Terceiro Trabalho de Geometria Computacional

Fabricio Schiavon Kolberg

Novembro de 2016

1 Problema a ser Resolvido

O problema proposto para o terceiro trabalho de geometria computacional foi o de buscar *segmentos de reta sobre o plano*: dado um conjunto de segmentos sobre o plano sem interiores intersectantes, encontrar todos os segmentos que intersectam um certo *retângulo de busca*. Para cada entrada passada ao programa, múltiplos retângulos de busca também são passados. As restrições impostas sobre nossa solução são que a construção das estruturas necessárias para a busca deve ter uma complexidade $O(n \log n)$, e a busca em si deve ter complexidade $O(\log^2 n + k)$, onde k é o tamanho da saída.

2 Detalhes da Implementação

2.1 Como Rodar

Para preparar o programa *windowing* para rodar, deve-se compilar o programa usando o compilador *gcc*. O Makefile incluso neste pacote já o faz por si próprio ao se digitar o comando **make**.

Para rodar o programa, simplesmente chame **./windowing < INPUT**, onde o **INPUT** é o arquivo contendo a entrada segundo descrito na especificação do trabalho.

2.2 Estruturas de Dados

Para poder cobrir todos os casos da busca de segmentos dentro de retângulos na complexidade especificada, utilizamos duas estruturas de dados:

1. Uma árvore *Range 2D* para encontrar segmentos para os quais um ou dois extremos estejam dentro dos retângulos [Berg et al., 2008].
2. Uma *árvore de segmentos* para encontrar segmentos que intersectam os retângulos sem que nenhum de seus extremos esteja contido no mesmo [Berg et al., 2008].

2.2.1 Construção da Range 2D

A árvore Range 2D que utilizamos para o primeiro caso da busca é composta de uma árvore binária onde os extremos de todos os segmentos são ordenados pela sua coordenada x, tal que a cada nodo dessa árvore é acoplada uma árvore binária do seu conjunto canônico, ordenada pela coordenada y. Para um nodo qualquer v em uma árvore, o seu *conjunto canônico* é o conjunto de todos os nodos tais que existe um caminho *descendente* de v até eles. Ou seja, o conjunto canônico é o conjunto de "descendentes" de v [Berg et al., 2008].

A árvore em questão, se construída de modo balanceado, tem tamanho $O(n \log n)$ [Berg et al., 2008]. Portanto, para que o tempo de sua construção não exceda essa complexidade, basta que o tempo de construção seja linear sobre o tamanho da estrutura a ser construída.

Para auxiliar na construção da árvore em tempo linear sobre seu tamanho final, utilizamos dois *vetores* (arrays) contendo os extremos dos segmentos: um ordenado na coordenada x, e outro ordenado

na coordenada y. Ambos os vetores são ordenados por *QuickSort*, retendo a complexidade $O(n \log n)$. Cada entrada dos vetores contém, além das coordenadas dos pontos, o índice do segmento que contém o ponto no vetor de segmentos e, no caso do vetor ordenado por y, o índice no vetor x que contém o mesmo ponto. Para construir, de modo *top-down*, a árvore "principal" da Range 2D, o algoritmo recursivo

Construir_Range2D(C_x, C_y**)** é utilizado, onde C_x e C_y representam os vetores contendo os elementos do conjunto canônico ordenados por x e y, respectivamente. Note que, para a raiz da árvore, C_x e C_y serão os vetores ordenados auxiliares mencionados anteriormente, contendo todos os pontos.

Algorithm 1: Construir_Range2D(C_x, C_y)

```
n ← novo nodo X
n.P ← mediana de C_x
n.T_y ← Construir_Range1D(C_y)
Se a árvore T_y só contem um nodo
    n.filho_esquerdo ← ∅
    n.filho_direito ← ∅
Senão
    Divida C_x em C_xl, C_xr, com C_xl contendo os pontos abaixo da mediana, e C_xr os acima.
    Divida C_y em C_yl, C_yr, com C_yl contendo os pontos de C_xl ordenados por y, e C_yr os de C_xr.
    Se |C_xl| ≠ ∅
        n.filho_esquerdo ←Construir_Range2D(C_xl, C_yl)
    Senão
        n.filho_esquerdo ← ∅
    Se |C_xr| ≠ ∅
        n.filho_direito ←Construir_Range2D(C_xr, C_yr)
    Senão
        n.filho_direito ← ∅
Devolva n
```

Uma árvore binária assim construída, onde cada nodo contém o ponto que é a mediana de seu conjunto canônico, tem altura trivialmente $O(\log n)$.

Para manter a complexidade $O(n \log n)$, o tempo gasto com cada nodo na construção da árvore principal, ignorando-se as chamadas recursivas, deve ser linear sobre o tamanho de seu conjunto canônico. É fácil verificar que a operação de divisão dos vetores mencionada no algoritmo pode ser feita em tempo linear tendo-se C_x e C_y ordenados. Portanto, basta que a construção da árvore 1D que é acoplada a cada nodo também tenha complexidade linear sobre seu conjunto canônico. A construção das árvores em questão é feita de modo *bottom up*, com o algoritmo Construir_Range1D(C_y), onde C_y é o conjunto canônico ordenado em y, e % representa a operação de *resto de divisão*.

Algorithm 2: Construir_Range1D(C_y)

```
n ← vetor vazio de nodos Y
atual ← vetor vazio de nodos Y
proximo ← vetor vazio de nodos Y
/* os pontos são pegos na ordem em que estão no vetor */
Para cada ponto P de C_y
    y ← novo nodo Y
    y.P ← P
    y.filho_esquerdo ← ∅
    y.filho_direito ← ∅
    adicione y ao vetor atual
Enquanto |atual| > 1
    Para cada i ∈ {1, ..., |atual|}
        Se i é ímpar e (i < |atual| ou i%4 ≠ 1)
            Se i%4 = 1
                atual[i + 1].filho_esquerdo ← atual[i]
            Senão
                atual[i − 1].filho_direito ← atual[i]
        Senão
            /* atual[i] não será marcado como filho nessa iteração */
            adicione atual[i] ao vetor próximo
    atual ← novo /* transferência de conteúdo dos vetores */
    novo ← ∅
Devolva atual[1]
```

Proposição 1. *Dado um vetor de pontos C_y ordenado pela coordenada y, com $|C_y| = n$, o algoritmo Construir_Range1D(C_y) constrói uma árvore binária ordenada pela coordenada y de altura no máximo* $\lceil \log n \rceil$.

Demonstração. Primeiro de tudo, é fácil verificar que o algoritmo termina, já que a cada iteração do "Enquanto", o tamanho do vetor *atual* diminui pelo menos em 1.

Também é fácil verificar que, ao início de cada iteração do "Enquanto", o vetor *atual* está ordenado de acordo com a coordenada y, o que implica que sempre que um nodo recebe um filho esquerdo, o filho tem a coordenada y menor que ele, e sempre que um nodo recebe um filho direito, o filho tem a coordenda y maior que ele, já que a única atribuição de filhos ocorre dentro do "Se" mais interno. Além disso, sabemos que a estrutura não fechará ciclo em momento algum, já que todo nodo só será atribuído como filho uma vez (pois não estará no vetor "atual" na próxima iteração após ser atribuído como filho pela primeira vez).

Note que os nodos que recebem filhos em cada iteração do "Enquanto" são precisamente os nodos que estão em índices pares do vetor *atual* que não são múltiplos de 4. Os nodos em questão necessariamente não foram marcados como filhos de ninguém ainda e, na próxima iteração, cairão em índices ímpares do vetor. Caso um nodo qualquer v que tenha recebido filhos caia em um índice ímpar que não é o último índice do vetor ou não é da forma $4k+1$, então ele sairá do vetor na próxima iteração. Caso contrário, e o nodo cair no final do vetor com um índice ímpar $4k+1$, isso significa que existirão $2k$ índices ímpares antes dele, significando que na próxima iteração, seu índice será $2k+1$. O mesmo argumento se repete caso k seja par, significando que o índice do nodo em questão permanecerá ímpar até deixar de ser da forma $4k+1$, em cujo caso ele não estará no vetor *atual* na iteração seguinte. Logo, todo nodo só será atribuído filhos no máximo uma vez para cada "lado".

Com o que provamos acima, é possível concluir que o grafo dos nodos (com arestas entre pais e filhos) é sempre acíclico. Além disso, também é fácil verificar indutivamente que o número de componentes conexas do mesmo é sempre igual ao número de elementos do vetor *atual*, significando que ao final da última iteração, o resultado será uma árvore binária (conexa, acíclica, com no máximo dois filhos por nodo). Mais que isso, toda componente conexa será tal que seus nodos formam uma sequência consecutiva em C_y, o que implica que a árvore em questão é tal que, para qualquer nodo, a sub-árvore

esquerda só contém nodos cujo ponto correspondente tem o índice y menor ou igual ao do ponto do nodo em questão, e a sub-árvore direita só contém nodos com o índice y maior ou igual. Ou seja, a árvore é completamente consistente e correta.

Além disso, é fácil verificar que o número de iterações que o "Enquanto" fará é no máximo $\lceil \log n \rceil$ (pois ao fim de cada iteração, o tamanho do vetor *atual* será no máximo o teto da metade do tamanho no início da iteração). Associe isso ao fato (também facilmente verificável) que a distância máxima entre qualquer nodo e seus descendentes é sempre menor ou igual ao número de iterações, e temos que a árvore terá altura no máximo $\lceil \log n \rceil$. □

2.2.2 Construção da Árvore de Segmentos

No nosso trabalho, implementamos duas árvores de segmentos: uma para busca de segmentos que intersectam arestas verticais, e outra para as arestas horizontais. A primeira é ordenada pela coordenada x, onde as folhas são os possíveis valores de x que um extremo de segmento podem assumir, além de folhas que representam posições entre dois valores consecutivos. Cada segmento é, então, armazenado em nodos que, minimalmente, sejam tais que as folhas que deles descendem são exatamente todos os valores de x (e posições intermediárias) pelos quais o segmento passa. Cada lista de segmentos armazenada em um nodo é então ordenada em um vetor pela coordenada y. A segunda árvore é análoga, mas invertendo a ordem das coordenadas. Maiores detalhes sobre árvores de segmentos podem ser encontrados na página 223 de [Berg et al., 2008].

Para auxiliar na construção das árvores de segmentos, mantemos um vetor ordenado horizontalmente, e outro verticalmente, para conseguir manter a construção da estrutura em complexidade $O(n \log n)$ [Berg et al., 2008]. A ordenação do vetor vertical é dada pelo seguinte processo:

1. Primeiro fazemos um *line sweep* dos extremos dos segmentos na coordenada x, inserindo os segmentos "ativos" a cada passo da varredura em uma árvore rubro-negra [Cormen, 2001] ordenada com a ordem na qual $s > t$ se e somente se existe uma linha vertical que intersecta s em um ponto de maior y do que o ponto no qual a mesma intersecta t. Com essa varredura, construímos um grafo direcionado ao adicionar arestas direcionadas entre cada nodo recém adicionado, seu nodo antecessor e seu nodo sucessor, ambos os quais podem ser encontrados em $O(\log n)$. Qualquer remoção de nodo também é feita em $O(\log n)$, e no máximo uma acontece a cada iteração da varredura.

2. Fazemos, então, uma ordenação topológica com o algoritmo de Kahn [Kahn, 1962] do grafo construído.

O algoritmo de *line sweep* terá complexidade $O(n \log n)$, pois a cada passo da varredura, teremos que inserir ou remover no máximo um vértice da árvore rubro-negra, com complexidade $O(\log n)$, e adicionaremos no máximo duas arestas ao grafo direcionado dos nodos. No final das contas, teremos que o número de arestas do grafo direcionado será $O(n)$ por si só também, significando que o algoritmo de Kahn, que tem complexidade $O(v + e)$ [Kahn, 1962], terá complexidade linear sobre o grafo todo. O caso da ordenação horizontal é análogo, apenas invertendo as coordenadas x e y.

Basta, então, provar o seguinte.

Proposição 2. *O vetor criado pelo método descrito acima é tal que, se existe uma linha vertical que intersecta s em um ponto acima do qual ela intersecta t, então o índice no qual s aparece no vetor é maior que o índice no qual t aparece.*

Demonstração. Primeiro de tudo, é fácil verificar que, se existe um caminho direcionado do nodo correspondente a t para o correspondente a s, então t aparecerá primeiro no vetor da ordenação topológica. Se s intersecta uma linha vertical em um ponto acima de onde t intersecta, isso implica que, em alguma iteração do algoritmo, os nodos correspondentes a s e t estão na árvore ao mesmo tempo. Observe o seguinte invariante: a qualquer modificação da árvore, se um nodo n é tal que seu segmento está imediatamente acima do segmento de um nodo m com relação à reta paralela ao eixo y correspondente ao ponto atual da varredura, então existe um caminho de m para n no grafo que é construído durante

a varredura. Isso se preserva em operações de adição, pois os únicos nodos que recebem um novo antecessor e/ou sucessor em uma adição de nodo são justamente o antecessor e sucessor do novo nodo, que recebem arestas de saída e entrada (respectivamente) para o mesmo. Isso também se preserva em operações de remoção, pois para qualquer nodo removido b, tal que seu nodo antecessor é a e seu sucessor é c, temos que, após a remoção, a passa a ser anterior a c; pelo invariante, existe um caminho de a para b, e outro de b para c, logo, um de a para c.

Desse modo, existe um caminho de qualquer nodo para seu sucessor dentre os nodos que se encontraram na árvore simultaneamente em algum momento. Se s se encontra acima de t em alguma reta paralela ao eixo y que intersecta ambos, então ou s é sucessor de t, ou existe uma sequência de sucessores começando em t e terminando em s, o que implica na existência de um caminho pelo invariante. □

Outra estrutura auxiliar utilizada são vetores para todos os possíveis valores de x e y, para construirmos inicialmente as árvores de modo bottom-up. Esses são, também, ordenados com QuickSort, utilizando-se a ordem aritmética dos valores.

Primeiro, as árvores de segmentos são construídas de modo bottom up (em $O(n)$) utilizando-se os vetores de possíveis valores de x e y. A construção da árvore de segmentos em si é simples: em cada iteração, juntamos nodos consecutivos com um nodo pai em comum de ambos (que é mandado para a próxima iteração), e caso o último nodo sobre, ele também é mandado para a próxima iteração. Assim sucessivamente até que só sobre um nodo. É fácil verificar que esse procedimento é linear e gera uma árvore de altura $O(\log n)$.

Depois disso, os segmentos, na ordem construída com a ordenação topológica supracitada, são adicionados à árvore com o procedimento *InsertSegmentTree* descrito em [Berg et al., 2008], com a inserção dentro do nodo em si sendo feita em tempo constante, pois os segmentos já são adicionados na ordem em que têm que ficar.

2.3 Algoritmos de Busca

2.3.1 Algoritmo de Busca Sobre Range 2D

O algoritmo de busca sobre a árvore Range 2D que utilizamos é o algoritmo *2DRangeQuery* da página 108 de [Berg et al., 2008], exceto com algumas adaptações. Como estamos reportando segmentos ao invés de pontos, o que fazemos é reportar o índice do segmento do qual o ponto é extremo. Desse modo, garantimos que todos os segmentos cujos pontos estão contidos no espaço de busca serão reportados exatamente uma vez. O algoritmo é $O(\log^2 n + k)$ [Berg et al., 2008].

2.3.2 Algoritmo de Busca Sobre Árvore de Segmentos

O algoritmo de busca sobre árvore de segmentos envolve buscar, para cada aresta do retângulo de busca, quais segmentos têm seu retângulo correspondente intersectando com a mesma. A busca é feita do seguinte modo: primeiro, encontra-se, através de uma busca binária, um valor ou par de valores de x (se a aresta for vertical) ou y (se a aresta for horizontal) tal que existem segmentos que possuem extremos nesses valores, e o valor de x (ou de y) da aresta em si esteja entre esses valores. Depois disso, é feita uma busca em $O(\log n)$ até a folha da árvore de segmentos correspondente a esse valor ou par de valores. Para cada nodo da árvore pelo qual a busca passa, é feita uma busca binária, dentro do seu vetor de segmentos, pelo vetor com o maior valor de y (se a aresta for vertical) ou x (se a aresta for horizontal) que é menor ou igual ao maior y (ou x) do retângulo, e outra pelo menor valor de y (ou de x) que é maior ou igual ao menor y (ou x) do retângulo, e então verifica-se, para cada segmento dentro do intervalo entre os dois índices encontrados, se ele deve ser reportado ou não (sendo $O(\log n + l)$, onde l é o número de segmentos que estarão dentre os dois índices encontrados.

É fácil verificar que a soma de todos os valores de l é $O(k)$, e o fato que o algoritmo de busca assim descrito encontra todas as interseções com as arestas buscadas é provado em [Berg et al., 2008].

2.3.3 Evitando Reportagens Duplicadas

Para evitar reportar o mesmo segmento mais de uma vez em cada retângulo, mantemos um vetor de inteiros chamado *last_report* tal que, para o segmento de índice i no vetor de segmentos, o valor de *last_report[i]* é igual ao valor do índice do último retângulo no qual ele foi reportado. Desse modo, caso uma busca no retângulo r encontre o segmento i, verifica-se se *last_report[i]* é igual a r. Se não for, reporta-se o segmento, e altera-se o valor de *last_report[i]* para r.

A alocação e inicialização do vetor são feitas em $O(n)$, durante a construção das estruturas auxiliares, e qualquer consulta ou alteração posterior tem custo $O(1)$.

3 Conclusão

Implementamos um programa que busca, dado um conjunto de segmentos sobre o plano, os segmentos que intersectam cada retângulo de um conjunto, com complexidades de leitura de dados e de busca limitadas, respectivamente, por $O(n \log n)$ e $O(\log^2 n + k)$. Utilizamos duas estruturas de dados diferentes para atingir esse objetivo, cobrindo tanto os casos em que um extremo de segmento está contido no conjunto quanto nos casos em que a interseção se dá no "meio" do segmento.

Possíveis maneira de melhorar a implementação aqui descrita envolvem encontrar uma estrutura que cubra ambos os casos com a complexidade pedida, assim diminuindo o tempo de computação significativamente, ou descobrir possíveis operações que podem ser simplificadas sob uma inspeção meticulosa.

Referências

Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008. ISBN 3540779736, 9783540779735.

Thomas Cormen. *Introduction to algorithms*. MIT Press, Cambridge, Mass, 2001. ISBN 0262032937.

A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, November 1962. ISSN 0001-0782. doi: 10.1145/368996.369025. URL `http://doi.acm.org/10.1145/368996.369025`.