Lucas Rodrigo da Silva Suppes

Algoritmos e estruturas de dados para visualização de polígonos em aplicações de tempo real

Brasil

2021

Lucas Rodrigo da Silva Suppes

Algoritmos e estruturas de dados para visualização de polígonos em aplicações de tempo real

Estruturas de dados eficientes para consultas geométricas para aplicações interativas em tempo real. UFSC - Ciências da computação

Universidade Federal de Santa Catarina

Instituto de Estatística e Informática - Centro de Tecnologia

Orientador: Prof. Álvaro Junio Pereira Franco, Dr.

Brasil

2021

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Suppes, Lucas Rodrigo da Silva
Algoritmos e estruturas de dados para visualização de polígonos em aplicações de tempo real : Estruturas de dados eficientes para consultas geométricas para aplicações interativas em tempo real. / Lucas Rodrigo da Silva Suppes ; orientador, Álvaro Junio Pereira Franco, 2021.
67 p.

Tese (doutorado) - Universidade Federal de Santa Catarina, , Programa de Pós-Graduação em , Florianópolis, 2021.

Inclui referências.

1. . 2. Estruturas de dados. 3. Geometria computacional. 4. Jogos. 5. Árvores. I. Franco, Álvaro Junio Pereira. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em . III. Título.

Lucas Rodrigo da Silva Suppes

Algoritmos e estruturas de dados para visualização de polígonos em aplicações de tempo real

Estruturas de dados eficientes para consultas geométricas para aplicações interativas em tempo real. UFSC - Ciências da computação

Trabalho submetido à banca. Florianópolis, 21 de maio de 2021:

Prof. Álvaro Junio Pereira Franco, Dr.
Orientador

Brasil
2021

Este trabalho é dedicado às crianças adultas que,
quando pequenas, sonharam em se tornar cientistas.

Agradecimentos

À minha mãe Cleonice que em nenhum momento teve duvidas de minha capacidade e sempre incentivou que eu trilhasse meu rumo qualquer que fosse a direção. Sempre confiou na minha bússola e compasso moral que me guiaram até onde me encontro.

À meu pai Bernardino que sem seu sopro e força jamais teria ido longe. *“ Vós sois os arcos dos quais vossos filhos são arremessados como flechas vivas.” - Khalil Gibran*

Ao meu orientador Álvaro por esse ano de trabalho incessante e seu grande carinho e suporte para que eu conseguisse executar esta obra.

À cada amigo e colega que conheci durante essa jornada. Dos que ficaram e dos que se foram. Cada letra deste trabalho tem um pouco de cada um.

À cada professor e profissional desta universidade que me passou um pouco de seu conhecimento, e em especial, à aqueles poucos professores que conseguiram de forma especial cativar minha curiosidade para esta sublime ciência.

À flor que sem ela este trabalho jamais teria vindo a fruição. Por sempre ter acreditado em mim mesmo quando nem eu mesmo acreditava; por cada sorriso mesmo quando eu era só lagrimas; por ter sido meu alicerce quando eu era fraco.

À Ele que dá-me forças, guia-me, protege-me e orienta-me sempre.

"Um processo computacional é de fato muito parecido como a ideia de um espirito para os magos. Não pode ser visto ou tocado. Não é composto de matéria. Entretanto, é real. Pode realizar trabalho intelectual. Pode responder questões. Pode afetar o mundo entregando dinheiro em um caixa de banco ou controlando um braço robótico em uma fábrica. Nós conjuramos programas como os magos conjuram suas magias. Eles são cuidadosamente compostos de expressões simbólicas em linguagens de programação arcanas e esotéricas que descrevem o intento para os processos executarem ."
(Structure and Interpretation of Computer Programs, pagina 2)

"Ainda que eu andasse pelo vale da sombra da morte,
não temeria mal algum, porque tu estás comigo;
a tua vara e o teu cajado me consolam.
(Bíblia Sagrada, Salmos 23:4)

Resumo

Este trabalho apresenta um estudo de algumas estruturas de dados e técnicas para processamento de janelas. Estudamos maneiras de estruturar objetos geométricos de tal forma que consultas geométricas são respondidas com eficiência. As estruturas de dados que estudamos foram Árvore KD, Árvore de Alcance, Árvore de Intervalos e Árvore de Segmentos. Estamos interessados em consultar objetos no plano portanto este trabalho preparou as estruturas de dados para recuperar objetos no plano. As estruturas de dados e algoritmos de construção e consulta foram implementados. Por fim, utilizamos nossas implementações em uma aplicação que processa pontos e segmentos no plano. Demonstramos que são estruturas eficientes para consultas espaciais de pontos em aplicações com restrições temporais.

Palavras-chave: Estruturas de Dados. Árvores. Geometria Computacional.

Abstract

This work presents a study of some data structures and techniques to process windows. We study ways to structure geometric objects in such a way that queries in windows are quickly answered. The data structures that we study were KD Tree, Range Tree, Interval Tree and Segment Tree. This work used all the data structures on the plan. The data structures and the algorithms of construction and query were implemented. Finally, we used our implementation in an application that processes points and segments on the plan. We showed that the structures are efficient for query spatial points for time constraint applications.

Keywords: Data Structures. Trees. Computational Geometry.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

A cada nova geração de consoles, desenvolvedores e artistas gráficos aumentam a qualidade gráfica incrementando a contagem de polígonos em cada objeto 3D[1]. Com objetos tridimensionais e cenas cada vez mais complexas, sendo jogos aplicações de tempo real, se faz necessário que os algoritmos realizem consultas a esses objetos rapidamente pois o tempo da consulta não pode extrapolar o tempo limite esperado para desenhar cada quadro da aplicação. Este trabalho visa apresentar estruturas de dados conhecidas a fim de produzir consultas aos objetos e figuras geométricas de forma eficiente. Sabemos que cada objeto tridimensional é composto de figuras simples como triângulos, vértices e arestas (VRIES, 2015). Portanto, se pudermos consultar tais figuras de forma eficiente, poderemos estender para uma aplicação tridimensional pois podemos consultar os objetos tridimensionais por seus vértices e arestas. O nosso trabalho considera a decomposição de polígonos sobre estes dois objetos geométricos mais simples: pontos e segmentos de reta.

Ao longo deste trabalho vimos estruturas de dados para consultar pontos bidimensionais que podem ser facilmente estendidas para três dimensões e utilizadas assim para jogos tridimensionais. O objetivo de consultas rápidas pode ser variado: consultar quais objetos devem ser desenhados sem o custo de consultar linearmente quais objetos estão presentes dentro do raio da câmera permitindo construção de grandes cenas sem um grande custo de consulta. Podemos utilizar também para otimizar algoritmos que preenchem triângulos [2] informando para o pipeline gráfico apenas os vértices dentro da consulta (VRIES, 2015). Estudamos árvore KD que subdivide o espaço em cada K dimensões permitindo consultas espaciais.Em seguida, estudamos árvores de alcance, sendo esta uma árvore binária balanceada multinível. Isto é, cada nó da árvore tem uma outra árvore anexa representando outra dimensão. Sua construção é pensada em fazer consultas em intervalos, sendo uma estrutura eficiente para consultas em janela. Em seguida vemos árvores de intervalo que permite consultas de segmentos paralelos aos eixos de uma janela. Podendo esta ser utilizada para consulta de objetos considerando apenas suas grandezas espaciais. E por fim, estudaremos árvores de segmentos, que permite a consulta de segmentos com quaisquer inclinação.

As nossas implementações foram aplicadas em dois cenários. No primeiro, temos um conjunto de pontos no plano que representam objetos em um mapa e um outro ponto que percorre esse mapa e assim, deslocando a janela pelo mapa. No segundo, temos um conjunto de segmentos no plano que descrevem o mapa do Brasil, fixamos um ponto no

[1] A cada geração a contagem de polígonos chega a quintuplicar https://blog.playstation.com/2019/12/16/the-polygonal-evolution-of-5-iconic-playstation-characters/

[2] Podemos enviar para a V-RAM apenas o que a nossa consulta retornar https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

mapa, definimos uma janela a partir deste ponto e desenhamos somente aqueles segmentos do mapa que estão na janela.

Para introduzirmos o assunto de consulta por figuras geométricas iremos passar uma visão geral das estruturas estudadas e sobre computação gráfica, pois este é o nosso objetivo fim.

1.1 Árvores binária de busca balanceada

Neste trabalho estudamos árvores binária de busca especiais para objetos geométricos. Uma árvore binária é uma estrutura de dados que possui nós e arestas. Alguns nós são internos, enquanto que outros são folhas. Os nós internos possuem no máximo dois filhos e os nós folhas não possuem filhos. Supondo chaves armazenadas em cada nó e em árvores de busca, os nós das subárvores à esquerda tem valores menores ou iguais à chave do nó pai. E, respectivamente, as subárvores à direita tem chaves maiores que a do nó pai. O ideal em uma árvore binária de busca é estruturar dados de forma que seja eficiente uma consulta por uma chave. A construção dá-se dado um conjunto de valores $S = x_1, x_2, ...x_n$, constrói-se um nó e chama-se raiz v. Guardamos o primeiro valor x_1 neste nó. Obtemos o próximo valor x_2 e comparamos com o valor da raiz v, caso seja $x_2 < v = x_1$, criamos um novo nó e guardamos o valor de x_2 neste nó e inserimos na subárvore à esquerda de v. Reciprocamente caso operamos com $x_2 > v = x_1$, inserimos o novo nó em uma subárvore à direita de v. Obtemos o próximo valor x_3. Suponha que este é $x_3 > v = x_1$. Portanto inserimos à direita de v. Seguimos assim até x_n.

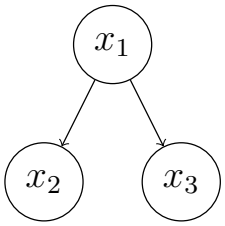


Figura 1 – Árvore binária exemplo

É importante dizer que a construção de uma árvore binária de busca desta forma não garante que uma consulta na árvore será feita de forma eficiente.Dizemos que um nó v é alcançável a partir de um nó v' quando há ao menos um caminho entre dois nós v e v'. Chamaremos a altura de uma árvore binária como o comprimento do maior caminho da raiz da árvore até a folha alcançável a partir da raiz. Portanto, para construir árvores binárias balanceadas existem outros algoritmos que podem rotacionar subárvores de tal forma que garanta sempre uma altura ótima para a árvore. Logo descreveremos sobre árvores binárias de busca balanceadas.

A consulta em uma árvore pode ser feita de através de chamadas recursivas, onde dado um valor de consulta s, compara o valor do nó v com o valor s. Caso $s > v$, isto implica que a consulta deve seguir pela subárvore à direita de v. Simetricamente, caso $s < v$ implica que a consulta deve seguir pela subárvore esquerda, fazendo uma chamada recursiva com os respectivos nós das subárvores determinadas pela comparação. A condição de parada da consulta é se $s = v$, ou se a árvore é vazia.

Uma árvore binária de busca é dita balanceada quando a altura da árvore é da ordem $\log_2 n$ para uma árvore com n elementos. Uma árvore binária de busca não necessariamente está balanceada. Uma árvore que não esteja balanceada pode ter, nos piores casos de uma consulta, tempo de consulta igual o de uma lista. Ou seja, uma consulta linear e portanto perdendo as qualidades de uma árvore de busca. Existem técnicas para evitar que isso aconteça. A que usamos para atingir este fim neste estudo é pre-ordenar o conjunto de valores (chaves) a serem inseridos na árvore. Isto garante que a árvore que será construída será balanceada (CORMEN et al., 2002). A ordenação garante que as folhas da árvore $v_1, v_2, \ldots, v_n$, quando vistas da esquerda para a direita, atendam ao seguinte $v_1 \leq v_2 \leq \cdots \leq v_n$.

1.2 Computação Gráfica

Afim de compreendermos as implicações e como podemos utilizar as estruturas usadas em jogos e em aplicações gráficas de tempo real queremos ter uma ideia básica de onde podemos utilizar as estruturas de dados. O componente central de um sistema de gráficos em tempo real é o que se chama de pipeline gráfico representado na Figura 2.

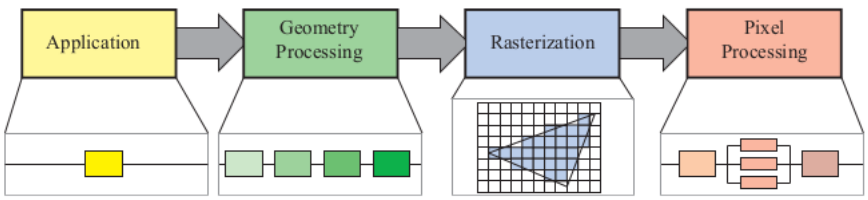


Figura 2 – (TOMAS et al., 2018) Pipeline gráfico para renderização de imagens em tempo real

Sua principal função é desenhar uma imagem bidimensional na tela, dado objetos tridimensionais, uma câmera, fontes de iluminação, etc. O tempo de desenho de cada quadro é expresso por *Quadros Por Segundo* (QPS), isto é, a quantidade de quadros produzidos a cada segundo. No primeiro estágio, *aplicação* é onde o programa lógico da aplicação está sendo executado, por exemplo, checagem de colisão, simulação de física, animação e outros. O estágio seguinte é o *processamento de geometria* que lida com as transformações geométricas, projeções, como um objeto (*vertex shader*) vai ser desenhado

e onde vai ser desenhado. O estágio de *rasterização* toma como entrada três vértices de um triângulo, encontra todos os pixel que definem este triângulo e executa um programa (*fragment shader*) para cada pixel determinando sua cor. Finalmente, o *processamento por pixel* executa um programa por pixel para determinar se é visível ou não, além de outros artifícios para coloração. O desenvolvedor tem total domínio dos triângulos que serão desenhados já no estágio de aplicação podendo realizar consultas em nível de aplicação e entregar para o estágio de *processamento de geometria* somente os pontos que consultamos.

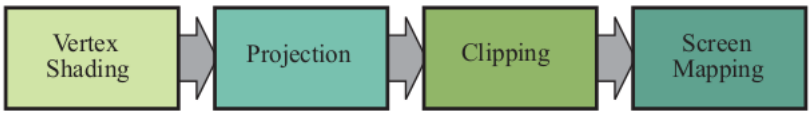


Figura 3 – Estágio de geometria subdividido

Sabemos que o estágio de processamento de geometria executa um algoritmo chamado *recorte (clipping)*, que recorta os segmentos que estão fora da janela visível, desenhando somente o que está dentro da janela. Porém, o algoritmo será aplicado a todos os vértices que foram enviados para o pipeline. Além disso, o maior gargalo em uma das partes do pipeline atrasa todo o resto do processamento gráfico. Portanto, com nossas estruturas de dados conseguimos deixar eficiente não só o processamento de geometria por ter os pontos dentro da janela a ser trabalhada como por consequência otimizamos o resto de todo o pipeline.

1.3 Objetivos

O objetivo deste trabalho é encontrar e provar a eficácia de estruturas de dados para consultas espaciais visando aplicações em tempo real gráficas. Dado o pipeline gráfico precisamos enviar para o pipeline gráfico apenas os vértices dos polígonos que estão dentro da janela de consulta. Otimizando, portanto, o pipeline gráfico e a aplicação em si caso precise efetuar cálculos sobre os polígonos que estiverem sendo exibidos. Um segundo objetivo deste trabalho é mostrar que existe classe de problemas de consulta sobre polígonos onde há uma grande densidade de segmentos. Para tal classe de problemas o objetivo é mostrar que sem as estruturas estudadas a aplicação não atinge as restrições temporais. E, portanto, apresentar uma solução para esta classe de problemas.

1.3.1 Objetivos específicos

Inicialmente o alvo de estudo são árvores KD que repartem o espaço em cada nível da árvore alternando eixos. Temos com esta portanto qual seria o pior caso de uma consulta em janela. Em seguida uma árvore mais apta para consultas em janela com um

tempo de consulta mais eficiente. Por fim consultas por segmentos em janela. Inicialmente é demonstrado um caso mais restrito em que os segmentos são necessariamente paralelos aos eixos de consulta. E é concluído o estudo com uma estrutura mais completa que permite consultar segmentos com quaisquer inclinação em relação ao eixo de consulta. Comparando assim por fim as técnicas de consulta de segmentos no plano e suas respectivas restrições.

1.4 Metodologia

Fizemos um levantamento bibliográfico de estruturas de geometria computacional que poderiam auxiliar na construção de jogos e quais classes de problemas seriam relevantes. Para cada estrutura estudada foi implementada sob a linguagem Python a fim de validar-las. Escolhemos as mais eficientes e mais abrangentes para realizar nossos experimentos. Para consulta de pontos escolhemos as árvores de alcance. Para consulta de segmentos com qualquer inclinação em relação aos eixos de consulta escolhemos as árvores de segmentos. Aplicamos as estruturas escolhidas em duas aplicações para validarmos a sua eficiência. Uma aplicação de consulta de pontos em janela e uma aplicação que consulta os segmentos do mapa do Brasil.

1.5 Notação

Como estamos sempre trabalho com árvores binárias neste trabalho, vamos introduzir as notações utilizadas. Denotaremos um nó de uma árvore binária por nó$_i^h$ onde h é o nível da árvore e i é o i-ésimo nó da esquerda para a direita nesta árvore. Chamaremos de val(nó) o valor armazenado em um nó de uma árvore binária. Para consultarmos por intervalos unidimensionais, definimos uma janela J como um intervalo unidimensional $J = [x, x']$ onde $[x, x']$ é um intervalo. Definimos uma janela bidimensional J como o produto cartesiano de dois intervalos: $J = [x, x'] \times [y, y']$.

2 Algoritmos e estruturas de dados para consulta de pontos no plano

Nesta seção, veremos formas de construir estruturas de dados para um conjunto de pontos no plano. Consideramos que os pontos são fixos. Para cada estrutura construída, veremos também como utilizá-la para consultar pontos dentro de uma janela.

2.1 Árvore KD

Uma árvore KD (BENTLEY, 1975) é uma árvore binária onde cada folha é um ponto *k-dimensional*. Cada nó não-folha guarda um valor v em uma dimensão d. Pontos cujo os valores na dimensão d são iguais ou menores a v estão na subárvore à esquerda, e respectivamente, pontos com valores na dimensão d maiores que v estão na subárvore à direita. Cada nível da árvore é associado a uma das k *dimensões*. Então, a citar um exemplo no plano, se dado nível considera o eixo x, a subárvore à esquerda contém os pontos com o eixo x menor que o valor v. Similarmente, à direita contém os pontos com o eixo x maior que o valor v. Nesse caso do plano, o eixo y é considerado no próximo nível da árvore. Uma árvore KD (para $K = 2$) pode ser construída em tempo $O(n \log n)$ onde n é o número de pontos dado na entrada. E o tempo de consulta é da ordem de $O(\sqrt{n} + k)$ onde k é o número de pontos dentro da janela (BERG et al., 2008a).

2.1.1 Árvore $2D$

Uma árvore $2D$ é a versão com duas dimensões para árvores KD. A sua construção considera um dado conjunto de pontos no plano (P) e pode ser feita da seguinte forma. Na construção de uma árvore para 2 dimensões, cada ponto tem uma forma $p = (p_x, p_y)$. Escolhemos um eixo para iniciar a construção da árvore. Ao iniciar a construção, realizamos uma x-ordenação e uma y-ordenação nos pontos de P (no caso n dimensional, faremos n ordenações, uma para cada dimensão). Chamaremos o conjunto de pontos ordenados pelo eixo x de $P_{ord(x)}$, e ordenados por y de $P_{ord(y)}$.

A seguir, apresentamos uma forma de construir uma árvore $2D$ como em (BERG et al., 2008a). Fixado um eixo, o valor da mediana dos pontos ordenados neste eixo será o valor v escolhido para dividir P em dois subconjuntos, e serão criados dois subconjuntos. Chamamos recursivamente a construção das subárvores à esquerda e à direita de v alternando o eixo e passando os subconjuntos. A base do algoritmo ocorre quando o conjunto de pontos possui um único ponto. Neste caso, armazenamos este ponto na folha

da árvore. Vamos ilustrar uma construção de uma árvore $2D$ considerando o conjunto de pontos da Figura 4.

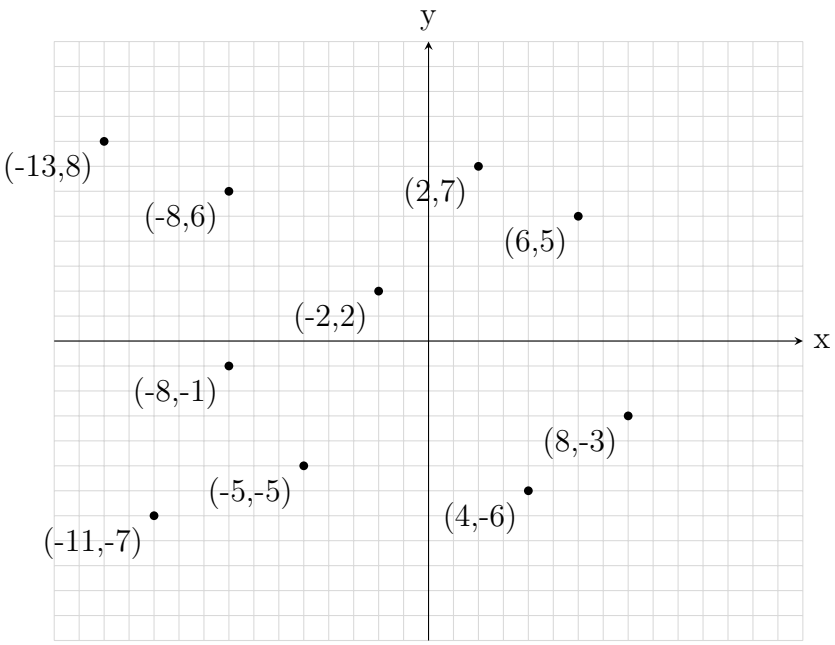


Figura 4 – Plano com os pontos P.

Inicialmente escolhemos o eixo x. Acompanharemos a troca de eixos de acordo com o nível da árvore. Caso o nível seja *par*, consideramos o eixo x, do contrário, o eixo y. Realizamos uma x-ordenação em P obtendo $P_{ord(x)} = ((-13, 8), (-11, -7), (-8, 6), (-8, -1), (-5, 5), (-2, 2), (2, 7), (4, -6), (6, 5), (8, -3))$. Com isso, sabemos que a x-mediana é $v = -5$ (veja a Figura 5). Criamos dois subconjuntos P_1 e P_2 tal que: $P_1 = \{p \in P : p_x \leq v\}$ e $P_2 = \{p \in P : p_x > v\}$; e, de forma recursiva, obtemos as árvores $2D$ dos dois novos subconjuntos P_1 e P_2 (as raízes dessas novas árvores consideram o eixo y). Por fim, criamos um nó r e armazenamos nele a x-mediana (ou seja, $val(v) = -5$). Chamaremos a x-mediana de x_{med} e, respectivamente, y_{med} a y-mediana; a subárvore à esquerda de r será uma árvore $2D$ sobre P_1; e a subárvore à direita de r será uma árvore $2D$ sobre P_2.

Vale reforçar que o eixo y será considerado para os subconjuntos P_1 e P_2. Para P_1, realizamos uma y-ordenação dos pontos e obtemos $P_{1ord(y)} = ((-11, -7), (-5, -5), (-8, -1), (-8, 6), (-13, 8))$. A y-mediana é $v = -1$. O processo continua com a criação dos subconjuntos considerando P_1 e depois considerando os pontos em P_2. A Figura 6 mostra os cortes no plano utilizados para construir os subconjuntos de pontos.

Este procedimento é repetido até que o conjunto de pontos tenha somente um ponto. Neste caso, criamos um nó folha, contendo o ponto p. A Figura 7 ilustra todos os cortes sobre o conjunto de pontos inicial. Estes cortes foram utilizados na construção dos subconjuntos de pontos.

A Figura 8 ilustra a árvore $2D$ completa para o conjunto de pontos considerados

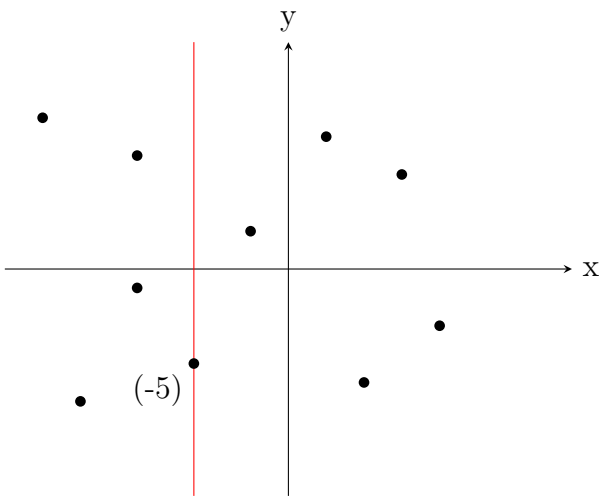


Figura 5 – A linha vermelha é o corte no eixo x.

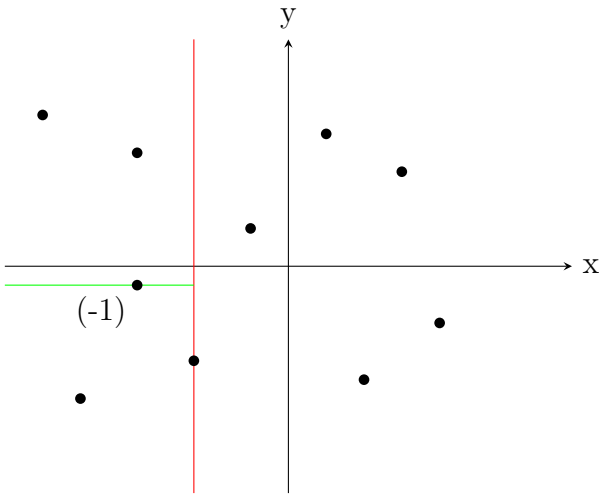


Figura 6 – A linha verde é o corte no eixo y que divide em relação ao ponto $(-8, -1)$.

anteriormente. Logo depois, descrevemos um algoritmo que constrói uma árvore $2D$ para um dado conjunto de pontos no plano.

2.1.2 Consulta de pontos em árvores $2D$

Uma consulta em uma árvore $2D$ construída sobre um conjunto de pontos P é uma busca de quais pontos de P estão entre um retângulo de consulta $[x, x'] \times [y, y']$ que chamaremos de janela. Um ponto $p = (p_x, p_y)$ está dentro de um retângulo de busca se $p_x \in [x, x']$ e $p_y \in [y, y']$. Podemos dizer que uma consulta no plano é composta de duas subconsultas nos eixos: uma no eixo x e outra no eixo y.

Seja P^r o conjunto de pontos em folhas alcançáveis a partir de r. Podemos definir uma *região* retangular a partir dos pontos em P^r (denotamos tal região por região(r)):

$$\text{região}(r) = [min_x(p), max_x(p)] \times [min_y(p), max_y(p)], p \in P^r$$

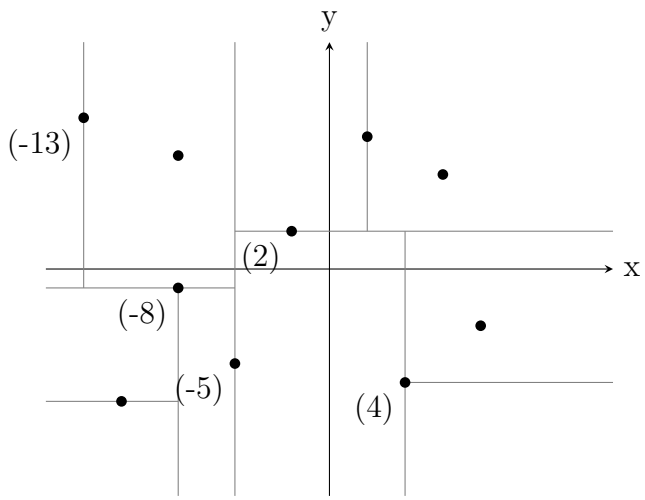


Figura 7 – Plano P com a subdivisão dos pontos e as linhas cinza são os cortes dos nós da árvore $2D$. Cada valor nesta imagem é um nó da árvore da Figura 8.

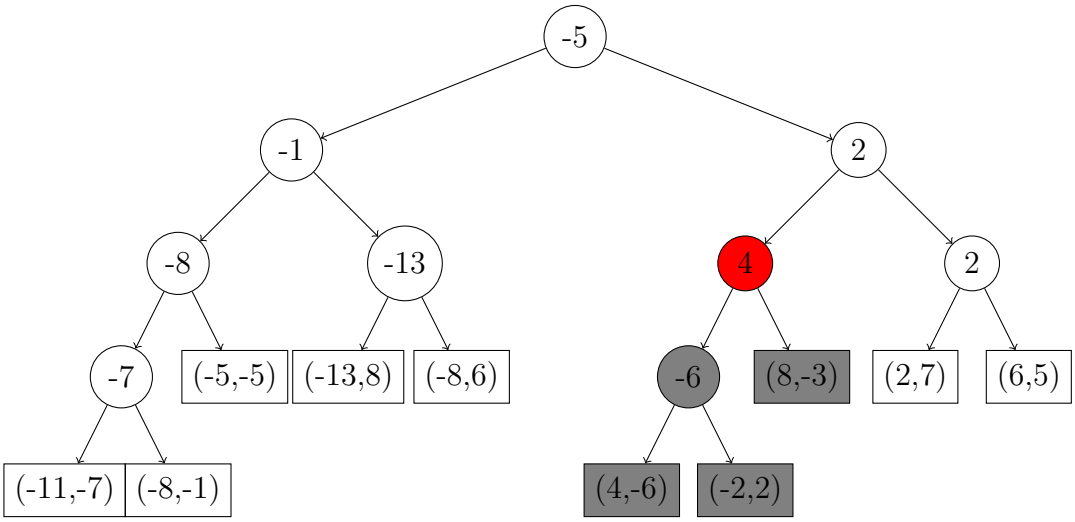


Figura 8 – Árvore $2D$ construída. O nó$_3^2$ indica a área da Figura 9

O algoritmo de consulta buscará pela subárvore com raiz r somente se o retângulo de busca intersectar a região(r). Um exemplo de uma região de um nó de uma árvore $2D$ pode ser observado na Figura 8. A região do nó$_3^2$ (que guarda o valor $val(\text{nó}_3^2) = 4$) é

$$\text{região}(\text{nó}_3^2) = [-2, 8] \times [-6, 2]$$

como pode ser observado na Figura 9 .

Dada uma janela de consulta, o algoritmo de busca funciona descendo a árvore, mas visitando somente os nós r cuja região(r) intersecta a janela. Quando uma região(r) está contida na janela de consulta, devolvemos todos os pontos na subárvore enraizada em r. No caso em que chegamos em um nó folha, temos que verificar se o ponto armazenado neste nó está dentro da janela, e se estiver, contabilizá-lo.

Segue o algoritmo que recebe como parâmetros a raiz de uma árvore $2D$ e uma

Algorithm 1 Recebe um conjunto de pontos P no plano e uma profundidade da árvore. Devolve a raiz de uma árvore $2D$.

```
1: function ConstróiÁrvore2D(P, profundidade)
2:     if P contém apenas um ponto then return nó folha que armazena o ponto de P
3:     else
4:         if profundidade é par then
5:             Faça uma x-ordenação sobre P
6:             Divide P em dois subconjuntos pela x_med
7:             Seja P_1 o conjunto dos pontos à esquerda de v incluindo o próprio v
8:             Seja P_2 o conjunto de pontos à direita de v
9:         else
10:            Faça uma y-ordenação sobre P
11:            Divide P em dois subconjuntos pela y_med
12:            Seja P_1 o conjunto dos pontos abaixo de v incluindo o próprio v
13:            Seja P_2 o conjunto de pontos acima de v
14:        end if
15:    end if
16:    v_esquerda ← ConstróiÁrvore2D(P_1, profundidade + 1)
17:    v_direita ← ConstróiÁrvore2D(P_2, profundidade + 1)
18:    Cria um nó r, armazene em r o valor v e associe os filhos r_esquerda e r_direita
19:    return r
20: end function
```

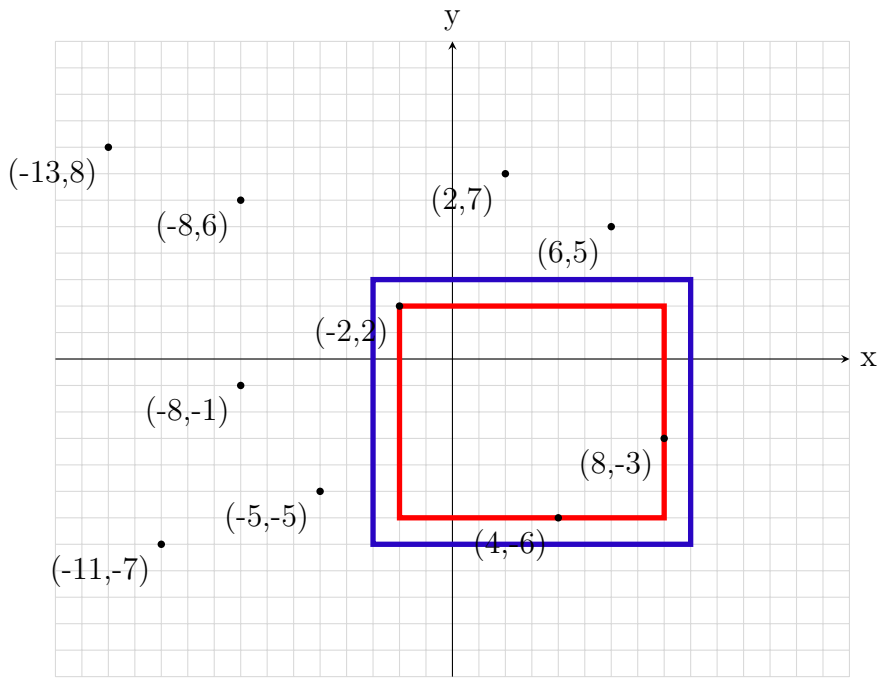


Figura 9 – Em vermelho a região do nó da árvore anterior com o valor 4 armazenado.

janela R. Usamos uma chamada ReportaSubárvore(r) que atravessa a árvore do nó r e devolve todos os pontos nas suas folhas. Segue como notação $filho_{esq}(r)$ sendo o filho à esquerda e $filho_{dir}(r)$ o filho à direita do nó r.

Algorithm 2 Recebe como parâmetro um nó r e uma janela R. Devolve todos os pontos dentro de R.

```
1: function ConsultaEmÁrvore2D(r, R)
2:     if r é folha then return r se estiver dentro da R
3:     else
4:         if região(filho_esq(r)) está contido na R then
5:             return ReportaSubárvore(filho_esq(r))
6:         else
7:             if região(filho_esq(r)) intersecta R then
8:                 return ConsultaEmÁrvore2D(filho_esq(r), R)
9:             end if
10:        end if
11:        if região(filho_dir(r)) está contido na R then
12:            return ReportaSubávore(filho_dir(r), R)
13:        else
14:            if região(filho_dir(r)) intersecta R then
15:                return ConsultaEmÁrvore2D(filho_dir(r), R)
16:            end if
17:        end if
18:    end if
19: end function
```

Vamos acompanhar uma consulta na árvore construída da Figura 8. Considere uma janela $[-3, 9] \times [-7, 3]$. Iniciamos no nó$_1^0$. Como o nó não é folha, checamos se a região do filho à esquerda esta dentro da consulta. O nó$_1^1$ tem região $-13 \leq x \leq -5$ e $-7 \leq y \leq 8$, logo a região não está contida na consulta. Assim como a região do nó$_2^1$. Porém, as regiões dos nó$_1^1$ e nó$_2^1$ intersectam o retângulo da consulta. Então procedemos com a consulta em ambos os nós. Note que o filho à esquerda do nó$_2^1$ (nó$_3^2$ em destaque na Figura 8) possui região contida na janela de consulta. Portanto todos os pontos das folha alcançáveis a partir deste nó estão na janela.

2.1.3 Otimização para a construção de uma árvore $2D$

A forma como implementamos o algoritmo de construção realiza ordenações em cada execução do algoritmo que não é o caso da base da recursão (linhas 5 e 10 do Algoritmo 1). Uma otimização para a implementação da construção da árvore $2D$ considera apenas 2 ordenações, uma para cada eixo. Para isto, é preciso ordenar os pontos nos dois eixos antes do início da construção da árvore e a cada chamada do algoritmo que constrói a árvore, conseguimos construir os subconjuntos de pontos ordenados em tempo linear, sempre considerando uma ordenação do conjunto de pontos que originou essa chamada. Por último, a Figura 10 corresponde a uma figura gerada por nossa implementação da árvore $2D$. Como esperado, os pontos dentro de uma janela foram encontrados.

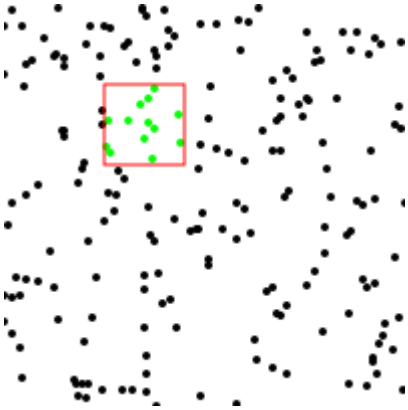


Figura 10 – Resultados de busca em uma árvore $2D$

2.2 Árvore de Alcance

Uma árvore de alcance (BENTLEY, 1979) pode ser implementada considerando várias dimensões. No caso geral, com k dimensões, teremos k níveis de árvores binárias de busca todas balanceadas. Uma dimensão está relacionada a cada nível. O primeiro nível é formado por uma árvore binária de busca balanceada relacionada a dimensão, digamos, 1. Cada nó desta árvore guarda um valor da dimensão 1, mais uma outra árvore binária de busca balanceada relacionadas à dimensão 2. Cada nó do segundo nível, guarda uma valor da dimensão 2, outra árvore relacionada à dimensão 3. E assim por diante. Neste trabalho consideramos uma árvore de alcance para 2 dimensões ($2D$). Denotamos por τ_r a raiz de uma árvore relacionada a um nó r. Neste caso, o nó r na árvore mais externa e τ_r em uma árvore auxiliar ao nó r. A Figura 11 ilustra uma árvore de alcance $2D$. Note que as folhas da árvore do primeiro nível (à esquerda) aparecem como folhas em uma árvore do segundo nível (à direita). É uma alternativa para consulta de pontos cujo tempo de consulta é superior ao da árvore KD.

2.2.1 Árvore de Alcance $2D$

Uma árvore de alcance $2D$ é uma versão de duas dimensões para árvores de alcance. A construção considera um dado conjunto de pontos P e pode ser feita da seguinte forma. Temos os pontos na forma $p = (p_x, p_y)$. Ao iniciar a construção, fixamos o eixo x para o primeiro nível; construímos uma árvore binária de busca balanceada T considerando os valores na x-coordenada dos pontos de P; os pontos p serão salvos nos nós folha; para cada nó não folha r de T construiremos uma árvore auxiliar τ_r com todas as folhas alcançáveis a partir de r. Estas árvores serão construídas considerando os valores da y-coordenada. Por fim, associamos τ_r ao nó r. Os nós folhas das árvores guardarão os pontos de P. Uma árvore de alcance pode ser construída em tempo $O(n \log(n))$, onde n é o número de pontos dados no plano. O tempo de uma consulta em uma árvore de alcance é da ordem de

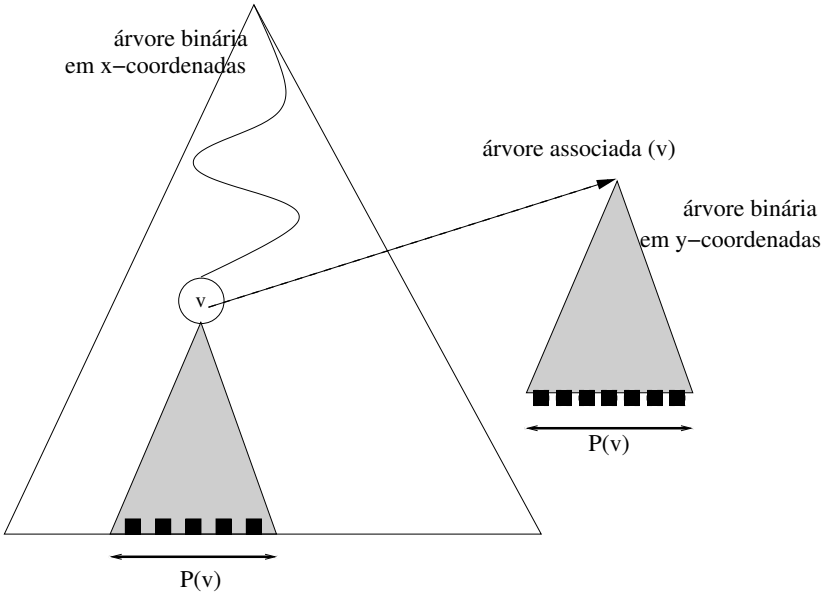


Figura 11 – Árvore de Alcance $2D$. Onde $P(v)$ são os pontos alcançáveis a partir de um nó v.

$O(\log^2(n) + k)$ onde k são os pontos dentro da janela (BERG et al., 2008a).

Vamos considerar o seguinte conjunto de pontos P da Figura 12 para ilustrar a construção de uma árvore de alcance $2D$.

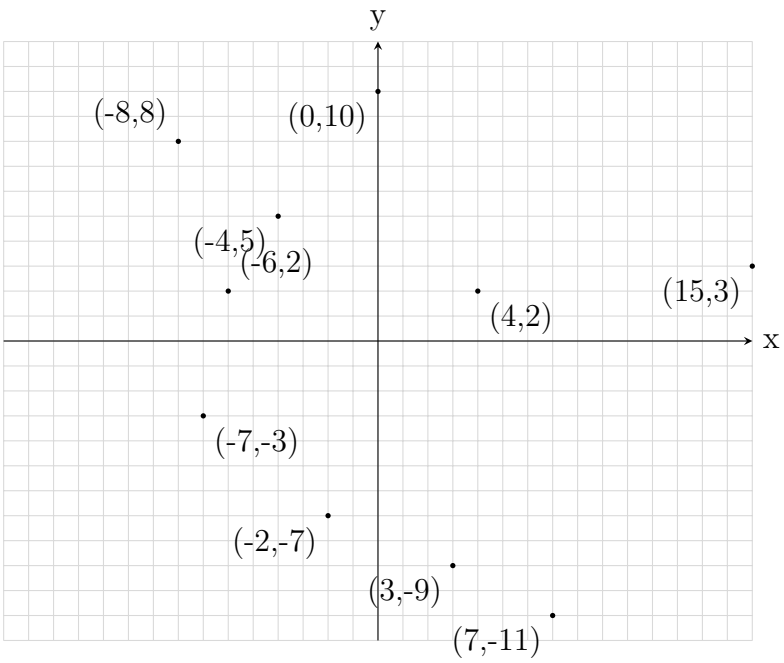


Figura 12 – Pontos P no plano.

Para construir uma árvore binária de busca balanceada do primeiro nível, podemos inicialmente ordenar os pontos de P pela x-coordenada. Pegamos a mediana da x-

coordenada, x_{med}, e o guardamos em um nó r da árvore. Depois, criamos dois subconjuntos P_1 e P_2 tal que $P_1 = \{p \in P : p_x \leq x_{med}\}$ e $P_2 = \{p \in P : p_x > x_{med}\}$. Criamos uma árvore binária de busca balanceada τ_r considerando o valor da y-coordenada dos pontos de P. A árvore de segundo nível associada ao vértice r será τ_r. O processo continua de forma recursiva sobre os dois novos conjuntos P_1 e P_2. Em seguida, adicionamos o pseudocódigo deste procedimento.

Algorithm 3 Recebe como entrada um conjunto de pontos P. Devolve o nó raiz de uma árvore de alcance $2D$.

1: **function** ConstróiÁrvoreAlcance2D(P)
2: Criamos um novo nó r
3: Construímos uma árvore binária de busca balanceada associada ao nó r sobre os pontos P e considerando a y-coordenada. A árvore será denotada por τ_r.
4: **if** P contém apenas um ponto **then**
5: Guardamos o ponto de P em r.
6: Associamos τ_r ao nó r.
7: **else**
8: Dividimos P em dois subconjuntos.
9: P_1 contém os pontos com a x-coordenada menor ou igual que x_{med}.
10: P_2 contém os pontos com x-coordenada maior que x_{med}.
11: $v_{esq} \leftarrow$ ConstróiÁrvoreAlcance2D(P_1).
12: $v_{dir} \leftarrow$ ConstróiÁrvoreAlcance2D(P_2).
13: Criamos um nó v guardando x_{med}.
14: Fazemos v_{esq} e v_{dir} filhos à esquerda e à direita de v.
15: Associamos τ_r ao nó r.
16: **end if**
17: **return** v
18: **end function**

Vamos seguir alguns passos da construção de uma árvore de alcance. Considere os os pontos da Figura 12 ordenados pelo eixo x: $P_{ord(x)} = \{(-8, 8), (-7, -3), (-6, 2), (-4, 5), (-2, -7), (0, 10), (3, -9), (4, 2), (7, -11), (15, 3)\}$. Criamos um nó raiz r e nele armazenamos o valor da $x_{med} = -2$. Dividimos o conjunto de pontos em dois subconjuntos P_1 com os pontos $p_x \leq -2$ e P_2 com os valores $p_x > -2$. Com relação à árvore de segundo nível relacionada ao conjunto inicial, fazemos uma y-ordenação nos pontos de P e construímos uma árvore binária de busca balanceada considerando os valores do eixo y e associamos a árvore resultante à raiz r (ou nó$_1^0$) da árvore do primeiro nível. A subárvore à esquerda de r será uma árvore de alcance sobre P_1; e a subárvore à direita de r será uma árvore de alcance sobre P_2. A Figura 13 ilustra o nó$_1^1$ e a sua árvore associada de segundo nível.

Este procedimento é repetido até que caia na base da recursão que ocorre quando o conjunto de pontos tem somente um ponto p. Neste caso, criamos um nó folha contendo o ponto p e a sua árvore associada de segundo nível também contendo o ponto p. A Figura 14 a seguir é a representação dos pontos no plano em uma árvore de alcance (primeiro nível) sobre os pontos da Figura 12.

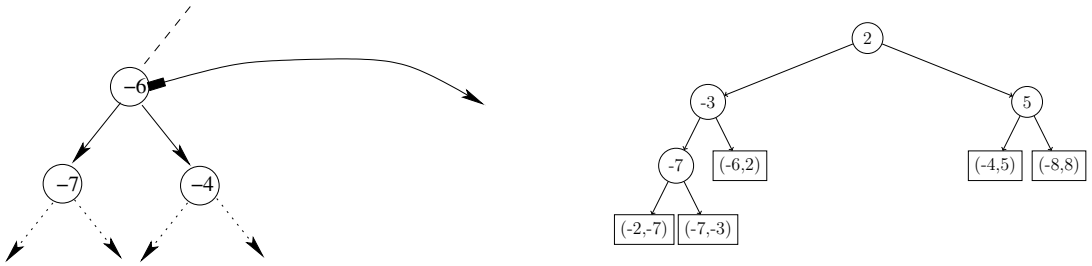


Figura 13 – nó$_1^1$ e sua árvore de segundo nível associada da Figura 14.

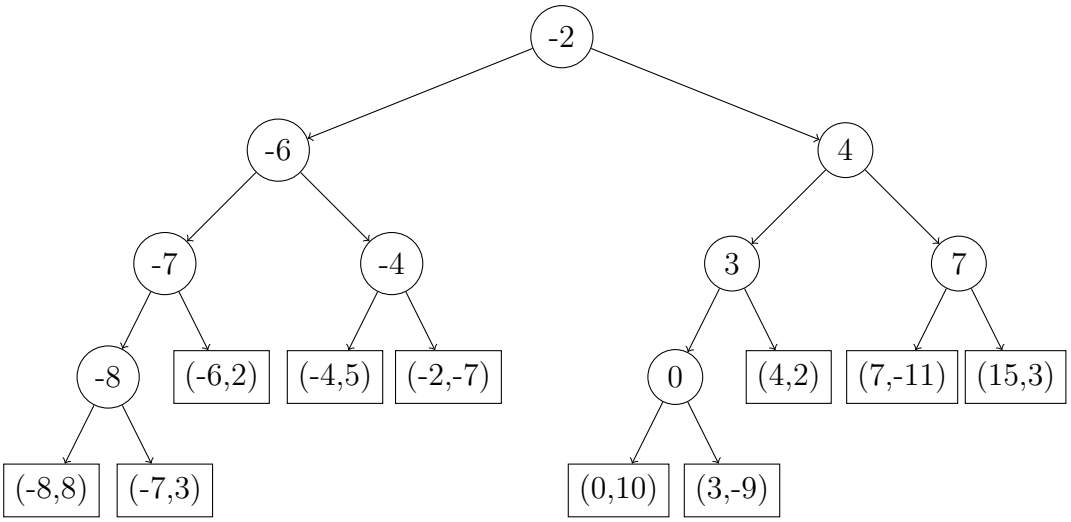


Figura 14 – Árvore construída com os pontos P pela x-coordenada.

2.2.2 Consulta dos pontos em árvores de alcance.

Uma consulta 2-dimensional em P é uma busca de quais pontos de P estão entre uma janela de consulta $[x, x'] \times [y, y']$. Um ponto $p = (p_x, p_y)$ está dentro de um retângulo de consulta se $p_x \in [x, x']$ e $p_y \in [y, y']$.

Uma consulta em uma árvore de alcance é a combinação de n consultas em árvore, onde n é a dimensão da árvore de alcance. Na árvore de alcance 2-dimensional temos uma consulta no eixo x seguida por uma consulta na árvore auxiliar τ que foi construída considerando y-coordenada. A consulta com janela, consistirá portanto na união de duas consultas de intervalo unidimensional em árvore.

Uma consulta unidimensional em uma árvore de alcance pode ser visualizada como uma consulta de quais pontos em uma reta s estão em um intervalo de consulta $[x, x']$.

2.2.3 Consulta de intervalo unidimensional

Seja uma árvore binária balanceada T. Uma consulta de intervalo em T é tal que queremos todos os nós folhas de T cujo valor esteja dentro do intervalo consultado $[x, x']$.

Dado o intervalo da consulta $R = [x, x']$ e a raiz de T realizaremos o seguinte

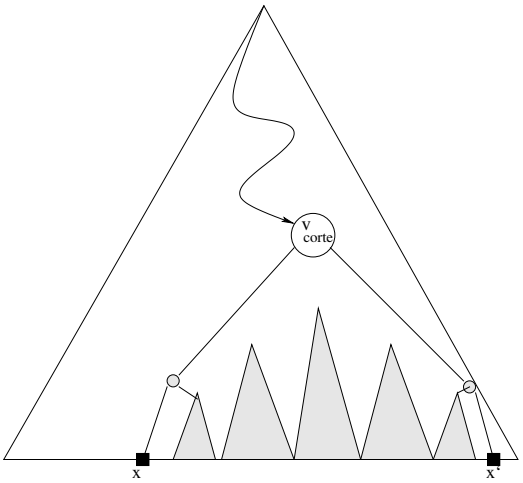


Figura 15 – Consulta de alcance unidimensional, e o nó$_{corte}$ sendo o primeiro nó da consulta.

algoritmo: inicialmente precisamos encontrar o primeiro nó cujo valor está contido na consulta. Chamaremos este nó de nó$_{corte}$. Para encontrar o nó$_{corte}$ faremos uma consulta simples que a partir da raiz v checaremos se o valor armazenado em v está dentro do intervalo buscado. Se estiver será nosso nó$_{corte}$. Do contrário checaremos se o valor armazenado em v é maior que x' (denotaremos por $v > x'$). Caso seja, o valor em v é maior que o maior valor da consulta, então faremos recursivamente a busca por nó$_{corte}$ passando o nó$_{esquerda}$ de v. Caso $v \leq x$ realizaremos a consulta por nó$_{corte}$ recursivamente em nó$_{direita}$ de v.

Munidos de nó$_{corte}$, faremos a consulta para retornar todos os pontos contidos dentro de R na árvore T. A partir de nó$_{corte}$, queremos recursivamente buscar os pontos que são menores que nó$_{corte}$ porém ainda dentro da consulta, e similarmente, os pontos maiores que n_{corte} ainda dentro do intervalo. Assim como buscar os pontos maiores que nó$_{corte}$ e dentro da consulta. (BERG et al., 2008b)

Para buscarmos os valores menores que nó$_{corte}$ iremos realizar uma consulta em profundidade considerando os nós não folha à esquerda a partir de nó$_{corte}$ checando se valor do nó v é maior que x. Quando essa condição não for mais atendida, isso significa que já não mais está dentro da consulta. Neste ponto, checaremos se o nó à direita está dentro da consulta. Caso esteja, reportamos. Durante a busca em profundidade todos os pontos que atendem $v > x$, reportamos a subárvore à direita deste nó, ou seja, temos certeza que os valores desta subárvore está dentro do intervalo consultado.

A mesma busca em profundidade será realizada para os valores maiores que nó$_{corte}$, checando se o valor armazenado no nó é menor ou igual que x' e reportando todas as subárvores à esquerda que atendem essa condição. E no caso onde essa condição seja falsa, checando o nó à esquerda e reportando caso o valor esteja dentro do intervalo. A condição de parada dessa busca em profundidade é caso seja um nó folha, onde deve-se checar se o

ponto armazenado na folha está dentro do intervalo. Caso esteja, devemos reportá-lo.

Em seguida apresentamos os pseudocódigos para buscar o nó$_{corte}$ e para consultar de forma unidimensional.

Algorithm 4 Recebe como parâmetro um nó e uma janela. Devolve o primeiro nó cujo valor armazenado esteja dentro do intervalo de consulta.

```
1: function EncontraNóCorte(v, R : [x, x'])
2:     while v não é folha do
3:         if v ∈ R then return v
4:         else
5:             if v > x' then
6:                 v ← v_esquerda
7:             else
8:                 v ← v_direita
9:             end if
10:        end if
11:    end while
12: end function
```

Algorithm 5 Recebe um nó e uma consulta. Devolve todos os pontos dentro da consulta.

```
1: function BuscaEmAlcance1D(v_corte, R : [x, x'])
2:     if v_corte é folha then
3:         if v_corte ∈ R then
4:             Devolve ponto v_corte
5:         end if
6:     else
7:         v ← filho_esq(v_corte)
8:         while v não for folha do
9:             if x ≤ val(v) then
10:                ReportaSubárvore(v)
11:                v ← filho_esq(v)
12:            else
13:                v ← filho_dir(v)
14:            end if
15:        end while
16:        if v é folha e val(v) ∈ R then
17:            Reporta ponto v
18:        end if
19:        Repete-se as linhas 7 até 18 de forma simétrica, trocando o símbolo > por ≤,
       esquerda por direita e direita por esquerda.
20:    end if
21: end function
```

Na Figura 16 temos a árvore construída em relação a coordenada y.

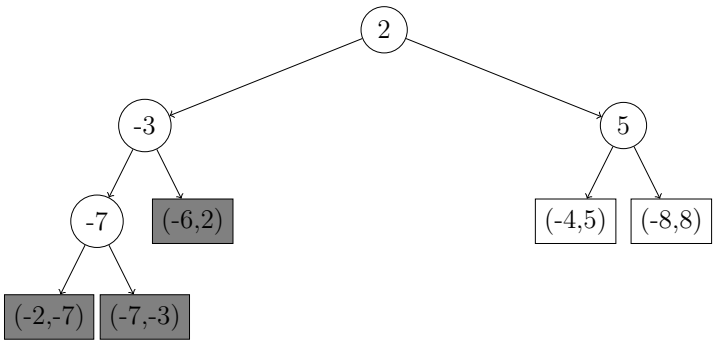
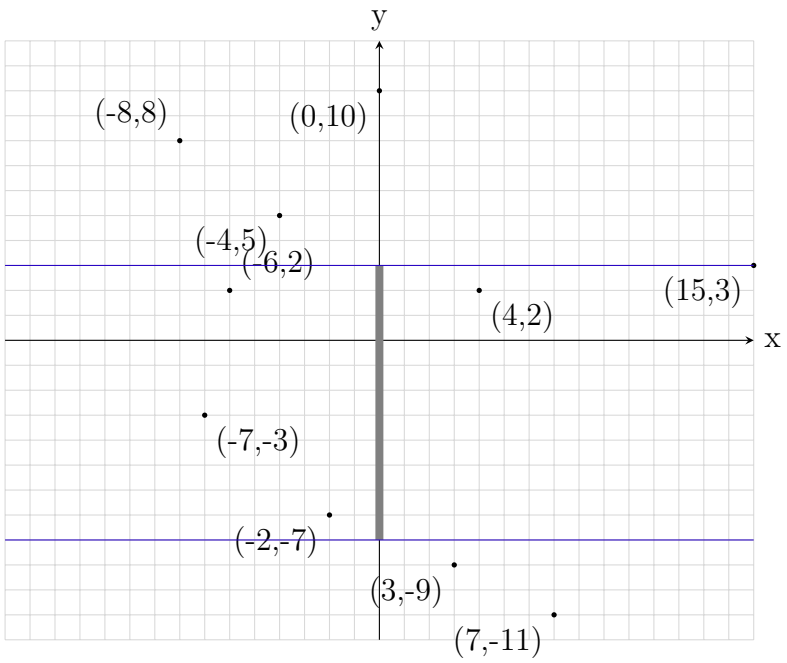


Figura 16 – Pontos na y-árvore do nó$_1^1$ da Figura 14.

Vamos ilustrar uma consulta unidimensional considerando está árvore. Seja o intervalo de consulta $R = [-8, 3]$ (veja na Figura 17):

Figura 17 – Em azul, a consulta unidimensional em y nos pontos em P.



Começamos pelo nó raiz com valor 2. A condição $-8 \leq 2 \leq 3$ é verdadeira. Portanto nó$_1^0$ será nosso nó$_{corte}$. A partir dele, primeiramente iremos obter os nós v tal que $v \leq 2$. Começamos a busca em profundidade: o nó a esquerda de nó$_1^0$ é nó$_1^1$. Reportamos a subárvore à direita nó$_1^1 = \{(-6, 2)\}$.O valor val(nó$_1^1$) $= -3$ ainda mantém $-8 \leq -3$, e, portanto, continuamos a busca em profundidade. Agora temos o nó$_1^3$. Por sua vez, satisfaz $-8 \leq -7$, reportamos a subárvore à direita $=\{(-7, -3)\}$. A busca em profundidade continuará até o nó$_{folha} = \{(-2, -7)\}$, que satisfaz a consulta.Realizaremos o mesmo procedimento, porem, á partir de nó$_{corte}$ iremos realizar busca em profundidade pela direita. Consultamos que val(nó$_2^1$) > 3, portanto este nó não está dentro do intervalo de consulta. Vamos consultar se a subárvore à esquerda de nó$_2^1$ está dentro da consulta. Este é um nó

folha $val(\text{nó}_3^2) = (-4, 5)$ e verificamos que este também não está dentro da consulta. Por fim teremos os pontos: $\{(-2, -7), (-7, -3), (-6, 2)\}$.

2.2.4 Consulta Bidimensional

Para realizar uma consulta bidimensional, iremos realizar uma consulta de intervalos unidimensional para cada dimensão. Para cada nó visitado que está dentro da janela de consulta, será feita outra busca unidimensional na árvore τ associada ao nó. Segue pseudocódigo pra consulta bidimensional.

Algorithm 6 Recebe um nó e uma janela. Devolve todos os pontos dentro da consulta.

1: **function** BuscaEmAlcance2D(v_{corte}, $R : [x, x'] \times [y, y']$)
2: $v_{corte} \leftarrow$ EncontraNóCorte(τ, x, x')
3: **if** v_{corte} é folha **then return** $val(n_{corte})$ se $val(v_{corte}) \in R$
4: **else**
5: $v \leftarrow filho_{esq}(v_{corte})$
6: **while** v não é folha **do**
7: **if** $x \leq v$ **then**
8: BuscaEmAlcance1D($filho_{dir}(v) \rightarrow \tau_{associada}, [y, y']$)
9: $v \leftarrow filho_{esq}(v)$
10: **else**
11: $v \leftarrow filho_{dir}(v)$
12: **end if**
13: **end while**
14: Checar se o ponto na folha está dentro da consulta.
15: De forma simétrica às linhas 6 até 15, seguimos pelo caminho a partir de $filho_{dir}(v_{corte})$.
16: **end if**
17: **end function**

Iremos agora exemplificar uma consulta bidimensional. Seja o intervalo de consulta $R = [-6, 1] \times [-8, 3]$. Como explicitado, será uma junção de consultas unidimensionais na x-árvore com a consulta $[-6, 1]$ e uma consulta nas y-árvores associadas entre $[-8, 3]$.

Iniciamos a busca no nó raiz n_1^0. Temos que inicialmente encontrar o n_{corte} e n_0^1 está dentro de $[-6, 1]$. A partir dele faremos uma busca em profundidade para esquerda e para a direita para encontrar todos os valores dentro do intervalo $[-6, 1]$.

Como na Figura 14, agora estamos no nó_1^0, este não é folha, então prosseguimos. Consultaremos em profundidade iniciando pela esquerda. À esquerda de nó_1^0 é nó_1^1, não folha, e portanto checaremos se este ainda é menor que a consulta, ou seja, se a busca em profundidade pode prosseguir consultando os ramos à esquerda da árvore. $-6 \leq -6$, logo, faremos uma Busca1DEmAlcance($\text{nó}_2^2 \rightarrow \tau_{assoc}, -8, 3$). A árvore auxiliar do nó_2^2 é τ_{assoc} é a árvore que segue:

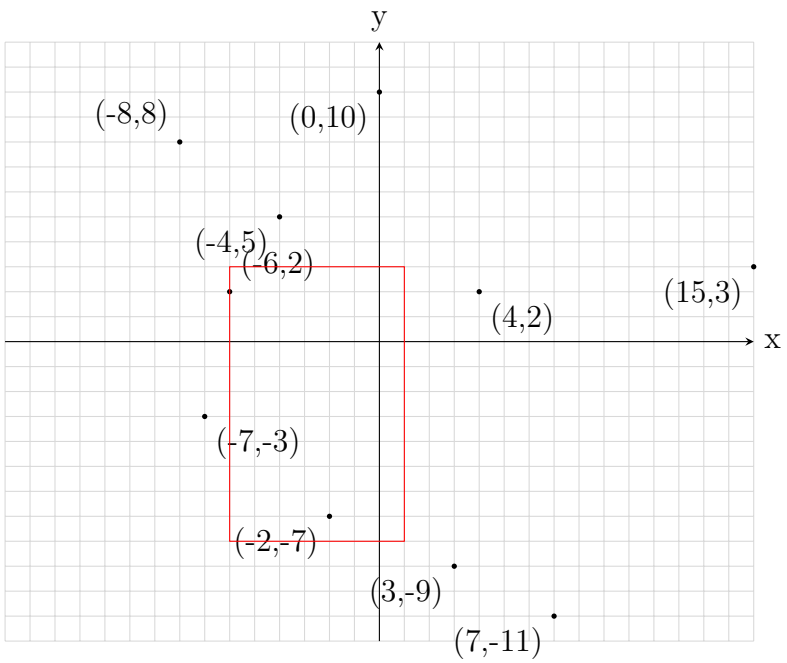


Figura 18 – Em vermelho a região do retângulo de consulta

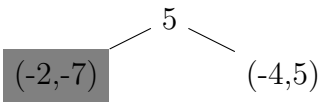


Figura 19 – Pontos na y-árvore do nó$_1^1$ da Figura 14.

Consultaremos nesta árvore os pontos dentro do intervalo $[-8, 3]$, que retornará os pontos em cinza $= \{(-2, -7)\}$. Consultaremos o próximo nó à esquerda, nó$_1^3$. Por sua vez a consulta $-6 \leq -7$ é falso e iremos avaliar o nó à direita de nó$_1^2$ que é o ponto $(-6, 2)$ e checamos se está na janela e o retornamos. Prosseguiremos com a consulta à direita do nó$_{corte}$ = nó$_1^0$. A figura a seguir ilustra os resultados da nossa implementação.

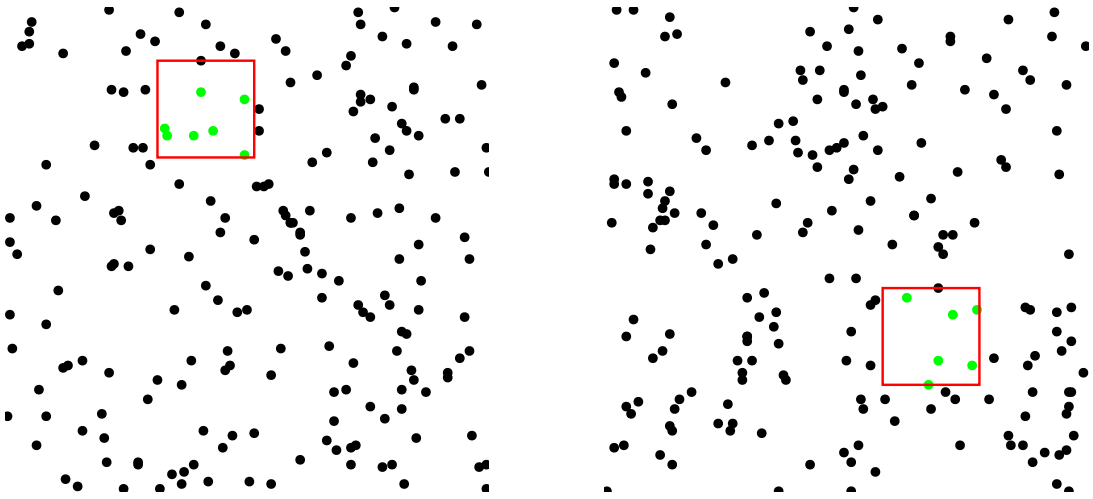


Figura 20 – Resultados das consultas em janela usando Árvore de Alcance $2D$

3 Algoritmos e estruturas de dados para consulta de segmentos no plano

Nesta seção, veremos formas de construir estruturas de dados que ajudam a responder consultas da seguinte forma: Dado um conjunto de segmentos no plano, quais destes segmentos estão em uma janela? Consideramos que o conjunto de segmentos é fixo. Portanto, uma vez construída, a estrutura de dados estará fixa e preparada para responder muitas consultas diferentes. Veremos duas estruturas de dados neste capítulo: Árvore de Intervalos e Árvore de Segmentos. Para cada estrutura veremos como podemos construí-las e usá-las para responder consultas em janelas.

3.1 Árvore de Intervalos

Seja I um conjunto de intervalos. A mediana deste conjunto de intervalos I é a media das medianas de cada intervalo em I. Denotaremos que um intervalo $[x, x']$ é menor que θ se $x' < \theta$. Reciprocamente, denotaremos que o intervalo $[x, x']$ é maior que θ se $x > \theta$. Uma árvore de intervalos é uma árvore binária balanceada multinível onde cada nó não folha guarda um valor v que representa a mediana de um conjunto de intervalos I. Denotaremos o conjunto de intervalos que contém o valor v de I_{med}. Denotaremos também o conjunto de intervalos que são menores que v e não contém v de I_{esq}, e similarmente, o conjunto de intervalos maiores que v e que não o contém de I_{dir}. Cada nó não folha r possui uma estrutura auxiliar para consultar os intervalos I_{med}. Uma árvore de intervalos pode ser construída em tempo $O(n \log(n))$. Uma árvore de intervalos pode reportar todos os k segmentos que estão em uma janela em tempo $O(\log^2(n) + k)$ (BERG et al., 2008a). A fim de encontrar quais segmentos estão sendo visualizados em uma janela temos três casos. Os segmentos possuem dois pontos dentro da janela; Os segmentos possuem ao menos um dos pontos extremos dentro da janela; Os segmentos possuem nenhum ponto extremo dentro da janela, porem, cruzam a janela. Utilizaremos a árvore de intervalo para encontrar o ultimo caso. Para encontrarmos os dois primeiros utilizaremos as estruturas estudadas na seção 2.2.

3.1.1 Árvore de Intervalos unidimensional

A construção considera um dado conjunto de intervalos I e pode ser feita da seguinte forma. Temos os intervalos da forma $i = [x, x']$. Ao iniciar a construção, calculamos o valor da mediana, x_{med}, dos intervalos em I e o guardamos em um nó r. Criamos três subconjuntos I_{esq} e I_{dir} tal que $I_{esq} = \{[x, x'] \in I : x' < x_{med}\}$, $I_{dir} = \{[x, x'] \in I : x >$

$x_{med}\}$ e $I_{med} = \{[x, x'] \in I : x \leq x_{med} \leq x'\}$. Construímos duas estruturas de dados auxiliares $\tau_{esq}(r)$ com todos os intervalos de I_{med} ordenada considerando o ponto mais à esquerda de cada intervalo e $\tau_{dir}(r)$ considerando o ponto mais à direita de cada intervalo. Por fim associamos ambas $\tau_{esq}(r)$ e $\tau_{dir}(r)$ ao nó r. O processo continua de forma recursiva sobre os dois novos conjuntos I_{esq} e I_{dir}. Vamos considerar o seguinte conjunto de intervalos da Figura 21 para ilustrar a construção de uma árvore de intervalos.

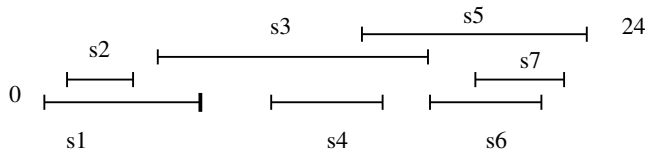


Figura 21 – Conjuntos de Intervalos na reta real.

Algorithm 7 Recebe um conjunto de intervalos I na reta real. Devolve a raiz de uma árvore de intervalos.

```
1: function ConstróiÁrvoreDeIntervalos(I)
2:     if I for vazio then return nó folha vazio
3:     else
4:         Crie um nó r
5:         Faça uma ordenação sobre I pelo ponto mais à esquerda
6:         Calcule x_med e guarde em r
7:         Calcule I_med construa duas listas ordenadas para I_med
8:         τ_esq(r) ordenado pelo ponto mais à esquerda dos intervalos crescente
9:         τ_dir(r) ordenado pelo ponto mais à direita dos intervalos decrescente
10:        Guarde τ_esq(r) e τ_dir(r) no nó r
11:        Cria os nós r_esquerda e r_direita
12:        r_esquerda ← ConstróiÁrvoreDeIntervalos(I_esq)
13:        r_direita ← ConstróiÁrvoreDeIntervalos(I_dir)
14:        Associe r_esquerda como filho à esquerda de r
15:        Associe r_direita como filho à direita de r
16:        return r
17:    end if
18: end function
```

Vamos seguir alguns passos da construção de uma árvore de intervalos como feito em (BERG et al., 1985). Considerando os intervalos da Figura 22 inicialmente vamos ordenar os intervalos de I pelo ponto mais à esquerda: $s_1 = [0, 7]$, $s_2 = [1, 4]$, $s_3 = [5, 12]$, $s_4 = [11, 16]$, $s_5 = [14, 24]$, $s_6 = [17, 22]$, $s_7 = [19, 23]$. Criamos um nó raiz r e nele armazenamos o valor da $x_{med} = 14$. Dividimos o conjunto de intervalos em três subconjuntos: I_{esq} com os segmentos $\{s_1, s_2\}$, I_{dir} com os segmentos $\{s_6, s_7\}$ por fim I_{med} com os segmentos que contêm x_{med} $\{s_3, s_4, s_5\}$. Com relação a I_{med} construímos duas listas: $\tau_{esq}(r)$ com os intervalos ordenados pelos pontos mais à esquerda $\{s_3, s_4, s_5\}$, e $\tau_{dir}(r)$ com os intervalos ordenados pelos pontos mais à direita de forma decrescente $\{s_5, s_3, s_4\}$. Ordenamos à

esquerda de forma crescente nos garantirá que em uma consulta se o valor consultado for menor que o primeiro intervalos consultado, não haverá necessidade de consultar os demais intervalos da lista associada. Respectivamente, os segmentos ordenados de forma decrescente pelos pontos extremos à direita nos garante que caso a consulta seja maior que o primeiro valor da lista não haverá necessidade de continuar a consulta na lista associada à direita. A subárvore à esquerda será uma árvore de intervalos sobre I_{esq}; e a subárvore à direita será uma árvore de intervalos sobre I_{dir}. Este procedimento é repetido até que o conjunto de intervalos seja vazio. Neste caso, criamos um nó folha também vazio. A Figura 22 ilustra a árvore de intervalos construída.

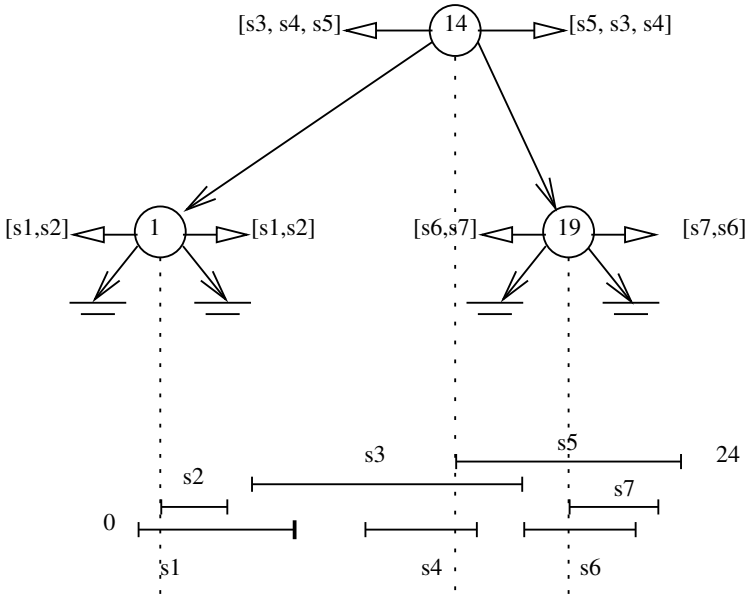


Figura 22 – Árvore de Intervalos.

3.1.2 Consulta de ponto em árvores de intervalo

Seja uma árvore binária de intervalos T. Uma consulta nesta árvore é tal que queremos todos os intervalos que contenham o valor q_x consultado. Dado o valor q_x da consulta realizamos o seguinte algoritmo: inicialmente checamos se q_x é menor que o valor armazenado em r (denotaremos por $q_x < r$); Caso seja, consultaremos a lista auxiliar $\tau_{esq}(r)$ e retornamos todos os intervalos que contenham q_x; então fazemos recursivamente a consulta em $r_{esquerda}$. De forma simétrica, caso $q_x \geq r$ consultaremos a lista auxiliar $\tau_{dir}(r)$ retornando os intervalos que contenham q_x e recursivamente consultando em $r_{direita}$.

Segue o algoritmo (Algoritmo 8) que recebe como parâmetros a raiz de uma árvore de intervalos r e ponto para consulta q_x.

Vamos acompanhar uma consulta na árvore construída da Figura 22. Considere um ponto de consulta $q_x = 1$. Iniciamos no nó$_1^0$. Como o nó não é folha, checamos se

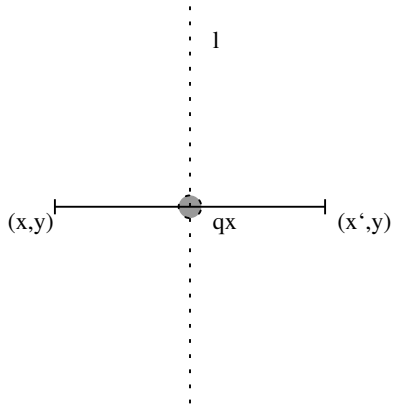


Figura 23 – Consulta de um valor q_x em um intervalo $[x, x']$

Algorithm 8 Recebe a raiz de uma árvore de intervalos r e um ponto de consulta q_x. Devolve todos os segmentos que contêm q_x.

```
1: function ConsultaÁrvoreDeIntervalos(r, q_x)
2:     if r não for folha then
3:         if q_x for < r then
4:             Consulte τ_esq(r) começando pelo intervalo com o ponto mais à esquerda e
   reporte todos intervalos que contenham q_x. Pare a consulta no primeiro intervalo que
   não contenha q_x.
5:             ConsultaÁrvoreDeIntervalos(r_esq, q_x)
6:         else
7:             Consulte τ_dir(r) começando pelo intervalo com o ponto mais à direita e
   reporte todos intervalos que contenham q_x. Pare a consulta no primeiro intervalo que
   não contenha q_x.
8:             ConsultaÁrvoreDeIntervalos(r_dir, q_x)
9:         end if
10:    end if
11: end function
```

valor q_x é menor que o x_{med} salvo. O x_{med} salvo é 14, logo o ponto de consulta está à esquerda do nó$_1^0$. Consultamos τ_{esq}(nó$_1^0$) e o intervalo s_3 não contém $q_x = 1$, portanto paramos a consulta em τ_{esq}(nó$_1^0$). Então procedemos com a consulta no nó à esquerda. Note que ambos $q_x = 1 \geq 1$ então consultaremos a lista $\tau_{dir}(r)$; reportando ambos $\{s_1, s_2\}$ que contêm 1.

3.1.3 Árvore de Intervalos para consulta de segmentos no plano

Seja S o conjunto de n segmentos no plano P paralelos aos eixos de consulta. Para consultar quais segmentos de forma $s = (s_x, s_y)(s'_x, s_y)$ estão dentro de uma janela de consulta $J = [x, x'] \times [y, y']$ podemos utilizar a estrutura de dados construída no capítulo 2, as árvores de alcance, para encontrar quais dos $2n$ pontos extremos dos segmentos estão dentro da janela J. Contudo essa consulta não conseguiria encontrar segmentos que

cruzam a janela, isto é, ambos pontos extremos estão fora da janela, porém passam por ela.

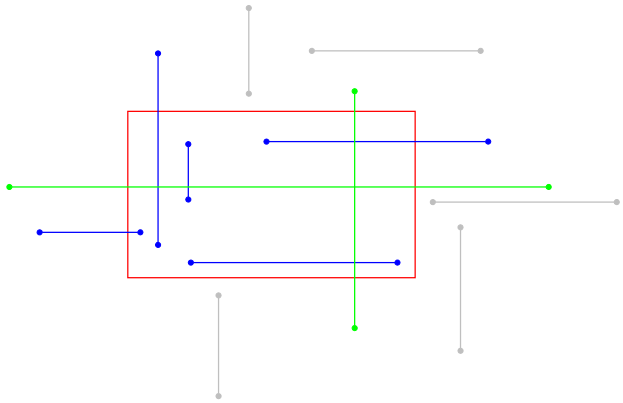


Figura 24 – Em azul: os segmentos encontrados pela árvore de alcance e que possuem pelo menos um ponto extremo dentro da janela. Em verde: os segmentos que cruzam as extremidades da janela e reportados pela árvore de intervalos. Em cinza, os segmentos que não devem ser reportados. Em vermelho : A janela de consulta

Para os segmentos que cruzam a janela, vistos em verde na Figura 24, utilizaremos uma árvore de intervalos. Contudo, para uma consulta no plano, ela será modificada para que leve em consideração o comportamento bidimensional da consulta. Para isso adaptaremos as estruturas auxiliares $\tau_{esq}(r)$ e $\tau_{dir}(r)$. Iremos montar o caso dos segmentos horizontais. Os verticais são simétricos; porém, na orientação vertical. Um segmento $s = (s_x, s_y)(s'_x, s_y)$ que cruza as duas extremidades da janela $[x, x'] \times [y, y']$ tem seu ponto à esquerda de x, isto é, $s_x < x$ e similarmente seu ponto mais à direita é maior que x', isto é, $s'_x > x'$. Portanto, para saber se um segmento cruza a janela podemos escolher uma das extremidades da janela e fazer uma consulta para saber se o intervalo $[s_x, s'_x]$ contém uma das extremidades da janela. Neste trabalho implementamos como (BERG et al., 1985) e consideramos a extremidade mais à esquerda da janela para consultas de segmentos horizontais. E a extremidade inferior da janela para consultas de segmentos verticais.

A construção de uma árvore de intervalos para consulta de segmentos no plano é a mesma para intervalos na reta real. A diferença é a estrutura auxiliar. Ao invés de uma lista ordenada, usaremos uma árvore de alcance. Construímos uma árvore de alcance com os pontos mais à esquerda dos segmentos de I_{med} e guardamos em τ_{esq}; De forma análoga construímos uma árvore de alcance com os pontos mais à direita dos segmentos de I_{med}.

Vamos considerar o conjunto de segmentos no plano da Figura 25 para ilustrar a construção da árvore de intervalos para segmentos no plano. Inicialmente vamos ordenar os intervalos de I pelo ponto mais à esquerda: $s_1 = [(-8, -5)(-3, -5)]$, $s_2 = [(-3, -1)(6, -1)]$, $s_3 = [(0, 4)(4, 4)]$, $s_4 = [(1, 1)(3, 1)]$, $s_5 = [(6, 9)(12, 9)]$. Criamos um nó raiz r e nele armazenamos o valor da $x_{med} = 1$. Dividimos o conjunto de intervalos em três subconjuntos:

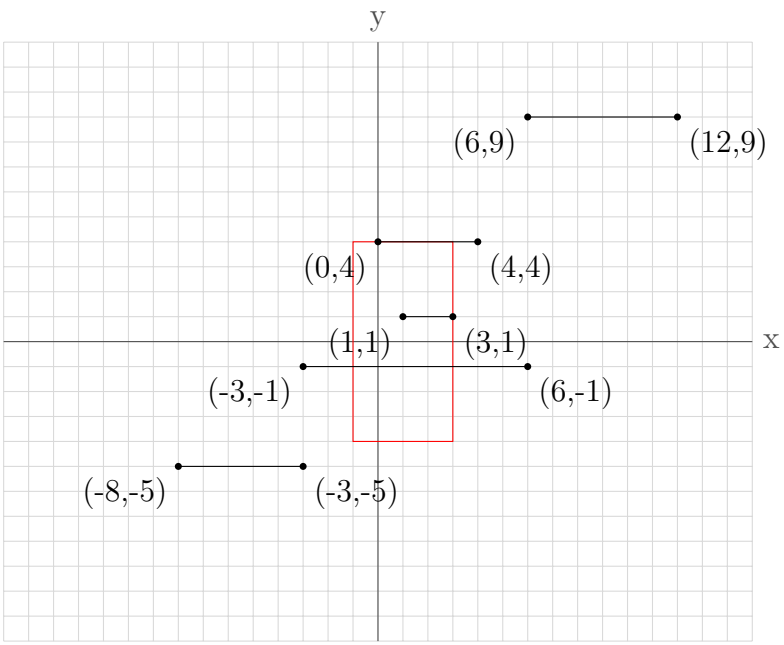


Figura 25 – Em vermelho a região do retângulo de consulta

I_{esq} com o segmento $\{s_1\}$, I_{dir} com o segmento $\{s_5\}$ por fim I_{med} com os segmentos, que contem x_{med}, $\{s_2, s_3, s_4\}$. Isto é, a x-coordenada do segmento está entre o valor x_{med}.Construímos uma árvore de alcance com os segmentos de I_{med} em relação aos pontos mais à esquerda dos segmentos e associamos esta árvore a $\tau_{esq}(r)$. Construímos outra árvore de alcance com os segmentos de I_{med} considerando os pontos mais à direita e associamos a $\tau_{dir}(r)$. A subárvore à esquerda será uma árvore de intervalos sobre I_{esq}; e a subárvore à direita será uma árvore de intervalos sobre I_{dir}. A seguir a Figura 26 ilustra a árvore de intervalos construída. Na Figura 27 temos um exemplo da árvore mais externa da árvore de alcance construída sob os pontos extremos mais à direita dos intervalos no nó$_1^0$.

3.1.4 Consulta para encontrar segmentos em janela

Para obter os segmentos do conjunto S com n segmentos no plano e que estão contidos em uma a janela $J = [x, x'] \times [y, y']$ iremos primeiro consultar uma árvore de alcance construída sobre os $2n$ pontos extremos (todos os pontos de todos os segmentos). Neste trabalho modificamos a árvore de alcance para que construa na raiz um mapa *ponto* $\rightarrow$ *segmento*. Durante a consulta da árvore de alcance, está retornará todos os pontos dentro da janela e acessamos esse mapa da raiz para saber quais são os segmentos associados aos pontos encontrados. Obtemos assim os segmentos que tem ao menos um ponto dentro da janela.

Para obtermos os segmentos cujos pontos extremos estão fora da janela de consulta utilizaremos a árvore de intervalos construída anteriormente. Consultamos a estrutura

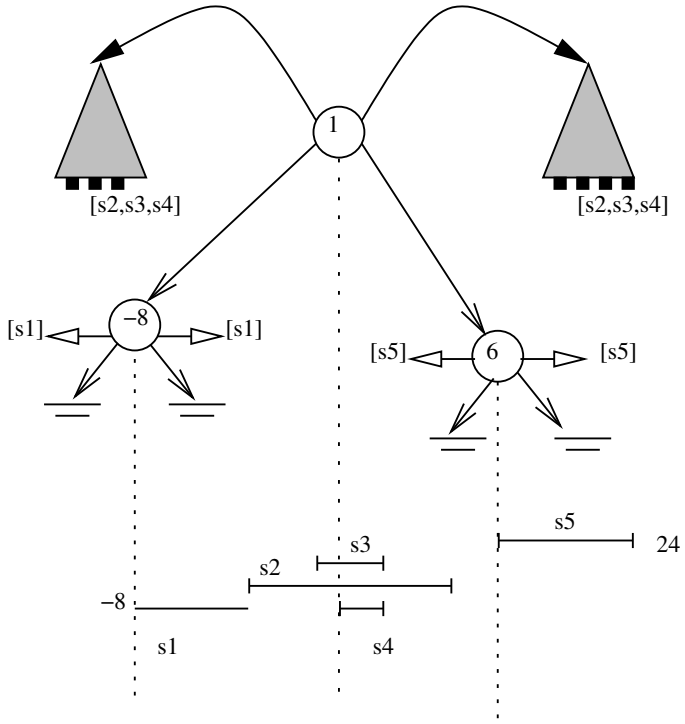


Figura 26 – Árvore de intervalos construída com os segmentos da Figura 25

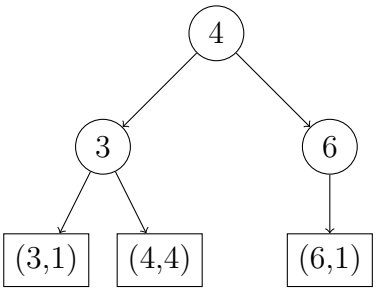


Figura 27 – A árvore de alcance mais externa construída com os pontos extremos à direita associada à direita do nó$_1^0$ da árvore de intervalos da Figura 26.

auxiliar com uma janela que garante que o ponto do segmento cruza uma das laterais da janela. A consulta em $\tau_{esq}(r)$ será com a janela $J'_{-\infty} =]-\infty, x] \times [y, y']$. Similarmente a consulta na árvore associada $\tau_{dir}(r)$ $J'_{\infty} = [x', \infty[\times[y, y']$. A Figura 28 ilustra a janela $J'_{-\infty}$ de consulta. Logo depois descrevemos um algoritmo que consulta a árvore de intervalos e as estruturas associadas.

Vamos acompanhar uma consulta na árvore construída para a Figura 25. Inicialmente queremos descobrir quais segmentos tem ao menos um ponto extremo dentro da janela. Consultamos a árvore de alcance construída com os $2n$ pontos com a janela $J = [-1, 3] \times [-4, 4]$. Esta consulta devolve $(1, 1), (3, 1), (0, 4)$; consultamos o mapa na raiz da árvore e devolvemos os segmentos $[s_3, s_4]$. Seguimos para a consulta da árvore de intervalos. Iniciamos na raiz nó$_1^0$. Como o nó não é folha, checamos se -1 (a extre-

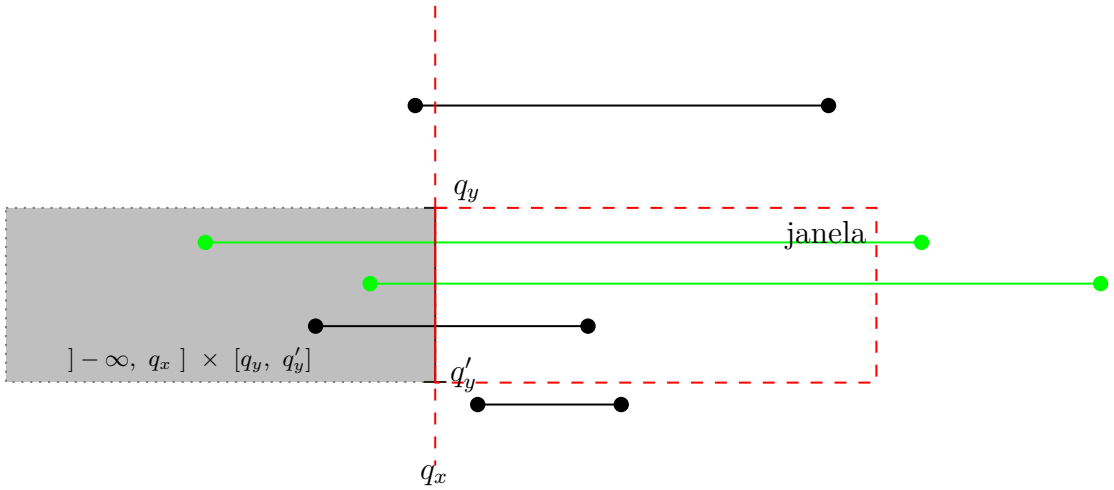


Figura 28 – Consulta na árvore de intervalos para segmentos que cruzam a janela. Em verde os segmentos que queremos reportar com esta consulta

Algorithm 9 Recebe a raiz de uma árvore de intervalos r e uma janela de consulta $J = [x, x'] \times [y, y']$. Devolve todos os segmentos que atravessam J.

```
1: function ConsultaÁrvoreDeIntervalosNoPlano(r, J)
2:     if r não for folha then
3:         if x < r then
4:             L_e ←BuscaEmAlcance2D(τ_esq(r) , ]−∞, x] × [y, y'])
5:             Verifique se o ponto direito de cada elemento de L_e é maior que x'.
6:             ConsultaÁrvoreDeIntervalosNoPlano(r_esq, J)
7:         else
8:             L_d ← BuscaEmAlcance2D(τ_dir(r) , [x', ∞[×[y, y'])
9:             Verifique se o ponto esquerdo de cada elemento de L_d é menor que x.
10:            ConsultaÁrvoreDeIntervalosNoPlano(r_dir, J)
11:        end if
12:    end if
13: end function
```

midade esquerda da janela) é menor que o x_{med} salvo no nó. O x_{med} salvo é 1, logo a consulta está à esquerda de nó$_1^0$. Consultamos a árvore associada τ_{esq}(nó$_1^0$) com a janela $]-\infty, -1] \times [-4, 4]$ devolvendo $(-3, -1)$; utilizando o mapa associado à raiz devolvemos o segmento $[s_2]$. A Figura 29 mostra os resultados obtidos com a nossa implementação para consultar segmentos horizontais no plano.

3.2 Árvore de Segmentos

Em uma consulta em janela, na seção 3.1, a Árvore de Intervalos nos foi útil para encontrar segmentos paralelos aos eixos da consulta. A primeira vista podemos pensar que conseguimos encontrar segmentos com quaisquer orientações com esta estrutura considerando cada segmento apenas suas componentes cartesianas x-intervalo e y-intervalo.

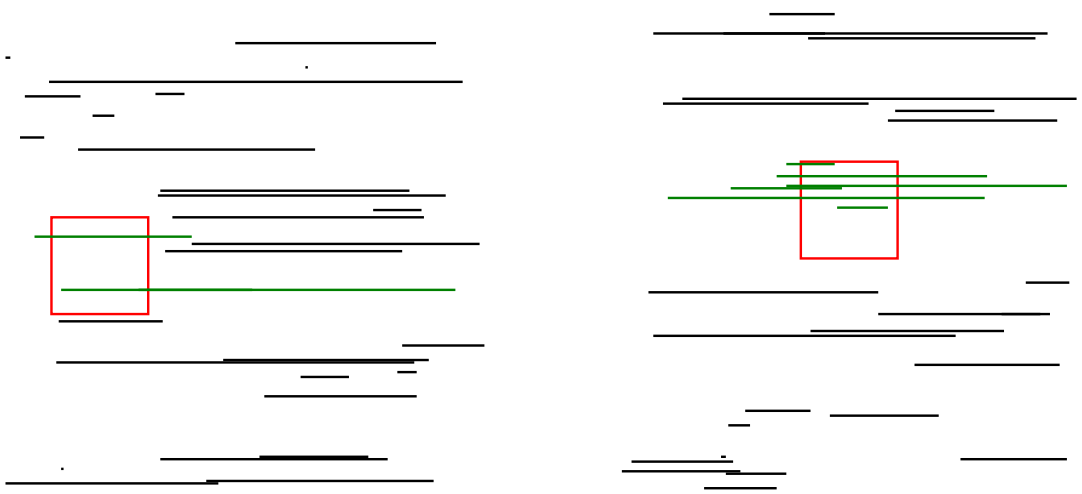


Figura 29 – Resultados das consultas em janela com Árvore de Intervalos no plano

No melhor caso, esta alternativa funcionará bem e a maioria dos segmentos terá suas componentes intersectando a janela. No pior dos casos, a consulta reportaria segmentos que possui componentes que tanto seu x-intervalo quanto seu y-intervalo respeitam a consulta em janela, porém, o segmento não necessariamente cruza a consulta. Como por exemplo na figura 30.

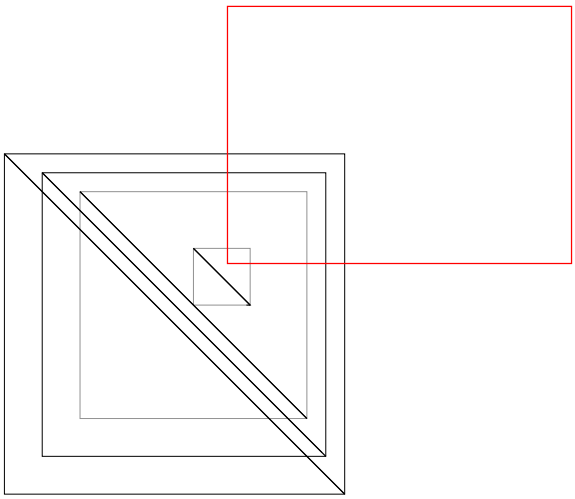


Figura 30 – Consulta de segmentos com inclinação e com árvore de intervalos no pior caso. Em vermelho a janela de consulta.

Para conseguirmos consultar segmentos com inclinação adotaremos a mesma estratégia da seção 3.1. Iremos portanto distinguir segmentos que tem um ponto extremo dentro da janela de segmentos que cruzam as extremidades da janela. O primeiro caso conseguimos reportar com uma árvore de alcance. Para encontrar o segundo tipo de segmento iremos realizar uma consulta de intersecção com cada uma das quatro arestas que compõem a janela. Iremos demonstrar como realizar consultas apenas com uma borda vertical. Para consultas sobre as bordas horizontais, uma abordagem simétrica pode ser

utilizada. Uma árvore de segmentos pode ser construída em tempo de ordem $O(n\log(n))$. A consulta consegue reportar todos os segmentos que intersectam a janela em tempo $O(\log^2(n)+k)$ onde k é o numero de segmentos reportados (BERG et al., 2008a).

3.2.1 Árvores de Segmentos para consultas unidimensionais

Vamos reduzir o problema de consulta em uma janela $J = [q_x, q'_x] \times [q_y, q'_y]$ para quatro consultas unidimensionais. Vamos construir uma árvore de segmentos para reta real, e depois expandimos para o plano. Seja $I = [x_1, x'_1], [x_2, x'_2], ..., [x_n, x'_n]$ o conjunto dos n intervalos na reta real. Queremos construir uma estrutura de dados capaz de retornar os intervalos que contêm q_x. Seja $p_1, p_2, ..., p_m$ a lista de todos os pontos extremos de cada intervalo, ordenados do menor para o maior. Iremos construir a partir dos m pontos dos intervalos de I um conjunto de intervalos elementares. Os intervalos elementares consistem de intervalos abertos entre dois pontos extremos consecutivos p_i e p_{i+1}, alternados com um intervalo unitário fechado de apenas um ponto extremo $[p_i, p_i]$. A razão para tratarmos os pontos extremos como intervalos fechados é pelo fato de a resposta da consulta não ser necessariamente a mesma dentro de um intervalo e nos seus pontos extremos. Iremos então construir uma árvore binaria τ em que suas folhas armazenam os intervalos elementares. Denotaremos o intervalo correspondente de uma folha μ como $Int(\mu)$.

$$]-\infty, p_1[, [p_1, p_1],]p_1, p_2[, [p_2, p_2]...,]p_{m-1}, p_m[, [p_m, p_m],]p_m, +\infty[$$

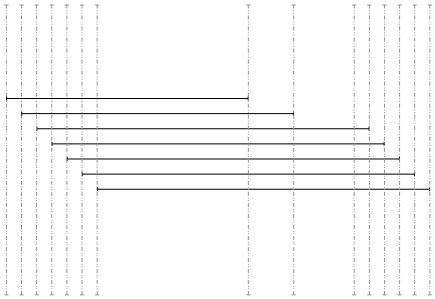


Figura 31 – Intervalos no plano seccionados em cada p_n

Há também um intervalo associado a cada nó não folha que é formado pela a união dos intervalos dos nós filhos. Logo, o intervalo associado à raiz da árvore de segmentos é o intervalo $]-\infty, +\infty[$. Chamaremos o nó v que guarda o valor $[p, p'_i]$ de v_{p,p'_i}. Munidos da árvore binária balanceada τ construída com os intervalos elementares, inserimos cada intervalo i na árvore τ de forma que o nó $v_{p_i,p'_i} \subseteq i$. Como um intervalo pode conter muitos intervalos elementares. A figura 32 mostra a quantidade de intervalos elementares contidos em um segmento s. Note que é bom armazenar s no nó v pois s contém todos os intervalos elementares obtidos a partir de v.

A construção de uma árvore binária balanceada será feita de baixo-para-cima. Construímos uma fila f com os intervalos elementares ordenados pelo seu valor mais

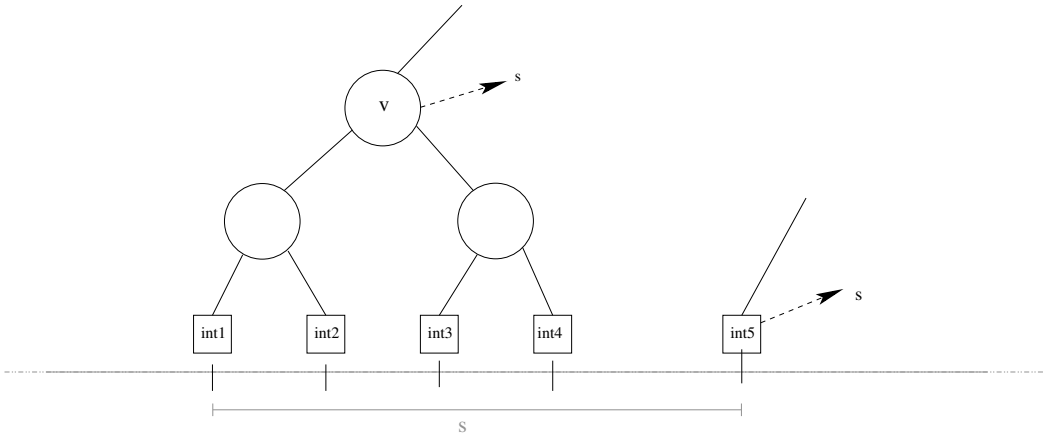


Figura 32 – Árvore de segmentos representando um intervalo s

à esquerda. Ao iniciar a construção pegaremos os dois primeiros valores de f, unimos seus intervalos e colocamos novamente no fim da fila. Fazemos isto até restar somente um intervalo. Segue um procedimento na Figura 33 para a construção da árvore de segmentos para intervalos unidimensionais. Porém, o numero de intervalos elementares nesta fila deve ser potencia de 2 pois somente assim conseguiremos construir a árvore par-a-par. Ou seja, para conseguirmos construir a árvore baixo-cima precisamos antes unir intervalos elementares até que o numero de intervalos seja uma potencia de 2. Fazemos isso percorrendo a fila e unindo intervalos par-a-par ordenadamente e preservando sua posição na fila.

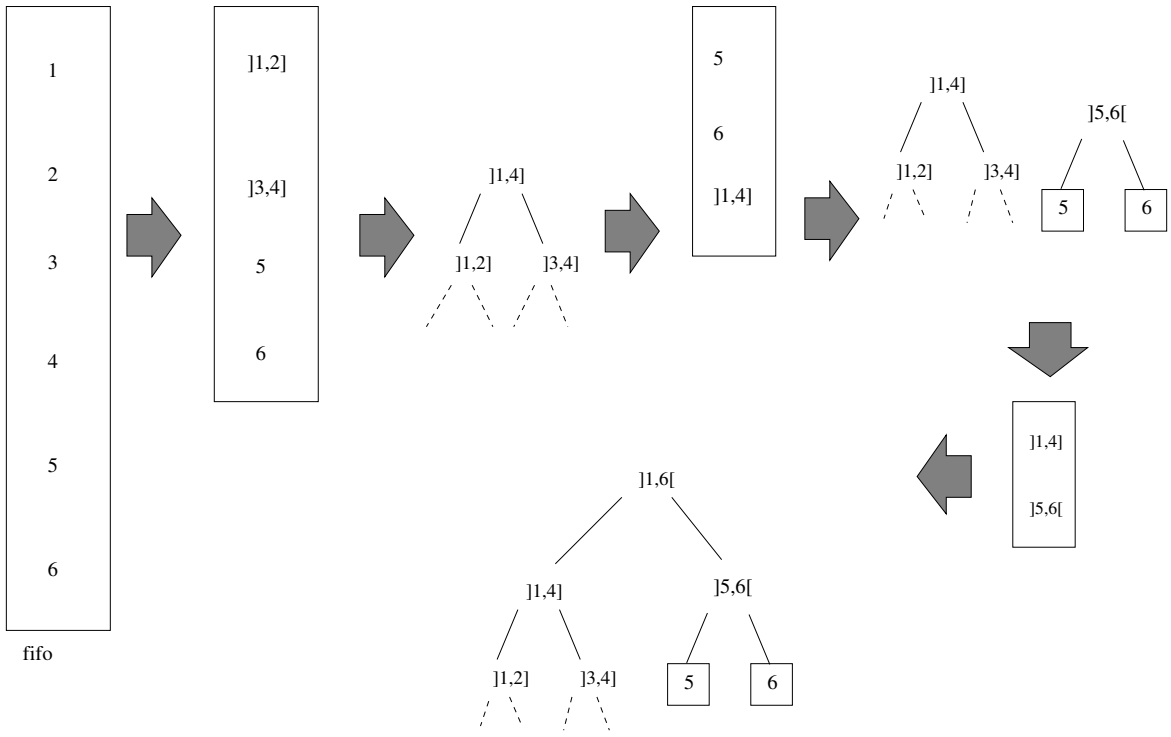


Figura 33 – Construímos a árvore de baixo-para-cima juntando elementos da fila par-a-par

Algorithm 10 Recebe uma lista de intervalos elementares I ordenada, retorna a fila de nós de tamanho $\lfloor(\log_2 n)\rfloor$ com intervalos unidos

```
1: function ConstróiFilaAuxiliar(I)
2:     Coloque todos os intervalos elementares (em ordem) em uma fila
3:     Calcule tamanho mínimo da fila: t_min = 2^⌊(log2 |I|)⌋)
4:     Inicie um contador
5:     while |I| > t_min do
6:         Remova da fila os intervalos elementares n e n + 1
7:         Crie um nó v, e guarde o seu valor com a união do intervalo elementar n e de
       n + 1
8:         Associe à subárvore a esquerda de v um nó folha com o valor do intervalo n
9:         Associe à subárvore a direita de v um nó folha com o valor do intervalo n + 1
10:        Insira o nó v na posição n.
11:        Incremente o contador
12:    end while
13: end function
```

Por fim, teremos uma árvore binária τ com intervalos elementares como folhas. Precisamos inserir os intervalos nos nós. Para cada nó v da árvore iremos inserir o intervalo $[x, x']$ se o $Int(v) \subseteq [x, x']$ e o $pai(v) \nsubseteq [x, x']$. A árvore seguindo estes princípios é chamada de árvore de segmentos. A seguir o algoritmo para inserir os intervalos na árvore τ.

Algorithm 11 Recebe a raiz de uma árvore binária de intervalos elementares v, e um intervalo $s = [x, x']$. Retorna a raiz v' com o intervalo inserido nos nós cujo $Int(v) \subseteq s$

```
1: function InsereSegmentoÁrvoreSegmentos(v, [x, x'])
2:     if Int(v) ⊆ [x, x'] then
3:         Guarde o valor de s em v
4:     else
5:         if filho_esq(v) ∩ [x, x'] ≠ ∅ then
6:             InsereSegmentoÁrvoreSegmentos(filho_esq(v), s)
7:         end if
8:         if filho_dir(v) ∩ [x, x'] ≠ ∅ then
9:             InsereSegmentoÁrvoreSegmentos(filho_dir(v), s)
10:        end if
11:    end if
12: end function
```

Seja I o conjunto de intervalos da Figura 34. Iniciamos construindo o conjunto de intervalos elementares: $]-\infty, -3[$, $[-3, -3]$, $]-3, -2[$, $[-2, -2]$, $]-2, -1[$, $[-1, -1], \ldots,]6, 7[$, $[7, 7]$, $]7, \infty[$. Inserimos na fila f ordenadamente. Neste caso, o tamanho da fila é 15, e para construirmos a árvore, precisamos reduzir para $2^{\lfloor(\log_2 15)\rfloor} = 2^3 = 8$ elementos na fila. Para isto, vamos iterar sobre nossa fila e unir os intervalos elementares $]-\infty, -3[$ e $[-3, -3]$. Criamos um nó v e valor da união dos intervalos elementares é guardado no nó. $Int(v) =]-\infty, -3]$. Atribuímos a subárvore à esquerda com o menor intervalo elementar e à

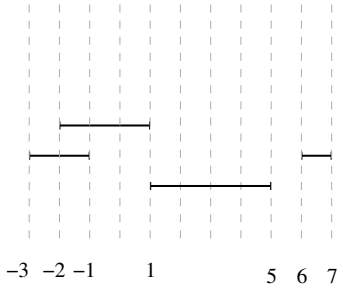


Figura 34 – Intervalos na reta real

direita o maior. E iteramos o próximo valor da fila. Com a fila com tamanho 8, conseguimos construir a árvore com a estratégia de baixo-para-cima. Iniciamos pegando da fila os dois primeiros elementos, unimos e adicionamos no fim da fila. Repetimos até restar apenas um elemento. Por fim, teremos a árvore da Figura 35 a seguir. Agora iremos inserir na árvore os intervalos do conjunto I. Para inserir $[-3,-1]$, começamos pela raiz $v_{-\infty,+\infty}$. Checamos se $]-\infty,+\infty[\subseteq[-3,-1]$, como não é verdade, checamos se o $filho_{esq}(v)\cap[-3,-1]\neq\emptyset$ e se $filho_{esq}(v)\cap[-3,-1]\neq\emptyset$. Sendo verdade apenas para a primeira verificação. Chamamos recursivamente InsereSegmentoNaÁrvore($filho_{esq}(v_{-\infty,+\infty})$, $[-3,-1]$) e seguimos com a inserção. O intervalo $[-3,-1]$ não contém o nó $]-\infty,1]$, e ambos $filho_{esq}(v)=]-\infty,-2]$ e $filho_{dir}(v)=]-2,1]$ intersectam o intervalo $s=[-3,-1]$. Chamamos recursivamente InsereSegmentoNaÁrvore($filho_{esq}(v_{-\infty,1},[-3,-1])$) e InsereSegmentoNaÁrvore($filho_{dir}(v_{-\infty,1},[-3,-1])$). Segue-se estas consultas recursivas até chegar aos nós folhas. Os nós $v_{-3-3}, v_{-3,-2}, v_{-2,-1}, v_{-1,-1}$ contêm o intervalo e portanto o intervalo é associado a estes nós. A figura a seguir é uma ilustração da árvore de segmento construída.

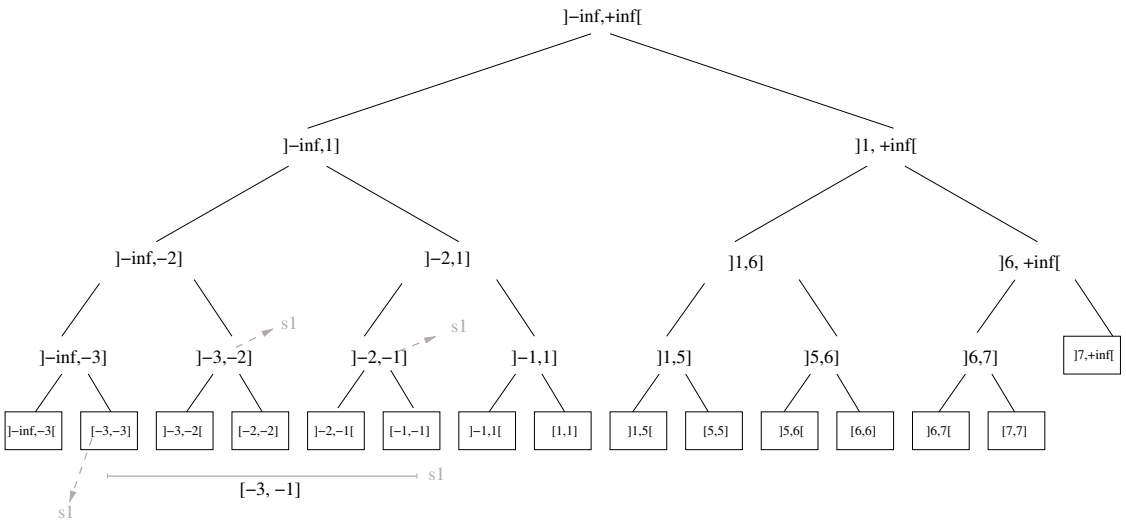


Figura 35 – Árvore de segmentos construída

3.2.2 Consulta unidimensional na Árvore de Segmentos

Para consultar na Árvore construída queremos reportar todos os intervalos $s = [x, x']$ que intersectam a consulta $q = q_x$. Iremos consultar cada nó da árvore e enquanto não for nó folha, checaremos se o $Int(v)$ intersecta a consulta. Em cada visita a um nó, reportamos (se houver algum) os intervalos guardados no nó. A seguir o algoritmo para consulta em uma árvore de segmentos.

Algorithm 12 Recebe a raiz de uma árvore de segmentos v, e um valor de consulta q_x. Retorna todos os intervalos que contem o ponto q_x

```
1: function ConsultaArvoreSegmentos(v, q_x)
2:     Reporte os segmentos guardados em v.
3:     if v não é folha then
4:         if q_x ∈Int(filho_esq(v)) then
5:             ConsultaArvoreSegmentos(filho_esq(v), q_x)
6:         else
7:             ConsultaArvoreSegmentos(filho_dir(v), q_x)
8:         end if
9:     end if
10: end function
```

Vamos acompanhar uma consulta na árvore construída para a Figura 35 para o ponto $q_x = -1$. Iniciamos na raiz $v_{-\infty,+\infty}$, e não há segmentos para serem reportados. Consultamos se o intervalo da subárvore à esquerda intersecta -1. Consultamos recursivamente agora a subárvore à esquerda. Não há intervalos para serem reportados. Consultamos se $-1 \in filho_{esq}(v_{-\infty,-2})$, o que é falso então consultamos $filho_{dir}(v)$. Seguimos recursivamente para $v_{-\infty,-3}$ reportamos os segmentos e seguimos com a consulta recursiva. Consultamos recursivamente se $-1 \in filho_{esq}(v) =]-\infty, -3]$, o que é falso. E consultamos o nó $filho_{dir}(v) = [-3, -3]$ e retornamos o segmento associado S_1.

3.2.3 Estendendo a Árvore de segmentos para janelas 2D

Como dito na introdução da árvore de segmentos, usaremos a árvore de segmentos para consultar as arestas da janela. Iremos analisar como consultar uma aresta da janela usando a árvore de segmentos. Queremos estender o caso unidimensional para consultarmos os segmentos com inclinação. Para isso queremos que a consulta $q = q_x \times [y, y']$ retorne os segmentos inclinados que cruzam esta consulta. Para atingirmos isso, consultaremos se os x-intervalos dos segmentos orientados que contém q_x estão na árvore de segmentos. Para cada segmento encontrado, consultaremos uma estrutura auxiliar ρ que responderá a consulta se o y-intervalo do segmento está dentro da consulta. A estrutura auxiliar para consultar um intervalo será a árvore de alcance apenas para uma dimensão. Para isto usaremos os segmentos salvos nos nós da árvore de segmentos, e atualizaremos a lista de segmentos para uma árvore de alcance.

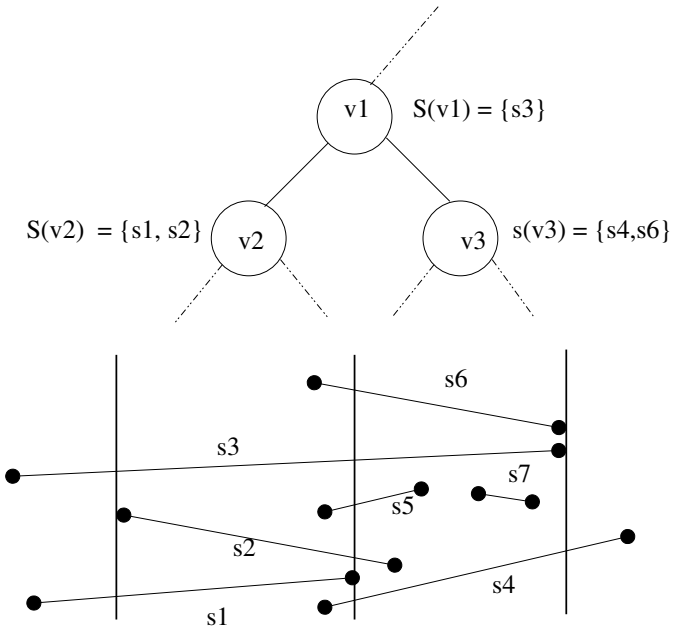


Figura 36 – Árvore de segmentos consultando 2D

Algorithm 13 Recebe a raiz de uma árvore de segmentos v

```
1: function AtualizaEstruturasÁrvoreSegmentos(v)
2:     if houver lista de segmentos s em v then
3:         ConstroiÁrvoreAlcance(s_{y-intervalos})
4:     end if
5:     if houver segmento em filho_{esq}(v) then
6:         AtualizaEstruturasArvoreSegmentos(v)
7:     end if
8:     if houver segmento em filho_{dir}(v) then
9:         AtualizaEstruturasArvoreSegmentos(v)
10:    end if
11: end function
```

A consulta portanto terá uma adaptação. Ao invés de retornar todos os segmentos nos nós, realizaremos uma consulta dos segmentos nos nós quais destes estão dentro do intervalo de consulta $[y, y']$ na árvore ρ.

Por fim, para encontrar os segmentos inclinados em uma janela $J = [x, x'] \times [y, y']$, são necessárias duas árvores de segmento para cada eixo. E serão realizadas quatro consultas para cada aresta da janela. Detectando os segmentos que cruzam a janela e que não necessariamente tem um ponto dentro da janela. Enquanto que para detectar os pontos dentro da janela utilizamos uma árvore de alcance bidimensional e procuramos ao menos um ponto dentro da janela.

Algorithm 14 Recebe a raiz de uma árvore de segmentos v, e um valor de consulta q_x. Retorna todos os segmentos que contêm o ponto q_x

```
1: function ConsultaArvoreSegmentos2D(v, q_x)
2:     Reporte segmentos de Busca1DEmAlcance(ρ, [y, y']).
3:     if v não é folha then
4:         if q_x ∈ Int(filho_esq(v)) then
5:             ConsultaArvoreSegmentos2D(filho_esq(v), q_x)
6:         else
7:             ConsultaArvoreSegmentos2D(filho_dir(v), q_x)
8:         end if
9:     end if
10: end function
```

3.2.4 Resultados

A fim de demonstrar o funcionamento da árvore desenvolvemos uma aplicação que constrói o mapa do Brasil com suas divisas intermunicipais, e conseguimos mostrar esse grande conjunto de pontos de forma eficiente. Inicialmente lemos o arquivo que contem os pontos, criamos segmentos que os representam. Construímos uma árvore de alcance para detectarmos quais pontos estão dentro da janela, e uma árvore para identificarmos as bodas verticais da janela. Criamos um laço que se repete até sairmos da aplicação e para cada iteração consultamos ambas árvores e obtemos um conjunto de segmentos que podemos desenhar. Desenhamos cada um e o programa volta a iterar. Na Figura 37 (à direita) vemos a árvore de segmentos consultando os segmentos que cruzam a janela. Enquanto na Figura 37 (à esquerda) é a união das consultas da árvores de alcance para consultar os pontos extremos dos segmentos com o resultado da consulta na árvore de segmentos e quais segmentos cruzam a janela.

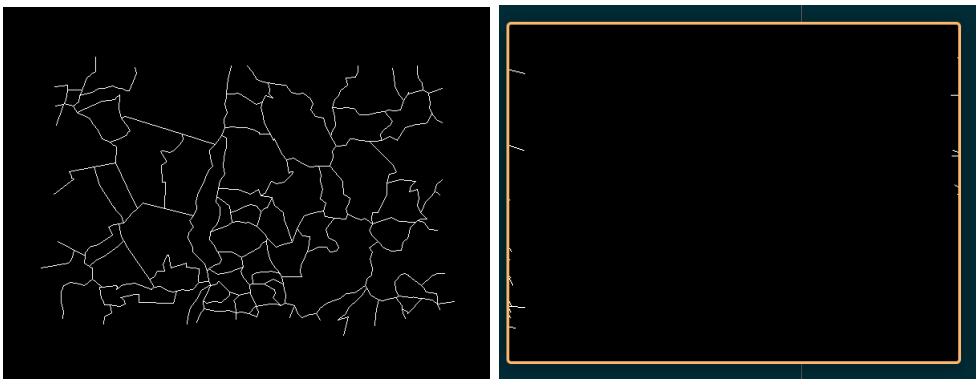


Figura 37 – Janela de consulta do mapa do Brasil (à esquerda). Apenas a árvore de segmentos retornando os segmentos que cruzam as extremidades verticais da janela (à direita).

4 Resultados

Neste capitulo falaremos das nossas implementações e de nossos resultados com algumas delas. As estruturas[1] foram implementadas na linguagem Python, e para validação visual preferimos imagens SVG por serem fáceis de interpretar e de gerar imagens teste. Além da implementação das estruturas desenvolvemos dois casos de testes para validar as estruturas. O primeiro é uma aplicação onde há um mapa com movimento livre com inúmeros pontos. A ideia deste programa era validar tanto a aplicação para jogos 2D quanto 3D. Para um jogo 2D, poderíamos substituir cada ponto por texturas do jogo, e teríamos um mapa virtualmente infinito em dimensões. E portanto, buscamos neste capitulo validar que podemos consultar no plano grandes ordens de grandeza de pontos sem grandes impactos na performance. Validamos esta ideia mostrando os tempos de consulta para grandes valores de pontos. O segundo programa é um mapa do Brasil com grande resolução de segmentos e movimento de câmera livre por este mapa. Validamos a aplicação mostrando que seria inviável ter uma aplicação de tempo real sem as estruturas utilizadas. Construímos cada uma das estruturas de dados apresentadas no texto e cada uma delas tem métodos auxiliares para construir figura SVG com pontos aleatórios com janelas aleatórias, e a construção da árvore em cima do arranjo de pontos desta imagem e a saída do programa como outra figura SVG com os pontos dentro da janela indicados com a cor verde.Todas as estruturas foram construídas visando apenas a consulta em janela. Portanto como demonstrado no texto nos atentamos apenas aos métodos de construção e consulta. As estruturas de consulta para segmentos por sua vez foram construídas, como visto no trabalho até aqui, para consultas das bordas e portanto trabalham em conjunto com as estruturas de consultas de pontos. Para interpretarmos as imagens utilizamos a biblioteca *xml*[2] e interpretamos as figuras SVG como XML.Usamos a mesma biblioteca tanto para a leitura quanto escrita das figuras após a consulta. Nos resultados obtidos utilizamos um computador com as configurações: Processador Intelő Core i7-7500U[3] com 8GB de RAM DDR4 2666MHz.

[1] Afim de a aplicação ser agnóstica de sistema, utilizamos o programa pipenv que permite criar ambientes Python com as dependencias necessárias. Instruções de uso estão disponíveis no arquivo README do projeto. https://github.com/lrdass/theia

[2] Biblioteca built-in Python para lídar com arquivos XML https://docs.python.org/3/library/xml.etree.elementtree.html

[3] Referencia completa do CPU utilizado https://ark.intel.com/content/www/br/pt/ark/products/95451/intel-core-i7-7500u-processor-4m-cache-up-to-3-50-ghz.html

4.1 Aplicação árvore de alcance

Em uma aplicação tridimensional poderíamos construir cenas arbitrariamente grandes de tal forma que organizaríamos os objetos da cena $3D$ com uma árvore de alcance de 3 dimensões. Consultaríamos nesta árvore, portanto, o cubo representado pela câmera como na Figura 38. Reportando somente quais estruturas estão dentro da janela e então enviaríamos para o pipeline gráfico 1.2 para desenharmos na tela. Podemos pensar em uma limitação para um jogo que precisa manter todos os objetos geométricos no pipeline. O número de objetos geométricos seria limitado pela memória do pipeline. A partir destas constatações podemos recriar este comportamento construindo uma árvore de alcance tridimensional com todos os seus objetos, e carregamos para memória da placa de video apenas o que é retornado da consulta. Necessitando de apenas uma tela de carregamento para construir a árvore.

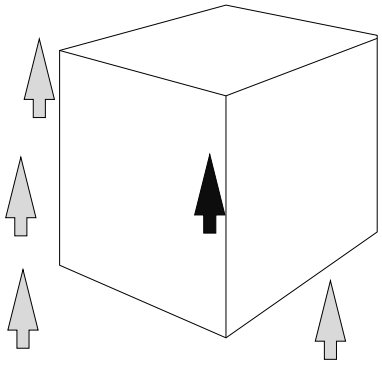


Figura 38 – Exemplo de cena tridimensional com uma consulta retornando apenas os objetos dentro da janela. Considere as setas como figuras tridimensionais no espaço. E o cubo como a janela de consulta.

Para justificarmos esta aplicação, construímos uma versão simplificada do problema em duas dimensões visto na Figura 39. Criamos aleatoriamente pontos no plano e construímos uma árvore de alcance bidimensional. Em cada laço de execução do programa, consultamos a árvore com a janela, e desenhamos apenas os pontos dentro da janela. A solução trivial deste problema sem as árvores de alcance é consultar cada ponto e então desenha-lo e deixar o algoritmo de recorte (HUGHES et al., 2014) desenhar na tela. Porem, ainda iteraria sobre estes para poder constatar que não estão na janela. Enquanto utilizando a árvore, temos uma maneira eficiente de consultar e desenhar apenas os pontos dentro da janela.

Como jogos são aplicações de tempo real, temos que pensar em restrições de tempo. Jogos modernos tem objetivos de entregar entre 30 e 60 quadros por segundo. Considerando o pior caso temos $\frac{1}{30} \approx 0.0334$ segundos para computações entre cada quadro.

Com base na Tabela 1 temos bastante confiança de que a estrutura de dados está dentro do tempo limite de computação para cada quadro desenhado, até mesmo

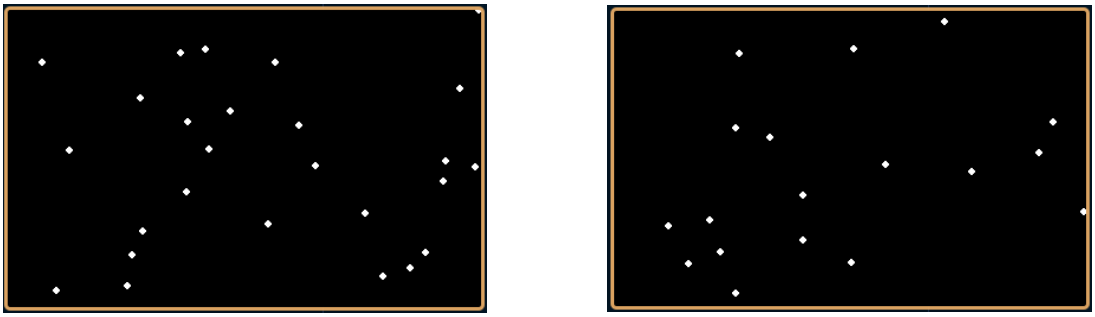


Figura 39 – Aplicação construída com pontos no plano e consultas em tempo real

Pontos	Média do Tempo (s)	Desvio Padrão (s)
1000	0.00012955069541931	$1.3212002262037 \times 10^{-5}$
10000	0.00013832251230876	$2.0937153306017 \times 10^{-5}$
100000	0.00015597189626386	$2.8073955196226 \times 10^{-5}$

Tabela 1 – Tabela comparativa do número de pontos e o tempo para reportar os pontos em uma janela proporcional ao tamanho do conjunto de pontos testado

Incremento médio tempo consulta(s) para cada 10^n pontos	Desvio Padrão (s)
$0,00001321$	$6.276986896593 \times 10^{-6}$

Tabela 2 – Há um incremento médio de 13,21 microssegundos para cada 10^n pontos na consulta

aumentando o número de pontos. Mostrando que o crescimento com um fator de 10^n o tempo da consulta ainda permanece na casa dos 0.1 milissegundos.

4.2 Aplicação árvore de segmentos

Em aplicações de tempo real pode ser que exista apenas um objeto com grande complexidade. Programas que permitem ilustração em tempo real, por exemplo, estão mais interessados em conhecer se determinado segmento do objeto sendo desenhado está dentro da janela. Ou mapeamento tridimensional de um sistema cardiovascular em tempo real em que temos uma malha de um único objeto com grande complexidade e, para visualizá-lo, temos que carregar apenas um recorte deste objeto complexo. Em OpenGL (VRIES, 2015) temos um arranjo com as posições dos vértices chamado $VertexArray$ e um segundo arranjo que contém uma ordem de cada vértice para formar os triângulos chamado *ElementArray*. Podemos portanto interpretar estes dois arranjos e construir os segmentos pois sabemos que cada aresta de um triângulo é um segmento. E assim utilizar estes para construir e consultar com a árvore de segmentos. Para simplificarmos este problema tridimensional, desenvolvemos uma aplicação mostrado na Figura 37 (à direita) que navega pelo mapa do Brasil com alta resolução de segmentos em tempo real.

A consulta linear está inviável para uma aplicação de tempo real demonstrado na

Média do Tempo (s)	Desvio Padrão (s)
0.005857	0.003753

Tabela 3 – Utilizando a estrutura de dados para consultar os segmentos

Media do Tempo (s)	Desvio Padrão (s)
0.148667	0.017494

Tabela 4 – Sem a estrutura de dados consultando linearmente

Tabela 4. Imaginando a mesma restrição de 30 quadros por segundo tendo $\frac{1}{30} \approx 0.0334$ segundos para calcular um quadro seria inviável atingir o objetivo sem a árvore de segmentos. A consulta linear alcançaria no máximo $\frac{1}{0.1487} \approx 6.72$ quadros por segundo nos nossos experimentos.

5 Conclusão

Objetivo deste trabalho era buscar formas de consultar por polígonos para o desenvolvimento de jogos e aplicações gráficas de tempo real. O objetivo era demonstrar que não só haveria um ganho de performance para determinados cenários quanto demonstrar que existem aplicações que sem otimizações como as que foram apresentadas simplesmente os problemas não poderiam ser executadas em tempo hábil.

Com a implementação de árvores KD constatamos qual seria o pior caso para aceitar entre as consultas de janela em árvore. Sendo ambas, árvore KD e árvore de alcance flexíveis e úteis para consulta de pontos em N-dimensões com uma janela. A partir do momento que estudamos árvores de alcance, sempre utilizávamos esta estrutura quando precisávamos consultar por pontos em janelas por ser mais otimizada para este objetivo fim, sendo mais performática até no pior caso de sua consulta. Com o experimento descrito na seção 4.1 da aplicação de pontos em um plano validamos a ideia inicial do texto. Visto que a consulta cresce de forma marginal com o crescimento dos pontos que iremos consultar, e portanto, a estrutura atinge os objetivos esperados. Poderíamos, portanto, realizar a construção de cenas arbitrariamente grandes para jogos 2D e 3D com custo marginal de tempo para aplicação. A árvore de intervalos se mostrou um excelente caso base para a árvore de segmentos. Conseguimos encontrar aplicações que seria perfeitamente aplicável para algum projeto gráfico de tempo real. Como por exemplo, poderíamos consultar em uma cena 2D apenas objetos cujas dimensões espaciais intersectem com a janela, ou seja, onde a geometria do objeto não seja de interesse apenas se está ou não presente na janela. Em jogos 2D os objetos da cena são usualmente imagens retangulares e portanto seria um excelente caso de uso. A aplicação de árvore de segmentos da Seção 4.2 nós demonstramos que tal classe de aplicação só é reproduzível com alguma estrutura para consulta como a estudada neste trabalho.

Demonstramos que para objetos com grande complexidade de segmentos, uma consulta linear em grandes resoluções de segmentos inviabilizaria uma aplicação de tempo real. Com as estruturas de dados apresentamos uma solução possível para esta classe de problemas e demonstramos que ela executa as consultas dentro da janela limite da aplicação. Percebemos que quando a aplicação trabalhava com objetos geométricos mais complexos e em dimensões maiores (já para $2D$), as estruturas de dados eram compostas por outras estruturas auxiliares. Notamos que são estruturas que não estão limitadas a alguma dimensão específica e portanto são facilmente portáveis para dimensões maiores, e para este trabalho, são adequadas para o uso em jogos tridimensionais. As estruturas estudadas portanto permitem aceleração do pipeline gráfico por encurtarem o número de operações necessárias na aplicação e por diminuir o número de vértices enviados à placa de

video para serem computados e recortados. Além disso, permite aplicações que sem estas estruturas seriam inviáveis. Com isso, concluímos que as estruturas de dados estudadas dos experimentos das seções 4.1 e 4.2 são eficientes na recuperação de objetos geométricos e poderiam ser aplicadas no desenvolvimento de jogos, e para otimizar e extrair o máximo de desempenho do sistema gráfico que esteja sendo produzido.

Referências

BENTLEY, Jon Louis. **Decomposable Searching Problems**. Pittsburgh: Carnegie-Mellon University, 1979. P. 1.

_____. Multidimensional binary search trees used for associative searching. **Communications of the ACM**, ACM New York, NY, USA, v. 18, n. 9, p. 509–517, 1975.

BERG, Mark de et al. **Computational Geometry - Algorithms and Applications**. Berlin: Springer, 2008.

_____._____. Berlin: Springer, 2008. P. 96.

_____. **Computational Geometry - An Introduction**. New York: Springer-Verlag, 1985. P. 331, 360–363.

CORMEN, Thomas H. et al. **Introduction to Algorithms**. Massachusetts: Elsevier, 2002. P. 204, 205, 206, 207.

HUGHES, John F. et al. **Computer Graphics Principles and Practice**. New Jersey: Pearson Education, 2014. P. 1044–1047.

TOMAS, Akenine-Moller et al. **Real-time Rendering**. Boca Raton, FL: CRC, 2018. P. 12–27.

VRIES, Joey de. **Learn OpenGL**. New York: Creative Commons, 2015. P. 37, 38, 39.

ANEXO A – Artigo SBC

Algoritmos e estruturas de dados para visualização de polígonos em aplicações de tempo real

Lucas R. S. Suppes[1], Álvaro J. P. Franco[2],

[1]Instituto de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)

lucas.suppes@gmail.com

Abstract. This work presents a study of some data structures and techniques to process windows. We study ways to structure geometric objects in such a way that queries in windows are quickly answered. The data structures that we study were KD Tree, Range Tree, Interval Tree and Segment Tree. This work used all the data structures on the plan. The data structures and the algorithms of construction and query were implemented. Finally, we used our implementation in an application that processes points and segments on the plan. We showed that the structures are efficient for query spatial points for time constraint applications.

Resumo. Este trabalho apresenta um estudo de algumas estruturas de dados e técnicas para processamento de janelas. Estudamos maneiras de estruturar objetos geométricos de tal forma que consultas geométricas são respondidas com eficiência. Neste texto vamos apresentar duas estruturas de dados que estudamos as Árvores de Alcance, e as Árvores de Segmentos. Vamos utilizá-las para consultar objetos no plano portanto este trabalho preparou as estruturas de dados para recuperar objetos no plano. As estruturas de dados e algoritmos de construção e consulta foram implementados. Por fim, utilizamos nossas implementações em uma aplicação que processa pontos e segmentos no plano. Demonstramos que são estruturas eficientes para consultas espaciais de pontos em aplicações com restrições temporais.

1. Objetivos

O objetivo deste trabalho é encontrar e provar a eficácia de estruturas de dados para consultas espaciais visando aplicações em tempo real gráficas. Dado o pipeline gráfico precisamos enviar para o pipeline gráfico apenas os vértices dos polígonos que estão dentro da janela de consulta. Otimizando, portanto, o pipeline gráfico e a aplicação em si caso precise efetuar cálculos sobre os polígonos que estiverem sendo exibidos. Um segundo objetivo deste trabalho é mostrar que existe classe de problemas de consulta sobre polígonos onde há uma grande densidade de segmentos. Para tal classe de problemas o objetivo é mostrar que sem as estruturas estudadas a aplicação não atinge as restrições temporais. E, portanto, apresentar uma solução para esta classe de problemas.

2. Árvores de Alcance

Uma árvore de alcance [Bentley 1979] pode ser implementada considerando várias dimensões. No caso geral, com k dimensões, teremos k níveis de árvores binárias de

busca todas balanceadas. Uma dimensão está relacionada a cada nível. O primeiro nível é formado por uma árvore binária de busca balanceada relacionada a dimensão, digamos, 1. Cada nó desta árvore guarda um valor da dimensão 1, mais uma outra árvore binária de busca balanceada relacionadas à dimensão 2. Cada nó do segundo nível, guarda uma valor da dimensão 2, outra árvore relacionada à dimensão 3. E assim por diante. Neste trabalho consideramos uma árvore de alcance para 2 dimensões ($2D$). Denotamos por τ_r a raiz de uma árvore relacionada a um nó r. Neste caso, o nó r na árvore mais externa e τ_r em uma árvore auxiliar ao nó r. A Figura 1 ilustra uma árvore de alcance $2D$. Note que as folhas da árvore do primeiro nível (à esquerda) aparecem como folhas em uma árvore do segundo nível (à direita). É uma alternativa para consulta de pontos cujo tempo de consulta é superior ao da árvore KD.

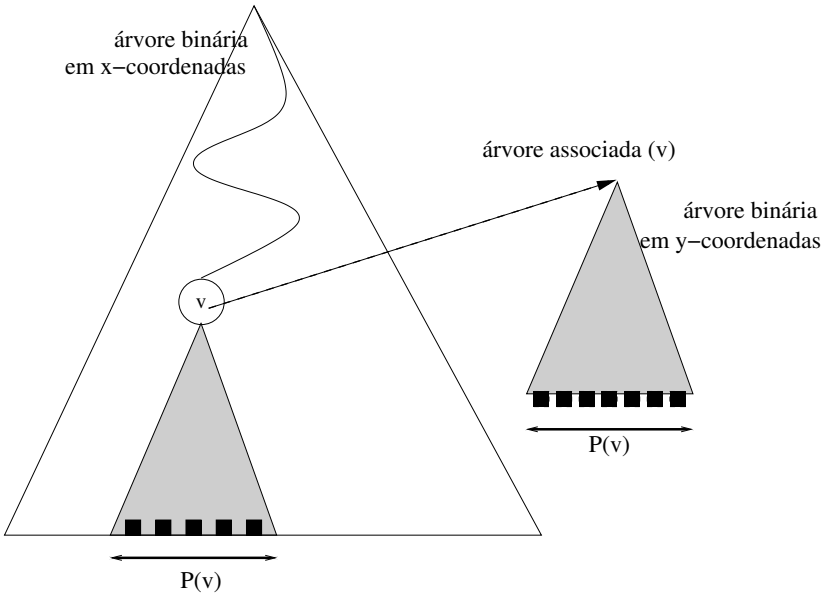


Figura 1. — Árvore de Alcance $2D$. Onde $P(v)$ são os pontos alcançáveis a partir de um nó v.

2.1. Árvore de Alcance $\mathbf{2D}$

Uma árvore de alcance $2D$ é uma versão de duas dimensões para árvores de alcance. A construção considera um dado conjunto de pontos P e pode ser feita da seguinte forma. Temos os pontos na forma $p = (p_x, p_y)$. Ao iniciar a construção, fixamos o eixo x para o primeiro nível; construímos uma árvore binária de busca balanceada T considerando os valores na x-coordenada dos pontos de P; os pontos p serão salvos nos nós folha; para cada nó não folha r de T construiremos uma árvore auxiliar τ_r com todas as folhas alcançáveis a partir de r. Estas árvores serão construídas considerando os valores da y-coordenada. Por fim, associamos τ_r ao nó r. Os nós folhas das árvores guardarão os pontos de P. Uma árvore de alcance pode ser construída em tempo $O(n \log(n))$, onde n é o número de pontos dados no plano. O tempo de uma consulta em uma árvore de alcance é da ordem de $O(\log^2(n) + k)$ onde k são os pontos dentro da janela [Berg et al. 2008a].

Vamos considerar o seguinte conjunto de pontos P da Figura 2 para ilustrar a construção de uma árvore de alcance $2D$.

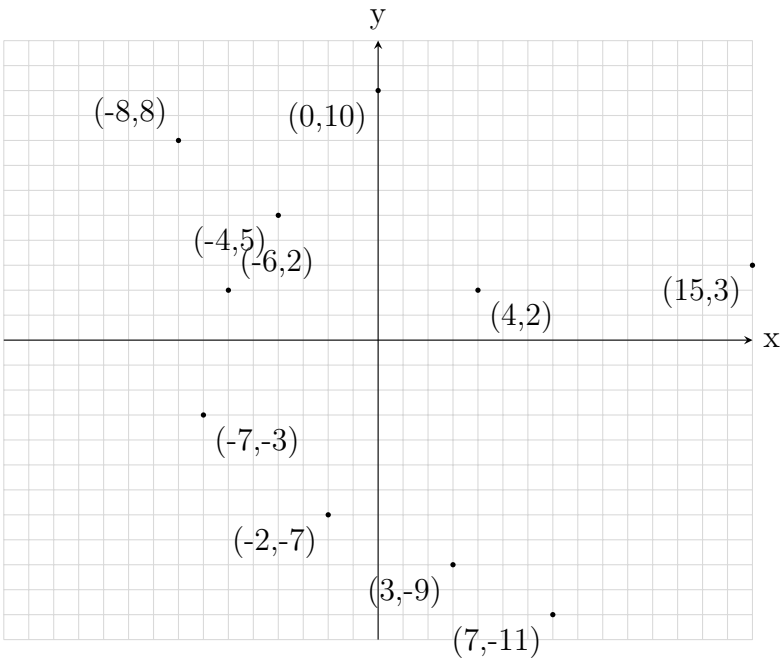


Figura 2. Pontos P no plano.

Para construir uma árvore binária de busca balanceada do primeiro nível, podemos inicialmente ordenar os pontos de P pela x-coordenada. Pegamos a mediana da x-coordenada, x_{med}, e o guardamos em um nó r da árvore. Depois, criamos dois subconjuntos P_1 e P_2 tal que $P_1 = \{p \in P : p_x \leq x_{med}\}$ e $P_2 = \{p \in P : p_x > x_{med}\}$. Criamos uma árvore binária de busca balanceada τ_r considerando o valor da y-coordenada dos pontos de P. A árvore de segundo nível associada ao vértice r será τ_r. O processo continua de forma recursiva sobre os dois novos conjuntos P_1 e P_2. Em seguida, adicionamos o pseudocódigo deste procedimento.

Vamos seguir alguns passos da construção de uma árvore de alcance. Considere os os pontos da Figura 2 ordenados pelo eixo x: $P_{ord(x)} = \{(-8,8), (-7,-3), (-6,2), (-4,5), (-2,-7), (0,10), (3,-9), (4,2), (7,-11), (15,3)\}$. Criamos um nó raiz r e nele armazenamos o valor da $x_{med} = -2$. Dividimos o conjunto de pontos em dois subconjuntos P_1 com os pontos $p_x \leq -2$ e P_2 com os valores $p_x > -2$. Com relação à árvore de segundo nível relacionada ao conjunto inicial, fazemos uma y-ordenação nos pontos de P e construímos uma árvore binária de busca balanceada considerando os valores do eixo y e associamos a árvore resultante à raiz r (ou nó$_1^0$) da árvore do primeiro nível. A subárvore à esquerda de r será uma árvore de alcance sobre P_1; e a subárvore à direita de r será uma árvore de alcance sobre P_2. A Figura 3 ilustra o nó$_1^1$ e a sua árvore associada de segundo nível.

Este procedimento é repetido até que caia na base da recursão que ocorre quando o conjunto de pontos tem somente um ponto p. Neste caso, criamos um nó folha contendo o ponto p e a sua árvore associada de segundo nível também contendo

Algorithm 1 Recebe como entrada um conjunto de pontos P. Devolve o nó raiz de uma árvore de alcance $2D$.

```
1: function ConstróiÁrvoreAlcance2D(P)
2:     Criamos um novo nó r
3:     Construímos uma árvore binária de busca balanceada associada ao nó r sobre
       os pontos P e considerando a y-coordenada. A árvore será denotada por τ_r.
4:     if P contém apenas um ponto then
5:         Guardamos o ponto de P em r.
6:         Associamos τ_r ao nó r.
7:     else
8:         Dividimos P em dois subconjuntos.
9:         P_1 contém os pontos com a x-coordenada menor ou igual que x_med.
10:        P_2 contém os pontos com x-coordenada maior que x_med.
11:        v_esq ← ConstróiÁrvoreAlcance2D(P_1).
12:        v_dir ← ConstróiÁrvoreAlcance2D(P_2).
13:        Criamos um nó v guardando x_med.
14:        Fazemos v_esq e v_dir filhos à esquerda e à direita de v.
15:        Associamos τ_r ao nó r.
16:    end if
17:    return v
18: end function
```

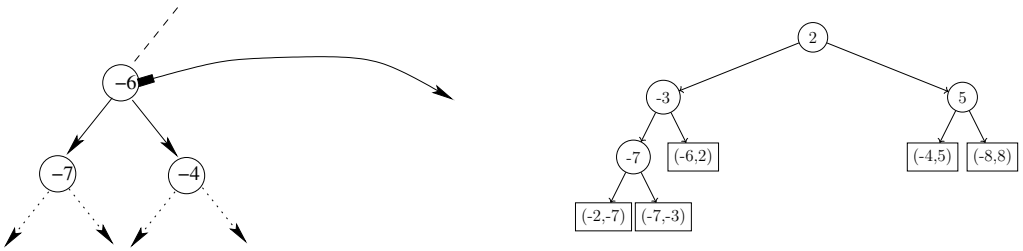


Figura 3. nó$_1^1$ e sua árvore de segundo nível associada da Figura 4.

o ponto p. A Figura 4 a seguir é a representação dos pontos no plano em uma árvore de alcance (primeiro nível) sobre os pontos da Figura 2.

2.2. Consulta dos pontos em árvores de alcance.

Uma consulta 2-dimensional em P é uma busca de quais pontos de P estão entre uma janela de consulta $[x, x'] \times [y, y']$. Um ponto $p = (p_x, p_y)$ está dentro de um retângulo de consulta se $p_x \in [x, x']$ e $p_y \in [y, y']$.

Uma consulta em uma árvore de alcance é a combinação de n consultas em árvore, onde n é a dimensão da árvore de alcance. Na árvore de alcance 2-dimensional temos uma consulta no eixo x seguida por uma consulta na árvore auxiliar τ que foi construída considerando y-coordenada. A consulta com janela, consistirá portanto na união de duas consultas de intervalo unidimensional em árvore.

Uma consulta unidimensional em uma árvore de alcance pode ser visualizada como uma consulta de quais pontos em uma reta s estão em um intervalo de consulta $[x, x']$.

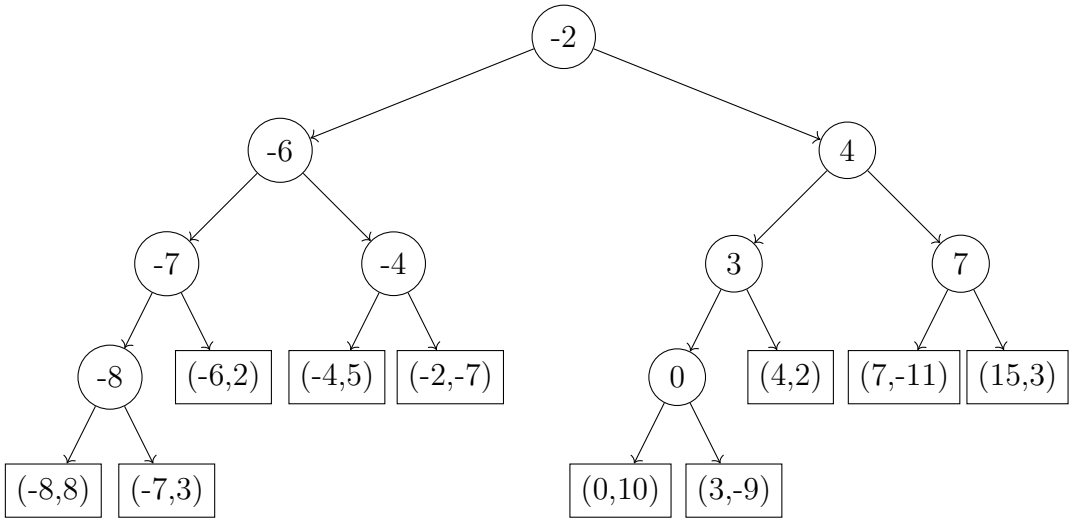


Figura 4. Árvore construída com os pontos P pela x-coordenada.

2.3. Consulta de intervalo unidimensional

Seja uma árvore binária balanceada T. Uma consulta de intervalo em T é tal que queremos todos os nós folhas de T cujo valor esteja dentro do intervalo consultado $[x, x']$.

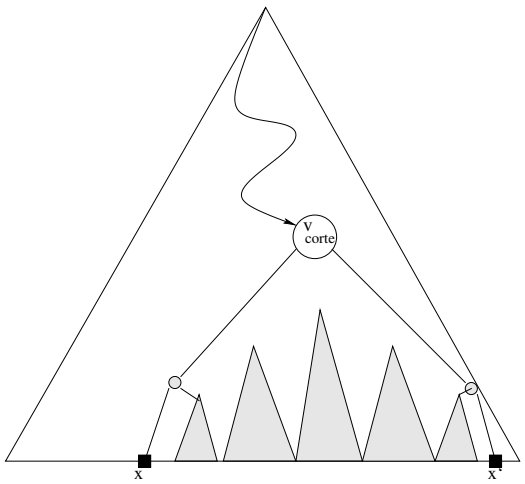


Figura 5. Consulta de alcance unidimensional, e o nó$_{corte}$ sendo o primeiro nó da consulta.

Dado o intervalo da consulta $R = [x, x']$ e a raiz de T realizaremos o seguinte algoritmo: inicialmente precisamos encontrar o primeiro nó cujo valor está contido na consulta. Chamaremos este nó de nó$_{corte}$. Para encontrar o nó$_{corte}$ faremos uma consulta simples que a partir da raiz v checaremos se o valor armazenado em v está dentro do intervalo buscado. Se estiver será nosso nó$_{corte}$. Do contrário checaremos se o valor armazenado em v é maior que x' (denotaremos por $v > x'$). Caso seja, o valor em v é maior que o maior valor da consulta, então faremos recursivamente a busca por nó$_{corte}$ passando o nó$_{esquerda}$ de v. Caso $v \leq x$ realizaremos a consulta por nó$_{corte}$ recursivamente em nó$_{direita}$ de v.

Munidos de nó$_{corte}$, faremos a consulta para retornar todos os pontos contidos dentro de R na árvore T. A partir de nó$_{corte}$, queremos recursivamente buscar os pontos que são menores que nó$_{corte}$ porém ainda dentro da consulta, e similarmente, os pontos maiores que n_{corte} ainda dentro do intervalo. Assim como buscar os pontos maiores que nó$_{corte}$ e dentro da consulta. [Berg et al. 2008b]

Para buscarmos os valores menores que nó$_{corte}$ iremos realizar uma consulta em profundidade considerando os nós não folha à esquerda a partir de nó$_{corte}$ checando se valor do nó v é maior que x. Quando essa condição não for mais atendida, isso significa que já não mais está dentro da consulta. Neste ponto, checaremos se o nó à direita está dentro da consulta. Caso esteja, reportamos. Durante a busca em profundidade todos os pontos que atendem $v > x$, reportamos a subárvore à direita deste nó, ou seja, temos certeza que os valores desta subárvore está dentro do intervalo consultado.

A mesma busca em profundidade será realizada para os valores maiores que nó$_{corte}$, checando se o valor armazenado no nó é menor ou igual que x' e reportando todas as subárvores à esquerda que atendem essa condição. E no caso onde essa condição seja falsa, checando o nó à esquerda e reportando caso o valor esteja dentro do intervalo. A condição de parada dessa busca em profundidade é caso seja um nó folha, onde deve-se checar se o ponto armazenado na folha está dentro do intervalo. Caso esteja, devemos reportá-lo.

Em seguida apresentamos os pseudocódigos para buscar o nó$_{corte}$ e para consultar de forma unidimensional.

Algorithm 2 Recebe como parâmetro um nó e uma janela. Devolve o primeiro nó cujo valor armazenado esteja dentro do intervalo de consulta.

```
1: function EncontraNóCorte(v, R : [x, x'])
2:     while v não é folha do
3:         if v ∈ R  then return v
4:         else
5:             if v > x' then
6:                 v ← v_esquerda
7:             else
8:                 v ← v_direita
9:             end if
10:        end if
11:    end while
12: end function
```

Algorithm 3 Recebe um nó e uma consulta. Devolve todos os pontos dentro da consulta.

```
1: function BuscaEmAlcance1D(v_corte, R : [x, x'])
2:     if v_corte é folha then
3:         if v_corte ∈ R then
4:             Devolve ponto v_corte
5:         end if
6:     else
7:         v ← filho_esq(v_corte)
8:         while v não for folha do
9:             if x ≤ val(v) then
10:                ReportaSubárvore(v)
11:                v ← filho_esq(v)
12:            else
13:                v ← filho_dir(v)
14:            end if
15:        end while
16:        if v é folha e val(v) ∈ R then
17:            Reporta ponto v
18:        end if
19:        Repete-se as linhas 7 até 18 de forma simétrica, trocando o símbolo >
       por ≤, esquerda por direita e direita por esquerda.
20:    end if
21: end function
```

Na Figura 6 temos a árvore construída em relação a coordenada y.

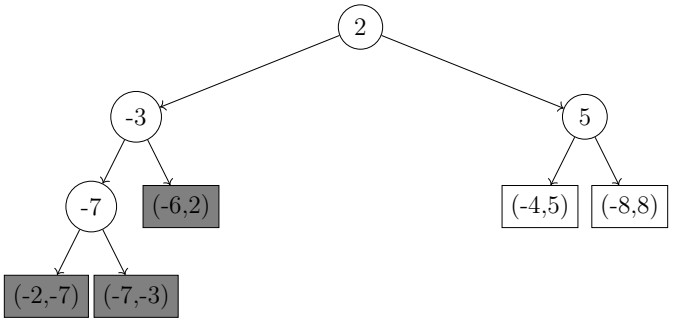
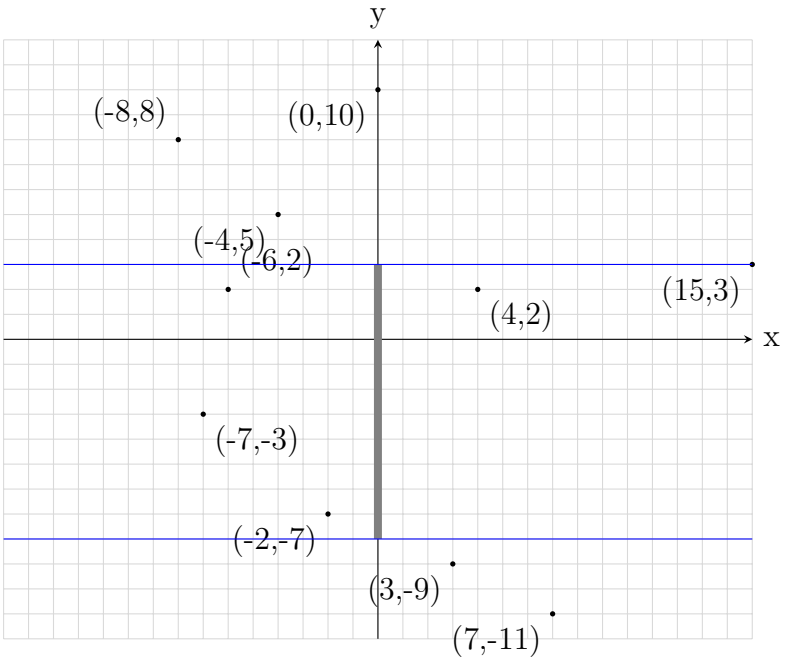


Figura 6. Pontos na y-árvore do nó$_1^1$ da Figura 4.

Vamos ilustrar uma consulta unidimensional considerando está árvore. Seja o intervalo de consulta $R = [-8, 3]$ (veja na Figura 7):

Figura 7. Em azul, a consulta unidimensional em y nos pontos em P.



Começamos pelo nó raiz com valor 2. A condição $-8 \leq 2 \leq 3$ é verdadeira. Portanto nó$_1^0$ será nosso nó$_{corte}$. A partir dele, primeiramente iremos obter os nós v tal que $v \leq 2$. Começamos a busca em profundidade: o nó a esquerda de nó$_1^0$ é nó$_1^1$. Reportamos a subárvore à direita nó$_1^1 = \{(-6, 2)\}$.O valor val(nó$_1^1$) $= -3$ ainda mantém $-8 \leq -3$, e, portanto, continuamos a busca em profundidade. Agora temos o nó$_1^3$. Por sua vez, satisfaz $-8 \leq -7$, reportamos a subárvore à direita $=\{(-7, -3)\}$. A busca em profundidade continuará até o nó$_{folha} = \{(-2, -7)\}$, que satisfaz a consulta.Realizaremos o mesmo procedimento, porem, á partir de nó$_{corte}$ iremos realizar busca em profundidade pela direita. Consultamos que val(nó$_2^1$) > 3, portanto este nó não está dentro do intervalo de consulta. Vamos consultar se a subárvore à esquerda de nó$_2^1$ está dentro da consulta. Este é um nó folha val(nó$_3^2$) $=$

$(-4, 5)$ e verificamos que este também não está dentro da consulta. Por fim teremos os pontos: $\{(-2, -7), (-7, -3), (-6, 2)\}$.

2.4. Consulta Bidimensional

Para realizar uma consulta bidimensional, iremos realizar uma consulta de intervalos unidimensional para cada dimensão. Para cada nó visitado que está dentro da janela de consulta, será feita outra busca unidimensional na árvore τ associada ao nó. Segue pseudocódigo pra consulta bidimensional.

Algorithm 4 Recebe um nó e uma janela. Devolve todos os pontos dentro da consulta.

```
1: function BuscaEmAlcance2D(v_corte, R : [x, x'] × [y, y'])
2:     v_corte ← EncontraNóCorte(τ, x, x')
3:     if v_corte é folha then return val(n_corte) se val(v_corte) ∈ R
4:     else
5:         v ← filho_esq(v_corte)
6:         while v não é folha do
7:             if x ≤ v then
8:                 BuscaEmAlcance1D(filho_dir(v) → τ_associada, [y, y'])
9:                 v ← filho_esq(v)
10:            else
11:                v ← filho_dir(v)
12:            end if
13:        end while
14:        Checar se o ponto na folha está dentro da consulta.
15:        De forma simétrica às linhas 6 até 15, seguimos pelo caminho a partir de
       filho_dir(v_corte).
16:    end if
17: end function
```

Iremos agora exemplificar uma consulta bidimensional. Seja o intervalo de consulta $R = [-6, 1] \times [-8, 3]$. Como explicitado, será uma junção de consultas unidimensionais na x-árvore com a consulta $[-6, 1]$ e uma consulta nas y-árvores associadas entre $[-8, 3]$.

Iniciamos a busca no nó raiz n_1^0. Temos que inicialmente encontrar o n_{corte} e n_0^1 está dentro de $[-6, 1]$. A partir dele faremos uma busca em profundidade para esquerda e para a direita para encontrar todos os valores dentro do intervalo $[-6, 1]$.

Como na Figura 4, agora estamos no nó$_1^0$, este não é folha, então prosseguimos. Consultaremos em profundidade iniciando pela esquerda. À esquerda de nó$_1^0$ é nó$_1^1$, não folha, e portanto checaremos se este ainda é menor que a consulta, ou seja, se a busca em profundidade pode prosseguir consultando os ramos à esquerda da árvore. $-6 \leq -6$, logo, faremos uma Busca1DEmAlcance(nó$_2^2 \rightarrow \tau_{assoc}, -8, 3$). A árvore auxiliar do nó$_2^2$ é τ_{assoc} é a árvore que segue:

Consultaremos nesta árvore os pontos dentro do intervalo $[-8, 3]$, que retornará os pontos em cinza = $\{(-2, -7)\}$. Consultaremos o próximo nó à esquerda,

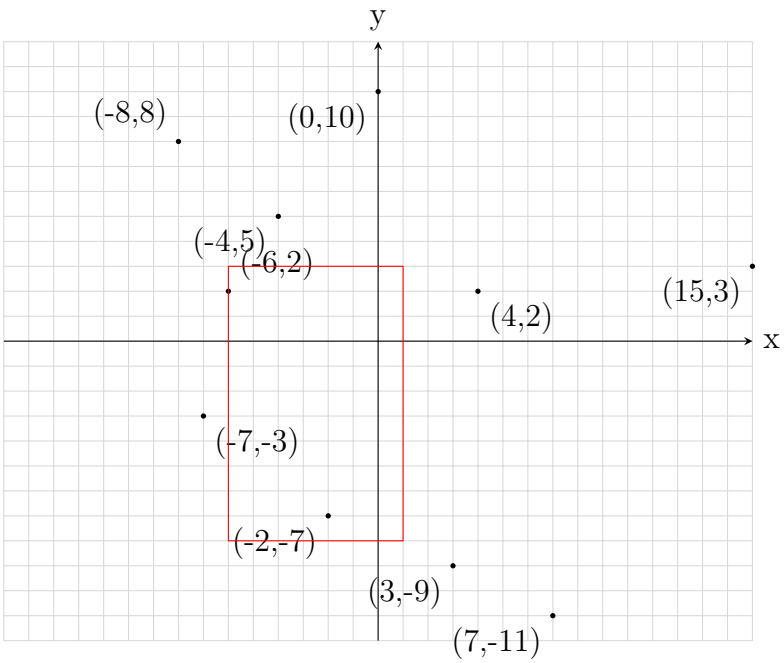


Figura 8. Em vermelho a região do retângulo de consulta

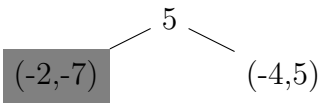


Figura 9. Pontos na y-árvore do nó$_1^1$ da Figura 4.

nó$_1^3$. Por sua vez a consulta $-6 \leq -7$ é falso e iremos avaliar o nó à direita de nó$_1^2$ que é o ponto $(-6, 2)$ e checamos se está na janela e o retornamos. Prosseguiremos com a consulta à direita do nó$_{corte}$ = nó$_1^0$. A figura a seguir ilustra os resultados da nossa implementação.

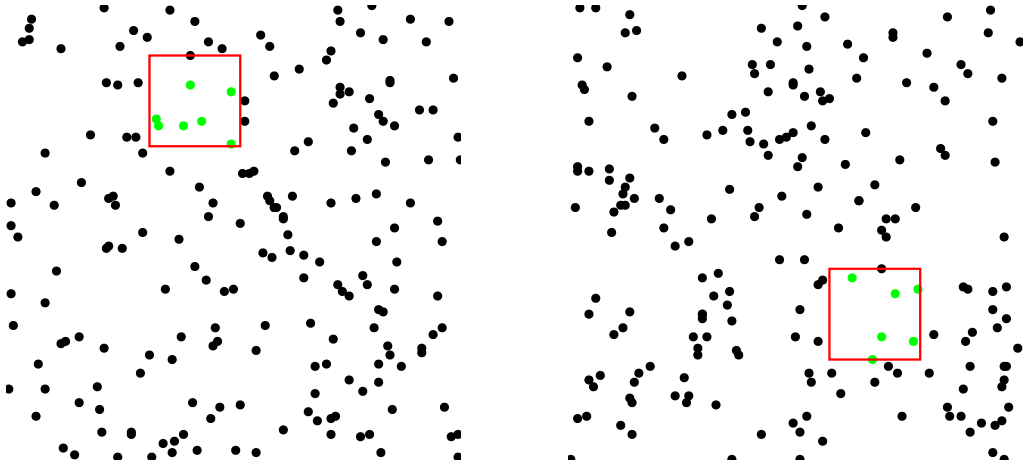


Figura 10. Resultados das consultas em janela usando Árvore de Alcance $2D$

2.5. Árvores de Segmentos

Em uma consulta em janela, a Árvore de Intervalos nos foi útil para encontrar segmentos paralelos aos eixos da consulta. A primeira vista podemos pensar que conseguimos encontrar segmentos com quaisquer orientações com esta estrutura considerando cada segmento apenas suas componentes cartesianas x-intervalo e y-intervalo. No melhor caso, esta alternativa funcionará bem e a maioria dos segmentos terá suas componentes intersectando a janela. No pior dos casos, a consulta reportaria segmentos que possui componentes que tanto seu x-intervalo quanto seu y-intervalo respeitam a consulta em janela, porém, o segmento não necessariamente cruza a consulta. Como por exemplo na figura 11.

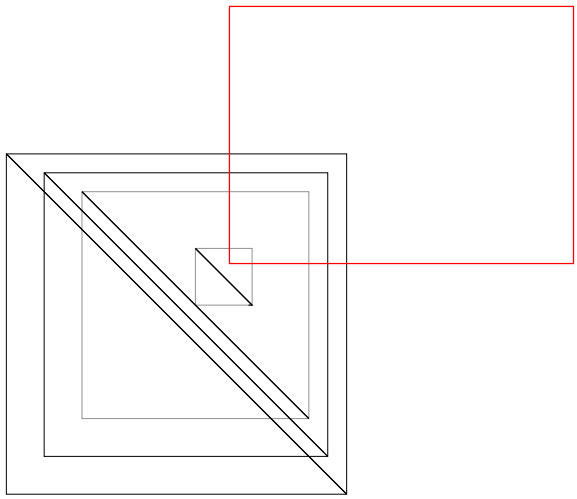


Figura 11. Consulta de segmentos com inclinação e com árvore de intervalos no pior caso. Em vermelho a janela de consulta.

Para conseguirmos consultar segmentos com inclinação adotaremos a mesma estratégia das árvores de intervalo. Iremos portanto distinguir segmentos que tem um ponto extremo dentro da janela de segmentos que cruzam as extremidades da janela. O primeiro caso conseguimos reportar com uma árvore de alcance. Para encontrar o segundo tipo de segmento iremos realizar uma consulta de intersecção com cada uma das quatro arestas que compõem a janela. Iremos demonstrar como realizar consultas apenas com uma borda vertical. Para consultas sobre as bordas horizontais, uma abordagem simétrica pode ser utilizada. Uma árvore de segmentos pode ser construída em tempo de ordem $O(n \log(n))$. A consulta consegue reportar todos os segmentos que intersectam a janela em tempo $O(\log^2(n) + k)$ onde k é o numero de segmentos reportados [Berg et al. 2008a].

2.6. Árvores de Segmentos para consultas unidimensionais

Vamos reduzir o problema de consulta em uma janela $J = [q_x, q'_x] \times [q_y, q'_y]$ para quatro consultas unidimensionais. Vamos construir uma árvore de segmentos para reta real, e depois expandimos para o plano. Seja $I = [x_1, x'_1], [x_2, x'_2], ..., [x_n, x'_n]$ o conjunto dos n intervalos na reta real. Queremos construir uma estrutura de dados capaz de retornar os intervalos que contêm q_x. Seja $p_1, p_2, ..., p_m$ a lista de todos os pontos extremos de cada intervalo, ordenados do menor para o maior. Iremos

construir a partir dos m pontos dos intervalos de I um conjunto de intervalos elementares. Os intervalos elementares consistem de intervalos abertos entre dois pontos extremos consecutivos p_i e p_{i+1}, alternados com um intervalo unitário fechado de apenas um ponto extremo $[p_i, p_i]$. A razão para tratarmos os pontos extremos como intervalos fechados é pelo fato de a resposta da consulta não ser necessariamente a mesma dentro de um intervalo e nos seus pontos extremos. Iremos então construir uma árvore binaria τ em que suas folhas armazenam os intervalos elementares. Denotaremos o intervalo correspondente de uma folha μ como $Int(\mu)$.

$$]-\infty, p_1[, [p_1, p_1],]p_1, p_2[, [p_2, p_2]...,]p_{m-1}, p_m[, [p_m, p_m],]p_m, +\infty[$$

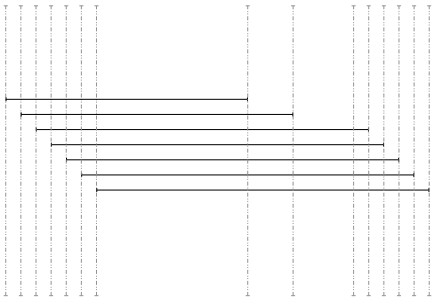


Figura 12. Intervalos no plano seccionados em cada p_n

Há também um intervalo associado a cada nó não folha que é formado pela a união dos intervalos dos nós filhos. Logo, o intervalo associado à raiz da árvore de segmentos é o intervalo $]-\infty, +\infty[$. Chamaremos o nó v que guarda o valor $[p, p'_i]$ de v_{p,p'_i}. Munidos da árvore binária balanceada τ construída com os intervalos elementares, inserimos cada intervalo i na árvore τ de forma que o nó $v_{p_i,p'_i} \subseteq i$. Como um intervalo pode conter muitos intervalos elementares. A figura 13 mostra a quantidade de intervalos elementares contidos em um segmento s. Note que é bom armazenar s no nó v pois s contém todos os intervalos elementares obtidos a partir de v.

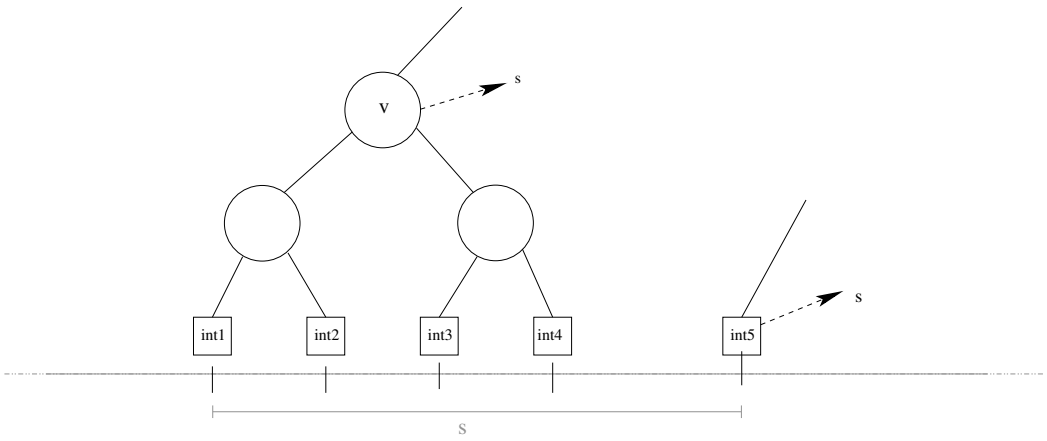


Figura 13. Árvore de segmentos representando um intervalo s

A construção de uma árvore binária balanceada será feita de baixo-para-cima. Construímos uma fila f com os intervalos elementares ordenados pelo seu valor

mais à esquerda. Ao iniciar a construção pegaremos os dois primeiros valores de f, unimos seus intervalos e colocamos novamente no fim da fila. Fazemos isto até restar somente um intervalo. Segue um procedimento na Figura 14 para a construção da árvore de segmentos para intervalos unidimensionais. Porém, o numero de intervalos elementares nesta fila deve ser potencia de 2 pois somente assim conseguiremos construir a árvore par-a-par. Ou seja, para conseguirmos construir a árvore baixo-cima precisamos antes unir intervalos elementares até que o numero de intervalos seja uma potencia de 2. Fazemos isso percorrendo a fila e unindo intervalos par-a-par ordenadamente e preservando sua posição na fila.

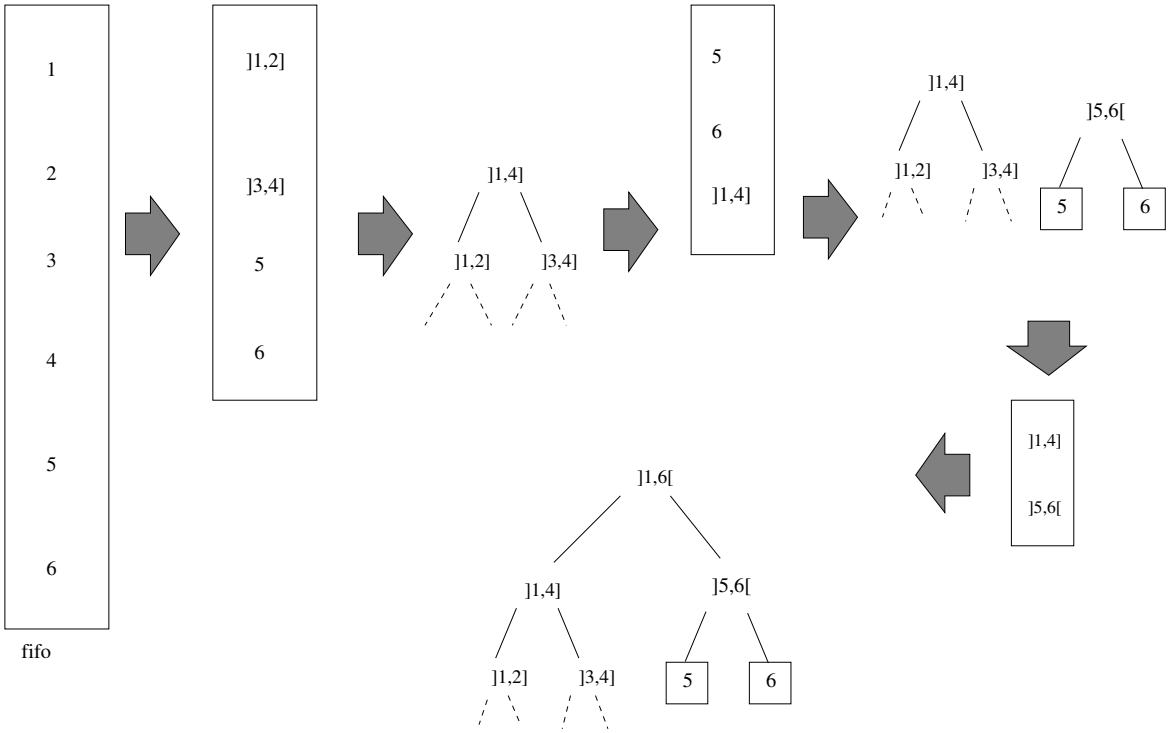


Figura 14. Construímos a árvore de baixo-para-cima juntando elementos da fila par-a-par

Seja I o conjunto de intervalos da Figura 15. Iniciamos construindo o conjunto de intervalos elementares: $]-\infty,-3[$, $[-3,-3]$, $]-3,-2[$, $[-2,-2]$, $]-2,-1[$, $[-1,-1],\ldots,]6,7[$, $[7,7]$, $]7,\infty[$. Inserimos na fila f ordenadamente. Neste caso, o tamanho da fila é 15, e para construirmos a árvore, precisamos reduzir para

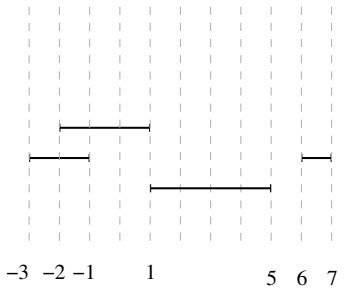


Figura 15. Intervalos na reta real

Algorithm 5 Recebe uma lista de intervalos elementares I ordenada, retorna a fila de nós de tamanho $\lfloor(\log_2 n)\rfloor$ com intervalos unidos

```
1: function ConstróiFilaAuxiliar(I)
2:     Coloque todos os intervalos elementares (em ordem) em uma fila
3:     Calcule tamanho mínimo da fila: t_min = 2^⌊(log2 |I|)⌋)
4:     Inicie um contador
5:     while |I| > t_min do
6:         Remova da fila os intervalos elementares n e n + 1
7:         Crie um nó v, e guarde o seu valor com a união do intervalo elementar n
       e de n + 1
8:         Associe à subárvore a esquerda de v um nó folha com o valor do intervalo
       n
9:         Associe à subárvore a direita de v um nó folha com o valor do intervalo
       n + 1
10:        Insira o nó v na posição n.
11:        Incremente o contador
12:    end while
13: end function
```

Algorithm 6 Recebe a raiz de uma árvore binária de intervalos elementares v, e um intervalo $s = [x, x']$. Retorna a raiz v' com o intervalo inserido nos nós cujo $Int(v) \subseteq s$

```
1: function InsereSegmentoÁrvoreSegmentos(v, [x, x'])
2:     if Int(v) ⊆ [x, x'] then
3:         Guarde o valor de s em v
4:     else
5:         if filho_esq(v) ∩ [x, x'] ≠ ∅ then
6:             InsereSegmentoÁrvoreSegmentos(filho_esq(v), s)
7:         end if
8:         if filho_dir(v) ∩ [x, x'] ≠ ∅ then
9:             InsereSegmentoÁrvoreSegmentos(filho_dir(v), s)
10:        end if
11:    end if
12: end function
```

$2^{\lfloor(\log_2 15)\rfloor} = 2^3 = 8$ elementos na fila. Para isto, vamos iterar sobre nossa fila e unir os intervalos elementares $]-\infty, -3[$ e $[-3, -3]$. Criamos um nó v e valor da união dos intervalos elementares é guardado no nó. $Int(v) =]-\infty, -3]$. Atribuímos a subárvore à esquerda com o menor intervalo elementar e à direita o maior. E iteramos o próximo valor da fila. Com a fila com tamanho 8, conseguimos construir a árvore com a estratégia de baixo-para-cima. Iniciamos pegando da fila os dois primeiros elementos, unimos e adicionamos no fim da fila. Repetimos até restar apenas um elemento. Por fim, teremos a árvore da Figura 16 a seguir. Agora iremos inserir na árvore os intervalos do conjunto I. Para inserir $[-3, -1]$, começamos pela raiz $v_{-\infty,+\infty}$. Checamos se $]-\infty, +\infty[\subseteq [-3, -1]$, como não é verdade, checamos se o $filho_{esq}(v) \cap [-3, -1] \neq \emptyset$ e se $filho_{esq}(v) \cap [-3, -1] \neq \emptyset$.

Sendo verdade apenas para a primeira verificação. Chamamos recursivamente InsereSegmentoNaÁrvore($filho_{esq}(v_{-\infty,+\infty})$, $[-3,-1]$) e seguimos com a inserção. O intervalo $[-3,-1]$ não contém o nó $]-\infty,1]$, e ambos $filho_{esq}(v) =]-\infty,-2]$ e $filho_{dir}(v) =]-2,1]$ intersectam o intervalo $s = [-3,-1]$. Chamamos recursivamente InsereSegmentoNaÁrvore($filho_{esq}(v_{-\infty,1}, [-3,-1])$) e InsereSegmentoNaÁrvore($filho_{dir}(v_{-\infty,1}, [-3,-1])$). Segue-se estas consultas recursivas até chegar aos nós folhas. Os nós $v_{-3-3}, v_{-3,-2}, v_{-2,-1}, v_{-1,-1}$ contêm o intervalo e portanto o intervalo é associado a estes nós. A figura a seguir é uma ilustração da árvore de segmento construída.

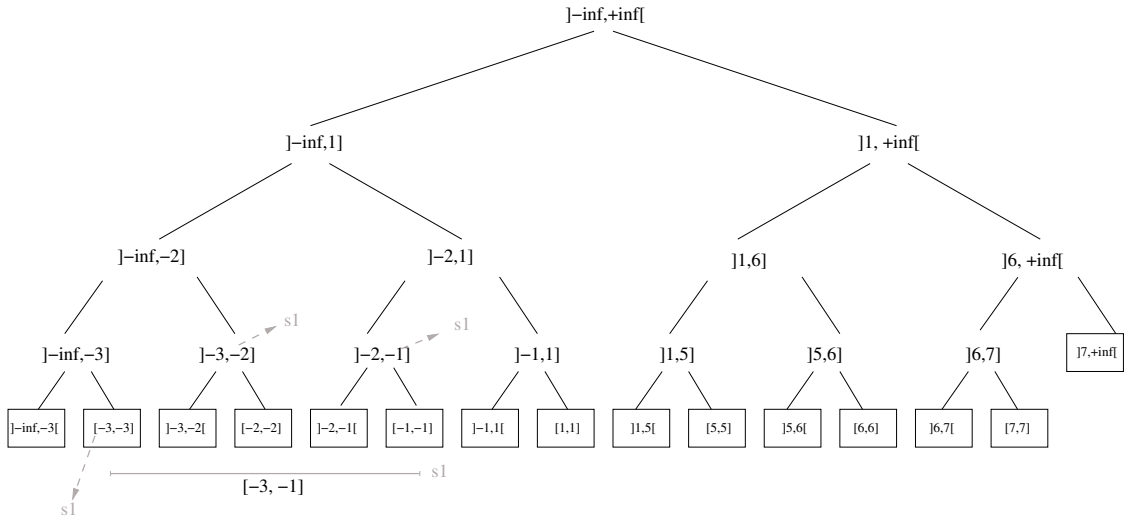


Figura 16. Árvore de segmentos construída

2.7. Consulta unidimensional na Árvore de Segmentos

Para consultar na Árvore construída queremos reportar todos os intervalos $s = [x, x']$ que intersectam a consulta $q = q_x$. Iremos consultar cada nó da árvore e enquanto não for nó folha, checaremos se o $Int(v)$ intersecta a consulta. Em cada visita a um nó, reportamos (se houver algum) os intervalos guardados no nó. A seguir o algoritmo para consulta em uma árvore de segmentos.

Algorithm 7 Recebe a raiz de uma árvore de segmentos v, e um valor de consulta q_x. Retorna todos os intervalos que contem o ponto q_x

```
1: function ConsultaArvoreSegmentos(v, q_x)
2:     Reporte os segmentos guardados em v.
3:     if v não é folha then
4:         if q_x ∈Int(filho_esq(v)) then
5:             ConsultaArvoreSegmentos(filho_esq(v), q_x)
6:         else
7:             ConsultaArvoreSegmentos(filho_dir(v), q_x)
8:         end if
9:     end if
10: end function
```

Vamos acompanhar uma consulta na árvore construída para a Figura 16 para o ponto $q_x = -1$. Iniciamos na raiz $v_{-\infty,+\infty}$, e não há segmentos para serem reportados. Consultamos se o intervalo da subárvore à esquerda intersecta -1. Consultamos recursivamente agora a subárvore à esquerda. Não há intervalos para serem reportados. Consultamos se $-1 \in filho_{esq}(v_{-\infty,-2})$, o que é falso então consultamos $filho_{dir}(v)$. Seguimos recursivamente para $v_{-\infty,-3}$ reportamos os segmentos e seguimos com a consulta recursiva. Consultamos recursivamente se $-1 \in filho_{esq}(v) =]-\infty,-3]$, o que é falso. E consultamos o nó $filho_{dir}(v) = [-3,-3]$ e retornamos o segmento associado S_1.

2.8. Estendendo a Árvore de segmentos para janelas 2D

Como dito na introdução da árvore de segmentos, usaremos a árvore de segmentos para consultar as arestas da janela. Iremos analisar como consultar uma aresta da janela usando a árvore de segmentos. Queremos estender o caso unidimensional para consultarmos os segmentos com inclinação. Para isso queremos que a consulta $q = q_x \times [y, y']$ retorne os segmentos inclinados que cruzam esta consulta. Para atingirmos isso, consultaremos se os x-intervalos dos segmentos orientados que contém q_x estão na árvore de segmentos. Para cada segmento encontrado, consultaremos uma estrutura auxiliar ρ que responderá a consulta se o y-intervalo do segmento está dentro da consulta. A estrutura auxiliar para consultar um intervalo será a árvore de alcance apenas para uma dimensão. Para isto usaremos os segmentos salvos nos nós da árvore de segmentos, e atualizaremos a lista de segmentos para uma árvore de alcance.

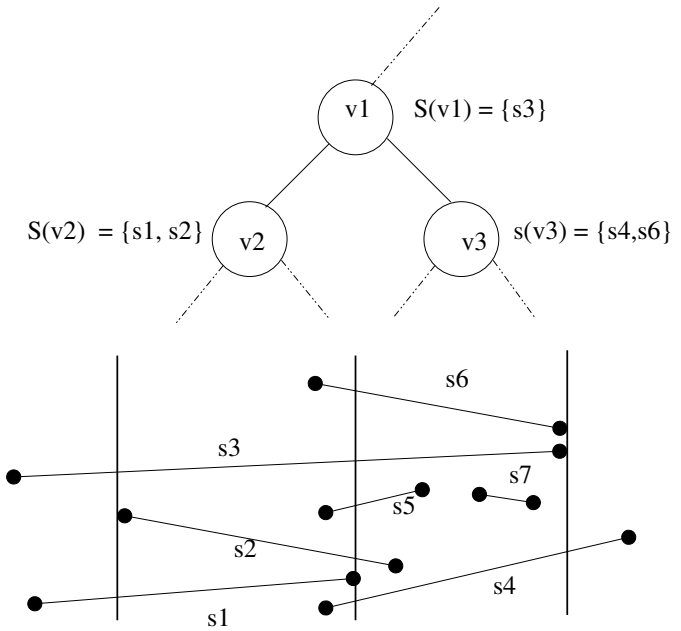


Figura 17. Árvore de segmentos consultando 2D

A consulta portanto terá uma adaptação. Ao invés de retornar todos os segmentos nos nós, realizaremos uma consulta dos segmentos nos nós quais destes estão dentro do intervalo de consulta $[y, y']$ na árvore ρ.

Algorithm 8 Recebe a raiz de uma árvore de segmentos v

```
1: function AtualizaEstruturasÁrvoreSegmentos(v)
2:     if houver lista de segmentos s em v then
3:         ConstroiÁrvoreAlcance(s_{y-intervalos})
4:     end if
5:     if houver segmento em filho_esq(v) then
6:         AtualizaEstruturasArvoreSegmentos(v)
7:     end if
8:     if houver segmento em filho_dir(v) then
9:         AtualizaEstruturasArvoreSegmentos(v)
10:    end if
11: end function
```

Algorithm 9 Recebe a raiz de uma árvore de segmentos v, e um valor de consulta q_x. Retorna todos os segmentos que contêm o ponto q_x

```
1: function ConsultaArvoreSegmentos2D(v, q_x)
2:     Reporte segmentos de Busca1DEmAlcance(ρ, [y, y']).
3:     if v não é folha then
4:         if q_x ∈Int(filho_esq(v)) then
5:             ConsultaArvoreSegmentos2D(filho_esq(v), q_x)
6:         else
7:             ConsultaArvoreSegmentos2D(filho_dir(v), q_x)
8:         end if
9:     end if
10: end function
```

Por fim, para encontrar os segmentos inclinados em uma janela $J = [x, x'] \times [y, y']$, são necessárias duas árvores de segmento para cada eixo. E serão realizadas quatro consultas para cada aresta da janela. Detectando os segmentos que cruzam a janela e que não necessariamente tem um ponto dentro da janela. Enquanto que para detectar os pontos dentro da janela utilizamos uma árvore de alcance bidimensional e procuramos ao menos um ponto dentro da janela.

2.9. Resultados

A fim de demonstrar o funcionamento da árvore desenvolvemos uma aplicação que constrói o mapa do Brasil com suas divisas intermunicipais, e conseguimos mostrar esse grande conjunto de pontos de forma eficiente. Inicialmente lemos o arquivo que contem os pontos, criamos segmentos que os representam. Construímos uma árvore de alcance para detectarmos quais pontos estão dentro da janela, e uma árvore para identificarmos as bodas verticais da janela. Criamos um laço que se repete até sairmos da aplicação e para cada iteração consultamos ambas árvores e obtemos um conjunto de segmentos que podemos desenhar. Desenhamos cada um e o programa volta a iterar. Na Figura 18 (à direita) vemos a árvore de segmentos consultando os segmentos que cruzam a janela. Enquanto na Figura 18 (à esquerda) é a união das consultas da árvores de alcance para consultar os pontos extremos dos segmentos com o resultado da consulta na árvore de segmentos e quais segmentos cruzam a

janela.

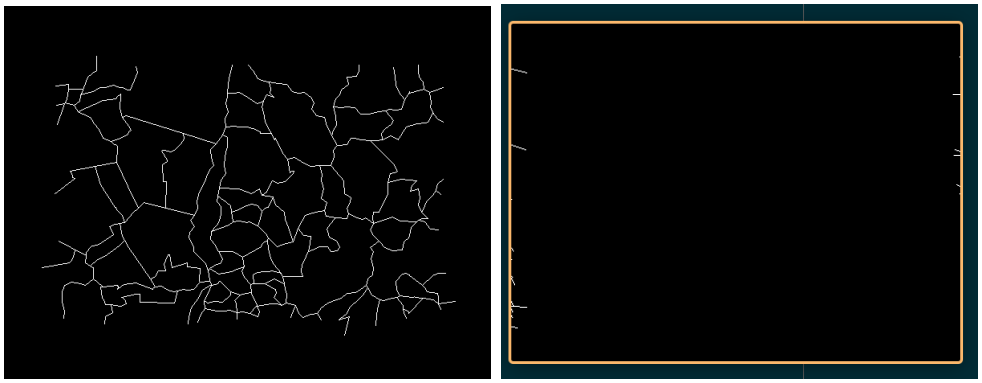


Figura 18. Janela de consulta do mapa do Brasil (à esquerda). Apenas a árvore de segmentos retornando os segmentos que cruzam as extremidades verticais da janela (à direita).

3. Resultados

Neste capitulo falaremos das nossas implementações e de nossos resultados com algumas delas. As estruturas[1] foram implementadas na linguagem Python, e para validação visual preferimos imagens SVG por serem fáceis de interpretar e de gerar imagens teste. Além da implementação das estruturas desenvolvemos dois casos de testes para validar as estruturas. O primeiro é uma aplicação onde há um mapa com movimento livre com inúmeros pontos. A ideia deste programa era validar tanto a aplicação para jogos 2D quanto 3D. Para um jogo 2D, poderíamos substituir cada ponto por texturas do jogo, e teríamos um mapa virtualmente infinito em dimensões. E portanto, buscamos neste capitulo validar que podemos consultar no plano grandes ordens de grandeza de pontos sem grandes impactos na performance. Validamos esta ideia mostrando os tempos de consulta para grandes valores de pontos. O segundo programa é um mapa do Brasil com grande resolução de segmentos e movimento de câmera livre por este mapa. Validamos a aplicação mostrando que seria inviável ter uma aplicação de tempo real sem as estruturas utilizadas. Construímos cada uma das estruturas de dados apresentadas no texto e cada uma delas tem métodos auxiliares para construir figura SVG com pontos aleatórios com janelas aleatórias, e a construção da árvore em cima do arranjo de pontos desta imagem e a saída do programa como outra figura SVG com os pontos dentro da janela indicados com a cor verde.Todas as estruturas foram construídas visando apenas a consulta em janela. Portanto como demonstrado no texto nos atentamos apenas aos métodos de construção e consulta. As estruturas de consulta para segmentos por sua vez foram construídas, como visto no trabalho até aqui, para consultas das bordas e portanto trabalham em conjunto com as estruturas de consultas de pontos. Para interpretarmos as imagens utilizamos a biblioteca xml[2] e interpretamos as figuras

[1] Afim de a aplicação ser agnóstica de sistema, utilizamos o programa pipenv que permite criar ambientes Python com as dependencias necessárias. Instruções de uso estão disponíveis no arquivo README do projeto. https://github.com/lrdass/theia

[2] Biblioteca built-in Python para lídar com arquivos XML https://docs.python.org/3/library/xml.etree.elementtree.html

SVG como XML.Usamos a mesma biblioteca tanto para a leitura quanto escrita das figuras após a consulta. Nos resultados obtidos utilizamos um computador com as configurações: Processador Intel® Core™ i7-7500U[3] com 8GB de RAM DDR4 2666MHz.

4. Aplicação árvore de intervalos

Em uma aplicação tridimensional poderíamos construir cenas arbitrariamente grandes de tal forma que organizaríamos os objetos da cena $3D$ com uma árvore de alcance de 3 dimensões. Consultaríamos nesta árvore, portanto, o cubo representado pela câmera como na Figura 19. Reportando somente quais estruturas estão dentro da janela e então enviaríamos para o pipeline gráfico para desenharmos na tela. Podemos pensar em uma limitação para um jogo que precisa manter todos os objetos geométricos no pipeline. O número de objetos geométricos seria limitado pela memória do pipeline. A partir destas constatações podemos recriar este comportamento construindo uma árvore de alcance tridimensional com todos os seus objetos, e carregamos para memória da placa de video apenas o que é retornado da consulta. Necessitando de apenas uma tela de carregamento para construir a árvore.

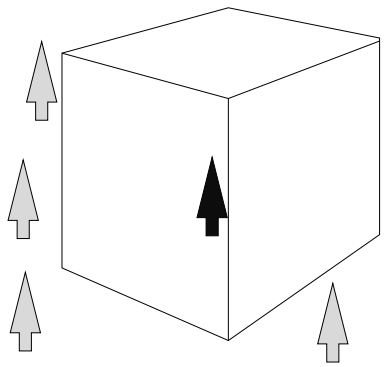


Figura 19. Exemplo de cena tridimensional com uma consulta retornando apenas os objetos dentro da janela. Considere as setas como figuras tridimensionais no espaço. E o cubo como a janela de consulta.

Para justificarmos esta aplicação, construímos uma versão simplificada do problema em duas dimensões visto na Figura 20. Criamos aleatoriamente pontos no plano e construímos uma árvore de alcance bidimensional. Em cada laço de execução do programa, consultamos a árvore com a janela, e desenhamos apenas os pontos dentro da janela. A solução trivial deste problema sem as árvores de alcance é consultar cada ponto e então desenha-lo e deixar o algoritmo de recorte [Hughes et al. 2014] desenhar na tela. Porem, ainda iteraria sobre estes para poder constatar que não estão na janela. Enquanto utilizando a árvore, temos uma maneira eficiente de consultar e desenhar apenas os pontos dentro da janela.

Como jogos são aplicações de tempo real, temos que pensar em restrições de tempo. Jogos modernos tem objetivos de entregar entre 30 e 60 quadros por segundo. Considerando o pior caso temos $\frac{1}{30} \approx 0.0334$ segundos para computações entre cada quadro.

[3]Referencia completa do CPU utilizado https://ark.intel.com/content/www/br/pt/ark/products/95451/intel-core-i7-7500u-processor-4m-cache-up-to-3-50-ghz.html

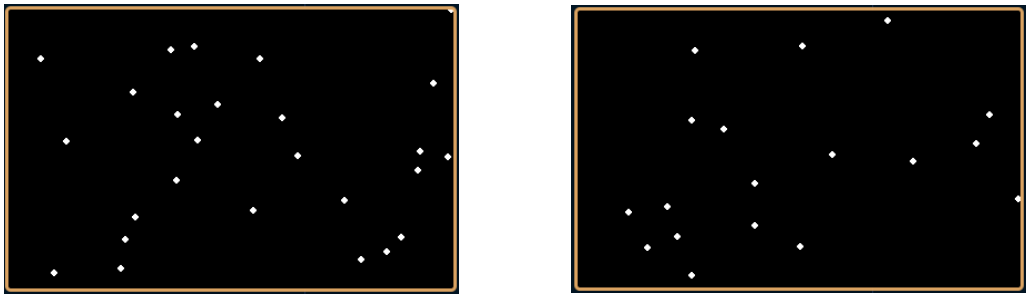


Figura 20. Aplicação construída com pontos no plano e consultas em tempo real

Pontos	Média do Tempo (s)	Desvio Padrão (s)
1000	0.00012955069541931	$1.3212002262037 \times 10^{-5}$
10000	0.00013832251230876	$2.0937153306017 \times 10^{-5}$
100000	0.00015597189626386	$2.8073955196226 \times 10^{-5}$

Tabela 1. Tabela comparativa do número de pontos e o tempo para reportar os pontos em uma janela proporcional ao tamanho do conjunto de pontos testado

Incremento médio tempo consulta(s) para cada 10^n pontos	Desvio Padrão (s)
$0,00001321$	$6.276986896593 \times 10^{-6}$

Tabela 2. Há um incremento médio de 13,21 microssegundos para cada 10^n pontos na consulta

Com base na Tabela 1 temos bastante confiança de que a estrutura de dados está dentro do tempo limite de computação para cada quadro desenhado, até mesmo aumentando o número de pontos. Mostrando que o crescimento com um fator de 10^n o tempo da consulta ainda permanece na casa dos 0.1 milissegundos.

5. Aplicação árvore de segmentos

Em aplicações de tempo real pode ser que exista apenas um objeto com grande complexidade. Programas que permitem ilustração em tempo real, por exemplo, estão mais interessados em conhecer se determinado segmento do objeto sendo desenhado está dentro da janela. Ou mapeamento tridimensional de um sistema cardiovascular em tempo real em que temos uma malha de um único objeto com grande complexidade e, para visualizá-lo, temos que carregar apenas um recorte deste objeto complexo. Em OpenGL [Vries 2015] temos um arranjo com as posições dos vértices chamado $VertexArray$ e um segundo arranjo que contém uma ordem de cada vértice para formar os triângulos chamado $ElementArray$. Podemos portanto interpretar estes dois arranjos e construir os segmentos pois sabemos que cada aresta de um triângulo é um segmento. E assim utilizar estes para construir e consultar com a árvore de segmentos. Para simplificarmos este problema tridimensional, desenvolvemos uma aplicação mostrado na Figura 18 (à direita) que navega pelo mapa do Brasil com alta resolução de segmentos em tempo real.

Média do Tempo (s)	Desvio Padrão (s)
0.005857	0.003753

Tabela 3. Utilizando a estrutura de dados para consultar os segmentos

Media do Tempo (s)	Desvio Padrão (s)
0.148667	0.017494

Tabela 4. Sem a estrutura de dados consultando linearmente

A consulta linear está inviável para uma aplicação de tempo real demonstrado na Tabela 4. Imaginando a mesma restrição de 30 quadros por segundo tendo $\frac{1}{30} \approx 0.0334$ segundos para calcular um quadro seria inviável atingir o objetivo sem a árvore de segmentos. A consulta linear alcançaria no máximo $\frac{1}{0.1487} \approx 6.72$ quadros por segundo nos nossos experimentos.

Referências

Bentley, J. L. (1979). Decomposable Searching Problems. Carnegie-Mellon University, Pittsburgh.

Berg, M. d., Cheong, O., Kreveld, M. v., and Overmars, M. (2008a). Computational Geometry - Algorithms and Applications. Springer, Berlin.

Berg, M. d., Cheong, O., Kreveld, M. v., and Overmars, M. (2008b). Computational Geometry - Algorithms and Applications. Springer, Berlin.

Hughes, J. F., Van Dam, A., McGuire, M., Sklar, D. F., Feiner, S. K., and Akeley, K. (2014). Computer Graphics Principles and Practice. Pearson Education, New Jersey.

Vries, J. d. (2015). Learn OpenGL. Creative Commons, New York.