Consultas de segmentos em janelas: algoritmos e estruturas de dados

Álvaro Junio Pereira Franco

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título de mestre
em
Ciências

Programa: Ciência da Computação
Orientador: Prof. Dr. Carlos Eduardo Ferreira

Durante o desenvolvimento deste trabalho o autor recebeu auxílio financeiro do CNPq

São Paulo, agosto de 2009

Consultas de segmentos em janelas: algoritmos e estruturas de dados

Este exemplar corresponde à redação
final da dissertação defendida
por Álvaro Junio Pereira Franco
e aprovada pela Comissão Julgadora.

São Paulo, agosto de 2009

Banca Examinadora:

- Prof. Dr. Carlos Eduardo Ferreira (Orientador) IME-USP
- Prof. Dr. José Coelho de Pina Junior IME-USP
- Prof. Dr. Pedro Jussieu de Rezende IC-UNICAMP
- Profa. Dra. Cristina Gomes Fernandes IME-USP (Suplente)
- Prof. Dr. Jorge Stolfi IC-UNICAMP (Suplente)
- Prof. Dr. Luiz Henrique de Figueiredo IMPA (Suplente)

Agradecimentos

Gostaria de agradecer a Deus, à minha família e aos meus amigos e professores. Em particular, agradeço ao Instituto de Matemática e Estatística da Universidade de São Paulo pela oportunidade, ao COSEAS pelo auxílio moradia, ao CNPq pelo apoio financeiro, aos meus pais Álvaro Campos e Suely pelo apoio nos momentos difíceis, aos meus irmãos Rakel e Ricardo pelo incentivo, à minha namorada Priscila pelo carinho, aos meus amigos Alexandre, Wanderley, Carlos, Anderson, Márcio, Lobato e José Francisco pelos momentos de diversão e estudo, aos professores (e amigos) Coelho, Cris, Yoshiko, Alair, Mandel, José Augusto, Yoshiharu, Leliane e Eloi pelo aprendizado em conversas, seminários e aulas, ao professor Luiz Henrique pela instância do mapa dos municípios do Brasil, muito importante para este trabalho, e em especial aos professores Nami e Carlinhos que aceitaram a dura tarefa de me orientar.

Resumo

Neste trabalho estudamos problemas relacionados com a busca de pontos e segmentos em janelas retangulares com os lados paralelos aos eixos. É dado um conjunto de segmentos (ou pontos) no plano. Em uma primeira fase estes segmentos são organizados em estruturas de dados de tal forma a tornar buscas por aqueles que estão contidos em janelas retangulares mais eficiente. Na segunda fase são dadas as janelas de maneira *online*. Várias destas estruturas de dados são baseadas em árvores balanceadas, tais como, árvore limite, árvore de busca com prioridade, árvore de intervalos e árvore de segmentos. Na dissertação mostramos detalhadamente estas estruturas de dados e os algoritmos para resolver este problema para conjuntos de pontos (versão unidimensional do problema) e para segmentos no plano, tanto horizontais e verticais como com qualquer orientação (sem cruzamentos). Os algoritmos são analisados de forma rigorosa quanto ao seu uso de espaço e de tempo. Implementamos também os vários algoritmos estudados, construindo uma biblioteca destas estruturas de dados. Apresentamos, finalmente os resultados de experimentos computacionais com instâncias do problema.

Abstract

In this work we study problems about point and segment query in rectangular windows whose edges are parallel to the axis. Given a set of segments (or points) in the plane. In a first phase these segments are organized in data structures such that queries for segments in windows are done more efficiently. In the second phase windows are given online. The data structures are balanced trees as range tree, priority search tree, interval tree and segment tree. In this master's thesis we show in details data structures and algorithms for solving windowing queries to sets of points (unidimensional version of the problem) and of segments in the plane, as horizontal and vertical as any orientation (without crossings). The algorithms are analysed rigorously regarding their space and time used. We implement the algorithms studied, building a library of these data structures. Finally, we present, the results of computational experiments with instances of the problem.

Sumário

Introdução

Projetistas e analistas de algoritmos querem desenvolver algoritmos eficientes. Neste trabalho descrevemos algoritmos eficientes encontrados na literatura que listam pontos e segmentos que estão em janelas. Em alguns casos podemos comparar a eficiência de algoritmos que resolvem um mesmo problema. Por exemplo, no Capítulo 2 descrevemos dois algoritmos rápidos que listam pontos em janelas no plano, porém, um consome tempo aproximadamente igual a $c_1(\log^2 n + k)$ enquanto que o outro consome tempo aproximadamente igual a $c_2(\log n + k)$, onde n é o número de pontos de um conjunto, k é o número de pontos listados e c_1 e c_2 são constantes independentes do valor de n. Note que um algoritmo possui um fator $\log n$ a mais que o outro e isso faz com que um seja mais rápido que o outro (quando o valor de n é muito grande, o algoritmo que consome tempo aproximadamente $c_2(\log n + k)$ é mais rápido que o que consome $c_1(\log^2 n + k)$, mesmo se $c_1 < c_2$). Outra preocupação de projetistas e analistas é a quantidade de espaço consumida por estruturas de dados. Geralmente, o uso de estruturas de dados acelera o consumo de tempo de um algoritmo, porém o consumo de espaço cresce. Um desafio para projetistas e analistas de algoritmos é encontrar um "ponto de equilíbrio" entre os dois consumos.

Emergente da área de análise de algoritmos [8], a disciplina geometria computacional trata da complexidade de problemas geométricos com aplicações em: computação gráfica, reconhecimento de padrões, processamento de imagens, pesquisa operacional, estatística, robótica, entre outros [17]. Este texto se concentra no projeto e na análise de alguns algoritmos e nas estruturas de dados por eles usadas para resolver um problema específico de geometria computacional classificado como um problema de busca geométrico. Desejamos localizar segmentos e pontos de uma dada coleção de segmentos e pontos. No decorrer do texto, descrevemos tais algoritmos, assim como algumas estruturas de dados que contribuem nos seus desempenhos, verificando o consumo de espaço de cada estrutura estudada. Problemas clássicos como busca por pontos em uma região retangular no plano, busca por pontos em um intervalo na reta, busca por intervalos que contém um ponto, busca por segmentos que atravessam uma reta vertical ou horizontal são alguns problemas que estudamos neste trabalho. A solução deles contribui na solução de um problema maior que chamamos de consultas em janelas. Começamos a falar sobre este problema descrevendo duas aplicações.

Nos tempos de hoje existem programas de computador que ajudam pessoas a se localizar. Esses programas são chamados de sistemas de navegação [8]. Geralmente estes sistemas armazenam um mapa de uma determinada região e mostram em uma tela a localização de

um ponto assim como uma parte do mapa próxima a este ponto. Um mapa de uma grande cidade como São Paulo possui uma grande quantidade de dados. Porém, uma pequena porção destes dados são mostradas ao usuário. Como podemos obter de forma rápida somente os dados que serão mostrados ao usuário? Uma estratégia ingênua é verificar, a cada consulta, todos os segmentos e pontos do mapa e mostrar aqueles que estão próximos do ponto. O que pretendemos fazer neste trabalho é apresentar algumas estruturas de dados que ajudam a responder à consulta mais eficientemente.

Outra aplicação do problema que tratamos surge no projeto de circuitos integrados. Em placas de circuitos impressos encontramos algumas dificuldades em identificar qual trecho do circuito se liga com um determinado componente. Muitas vezes, um trecho fica muito próximo de outro dificultando a identificação das ligações [8]. O mesmo ocorre com alguns componentes eletrônicos que com o avanço da nanotecnologia ficaram menores. Um programa de computador que mostrasse em uma tela os componentes e os trechos de um circuito de uma placa em uma escala maior seria útil.

Figura 1: Um sistema de navegação e um circuito impresso.

Note que tanto na primeira aplicação (sistema de navegação) como na segunda (aumento da escala em circuitos impressos) temos que mostrar em uma tela alguns objetos que pertencem a um conjunto, digamos, S. Em verdade, os objetos que gostaríamos de mostrar pertencem a um subconjunto de S que se encontra na região de interesse do usuário. Mas o problema é que S pode ser muito grande e a implementação de um programa que usa estruturas de dados robustas se faz necessária para obtermos este subconjunto de forma rápida.

Resumindo, temos o seguinte problema. Seja S um conjunto de pontos e segmentos. Repetidas consultas são feitas sobre S. Cada consulta é constituída por uma região retangular com os lados paralelos aos eixos do plano. Chamamos essa região de janela. Desejamos responder a cada consulta quais elementos de S interceptam sua determinada janela. É importante perceber que uma consulta sobre S deve ser respondida rapidamente. Para isso,

armazenamos os elementos de S em estruturas de dados que garantem uma busca rápida. É claro que gastamos tempo para organizar os elementos de S nessas estruturas, porém tal organização é realizada somente uma vez enquanto que podem ocorrer várias consultas. Com isso, passa a ser vantajoso organizar os elementos de S em uma fase de pré-processamento dos dados, para alcançar uma melhor eficiência na fase de consultas. Preparata e Shamos [22] classificaram a fase de consultas como consultas modo-repetitivo (ou *online*).

As estruturas de dados que vamos estudar são árvores limite, árvores de busca com prioridade, árvores de intervalos e árvores de segmentos. Um bom motivo para o uso de árvores é que podemos realizar buscas eficientemente nestas estruturas caso as árvores estejam balanceadas. As árvores estudadas neste trabalho são binárias e suas construções, sobre um conjunto com n elementos, garantem que tenham altura $O(\log n)$. Os algoritmos que constroem cada uma dessas estruturas e que realizam buscas em janelas foram implementados e com eles construímos uma biblioteca que pode ser usada para resolver rapidamente buscas por segmentos ou pontos em janelas.

No Capítulo 1 definimos os objetos geométricos considerados neste trabalho, falamos sobre nosso problema de interesse, sobre algumas técnicas algorítmicas e sobre algumas estruturas de dados fundamentais. No Capítulo 2 resolvemos o problema busca por pontos em janelas em uma e duas dimensões. Na versão unidimensional, buscar pontos em janelas é equivalente a buscar pontos em intervalos na reta. Na versão bidimensional, buscar pontos em janelas é buscar pontos em regiões retangulares no plano com lados paralelos aos eixos. No Capítulo 3 resolvemos o problema de busca por intervalos em janelas (unidimensional) e de busca por segmentos em janelas (bidimensional). No Capítulo 4 resolvemos consultas sobre conjuntos de segmentos usando as estruturas de dados e os algoritmos vistos nos Capítulos 2 e 3 e apresentamos resultados de alguns experimentos computacionais com instâncias do problema. E no final deste texto descrevemos nossas considerações finais listando alguns trabalhos que dão continuidade a este e citando alguns problemas em aberto.

Capítulo 1

Preliminares

Os algoritmos que vamos estudar nos capítulos seguintes usam técnicas como ordenação, recursão, divisão-e-conquista, entre outras e algumas estruturas de dados fundamentais como vetores, listas ligadas, árvores de busca binária e *heaps*. Por isso, neste capítulo falamos brevemente sobre essas técnicas e estruturas. É também neste capítulo que introduzimos o assunto estudado nesta dissertação quando falamos sobre uma estrutura geométrica clássica chamada *kd-tree*.

Começamos este capítulo introduzindo os objetos geométricos usados neste trabalho: pontos, intervalos e segmentos; e o problema de nosso interesse com algumas ilustrações que ajudam a entendê-lo.

1.1 Objetos geométricos

Os objetos geométricos usados neste trabalho são pontos e intervalos na reta e pontos e segmentos no plano. Um ponto na reta possui um valor x. Um ponto no plano possui dois valores, um para cada eixo do plano, denotado por $p = (x, y)$. Um intervalo int é definido sobre a reta, é formado por dois valores x e y e é denotado por $int = [x : y]$. Um segmento no plano s é formado por dois pontos no plano (chamados também de pontos extremos de s) $p_1 = (x_1, y_1)$ e $p_2 = (x_2, y_2)$ e é denotado por $s = \overline{(x_1, y_1)(x_2, y_2)}$. O ponto extremo esquerdo de s é aquele que possui menor x coordenada ou, em caso de empate, o que possui menor y coordenada. Denotamos esse ponto por $p_e(s)$. O outro ponto é o ponto extremo direito de s, denotado por $p_d(s)$.

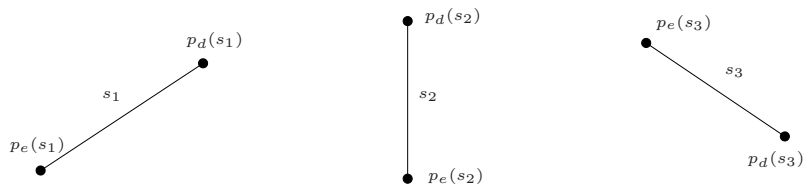


Figura 1.1: Pontos extremos esquerdo e direito de alguns segmentos.

Uma janela na reta é um intervalo e no plano é uma região retangular com os lados paralelos aos eixos. Uma janela W é formada por dois pontos, w_1 e w_2. Muitas vezes escrevemos $W = (w_1, w_2)$. O ponto w_1 é o menor ponto de uma janela e w_2 é o maior ponto. Para evitar problemas relacionados à precisão numérica assumimos que os valores dos pontos, intervalos, segmentos e janelas são inteiros.

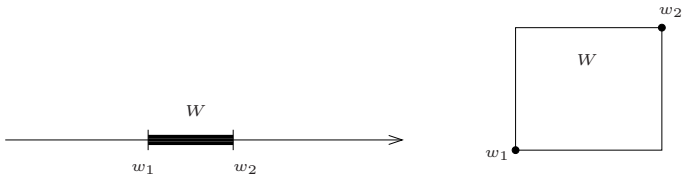


Figura 1.2: Uma janela na reta e no plano.

1.2 Problema

O problema se resume em organizar os elementos de um conjunto que podem ser pontos ou segmentos em estruturas de dados e resolver repetidas consultas sobre este conjunto. Em geral, denotamos o conjunto ou por S quando é um conjunto de segmentos ou por P quando é um conjunto de pontos. Por enquanto vamos denotar por S. A organização dos elementos de S é feita em uma fase chamada de pré-processamento. É nesta fase que criamos estruturas de dados que organizam os elementos de S. Depois disso, repetidas consultas sobre S são resolvidas. Uma consulta sobre S é formada por uma janela W no plano denotada por W-consulta sobre S. A solução de tal consulta é obtida quando listamos todos os elementos de S que interceptam W. Queremos consumir pouco tempo para realizar essa organização e para resolver cada consulta. As figuras 1.3, 1.4 e 1.5 exemplificam o nosso problema e suas respostas.

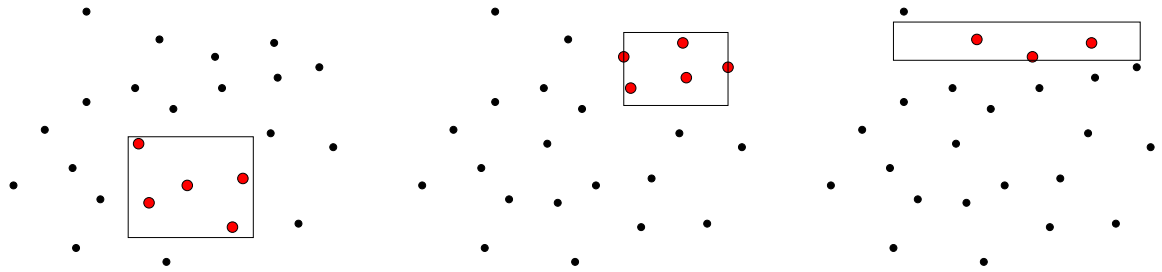


Figura 1.3: Um conjunto de pontos e repetidas janelas.

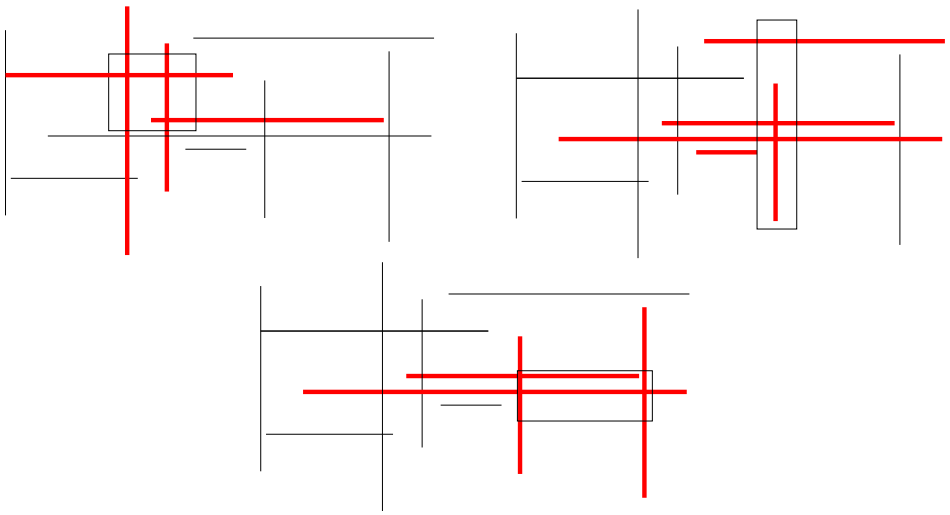


Figura 1.4: Um conjunto de segmentos horizontais e verticais e repetidas janelas.

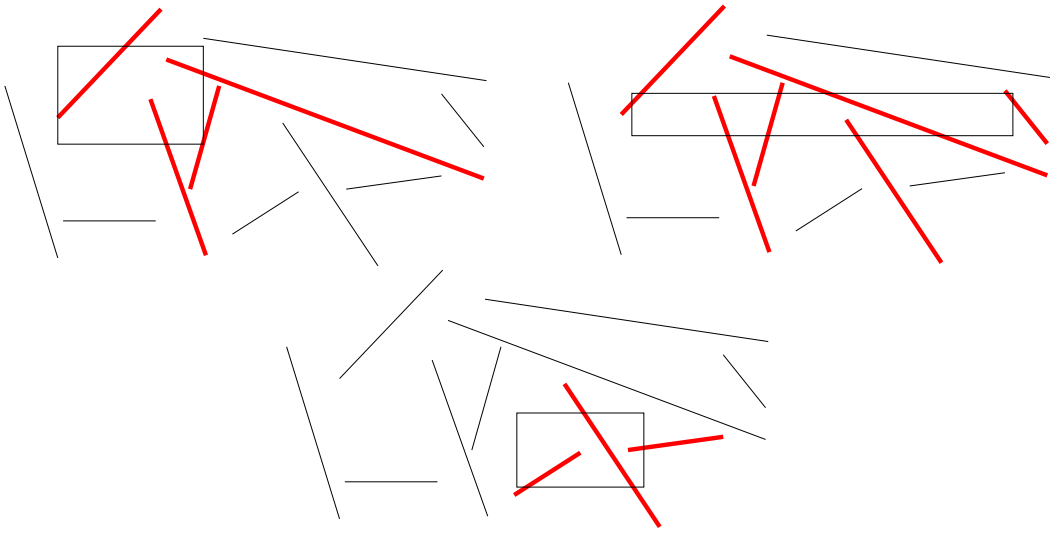


Figura 1.5: Um conjunto de segmentos sem cruzamentos e repetidas janelas.

1.3 Técnicas algorítmicas

Técnicas algorítmicas como ordenação, enumeração, recursão, divisão-e-conquista, pesquisa em árvores de busca binária e busca binária são usadas nos algoritmos descritos neste trabalho. Por exemplo, na construção das estruturas de dados estudadas são usadas as técnicas de ordenação, recursão e divisão-e-conquista. Normalmente construímos um nó de uma es-

trutura sobre um dado conjunto ordenado de pontos ou segmentos S, particionamos S e construímos os outros nós desta estrutura sobre estas partições, recursivamente. O particionamento de S fica fácil quando S está ordenado. Sempre particionamos um conjunto com relação à x ou y-coordenada de seus elemetos. Um x-particionamento de S é feito dividindo os elementos que estão à esquerda (ou sobre) e à direita de uma reta vertical que passa pela x-mediana em S. De forma simétrica, um y-particionamento de S é feito dividindo os elementos que estão abaixo (ou sobre) e acima de uma reta horizontal que passa pela y-mediana em S. Na listagem dos elementos que estão em uma janela usamos pesquisa em árvores de busca binária quando visitamos alguns nós de uma árvore de busca binária, enumeração quando visitamos todos os nós e listamos todas as folhas de uma árvore de busca binária e busca binária quando queremos buscar um elemento em um vetor ordenado. No decorrer deste texto e nos livros [16] e [6] encontramos o uso de cada uma dessas técnicas.

1.4 Estruturas de dados fundamentais

Consideramos neste texto que cada estrutura é composta por nós. Um vetor é composto por uma sequência consecutiva de nós, uma lista ligada é composta por uma sequência (não necessariamente consecutiva) de nós, uma árvore é composta por nós e assim por diante. Um nó é composto por membros e possui pelo menos o membro chave.

1.4.1 Vetores e listas ligadas

Um vetor é a estrutura de dados mais simples usada neste trabalho. Ele é formado por uma sequência de nós, em posições consecutivas da memória, acessados diretamente por um índice. O primeiro nó de um vetor V com n nós é obtido usando a notação $V[0]$ e o último usando $V[n-1]$. Com isso, o seu i-ésimo nó está armazenado em $V[i-1]$. Em alguns momentos, mantemos um vetor ordenado. Por simplicidade, considere um conjunto S formado por pontos distintos e $p_1 < p_2 < \ldots < p_n$ uma ordenação deles. Um vetor que armazena os elementos de S ordenados é chamado de vetor ordenado e é denotado por $V = [p_1, p_2, \ldots, p_n]$, onde p_1 está em $V[0]$, p_2 está em $V[1]$ e assim por diante.

Uma lista ligada, como em um vetor, é formada por uma sequência de nós, porém, o acesso aos nós não é feito diretamente por um índice. É natural acessar um elemento em um nó de uma lista percorrendo-a. Uma lista pode ser simplesmente ligada (e neste caso é chamada de lista ligada) ou duplamente ligada. Um nó de uma lista ligada tem um apontador para o próximo nó da lista. Já um nó de uma lista duplamente ligada tem dois apontadores, um para o próximo nó e um para o anterior. O tamanho de uma lista pode aumentar ou diminuir, adicionando ou eliminando um nó a ela. Isso é essencial quando precisamos de uma estrutura cujo tamanho pode variar. Em uma lista circular duplamente ligada, o apontador para o nó anterior do primeiro nó, aponta para o último nó da lista e o apontador para o próximo nó do último nó, aponta para o primeiro nó. Para simplificar as operações realizadas sobre uma lista (para mais detalhes veja [6]), criamos um nó chamado cabeça, ou como chamam em [6], sentinela.

1.4.2 Árvores de busca binária

Um nó típico de uma árvore binária possui como membros uma chave e dois apontadores, cada um apontando para uma árvore binária que chamamos de subárvore à sua esquerda e subárvore à sua direita. Uma árvore binária é definida recursivamente como: um nó vazio, ou um nó com uma chave, um apontador para uma árvore binária à sua esquerda e um apontador para uma árvore binária à sua direita [30]. Dizemos que o primeiro nó de uma árvore binária à esquerda de um nó v é o filho à esquerda de v. O primeiro nó de uma árvore binária à direita de um nó v é o filho à direita de v. Os filhos à esquerda e à direita de um nó v são denotados por $e(v)$ e $d(v)$, respectivamente. Se um nó u é filho de um nó v, então v é pai de u. Cada nó de uma árvore pode ser classificado como interno ou folha. Um nó interno possui pelo menos um filho. Uma folha não tem filhos. O nó que não é filho de qualquer outro nó é chamado de raiz. Vamos denotar por $p(v)$ a chave de um nó v. Uma árvore binária torna-se de busca quando vale a seguinte propriedade: para todo nó v, $p(u) < p(v) < p(w)$, onde u é qualquer nó na subárvore com raiz em $e(v)$ e w é qualquer nó na subárvore com raiz em $d(v)$ [30]. Neste trabalho, permitimos que existam nós com a mesma chave. Com isso, a propriedade anterior é levemente alterada, para todo nó v, $p(u) \leq p(v) < p(w)$, onde u é qualquer nó na subárvore com raiz em $e(v)$ e w é qualquer nó na subárvore com raiz em $d(v)$.

Seja T uma árvore de busca binária. A profundidade de um nó v de T é o comprimento do caminho entre a raiz de T e v. O conjunto dos nós de T que têm profundidade igual a i formam o nível i de T. Dados dois níveis i e j, $i < j$, dizemos que o nível j é mais alto que o nível i ou que o nível i é mais baixo que o nível j. A altura de um nó v é o comprimento do maior caminho entre v e uma folha descendente de v. A altura de T é igual à altura da raiz de T. Se T armazena os n elementos de um conjunto e sua altura é $O(\log n)$, então dizemos que T está balanceada. Uma árvore binária completa é uma árvore binária onde todos os nós internos possuem dois filhos e todas as folhas estão no mesmo nível.

Em alguns momentos, usamos o jargão da teoria dos grafos para falar sobre árvores. Uma árvore é formada por dois conjuntos V e A. Os elementos do conjunto V são chamados de vértices (ou nós) e os elementos do conjunto A de arestas. Os elementos do conjunto A são formados por pares do conjunto V. Dizemos que uma aresta uv incide nos vértices distintos u e v. O grau de um vértice v é o número de arestas que incidem em v. Uma introdução sucinta à teoria dos grafos pode ser encontrada em [13].

1.4.3 *Heap*

Um *heap* é uma estrutura de dados que pode ser vista como uma árvore binária quase completa, ou seja, todo nó interno possui dois filhos (excluindo possivelmente o nó interno mais profundo e mais à direita que pode ter somente o filho à esquerda) e toda folha está ou no último ou no penúltimo nível. Com isso a árvore binária representada por um *heap* que armazena n elementos está balanceada e podemos usar um vetor para representá-la. Os nós internos de um *heap* com n nós estão nas posições $0,\ 1,\ \ldots,$ e $\lfloor (n-2)/2 \rfloor$ do vetor, enquanto

que as folhas estão nas posições $\lfloor (n-2)/2 \rfloor + 1, \ldots,$ e $n-1$. O filho à direita de um nó interno armazenado na posição i de um *heap*, está armazenado na posição $2i+2$ e o filho à esquerda, na posição $2i+1$.

Outra propriedade de um *heap* é: a chave de um nó interno v é maior que a chave dos seus filhos (*maxheap*) que pode variar para, a chave de v é menor que a chave de seus filhos (*minheap*). Na Figura 1.6 ilustramos um *minheap*.

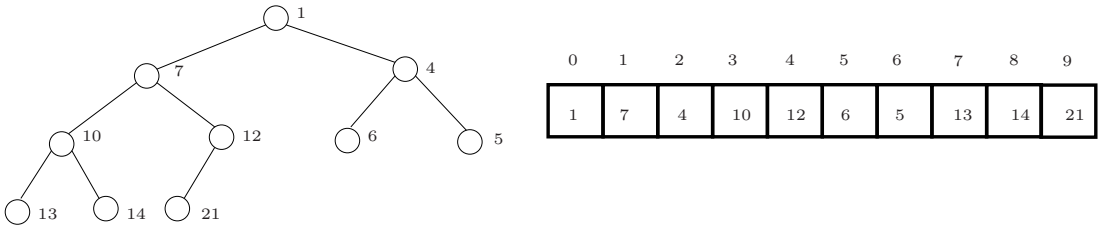


Figura 1.6: Um exemplo de um *minheap*.

1.5 Uma estrutura geométrica - *kd-tree*

A estrutura *kd-tree* é uma estrutura clássica usada para resolver o problema de consultas em janelas. Dado um conjunto P de pontos no plano sua construção é feita dividindo o plano em semiplanos e armazenando em cada nó interno uma região do plano e em cada folha um ponto de P. Uma *kd-tree* com raiz em v é construída sobre os pontos que estão na região de v. O conjunto que contém esses pontos é denotado por $P(v)$. Assim, a região da raiz de uma *kd-tree* corresponde a todo o plano. A região do filho à esquerda da raiz corresponde ao semiplano à esquerda (e inclusive) a uma reta vertical r_v que passa pela x-mediana de P, e a região do filho à direita da raiz corresponde ao semiplano à direita de r_v. A raiz da subárvore à esquerda do filho à esquerda da raiz armazena uma região formada pela intersecção entre a região de seu pai (no nosso caso essa região é o semiplano à esquerda de r_v) e o semiplano abaixo de uma reta horizontal r_h que passa pela y-mediana dos pontos em $P(e(v))$. A subárvore à direita do filho à esquerda da raiz armazena uma região formada pela intersecção entre a região de seu pai e o semiplano acima de r_h. As subárvores à esquerda e à direita do filho à direita da raiz são construídas de maneira simétrica. Note que em um nó de um nível par armazenamos uma região formada pela intersecção entre seu pai e um semiplano definido por uma reta vertical enquanto que em um nó de um nível ímpar armazenamos uma região formada pela intersecção entre seu pai e um semiplano definido por uma reta horizontal. O consumo de tempo da construção de uma *kd-tree* sobre um conjunto com n pontos é $\Theta(n \log n)$ e o consumo de espaço é $\Theta(n)$. A Figura 1.7 ilustra uma *kd-tree* construída sobre um conjunto de pontos no plano.

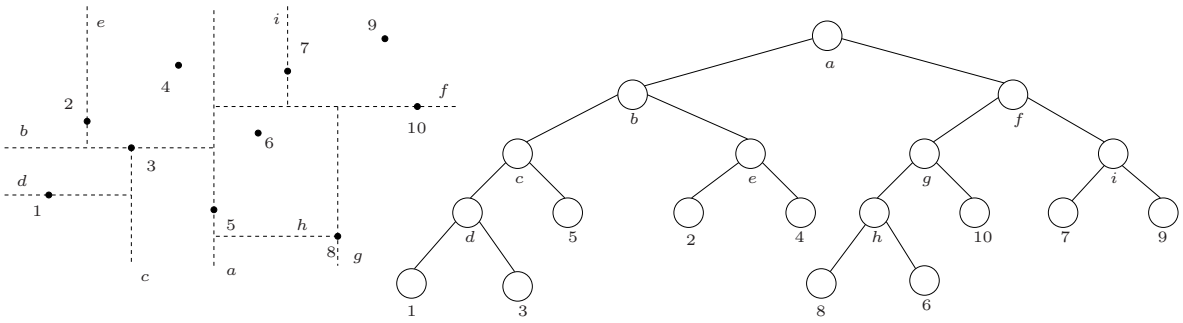


Figura 1.7: Uma *kd-tree* construída sobre um conjunto de pontos no plano.

Uma consulta é resolvida da seguinte forma. Dada uma janela no plano W e um nó v de uma *kd-tree*, se W contém a região armazenada em v então liste todos os pontos que estão armazenados nas folhas da subárvore com raiz em v. Se W não contém a região armazenada em v então busque pelos pontos que estão nas subárvores com raiz em $e(v)$ e/ou $d(v)$ desde que W intercepte as regiões $e(v)$ e/ou $d(v)$. A Figura 1.8 ilustra uma consulta em uma *kd-tree*. As arestas em destaque são usadas nesta consulta. Note que a janela ilimitada contém a região armazenada no nó $*$. Sendo assim, os pontos armazenados nas folhas da subárvore com raiz em $*$ estão na janela.

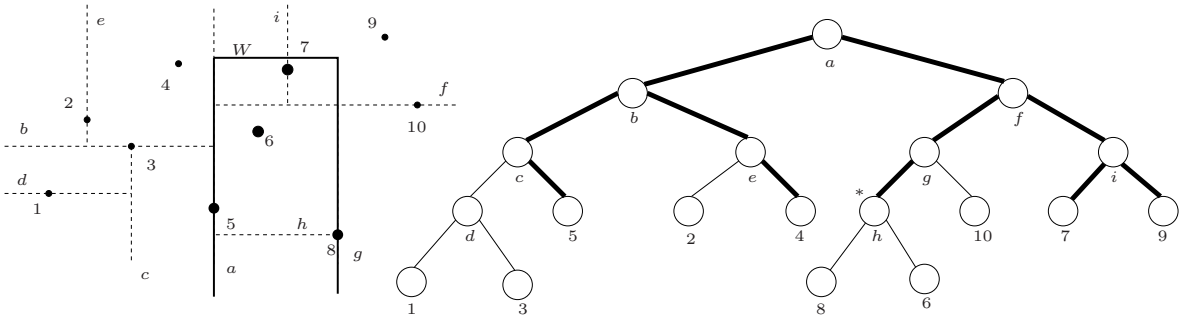


Figura 1.8: Uma janela ilimitada e uma consulta em uma *kd-tree*.

Para resolver uma consulta consumimos tempo $O(\sqrt{n}+k)$, onde n é o número de pontos de um conjunto P e k é o número de pontos de P que estão em W. No próximo capítulo vamos estudar estruturas de dados e algoritmos que melhoram esse consumo de tempo.

Capítulo 2

Pontos em janelas

O primeiro subproblema que vamos resolver é a busca por pontos em uma janela. Dado um conjunto P com n pontos, inicialmente organizamos os elementos de P em uma estrutura de dados chamada árvore limite (fase de pré-processamento). Em seguida, dizemos como resolver uma W-consulta sobre P usando a estrutura criada e armazenando em uma lista todos os pontos de P que estão em uma janela W (fase de consultas). A estrutura de dados árvore limite foi criada independentemente por [4, 18, 19, 29]. Ela é uma árvore de busca binária balanceada multinível. Neste tipo de estrutura temos uma árvore de busca binária (a de primeiro nível) em que cada nó aponta para a raiz de uma outra árvore de busca binária (a de segundo nível), em que os nós podem apontar para outras árvores e assim por diante. Trabalhamos com árvores limite com até dois níveis e as usamos para resolver o subproblema de interesse deste capítulo. A estrutura de dados árvore de busca com prioridade [21] é uma árvore de busca binária balanceada. As vantagens e desvantagens destas duas estruturas serão percebidas durante este capítulo. Começamos considerando o subproblema em uma dimensão. Assim, os pontos estão na reta e W é um intervalo. Depois estendemos o subproblema para duas dimensões tornando W uma região retangular sobre o plano. Em ambos os casos chamamos W de janela.

2.1 Pontos em janelas - unidimensional

Seja P um conjunto com n pontos distintos sobre a reta. Sejam w_1 e w_2 pontos na reta onde $w_1 \leq w_2$. Uma janela W na reta é formada por um intervalo fechado com limites em w_1 e w_2. Como podemos organizar os elementos de P tal que uma W-consulta sobre P possa ser feita rapidamente? Na fase de pré-processamento podemos armazenar os elementos de P em um vetor V e ordená-lo de tal forma que $p(V[0]) < p(V[1]) < \ldots < p(V[n-1])$. Com isso, consumimos tempo $\Theta(n \log n)$. Na fase de consultas, um usuário define uma janela W através de dois pontos, w_1 e w_2, e realiza uma W-consulta sobre P que pode ser resolvida da seguinte forma: realizamos uma busca binária por w_1 em V e encontramos uma posição i de V que armazena o ponto com menor valor maior ou igual a w_1. Percorremos V a partir da posição i listando todos os pontos com valor menor ou igual a w_2. Esta solução consome tempo $O(\log n + k)$ para listar k pontos de um conjunto que estão em uma janela. Vamos mostrar uma solução alternativa que faz uso de uma árvore de busca binária T. Primeiramente,

organizamos os pontos de P em T na fase de pré-processamento de forma que as folhas de T armazenam os pontos de P. Ao percorrer as folhas da esquerda para a direita, percorremos os ponto de P ordenados. Os nós internos de T armazenam pontos que direcionam uma busca na árvore. Na Figura 2.1, ilustramos uma árvore construída com essa ideia.

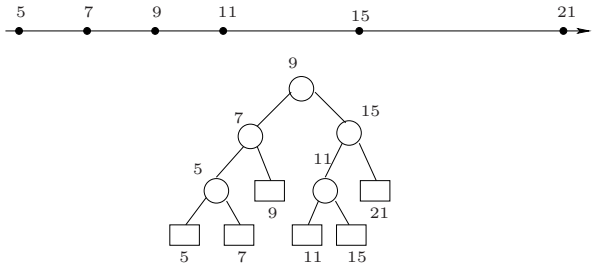


Figura 2.1: Uma árvore limite com 1-nível.

Em seguida, o algoritmo ConstroiÁrvoreLimite1D descreve formalmente a construção de uma árvore limite sobre um conjunto de pontos na reta (árvore limite 1-nível). Na construção usamos um vetor V de pontos ordenados. Como consequência, obtemos um ponto médio de V (um ponto que divide V em dois vetores com tamanhos quase iguais) em tempo constante. Esta ordenação prévia pode ser evitada se usarmos uma árvore rubro-negra [2] ou AVL [1].

Algoritmo 1 ConstroiÁrvoreLimite1D(vetor V)

Entrada: Um conjunto não vazio P de pontos distintos através de um vetor ordenado $V = [p_1, p_2, \ldots, p_n]$, com $p_1 < p_2 < \ldots < p_n$.
Saída: A raiz v de uma árvore limite 1-nível.

1. Crie um nó v.
2. Crie dois vetores $V_e = [p_1, p_2, \ldots, p_{\lceil \frac{n}{2} \rceil}]$ e $V_d = [p_{\lceil \frac{n}{2} \rceil + 1}, p_{\lceil \frac{n}{2} \rceil + 2}, \ldots, p_n]$
3. $p(v) \leftarrow p_{\lceil \frac{n}{2} \rceil}$
4. **se** V contém um único ponto **então**
5. Marque v como folha.
6. **senão**
7. $e(v) \leftarrow$ ConstroiÁrvoreLimite1D(V_e)
8. $d(v) \leftarrow$ ConstroiÁrvoreLimite1D(V_d)
9. **fim se**
10. **devolva** v

Lema 1 **[4, 18, 19, 29]** *Dado um conjunto de n pontos distintos, $n \geq 1$, o algoritmo* ConstroiÁrvoreLimite1D *constroi uma árvore limite 1-nível usando um vetor ordenado em tempo $\Theta(n \log n)$.*

Demonstração: A demonstração segue da solução da seguinte recorrência.

$$T(n) = \begin{cases} \Theta(1), & \text{se} \quad n = 1 \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n), & \text{se} \quad n \geq 2 \end{cases}$$ □

Considere o seguinte algoritmo que constroi uma árvore sobre um conjunto com n elementos.

Algoritmo 2 ConstroiÁrvoreSobreConjunto(conjunto P)

Entrada: Um conjunto ordenado não vazio $P = \{p_1, p_2, \ldots, p_n\}$ com n elementos.
Saída: A raiz v de uma árvore de busca binária.

1. Crie um nó v.
2. **se** $n = 1$ **então**
3. Marque v como folha.
4. **senão**
5. $e(v) \leftarrow$ ConstroiÁrvoreSobreConjunto($P_1 = \{p_1, p_2, \ldots, p_{\lceil \frac{n}{2} \rceil}\}$)
6. $d(v) \leftarrow$ ConstroiÁrvoreSobreConjunto($P_2 = \{p_{\lceil \frac{n}{2} \rceil + 1}, \ldots, p_n\}$)
7. **fim se**
8. **devolva** v

Lema 2 *Qualquer árvore construída sobre um conjunto com n elementos, $n \geq 1$, pelo algoritmo* ConstroiÁrvoreSobreConjunto *possui altura no máximo $\lceil \log n \rceil$.*

Demontração: Vamos mostrar este resultado por indução em n.

Para $n = 1$ temos que a altura de T é 0 e $\lceil \log 1 \rceil = 0$.

Figura 2.2: Uma árvore construída pelo algoritmo com $P = \{p_1\}$ e $n = 1$.

Agora, suponha que T foi construída sobre um conjunto com $n \geq 2$ elementos, então $T' = T - raiz(T)$ é uma floresta com duas árvores T'_1 e T'_2. Sabemos que T'_1 e T'_2 foram construídas, respectivamente, sobre conjuntos com $\lceil \frac{n}{2} \rceil$ e $\lfloor \frac{n}{2} \rfloor$ elementos. Logo, por indução, a altura de T'_1 é no máximo $\lceil \log \lceil \frac{n}{2} \rceil \rceil$ e a altura de T'_2 é no máximo $\lceil \log \lfloor \frac{n}{2} \rfloor \rceil \leq \lceil \log \lceil \frac{n}{2} \rceil \rceil$. Com isso, a altura h de T é no máximo $\lceil \log \lceil \frac{n}{2} \rceil \rceil + 1$. Agora, vamos mostrar que $\lceil \log \lceil \frac{n}{2} \rceil \rceil + 1 = \lceil \log n \rceil$, para todo $n \geq 2$ e assim, segue o lema.

Suponha que n seja par. Então $\lceil \frac{n}{2} \rceil = \frac{n}{2}$, logo $\lceil \log \lceil \frac{n}{2} \rceil \rceil + 1 = \lceil \log \frac{n}{2} \rceil + 1 = \lceil \log n - \log 2 + 1 \rceil = \lceil \log n \rceil$.

Agora, suponha que n seja ímpar. Então $\lceil \frac{n}{2} \rceil = \frac{n+1}{2}$, logo $\lceil \log \lceil \frac{n}{2} \rceil \rceil + 1 = \lceil \log \frac{n+1}{2} \rceil + 1 = \lceil \log(n+1) - \log 2 + 1 \rceil = \lceil \log(n+1) \rceil = \lceil \log n \rceil$. A última igualdade é válida pois, n é ímpar e assim $n > 2$ e n está entre duas potências de 2, $2^{k-1} < n < 2^k$, para k inteiro maior ou igual a 2. Logo, $\lceil \log n \rceil = k$. Por outro lado, somando 1 ao valor de n temos que $2^{k-1} < n+1 \leq 2^k$ e a seguinte igualdade vale $\lceil \log(n+1) \rceil = k$. □

Já na fase de consultas, uma W-consulta sobre P é resolvida da seguinte forma. Procuramos por w_1 e w_2 na árvore T e encontramos dois caminhos C e C'. Temos duas possibilidades para esses dois caminhos:

- $C = C'$. Neste caso, os caminhos terminam em uma folha u de T e basta verificar se este ponto é maior ou igual a w_1 e menor ou igual a w_2. Note que, se a árvore está balanceada, o algoritmo consome tempo $O(\log n)$.
- $C \neq C'$. Com isso, existe um nó v_{div} que divide os caminhos C e C', ou seja, C segue por $e(v_{div})$ e C' por $d(v_{div})$.

Afirmamos que, os pontos que pertencem à janela W estão armazenados na subárvore com raiz em v_{div}. Vamos ver que isso é verdade. Considere um ponto p tal que p pertence a uma janela W e p não pertence a $T_{v_{div}}$ (subárvore de T com raiz em v_{div}). Considere o caminho C_p da raiz até p e o primeiro vértice em que C_p difere de C e C'. Como p não pertence a $T_{v_{div}}$, será um vértice v' anterior a v_{div}. Supomos, sem perda de generalidade, que os caminhos C e C' seguem à esquerda de v' e C_p à direita.

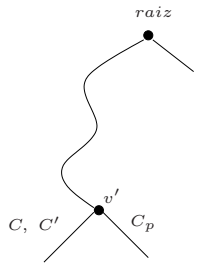


Figura 2.3: Como C e C' seguem à esquerda de v' e C_p à direita temos que $w_2 \leq p(v') < p$.

Assim, $w_2 \leq p(v') < p$. Mas, como p pertence a W, $p \leq w_2$ que é uma contradição. Com isso, podemos começar uma busca por pontos que estão em uma janela a partir deste nó. O algoritmo EncontraNóDivide encontra ou o nó v_{div}, ou uma folha onde os dois caminhos C e C' terminam.

Considere que v_{div} existe. Então, $w_1 \leq p(v_{div}) < w_2$ e assim sabemos que $w_1 \neq w_2$. Sejam f e f' as folhas dos caminhos C e C'. Uma informação importante é que os pontos

Algoritmo 3 EncontraNóDivide(árvore_limite v_{raiz}, janela W)

Entrada: Uma árvore limite com raiz em v_{raiz} e uma janela $W = (w_1, w_2)$.

Saída: O nó v_{div} que divide os caminhos C e C' ou uma folha onde ambos caminhos terminam.

1. $v_{div} \leftarrow v_{raiz}$
2. **enquanto** v_{div} não é uma folha **e** ($w_1 > p(v_{div})$ **ou** $w_2 \leq p(v_{div})$) **faça**
3. **se** $w_2 \leq p(v_{div})$ **então**
4. $v_{div} \leftarrow e(v_{div})$
5. **senão**
6. $v_{div} \leftarrow d(v_{div})$
7. **fim se**
8. **fim enquanto**
9. **devolva** v_{div}

armazenados na subárvore à esquerda de v_{div} têm valor menor que w_2 e os pontos armazenados na subárvore à direita têm valor maior que w_1. Considere um nó v descendente à esquerda de v_{div} pertencente ao caminho C. Se $w_1 \leq p(v)$, então os pontos armazenados na $T_{d(v)}$ têm valor entre w_1 e w_2. Portanto, devemos listá-los (veja a Figura 2.4).

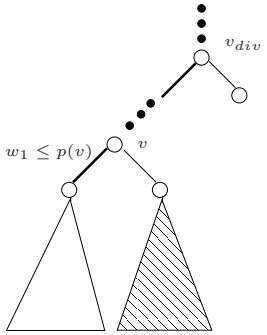


Figura 2.4: Os pontos na subárvore hachurada estão em W.

Se $w_1 > p(v)$, então os pontos que estão em $T_{e(v)}$ não estão no intervalo e a busca continua para a direita de v. Na folha f, verificamos se o ponto nela armazenado está no intervalo. De modo similar ocorre com um nó v descendente à direita de v_{div}. Se $w_2 > p(v)$, então os pontos armazenados na $T_{e(v)}$ têm valor na janela e devemos listá-los. Se $w_2 \leq p(v)$, então continuamos a busca na subárvore à esquerda de v, pois os pontos armazenados na $T_{d(v)}$ não podem estar na janela. Quando chegamos em f', verificamos se o ponto armazenado nesta folha tem valor na janela. O algoritmo ConsultaÁrvoreLimite1D realiza essa busca. Nele, usamos uma subrotina chamada PontosSubárvore que recebe uma árvore limite T_v com raiz em v e devolve todos os pontos armazenados nas suas folhas. Suponha que esta subrotina visite cada nó u de T_v e liste somente os pontos que estão armazenados em u quando u é

uma folha. Note que o consumo de tempo desta subrotina é $O(k_v)$, onde k_v é o número de folhas de T_v. Vamos ver porque isso é verdade mostrando que o número de nós internos de uma árvore limite 1-nível é menor que o seu número de folhas.

Lema 3 *O número de nós internos de uma árvore limite* 1*-nível é menor que o seu número de folhas.*

Demonstração: Seja T uma árvore limite 1-nível com N nós. Sejam k e l os números de nós internos e folhas, respectivamente, de T. Sabemos que $k + l = N$. Em toda árvore, o seu número de arestas é igual ao seu número de nós menos um. No nosso caso, o número de arestas de T é $N - 1$. Pela construção de T, sabemos que todo nó interno possui dois filhos. Logo, o grau de cada nó interno de T é três (dois filhos e um pai), com exceção da raiz cujo grau é dois (a raiz não tem pai). Cada folha possui grau igual a um. Assim, somando as arestas de cada nó de T temos $3(k-1)+2+l = 2(N-1)$. Resolvendo esse sistema encontramos $k = \frac{(N-1)}{2}$ e $l = \frac{(N+1)}{2}$. Com isso, $k = l - 1$ e, portanto, o número de nós internos de uma árvore limite 1-nível é menor que o seu número de folhas. □

Podemos usar outra estrutura de dados para obter os pontos armazenados em cada folha de T_v. Esta nova estrutura é uma varição da árvore com fios (*threaded tree*) [30] e chamamos de árvore limite 1-nível com fios. Sua construção pode ser feita da seguinte forma. Seja T_v uma árvore limite 1-nível sobre um conjunto com n pontos. Realizamos sobre T_v um percurso em-ordem (*in-order*) mantendo a última folha visitada f'. Ao visitar uma nova folha f, f' aponta para f que, a partir deste momento, passa ser a última folha visitada. Este procedimento consome o mesmo tempo de um algoritmo que percorre T_v em-ordem, ou seja, $\Theta(n)$, pois, visitamos cada nó v da árvore uma única vez e em cada visita consumimos tempo constante, ou para criar os fios e atualizar f', caso v seja uma folha, ou para chamar os dois filhos de v, caso v seja um nó interno. Um exemplo de uma árvore limite 1-nível com fios pode ser visto na figura 2.5.

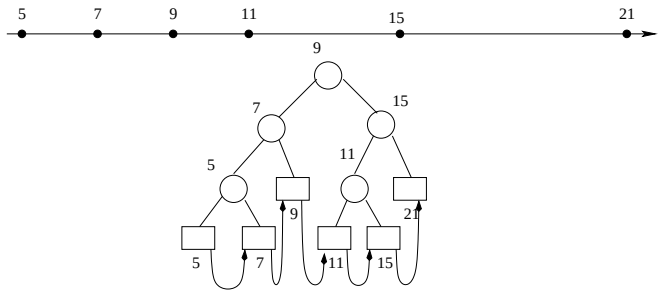


Figura 2.5: Uma árvore limite 1-nível com fios.

Os pontos armazenados nas folhas de tal estrutura podem ser obtidos assim: primeiramente encontramos a sua folha mais à esquerda e através dela visitamos cada folha usando os ponteiros criados entre elas. Em cada folha, listamos os pontos nelas armazenados. Como

Algoritmo 4 ConsultaÁrvoreLimite1D(árvore_limite v_{raiz}, janela W)

Entrada: Um conjunto não vazio P de pontos distintos armazenado em uma árvore limite T com raiz em v_{raiz} e uma janela W.

Saída: Uma lista L que contém os pontos de P que pertencem a W.

1. $L \leftarrow \emptyset$
2. $v_{div} \leftarrow$ EncontraNóDivide(v_{raiz}, W)
3. **se** v_{div} é uma folha **então**
4. ▷ Verifique se o ponto associado à folha v_{div} deve ser armazenado em L.
5. **se** $w_1 \leq p(v_{div}) \leq w_2$ **então**
6. $L \leftarrow \{p(v_{div})\}$
7. **fim se**
8. **senão**
9. ▷ Ande no caminho C.
10. $v \leftarrow e(v_{div})$
11. **enquanto** v não é uma folha **faça**
12. **se** $w_1 \leq p(v)$ **então**
13. ▷ Armazene todos os pontos na subárvore com raiz em $d(v)$ em L.
14. $L \leftarrow L \cup$ PontosSubárvore($d(v)$)
15. $v \leftarrow e(v)$
16. **senão**
17. $v \leftarrow d(v)$
18. **fim se**
19. **fim enquanto**
20. ▷ Verifique se o ponto associado à folha v deve ser armazenado em L.
21. **se** $w_1 \leq p(v) \leq w_2$ **então**
22. $L \leftarrow L \cup \{p(v)\}$
23. **fim se**
24. ▷ Ande no caminho C'.
25. $v \leftarrow d(v_{div})$
26. **enquanto** v não é uma folha **faça**
27. **se** $w_2 > p(v)$ **então**
28. ▷ Armazene todos os pontos na subárvore com raiz em $e(v)$ em L.
29. $L \leftarrow L \cup$ PontosSubárvore($e(v)$)
30. $v \leftarrow d(v)$
31. **senão**
32. $v \leftarrow e(v)$
33. **fim se**
34. **fim enquanto**
35. ▷ Verifique se o ponto associado à folha v deve ser armazenado em L.
36. **se** $w_1 \leq p(v) \leq w_2$ **então**
37. $L \leftarrow L \cup \{p(v)\}$
38. **fim se**
39. **fim se**
40. **devolva** L

T_v possui altura $O(\log n)$, este procedimento consome tempo $O(\log n)$ para encontrar a folha mais à esquerda de T_v mais k para listar as folhas de T_v.

O algoritmo CriaFios, recebe uma árvore limite 1-nível T com raiz em v e devolve T com fios. Nele a folha f' representa a última folha visitada em um percurso em-ordem. Antes da primeira chamada deste algoritmo, f' deve apontar para um valor $NULO$.

Algoritmo 5 CriaFios(árvore_limite v, folha f')

Entrada: A raiz v de uma árvore limite 1-nível T.
Saída: T com fios e a folha mais à direita de T.

```
1. se v é uma folha então
2.    se f' ≠ NULO então
3.       d(f') ← v
4.    fim se
5.    f' ← v
6. senão
7.    e(v), f' ← CriaFios(e(v), f')
8.    d(v), f' ← CriaFios(d(v), f')
9. fim se
10. devolva v, f'
```

Lema 4 **[4, 18, 19, 29]** *Sejam P um conjunto com n pontos distintos e W uma janela, ambos sobre a reta. Seja T uma árvore limite 1-nível construída sobre P. O algoritmo* ConsultaÁrvoreLimite1D *consome tempo $O(\log n + k)$, onde k é o número de pontos de P que pertencem a W.*

Demonstração: Pelo Lema 2, T possui altura $O(\log n)$ e assim, consumimos tempo $O(\log n)$ para encontrar v_{div} em T e visitamos $O(\log n)$ nós nas subárvores à esquerda e à direita de v_{div} e, possivelmente, em um nó v, descendente de v_{div}, consumimos tempo $O(k_v)$ para listar os pontos armazenados nas folhas da subárvore com raiz em v que estão na janela W. Como $\sum_v ck_v = c\sum_v k_v = ck = O(k)$, para alguma constante c positiva, segue que o consumo de tempo do algoritmo é $O(\log n + k)$. □

Na próxima seção, falaremos sobre busca por pontos no plano. Para isso, devemos estender uma árvore limite 1-nível para uma 2-níveis.

2.2 Pontos em janelas - bidimensional

Sejam $w_1 = (x, y)$ e $w_2 = (x', y')$ pontos no plano onde $x < x'$ e $y < y'$. Os segmentos de reta que formam um retângulo com os lados paralelos aos eixos são: $s_1 = \overline{(x, y)(x, y')}$, $s_2 = \overline{(x, y')(x', y')}$, $s_3 = \overline{(x', y')(x', y)}$ e $s_4 = \overline{(x', y)(x, y)}$. Uma janela W é formada pelos segmentos de reta de um retângulo e sua região interna. Ao longo deste texto, usaremos alguns símbolos de comparação. São eles: $\leq_x$, $>_x$, $\leq_y$, $>_y$. Usamos estes símbolos para

comparar dois pontos no plano. Dizemos que um ponto p é x-(menor ou igual) a um ponto q ($p \leq_x q$) se, e somente se, $x(p) < x(q)$ ou ($x(p) = x(q)$ e $y(p) \leq y(q)$). Um ponto p é x-maior que um ponto q se, e somente se, p não é x-(menor ou igual) a q. De forma análoga definimos quando um ponto é y-(menor ou igual) e y-maior que outro.

Uma árvore limite T construída sobre um conjunto de pontos no plano tem dois níveis. A árvore do primeiro nível (ou árvore principal) é construída sobre as x-coordenadas dos pontos de P. Novamente, os pontos de P são armazenados nas suas folhas e os nós internos armazenam pontos que direcionam uma busca no eixo x. Cada nó da árvore principal aponta agora, adicionalmente, para uma estrutura associada que também é uma árvore de busca binária balanceada. Denotamos por $P(v)$ os pontos armazenados nas folhas da subárvore com raiz em um nó v da árvore principal. A estrutura associada de v, ou seja, a árvore do segundo nível de v, é construída sobre os pontos de $P(v)$. As suas folhas armazenam os pontos de $P(v)$ e seus nós internos armazenam pontos que orientam uma busca no eixo y. Uma estrutura associada a um nó v de T é denotada por $T_y(v)$. Um exemplo parcial de uma árvore limite 2-níveis pode ser visto na Figura 2.6.

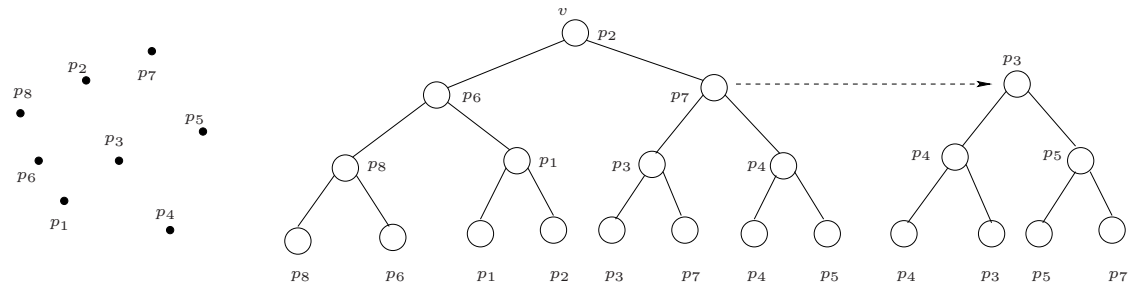


Figura 2.6: Uma árvore limite 2-níveis parcial.

Vamos falar mais sobre a construção de uma árvore limite 2-níveis. Considere um conjunto de pontos distintos no plano P e dois vetores ordenados sobre P. O primeiro vetor denotado por $V^x = [p_1, p_2, \ldots, p_n]$ ordenado de tal forma que $p_1 <_x p_2 <_x \ldots <_x p_n$ e o segundo vetor $V^y = [q_1, q_2, \ldots, q_n]$ tal que $q_1 <_y q_2 <_y \ldots <_y q_n$. Dizemos que o vetor V^x está x-ordenado e que V^y está y-ordenado. Uma árvore limite 2-níveis sobre P é construída recursivamente usando os vetores V^x e V^y da seguinte forma. Crie um nó v e armazene nele o ponto $p_{\lceil \frac{n}{2} \rceil}$. Construa a estrutura associada de v usando o vetor V^y. Construa dois vetores x-ordenados, $V^x_e = [p_1, p_2, \ldots, p_{\lceil \frac{n}{2} \rceil}]$ e $V^x_d = [p_{\lceil \frac{n}{2} \rceil + 1}, p_{\lceil \frac{n}{2} \rceil + 2}, \ldots, p_n]$ e dois vetores y-ordenados, V^y_e e V^y_d, com os mesmos pontos em V^x_e e V^x_d mas ordenados pela y-coordenada. A subárvore à esquerda de v é construída da mesma forma usando os vetores V^x_e e V^y_e e a subárvore à direita de v, usando os vetores V^x_d e V^y_d. Conseguimos construir em tempo linear os vetores V^x_e, V^x_d, V^y_e e V^y_d percorrendo os vetores V^x e V^y. A construção dos vetores V^x_e e V^x_d é trivial. Os vetores V^y_e e V^y_d podem ser construídos da seguinte forma. Para cada ponto q em V^y, compare a coordenada x de q com a de $p_{\lceil \frac{n}{2} \rceil}$. Se $q \leq_x p_{\lceil \frac{n}{2} \rceil}$ então q pertence ao vetor V^x_e, e assim q é armazenado no vetor V^y_e. Caso contrário, q é armazenado no vetor V^y_d. Considere a Figura

2.6 para interpretar a Figura 2.7.

O algoritmo ConstroiÁrvoreLimite descreve formalmente uma maneira de construir uma árvore limite 2-níveis. Note que, se a linha 2 deste algoritmo consumir tempo linear então a recorrência que define o seu consumo de tempo é igual à recorrência da demonstração do Lema 1. Com isso, o seu consumo de tempo é $\Theta(n \log n)$.

$$V^x = [p_8, p_6, p_1, p_2, p_3, p_7, p_4, p_5] \longrightarrow \begin{array}{l} V_e^x = [p_8, p_6, p_1, p_2] \\ V_d^x = [p_3, p_7, p_4, p_5] \end{array}$$

$$V^y = [p_4, p_1, p_6, p_3, p_5, p_8, p_2, p_7] \longrightarrow \begin{array}{l} V_e^y = [p_1, p_6, p_8, p_2] \\ V_d^y = [p_4, p_3, p_5, p_7] \end{array}$$

Figura 2.7: Os pontos do conjunto $P(v)$ estão representados nos vetores x-ordenado V^x e y-ordenado V^y, os pontos do conjunto $P(e(v))$ nos vetores ordenados V_e^x e V_e^y e os pontos do conjunto $P(d(v))$ nos vetores ordenados V_d^x e V_d^y.

Algoritmo 6 ConstroiÁrvoreLimite(vetor V^x, vetor V^y)

Entrada: Um conjunto de pontos distintos não vazio P através de dois vetores: um x-ordenado $V^x = [p_1, p_2, \ldots, p_n]$ e outro y-ordenado $V^y = [q_1, q_2, \ldots, q_n]$.
Saída: A raiz v de uma árvore limite 2-níveis.

1. Crie um nó v.
2. $T_y(v) \leftarrow$ ConstroiEstruturaAssociada(V^y)
3. Crie quatro vetores vazios: V_e^x, V_d^x, V_e^y e V_d^y
4. $V_e^x \leftarrow [p_1, p_2, \ldots, p_{\lceil \frac{n}{2} \rceil}]$.
5. $V_d^x \leftarrow [p_{\lceil \frac{n}{2} \rceil + 1}, p_{\lceil \frac{n}{2} \rceil + 2}, \ldots, p_n]$.
6. **para** $i \leftarrow 1$ até n **faça**
7. **se** $q_i \leq_x p_{\lceil \frac{n}{2} \rceil}$ **então**
8. Insira no final do vetor V_e^y o ponto q_i.
9. **senão**
10. Insira no final do vetor V_d^y o ponto q_i.
11. **fim se**
12. **fim para**
13. $p(v) \leftarrow p_{\lceil \frac{n}{2} \rceil}$
14. **se** V^x contém um único ponto **então**
15. Marque v como folha.
16. **senão**
17. $e(v) \leftarrow$ ConstroiÁrvoreLimite(V_e^x, V_e^y)
18. $d(v) \leftarrow$ ConstroiÁrvoreLimite(V_d^x, V_d^y)
19. **fim se**
20. **devolva** v

Falta então, descrever como construímos uma estrutura associada de um nó v de T. Ela deve ser uma árvore de busca binária. Podemos construí-la baseados na ideia de construção de *heap*. Os membros de cada nó u desta árvore são:

- um ponto, $p(u)$; e
- o ponto armazenado na folha mais à direita do seu filho à direita, que vamos denotar por $p_d(u)$.

Como na árvore principal, $p(u)$ é usado para orientar uma busca. Já $p_d(u)$ é uma informação usada durante sua construção. Para garantir um consumo de tempo linear, vamos construir T_y usando uma estratégia "de baixo para cima".

Seja P um conjunto com n pontos distintos. Sejam l o número de folhas do último nível h e l' o número de folhas do penúltimo nível $h-1$ de T_y. Como o número de folhas é n temos que, $l + l' = n$. Sabemos que, ao somar l com $2l'$ teremos o número máximo de nós no nível h, ou seja, $l + 2l' = 2^h$. Resolvendo este sistema, encontramos o número de folhas nos níveis $h-1$ ($l' = 2^h - n$) e h ($l = n - l'$). O valor de h é dado pelo primeiro nível de T_y tal que o seu número de nós é maior ou igual a n, ou seja, h é o menor inteiro tal que $2^h \geq n$. Mais precisamente, $h = \lceil \log n \rceil$. h também é a altura de T_y.

Pelo Lema 3 sabemos que o números de nós da árvore T_y é $N = 2n - 1$. Com as informações: o número de nós; o número de folhas em h; e o número de folhas em $h-1$, podemos efetivamente construir T_y começando pelas suas n folhas. Suponha que os pontos de P estão armazenados em um vetor y-ordenado $V^y = [q_1, q_2, \ldots, q_n]$. Os pontos armazenados nas l folhas do nível h são os l primeiros pontos do vetor V^y, ou seja, $q_1, q_2, \ldots, q_l$. Eles devem ser armazenados nas l últimas posições de T_y. Os pontos armazenados nas l' folhas restantes são os l' pontos restantes de V^y, $q_{l+1}, q_{l+2}, \ldots, q_n$. Eles devem ser armazenados nas posições que antecedem as l últimas posições do vetor. Note que, ao percorremos as folhas da árvore T_y da esquerda para a direita temos uma y-ordenação dos pontos nelas armazenados. O exemplo da Figura 2.9 ilustra o preenchimento das folhas.

Completando a construção, para cada nó interno u de T_y, $p(u) = p_d(e(u))$ e $p_d(u) = p_d(d(u))$. Os primeiros nós internos a serem construídos estão no nível $h-1$. Em seguida, construímos os nós do nível $h-2$, e assim por diante. Perceba que, a manutenção de $p_d(u)$ é importante, pois, se u é filho à esquerda de um nó v, então $p(v)$ deve ser igual a $p_d(u)$. Veja como fica a estrutura T_y da Figura 2.9 na Figura 2.10 depois de completada sua construção.

Lema 5 *Uma estrutura associada T_y é uma árvore de busca binária com altura $O(\log n)$.*

Demonstração: Vamos mostrar que uma estrutura construída como descrevemos acima é uma árvore de busca binária, mostrando que o ponto armazenado em qualquer nó da estrutura é y-(maior ou igual) a um ponto armazenado em qualquer nó da sua subárvore à esquerda e y-menor que um ponto armazenado em qualquer nó da sua subárvore à direita.

Seja u um nó de uma estrutura associada T_y. Sejam u_e e u_d dois nós descendentes à esquerda e à direita de u, respectivamente. Pela construção, $p(u) = p_d(e(u))$, $p(u_e) = p_d(e(u_e))$ e $p(u_d) = p_d(e(u_d))$ e $p_d(e(u))$, $p_d(e(u_e))$ e $p_d(e(u_d))$ são pontos armazenados em uma folha de T_y. Então temos que mostrar que $p_d(e(u_e)) \leq_y p_d(e(u)) <_y p_d(e(u_d))$. Pela y-ordenação das folhas é direto perceber isso (veja a Figura 2.8). Logo, a estrutura associada é uma árvore de busca binária. Pela sua construção, $h = \lceil \log n \rceil$ e assim ela é balanceada. □

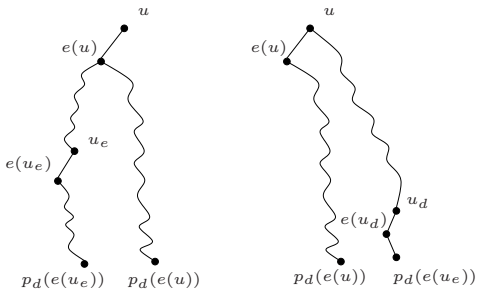


Figura 2.8: Pela y-ordenação das folhas temos que $p_d(e(u_e)) = p(u_e) \leq_y p_d(e(u)) = p(u) <_y p_d(e(u_d)) = p(u_d)$.

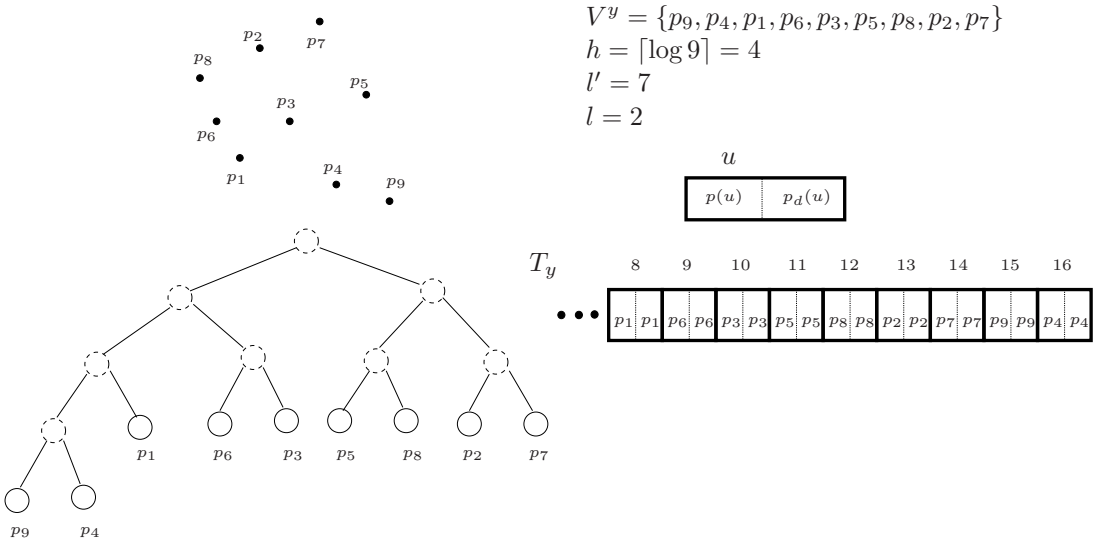


Figura 2.9: Preenchimento das folhas de uma estrutura associada, T_y, baseada em *heap*. Perceba que as folhas do nível h são filhas dos nós internos mais à esquerda do nível $h-1$.

O algoritmo ConstroiEstruturaAssociada constroi tal árvore. Note que as linhas 8 – 12 e 15 – 19 juntas executam n vezes. As linhas 7 e 14 executam no total duas vezes mais, ou seja, $n+2$. Nas linhas 11 e 18 a variável i é decrementada. Como i possui inicialmente valor igual a $2n-1$ teremos na primeira iteraçao do enquanto da linha 21 i com valor $n-1$. Com isso, as linhas 22 – 25 são executadas $n-1$ vezes e a linha 21 é executada

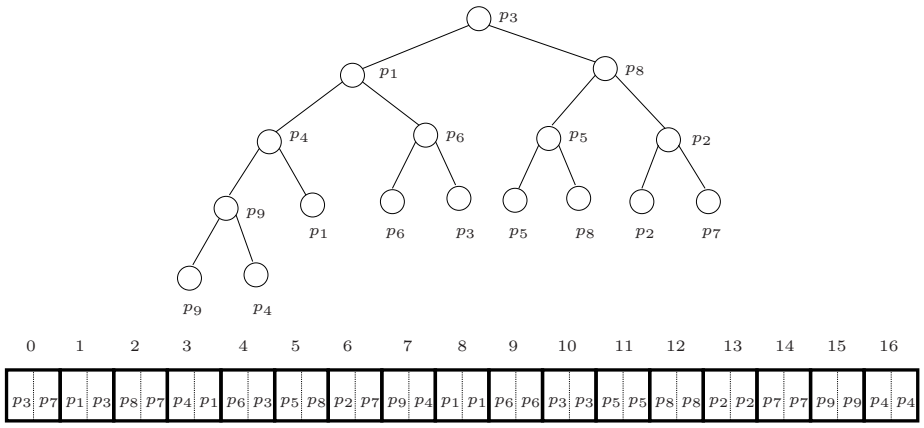


Figura 2.10: T_y totalmente preenchida. Está ilustrado em cada nó v da árvore somente os pontos que orientam uma busca, ou seja, $p(v)$. Em cada nó v do vetor ilustramos $p(v)$ e $p_d(v)$.

n vezes. Como toda linha consome tempo constante, o consumo de tempo do algoritmo ConstroiEstruturaAssociada é $\Theta(n)$.

Passada a fase de pré-processamento, mostraremos como responder a uma consulta. Seja T uma árvore limite 2-níveis construída sobre um conjunto de pontos distintos P com cardinalidade igual a n. Uma W-consulta sobre P devolve todos os pontos pertencentes a P que estão em W. É importante notar que a árvore principal de T orienta uma busca no eixo x e cada estrutura associada orienta uma busca sobre o eixo y. Realizamos uma W-consulta sobre P da seguinte forma. Primeiramente, encontramos o nó mais alto de T, v_{div}, tal que $w_1 \leq_x p(v_{div}) <_x w_2$. Isso é feito chamando EncontraNóDivide($raiz(T)$, W). Uma pequena alteração deve ser feita neste algoritmo. Os símbolos $\leq$ e $>$ devem ser trocados por $\leq_x$ e $>_x$. Assim, $p(v_{div})$ está na área hachurada mostrada na Figura 2.11.

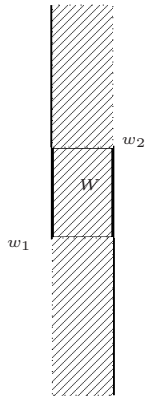


Figura 2.11: As buscas sobre o eixo x encontram pontos na área hachurada.

Algoritmo 7 ConstroiEstruturaAssociada(vetor V^y)

Entrada: Um conjunto de pontos distintos não vazio P através de um vetor y-ordenado $V^y = [q_1, q_2, \ldots, q_n]$.

Saída: Uma estrutura associada sobre P.

1. Crie um vetor T_y com $2n - 1$ posições.
2. $h \leftarrow \lceil \log(n) \rceil$
3. $l' \leftarrow 2^h - n$
4. $l \leftarrow n - l'$
5. $i \leftarrow 2n - 2$
6. ▷ Nas linhas 7 – 12 construímos as folhas do nível h.
7. **para** $j \leftarrow l - 1$ decrescendo até 0 **faça**
8. $p(T_y[i]) \leftarrow V^y[j]$
9. $p_d(T_y[i]) \leftarrow V^y[j]$
10. Marque $T_y[i]$ como folha.
11. $i \leftarrow i - 1$
12. **fim para**
13. ▷ Nas linhas 14 – 19 construímos as folhas do nível $h - 1$.
14. **para** $j \leftarrow n - 1$ decrescendo até l **faça**
15. $p(T_y[i]) \leftarrow V^y[j]$
16. $p_d(T_y[i]) \leftarrow V^y[j]$
17. Marque $T_y[i]$ como folha.
18. $i \leftarrow i - 1$
19. **fim para**
20. ▷ Nas linhas 21 – 25 construímos os nós internos.
21. **enquanto** $i \geq 0$ **faça**
22. $p(T_y[i]) \leftarrow p_d(T_y[2i + 1])$
23. $p_d(T_y[i]) \leftarrow p_d(T_y[2i + 2])$
24. $i \leftarrow i - 1$
25. **fim enquanto**
26. **devolva** T_y

Em seguida, percorremos as subárvores à esquerda e à direita de v_{div} buscando pelos pontos w_1 e w_2, respectivamente. Vamos falar como obtemos os pontos que estão em W e armazenados nas folhas da subárvore à esquerda de v_{div}. Os pontos armazenados na subárvore à sua direita são obtidos de forma semelhante. Pela construção de T, a subárvore à esquerda de v_{div} armazena pontos de P que são x-(menores ou iguais) a $p(v_{div})$. Logo, eles são x-menores que w_2. Suponha que estamos em um nó v descendente à esquerda de v_{div}. Se $w_1 >_x p(v)$, então nada podemos fazer a não ser continuar a busca na subárvore com raiz em $d(v)$. No entanto, se $w_1 \leq_x p(v)$ então a busca continua na subárvore com raiz em $e(v)$ e $w_1 <_x p(r) <_x w_2$, para todo ponto r armazenado na subárvore com raiz em $d(v)$. Sabemos que $T_y(d(v))$ armazena tais pontos. Logo, de forma simétrica, encontramos em $T_y(d(v))$ o nó mais alto v'_{div} tal que $w_1 \leq_y p(v'_{div}) <_y w_2$ chamando EncontraNóDivide($raiz(T_y(d(v))),\ W$)

com os símbolos $\leq$ e $>$ modificados para $\leq_y$ e $>_y$. Percorremos as subárvores à esquerda e à direita de v'_{div} buscando pelos pontos w_1 e w_2 e encontramos alguns pontos que estão em W. Veja a Figura 2.12.

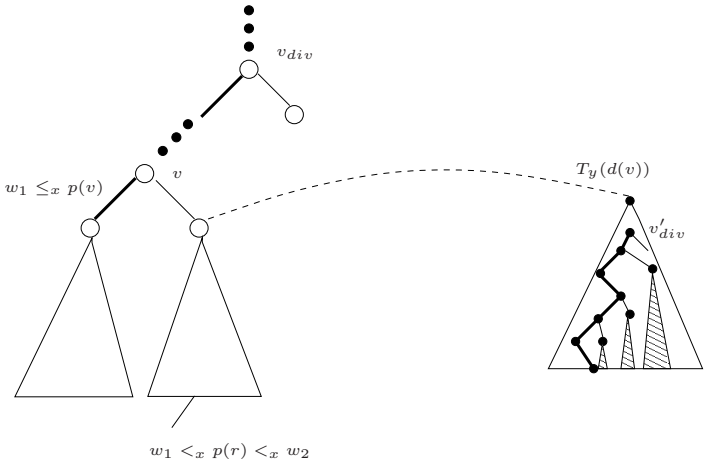


Figura 2.12: Os pontos nas árvores hachuradas estão em W.

As buscas por w_1 e w_2, tanto na árvore principal quanto em uma estrutura associada, continuam até atingirem uma folha f. Por fim, verificamos se $p(f)$ pertence a W. Mais detalhes são encontrados no algoritmo ConsultaÁrvoreLimite. Ele chama o algoritmo ConsultaÁrvoreLimite1D que deve ter os símbolos $\leq$ e $>$ alterados para $\leq_y$, e $>_y$.

Lema 6 **[4, 18, 19, 29]** *O algoritmo* ConsultaÁrvoreLimite *realiza uma W-consulta sobre um conjunto de pontos distintos P em tempo $O(\log^2 n + k)$, onde n é o número de pontos de P e k é o número de pontos de P que estão na janela W.*

Demonstração: Seja T uma árvore limite 2-níveis construída sobre P. Sabemos que consumimos tempo $O(\log n)$ para encontrar o nó v_{div} em T. As buscas por w_1 e w_2 nas subárvores à esquerda e à direita de v_{div} produzem dois caminhos C_x e C'_x com comprimento $O(\log n)$, pois T é balanceada. Para cada nó v de C_x ou C'_x, possivelmente chamamos o algoritmo ConsultaÁrvoreLimite1D passando como parâmetro a estrutura associada do nó $d(v)$ ou $e(v)$. Pelo Lema 4, o consumo de tempo de cada chamada é $O(\log n + k_v)$, onde k_v é o número de pontos armazenados nas subárvores com raiz em $d(v)$ ou $e(v)$ que estão em W. Como a soma dos k_v's é igual a k, segue que o consumo de tempo do algoritmo é $O(\log^2 n + k)$. $\square$

Conseguimos melhorar o consumo de tempo de uma W-consulta sobre um conjunto de pontos distintos P se usarmos uma técnica chamada cascateamento fracionário, tema da nossa próxima seção.

Algoritmo 8 ConsultaÁrvoreLimite(árvore_limite v_{raiz}, janela W)

Entrada: Um conjunto de pontos distintos não vazio P armazenados em uma árvore limite 2-níveis T com raiz no nó v_{raiz} e uma janela W.

Saída: Uma lista L que contém os pontos de P que pertencem a W.

1. $L \leftarrow \emptyset$
2. $v_{div} \leftarrow$ EncontraNóDivide(v_{raiz}, W)
3. **se** v_{div} é uma folha **então**
4. ▷ Verifique se o ponto associado à folha v_{div} deve ser armazenado em L.
5. **se** $w_1 \leq_x p(v_{div}) \leq_x w_2$ **e** $w_1 \leq_y p(v_{div}) \leq_y w_2$ **então**
6. $L \leftarrow \{p(v_{div})\}$
7. **fim se**
8. **senão**
9. ▷ Ande no caminho C_x.
10. $v \leftarrow e(v_{div})$
11. **enquanto** v não é uma folha **faça**
12. **se** $w_1 \leq_x p(v)$ **então**
13. ▷ Verifique a y-coordenada dos pontos armazenados na subárvore com raiz em $T_y(d(v))$.
14. $L \leftarrow L \cup$ ConsultaÁrvoreLimite1D($T_y(d(v))$, W)
15. $v \leftarrow e(v)$
16. **senão**
17. $v \leftarrow d(v)$
18. **fim se**
19. **fim enquanto**
20. ▷ Verifique se o ponto associado à folha v deve ser armazenado em L.
21. **se** $w_1 \leq_x p(v) \leq_x w_2$ **e** $w_1 \leq_y p(v) \leq_y w_2$ **então**
22. $L \leftarrow L \cup \{p(v)\}$
23. **fim se**
24. ▷ Ande no caminho C'_x.
25. $v \leftarrow d(v_{div})$
26. **enquanto** v não é uma folha **faça**
27. **se** $w_2 >_x p(v)$ **então**
28. ▷ Verifique a y-coordenada dos pontos armazenados na subárvore com raiz em $T_y(e(v))$.
29. $L \leftarrow L \cup$ ConsultaÁrvoreLimite1D($T_y(e(v))$, W)
30. $v \leftarrow d(v)$
31. **senão**
32. $v \leftarrow e(v)$
33. **fim se**
34. **fim enquanto**
35. ▷ Verifique se o ponto associado à folha v deve ser armazenado em L.
36. **se** $w_1 \leq_x p(v) \leq_x w_2$ **e** $w_1 \leq_y p(v) \leq_y w_2$ **então**
37. $L \leftarrow L \cup \{p(v)\}$
38. **fim se**
39. **fim se**
40. **devolva** L

2.3 Cascateamento fracionário

A técnica que vamos estudar nesta seção, cascateamento fracionário, criada independentemente por Lueker [19] e Willard [28], reduz o consumo de tempo gasto para resolver uma W-consulta sobre um conjunto de pontos distintos P com n pontos de $O(\log^2 n + k)$ para $O(\log n + k)$, onde k é o número de pontos de P em W. A principal modificação a ser feita é na estrutura de dados árvore limite porém, antes de falar sobre esta modificação, exemplificamos a ideia dessa técnica na reta e depois estendemos para o plano.

Sejam P_1 e P_2 dois conjuntos de pontos na reta tais que P_2 é um subconjunto de P_1. Suponha que os elementos de P_1 e P_2 estão armazenados em vetores ordenados, V_1 e V_2. Considere que verificamos os pontos de V_1 que estão em uma dada janela W. Ou seja, encontramos o menor ponto armazenado em um nó v_1 de V_1 tal que $p(v_1) \geq w_1$ e listamos os pontos até encontrar um nó v_1' com $p(v_1') > w_2$. Se quisermos agora obter os pontos de V_2 nesta mesma janela temos de repetir o mesmo processo, executando uma busca binária no conjunto. Entretanto, com o uso da técnica podemos eliminar esta segunda busca. Considere que cada nó v_1 de V_1 aponta para um nó v_2 de V_2 tal que o ponto $p(v_2)$ é o menor ponto de P_2 com $p(v_2) \geq p(v_1)$. Veja a Figura 2.13.

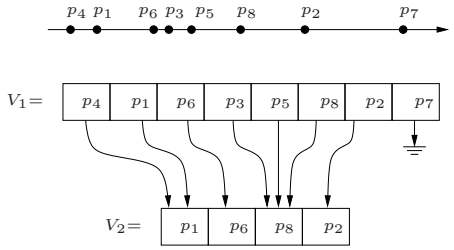


Figura 2.13: P_1 e P_2 são dois conjuntos de pontos na reta e P_2 é um subconjunto de P_1. Eles são representados através de V_1 e V_2. Cada nó v_1 de V_1 aponta para um nó v_2 de V_2 que armazena o menor ponto de P_2 tal que $p(v_2) \geq p(v_1)$.

Agora, de posse do resultado v_1 da busca binária em V_1, pelo limite inferior da janela podemos obter os elementos em V_2 da seguinte forma. Seja v_2 o nó em V_2 apontado por v_1. Sabemos que $p(v_2)$ é o menor ponto em P_2, maior ou igual a $p(v_1)$. Como P_2 é um subconjunto de P_1, temos que $p(v_2)$ é o menor ponto em P_2, maior ou igual a w_1. Logo, os elementos de V_2 que estão em W são listados percorrendo V_2 a partir de v_2 e assim, obtemos os pontos de P_2 que estão em W aproveitando o resultado de uma única busca binária. Vamos mostrar em seguida como a ideia pode ser aplicada em nosso problema.

Na seção 2.2, definimos uma árvore limite 2-níveis como uma estrutura multinível. Temos uma árvore de busca binária balanceada no primeiro e no segundo nível. A modificação na estrutura a ser feita está no segundo nível. Trocamos cada árvore de busca binária balanceada do segundo nível por um vetor y-ordenado. Suponha que T seja uma árvore limite. Seja v um nó de T. Lembre que definimos $P(v)$ como sendo o conjunto de pontos armazenados

na subárvore com raiz em v. Agora, $T_y(v)$ é um vetor que armazena os pontos de $P(v)$ y-ordenados. Um nó w de $T_y(v)$ possui como membros um ponto e dois ponteiros, $pt_e(w)$ e $pt_d(w)$. O ponteiro $pt_e(w)$ aponta para um nó w_e de $T_y(e(v))$ que armazena o menor ponto de $P(e(v))$ tal que $p(w_e) \geq_y p(w)$. Da mesma forma, o ponteiro $pt_d(w)$ aponta para um nó w_d de $T_y(d(v))$ que armazena o menor ponto de $P(d(v))$ tal que $p(w_d) \geq_y p(w)$. A Figura 2.14 ilustra nossa nova estrutura.

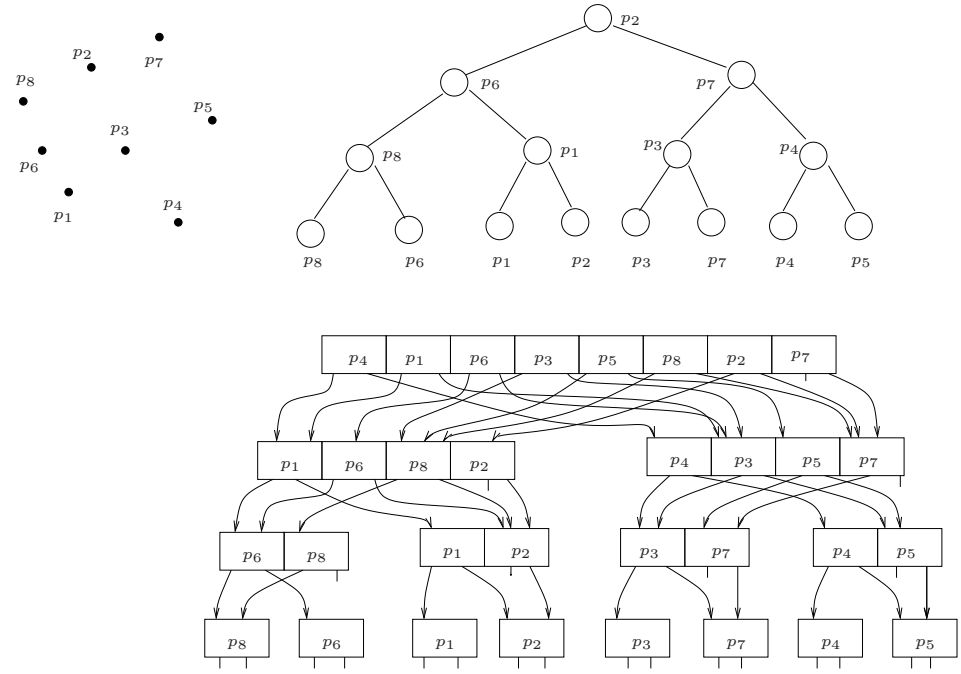


Figura 2.14: Árvore limite com camadas.

Essa nova estrutura é conhecida como árvore limite com camadas. Sua construção, feita pelo algoritmo ConstroiÁrvoreLimiteComCamadas, é similar à construção da árvore limite 2-níveis. A diferença está relacionada à estrutura associada de cada nó v da árvore principal, ou seja, $T_y(v)$. Agora, $T_y(v)$ é um vetor y-ordenado que armazena os pontos de $P(v)$. Note no algoritmo que V^y é um vetor com tal característica e assim, o associamos a $T_y(v)$ (linha 2 do algoritmo). A partir do momento que temos V_e^y e V_d^y, podemos definir os ponteiros de V^y para V_e^y e V_d^y. Isso ocorre na chamada do algoritmo CriaPonteiros. Observe que, se este algoritmo consumir tempo linear, então o algoritmo ConstroiÁrvoreLimiteComCamadas cosumirá tempo $\Theta(n \log n)$.

Vamos descrever o algoritmo CriaPonteiros de tal forma que seu consumo de tempo seja linear. Este algoritmo recebe os vetores V^y, V_e^y e V_d^y e devolve V^y com ponteiros para V_e^y e V_d^y. Lembre que os membros de cada nó v destes vetores são $p(v)$, $pt_e(v)$ e $pt_d(v)$. No início do algoritmo somente $p(v)$ está definido. No final, cada nó v de V^y possui definido

Algoritmo 9 ConstroiÁrvoreLimiteComCamadas(vetor V^x, vetor V^y)

Entrada: Um conjunto de pontos distintos não vazio P através de dois vetores: um x-ordenado $V^x = [p_1, p_2, \ldots, p_n]$ e outro y-ordenado $V^y = [q_1, q_2, \ldots, q_n]$.

Saída: A raiz v de uma árvore limite com camadas.

1. Crie um nó v.
2. $T_y(v) \leftarrow V^y$.
3. Crie quatro vetores vazios: V_e^x, V_d^x, V_e^y e V_d^y
4. $V_e^x \leftarrow [p_1, p_2, \ldots, p_{\lceil \frac{n}{2} \rceil}]$.
5. $V_d^x \leftarrow [p_{\lceil \frac{n}{2} \rceil + 1}, p_{\lceil \frac{n}{2} \rceil + 2}, \ldots, p_n]$.
6. **para** $i \leftarrow 1$ até n **faça**
7. **se** $q_i \leq_x p_{\lceil \frac{n}{2} \rceil}$ **então**
8. Insira no final do vetor V_e^y o ponto q_i.
9. **senão**
10. Insira no final do vetor V_d^y o ponto q_i.
11. **fim se**
12. **fim para**
13. $V^y \leftarrow$ CriaPonteiros(V^y, V_e^y, V_d^y)
14. $p(v) \leftarrow p_{\lceil \frac{n}{2} \rceil}$
15. **se** V^x contém um único ponto **então**
16. Marque v como folha.
17. **senão**
18. $e(v) \leftarrow$ ConstroiÁrvoreLimiteComCamadas(V_e^x, V_e^y)
19. $d(v) \leftarrow$ ConstroiÁrvoreLimiteComCamadas(V_d^x, V_d^y)
20. **fim se**
21. **devolva** v

$p(v)$, $pt_e(v)$ apontando para um nó de V_e^y e $pt_d(v)$ apontando para um nó de V_d^y.

Primeiramente, observe que V_e^y e V_d^y formam uma partição de V^y, isto é, o conjunto dos pontos armazenados em V^y ($P(V^y)$) é formado pela união dos conjuntos dos pontos armazenados em V_e^y ($P(V_e^y)$) e V_d^y ($P(V_d^y)$). Sejam q, q_e e q_d os menores pontos de $P(V^y)$, $P(V_e^y)$ e $P(V_d^y)$. Pela y-ordenação dos vetores, encontramos tais pontos em tempo constante. Pela partição, temos duas possibilidades:

- $q = q_e$ e q_d é o menor ponto em $P(V_d^y)$ y-maior que q; ou
- $q = q_d$ e q_e é o menor ponto em $P(V_e^y)$ y-maior que q,

e assim, descrevemos como definir os ponteiros do nó de V^y que armazena q.

Note que uma nova partição pode ser formada:

$$P(V^y) - \{q\} = \begin{cases} P(V_e^y) - \{q\} \cup P(V_d^y), & \text{se} \quad q \in P(V_e^y) \\ P(V_e^y) \cup P(V_d^y) - \{q\}, & \text{se} \quad q \in P(V_d^y). \end{cases}$$

Repetindo este processo $n = |P(V^y)|$ vezes, criamos os ponteiros para cada nó de V^y em tempo $\Theta(n)$.

Uma abordagem de tal estratégia que trabalha com índices sobre os vetores V^y, V_e^y e V_d^y pode ser vista formalmente no algoritmo CriaPonteiros.

Algoritmo 10 CriaPonteiros(vetor V, vetor V_e, vetor V_d)

Entrada: Três vetores y-ordenados, V, V_e e V_d com tamanhos n, n_e e n_d, respectivamente. Os vetores V_e e V_d são disjuntos e formam uma partição de V.
Saída: O vetor V com os ponteiros pt_e e pt_d definidos.

1. $i_e \leftarrow 0$, $i_d \leftarrow 0$, $i \leftarrow 0$
2. **enquanto** $i < n$ **faça**
3. **se** $i_e < n_e$ **então**
4. $pt_e(V[i])$ aponta para o nó $V_e[i_e]$.
5. **senão**
6. $pt_e(V[i])$ não aponta para nenhum nó.
7. **fim se**
8. **se** $i_d < n_d$ **então**
9. $pt_d(V[i])$ aponta para o nó $V_d[i_d]$.
10. **senão**
11. $pt_d(V[i])$ não aponta para nenhum nó.
12. **fim se**
13. **se** $i_e < n_e$ **e** $p(V[i]) = p(V_e[i_e])$ **então**
14. $i_e \leftarrow i_e + 1$
15. **senão**
16. $i_d \leftarrow i_d + 1$
17. **fim se**
18. $i \leftarrow i + 1$
19. **fim enquanto**
20. **devolva** V

Lema 7 [19,28] *O algoritmo* ConstroiÁrvoreLimiteComCamadas *constroi uma árvore limite com camadas sobre um conjunto não vazio de pontos distintos P com cardinalidade igual a n usando dois vetores ordenados em tempo $\Theta(n \log n)$.* □

Terminada a fase de pré-processamento, teremos uma árvore limite com camadas T sobre P. Uma W-consulta sobre P é realizada da seguinte forma. Procure em T o nó v_{div} como anteriormente. No vetor $T_y(v_{div})$ busque por v'_{div} tal que $p(v'_{div})$ é o menor ponto y-(maior ou igual) a w_1. Com uma busca binária em $T_y(v_{div})$ encontramos tal nó consumindo tempo $O(\log n)$. Novamente, ao buscar por w_1 e por w_2 nas subárvores à esquerda e à direita de v_{div}, respectivamente, encontramos dois caminhos C e C' com $O(\log n)$ nós cada. Para cada nó v de C e C', obtemos em tempo constante o menor ponto y-(maior ou igual) a w_1 em $T_y(v)$ seguindo os ponteiros. Mais uma vez, vamos falar da resolução de uma W-consulta

sobre P olhando somente para o caminho C. Podemos encontrar os pontos que estão em W e nas subárvores cuja raiz é um nó de C' de maneira simétrica. Então, suponha que v é um nó de C e v' é o menor ponto y-(maior ou igual) a w_1 em $T_y(v)$. Temos duas possibilidades. Se $w_1 >_x p(v)$, então continuamos a busca na subárvore com raiz em $d(v)$ e seguimos para o nó apontado por $pt_d(v')$ em $T_y(d(v))$. Se $w_1 \leq_x p(v)$, então listamos todos os pontos y-(menores ou iguais) a w_2 armazenados no vetor $T_y(d(v))$ a partir do nó apontado por $pt_d(v')$, continuamos a busca na subárvore com raiz em $e(v)$ e seguimos para o nó apontado por $pt_e(v')$ em $T_y(e(v))$. Veja a Figura 2.15. Observe que não é necessária nenhuma busca adicional na estrutura associada, mas apenas seguir no vetor até encontrar um ponto que seja maior que w_2. Veja o algoritmo ConsultaÁrvoreLimiteComCamadas para melhor esclarecimento.

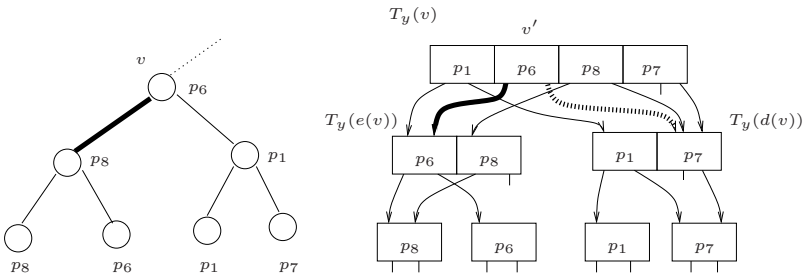


Figura 2.15: $w_1 \leq_x p(v)$. A busca continua na subárvore com raiz em $e(v)$, mantemos o menor ponto y-(maior ou igual) a w_1 armazenado no vetor $T_y(e(v))$ seguindo o apontador $pt_e(v')$ e listamos todos os pontos em $T_y(d(v))$ y-(menores ou iguais) a w_2, a partir do nó apontado por $pt_d(v')$.

Lema 8 **[19, 28]** *Seja P um conjunto com n pontos distintos. Seja T uma árvore limite com camadas sobre P. Uma W-consulta sobre P usando T consome tempo $O(\log n + k)$, onde k é o número de pontos de P em W.*

Demonstração: Novamente, consumimos tempo $O(\log n)$ para encontrar v_{div} em T. Também consumimos esse tempo para encontrar v'_{div} em $T_y(v_{div})$. As buscas por w_1 e w_2 nas subárvores à esquerda e à direita de v_{div} produzem dois caminhos, C e C' com $O(\log n)$ nós cada. Para cada nó v de C ou C', consumimos tempo proporcional ao número de pontos (k_v) armazenados na subárvore com raiz em $d(v)$ ou $e(v)$ que estão em W. Este consumo é obtido porque mantemos o menor ponto em $T_y(v)$ y-(maior ou igual) a w_1 através de ponteiros. A soma de todos os k_v's pontos é igual a k pontos e assim, segue que o consumo de tempo de uma W-consulta é $O(\log n + k)$. □

O que podemos dizer sobre o consumo de espaço de uma árvore limite com camadas? Seja T uma árvore limite com camadas construída sobre um conjunto P com n pontos. Sabemos que o número total de nós da árvore principal de T é $N = 2n - 1$ (seção 2.1). Observe que cada nível i da árvore apresenta, em seus y-vetores, uma partição de P. Como sabemos, a altura de T é $O(\log n)$ e portanto o consumo total de espaço da árvore limite com camadas é $\Theta(n \log n)$. Com isso, segue o próximo lema.

Algoritmo 11 ConsultaÁrvoreLimiteComCamadas(árvore_limite_camadas v_{raiz}, janela W)

Entrada: Um conjunto de pontos distintos não vazio P armazenados em uma árvore limite com camadas T com raiz no nó v_{raiz} e uma janela W.

Saída: Uma lista L que contém os pontos de P que pertencem a W.

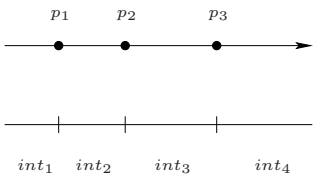
1. $L \leftarrow \emptyset$
2. $v_{div} \leftarrow$ EncontraNóDivide(v_{raiz}, W)
3. **se** v_{div} é uma folha **então**
4. **se** $w_1 \leq_x p(v_{div}) \leq_x w_2$ **e** $w_1 \leq_y p(v_{div}) \leq_y w_2$ **então**
5. $L \leftarrow \{p(v_{div})\}$
6. **fim se**
7. **senão**
8. ▷ Encontre o menor ponto em $T_y(v_{div})$ y-(maior ou igual) a w_1 realizando uma busca binária.
9. $v'_{div} \leftarrow$ BuscaBinária($T_y(v_{div})$, w_1)
10. **se** $v'_{div} \neq NULO$ **então**
11. ▷ Ande no caminho C.
12. $v \leftarrow e(v_{div})$
13. $v' \leftarrow pt_e(v'_{div})$
14. ▷ Se $v' = NULO$, então não existe $q \in P(v)$ tal que $q \geq_y w_1$.
15. **enquanto** v não é uma folha **e** $v' \neq NULO$ **faça**
16. **se** $w_1 \leq_x p(v)$ **então**
17. $u \leftarrow pt_d(v')$
18. **enquanto** u é um nó do vetor $T_y(d(v))$ **e** $p(u) \leq_y w_2$ **faça**
19. $L \leftarrow L \cup \{p(u)\}$
20. $u \leftarrow u + 1$ ▷ Vá para o próximo nó do vetor.
21. **fim enquanto**
22. $v \leftarrow e(v)$
23. $v' \leftarrow pt_e(v')$
24. **senão**
25. $v \leftarrow d(v)$
26. $v' \leftarrow pt_d(v')$
27. **fim se**
28. **fim enquanto**
29. **se** $v' \neq NULO$ **e** $(w_1 \leq_x p(v) \leq_x w_2$ **e** $w_1 \leq_y p(v) \leq_y w_2)$ **então**
30. $L \leftarrow L \cup \{p(v)\}$
31. **fim se**
32. ▷ De forma simétrica encontre os pontos que estão armazenados na subárvore à direita de v_{div} andando em C'.
33. **fim se**
34. **fim se**
35. **devolva** L

Lema 9 **[19,28]** *Uma árvore limite com camadas construída sobre um conjunto com n pontos no plano consome espaço* $\Theta(n \log n)$. □

2.4 Cota inferior

A estratégia adotada nesta seção consiste em mostrar um conjunto de pontos no plano cujo número de W-consultas distintas possíveis sobre ele é grande. Duas W-consultas são distintas se, e somente se, elas devolvem diferentes conjuntos de pontos. Sabemos que uma W-consulta sobre um conjunto com n pontos no plano pode ser respondida em tempo $O(\log n + k)$, onde k é o número de pontos em W. Como em Preparata e Shamos [22], dividimos este consumo de tempo em dois: consumo de tempo da busca pelos pontos que estão em W - $O(\log n)$ - e consumo de tempo da listagem dos pontos que estão em W - $O(k)$. Uma cota inferior para o segundo consumo de tempo é $\Omega(k)$, pois, k pontos devem ser listados. Uma cota inferior para o primeiro consumo é estudada a seguir.

Seja $P = \{p_1, p_2, \ldots, p_n\}$ um conjunto de pontos distintos na reta. Sejam $int_1 = (-\infty : p_1)$, $int_2 = (p_1 : p_2)$, $\ldots$, $int_n = (p_{n-1} : p_n)$ e $int_{n+1} = (p_n : +\infty)$, $n+1$ intervalos abertos na reta formados pelos pontos de P. Afirmamos que o conjunto P admite $\binom{n+1}{2} + 1$ W-consultas distintas. Isso porque a resposta para cada W-consulta é ou o conjunto vazio, ou um conjunto de pontos que estão em uma janela defina por dois pontos localizados em dois intervalos distintos dos $n+1$ intervalos abertos. Por exemplo, na Figura 2.16 temos três pontos p_1, p_2 e p_3, e quatro intervalos abertos $int_1 = (-\infty : p_1)$, $int_2 = (p_1 : p_2)$, $int_3 = (p_2 : p_3)$ e $int_4 = (p_3 : +\infty)$. Combinando dois pontos localizados em dois dos quatro intervalos definimos seis janelas distintas cuja resposta é diferente do conjunto vazio. Ao considerar uma janela cuja resposta é o conjunto vazio temos o número de janelas distintas desejado.



$q_1 \in$	int_1	int_1	int_1	int_2	int_2	int_3
$q_2 \in$	int_2	int_3	int_4	int_3	int_4	int_4
$W = [q_1 : q_2]$	$\{p_1\}$	$\{p_1, p_2\}$	$\{p_1, p_2, p_3\}$	$\{p_2\}$	$\{p_2, p_3\}$	$\{p_3\}$

Figura 2.16: Resposta para cada W-consulta distinta formada pelos intervalos. A W-consulta cuja resposta é o conjunto vazio não está na tabela.

Observe que dado um conjunto de pontos P na reta, o número de janelas distintas que P admite depende somente do seu tamanho. Isso não ocorre no plano. Se, por exemplo, todos os n pontos no plano estiverem na mesma reta, o número de janelas distintas será o mesmo que o caso anterior. Veremos no próximo teorema que, se P é um conjunto no plano, então o número de janelas distintas pode ser bem grande dependendo da disposição de seus pontos.

Teorema 10 **[25]** *Dado n pontos no plano, o número de subconjuntos desses pontos que são respostas para alguma W-consultas é pelo menos $\frac{n^4}{64}$.*

Demonstração: Considere a Figura 2.17.

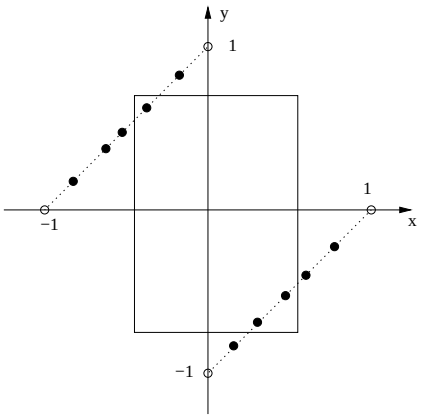


Figura 2.17: Um conjunto de pontos no plano.

Sejam $s_1 = \overline{(1,0)(0,-1)}$ e $s_2 = \overline{(0,1)(-1,0)}$ segmentos de retas abertas nas extremidades, isto é, os pontos $(1,0)$ e $(0,-1)$ não pertencem a s_1 e os pontos $(0,1)$ e $(-1,0)$ não pertencem a s_2. Seja P um conjunto com n pontos no plano, sendo que cada um deles pertencem ou ao segmento de reta s_1 ou ao segmento de reta s_2. Considere que o número de pontos em s_1 é $\lceil n/2 \rceil$ e em s_2 é $\lfloor n/2 \rfloor$. Sejam Q e Q' os conjuntos formados pelas janelas distintas no quadrante inferior direito e superior esquerdo, respectivamente. Observe que podemos encontrar como anteriormente o número de janelas distintas em Q e Q'. Com isso, $|Q| = \binom{\lceil n/2 \rceil + 1}{2} + 1$ e $|Q'| = \binom{\lfloor n/2 \rfloor + 1}{2} + 1$. Perceba também que, pela disposição dos pontos de P, podemos combinar cada janela W_Q de Q com cada janela $W_{Q'}$ de Q' formando uma nova janela W cuja solução da W-consulta é a união da solução da W_Q-consulta e da $W_{Q'}$-consulta. Assim, o número total de janela distintas formadas por P é $|Q||Q'|$, ou seja:

$$\left[\binom{\lceil n/2 \rceil + 1}{2} + 1\right]\left[\binom{\lfloor n/2 \rfloor + 1}{2} + 1\right] \geq \left[\frac{(\lceil n/2 \rceil + 1)!}{2(\lceil n/2 \rceil - 1)!}\right]\left[\frac{(\lfloor n/2 \rfloor + 1)!}{2(\lfloor n/2 \rfloor - 1)!}\right] =$$

$$= \left[\frac{(\lceil n/2 \rceil + 1)(\lceil n/2 \rceil)(\lceil n/2 \rceil - 1)!}{2(\lceil n/2 \rceil - 1)!}\right]\left[\frac{(\lfloor n/2 \rfloor + 1)(\lfloor n/2 \rfloor)(\lfloor n/2 \rfloor - 1)!}{2(\lfloor n/2 \rfloor - 1)!}\right] =$$

$$= \left[\frac{(\lceil n/2 \rceil + 1)(\lceil n/2 \rceil)}{2}\right]\left[\frac{(\lfloor n/2 \rfloor + 1)(\lfloor n/2 \rfloor)}{2}\right] \geq \left[\frac{(\lfloor n/2 \rfloor^2 + \lfloor n/2 \rfloor)^2}{4}\right] =$$

$$= \frac{\lfloor n/2 \rfloor^4 + 2\lfloor n/2 \rfloor^3 + \lfloor n/2 \rfloor^2}{4} \geq \frac{\lfloor n/2 \rfloor^4}{4} \cong \frac{n^4}{64}.$$

□

Qualquer algorimo correto que busca pontos em uma janela, deve ser capaz de encontrar todos os $\frac{n^4}{64}$ conjuntos de pontos da Figura 2.17. Podemos visualizar a estrutura da execução de um algoritmo qualquer como uma árvore. Em cada nó da árvore há comparações e as folhas representam cada um dos $\frac{n^4}{64}$ conjuntos de pontos. Preparata e Shamos [22] chamam esta estrutura de árvore de decisão algébrica e este é um importante modelo computacional [24], [23] e [10]. Só faz sentido usar este modelo se o problema em questão for de decisão. Então vamos desviar por um momento o nosso foco para definirmos um problema de decisão correspondente ao nosso problema. Sejam P um conjunto não vazio de pontos no plano, W uma janela no plano e T uma árvore limite com camadas que armazena os pontos de P. Seja k um inteiro positivo. Existem pelo menos k pontos de P em W? Podemos resolver este problema encontrando todos os pontos que pertencem a W usando o algoritmo ConsultaÁrvoreLimiteComCamadas que devolve um subconjunto P' de P e de W e, em seguida, consumimos tempo constante para responder sim, caso $k \leq |P'|$ ou não, caso contrário. Agora, vamos voltar à árvore de decisão algébrica T_a definida essencialmente pela execução de qualquer algoritmo para o problema. O consumo deste algoritmo é pelo menos proporcional ao maior caminho da raiz até uma folha em T_a, ou seja, proporcional à altura de T_a. Suponha que T_a é binária e que sua altura seja h. Então, T_a possui no máximo 2^h folhas. Logo,

$$\begin{array}{rcll} 2^h & \geq & \frac{n^4}{64} & \\ \log 2^h & \geq & \log(\frac{n^4}{64}) & \\ h & \geq & \log n^4 - \log 64 & \\ h & \geq & 4\log n - \log n, & \text{para } n \geq 64 \\ h & \geq & 3\log n & \\ h & = & \Omega(\log n). & \end{array}$$

Com isso, segue o próximo teorema.

Teorema 11 **[22]** *Seja P um conjunto não vazio com n pontos distintos no plano. Em um modelo de árvore de decisão algébrica, o número de comparações realizadas para listar pontos que estão em uma janela é $\Omega(\log n)$ no pior caso.* □

O que podemos falar sobre cota inferior no consumo de espaço de uma estrutura que ajuda a resolver uma W-consulta? Uma cota inferior direta do consumo de espaço de uma estrutura de dados que armazena um conjunto com n pontos é $\Omega(n)$. Para um modelo de computação diferente definido em [26] e conhecida como máquina ponteiro (*pointer machine*), Chazelle [5] mostrou que, se uma estrutura de dados encontra k pontos em uma janela no plano em tempo $O(\log n + k)$, então ela consome espaço $\Omega(n(\log n/\log\log n))$. Note que a cota inferior de Chazelle é mais forte que a nossa cota inferior. Um fato interessante é, ao criar uma restrição em uma janela W podemos usar uma estrutura de dados que resolve uma W-consulta consumindo espaço $O(n)$. Veremos como isso é possível nas duas próximas seções.

2.5 Pontos em janelas ilimitadas - unidimensional

Seja P um conjunto com n pontos distintos sobre a reta. Seja w um ponto na reta. Uma janela ilimitada W^+ na reta é formada por um intervalo aberto $[w : +\infty)$. Da mesma forma, uma janela W^- é formada por um intervalo aberto $(-\infty : w]$. Os algoritmos que vamos estudar encontram pontos que pertencem a uma janela W^-. Algoritmos que encontram pontos em uma janela W^+ são simétricos. Vamos omitir a fase de pré-processamento pois ela se resume em construir um *minheap* T sobre os pontos de P. Isso pode ser encontrado em [6]. Na fase de consultas, precisamos buscar pontos que pertencem à janela ilimitada. Para isso, verificamos o ponto armazenado na raiz v de T, $p_{min}(v)$. Se $p_{min}(v) \leq w$ então $p_{min}(v)$ pertence a W^- e, recursivamente, verificamos os pontos armazenados nos dois filhos de v. Se $p_{min}(v) > w$ então $p_{min}(v)$ não está em W^- e os pontos armazenados em seus filhos $p_{min}(e(v))$ e $p_{min}(d(v))$ também não estão, pois, pela propriedade de T, $p_{min}(e(v))$ e $p_{min}(d(v))$ são maiores que $p_{min}(v)$. O algoritmo PontosMinheap recebe como parâmetro um *minheap* T construído sobre um conjunto de pontos distintos na reta P e uma janela W^- e devolve os pontos de P que pertencem a W^-.

Algoritmo 12 PontosMinheap(minheap v, janela_ilimitada W^-)

Entrada: Um *minheap* com raiz em v construído sobre um conjunto de pontos distintos na reta P e uma janela ilimitada W^-.

Saída: Uma lista L que contém os pontos de P que pertencem a $W^- = (-\infty : w]$.

1. $L \leftarrow \emptyset$
2. **se** v é diferente de $NULO$ **então**
3. **se** $p(v) \leq w$ **então**
4. $L \leftarrow \{p(v)\} \cup$ PontosMinheap$(e(v),\ W^-) \cup$ PontosMinheap$(d(v),\ W^-)$
5. **fim se**
6. **fim se**
7. **devolva** L

Lema 12 *Seja P um conjunto de pontos distintos na reta. O consumo de tempo do algoritmo* PontosMinheap *é $O(k)$, onde k é o número de pontos de P que pertencem a W^-.*

Demonstração: Seja T um *minheap* construído sobre P. Seja r o número de nós de T visitados pelo algoritmo. Sabemos que $r = k + l$, onde k é o número de nós visitados ($V = \{v_1,\ v_2,\ \ldots,\ v_k\}$) que armazenam pontos que pertencem a W^- e l é o número de nós visitados ($U = \{u_1,\ u_2,\ \ldots,\ u_l\}$) que armazenam os pontos que não pertencem a W^-. Sabemos que não existe nó u_i, com $1 \leq i \leq l$, tal que $u_i = e(u_j)$ ou $u_i = d(u_j)$ para algum u_j, com $1 \leq j \leq l$ e $j \neq i$. Isso é verdade pois, para qualquer u_i, $1 \leq i \leq l$, $p(u_i) > w_2$ e assim, o algoritmo não visita nenhum filho de u_i. Então, para cada elemento $u_i \in U$ existe um elemento $v_j \in V$ tal que $u_i = e(v_j)$ ou $u_i = d(v_j)$. Portanto, $l \leq 2k$. Assim, $r \leq 3k$. Como em cada nó consumimos tempo constante (no máximo duas comparações e duas operações de união que podemos realizar como concatenação de listas ligadas) o algoritmo PontosMinheap consome tempo $O(k)$. □

Até agora mostramos como armazenar os elementos de um conjunto de pontos distintos na reta de tal forma que uma W^--consulta seja resolvida rapidamente. Na próxima seção, vamos estender a ideia aqui estudada para o plano.

2.6 Pontos em janelas ilimitadas - bidimensional

Seja P um conjunto de pontos distintos no plano. Sejam p e p' pontos de P com menor e maior coordenada x e q e q' pontos de P com menor e maior coordenada y, respectivamente. Dizemos que os valores $x(p) - 1$ e $y(q) - 1$ fazem o papel de $-\infty$ no eixo x e no eixo y e que $x(p') + 1$ e $y(q') + 1$ fazem o papel de $+\infty$ no eixo x e no eixo y. Sejam $w_1 = (x, y)$ e $w_2 = (x', y')$ pontos no plano onde $x < x'$ e $y < y'$. Uma janela ilimitada no plano é formada pelos segmentos de reta de um retângulo "infinito" e sua região interna, similar à definição na seção 2.2, porém, uma, e somente uma, das seguintes alternativas ocorre: $x = x(p) - 1$ ou $x' = x(p') + 1$ ou $y = y(q) - 1$ ou $y' = y(q') + 1$. Vamos considerar nos algoritmos estudados, somente uma janela ilimitada W^- com $x = x(p) - 1$. Os algoritmos que encontram pontos em janelas com $x' = x(p') + 1$ ou $y = y(q) - 1$ ou $y' = y(q') + 1$ são simétricos. Em alguns momentos abusamos da linguagem e tratamos, por exemplo, $w_1 = (-\infty, y)$ como um ponto.

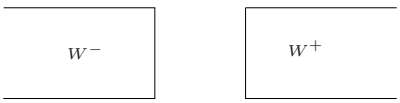


Figura 2.18: Duas janelas ilimitadas: W^- com $w_1 = (-\infty, y)$ e $w_2 = (x', y')$ e W^+ com $w_1 = (x, y)$ e $w_2 = (+\infty, y')$.

Em seguida falaremos sobre a fase de pré-processamento onde mostramos como armazenar pontos de P em uma estrutura de dados chamada árvore de busca com prioridade [21]. Um nó v de uma árvore de busca com prioridade T possui quatro membros: seus dois filhos e dois pontos de P, $p(v)$ e $p_{min}(v)$. O ponto $p(v)$ é usado para orientar uma busca em T e o ponto $p_{min}(v)$ é usado para encontrar os pontos de P que pertencem a uma janela ilimitada. Uma propriedade de um *minheap* é mantida em T através do ponto $p_{min}(v)$. Para todo nó v de T temos que $p_{min}(v) <_x p_{min}(e(v))$ e $p_{min}(v) <_x p_{min}(d(v))$. Isso justifica o nome da estrutura uma vez que um *heap* pode ser usado para armazenar uma fila de prioridade. Garantimos T balanceada através de sua construção. Primeiramente, criamos um nó v e armazenamos nele o x-menor ponto de P em $p_{min}(v)$. Em $p(v)$ armazenamos um ponto q de $P - \{p_{min}(v)\}$ tal que os conjuntos $P(e(v)) = \{p \in P - \{p_{min}(v)\} : p \leq_y q\}$ e $P(d(v)) = \{p \in P - \{p_{min}(v)\} : p >_y q\}$ tenham tamanhos que diferem de no máximo um elemento. Recursivamente definimos os filhos de v através dos conjuntos $P(e(v))$ e $P(d(v))$. O algoritmo ConstroiMinÁrvoreBuscaPrioridade formaliza a construção descrita acima. Ele recebe um conjunto de pontos distintos no plano P, através de dois vetores: um x-ordenado V^x e outro y-ordenado V^y e devolve uma árvore de busca com prioridade com raiz em v.

Algoritmo 13 ConstroiMinÁrvoreBuscaPrioridade(vetor V^x, vetor V^y)

Entrada: Um conjunto P de pontos distintos através de dois vetores: um x-ordenado $V^x = [p_1,\ p_2,\ \ldots,\ p_n]$ e outro y-ordenado $V^y = [q_1,\ q_2,\ \ldots,\ q_n]$.

Saída: Uma árvore de busca com prioridade T sobre P com raiz em v ou $NULO$ caso $n = 0$.

1. Crie um nó v.
2. **se** $n > 0$ **então**
3. Crie quatro vetores vazios: $V_e^x,\ V_e^y,\ V_d^x,\ V_d^y$.
4. $p_{min}(v) \leftarrow p_1$ ▷ Nesta linha, armazenamos o x-menor ponto de P em v.
5. $d \leftarrow 0$ ▷ Nesta linha, inicializamos uma variável que auxilia a encontrar $p(v)$.
6. ▷ Nas linhas 7 – 18, definimos os conjuntos $P(e(v))$ e $P(d(v))$ ordenados pela y-coordenada através dos vetores V_e^y e V_d^y.
7. **para** $i \leftarrow 1$ até $\lceil \frac{n-1}{2} \rceil + d$ **faça**
8. **se** $q_i \neq p_{min}(v)$ **então**
9. Insira q_i no final de V_e^y.
10. **senão**
11. $d \leftarrow d + 1$
12. **fim se**
13. **fim para**
14. **para** $i \leftarrow \lceil \frac{n-1}{2} \rceil + d + 1$ até n **faça**
15. **se** $q_i \neq p_{min}(v)$ **então**
16. Insira q_i no final de V_d^y.
17. **fim se**
18. **fim para**
19. ▷ Na linha 20, armazenamos um ponto de P que o divide em dois conjuntos com tamanhos quase iguais. Observe que se $n = 1$ então $d = 0$ e $p(v)$ não armazena nada.
20. $p(v) \leftarrow q_{\lceil \frac{n-1}{2} \rceil + d}$
21. ▷ Nas linhas 22 – 28, definimos os conjuntos $P(e(v))$ e $P(d(v))$ ordenados pela x-coordenada através dos vetores V_e^x e V_d^x.
22. **para** $i \leftarrow 2$ até n **faça**
23. **se** $p_i \leq_y p(v)$ **então**
24. Insira p_i no final de V_e^x.
25. **senão**
26. Insira p_i no final de V_d^x.
27. **fim se**
28. **fim para**
29. $e(v) \leftarrow$ ConstroiMinÁrvoreBuscaPrioridade($V_e^x,\ V_e^y$)
30. $d(v) \leftarrow$ ConstroiMinÁrvoreBuscaPrioridade($V_d^x,\ V_d^y$)
31. **senão**
32. $v \leftarrow NULO$
33. **fim se**
34. **devolva** v

Na linha 5 do algoritmo ConstroiMinÁrvoreBuscaPrioridade, inicializamos uma variável chamada d. Ela ajuda a encontrar o ponto $p(v)$ em V^y. Note que, $p(v) = q_{\lceil \frac{n-1}{2} \rceil + 1}$, se $p_{min}(v) = q_i$ para algum i entre 1 e $\lceil \frac{n-1}{2} \rceil$ inclusive, ou $p(v) = q_{\lceil \frac{n-1}{2} \rceil}$, caso contrário. A Figura 2.19 ilustra as duas situações.

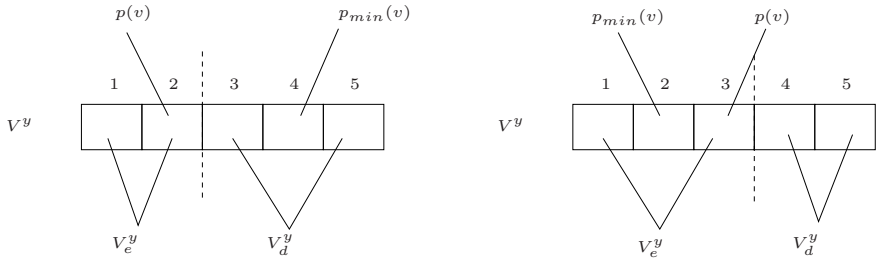


Figura 2.19: No primeiro vetor $p_{min}(v) = q_4$. Com isso, $p(v) = q_{\lceil \frac{n-1}{2} \rceil} = q_2$. No segundo, $p_{min}(v) = q_2$ e $p(v) = q_{\lceil \frac{n-1}{2} \rceil + 1} = q_3$.

Lema 13 **[21]** *O algoritmo* ConstroiMinÁrvoreBuscaPrioridade *constroi uma árvore de busca com prioridade sobre um conjunto P de pontos distintos com cardinalidade n usando dois vetores ordenados em tempo $\Theta(n \log n)$.*

Demonstração: A demonstração segue da solução da seguinte recorrência:

$$T(n) = \begin{cases} \Theta(1), & \text{se} \quad n = 0 \\ T(\lceil \frac{n-1}{2} \rceil) + T(\lfloor \frac{n-1}{2} \rfloor) + \Theta(n), & \text{se} \quad n > 0 \end{cases}$$ □

Lema 14 **[21]** *Uma árvore de busca com prioridade construída sobre um conjunto P de pontos distintos com cardinalidade n consome espaço $\Theta(n)$.*

Demonstração: Qualquer estrutura construída sobre P consome espaço $\Omega(n)$ pois, pelo menos cada pontos de P deve ser armazenado. Agora vamos mostrar que uma árvore de busca com prioridade T consome espaço $O(n)$ e assim, segue o lema.

Em cada chamada do algoritmo ConstroiMinÁrvoreBuscaPrioridade criamos um nó de T e diminuímos de um o número de elementos de P. Como P possui n elementos, então teremos $n+r$ chamadas ao algoritmo, onde r é o número de chamadas feitas por folhas. Logo teremos $n + r$ nós, porém, r nós não armazenam nada. Logo, T possui n nós que consomem no total espaço linear (cada um destes n nós armazena dois pontos). Portanto, o consumo de espaço de T é $O(n)$. □

Diferente de uma árvore limite onde cada nó interno possui dois filhos, um nó interno v de uma árvore de busca com prioridade pode possuir somente uma folha como filho à

esquerda. Chamamos este nó de "quase" folha. Para verificar isso, basta simular o algoritmo ConstroiMinÁrvoreBuscaPrioridade considerando um conjunto P com 2 elementos.

Agora, vamos à fase de consultas. Seja P um conjunto não vazio com n pontos distintos no plano, e W^- uma janela ilimitada. Considere que construímos uma árvore de busca com prioridade T sobre P. Lembramos que a árvore de busca com prioridade está organizada com base na y-coordenada dos pontos, e, além disso, cada nó da árvore armazena um ponto: o de x-coordenada mínima. Inicialmente encontramos v_{div}, o nó mais alto da árvore tal que $w_1 \leq_y p(v_{div}) <_y w_2$. Assim, encontramos um ponto na área hachurada da Figura 2.20.

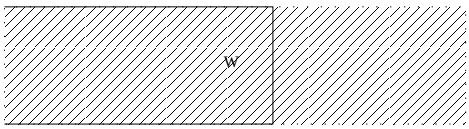


Figura 2.20: As buscas sobre o eixo y encontram pontos na área hachurada.

Em seguida, buscamos por w_1 e w_2 nas subárvores à esquerda e à direita de v_{div}. Seja v um nó à esquerda de v_{div}. Se $w_1 >_y p(v)$ então a busca continua para a direita de v e todo ponto armazenado na subárvore à esquerda de v é y-menor que w_1 e não pode pertencer a W^-. Se $w_1 \leq_y p(v))$ então a busca continua para a esquerda de v e todo ponto armazenado na subárvore à direita de v é y-maior que w_1 e y-menor que w_2. Então devemos listar os pontos que estão armazenados na subárvore com raiz em $d(v)$ e que são x-(menores ou iguais) que w_2. Para isso, verificamos inicialmente se $p_{min}(d(v)) \leq_x w_2$. Se sim então $p_{min}(d(v))$ pertence a W^- e, recursivamente, verificamos os pontos armazenados nos filhos do nó $d(v)$. Se não então $p_{min}(d(v))$ não pertence a W^- e nem os pontos armazenados em seus filhos pela propriedade de uma árvore de busca com prioridade. Este procedimento está descrito no algoritmo PontosMinheap. O algoritmo ConsultaMinÁrvoreBuscaPrioridade formaliza esta ideia. Note que ele usa os algoritmos EncontraNóDivide e PontosMinheap levemente alterados. O primeiro, devolve, além de v_{div}, uma lista L, enquanto que o segundo, recebe como parâmetro uma árvore de busca com prioridade e uma janela ilimitada no plano.

Agora, considere o exemplo da Figura 2.21 e a árvore de busca com prioridade sobre este exemplo na Figura 2.22.

A aresta mais escura na árvore de busca com prioridade T da Figura 2.22 representa a aresta pertencente ao caminho da raiz de T até o nó v_{div}. Este caminho é denotado por C_r. As arestas pontilhadas representam os caminhos C e C' e as arestas tracejadas representam os momentos em que o algoritmo PontosMinheap é chamado. Os nós hachurados armazenam pontos que estão na janela da Figura 2.21. Observe que a raiz de T, v_{raiz}, está em C_r e o ponto $p_{min}(v_{raiz})$ pertence a W^-. Isso indica que devemos verificar se um $p_{min}(v)$ armazenado em um nó v de C_r pertence a W^-. Perceba que os pontos $p_{min}(v)$ armazenados em um nó v ou do caminho C ou do caminho C' também deverão ser verificados. Resumindo, para cada nó v dos caminhos C_r, C e C' devemos verificar se $p_{min}(v)$ pertence a W^-. No exemplo da Figura 2.21 os pontos p_0 e p_{21} são encontrados através de C_r, p_{15}, p_{13} e p_{14} através de

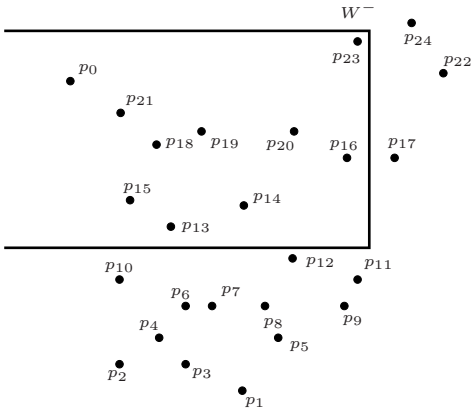


Figura 2.21: Um conjunto de pontos no plano.

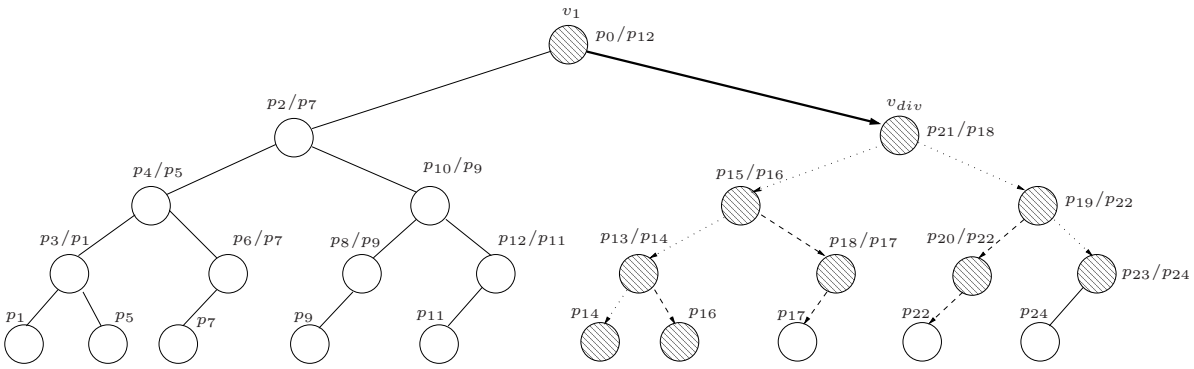


Figura 2.22: Árvore de busca com prioridade construída sobre o exemplo da Figura 2.21. Lembre que armazenamos dois pontos (p_i/p_j) em um nó interno v, onde $p_i = p_{min}(v)$ e $p_j = p(v)$. As folhas não possuem $p(v)$.

C, p_{19} e p_{23} através de C'. Os pontos p_{16}, p_{18} e p_{20} são encontrados através do algoritmo PontosMinheap.

Lema 15 **[21]** *O algoritmo* ConsultaMinÁrvoreBuscaPrioridade *resolve uma W^--consulta sobre um conjunto de pontos distintos não vazio P em tempo $O(\log n + k)$, onde n é o número de pontos de P e k é o número de pontos de P que pertencem a W^-.*

Demonstração: Seja T uma árvore de busca com prioridade construída sobre P. Pela sua construção, T tem altura $O(\log n)$. Com isso, consumimos tempo $O(\log n)$ para encontrar v_{div}. Isso porque consumimos tempo constante em cada nó v do caminho da raiz até v_{div} (C_r), isto é, em v decidimos se continuamos para sua esquerda ou para sua direita e verificamos se $p_{min}(v)$ pertence a W^-, armazenando-o em uma lista, caso verdadeiro. A partir de v_{div},

Algoritmo 14 ConsultaMinÁrvoreBuscaPrioridade (a_b_p v_{raiz}, janela_ilimitada W^-)

Entrada: Um conjunto P de pontos distintos não vazio armazenados em uma árvore de busca com prioridade (a_b_p) com raiz em v_{raiz} e uma janela ilimitada W^-.

Saída: Uma lista L que contém os pontos de P que pertencem a W^-.

1. $L,\ v_{div} \leftarrow$ EncontraNóDivide($v_{raiz},\ W^-$)
2. **se** v_{div} não é folha **e** nem quase folha **então**
3. ▷ Na linha seguinte, verificamos se o ponto armazenado em v_{div} pertence a W^-.
4. **se** $p_{min}(v_{div}) \in W^-$ **então**
5. $L \leftarrow L \cup \{p_{min}(v_{div})\}$
6. **fim se**
7. ▷ Nas linhas 8 - 29 andamos sobre o caminho C.
8. $u \leftarrow e(v_{div})$
9. **enquanto** u não é uma folha **e** nem uma quase folha **faça**
10. ▷ Na linha seguinte, verificamos se um ponto armazenado em um nó de C pertence a W^-.
11. **se** $p_{min}(v) \in W^-$ **então**
12. $L \leftarrow L \cup \{p_{min}(u)\}$
13. **fim se**
14. **se** $w_1 \leq_y p_{min}(u)$ **então**
15. $L \leftarrow L \cup$ PontosMinheaps($d(u),\ W^-$)
16. $u \leftarrow e(u)$
17. **senão**
18. $u \leftarrow d(u)$
19. **fim se**
20. **fim enquanto**
21. ▷ Aqui u é ou uma folha ou uma quase folha.
22. **se** $p_{min}(u) \in W^-$ **então**
23. $L \leftarrow L \cup \{p_{min}(u)\}$
24. **fim se**
25. **se** u é uma quase folha **então**
26. **se** $p_{min}(e(u)) \in W^-$ **então**
27. $L \leftarrow L \cup \{p_{min}(e(u))\}$
28. **fim se**
29. **fim se**
30. ▷ De forma simétrica encontre os pontos que estão armazenados na subárvore à direita de v_{div} andando em C'.
31. **senão**
32. ▷ Aqui v_{div} é ou uma folha ou uma quase folha.
33. **se** $p_{min}(v_{div}) \in W^-$ **então**
34. $L \leftarrow L \cup \{p_{min}(v_{div})\}$
35. **fim se**
36. **se** v_{div} é uma quase folha **então**
37. **se** $p_{min}(e(v_{div})) \in W^-$ **então**
38. $L \leftarrow L \cup \{p_{min}(e(v_{div}))\}$
39. **fim se**
40. **fim se**
41. **fim se**
42. **devolva** L

Algoritmo 15 EncontraNóDivide(árvore_busca_prioridade v_{raiz}, janela_ilimitada W^-)

Entrada: Um conjunto de pontos distintos não vazio P armazenados em uma árvore de busca com prioridade com raiz em v_{raiz} e uma janela ilimitada W^-.

Saída: Uma lista L e um nó v_{div}. L contém os pontos de P que estão armazenados em um nó do caminho C_r e pertencem a W^-. v_{div} é: ou um nó que divide os caminhos C e C'; ou uma folha; ou uma quase folha.

1. $L \leftarrow \emptyset$
2. $v_{div} \leftarrow v_{raiz}$
3. **enquanto** v_{div} não é folha **e** nem quase folha **e** $w_1 >_y p(v_{div})$ **e** $w_2 \leq_y p(v_{div})$ **faça**
4. ▷ Na linha seguinte, verificamos se um ponto armazenado em um nó de C_r pertence a W^-.
5. **se** $p_{min}(v_{div}) \in W^-$ **então**
6. $L \leftarrow L \cup \{p_{min}(v_{div})\}$
7. **fim se**
8. **se** $w_2 \leq_y p(v_{div})$ **então**
9. $v_{div} \leftarrow e(v_{div})$
10. **senão**
11. $v_{div} \leftarrow d(v_{div})$
12. **fim se**
13. **fim enquanto**
14. **devolva** $L,\ v_{div}$

buscamos por w_1 e w_2 nas subárvores à esquerda e à direita de v_{div}, encontrando dois caminhos C e C' com tamanhos $O(\log n)$ cada. Para cada nó v de C ou C' consumimos tempo constante para verificar se $p_{min}(v)$ pertence a W^-, armazenando-o caso verdadeiro. Se v pertence a C e $w_1 \leq_y p(v)$ (ou v pertence a C' e $w_2 >_y p(v)$), então chamamos o algoritmo PontosMinheap que consome tempo $O(k_v)$, onde k_v é o número de pontos de P que pertencem a W^- e estão armazenados na subárvore com raiz em $d(v)$ (ou $e(v)$). O consumo de tempo de todas as chamadas deste algoritmo é $O(k)$. Com isso, temos que o consumo de tempo do algoritmo ConsultaMinÁrvoreBuscaPrioridade é $O(\log n + k)$. □

Capítulo 3

Segmentos em janelas

Estamos interessados em encontrar objetos geométricos em uma janela no plano: pontos e segmentos de reta. No Capítulo 2 falamos sobre algoritmos e estruturas de dados que encontram pontos em uma janela. Neste capítulo falamos sobre algoritmos e estruturas de dados que encontram segmentos em uma janela.

Chamamos o subproblema que vamos resolver neste capítulo de busca por segmentos em uma janela e como no capítulo anterior, este problema apresenta duas fases. Dado um conjunto S com n segmentos, na primeira fase organizamos os elementos de S em uma estrutura de dados. Na segunda fase resolvemos várias W-consultas sobre S, isto é, para cada janela W listamos os segmentos de S que interceptam W.

Podemos usar os algoritmos e as estruturas de dados estudadas no capítulo anterior para listar os segmentos de S que possuem pelo menos um ponto extremo em W. Os algoritmos e as estruturas estudadas neste capítulo listam os segmentos restantes. Seja $S' \subseteq S$ o conjunto dos segmentos de reta de S que não possuem pontos extremos em W. Com isso um segmento s' de S' intercepta W se, e somente se, s' intercepta pelo menos duas "bordas" (s_1, s_2, s_3 ou s_4 - veja seção 2.2) de W. Então, uma consulta sobre S deve verificar quais segmentos interceptam pelo menos duas bordas de uma janela. Denotamos por s_i-consulta uma consulta que lista todos os segmentos de S que interceptam a borda s_i de uma janela.

Inicialmente supomos que os segmentos de reta do conjunto S são horizontais e listamos os segmentos de S que interceptam uma borda vertical de uma janela. É neste momento que estudamos a estrutura de dados árvore de intervalos [12, 20]. Depois consideramos que os segmentos tenham qualquer orientação porém, sem intersecção entre eles. Essa restrição nos permite em alguns momentos ordenar os segmentos de S. Sem ela essa ordenação não seria possível. Neste momento estudamos a estrutura de dados árvore de segmentos [3].

Como fizemos no capítulo anterior, começamos considerando o subproblema em uma dimensão. Assim, os segmentos são intervalos sobre a reta e s_i é um ou dois pontos na reta (se s_i é vertical ou horizontal). Neste caso o problema se resume em encontrar os intervalos de um conjunto que contêm um ponto (pois um intervalo de S intercepta um intervalo se contém pelo menos um de seus extremos). Depois estendemos o subproblema para duas dimensões.

3.1 Intervalos em janelas

Seja S um conjunto de intervalos na reta. Para cada intervalo s de S, denotamos por $p_e(s)$ o seu ponto extremo esquerdo e por $p_d(s)$ o seu ponto extremo direito. Dizemos que um vetor $V = [s_1, s_2, \ldots, s_n]$ que armazena intervalos é p_e-ordenado se $p_e(s_1) \leq p_e(s_2) \leq \ldots \leq p_e(s_n)$. Vamos organizar estes intervalos em uma estrutura de dados que chamamos de árvore de intervalos. Nessa árvore cada nó v armazena um ponto denotado por $p_m(v)$, dois apontadores para duas estruturas auxiliares e dois apontadores para os seus filhos à esquerda e à direita. O algoritmo que constroi uma árvore de intervalos recebe um conjunto S com n intervalos e devolve uma árvore de intervalos. Caso $n > 0$ então criamos um nó v e escolhemos um ponto para $p_m(v)$ tal que o tamanho dos conjuntos $S_e(v) = \{s \in S : p_d(s) < p_m(v)\}$ e $S_d(v) = \{s \in S : p_e(s) > p_m(v)\}$ não passe de $\lceil \frac{n}{2} \rceil$. Como os intervalos de S estão armazenados em um vetor p_e-ordenado $V = [s_1, s_2, \ldots, s_n]$, $p_m(v)$ pode ser o ponto $p_e(s_{\lceil \frac{n}{2} \rceil})$. Dizemos que os intervalos de $S_e(v)$ estão à esquerda de $p_m(v)$ e os de $S_d(v)$ estão à direita. Seja $S_m(v) = \{s \in S : p_e(s) \leq p_m(v) \leq p_d(s)\}$ o conjunto dos intervalos que contêm $p_m(v)$. Armazenamos os intervalos de $S_m(v)$ em duas estruturas auxiliares $L_1(v)$ e $L_2(v)$ (suponha inicialmente que estas estruturas são dois vetores ordenados. $L_1(v)$ é p_e-ordenado e $L_2(v)$ é p_d-ordenado). Em seguida criamos a subárvore à esquerda e à direita de v chamando recursivamente este procedimento que recebe, respectivamente, $S_e(v)$ e $S_d(v)$ e devolvemos v. Se $n = 0$ então o algoritmo devolve uma árvore de intervalos vazia. O algoritmo ConstroiÁrvoreIntervalos1D descreve formalmente esta ideia e a Figura 3.1 ilustra uma árvore de intervalos $1D$.

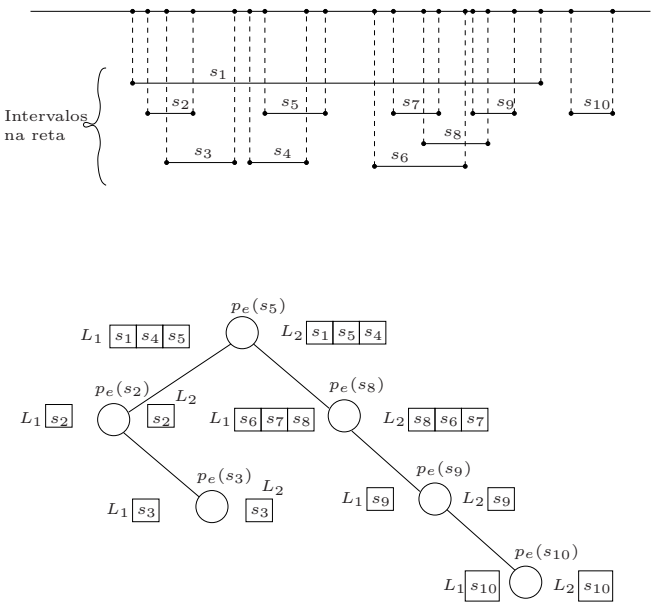


Figura 3.1: Um exemplo de uma árvore de intervalos $1D$.

Algoritmo 16 ConstroiÁrvoreIntervalos1D (vetor V)

Entrada: Um conjunto de intervalos S através de um vetor p_e-ordenado $V = [s_1, s_2, \ldots, s_n]$.
Saída: A raiz v de uma árvore de intervalos.

1. **se** $n > 0$ **então**
2. Crie um nó v.
3. Crie dois vetores vazios V_e e V_d para armazenar intervalos.
4. $p_m(v) \leftarrow p_e(s_{\lceil \frac{n}{2} \rceil})$
5. $i \leftarrow 1$
6. **enquanto** $i \leq n$ **e** $p_e(s_i) \leq p_m(v)$ **faça**
7. **se** $p_d(s_i) < p_m(v)$ **então**
8. Insira s_i no final de V_e.
9. **senão**
10. Insira s_i no final de $L_1(v)$ e $L_2(v)$.
11. **fim se**
12. $i \leftarrow i + 1$
13. **fim enquanto**
14. ▷ A subrotina OrdenaDecrescente ($L_2(v)$) ordena os intervalos de $L_2(v)$ pelo ponto extremo direito em ordem decrescente.
15. OrdenaDecrescente ($L_2(v)$)
16. **enquanto** $i \leq n$ **faça**
17. Insira s_i no final de V_d.
18. $i \leftarrow i + 1$
19. **fim enquanto**
20. $e(v) \leftarrow$ ConstroiÁrvoreIntervalos1D (V_e)
21. $d(v) \leftarrow$ ConstroiÁrvoreIntervalos1D (V_d)
22. **senão**
23. $v \leftarrow NULO$
24. **fim se**
25. **devolva** v

Vamos analisar o consumo de tempo do algoritmo ConstroiÁrvoreIntervalos1D considerando L_1 e L_2 como dois vetores ordenados. Como V é p_e-ordenado, não temos problemas para manter L_1 ordenado, pois basta inserir um intervalo sempre no seu final. Isso não ocorre com o vetor L_2. Os intervalos que armazenamos em L_2 na linha 10 do algoritmo não estão p_d-ordenados. Por isso, na linha 15 chamamos um algoritmo que ordena em ordem decrescente os intervalos em L_2.

Como V armazena n elementos, teremos n pontos extremos direitos. Para cada nó v, encontramos um conjunto de intervalos $S_m(v)$ tal que todo intervalo s de $S_m(v)$ contém o ponto $p_m(v)$. Note que ao associar um intervalo s a um conjunto $S_m(v)$, s não é associado a qualquer outro conjunto de intervalos. Isso quer dizer que um intervalo é armazenado uma única vez nos vetores L_1 e L_2. Com isso, um intervalo participa de uma ordenação uma

única vez. Portanto, o consumo de tempo de todas as ordenações decrescentes é $\Theta(n \log n)$. Perceba que a recorrência definida pelo algoritmo ConstroiÁrvoreIntervalos1D (desconsiderando as ordenações pois já sabemos quanto custa todas as chamadas ao algoritmo que ordena decrescente) nos mostra que o consumo de tempo do algoritmo é $\Theta(n \log n)$. Com isso, segue o próximo lema.

Lema 16 **[12, 20]** *O algoritmo* ConstroiÁrvoreIntervalos1D *constroi uma árvore de intervalos sobre um conjunto S com n intervalos usando um vetor p_e-ordenado consumindo tempo $\Theta(n \log n)$.* □

Agora vamos falar como usamos uma árvore de intervalos para resolver uma s_i-consulta. Considere que s_i é um ponto sobre a reta. Seja T uma árvore de intervalos construída sobre um conjunto de intervalos S. Seja v um nó de T. Suponha que $s_i > p_m(v)$. Então, qualquer intervalo armazenado na subárvore à esquerda de v não pode conter s_i. Com isso, a busca continua para a direita de v e analisamos os intervalos que contêm $p_m(v)$ armazenados nas estruturas $L_1(v)$ e $L_2(v)$. Lembre que os intervalos em $S_m(v)$ contêm $p_m(v)$, isto é, $p_e(s) \leq p_m(v) \leq p_d(s)$ para todo s em $S_m(v)$. Note que $s_i > p_m(v) \geq p_e(s)$, para todo s em $S_m(v)$. Com isso, basta encontrar os intervalos de $S_m(v)$ cujo ponto extremo direito é maior ou igual que s_i. O que é equivalente a encontrar os pontos extremos direitos dos intervalos de $L_2(v)$ que estão em uma janela W^+ com $w_1 = s_i$ e $w_2 = +\infty$. Assim, um intervalo s de $L_2(v)$ contém s_i se $p_d(s)$ pertence a W^+. Para verificar isso basta percorrer o vetor p_d-ordenado decrescente $L_2(v)$ listando os intervalos cujo ponto extremo direito é maior ou igual que s_i. Por outro lado, caso $s_i \leq p_m(v)$ então a busca continua para a esquerda de v e devemos listar um intervalo s de $L_1(v)$ caso $p_e(s)$ pertença a uma janela W^- com $w_1 = -\infty$ e $w_2 = s_i$.

Descrevemos formalmente em seguida um algoritmo (ConsultaÁrvoreIntervalos1D) que busca os intervalos de um conjunto S que contêm um ponto s_i. Ele usa dois algoritmos ConsultaVetorOrdenadoDecrescente e ConsultaVetorOrdenadoCrescente. O primeiro recebe um vetor de intervalos p_d-ordenado em ordem decrescente e uma janela W^+ com $w_1 = s_i$ e $w_2 = +\infty$ e devolve todos os intervalos que estão no vetor e que contêm s_i. O segundo recebe um vetor de intervalos p_e-ordenado em ordem crescente e uma janela W^- com $w_1 = -\infty$ e $w_2 = s_i$ e devolve todos os intervalos que estão no vetor e que contêm s_i.

Lema 17 **[12, 20]** *Sejam S um conjunto com n intervalos e s_i um ponto, ambos sobre a reta. O algoritmo* ConsultaÁrvoreIntervalos1D *consome tempo $O(\log n + k)$, onde k é o número de intervalos de S que contêm s_i.*

Demonstração: Mais uma vez vamos dividir o tempo gasto na busca entre o percurso da estrutura T e o tempo para listar os k intervalos procurados. Pelo Lema 2, sabemos que T possui altura $O(\log n)$. Pelo algoritmo sabemos que visitamos um nó em cada nível de T e, portanto, visitamos $O(\log n)$ nós. Em cada nó v consumimos tempo constante para definir uma janela e realizar uma operação de união de duas listas e consumimos

tempo $O(k_v)$ na chamada ou do algoritmo ConsultaVetorOrdenadoDecrescente ou do ConsultaVetorOrdenadoCrescente, onde k_v é o número de intervalos que estão em um vetor de um nó v e que contêm s_i. Como a soma de todos k_v's é igual a k, o consumo de tempo de todas as chamadas destes algoritmos é $O(k)$. Portanto, o consumo de tempo do algoritmo ConsultaÁrvoreIntervalos1D é $O(\log n+k)$. □

Algoritmo 17 ConsultaÁrvoreIntervalos1D (árvore_intervalo v, ponto s_i)

Entrada: Um conjunto de intervalos S armazenados em uma árvore de intervalos com raiz em v e um ponto s_i.

Saída: Uma lista L que armazena os intervalos de S que contêm o ponto s_i.

```
1.  L ← ∅
2.  se v ≠ NULO então
3.     se s_i > p_m(v) então
4.        Defina W+ com w1 = s_i e w2 = +∞.
5.        L ← ConsultaVetorOrdenadoDecrescente (L2(v), W+)
6.        L ← L ∪ ConsultaÁrvoreIntervalos1D (d(v), s_i)
7.     senão
8.        Defina W− com w1 = −∞ e w2 = s_i.
9.        L ← ConsultaVetorOrdenadoCrescente (L1(v), W−)
10.       L ← L ∪ ConsultaÁrvoreIntervalos1D (e(v), s_i)
11.    fim se
12. fim se
13. devolva L
```

Note que as estruturas auxiliares $L_1(v)$ e $L_2(v)$ de um nó v devem ser capazes de resolver, respectivamente, os seguintes problemas: dados um conjunto de pontos S e um ponto s, quais pontos de S são menores ou iguais a s, e quais pontos de S são maiores ou iguais a s? Note que estes problemas são um caso particular de um problema mais geral estudado no Capítulo 2, busca por pontos em uma janela W. Para o primeiro problema W possui $w_1 = -\infty$ e $w_2 = s$ e para o segundo $w_1 = s$ e $w_2 = +\infty$.

Podemos ver na Tabela 3.1 o consumo de tempo da construção de uma árvore de intervalos sobre S quando L_1 e L_2 são árvores limites, *heaps* ou vetores ordenados e o consumo de tempo de uma s_i-consulta sobre uma árvore de intervalos em cada estrutura.

Árvore de intervalos		
Estrutura auxiliar	Construção	Consulta
Árvore limite	$\Theta(n \log n)$	$O(\log^2 n + k)$
Heap	$\Theta(n \log n)$	$O(\log n + k)$
Vetor ordenado	$\Theta(n \log n)$	$O(\log n + k)$

Tabela 3.1: Consumo de tempo da construção e de uma consulta em uma árvore de intervalos construída sobre um conjunto com n intervalos de acordo com sua estrutura auxiliar.

3.2 Segmentos horizontais e verticais em janelas

Seja S um conjunto com n segmentos horizontais no plano e considere-os p_e-ordenados por x. Uma árvore de intervalos sobre S é construída da seguinte forma. Se $n > 0$ então criamos um nó v e encontramos um ponto $p_m(v)$ tal que o tamanho dos conjuntos $S_e(v) = \{s \in S : x(p_d(s)) < x(p_m(v))\}$ e $S_d(v) = \{s \in S : x(p_e(s)) > x(p_m(v))\}$ não passe de $\lceil \frac{n}{2} \rceil$. Mais uma vez observe que podemos usar $p_m(v) = p_e(s_{\lceil \frac{n}{2} \rceil})$. Seja $S_m(v) = \{s \in S : x(p_e(s)) \leq x(p_m(v)) \leq x(p_d(s))\}$ o conjunto dos segmentos que interceptam a reta $x(p_m(v))$. Construímos duas árvores de busca com prioridade (veja seção 2.6) $L_1(v)$ e $L_2(v)$, respectivamente, sobre os conjuntos dos pontos extremos esquerdo e direito dos segmentos de $S_m(v)$. Definimos $L_1(v)$ e $L_2(v)$ como duas árvores de busca com prioridade porém, mais adiante veremos que elas podem ser qualquer estrutura que resolva uma W-consulta. Por fim, da mesma forma construímos a subárvore à esquerda e à direita de v sobre os conjuntos $S_e(v)$ e $S_d(v)$, respectivamente. Formalmente temos o Algoritmo 18 ConstroiÁrvoreIntervalosHorizontal.

A Figura 3.2 ilustra uma árvore de intervalos $2D$ T construída sobre um conjunto de segmentos no plano. A árvore principal de T é formada pelos nós hachurados. Cada nó v da árvore principal de T armazena um ponto e aponta para duas estruturas auxiliares. A estrutura $L_1(v)$ é uma árvore de busca com prioridade mínima construída sobre os pontos extremos esquerdo dos segmentos em $S_m(v)$ e a estrutura $L_2(v)$ é uma árvore de busca com prioridade máxima construída sobre os pontos extremos direito dos segmentos em $S_m(v)$. Veja na seção 2.6 mais informações sobre uma árvore de busca com prioridade mínima. Uma árvore de busca com prioridade máxima é sua correspondente simétrica. Veremos na segunda fase (fase de consultas) que o eixo y é considerado somente nas estruturas auxiliares.

Lema 18 [12, 20] *O algoritmo* ConstroiÁrvoreIntervalosHorizontal *constroi uma árvore de intervalos sobre um conjunto de segmentos horizontais S com cardinalidade n usando um vetor p_e-ordenado em tempo $\Theta(n \log n)$.*

Demontração: Primeiramente vamos verificar o consumo de tempo de todas as chamadas aos algoritmos Ordena na linha 20, ConstroiMinÁrvoreBuscaPrioridade e o seu correspondente simétrico, ConstroiMaxÁrvoreBuscaPrioridade, nas linhas 21 e 22. Seja v um nó de uma árvore de intervalos T. Considere que um segmento s de S intercepte a reta $x(p_m(v))$. Pela linha 11 do algoritmo, o segmento s pertence aos vetores $P_{xe}(v)$, $P_{ye}(v)$, $P_{dx}(v)$ e $P_{dy}(v)$. Vamos denotar por n_v o número de elementos existentes em $P_{xe}(v)$, $P_{ye}(v)$, $P_{dx}(v)$ e $P_{dy}(v)$ de um nó v. Então, durante a construção de um nó v, na linha 20, o consumo de tempo é $\Theta(n_v \log n_v)$. Nas linhas 21 e 22 também o consumo de tempo é $\Theta(n_v \log n_v)$, pelo Lema 13. Como s não é armazenado em V_e e nem em V_d então s não aparece em qualquer outro nó de T e os seus pontos extremos não serão mais ordenados e nem armazenados em outras estruturas auxiliares. Logo, podemos pensar nestas execuções independentes e ver que o consumo total de tempo de todas as chamadas aos algoritmos Ordena, ConstroiMinÁrvoreBuscaPrioridade e ConstroiMaxÁrvoreBuscaPrioridade é

$\sum_{v \in T} \Theta(n_v \log n_v) = \Theta(n \log n)$. O consumo de tempo do algoritmo que nos falta calcular é dado pela solução de uma recorrência $T'(n)$ como definida abaixo.

$$T'(n) = \begin{cases} \Theta(1), & \text{se} \quad n = 0 \\ T'(n_e) + T'(n_d) + \Theta(n), & \text{se} \quad n > 0, \end{cases}$$

onde $n_e, n_d \leq \lfloor \frac{n}{2} \rfloor$. Observe que $T'(n)$ é limitada superiormente por $T(n)$.

$$T(n) = \begin{cases} \Theta(1), & \text{se} \quad n = 0 \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n), & \text{se} \quad n > 0. \end{cases}$$

Portanto, o consumo de tempo do algoritmo é $\Theta(n \log n)$. □

O algoritmo ConstroiMaxÁrvoreBuscaPrioridade é o correspondente simétrico do algoritmo ConstroiMinÁrvoreBuscaPrioridade descrito na seção 2.6. Lembre que uma árvore construída pelo algoritmo ConstroiMinÁrvoreBuscaPrioridade é usada para listar pontos em uma janela ilimitada W^- e uma árvore construída pelo algoritmo simétrico, em uma janela ilimitada W^+.

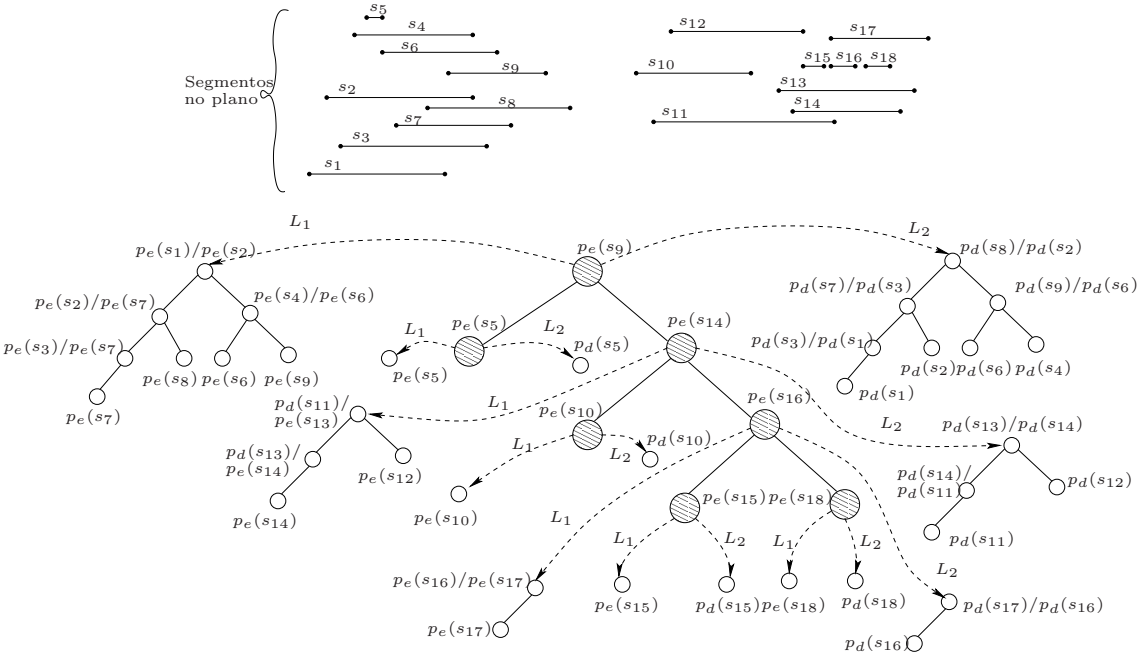


Figura 3.2: Um exemplo de uma árvore de intervalos $2D$.

Algoritmo 18 ConstroiÁrvoreIntervalosHorizontal (vetor V)

Entrada: Um conjunto de segmentos horizontais S através de um vetor p_e-ordenado $V = [s_1, s_2, \ldots, s_n]$.
Saída: A raiz v de uma árvore de intervalos.

1. **se** $n > 0$ **então**
2. Crie um nó v.
3. Crie dois vetores vazios V_e e V_d que armazenam segmentos.
4. Crie quatro vetores vazios P_{xe}, P_{ye} P_{xd} e P_{yd} que armazenam segmentos.
5. $p_m(v) \leftarrow p_e(s_{\lceil \frac{n}{2} \rceil})$
6. $i \leftarrow 1$
7. **enquanto** $i \leq n$ **e** $x(p_e(s_i)) \leq x(p_m(v))$ **faça**
8. **se** $x(p_d(s_i)) < x(p_m(v))$ **então**
9. Insira s_i no final de V_e.
10. **senão**
11. Insira s_i no final de P_{xe}, P_{ye}, P_{xd} e P_{yd}.
12. **fim se**
13. $i \leftarrow i + 1$
14. **fim enquanto**
15. **enquanto** $i \leq n$ **faça**
16. Insira s_i no final de V_d.
17. $i \leftarrow i + 1$
18. **fim enquanto**
19. ▷ Neste ponto P_{xe} está p_e-ordenado pela x-coordenada.
20. Ordena (P_{ye}), Ordena (P_{xd}), Ordena (P_{yd})
21. $L_1(v) \leftarrow$ ConstroiMinÁrvoreBuscaPrioridade (P_{xe}, P_{ye})
22. $L_2(v) \leftarrow$ ConstroiMaxÁrvoreBuscaPrioridade (P_{xd}, P_{yd})
23. $e(v) \leftarrow$ ConstroiÁrvoreIntervalosHorizontal (V_e)
24. $d(v) \leftarrow$ ConstroiÁrvoreIntervalosHorizontal (V_d)
25. **senão**
26. $v \leftarrow NULO$
27. **fim se**
28. **devolva** v

Sejam S um conjunto de segmentos horizontais e $s_i = \overline{(x, y)(x, y')}$ com $y < y'$, um segmento vertical no plano. Uma s_i-consulta pode ser resolvida da seguinte forma. Seja T uma árvore de intervalos construída sobre S. Seja v um nó de T. Suponha que $x > x(p_m(v))$. Então a busca continua na subárvore com raiz em $d(v)$ e qualquer segmento armazenado na subárvore com raiz em $e(v)$ não pode interceptar s_i. Sabemos também que o ponto extremo esquerdo dos segmentos em $S_m(v)$ estão à esquerda do segmento s_i, pois $x(p_e(s)) \leq x(p_m(v)) < x$ para todo s em $S_m(v)$. Logo, devemos listar os **segmentos** em $S_m(v)$ que têm seu **ponto extremo direito** dentro de uma janela ilimitada W^+ com $w_1 = (x, y)$ e $w_2 = (+\infty, y')$. Para isso, o algoritmo ConsultaMaxÁrvoreBuscaPrioridade deverá devolver uma lista que contém todos os segmentos $s \in L_2(v)$ cujo $p_d(s) \in W^+$. Caso $x \leq x(p_m(v))$, podemos obter os segmentos que interceptam s_i de forma simétrica, ou seja, a busca continua na subárvore com raiz em $e(v)$, os segmentos armazenados na subárvore com raiz em $d(v)$ não podem interceptar s_i e devemos listar os **segmentos** de $S_m(v)$ que possuem **ponto extremo esquerdo** dentro de uma janela ilimitada W^- com $w_1 = (-\infty, y)$ e $w_2 = (x, y')$. A Figura 3.3 ilustra tais situações.

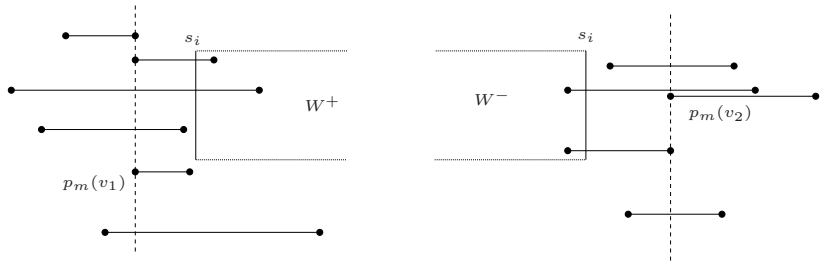


Figura 3.3: Segmentos horizontais em $S_m(v_1)$ e $S_m(v_2)$ que interceptam uma janela ilimitada W^+ e W^-.

O algoritmo ConsultaÁrvoreIntervalosHorizontal formaliza essa ideia. Ele recebe uma árvore de intervalos construída sobre um conjunto de segmentos horizontais S e um segmento vertical s_i e devolve uma lista que contém os segmentos de S que interceptam s_i. A coordenada y é verificada nas linhas 5 ou 9 do algoritmo. Nessas linhas buscamos por pontos em uma janela ilimitada usando a estrutura árvore de busca com prioridade.

Lema 19 [12, 20] *Sejam S um conjunto com n segmentos horizontais e s_i um segmento vertical. Seja T uma árvore de intervalos construída sobre S. O consumo de tempo do algoritmo* ConsultaÁrvoreIntervalosHorizontal *é $O(\log^2 n + k)$, onde k é o número de segmentos de S que interceptam s_i.*

Demonstração: Pelo Lema 2, a altura de T é $O(\log n)$. Pelo algoritmo, sabemos que exatamente um nó é visitado em cada nível de T. Logo, visitamos $O(\log n)$ nós em T. Vamos chamar o caminho formado por esses nós de C. Em cada nó de C realizamos uma comparação, uma definição de uma janela, uma consulta em uma árvore de busca com prioridade

Algoritmo 19 ConsultaÁrvoreIntervalosHorizontal (árvore_int v, segmento_vert s_i)
Entrada: Um conjunto de segmentos horizontais S armazenados em uma árvore de intervalos com raiz em v e um segmento vertical $s_i = \overline{(x,y)(x,y')}$, com $y < y'$.
Saída: Uma lista L que armazena os segmentos de S que interceptam s_i.

```
1.  L ← ∅
2.  se v ≠ NULO então
3.     se x > x(p_m(v)) então
4.        Defina W+ com w1 = (x, y) e w2 = (+∞, y').
5.        L ← ConsultaMaxÁrvoreBuscaPrioridade (L2(v), W+)
6.        L ← L ∪ ConsultaÁrvoreIntervalosHorizontal (d(v), s_i)
7.     senão
8.        Defina W− com w1 = (−∞, y) e w2 = (x, y').
9.        L ← ConsultaMinÁrvoreBuscaPrioridade (L1(v), W−)
10.       L ← L ∪ ConsultaÁrvoreIntervalosHorizontal (e(v), s_i)
11.    fim se
12. fim se
13. devolva L
```

e uma operação de união. Com exceção da consulta em uma árvore de busca com prioridade, todas as outras operações consomem tempo constante. Pelo Lema 15, uma consulta em uma árvore de busca com prioridade consome tempo $O(\log n_v + k_v) = O(\log n + k_v)$, onde n_v é o número de segmentos armazenados na árvore de busca com prioridade associada a um nó v e k_v é o número de segmentos dessa árvore que interceptam s_i. Como a soma dos k_v's para todo v de C é k temos que o consumo de tempo do algoritmo é $O(\log^2 n + k)$. □

Sobre o consumo de espaço de tal árvore podemos dizer que, dado um conjunto com n segmentos horizontais S, uma árvore de intervalos T sobre S armazena em um nó v um ponto extremo de um segmento de S, $p_m(v)$, dois apontadores para árvores de busca com prioridade sobre o conjunto $S_m(v) \subseteq S$, $L_1(v)$ e $L_2(v)$, e dois apontadores para os filhos à esquerda e à direita de v, $e(v)$ e $d(v)$. Os apontadores $e(v)$ ou de $d(v)$ e o membro $p_m(v)$ consomem espaço constante. O consumo de espaço de $L_1(v)$ e $L_2(v)$ é $\Theta(|S_m(v)|)$ (Lema 14). Primeiramente vamos analisar o consumo de espaço de $L_1(v)$ e $L_2(v)$ para todo v de T e por fim o consumo de espaço de todos os nós de T.

Sejam $v_1, v_2, \ldots, v_k$ todos os nós de T. Pela construção de T sabemos que $S_m(v_1) \cup S_m(v_2) \cup \ldots \cup S_m(v_k) = S$ e $S_m(v_1) \cap S_m(v_2) \cap \ldots \cap S_m(v_k) = \emptyset$. Portanto, o consumo de espaço das estruturas $L_1(v_1), L_2(v_1), L_1(v_2), L_2(v_2), \ldots, L_1(v_k)$ e $L_2(v_k)$ é $\sum_{i=1}^{k} \Theta(|S_m(v_i)|) = \Theta(|S|) = \Theta(n)$. Para mostrar que k é no máximo n basta perceber que na construção de um nó, armazenamos um ponto extremo esquerdo de um segmento que não será mais armazenado em qualquer outro nó. Portanto, teremos no máximo n nós, ou seja, $k \leq n$.

Podemos encontrar um exemplo onde k é exatamente n. Veja a Figura 3.4. Nesta figura, não existem dois segmentos horizontais que interceptam uma reta vertical. Com isso, o

conjunto $|S_m(v)| = 1$ para todo v de T e assim teremos n nós em T. Com isso, segue o próximo lema.

Lema 20 **[12, 20]** *Seja S um conjunto com n segmentos horizontais. Uma árvore de intervalos com árvores de busca com prioridade como estrutura associada construída sobre S consome espaço $\Theta(n)$.* □

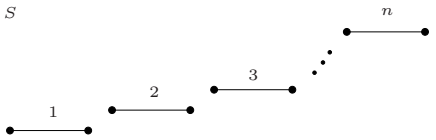


Figura 3.4: Uma árvore de intervalos construída sobre S possui n nós.

Como feito na seção anterior podemos ver na Tabela 3.2 o consumo de tempo da construção de uma árvore de intervalos sobre um conjunto S com n segmentos horizontais com L_1 e L_2 como árvores limites, árvores limites com camadas ou árvores de busca com prioridade, o consumo de tempo de uma s_i-consulta sobre uma árvore de intervalos, e o consumo de espaço de tal árvore.

Árvore de intervalos			
Estrutura auxiliar	Construção	Consulta	Espaço
Árvore limite	$\Theta(n \log n)$	$O(\log^3 n + k)$	$\Theta(n \log n)$
Árvore limite com camadas	$\Theta(n \log n)$	$O(\log^2 n + k)$	$\Theta(n \log n)$
Árvore de busca com prioridade	$\Theta(n \log n)$	$O(\log^2 n + k)$	$\Theta(n)$

Tabela 3.2: Consumo de tempo da construção, de uma consulta e de espaço de uma árvore de intervalos construída sobre um conjunto com n segmentos horizontais de acordo com sua estrutura auxiliar.

De maneira simétrica podemos encontrar os segmentos de um conjunto de segmentos verticais que interceptam um segmento horizontal.

3.3 Mais intervalos em janelas

Nesta seção vamos encontrar os intervalos de um conjunto que contêm um ponto como feito na seção 3.1, porém aqui vamos usar uma nova estrutura de dados chamada árvore de segmentos. Antes de construir uma árvore de segmentos devemos definir um intervalo elementar. Seja $S = \{s_1, s_2, \ldots, s_n\}$ um conjunto de intervalos sobre a reta. Considere cada ponto extremo distinto de $s_1, \ldots, s_n$. Suponha que esses pontos sejam $p_1, p_2, \ldots, p_k$, onde k é no máximo $2n$ e suponha também que eles estão ordenados em ordem crescente, $p_1 < p_2 < \ldots < p_k$. Os intervalos elementares formados pelos intervalos $s_1, \ldots, s_n$ são:

$$(-\infty : p_1), [p_1 : p_1], (p_1 : p_2), [p_2 : p_2], \ldots, (p_{k-1} : p_k), [p_k : p_k], (p_k : +\infty).$$

Perceba que eles particionam a reta em, no máximo, $4n+1$ intervalos. Isso ocorre quando não existem pontos extremos coincidentes entre quaisquer dois intervalos de S, ou seja, $k = 2n$. Assim, temos os seguintes intervalos elementares: cada ponto p_i constitui um intervalo elementar que só contém o ponto (são $2n$ destes intervalos); cada intervalo formado entre dois pontos consecutivos $(p_i : p_{i+1})$ é um intervalo elementar (são $2n-1$ destes intervalos); e dois intervalos da forma $(-\infty : p_1)$ e $(p_m : +\infty)$ são intervalos elementares. Agora vamos falar sobre uma construção de uma árvore de segmentos. Suponha que um vetor p_e-ordenado $V = [s_1, \ldots, s_n]$ armazena os n intervalos de S. Um nó v de uma árvore de segmentos possui os seguintes membros: um intervalo $int(v)$ (não é um intervalo de S), uma lista ligada $L(v)$ de intervalos de S e dois apontadores para o filho à esquerda e à direita de v, $e(v)$ e $d(v)$. Algumas propriedades devem ser obedecidas: 1) $int(f)$ é um intervalo elementar, onde f é uma folha; 2) $int(v)$ é um intervalo formado pelo ponto extremo esquerdo do intervalo do seu filho à esquerda e pelo ponto extremo direito do seu filho à direita, onde v é um nó interno, assim o intervalo $int(v)$ é a união dos intervalos dos seus filhos; e 3) todo intervalo em $L(v)$ é um intervalo de S que contém $int(v)$ mas não contém $int(\text{pai}(v))$, onde v é um nó qualquer. A Figura 3.7 ilustra uma árvore de segmentos construída sobre um conjunto de intervalos na reta. Na figura os intervalos estão em níveis diferentes para melhor visualização.

Uma construção de tal árvore é dada em três etapas. Na primeira etapa, dado um conjunto de intervalos S, obtemos o conjunto de intervalos elementares correspondente. Na segunda, construímos uma árvore de busca binária balanceada T similar a um *heap*, preenchendo o membro $int(v)$ de cada nó v. Consumimos tempo linear com uma construção feita de baixo para cima. Primeiro preenchemos as folhas com os intervalos elementares. A folha mais à esquerda de T é preenchida com o menor intervalo elementar, ou seja, $(-\infty : p_1)$. A segunda folha mais à esquerda é prenchida com o intervalo elementar $[p_1 : p_1]$ e assim por diante até a folha mais à direita que é preenchida com o intervalo elementar $(p_m : +\infty)$. Os pais de cada nó de T são preenchidos com a união dos seus filhos, ou seja, o intervalo formado pelo ponto extremo esquerdo do intervalo do seu filho à esquerda e com o ponto extremo direito do intervalo do seu filho à direita. Esta construção é similar à construção vista no Capítulo 2, Algoritmo 7.

Na terceira etapa os intervalos de S devem ser inseridos em T. Dado um nó v de T e um intervalo s de S, inserimos s em $L(v)$ caso s contenha $int(v)$ e não contenha $int(pai(v))$. Caso contrário, inserimos s na subárvore à esquerda de v se $s \cap int(e(v)) \neq \emptyset$ e à direita de v se $s \cap int(d(v)) \neq \emptyset$. O algoritmo InsereIntervalo insere um intervalo em uma árvore de busca binária balanceada construída pelo algoritmo ConstroiÁrvore.

Lema 21 **[8]** *Seja T uma árvore de segmentos. Seja S um conjunto de intervalos na reta. Cada intervalo s de S é inserido no máximo duas vezes em um nível de T.*

Demonstração: Suponha por um instante que existam em um mesmo nível, nós de T, $v_1, v_2, \ldots, v_j$, $j \geq 3$ que armazenam um intervalo s. Sem perda de generalidade, suponha que $int(v_1) < int(v_2) < \ldots < int(v_j)$, isto é, $int(v_1)$ é o intervalo mais à esquerda dentre os

intervalos $int(v_1), int(v_2), \ldots, int(v_j)$ seguido pelo $int(v_2)$ e assim por diante. Como s está armazenado em v_1 e v_j, temos que s começa antes de $int(v_1)$ e termina depois de $int(v_j)$.

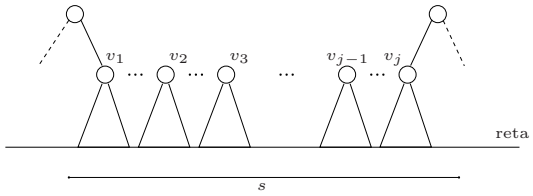


Figura 3.5: O intervalo s começa antes de $int(v_1)$ e termina depois de $int(v_j)$.

Sabemos que v_i e v_{i+1} para $i = 1, \ldots, j-1$ não podem ser irmãos, caso contrário, s conteria os intervalos disjuntos de v_i e v_{i+1} e assim, o pai de v_i e v_{i+1} deveria armazenar s. Então, considere v_i e seu irmão, denotado por v_i'. O intervalo armazenado no pai de v_i e v_i' é formado pela união dos intervalos de v_i e de v_i'. Chamemos este intervalo de s'.

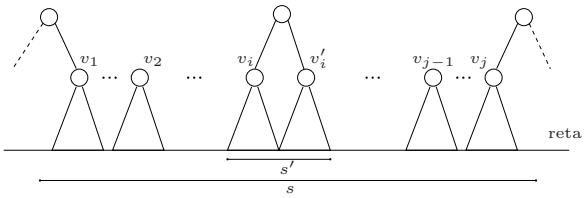


Figura 3.6: Neste caso v_i' é filho à direita porém, o mesmo vale quando v_i' é filho à esquerda.

Note que s contém s' e portanto s deve ser armazenado no pai de v_i e não em v_i. Isso vale para todo $i = 2, \ldots, j-1$, enquanto que os nós v_1 e v_j devem ser filhos, respectivamente, à direita e à esquerda. Portanto, um intervalo s é armazenado no máximo duas vezes em um mesmo nível. □

Lema 22 *Seja T uma árvore de segmentos. Seja S um conjunto de intervalos na reta. Ao inserir um intervalo s de S em T visitamos no máximo quatro nós em um mesmo nível de T.*

Demonstração: Suponha que existam em um mesmo nível, nós de T, $v_1, v_2, \ldots, v_j$, $j \geq 5$ que foram visitados ao inserir um intervalo s em T. Suponha que $int(v_1) < int(v_2) < \ldots < int(v_j)$, como no lema anterior. Como visitamos v_1 e v_j, sabemos que $s \cap int(v_1) \neq \emptyset$ e $s \cap int(v_j) \neq \emptyset$ e assim, s contém $int(v_i)$, para $i = 2, \ldots, j-1$. Seja v_i' o irmão de v_i. Note que, se s contém $int(v_i')$ então v_i não poderia ser visitado, pois s conteria o pai de v_i. Logo, v_1 e v_2 devem ser irmãos, v_{j-1} e v_j devem também ser irmãos, s não pode conter v_1 e v_j e os nós v_i, para $i = 3, \ldots, j-2$ não podem ser visitados, caso contrário, s conteria $int(v_i)$ e

$int(v'_i)$ e assim, s conteria o pai de v_i e v'_i. Portanto, ao inserir um intervalo s em T visitamos no máximo quatro nós em um mesmo nível. □

Uma construção de uma árvore de segmentos 1D é dada pelo Algoritmo 22. Ele usa um algoritmo que recebe um conjunto de intervalos S e devolve os intervalos elementares p_e-ordenados correspondentes a S.

Algoritmo 20 ConstroiÁrvore(vetor V)

Entrada: Um conjunto não vazio com $m \leq 4n + 1$ intervalos elementares através de um vetor p_e-ordenado V.

Saída: Uma árvore de busca binária balanceada, similar a um *heap*, sobre os elementos de V.

1. Crie um vetor T com $2m - 1$ posições.
2. $h \leftarrow \lceil \log(m) \rceil$
3. $l' \leftarrow 2^h - m$
4. $l \leftarrow m - l'$
5. $i \leftarrow 2m - 2$
6. ▷ Nas linhas 7 – 11 construímos as folhas do nível h.
7. **para** $j \leftarrow l - 1$ decrescendo até 0 **faça**
8. $int(T[i]) \leftarrow V[j]$
9. Marque $T[i]$ como folha.
10. $i \leftarrow i - 1$
11. **fim para**
12. ▷ Nas linhas 13 – 17 construímos as folhas do nível $h - 1$.
13. **para** $j \leftarrow m - 1$ decrescendo até l **faça**
14. $int(T[i]) \leftarrow V[j]$
15. Marque $T_y[i]$ como folha.
16. $i \leftarrow i - 1$
17. **fim para**
18. ▷ Nas linhas 19 – 25 construímos os nós internos.
19. **enquanto** $i \geq 0$ **faça**
20. ▷ O ponto extremo esquerdo do intervalo armazenado em um nó interno recebe o ponto extremo esquerdo do intervalo do seu filho à esquerda.
21. $p_e(int(T[i])) \leftarrow p_e(int(T[2i + 1]))$
22. ▷ O ponto extremo direito do intervalo armazenado em um nó interno recebe o ponto extremo direito do intervalo do seu filho à direita.
23. $p_d(int(T[i])) \leftarrow p_d(int(T[2i + 2]))$
24. $i \leftarrow i - 1$
25. **fim enquanto**
26. **devolva** T

Algoritmo 21 InsereIntervalo(arvore v, intervalo s)

Entrada: Um árvore de busca binária balanceada com raiz em v construída pelo algoritmo ConstroiÁrvore e um intervalo s.
Saída: O intervalo s inserido na árvore de segmentos com raiz em v.

```
1. u ← v
2. se Contém(s, int(u)) então
3.    Insira s em L(u).
4. senão
5.    se Intersecção(s, int(e(u))) então
6.       InsereIntervalo(e(u), s)
7.    fim se
8.    se Intersecção(s, int(d(u))) então
9.       InsereIntervalo(d(u), s)
10.   fim se
11. fim se
```

Algoritmo 22 ConstroiÁrvoreSegmentos1D(vetor V)

Entrada: Um conjunto com n intervalos através do vetor V.
Saída: Uma árvore de segmentos T.

```
1. V_el ← ConstroiIntervalosElementares(V)
2. T ← ConstroiÁrvore(V_el)
3. para i ← 1 até n faça
4.    InsereIntervalo(T, V[i])
5. fim para
6. devolva T
```

Lema 23 **[8]** *Seja S um conjunto com n intervalos. Uma árvore de segmentos sobre S é construída pelo algoritmo* ConstroiÁrvoreSegmentos1D *que consome tempo $\Theta(n \log n)$.*

Demontração: O consumo de tempo da linha 1 é dado pelo consumo de tempo do algoritmo ConstroiIntervalosElementares. Ele realiza uma ordenação em no máximo $2n$ pontos (linha 7). As demais linhas deste algoritmo consomem tempo proporcional ao número de intervalos elementares que pode ser no máximo $4n + 1$. Logo, o consumo de tempo de linha 1 é $\Theta(n \log n)$. Como visto na seção 2.2, o consumo de tempo da linha 2 é $\Theta(n)$. A linha 6 consome tempo constante. Falta analisar o consumo de tempo das linhas 3, 4 e 5. Nestas linhas inserimos cada um dos n intervalos de S. Pelos Lemas 21 e 22 sabemos que inserimos um intervalo s de S em T no máximo duas vezes em um mesmo nível e, para isso, visitamos no máximo quatro nós por nível de T que possui altura $O(\log n)$. Como o consumo de tempo para inserir s em $L(v)$ é constante temos que o consumo de tempo para inserir s é $O(\log n)$. Com isso, consumimos tempo $O(n \log n)$ para inserir os n intervalos de S em T. Portanto, o consumo de tempo do algoritmo é $\Theta(n \log n)$. □

As primitivas usadas pelo Algoritmo 21 são descritas em seguida.

Algoritmo 23 INTERSECÇÃO(intervalo s, intervalo s')

Entrada: Dois intervalos s e s'.
Saída: VERDADEIRO se $s \cap s' = \emptyset$. FALSO caso contrário.

1. **se** CONTÉM(s, s') **ou** PERTENCE($p_e(s)$, s') **ou** PERTENCE($p_d(s)$, s') **então**
2. **devolva** VERDADEIRO
3. **senão**
4. **devolva** FALSO
5. **fim se**

Algoritmo 24 CONTÉM(intervalo s, intervalo s')

Entrada: Dois intervalos s e s'.
Saída: VERDADEIRO se s contém s'. FALSO caso contrário.

1. **se** $p_e(s) \leq p_e(s')$ **e** $p_d(s) \geq p_d(s')$ **então**
2. **devolva** VERDADEIRO
3. **senão**
4. **devolva** FALSO
5. **fim se**

Algoritmo 25 PERTENCE(ponto q, intervalo s)

Entrada: Um ponto q e um intervalo s.
Saída: VERDADEIRO se q pertence a s. FALSO caso contrário.

1. **se** $p_e(s) < q < p_d(s)$ **ou** ($p_e(s)$ é fechado **e** $p_e(s) = q$) **ou** ($p_d(s)$ é fechado **e** $p_d(s) = q$) **então**
2. **devolva** VERDADEIRO
3. **senão**
4. **devolva** FALSO
5. **fim se**

Lema 24 [8] *Seja S um conjunto com n intervalos. Uma árvore de segmentos 1D T construída sobre S consome espaço $O(n \log n)$.*

Demonstração: Pelo Lema 21 um intervalo s de S é inserido no máximo duas vezes em um nível de T. Então, um intervalo é armazenado $O(\log n)$ vezes, pois T possui altura $O(\log n)$. Portanto, n intervalos são armazenados $O(n \log n)$ vezes. Pelo Lema 3 temos que o número de nós de T é no máximo $2(4n+1) - 1 = 8n + 1 = \Theta(n)$. Portanto, o consumo de espaço de T é $O(n \log n)$. □

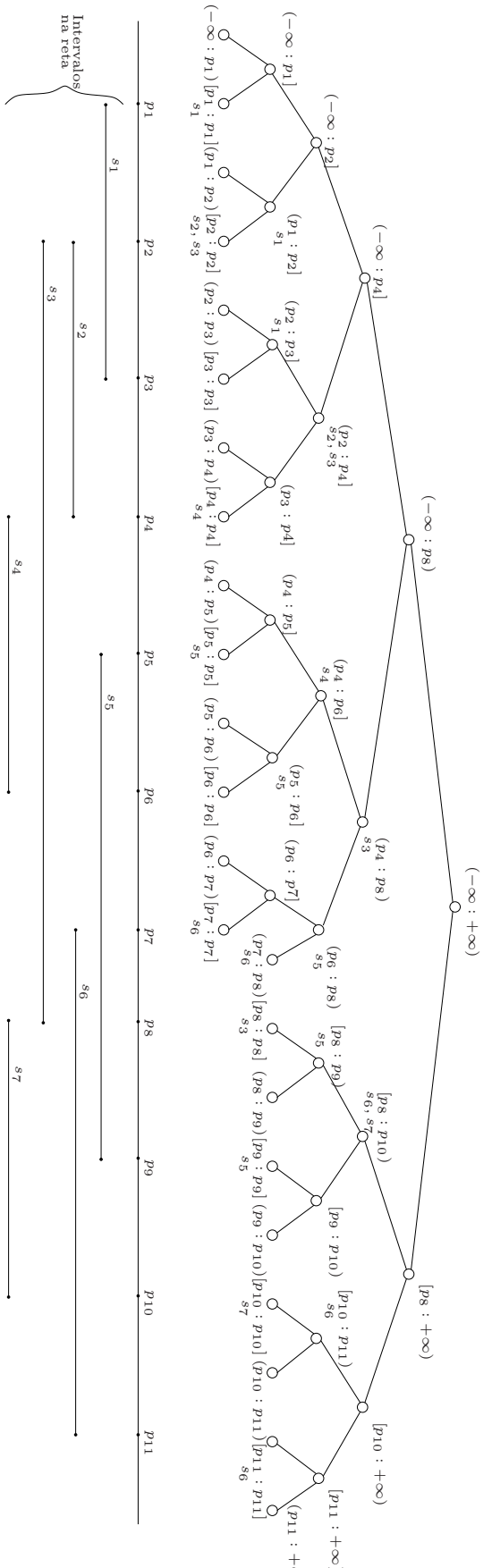


Figura 3.7: Uma árvore de segmentos construída sobre um conjunto de intervalos.

Algoritmo 26 ConstroiIntervalosElementares(vetor V)

Entrada: Um conjunto S com n intervalos através do vetor V.
Saída: O conjunto de intervalos elementares p_e-ordenados correspondentes a S.

1. Crie um vetor P que armazena pontos.
2. Crie um vetor Q que armazena intervalos.
3. **para** $i \leftarrow 1$ até n **faça**
4. Insira no final de P o ponto $p_e(V[i])$.
5. Insira no final de P o ponto $p_d(V[i])$.
6. **fim para**
7. Ordena(P)
8. $m \leftarrow$ RemoveRepetições(P)
9. ▷ A partir desta linha P não possui repetições e o seu tamanho é $m \leq 2n$.
10. $p_e \leftarrow -\infty$
11. **para** $i \leftarrow 1$ até m **faça**
12. $p_d \leftarrow P[i]$
13. Insira no final de Q o intervalo aberto $(p_e : p_d)$.
14. Insira no final de Q o intervalo fechado $[p_d : p_d]$.
15. $p_e \leftarrow p_d$
16. **fim para**
17. $p_d \leftarrow +\infty$
18. Insira no final de Q o intervalo aberto $(p_e : p_d)$.
19. **devolva** Q.

Na fase de consultas, queremos encontrar os intervalos de um conjunto S que contêm um ponto s_i. Considere uma árvore de segmentos T construída sobre S. Vamos realizar uma busca por s_i em T. Seja v um nó do caminho desta busca. Se s_i não pertence a $int(v)$, então s_i não está em nenhum intervalo armazenado nessa subárvore. Assim, considere agora que s_i pertence ao intervalo $int(v)$. Como todo intervalo de S armazenado em $L(v)$ contém $int(v)$ temos que s_i pertence a todo intervalo em $L(v)$. Logo, devemos listá-los. Em seguida, continuamos à esquerda ou à direita de v dependendo se s_i pertence a $int(e(v))$ ou a $int(d(v))$ (lembre que esses intervalos são disjuntos e particionam $int(v)$, portanto o ponto s_i estará em somente um deles). Começamos essa busca na raiz de T (v_{raiz}) e sabemos que $int(v_{raiz})$ é igual a $(-\infty : +\infty)$. Assim, para qualquer busca, s_i está no intervalo da raiz. O algoritmo ConsultaÁrvoreSegmentos1D formaliza essa ideia.

Lema 25 **[8]** *Sejam S um conjunto com n intervalos na reta e s_i um ponto. Seja T uma árvore de segmentos $1D$ construída sobre S. O algoritmo* ConsultaÁrvoreSegmentos1D *consome tempo $O(\log n + k)$, onde k é o número de intervalos de S que contêm s_i.*

Demonstração: Pela linha 2 do algoritmo ConstroiÁrvore sabemos que T possui altura $h = O(\log m)$. Como $m \leq 4n + 1$ temos que $h = O(\log n)$. Assim, o caminho C formado pela busca por s_i em T possui comprimento $O(\log n)$. Em cada nó v de C consumimos tempo $O(k_v)$

para listar os k_v intervalos de $L(v)$ e consumimos tempo constante para unir duas listas. Como $\sum_{v \in C} k_v = k$ temos que o consumo de tempo do algoritmo ConsultaÁrvoreSegmentos1D é $O(\log n + k)$. □

Algoritmo 27 ConsultaÁrvoreSegmentos1D(árvore_segmentos v, ponto s_i)

Entrada: Uma árvore de segmentos $1D$ não nula com raiz em v construída sobre um conjunto com n intervalos S e um ponto s_i.

Saída: Uma lista L com os intervalos de S que contêm o ponto s_i.

```
1.  u ← v
2.  L ← L(u)
3.  se u não é folha então
4.     se Pertence(s_i, int(e(u))) então
5.        devolva L ∪ ConsultaÁrvoreSegmentos1D(e(u), s_i)
6.     senão
7.        devolva L ∪ ConsultaÁrvoreSegmentos1D(d(u), s_i)
8.     fim se
9.  senão
10.    devolva L
11. fim se
```

A Figura 3.8 ilustra uma s_i-consulta realizada na árvore da Figura 3.7. As arestas escuras correspondem às arestas do caminho formado pela busca de s_i na árvore de segmentos. L é uma lista que possui os intervalos que contêm s_i.

3.4 Segmentos em janelas

Nesta seção trabalhamos com um conjunto S com n segmentos quaisquer no plano (não necessariamente horizontais e verticais). Supomos que os segmentos em S não se interceptam dois a dois. Neste caso dizemos que S é um conjunto sem intersecções. Vamos usar uma árvore de segmentos para armazenar os segmentos de S. Os membros de um nó v de uma árvore de segmentos $2D$ são: um intervalo ($int(v)$); um vetor ordenado ($L_{ord}(v)$) que armazena segmentos de S; e dois apontadores para os filhos à esquerda e à direita de v. Existe uma pequena diferença em um membro entre os nós v de uma árvore de segmentos $1D$ e de uma árvore de segmentos $2D$. Na árvore de segmentos $1D$ temos uma lista $L(v)$ enquanto que na árvore de segmentos $2D$ temos um vetor ordenado $L_{ord}(v)$. Uma construção de uma árvore de segmentos $2D$ é similar à construção de uma árvore de segmentos $1D$. Ela é dividida em quatro etapas. As três primeiras etapas são iguais às etapas da construção de uma árvore de segmentos $1D$ (construção dos intervalos elementares definidos pela projeção dos segmentos de S no eixo x; construção de uma árvore de busca binária balanceada T baseada em um *heap*; e a inserção de cada segmento s de S em T). A última etapa consiste na ordenação dos vetores $L_{ord}(v)$ de cada nó v de T. Antes de realizar esta ordenação vamos definir alguns conceitos que serão usados posteriormente. Para cada conceito ilustramos um possível cenário.

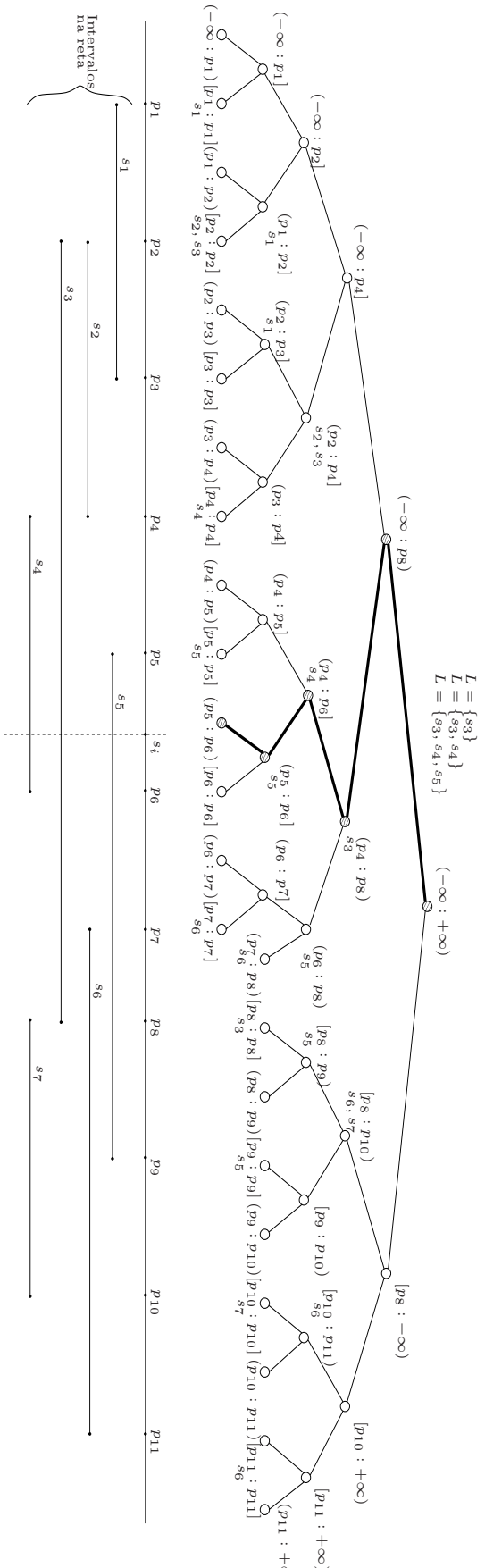


Figura 3.8: Uma s_i-consulta em uma árvore de segmentos.

Definição: Seja s um segmento no plano e p um ponto no plano. Se o passeio fechado $p_e(s)$, $p_d(s)$, p, $p_e(s)$ formar um triângulo no sentido anti-horário então dizemos que p está à esquerda da reta r que contém s. Se este passeio formar um triângulo no sentido horário então p está à direita de r (veja Figura 3.9).

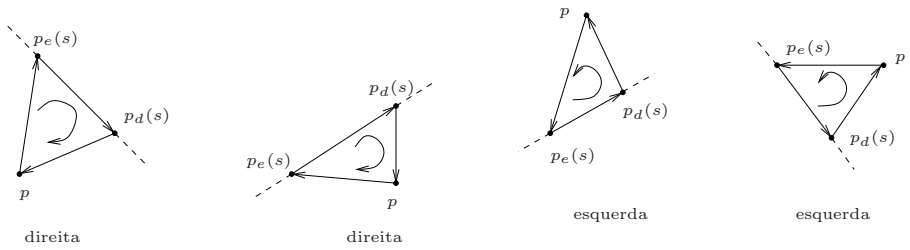


Figura 3.9: Os sentidos de alguns triângulos formados pelo passeio fechado $p_e(s)$, $p_d(s)$, p, $p_e(s)$.

A partir daqui, abusamos da linguagem e dizemos que um ponto p está à esquerda (ou à direita) de um segmento s.

Podem existir pontos que não estão nem à direita e nem à esquerda de um segmento. Nestes casos dizemos que eles estão sobre a reta que contém um segmento (ou sobre um segmento). Note que o triângulo formado pelo passeio possui área igual a zero. Na Figura 3.10 ilustramos alguns cenários.

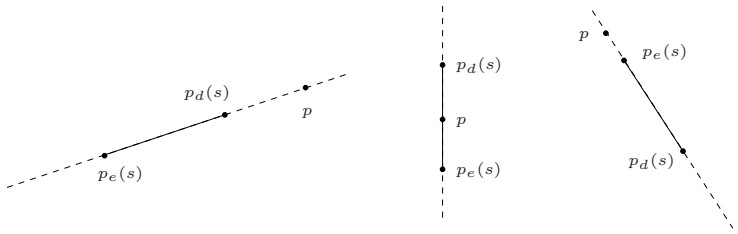


Figura 3.10: Alguns triângulos formados pelo passeio fechado $p_e(s)$, $p_d(s)$, p, $p_e(s)$ com área igual a zero.

Definição: Dados dois segmentos s_1 e s_2 no plano, dizemos que s_1 está à esquerda de s_2 se ambos os pontos extremos estão à esquerda de s_2 ou se um ponto extremo está à esquerda e o outro sobre s_2. De forma simétrica, s_1 está à direita de s_2 se ambos os pontos extremos estão à direita de s_2 ou um ponto extremo está à direita e o outro sobre s_2 (veja Figura 3.11).

No primeiro cenário da Figura 3.11 temos o segmento s_2 à direita do segmento s_1 porém, nos dois últimos cenários, o segmento s_2 não está à direita nem à esquerda de s_1. Com isso, segue a seguinte definição.

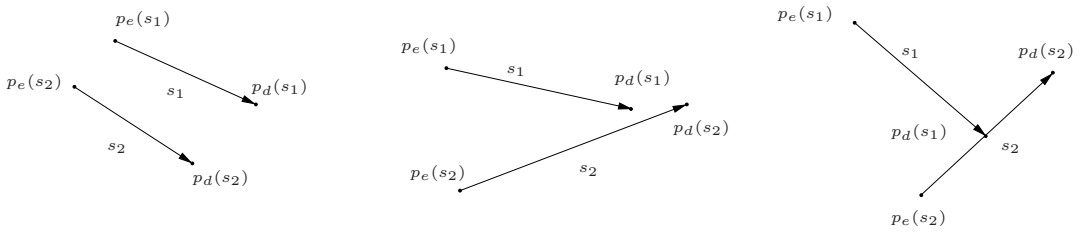


Figura 3.11: Nos três casos s_1 está à esquerda de s_2.

Definição: Dados dois segmentos s_1 e s_2 no plano, dizemos que s_1 pseudo-intercepta s_2 se $p_e(s_1)$ está à esquerda (ou à direita) de s_2 e $p_d(s_1)$ está à direita (ou à esquerda) de s_2 (veja Figura 3.12).

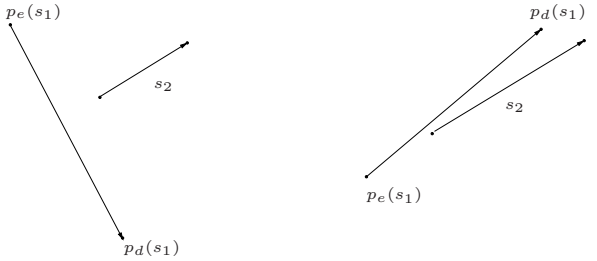


Figura 3.12: Nos dois casos s_1 pseudo-intercepta s_2. Nos dois últimos cenários da Figura 3.11 s_2 pseudo-intercepta s_1.

Lema 26 *Seja S um conjunto de segmentos sem intersecção. Seja s_2 pertencente a S. Se existe s_1 que pseudo-intercepta s_2 então s_2 está ou à direita ou à esquerda de s_1.*

Demonstração: Seja r a reta que contém s_2. Para qualquer p pertencente a r mas não pertencente a s_2 e qualquer reta r' passando por p, que não seja coincidente com r, s_2 está ou à esquerda ou à direita de r' (veja a Figura 3.13).

Seja s_1 um segmento que pseudo-intercepta s_2. Logo, um ponto extremo de s_1 está à esquerda de s_2 e o outro está à direita. Então, existem um ponto p sobre r mas não sobre s_2 (a intersecção entre s_1 e r) e uma reta r' não coincidente com a reta r que contém p e s_1. Portanto, s_2 está ou à esquerda ou à direita de r' e, consequentemente, de s_1. □

Agora podemos realizar a quarta etapa da nossa construção de um árvore de segmentos $2D$ T. Lembre que devemos ordenar $L_{ord}(v)$ de todo nó v de T. Então, considere $int(v)$ e $L_{ord}(v)$ de um nó v de T. (veja a Figura 3.14).

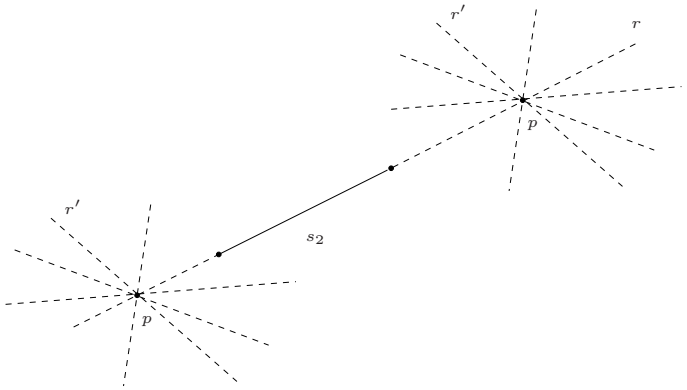


Figura 3.13: Vemos na figura que s_2 está ou à esquerda ou à direita de r'.

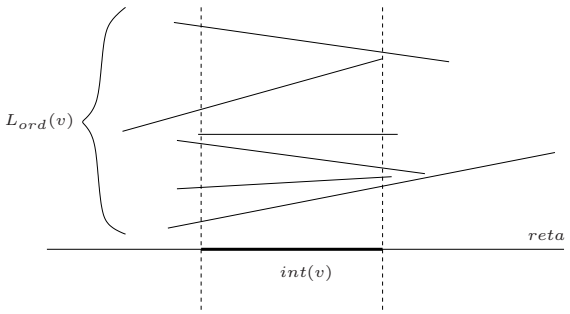


Figura 3.14: A janela $W_{int(v)}$ de um nó v e os segmentos que a atravessam.

Denotamos por $W_{int(v)} = ((p_e(int(v)), -\infty), (p_d(int(v)), +\infty))$ a janela ilimitada formada pelos pontos extremos de $int(v)$. Note que todo segmento de $L_{ord}(v)$ atravessa $W_{int(v)}$. Como S não possui intersecção podemos ordenar os segmentos de $L_{ord}(v)$ da seguinte forma. Dados s_1 e s_2 pertencentes a $L_{ord}(v)$ dizemos que $s_1 < s_2$ se s_1 está à direita de s_2 ou s_1 pseudo-intercepta s_2 e s_2 está à esquerda de s_1. Descrevemos em seguida um algoritmo que constroi uma árvore de segmentos $2D$ sobre um conjunto de segmentos sem intersecção S.

Lema 27 *Seja S um conjunto com n segmentos sem intersecção. A construção de uma árvore de segmentos $2D$ T sobre S feita pelo algoritmo* ConstroiÁrvoreSegmentos *consome tempo $O(n \log^2 n)$.*

Demonstração: Vamos analisar as linhas 6 – 8 pois as demais já foram analisadas no Lema 23. Sejam $v_1, v_2, \ldots v_k$ os k nós de um mesmo nível de T. Sejam $n_{v_1}, n_{v_2}, \ldots, n_{v_k}$ o número de segmentos armazenados nos nós $v_1, v_2, \ldots v_k$. Pelo Lema 21, sabemos que um segmento de S é inserido no máximo duas vezes em um mesmo nível de T. Então, $n_{v_1} + n_{v_2} + \ldots + n_{v_k} \leq 2n$. Consumimos tempo $\Theta(n_{v_i} \log n_{v_i}) \leq c n_{v_i} \log n_{v_i}$, para uma constante positiva c, para ordenar

os segmentos em v_i, para $i = 1, \ldots, k$. Com isso, o consumo de tempo para ordenar todos os segmentos em um mesmo nível de T é:

$$\leq \sum_{i=1}^{k} cn_{v_i} \log n_{v_i} \leq c \sum_{i=1}^{k} n_{v_i} \log n = c \log n \sum_{i=1}^{k} n_{v_i} \leq 2cn \log n = \Theta(n \log n).$$

Como T possui $O(\log n)$ níveis, temos que o consumo de tempo do algoritmo é $O(n \log^2 n)$.□

Na fase de consultas, dados uma árvore de segmentos $2D$ T construída sobre um conjunto de segmentos sem intersecção S e um segmento vertical s_i, podemos encontrar os segmentos de S que interceptam $s_i = \overline{(x, y)(x, y')}$, com $y < y'$, da seguinte forma. Seja v um nó de T. Se x não pertence a $int(v)$, então não precisamos continuar a busca na subárvore com raíz em v. Se x pertence a $int(v)$, então a reta que contém s_i intercepta todo segmento em $L_{ord}(v)$. Como $L_{ord}(v)$ está ordenado, realizamos uma busca pelo primeiro segmento, digamos s', em $L_{ord}(v)$ cujo $p_e(s_i) = (x, y)$ está ou sobre s' ou à sua direita. Essa busca pode ser realizada em tempo $O(\log n_v)$ (busca binária), onde n_v é o número de segmentos em $L_{ord}(v)$. Com isso, sabemos que o ponto extremo esquerdo de s_i está à direita de todo segmento em $L_{ord}(v)$ a partir de s'. Então, temos que listar tais segmentos enquanto o ponto extremo direito de s_i estiver ou sobre s' ou à sua esquerda. Na Figura 3.15 ilustramos um possível cenário e em seguida descrevemos um algoritmo que resolve uma s_i-consulta.

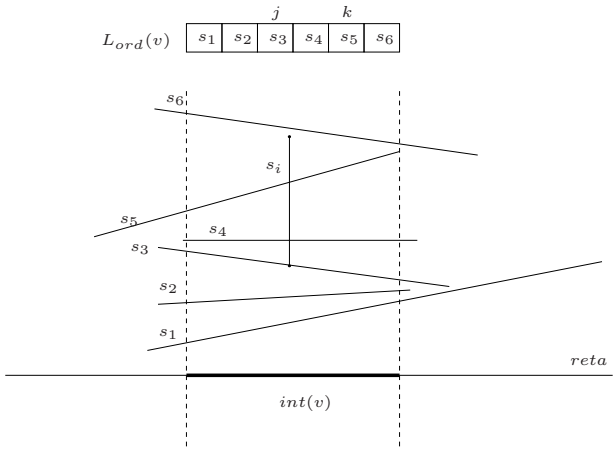


Figura 3.15: Uma busca binária em $L_{ord}(v)$ devolve o índice j. Os segmentos entre os índices j e k interceptam s_i.

Lema 28 **[8]** *Sejam S um conjunto com n segmentos no plano sem intersecção e $s_i = \overline{(x, y)(x, y')}$, com $y < y'$, um segmento vertical no plano. Seja T uma árvore de segmentos $2D$ construída sobre S. O algoritmo* ConsultaÁrvoreSegmentos *consome tempo $O(\log^2 n + k)$, onde k é o número de intervalos de S que interceptam s_i.*

Algoritmo 28 ConstroiÁrvoreSegmentos(vetor V)

Entrada: Um conjunto S com n segmentos sem intesecção através do vetor V.
Saída: Uma árvore de segmentos $2D$ T.

1. $V_{el} \leftarrow$ ConstroiIntervalosElementares(V)
2. $T \leftarrow$ ConstroiÁrvore(V_{el})
3. **para** $i \leftarrow 1$ até n **faça**
4. InsereIntervalo(T, $V[i]$)
5. **fim para**
6. **para cada** nó v de T **faça**
7. OrdenaBarra($L_{ord}(v)$)
8. **fim para**
9. **devolva** T

Algoritmo 29 ConsultaÁrvoreSegmentos(árvore_segmentos v, segmento_vertical s_i)

Entrada: Uma árvore de segmentos $2D$ não nula com raiz em v construída sobre um conjunto com n segmentos sem intersecção S e um segmento vertical $s_i = \overline{(x,y)(x,y')}$, com $y < y'$.
Saída: Uma lista L com os segmentos de S que interceptam s_i.

1. $u \leftarrow v$
2. $L \leftarrow \emptyset$
3. ▷ Encontre o menor índice j no vetor $L_{ord}(u)$ tal que $p_e(s_i) = (x,y)$ está à direita ou sobre $L_{ord}(u)[j]$.
4. $j \leftarrow$ BuscaBinária($L_{ord}(u)$)
5. **enquanto** $p_d(s_i) = (x,y')$ está à esquerda ou sobre $L_{ord}(u)[j]$ **faça**
6. Insira $L_{ord}(u)[j]$ no final de L.
7. $j \leftarrow j+1$
8. **fim enquanto**
9. **se** u não é folha **então**
10. **se** Pertence(x, $int(e(u))$) **então**
11. **devolva** $L \cup$ ConsultaÁrvoreSegmentos($e(u)$, s_i)
12. **senão**
13. **devolva** $L \cup$ ConsultaÁrvoreSegmentos($d(u)$, s_i)
14. **fim se**
15. **senão**
16. **devolva** L
17. **fim se**

Demonstração: Uma busca por x em T visita $O(\log n)$ nós que formam um caminho (C) da raiz até uma folha. Em cada nó v de C realizamos uma busca binária e percorremos k_v elementos em $L_{ord}(v)$ consumindo tempo, respectivamente, $O(\log n_v)$ e $O(k_v)$, onde n_v é o número de elementos em $L_{ord}(v)$ e k_v é o números de elementos em $L_{ord}(v)$ que atravessam s_i. Logo, em um nó consumimos tempo $O(\log n_v + k_v) = O(\log n + k_v)$. Como $\sum_{v \in C} k_v = k$ temos que o

consumo de tempo do algoritmo é $O(\log^2 n + k)$. □

Os segmentos de um conjunto S sem intersecção que interceptam um segmento horizontal podem ser obtidos de maneira similar.

Capítulo 4

Consultas em janelas

Neste capítulo vamos usar alguns algoritmos e estruturas de dados estudadas para listar segmentos em janelas. Até agora sabemos encontrar pontos em janelas, segmentos horizontais (verticais) que interceptam um segmento vertical (horizontal) e segmentos com qualquer orientação que interceptam um segmento horizontal ou vertical. Chegou o momento em que juntamos o que estudamos nos capítulos anteriores para resolver nosso problema principal: segmentos em janelas. As estruturas que usamos neste capítulo são aquelas que proporcionam um melhor desempenho, por exemplo, uma árvore limite com a técnica cascateamento fracionário possui melhor consumo de tempo para buscar pontos em janelas na fase de consultas então, nos algoritmos deste capítulo, referimos a árvores limite com camadas. Outro exemplo, uma árvore de intervalos possui como estrutura associada uma estrutura que busca pontos em janelas ilimitadas então, referimos a uma árvore de intervalos com árvores de busca com prioridade, pois essa estrutura consome menos espaço que as outras estudadas.

Vamos considerar dois casos:

- apenas segmentos horizontais e verticais: neste caso os segmentos são armazenados em três estruturas de dados, árvore limite com camadas, árvore de busca com prioridade e árvore de intervalos.
- segmentos com qualquer orientação mas sem cruzamentos (os segmentos podem se tocar): neste caso os segmentos são armazenados em duas estruturas de dados, árvore limite com camadas e árvore de segmentos.

Vamos mostrar neste capítulo como utilizar essas estruturas para responder eficientemente a consultas sobre que segmentos interceptam janelas dadas. No final do capítulo apresentamos resultados com algumas instâncias do problema.

4.1 Segmentos horizontais e verticais em janelas

Sejam S^h e S^v dois conjuntos de segmentos horizontais e verticais, respectivamente. Na primeira fase, temos que armazenar tais segmentos em estruturas de dados. Vamos armazenar os segmentos de S^h em uma árvore de intervalos horizontal com árvores de busca

com prioridade como estrutura associada denotada por T^h_{AI} e os segmentos de S^v em uma árvore de intervalos vertical com árvores de busca com prioridade como estrutura associada denotada por T^v_{AI}. Armazenamos em duas árvores limite com camadas os pontos extremos esquerdos dos segmentos de S_h e S_v e denotamos, respectivamente por T^h_{AL} e T^v_{AL}. Lembramos que definimos como ponto extremo esquerdo de um segmento vertical o ponto de menor y-coordenada. O algoritmo ConstroiEstruturasHorVer recebe os conjuntos S^h e S^v e devolve essas estruturas.

Algoritmo 30 ConstroiEstruturasHorVer (vetor S^h, vetor S^v)

Entrada: Um conjunto com n_h segmentos horizontais e um conjunto com n_v segmentos verticais através dos vetores S^h e S^v.

Saída: Uma árvore de intervalos horizontal T^h_{AI} construída sobre os elementos de S^h, uma árvore de intervalos vertical T^v_{AI} construída sobre os elementos de S^v. Ambas possuem uma árvore de busca com prioridade como estrutura associada. E duas árvores limite com camadas T^h_{AL} e T^v_{AL} construídas sobre os pontos extremos esquerdos dos elementos de S^h e S^v, respectivamente.

1. $P^h \leftarrow$ PontosExtremosEsquerdos (S^h)
2. $P^h_x \leftarrow$ Ordena_x (P^h)
3. $P^h_y \leftarrow$ Ordena_y (P^h)
4. $T^h_{AL} \leftarrow$ ConstroiÁrvoreLimiteComCamadas $(P^h_x,\ P^h_y)$
5. $P^v \leftarrow$ PontosExtremosEsquerdos (S^v)
6. $P^v_x \leftarrow$ Ordena_x (P^v)
7. $P^v_y \leftarrow$ Ordena_y (P^v)
8. $T^v_{AL} \leftarrow$ ConstroiÁrvoreLimiteComCamadas $(P^v_x,\ P^v_y)$
9. $S^h \leftarrow$ OrdenaPontoExtremoEsquerdo_x (S^h)
10. $T^h_{AI} \leftarrow$ ConstroiÁrvoreIntervalosHorizontal (S^h)
11. $S^v \leftarrow$ OrdenaPontoExtremoEsquerdo_x (S^v)
12. $T^v_{AI} \leftarrow$ ConstroiÁrvoreIntervalosVertical (S^v)
13. **devolva** $T^h_{AI},\ T^v_{AI},\ T^h_{AL},\ T^v_{AL}$

Na segunda fase, vamos responder consultas sobre que segmentos interceptam uma janela dada W. Isso se dá em 4 passos:

1 - Listamos os segmentos horizontais que interceptam s_1 de W (sua borda esquerda);

2 - Listamos os segmentos verticais que interceptam s_4 de W (sua borda inferior);

3 - Listamos os segmentos horizontais cujos extremos esquerdo estão em W.

4 - Listamos os segmentos verticais cujos extremos esquerdo estão em W.

Descrevemos este algoritmo formalmente em SegmentosHorVerEmJanela.

Perceba que alteramos levemente W nas linhas 3 e 7. Sem esta alteração o algoritmo listaria duas vezes um segmento horizontal (vertical) que intercepta s_1 (s_4) e que possui ponto extremo esquerdo em W, isto é, um segmento horizontal (vertical) que possui ponto extremo esquerdo sobre s_1 (s_4). Essa alteração deve ser feita por partes. Primeiro, antes de realizar a busca na linha 4, mudamos o valor $x(s_1)$ para $x(s_1)+\epsilon$. Devemos ter muito cuidado na escolha do valor ϵ[1] pois, um valor grande pode ocasionar perdas de segmentos que estão em W. Segundo, antes de realizar a busca na linha 8, alteramos o valor $y(s_4)$ para $y(s_4)+\epsilon$. Na Figura 4.1 vemos uma janela W ilustrada duas vezes. Na janela à esquerda buscamos por segmentos horizontais e à direita por segmentos verticais. O segmento 1 possui ponto extremo esquerdo sobre s_1. Logo, este segmento intercepta s_1, porém, o seu ponto extremo esquerdo não está na nova janela W'. O segmento 7 possui ponto extremo esquerdo sobre s_4, assim ele intercepta s_4 mas o seu ponto extremo esquerdo não está na nova janela W''.

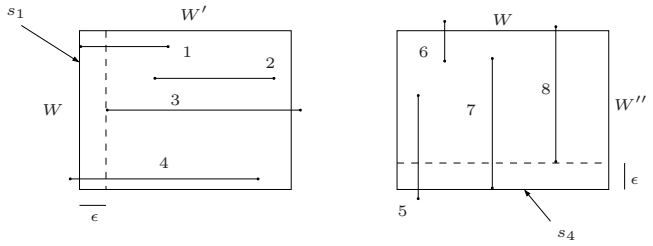


Figura 4.1: Segmentos em W, W' e W''.

Teorema 29 **[8]** *Sejam S^h e S^v conjuntos com n_h e n_v segmentos horizontais e verticais, respectivamente. Considere $n = n_h + n_v$. Os segmentos de S^h e S^v que estão em uma janela W podem ser listados consumindo tempo $O(\log^2 n + k)$, onde k é o número de segmentos de S^h e S^v em W. Para isso, usamos estruturas de dados que podem ser construídas consumindo tempo e espaço $\Theta(n \log n)$.*

Demonstração: Vamos mostrar que o algoritmo SegmentosHorVerEmJanelas consome tempo $O(\log^2 n + k)$, onde k é o número de segmentos em W. As linhas 1, 3, 5, 7 e 9 consomem tempo constante. As linhas 2 e 4 buscam pelos segmentos horizontais em W. Pelos Lemas 19 e 8, essas linhas consomem tempo $O(\log^2 n_h + k'_h) \leq c_1(\log^2 n_h + k'_h)$ e $O(\log n_h + k''_h) \leq c_2(\log n_h + k''_h)$, repectivamente, onde k'_h são os segmentos horizontais que cruzam s_1, k''_h são os segmentos horizontais que possuem ponto extremo esquerdo em W. Considere $k_h = k'_h + k''_h$. Com isso, para encontrar os segmentos horizontais em W consumimos tempo no máximo $c_1(\log^2 n_h + k'_h) + c_2(\log n_h + k''_h) \leq c_1(\log^2 n_h + k'_h) + c_2(\log^2 n_h + k''_h) \leq$

[1]Supondo que o ponto w_1 de uma janela e que os segmentos de um conjunto possuem coordenadas inteiras, podemos definir ϵ com valor inteiro 1. Em um caso mais geral, ϵ deve ser menor ou igual que a distância entre $x(w_1)$ e $x(s)$, onde s é o primeiro segmento cujo ponto extremo esquerdo está à direita da borda esquerda de uma janela. De maneira semelhante isso vale para ϵ da y-coordenada. Note que s pode ser obtido por busca binária supondo ordenado o conjunto de segmentos.

Algoritmo 31 SegmentosHorVerEmJanelas ($T^h_{AI},\ T^v_{AI},\ T^h_{AL},\ T^v_{AL},\ W$)

Entrada: Uma árvore de intervalos horizontal T^h_{AI} construída sobre os elementos de S^h, uma árvore de intervalos vertical T^v_{AI} construída sobre os elementos de S^v. Duas árvores limite com camadas T^h_{AL} e T^v_{AL} construídas sobre os pontos extremos esquerdos dos elementos de S^h e S^v, respectivamente, e uma janela W.

Saída: Uma lista l com os segmentos de S^h e S^v em W

1. $s_1 \leftarrow$ BordaEsquerda (W)
2. $l \leftarrow$ ConsultaÁrvoreIntervalosHorizontal $(T^h_{AI},\ s_1)$
3. $W' \leftarrow$ AlteraJanela$(W,\ s_1,\ \epsilon)$
4. $l \leftarrow l \cup$ ConsultaÁrvoreLimiteComCamadas $(T^h_{AL},\ W')$
5. $s_4 \leftarrow$ BordaInferior (W)
6. $l \leftarrow l \cup$ ConsultaÁrvoreIntervalosVertical $(T^v_{AI},\ s_4)$
7. $W'' \leftarrow$ AlteraJanela$(W,\ s_4,\ \epsilon)$
8. $l \leftarrow l \cup$ ConsultaÁrvoreLimiteComCamadas $(T^v_{AL},\ W'')$
9. **devolva** l

$\max\{c_1, c_2\}(2\log^2 n_h + k_h) = O(\log^2 n_h + k_h)$. As linhas 6 e 8 buscam pelos segmentos verticais em W. Pelos Lemas 19 (simétrico) e 8, essas linhas consomem tempo $O(\log^2 n_v + k'_v) \leq c_1(\log^2 n_v + k'_v)$ e $O(\log n_v + k''_v) \leq c_2(\log n_v + k''_v)$, repectivamente, onde k'_v são os segmentos verticais que cruzam s_4, k''_h são os segmentos verticais que possuem ponto extremo esquerdo em W. Considere $k_v = k'_v + k''_v$. Com isso, para encontrar os segmentos verticais em W consumimos tempo no máximo $c_1(\log^2 n_v + k'_v) + c_2(\log n_v + k''_v) \leq c_1(\log^2 n_v + k'_v) + c_2(\log^2 n_v + k''_v) \leq \max\{c_1, c_2\}(2\log^2 n_v + k_v) = O(\log^2 n_v + k_v)$. Assim, para listar os $k = k_h + k_v$ segmentos de S^h e S^v que estão em W consumimos tempo $O(\log^2 n_h + k_h) + O(\log^2 n_v + k_v) \leq c(\log^2 n_h + k_h) + c(\log^2 n_v + k_v) \leq c(2\log^2 n + k) \leq 2c(\log^2 n + k) = O(\log^2 n + k)$.

Agora vamos mostrar que o algoritmo ConstroiEstruturasHorVer consome tempo $\Theta(n \log n)$. As linhas 1 e 5 consomem tempo $O(n)$, no total. As linhas 2, 3, 6, 7, 9 e 11, consomem tempo $\Theta(n \log n)$ (ordenação). Pelo Lema 7, as linhas 4 e 8 consomem tempo $\Theta(n_h \log n_h) \leq c_1(n_h \log n_h)$ e $\Theta(n_v \log n_v) \leq c_1(n_v \log n_v)$, respectivamente e pelo Lema 18, as linhas 10 e 12 consomem tempo $\Theta(n_h \log n_h) \leq c_2(n_h \log n_h)$ e $\Theta(n_v \log n_v) \leq c_2(n_v \log n_v)$, respectivamente. Então, o consumo de tempo total dessas linhas é no máximo $c_1(n_h \log n_h) + c_1(n_v \log n_v) + c_2(n_h \log n_h) + c_2(n_v \log n_v) = (c_1 + c_2)(n_h \log n_h) + (c_1 + c_2)(n_v \log n_v) = (c_1 + c_2)(n_h \log n_h + n_v \log n_v) \leq (c_1 + c_2)(n_h \log n + n_v \log n) = (c_1 + c_2)((n_h + n_v) \log n) = (c_1 + c_2)(n \log n) = \Theta(n \log n)$. Logo, o consumo de tempo deste algoritmo é $\Theta(n \log n)$.

E agora vamos mostrar que o consumo de espaço considerando todas as árvores é $\Theta(n \log n)$. Pelo Lema 9, as árvores T^h_{AL} e T^v_{AL} consomem espaço, respectivamente, $\Theta(n_h \log n_h) \leq c_1(n_h \log n_h)$ e $\Theta(n_v \log n_v) \leq c_1(n_v \log n_v)$ e pelo Lema 20, as árvores T^h_{AI} e T^v_{AI} consomem espaço $\Theta(n_h)$ e $\Theta(n_v)$, respectivamente. Então o consumo total é no máximo, $c_1(n_h \log n_h) + c_1(n_v \log n_v) \leq c_1(n_h \log n_h + n_v \log n_v) \leq c_1((n_h + n_v) \log n) = c_1(n \log n) = \Theta(n \log n)$. □

4.2 Segmentos em janelas

Seja S um conjunto de segmentos com qualquer orientação sem intersecção. Para encontrar os segmentos que interceptam uma janela dividimos em 5 consultas:

1 - encontramos os segmentos com um ponto extremo na janela (duas consultas).

2 - encontramos os segmentos que interceptam os segmentos verticais de W (duas consultas).

3 - encontramos os segmentos que interceptam um dos segmentos horizontais de W.

Vamos construir quatro estruturas de dados: duas árvores limite com camadas (T^e_{AL} e T^d_{AL}) e duas árvores de segmentos (T^v_{AS} e T^h_{AS}). As árvores T^e_{AL} e T^d_{AL} são construídas sobre os conjuntos de pontos extremos esquerdo e direito dos segmentos em S, respectivamente, enquanto que as árvores T^v_{AS} e T^h_{AS} são construídas sobre o conjunto S. Com isso, podemos encontrar os segmentos de S que possuem pelo menos um ponto extremo em uma janela W (usando as estruturas T^e_{AL} e T^d_{AL}) e os segmentos que interceptam um segmento vertical de W (usando a estrutura T^v_{AS}) e os segmentos que interceptam um segmento horizontal de W (usando a estrutura T^h_{AS}). O algoritmo ConstroiEstruturas recebe um conjunto de segmentos com qualquer orientação sem intersecção e devolve tais estruturas.

Algoritmo 32 ConstroiEstruturas (vetor S)

Entrada: Um conjunto com n segmentos com qualquer orientação sem intersecção através do vetor S.

Saída: Uma árvore de segmentos T^v_{AS} e uma árvore de segmentos T^h_{AS} construídas sobre os elementos de S. E duas árvores limite com camadas T^e_{AL} e T^d_{AL} construídas, respectivamente, sobre os pontos extremos esquerdos e direitos dos elementos de S.

1. $P^e \leftarrow$ PontosExtremosEsquerdos (S)
2. $P^e_x \leftarrow$ Ordena_x (P^e)
3. $P^e_y \leftarrow$ Ordena_y (P^e)
4. $T^e_{AL} \leftarrow$ ConstroiÁrvoreLimiteComCamadas ($P^e_x,\ P^e_y$)
5. $P^d \leftarrow$ PontosExtremosDireitos (S)
6. $P^d_x \leftarrow$ Ordena_x (P^d)
7. $P^d_y \leftarrow$ Ordena_y (P^d)
8. $T^d_{AL} \leftarrow$ ConstroiÁrvoreLimiteComCamadas ($P^d_x,\ P^d_y$)
9. $T^v_{AS} \leftarrow$ ConstroiÁrvoreSegmentos (S)
10. $T^h_{AS} \leftarrow$ ConstroiÁrvoreSegmentosSimétrica (S)
11. **devolva** $T^v_{AS},\ T^h_{AS},\ T^e_{AL},\ T^d_{AL}$

Como descrito anteriormente, na fase de consultas, podemos listar os segmentos de S que estão em uma janela W da seguinte forma. Primeiro, buscamos pelos segmentos que possuem pelo menos um ponto extremo em W, isto é, armazenamos em uma lista l_e os

segmentos de S que possuem ponto extremo esquerdo em W usando a árvore T^e_{AL} e em uma lista l_d os segmentos que possuem ponto extremo direito em W usando a árvore T^d_{AL}. Note que os segmentos que possuem ambos pontos extremos em W aparecem em l_e e em l_d. Logo, devemos remover tais repetições. Ilustramos na Figura 4.2 alguns segmentos que podem estar nas listas l_e e l_d.

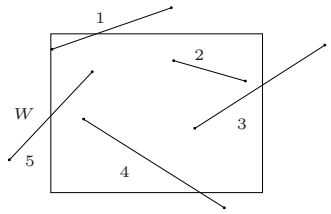


Figura 4.2: Segmentos em l_e e em l_d.

A lista l_e contém os segmentos 1, 2, 3 e 4 e a lista l_d contém os segmentos 2 e 5. Note que o segmento 2 está nas duas listas. Para remover essas repetições basta passar por cada segmento s em l_e, se s possui ponto extremo direito em W então remova s de l_e. Falta listar os segmentos que interceptam pelo menos duas bordas de W. Para isso, usamos três listas l_{s_1}, l_{s_2} e l_{s_3} para armazenar os segmentos de S que não possuem ponto extremo em W mas interceptam pelo menos duas bordas de W. Na lista l_{s_1} armazenamos os segmentos que interceptam a borda s_1 e outra borda de W. Da mesma forma, na lista l_{s_2} armazenamos os segmentos que interceptam a borda s_2 e outra borda de W e na lista l_{s_3} os segmentos que interceptam a borda s_3 e outra borda de W. Os segmentos que interceptam a borda s_4 e outra borda de W devem estar em alguma dessas listas. Mais uma vez, pode ocorrer que um segmento seja armazenado em mais de uma destas listas assim teremos repetições. Por exemplo, um segmento de S que intercepta s_1 e s_2 estará em l_{s_1} e em l_{s_2}. As Figuras 4.3, 4.4 e 4.5 ilustram possíveis repetições de um segmento em l_{s_1}, l_{s_2} e l_{s_3}.

Tratamos estas repetições da seguinte forma. Primeiro, listamos os segmentos que interceptam s_1 e s_3 (as bordas verticais de W). Removemos as repetições que ocorrem em l_{s_1} e l_{s_3} passando por cada segmento s em l_{s_1}. Se s possui um ponto extremo em W ou intercepta s_3, então removemos s de l_{s_1}. Passamos por cada segmento s em l_{s_3}, se s possui um ponto extremo em W, então removemos s de l_{s_3}. Com isso, teremos sem repetição em l_{s_1} e em l_{s_3} os segmentos que interceptam s_1 e outra borda de W e s_3 e outra borda de W. Depois disso, listamos os segmentos que interceptam s_2 (a borda superior - horizontal - de W). Passamos por cada segmento s em l_{s_2}, se s possui um ponto extremo em W ou não intercepta s_4, então removemos s de l_{s_2}. Também removemos s caso ele seja colinear a algum ponto extremo de W (os cantos de uma janela). Note que para remover as repetições nas listas visitamos cada posição de l_e, l_{s_1}, l_{s_2} e l_{s_3} consumindo tempo constante em cada visita. Então, visitamos no máximo $4k$ posições (no total) supondo que existam k segmentos de S em W. O algoritmo SegmentosEmJanelas lista os segmentos de S sem repetição que estão em uma janela W.

Algoritmo 33 SegmentosEmJanelas ($T^v_{AS},\ T^h_{AS},\ T^e_{AL},\ T^d_{AL},\ W$)

Entrada: Uma árvore de segmentos T^v_{AS} construída sobre os elementos de S, uma árvore de segmentos (simétrica) T^h_{AS} construída sobre os elementos de S. Duas árvores limite com camadas T^e_{AL} e T^d_{AL} construídas sobre os pontos extremos esquerdos e direitos, respetivamente, dos elementos de S e uma janela W.

Saída: Uma lista l com os segmentos de S em W

1. $l_e \leftarrow$ ConsultaÁrvoreLimiteComCamadas ($T^e_{AL},\ W$) ▷ extremo esquerdo em W.
2. $l_d \leftarrow$ ConsultaÁrvoreLimiteComCamadas ($T^d_{AL},\ W$) ▷ extremo direito em W.
3. $s_1 \leftarrow$ BordaEsquerda (W), $s_2 \leftarrow$ BordaSuperior (W), $s_3 \leftarrow$ BordaDireita (W)
4. $l_{s_1} \leftarrow$ ConsultaÁrvoreSegmentos ($T^v_{AS},\ s_1$) ▷ cruza borda esquerda de W.
5. $l_{s_3} \leftarrow$ ConsultaÁrvoreSegmentos ($T^v_{AS},\ s_3$) ▷ cruza borda direita de W.
6. $l_{s_2} \leftarrow$ ConsultaÁrvoreSegmentosSimétrica ($T^h_{AS},\ s_2$) ▷ cruza borda superior de W.
7. RemoveRepetições ($l_e,\ l_d,\ l_{s_1},\ l_{s_2},\ l_{s_3}$)
8. $l \leftarrow l_e \cup l_d \cup l_{s_1} \cup l_{s_2} \cup l_{s_3}$
9. **devolva** l

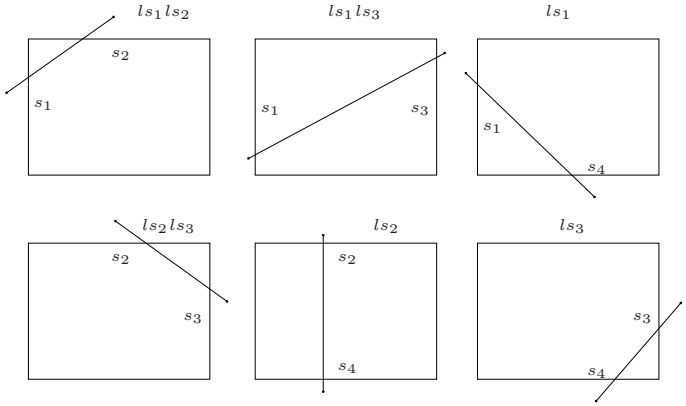


Figura 4.3: Um segmento intercepta duas bordas e é listado no máximo duas vezes.

Teorema 30 **[8]** *Seja S um conjunto com n segmentos com qualquer orientação mas sem intersecção. Os segmentos de S que estão em uma janela W podem ser listados consumindo tempo $O(\log^2 n + k)$, onde k é o número de segmentos de S em W. Para isso, usamos estruturas de dados que podem ser construídas consumindo tempo $O(n \log^2 n)$ e espaço $\Theta(n \log n)$.*

Demonstração: Vamos mostrar que o algoritmo SegmentosEmJanelas consome tempo $O(\log^2 n + k)$. As linhas 3, 8 e 9 consomem tempo constante. Pelo Lema 8, as linhas 1 e 2 consomem tempo $O(\log n + k_e)$, e $O(\log n + k_d)$, respectivamente, onde k_e são os segmentos de S que possuem ponto extremo esquerdo em W e k_d são os segmentos que possuem ponto extremo direito em W. Pelo Lema 28, as linhas 4, 5 e 6 consomem tempo $O(\log n + k_{s_1})$, $O(\log n + k_{s_3})$ e $O(\log n + k_{s_2})$, respectivamente, onde k_{s_i} são os segmentos de S que atravessam

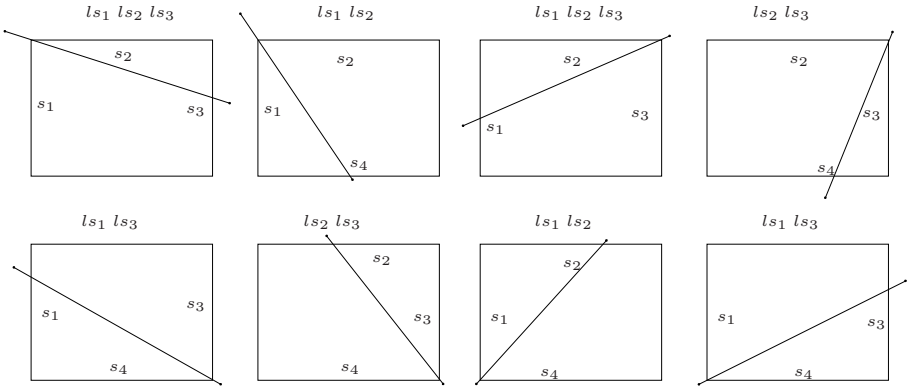


Figura 4.4: Um segmento intercepta três bordas e é listado no máximo três vezes.

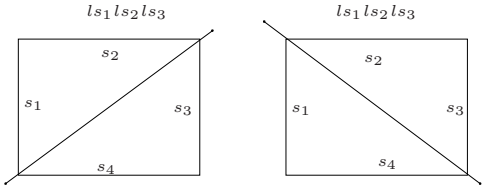


Figura 4.5: Um segmento intercepta quatro bordas e é listado três vezes.

a borda s_i de W. A linha 7 consome tempo no máximo $k_e + k_{s_1} + k_{s_2} + k_{s_3} \leq 4k = O(k)$. Logo, o consumo de tempo do algoritmo é $O(\log^2 n + k)$.

Agora vamos mostrar que o algoritmo ConstroiEstruturas consome tempo $O(n \log^2 n)$. As linhas 1, 2, 6 e 7 consomem tempo $\Theta(n \log n)$ (ordenação). As linhas 1 e 5 consomem tempo $\Theta(n)$. Pelo Lema 7, as linhas 4 e 8 consomem tempo $\Theta(n \log n)$. Pelo Lema 27, as linhas 9 e 10 consomem tempo $O(n \log^2 n)$. E a linha 11 consome tempo constante. Portanto, o consumo de tempo é $O(n \log^2 n)$.

Por fim, vamos mostrar que o consumo de espaço das estruturas construídas pelo algoritmo ConstroiEstruturas, no total, é $\Theta(n \log n)$. Pelo Lema 24, T^v_{AS} e T^h_{AS} consomem espaço $O(n \log n)$ e pelo Lema 9, T^e_{AL} e T^d_{AL} consomem espaço $\Theta(n \log n)$. Portanto, o consumo de espaço total é $\Theta(n \log n)$. □

Podemos enfraquecer a hipótese sobre o conjunto S permitindo que os segmentos de S se toquem. Com isso, podem ocorrer os casos ilustrados na Figura 4.6.

Podemos usar as estruturas árvore de segmentos e árvore limite (com camadas) para encontrar tais segmentos que estão em uma janela, porém, um pré-processamento sobre os pontos extremos de cada segmento deve ser feito, pois, uma árvore limite é construída sobre um conjunto de pontos não coincidentes. Para resolver esse problema interpretamos cada

Figura 4.6: Um ponto extremo pode tocar em um outro segmento (em um extremo ou internamente).

ponto extremo de um segmento como único apesar deste possivelmente ser ponto extremo de outros segmentos.

No trabalho de Derraik [9] podemos ver que alguns bancos de dados cartográficos armazenam um conjunto de segmentos como linhas poligonais (veja a Figura 4.7).

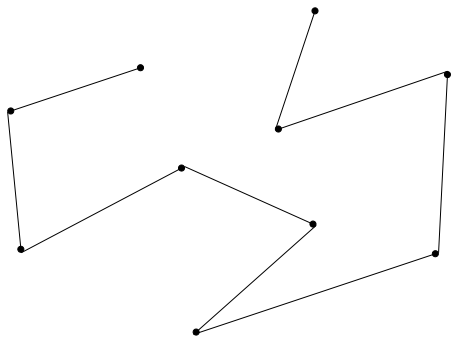


Figura 4.7: Uma linha poligonal.

A Figura 4.8 mostra o mapa dos municípios do Brasil[2] constituído por 56.948 segmentos armazenados em um banco de dados cartográfico. Na primeira figura, todos os segmentos são visualizados. Na segunda, os segmentos que estão dentro de uma janela estão em destaque. Para obter os segmentos dentro da janela na segunda figura, usamos alguns algoritmos descritos neste trabalho, como o algoritmo que constroi uma árvore de segmentos, que constroi uma árvore limite e os algoritmos que buscam segmentos em uma janela usando estas estruturas.

4.3 Consumo de tempo e espaço na prática

Escrevemos os algoritmos deste trabalho na linguagem $C++$ construindo assim uma biblioteca de consultas em janelas. Medimos o consumo de tempo da construção de cada estrutura (executando 50 vezes cada construção) em um computador com a seguinte configuração: AMD Turion 64 bits X2, 2GB de memória RAM e sistema operacional linux

[2]Agradecemos ao professor Luiz Henrique de Figueiredo que nos forneceu esse mapa.

AMD64 distribuição ubuntu 7.04. Na Figura 4.9 visualizamos o consumo de tempo da construção das árvores: de intervalos ($+$), limite com camadas ($\times$), limite ($*$) e de busca com prioridade ($\square$). Observe que o comportamento da nossa implementação, para os casos citados, é limitado por uma função em $\Theta(n \log n)$ e parece que sempre será limitado por ela, como esperado. A árvore de intervalos considerada tem como estrutura associada árvores de busca com prioridade.

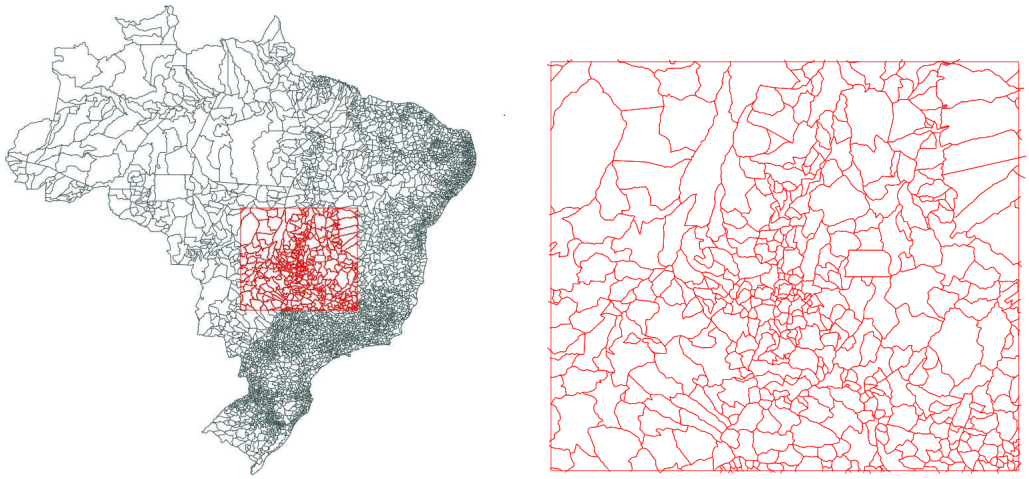


Figura 4.8: Mapa dos municípios do Brasil. Em destaque estão os segmentos dentro de uma janela.

Na Figura 4.10 visualizamos o consumo de tempo da construção da árvore de segmentos. Perceba que ela parece sempre estar limitada por uma função em $O(n \log^2 n)$, como esperado.

O consumo de tempo das consultas são todos muito rápidos, no máximo 1 segundo para um exemplo com 1.024.000 pontos ou segmentos, todos eles em uma janela.

O consumo de espaço das estruturas implementadas pode ser visualizado na Figura 4.11. Lembre que o consumo de espaço de uma árvore de segmentos é $O(n \log n)$, de uma árvore limite com camadas e uma árvore limite (simples) é $\Theta(n \log n)$ e de uma árvore de intervalos (com árvores de busca com prioridade como estrutura associada) e uma árvore de busca com prioridade é $\Theta(n)$.

Esta nossa análise empírica dos algoritmos mostrou os resultados esperados.

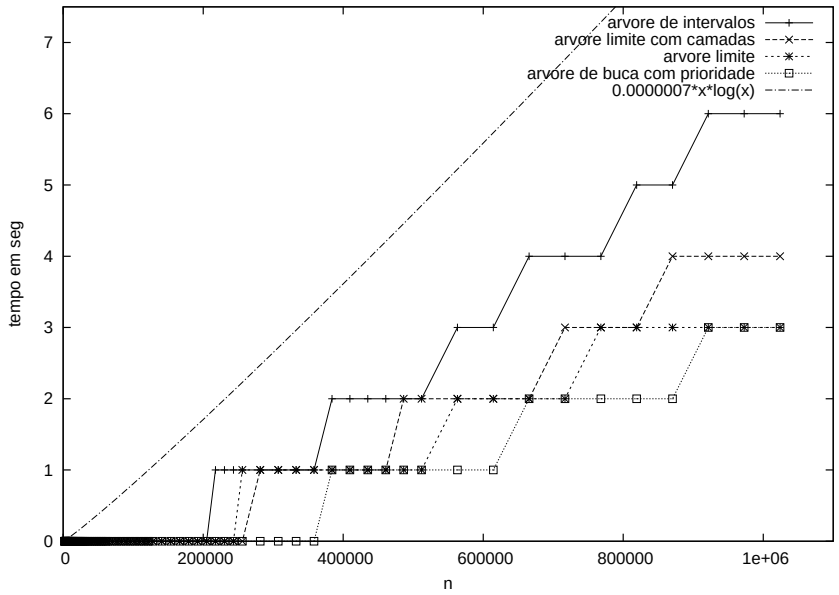


Figura 4.9: Consumo de tempo da construção das árvores de intervalos, limite com camadas, limite (simples) e de busca com prioridade.

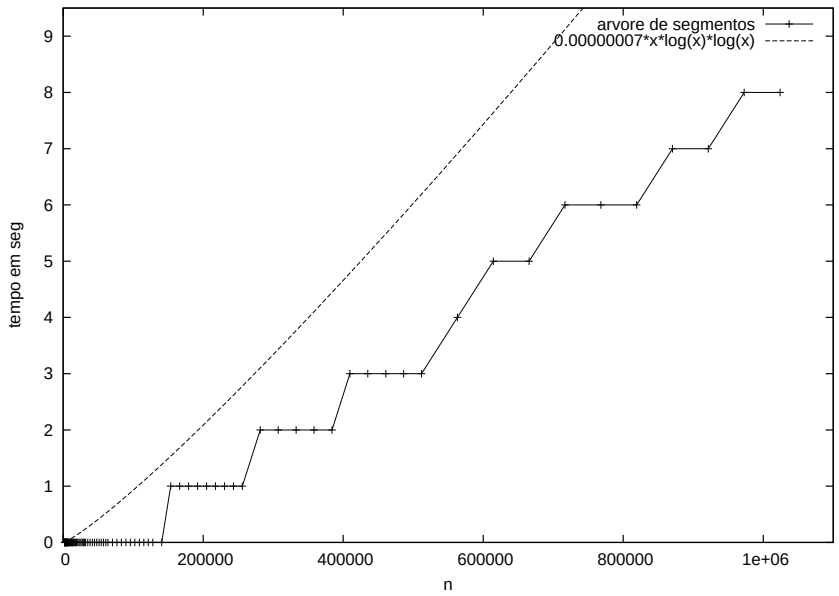


Figura 4.10: Consumo de tempo da construção da árvore de segmentos.

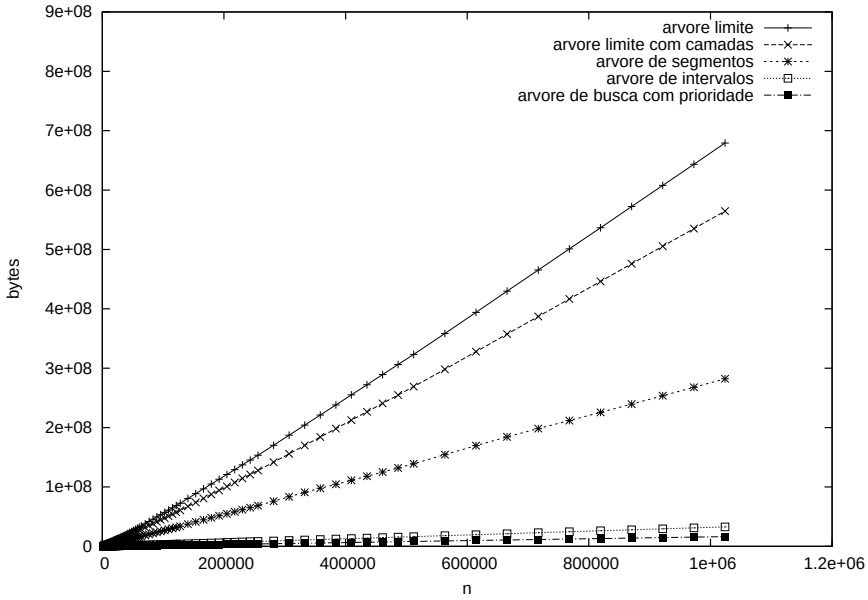


Figura 4.11: Consumo de espaço das estruturas implementadas.

Considerações finais

Nós descrevemos alguns algoritmos para resolver o problema de consultas em janelas. Analisamos o consumo de tempo de cada algoritmo e de espaço de cada estrutura. Resumimos esses consumos na Tabela 4.1, onde n é o número de segmentos (ou pontos) de um conjunto S e k é o número de elementos de S que estão em uma janela.

Dimensão	Estrutura de dados	Construção	Consulta	Espaço
1D	Árvore limite	$\Theta(n \log n)$	$O(\log n + k)$	$\Theta(n)$
	Árvore de intervalos com vetores ordenados	$\Theta(n \log n)$	$O(\log n + k)$	$\Theta(n)$
	Árvore de segmentos	$\Theta(n \log n)$	$O(\log n + k)$	$O(n \log n)$
2D	Árvore limite	$\Theta(n \log n)$	$O(\log^2 n + k)$	$\Theta(n \log n)$
	Árvore limite com camadas	$\Theta(n \log n)$	$O(\log n + k)$	$\Theta(n \log n)$
	Árvore de busca com prioridade	$\Theta(n \log n)$	$O(\log n + k)$	$\Theta(n)$
	Árvore de intervalos com árvore de busca com prioridade	$\Theta(n \log n)$	$O(\log^2 n + k)$	$\Theta(n)$
	Árvore de segmentos	$O(n \log^2 n)$	$O(\log^2 n + k)$	$O(n \log n)$

Tabela 4.1: Consumo de tempo dos algoritmos e de espaço das estruturas estudadas.

Comprovamos que a construção de algumas estruturas de dados que armazenam segmentos e pontos no plano (ou intervalos e pontos na reta) é feita rapidamente. A maioria delas, como por exemplo, árvore limite (simples), árvore limite com camadas, árvore de busca com prioridade e árvore de intervalos, consomem tempo $\Theta(n \log n)$ enquanto que uma árvore de segmentos é construída consumindo tempo $O(n \log^2 n)$[3]. Comprovamos que é consumido pouco espaço por essas estruturas, da ordem de $\Theta(n)$ para árvores de busca com prioridade e árvore de intervalos e de $\Theta(n \log n)$ para as demais estruturas. Sem dúvidas as estruturas árvore limite com camadas e árvore de segmentos são as estruturas mais poderosas e que minimizam restrições sobre um conjunto de segmentos e pontos. A árvore de intervalos possui

[3]Os autores do livro [8] dizem que uma árvore de segmentos pode ser construída em tempo $O(n \log n)$ mas não sabemos como obter esse consumo de tempo.

a vantagem de consumir espaço $\Theta(n)$, porém, ela restringe muito o conjunto de segmentos da entrada que devem ser horizontais ou verticais.

Desenvolvemos uma biblioteca com os algoritmos que constroem as estruturas árvore limite, árvore limite com camadas, árvore de busca com prioridade, árvore de intervalos com árvores de busca com prioridade e árvore de segmentos, e com os algoritmos que usam tais estruturas e resolvem o problema de consultas em janelas.

Em seguida, citamos alguns problemas em aberto que se relacionam com o nosso trabalho e com a área de geometria computacional.

PROBLEMAS EM ABERTO: Como podemos estender a estrutura árvore de busca com prioridade para resolver problemas de consultas em dimensões maiores? [8] Como podemos estender a estrutura árvore de intervalos para resolver problemas de consultas em dimensões maiores? [8] Existe uma estrutura de dados que usa $o(n \log n)$ espaço e que resolve uma generalização de consultas em janelas ilimitadas 2-lados em tempo $O(\log n)$? (essa generalização pode ser obtida em [14] e uma janela ilimitada 2-lados é formada somente pelo ponto w_1 e por duas semirretas, uma vertical e outra horizontal começando em w_1) [11]. Neste mesmo trabalho, [11], são listados alguns problemas em aberto considerando uma variação do problema consulta em janelas. Outros problemas em aberto da área de geometria computacional podem ser vistos em [7, 15, 17, 27].

Por fim, podemos citar alguns trabalhos que dão continuidade a este como por exemplo, usar outras estruturas de dados como *quadtree* e *kd-tree* para resolver o problema consultas em janelas, estudar e implementar as versões para dimensões maiores das estruturas árvore limite e árvore de segmentos, resolver consultas em janelas considerando pontos em movimentos, adequar as estruturas estudadas para suportar adição e remoção de pontos e segmentos mantendo a estrutura correta e balanceada e resolver consultas em diferentes formas geométricas como círculos, triângulos e polígonos.

Referências Bibliográficas

[1] G. M. Adel'son-Vel'skii and E. M. Landis, *An algorithm for the organization of information*, Soviet Mathematics Doklady (1962), 3:1259–1263.

[2] R. Bayer, *Symmetric binary B-trees: Data structure and maintenance algorithms*, Acta Informatica (1972), 1:290–306.

[3] J. L. Bentley, *Solutions to Klee's rectangle problems*, Tech. report, Carnegie-Melon Univ., Pittsburgh, PA, 1977.

[4] ———, *Decomposable searching problems*, Inform. Process. Lett., 8 (1979), 244–251.

[5] B. Chazelle, *Lower bounds for orthogonal range searching: part I. The reporting case*, J. ACM **37** (1990), no. 2, 200–212.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms, 2nd edition*, MIT Press, McGraw-Hill Book Company, 2001.

[7] H. T. Croft, K. J. Falconer, and R. K. Guy, *Unsolved problems in geometry*, Springer-Verlag, New York, 1994, Corrected reprint of the 1991 original [MR 92c:52001], Unsolved Problems in Intuitive Mathematics, II. MR 95k:52001

[8] M. de Berge, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, second ed., Springer, 2000.

[9] A. L. B. Derraik, *Um estudo comparativo de representações de multi-resolução para linhas poligonais*, Master's thesis, Pontifícia Universidade Católica do Rio de Janeiro, 1997.

[10] D. Dobkin and R. Lipton, *On the complexity of computations under varying set of primitives*, Journal of Computer and System Sciences **18** (1979), no. 18, 86–91.

[11] V. Dujmović, J. Howat, and P. Morin, *Biased range trees*, SODA '09: Proceedings of the Nineteenth Annual ACM -SIAM Symposium on Discrete Algorithms (Philadelphia, PA, USA), Society for Industrial and Applied Mathematics, 2009, pp. 486–495.

[12] H. Edelsbrunner, *Dynamic data structures for orthogonal intersection queries*, Tech. report, Tech. Univ. Graz, 1980.

[13] P. Feofiloff, Y. Kohayakawa, and Y. Wakabayashi, *Uma introdução sucinta à Teoria dos Grafos*, Escrito para um minicurso na II Bienal da SBM realizado em Salvador, Outubro 2004.

[14] M. L. Fredman, *A lower bound on the complexity of orthogonal range queries*, J. ACM **28** (1981), no. 4, 696–705.

[15] V. Klee and S. Wagon, *Old and new unsolved problems in plane geometry and number theory*, The Mathematical Association of America, 1996.

[16] D. E. Knuth, *The art of computer programming*, vol. 3 sorting and searching, Addison-Wiley, Reading, Mass., :, 1973.

[17] D. T. Lee and F. P. Preparata, *Computational Geometry – A Survey*, IEEE Trans. Comput. **33** (1984), no. 12, 1072–1101.

[18] D. T. Lee and C. K. Wong, *Quintary trees: a file structure for multidimensional database sytems*, ACM Trans. Database Syst. **5** (1980), no. 3, 339–353.

[19] G. S. Lueker, *A Data Structure for Orthogonal Range Queries*, FOCS, IEEE, 1978, pp. 28–34.

[20] E. M. McCreight, *Efficient algorithms for enumerating intersecting intervals and rectangles*, Tech. report, Xerox Palo Alto Res. Center, 1980.

[21] ———, *Priority search trees*, SIAM J. Comput. (1985), 257–276.

[22] P. F. Preparata and I. M. Shamos, *Computational geometry: An introduction (monographs in computer science)*, Springer, August 1985.

[23] M. O. Rabin, *Proving simultaneous positivity of linear forms*, Journal of Computer and System Sciences **6** (1972), no. 6, 639–650.

[24] E. M. Reingold, *On the optimality of some set algorithms*, J. ACM **19** (1972), no. 4, 649–659.

[25] J. B. Saxe, *On the number of range queries in k-space*, Discrete Applied Mathematics 1 (1979), 217–225.

[26] R. E. Tarjan, *A class of algorithms which require nonlinear time to maintain disjoint sets*, Journal of Computer and System Sciences **18** (1979), no. 2, 110–127.

[27] J. Urrutia, *Open problems in computational geometry*, LATIN '02: Proceedings of the 5th Latin American Symposium on Theoretical Informatics (London, UK), Springer-Verlag, 2002, pp. 4–11.

[28] D. E. Willard, *Predicate-oriented database search algorithms*, Ph.D. thesis, Harvard Univ., 1978.

[29] ———, *The super-B-tree algorithm*, Tech. report, Harvard Univ., Cambridge, MA, 1979.

[30] C. J. Van Wyk, *Data Structures and C Programs*, Addison-Wesley Publishing Co., Inc., New Jersey, USA, 1988.