Relatório Terceiro Trabalho - Geometria Computacional

Eduardo Mathias - ems19, Gustavo Frehse - gsf20,
Thiago Senff - tcps22, André Thomal - agt20,
Isadora Botassari - ibs20, Pedro - pvss19, Nico - nigr21

June 2023

1 Introdução

O trabalho consiste na implementação do problema da seleção de segmentos em uma janela retangular. Este problema consiste em dado um conjunto S com n segmentos de reta no plano, responder w consultas onde é dada uma janela retangular e deve-se responder quais segmentos intersectam esta janela. O objetivo deste trabalho é fazer um pré-processamento dos segmentos em tempo $\mathcal{O}(n \log n)$ e armazenar em uma estrutura de dados que ocupa espaço $\mathcal{O}(n \log n)$. Após isso, dadas as consultas (janelas retangulares), realizar cada busca em tempo $\mathcal{O}(\log^2 n + k)$, onde k é o número de segmentos da resposta.

2 Implementação

Para resolver o problema, dividimos em dois casos: (1) encontrar os segmentos que tem alguma das pontas (inicial ou final) dentro da janela e (2) pontos que atravessam a janela. Para (1) utilizamos uma *Range Tree 2D*, visto que esta estrutura resolve o problema de encontrar quais pontos estão dentro de uma janela retangular com as complexidades desejadas. Adaptamos a estrutura para resolver (1), quebrando um segmento nos pontos que o identificam, e acoplamos a cada ponto o id do segmento que ele faz parte. Para (2) utilizamos uma *Segment Tree* para encontrar os segmentos que atravessam a janela, atravessar nesse caso, significa que um segmento pode cruzar qualquer um dos quatro lados do janela retangular. Com essa implementação, são realizadas cinco *queries* ao total: uma para a *Range Tree* e quatro para *Segment Tree*, sendo um query para cada aresta da janela.

2.1 Range Tree 2D

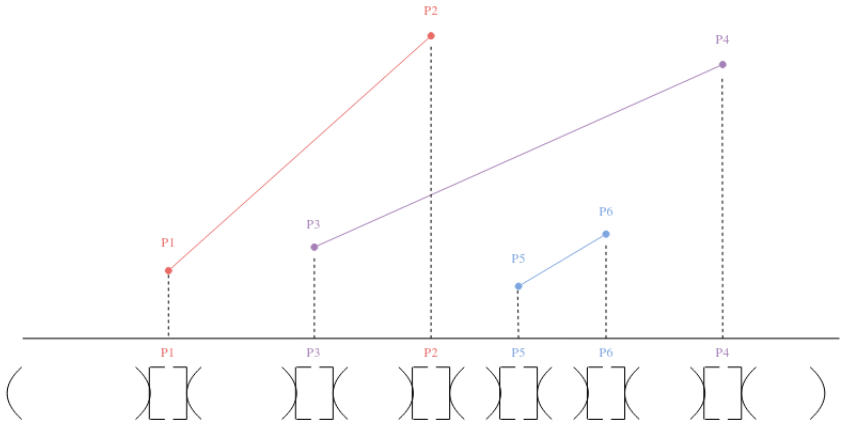
Esta árvore foi construída em C++ como uma struct RangeTree2D. Ela possui como campo um vetor de pontos do tipo struct item (vector< *item* > pon-

tos). Em adição, o struct item serve para representar cada ponto da árvore da RangeTree2D. Esse struct possui os campos RangeTree1D associada, o booleano eh_ponto para conferir se aquele ponto é uma folha e o inteiro xmid para identificar o x médio do intervalo. Para a construção da Range Tree 2D, temos a função recursiva build2dRangeTree que recebe os pontos e posição, a última com valor inicial 0. A função inicia com a criação de uma RangeTree 1D para as coordenadas y, que será associada ao ponto na atual posição. A criação dessa árvore demora O($nlog^2n$) uma vez que custa O($nlogn$) para montar a Range Tree 1d que é necessário para armazenar os pontos por y em cada nó da range tree 2d que guarda os pontos que estão abaixo dela o que gera um custo de memoria de O($nlogn$).

A query da Range Tree recebe as pontas de uma janela e retorna os pontos que se encontram nela. A função percorre a árvore e, caso o valor x de um ponto esteja entre os limite do retângulo, outra query na árvore 1d associada ao nó atual é feita, testando se o ponto está entre os valores y. No total, o custo acaba sendo de O($log^2n + k$).

2.2 Segment Tree

A SegmentTree foi implementada usando uma estrutura de árvore semelhante à estrutura de uma heap. A função de criação da árvore recebe como parâmetro um vetor de segmentos; em cada segmento, além dos pontos de início e fim, possui um valor de id associado a si. Após a criação da SegmentTree, os nós folha são definidos levando em consideração os endpoints distintos dos segmentos. Cada nó folha representa ou um intervalo fechado, ou um intervalo aberto, seguindo a subdivisão da imagem (para cada endpoint, há um nó folha com um intervalo fechado, e entre endpoints, há um nó folha representando um intervalo aberto).



No final da função de criação, é inserido o segmento o mais alto possível na árvore, garantindo que cada segmento está em no máximo 2 nós em cada nível da árvore. Essa função tem complexidade O($nlog^2n$), pois depois de fazermos o

build da estrutura em O($nlogn$), passamos novamente pela estrutura ordenando os segmentos associados a um nó para depois possamos fazer busca binária, e a Segment Tree ocupa um espaço O($nlogn$).

Na query da Segment tree passamos pelo caminho dos nós que intersectam o segmento vertical dado, e para cada nó no caminho encontramos o primeiro segmento que o intersecta.

2.3 Conclusão

Com essa implementação, conseguimos fazer um pré-processamento em tempo O($nlog^2n$) e gasto de memória O($nlogn$). Assintoticamente, o fato de mais de uma query ser realizada nas árvores não influencia no custo de tempo. Além disso, o uso de mais de uma arvore não muda o custo de armazenamento, uma vez que estamos guardando o dobro de memória, o que, novamente, pode ser ignorado assintoticamente. As duas estruturas combinadas são capazes de responder satisfatoriamente o problema proposto no enunciado.

3 Exemplos para teste

Aqui temos uma seleção dos exemplos mais interessantes que foram utilizados para testes dos algoritmos e alguns comentários sobre eles.

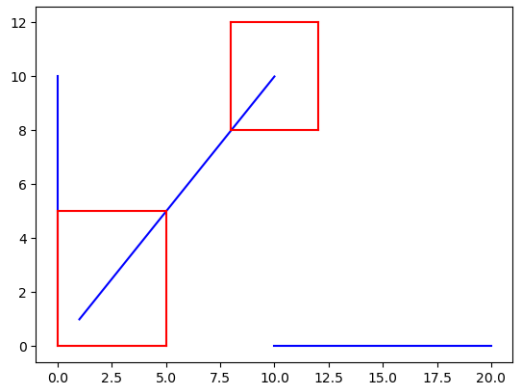
3.1 Exemplo 1

Este exemplo foi fornecido no enunciado do trabalho e possui como entrada 3 segmentos:

1. 0 0 1 10
2. 10 20 0 0
3. 10 1 10 1

As buscas são feitas nessas duas janelas:

1. 8 12 8 12
2. 5 0 5 0



O programa consegue nos responder corretamente o segmento da janela 1 como sendo 1, e os segmentos da janela 2 como sendo 3 e 1.

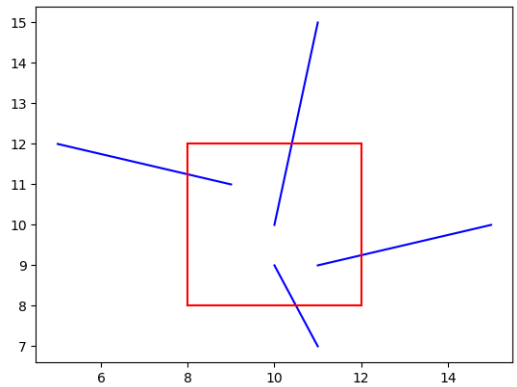
3.2 Exemplo 2

Esse exemplo foi feito para garantir que não importasse com qual aresta da janela de busca o segmento intersectasse ele ainda sim seria detectado, esse exemplo possui como entrada 4 segmentos:

1. 11 15 9 10
2. 10 11 10 15
3. 10 11 9 7
4. 9 5 11 12

A busca é realizada na seguinte janela:

1. 8 12 8 12



O programa consegue responder corretamente que todos os segmentos estão na janela.

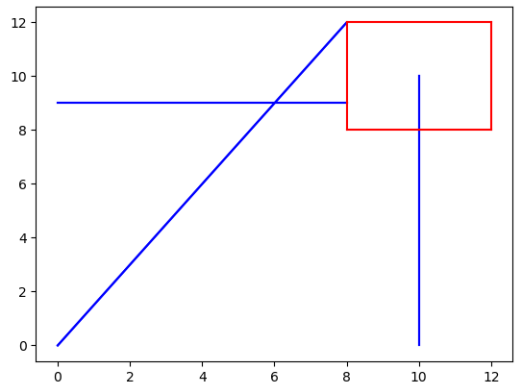
3.3 Exemplo 3

O exemplo 3 é um compilado de exemplos que foram modelados com intuito de testar se a ordem dos segmentos invalidaria a resposta. Há 3 segmentos de reta distintos, e cada um deles é dado duas vezes como entrada: primeiro, os pontos do segmento são dados na ordem convencional, e depois, numa ordem invertida: Temos então 6 segmentos na entrada:

1. 0 8 0 12
2. 8 0 12 0
3. 0 8 9 9
4. 8 0 9 9
5. 10 10 0 10
6. 10 10 10 0

E a janela na qual será feita a busca:

1. 8 12 8 12



O programa consegue resolver corretamente tanto os casos dos segmentos em ordem convencional quanto suas os casos de suas versões invertidas.

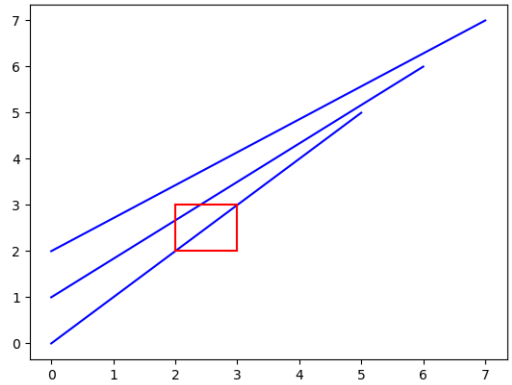
3.4 Exemplo 4

O exemplo 4 verifica casos em que os segmentos não têm nenhum ponto na janela e somente a atravessam. Temos 3 segmentos na entrada:

1. 0 5 0 5
2. 0 6 1 6
3. 0 7 2 7

E a janela na qual será feita a busca:

1. 2 3 2 3



O programa consegue resolver esses casos corretamente apontando os segmentos na janela como sendo o segmento 1 e 2.

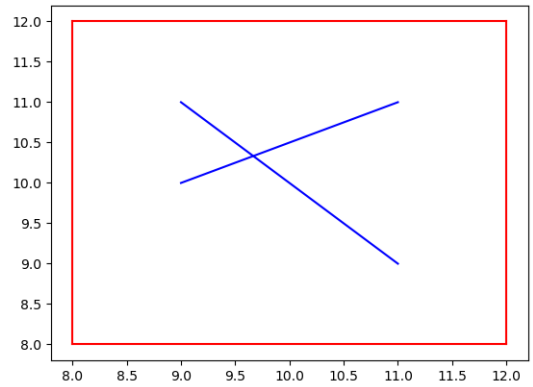
3.5 Exemplo 5

O exemplo 5 verifica casos em que os segmentos estão inteiramente dentro da janela. Temos 2 segmentos na entrada:

1. 9 11 11 9
2. 9 11 10 11

E a janela na qual será feita a busca:

1. 8 12 8 12



O programa consegue responder corretamente que todos os segmentos estão na janela.