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

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 J. P. Franco (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 $\{p \in \mathbb{R}^2 \colon p = u + t * v$ 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 um reta por cima para dar destaque: $s \coloneqq \overline{(x_1, y_1)(x_2, y_2)}$, onde $u = (x_1, y_1)$ e $v = (x_2, y_2)$ são pontos chamados de **pontos extremos** de s.

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 $\boldsymbol{x}$**. 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 $p \coloneqq (x_1, y_1) \in \mathbb{R}^2$, $s \coloneqq \overline{(x_2, y_2)(x_3, y_3)}$ e $d \coloneqq \det \begin{pmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{pmatrix}$

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 J. P. Franco (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 raíz, conseguimos assim altura $\mathcal{O}(\log n)$. A seguir está o trecho de código referente à construção dessa árvore:

Algoritmo 3 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. Construímos 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 $w_1 \leq p \leq w_2$

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

Algoritmo 5 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 6 Devolve uma lista com as folhas de uma dada árvore.

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

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

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

Algoritmo 7 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

         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 em cada subàrvore 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 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 Álvaro J. P. Franco (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.

```
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. Construímos 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*.

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

- 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^y_e e V^y_d. Essa partição é feita da seguinte forma: Seja v_{max} o maior ponto de V^x, seja $q \in V^y$, se $q \leq_x v_{max},\ q \in V^y_e,$ caso contrário $q \in V^y_d$.

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^y_e e V^y_d 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^y_e tal que $q \geq_y p$, um ponteiro $pt_d(u)$ para o menor ponto u em V^y_d tal que $u \geq_y p$, uma variável booleana que indica se o vetor ao qual o nó pertence é V^y_e ou V^y_d 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 J. P. Franco (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.

```
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 12 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. 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*.

```
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 13 Continuação do algoritmo 13.

```
        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

- 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 $O(k)$. 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 raíz 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*.

```
def query(self, v, rng):
    l = []
    w = rng[1]
    if p is not None:
        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 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: Sejam x_{min} o menor e 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.

Capítulo 4

Consultas sobre segmentos em janelas

Referências Bibliográficas

Álvaro J. P. Franco(2009) Álvaro J. P. Franco. 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