervalos na reta.

Resolveremos agora o mesmo problema apresentado na seção 4.1 utilizando uma nova estrutura de dados. Definiremos primeiro uma noção que será utilizada nessa estrutura: Seja $S := \{s_1, s_2, \ldots, s_n\}$ um conjunto de intervalos (ou segmentos) e seja $P := \{p_1, p_2, \ldots, p_{2n}\}$ o conjunto de pontos extremos de S onde $p_1 \leq p_2 \leq \cdots \leq p_{2n}$, dizemos que o conjunto $E := \{(-\infty; p_1), [p_1; p_1], (p_1; p_2), [p_2; p_2], \ldots, [p_n; p_n], (p_n; +\infty)\}$ é o conjunto de **intervalos elementares** sobre o conjunto S. Note que esse conjunto tem tamanho no máximo $4n + 1$, caso todos os pontos extremos de S sejam distintos.

4.3.1 Pré-processamento

A estrutura que iremos utilizar é chamada **árvore de segmentos**. Cada nó dessa árvore terá os seguintes campos: um intervalo associado, uma lista de segmentos, um ponteiro para o filho esquerdo e um ponteiro para o filho direito desse nó. Seja v um nó da árvore de segmentos e seja $int(v)$ o intervalo associado ao nó v, esse intervalo terá a seguinte forma: Caso v seja uma folha, $int(v)$ será um intervalo elementar, caso contrário, $int(v)$ será a união dos intervalos dos seus filhos esquerdo e direito. A construção dessa estrutura se dará em 3 partes:

1. Seja S um conjunto de intervalos, obtemos o conjunto E de intervalos elementares sobre esse conjunto.

2. Construímos uma árvore binária de baixo para cima (similar a um *heap*), colocando os intervalos elementares nas folhas e fazendo as uniões à medida que subimos na árvore.

3. Seja v um nó da árvore de segmentos e seja $s \in S$. Se $int(v) \subset s$, inserimos s na lista de v. E repetimos para seus filhos caso o intervalo associado a eles intersectem s. Note que a raiz terá, por construção, intervalo $(-\infty; +\infty)$ e portanto, terá sempre sua lista vazia.

Seguem os algoritmos descritos acima:

Algoritmo 26 Devolve uma lista de intervalos elementares q construída sobre o conjunto v.

```
def buildElementaryIntervals(self,v):
    p = []
    q = []
    n = len(v)

    for i in range(n):
        p.append(v[i].beg)
        p.append(v[i].end)

    sort(p)

    m = self.removeDuplicates(p)

    l = Point(-inf,0)

    for i in range(len(m)):
        r = p[i]
        q.append(Segment(l,r,True))
        q.append(Segment(r,r))
        l = r

    r = Point(inf,0)

    q.append(Segment(l,r,True))

    return q
```

Algoritmo 27 Devolve uma árvore binária T construída a partir do conjunto de intervalos elementares e.

```
def buildTree(self,e):
    m = len(e)
    h = ceil(log(m,2))
    l2 = 2**h - m
    l = m - l2
    i = 2*m - 2
    T = []
    for k in range(2*m-1):
        T.append(0)
        T[k] = SegmentTreeNode()

    for j in range(l-1,-1,-1):
        T[i].interval = e[j]
        T[i].leaf = True
        i -= 1

    for j in range(m-1,-1,-1):
        T[i].interval = e[j]
        T[i].leaf = True
        i -= 1

    while i >= 0:
        T[i].interval.beg = T[2*i+1].interval.beg
        T[i].interval.end = T[2*i+2].interval.end
        i -= 1

    return T
```

Algoritmo 28 Insere um dado intervalo s no nó v de uma árvore de segmentos.

```
def insertInterval(self,v,s):
    u = v

    if self.contains(s,u.interval):
        u.L.append(s)
    else:
        if self.intersects(s,u.l.interval):
            self.insertInterval(u.l,s)

        if self.intersects(s,u.r.interval):
            self.insertInterval(u.r,s)
```

Algoritmo 29 Devolve um ponteiro T para uma árvore de segmentos construída sobre a lista de segmentos v.

```
def buildSegmentTree(self,v):
    v2 = self.buildElementaryIntervals(v)
    T = self.buildTree(v2)

    for i in range(len(v)):
        self.insertInterval(T,v[i])

    return T
```

4.3.2 Realizando a consulta

Seja p um ponto e seja S um conjunto de intervalos, podemos encontrar todos os intervalos de $S' := \{i \in S : i \ni p\}$ da seguinte forma: Começamos a chamada na raiz e adicionamos sua lista na resposta (que por construção será vazia), verificamos então se seus filhos esquerdo e direito contém o ponto p, caso afirmativo, chamamos recursivamente para esses nós.

Segue o algoritmo que foi descrito acima:

Algoritmo 30 Devolve uma lista l de segmentos que contêm um dado ponto p.

```
def query(self,p):
    return self.query_r(self.root,p)

def query_r(self,v,p):
    u = v

    l = u.L

    if not u.isLeaf():
        if self.belongsTo(p,u.l.interval):
            l += self.query_r(u.l,p)
            return l
        else:
            l += self.query_r(u.r,p)
            return l

    return l
```

4.3.3 Análise

- Inicialmente construímos os intervalos elementares, que requer uma ordenação sobre o conjunto de pontos extremos, levando portanto tempo $\mathcal{O}(n \log n)$. Construir a árvore de baixo para cima e inserir os intervalos de S na árvore levam ambos tempo proporcional ao número de intervalos elementares, isto é, tempo $\mathcal{O}(n)$. Assim, chegamos que o consumo de tempo total da construção dessa estrutura é $\mathcal{O}(n \log n)$.

- Por construção, teremos que a árvore terá altura $\mathcal{O}(\log n)$. Em cada nível da árvore levamos tempo $\mathcal{O}(k')$ ($\sum k' = k$) para adicionar todos os segmentos da lista associada ao nó na resposta e tempo constante nas demais operações. Portanto, o consumo de tempo total da consulta é $\mathcal{O}(\log n + k)$.

4.4 Consultas sobre segmentos com qualquer orientação

Discutiremos agora como estender o problema apresentado na seção 4.2. Agora nosso conjunto S conterá segmentos com qualquer orientação, porém ainda não-intersectantes e nossa "janela" será um segmento vertical (ou horizontal).

4.4.1 Pré-processamento

Utilizaremos novamente uma árvore de segmentos como na seção anterior, com a alteração que a lista associada a cada nó é agora um vetor ordenado. Chamaremos essa árvore de **árvore de segmentos 2D horizontal** quando utilizada para responder uma consulta sobre um segmento vertical (Definição simétrica para segmentos horizontais). A relação de ordem que usaremos é: Sejam u e v segmentos, diremos que $u < v$ caso u esteja à direita de v, ou caso u pseudo-intercepte v e v esteja à esquerda de u.

Segue a implementação dessa árvore:

Algoritmo 31 Devolve um ponteiro para uma raiz v de uma árvore de segmentos 2D.

```
1  def buildSegmentTree(self,v):
2      v2 = self.buildElementaryIntervals(v)
3      t = self.buildTree(v2)
4
5      for i in range(len(v)):
6          self.insertInterval(t,v[i])
7
8      self.sortLists(t)
9
10     return t
```

(As chamadas nas linhas 2,3 e 6 são as mesmas descritas na seção 4.3 e a chamada da linha 8 ordena as listas de todos os nós da árvore, seguindo a relação de ordem descrita acima.)

4.4.2 Realizando a consulta

Seja S um conjunto de segmentos não-intersectantes e seja w um segmento vertical. Podemos consultar todos os segmentos de S que intersectam w da seguinte forma:

Seja v um nó da árvore e seja L_{ord} o vetor ordenado de v. Achamos o menor índice j tal que $p_e(L_{ord}[j])$ está à esquerda do ponto extremo esquerdo de w. A partir de $L_{ord}[j]$, adicionamos todos os segmentos s_i de L_{ord} tais que $p_d(w)$ esteja à esquerda de s_i ou que $s_i \ni p_d(w)$. Verificamos então se $p_e(w)$ está contido no intervalo associado ao filho esquerdo de v, caso afirmativo chamamos a função para seu filho esquerdo, caso contrário chamamos a função para seu filho direito.

Inicialmente a implementação continha um caso patológico: Caso existisse algum $u \in S$ tal que $u \subseteq w$ ou $w \subseteq u$, esse elemento não seria incluso na consulta, pois pela definição de posição relativa entre pontos e segmentos, os pontos extremos de u não estariam nem à esquerda ou à direita de w. Esse problema foi resolvido estendendo o conceito de esquerda e direita, e pode ser visto na seção 2.3.

Segue a implementação da rotina descrita acima:

Algoritmo 32 Devolve uma lista l de todos os segmentos que interceptam um dado segmento s.

```
1 def query(self,s):
2     return self.query_r(self.root,s)
3
4 def query_r(self,v,s):
5     u = v
6     l = []
7     j = self.binarySearch(s.beg,u.L)
8
9     while j < len(u.L) and (left(s.end,u.L[j]) or
10     self.inside(s.end,u.L[j])):
11         l.append(u.L[j])
12         j += 1
13
14     x = s.beg
15     if not u.isLeaf():
16         if belongsTo(x,u.l.interval):
17             l += self.query_r(u.l,s)
18             return l
19         else:
20             l += self.query_r(u.r,s)
21             return l
22
23     return l
```

4.4.3 Análise

- O único trecho novo que precisamos analisar na construção é a chamada da linha 8. Sejam v_i, com $i = \{1, \ldots, k\}$, $k \leq 2n$, nós da árvore. Para cada v_i realizamos uma ordenação que terá complexidade $\mathcal{O}(n_i \log n_i)$, onde n_i é o número de segmentos armazenados em v_i. Teremos que a ordenação de um dado nível da árvore consumirá tempo $\sum n_i \log n_i \leq 2n \log n = \mathcal{O}(n \log n)$, como temos $\mathcal{O}(\log n)$ níveis, chegamos à complexidade total de $\mathcal{O}(n \log^2 n)$.

- Visitamos $\mathcal{O}(\log n)$ nós da árvore na consulta. Em cada nó i realizamos uma busca binária que consome tempo $\mathcal{O}(\log n_i) = \mathcal{O}(\log n)$ e percorremos $O(k_i)$ elementos de $L_{ord}(i)$, onde n_i é o número de elementos em $L_{ord}(i)$ e k_i o número de elementos de $L_{ord}(i)$ que está na resposta. Teremos entao complexidade total na consulta da ordem de $\mathcal{O}(\log^2 n + k)$ (Pois $\sum k_i = k$).

Capítulo 5

Consultas sobre segmentos em janelas

Nesse capítulo iremos finalmente resolver o problema inicialmente proposto: Dado um conjunto de segmentos não-intersectantes S, quais segmentos de S estão contidos numa certa janela de lados paralelos W? As estruturas que utilizaremos são as versões mais eficientes de todos os algoritmos que apresentamos até este ponto.

5.1 Pré-processamento

Utilizaremos 4 árvores construídas a partir do conjunto S: Duas árvores limite com camadas, uma sobre o conjunto de pontos esquerdos de S e outra sobre o conjunto de pontos direitos de S, e duas árvores de segmentos 2D, uma horizontal e a outra vertical.

Segue o trecho de código referente às chamadas das construções dessas estruturas:

Algoritmo 33 Constroi as 4 estruturas auxiliares a serem utilizadas na consulta a partir de uma lista de segmentos s.

```
1 def __init__(self,s):
2     self.l = []
3     self.r = []
4     for seg in s:
5         self.l.append(Point(seg.beg.x,seg.beg.y,seg))
6         self.r.append(Point(seg.end.x,seg.end.y,seg))
7
8     self.layer_l = LayerTree(self.l)
9     self.layer_r = LayerTree(self.r)
10
11    self.seg_v = SegmentTree2Dx(s)
12    self.seg_h = SegmentTree2Dy(s)
```

(As chamadas nas linhas 8 e 9 referem-se à estrutura descrita na seção 3.3.1, e as chamadas das linhas 11 e 12 à estrutura descrita na seção 4.4.1.)

5.2 Realizando a consulta

Seja $W = \overline{(x_1, y_1), (x_2, y_2)}$ com lados: $w_1 = \overline{(x_1, y_1)(x_1, y_2)}$, $w_2 = \overline{(x_1, y_2)(x_2, y_2)}$, $w_3 = \overline{(x_2, y_2)(x_2, y_1)}$ e $w_4 = \overline{(x_2, y_1)(x_1, y_1)}$. A consulta será divida em 5 subconsultas:

1. Encontramos todos os $s \in S$ tais que $p_e(s) \in W$.
2. Encontramos todos os $s \in S$ tais que $p_d(s) \in W$.
3. Encontramos todos os $s \in S$ que interceptam w_1.
4. Encontramos todos os $s \in S$ que interceptam w_3.
5. Encontramos todos os $s \in S$ que interceptam w_2.

Perceba que poderão haver repetições entre essas consultas, portanto apenas retiramos os segmentos repetidos no final. Segue a implementação referente à rotina descrita:

Algoritmo 34 Constroi as 4 estruturas auxiliares a serem utilizadas na consulta a partir de uma lista de segmentos s.

```
1 def query(self,rng):
2     L = self.layer_l.query(rng)
3     R = self.layer_r.query(rng)
4     l = []
5     r = []
6
7     for p in L: l.append(p.seg)
8     for p in R: r.append(p.seg)
9
10    a = rng[0].x
11    b = rng[0].y
12    c = rng[1].x
13    d = rng[1].y
14
15    seg1 = Segment(Point(a,b),Point(a,d))
16    seg2 = Segment(Point(a,d),Point(c,d))
17    seg3 = Segment(Point(c,b),Point(c,d))
18    seg4 = Segment(Point(a,b),Point(c,b))
19
20    s1 = self.seg_v.query(seg1)
21    s3 = self.seg_v.query(seg3)
22    s2 = self.seg_h.query(seg2)
23
24    return list(set(l+r+s1+s2+s3))
```

(As chamadas nas linhas 2 e 3 referem-se ao algoritmo descrito na seção 3.3.2, e as chamadas das linhas 20, 21 e 22 referem-se ao algoritmo descrito na seção 4.4.2.)

5.3 Análise

- Na construção, preenchemos listas com os pontos extremos dos segmentos consumindo tempo $\mathcal{O}(n)$ e realizamos ordenações consumindo tempo $\mathcal{O}(n \log n)$. Além disso, construímos 2 árvores limite com camadas consumindo tempo $\mathcal{O}(n \log n)$ e 2 árvores de segmentos 2D, consumido tempo $\mathcal{O}(n \log^2 n)$. Portanto, o consumo total será da ordem de $\mathcal{O}(n \log^2 n)$.

- Na consulta, separamos os elementos de s em duas listas com seus pontos extremos, consumindo tempo $\mathcal{O}(n)$. Realizamos uma consulta em árvore limite com camadas que levará tempo $\mathcal{O}(\log n + k_e)$ e outra que levará tempo $\mathcal{O}(\log n + k_d)$, onde k_e e k_d são o número de segmentos de s que satisfez cada uma dessas consultas. Além disso realizamos 3 consultas em árvores de segmentos 2D, consumindo tempos $\mathcal{O}(\log^2 n + k_{w_1})$, $\mathcal{O}(\log^2 n + k_{w_2})$ e $\mathcal{O}(\log^2 n + k_{w_3})$ respectivamente. Teremos que $k_e + k_d + k_{w_1} + k_{w_2} + k_{w_3} \leq 5k = \mathcal{O}(k)$. Portanto, o consumo de tempo total será $\mathcal{O}(\log^2 n + k)$.

Capítulo 6

Conclusão

Neste trabalho foi feito

Referências Bibliográficas

[1] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*. Springer, 3 ed., 2008. 1

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 3 ed., 2009. 1

[3] Álvaro J. P. Franco, "Consultas de segmentos em janelas: algoritmos e estruturas de dados," Master's thesis, Instituto de Matemática e Estatística, Universidade de São Paulo, Brasil, Aug. 2009. 2, 7, 11, 13