Problemas Cinéticos em Geometria Computacional

Eduardo Garcia de Freitas

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO GRAU DE MESTRE
EM
CIÊNCIA DA COMPUTAÇÃO

Orientador: **Prof. Dr. Carlos Eduardo Ferreira**

Durante a elaboração deste trabalho o autor recebeu apoio financeiro da FAPESP processo: 98/16273-4

– São Paulo, Novembro de 2000 –

Resumo

Problemas em Geometria Computacional permitem a modelagem de situações do mundo físico no computador, de forma que esses problemas possam ser resolvidos eficientemente. Estudamos algoritmos e estruturas de dados para a solução de problemas de Geometria Computacional no âmbito cinético, ou seja, onde admitimos que os objetos geométricos (pontos, retas, polígonos, etc.) possuam movimento associado. Com isso, nos problemas cinéticos o valor dos atributos geométricos, que são propriedades geométricas de um conjunto, se altera com o passar do tempo. Nesta dissertação abordamos, no cenário cinético, o problema de se manter o máximo de um conjunto, o par de pontos mais próximo, o fecho convexo e o diagrama de Voronoi. Esses são exemplos de atributos geométricos.

Para que possamos manter atributos geométricos sobre um conjunto de objetos em movimento de forma eficiente, apresentamos um modelo proposto por Basch, Guibas e Hershberger, que introduz as estruturas de dados cinéticas. Elas são compostas de uma prova da corretude do atributo sendo "animada" através do tempo. Apesar do movimento contínuo de cada objeto, o atributo somente será alterado pela ocorrência de eventos em momentos discretos no tempo. O modelo também introduz medidas para a análise do desempenho de tais estruturas sob quatro diferentes pontos de vista. Uma estrutura de dados cinética, segundo o modelo, deve ser eficiente, local, compacta e ter resposta rápida.

Apresentamos também uma estratégia de implementação para as estruturas de dados cinéticas e exemplificamos sua utilização no problema do máximo.

Abstract

Computational Geometry problems allow modeling real world's situations in the computer as this problems can be solved efficiently. We studied algorithms and data structures to solution Computational Geometry problems at the kinetic scenario. In this kind of scenario, we allow that geometric objects (points, edges, polygons, etc.) have associated movement. Thus, in the kinetic problems, the value of geometric attributes change itself while the time advance. In this dissertation we approach, under the kinetic scenario, the problem of maintenance of the maximum element of a set, the closest pair of points, the convex hull and the Voronoi diagram. These are geometric attribute examples.

To maintain geometric attributes about a set of moving objects, we present a model proposed by Basch, Guibas and Hershberger, that introduce the kinetic data structures. They are composed by a proof of the correctness being "animated" over the time. Although the continuous movement of each object, the attribute only will be changed by the occurrence of events in discrete instants. The model also introduce techniques to measure performance of these kinetic data structures under four different points of view. A kinetic data structure, under the model, must be efficient, local, compact and responsive.

We also present a strategy for implement kinetic data structures. We exemplify the usability of this strategy at the problem of maintenance of the maximum element of a set of moving points.

Agradecimentos

Gostaria de prestar aqui meus agradecimentos às pessoas que foram importantes na contribuição para o resultado final desta dissertação de mestrado.

Primeiramente gostaria de agradecer à minha família, principalmente aos meus pais, Vera e Waldir, e meus tios, Humberto, Dácio e Valdir, que tanto colaboraram em todos os momentos desde o princípio da minha trajetória e que certamente continuarão colaborando ao longo da vida. Sem eles este trabalho não seria possível.

Devo prestar meus agradecimentos aos meus orientadores: o Carlinhos, orientador de fato, e o Coelho, que juntos fizeram um grande trabalho ao conduzir-me durante o programa de mestrado. Acho que sem eles, dificilmente esta dissertação estaria pronta em dois anos.

Paralelamente quero registrar meus agradecimentos à todos os amigos e colegas que contribuíram de alguma forma no enriquecimento deste trabalho. Partilularmente devo agradecer ao Aritanan (Pil), ao Cássio, ao Marcos e ao Ronney pelo companheirismo e pelas sugestões feitas nos últimos tempos.

Finalmente agradeço aos desbravadores da linha de pesquisa seguida nesta dissertação: Basch, Guibas entre outros. Afinal sem o trabalho anterior deles não seria possível escrever esta dissertação da maneira como foi escrita.

Sumário

1 Introdução **11**

1.1 O Modelo Cinético 13

1.2 Estrutura de Dados Cinética 17

1.2.1 Análise de Desempenho 18

1.2.2 Nomenclatura 19

2 Arranjos e Envelopes **20**

2.1 Envelopes 21

2.2 Seqüências de Davenport-Schinzel 22

2.3 Algoritmo para Calcular o Envelope Superior 25

2.4 Dualidade 27

3 O Problema do Máximo **29**

3.1 Lista Ordenada 31

3.2 Heap Cinético 32

3.3 Torneio Cinético 36

3.4 Considerações Finais 39

4 Par Mais Próximo **40**

4.1 Algoritmo Estático 42

4.2 Resolvendo o Problema Cinético 47

4.3 Problema Dinâmico-Cinético 52

4.4 Considerações Finais 53

5 Fecho Convexo **55**

5.1 Certificados para o Envelope Superior de Duas Cadeias 56

5.2 Tratamento de Eventos 61
5.3 Cinetizando o Envelope Superior Global 65
5.4 Manutenção do Fecho Convexo 67
5.5 Problemas Correlatos 68
5.5.1 Par de Pontos Mais Distante 68
5.5.2 Largura de um Conjunto de Pontos 74
5.6 Considerações Finais 77

6 Diagrama de Voronoi e Triangularização de Delaunay **79**
6.1 Problema Estático 79
6.2 Problema Cinético 82
6.3 Considerações Finais 85

7 Implementações **86**
7.1 Implementando Estruturas de Dados Cinéticas 87
7.1.1 As classes básicas 87
7.1.2 Estendendo as classes básicas 91
7.2 Implementações para o Problema do Máximo 92
7.2.1 Torneio cinético 94
7.2.2 Heap cinético 97
7.2.3 Lista cinética 98
7.3 Exemplo de Outra Implementação 99
7.4 Detalhes de Implementação 100
7.5 Testes 103
7.6 Considerações Finais 111

8 Considerações Finais **112**
8.1 Conclusões 112
8.2 Outros Problemas e Resultados 114

A Heap **119**
A.1 Heapsort 120
A.2 Fila de Prioridade 122

Referências Bibliográficas **125**

Lista de Figuras

1.1 Exemplos de certificados 16

2.1 Um arranjo de linhas retas no plano 20
2.2 Um conjunto de curvas no plano e o seu envelope superior 21
2.3 Seqüências de Davenport-Schinzel descrevendo um envelope 23
2.4 Possibilidades para conjuntos de 2 linhas retas 23
2.5 Envelope superior de um conjunto de segmentos de curvas 24
2.6 Transformando funções parcialmente definidas em funções totalmente definidas 25
2.7 Dualidade ponto-reta 28

3.1 Pontos movendo-se sobre o plano ty 30
3.2 Dependências entre certificados na lista ordenada 31
3.3 Um caso ruim para a lista ordenada 32
3.4 Heap cinético 33
3.5 Torneio cinético 37

4.1 Algoritmo de Shamos 41
4.2 Divisão em cones em torno de um ponto 42
4.3 Os Conjuntos $Cands(p)$ e $Maxima(p)$ 44
4.4 Figuras auxiliares para o Lema 4.2 e o Lema 4.3 44
4.5 Evento na Direção Horizontal 49
4.6 Evento na Direção 60° 52

5.1 Dualidade e Fecho Convexo 57
5.2 Vértices e retas inversas 57
5.3 Certificados para a manutenção do envelope superior 59
5.4 Alguns eventos 66

5.5 Correspondência entre fecho convexo e envelope superior 69
5.6 O par mais distante é formado por dois vértices do fecho convexo . . . 70
5.7 Cadeias superior e inferior do fecho convexo 71
5.8 Os pares opostos no plano dual 72
5.9 Contra-exemplo para a localidade da estrutura 74
5.10 As retas suporte paralelas entre dois vértices do fecho convexo 76
5.11 Os pares opostos reta-vértice no plano dual 77

6.1 Diagrama de Voronoi 80
6.2 Triangularização de Delaunay 80
6.3 Diagrama de Voronoi e Triangularização de Delaunay juntos 81
6.4 Violação de um certificado InCircle 83
6.5 Contra-exemplo para a localidade da triangularização de Delaunay . . . 85

7.1 Diagrama de implementação 92
7.2 Gráfico: Número de eventos × Tamanho da instância 106
7.3 Gráfico: Alterações na estrutura × Número de eventos 106

8.1 Exemplos de envoltórias em polígonos 115
8.2 Exemplo de consultas retangulares 116

A.1 Exemplo de heap binário e heap ternário 119

Lista de Algoritmos

2.1 Calcula Envelope Superior . . . 26
3.1 Encontra Máximo Divisão-e-Conquista . . . 36
4.1 Encontrando pares candidatos . . . 45
4.2 Algoritmo estático para o par mais próximo . . . 46
4.3 Processando eventos na direção horizontal . . . 50
4.4 Processando um evento no qual q entra em $Dom(p)$ . . . 51
4.5 Processando um evento no qual q sai de $Dom(p)$ . . . 51
5.1 Tratamento de Evento Horizontal . . . 61
5.2 Atualização dos Certificados de Intersecção . . . 61
5.3 Atualização da Lista Horizontal de Vértices . . . 62
5.4 Atualização da Lista Duplamente Ligada . . . 62
5.5 Atualização dos Certificados de Inclinação . . . 63
5.6 Tratamento de Evento de Intersecção . . . 63
5.7 Tratamento de Evento de Tangência . . . 64
5.8 Tratamento de Evento de Tangência/Inclinação . . . 64
5.9 Tratamento de Evento de Inclinação . . . 64
5.10 Algoritmo estático para o par mais distante . . . 73
5.11 Algoritmo estático para a largura de um conjunto de pontos . . . 78
A.1 Conserta Heap . . . 120
A.2 Constrói Heap . . . 121
A.3 Heapsort . . . 122
A.4 Extrai Máximo . . . 123
A.5 Insere Heap . . . 123

Capítulo 1

Introdução

Em Geometria Computacional estamos interessados em desenvolver algoritmos eficientes para resolvermos problemas geométricos. Muitos desses problemas têm suas origens em outras áreas como computação gráfica, telefonia celular, controle de tráfego aéreo [CKLM97], robótica, movimento de partículas, *computer-aided design* e processamento de imagens entre outras. No desenvolvimento de tais algoritmos são utilizados resultados de diversas áreas como geometria euclidiana, combinatória, teoria dos grafos, análise de algoritmos e estruturas de dados.

Problemas em Geometria Computacional têm como instância um conjunto de *objetos* que podem representar entidades do mundo físico. Um objeto geométrico é a entidade básica, variando conforme a natureza do problema abordado. Exemplos típicos de objetos geométricos são: pontos, retas, polígonos e outros. Em um modelo real, por exemplo, pontos podem representar agências do correio, pessoas ou aviões, retas podem representar rotas aéreas e polígonos podem representar robôs, edifícios, fronteiras ou objetos complexos.

A resposta para um dado problema de Geometria Computacional é chamada de *atributo geométrico* do conjunto de objetos analisado. Exemplos de atributos geométricos são: o fecho convexo, o par de pontos mais próximo, ou ainda o diagrama de Voronoi de um conjunto de pontos. Um atributo geométrico é computacionalmente descrito por uma *descrição combinatória* ou representação combinatória que abrange as estruturas de dados necessárias para se armazenar um atributo geométrico. Uma descrição combinatória típica para um fecho convexo é um polígono representado por uma lista de pontos em que cada dois pontos vizinhos na lista formam uma aresta do fecho convexo.

Quando é dado um conjunto de objetos geométricos e deseja-se consultar um determinado atributo desse conjunto (como por exemplo o par de pontos com distância mínima em um conjunto de n pontos), dizemos que esta é a versão *estática* do problema.

O mesmo problema pode ser formulado sobre conjuntos mutáveis de objetos geométricos. Nesse caso podemos ter uma classe de problemas em que existem operações como inserções e remoções de objetos no conjunto e gostaríamos de continuar com a

solução correta sem precisar recalculá-la como um novo problema estático. Esta é a chamada versão *dinâmica* do problema, ou *on-line* no sentido de inserções e remoções de objetos ao longo do tempo.

Uma terceira formulação, que é o objeto do nosso estudo nesta dissertação, é a versão *cinética* do problema, onde os objetos geométricos podem estar em movimento contínuo. Neste caso, queremos encontrar algoritmos e estruturas de dados que mantenham correto ao longo do tempo, de forma eficiente, o atributo desejado sobre o conjunto de objetos geométricos. Este constitui um outro tipo de problema *on-line* no sentido que as posições dos objetos se alteram a todo instante.

Nesta dissertação de mestrado apresentamos um modelo para a resolução de problemas cinéticos. Estudamos versões cinéticas de alguns problemas em Geometria Computacional sobre objetos em movimento no plano que são abordados através de estruturas de dados cinéticas segundo o modelo apresentado. Nosso estudo foi principalmente de análise de artigos relacionados com o tema, porém também incluímos algumas experiências com implementações de algumas estruturas de dados cinéticas básicas apresentadas, elaborando uma estratégia de implementação que acreditamos aplicar-se bem a qualquer implementação de estruturas de dados cinéticas.

Nas próximas seções vamos apresentar o modelo estudado e os critérios que são propostos por esse modelo para análise de desempenho de estruturas de dados cinéticas. Vamos também definir a nomenclatura usual que é usada nos capítulos seguintes. Nesses capítulos apresentaremos diversas estruturas de dados cinéticas para diferentes problemas cinéticos em Geometria Computacional.

No Capítulo 2 concentramos nossa apresentação no estudo preliminar de arranjos de linhas no plano e envelopes, que são bastante relevantes para o restante da dissertação. Apresentamos as seqüências de Davenport-Schinzel, que são ferramentas importantes na análise de desempenho, um algoritmo para encontrar envelopes e introduzimos o conceito de dualidade entre ponto e reta.

O Capítulo 3 apresenta o problema da manutenção do máximo de um conjunto de pontos unidimensionais em movimento contínuo. Esse problema serve como exemplo de aplicação do modelo cinético apresentado, assim como introduz estruturas de dados cinéticas que são comumente utilizadas nos problemas mais complexos.

No Capítulo 4 destacamos o primeiro problema em duas dimensões: o problema da manutenção do par mais próximo de um conjunto de pontos em movimento contínuo no plano. Apresentamos um novo algoritmo para resolver o problema estático que foi desenvolvido especialmente para funcionar bem com a versão cinética do problema.

O Capítulo 5 expõe uma estrutura de dados cinética para resolver o problema do fecho convexo de um conjunto de pontos em movimento no plano. Nesse capítulo mostramos que o problema dos envelopes é o problema dual do problema do fecho convexo e como uma estrutura de dados cinética para o problema cinético dos envelopes é utilizada na manutenção do fecho convexo cinético. Ainda no Capítulo 5 exploramos alguns problemas que usam o problema do fecho convexo como sub-problema tais como

o problema do par de pontos mais distante e o problema de se determinar a largura de um conjunto de pontos.

No Capítulo 6 apresentamos uma estrutura de dados cinética para a manutenção da triangularização de Delaunay de um conjunto de pontos em movimento. Explorando a dualidade entre a triangularização de Delaunay e o diagrama de Voronoi podemos aplicar tal estrutura de dados cinética para manter também o diagrama de Voronoi.

O Capítulo 7 destaca uma estratégia de implementação utilizada para a construção de estruturas de dados cinéticas. Neste capítulo apresentamos essa estratégia desenvolvida pelo aluno e como ela foi utilizada para a implementação das estruturas de dados cinéticas apresentadas para o problema do máximo. Ainda neste capítulo apresentamos alguns resultados baseados em testes comparativos entre as implementações das diferentes estruturas de dados cinéticas para o problema do máximo a fim de comparar o desempenho de tais estruturas.

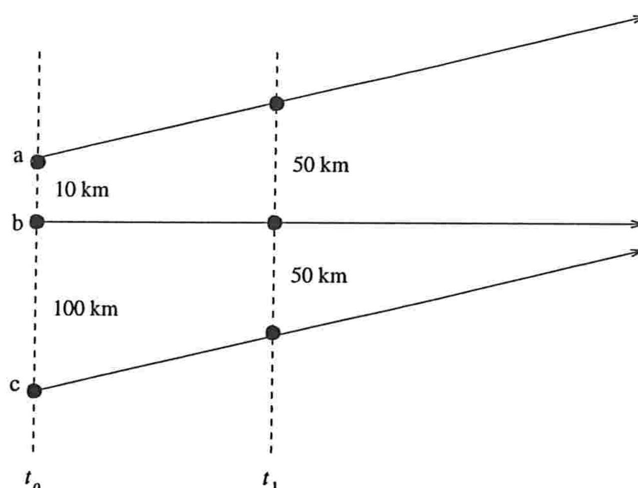
No Capítulo 8 apresentamos as conclusões de todo o trabalho e discutimos rapidamente alguns assuntos relacionados, pretendendo apresentar uma visão geral sobre o desenvolvimento atual da área de problemas cinéticos em Geometria Computacional.

1.1 O Modelo Cinético

Quando trabalhamos com objetos em movimento, em geral estamos interessados em um determinado atributo do conjunto de objetos considerado. Vamos considerar como um exemplo inicial uma aplicação de controle de tráfego aéreo onde desejamos saber qual o par de aviões que está mais próximo a cada instante visando a evitar acidentes. Não é difícil perceber que a distância mínima entre os pares de aviões pode variar continuamente. Porém, o par de aviões responsável por essa distância mínima não muda a todo instante, mas apenas em determinados instantes especiais do tempo.

Imagine um conjunto de três aviões a, b e c com trajetórias retilíneas conforme a Figura 1.1. Em um dado instante t_0 temos que a distância $d(a,b)$ entre a e b é 10 km e está aumentando devido ao movimento das duas aeronaves e que a distância $d(b,c)$ é 100km e está diminuindo. Em algum instante $t_1 > t_0$ as duas distâncias $d(a,b)$ e $d(b,c)$ terão o mesmo valor e, para $t > t_1$ o par de aviões mais próximo será sempre o par (b,c) e não mais o par (a,b) a menos que algum dos aviões altere a sua trajetória. Durante todo o intervalo de tempo $[t_0, t_1[$ o par de aviões mais próximo não se alterou, apesar da distância entre eles ser modificada a todo momento. E, no intervalo $]t_1, +\infty]$ o par de aviões mais próximo também não será mudado. O instante t_1 é um instante especial, onde acontece uma alteração do par de aviões de interesse.

No caso descrito o atributo geométrico que desejamos monitorar é o par de pontos (aviões) mais próximo. A descrição combinatória desse atributo é dada por um par de aviões que é o par mais próximo. Caso seja necessário saber o valor da distância atual entre este par de aviões, basta que se calcule a distância entre eles no instante desejado



a partir da descrição combinatória (o par) mantida. Tudo o que a aplicação precisaria fazer seria manter uma estrutura que monitorasse estes instantes especiais onde ocorrem mudanças significativas nas distâncias entre os pares de aviões, devolvendo a descrição do par mais próximo.

O mesmo pensamento pode ser estendido a outros problemas geométricos que envolvam objetos em movimento. No exemplo dos aviões, estamos interessados em manter o par de pontos mais próximo ao longo do tempo. Porém, poderíamos estar interessados em manter outros atributos sobre o conjunto de objetos tais como o fecho convexo, ou o par de objetos mais distante, o diagrama de Voronoi, entre outros. O que se percebe é que, apesar de as distâncias entre os objetos estarem em contínua modificação, a descrição combinatória dos objetos permanece a mesma durante um certo intervalo de tempo até que se alcance um instante especial que depende do movimento de um certo número de objetos.

Seguindo essa idéia de monitorar esses instantes especiais, um modelo para problemas que envolvem objetos em movimento contínuo foi exposto recentemente por Basch, Guibas e Hershberger [BGH98, BGH97]. No modelo proposto, assume-se que cada objeto tem um *plano de vôo* inicial que pode ser alterado a qualquer instante. O plano de vôo tipicamente é uma equação descrevendo o movimento do objeto ao longo do tempo. As alterações não são sabidas de antemão, sendo acrescentadas *on-line*. Assim, os dados de entrada para o tipo de problema considerado são:

- (objetos) n objetos $p_1, \ldots, p_n$ em movimento contínuo;
- (informação sobre o movimento) um plano de vôo $v_i(t)$ para cada objeto p_i ($i = 1, \ldots, n$), onde $v_i(t)$ é a posição do ponto p_i no instante t;
- (atualização) uma seqüência de alterações de planos de vôo que são fornecidas *on-line*.

Note que não são incluídas consultas na entrada já que a descrição combinatória do atributo geométrico de interesse deve estar prontamente disponível a todo instante.

A idéia chave por trás do modelo é que apesar de os objetos estarem em movimentação contínua e, conseqüentemente, o atributo geométrico sendo considerado também estar sendo alterado continuamente, a descrição combinatória deste atributo muda somente em certos momentos críticos discretos que são instantes especiais. A técnica utilizada por Basch, Guibas e Hershberger [BGH98, BGH97] para se manter a descrição combinatória de um atributo geométrico em movimento contínuo se concentra exatamente nestes momentos críticos. Os três autores propuseram um modelo de estrutura de dados, o qual denominaram *Estrutura de Dados Cinética* (*Kinetic Data Structure*). Além da descrição combinatória do atributo desejado, tal estrutura mantém um conjunto de *certificados* que constituem uma prova da corretude da descrição combinatória que está sendo mantida. Estes certificados, que são predicados geométricos com um número fixo de parâmetros, são calculados a partir dos planos de vôo dos pontos e possuem determinados *prazos de validade* que terminam nos instantes onde estes certificados deixam de ser capazes de atestar a corretude da descrição combinatória atual. Esses instantes em que um certificado perde a sua validade são os instantes especiais onde em geral ocorrem alterações na estrutura, constituindo um *evento*.

Na realidade, as estruturas de dados cinéticas podem ser encaradas como extensões de um tipo de algoritmo muito empregado na resolução de problemas estáticos em Geometria Computacional: os algoritmos de varredura. Algoritmos desse tipo geralmente fazem uma pré-ordenação dos objetos envolvidos em relação a alguma orientação e resolvem o problema incrementamente varrendo o plano (ou o espaço) com uma linha de varredura (plano de varredura) que se movimenta em sentido ortogonal ao da pré-ordenação. A pré-ordenação é necessária para que possamos descobrir facilmente quais são os pontos de parada da varredura e esses pontos ficam armazenados em uma fila auxiliar. Exatamente nos pontos de parada, e apenas quando a linha de varredura encontra um deles, é que o algoritmo faz alguma atualização na estrutura que está sendo mantida. Exemplos de algoritmos de varredura são encontrados para resolução de problemas tais como o problema de se encontrar as intersecções de um conjunto de segmentos [CLR90, BKOS98, Or98, BO79], o problema da triangularização de polígonos [BKOS98, Or98] ou o problema do diagrama de Voronoi [Sk98, BKOS98, Or98], entre outros.

No caso do modelo cinético a idéia é exatamente a mesma. A diferença é que a varredura é realizada no eixo do tempo. A fila de pontos de parada é substituída por uma fila que armazena os eventos. O algoritmo continua fazendo as alterações em uma estrutura que é mantida para a exibição do atributo geométrico desejado nos pontos de parada que agora são os instantes em que algum certificado na fila de eventos perde a sua validade.

Como exemplo, uma prova para a corretude da descrição combinatória do fecho convexo no plano pode ser dada através de certificados do tipo $\texttt{Left}(p_1, p_2, p_3)$ que, dados três pontos p_1, p_2, p_3, é verdadeiro se "o ponto p_3 está à esquerda do segmento orientado

de p_1 a p_2". Já uma prova da corretude da descrição combinatória da Triangularização de Delaunay pode ser obtida através de certificados do tipo In-Circle(p_1, p_2, p_3, p_4) que, dados quatro pontos p_1, p_2, p_3, p_4, é verdadeiro se "o ponto p_4 está dentro do círculo definido pelos pontos p_1, p_2, p_3". A Figura 1.1 ilustra esses dois tipos de certificados. Os certificados são tipicamente obtidos de algoritmos que resolvem o correspondente problema estático.

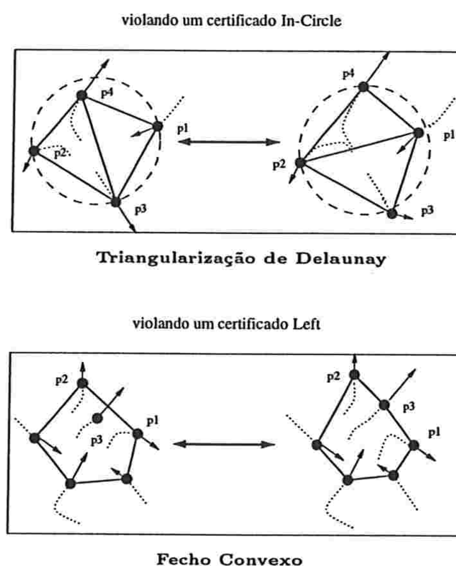


Figura 1.1: Exemplos de certificados para a Triangularização de Delaunay e o Fecho Convexo.

Voltando ao exemplo dos aviões mencionado anteriormente (Figura 1.1), a estrutura de dados cinética utilizada seria construída resolvendo o problema estático do par de pontos mais próximo para o conjunto de aviões nas suas posições iniciais ($t = t_0$), construindo também a estrutura de certificados durante a resolução. No exemplo haveria um certificado provando que o par de aviões (a, b) possui menor distância que qualquer outro par de aviões do conjunto até um determinado instante $t > t_0$ calculado com base nos planos de vôo dos aviões. Esse instante torna-se o prazo de validade do certificado. Quando o tempo decorrido atinge tal instante o certificado se torna inválido, deixando de provar que o par (a, b) é o mais próximo do conjunto de aviões. Nesse instante é necessário, portanto, removê-lo da estrutura e atualizá-la de forma a corrigir o erro causado pela falha no certificado. As atualizações na estrutura têm de ser feitas de modo a mantê-la correta e novos certificados serão criados provando a corretude da estrutura.

1.2 Estrutura de Dados Cinética

Basch, Guibas e Hershberger [BGH98, BGH97] chamaram a descrição combinatória de um certo atributo geométrico de *configuração* dos dados em movimento. Suas estruturas de dados foram denominadas *estruturas de dados cinéticas* para distingüi-las das clássicas estruturas estáticas e dinâmicas (no sentido de problemas estáticos e dinâmicos). Ao processo de transformar a resposta fornecida por um algoritmo que resolve um problema estático em uma estrutura de dados cinética que é válida para a versão cinética do mesmo problema eles deram o nome de *cinetização*.

Assume-se que cada objeto tem um *plano de vôo* que dá informação sobre o seu movimento atual. Esse plano de vôo é tipicamente uma equação que descreve o seu movimento. Nada impede que, em determinado instante, o plano de vôo de um objeto se altere.

Um *certificado* é uma prova local da corretude da configuração que está sendo mantida. Usando os planos de vôo dos objetos podemos determinar o seu *prazo de validade*. O instante do término do prazo de validade de um certificado constitui um *evento*. Quando um evento causa uma mudança na configuração ele é chamado de *evento externo*, caso contrário é um *evento interno*. O processamento de um evento mantém a consistência da estrutura.

Os certificados são armazenados em uma fila de prioridade chamada *fila de eventos* que os mantém em ordem de prazo de validade. Quando um certificado perde a sua validade, ele é removido da fila e os certificados que incluem os objetos envolvidos neste certificado devem ser recalculados. Talvez esses certificados também tenham que ser removidos da fila e alguns outros novos tenham que ser inseridos. Quando um objeto muda o seu plano de vôo, todos os certificados que dependem dele na fila de eventos devem ser recalculados e ter suas posições na fila atualizadas.

O conjunto de certificados forma uma estrutura que atesta a validade da solução. O instante em que um certificado perde a sua validade corresponde a um evento. Em um evento tipicamente alteramos outros certificados que são removidos da fila de eventos e reinseridos, mas os seus eventos correspondentes não são processados pois os seus prazos de validade não venceram.

Em certos casos, a modelagem do problema cinético tratado permite adicionarmos a uma estrutura de dados cinética a propriedade de resolver também o problema dinâmico, ou seja, além de resolvermos o problema cinético, podemos ter inclusões e remoções de objetos no conjunto de elementos considerado. Essa é uma situação que não faz parte do modelo cinético estudado, figurando nesta dissertação como uma extensão nos problemas em que é possível aplicá-la.

1.2.1 Análise de Desempenho

A análise de desempenho para estruturas de dados cinéticas segundo o modelo cinético proposto por Basch, Guibas e Hershberger [BGH98, BGH97] é feita sob quatro diferentes pontos de vista: *resposta rápida, eficiência, compacidade* e *localidade*. Vamos a seguir apresentar cada um desses conceitos que indicam se uma estrutura de dados cinética é boa ou não para resolver um problema cinético segundo o modelo.

A primeira propriedade tem a ver com o custo de processar um certificado que perdeu a sua validade (um evento). Dizemos que uma estrutura de dados cinética é de *resposta rápida* se o custo de processar um evento é polilogarítmico no número de objetos geométricos no pior caso. Em geral, o processamento de um evento envolve remoções e inserções de outros certificados na fila de eventos. Essa propriedade leva em conta o tempo necessário para a atualização da estrutura quando ocorre um evento.

Uma segunda medida de desempenho diz respeito ao número de eventos processados. O objetivo é desenvolver estruturas em que a razão entre o número total de eventos no pior caso e o número de eventos externos no pior caso seja polilogarítmica no número de objetos em movimento. Em suma, pretende-se minimizar a ocorrência de eventos internos. Uma estrutura de dados cinética que apresente esse comportamento é considerada *eficiente* segundo o modelo cinético.

Uma estrutura de dados cinética é dita *compacta* se o número máximo de certificados na fila de eventos é "aproximadamente" igual ao espaço necessário para armazenarmos a menor prova de corretude do atributo desejado. Uma estrutura de dados cinética que mantém um atributo que precise de $O(n)$ certificados para ser representada e que possua no máximo $O(n)$ certificados a qualquer instante é considerada compacta. Outra que possua $O(n \log^* n)$ certificados para a manutenção do mesmo atributo também, mas uma estrutura para o mesmo problema que tenha compacidade $O(n \log n)$ não é considerada compacta. A idéia é permitir que estruturas gastem espaço "pouco mais" que o estritamente necessário com a fila de eventos, evitando a ocorrência de "eventos supérfluos".

Finalmente, dizemos que uma estrutura de dados cinética é *local* se, a qualquer momento, o número máximo de eventos na fila que são afetados por um único objeto em movimento é polilogarítmico no número total de objetos. Essa propriedade é particularmente crítica se a estrutura precisa dar suporte a muitas atualizações nos planos de vôo.

É sempre desejável que uma estrutura de dados cinética apresente as quatro propriedades: tenha localidade, eficiência, compacidade e seja de resposta rápida. Porém, na prática, nem sempre ocorre de termos à disposição uma estrutura de dados cinética que satisfaça aos quatro critérios estabelecidos.

1.2.2 Nomenclatura

Antes de prosseguirmos, é tempo de definir alguma nomenclatura que será usada no decorrer desta dissertação:

Um (δ, n)-**cenário** corresponde a um conjunto S de n elementos em movimento onde o movimento de cada elemento $s \in S$ é regido por uma função polinomial de grau máximo δ.

Nos casos em que a modelagem do problema tratado permite adicionarmos a uma estrutura de dados cinética a propriedade de resolver também o problema dinâmico, faz-se necessário um modelo de movimento apropriado para que possamos analisar estruturas de dados que são cinéticas e dinâmicas. Então, vamos definir $\overline{\mathbb{R}}^2 = \mathbb{R}^2 \cup \{\omega\}$, onde $\{\omega\}$ é um símbolo especial que significa que o item está "oculto". Um (δ, n, m)-**cenário dinâmico** corresponde a um conjunto de m elementos cujas posições estão contidas em $\overline{\mathbb{R}}^2$, ou seja, cada elemento possui um plano de vôo sobre $\mathbb{R}^2$ com grau máximo δ ou está em ω. O número total de objetos que participam do cenário é m e o número máximo de objetos que não estão ocultos a qualquer instante é n. Nesse contexto, um (δ, n)-cenário é um (δ, n, n)-cenário. Uma remoção acontece quando um determinado elemento sai da dimensão dos reais e vai para $\{\omega\}$. Uma inserção segue o caminho inverso.

Em determinadas aplicações pode ser que uma alteração no plano de vôo de um ponto se faça de maneira descontínua, ou seja, o ponto some e aparece em outro local do espaço assumindo o novo plano de vôo. Um exemplo rápido seria de um ponto 1-dimensional cujo plano de vôo é dado pela equação em função do tempo $y = t + 10$. No instante $t = 5$ seu movimento passa a ser regido pela equação $y = t + 20$. Portanto, no instante $t = 5$ o mesmo ponto "evapora" na posição $y = 15$ e aparece na posição $y = 25$. Chamamos esse tipo de alteração de plano de vôo de *alteração descontínua*. O tratamento de alterações descontínuas tipicamente envolve uma remoção seguida de uma inserção do mesmo elemento na estrutura de dados. Uma estrutura de dados cinética que dê suporte a esse tipo de alteração é considerada também uma estrutura que admitem operações dinâmicas.

Poderíamos pensar em resolver o problema de alterações no plano de vôo de um objeto sempre utilizando inserções e remoções. Porém, tal artifício se torna ineficiente na maior parte dos casos em que o movimento do objeto é alterado de forma que não ocorra um alteração descontínua. Nesses casos o objeto seria removido da estrutura e posteriormente reinserido no mesmo local onde estava. Portanto, quando sabemos que a alteração do movimento não é descontínua, ou seja, o objeto continua na mesma posição no instante de alteração no seu movimento, não precisamos removê-lo da estrutura pois ele ocuparia o mesmo local quando fosse reinserido. Por esse motivo toda estrutura de dados cinética que siga o modelo cinético proposto é capaz de permitir alterações no plano de vôo dos objetos mas não necessariamente é capaz de permitir inserções e remoções de objetos na estrutura de forma eficiente.

Capítulo 2

Arranjos e Envelopes

Os envelopes (superior e inferior) fazem parte do estudo de *arranjos* (arrangements) de curvas e superfícies em Geometria Computacional. Um arranjo é uma coleção de curvas no espaço que induz uma partição do espaço em regiões. Por exemplo, um arranjo de linhas retas no plano induz uma partição do plano em faces poligonais, arestas e vértices conforme mostra a Figura 2.1. Os vértices são as intersecções entre as linhas. As arestas aparecem em dois tipos: limitadas e ilimitadas. Arestas limitadas são aquelas que ligam dois vértices, as ilimitadas englobam as arestas que ligam um vértice ao infinito e as que não possuem vértice (imagine uma linha que não cruza com nenhuma outra). As faces também podem ocorrer como faces limitadas e faces ilimitadas. As faces limitadas são as áreas poligonais cuja fronteira é totalmente formada por arestas limitadas. As faces que possuam alguma aresta ilimitada na sua fronteira são chamadas ilimitadas.

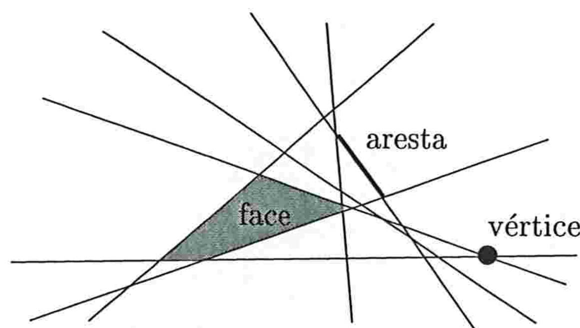


Figura 2.1: Um arranjo de 7 linhas retas no plano com 21 vértices, 49 arestas (14 ilimitadas) e 29 faces (14 ilimitadas)

Os arranjos são úteis na resolução de diversos problemas em Geometria Computacional e Computação Gráfica, tais como Visibilidade, Recorte em Cenas, Movimento de Robôs, Diagramas de Voronoi, entre outros [Ha97, Or98]. Em nosso estudo sobre objetos em movimento, os envelopes são usados como ferramenta para a resolução do Problema do Máximo (Capítulo 3), do Fecho Convexo (Capítulo 5), etc.

2.1 Envelopes

Considere um arranjo C de curvas no espaço, onde cada curva de C é o gráfico de uma função definida no espaço. O envelope superior $E_{sup}(C)$ é uma seqüência de curvas que limita C superiormente para alguma orientação do espaço. Para que isso faça sentido é necessário que as curvas sejam monótonas[1] na orientação do espaço considerada. Conjuntos de funções de $\mathbb{R}^n$ em $\mathbb{R}$ são exemplos típicos de conjuntos de curvas monótonas. Analogamente pode-se definir o envelope inferior $E_{inf}(C)$ que limita C inferiormente.

Particularizando o problema sobre um conjunto C de curvas c_i no plano (linhas - funções de $\mathbb{R}$ em $\mathbb{R}$), o envelope superior pode ser combinatorialmente descrito simplesmente por uma seqüência dos índices i das linhas que limitam C superiormente da esquerda para a direita conforme visto na Figura 2.2.

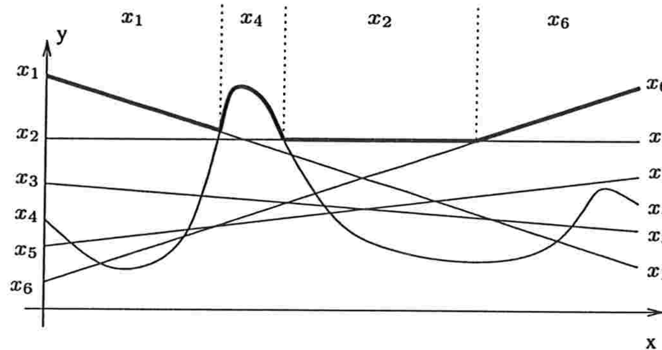


Figura 2.2: Um conjunto de curvas no plano e o seu envelope superior $\langle x_1, x_4, x_2, x_6 \rangle$ em destaque.

No caso de curvas no espaço 3D (superfícies - funções de $\mathbb{R}^2$ em $\mathbb{R}$) o envelope superior será constituído por um conjunto de superfícies interconectadas, formando uma espécie de colcha. A descrição combinatória do envelope superior pode ser feita, então, através de um diagrama planar composto das regiões interconectadas (faces, arestas e vértices) que limitam superiormente C no espaço.

O estudo de arranjos de curvas em duas dimensões generaliza a noção de envelope superior e inferior culminando na definição do k-nível de um arranjo. O k-nível é uma seqüência de curvas de C que particiona o plano em duas partes para alguma orientação do plano. Se o k-nível de um arranjo é definido sobre o eixo x do plano xy, interceptamos exatamente $k-1$ linhas acima do k-nível para quando o cortamos com uma linha vertical. O envelope superior é, portanto, o 1-nível de um arranjo de curvas no plano.

[1]Para cada ponto na orientação considerada, existe no máximo um ponto correspondente na curva. No plano xy uma curva é *x-monótona* se cada linha paralela ao eixo y intercepta a curva em no máximo um ponto, ou seja, se a curva é o gráfico de uma função.

2.2 Seqüências de Davenport-Schinzel

As seqüências de Davenport-Schinzel são estruturas combinatórias interessantes e poderosas que são aplicáveis na estimativa da complexidade combinatória de estruturas como envelopes ou arranjos de funções no plano. Por extensão, encontram aplicações em inúmeros problemas geométricos, incluindo vários problemas cinéticos que podem ser reduzidos à manutenção de envelopes, como o Fecho Convexo.

Definição 2.1 *Sejam n e s inteiros positivos, a seqüência $U = \langle u_1, \ldots, u_m \rangle$ é uma (n, s)-seqüência de Davenport-Schinzel se U obedece às seguintes propriedades:*

1. Um elemento u_i pertence a um alfabeto Σ onde $|\Sigma| = n$;

2. Dois elementos adjacentes são diferentes, ou seja, $u_i \neq u_{i+1}$ para todo $i = 1, \ldots, m-1$;

3. Qualquer subseqüência alternante de U constituída de somente dois elementos tem comprimento máximo $s+1$, isto é, não podem existir $s+2$ índices $i_1 < i_2 < \ldots < i_{s+2}$ tal que $u_{i_1} = u_{i_3} = u_{i_5} = \ldots = a \in \Sigma$ e $u_{i_2} = u_{i_4} = \ldots = b \in \Sigma$ para $a \neq b$.

Assim, uma $(n, 2)$-seqüência sobre um alfabeto $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ não pode conter qualquer subseqüência que possua a forma $\langle \sigma_i, \ldots, \sigma_{i+1}, \ldots, \sigma_i, \ldots, \sigma_{i+1} \rangle$ para $1 \leq i \leq n-1$. Vamos tomar como exemplo a seqüência $U = \langle a, b, c, a, d, b \rangle$ sobre o alfabeto $\Sigma = \{a, b, c, d\}$. U é uma $(4, 3)$-seqüência porque a subseqüência $\langle a, b, a, b \rangle$ de U é a maior subseqüência com elementos distintos em U e tem comprimento 4 $(= s+1)$, mas não é uma $(4, 2)$-seqüência porque nesse caso apenas são permitidas subseqüências com comprimento máximo 3.

Denotamos $\lambda_s(n)$ o comprimento máximo de uma (n, s)-seqüência de Davenport-Schinzel. Assim, se considerarmos o alfabeto $\Sigma = \{a, b\}$ e $s = 1$, as maiores seqüências permitidas são $\langle a, b \rangle$ e $\langle b, a \rangle$. Portanto, $\lambda_1(2) = 2$. Se considerarmos $s = 0$ sobre o mesmo alfabeto, as únicas seqüências permitidas são $\langle a \rangle$ e $\langle b \rangle$ que têm comprimento 1, assim, $\lambda_0(2) = 1$. Já para um alfabeto $\Sigma = \{a, b, c\}$ e $s = 1$, as maiores seqüências permitidas são $\langle a, b, c \rangle$, $\langle a, c, b \rangle$, $\langle b, a, c \rangle$, $\langle b, c, a \rangle$, $\langle c, a, b \rangle$ e $\langle c, b, a \rangle$, o que leva a $\lambda_1(3) = 3$.

As seqüências de Davenport-Schinzel são importantes na sua relação com a estrutura combinatória da descrição de um envelope superior ou inferior de um conjunto de funções no plano. Para qualquer conjunto de n funções contínuas $f_1, \ldots, f_n$ de $\mathbb{R}$ em $\mathbb{R}$ com a propriedade de que cada par entre as funções consideradas intercepta-se em, no máximo, s pontos, pode-se mostrar que a seqüência dos índices i das funções na ordem em que elas aparecem na descrição de um envelope da esquerda para a direita é uma (n, s)-seqüência de Davenport-Schinzel [SA95]. Também é possível mostrar que a cada

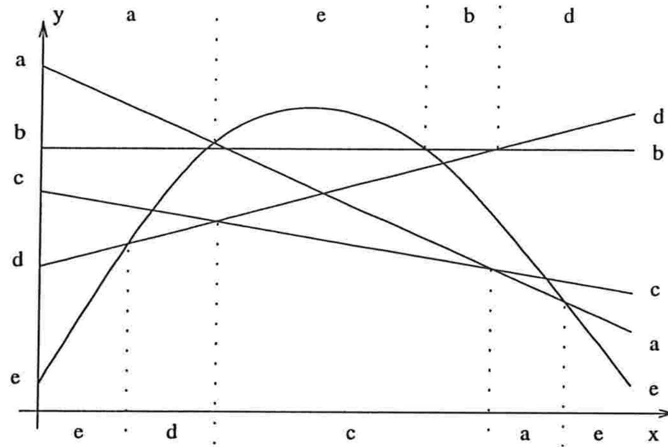


Figura 2.3: Um exemplo com 4 linhas retas e uma parábola no plano. O envelope superior é descrito por $\langle a, e, b, d\rangle$ e o inferior por $\langle e, d, c, a, e\rangle$. Ambas são $(5, 2)$-seqüências.

(n, s)-seqüência corresponde um conjunto de n funções que duas a duas interceptam-se em no máximo s pontos.

Vamos analisar um caso muito simples: suponha um conjunto S de duas retas no plano. Podemos criar uma associação entre cada reta a um elemento do alfabeto $\Sigma = \{a, b\}$. Temos então duas possibilidades: a é paralela a b ou a intercepta b em um único ponto conforme podemos conferir na Figura 2.4. No primeiro caso (retas paralelas) não há qualquer cruzamento ($s = 0$) entre a e b no intervalo $]-\infty, +\infty[$, implicando que o envelope superior será totalmente descrito por uma seqüência de tamanho 1 que é o tamanho máximo para $(2, 0)$-seqüências de Davenport-Schinzel.

Caso as retas não sejam paralelas, o envelope será dado por uma seqüência de tamanho 2 pois em $-\infty$ começa com uma das retas e termina em $+\infty$ com a outra reta porque elas se cruzam em algum ponto intermediário uma única vez ($s = 1$). Como visto anteriormente, as maiores $(2, 1)$-seqüências de Davenport-Schinzel são $\langle a, b\rangle$ e $\langle b, a\rangle$ que são exatamente as únicas seqüências possíveis que descrevem o envelope.

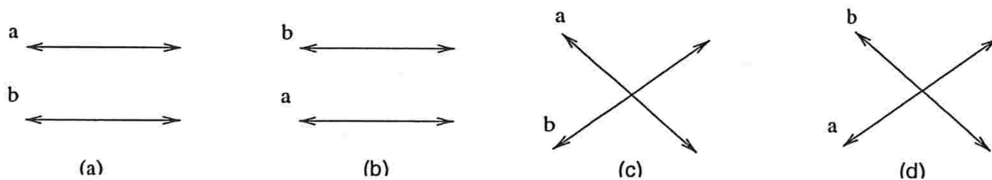


Figura 2.4: Todas as possibilidades para conjuntos de 2 linhas retas: em (a) o envelope superior é descrito pela seqüência $\langle a\rangle$; em (b) por $\langle b\rangle$; em (c) por $\langle a, b\rangle$ e em (d) é descrito pela seqüência $\langle b, a\rangle$.

Esse tipo de construção pode ser estendido para qualquer conjunto de n curvas, que são associadas com elementos de um alfabeto de tamanho n. O parâmetro s é fixado pelo número máximo de intersecções entre duas curvas do conjunto. Assim, um conjunto de n linhas que tomadas duas a duas interceptam-se no máximo s vezes tem o seu envelope superior descrito por uma (n, s)-seqüência de Davenport-Schinzel que possui comprimento máximo $\lambda_s(n)$. Um fato surpreendente no estudo das seqüências de Davenport-Schinzel é que, para um valor fixo de s, $\lambda_s(n)$ é aproximadamente linear em n, crescendo extremamente devagar. Especificamente temos [Ba99, SA95]:

$$\begin{aligned}
\lambda_0(n) &= 1 \\
\lambda_1(n) &= n \\
\lambda_2(n) &= 2n-1 \\
\lambda_3(n) &= \Theta(n\alpha(n)) \\
\lambda_4(n) &= \Theta(n \cdot 2^{\alpha(n)}) \\
\lambda_s(n) &\leq n \cdot 2^{(1+o(1))\alpha(n)^{\frac{s-2}{2}}} \quad \text{se } s \text{ é par} \\
\lambda_s(n) &\leq n \cdot 2^{(1+o(1))\log\alpha(n)\cdot\alpha(n)^{\frac{s-2}{2}}} \quad \text{se } s \text{ é ímpar}
\end{aligned}$$

onde $\alpha(n)$ é a inversa da função de Ackermann. A função de Ackermann $A(n)$ cresce incrívelmente rápido. Para se ter uma idéia, $A(4)$ é uma torre exponencial de 65636 dois. Assim, $\alpha(n)$ é uma função que cresce muito vagarosamente, sendo $\alpha(n) \leq 5$ para valores práticos de n. Para mais detalhes sobre a função de Ackermann veja [SA95, CLR90]. Outro resultado diz que $\lambda_s(n) = O(n \log^* n)$ [Go97].

Resultados parecidos são conhecidos para o caso de querermos saber o envelope de um conjunto S de n funções contínuas definidas parcialmente (segmentos) como mostrado na Figura 2.5. Nesse tipo de cenário a complexidade do tamanho do envelope é limitada por $\lambda_{s+2}(n)$, onde s é o número máximo de intersecções entre qualquer par de funções [SA95, Sh97].

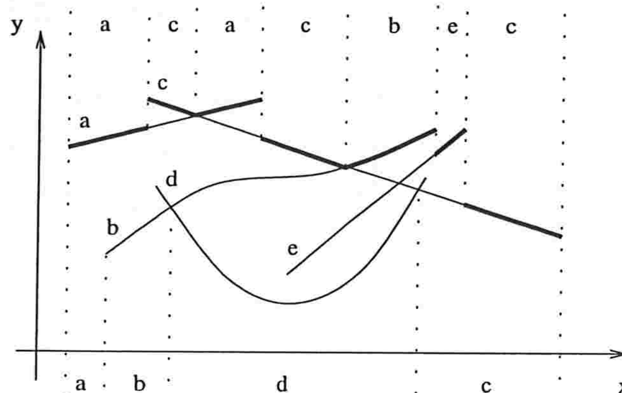


Figura 2.5: Nesse exemplo de conjunto de segmentos de curvas o envelope superior (em destaque) é dado por $\langle a, c, a, c, b, e, c\rangle$ e o envelope inferior por $\langle a, b, d, c\rangle$

Para verificar a validade da propriedade devemos primeiro fixar que tipo de envelope queremos obter. Caso estejamos interessados no envelope superior, basta que se defina um conjunto S^* contendo as funções de S totalmente definidas da seguinte forma: em cada extremo de cada função de S, conectamos uma reta de inclinação suficientemente grande (quase vertical) para baixo conforme mostra a Figura 2.6. A inclinação escolhida deve ser a mesma para o complemento de todas as funções. Se quisermos obter o envelope inferior, então basta que se conecte uma reta de inclinação suficientemente grande para cima em cada extremo.

Note que o envelope continua tendo a mesma descrição e que no máximo 2 novas intersecções entre cada par de funções de S são introduzidas na criação de S^*. Portanto o envelope nessa situação é dado por uma $(n, s+2)$ seqüência de Davenport-Schinzel. Assim, se estivermos trabalhando com uma coleção de n segmentos de reta ($s = 1$), o envelope superior (ou inferior) será completamente descrito por $O(n\alpha(n))$ subsegmentos.

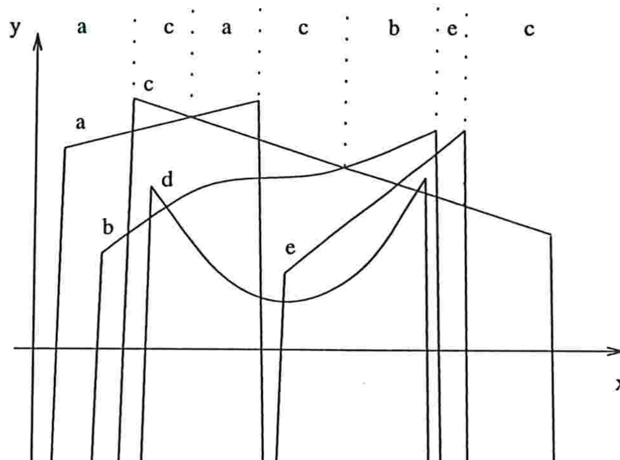


Figura 2.6: Transformando as funções parcialmente definidas da Figura 2.5 em funções totalmente definidas, tendo como objetivo o envelope superior.

2.3 Algoritmo para Calcular o Envelope Superior

Descreveremos agora um algoritmo simples, que usa a técnica de divisão-e-conquista, para calcular o envelope superior de um conjunto estático de n curvas x-monótonas totalmente definidas no plano. O algoritmo possui complexidade de tempo $O(\lambda_s(n) \log n)$, onde a função $\lambda_s(n)$ é o valor do maior comprimento possível para uma (n, s)-seqüência de Davenport-Schinzel. Considerando apenas funções lineares, o tempo de execução do algoritmo é $O(n \log n)$.

Este algoritmo foi proposto inicialmente proposto por Atallah [At85] e possui uma

versão que calcula o envelope superior de um conjunto de curvas parcialmente definidas (segmentos) em tempos $O(\lambda_{s+2}(n)\log n)$. Hershberger [He89] obteve um algoritmo que melhora o limitante para $O(\lambda_{s+1}(n)\log n)$ para o mesmo problema de curvas parcialmente definidas. O envelope inferior pode ser encontrado usando o mesmo algoritmo, mas invertendo as comparações "para cima" por comparações "para baixo".

Algoritmo 2.1 EnvelopeSuperior(S)

Entrada: Um conjunto S de curvas no plano.
Saída: Envelope superior $E_{sup}(S)$.

se $|S| = 1$ **então**
  retorna $\langle -\infty[s_1] \rangle$
$S_1 \leftarrow \{s_1, \cdots, s_{|S|/2}\}$
$E_{S_1} \leftarrow$ `EnvelopeSuperior`(S_1)
$E_{S_2} \leftarrow$ `EnvelopeSuperior`$(S - S_1)$
$E_S \leftarrow \phi$
$x \leftarrow -\infty$
$i, j \leftarrow 1$
enquanto $i \leq |E_{S_1}|$ e $j \leq |E_{S_2}|$ **faça**
  $p_1 \leftarrow E_{S_{1i}}$
  $s_1 \leftarrow$ `CurvaAssociada`(p_1, S_1)
  $p_2 \leftarrow E_{S_{2j}}$
  $s_2 \leftarrow$ `CurvaAssociada`(p_2, S_2)
  $q \leftarrow$ `ProximaInterseccao`(x, s_1, s_2)
  se q é mais próximo de x que p_1 e p_2 **então**
    $E_S \leftarrow E_S \cup q[$`CurvaAcima`$(q, s_1, s_2)]$
    $x \leftarrow q$
  senão se p_1 é mais próximo de x que q e p_2 **e** `CurvaAcima`$(p_1, s_1, s_2) = s_1$ **então**
    $E_S \leftarrow E_S \cup p_1[s_1]$
    $i \leftarrow i + 1$
    $x \leftarrow p_1$
  senão se p_2 é mais próximo de x que q e p_1 **e** `CurvaAcima`$(p_2, s_1, s_2) = s_2$ **então**
    $E_S \leftarrow E_S \cup p_2[s_2]$
    $j \leftarrow j + 1$
    $x \leftarrow p_2$
retorna E_S

No Algoritmo 2.1, usamos três algoritmos auxiliares: `CurvaAssociada` recebe um ponto p e o seu envelope E e retorna a curva associada a p em E; `CurvaAcima` recebe duas curvas a e b e um ponto p e responde qual das duas curvas está acima na reta vertical que passa por p; `ProximaInterseccao` recebe um ponto p e duas curvas a e b, retornando o ponto q à direita de p onde ocorre a primeira intersecção entre as curvas a e b. Todos os algoritmos devem executar com complexidade de tempo $O(1)$.

Seja S um conjunto de n curvas x-monótonas e seja E_S o seu envelope superior.

Dividimos S em dois subconjuntos S_1 e S_2 de comprimento máximo $\lceil n/2 \rceil$, calculamos o envelope superior de cada subconjunto recursivamente e então devemos compôr os dois sub-envelopes E_{S_1} e E_{S_2} para obter o envelope final E_S. Representamos E_S como uma lista duplamente ligada dos pontos de transição ordenados da esquerda para a direita associados à curva que aparece imediatamente à sua direita no envelope. Para isso devemos incluir um ponto virtual $x = -\infty$. O comprimento dessa lista é $O(\lambda_s(n))$. A base da recursão se dá quando o conjunto de curvas tem tamanho 1 e, nesse caso, o envelope superior é a própria curva que forma o conjunto. Digamos que tal curva seja $s \in S$, a lista que representa esse envelope é dada por $\langle -\infty[s] \rangle$ que diz que após $-\infty$ o envelope é dado pela curva s. Uma lista desse tipo é denominada *cadeia*.

Para combinarmos as duas cadeias representando sub-envelopes na fase de conquista do algoritmo, realizamos uma varredura da esquerda para a direita através de uma linha vertical. A cada ponto x a linha de varredura intercepta uma curva $s_1 \in E_{S_1}$ e uma curva $s_2 \in E_{S_2}$. Sejam p_1 e p_2 os próximos pontos de transição em E_{S_1} e E_{S_2} respectivamente e seja q o ponto de intersecção entre s_1 e s_2 mais próximo a direita de x. Se esse ponto não existe, fazemos $q = +\infty$. Agora devemos mover a linha de varredura até o ponto x' mais a esquerda entre q, p_1 e p_2 e analisar cada caso.

Se $x' = q$, inserimos q na lista dos pontos de transição de E_S associado à curva entre s_1 e s_2 que estiver acima imediatamente à direita de q. Se $x' = p_1$ e s_1 está acima de s_2, inserimos p_1 na lista associado à s_1, caso contrário p_1 não será um ponto de transição em E_S. O caso para $x' = p_2$ é simétrico.

Assim, a linha de varredura avança passo a passo por todos os intervalos determinados pelos pontos de transição de E_{S_1} e E_{S_2} e por todas as intersecções entre os dois sub-envelopes. Sendo $T(n)$ o tempo de execução do algoritmo sobre um conjunto S de n curvas, obtemos a seguinte recorrência:

$$T(n) = \begin{cases} O(1) & \text{se } n = 1 \\ 2T(n/2) + O(\lambda_s(n)) & \text{se } n > 1 \end{cases}$$

cuja solução é $O(\lambda_s(n) \log n)$. Considerando apenas funções lineares, temos que $\lambda_s(n) = \lambda_1(n) = O(n)$, o que faz com que o algoritmo assuma complexidade de tempo $O(n \log n)$ nessas condições.

2.4 Dualidade

Dualidade no estudo de arranjos serve para podermos transformar um conjunto de pontos de $\mathbb{R}^d$ (o espaço primal) em um conjunto de hiperplanos em $\mathbb{R}^d$ (o espaço dual) e vice-versa. Muitas vezes, um problema que lida com relações entre pontos pode ser melhor compreendido aplicando uma transformação dual-primal sobre os pontos.

Restringindo a dualidade a $\mathbb{R}^2$, podemos mapear um ponto em uma linha reta dual e vice-versa. Uma razão para esse tipo de conversão é que muitas vezes as relações

entre pontos são melhor compreendidas através do arranjo de linhas correspondente. Existem diversas transformações primal-dual possíveis [BKOS98, Or98, Ha97, RS94] dependendo da representação da reta e do contexto do problema, mas o conceito permanece em qualquer caso: como um ponto em $\mathbb{R}^2$ é determinado por duas coordenadas, ele pode ser associado com uma reta (cuja equação é composta de dois coeficientes). Sendo p^* a reta dual do ponto p e r^* o ponto dual da reta r, a transformação que adotamos é descrita como:

$$p = (a, b) \Rightarrow p^* : y = ax + b$$

$$r : y = ax + b \Rightarrow r^* = (a, b)$$

Um exemplo da transformação descrita acima pode ser visto na Figura 2.7. Note que pontos sobre o eixo vertical são mapeados para linhas horizontais e pontos sobre o eixo horizontal para linhas que passam pela origem no plano dual.

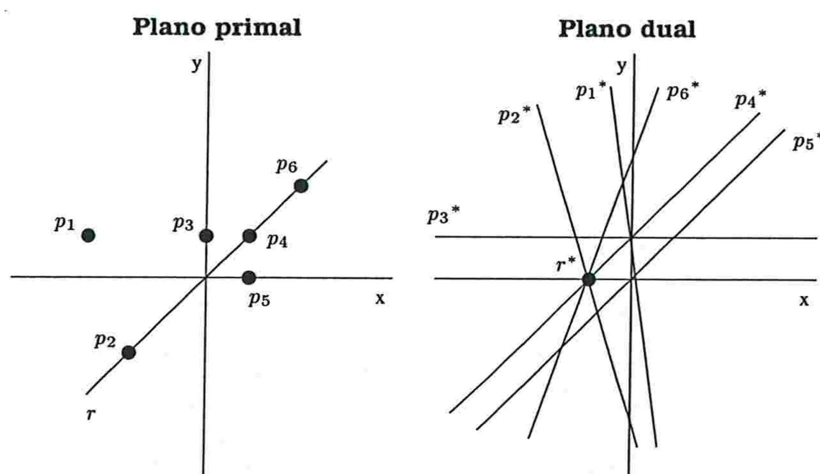


Figura 2.7: Um exemplo da dualidade entre pontos e retas em $\mathbb{R}^2$.

Sendo $\mathbb{T}$ a transformação de dualidade, podemos descrever algumas propriedades sobre $\mathbb{T}$. Por exemplo, $\mathbb{T}$ é a sua própria inversa, isto é, $\mathbb{T}(\mathbb{T}(x)) = x$ onde x é um ponto ou uma reta. Outra propriedade é de que $\mathbb{T}$ é um-pra-um entre todos os pontos do plano e todas as retas não-verticais (a dualidade sobre linhas verticais não é definida). Além disso, considerando um ponto p sobre uma reta r, $\mathbb{T}(r)$ será um ponto sobre a reta $\mathbb{T}(p)$. Se p estiver acima de r, então $\mathbb{T}(r)$ estará abaixo de $\mathbb{T}(p)$ e o caso em que p está abaixo de r é simétrico. Pontos colineares sobre uma reta r são transformados em retas que se cruzam em um mesmo ponto $\mathbb{T}(r)$.

Capítulo 3

O Problema do Máximo

Para tornar mais concretos os conceitos apresentados noCapítulo 1, vamos apresentar um problema cinético simples unidimensional, denominado o problema do máximo:

Problema 3.1 (Problema do Máximo) *Dado um conjunto S de n pontos movendo-se ao longo de um eixo, deseja-se saber a todo instante qual o ponto com maior coordenada no eixo considerado.*

Vamos fazer aqui algumas suposições sobre o problema apresentado:

1. quando dois pontos colidem eles atravessam um ao outro
2. a configuração inicial é arbitrária
3. não existem, em nenhum momento, dois pontos com o mesmo plano de vôo

Se fixarmos que o eixo sobre o qual os pontos estão se movendo é o eixo y, e tomarmos o tempo como um outro eixo t, podemos produzir um gráfico das posições em função do tempo no plano ty, obtendo um mapa das trajetórias dos pontos ao longo do tempo. Nesse contexto, o problema passa a ser computar o envelope superior (veja Capítulo 2) de um conjunto de curvas no plano a partir de um instante t_0 inicial. Na Figura 3.1 podemos ver um caso particular onde os pontos possuem movimentos lineares.

Considerando apenas o caso onde os pontos possuem movimentos lineares (um $(1, n)$-cenário), poderíamos resolver o problema usando um algoritmo divisão-e-conquista que obtém o envelope superior de retas no plano em tempo $O(n \log n)$, onde n é o número de retas [At85, He89] em S (veja Algoritmo 2.1). O algoritmo cria a noção de cadeias, que particionam o plano em duas partes e são formadas por "pedaços" de retas. A cadeia mais simples é formada por uma única reta totalmente definida no intervalo $]-\infty, +\infty[$. O envelope superior de um conjunto de retas é um outro exemplo

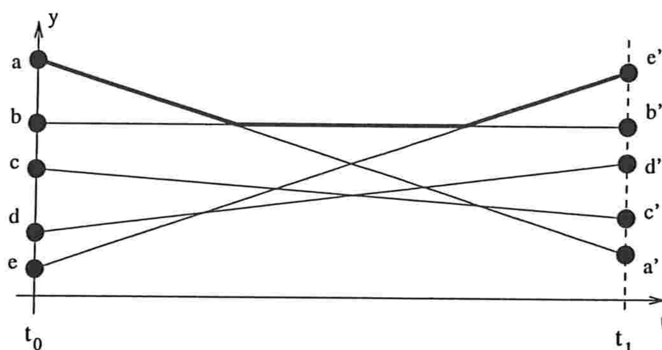


Figura 3.1: Pontos movendo-se linearmente sobre o plano ty e envelope superior em destaque.

de cadeia, sendo formado por pedaços de várias retas do conjunto e particiona o plano em duas partes.

O número de vezes que o elemento máximo de S pode mudar é exatamente igual ao maior tamanho da descrição do envelope superior das retas no plano ty. O estudo das seqüências de Davenport-Schinzel (veja Seção 2.2) mostra que em um (δ, n)-cenário o máximo de um conjunto de n curvas pode mudar no máximo $\lambda_\delta(n)$ vezes. Em um $(1, n)$-cenário, o número de vezes que o máximo do conjunto pode mudar é $\Theta(n)$.

Usando o algoritmo divisão-e-conquista citado acima que obtém o envelope superior de n retas em tempo $O(n \log n)$[1], obtemos um método que mantém o atributo desejado (o máximo) a cada instante com uma complexidade de tempo que é somente um fator logarítmico mais alto que o número máximo de mudanças no atributo. Porém, esse algoritmo não se encaixa nas exigências do modelo cinético apresentado porque assume que temos informação completa sobre o movimento de cada ponto durante todo o tempo desde o instante inicial, não permitindo que façamos quaisquer alterações durante a simulação. Então temos que utilizar um método que admita alterações on-line no movimento de cada ponto.

A seguir apresentaremos três métodos distintos descritos em [BGH98] para a solução do problema do máximo cinético. Ao invés de tentar criar um algoritmo especial para a versão cinética do problema, como feito acima com o algoritmo divisão-e-conquista que descobre o envelope superior, cada método tenta cinetizar um algoritmo que resolve o problema estático. O primeiro método a ser apresentado mantém a lista ordenada dos pontos, o segundo mantém um heap e um terceiro método utiliza uma outra idéia de divisão-e-conquista. Para cada solução será apresentada a sua análise quanto às quatro medidas de desempenho introduzidas pelo modelo cinético. Onde estivermos contando o número de eventos processados, o tempo gasto no processamento deve ser acrescido pelo tempo correspondente a inserções e remoções de um certificado na fila

[1] E que obtém o envelope superior de n curvas em tempo $O(\lambda_s(n) \log n)$, onde s é o número máximo de cruzamentos entre cada par de curvas do conjunto de n curvas considerado.

de eventos. Uma boa implementação para a fila de eventos não deve gastar tempo maior que $O(\log n)$ em ambas as operações.

3.1 Lista Ordenada

Vamos inicialmente pensar na versão estática do problema do máximo: encontrar o elemento que é o máximo de um conjunto de n elementos. Uma alternativa é ordenar os elementos e um dos extremos da lista ordenada certamente será o atributo desejado. Sabe-se que a construção da lista ordenada consome tempo $O(n \log n)$ usando um bom algoritmo de ordenação. Podemos tentar manter uma lista ordenada com todos os pontos ao longo do eixo y.

Na fase de cinetização, devemos definir quais serão os certificados que atestarão a validade da resposta mantida pela estrutura de dados cinética. A estratégia adotada é a de que para todo par de pontos consecutivos na lista teremos um certificado que diz que o ponto com índice menor na lista ordenada estará acima do outro (possui coordenada y maior) até um determinado instante. Esse instante é calculado através dos planos de vôo que descrevem o movimento de cada ponto. O elemento máximo terá um certificado associado ao segundo maior elemento, que por sua vez terá um outro certificado associado ao terceiro maior, e assim por diante até o menor elemento da lista.

Figura 3.2: O certificado *bc* afeta três certificados: *ab*, *bc* e *cd*.

Quando dois pontos se encontram um certificado deixa de ser válido pois a ordem relativa entre esses dois pontos que era atestada por tal certificado deixa de existir e ocorre um evento. Nesse instante é necessário processar esse evento, invertendo as posições dos pontos que se encontraram na lista ordenada e recalculando outros dois certificados que são afetados por essa inversão na lista ordenada. Assim, até três certificados são destruídos (retirados da fila de eventos) e três novos são criados (inseridos na fila). Os três certificados envolvidos são: o certificado que existe entre os dois pontos que se encontraram (e que é o responsável pelo evento), mais os certificados entre cada um dos dois pontos envolvidos e os seus respectivos vizinhos conforme visto na Figura 3.2. Pode acontecer de um dos pontos não ter um segundo vizinho (ser um extremo da lista ordenada) e, nesse caso, somente dois certificados serão atualizados.

A estrutura de dados cinética obtida é de resposta rápida pois o custo de se processar um evento é o custo de se alterar três certificados. O custo de alterar um certificado (destruí-lo e criar outro para o seu lugar) depende exclusivamente do custo de se calcular o seu novo prazo de validade. Tipicamente, esse custo é associado ao trabalho de se

calcular zeros de funções, fato que não depende do número de pontos no cenário e que pode ser considerado de custo constante. Então o custo de se processar um evento é $O(1)$.

A estrutura também é compacta, pois o número máximo de eventos na fila é igual ao número de adjacências na lista ordenada que é $n-1$. Portanto o tamanho da fila de eventos é $\Theta(n)$ e a estrutura é compacta.

Um determinado ponto aparece em, no máximo, dois certificados (um com cada vizinho na lista ordenada). Por isso a estrutura também é local, pois um único ponto aparece em $O(1)$ eventos na fila, o que quer dizer que alterações no movimento de algum ponto podem ser implementadas em tempo constante em relação ao número de objetos no cenário.

Porém, o total de eventos externos corresponde ao número de alterações no valor máximo que é $\lambda_\delta(n)$ em um (δ, n)-cenário. Se considerarmos um $(1, n)$-cenário, então o número de eventos externos é $\Theta(n)$ (cada ponto aparece como valor máximo uma vez). Já o número total de eventos é igual ao número de trocas entre elementos adjacentes na lista. No pior caso, todos os pontos cruzam com todos os pontos, totalizando $\Theta(n^2)$ eventos no total, conforme mostra a Figura 3.3. Assim, se dividirmos o número de eventos no pior caso pelo número de eventos externos no pior caso obtemos um fator de $O(n)$, o que impede que a estrutura possa ser considerada eficiente segundo o modelo cinético.

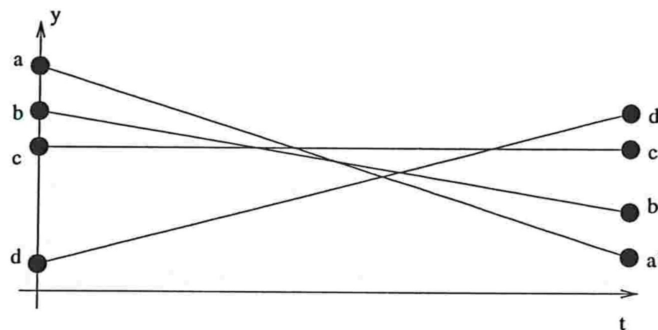


Figura 3.3: No pior caso temos todas as n linhas cruzando com todas as outras $n-1$ linhas, o que resulta em $O(n^2)$ cruzamentos, correspondendo a $O(n^2)$ eventos na lista ordenada.

3.2 Heap Cinético

Outra solução para o problema do máximo estático é construir um *heap* binário em cuja raiz encontramos o elemento máximo, exatamente como acontece em uma iteração do algoritmo de ordenação Heapsort (veja Apêndice A). A construção de um heap de

n elementos demora tempo $O(n)^2$. A idéia agora é manter a estrutura desse heap com o tempo, criando a estrutura de dados cinética denominada *heap cinético*.

A cinetização do heap é feita da seguinte forma: para cada ligação no heap temos um certificado que garante que o ponto filho está abaixo do ponto pai na ordem dos pontos até um determinado instante que é dado pelo cruzamento entre os planos de vôo dos dois pontos em questão.

Quando um certificado perde sua validade ocorre um evento cujo processamento forçará uma troca de posições entre pai e filho na ordem dos pontos. Conseqüentemente, nesse instante teremos uma inversão entre os elementos das duas células envolvidas do heap. No processamento de tal evento é necessário que se atualizem até cinco certificados: o certificado responsável pelo evento (entre os dois pontos que estão se invertendo), os certificados existentes entre o ponto pai e o seu outro filho e entre o ponto pai e o seu pai, além dos certificados entre o ponto filho e os seus dois filhos.

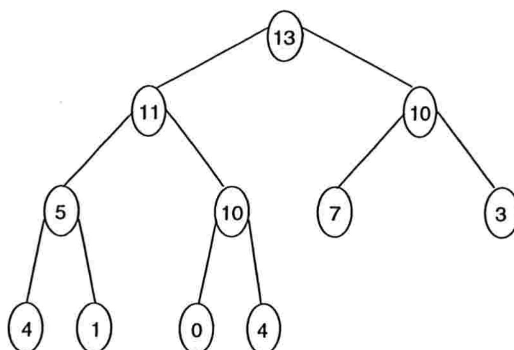


Figura 3.4: Heap que seleciona o máximo sobre um conjunto de 11 números. Cada aresta corresponde a uma comparação e é associada a um certificado.

Podemos notar na Figura 3.4 que dois vizinhos na ordem dos pontos podem não ter um certificado correspondente naquele instante simplesmente por se situarem em sub-árvores diferentes do heap. O único par de pontos que são vizinhos na ordem dos pontos e que deve ter necessariamente um certificado correspondente é constituído pelo máximo e o segundo maior elemento. Isso ocorre porque o maior elemento, deve estar na raiz e ser maior que os seus filhos. Como o segundo maior elemento do conjunto é maior que todos os outros exceto o máximo, ele só pode ser filho do nó que contém o máximo no heap, caso contrário a propriedade do heap estaria violada.

O heap cinético é uma estrutura de resposta rápida pois o custo de se processar um evento é igual ao custo de se alterar no máximo cinco certificados. Como o custo de alterarmos um certificado é constante, então o tempo gasto ao se processar um evento é $O(1)$.

[2]Para saber mais sobre a construção de um heap em tempo $O(n)$ veja Algoritmo A.2

O número máximo de eventos na fila é igual ao número de arestas no heap que é $n-1$. Portanto o tamanho da fila de eventos é $\Theta(n)$ e o heap cinético é considerado compacto.

Um determinado ponto aparece, no máximo, em três certificados (um com o seu pai e outro para cada um dos dois filhos). Portanto o heap cinético também é local, pois um único ponto aparece em $O(1)$ eventos na fila.

Para analisar a eficiência da estrutura precisamos contar o número total de eventos que possam ocorrer na estrutura. Vamos mostrar a seguir que, no pior caso, é necessário processar um total de $O(n \log^2 n)$ eventos em um $(1, n)$-cenário, contra $\Theta(n)$ eventos externos, resultando em uma razão de $O(\log^2 n)$ (que é polilogarítmica em n) entre o número total de eventos no pior caso e o número de eventos externos no pior caso. Por esse motivo o heap cinético é considerado também uma estrutura de dados cinética eficiente. Ainda não se conhece nenhum resultado teórico para movimentos não-lineares [Ba99].

Para chegarmos ao limite de $O(n \log^2 n)$ eventos no heap cinético, precisamos antes definir o *nível* de um ponto a no heap no instante t, que chamaremos de $\gamma_t(a)$, como a distância do nó contendo a até a folha descendente de a mais profunda no instante t. Assim, $1 \leq \gamma_t(a) \leq \lceil \log n \rceil$, para qualquer nó a do heap. Vamos definir por $\hat{\gamma}_t(a)$ o maior nível atingido pelo ponto a em qualquer instante entre t_0 e t.

Lema 3.2 *Em um $(1, n)$-cenário, se $[a > b]$ é um certificado que perde sua validade no instante t, então $\hat{\gamma}_t(a) \geq \hat{\gamma}_t(b) + 1$.*

Prova: Considere o instante x antes de t quando b esteve em seu nível máximo $\hat{\gamma}_t(b)$. Em x, b estava em algum nó v do heap e desde então b moveu-se somente dentro de uma sub-árvore T com raiz em v. Existem, então, dois casos para b:

1. Se b está no nó v no instante t, então a deve estar no nó pai de v nesse mesmo instante, e a afirmação vale.

2. Se b está em algum nó u em T, $u \neq v$, no instante t, a não poderia estar em T no instante x pois os movimentos são lineares. O caminho percorrido por a no heap deve entrar em T em algum momento, passando obrigatoriamente por v e pelo nó pai de v. Portanto, a afirmação também vale nesse caso. □

Teorema 3.3 *O número total de eventos no heap cinético sobre n pontos em um $(1, n)$-cenário é $O(n \log^2 n)$.*

Prova: Vamos fazer uma prova por função potencial. Para um ponto a no heap, definimos o seu potencial $\phi_t(a)$ e o potencial do heap inteiro Φ_t no instante t como

$$\phi_t(a) = \hat{\gamma}_t(a)(\gamma_t(a) - \hat{\gamma}_t(a))$$

$$\Phi_t = |\sum_a \phi_t(a)|$$

Note que o potencial de qualquer ponto é zero no instante inicial ($\gamma_{t_0}(a) = \hat{\gamma}_{t_0}(a)$) e é não-positivo para qualquer instante t ($\gamma_t(a) \leq \hat{\gamma}_t(a)$). O valor máximo do potencial do heap é $O(n \log^2 n)$ como se segue:

$$\Phi_t = |\sum_a \hat{\gamma}_t(a)(\gamma_t(a) - \hat{\gamma}_t(a))|$$

$$\Phi_t \leq |\lceil \log n \rceil (1 - \lceil \log n \rceil) \cdot 2^{\lceil \log n \rceil - 1} + \ldots + \lceil \log n \rceil (\lceil \log n \rceil - \lceil \log n \rceil) \cdot 2^0|$$

$$\Phi_t \leq |\sum_{k=0}^{\lceil \log n \rceil - 1} \lceil \log n \rceil \cdot (-k) \cdot 2^k| = |-\lceil \log n \rceil| \cdot \sum_{k=0}^{\lceil \log n \rceil - 1} k \cdot 2^k$$

$$\Phi_t \leq \lceil \log n \rceil \cdot \sum_{k=0}^{\lceil \log n \rceil - 1} k \cdot 2^k \leq \lceil \log n \rceil \cdot \int_0^{\lceil \log n \rceil} x \cdot 2^x \cdot dx$$

$$\Phi_t \leq \lceil \log n \rceil \cdot O(n \log n) = O(n \log^2 n)$$

Agora precisamos mostrar que cada evento processado contribui com um decréscimo de 1 no potencial total do heap. Considere uma troca no instante t entre pontos a e b, sendo a pai de b no heap. O potencial de qualquer outro ponto permanece o mesmo, $\hat{\gamma}_{t^+}(a) = \hat{\gamma}_{t^-}(a)$ e a mudança no potencial de a é $\phi_{t^+}(a) - \phi_{t^-}(a) = -\hat{\gamma}_{t^-}(a)$, onde t^- é o instante imediatamente anterior a t e t^+ é o instante imediatamente posterior a t.

Há dois casos para b, ou ele estava no seu maior nível em t^- ($\gamma_{t^-}(b) = \hat{\gamma}_{t^-}(b)$) ou não. No primeiro caso, $\hat{\gamma}_{t^-}(b) = \gamma_{t^-}(b)$ e $\hat{\gamma}_{t^+}(b) = \gamma_{t^+}(b)$, então $\phi_{t^-}(b) = \phi_{t^+}(b) = 0$. Assim o decréscimo no potencial total é $\hat{\gamma}_{t^-}(a)$, que é no mínimo 1 porque a não está no nível base do heap antes da troca.

No segundo caso, o potencial de b cresce $\hat{\gamma}_{t^-}(b)$, mudando o potencial total do heap de $\hat{\gamma}_{t^-}(b) - \hat{\gamma}_{t^-}(a)$ que corresponde a um decréscimo de pelo menos 1 de acordo com o Lema 3.2. Então, o número de eventos é proporcional ao valor do potencial total do heap que é $O(n \log^2 n)$. □

Considerando um $(1, n, m)$-cenário dinâmico, estamos trabalhando com segmentos de reta e a análise anterior pouco vale. Em [Ba99], Basch mostra que o número total de eventos no heap cinético em um $(1, n, m)$-cenário dinâmico é $O(m\sqrt{n \log n})$ usando um argumento que é uma mescla dos argumentos usados em [EW85, ELSS85] para provar limitantes para o k-nível (veja Capítulo 2) de um arranjo de linhas retas no plano.

3.3 Torneio Cinético

Apesar de o heap cinético atender a todas as exigências para que uma estrutura de dados cinética tenha um bom desempenho segundo o modelo, vamos apresentar uma terceira estratégia para a solução cinética do problema do máximo cujos limitantes comprovadamente valem para (δ, n)-cenários e que será particularmente importante para a solução de outros problemas cinéticos.

Algoritmo 3.1 EncontraMaximo(S, i, f)

Entrada: $i \leq f$, índices início e fim do conjunto indexado S.
Saída: o índice j do elemento S_j que é o máximo entre $S_i, \ldots, S_f$.

```
se i = f então
  retorna i
senão
  m ← ⌊(f − i)/2⌋
  l ← EncontraMaximo(S, i, m)
  r ← EncontraMaximo(S, m + 1, f)
  se S_l > S_r então
    retorna l
  senão
    retorna r
```

Considere um algoritmo divisão-e-conquista para obtermos o máximo em um conjunto de n pontos (ou o seu índice no conjunto) como o Algoritmo 3.1. Nesse algoritmo, o máximo é dado pela comparação entre os máximos de dois subconjuntos de $n/2$ pontos, e assim recursivamente. Podemos representar a execução do algoritmo através de uma árvore de recursão onde cada nó contém o valor retornado pela chamada recursiva e cada aresta corresponde a uma comparação entre dois elementos do conjunto.

No nível mais baixo temos n elementos, causando $n/2$ comparações para decidir quais serão os elementos que compõem o segundo nível, o que causa $n/4$ comparações nesse nível e assim por diante. Desse modo, o número total de comparações que o algoritmo faz para encontrar o máximo entre n elementos é $C(n) = n - 1$.

No final o algoritmo realizou $n - 1$ comparações. A idéia na cinetização é usar essas comparações como certificados. Vista de baixo para cima a árvore da recursão forma algo semelhante a um torneio eliminatório, onde vence o elemento que tem maior coordenada y, classificando-se para a próxima fase. As comparações são os jogos eliminatórios e o campeão é o atributo procurado (o máximo do conjunto). Veja a Figura 3.5.

Um certificado corresponde a uma comparação no algoritmo estático e o conjunto de todos eles atesta a validade da resposta contida na raiz do torneio. Novamente os certificados têm um prazo de validade que é calculado a partir dos planos de vôo dos pontos no cenário considerado. Quando um certificado perde a sua validade, temos a

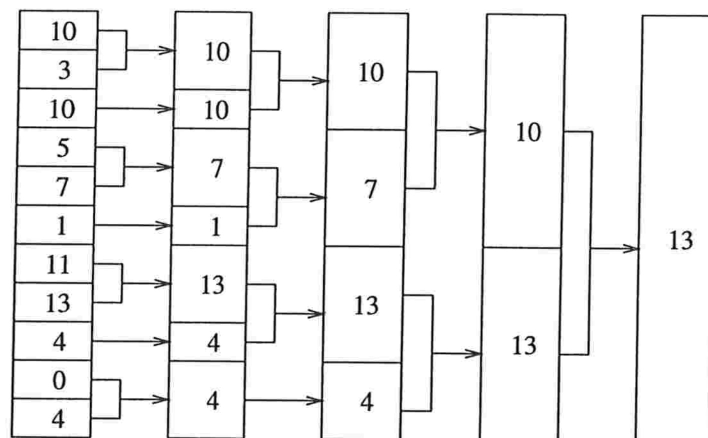


Figura 3.5: Torneio que seleciona o máximo sobre um conjunto de 11 números. Cada junção corresponde a uma comparação e é associada a um certificado.

inversão entre o vencedor e o perdedor de uma das partidas (comparações) do torneio. Nesse caso, basta atualizarmos os certificados no caminho entre o nó onde ocorreu a inversão e a raiz. Vamos atualizando os certificados com o novo vencedor até que ele perca uma partida ou seja o novo campeão do torneio. Como a árvore do torneio é balanceada, a atualização gasta tempo $O(\log n)$ e envolve no máximo $O(\log n)$ dos certificados existentes.

Essa estrutura, chamada de *torneio cinético*, é uma estrutura de resposta rápida pois o custo de se processar um evento é o custo de se alterar $O(\log n)$ certificados. Como o custo de alterarmos um certificado é constante, então o custo de se processar um evento é $O(\log n)$.

A fila de eventos contém, no máximo, $n-1$ eventos (que é o número de comparações necessárias para descobrirmos o máximo do conjunto). Assim, o torneio cinético pode ser considerado uma estrutura de dados cinética compacta porque o tamanho da fila de eventos é $\Theta(n)$.

Um determinado ponto aparece em, no máximo, $O(\log n)$ certificados, sendo um para cada partida sua no torneio, correspondendo a um nível na árvore. Então o torneio cinético é uma estrutura local, pois um único ponto aparece em $O(\log n)$ eventos na fila.

O número total de eventos no pior caso do torneio cinético é $O(n \log n)$ em um $(1, n)$-cenário conforme mostra o Teorema 3.4, levando a um fator de $O(\log n)$ na razão entre esse total e o total de eventos externos no pior caso. Assim, o torneio cinético é uma estrutura considerada eficiente segundo o modelo cinético.

Teorema 3.4 *O número total de eventos no torneio cinético sobre n pontos em um* $(1, n)$*-cenário é* $O(n \log n)$.

Prova: Cada evento no torneio cinético corresponde a uma troca no vencedor de cada partida (nó). Então precisamos contar o número de vezes que o conteúdo de um nó do torneio pode mudar. Vamos assumir que as folhas são os próprios elementos. Nos nós de altura 0 nunca teremos mudanças no ponto ali armazenado. Nos nós de altura 1, poderemos ter uma única alteração. Indutivamente, o conteúdo dos nós de altura h poderão ser alterados $k-1$ vezes, onde k é o número de folhas da sub-árvore com raiz em um nó fixo de altura h. Assim, o total de eventos E é dado por:

$$E = \sum_{h=0}^{\lceil \log n \rceil} \frac{n}{2^h} \cdot (2^h - 1)$$

$$E = \sum_{h=0}^{\lceil \log n \rceil} \frac{n}{2^h} \cdot 2^h - \sum_{h=0}^{\lceil \log n \rceil} \frac{n}{2^h} \leq \sum_{h=0}^{\lceil \log n \rceil} \frac{n}{2^h} \cdot 2^h$$

$$E \leq n \cdot \sum_{h=0}^{\lceil \log n \rceil} 1 = n \cdot \log n$$

$$E = O(n \log n) \quad \square$$

Dinamizando o Torneio Cinético

Podemos analisar o comportamento da estrutura do torneio cinético para outros tipos de cenário (onde as equações dos planos de vôo podem assumir graus maiores que 1) e incluir ainda suporte à inserções e remoções durante o transcorrer do tempo. Nesse caso estamos em um (δ, n, m)-cenário dinâmico e o número total de eventos externos é $O(\lambda_{\delta+2}(m))$. O Teorema 3.5 mostra que ainda assim o torneio cinético é uma estrutura cinética eficiente.

Teorema 3.5 *O número total de eventos no torneio cinético em um (δ, n, m)-cenário dinâmico é $O(\lambda_{\delta+2}(m) \log n)$.*

Prova: Quando processamos um evento, o número de certificados que temos que alterar é proporcional ao número de nós cujos conteúdos mudam durante o processamento. Considere um nó v na árvore do torneio e seja n_v o número de elementos que aparecem na sub-árvore com raiz em v durante todo o tempo. O conteúdo de v, que é o máximo na sub-árvore, muda no máximo $\lambda_{\delta+2}(n_v)$ vezes. Seja L_i o conjunto de nós no nível i, temos

$$\sum_{v \in L_i} n_v \leq m$$

E o número total de mudanças no nível i é no máximo

$$\sum_{v \in L_i} \lambda_{\delta+2}(n_v) \leq \lambda_{\delta+2}(m)$$

Somando sobre todo o torneio (que tem profundidade $\log n$), o número total de mudanças é $O(\lambda_{\delta+2}(m)\log n)$. □

3.4 Considerações Finais

Como se vê, o torneio cinético mostra-se mais eficiente que o heap segundo o modelo cinético pois gera um fator de eficiência logarítmico, enquanto que o heap cinético gera um fator de eficiência $O(\log^2 n)$. Por outro lado, o heap possui localidade e resposta rápida ótimas ($O(1)$), o que não ocorre com o torneio.

Apesar da eficiência em relação ao número de eventos totais/externos ser melhor no torneio cinético que no heap cinético, o torneio não possui localidade e resposta rápida ótimas. Por esse motivo, em uma aplicação onde o movimento dos pontos se altera muitas vezes pode ser preferível optar pelo heap. Já uma aplicação em que sabemos de antemão que o atributo será alterado muitas vezes por força do movimento dos objetos (e não por alterações on-line) será melhor implementada usando o torneio cinético. Uma vantagem da estrutura usando torneio é que o resultado é conhecido para qualquer tipo de movimento, enquanto no heap apenas temos resultados conhecidos para movimentos lineares. Além disso, o torneio cinético parece ser uma estrutura mais natural para implementações em arquitetura paralela.

Tanto a estrutura do heap quanto o torneio cinético oferecem suporte a alterações descontínuas. Esse tipo de alteração envolve remoção seguida de inserção na estrutura de dados cinética pois um determinado ponto que sofra tal alteração pode se "teleportar" no espaço saindo de uma posição e aparecendo em outra completamente diferente no mesmo instante. Como as duas estruturas de dados são eficientes em um $(1, n, m)$-cenário dinâmico elas podem ser usadas normalmente no caso em que são permitidas alterações descontínuas. Porém, em uma aplicação desse tipo é preferível usar o torneio cinético por sua melhor eficiência.

Em [Ba99] é apresentada uma outra estrutura de dados cinética para a manutenção do máximo no cenário cinético denominada *kinetic heater*. Essa estrutura é baseada em árvores aleatórias com propriedades muito parecidas com as do heap cinético. Uma desvantagem dessa estrutura é não ser determinística, baseando-se nas leis das probabilidades.

Capítulo 4

Par Mais Próximo

Neste capítulo vamos apresentar nosso primeiro problema em Geometria Computacional bidimensional, o problema do Par de Pontos Mais Próximo:

Problema 4.1 (Problema do Par de Pontos Mais Próximo) *Dado um conjunto S de n pontos movendo-se no plano, deseja-se saber a todo instante qual o par de pontos (p, q) de S com menor distância $d(p, q)$.*

Possíveis aplicações para esse problema seriam em tráfego aéreo e telefonia. Imagine que estejamos desenvolvendo um sistema de controle de tráfego aéreo para um determinado aeroporto e precisamos monitorar todo o trânsito de aviões nas proximidades desse aeroporto. Assumindo que as aeronaves sobrevoam a região monitorada em altitudes praticamente iguais, podemos associar as posições de cada avião a posições no plano bidimensional como se estivéssemos vendo o tráfego aéreo de cima (ou de baixo). A idéia seria prestar atenção nos pares de aviões que estejam mais próximos entre si, comunicando-os do fato a fim de evitar a ocorrência de acidentes.

Antes de começar a desenvolver o assunto, convém ressaltar que estaremos assumindo que apenas um par de pontos é o mais próximo dentre todos os pares de pontos do conjunto considerado a cada instante, exceto no instante da ocorrência de eventos relacionados ao par de pontos mais próximo. As distâncias consideradas são distâncias euclidianas.

A primeira idéia que poderia vir à cabeça para solucionar o problema cinético do par de pontos mais próximo no plano seria usar uma versão cinética do algoritmo quadrático que resolve o problema estático selecionando o par de pontos mais próximo dentre todos os pares possíveis. Para isso, basta que se gerem todos os pares de pontos possíveis em S e armazená-los em um heap (veja Apêndice A) que minimize as distâncias. A versão cinética desse algoritmo teria como certificados as arestas de ligação do heap, como no problema do máximo cinético (veja Capítulo 3). A cada evento, tudo o que teríamos a fazer seria atualizar o heap invertendo os elementos que formavam o certificado processado e alterando um número constante de outros

certificados. Porém, existem $O(n^2)$ pares de pontos obrigando-nos a manter $O(n^2)$ certificados, o que resultaria em uma estrutura de dados cinética que não é local nem compacta pois cada ponto aparece em $O(n)$ certificados e o total de certificados na fila de eventos a qualquer instante é $O(n^2)$. Portanto, devemos procurar um algoritmo que obtenha o par mais próximo estático de forma mais eficiente para transformá-lo em uma boa estrutura de dados cinética.

Uma idéia seguinte seria utilizar o algoritmo apresentado por Preparata e Shamos [PS85] que resolve o problema através de uma técnica de divisão-e-conquista com complexidade de tempo $O(n \log n)$. Em poucas palavras, o algoritmo divide o conjunto de pontos ao meio com uma linha vertical $x = x_0$ e calcula recursivamente o par mais próximo na metade esquerda (cuja distância é δ_L) e na metade direita (cuja distância é δ_R). Na fase da conquista o algoritmo verifica se algum par formado por pontos que estão em metades diferentes possui distância menor que $\delta = \min(\delta_L, \delta_R)$ como mostra a Figura 4.1.

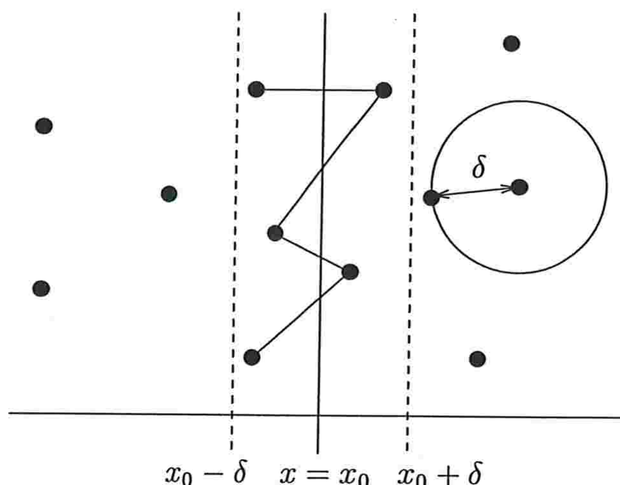


Figura 4.1: Em uma iteração do algoritmo de Shamos procuramos um par de pontos com distância menor que δ na faixa $[x_0 - \delta, x_0 + \delta]$.

Em uma versão cinética, o algoritmo de Shamos forneceria certificados do tipo $x_p < x_0 - \delta$ ou $x_p > x_0 + \delta$ para cada ponto p em cada fase de divisão do algoritmo. Nesse caso, δ é a distância entre os pontos que formam o par mais próximo considerando as duas metades e x_0 representa a linha vertical da divisão no passo correspondente do algoritmo. Dessa forma um mesmo ponto poderia aparecer em $O(\log n)$ certificados.

Porém, em cada nível de recursão do algoritmo o total de pontos envolvidos é n, o que resultaria num total de $O(n \log n)$ certificados na fila de eventos a qualquer instante, caracterizando a estrutura coma uma estrutura de dados cinética não-compacta. Além disso, uma mudança no par mais próximo, por exemplo, alteraria todos os certificados que dependem desse par. Assim, pelo menos os $O(n)$ certificados do primeiro nível de recursão deveriam ser atualizados. Portanto, a estrutura de dados cinética obtida

também não é local e nem de resposta rápida pois teríamos um número pelo menos linear de certificados a serem atualizados no processamento de um evento que afete o par mais próximo. Esse exemplo mostra que nem sempre um bom algoritmo que resolve o problema estático se apresenta como uma boa alternativa para a geração da estrutura de certificados para a versão cinética do mesmo problema.

Basch, Guibas e Hershberger descrevem em [BGH97, BGH98] um novo algoritmo que resolve o problema do par mais próximo estático também em tempo $O(n \log n)$ e que admite uma cinetização compacta, local, eficiente e de resposta rápida. Trata-se de um algoritmo sofisticado que usa a idéia de linha de varredura e que foi desenvolvido especialmente para produzir uma estrutura de dados cinética que seja eficiente sob as quatro medidas de desempenho introduzidas pelo modelo cinético. Vamos apresentar a versão estática desse algoritmo na próxima seção e discutir a sua versão cinética mais adiante. No final mostraremos algo em relação ao funcionamento do algoritmo em cenários cinético-dinâmicos.

4.1 Algoritmo Estático

O algoritmo proposto por Basch, Guibas e Hershberger para resolver o problema do par mais próximo é baseado na idéia de dividir a área ao redor de cada ponto p em seis cones, cada um com abertura de 60°. Os cones são formados pelas áreas definidas pelo eixo y e as retas $x \pm 30°$ se interceptando em p. Cada cone possui um eixo central que são as retas x, $x + 60°$ e $x - 60°$ (veja a Figura 4.2). A idéia é selecionar o vizinho que está mais próximo de p em cada um dos cones ao redor de p. Com certeza o vizinho mais próximo de p será selecionado por um dos cones e do conjunto de vizinhos mais próximos de cada ponto de S podemos extrair o par de pontos mais próximo. Vamos assumir que dois pares não possuem distâncias iguais, resultando em um único par de pontos mais próximo em S.

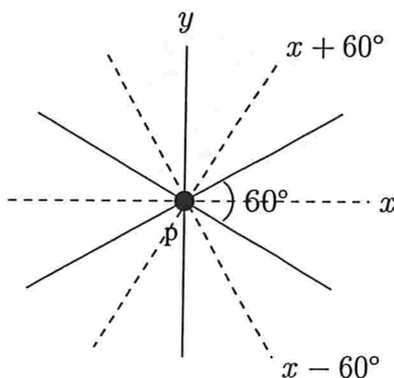


Figura 4.2: Divisão em cones em torno do ponto p.

Apesar de dividirmos a área ao redor de cada ponto p em seis cones, estaremos interessados em apenas três deles, mais especificamente os cones situados à direita de p. Esses cones, juntos, examinarão todos os pontos no semi-plano à direita de p. Os vizinhos em relação aos três cones da esquerda serão obtidos nas vizinhanças dos pontos que estão à esquerda de p. A seguir vamos nos concentrar em mostrar como as coisas funcionam para o cone cujo eixo central é o eixo x. Tudo se aplicará aos outros dois cones por rotação.

Definimos *dominância* de um ponto p como sendo o cone à direita de p limitado pelas retas que atravessam p e formam ângulos de $\pm 30°$ com o eixo x. A dominância de p será denotada por $Dom(p)$. Se um ponto q está dentro de $Dom(p)$, dizemos que q é dominado por p, ou que p domina q.

Se p é o ponto mais à direita dentre o conjunto dos pontos que dominam q, dizemos que p *cobre* q. Para que essa definição funcione para qualquer ponto, suponha que exista um ponto extra com coordenadas $(-\infty, 0)$. O conjunto dos pontos que são cobertos por p é o *conjunto de candidatos* do ponto p, denotado por $Cands(p)$. O elemento mais à esquerda em $Cands(p)$ é chamado *candidato esquerdo* de p e é denotado por $lcand(p)$. Dizemos que um par $(p, lcand(p))$ é um *par candidato* a ser o par mais próximo do conjunto de pontos. Finalmente, o conjunto $Maxima(p)$ contém todos os pontos à direita de p que não são dominados por nenhum ponto à direita de p. O ponto $q \in Maxima(p)$ com menor coordenada y acima de $Dom(p)$ é chamado $up(p)$ e, simetricamente, o ponto $q' \in Maxima(p)$ com maior coordenada y abaixo de $Dom(p)$ é chamado $low(p)$. Para que sempre tenhamos algum ponto como resposta para $low(p)$ e $up(p)$, devemos supor que existam dois pontos extras com coordenadas $(+\infty, +\infty)$ e $(+\infty, -\infty)$. Os conjuntos e pontos especiais para um ponto p fixado podem ser vistos na Figura 4.3.

Lema 4.2 *Sejam T um triângulo equilátero cujos vértices são i, j, k e m um ponto na aresta jk de T (oposta a i). Então para todo ponto $l \neq i$ no interior de T, $d(l, m) < d(i, m)$.*

Prova: Traçando um círculo de raio $d(i, m)$ centrado em m, teremos um círculo que compreende totalmente o triângulo (veja Figura 4.4 (*a*)). Se l está dentro desse círculo, então $d(l, m) \leq d(i, m)$. Como $l \neq i$, então $d(l, m) < d(i, m)$. □

Lema 4.3 *Sejam S um conjunto de pontos e (a, b) o par de pontos mais próximo em S. Supondo que a está à esquerda de b, então $b \notin Dom(p)$ para qualquer outro ponto p à direita de a.*

Prova: Tome o círculo centrado em a com raio $d(a, b)$. Certamente b estará localizado na fronteira deste círculo. Qualquer ponto p à direita de a tal que $b \in Dom(p)$ deve estar à esquerda de b e fora de $Dom(a)$, caso contrário $d(p, b) < d(a, b)$ e (a, b) não seria o par mais próximo. Portanto, qualquer ponto p nestas condições deve estar em uma

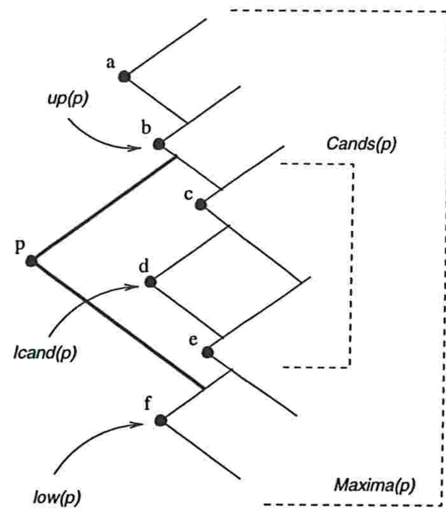


Figura 4.3: Os conjuntos $Cands(p)$ e $Maxima(p)$. Também são indicados os pontos especiais $lcand(p), up(p)$ e $low(p)$.

das regiões indicadas na Figura 4.4 (b). Porém estas regiões estão totalmente dentro do círculo, o que também não possibilitaria que (a, b) fosse o par mais próximo. □

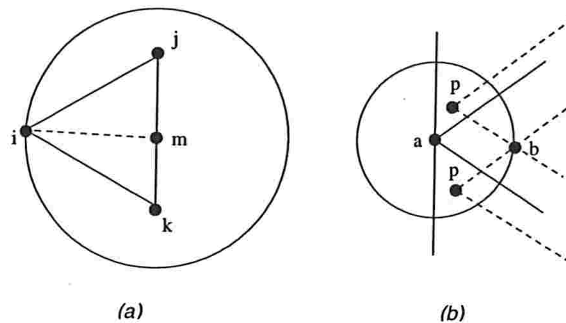


Figura 4.4: Figuras auxiliares para o Lema 4.2 e o Lema 4.3

Lema 4.4 *Sejam S um conjunto de pontos e (a, b) o par de pontos mais próximo em S nas condições do lema anterior. Se $b \in Dom(a)$, então b é $lcand(a)$.*

Prova: Pelo Lema 4.3 sabemos que b é coberto por a, ou seja, não existe nenhum ponto p à direita de a que domina b. Agora assuma que exista um ponto q dominado por a à esquerda de b. q certamente deve estar dentro do triângulo equilátero formado pelos limites de $Dom(a)$ e a linha vertical que passa por b. Porém, pelo Lema 4.2, $d(q, b) < d(a, b)$ contrariando a hipótese de que (a, b) é o par mais próximo. □

Com base nos lemas anteriores, conclui-se que o vizinho mais próximo de um dado ponto p dentro de $Dom(p)$ é exatamente $lcand(p)$, e $lcand(p)$ é único (por hipótese). Assim, como o par $(p, lcand(p))$ é um par candidato, temos no máximo n pares candidatos em um conjunto de n pontos considerando o cone centrado pelo eixo x. Podemos estender o conceito de dominância para os outros dois cones (nas direções $\pm 60°$) por rotação, o que leva a um máximo de $3n$ pares candidatos.

Corolário 4.5 *O par de pontos mais próximo de um conjunto S de n pontos é um dos $O(n)$ pares candidatos associados com as direções $0°, 60°$ e $-60°$.*

Prova: Suponha que (a, b) é o par de pontos mais próximo de S, com a à esquerda de b. Os três cones de dominância de a cobrem o semi-plano direito de a completamente. Assim, b deve estar em um deles. Pelo Lema 4.4, (a, b) deve ser um dos pares candidatos em uma das direções dos cones de dominância ($0°, 60°$ e $-60°$). □

O algoritmo é constituído de duas fases. Vamos começar descrevendo a primeira fase, que é executada em três passos: um calcula todos os pares candidatos referentes à direção horizontal, outro calcula os pares relativos à direção 60° e um terceiro passo os correspondentes à direção $-60°$. No passo relativo à direção horizontal a linha de varredura percorre o plano da direita para a esquerda. Nos demais passos tudo deve ser rotacionado apropriadamente ($\pm 60°$). O Algoritmo 4.1 apresenta esse procedimento para a direção horizontal. Ele utiliza $Maxima$ como estrutura principal para a varredura, que é executada da direita para a esquerda. A cada passo do algoritmo o conteúdo de $Maxima$ é exatamente $Maxima(p)$.

Algoritmo 4.1 ParesCandidatosHorizontal(S)

Entrada: Conjunto S de pontos no plano.
Saída: Lista PC de pares candidatos relativos à direção horizontal.

```
  Ordene os elementos de S na direção horizontal
  PC ← ∅
  Maxima ← ∅
  para cada ponto p ∈ S da direita para a esquerda faça
    Cands(p) ← Maxima ∩ Dom(p)
    lcand(p) ← elemento mais à esquerda em Cands(p)
    Maxima ← (Maxima \ Cands(p)) ∪ p
    PC ← PC ∪ (p, lcand(p))
  retorna PC
```

Inicialmente é necessário que ordenemos os objetos na direção que está sendo verificada. Esse passo de ordenação pode ser implementado para execução em tempo $O(n \log n)$ usando Heapsort (veja Apêndice A) ou Mergesort. Se $Maxima$ for arma-

zenado em uma árvore balanceada que forneça busca, inserção, remoção, *join*[1] e *split*[2] em tempo logarítmico [Kn73], podemos encontrar $up(p)$ e $low(p)$ em tempo $O(\log n)$. Como $Cands(p)$ é uma porção consecutiva dentro de $Maxima$ e é limitado por $up(p)$ e $low(p)$, encontrando $up(p)$ e $low(p)$ em tempo logarítmico, podemos extrair $Cands(p)$ de $Maxima$ também em tempo $O(\log n)$. $lcand(p)$ será o elemento de $Cands(p)$ que foi inserido por último durante a varredura, e o tempo total de execução do Algoritmo 4.1 é $O(n \log n)$.

O mesmo algoritmo é aplicado com o plano rotacionado $\pm 60°$ para obter todos os pares candidatos. Cada direção contribui com n pares candidatos, totalizando $3n$ pares candidatos.

Na segunda fase do algoritmo devemos selecionar o par mais próximo dentre todos os $3n$ pares candidatos. Esse passo pode ser feito em tempo linear. Dessa forma podemos obter o par de pontos mais próximo de um conjunto de n pontos consumindo tempo $O(n \log n)$. O procedimento completo para resolver o problema estático do par de pontos mais próximo é descrito no Algoritmo 4.2.

Algoritmo 4.2 ParMaisProximoEstático(S)

Entrada: Conjunto S de pontos no plano.
Saída: Par de pontos de S com menor distância.

```
P ← ParesCandidatosHorizontal(S)
P ← P ∪ ParesCandidatosDir+60°(S)
P ← P ∪ ParesCandidatosDir-60°(S)
retorna ExtraiMinimo(P)
```

`ParesCandidatosHorizontal` devolve uma lista com os pares candidatos relativos ao eixo horizontal. As duas chamadas seguintes invocam as versões do mesmo algoritmo para as direções $\pm 60°$, devolvendo as suas respectivas listas de pares candidatos. `ExtraiMinimo` recebe o conjunto de todos os pares candidatos e organiza o torneio entre eles de modo semelhante ao que é feito no Algoritmo 3.1. Porém, desta vez estamos interessados em minimizar (a distância) ao invés de maximizar, e `ExtraiMinimo` retorna o par candidato com menor distância. Esse par é o par de pontos mais próximo do conjunto de pontos S.

Na segunda fase do algoritmo para o problema estático selecionamos o par mais próximo dentre os $3n$ pares candidatos em tempo linear. Na versão cinética do problema deveremos manter o par com menor distância, ou seja, teremos que manter o par com menor distância em um conjunto de pares de pontos candidatos. Para isso podemos utilizar alguma das estruturas discutidas para o problema do máximo (veja Capítulo 3) como a lista ordenada, o torneio cinético ou o heap cinético.

[1] *join* é a operação que concatena duas árvores disjuntas em uma única árvore
[2] *split* é a operação que particiona uma árvore em duas, sendo a operação inversa de *join*.

4.2 Resolvendo o Problema Cinético

Na fase de cinetização do algoritmo estático apresentado na seção anterior vamos considerar a utilização de um *torneio cinético* entre os pares candidatos na fase final do algoritmo estático. Isso será feito por motivos de análise da estrutura para os tipos de movimento permitidos. Tal atitude se justifica pelo fato de que não se conhecem limites de eficiência comprovados na utilização do heap cinético para movimentos não-lineares (veja Capítulo 3) e também pela falta de eficiência demonstrada pela lista ordenada.

O algoritmo estático fornece um conjunto de certificados que atestam a validade da solução obtida. Esses certificados se apresentam em quatro tipos: os que certificam a ordenação dos elementos de S quanto à coordenada x; os certificados para a ordem das projeções dos pontos no eixo $x + 60°$; os certificados para a ordem das projeções dos pontos no eixo $x - 60°$; e os que certificam o torneio dos pares candidatos.

Analisando a localidade da estrutura, verificamos que cada ponto está envolvido em no máximo 2 certificados em cada uma das 3 listas ordenadas, resultando em um máximo de 6 certificados por ponto referentes às listas ordenadas das direções consideradas. Além disso, para cada direção, cada ponto está presente em no máximo 2 pares candidatos (um par onde ele é dominado pelo seu "parceiro" e outro onde ele domina o parceiro), totalizando 6 pares candidatos possíveis para cada elemento de S. Como um mesmo elemento está envolvido em um número constante de pares candidatos, esse elemento aparece em $O(1)$ folhas do torneio e em $O(\log n)$ nós internos (veja Capítulo 3). Assim a estrutura de certificados tem localidade $O(\log n)$.

De posse da estrutura que contém o atributo geométrico desejado (o par mais próximo) e o conjunto de certificados que comprovam que o conteúdo de tal estrutura é válido, precisamos mostrar agora como manter essa estrutura quando há a necessidade de se processar um evento.

Vamos começar descrevendo o que devemos fazer para atualizar a estrutura quando um certificado para a lista ordenada relativa à orientação horizontal perde a sua validade. Supondo que p e q são os pontos que constituem tal certificado, uma troca entre eles na ordenação horizontal pode causar uma mudança na cobertura de um número linear de outros pontos (veja Figura 4.5). Do mesmo modo, uma mudança na ordenação relativa à orientação $+60°$ pode mudar um número linear de up's e low's (Figura 4.6). Para obtermos uma estrutura de dados cinética que seja de resposta rápida, vamos precisar manter três estruturas em forma de árvores binárias balanceadas para cada ponto. Antes de descrevê-las vamos apresentar algumas conclusões que podemos tirar com relação a $Cands(p)$, $up(p)$ e $low(p)$ que facilitarão as coisas:

Proposição 4.6 *Se $q, r \in Cands(p)$ então q é antecessor de r na $+60°$-ordem se e somente se q é sucessor de r na $-60°$-ordem. E isso ocorre se e somente se q tem coordenada y menor que a de r.*

Prova: Da hipótese podemos concluir que q não está na dominância de r pois se tal

fato ocorre, $q \notin Cands(p)$. Do mesmo modo r não está na dominância de q. Então r está acima ou abaixo de $Dom(q)$. Nesse caso as $+60°$-ordem e $-60°$-ordem são opostas uma da outra. Além disso, a $+60°$-ordem entre q e r é a mesma que a ordem vertical pois q e r pertencem a $Cands(p)$. □

Proposição 4.7 *Se $up(p) = up(q)$ e p está à esquerda de q, então p é antecessor de q na $-60°$-ordem. Simetricamente, se $low(p) = low(q)$ e p está à esquerda de q, então p é antecessor de q na $+60°$-ordem.*

Prova: Seja $r = up(p) = up(q)$. Se q é antecessor de p na $-60°$ então q está acima de $Dom(p)$, mas q está abaixo de r, então r não pode ser $up(p)$, o que é uma contradição. A prova para $low(p) = low(q)$ é simétrica. □

Com esses resultados, podemos descrever as três estruturas em forma de árvores binárias que devem ser armazenadas para cada ponto p. $Cands(p)$ contém os candidatos de p como uma seqüência ordenada pela coordenada y e é fornecido em um passo do loop em `ParesCandidatosHorizontal` (Algoritmo 4.1). De acordo com a Proposição 4.6 essa ordem de $Cands(p)$ é a mesma que as $\pm 60°$-ordens. Portanto, basta que se guarde essa ordenação em uma árvore binária balanceada que forneça as operações de busca e atualização em tempo logarítmico. Além disso, cada nó da árvore deve apontar para o nó pai e a raiz para p, o que facilitará o tratamento de trocas entre os elementos da árvore no processamento de um evento, e cada nó deve apontar também para o ponto mais à esquerda (no plano) da sua sub-árvore na ordem horizontal. Assim podemos encontrar a cobertura de cada ponto em tempo $O(\log n)$ caminhando na árvore e $lcand(p)$ em tempo $O(1)$ usando o apontador para o descendente cuja posição no plano é a posição mais à esquerda, e torna-se fácil atualizar $lcand(p)$ ao processar um evento.

Cada ponto p do conjunto também deve ter outras duas árvores associadas: $Hits_{up}(p)$ e $Hits_{low}(p)$. A primeira estrutura deve armazenar todos os pontos q para os quais $up(q) = p$, ordenados pela coordenada x. De acordo com a Proposição 4.7 essa ordem é suficiente para sabermos a ordenação nas $\pm 60°$-ordens. Novamente cada nó da árvore deve apontar para o nó pai e a raiz deve apontar para p. Dessa forma podemos encontrar $up(q)$ para qualquer ponto q em tempo logarítmico. $Hits_{low}(p)$ é uma estrutura similar que armazena todos os pontos q para os quais $low(q) = p$ e onde é possível encontrar $low(q)$ em tempo $O(\log n)$ qualquer que seja o ponto q.

Essas são as únicas estruturas de que precisaremos para a cinetização. Note que não precisamos usar $Maxima$ na cinetização porque toda a informação necessária é descrita pelas 3 estruturas: $Cands, Hits_{up}$ e $Hits_{low}$. O algoritmo estático pode criar essas estruturas em tempo $O(n \log n)$ da seguinte forma: $Cands(p)$ é fornecido a cada passo da linha de varredura e as árvore $Hits$ podem ser construídas incrementalmente a cada vez que um up ou um low é encontrado para arrancar $Cands(p)$ de $Maxima$.

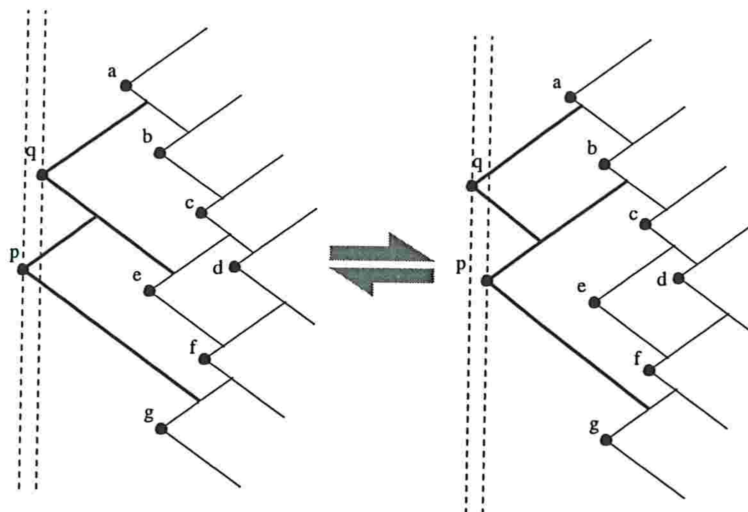


Figura 4.5: Um evento entre p e q na direção horizontal muda $Cands(p)$ e $Cands(q)$.

A seguir apresentaremos como o processamento de um evento pode afetar as estruturas $Cands$ e $Hits$ de cada ponto. Começaremos pelos eventos de troca de posições na lista que contém a ordenação horizontal. Vamos assumir que p está à esquerda de q e que p está abaixo de q no instante imediatamente anterior à ocorrência do evento, conforme mostra a Figura 4.5. O caso em que p estiver acima de q é tratado de modo semelhante. O pseudo-código é apresentado no Algoritmo 4.3 que usa a sub-rotina `Cover`. Ela recebe um ponto e retorna a cobertura desse ponto.

Se q é $up(p)$, como na Figura 4.5, então a inversão na ordem horizontal mudará a estrutura de ambos os pontos. Especificamente, os itens de $Maxima(q)$ em $Dom(q) \cap Dom(p)$ são transferidos de $Cands(q)$ para $Cands(p)$ e há uma mudança em $up(p)$. O novo ponto $up(p)$ deve obrigatoriamente estar em $Dom(q)$. Inversamente, p passa a ser $low(q)$. A fase 1 do Algoritmo 4.3 toma conta dessas mudanças, fazendo com que tudo seja arrumado em tempo $O(\log n)$. Se q não é $up(p)$, então não haverá nenhuma mudança nas estruturas $Cands$ e $Hits$.

Havendo ou não as mudanças apresentadas acima, um $lcand$ pode ter sido alterado na ocorrência do evento. Se $p, q \in Cands(u)$ para algum ponto u, então é necessário atualizar os campos que indicam o elemento mais à esquerda em u e em $Dom(z)$ para cada ponto $z \in Dom(u)$. Assim, devemos atualizar os elementos na árvore $Cands(u)$. Como as comparações entre p e q são recalculadas, os ancestrais comuns dos dois pontos devem ser alterados, o que pode causar uma mudança em $lcand(u)$. Na fase 2 do Algoritmo 4.3 essas manipulações são tratadas em tempo $O(\log n)$. Uma mudança em $lcand(u)$ acarreta em alterações nos eventos relativos ao torneio de pares candidatos como se houvesse uma alteração descontínua em um torneio 1-dimensional. Voltaremos a esse detalhe mais adiante.

Agora passaremos à apresentação do procedimento de atualização das estruturas

Algoritmo 4.3 ProcessaEventoHorizontal(p, q)

Entrada: Pontos p e q responsáveis pelo evento, com p à esquerda de q e p abaixo de q no instante anterior ao evento.

Saída: As estruturas $Cands, Hits_{up}$ e $Hits_{low}$ devidamente atualizadas.

se $q = up(p)$ **então** {fase 1}
  $A \leftarrow Cands(q) \cap Dom(p)$
  $Cands(q) \leftarrow Cands(q) \setminus A$
  $Cands(p) \leftarrow Cands(p) \cup A$
  $w \leftarrow low(q)$
  Remova q de $Hits_{low}(w)$
  Insira q em $Hits_{low}(p)$
  $v \leftarrow$ item de $Cands(q)$ com menor coordenada y
  Remova p de $Hits_{up}(q)$
  Insira p em $Hits_{up}(v)$
$p' \leftarrow \texttt{Cover}(p)$
$q' \leftarrow \texttt{Cover}(q)$
se $p' = q'$ **então** {fase 2}
  Atualize a árvore $Cands(p')$ começando do ancestral comum de p e q na árvore $Cands(p')$.

quando ocorre um evento na $+60°$-ordem, ou seja, no instante da ocorrência do evento, a linha que atravessa p e q (os pontos envolvidos no evento em questão) faz um ângulo de $-30°$ com o eixo x. Desta vez vamos assumir que p está à esquerda de q e acima de q, conforme mostra a Figura 4.6. Existem dois casos a se considerar: q entra ou sai de $Dom(p)$. O Algoritmo 4.4 mostra o que deve ser feito no caso em que q entra em $Dom(p)$.

Se q não é $low(p)$, então a troca entre p e q na $+60°$-ordem não afeta as estruturas de ambos os pontos, tornando desnecessária qualquer atualização nas estruturas.

Já no caso em que q é $low(p)$, a troca entre p e q na $+60°$-ordem não muda up ou low de nenhum ponto à direita de p. Somente $low(p)$ precisa ser modificado. Dos pontos à esquerda de p, somente os que têm q como up (todos os elementos em $Hits_{up}(q)$) precisam ter seus up's mudados para p. As únicas mudanças em $Cands$ ocorrem em $Cands(p)$, pois q está entrando em $Dom(p)$, e em $Cands(v)$ onde v é o ponto de cuja dominância q está saindo. Os campos que indicam o ponto mais à esquerda em $Cands$ de cada ponto devem ser atualizados durante essa modificação, de forma que continuem corretos.

No caso em que q sai de $Dom(p)$ as mudanças são reversas às descritas acima. De qualquer forma, apresentamos tais mudanças no Algoritmo 4.5.

Todas as operações descritas acima podem ser feitas em tempo logarítmico. O tratamento da perda de validade de certificados sobre a $-60°$-ordem é feito de forma simétrica aos certificados sobre a $+60°$-ordem. O último tipo de certificado utilizado

Algoritmo 4.4 ProcessaEvento60°Entrada(p, q)

Entrada: Pontos p e q responsáveis pelo evento, com p à esquerda de q e p acima de q no instante anterior ao evento com q entrando em $Dom(p)$.
Saída: As estruturas $Cands, Hits_{up}$ e $Hits_{low}$ devidamente atualizadas.

se $q = low(p)$ **então**
 $v \leftarrow$`Cover`(q)
 Remova q de $Cands(v)$
 Insira q em $Cands(p)$
 $t \leftarrow$ ponto mais à esquerda em $Hits_{up}(q)$ que esteja a direita de p
 Remova p de $Hits_{low}(q)$
 Insira p em $Hits_{low}(t)$
 $A \leftarrow$ pontos de $Hits_{up}(q)$ que estão à esquerda de t
 $Hits_{up}(q) \leftarrow Hits_{up}(q) \setminus A$
 $Hits_{up}(p) \leftarrow Hits_{up}(p) \cup A$

Algoritmo 4.5 ProcessaEvento60°Saida(p, q)

Entrada: Pontos p e q responsáveis pelo evento, com p à esquerda de q e p acima de q no instante anterior ao evento com q saindo de $Dom(p)$.
Saída: As estruturas $Cands, Hits_{up}$ e $Hits_{low}$ devidamente atualizadas.

se $q \in Cands(p)$ **então**
 $t \leftarrow low(p)$
 Remova p de $Hits_{low}(t)$
 Insira p em $Hits_{low}(q)$
 Insira q em $Hits_{low}(t)$
 $A \leftarrow$ pontos de $Hits_{up}(p)$ maiores que q na $-60°$-ordem
 $Hits_{up}(p) \leftarrow Hits_{up}(p) \setminus A$
 $Hits_{up}(q) \leftarrow Hits_{up}(q) \cup A$
 $v \leftarrow$ ponto mais à direita em $Hits_{up}(p)$
 Remova q de $Cands(p)$
 Insira q em $Cands(v)$

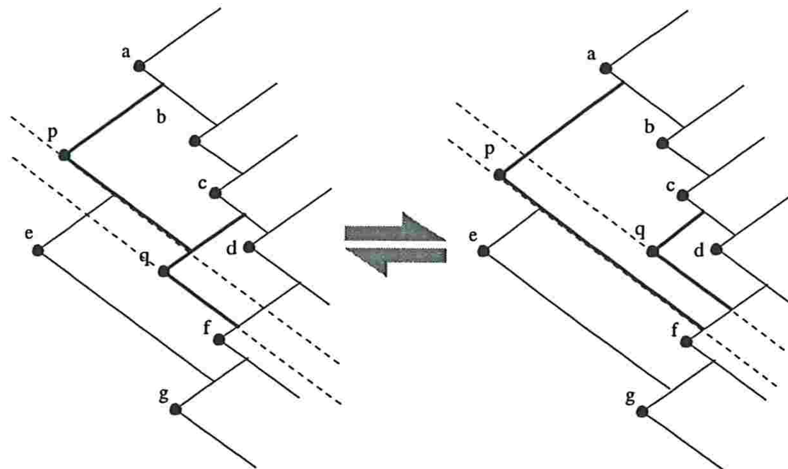


Figura 4.6: Um evento entre p e q na direção 60° faz q entrar/sair de $Dom(p)$.

nesta cinetização para o problema do par de pontos mais próximo é aquele que age sobre o torneio cinético entre os $3n$ pares candidatos. A manipulação de certificados no torneio foi mostrada no Capítulo 3 e o processamento de seu evento associado também consome tempo $O(\log n)$. Dessa forma, temos que qualquer evento possível nesta cinetização do problema pode ser processado em tempo logarítmico. Assim a estrutura é de resposta rápida.

4.3 Problema Dinâmico-Cinético

A raiz do torneio de pares candidatos $(p, lcand(p))$ contém o par de pontos mais próximo do conjunto S de n pontos em movimento. Podemos notar que quando $lcand(p)$ muda, pode haver uma alteração descontínua da distância associada no torneio cinético, semelhante a uma alteração descontínua no movimento de um ponto 1-dimensional, correspondendo a uma remoção do par antigo e posterior inclusão do novo par $(p, lcand(p))$. Como o torneio cinético dá suporte esse tipo de alteração, podemos desenvolver uma estrutura cinético-dinâmica.

Para que nossa estrutura de dados cinética apresentada seja eficiente no cenário cinético-dinâmico, devemos garantir que inserções e remoções nas 3 listas ordenadas das 3 direções consideradas sejam executadas em tempo logarítmico. Por esse motivo as 3 listas são armazenadas em árvores binárias balanceadas cujos elementos estão conectados por uma lista duplamente ligada. Assim é possível realizar inserções e remoções em tempo $O(\log n)$ e podemos acessar os vizinhos de um determinado elemento em tempo constante. À cada inserção corresponde o surgimento de no máximo 2 novos pares candidatos por direção (totalizando 6) que por sua vez devem ser inseridos no torneio de pares candidatos. Desse modo, no total a inserção de um novo ponto gasta

tempo $O(\log n)$ e cria $O(\log n)$ novos certificados. Na remoção as coisas acontecem de modo contrário, onde precisamos remover $O(\log n)$ certificados da fila de eventos, um número constante de pares candidatos do torneio, além de remover o ponto das 3 listas ordenadas. A remoção também gasta tempo logarítmico.

Teorema 4.8 *O número total de eventos na estrutura de dados cinética para o problema do par mais próximo é $O(\lambda_{2\delta+2}(n^2)\log(n))$ em um (δ, n, m)-cenário dinâmico.*

Prova: Um evento que não seja sobre o torneio cinético provoca uma troca entre dois elementos em uma das 3 ordenações mantidas. Como um par de itens pode trocar de posição no máximo δ vezes e temos n^2 pares de pontos, o número de eventos desse tipo é $O(n^2)$.

Como o movimento de cada ponto é descrito por um polinômio de grau máximo δ, o quadrado da distância entre dois pontos é dado por um polinômio de grau máximo 2δ. Dessa forma o torneio cinético entre os pares candidatos age sobre um $(2\delta, 3n, n^2)$-cenário dinâmico induzido e, de acordo com o Teorema 3.5, o número de eventos no torneio é $O(\lambda_{2\delta+2}(n^2)\log(n))$. Portanto o número total de eventos é $O(n^2) + O(\lambda_{2\delta+2}(n^2)\log(n))$ que é $O(\lambda_{2\delta+2}(n^2)\log(n))$. □

4.4 Considerações Finais

O número total de eventos externos associados às listas ligadas é $O(n^2)$ pois temos n^2 pares de pontos cujas trajetórias podem se cruzar no máximo δ vezes. O total de eventos externos relativos ao torneio é $O(\lambda_\delta(n))$ que é "sobrelinear". Portanto o número total de eventos externos é $O(n^2)$. Como $\lambda_{2\delta+2}(n^2)$ é "sobrequadrático" (é $n^2 \cdot \epsilon$, veja Seção 2.2), o quociente entre o número total de eventos e o número de eventos externos é $O(\epsilon \log(n))$, resultando em uma estrutura eficiente.

O número total de certificados associados a um único ponto é no máximo $O(\log n)$, pois cada ponto aparece em no máximo 2 certificados em cada lista ordenada, totalizando 6 certificados, e em até $O(\log n)$ certificados no torneio de pares candidatos. Portanto a estrutura é local.

A estrutura também é compacta pois o número total de eventos na fila de eventos a qualquer instante é linear: temos cada um dos $3n$ certificados (no máximo), mais $O(n)$ eventos do torneio cinético.

Finalmente, vamos analisar a quantidade de espaço total utilizado pela estrutura de dados cinética apresentada. O espaço total é dado pelas 3 listas ordenadas, as 3 árvores armazenadas em cada ponto, além do torneio cinético. Tanto o torneio como cada uma das listas ordenadas consomem espaço $O(n)$. Cada árvore também consome espaço $O(n)$ e, como existem n árvores, a estrutura consumiria $O(n^2)$ de espaço total. Porém isso não é verdade, pois como cada ponto é coberto por um unico ponto à sua

esquerda, um mesmo ponto aparece em apenas uma árvore $Cands$. Entào a contagem de todos os nós de todas as árvores $Cands$ é $O(n)$. Da mesma forma, cada ponto possui um único *up* e um único *low*, e a contagem de todos os nós de todas as árvores $Hits_{up}$ e $Hits_{low}$ também é $O(n)$. Então o espaço total utilizado pela estrutura é $O(n)$.

Capítulo 5

Fecho Convexo

Passamos agora à descrição de uma estrutura de dados cinética que mantém eficientemente a solução cinética para um segundo problema clássico em Geometria Computacional: o Problema do Fecho Convexo de um conjunto de pontos no plano.

Problema 5.1 (Problema do Fecho Convexo) *Dado um conjunto S de n pontos movendo-se no plano, deseja-se saber a todo instante o menor polígono convexo que contém todos os pontos de S. Esse polígono receberá a denominação de fecho convexo de S. A fronteira faz parte do fecho.*

Uma aplicação prática para o problema do fecho convexo cinético pode vir da robótica. Imagine que precisemos criar os mecanismos de controle de um robô que estará realizando movimentos em algum local cheio de obstáculos. Imagine também que esse robô possui partes articuláveis, podendo girar um "braço" ou carregar outros objetos. Um possível modelagem do movimento desse robô poderia levar em conta o fecho convexo que contenha o robô. Com a capacidade de mover seus membros e carregar objetos, o fecho convexo que contém o robô e sua carga poderá sofrer constantes modificações. Se esse fecho convexo não colidir com nenhum obstáculo no caminho, então o robô também não colidirá.

Existem diversos algoritmos que resolvem eficientemente a versão estática do problema. Dentre eles, devemos encontrar aquele que produza o melhor conjunto de certificados na fase de cinetização para que possamos construir uma estrutura de dados cinética eficiente sob os quatro pontos de vista estabelecidos pelo modelo cinético.

Basch, Guibas e Hershberger [BGH98, BGH97] consideraram um algoritmo divisão-e-conquista que resolve o problema estático em tempo $O(n \log n)$. O algoritmo divide o fecho convexo em duas partes: a parte de cima e a de baixo, e cada uma é calculada separadamente. O fecho convexo final é dado pela concatenação entre as duas partes obtidas.

Para facilitar as coisas, vamos assumir que dois pontos não possuem a mesma coordenada x. As duas partes do fecho convexo são então definidas da seguinte forma:

encontram-se os pontos extremos esquerdo e direito do conjunto S. A parte do fecho convexo que ficar acima da linha que interliga esses dois pontos é o fecho de cima. A parte complementar é o fecho de baixo (veja a Figura 5.1).

O algoritmo então parte em busca do fecho de cima recursivamente, dividindo ao meio o conjunto de pontos e construindo a solução na fase de conquista. A mesma coisa é feita para encontrar o fecho de baixo. Cada uma das partes é encontrada em tempo $O(n \log n)$. Porém a apresentação do algoritmo sobre o plano no qual estão as coordenadas dos pontos não é tão clara. Este algoritmo é apresentado, então, sobre o plano dual, aplicando o conceito de dualidade entre ponto e reta apresentado na Seção 2.4, onde o algoritmo se mostra mais natural, tendo a sua descrição mais facilmente entendida.

Cada ponto (a, b) do plano original é associado a uma reta cuja equação é $y = ax + b$. Não é difícil ver que o fecho convexo de cima no plano primal corresponde ao envelope superior no plano dual (Figura 5.1). Analogamente, a parte de baixo do fecho corresponde ao envelope inferior no plano dual. Utilizando o Algoritmo 2.1 que encontra o envelope superior/inferior de n retas em tempo $O(n \log n)$, podemos encontrar o fecho convexo de n pontos com a mesma complexidade de tempo. A seguir vamos explicar a cinetização do algoritmo que encontra o envelope, mostrando que ela constitui uma boa estrutura de dados cinética para o problema do fecho convexo.

5.1 Certificados para o Envelope Superior de Duas Cadeias

Como os objetos primários do problema são os pontos do plano primal que são associados com retas no plano dual, vamos manter a mesma nomenclatura para as retas do plano dual. Assim, os pontos a e b do plano primal são levados às retas a e b no plano dual e a sua intersecção produz um vértice no plano dual ao qual chamaremos vértice ab.

O Algoritmo 2.1 introduz o conceito de *cadeia* durante a sua implementação (veja a Seção 2.3). Uma cadeia é representada por uma lista duplamente ligada que descreve um envelope da esquerda para a direita. Ela é composta dos vértices do envelope, cada um associado à reta imediatamente à sua direita. Dessa forma, considerando o exemplo fornecido pela Figura 5.1, temos que a cadeia que descreve o envelope superior possui a seguinte descrição: $\langle -\infty[b], ba[a], ac[c], cg[g] \rangle$. O algoritmo que calcula o envelope superior resolve o problema recursivamente e, a cada nível da recursão há a combinação de duas cadeias produzindo uma nova cadeia que representa o envelope superior das duas cadeias.

Na fase de combinação o algoritmo faz uma varredura da esquerda para a direita nas duas cadeias simultaneamente, obtendo a nova cadeia. Os pontos de parada são os vértices de cada cadeia e, assumindo que não existem vértices com mesma coordenada x

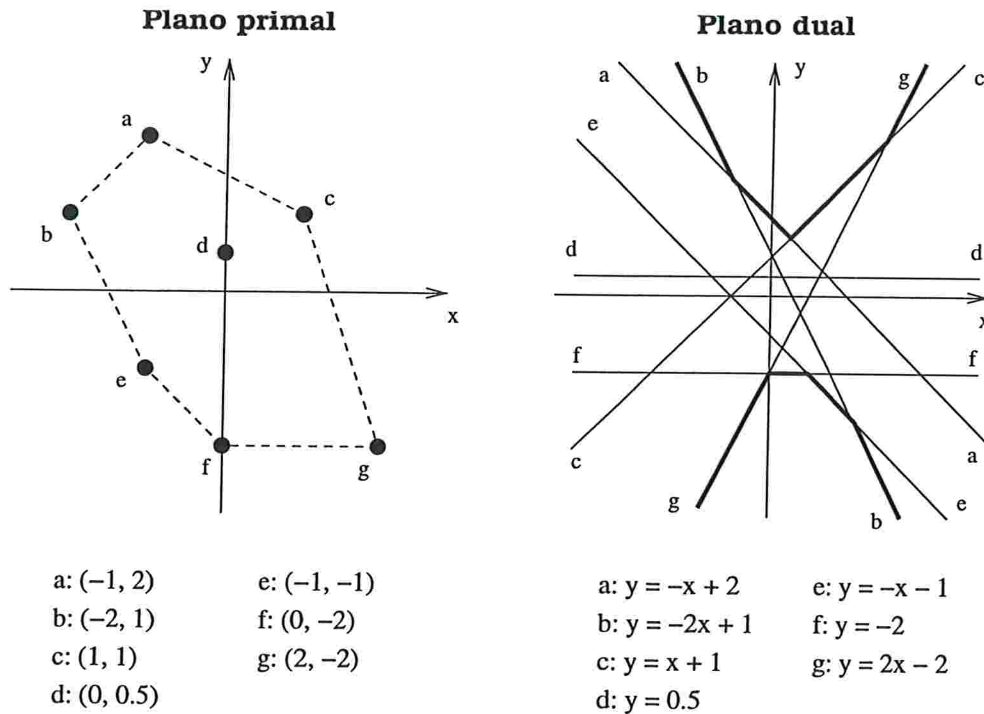


Figura 5.1: A parte de cima do fecho convexo corresponde ao envelope superior $\langle b, a, c, g\rangle$ no plano dual. Analogamente a parte de baixo do fecho corresponde ao envelope inferior $\langle g, f, e, b\rangle$.

em cadeias diferentes, para cada vértice a linha de varredura deve interceptar uma reta na outra cadeia. Para um vértice ab, a reta da outra cadeia que está acima ou abaixo de ab é chamada *reta inversa* (*contender edge*) de ab, conforme mostra a Figura 5.2, sendo denotada como $ce(ab)$. Para a cinetização do algoritmo vamos precisar que cada vértice possua um apontador para a sua reta inversa. Também vamos precisar que cada vértice aponte para o próximo ponto de parada (*next*) e para o anterior (*prev*). O próximo ponto de parada pode ser o próximo vértice da mesma cadeia ou da outra cadeia. Nesta seção vamos considerar apenas duas cadeias e apresentar uma estrutura de dados cinética que mantém a cadeia que representa o envelope superior das duas cadeias de entrada.

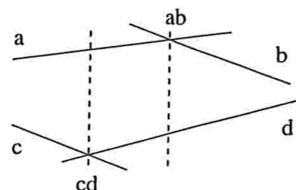


Figura 5.2: A reta inversa do vértice ab é d. A reta inversa do vértice cd é a.

<table>
<tr><th>Tipo</th><th>Comparação</th><th>Condições</th></tr>
<tr><td>$x[ab]$</td><td>$[ab <_x cd]$</td><td>$cd = next(ab)$ e
ab e cd estão em cadeias diferentes</td></tr>
<tr><td>$yli[ab]$</td><td>$[ab <_y ce(ab)]$ ou
$[ab >_y ce(ab)]$</td><td>b cruza $ce(ab)$</td></tr>
<tr><td>$yri[ab]$</td><td>$[ab <_y ce(ab)]$ ou
$[ab >_y ce(ab)]$</td><td>a cruza $ce(ab)$</td></tr>
<tr><td>$yt[ab]$</td><td>$[ce(ab) <_y ab]$</td><td rowspan="3">$a <_s ce(ab)$ e
$ce(ab) <_s b$ e
$ce(ab) <_y ab$</td></tr>
<tr><td>$slt[ab]$</td><td>$[a <_s ce(ab)]$</td></tr>
<tr><td>$srt[ab]$</td><td>$[ce(ab) <_s b]$</td></tr>
<tr><td>$sl[ab]$</td><td>$[b <_s ce(ab)]$</td><td>$b <_s ce(ab)$ e
$ab <_y ce(ab)$ e
ab e $next(ab)$ estão em cadeias diferentes</td></tr>
<tr><td>$sr[ab]$</td><td>$[ce(ab) <_s a]$</td><td>$ce(ab) <_s a$ e
$ab <_y ce(ab)$ e
ab e $prev(ab)$ estão em cadeias diferentes</td></tr>
</table>

Tabela 5.1: Tipos de certificados necessários para se manter o envelope superior de duas cadeias.

As comparações realizadas na fase de combinação das duas cadeias produzem dois tipos de certificados: x-certificados provando a ordem horizontal dos vértices (as comparações em x serão denotadas por $<_x$) e y-certificados atestando a posição vertical de um vértice com respeito à sua reta inversa (comparações denotadas por $<_y$). Porém, se armazenássemos todas essas comparações em y como certificados, a estrutura de dados cinética obtida não seria local, pois uma certa reta poderia ser inversa de um número linear de vértices da outra cadeia (imagine uma cadeia formada por uma única reta e outra formada por vários vértices e retas). Então não vamos armazenar todos estes certificados. Temos, então, que contar com uma lista de certificados alternativa que também considere comparações entre as inclinações das retas (*slopes*). Comparações entre inclinações serão denotadas por $<_s$.

A Tabela 5.1 lista todos os tipos de certificados que constituem a prova da corretude do Algoritmo 2.1. A primeira coluna mostra o nome do certificado, a segunda mostra a comparação que esse certificado comprova e a terceira expõe as condições para que tal certificado esteja presente na estrutura de dados cinética de forma que ela seja uma estrutura local. Isso será melhor detalhado mais adiante.

A primeira linha da tabela diz que um certificado do tipo $x[ab]$ é armazenado na estrutura de dados cinética somente quando os vértices ab e cd são vizinhos na ordem horizontal e estão em cadeias diferentes. Nesse caso a comparação certifica a ordenação horizontal entre ab e cd. Não é necessário certificar a ordenação horizontal entre vértices da mesma cadeia.

A segunda e terceira linhas da tabela dizem respeito aos certificados que atestam as intersecções entre as cadeias: `yli[...]` e `yri[...]` (y left/right intersection). Tais certificados somente existirão quando ocorrer uma intersecção entre retas de cadeias diferentes. À cada instante, o conjunto dos certificados `yli[...]` e `yri[...]` deve cobrir todas as intersecções entre cadeias na produção do envelope superior naquele instante.

Caso uma reta $ce(ab)$ não faça parte do envelope superior (considerando-se a coordenada x do vértice ab de parada na varredura), teremos três certificados atestando que existe uma espécie de "tangência" da cadeia que contém ab (certificados `yt[...]`, `slt[...]` e `srt[...]`) ou um certificado provando que não exista tal tangência (`sl[...]` ou `sr[...]`). Todos os tipos de certificados listados pela Tabela 5.1 aparecem na Figura 5.3.

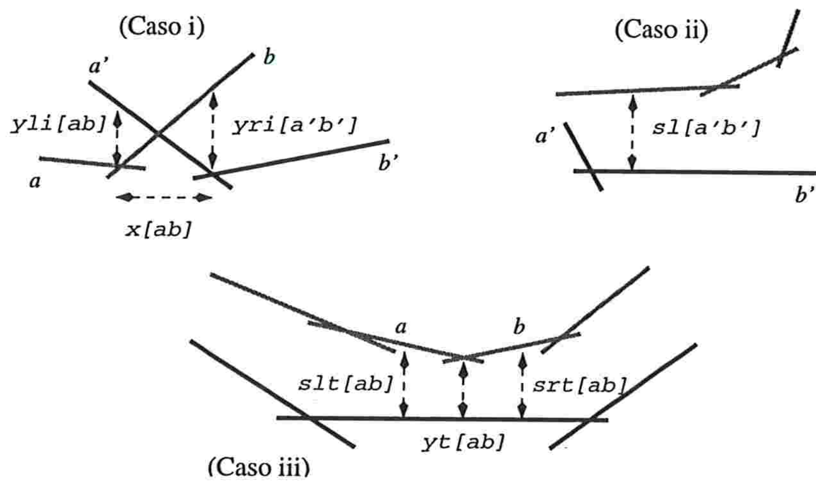


Figura 5.3: Dependendo das posições relativas das duas cadeias, diferentes certificados são usados para atestar intersecções (caso i) ou ausência de intersecção (casos ii e iii).

Lema 5.2 *A lista de certificados apresentada é suficiente para a manutenção do envelope superior.*

Prova: Seja $\mathcal{C}$ uma configuração cuja lista de certificados é $\mathcal{L}$ e seja $\mathcal{C}'$ uma outra configuração para a qual a lista de certificados $\mathcal{L}$ também vale. Devemos mostrar que a cadeia que descreve o envelope superior em $\mathcal{C}$ é exatamente a mesma que a que descreve o envelope superior em $\mathcal{C}'$.

Todas as intersecções em $\mathcal{C}$ são intersecções em $\mathcal{C}'$. Isso é forçadamente verdade devido aos certificados do tipo `yli[...]` e `yri[...]` e aos x-certificados. Os x-certificados certificam a ordenação horizontal e provam a corretude dos apontadores para as retas inversas de cada vértice. Além disso, qualquer vértice que possua um y-certificado em $\mathcal{L}$ deverá estar em uma posição relativa semelhante tanto em $\mathcal{C}$ quanto em $\mathcal{C}'$.

Vamos assumir agora que exista uma intersecção i em $\mathcal{C}'$ que não esteja em $\mathcal{C}$. Suponha também que em $\mathcal{C}$, a cadeia C_a está acima da cadeia C_b no ponto que possui

coordenada x igual à coordenada x de i. Considere a porção completa do envelope superior em $\mathcal{C}$ na região que contém i. Como devem existir y-certificados nas duas extremidades dessa região, a cadeia C_a deverá estar acima da cadeia C_b nas duas pontas da região considerada. Conclui-se que tanto em $\mathcal{C}$ quanto em $\mathcal{C}'$, C_a deve estar acima de C_b nas duas pontas da região. Então deve existir um número par de intersecções nessa região em $\mathcal{C}'$, pois uma intersecção entre as duas cadeias faz C_a passar para baixo de C_b e é necessário que exista outra intersecção na região para que C_a volte a ficar por cima de C_b.

Considere, então, duas intersecções entre as quais a cadeia C_b esteja acima da cadeia C_a e suponha que a seqüencia de retas da cadeia C_a nessa região seja $a_1, a_2, \ldots, a_j$ e que a seqüência de retas da cadeia C_b seja $b_1, b_2, \ldots, b_k$, incluindo as arestas que se interceptam. Temos que $b_1 <_s a_1$ e $a_j <_s b_k$. Então existe pelo menos um vértice na região onde a ordem relativa das inclinações entre as duas cadeias se inverte. Suponha que o vértice mais à esquerda onde isso ocorre seja um vértice da cadeia C_a e a sua reta inversa é b_l. Em $\mathcal{C}$ a reta b_l está inteiramente abaixo da cadeia C_a. Então existe um certificado em $\mathcal{C}$ do tipo `sl[...]` (ou `sr[...]`) que prova que a reta b_l tem menor (ou maior) inclinação que a reta correspondente na cadeia C_a, ou existe um certificado tangente em $\mathcal{C}$ (do tipo `yt[...]`) que prova que b_l está abaixo da cadeia C_a. Em ambos os casos, $\mathcal{C}'$ não pode ter tais certificados, contrariando a hipótese. □

Lema 5.3 *A localidade da lista de certificados para a manutenção do envelope superior é $O(1)$.*

Prova: Considere uma reta a. Ela estará envolvida em um certificado porque um dos seus vértices extremos estará. Mas cada vértice pode estar envolvido em somente dois certificados do tipo `x[...]` (um para a esquerda e outro para a direita) e em um dos outros tipos.

Se a é interceptada pela outra cadeia, ela estará envolvida em certificados do tipo `yli[...]` e `yri[...]` como reta inversa. Porém, para cada reta nessas condições existe apenas um certificado de cada um desses tipos.

Suponha que a seja a reta inversa de muitos vértices. Esses vértices todos têm de estar na outra cadeia. Assim, somente participarão da lista de certificados os que envolvem a com os vértices mais à esquerda e mais à direita dentre os considerados.

No caso em que a não intercepta a outra cadeia, ela pode estar envolvida em no máximo duas triplas de certificados de tangência (`yt[...]`, `slt[...]` e `srt[...]`) ou, conforme o parágrafo anterior, em no máximo dois certificados de inclinação (`sl[...]` e `sr[...]`). Assim a localidade da lista de certificados para duas cadeias é $O(1)$. □

5.2 Tratamento de Eventos

O Lema 5.2 diz que a lista de certificados definida na seção anterior é suficiente para mantermos o envelope superior de duas cadeias. Como acontece em toda estrutura de dados cinética, os certificados devem ser colocados em uma fila de eventos e cada certificado tem um tempo de vida calculado no instante em que é criado, com base no movimento dos objetos que o compõem. Quando um certificado atinge o seu prazo de validade, ocorre um evento que deve ser tratado para que a estrutura continue correta. O tratamento de um evento requer atualizações na lista de certificados, podendo ocasionar alterações em certificados existentes. Dependendo do tipo do certificado responsável pelo evento em questão, diferentes ações são necessárias. A seguir são exibidos algoritmos que se responsabilizam pelo tratamento de cada tipo de evento possível sobre a lista de certificados descrita na seção anterior. Uma descrição ilustrativa aparece na Figura 5.4.

O tratamento para eventos correspondentes a certificados dos tipos $yri\,[ab]$, $srt\,[ab]$ e $sr\,[ab]$ é simétrico ao dos tipos $yli\,[ab]$, $slt\,[ab]$ e $sl\,[ab]$ respectivamente.

Algoritmo 5.1 TrataEvento $x\,[ab]$

Entrada: evento causado por um certificado do tipo $x\,[ab]$.
Saída: a lista de certificados e o envelope superior devidamente atualizados.

$cd \leftarrow next(ab)$
`AtualizaCertHorizontal`$(ab,\ cd)$
`AtualizaCertInterseccao`$(ab,\ cd)$
{atualização dos certificados de inclinação se ab está abaixo de d}
se $ab <_s d$ **então**
  `AtualizaCertInclinacao`$(ab,\ cd)$
{tratamento simétrico se cd está abaixo de a}
se $cd <_s a$ **então**
  `AtualizaCertInclinacao`$(cd,\ ab)$

Algoritmo 5.2 AtualizaCertInterseccao

Entrada: vértices ab e cd responsáveis por um evento horizontal.
Saída: os certificados referentes a intersecções devidamente atualizados.

se existe $yri\,[ab]$ **então**
  remova $yri\,[ab]$
  crie $yri\,[cd]$
se existe $yli\,[cd]$ **então**
  remova $yli\,[cd]$
  crie $yli\,[ab]$

Algoritmo 5.3 AtualizaCertHorizontal

Entrada: vértices ab e cd responsáveis por um evento horizontal.
Saída: a lista horizontal de vértices e os certificados correspondentes atualizados.

$criaPrev \leftarrow$ **true**
$criaNext \leftarrow$ **true**
remova $x[ab]$ e **crie** $x[cd]$
se existe $x[prev(ab)]$ **então**
  remova $x[prev(ab)]$
  $criaPrev \leftarrow$ **false**
se existe $x[next(ab)]$ **então**
  remova $x[next(ab)]$
  $criaNext \leftarrow$ **false**
`AtualizaLista`(ab, cd)
se $criaPrev$ **então**
  crie $x[prev(cd)]$
se $criaNext$ **então**
  crie $x[next(cd)]$

Algoritmo 5.4 AtualizaLista

Entrada: vértices ab e cd responsáveis por um evento horizontal.
Saída: a lista ordenada atualizada

$prev(cd) \leftarrow prev(ab)$
$next(ab) \leftarrow next(cd)$
$next(cd) \leftarrow ab$
$prev(ab) \leftarrow cd$
$ce(ab) \leftarrow d$
$ce(cd) \leftarrow a$

Algoritmo 5.5 AtualizaCertInclinacao

Entrada: vértices ab e cd responsáveis por um evento horizontal, com ab abaixo de d.
Saída: os certificados referentes a inclinações devidamente atualizados.

se existe $yt\,[cd]$ **então**
  remova $yt\,[cd]$
  remova $slt\,[cd]$
  remova $srt\,[cd]$
se existe $sl\,[ab]$ **então**
  remova $sl\,[ab]$
se cadeia de $ab \neq$ cadeia de $next(ab)$ e $b <_s d$ **então**
  crie $sl\,[ab]$
se cadeia de $prev(cd) \neq$ cadeia de cd e $prev(cd) <_y ce(prev(cd))$ e $a <_s c$ **então**
  crie $sl\,[prev(cd)]$
senão se existe $sr\,[ab]$ **então**
  se $a <_s d$ **então**
    remova $sr\,[ab]$
    crie $yt\,[cd]$
    crie $slt\,[cd]$
    crie $srt\,[cd]$
  senão
    faça $sr\,[ab]$ apontar para novo $ce(ab)$

Algoritmo 5.6 TrataEvento $yli\,[ab]$

Entrada: evento causado por um certificado do tipo $yli\,[ab]$.
Saída: a lista de certificados e o envelope superior devidamente atualizados.

remova $yri\,[next(ab)]$
remova $yli\,[ab]$
se existe $yri\,[ab]$ **então**
  remova $yli\,[prev(ab)]$
  remova $yri\,[ab]$
  crie $slt\,[ab]$
  crie $srt\,[ab]$
  crie $yt\,[ab]$
  remova $ce(ab)$ do envelope superior
senão
  crie $yri\,[ab]$
  crie $yli\,[prev(ab)]$
  insira a no envelope superior

Algoritmo 5.7 TrataEvento $yt\,[ab]$

Entrada: evento causado por um certificado do tipo $yt\,[ab]$.
Saída: a lista de certificados e o envelope superior devidamente atualizados.

remova $yt\,[ab]$
remova $slt\,[ab]$
remova $srt\,[ab]$
crie $yli\,[ab]$
crie $yri\,[next(ab)]$
crie $yri\,[ab]$
crie $yli\,[prev(ab)]$
insira $ce(ab)$ no envelope superior

Algoritmo 5.8 TrataEvento $slt\,[ab]$

Entrada: evento causado por um certificado do tipo $slt\,[ab]$.
Saída: a lista de certificados e o envelope superior devidamente atualizados.

remova $yt\,[ab]$
remova $slt\,[ab]$
remova $srt\,[ab]$
$cd \leftarrow prev(ab)$
se cadeia de d = cadeia de a **então**
  crie $yt\,[cd]$
  crie $slt\,[cd]$
  crie $srt\,[cd]$
senão
  crie $sl\,[cd]$

Algoritmo 5.9 TrataEvento $sl\,[ab]$

Entrada: evento causado por um certificado do tipo $sl\,[ab]$.
Saída: a lista de certificados e o envelope superior devidamente atualizados.

remova $sl\,[ab]$
$cd \leftarrow next(ab)$
se cadeia de $cd \neq$ cadeia de ab **então**
  crie $yt\,[cd]$
  crie $slt\,[cd]$
  crie $srt\,[cd]$
senão se cadeia de $ce(cd) \neq$ cadeia de $ce(ab)$ **então**
  crie $sr\,[cd]$

Considere, por exemplo, um evento causado pela perda de validade de um certificado do tipo $\mathtt{slt}[ab]$. Trata-se de um certificado de inclinação relacionado a uma "tangência" da cadeia que contém ab em relação à outra. Devemos executar o Algoritmo 5.8 para tomar conta desse evento. Aqui uma reta que possuía coeficiente angular negativo assume um novo coeficiente angular (positivo). Essa troca de sinal no valor do coeficiente angular é o que caracteriza o evento. Primeiramente devemos remover o certificado inválido, além de seus "companheiros de tangência" $\mathtt{yt}[ab]$ e $\mathtt{srt}[ab]$. Agora devemos testar se o ponto de parada cd anterior a ab está na mesma cadeia ou não. Se estiver na mesma cadeia, então estamos no caso (iv) da Figura 5.4 (da esquerda para a direita) e devemos criar um novo trio de certificados para a nova tangência criada em cd. Caso cd e ab estejam em cadeias diferentes, então estamos lidando com o caso (v) da Figura 5.4 (também da esquerda para a direita) e a tangência que existia antes deixa de existir. Nesse caso um certificado de inclinação do tipo $\mathtt{sr}[cd]$ deve ser adicionado à lista de eventos. Repare que neste tipo de evento a cadeia que descreve o envelope superior produzido pelas duas cadeias não é alterado.

Como mais um exemplo vamos considerar um evento que altere a descrição do envelope superior. Considere o evento $\mathtt{yt}[ab]$, que corresponde ao caso (iii) da Figura 5.4 (da direita para a esquerda). Esse evento é caracterizado pela destruição de uma "tangência" devido ao vértice de tangência trocar de lado com a sua reta inversa. Nesse caso a tangência é substituída por duas intersecções entre as cadeias, substituindo três certificados por outros quatro (veja o Algoritmo 5.7). Além de tudo, uma nova reta é introduzida à cadeia que descreve o envelope superior exatamente no local onde estava o vértice ab.

Analisando os algoritmos para o tratamento de cada tipo de evento, podemos notar que cada evento pode ser resolvido em tempo constante pois basta que se façam alterações locais na lista de certificados. As inserções/remoções de novos objetos na cadeia que descreve o envelope superior devem ser implementadas para executar também em tempo constante. Em geral uma cadeia será implementada como uma lista duplamente ligada de vértices, cada vértice com um campo para a reta à sua direita. Inserções e remoções nesta lista ligada podem ser implementadas para execução em tempo constante se cada certificado armazenar apontadores para as posições da lista ligada ocupadas pelos objetos envolvidos em tal certificado. Assim, o tempo consumido no processamento de um evento é $O(1)$.

5.3 Cinetizando o Envelope Superior Global

Para cinetizar o algoritmo divisão-e-conquista que obtém o envelope superior de um arranjo de n retas em tempo $O(n \log n)$ (o Algoritmo 2.1) precisamos manter uma árvore binária balanceada que conta a história da execução do algoritmo. Cada nó dessa árvore corresponde a uma fase de conquista da execução do algoritmo cuja divisão

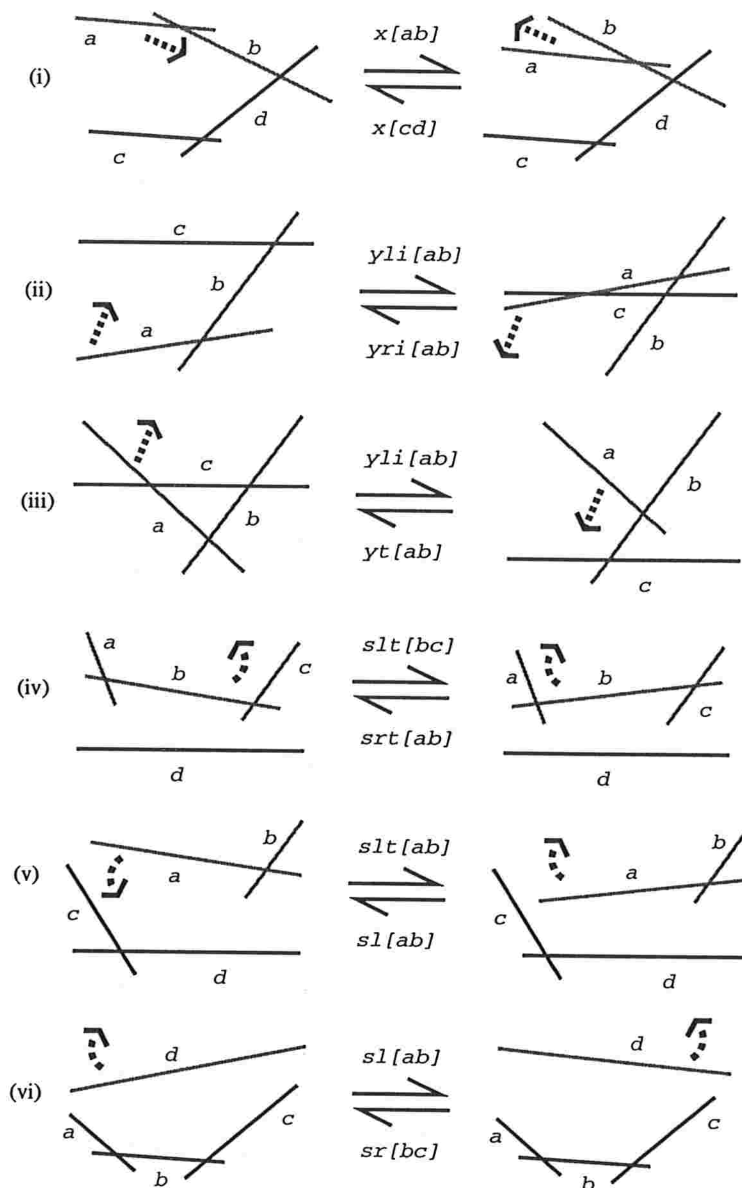


Figura 5.4: Alguns dos eventos que podem ocorrer durante a manutenção do envelope superior. O certificado responsável pelo evento é indicado em cada transição. Existem dois casos não mostrados que correspondem aos casos (ii) e (v) espelhados.

particionou o arranjo de retas entre os dois filhos desse nó. A árvore conta a história da recursão aplicada ao algoritmo e cada nó contém a computação de uma cadeia que descreve o envelope superior de outras duas cadeias (que vêm como resposta dos dois filhos do nó considerado). O envelope superior de todo o arranjo é obtido da cadeia produzida na raiz dessa árvore.

Caso um evento cause uma mudança no envelope de um dado nó, esse nó deve

passar o evento para o nó pai como uma mudança local na estrutura. O mesmo evento é então repassado para cima na árvore, ancestral por ancestral, até que não precise mais ser propagado ou chegue até a raiz. Conforme o Lema 5.3, cada nó possui uma sub-estrutura de dados cinética com localidade $O(1)$. Assim a estrutura toda para manter o envelope superior de um arranjo de n retas possui localidade $O(\log n)$ pois a árvore de recursão é balanceada.

A estrutura também é de resposta rápida, pois o custo de se processar um evento em cada um dos $O(\log n)$ nós no caminho de propagação de um evento pela árvore, é $O(1)$, o que resulta em uma estrutura de dados cinética cujo tempo de processamento de um evento é $O(\log n)$.

Podemos analisar a eficiência da estrutura tomando o tempo como uma terceira dimensão estática e extrapolando cada evento para que ele figure como uma estrutura tridimensional com resultados conhecidos de complexidade. A estrutura descrita pelo envelope superior de um arranjo de linhas através do tempo forma exatamente o envelope superior de um arranjo de superfícies [Sh94, HS94]. Em [BGH97, Ba99] usam-se resultados que provam a complexidade "aproximadamente quadrática" para o envelope superior de superfícies algébricas [Sh94, At85]. Também se faz uso de resultados recentes sobre a complexidade "aproximadamente quadrática" da composição de dois envelopes superiores para obter limitantes elegantes no número de eventos para x-certificados. Esses resultados são usados para se provar que o número total de eventos na estrutura é $O(n^{2+\epsilon})$ para qualquer $\epsilon > 0$ em um (δ, n)-cenário e que a estrutura é compacta.

5.4 Manutenção do Fecho Convexo

Finalmente podemos voltar ao nosso problema original que é manter eficientemente ao longo do tempo o fecho convexo de um conjunto de n pontos em movimento no plano. O problema de se encontrar o fecho convexo pode ser dividido em encontrarmos a parte de baixo e a parte de cima do fecho. Através da dualidade entre ponto e reta, podemos reduzir o problema de se encontrar a parte de cima do fecho convexo de n pontos ao problema de se obter o envelope superior de n retas. A parte de baixo é encontrada simetricamente através do envelope inferior. Através de um algoritmo divisão-e-conquista é possível obter esse envelope superior em tempo $O(n \log n)$. O envelope inferior é obtido através de um algoritmo simetricamente idêntico. Portanto podemos encontrar o fecho convexo em tempo $O(n \log n)$.

Nas seções anteriores vimos que o algoritmo que encontra o envelope superior admite uma boa cinetização segundo o modelo cinético. Para que os mesmos resultados valham também para o problema do fecho convexo, devemos mostrar que a estrutura continua eficiente. De acordo com [BGH97, Ba99] o número de eventos possíveis na estrutura é $O(n^{2+\epsilon})$. Outro resultado mostra que, no pior caso, o fecho convexo de n pontos em movimento regido por polinômios de grau limitado muda $\Theta(n^2)$ vezes [AGHV97]. Há

um fator de eficiência para a estrutura de dados cinética do fecho convexo de $O(n^\epsilon)$. Como ϵ é pequeno, a estrutura obtida para a manutenção do fecho convexo é eficiente segundo o modelo cinético.

Assim, o fecho convexo pode ser mantido através do tempo de forma eficiente utilizando-se a estrutura de dados cinética desenvolvida para o envelope superior. Basta que tenhamos, então, duas estruturas de dados cinéticas mantidas separadamente: uma mantém o envelope superior e outra mantém o envelope inferior. O fecho convexo é obtido então através da concatenação entre as listas que descrevem os dois envelopes. Para que a resposta final seja montada eficientemente, tais listas devem estar armazenadas em estruturas em forma de árvores balanceadas ao invés de listas circulares. Uma sugestão vista em [Ba99] é utilizar árvores dinâmicas, pois dessa forma a estrutura também se mostra capaz de oferecer suporte a consultas de localização de ponto.

5.5 Problemas Correlatos

Existem outros problemas cineticos em Geometria Computacional que podem ser resolvidos através da estrutura de dados cinética desenvolvida para o fecho convexo. Nesta seção vamos apresentar dois problemas que descrevem o comportamento e a localização de um conjunto de pontos, como o problema do par de pontos mais distante e o problema da largura de um conjunto de pontos. Também veremos que a estrutura para a manutenção desses atributos geométricos está intimamente relacionada com a estrutura do fecho convexo. Conseqüentemente a estrutura de dados cinética apresentada para tais problemas também será baseada na estrutura utilizada na manutenção do fecho convexo.

5.5.1 Par de Pontos Mais Distante

Problema 5.4 (Problema do Par de Pontos Mais Distante) *Dado um conjunto S de n pontos em movimento no plano, deseja-se saber a todo instante qual o par de pontos (p, q) de S cuja distância entre p e q é a maior possível.*

Através do par de pontos (p, q) mais distante de um conjunto, cuja distância é $d(p, q)$, podemos facilmente gerar um quadrado de lado $d(p, q) + \epsilon$ que compreende todos os pontos do conjunto de pontos considerado. A versão estática desse problema tem várias aplicações em técnicas de Computação Gráfica como recorte, ou *clipping*, em cenas. Nessas técnicas tipicamente estamos interessados em obter um retângulo que compreenda todo o conjunto de objetos de interesse para destacá-los do restante da cena. Considerando um conjunto de pontos em movimento, aplicações equivalentes seriam de recorte em cenas em movimento que podem ser utilizadas em realidade virtual. A versão cinética do problema também encontra aplicações em detecção de colisão entre pontos e entre polígonos.

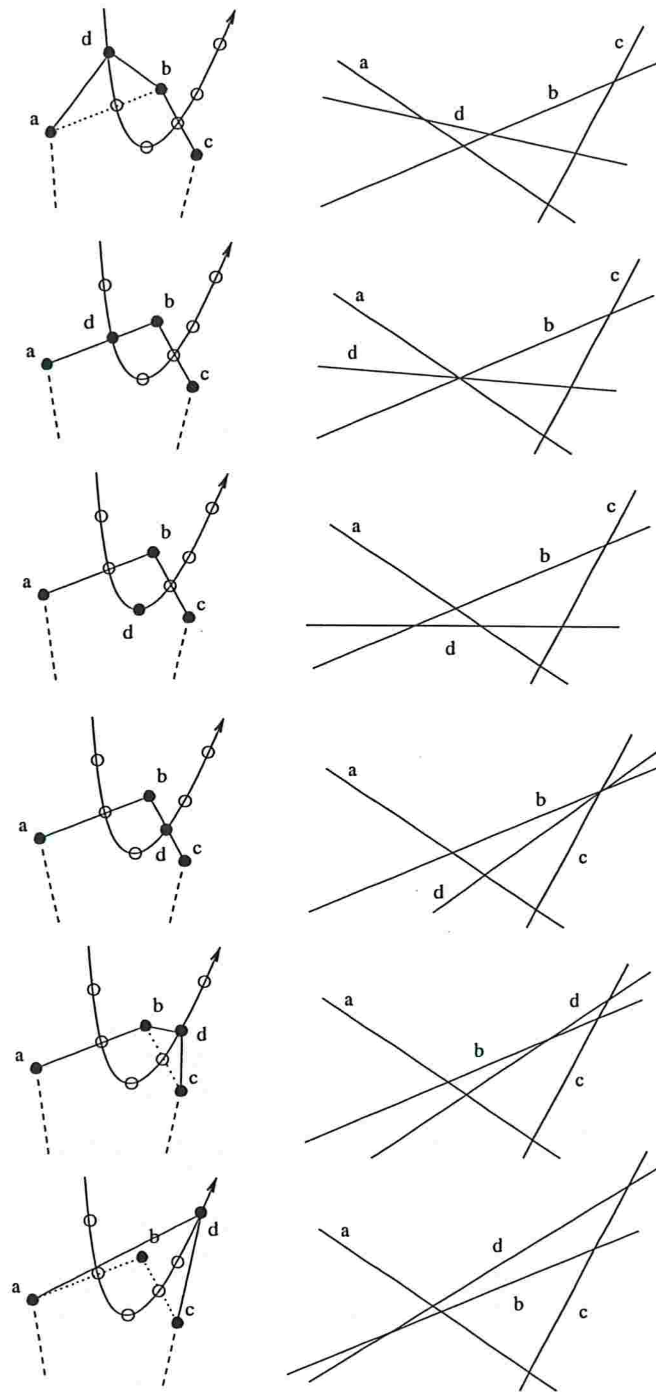


Figura 5.5: Um vértice (d) se movimenta rumo ao interior do fecho convexo e sai por outro local. À direita é possível acompanhar as mudanças correspondentes no envelope superior do plano dual.

Outra aplicação é encontrada na redução da complexidade de algoritmos espaciais. Ao invés de trabalharmos com uma entidade geométrica complicada, muitas vezes é muito melhor aproximarmos essa entidade por uma caixa que englobe todos os pontos que formam a entidade geométrica. No caso bidimensional, podemos aproximar uma figura complicada por um quadrado. O tamanho dos lados desse quadrado pode ser dado pela distância entre o par de pontos mais distante do conjunto de pontos considerado.

Como os pontos estão movendo-se continuamente, a distância do par de pontos mais distante também se altera continuamente, mas, como em todos os casos considerados, a descrição combinatória do par somente é alterada em instantes especiais que constituem os eventos.

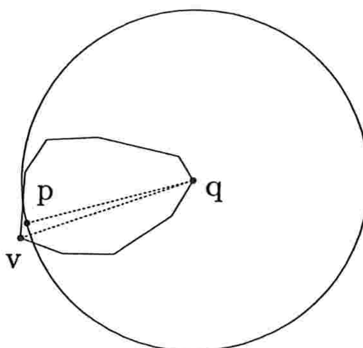


Figura 5.6: O par mais distante é formado por dois vértices do fecho convexo

Lema 5.5 *O par de pontos mais distante de um conjunto S de n pontos é formado por vértices do fecho convexo de S.*

Prova: Seja (p, q) o par de pontos mais distante em S. Suponha que p não é um vértice do fecho convexo, sendo interior ao fecho convexo. Devemos mostrar que existe uma diagonal no fecho convexo cujo comprimento é maior que $d(p, q)$. Aqui estamos supondo que um lado do fecho também pode ser considerado uma diagonal.

Considere o círculo centrado em q com raio $d(p, q)$. Esse círculo deve compreender todos os pontos de S, pois (p, q) é o par de pontos mais distante de S. Porém, como p é interno ao fecho convexo, pelo menos um vértice v do fecho convexo fica de fora do círculo considerado, conforme mostra a figura Figura 5.6. Portanto, a diagonal entre q e v possui comprimento igual a $d(q, v)$ que é maior que $d(p, q)$. Dessa forma, (p, q) não pode ser o par de pontos mais distante em S. □

De acordo com o Lema 5.5, o par mais distante é dado por vértices do fecho convexo. Uma solução natural para a versão estática do problema consiste então em encontrar o fecho convexo e descobrir qual o par de vértices do fecho com maior distância. Porém,

nem todo par de vértices do fecho convexo é candidato a ser esse par de pontos, apenas os chamados pares de pontos *opostos* (*antipodals*).

Tome dois vértices em extremos opostos do fecho convexo, como o vértice mais à esquerda v_l e o mais à direita v_r. A *cadeia superior* do fecho convexo é formada pelas arestas e vértices da fronteira do fecho que limitam S superiormente, ligando v_l a v_r. A outra cadeia que conecta v_l a v_r limitando S inferiormente é a *cadeia inferior*. Um par de pontos é considerado um *par oposto* se um dos pontos é vértice da cadeia superior e o outro da cadeia inferior e os dois vértices admitem linhas suporte paralelas. A intersecção da linha suporte de um vértice com o fecho convexo deve conter o vértice e, eventualmente, uma aresta do fecho convexo (conseqüentemente contendo também um outro vértice vizinho ao vértice considerado, servindo também de reta suporte para esse segundo vértice). Um desenho das cadeias que mostra também a idéia de pares opostos e linhas suporte está na Figura 5.7. Note que v_l e v_r devem pertencer às duas cadeias ao mesmo tempo.

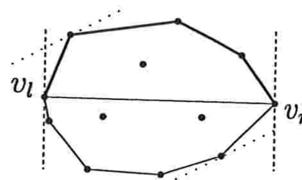


Figura 5.7: Cadeias superior e inferior do fecho convexo e vértices opostos com suas linhas suporte externas ao fecho convexo.

A cadeia superior é descrita por uma lista de vértices que começa em v_l e vai até v_r. Já a cadeia inferior é descrita por outra lista de vértices que vai no sentido contrário, de v_r a v_l. Note que a concatenação das duas cadeias dá a descrição do fecho convexo e isso não é por acaso, pois as duas cadeias correspondem exatamente aos envelopes superior e inferior no plano dual. De posse das cadeias inferior e superior, é possível enumerar todos os pares opostos através de uma varredura na fronteira do fecho que pode ser feita em tempo linear. Para encontrar o par de pontos mais distante devemos analisar as distâncias entre os pares opostos, identificando aquele com maior distância.

Podemos resolver o problema diretamente no plano dual. As cadeias superior e inferior são mapeadas para seus respectivos envelopes. Cada vértice converte-se em uma reta e cada aresta do fecho transforma-se em um vértice no plano dual conforme descrito no Capítulo 2. Uma varredura através do eixo x no plano dual é suficiente para percorrermos todos os pares opostos, um a cada intervalo definido pela dualidade das linhas suporte de cada vértice. A Figura 5.8 mostra os pares opostos no plano dual para o exemplo da Figura 5.1. Repare que o par formado pelo vértice mais à esquerda e o vértice mais à direita aparece duas vezes porque ambos os vértices pertencem aos extremos das duas cadeias.

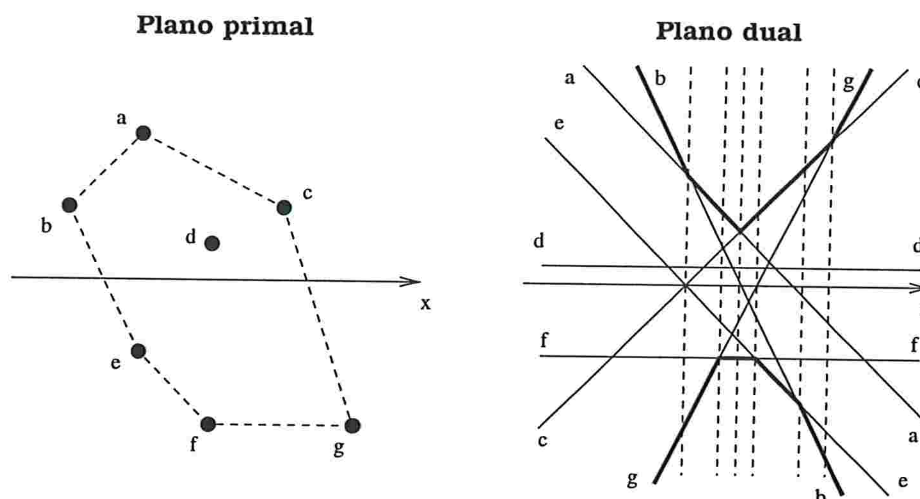


Figura 5.8: Os pares opostos aparecem no plano dual em seqüência quando fazemos uma varredura no eixo x. Cada intervalo entre as linhas verticais tracejadas corresponde a um par oposto. Os pares opostos na figura são: (b, g), (a, g), (a, f), (c, f), (c, e), (b, c) e novamente (b, g).

O algoritmo que resolve o problema estático, encontrando o par de pontos mais distante de um conjunto S de n pontos é exibido no Algoritmo 5.10. O algoritmo utiliza algumas sub-rotinas tais como `Dualiza` que recebe um conjunto de pontos e devolve um conjunto de retas duais; `EnvelopeSuperior` que nada mais é que o Algoritmo 2.1 que retorna o envelope superior de um conjunto de retas; `EnvelopeInferior` é o procedimento simétrico à `EnvelopeSuperior`; `ProximaParada` retorna o vértice da cadeia inferior ou superior imediatamente à direita de um dado ponto, e `SelMaiorDistancia` seleciona o par com maior distância dentre um conjunto de pares de vértices (considerando os vértices no plano primal). Finalmente, `next` recebe uma reta em um dos envelopes e retorna a próxima reta na descrição do mesmo envelope.

O processo de dualização consome tempo $O(n)$, pois precisamos apenas mapear cada ponto em uma reta dual baseado nas suas próprias coordenadas. Cada envelope pode ser encontrado em tempo $O(n \log n)$ utilizando o Algoritmo 2.1. A varredura através dos pares opostos é realizada em tempo linear, bem como a seleção do par mais distante, pois temos no máximo $O(n)$ pares opostos. Portanto o par de pontos mais distante de um conjunto de n pontos em um cenário estático pode ser obtido em tempo $O(n \log n)$.

Agarwal, Guibas, Hershberger e Veach [AGHV97] descreveram uma estrutura de dados cinética para problema do par mais distante de pontos em movimento cinetizando o algoritmo estático descrito acima. A estrutura faz uso da ordenação pela coordenada x entre os vértices das cadeias superior e inferior que atestam a ordenação entre os pares opostos. Essa ordenação é armazenada em uma lista duplamente ligada distribuída em uma árvore binária balanceada, podendo ser atualizada em tempo $O(\log n)$ mais tempo

Algoritmo 5.10 ParMaisDistante

Entrada: um conjunto S de pontos no plano.
Saída: o par de pontos mais distante.

```
D ← Dualiza(S)
E_sup ← EnvelopeSuperior(D)
E_inf ← EnvelopeInferior(D)
s ← inicio(E_sup)
i ← inicio(E_inf)
P ← (s, i)
x ← −∞
enquanto ProximaParada(x) < +∞ faça
  x ← ProximaParada(x)
  se x está em E_sup então
    s ← next(s)
  senão
    i ← next(i)
  P ← P ∪ (s, i)
p ← SelMaiorDistancia(P)
retorna p
```

proporcional ao número de pares opostos afetados a cada evento sobre a lista ordenada. A procura pelo par mais distante dentre os pares opostos pode ser realizada por um torneio cinético.

Os eventos podem ocorrer em três camadas: a primeira camada diz respeito aos eventos sobre o torneio cinético realizado entre os pares opostos. Esses eventos são tratados em tempo $O(\log n)$ conforme visto na Seção 3.3.

A segunda camada de eventos diz respeito à ocorrência de um evento sobre a lista ordenada, que significa que dois vértices inverteram suas posições na lista ordenada. Nesse caso, três certificados devem ser atualizados para restaurar a corretude da estrutura, exatamente como acontece em eventos na lista ordenada para o problema do máximo (veja Seção 3.1). A árvore que contém a lista é atualizada em tempo $O(\log n)$ e o torneio cinético para busca do par mais distante também é atualizado em tempo logarítmico.

A terceira camada de eventos é relativa aos eventos sobre a descrição do fecho convexo. Quando a descrição do fecho convexo muda, um novo vértice é incluído ou excluído do fecho. Conseqüentemente, uma reta aparece ou desaparece de um dos envelopes e pares opostos podem ser adicionados ou removidos da lista ordenada. Nesse caso devemos percorrer a árvore buscando o área afetada e modificar a estrutura de certificados, além de eventuais mudanças no torneio cinético. A árvore e o torneio são atualizados em tempo $O(\log n)$. Assim, a estrutura é de resposta rápida, pois o tempo de se processar um evento é $O(\log n)$.

O espaço gasto para manter a estrutura é proporcional ao tamanho da árvore mais o tamanho do torneio cinético mais o tamanho das estruturas cinéticas para os envelopes. As estruturas para os envelopes são compactas (veja seção anterior). A soma do número de vértices nos dois envelopes é o número de nós da árvore que contém a lista ordenada. Como o tamanho de um envelope é $O(n)$, o número de vértices também é $O(n)$. Esse também é o limitante para o número máximo de concorrentes no torneio cinético que descobre o par mais distante. Portanto a estrutura toda consome espaço $O(n)$. A fila de eventos também consome espaço linear, sendo uma estrutura compacta segundo o modelo cinético.

Para provar a eficiência da estrutura, devemos limitar o número de eventos na lista ordenada. De acordo com as seções anteriores, o fecho convexo de n pontos (e o envelope superior de n retas) sob movimentos algébricos muda $O(n^{2+\epsilon})$ vezes, para qualquer $\epsilon > 0$ em um (δ, n)-cenário. O número de pares opostos é determinado pela sobreposição das projeções dos envelopes superior e inferior no eixo x. Tomando o tempo como uma terceira dimensão é possível mostrar que essa quantidade obedece ao mesmo limitante [AGHV97, ASS96]. Então o número total de eventos é $O(n^{2+\epsilon})$.

O número máximo de eventos externos para o par de pontos mais distante é proporcional ao número de diagonais existentes no fecho convexo. Essa quantidade é $\Theta(n^2)$, o que leva a um fator extra de $O(n^\epsilon)$. Tomando $\epsilon < 1$, a estrutura é considerada eficiente sob o modelo cinético.

Porém, infelizmente a estrutura para a manutenção do par de pontos mais distante não é local, pois um único ponto pode pertencer a $O(n)$ pares opostos, como mostra o exemplo da Figura 5.9.

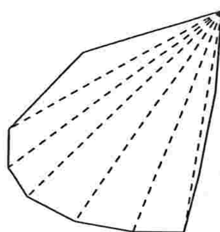


Figura 5.9: Contra-exemplo que desmente a localidade da estrutura de dados cinética para o problema do par de pontos mais distante. Um único ponto pertence a $O(n)$ pares opostos (exibidos nas linhas tracejadas).

5.5.2 Largura de um Conjunto de Pontos

Problema 5.6 (Largura de um conjunto de pontos) *Dado um conjunto S de n pontos no plano, determinar a distância mínima que duas retas paralelas devem ter entre si de modo que todos os pontos de S fiquem entre as duas retas.*

A largura de um conjunto de pontos no plano é definida como a separação mínima de duas linhas paralelas que comprimem todo o conjunto de pontos na área entre elas. Não é difícil ver que uma das retas se apóia sobre uma aresta do fecho convexo e a outra intercepta pelo menos um vértice do fecho.

Um primeiro candidato a ser o local onde devemos colocar tais retas paralelas é o par de pontos mais distante. Esse par de vértices admite um par de retas suporte externas ao fecho convexo, e por isso é um candidato a ser o par que determina a largura do conjunto de pontos. Se dispusermos as retas em posição perpendicular à diagonal entre o par de vértices mais distante, a distância entre as retas será exatamente igual à distância entre os dois vértices considerados.

Porém, em um vértice podemos rotacionar a reta suporte correspondente em torno do próprio vértice, obtendo inúmeras inclinações diferentes para a reta suporte. A limitação nessa rotação fica por conta da reta que também é suporte de uma das duas arestas do fecho convexo que incidem sobre o vértice considerado, pois ultrapassando essa limitação a reta deixaria de ser suporte passando a cruzar o interior do fecho convexo.

Considerando um par de vértices qualquer do fecho convexo, podemos rotacionar sincronizadamente as suas retas suporte de modo que elas continuem paralelas e selecionar qual inclinação proporciona a menor distância entre as duas retas. E a posição em que as duas retas paralelas estão mais próximas é exatamente quando uma delas está sobre uma aresta do fecho convexo, como mostra o Lema 5.7.

Lema 5.7 *Sejam r e s duas retas paralelas que são retas suporte de vértices a e b distintos do fecho convexo de S. A menor distância entre r e s é obtida na posição em que uma das retas também é reta suporte de uma das arestas do fecho convexo que são incidentes a a ou b.*

Prova: Sejam r e s retas suporte dos vértices a e b do fecho convexo, respectivamente, tal que r e s possam ser dispostas paralelamente. Suponha que rotacionamos sincronizadamente r e s de modo que s atinja um de seus limites, a aresta bc do fecho convexo, antes que r o faça. Nessa posição a distância entre a e s é igual ao comprimento de um dos catetos de um triângulo retângulo cuja hipotenusa é $d(a, b)$ (veja a Figura 5.10).

Pelo Teorema de Pitágoras sabemos que a distância entre s e a é menor que $d(a, b)$. Em qualquer outra posição em que s seja colocada, mantendo a relação de paralelismo com r, a linha que conecta a à s atravessa a aresta bc (ou a outra aresta que incide sobre b) e é maior que o cateto da situação anterior. Então a posição que corresponde à menor distância entre as retas suporte paralelas é obtida quando s é também a reta suporte de bc. □

De acordo com o Lema 5.7, o par de retas suporte que define a largura do conjunto de pontos será determinado por um par vértice-aresta. O valor da largura é dado pela distância entre um vértice e uma aresta do fecho convexo. Uma das retas pode ser encontrada através das coordenadas dos extremos da aresta considerada e a outra reta

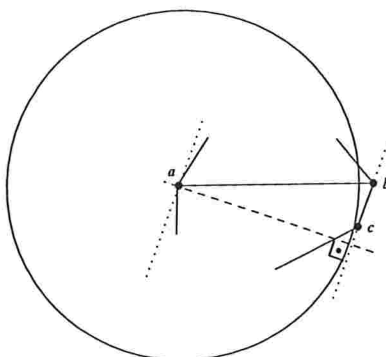


Figura 5.10: Exemplo de dois vértices a e b do fecho convexo e as retas suportes (pontilhadas). O triângulo retângulo tracejado mostra que a distância d entre as retas suporte nessa posição é menor que $d(a, b)$. O círculo mostra que d é a menor distância entre as retas suporte para o conjunto de inclinações possíveis.

é encontrada transladando a primeira reta através da distância obtida para representar a largura.

Como no problema do par de pontos mais distante, não estaremos interessados em todos os pares vértice-aresta do fecho convexo, mas apenas nos pares *opostos*, ou seja, pares em que um vértice e uma aresta admitem retas suportes paralelas (que são exatamente as retas procuradas). Devemos examinar todos os pares opostos vértice-aresta, selecionando aquele com menor distância.

Um algoritmo estático que resolve o problema utiliza o plano dual, exatamente como o algoritmo que busca pelo par mais distante. No plano dual temos duas cadeias, uma representando o envelope superior e outra representando o envelope inferior. A concatenação das duas cadeias nos dá o dual do fecho convexo. Cada vértice do fecho convexo é dualizado como uma reta que compõe um dos envelopes e cada aresta do fecho é mapeada em um vértice de um envelope. Assim, um par vértice-aresta no plano original tem um par reta-vértice como seu dual. Para encontrarmos o par reta-vértice procurado, devemos realizar uma varredura no plano dual similar à usada no caso do par mais distante, dessa vez enfocando os pares opostos reta-vértice e selecionar aquele com menor distância. Um par oposto reta-vértice corresponde, portanto, a um vértice de um dos envelopes e uma reta do outro envelope. O vértice dual em questão deve ter a sua coordenada x entre as coordenadas x dos extremos da porção da reta no envelope (veja a Figura 5.11).

O algoritmo estático que resolve o problema da largura de um conjunto de pontos é mostrado no Algoritmo 5.11. A exemplo do Algoritmo 5.10, utiliza como subrotinas `Dualiza`, `EnvelopeSuperior`, `EnvelopeInferior`, `ProximaParada` e `next` que são exatamente iguais neste caso. A novidade é a utilização de `SelMenorDistancia` que seleciona o par aresta-vértice com menor distância no plano original dentre um

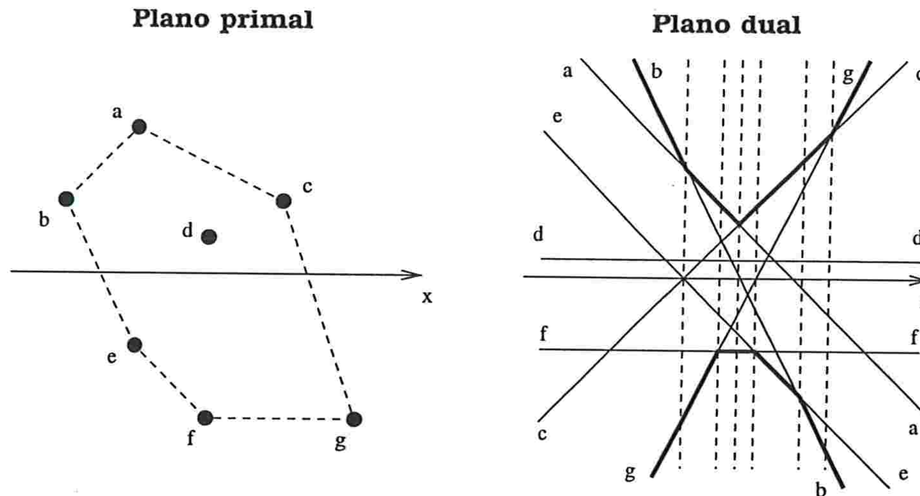


Figura 5.11: Os pares opostos reta-vértice aparecem no plano dual em seqüência quando fazemos uma varredura no eixo x. Cada linha vertical tracejada corresponde a um par oposto. Os pares opostos na figura são: (g, ba), (a, gf), (f, ac), (c, ef), (c, be) e (b, cg).

conjunto de pares opostos.

A dualização pode ser feita com complexidade de tempo $O(n)$. Cada envelope pode ser encontrado em tempo $O(n \log n)$ utilizando o Algoritmo 2.1. A varredura através dos pares opostos é realizada em tempo linear, bem como a seleção do par com menor distância, pois o número de pares opostos reta-vértice é proporcional ao de pares opostos reta-reta usados no problema do par mais distante, que é $O(n)$. Portanto o problema da largura de um conjunto de n pontos pode ser resolvido no cenário estático em tempo $O(n \log n)$.

A cinetização desse algoritmo é praticamente idêntica à usada para resolver o problema do par de pontos mais distante. As únicas diferenças é que precisamos organizar um torneio cinético com os pares opostos reta-vértice, e esse torneio precisa selecionar o par com distância mínima ao invés da máxima. A análise de desempenho da estrutura de dados cinética obtida também é muito similar. Novamente a estrutura é compacta, eficiente e de reposta rápida, mas não é local sob a ótica do modelo cinético.

5.6 Considerações Finais

É possível inserir e remover objetos da estrutura de dados cinética que mantém o fecho convexo, mas esse trabalho pode resultar em um número linear de mudanças na estrutura no pior caso. Por esse motivo, essa estrutura não pode ser considerada eficiente sob o ponto de vista do modelo cinético sobre cenários dinâmicos. Uma boa tentativa para se obter uma estrutura de dados cinética que tenha boas propriedades dinâmicas seria cinetizar a estrutura de dados dinâmica proposta por Overmars e Van

Algoritmo 5.11 Largura

Entrada: um conjunto S de pontos no plano.
Saída: o par aresta-vértice com menor distância.

```
D ← Dualiza(S)
E_sup ← EnvelopeSuperior(D)
E_inf ← EnvelopeInferior(D)
s ← reta associada ao ponto −∞ em E_sup
i ← reta associada ao ponto −∞ em E_inf
x ← −∞
enquanto ProximaParada(x) < +∞ faça
  x ← ProximaParada(x)
  se x está em E_sup então
    P ← P ∪ (x, i)
    s ← next(s)
  senão
    P ← P ∪ (x, s)
    i ← next(i)
p ← SelMenorDistancia(P)
retorna p
```

Leeuwen [OL81] que mantém o fecho convexo de um conjunto de n pontos no plano com custo $O(\log^2 n)$ por operação de inserção ou remoção. Porém, provar que um conjunto de certificados fornecido por essa estrutura é eficiente sob o ponto de vista do modelo cinético pode ser mais difícil do que aparenta, visto que até hoje não foi publicado nenhum artigo nesse sentido. Segundo Basch [Ba99], há a suspeita de a estrutura tenha que lidar com a manutenção da mediana de n valores em movimento. Nenhum limite combinatorial aceitável para a aplicação do modelo cinético é conhecido ainda para o problema da mediana de valores em movimento, permanecendo ainda um problema em aberto na área de Combinatória.

A utilização da dualidade entre o fecho convexo e a concatenação dos envelopes superior e inferior pode servir de ponto de partida para a elaboração de estruturas de dados cinéticas para o fecho convexo em dimensões maiores que $\mathbb{R}^2$. Talvez a exploração de um paralelo entre o fecho convexo tridimensional e o envelope de um arranjo de superfícies possa levar a um algoritmo que resolva o problema estático do fecho convexo tridimensional fornecendo uma lista de certificados que possibilite a obtenção de uma estrutura de dados cinética eficiente segundo o modelo cinético.

Capítulo 6

Diagrama de Voronoi e Triangularização de Delaunay

Neste capítulo vamos descrever uma estrutura de dados cinética que mantém a triangularização de Delaunay e o diagrama de Voronoi de um conjunto de pontos em movimento no plano.

6.1 Problema Estático

Problema 6.1 (Diagrama de Voronoi) *Dado um conjunto S de n pontos no plano, determinar para cada ponto p em S qual é a região $V(p)$ dos pontos do plano que estão mais próximos de p do que qualquer outro ponto em S. A união das n regiões $V(p_1), \ldots, V(p_n)$ forma uma partição do plano chamada* Diagrama de Voronoi *de S, que denotamos $Vor(S)$.*

O problema acima descreve o *Diagrama de Voronoi* no cenário estático. Tal diagrama é composto da reunião de diversos polígonos denominados polígonos de Voronoi, onde o interior de cada polígono representa a região de Voronoi $V(p_i)$ do ponto p_i, conforme mostra a Figura 6.1. As arestas dos polígonos são as arestas de Voronoi e representam pontos do plano que são equidistantes em relação a dois pontos de S. Os vértices dos polígonos, denominados vértices de Voronoi, correspondem a pontos do plano que equidistam de 3 pontos diferentes[1].

Considere o grafo dual do diagrama de Voronoi de um conjunto S de pontos no plano formado da seguinte forma: os vértices do grafo são os pontos de S e existe uma aresta entre dois vértices do grafo se os polígonos de Voronoi associados a tais vértices possuírem uma aresta em comum, ou seja, se as regiões correspondentes são vizinhas. Este grafo é denominado *Triangularização de Delaunay* de S, ou apenas

[1]supondo que não existam quatro pontos co-circulares em S.

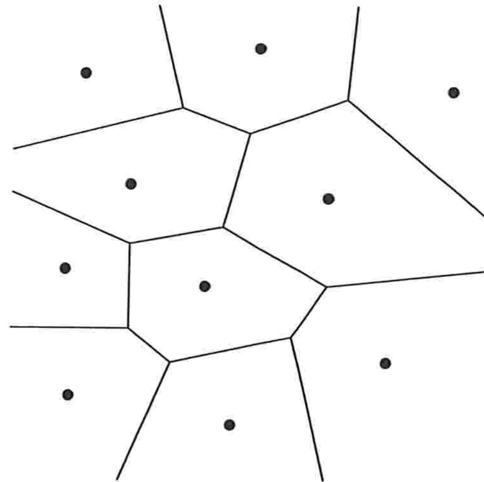


Figura 6.1: Diagrama de Voronoi de um conjunto de pontos.

$Del(S)$. Em 1934, o próprio Delaunay mostrou que o grafo definido acima produz uma triangularização do conjunto S de pontos no plano, isto é, produz um grafo planar cujas faces internas são triângulos. Um exemplo pode ser visto na Figura 6.2.

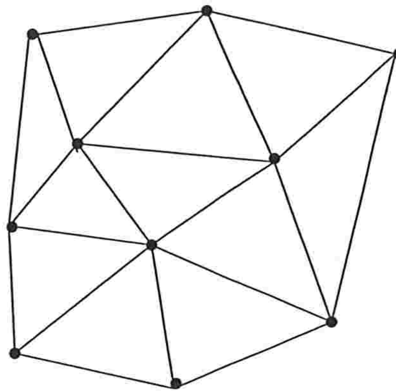


Figura 6.2: Triangularização de Delaunay do mesmo conjunto de pontos da Figura 6.1.

Assim como o problema dos envelopes está intimamente relacionado ao problema do fecho convexo, conforme visto no Capítulo 5, a triangularização de Delaunay está interligada ao diagrama de Voronoi (são problemas duais). É possível obter uma solução para um dos problemas a partir da solução para o outro em tempo linear no número de pontos. Cada vértice de Voronoi é dado pelo circuncentro de cada triângulo em $Del(S)$, e mais, assumindo que não existam quatro pontos co-circulares em S, existe um círculo definido pelos três vértices de cada triângulo em $Del(S)$ que não contém nenhum outro ponto de S. Uma aresta de Voronoi liga dois vértices de Voronoi se os triângulos correspondentes a tais vértices são vizinhos em $Del(S)$. A Figura 6.3

apresenta os dois atributos geométricos para o mesmo conjunto de pontos.

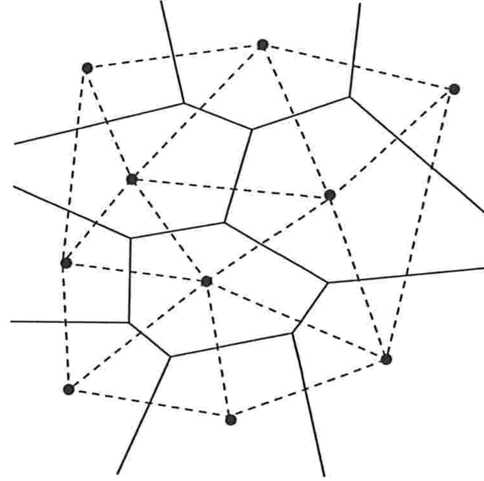


Figura 6.3: Diagrama de Voronoi e a sua triangularização de Delaunay dual para um mesmo conjunto de pontos.

Um algoritmo que resolve o problema da Triangularização de Delaunay no cenário estático com complexidade de tempo $O(n \log n)$, utilizando a técnica de divisão-e-conquista foi proposto por Lee [Lee80]. Nesse algoritmo assume-se que qualquer subconjunto de quatro pontos de S não é um conjunto de pontos co-circulares. O algoritmo recebe o conjunto S pré-ordenado em relação ao eixo x, divide o conjunto na metade, obtém a solução de cada metade recursivamente e combina estas soluções parciais, construindo a solução final. A obtenção da Triangularização de Delaunay usa como primitivas internas dois tipos de testes: `Left`(p_i, p_j, p_k) que diz se um ponto p_k está à esquerda de uma aresta orientada determinada pelos pontos p_i e p_j, e `InCircle`(p_i, p_j, p_k, p_l) que diz se um ponto p_l está dentro, fora ou sobre a circunferência orientada determinada pelos pontos p_i, p_j, p_k [GS86]. Sendo p_i descrito pelas coordenadas (x_{p_i}, y_{p_i}), temos:

$$Left(p_i, p_j, p_k) = \begin{vmatrix} x_{p_i} & y_{p_i} & 1 \\ x_{p_j} & y_{p_j} & 1 \\ x_{p_k} & y_{p_k} & 1 \end{vmatrix}$$

$$InCircle(p_i, p_j, p_k, p_l) = \begin{vmatrix} x_{p_i} & y_{p_i} & {x_{p_i}}^2 + {y_{p_i}}^2 & 1 \\ x_{p_j} & y_{p_j} & {x_{p_j}}^2 + {y_{p_j}}^2 & 1 \\ x_{p_k} & y_{p_k} & {x_{p_k}}^2 + {y_{p_k}}^2 & 1 \\ x_{p_l} & y_{p_l} & {x_{p_l}}^2 + {y_{p_l}}^2 & 1 \end{vmatrix}$$

`Left`(p_i, p_j, p_k) constitui um teste em que valores maiores que zero significam que o ponto p_k está à esquerda do segmento orientado entre p_i e p_j. Quando `Left`(p_i, p_j, p_k)

é igual a zero, sabemos que p_k está sobre o segmento orientado, ou seja, os 3 pontos são colineares, e valores menores que zero significam que p_k está à direita do segmento (p_i, p_j).

`InCircle`(p_i, p_j, p_k, p_l) testa se o ponto p_l está dentro do círculo orientado que passa pelos três pontos restantes. Valores maiores que zero nesse teste significam que o ponto p_l é interior a esse círculo orientado. Quando `InCircle`(p_i, p_j, p_k, p_l) é igual a zero, podemos dizer que p_l está sobre a fronteira do círculo orientado. Valores menores que zero significam que p_l está do lado de fora do círculo.

6.2 Problema Cinético

Se considerarmos o conjunto S de n pontos movendo-se continuamente ao longo do tempo temos que, a cada instante, os pontos definem uma triangularização de Delaunay que também muda continuamente com o tempo, mas, entre certos instantes especiais, a descrição combinatória da triangularização e, conseqüentemente, a do diagrama de Voronoi, permanece inalterada.

Guibas, Mitchell e Roos [GMR91] propuseram uma estrutura de dados cinética para o problema armazenando cada tipo de teste `Left` e `InCircle` como um certificado. Assim, os eventos considerados são os instantes onde as propriedades `Left` entre um ponto e uma aresta e `InCircle` entre um ponto e uma circunferência mudam de sinal. A figura 6.4 mostra mais claramente o que acontece quando um certificado `InCircle` perde a sua validade. Certificados do tipo `Left` são utilizados apenas na região da fronteira do fecho convexo de S.

Na verdade essa estrutura de dados cinética não é a cinetização do algoritmo de Lee, mas a animação da estrutura combinatória da triangularização de Delaunay. Como a própria estrutura é auto-certificada, podemos usar qualquer algoritmo que produza a triangularização de Delaunay no cenário estático antes de iniciar a contagem do tempo, obtendo o mesmo conjunto de certificados.

A violação de um certificado do tipo `InCircle` ocasiona uma inversão dos triângulos de Delaunay responsáveis pelo certificado que perdeu sua validade (veja a Figura 6.4). Já o processamento de um certificado do tipo `Left` envolve o aparecimento (ou desaparecimento) de um novo ponto no fecho convexo de S, produzindo a aparição de um novo triângulo em $Del(S)$ (ou o desaparecimento de um triângulo de $Del(S)$).

Para simplificar a discussão, vamos considerar o conjunto S com a inclusão de um novo ponto ∞. Assim, o conjunto que vamos considerar é $S' = S \cup \{\infty\}$. A triangularização de Delaunay estendida é dada por

$$Del(S') = Del(S) \cup \{(p_i, \infty) | p_i \in Conv(S)\}$$

ou seja, todo ponto na fronteira do fecho convexo de S é conectado a ∞ produzindo uma triangularização estendida onde todas as faces são internas e são triângulares. A

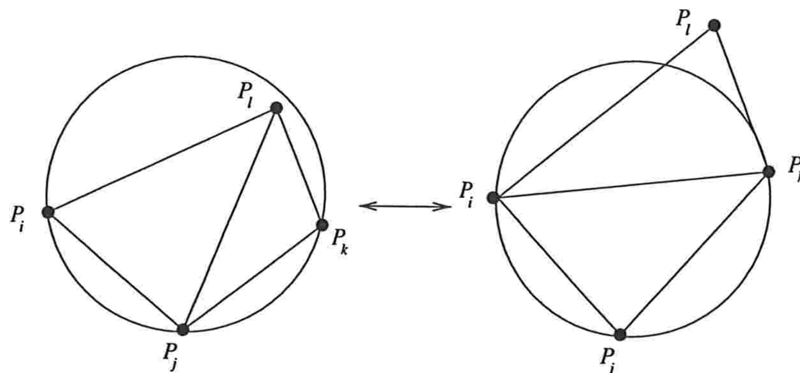


Figura 6.4: Violação de um certificado InCircle

vantagem dessa representação estendida é que podemos representar tudo em função de triplas, sem nos preocuparmos com a fronteira do fecho convexo de S como casos especiais. Essa representação permite que se elimine o tratamento diferenciado de eventos correspondentes a certificados do tipo `Left`, pois todos os tipos de certificados existentes serão do tipo `InCircle`[2]. Porém o cálculo do prazo de validade dos certificados permanece diferenciado.

Tomando $p_i, p_j, p_k \in S$, $\triangle(p_i, p_j, p_k)$ o circuncentro do triângulo formado por p_i, p_j, p_k e $C(p_i, p_j, p_k)$ o círculo definido pelos pontos p_i, p_j, p_k podemos caracterizar a dualidade entre $Del(S')$ e $Vor(S)$:

$$(p_i, p_j, p_k) \in Del(S') \iff \triangle(p_i, p_j, p_k) \in Vor(S).$$

$$(p_i, p_j, \infty) \in Del(S') \iff p_i \text{ e } p_j \text{ são vértices vizinhos em } Conv(S).$$

A transição mostrada na Figura 6.4 decorrente de um evento `InCircle` é equivalente a uma *fusão momentânea* de dois vértices de Voronoi que ocorre no instante t em que os quatro pontos se tornam co-circulares, seguida da criação de um novo vértice de Voronoi no instante $t + \epsilon$, para $\epsilon > 0$.

Um par de triângulos adjacentes em $Del(S')$ é denominado *quadrilátero.* Cada triângulo tem no máximo três triângulos vizinhos, participando então de no máximo 3 quadriláteros. Desse modo o número total de quadriláteros é linear e cada certificado é formado por um quadrilátero. A estrutura de dados cinética é formada então em três passos:

1. Calcular a triângularização de Delaunay de S' na posição inicial $t = t_0$.
2. Para cada quadrilátero em $Del(S'(t_0))$ calcular os prazos de validade para o certificado correspondente.

[2]Um círculo definido por p_i, p_j, ∞ é equivalente a uma linha reta passando pelos pontos p_i e p_j.

3. Inserir cada certificado na fila de eventos.

O primeiro passo acima pode ser feito em tempo $O(n \log n)$ usando o algoritmo de Lee. No passo seguinte, para os quadriláteros $\{p_i, p_j, p_k, p_l\} \in Del(S'(t_0))$ precisamos calcular os zeros da função InCircle(p_i, p_j, p_k, p_l) e para os quadriláteros da forma $\{p_i, p_j, p_k, \infty\} \in Del(S'(t_0))$ computar os zeros da função Left(p_i, p_j, p_k). Esse passo pode ser realizado em tempo $O(n)$. O terceiro passo corresponde a $O(n)$ inserções em uma fila de prioridade, resultando um tempo $O(n \log n)$.

O processamento de um evento envolve uma troca entre os triângulos de Delaunay envolvidos conforme mostra a Figura 6.4. Porém, em certos casos degenerados, pode ser que mais de um quadrilátero adjacente produza um evento ao mesmo tempo. Esse tipo de situação ocorre quando mais de quatro pontos numa certa vizinhança tornam-se co-circulares, podendo provocar o aparecimento de erros na estrutura interna da triangularização se não forem tratados diferentemente. Nesse caso a solução é utilizar o algoritmo proposto em [AGSS87] que retriangulariza o interior do polígono convexo formado pelos pontos que se tornaram co-circulares para o instante $t + \epsilon$ (quando não devem ser mais co-circulares) em tempo linear.

Ao criarmos um evento podemos descobrir se ele corresponde a uma troca simples ou a uma retriangularização local na fase de inserção do evento na fila de eventos. Se o prazo de validade de um certificado que está entrando na fila é igual ao de algum que já está na fila e os quadriláteros correspondentes são adjacentes, então o evento será de retriangularização (assim como o evento associado ao outro quadrilátero), caso contrário será de troca simples.

Cada evento de troca simples afeta outros quatro quadriláteros além do responsável pelo evento, correspondendo a quatro atualizações na fila de eventos que podem ser concluídas em tempo $O(\log n)$ cada uma. Um evento de retriangularização pode afetar muitos quadriláteros, mas em [GMR91, Ro93] é mostrado que o tempo amortizado por operação continua sendo $O(\log n)$.

A estrutura de dados cinética proposta para o problema é considerada, portanto, de resposta rápida pois o tempo amortizado de processamento de um evento é $O(\log n)$. A estrutura também é compacta pois o número de certificados na fila de eventos é proporcional ao número de quadriláteros, que é linear.

O número de eventos é igual ao número de eventos externos, pois a perda de validade dos certificados utilizados sempre promove mudanças na triangularização de Delaunay que é o atributo geométrico analisado. Portanto, a estrutura também é eficiente. De acordo com [GMR91, Ro93] o número total de eventos é $O(n^2 \lambda_s(n))$ assumindo que os planos de vôo de cada ponto são funções que tomadas duas a duas se cruzam no máximo s vezes. No caso de movimentos lineares temos que $\lambda_1(n) = n$, produzindo um número máximo de eventos $O(n^3)$. Porém, a estrutura não é local pois um único ponto pode aparecer em $O(n)$ certificados. Na Figura 6.5 podemos ver um exemplo que contém um ponto que pertence a $O(n)$ quadriláteros.

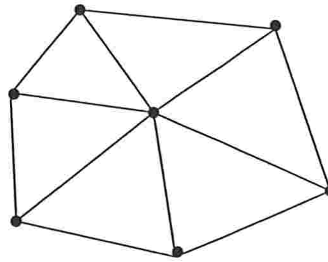


Figura 6.5: Um exemplo mostrando que a estrutura de dados cinética para a triangularização de Delaunay não é local. Repare que o ponto central é vértice de $O(n)$ triângulos.

6.3 Considerações Finais

A estrutura de dados cinética apresentada para a manutenção da triangularização de Dealunay de um conjunto de n pontos em movimento contínuo é um exemplo de estrutura auto-certificada, pois a própria estrutura trabalha como um conjunto de certificados para si própria. Estruturas que apresentam essa propriedade sempre serão eficientes porque o número de eventos internos é nulo.

A estrutura apresentada não é local, pois vértices de alto grau ($O(n)$) podem aparecer no grafo da triangularização. Porém esse é o melhor resultado anunciado até hoje. Poderíamos argumentar que, como a triangularização é um grafo planar, segundo a lei de Euler para cada vértice de alto grau deveremos ter muitos vértices de baixo grau para manter a relação proporcional entre o número de vértices e o número de faces e arestas. Com isso, em aplicações práticas onde espera-se que as alterações nos movimentos dos pontos sejam probabilísticamente espalhadas por todos os pontos do conjunto, poderíamos argumentar que, segundo a lei das probabilidades, a estrutura de dados cinética apresentada possua comportamento local. Para m atualizações nos planos de vôo dos pontos seria esperado que teríamos poucas atualizações em $O(n)$ certificados e muitas atualizações $O(1)$. Um próximo resultado sobre a estrutura apresentada seria a aplicabilidade de análise estatística sobre o seu comportamento.

Capítulo 7

Implementações

Todas as estruturas cinéticas descritas no Capítulo 3 foram implementadas: a lista ordenada cinética, o heap cinético e o torneio cinético. Nosso intento neste capítulo é o de descrever as estratégias adotadas no trabalho de implementação dessas estruturas de dados cinéticas, bem como apresentar alguns resultados obtidos em experimentos feitos com o resultado do trabalho de implementação para o problema do máximo cinético.

O desenvolvimento foi todo realizado em plataforma Unix-Solaris/Linux utilizando linguagem C++. Tomou-se o cuidado de conceber o código sob padrão ISO/ANSI-C++, tornando-o o mais portável possível entre diferentes plataformas. Constatou-se, por exemplo, que as implementações funcionam sob ambiente Windows98 usando o compilador Borland C++ Builder 4.0.

Outra preocupação foi a de que o código deveria ser o mais geral possível para que pudesse ser facilmente estendido ao desenvolver uma nova estrutura de dados cinética. Daí a escolha da linguagem para a implementação foi C++, pois o código fica facilmente extensível com a utilização de templates [De98, Se98, MS96]. Nenhuma interface gráfica foi desenvolvida. Os programas recebem entradas descritas em arquivos texto e devolvem as respostas também em formato texto. Com essas ferramentas elaborou-se uma modelagem orientada a objetos baseada nas idéias que aparecem nas implementações de Basch [Ba99].

Nossa intenção era a de criar uma biblioteca básica e extensível para a construção de estruturas de dados cinéticas. Com isso a fase de implementações centrou-se no problema do máximo exatamente por tratar-se de um problema básico utilizado como sub-problema de muitos outros problemas geométricos mais complicados. Também tínhamos como objetivo realizar um teste comparativo entre diferentes estruturas de dados cinéticas disponíveis para um mesmo problema a fim de comprovar na prática os limites teóricos apresentados.

7.1 Implementando Estruturas de Dados Cinéticas

Nesta seção vamos apresentar o modelo orientado a objeto desenvolvido. Tal modelo foi concebido com o intuito de tornar simples a criação de quaisquer novas estruturas de dados cinéticas. Para facilitar a vida do programador que vai construir uma nova estrutura de dados cinética, dividimos a implementação em dois níveis. No nível das classes mais básicas definimos toda a interação intrínseca entre os objetos que compõem uma estrutura de dados cinética, qualquer que seja ela. O nível mais alto é onde as classes básicas são estendidas e onde é inserida a particularização da estrutura de dados cinética que está sendo implementada.

7.1.1 As classes básicas

Uma estrutura de dados cinética é montada através de três componentes básicos: `kds-base`, `item-base` e `cert-base`. O primeiro componente refere-se à base da estrutura de dados cinética (*kinetic data structure - kds*) propriamente dita, o componente `item-base` é a base para o encapsulamento de um objeto geométrico em movimento no interior da estrutura (chamado de item) e o componente `cert-base` representa um certificado. Além desses três, existe um quarto componente básico, `obj-base`, que serve de base para representar qualquer objeto geométrico tais como pontos, retas, círculos, polígonos, etc. A seguir apresentamos a interface de cada uma dessas classes.

```
class obj_base {
 private:
  list<item_base *> items;    // lista de itens que referenciam o objeto
  char *id;                   // identificador

 public:
  obj_base(const char *str);
  ~obj_base();
  list<item_base *> &get_items_list();
  list<item_base *>::iterator add_item(item_base *i);
  char *get_id();
  void change_id(const char *str);
};
```

`obj-base` é uma classe base para um objeto em movimento. Qualquer objeto, seja ponto, reta ou polígono, deve estender esta classe no nível mais alto da implementação. Podemos notar na interface de `obj-base` que um objeto básico possui apenas uma lista dos itens que o referenciam na estrutura e um identificador que o distingue dos outros objetos na estrutura, além de alguns métodos de administração desses campos, onde se destaca o método `add-item` que é usado quando criamos um outro item que referencia o objeto.

```
class item_base {
 private:
  obj_base *obj;

  // posição na lista de itens do objeto
  list<item_base *>::iterator objlist_ptr;

  // posição na lista de itens do kds
  list<item_base *>::iterator kdslist_ptr;

 public:
  item_base(obj_base *o);
  list<item_base *>::iterator &obj_list_pointer();
  list<item_base *>::iterator &kds_list_pointer();
  obj_base *get_obj_base() const;
  virtual void flightplan_changed() = 0;
};
```

`item-base` é a classe que permite que um mesmo objeto `obj-base` faça parte de mais de um certificado e de mais que uma estrutura de dados cinética sem que precisemos replicá-lo. Na verdade tudo o que ela possui, essencialmente, é um apontador para o objeto que está referenciando, evitando a multiplicação desnecessária de dados. Imagine que um mesmo objeto participe de 15 certificados diferentes. Em cada um deles existe um `item-base` que referencia um mesmo objeto. Note que também armazenamos apontadores para a posição nas listas que contêm esse item de forma que possamos achá-lo em tempo constante para removê-lo ou alterá-lo. Repare o método virtual `flightplan-changed` que será implementado em todas as extensões de `item-base`. Esse método deve ser chamado pelo objeto que é o "dono" deste item logo no instante em que o plano de vôo é alterado e sua implementação será responsável pelas alterações na estrutura em decorrência dessa alteração no plano de vôo de um objeto.

```
class cert_base {
 private:
  time_type death;        // prazo de validade

  // posição na lista de certificados do kds
  list<cert_base *>::iterator kdslist_ptr;

 public:
  cert_base() { death = INFINITE; }
  time_type death_time() { return death; }
  void set_death_time(time_type d) { death = d; }
```

```
  list<cert_base *>::iterator &kds_list_pointer();

  // métodos virtuais (dependentes de cada implementação de cert_base)
  virtual time_type calc_death_time() const = 0;
  virtual void process_death() = 0;
};
```

cert-base é a classe mais básica para um certificado. Possui um apontador para a sua posição na lista de certificados do kds e um prazo de validade, além de fornecer interface para a edição desses campos. Note a presença de dois métodos virtuais: calc-death-time é responsável pelo cálculo do prazo de validade do certificado e process-death implementa os procedimentos que devem ser seguidos logo após o certificado ser removido da fila de eventos.

```
class kds_base {
 protected:
  clock_type *clock;                // relógio
  event_queue *eq;                  // fila de eventos
  list<cert_base *> cert_list;      // lista de certificados
  list<item_base *> item_list;      // lista de itens
  list<obj_base *>  obj_list;       // lista de objetos

 public:
  kds_base();
  virtual ~kds_base();
  event_queue *get_eventqueue() { return eq; }
  void set_eventqueue(event_queue *q) { eq = q; }
  void set_clock(clock_type *c) { clock = c; }
  time_type now() { return clock->now(); }
  list<item_base *> &get_item_list() { return item_list; }
  list<item_base *>::iterator add_item(item_base *i);
  list<cert_base *> &get_cert_list() { return cert_list; }
  list<cert_base *>::iterator add_certificate(cert_base *c);
  bool step_forward();

  virtual void add_object(obj_base *) = 0;
  virtual void create_certificates() = 0;
};
```

Note que a interface da classe kds-base contém apontadores para uma fila de eventos e um relógio e possui três listas: uma de objetos (obj-base), uma de itens (item-base) e outra de certificados (cert-base). Além disso provê uma infinidade de métodos que manipulam esses campos e outros métodos interessantes. step-forward

faz a estrutura de dados avançar até o próximo evento. `add-object` inclui um novo objeto na estrutura. Se for chamada após o início da simulação deve construir os itens e certificados associados ao novo objeto. Caso `add-object` seja chamada antes do disparo do relógio, então apenas insere o objeto na lista de objetos. `create-certificates` é chamado quando o relógio "dá a largada", sendo responsável pela montagem de toda a estrutura interna, aplicando o algoritmo estático. Nesse momento os certificados são criados e colocados na fila de eventos.

O objetivo da existência dessas classes básicas é o de definir em um nível básico a interligação que sempre deverá existir entre cada um dos componentes de uma estrutura de dados cinética, qualquer que seja ela. Essa interligação comum a toda estrutura de dados cinética foi implementada nesse nível básico. Dessa forma, ao criar uma nova estrutura, o programador não tem que se preocupar com detalhes de "baixo nível" da construção de uma estrutura de dados cinética. Basta que ele estenda cada uma das classes básicas, colocando a implementação da estrutura na criação das subclasses.

Além dos quatro componentes já citados, existe um quinto elemento que é a fila de eventos. Porém esse componente não precisará ser estendido, pois tudo o que a fila de eventos deve fazer é comportar diferentes certificados em uma fila de prioridade cuja chave é definida pelo prazo de validade de cada certificado. Assim, a fila de eventos foi implementada para armazenar várias instâncias da classe básica `cert-base`. A fila de eventos deve ser global em uma aplicação que use estruturas de dados cinéticas mesmo que haja mais que uma, podendo receber qualquer tipo de certificado pois todos os certificados devem ser subclasses de `cert-base`.

```
class event_queue : public priority_queue<kcert_base *> {
 public:
  event_queue();
  cert_base *next_event();
  time_type next_event_time();
  void ins_event(cert_base *e);
  void del_event(cert_base *e);
  void relocate_event(cert_base *e);
  int update(time_type t);
};
```

A fila de eventos possui alguns métodos que são utilizados pelas subclasses de `kds`, tais como retornar qual o certificado que será responsável pelo próximo evento (`next-event`), inserir (`ins-event`), remover (`del-event`) ou atualizar certificados (`relocate-event`). Uma atualização de um certificado em geral envolve uma remoção seguida de uma re-inserção do certificado removido na fila (com a prioridade atualizada). Um método de particular interesse na interface da fila de eventos com a estrutura é o método `update` que recebe um instante no futuro e avança o relógio até esse instante tomando conta de cada ocorrência de eventos até que o instante desejado seja alcançado.

A cada instante em que ocorre um evento, um método virtual do objeto `cert-base` responsável pelo evento é chamado para processar a morte desse certificado que é removido da fila (método `process-death`). Esse método é um método virtual para que a fila de eventos possa conter elementos `cert-base` genéricos e chamar esse método mesmo antes dele ter sido implementado. Aliás, tal método deverá ser implementado na definição de cada tipo de certificado, o que ocorre a cada extensão da classe `cert-base`.

7.1.2 Estendendo as classes básicas

Aplicações que implementem estruturas de dados cinéticas devem preocupar-se exclusivamente com este nível da implementação pois tudo o que acontece "nos bastidores" já foi implementado no nível mais baixo. Primeiramente devem definir o tipo de objeto geométrico em movimento utilizado estendendo a classe básica `obj-base` e definindo suas características e métodos virtuais. O mesmo deverá ser feito com relação às classes `kds-base`, `item-base` e `cert-base`.

Apenas um objeto k da subclasse de `kds-base` é criado na aplicação, sendo o responsável pelo gerenciamento da fila de eventos. Os objetos são criados antes de iniciada a contagem do tempo e inseridos em k através de chamadas sucessivas do método virtual `add-object`. Antes que o tempo possa começar a correr, é necessário que os certificados sejam calculados e colocados na fila de eventos, o que é feito executando o algoritmo que resolve o problema estático em `create-certificates`. A partir de então, k torna-se o dono de todos os itens (instâncias da subclasse de `item-base`) e certificados (instâncias da subclasse de `cert-base`) criados e o relógio pode ser acionado. k torna-se responsável pela manipulação dos eventos, usando a interface da fila de eventos, e pela correção da estrutura a cada ocorrência de eventos ou alteração de plano de vôo. A qualquer momento é possível obter o atributo que está sendo mantido por k simplesmente requisitando uma consulta.

k é criado por um *cliente* que está interessado no atributo geométrico que é mantido por k. Esse cliente pode ser uma aplicação ou, quando se utiliza uma combinação de diferentes estruturas de dados cinéticas, pode ser uma outra estrutura de dados cinética que usa o resultado de k como uma de suas entradas. Porém, mesmo nesse caso, apenas uma única fila de eventos global é utilizada, armazenando certificados de diversos tipos, associados às diferentes subclasses de `kds-base` que são utilizadas (conseqüentemente, diferentes subclasses de `cert-base` serão armazenadas na fila de eventos, uma para cada tipo de certificado).

A função do cliente é atuar como interface com uma estrutura de dados cinética. Ele pede à uma estrutura que realize alguma ação em um dado instante. O cliente pode solicitar a inclusão de um novo objeto no instante inicial (se a estrutura permitir operações dinâmicas então o cliente também pode solicitar inserções e remoções de objetos da estrutura a qualquer momento), a alteração do plano de vôo de um objeto na estrutura (alterações descontínuas só são permitidas se a estrutura permite operações dinâmicas) ou consultar o atributo mantido pela estrutura de dados cinética.

Para cada objeto em movimento, o cliente pede para que k crie um item que armazene tal objeto dentro da estrutura. Se o cenário sobre o qual a estrutura trabalha for um cenário cinético-dinâmico, esses pedidos podem ser feitos a qualquer instante. Caso contrário devem ser feitos sempre antes do disparo do relógio. Da mesma forma, o cliente pode remover um objeto da estrutura pedindo para que k destrua o item associado, bem como todos os certificados associados a esse item. Mudanças no plano de vôo dos objetos também são informadas pelo cliente e devem ser propagadas para todos os certificados que dependem de cada item associado ao objeto que tem seu plano de vôo alterado. Essa propagação é dependente da estrutura e deve ser implementada na extensão da classe `item-base`.

Podemos visualizar todos os componentes que fazem parte da implementação de uma estrutura de dados cinética segundo a nossa estratégia na Figura 7.1.

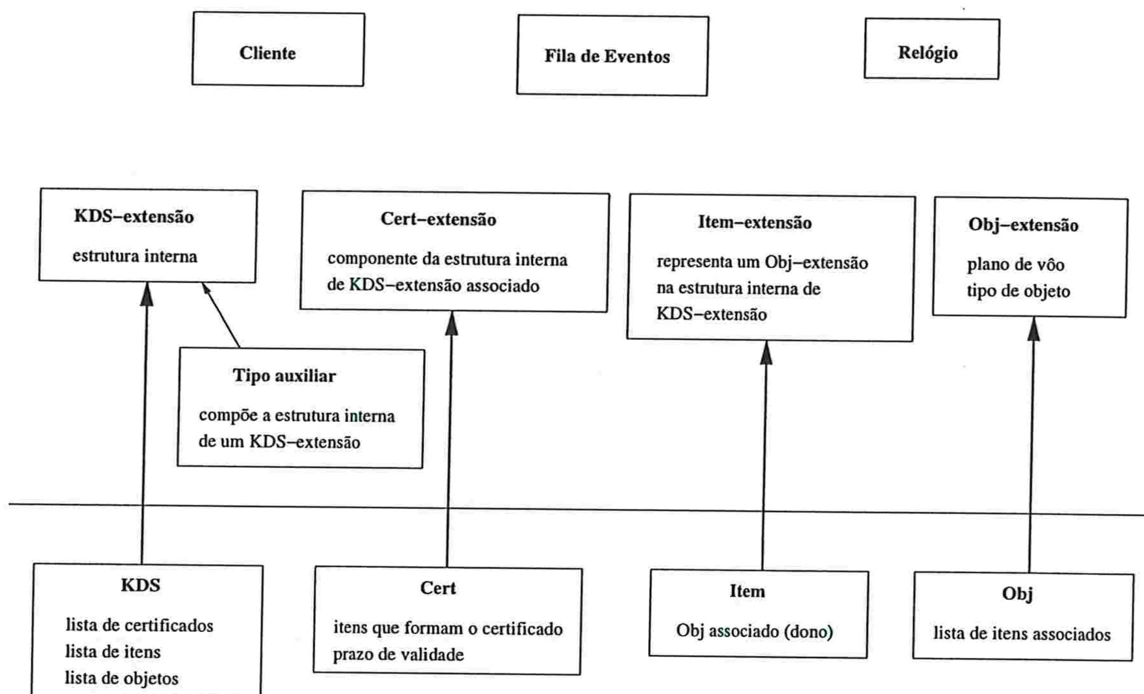


Figura 7.1: Todos os componentes que fazem parte de uma implementação de uma estrutura de dados cinética. Abaixo da linha horizontal está o nível mais básico e acima o nível mais alto.

7.2 Implementações para o Problema do Máximo

Implementamos as estruturas de dados cinéticas apresentadas para o problema do máximo (Capítulo 3). Para todas as estruturas de dados cinéticas implementadas, o cliente é representado por uma aplicação que recebe um arquivo texto contendo

instruções para a estrutura como entrada. A aplicação lê esse arquivo e, para cada instrução encontrada, aciona o método conveniente da estrutura para que a instrução seja executada. Nas primeiras linhas do arquivo definimos o conjunto de pontos inicial, um ponto por linha contendo uma string identificadora e um polinômio associado. A estrutura recebe esse conjunto e executa o respectivo algoritmo estático sobre esse conjunto, criando nessa etapa toda a estrutura de certificados e inserindo-os na fila de eventos. Um exemplo de um arquivo para o cliente é exibido a seguir:

```
5
um 1 2 50
dois 1 3 50
tres 1 50 2
quatro 2 1 0 3
cinco 2 -1 1 10

0.5 1
consulta

2 1
acelera zero 2

28 2
consulta
remove quatro

30 1
consulta

45 1
insere quatro 0 20

105 1
consulta
acelera tres -1

0 0
```

A primeira parte do arquivo de entrada exemplo diz que no instante inicial teremos 5 objetos na estrutura e em cada uma das 5 linhas seguintes há a descrição de um objeto através do seu identificador, o grau do polinômio que rege o seu plano de vôo seguido pelos coeficientes do polinômio.

A segunda parte do arquivo de entrada é representada pelas ações em ordem cronológica sobre a estrutura. Temos uma linha contendo o instante e o número n de

ações que devem ser realizadas naquele instante, seguida de n linhas com as ações. Uma linha com instante 0 e 0 ações encerra a entrada. Foram implementados quatro tipos diferentes de ações: `consulta`, `acelera`, `insere` e `remove`.

Uma ação do tipo `consulta` não possui nenhum parâmetro e apenas pede para a estrutura de dados cinética que emita um relatório (na saída padrão) mostrando a configuração do atributo geométrico que ela mantém naquele instante. No caso dos problemas implementados, a resposta à uma ação do tipo `consulta` devolve o ponto que é o máximo do conjunto no instante do pedido. Para isso é necessário atualizar a estrutura até o instante atual. Isso é feito usando o método interno da estrutura que avança o relógio da fila de eventos até o instante atual, realizando as alterações necessárias a cada ocorrência de um evento antes que o relógio chegue até o momento da consulta.

Ações do tipo `acelera` possuem dois parâmetros: um identificador válido, ou seja, que se refira a um objeto existente na estrutura, e a nova aceleração do movimento desse objeto, representada por um número. Essa ação age de forma que o objeto tenha a equação do seu movimento modificada para um polinômio de segundo grau (representando um movimento uniformemente acelerado) permanecendo na mesma posição em que estava antes de trocar de plano de vôo nesse mesmo instante, evitando alterações descontínuas na estrutura de dados cinética.

A instrução `insere` também possui dois parâmetros: um identificador válido (nesse caso um que não seja usado por nenhum ponto na estrutura) e um polinômio. Ao interpretar uma instrução desse tipo o cliente pede para a estrutura de dados cinética que insira um novo ponto passando o identificador e o seu plano de vôo.

Por fim, instruções do tipo `remove` são invocadas com apenas um parâmetro, que é um identificador válido, ou seja, que esteja sendo usado por algum ponto na estrutura. Neste caso, o cliente pede para que a estrutura de dados cinética remova o objeto do seu interior. O processo destrói todos os itens associados ao objeto, bem como os certificados que tinham algum dos itens destruídos como um de seus elementos.

7.2.1 Torneio cinético

Considere o torneio cinético para o problema do máximo. O tipo de objeto utilizado é o ponto unidimensional, cuja trajetória em função do tempo é descrita por um polinômio. Nesse caso, a implementação define a classe `point-1d`, que é uma extensão da classe base `obj-base`. O método construtor de `point-1d` recebe um polinômio, que será o plano de vôo inicial do objeto, e um identificador, que servirá para diferenciarmos cada um dos pontos envolvidos. A interface da classe `point-1d` é exibida a seguir.

```
// Classe ponto 1-dimensional em movimento
class point_1d : public obj_base {
 private:
```

```
  polynomial flightplan;  // plano de voo polinomial

 public:
  point_1d(const polynomial &p, clock_type *c, const char *id);
  double position() const;
  double speed() const;
  polynomial get_flightplan() const;
  void change_flightplan(const polynomial &p);
  void accelerated_flightplan(const double acc);
  time_type get_crash(const point_1d *x);
};
```

Repare que só precisamos dizer que o ponto unidimensional tem a sua coordenada dada por um polinômio. Os métodos que aplicam mudanças no plano de vôo são `change-flightplan`, que recebe um novo polinômio, e `accelerated-flightplan`, que recebe a nova aceleração e calcula o novo polinômio de grau 2 com coeficiente dominante igual ao parâmetro recebido de forma que não aconteça alteração descontínua na estrutura.

Para definir a estrutura de dados do torneio cinético, devemos implementar as subclasses de `kds-base`, `item-base` e `cert-base`. Vamos começar definindo a classe `kt-kds` que é subclasse de `kds-base`.

```
// estrutura para uma partida do torneio
typedef struct CelTorn {
  kt_cert *cert;                        // certificado associado
  kt_item *item;                        // item associado
  CelTorn *parent, *lchild, *rchild;    // links no torneio
  int height;                           // altura do nó no torneio
} CelTorn;

// Define a estrutura do torneio cinético
class kt_kds : public kds {
 private:
  CelTorn *max;            // o que queremos manter
  CelTorn *last;           // ultimo elemento inserido no torneio
  bool cert_mounted;       // diz se os certificados já foram criados

  Maximum *get_maximum(kt_item **pts, int length, int exp);
  void del_tournament_node(CelTorn *m);
  void add_in_tournament(kt_item *i);
  void remove_from_tournament();

 public:
```

```
  kt_kds(event_queue *q);
  point_1d *maximum();
  void add_object(obj_base *);
  void add_object(char *id, polynomial flightplan);
  void change_flightplan(char *id, polynomial new_flightplan);
  void remove_object(char *id);
  void create_certificates();
};
```

kds-base contém a lista de todos os objetos, itens e certificados que possui, então basta que kt-kds se preocupe em manter a estrutura do torneio. Para isso criamos uma estrutura CelTorn que representa uma partida do torneio e criamos uma árvore com vários CelTorn como nós internos para representar o torneio inteiro. Dentro dessa estrutura mantêm-se apontadores para o certificado associado à essa partida, para o item vencedor da partida e para as outras partidas do torneio relacionadas a essa partida: o pai e os filhos direito e esquerdo na árvore do torneio. Um objeto kt-kds mantém um apontador para a raiz do torneio (max), sendo capaz de responder a uma consulta pelo máximo a qualquer instante em tempo constante, além de definir os métodos que cuidam das atualizações internas no torneio. Porém toda a parte de buscar por itens e certificados internas já foi definida por kds-base que é a superclasse de kt-kds.

```
class kt_item : public item_base {
 public:
  // aponta para a folha do torneio que contem este item
  CelTorn *maximum;

  kt_item(kt_kds *k, point_1d *p);
  void flightplan_changed();
};
```

A classe kt-item é a extensão de item-base para o torneio cinético. Tudo o que precisa ser definido para ela é qual a partida associada a esse item no torneio, ou seja, qual estrutura CelTorn está associada à esse item, e o método flightplan-changed responsável pelas alterações em mudanças no plano de vôo deve ser implementado. Esse método deve ser capaz de informar à estrutura (uma instância de kt-kds) qual partida pode ter o seu resultado alterado e, nesse momento, a estrutura deve ser capaz de se auto-consertar através da chamada de um método interno da estrutura (não visível para o cliente).

```
class kt_cert : public cert_base {
 public:
  CelTorn *max;
```

```
    kt_cert(kt_kds *k, CelTorn *m);
    time_type calc_death_time() const;
    void process_death();
};
```

A classe `kt-cert` contém a definição do certificado permitido na estrutura, sendo uma subclasse de `cert-base`. Cada certificado está associado a uma partida do torneio (estrutura `CelTorn`) e contém informação sobre quando o resultado dessa partida deixa de ser válido, ou seja, o instante em que os dois pontos envolvidos estarão na mesma coordenada. Portanto, devemos definir dois métodos: um responsável pelo cálculo do prazo de validade do certificado e outro que notifica o objeto `kt-kds` de que a validade do certificado expirou, informando qual o nó do torneio que está com problemas.

Voltando à classe `kt-kds`, devemos criar o mecanismo de atualização no plano de vôo de um ponto dentro da estrutura. Esse mecanismo é proporcionado na forma de um método público `change-flightplan` que recebe o identificador do objeto e o novo polinômio que regerá o plano de vôo desse objeto. Esse método será responsável por procurar o objeto na estrutura e atualizar todos os nós do torneio em que um item associado apareça. Repare que se o polinômio alterar a posição do objeto geométrico na ordenação entre os objetos, a chamada a esse método constitui uma alteração descontínua (veja Capítulo 1) cuja execução representa uma remoção seguida de uma inserção em outro local na fase inicial do torneio. Então, como a estrutura também é dinâmica, devemos criar os mecanismos de inserção e remoção de objetos na estrutura. O mecanismo de inserção também é implementado através de um método público `add-object` que recebe um novo identificador e um polinômio, cria o novo objeto e o insere no torneio, criando um novo item associado para cada partida em que o objeto apareça. O método de remoção `remove-object` faz o trabalho inverso.

Em todos os casos, os identificadores devem ser válidos, ou seja, no caso da remoção ou da alteração do plano de vôo de um determinado objeto, é necessário que exista um objeto na estrutura que possua esse identificador. E no caso da inserção de um novo objeto, devemos examinar se esse identificador já não é usado por algum outro objeto na estrutura. Para isso, em nossas implementações, a aplicação cliente mantém uma árvore com os objetos que estão na estrutura, tornando mais eficiente a busca pelos seus identificadores.

7.2.2 Heap cinético

Agora vamos apresentar a implementação do heap cinético para o problema do máximo. O tipo de objeto utilizado é exatamente o ponto unidimensional `point-1d` descrito na seção anterior. Vamos omitir as interfaces das classes estendidas por elas serem bastante parecidas com as interfaces para o torneio cinético.

Ao criar a estrutura de dados para o heap cinético, a exemplo de qualquer outra

estrutura de dados cinética, também devemos implementar as subclasses de kds-base, item-base e cert-base. A classe kh-kds é a subclasse de kds-base para o heap cinético. Para a construção do heap usamos uma estrutura CelHeap que representa cada nó do heap. Como o heap foi implementado em um vetor, cada CelHeap c possui um campo que guarda o índice do vetor onde c está, além de um apontador para o item (uma instância de kh-item) que está armazenado nesta célula e um apontador para cada um dos três certificados que forma com a célula pai e as duas células filhas. As células pai, filha direita e filha esquerda são encontradas facilmente através de uma pequena conta com o índice da célula. Se a célula c está na posição i do vetor, seu pai estará na posição $\lfloor i/2 \rfloor$ e seus filhos nas posições $2i$ e $2i + 1$ do vetor. Um objeto kh-kds mantém um apontador para a raiz do heap, sendo capaz de informar o máximo do conjunto em tempo constante a qualquer instante. Os mecanismos de atualizações nos planos de vôo, inserções e remoções de objetos são análogos aos já descritos para o torneio cinético.

A classe kh-item é a extensão de item-base para o heap cinético. Tudo o que precisa ser definido para ela é qual o índice da célula no vetor heap que está associada a esse item, além do método responsável pelas alterações no heap quando ocorre mudanças no plano de vôo.

A classe kh-cert contém a definição do tipo de certificado permitido na estrutura, sendo uma subclasse de cert-base. Cada certificado está associado a uma aresta do heap, bastando saber quais os índices das células que o compõem. Dois métodos devem ser definidos: um que calcula o tempo de vida do certificado e outro que define as atualizações a serem feitas quando esse tempo de vida se esgota e o certificado provoca um evento, informando ao objeto kh-kds qual aresta do heap provocou o evento.

7.2.3 Lista cinética

Por fim apresentaremos a implementação da lista cinética para o problema do máximo. Novamente o tipo de objeto utilizado é o mesmo ponto unidimensional point-1d usado no torneio cinético e no heap cinético.

A classe kl-kds é a subclasse de kds-base para a lista cinética. Na declaração da lista usamos uma estrutura CelLista que representa cada célula da lista. Cada CelLista contém dois apontadores para os certificados que forma com cada célula vizinha na lista, além de um apontador para o item presente nesta célula da lista (uma instância de kl-item). Um objeto kl-kds mantém um apontador para o início da lista, sendo capaz de informar o máximo do conjunto em tempo constante a qualquer instante. Os mecanismos de atualizações nos planos de vôo, inserções e remoções de objetos são análogos aos utilizados para o torneio e o heap cinéticos.

A classe kl-item é a extensão de item-base para a lista cinética. Tudo o que ela precisa saber é onde a célula CelList associada está na lista. Isso pode ser feito através de um apontador. O método responsável pelas alterações na lista quando

ocorrem mudanças no plano de vôo também deve ser definido na classe `kl-item`.

A classe `kl-cert` contém a definição do tipo de certificado permitido na lista cinética, constituindo uma subclasse de `cert-base`. Cada certificado está associado a uma vizinhança na lista. Então basta que cada certificado saiba quais células o compõem através de apontadores para os elementos `CelList` correspondentes. Novamente, dois métodos devem ser definidos: um que calcula o tempo de vida do certificado e outro que define as atualizações na estrutura que devem ser realizadas quando esse o certificado provoca a ocorrência de um evento, informando ao objeto `kh-kds` qual vizinhança na lista está com problemas.

7.3 Exemplo de Outra Implementação

Uma vez que descrevemos o nosso método de implementação de estruturas de dados cinéticas e como ele foi utilizado para construirmos as estruturas para o problema do máximo, vamos expôr brevemente um exemplo de como é a implementação da estrutura de dados cinética para o par de pontos mais distante (Seção 5.5.1) segundo a estratégia de implementação desenvolvida nas seções anteriores.

Sabemos que a estrutura para o par mais distante usa a estrutura do fecho convexo (na verdade a estrutura do envelope superior) cinético como sub-estrutura. Portanto vamos planejar a implementação dessa estrutura primeiro.

O tipo de objeto utilizado para o problema do fecho convexo é o ponto bidimensional, cujas coordenadas estão no plano cartesiano. A trajetória do ponto em função do tempo é descrita por um par de polinômios, cada um responsável pelo movimento em uma das dimensões cartesianas: x ou y. A implementação tem que definir a classe `point-2d` como uma extensão da classe base `obj-base`. O construtor de `point-2d` receberá dois polinômios que formam o plano de vôo inicial do objeto, e um identificador, que serve para diferenciarmos cada um dos pontos na estrutura.

Vamos passar para a fase de planejamento da estrutura de dados cinética para o envelope superior. Para obter o fecho convexo, basta construir um cliente que receba os atributos mantidos por duas estruturas semelhantes: uma para o envelope superior e outra para o envelope inferior e faça a concatenação dos dados recebidos.

A classe `kse-kds` é a subclasse de `kds-base` nesse caso. Devemos implementar aqui todo o esquema de interação entre os pontos, bem como o processo de dualização (ida e volta) e o algoritmo estático que obtém o envelope inicial e constrói os certificados iniciais. Nesta parte da implementação fica a declaração de todas as estruturas internas.

A classe `kse-item` é a extensão de `item-base`. Nesta parte fica a implementação da comunicação entre um item e a estrutura, caracterizada pelo aviso de que um ponto teve o seu plano de vôo alterado à estrutura, de modo que ela possa ser consertada para continuar com o resultado correto e recalcular alguns certificados.

Nesta implementação temos vários tipos diferentes de certificados. Cada um deles

deve ser definido como uma subclasse diferente de `cert-base`. Cada tipo de certificado contém informação diferente e age diferentemente sobre a estrutura de dados quando perde a sua validade. Devemos definir como se calcula o prazo de validade e como comunicamos a estrutura central `kse-kds` no caso de que o prazo de validade seja alcançado. Provavelmente diferentes métodos da estrutura central sejam chamados por cada tipo de certificado para que sejam feitas as devidas atualizações.

Uma vez implementada a estrutura para o envelope superior cinético, devemos implementar o torneio de pares opostos para manter o par de pontos mais distante. Nesta estrutura o tipo de objeto utilizado seria um par oposto. A lista completa de pares opostos pode ser obtida através de um cliente que faz requisições às estruturas dos envelopes. A seguir devemos plenejar a implementação das subclasses de `kds-base`, `item-base` e `cert-base` para o torneio de pares opostos. Essa implementação pode aproveitar muito do esforço de implementação do torneio cinético para pontos unidimensionais apresentado na seção anterior.

7.4 Detalhes de Implementação

Nesta seção discutiremos alguns detalhes que apareceram no momento da implementação e que julgamos possuírem certa relevância como o cálculo de raízes de polinômios e as estruturas de dados auxiliares utilizadas.

Nos testes realizados sobre as implementações para o problema do máximo, apenas consideramos movimentos quadráticos que simulam movimentos físicos reais, sendo dominados pela aceleração. Portanto, em nossas implementações não precisaríamos calcular raízes de polinômios de grau maior que 2. Porém, implementamos um método de cálculo numérico para obtenção das raízes reais de polinômios de qualquer grau devido à dois motivos.

O primeiro motivo diz respeito à meta de apresentar um código que fosse geral o suficiente para dar suporte a movimentos regidos por polinômios de qualquer grau. O segundo motivo refere-se a prováveis extensões de nossa implementação. Suponha uma implementação da estrutura de dados cinética para o par mais próximo que constantemente calcula distâncias entre pares de pontos. Os cálculos de distância utilizam o quadrado das coordenadas dos pontos, e essas por sua vez são definidas por polinômios. Supondo que estamos restringindo o tipo de movimento para movimentos quadráticos, deveríamos ser capazes de calcular raízes de polinômios de grau 4. Outro caso seria a implementação de uma estrutura de dados cinética para o Diagrama de Voronoi. Essa estrutura necessita de testes `InCircle` entre 4 pontos que calcula o determinante de uma matriz 4×4 com 3 coordenadas definidas por polinômios e 1 constante por linha. O cálculo desse determinante precisa de um método que obtenha as raízes de polinômios de grau 6.

Para encontrar as raízes de polinômios de grau qualquer (maior que 2), usamos a implementação do método de Laguerre como implementado em "Numerical Recipes

in C" [PFTV93]. Esse método retorna todas as raízes, reais ou complexas, de um polinômio de qualquer grau. Talvez essa não seja a solução mais eficiente, mas é uma garantia de que nenhuma raiz seja perdida, por mais próximas que elas estejam. Porém, nem todos os problemas acabaram com a utilização desse método, pois ele não encontra todas as raízes corretamente se o polinômio estiver mal-condicionado[1]. Mesmo esse problema pode ser parcialmente contornado se multiplicarmos os coeficientes do polinômio por um fator suficientemente grande de modo que o mal-condicionamento desapareça.

O cálculo do prazo de validade de um certificado entre dois objetos a e b é feito da seguinte forma: dados os planos de vôo $y = p_a(t)$ e $y = p_b(t)$ e o instante t_0 atual, calculamos as raízes do polinômio diferença $p_a(t) - p_b(t)$ que têm valores maiores que t_0 a fim de evitar que o processamento de eventos possa causar loop infinito. O prazo de validade do certificado será a menor raiz do polinômio diferença que seja maior que t_0.

Vale a pena comentar também o tratamento aplicado às mudanças de plano de vôo dos objetos sem que ele saia do lugar, evitando alterações descontínuas. Nesse tipo de alteração assumimos sempre que o grau máximo do novo polinômio é 2. Ele será da forma $x = x_0 + v_0t + at^2/2$ que é a equação da posição em função do tempo aplicada em movimentos uniformemente variáveis na Física. x_0 é a posição no instante inicial $t = 0$ e v_0 a velocidade nesse mesmo instante.

O cliente fornece o valor da nova aceleração a e a nova equação é calculada internamente pelo programa. A estratégia adotada é executada em alguns passos. No primeiro passo calcula-se a posição x atual do objeto aplicando a equação $P(t)$ do movimento atual. Em seguida obtém-se o valor da velocidade v atual através da derivada da equação do plano de vôo atual $P'(t)$. A seguir devemos obter os coeficientes x_0, v_0.

Outra fórmula usada em Física para obtermos a velocidade em função do tempo em movimentos uniformemente variados é a equação $v = v_0 + at$ que, por sinal, corresponde à derivada da equação da posição em função do tempo. Sabemos qual é o instante e a velocidade atual, além da nova aceleração, então podemos calcular v_0 usando essa fórmula.

Para calcular x_0 podemos utilizar a equação da posição em função do tempo, pois sabemos os valores da posição atual x, da velocidade inicial v_0 calculada no passo anterior, do instante atual e a nova aceleração. A seguir basta montar a equação do novo plano de vôo $x(t) = p_0 + p_1t + p_2t^2$, onde os coeficientes são: $p_0 = x_0$, $p_1 = v_0$ e $p_2 = a/2$, e substituir a equação do plano de vôo atual pela nova. Esse último passo finaliza a troca de plano de vôo de um objeto. A seguir devemos atualizar a estrutura de certificados para manter a corretude da estrutura de dados cinética.

Outro aspecto referente à implementação que merece algum destaque é o uso de es-

[1]Um polinômio é mal-condicionado se existe grande discrepância entre os valores absolutos de seus coeficientes. Por exemplo, $1000x^2 + 2x + 66$ mostrou ser um polinômio mal-condicionado em nossos testes, enquanto $40x^2 - 12x + 27$ não.

truturas de dados auxiliares na implementação. Em nossas implementações utilizamos árvores, filas de prioridade e heaps.

Implementamos tais estruturas de dados utilizando templates, baseando-se na Standard Template Library que figura nas bibliotecas padrões para programação em C++ presentes em qualquer plataforma. Dessa forma podemos declarar estruturas genéricas que podem conter qualquer tipo de dado como elemento. A árvore balanceada implementada foi a árvore AVL que admite inserções, remoções, buscas, *splits* e *joins* em tempo $O(\log n)$. O heap foi implementado seguindo as idéias expostas no livro [CLR90]. A fila de prioridade foi implementada sobre a árvore AVL para dar suporte à remoções eficientes de qualquer elemento (não apenas do máximo como acontece em filas de prioridade que utilizam heap). Isso ocorreu porque ao processar mudanças no plano de vôo de objetos ou processar eventos, em geral precisamos alterar certificados que não estão na frente na fila de eventos. Esse procedimento envolve remoções e inserções desses certificados e daí o motivo da utilização da fila de prioridade sobre a nossa implementação da árvore AVL. Para as implementações de todas as estruturas de dados auxiliares utilizamos os livros de Knuth [Kn73], Cormen [CLR90], Musser [MS96], Sedgewick [Se98] e Waite [Wa98].

Vale a pena citar um problema que aparece na fase de implementação e que ocorre no processamento de eventos simultâneos na lista cinética. Pudemos observar que poucas vezes mais de um evento ocorre no mesmo instante e é muito difícil que tais eventos ocorram com pontos em comum. Porém esses casos existem e devem ser tratados corretamente.

No caso de eventos simultâneos sem objetos em comum, podemos processá-los em qualquer ordem que a estrutura não apresenta problemas. Já para o caso de eventos em que há objetos em comum decidir o que fazer se torna um problema de difícil resposta. Qual evento deve ser processado em primeiro lugar de modo que a estrutura permaneça correta?

Imagine três objetos a, b e c cujos planos de vôo são regidos, por exemplo, pelas equações $a : y = t$, $b : y = 10$ e $c : y = -t + 20$. No intervalo de tempo $[0, 10[$ a ordem entre os objetos é dada por c, b, a e dois eventos estão programados para o instante $t = 10$. Nesse instante o certificado existente entre a e b e o certificado entre b e c perdem a sua validade ao mesmo tempo e têm o ponto b em comum. O que fazer? Note que os três objetos têm a mesma posição nesse instante e que haverá inversão na ordem dos três objetos no instante $t + \epsilon$. O processamento comum de eventos disjuntos não se aplica, trazendo erros à estrutura.

A solução adotada foi a de remover todos os eventos simultâneos da fila de eventos e processá-los como se fossem um só. Nesse momento ordenamos todos os objetos envolvidos para o instante $t + \epsilon$ (pois no instante t eles têm a mesma posição) e recalculamos todos os certificados entre cada par de vizinhos do segmento recém ordenado. Essa abordagem resolve o problema.

Um problema parecido acontece na lista cinética quando um certificado entre a e b

perde a sua validade e os próximos k elementos após a possuem planos de vôo iguais ao de a. Nesse caso, mesmo tratando-se de um evento não simultâneo com qualquer outro, a simples inversão de posições entre os objetos a e b que é feita no processamento comum de eventos traz erros à estrutura pois a ordem entre os elementos apresentará erros. O certo seria passar b para o final do segmento de $k+1$ elementos com plano de vôo igual ao de a e continuar com o algoritmo. A estratégia apresentada no parágrafo anterior resolve o problema.

Esses problemas de implementação ocorrem apenas pelo motivo de calcularmos o prazo de validade de um certificado como a primeira raiz do polinômio diferença entre os planos de vôo dos objetos envolvidos maior que o instante t_0 onde calculamos o tempo de vida do certificado. Talvez se permitíssemos que um certificado tivesse um tempo de vida nulo, ou seja, fosse criado, inserido e removido da fila de eventos e processado no mesmo instante t_0 após processarmos alguns outros eventos simultâneos os problemas descritos acima não ocorressem, porém aumentaríamos as chances de o programa entrar em loop infinito.

7.5 Testes

Após a implementação das estruturas para o problema do máximo o próximo passo foi o de realizar experiências com tais estruturas a fim de descobrir qual o comportamento médio das estruturas implementadas para diferentes entradas e confrontar os resultados obtidos com os resultados teóricos expostos no Capítulo 3. Para a realização dos testes foi construído um programa gerador de testes que cria entradas (clientes) para as estruturas de dados cinéticas no formato exibido na Seção 7.2.

O programa construtor de testes recebe diversos parâmetros que configuram o aspecto do cliente que deve ser criado. Os parâmetros configuráveis são: o número de pontos no instante inicial, o número máximo de ações num mesmo instante da simulação, o tamanho máximo do conjunto de pontos, o valor máximo (em módulo) para os coeficientes dos polinômios, o incremento máximo e mínimo entre dois instantes em que existem ações a serem executadas e o instante com a última ação na entrada. Dentro desses parâmetros o programa sorteia aleatoriamente todo o conteúdo de um arquivo de teste.

Dividimos os testes em três categorias distintas: testes apenas consulta, testes cinéticos e testes completos. Nos testes do tipo consulta a única ação possível em instantes com ações é a ação de `consulta` que pede à estrutura de dados cinética que devolva o máximo atual do conjunto. Testes cinéticos admitem dois tipos de ações: `consulta` e `acelera`, onde o plano de vôo dos objetos pode ser alterado a qualquer instante. Testes completos adicionam ações dinâmicas: `insere` e `remove`.

Para cada categoria geramos 50 testes diferentes para os seguintes tamanhos iniciais de conjuntos: 10, 50, 100, 250, 500, 1000, 2000, 5000 e 10000 pontos cujos planos de vôo são sempre polinômios de grau menor igual a 2, simulando movimento uniforme

(grau 1), movimento uniformemente variado (grau 2) e posição estática (grau 0). Os parâmetros usados no programa de geração dos testes foram escolhidos aleatoriamente dentro dos seguintes intervalos: valor máximo dos coeficientes: entre 20 e 200, incremento mínimo entre os instantes com ações: entre 0.2 e 1, incremento máximo: entre 2 e 10. Para os testes com até 1000 elementos fixou-se o último instante com ação como o instante 200. Nos testes maiores fixamos o instante 400 como o último instante. O tamanho máximo dos conjuntos foi limitado no dobro do seu tamanho inicial.

Em cada execução, após o último instante com ação a estrutura de dados cinética utilizada processa todos os eventos que ainda estão na fila de eventos até que o conjunto de pontos atinja um estado de equilíbrio em que não é previsto nenhum outro evento. A partir dos testes de tamanho 2000 executamos os testes gerados apenas para as estruturas do heap e do torneio cinético. Para os testes até tamanho 1000 executamos também a estrutura da lista cinética. Um mesmo teste foi executado para cada estrutura de dados cinética disponível. A cada teste executado, contamos o total de eventos internos, de eventos externos, o tamanho máximo alcançado pela fila de eventos e o total de alterações na estrutura decorrentes de ocorrência de eventos, mudança em planos de vôo ou inserções e remoções. Com esses dados podemos avaliar cada estrutura em relação aos critérios de análise de desempenho introduzidos pelo modelo cinético: resposta rápida, localidade, compacidade e eficiência.

Testes tipo consulta

Vamos começar com os testes da categoria consulta. Nesse tipo de teste não existe qualquer alteração no conjunto de dados desde o instante inicial. Os eventos vão se sucedendo até que se alcance o estado de equilíbrio do conjunto de pontos. As informações relevantes para testes desse tipo, portanto, são as que referem-se aos eventos: o número de eventos internos, de eventos externos, de mudanças na estrutura relativas à ocorrência de eventos e o tamanho da fila de eventos. A Tabela 7.1 mostra a média dos resultados obtidos em 50 testes diferentes para cada tamanho de conjunto.

Quanto ao tamanho máximo da fila de eventos, podemos notar que a proporção dos resultados obtidos (coluna $max(|F|)$) em função do tamanho do conjunto é praticamente igual para todas as estruturas, ficando sempre perto de $n/2$. Esse resultado parece suficiente para comprovar que as três estruturas possuem compacidade $O(n)$.

Gostaríamos de verificar a eficiência teórica das estruturas segundo o modelo cinético através da coluna $|E_{tot}|/|E_{ext}|$ que contém o quociente entre o número total de eventos e o número de eventos externos. Porém, o modelo define eficiência como o quociente entre o pior caso para o total de eventos e o pior caso para o número de eventos externos. Como é difícil gerar um teste aleatório para pegar o pior caso e além disso estamos considerando a média de 50 testes para cada valor de n, parece que esse tipo de análise não se tornou possível dentro do nosso esquema de testes. Para conseguir medir satisfatoriamente a eficiência das estruturas precisaríamos construir testes em que se garantisse que o número de vezes em que o máximo do conjunto muda fosse

kds	n	$\|E_{tot}\|$	$\|E_{ext}\|$	$\|A_{evt}\|$	$max(\|F\|)$	$\|E_{tot}\|/\|E_{ext}\|$
Heap	10	6.20	1.80	18.60	4.56	3.44
Torneio	10	6.64	1.80	12.08	4.48	3.68
Lista	10	22.56	1.80	22.52	5.04	12.53
Heap	50	44.36	2.88	133.08	24.36	15.40
Torneio	50	33.84	2.88	76.64	24.12	11.75
Lista	50	617.64	2.88	617.12	29.12	214.46
Heap	100	99.64	3.48	298.92	51.32	28.63
Torneio	100	68.68	3.48	160.00	48.48	19.73
Lista	100	2585.28	3.48	2583.36	60.24	742.90
Heap	250	266.32	4.68	798.96	124.12	56.90
Torneio	250	172.92	4.68	415.52	122.96	36.95
Lista	250	16140.48	4.68	16127.92	149.32	3448.82
Heap	500	553.84	5.28	1661.52	251.68	104.89
Torneio	500	356.20	5.28	860.28	248.76	67.46
Lista	500	65208.76	5.28	65163.88	297.36	12350.14
Heap	1000	1109.40	5.56	3328.20	499.24	199.53
Torneio	1000	700.84	5.56	1702.72	492.96	126.05
Lista	1000	251652.92	5.56	251516.48	592.16	45261.32
Heap	2000	2259.16	6.32	6777.48	999.64	357.46
Torneio	2000	1398.36	6.32	3416.40	990.44	221.26
Heap	5000	5636.08	6.88	16908.24	2501.80	819.19
Torneio	5000	3474.76	6.88	8482.44	2461.00	505.05
Heap	10000	11350.96	7.24	34052.88	5001.12	1567.81
Torneio	10000	6942.08	7.24	17035.64	4932.64	958.85

Tabela 7.1: Tabela com a média dos dados de 50 testes do tipo consulta para cada valor de n, sendo n o número de pontos no conjunto, E_{tot} o conjunto de todos os eventos, E_{ext} o conjunto dos eventos externos, A_{evt} o conjunto das atualizações na estrutura proporcionadas pela ocorrência de eventos e F a fila de eventos.

$O(n)$.

Quanto ao número total de eventos, podemos notar que a lista cinética apresenta variação quadrática em relação ao tamanho do conjunto de dados. O torneio cinético apresenta variação linear, ao contrário do limite de pior caso que é $O(n \log n)$. Novamente trata-se de um problema com os testes para pior caso. Esse problema também ocorre com o total de eventos para o heap cinético. Os resultados obtidos para o heap apresentaram variação aproximadamente linear (veja a Figura 7.2).

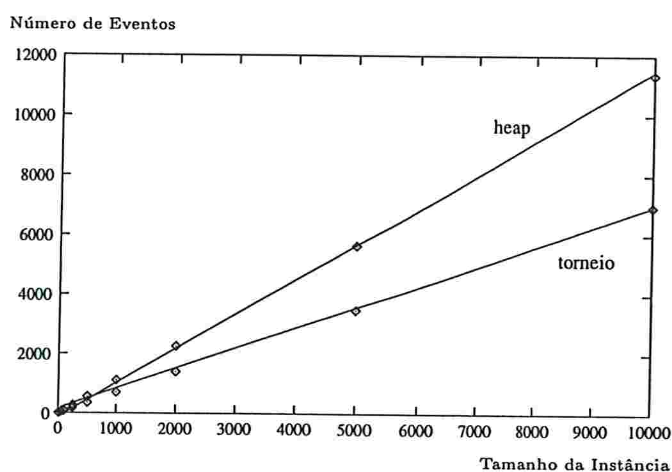


Figura 7.2: Gráfico que mostra os resultados obtidos para o número total de eventos em função do tamanho da instância para o Heap e o Torneio.

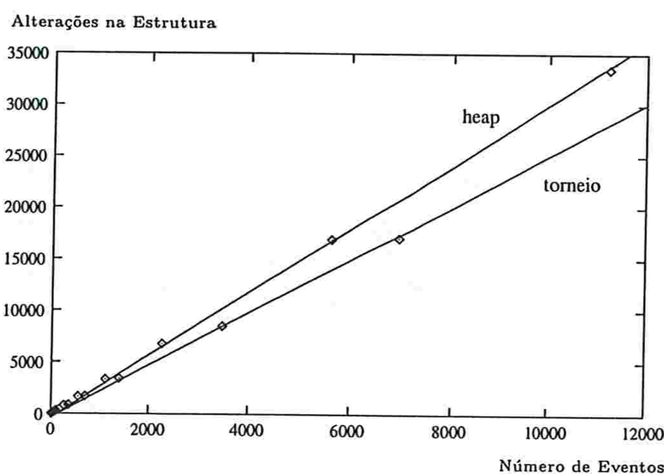


Figura 7.3: Gráfico que mostra os resultados obtidos para o número de alterações na estrutura em função do número total de eventos para o Heap e o Torneio.

Para verificar a qualidade de resposta rápida das estruturas devemos analisar a coluna $|A_{evt}|$ que mostra a média do número total de trocas entre elementos na estrutura. A

lista cinética apresenta variação quadrática em relação ao número de elementos no conjunto, mas apresenta variação proporcional ao número total de eventos. Na ocorrência de um evento na lista cinética, tudo o que deve ser feito é uma simples troca entre os objetos responsáveis pelo evento[2]. Assim, podemos concluir que a lista cinética gasta tempo constante para processar um evento confirmando o resultado teórico apresentado no Capítulo 3, sendo ela uma estrutura de resposta rápida.

O número de alterações na estrutura proporcionado pelo heap cinético também é diretamente proporcional ao número de eventos. Portanto o heap cinético também é uma estrutura de resposta rápida. Isso acontece devido à própria natureza do heap cinético que precisa fazer um número constante de trocas entre elementos do heap ao processar a ocorrência de um evento.

Já para o torneio cinético a história é um pouco diferente. O quociente entre o número de alterações na estrutura e o número total de eventos praticamente se estabiliza a partir de $n = 250$. Mas, baseado nos resultados teóricos, isso não significa que o tempo de processamento de um evento seja constante, e sim que estamos em mais uma situação enganosa de pior caso. Sabemos que, no pior caso, o processamento de um evento causa $O(\log n)$ trocas na estrutura do torneio. Porém, os resultados obtidos nos mostram que na prática esse número de trocas varia, em média, linearmente com o tamanho do conjunto de dados (veja o gráfico da Figura 7.3). Isso nos leva a crer que os eventos que afetam $O(\log n)$ nós do torneio acontecem raramente na prática.

As características dos testes do tipo consulta não fornecem dados para avaliarmos a localidade das estruturas, que está relacionada com o número de alterações na estrutura em virtude de mudanças nos planos de vôo dos objetos. Como os testes do tipo consulta não permitem alterações no plano de vôo, nossa análise de localidade será feita com os testes do tipo cinético.

Testes tipo cinético

Passemos agora para os testes do tipo cinético, onde permitimos que o plano de vôo dos objetos possa ser alterado a qualquer instante. Desta vez temos a possibilidade de ocorrerem alterações na estrutura por dois motivos: a ocorrência de um evento ou a troca do plano de vôo de um objeto. As alterações do segundo tipo nos dão subsídios para analisar a localidade das estruturas. A Tabela 7.2 mostra a média dos resultados obtidos em 50 testes diferentes para cada tamanho inicial de conjunto.

Quanto ao tamanho máximo da fila de eventos, nota-se que há um ligeiro crescimento para valores pequenos de n em relação aos testes do tipo consulta, mas que, na medida que n cresce, o comportamento retorna aos mesmos padrões dos mostrados na Tabela 7.1. Isso é explicado pelo motivo de estarmos limitando o número de instantes em que são permitidas ações de mudança no plano de vôo. Para testes pequenos

[2]Salvo raras exceções onde existem mais de dois objetos com o mesmo valor na lista ou eventos simultâneos entre objetos de posições vizinhas.

kds	n	$\|A_{evt}\|$	$\|A_{pv}\|$	$max(\|F\|)$	$\|E_{tot}\|$	$\|M_{pv}\|$
Heap	10	36.88	107.44	6.12	18.44	44.28
Torneio	10	33.16	80.56	6.76	23.36	44.28
Lista	10	70.88	44.28	6.40	72.04	44.28
Heap	50	152.24	147.92	25.24	76.12	45.48
Torneio	50	134.12	88.68	23.80	60.18	45.48
Lista	50	1075.16	45.48	30.32	1083.48	45.48
Heap	100	267.76	140.68	51.16	133.88	43.04
Torneio	100	230.88	82.00	50.52	99.56	43.04
Lista	100	3496.44	43.04	59.60	3508.16	43.04
Heap	250	585.20	136.08	123.08	292.60	42.84
Torneio	250	480.40	86.88	120.84	200.96	42.84
Lista	250	17624.25	42.84	146.54	17647.79	42.84
Heap	500	1156.48	136.80	250.56	578.24	43.84
Torneio	500	931.48	90.40	244.96	386.58	43.84
Lista	500	65945.24	43.84	298.52	65999.56	43.84
Heap	1000	2261.76	130.08	498.80	1130.88	42.80
Torneio	1000	1745.76	84.24	483.92	718.44	42.80
Lista	1000	256031.08	42.80	595.60	256182.68	42.80
Heap	2000	4610.00	258.32	999.12	2305.00	83.72
Torneio	2000	3539.20	166.88	984.60	1454.48	83.72
Heap	5000	11466.48	260.16	2500.12	5733.24	85.72
Torneio	5000	8684.68	172.12	2467.04	3549.08	85.72
Heap	10000	22817.28	244.56	5010.20	11408.64	81.16
Torneio	10000	17188.76	177.96	4927.72	7012.80	81.16

Tabela 7.2: Tabela com a média dos dados de 50 testes do tipo cinetico, sendo n o número de pontos no conjunto inicial, A_{evt} o conjunto de alterações na estrutura por ocorrência de eventos, A_{pv} o conjunto de alterações na estrutura por mudanças nos planos de vôo, F a fila de eventos, E_{tot} o conjunto de eventos e M_{pv} o conjunto de mudanças nos planos de vôo dos pontos.

esse número de alterações no plano de vôo chega a ser relevante para aumentar o tamanho máximo da fila de eventos. Em testes maiores, como o número de mudanças no plano de vôo é limitado (veja a coluna $|M_{pv}|$), essa contribuição fica diluída e o comportamento passa a ser o mesmo que para os testes do tipo consulta.

O mesmo fenômeno é verificado na análise do número total de eventos. Para testes pequenos o número de eventos cresce bastante em relação ao número de eventos em testes do tipo consulta para o mesmo tamanho do conjunto. Em testes pequenos, mudanças nos planos de vôo adiam significativamente a chegada do estado de equilíbrio do conjunto. Em testes maiores o mesmo comportamento só não se verifica porque o número de mudanças nos planos de vôo é limitado na criação dos testes e, conseqüentemente, o número de eventos cresce pouco quando comparado com o número de eventos para os testes do tipo consulta com o mesmo número de pontos. A saída talvez fosse gerar testes em que o número de mudanças no plano de vôo não é limitado por uma constante, mas sim pelo tamanho da instância.

Porém, mesmo com a limitação do número de alterações no plano de vôo, podemos analisar a localidade média das estruturas. Para isso devemos comparar a coluna do número de alterações na estrutura ($|A_{pv}|$) com o número médio de mudanças no plano de vôo (coluna $|M_{pv}|$). Note que o número de mudanças no plano de vôo durante a simulação é limitado e, por esse motivo, o número de alterações no heap cinético e na lista cinética permanecem aproximadamente constante, comprovando na prática a localidade $O(1)$ que essas duas estruturas possuem. Aliás, o número de alterações por plano de vôo na lista cinética é exatamente igual ao número de mudanças no plano de vôo dos objetos. Para o heap, o quociente entre as duas colunas gira em torno de 3.

O torneio cinético possui localidade de $O(\log n)$ no pior caso. Porém, os resultados práticos mostraram que o número de alterações por plano de vôo é constante para as entradas geradas conforme descrevemos no início da seção. Novamente estamos em uma situação enganosa de pior caso. Baseado nos resulados dos testes, podemos dizer que, em média, o número de alterações por mudança no plano de vôo é constante para o torneio cinético, apesar de sabermos que isso não é verdade no pior caso. Isso nos leva a crer que mudanças no plano de vôo de objetos escolhidos aleatoriamente que afetem $O(\log n)$ nós do torneio raramente acontecem na prática.

Testes tipo completo

Apresentamos agora os testes do tipo completo, onde tudo o que era permitido nos testes do tipo cinético continua permitido e também são aceitas inserções e remoções no conjunto de pontos em movimento a qualquer instante. Neste tipo de teste temos a possibilidade de ocorrerem alterações na estrutura de dados devido à essas ações dinâmicas de inserção e remoção. A Tabela 7.3 mostra a média dos resultados obtidos em 50 testes diferentes para cada tamanho inicial de conjunto.

Desta vez não apenas o número de alterações no plano de vôo é limitado como

kds	$n_{inicial}$	n_{tot}	$\|A_{evt}\|$	$\|A_{pv}\|$	$\|A_{din}\|$	$\|E_{tot}\|$	$max(\|F\|)$
Heap	10	35.50	25.04	70.70	390.70	12.53	5.17
Torneio	10	35.50	24.17	49.38	136.38	29.42	5.58
Lista	10	35.50	56.00	28.17	39.83	57.16	5.17
Heap	50	73.47	143.92	79.88	486.33	61.96	25.33
Torneio	50	73.47	107.32	51.68	169.56	53.24	24.36
Lista	50	73.47	860.41	26.82	45.41	864.88	28.59
Heap	100	123.00	305.12	89.00	478.08	122.56	50.16
Torneio	100	123.00	208.48	55.60	169.56	92.16	49.56
Lista	100	123.00	3126.94	28.47	45.65	3135.41	58.53
Heap	250	272.72	835.60	85.28	527.32	287.80	127.32
Torneio	250	272.72	467.16	57.24	187.24	195.64	122.36
Lista	250	272.72	17250.91	28.48	45.35	17270.07	150.04
Heap	500	522.36	1709.88	86.04	512.12	569.96	253.92
Torneio	500	522.36	894.00	56.12	189.24	371.79	246.40
Lista	500	522.36	66072.32	28.00	46.16	66122.92	296.84
Heap	1000	1023.28	3384.84	83.84	526.76	1128.28	497.76
Torneio	1000	1023.28	1738.88	55.60	180.20	715.04	485.92
Lista	1000	1023.28	254197.29	27.21	46.08	254341.92	589.88
Heap	2000	2044.06	6830.76	88.56	503.44	2276.92	1003.36
Torneio	2000	2044.06	3460.80	57.08	183.60	1417.96	986.08
Heap	5000	5043.94	17107.92	91.08	538.12	5702.64	2501.76
Torneio	5000	5043.94	8556.24	61.20	183.12	3501.60	2457.08
Heap	10000	10047.24	34112.04	79.98	514.80	11370.78	5012.76
Torneio	10000	10047.24	17058.64	52.32	176.04	6960.12	4937.48

Tabela 7.3: Tabela com a média dos dados de 50 testes do tipo completo, sendo $n_{inicial}$ o número de pontos no conjunto inicial, n_{tot} o número de objetos envolvidos em toda a simulação, A_{evt} o conjunto de alterações na estrutura por ocorrência de eventos, A_{pv} o conjunto de alterações na estrutura por mudanças nos planos de vôo, A_{din} o conjunto de alterações na estrutura por inclusão/remoção de elementos do conjunto, E_{tot} o conjunto de eventos e F a fila de eventos.

também o número de inserções e remoções no conjunto de pontos. De acordo com a tabela podemos notar que essas operações dinâmicas são baratas para a lista cinética, que apresentou o melhor resultado nesse quesito dentre as estruturas avaliadas.

Repare que houve uma queda no número de eventos para todas as estruturas em relação aos testes do tipo cinético. Isso aconteceu devido ao número de ações ter o mesmo limite para ambos os tipos de teste e, nos testes do tipo cinético não existirem ações de inserção e remoção, ocorrendo mais ações de mudança no plano de vôo. A disputa de um número maior de ações por um mesmo espaço nos testes foi responsável pela queda no número de eventos. Porém, de um modo geral, o número total de eventos permanece maior que nos testes do tipo consulta.

7.6 Considerações Finais

A criação de um modelo de implementação para estruturas de dados cinéticas parece ser uma boa idéia para facilitar o trabalho de implementação de estruturas mais complexas. Nossa idéia foi a de estabelecer uma biblioteca básica funcional e procedimentos de extensão que fossem fáceis de serem seguidos na criação de novas estruturas. Os resultados nos mostram que o objetivo parece ter sido alcançado.

Com relação aos testes, o ideal seria utilizar testes reais, mas não dispomos de tais testes. A saída encontrada foi gerar alguns testes que apresentam comportamento próximo ao que julgamos ser o comportamento de testes reais e aplicá-los às estruturas implementadas. É difícil criar testes que possam servir como um bom parâmetro para a avaliação dos diversos atributos contáveis na execução de uma simulação através de estruturas de dados cinéticas. Não é fácil comparar significativamente estruturas de dados cinéticas porque parece que sempre é possível construir um exemplo que beneficia o desempenho de uma estrutura em relação a outra e vice-versa. Pareceu-nos que o melhor a fazer seria gerar testes aleatórios dentro de certos parâmetros e manter o grau do polinômio de movimento dos pontos menor ou igual a dois, englobando apenas os movimentos que ocorrem na prática.

O problema com os testes permanece se queremos analisar o pior caso. Era desejável comprovar a eficiência das estruturas segundo o modelo cinético através dos testes, mas é difícil gerar testes de pior caso. Porém, testes empíricos dificilmente mostram o pior caso. Aliás, eles em geral mostram que as hipóteses de pior caso não acontecem freqüentemente, mas acontecem.

Mesmo com esses problemas, acreditamos que nossos testes servem para simular o efeito de aplicações práticas com as estruturas apresentadas para o problema do máximo. Dentro desse contexto, os testes demonstraram que a lista cinética não é uma boa estrutura de dados cinética para o problema do máximo devido ao grande número de eventos (quadrático) internos que ela apresenta, e que a escolha do torneio cinético parece ser a mais indicada, embora o heap cinético tenha vantagem teórica sobre o torneio quando o número de alterações nos planos de vôo não é limitado.

Capítulo 8

Considerações Finais

8.1 Conclusões

O modelo estudado introduz uma aproximação on-line para manutenção de um atributo geométrico de um conjunto de objetos que se movem continuamente com o tempo. Um ponto favorável importante é que com essa modelagem podemos evitar discretizações do tempo e manter a estrutura de dados correta todo o tempo preocupados apenas com os instantes que são efetivamente relevantes no decorrer do tempo.

Monitorar apenas esses instantes relevantes é a grande idéia por trás do modelo cinético apresentado. Apesar de os objetos estarem em movimento contínuo ao longo do tempo e, conseqüentemente, o atributo geométrico que está sendo monitorado também estar sendo alterado continuamente, a estrutura combinatória que descreve tal atributo só se altera em alguns instantes especiais do tempo. Portanto, a uma estrutura de dados cinética basta que sejam monitorados apenas esses instantes especiais que produzem mudanças significativas na descrição combinatória do atributo geométrico desejado.

Através dos problemas apresentados ao longo da dissertação é possível notar a generalidade do processo de cinetização que transforma um algoritmo que resolve um problema e devolve um atributo geométrico no cenário estático em uma estrutura de dados cinética que mantém o mesmo atributo geométrico no cenário cinético. Vimos também que, em certos casos, é possível estender os resultados também para o cenário dinâmico de forma eficiente como no caso do problema do máximo e do par mais próximo.

Na posição inicial é preciso que se resolva o problema estático a fim de obtermos a solução para o problema "antes de ser dada a largada". Particularmente, o processo de cinetização começa com uma prova da corretude do atributo geométrico no cenário estático e então analisa uma "animação" dessa prova através do tempo. Porém, segundo o modelo, existem provas (certificados) boas, ou seja, que se encaixam nos quatro critérios de análise propostos, e outras ruins, dependendo do algoritmo empregado para resolver o problema estático inicial. Por esse motivo, torna-se necessário, às

vezes, criar um novo algoritmo não tão intuitivo para resolver o problema estático de tal forma que os certificados fornecidos formem uma boa prova (no sentido do modelo) da corretude da solução para que possamos aplicar esse resultado ao cenário cinético. Os algoritmos que resolvem o problema estático inicial em geral são, portanto, cruciais para a cinetização posterior pois eles são os responsáveis pelos certificados iniciais. Devem, então, ser desenvolvidos com esse fim.

Vale ressaltar que existem outros modelos diferentes para problemas que envolvem objetos em movimento, tais como o modelo *off-line* proposto por Atallah [At83, At85] na década de 1980 que considera objetos com movimentos predeterminados (não se alteram) ou o modelo de *tempo real* tratado por Kahan [Ka91, Ka92] no início da década de 1990 que considera que uma consulta sobre o atributo geométrico desejado tenha um certo prazo limite para ser exibida. Não é nosso objetivo aqui apontar as vantagens do modelo cinético analisado sobre os demais, uma vez que as qualidades de um ou outro são destacadas na solução de problemas vindos de situações do mundo real.

Uma clara desvantagem do modelo cinético apresentado torna-se evidente ao considerarmos o caso de aplicações em tempo real onde não existe tempo suficiente para o processamento completo de um evento antes que o próximo evento aconteça. O ideal seria que tal processamento fosse instantâneo, o que, infelizmente, é impossível. O desenvolvimento de estruturas de dados cinéticas que sigam este modelo e que dêem suporte ao processamento de eventos em paralelo, mantendo estruturas parcialmente corretas pode ser uma alternativa para amenizar o problema.

Podemos destacar a possibilidade de uma estrutura de dados cinética ser utilizada como sub-estrutura de uma outra estrutura de dados cinética maior. Apresentamos o problema do fecho convexo cuja estrutura de dados cinética é formada por duas estruturas que mantêm os envelopes superior e inferior no plano dual. No caso do par de pontos mais próximo, temos um torneio cinético entre os pares candidatos. Outros problemas que utilizam sub-estruturas de dados cinéticas foram apresentados. De uma forma geral, a maior parte dos problemas em Geometria Computacional é resolvida utilizando alguma sub-estrutura dentre as apresentadas nesta dissertação. Por esse motivo é importante que ao criar uma estrutura de dados cinética para problemas básicos tenha-se em mente que ela pode ser utilizada como parte de estruturas mais elaboradas que são usadas para resolver problemas mais complexos.

Com relação às linhas de pesquisa futura podemos apontar a pesquisa sobre a manutenção da mediana de n valores em movimento, o que certamente ajudaria a obtermos uma estrutura de dados cinética a partir do algoritmo que mantém o fecho convexo dinâmico [OL81]. Outra linha de pesquisa seria a aplicação de análise probabilística para estimar a localidade da estrutura de dados cinética para a triangularização de Delaunay para conjuntos aleatórios de pontos em movimento. Mais um resultado que poderia ajudar o avanço da área seria a obtenção de limitantes para o heap cinético sob movimentos não-lineares.

Na parte de implementações contribuímos com a elaboração de uma estratégia geral para a construção de implementações de estruturas de dados cinéticas a partir de blocos básicos. A idéia de usar programação orientada a objetos nesse caso parece muito natural pois as relações de herança e polimorfismo possibilitam a concepção de um código suficientemente geral para os nossos objetivos.

Os testes da Seção 7.5 simularam o comportamento da lista cinética, torneio cinético e heap cinético para entradas aleatórias sob certas condições controladas, tais como a limitação do grau máximo dos planos de vôo dos objetos ou o número de ações sobre o conjunto de pontos no tempo. Os resultados obtidos nos mostraram que o torneio cinético parece ser a melhor escolha para a utilização de uma estrutura de dados cinética básica que selecione o máximo (ou o mínimo) de um conjunto dentro de aplicações mais complexas. Os testes também mostraram que a lista cinética não é uma boa estrutura a ser usada na prática devido ao número quadrático de eventos que ela proporciona.

8.2 Outros Problemas e Resultados

Ao longo do nosso estudo sobre problemas cinéticos em Geometria Computacional nos deparamos com muitos artigos relacionados com diferentes problemas geométricos apresentando resultados e estruturas de dados cinéticas para esses problemas. Nesta seção vamos enumerar alguns desses problemas encontrados na literatura, indicar algumas referências sobre esses problemas e introduzir brevemente algumas estruturas de dados cinéticas que são apresentadas, bem como apresentar novos resultados obtidos no avanço atual da linha de pesquisa.

Detecção de Colisões

Um problema interessante para o qual estão sendo estudadas estruturas de dados cinéticas é o problema da detecção de colisões. Nesse problema temos uma coleção de objetos tipicamente representados por polígonos movendo-se continuamente no plano e desejamos saber quando há (ou haverá) uma colisão entre dois desses objetos. Aplicações em controle de tráfego aéreo, movimento de robôs, realidade virtual ou video-games parecem imediatas.

Em geral o problema de proximidade é tratado em duas fases: uma mais relaxada e outra mais atenta aos objetos envolvidos. Na fase mais relaxada os polígonos são envolvidos por uma *envoltória* simples (um retângulo ou um círculo, veja a Figura 8.1) e a possibilidade de colisão entre dois objetos é determinada pela colisão entre duas envoltórias. Isso é importante pelo fato de podermos detectar colisão entre retângulos ou círculos em tempo constante, o que não acontece com os polígonos originais. A fase mais refinada entra em ação apenas quando as envoltórias colidem. Nesse caso temos a possibilidade de colisão entre os polígonos e um acompanhamento mais especializado se faz necessário. Esse tipo de abordagem do problema de detecção de colisão é chamada

de sensível à distância.

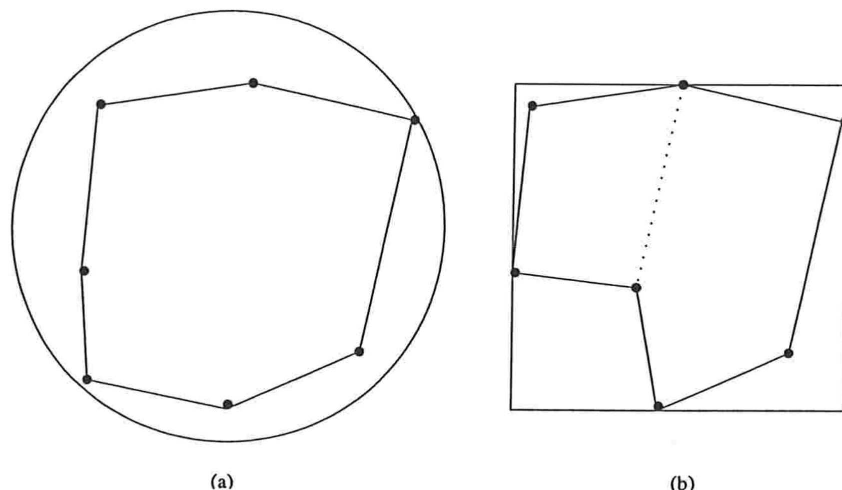


Figura 8.1: Exemplos de envoltórias para polígonos. A figura (*a*) mostra uma envoltória circular para um polígono convexo. A figura (*b*) mostra uma envoltória retangular para um polígono não-convexo que pode ser particionado em duas peças convexas através da linha pontilhada.

Em [EGSZ99, BEGHZ99], é apresentada uma estrutura de dados cinética para o problema sensível à distância descrito acima que lida com polígonos convexos. A estrutura apresentada é eficiente, local, compacta e de resposta rápida. Polígonos não-convexos podem ser tratados pela estrutura através de uma decomposição em partes convexas como mostra a Figura 8.1 (*b*).

Os certificados da estrutura são certificados de separação que existem em dois tipos: os que provam que duas envoltórias não se interceptam e os que provam que dois polígonos não se interceptam. Certificados do segundo tipo para um dado par de objetos somente existirão na fila de eventos se um certificado do primeiro tipo para as envoltórias dos dois objetos não existir. Portanto a perda de validade de um certificado entre envoltórias não significa necessariamente que os dois objetos colidiram, mas apenas que ele deve ser substituído por outros certificados mais detalhados que provarão que os dois polígonos ainda não colidiram. Uma colisão ocorrerá apenas quando algum dos certificados mais detalhados falhar. Tipicamente os certificados mais detalhados levam em consideração o par de vértices formado por um vértice de cada polígono que possui menor distância.

Podemos pensar no caso onde os polígonos são "maleáveis", isto é, aos vértices que formam o polígono é permitido que possuam movimento em relação aos outros vértices do mesmo polígono. Nesse caso a envoltória de cada polígono será constantemente alterada com o tempo. Tipicamente uma outra estrutura de dados cinética que monitora o par de pontos mais distante ou a largura de cada polígono deverá ser utilizada para tomar conta da envoltória de cada polígono.

O Problema de *Range Searching*

O problema de *Range Searching* consiste em, dado um conjunto de pontos no plano, responder a uma seqüência de consultas sobre os pontos que estão contidos em uma determinada região. Consultas típicas querem saber quantos ou quais os pontos do conjunto que estão no interior ou no exterior de um polígono no plano.

Uma versão simplificada do problema de Range Searching que possui aplicações imediatas no processamento de consultas em Bancos de Dados consiste em identificar os pontos que estão no interior de um retângulo no plano. Essa versão é chamada de *Rectangular Range Searching.* Imagine que queremos saber quais empregados de uma empresa ganham entre R$1000,00 e R$5000,00 e têm idade na faixa entre 25 e 40 anos. Em bancos de dados com esse tipo de informação temos uma busca por empregados em um retângulo no plano $salario \times idade$.

Uma consulta no problema retangular procura pelos pontos de um conjunto S no interior de um retângulo $R : [x, x'] \times [y, y']$. Um ponto $p = (x_p, y_p)$ está no interior de R se e somente se $x_p \in [x, x']$ e $y_p \in [y, y']$.

Existem diversas soluções para o problema retangular estático sobre um conjunto de n pontos no plano. Uma delas utiliza a estrutura de *kd-trees* que ocupam espaço $O(n)$ e processam uma consulta em tempo $O(\sqrt{n} + k)$, onde k é o número de pontos retornados pela consulta. Outra estrutura para o problema utiliza as *range trees* [BKOS98] que consomem tempo $O(\log n + k)$ para realizar uma operação de consulta, ocupando $O(n \log n)$ de espaço. Uma versão modificada das range trees foi descrita por Chazelle [Ch86], melhorando o limitante de espaço para $O(n \log n / \log \log n)$ e mantendo o mesmo tempo de consulta.

Se o retângulo de consulta é ilimitado, ou seja, um retângulo onde alguma das quatro coordenadas que o definem é igual a ∞ (veja a Figura 8.2 (*a*)), então é possível obter um tempo de consulta $O(\log n)$ consumindo espaço $O(n)$ utilizando *priority search trees* [Mc85, BKOS98].

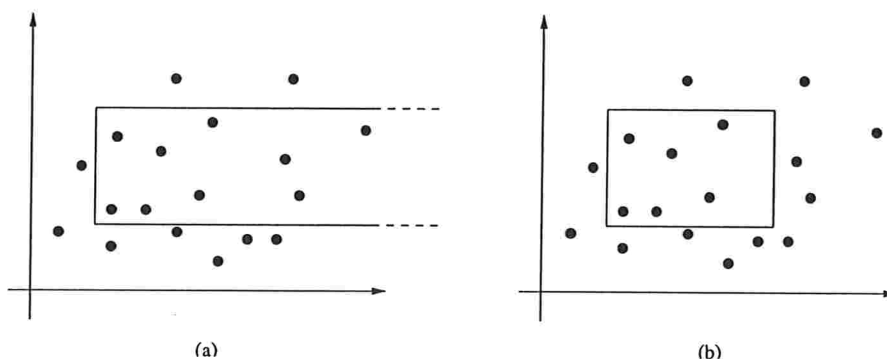


Figura 8.2: Exemplos de consultas retangulares. A figura (*a*) mostra uma consulta com retângulo ilimitado. A figura (*b*) mostra uma consulta com retângulo limitado.

No cenário dinâmico, são permitidas inclusões e remoções de pontos no conjunto. Arge, Samoladas e Vitter apresentaram em [ASV99] duas estruturas de dados que mantém o conjunto de pontos dinâmico gastando tempo $O(\log n)$ por consulta e $O(\log n)$ por operação de inserção ou remoção utilizando os resultados obtidos por Chazelle para o problema estático. A primeira estrutura é especial para consultas com os retângulos ilimitados, consumindo espaço $O(n)$. A segunda estrutura resolve o problema para retângulos limitados consumindo espaço $O(n \log n / \log \log n)$.

Em uma versão cinética do problema de Range Searching retangular os dados tipicamente possuem planos de vôo que são equações que comandam o sua trajetória no plano. Essas equações são calculadas no momento de cada consulta. Os planos de vôo podem ser alterados a qualquer momento. Em [AAE2000] é descrita uma estrutura de dados cinética para o problema que é uma cinetização da estrutura que mantém a versão dinâmica mencionada acima, obtendo uma estrutura de dados que funciona bem para cenários cinético-dinâmicos. Um evento nessa estrutura é processado em tempo amortizado $O(\log n)$ e mudanças no plano de vôo de um objeto podem ser feitas em tempo logarítmico no pior caso.

Em [BGZ97] é apresentada uma estrutura de dados cinética para o problema de Range Searching original. Trata-se de uma estrutura mais geral e que resolve eficientemente o problema cinético-dinâmico multidimensional de retornar o peso de uma região. A cada um dos n pontos em $\mathbb{R}^d$ é atribuído, de alguma forma, um peso. O peso de dada uma região é definido como a somatória dos pesos dos pontos que pertencem a essa região. A estrutura desenvolvida é capaz de responder rapidamente o peso de uma região de consulta. Assim, definindo o peso de cada ponto como o valor 1, por exemplo, é possível descobrir de forma eficiente quantos pontos pertencem a uma dada região. Essa estrutura é utilizada como ponto de partida para a resolução de vários problemas multidimensionais como o problema do par mais próximo ou o problema da árvore geradora mínima em d dimensões.

Árvore Geradora Mínima

Outro problema que nos últimos anos tem sido atacado no cenário cinético é o problema de se manter a árvore geradora mínima de um conjunto de pontos em movimento. O problema da árvore geradora mínima é originário da Teoria dos Grafos e consiste em, dado um grafo com peso nas arestas, determinar a árvore que conecta todos os vértices do grafo e que possua o menor peso. O peso de uma árvore é determinado pela soma das arestas que a compõem.

O problema da *árvore geradora mínima euclidiana* consiste em encontrarmos a árvore geradora mínima para o grafo induzido por pontos no espaço euclidiano cujas arestas são dadas por quaisquer ligações entre dois pontos do conjunto e o peso de uma aresta está associado à distância euclidiana entre os dois pontos que a determinam. Em $\mathbb{R}^2$ pode-se resolver esse problema em tempo $O(n \log n)$ através do fato de a árvore geradora mínima ser um subgrafo da triangularização de Delaunay [Mi97].

Em [BGZ97] é apresentada uma estrutura de dados cinética para o problema da árvore geradora mínima euclidiana para pontos em movimento em $\mathbb{R}^d$ que utiliza a estrutura para o problema de Range Searching multidimensional como sub-estrutura. A estrutura mantém uma árvore cujo peso é uma aproximação de $1+\epsilon$ vezes o peso da árvore geradora mínima real, para $\epsilon > 0$ no cenário cinético-dinâmico.

Em [AEGH98] é apresentado o problema da *árvore geradora mínima paramétrica*, onde os pesos de cada aresta de um grafo qualquer é uma função linear de um parâmetro λ. Nesse caso o problema cinético correspondente toma o tempo como parâmetro e os pesos das arestas tornam-se variáveis linearmente com o tempo. No mesmo artigo encontramos uma estrutura de dados cinética que dá suporte à inserções e remoções de vértices e arestas no grafo, bem como alterações na função de peso de qualquer aresta a qualquer instante.

Apêndice A

Heap

Uma *fila de prioridade* é uma estrutura de dados que armazena uma coleção de dados que possuem uma ordem total e que provê as seguintes operações:

1. inserção;
2. busca pelo maior elemento;
3. remoção do maior elemento.

Vamos supor durante todo este capítulo que queremos sempre buscar o elemento com maior prioridade dentre o conjunto de elementos na fila de prioridade. Porém, tudo funciona simetricamente para filas de prioridade que esteja m interessadas no menor elemento.

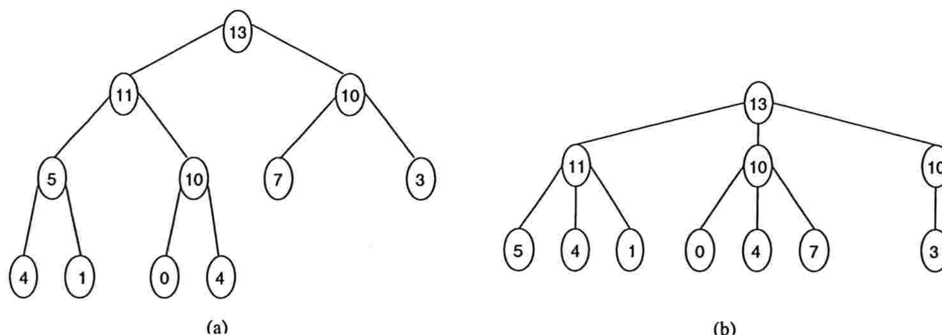


Figura A.1: Exemplos de heap: um heap binário com 11 elementos (a) e um heap ternário com os mesmos elementos (b).

O **heap** é uma estrutura de dados em que as operações de fila de prioridade são implementadas de forma eficiente através de uma estrutura em forma de árvore. Em

um heap os elementos são armazenados em uma árvore com a seguinte propriedade: o conteúdo de cada nó é maior ou igual que o de seus filhos. Além disso, a árvore do heap deve ser uma árvore quase completa, ou seja, a árvore é completamente preenchida em todos os níveis, exceto (possivelmente) no nível mais baixo que é preenchido da esquerda para a direita nó após nó. Esse fato permite que um heap seja implementado facilmente tanto em uma estrutura de apontadores como em uma estrutura seqüencial (vetor) [CLR90, Se98]. Um exemplo de heap binário e um de heap ternário podem ser vistos na Figura A.1.

Por ser uma árvore quase completa, a altura de um heap com n elementos é $\Theta(\log_b n)$, onde b é o número máximo de filhos de um nó. No caso de um heap binário, $b = 2$ e a altura do heap é $\Theta(\log_2 n)$, ou simplesmente $\Theta(\log n)$. Essa propriedade é fundamental para que as operações de uma fila de prioridade sejam implementadas eficientemente em um heap: a inserção pode ser feita em tempo $O(\log n)$, a busca pelo máximo em tempo $O(1)$ e a remoção do máximo em tempo $O(\log n)$ [CLR90, Se98].

A.1 Heapsort

Heapsort é um algoritmo de ordenação que se utiliza de um heap. A idéia é construir um heap com os n elementos a serem ordenados. A seguir, repete-se o procedimento de remover o maior elemento do heap e inseri-lo no início da lista ordenada sucessivamente até que o heap fique vazio. Note que a lista ordenada é preenchida do final para o começo. Esse fato é importante para que uma implementação do Heapsort não precise de estruturas adicionais mesmo que a entrada esteja sendo fornecida em um vetor, pois a soma dos espaços utilizados pelo heap e pela lista ordenada é sempre n. Vamos começar pelos algoritmos auxiliares:

Algoritmo A.1 `ConsertaHeap`(v)

Entrada: Um nó v do Heap que pode ter a propriedade do heap violada.
Saída: O sub-heap com raiz em v com a propriedade do heap válida em todos os nós.
  $u \leftarrow$ `SelecionaMaior`($v, filhos(v)$)
  se $u \neq v$ **então**
    $conteudo(v) \leftrightarrow conteudo(u)$
    `ConsertaHeap`(u)

`ConsertaHeap` (veja Algoritmo A.1) recebe como entrada um nó v do heap. Quando `ConsertaHeap` é chamado, assume-se que as subárvores de v são heaps, mas que o conteúdo de v pode ser menor que o de algum de seus filhos, violando a propriedade do heap. A função de `ConsertaHeap` é empurrar o conteúdo de v para algum de seus filhos, de forma que o nó v satisfaça a propriedade do heap. Assim, deve-se selecionar o nó u que possui maior conteúdo dentre v e os filhos de v (isso é feito em `SelecionaMaior`). Se $u = v$ então a propriedade do heap vale e não há nada a fazer, Caso $u \neq v$, então

devemos trocar os conteúdos de u e v e chamar ConsertaHeap recursivamente para u. A complexidade de tempo de ConsertaHeap é $O(h)$, onde h é a distância de v até uma folha. Uma chamada de ConsertaHeap para a raiz consome, portanto, tempo $O(\log n)$.

Algoritmo A.2 ConstroiHeap(S)

Entrada: Um conjunto S de elementos.
Saída: Um heap sobre o conjunto S.
  $H \leftarrow$ CriaLinks(S)
  para $i \leftarrow \lfloor |H|/b \rfloor$ **até** $i = 1$ **faça** {*b é o número de filhos de um nó do heap*}
    ConsertaHeap(H_i)
    $i \leftarrow i - 1$
  retorna H

ConstroiHeap (veja Algoritmo A.2) recebe como entrada o conjunto S de n valores que serão os elementos do heap. A chamada à CriaLinks serve para criar a estrutura de árvore completa H com todos os elementos de S e distribuir índices distintos para cada elemento, de modo que a raiz receba índice 1 e as folhas recebam os maiores índices. A numeração dos índices segue a idéia de preencher a árvore do heap nível após nível da raiz para as folhas e da esquerda para a direita. Nesse instante ainda não importa se a propriedade do heap está satisfeita, pois as chamadas à ConsertaHeap das folhas para a raiz resolverão o problema. CriaLinks executa em tempo $O(n)$ e cada chamada a ConsertaHeap gasta tempo $O(\log n)$, porém a complexidade de tempo de ConstroiHeap é $O(n)$, e não $O(n \log n)$ como parece ser [CLR90].

Podemos chegar ao limitante de $O(n)$ para a execução de ConstroiHeap observando que o tempo gasto na execução de ConsertaHeap para cada nó depende da altura desse nó na árvore. Muitos nós têm alturas baixas (estão próximos das folhas) e poucos nós têm alturas altas (estão próximos da raiz). Mais especificamente, temos um máximo de $\lceil \frac{n}{2^{h+1}} \rceil$ nós com altura h no heap. Dessa forma, o tempo consumido em ConstroiHeap pode ser expresso como

$$\sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil \cdot O(h) = O(n \cdot \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}) = O(n \cdot \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n)$$

Na expressão acima, $\sum_{h=0}^{\infty} \frac{h}{2^h}$ é a soma de uma soma de progressões geométricas de razão $\frac{1}{2}$. Essa soma tem valor limitado em 2. Assim, conclui-se que é possível construir um heap de n elementos em tempo linear.

Por fim, chegamos ao algoritmo Heapsort que recebe um conjunto S de n elementos a serem ordenados. A primeira coisa a se fazer é chamar ConstroiHeap, que devolve um heap H, em cuja raiz encontra-se o maior elemento de S. A partir daí, a cada iteração removemos o elemento máximo e o inserimos no início da lista que contém

os elementos ordenados até então. Nesse instante, o tamanho do heap diminui e o tamanho da lista ordenada aumenta.

Algoritmo A.3 `Heapsort`(S)

Entrada: Um conjunto S de elementos.
Saída: O conjunto S ordenado.

$H \leftarrow$ `ConstroiHeap`(S)
para $i \leftarrow |H|$ **até** $i = 1$ **faça**
  $S_i \leftarrow H_1$
  $H_1 \leftarrow H_i$
  `DestroiNo`(H_i)
  $i \leftarrow i - 1$
  `ConsertaHeap`(H_1)

Ao remover o elemento máximo de H que está em H_1, `Heapsort` sobe a última folha para a raiz e destrói o nó que continha tal folha. Nesse momento a propriedade do heap pode não ser mais válida e, por esse motivo é necessária a chamada de `ConsertaHeap` para afundar a nova raiz dentro do heap e restaurar a validade da propriedade do heap de que um nó tem conteúdo maior ou igual que os conteúdos dos seus filhos. No Algoritmo A.3 a variável i controla o tamanho do heap e a posição do início da lista ordenada ao mesmo tempo.

Ao final da execução de `Heapsort`, os elementos estão ordenados em S após um consumo de tempo $O(n \log n)$, pois `ConstroiHeap` consome tempo $O(n)$ e cada execução de `ConsertaHeap` gasta tempo $O(\log n)$. Como são $O(n)$ chamadas a `ConsertaHeap`, temos que `Heapsort` ordena um conjunto S de n elementos em tempo $O(n \log n)$.

A.2 Fila de Prioridade

Voltando às operações da fila de prioridade, vamos mostrar brevemente como implementá-las usando um heap. A operação de remoção (veja Algoritmo A.4) do elemento máximo do heap é feita exatamente como em uma iteração de `Heapsort`: subimos o último elemento para a raiz, reduzindo o tamanho do heap, e chamamos `ConsertaHeap` em seguida para afundar a nova raiz até que a propriedade de que a informação em cada nó é maior ou igual que a em seus filhos.

`ExtraiMaximo` assume que `TamanhoHeap` retorna o número de elementos no heap em tempo constante e que `DestroiNo` reduz o tamanho do heap também em tempo constante. Desse modo o tempo consumido por `ExtraiMaximo` é dominado pelo tempo de execução de `ConsertaHeap` que é $O(\log n)$. Portanto, `ExtraiMaximo` executa em tempo $O(\log n)$.

A operação de busca pelo maior elemento é implementada de modo trivial. Tudo o que precisamos fazer é retornar o elemento que está na raiz do heap e isso pode ser

Algoritmo A.4 `ExtraiMaximo`(H)

Entrada: Um heap H.
Saída: O elemento máximo em H.

$m \leftarrow H_1$
$s \leftarrow$ `TamanhoHeap`(H)
$H_1 \leftarrow H_s$
`DestroiNo`(H_s)
`ConsertaHeap`(H_1)
retorna m

feito em tempo constante.

A última operação necessária é a inserção. Nesse caso inserimos o novo elemento no próximo nó i disponível (após o último). É claro que o valor em i pode ser maior que o valor no nó j pai de i, o que fere a propriedade do heap. Devemos então subir no heap invertendo o novo elemento com o seu pai pelo caminho até a raiz, parando quando não for preciso realizar mais nenhuma inversão conforme mostra o Algoritmo A.5.

Algoritmo A.5 `InsereNoHeap`(H, x)

Entrada: Um heap H e um elemento x.
Saída: O heap H contendo o elemento x.

`CriaNo`(x)
$i \leftarrow$ `TamanhoHeap`(H)
$j \leftarrow$ `Pai`(H, i)
enquanto $i > 1$ e $H_j < x$ **faça**
  $H_i \leftarrow H_j$
  $i \leftarrow j$
  $j \leftarrow$ `Pai`(H, i)
$H_i \leftarrow x$

`InsereNoHeap` assume que `CriaNo` aloca espaço para o novo nó no heap e incrementa o tamanho do heap em tempo constante e que `Pai` retorna o índice do nó do heap que é pai do nó cujo índice foi recebido como parâmetro, também em tempo constante. Assim, o tempo de execução do algoritmo é dominado pelo loop que sobe pelo heap à procura do nó onde colocar o elemento a ser inserido. No pior caso, o novo elemento é o maior de todos e deve ser colocado na raiz. Nesse processo, percorremos o caminho desde a nova folha até a raiz e o tamanho desse caminho é igual à altura da raiz, que é $O(\log n)$. Dessa forma o tempo de execução de `InsereNoHeap` é $O(\log n)$.

Note que as operações de uma fila de prioridade foram implementadas com tempos de execução $O(\log n)$ no pior caso para inserções e remoções e tempo $O(1)$ para busca pelo maior elemento usando um heap. Existem outros tipos de estruturas de dados que podem ser usadas para a implementação eficiente de uma fila de prioridade que dê suporte a essas operações tais como uma árvore balanceada ou uma fila binomi-

al [Se98]. Os limitantes para os tempos de execução das operações podem mudar, mas as implementações obtidas também são consideradas eficientes. Porém a melhor implementação de uma fila de prioridade que possua apenas as operações descritas aqui é obtida usando um heap.

Referências Bibliográficas

[AGSS87] A. Aggarwal, L.J. Guibas, J. Saxe e P. Shor, *A Linear Time Algorithm for Computing the Voronoi Diagram of a Convex Polygon*, Proc. of the 19th Annual ACM Symposium on Theory of Computing, New York, 1987, pp. 39–45.

[Ag97] P.K. Agarwal, *Range Searching*, Handbook of Discrete and Computational Geometry, 1997, pp. 575–598.

[AAE2000] P.K. Agarwal, L. Arge e J. Erickson, *Indexing Moving Points*, Proc. 19th Annual ACM Symposium on Principles of Database Systems, 2000, http://compgeom.cs.uiuc.edu/~jeffe/pubs/pubs.html.

[AEGH98] P.K. Agarwal, D. Eppstein, L.J. Guibas e M.R. Henzinger, *Parametric and Kinetic Minimum Spanning Trees*, Proc. of FOCS'98, 1998.

[AGHV97] P.K. Agarwal, L.J. Guibas, J. Hershberger e E. Veach, *Maintaining the extent of a moving point set*, Proceedings 5th International Workshop on Algorithms and Data Structures (Berlin), Lecture Notes in Computer Science, vol. 1272, Springer-Verlag, 1997, http://www-graphics.stanford.edu/papers/extent/, pp. 31–44.

[ASS96] P.K. Agarwal, O. Schwarzkopf e M. Sharir, *The overlay of lower envelopes and its applications*, Discrete Computational Geometry **15** (1996) pp. 1–13.

[ASV99] L. Arge, V. Samoladas e J.S. Vitter, *On Two-Dimensional Indexability and Optimal Range Search Indexing*, Proc. 18th Annual ACM Sympos. Principles Database Systems, 1999, pp.346–357.

[At83] M.J. Atallah, *Dynamic computational geometry*, Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science (Tucson, Arizona), 1983, pp. 92–99.

[At85] M.J. Atallah, *Some dynamic computational geometry problems*, Computational Mathematics with Applications **11**, 1985, no. 12, 1171–1181.

[Ba99] J. Basch, *Kinetic Data Structures*, PhD Thesis, Stanford University, 1999.

[BEGHZ99] J. Basch, J. Erickson, L.J. Guibas, J. Hershberger e L. Zhang, *Kinetic collision detection between two simple polygons*, Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms, 1998, http://compgeom.cs.uiuc.edu/~jeffe/pubs/pubs.html.

[BGH98] J. Basch, L.J. Guibas e J. Hershberger, *Data structures for mobile data*, Journal of Algorithms, 1998, http://graphics.stanford.edu/~jbasch/publications/index.html.

[BGH97] J. Basch, L.J. Guibas e J. Hershberger, *Data structures for mobile data*, Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (New Orleans, Louisiana), ACM Press, 1997, http://graphics.stanford.edu/~jbasch/publications/index.html, pp. 747–756.

[BGZ97] J. Basch, L.J. Guibas e L. Zhang, *Proximity problems on moving points*, Proceedings of the 13th Annual Symposium on Computational Geometry (Nice, France), ACM Press, 1997, http://graphics.stanford.edu/~jbasch/publications/index.html, pp. 344–351.

[BO79] J.L. Bentley e T.A. Ottmann, *Algorithms for reporting and counting geometric intersections*, IEEE Trans. Comput. C-28, pp. 643–647, 1979.

[BKOS98] M. de Berg, M. van Kreveld, M. Overmars e O. Schwarzkopf, *Computational Geometry - Algorithms and Applications*, Springer, 1998.

[Ch86] B. Chazelle, *Filtering search: a new approach to query-answering*, SIAM J. Comput. 15, pp. 703–724, 1986.

[CKLM97] Y.J. Chiang, J.T. Klosowski, C. Lee e J.S.B. Mitchell, *Geometric algorithms for conflict detection/resolution in air traffic management*, Proceedings of the 36th IEEE Conference on Decision and Control, December 1997, http://cis.poly.edu/chiang/, pp. 1835–1840.

[CLR90] T.H. Cormen, C.E. Leiserson, R.L. Rivest, *Introduction to Algorithms*, MIT Press, McGrawHill, 1990.

[De98] H.M. Deitel e P.J. Deitel, *C++ How to Program*, Prentice Hall, 1998.

[EW85] H. Edelsbrunner e E. Welzl, *On the number of line separations of a finite set in the plane*, J. Combin. Theory Ser. 40, pp. 15–29, 1985.

[ELSS85] P. Erdös, L. Lovász, A. Simmons e E. Straus, *Dissection graphs of planar point sets*, A Survey of Combinatorial Theory, 1973, pp. 139–154.

[EGSZ99] J. Erickson, L.J. Guibas, J. Stolfi e L. Zhang, *Separation-sensitive collision detection for convex objects*, Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms, ACM Press, 1999, http://compgeom.cs.uiuc.edu/~jeffe/pubs/pubs.html.

[GO97] J.E. Goodman e J. O'Rourke (editores), *Handbook of Discrete and Computational Geometry*, CRC Press, 1997.

[Go97] M.T. Goodrich, *Parallel Algorithms in Geometry*, Handbook of Discrete and Computational Geometry, 1997, pp. 669–681.

[GMR91] L.J. Guibas, J.S.B. Mitchell e T. Roos, *Voronoi diagrams of moving points in the plane*, Proceedings of the 17th International Workshop Graph-Theoretic Concepts in Computer Science, Lecture Notes Computer Science, vol. 570, Springer-Verlag, pp. 113–125, 1991.

[GS86] L.J. Guibas e J. Stolfi, *Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams*, ACM Transactions on Graphics 4, pp. 74–123, 1986.

[Ha97] D. Halperin, *Arrangements*, Handbook of Discrete and Coputational Geometry, pp. 389–412, 1997.

[HS94] D. Halperin, M. Sharir, *New bounds for lower envelopes in three dimensions, with applications to visibility in terrains*, Discrete Computational Geometry 12, pp. 313–326, 1994.

[He89] J. Hershberger, *Finding the upper envelope of n line segments in $O(n \log n)$ time*, Information Processing Letter 33, 1989, pp. 169-174.

[Ka91] S. Kahan, *A model for data in motion*, Proceedings of the 23th Annual ACM Symposium on Theory of Computing (New Orleans, Lousiana), ACM Press, 1991, pp. 267–277.

[Ka92] S. Kahan, *Real-time closest pair of moving points*, Animation of Geometric Algorithms: A Video Review (M.H. Brown e J. Hershberger, eds.), no. 87a, Digital Equipment Corporation, Palo Alto, CA, June 1992, Companion to the video, pp. 1–3.

[Kn73] D. Knuth, *The Art of Computer Programming - Volume 3 / Sorting and Searching*, Addison-Wesley Publishing Company, 1973, pp. 451–471.

[Lee80] D.T. Lee e B.J. Schachter, *Two Algorithms for Constructing the Delaunay Triangulation*, International Journal of Computer Inf. Science, 1980.

[Mc85] E. McCreight, *Priority Search Trees*, SIAM Journal of Computing 14, pp. 257–276, 1985.

[Mi97] J.S.B. Mitchell, *Shortest Paths and Networks*, Handbook of Discrete and Computational Geometry, pp. 445–466, 1997.

[MS96] D. Musser e A. Saini, *STL Tutorial and Reference Guide*, Addison-Wesley Professional Computing Series, 1996.

[Or98] J. O'Rourke, *Computational geometry in C - Second Edition*, Cambridge University Press, Cambridge, 1998.

[OL81] M. Overmars e J. van Leeuwen, *Maintenance of configurations in the plane*, J. Comput. Syst. Sci. 23, pp. 166–204, 1981.

[PS85] F.P. Preparata e M.I. Shamos, *Computational Geometry*, Springer-Verlag, New York, 1985.

[PFTV93] W.H.Press, B.P.Flannery, S.A.Teukolsky e W.T.Vetterling, *Numerical Recipes in C - Second Edition*, Cambridge University Press, Cambridge, 1993.

[RS94] P.J. de Resende e J. Stolfi, *Fundamentos de geometria computacional*, IX Escola de Computação, 1994.

[Ro93] T. Roos, Voronoi diagrams over dynamic scenes, *Discrete Applied Mathematics* **43** (1993), 243–259.

[Se98] R. Sedgewick, *Algorithms in C++ – Parts 1-4*, Addison-Wesley, 1998.

[Sh94] M. Sharir, *Almost tight upper bounds for lower envelopes in higher dimensions*, Discrete Computational Geometry **12**, 1994, pp. 327–345.

[Sh97] M. Sharir, *Algorithm Motion Planning*, Handbook of Discrete and Computational Geometry, 1997, pp. 733–754.

[SA95] M. Sharir e P.K. Agarwal, *Davenport-Schinzel Sequences and Their Geometric Applications*, Cambridge University Press, 1995.

[Sk98] S. Skiena, *The Algorithm Design Manual*, Springer, 1998.

[Wa98] M. Waite, *Data Structures and Algorithms in Java*, Lafore, 1998.