LISTAS DE PRIORIDADES CINÉTICAS

Guilherme Dias da Fonseca

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS - GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Profª. Celina Miraglia Herrera de Figueiredo, D.Sc.

Prof. Paulo Cezar Pinto de Carvalho, Ph.D.

Prof. João Luiz Dihl Comba, Ph.D.

RIO DE JANEIRO, RJ - BRASIL

MARÇO DE 2003

FONSECA, GUILHERME DIAS DA

Listas de Prioridades Cinéticas [Rio de Janeiro] 2003

VII, 48 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2003)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1 - Estruturas de Dados

2 - Listas de Prioridades

3 - Geometria Computacional

I. COPPE/UFRJ II. Título (série)

À Lourdes, minha mãe,
que está sempre presente,
para tudo o que eu precisar,
dando-me carinho e apoio,
mas também permitindo-me
andar com minhas próprias pernas
(e cair, e levantar, e seguir)
na direção que eu escolher.
À Celina, minha orientadora,
que demonstrou muita confiança em mim,
dando-me total libertade neste trabalho,
e estando sempre à disposição
para longas conversas e revisões minuciosas.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

LISTAS DE PRIORIDADES CINÉTICAS

Guilherme Dias da Fonseca

Março/2003

Orientadora : Celina Miraglia Herrera de Figueiredo

Programa : Engenharia de Sistemas e Computação

Uma estrutura de dados cinética visa determinar, ao longo do tempo, um atributo geométrico de um conjunto de objetos em movimento contínuo. O atributo mais simples e natural de um conjunto de números reais é o elemento de valor máximo do conjunto. Uma lista de prioridades cinética é um tipo especial de estrutura de dados cinética que determina este elemento máximo ao longo do tempo, quando os elementos variam continuamente com o tempo. Além da variação dos elementos, as listas de prioridades cinéticas permitem que, a qualquer instante, elementos sejam inseridos ou removidos. Devido ao grande interesse prático e teórico nessas estruturas, várias construções foram sugeridas na literatura. Nesta tese, apresentamos as estruturas mais conhecidas, assim como uma nova estrutura descoberta pelo autor, o hanger cinético. Apresentamos também a análise de complexidade das estruturas, incluindo a análise justa de um caso do heap cinético realizada pelo autor.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

KINETIC PRIORITY QUEUES

Guilherme Dias da Fonseca

March/2003

Advisor : Celina Miraglia Herrera de Figueiredo

Department : Systems and Computers Engineering

Kinetic data structures compute a geometric atribute in a set of continuosly moving objects. The most natural atribute in a set of real numbers is the maximum element contained in the set. A kinetic priority queue is a special kinetic data structure, which determines this maximum element along time, when the elements change continuously with time. Kinetic priority queues also support insertions and deletions of elements, at any time. Due to the large practical and theoretical interest in these structures, many different constructions have been suggested in the literature. In this thesis, we present the most famous structures, and a new structure discovered by the author, the kinetic hanger. We also present the complexity analysis of the structures, including the author's tight analysis of a special case of the kinetic heap.

Conteúdo

CAPíTULO 1

Introdução

Simular o mundo físico é uma aplicação natural, e de grande importância, dos computadores. Este foi o problema que deu origem às *estruturas de dados cinéticas*. Uma estrutura de dados cinética visa determinar um atributo geométrico, ao longo do tempo, de objetos que se movem continuamente. Um exemplo é encontrar o par de objetos mais próximos em cada instante, de acordo com suas trajetórias.

Antes das estruturas de dados cinéticas, havia duas técnicas para atacar este tipo de problema. Em uma, considerava-se a trajetória dos objetos como previamente conhecida e transformava-se um problema de pontos em n dimensões em um problema de curvas em $n + 1$ dimensões, tratando o tempo como uma dimensão adicional. Esta abordagem, embora eficiente, exige que as trajetórias sejam previamente definidas. Não pode ser utilizada, por exemplo, quando as trajetórias dos objetos dependem do atributo que está sendo determinado. No nosso exemplo de encontrar o par de objetos mais próximos, quando a distância entre dois objetos for menor que um certo valor, pode haver uma colisão. Neste caso, as trajetórias dos objetos são alteradas e seria necessário reiniciar a simulação a partir deste instante de tempo.

A outra alternativa é discretizar o tempo. Resolve-se o problema estático em sucessivos instantes de tempo. Este método pode ser extremamente ineficiente por alguns motivos. Primeiro, a qualidade do resultado depende do intervalo de tempo entre as amostras. Determinar um intervalo aceitável costuma ser uma tarefa bastante complicada. Além disso, quanto menor o intervalo, maior o tempo de processamento. Em muitos casos é impossível

unir um resultado suficientemente preciso a um tempo de processamento aceitável.

Uma estrutura de dados cinética tem um conjunto mínimo de operações, que são: *inicialização*, *consulta de atributo*, *avanço no tempo* e *alteração de trajetória*. Na *inicialização* a estrutura recebe o instante de tempo a partir do qual deseja-se fazer a simulação, a posição dos objetos neste momento e suas trajetórias. O relógio interno da estrutura é ajustado para este tempo inicial. A operação de *consulta de atributo* informa o estado do atributo no tempo marcado pelo relógio interno. A operação de *avanço no tempo* avança o relógio interno da estrutura até o momento em que seu atributo seja alterado ou até um tempo prefixado. A operação de *alteração de trajetória* redefine a trajetória de um ou mais objetos, a partir do tempo marcado pelo relógio interno da estrutura. Uma observação sobre esta operação é que a trajetória dos objetos deve ser contínua. Portanto a nova trajetória deverá partir da mesma posição em que o objeto se encontrava.

Muitas vezes as estruturas de dados cinéticas são também dinâmicas, ou seja, possuem operações de *inserção* e *remoção* de elementos. Neste caso, a operação de inicialização pode criar uma estrutura vazia e a operação de inserção pode ser usada para introduzir os elementos.

Junto com a concepção das estruturas de dados cinéticas, surgiu um paradigma que se mostrou muito eficiente no projeto dessas estruturas. Este paradigma se baseia em resolver o problema estático que se deseja cinetizar e agendar, em uma lista de eventos, o instante de tempo em que cada um dos testes efetuados pelo algoritmo tem seu estado alterado. Esta lista de eventos é uma lista de prioridades. O algoritmo cinético procede examinando sempre o primeiro evento da lista e alterando toda a estrutura para o estado que seria obtido se, no algoritmo para o problema estático, o teste associado ao evento tivesse retornado um resultado diferente. Normalmente vários eventos têm que ser reagendados neste processo.

O atributo geométrico mais simples é, dada uma coleção de números reais (pontos na reta), determinar o maior número (ponto extremo em uma direção). Este é o atributo enfocado nesta tese e chamamos as estruturas de dados cinéticas que determinam este atributo de listas de prioridades cinéticas ou LPC. Embora o caso estático seja trivial, veremos que tanto as LPC quanto suas análises, mesmo quando os objetos estão em movimento uniforme, não são nada triviais. Além disso, a questão de desenvolver estruturas rápidas para este problema parece longe de fechada. As LPC assintoticamente mais eficientes só funcionam com funções lineares do tempo e usam estruturas de fecho convexo dinâmico extremamente complexas e pouco práticas. Em contrapartida, existem LPC um pouco menos eficientes que são extremamente simples e funcionam tanto com funções lineares quanto com curvas.

Se a motivação inicial para as estruturas de dados cinéticas foi simular o mundo físico, atualmente esta é apenas uma de suas aplicações. Como veremos, vários problemas de natureza estática podem ser resolvidos eficientemente com estruturas de dados cinéticas, especialmente com LPC. Um exemplo simples é o envelope superior de um conjunto de curvas. Outro bem mais refinado é o k-ésimo nível de um arranjo de curvas. Para este último problema, tanto o algoritmo determinístico assintoticamente mais eficiente quanto o algoritmo não trivial mais simples baseiam-se em LPC. Consequentemente, LPC mais eficientes implicam em algoritmos mais eficientes para estes problemas.

No capítulo 2, apresentamos os critérios que utilizaremos nos demais capítulos para analisarmos as LPC. No capítulo 3, explicamos detalhadamente o heap cinético [**3**], apresentamos uma análise justa proposta pelo autor [**9**] para o caso sem inserções ou remoções e uma análise de [**3**] que não foi provada justa para o caso com inserções e remoções. Apresentamos também uma variação $(\lg n)$-ária do heap cinético proposta pelo autor [**9**], que é mais eficiente que a versão binária. No capítulo 4, apresentamos o

torneio cinético e sua análise, juntando os resultados de [**4**] com a análise versão $(\lg n)$-ária proposta em [**6**]. No capítulo 5, apresentamos e analisamos o heater cinético [**2**], uma LPC randomizada, que tem localidade ótima e é tão eficiente quanto o torneio cinético. No capítulo 6, apresentamos e analisamos o hanger cinético, outra LPC randomizada, esta proposta pelo autor em [**10**]. As características assintóticas do hanger cinético são as mesmas do heater cinético, porém a estrutura é mais simples de ser implementada e acredita-se que seja mais eficiente na prática. No capítulo 7, mostramos como reduzir o problema de listas de prioridades cinéticas ao problema de fecho convexo dinâmico. Esta redução, proposta por Chan em [**6**], funciona apenas com funções lineares do tempo e não é muito eficiente para aplicações práticas, mas fornece as melhores complexidades de tempo conhecidas, sendo de grande valor teórico. No capítulo 8 exemplificamos uma aplicação das listas de prioridades cinéticas para resolver um problema de natureza estática, comparamos as várias estruturas apresentadas e mencionamos alguns problemas em aberto.

CAPíTULO 2

Analisando uma Lista de Prioridades Cinética

Para analisarmos a complexidade de tempo das listas de prioridades cinéticas (LPC), assim como de outras estruturas de dados cinéticas, precisamos definir precisamente que operações serão aplicadas e as propriedades que as trajetórias satisfazem.

As trajetórias dos objetos ao longo do tempo t são funções reais contínuas de t, que visualisamos no plano $t \times y$. Chamamos estas funções de curvas. Quando as funções são lineares, chamamos de retas. Uma propriedade combinatória importante destas curvas é o número de vezes que duas curvas se interceptam, que chamamos do grau da curva e denotamos por δ. Quando as curvas se interceptam no máximo 1 vez, chamamos de pseudo-retas. Podemos também falar em parábolas, pseudo-parábolas (curvas de grau 2), cúbicas etc, com definições análogas.

Quando estamos analisando estruturas dinâmicas, objetos podem ser criados ou destruídos ao longo do tempo. Neste caso, as trajetórias são funções definidas apenas em um intervalo de tempo. Chamamos estas funções de segmentos de curvas, segmentos de reta etc. Um detalhe que merece atenção especial é que segmentos de curvas onde cada par de segmentos se intercepta no máximo δ vezes *não* são necessariamente segmentos de curvas onde cada par de curvas se interceptam no máximo δ vezes. Veja, por exemplo, que os segmentos da figura 2.1 não são segmentos de nenhum conjunto de curvas de grau 1. Porém, para as nossas análises, o conjunto de curvas ao qual os segmentos pertence não é relevante. Assim, definimos como segmento de curvas de grau δ um segmento de um conjunto tal que cada par de segmentos se intercepte no máximo δ vezes.

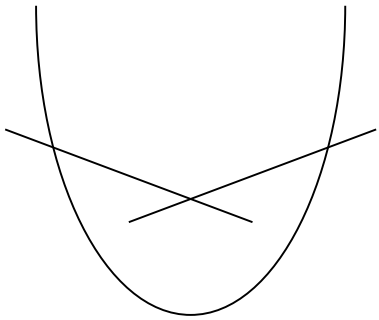

FIGURA 2.1. Segmentos de curvas que, embora os pares de segmentos só se interceptem uma vez, não são segmentos de curvas que só se interceptam uma vez. Pela nossa definição são segmentos de curvas de grau 1

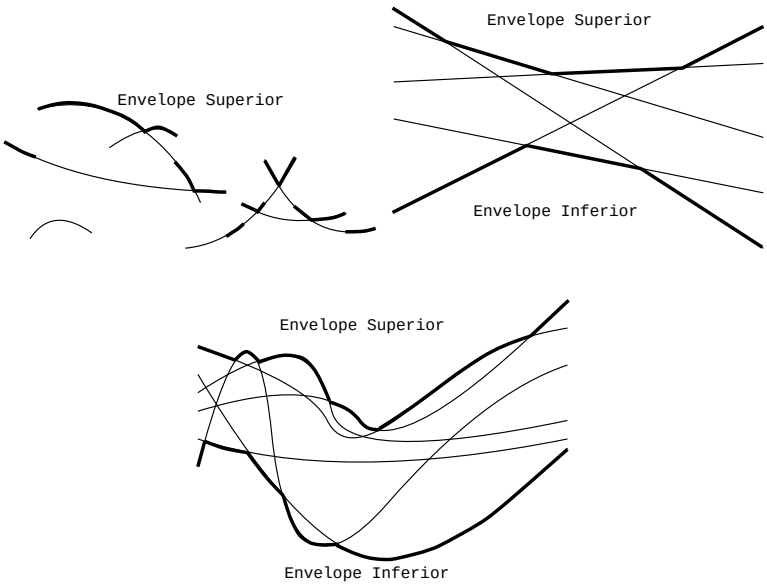

FIGURA 2.2. Envelopes superiores de conjuntos de curvas com $(\delta = 1, n = 3, m = 7)$, $(\delta = 1, n = 5)$ e $(\delta = 3, n = 6)$, respectivamente.

O envelope superior de um conjunto de curvas ou segmentos de curvas é a seqüência de segmentos que, ao longo do tempo, são máximos em y, como ilustra a figura 2.2.

A análise que tem se mostrado mais útil para as LPC é examinarmos a complexidade de tempo total da LPC determinar, usando o paradigma de linha de varredura, o envelope superior de um conjunto de curvas e segmentos de curvas. Esta análise será em função de, basicamente, três parâmetros, δ,

n e m. O parâmetro δ representa o número de vezes que cada par de curvas se intercepta. O parâmetro n representa o número máximo de curvas que podem ser interceptadas por uma reta vertical, ou seja, o número máximo de elementos armazenados simultaneamente na estrutura. O parâmetro m serve para limitar superiormente o número de inserções e remoções. Tanto o número de inserções quanto o de remoções devem ser menores ou iguais a m. No caso de não haver inserções ou remoções, o parâmetro m é omitido. Chamamos de complexidade de tempo de uma estrutura em um cenário (δ, n, m) (m podendo ser omitido) a complexidade de tempo da estrutura computar o envelope superior no cenário com os parâmetros δ, n e m.

Precisamos ressaltar que não estamos resolvendo um problema estático, ou seja, os diversos segmentos do cenário não precisam ser previamente conhecidos. Por isso não podemos usar os algoritmos estáticos que calculam este envelope superior.

Uma função que aparecerá várias vezes na análise de complexidade das listas de prioridade cinética é a função $\lambda_\delta(n)$, que é definida como o número máximo de segmentos no envelope superior de n curvas de grau δ. São conhecidos limites assintóticos bastante precisos para $\lambda_\delta(n)$, considerando sempre δ como constante. Estes limites envolvem a função de Ackerman. Primeiro definimos a seguinte família de funções recursivas:

$$A_1(n) = 2n;$$

$$A_k(n) = A_{k-1}^{(n)}(1), \text{ para } k \geq 2.$$

Nesta definição denotamos por $f^{(n)}$ a função composta por n iterações de f, por exemplo $f^{(3)}(n) = f(f(f(n)))$. Definimos então a função de Ackerman $A(n)$ como:

$$A(n) = A_n(n).$$

A função inversa de $A(n)$, denotada por $\alpha(n)$ aparece nos limites de $\lambda_\delta(n)$. Esta função cresce muito lentamente, sendo menor que 4 para todos

os propósitos práticos. Os limites para $\lambda_\delta(n)$ são:

$$\lambda_1(n) = n;$$

$$\lambda_2(n) = 2n - 1;$$

$$\lambda_3(n) = \Theta(n\alpha(n));$$

$$\lambda_s(n) \leq n2^{(1+o(1))\alpha(n)^{\frac{s-2}{2}}}, \text{ para n par;}$$

$$\lambda_s(n) \leq n2^{(1+o(1))\alpha(n)^{\frac{s-2}{2}}\log\alpha(n)}, \text{ para n ímpar.}$$

Para maiores detalhes sobre a função $\lambda_\delta(n)$, recomendamos a leitura de [**13**]. Um resultado que nos será bastante útil é que o número máximo de segmentos no envelope superior de n *segmentos de curvas* de grau δ é limitado por $\lambda_{\delta+2}(n)$.

A maioria das LPC terá sua complexidade descrita em função destes três parâmetros, δ, n e m. Porém, as LPC baseadas em estruturas de fecho convexo dinâmico funcionam apenas com segmentos de retas. Uma outra LPC que estudaremos, o heap cinético, não está limitado a retas ou pseudo-retas, porém analisar sua complexidade no caso $\delta > 1$ ainda é um problema de pesquisa, para o qual os resultados conhecidos não são satisfatórios.

Como mencionamos anteriormente, um paradigma usual para o projeto de estruturas de dados cinéticas usa uma lista de eventos. Esta lista de eventos armazena uma espécie de prazo de validade de cada teste do algoritmo estático. Se, a cada evento processado, precisamos alterar $O(f(n))$ eventos, dizemos que a estrutura tem localidade $O(f(n))$. A localidade da estrutura serve para avaliar o impacto causado pela falha de um teste, ou seja, pelo processamento de um evento.

Ao analisarmos estruturas desenvolvidas neste paradigma às vezes é útil separar a análise em duas partes. Em uma delas calculamos o número máximo de eventos processados e na outra o tempo gasto para processar um evento.

CAPíTULO 3

Heap Cinético

A estrutura de dados mais usada para se manter o máximo de um conjunto ordenado, submetido a operações de inserção e remoção, é o heap binário, ou simplesmente heap. Como veremos, é extremamente simples construirmos sua versão cinética, o chamado heap cinético. Porém, em contraste com a simplicidade algorítmica do heap cinético, sua análise se mostra bastante complicada e a maior parte dos resultados teóricos ainda são insatisfatórios. Algumas referências para o heap cinético são [**3, 4, 9**].

3.1. Versão Estática

Um heap binário é uma árvore binária cheia onde cada vértice está associado a um elemento de um conjunto ordenado, através de uma bijeção. Para tornarmos as explicações mais claras, muitas vezes não faremos distinção entre os vértices e os elementos associados a eles, como ilustrado a seguir. Em um heap binário, todo vértice é maior que seus descendentes (o elemento associado a todo vértice é maior que os elementos associados aos seus descendentes). Chamamos esta propriedade de ordenação de heap. Um heap binário está representado na figura 3.1.

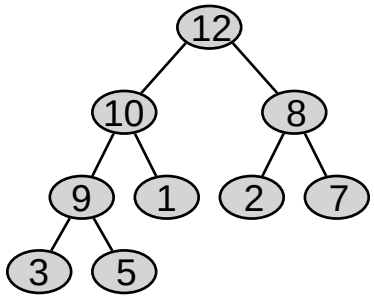

FIGURA 3.1. Heap binário com 9 elementos.

Este heap (estático) suporta as operações de inserção, remoção e determinação do elemento máximo. Determinar o máximo é bastante simples, pois o elemento máximo está sempre na raiz da árvore. Outra operação, chamada de extração do máximo, consiste em determinar o elemento máximo e em seguida removê-lo.

Para se inserir um novo elemento cria-se uma nova folha tomando cuidado para manter a árvore cheia. A esta folha associa-se o novo elemento. Isto pode violar a ordenação de heap. Para restabelecê-la, basta subir na árvore a partir da nova folha trocando, enquanto a ordenação for violada, os elementos associados ao pai e ao filho na árvore.

Para remover um elemento, a idéia mais intuitiva seria começar no elemento que se deseja remover e descer ao longo da árvore, subindo sempre um nível com o maior dos filhos. Esta estratégia preserva a ordenação de heap, mas não mantém a árvore cheia. Como o fato da árvore ser cheia é essencial para a análise de complexidade, adota-se outra estratégia. Remove-se da árvore uma folha que mantenha a árvore cheia. Possivelmente este não é o elemento que desejamos remover. Associa-se então o vértice do elemento que desejamos remover ao elemento da folha removida. A árvore agora é cheia e tem os elementos desejados, porém não respeita a ordenação de heap. Para restaurar a ordenação, deve-se descer na árvore, a partir do vértice alterado, trocando cada vértice com seu maior filho, parando quando a ordenação for restaurada.

Para analisarmos a eficiência de um heap binário com n elementos, basta notarmos que a operação de determinação do máximo pode ser feita em tempo $O(1)$, enquanto que inserções e remoções podem ser feitas em tempo proporcional à altura da árvore, que é $O(\lg n)$, pois a árvore é cheia.

3.2. Versão Cinética

No heap cinético, os elementos associados aos vértices são funções do tempo. Inicia-se a árvore como um heap estático, usando os valores das

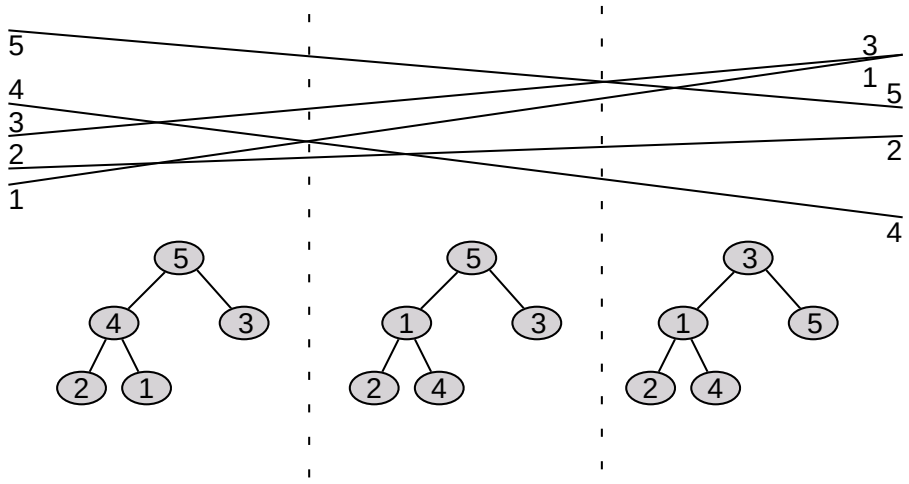

FIGURA 3.2. Heap cinético ao longo do tempo.

funções computados no tempo inicial. Para cada nó interno, agendamos um evento potencial no instante em que o primeiro filho vai se tornar maior que o nó interno. Os eventos potenciais são armazenados na lista de eventos, que é uma outra lista de prioridades (possivelmente um heap), permitindo a determinação do evento que ocorre primeiro no tempo.

Para avançar no tempo, é preciso fazer o que chamamos de processar um evento. Extrai-se o primeiro evento da lista de eventos, ajustando o relógio da estrutura para o tempo deste evento. Em seguida, troca-se a posição dos dois vértices do evento. Este comportamento está ilustrado na figura 3.2. Na troca, diz-se que o elemento que era filho antes do evento sobe e o elemento que era pai antes do evento desce. Com a troca, a ordenação de heap continuará válida. A lista de eventos também precisa ser atualizada. No máximo três eventos precisam ser reagendados, como mostra a figura 3.3. O tempo de nenhum outro evento é alterado. A localidade da estrutura é, portanto, $O(1)$. Pode-se repetir este procedimento até que o elemento da raiz seja alterado ou até um tempo pré-estabelecido, se for conveniente.

As operações de inserção e remoção no heap cinético são realizadas de modo semelhante ao caso estático. Vários eventos, porém, precisam ser reagendados. O número de eventos reagendados em uma inserção ou remoção

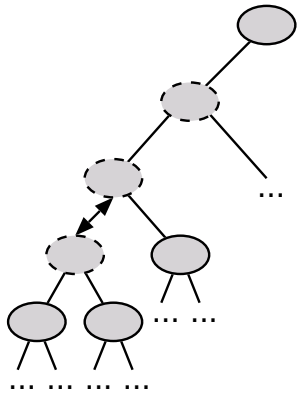

FIGURA 3.3. Processando um evento em um heap cinético. Apenas os eventos correspondentes aos vértices pontilhados precisam ser reagendados.

é, no máximo, proporcional à altura da árvore. Em um heap cinético com n elementos o número de eventos simultaneamente na lista de eventos é $O(n)$. Cada evento pode ser reagendado em tempo $O(\lg n)$. Multiplicando as complexidades, uma operação de inserção ou remoção em um heap binário com n elementos leva tempo $O(\lg^2 n)$.

Até aqui todas as análises foram bastante simples. Falta porém respondermos uma pergunta crucial: Quantos eventos são processados no máximo por um heap cinético? Esta pergunta mostrou-se surpreendentemente não trivial.

O caso mais simples é quando todos os n elementos são retas (ou pseudo-retas) e não há qualquer inserção ou remoção. Neste caso, o autor provou em [**9**] um limite justo de $\Theta(n \lg n)$ para o número de eventos de um heap cinético binário. Antes sabia-se que o número máximo de eventos de um heap cinético binário era $\Omega(n \lg n)$ e $O(n \lg^2 n)$.

Antes de provarmos este resultado, vamos generalizar o conceito de heap cinético, para o que definimos como uma árvore com ordenação de heap. Uma árvore com ordenação de heap é uma árvore enraizada qualquer com elementos associados aos seus vértices por uma bijeção. Como no heap binário, todo elemento deve ser maior que seus descendentes. Os algoritmos apresentados para o heap cinético também funcionam para produzirmos uma

versão cinética das árvores com ordenação de heap, embora as complexidades de tempo possam ser diferentes.

Ao tratarmos de uma árvore cinética com ordenação de heap K, os elementos mudam de posição ao longo do tempo. Portanto, quando a posição dos elementos é relevante, usamos K_t no lugar de K, onde K_t significa a árvore K no tempo t. Caso ocorra um evento no tempo t, a posição dos elementos na árvore não está definida neste instante. Neste caso denotamos por K_{t-} e K_{t+} as árvores imediatamente antes e imediatamente depois do tempo t. Denotamos por $K_{-\infty}$ e $K_{+\infty}$ as árvores antes do processamento de qualquer evento e depois do processamento de todos os eventos.

Definimos $l(K_t, v)$, o nível do vértice v na árvore K_t, como a distância (número de arestas) de v até a raiz da árvore K_t. Usamos a mesma notação $l(K_t, f)$ para o nível de um elemento f, que é definido como o nível do vértice correspondente a f.

LEMA 3.1. *Em uma árvore cinética com ordenação de heap K armazenando retas, para todo elemento $f \in K$, há no máximo $l(K_{-\infty}, f)$ eventos em que f sobe.*

DEMONSTRAÇÃO. Definimos $\phi(K_t, f)$ como o número de ancestrais de f em K_t com coeficiente angular menor que o coeficiente angular de f, ou seja, o número de ancestrais de f que assumirão valores menores que o de f em algum tempo maior que t. Claramente, para todo elemento f, temos $\phi(K_{-\infty}, f) = l(K_{-\infty}, f)$ e $\phi(K_{+\infty}, f) = 0$.

Digamos que, no evento ocorrido em um tempo t qualquer, o elemento p desce e o elemento c sobe. Para provarmos o lema, basta provarmos que $\phi(K_{t+}, c) < \phi(K_{t-}, c)$ e que, para todo elemento f, temos $\phi(K_{t+}, f) \leq \phi(K_{t-}, f)$.

Seja S o conjunto dos elementos de K. Vamos particionar $S \setminus \{p, c\}$ em três conjuntos, conforme a figura 3.4:

$S_1 = \{f \in S \setminus \{p, c\} : f$ *não* é descendente de p em $K_{t-}\}$,

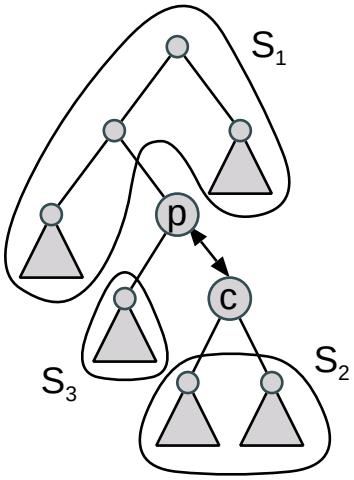

FIGURA 3.4. Particionamento da árvore usado na prova do lema 3.1.

$S_2 = \{f \in S \setminus \{p, c\} : f$ é descendente de c em $K_{t-}\}$,

$S_3 = \{f \in S \setminus \{p, c\} : f$ é descendente de p mas *não* de c em $K_{t-}\}$.

Claramente, se $f \in S_1 \cup S_2$, então $\phi(K_{t+}, f) = \phi(K_{t-}, f)$, pois os ancestrais de f em K_{t-} e em K_{t+} são os mesmos.

Para $f \in S_3$ notamos que se p desce e c sobe no evento, então o coeficiente angular de c é maior que o coeficiente angular de p. Segue então que $\phi(K_{t+}, f) \leq \phi(K_{t-}, f)$.

Temos também $\phi(K_{t+}, p) = \phi(K_{t-}, p)$, pois o único elemento acrescido ao conjunto de ancestrais de p foi c, o qual tem coeficiente angular maior que o de p.

Finalmente $\phi(K_{t+}, p) = \phi(K_{t-}, p) - 1$, pois p foi removido do conjunto de ancestrais de c e o coeficiente angular de p é menor que o coeficiente angular de c. □

Provamos, também, que os limites superiores obtidos acima são justos, usando um argumento análogo ao de [**3**]:

LEMA 3.2. *Existe árvore cinética com ordenação de heap K armazenando retas onde, para todo elemento $f \in K$, são processados pelo menos $l(K_{-\infty}, f)$ eventos em que f sobe.*

DEMONSTRAÇÃO. Vamos considerar que as funções lineares armazenadas no heap são retas tangentes à parábola $y = t^2$. Toda função é máxima no ponto de tangência, estando então na raiz da árvore neste instante. Como em cada evento apenas um elemento sobe na árvore, para que o elemento f chegue à raiz, partindo de sua posição inicial (quando $t = -\infty$), são necessários $l(K_{-\infty}, f)$ eventos em que f sobe. □

Destes lemas segue diretamente:

TEOREMA 3.3. *Um heap cinético binário armazenando n retas processa no máximo* $\Theta(n \lg n)$ *eventos.*

Embora tenhamos usado a terminologia de retas, todos os resultados acima também são válidos para heaps armazenando um conjunto de pseudo-retas. As demonstrações são análogas, tendo apenas o cuidado de, no lugar do coeficiente angular, usar a ordem em que as funções se interceptam.

Quando os n elementos são segmentos de retas (ou seja, há n inserções e n remoções), o único limite superior conhecido para o número de eventos é $O(n\sqrt{n \lg n})$ [**3**]. O único limite inferior conhecido é $\Omega(n \lg n)$ [**3**]. Antes de provar este limite superior, provaremos outro resultado de geometria combinatória, apenas para nos familiarizarmos com o tipo de demonstração.

Dado S, um conjunto de retas no plano, chamamos de k-ésimo nível de S a linha poligonal infinita formada por segmentos de S tal que cada segmento tem $k - 1$ retas acima dele (figura 3.5). Determinar o número máximo de segmentos do k-ésimo nível é um problema reconhecidamente difícil (referências para este problema podem ser encontradas na introdução de [**6**]). Provaremos um limite superior.

TEOREMA 3.4. *O k-ésimo nível de um conjunto S de n retas tem* $O(n\sqrt{k})$ *segmentos.*

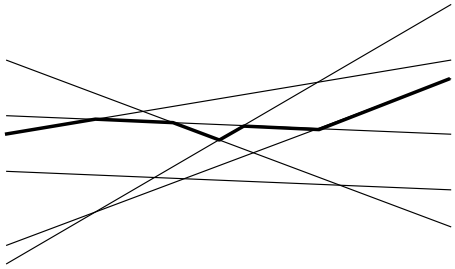

FIGURA 3.5. Um conjunto de retas com o terceiro nível destacado.

DEMONSTRAÇÃO. Consideraremos, como usual, as retas como função do tempo. Como usaremos uma notação muito semelhante a usada até agora, não definiremos explicitamente algumas notações.

Primeiro, notamos que existem dois tipos claramente distintos de trocas que podem definir o início de um novo segmento no k-ésimo nível: nas trocas do tipo (i) a reta que estava no nível $k-1$ troca de lugar com a reta no nível k, enquanto nas trocas do tipo (ii) a reta que estava no nível $k+1$ troca de lugar com a reta no nível k. Podemos considerar apenas as trocas do tipo (ii), pois as trocas do tipo (i) para o k-ésimo nível são trocas do tipo (ii) para o nível $k-1$. Portanto provar que o número de trocas do tipo (ii) é $O(n\sqrt{k})$, implica na prova do teorema.

Seja $p(r)$ a posição da reta r na lista das retas de S em ordem *decrescente* de coeficiente angular. Em um instante de tempo t, definimos R_t como o conjunto de retas acima ou sobre o k-ésimo nível no instante t e definimos:

$$P_t = \sum_{r \in R_t} p(r).$$

Claramente $P_{-\infty} = O(nk)$. Se no instante de tempo t, há uma troca da reta no k-ésimo nível com a reta no nível $k+1$, então $P_{t+} < P_{t-}$.

Usaremos um artifício para calcular um limite superior para o número de trocas do tipo (ii). Vamos particionar estas trocas em dois conjuntos, usando uma constante B. No primeiro conjunto colocamos as trocas que reduziram o valor de P_t em mais de B unidades e no segundo as trocas que reduziram o valor de P_t em menos de B ou exatamente B unidades.

O número de trocas do primeiro conjunto é claramente $O(nk/B)$. Já o número de trocas do segundo conjunto pode ser determinado usando o fato de que, para cada reta r, existem apenas $O(B)$ retas r' com coeficiente angular maior que r tais que $p(r) - p(r') \leq B$. Portanto, o número de trocas do segundo conjunto é $O(nB)$.

Fazendo $B = O(\sqrt{k})$ temos que o número de trocas do primeiro e do segundo conjunto são ambos $O(n\sqrt{k})$, concluindo a prova. □

Usando uma argumentação semelhante, em [**3**] prova-se:

LEMA 3.5. *Um heap cinético inicialmente contendo no máximo n elementos, submetido a no máximo n operações de inserção e remoção, onde todos os elementos são funções lineares do tempo, processa $O(n\sqrt{n \lg n})$ eventos.*

DEMONSTRAÇÃO. Definimos $r(f)$ como a posição de f na lista, ordenada crescentemente por coeficiente angular, de todos os elementos que serão inseridos. Definimos $\phi(K_t, f) = l(K_t, f)r(f)$ (onde $l(K_t, f)$ é a distância, no heap K_t de f até a raiz) e o potencial $\phi(K_t) = \sum_f \phi(K_t, f)$. Antes de inserirmos qualquer elemento, como o heap já inicia com $O(n)$ elementos, o potencial $\phi(K_t)$ é $O(n^2 \lg n)$. Vamos analisar a variação desta função potencial ao processarmos um evento, inserirmos um elemento e removermos um elemento.

Ao processarmos um evento, no tempo t, onde p desce e c sobe, como $r(c) > r(p)$ e $\phi(K_{t+}) = \phi(K_{t-}) - r(c) + r(p)$, temos $\phi(K_{t+}) < \phi(K_{t-})$.

Para inserirmos um elemento f, começamos colocando o elemento em uma nova folha, aumentando o potencial do heap em $l(K_t, f)r(f) = O(n \lg n)$. Então vamos trocando o novo elemento com seu pai até que a ordenação de heap seja restaurada. Fazemos $O(\lg n)$ trocas deste tipo, e cada troca aumenta o potencial em $O(n)$. Portanto, uma inserção aumenta o potencial em $O(n \lg n)$ e n inserções podem aumentar o potencial em $O(n^2 \lg n)$.

Para removermos um elemento do heap, colocamos uma folha no lugar do elemento que desejamos remover e descemos com este elemento até que a ordenação do heap seja restaurada. O potencial do heap, na remoção, certamente diminui, pois nenhum elemento terá seu nível aumentado (movido para baixo) neste processo.

Com isto, provamos que o heap cinético com n inserções e remoções processa no máximo $O(n^2 \lg n)$ eventos, mas já sabíamos, trivialmente, que o número de eventos é $O(n^2)$. Mas podemos usar um argumento análogo ao do teorema 3.4.

Vamos particionar os eventos em dois conjuntos, usando uma constante B. No primeiro conjunto colocamos os eventos que reduziram o potencial do heap em mais que B e no segundo conjunto os eventos que reduziram o potencial do heap em no máximo B.

Como o potencial inicial e os acréscimos no potencial somam no máximo $O(n^2 \lg n)$, o número de eventos no primeiro conjunto é $O(n^2 \lg n/B)$. Já o número de elementos no segundo conjunto é $O(nB)$. Fazendo $B = \sqrt{n \lg n}$, temos que o número total de eventos é $O(n\sqrt{n \lg n})$. □

Deste lema segue:

Teorema 3.6. *Um heap cinético sempre contendo no máximo n elementos, submetido a m operações de inserção e remoção, onde todos os elementos são funções lineares do tempo, processa $O(m\sqrt{n \lg n})$ eventos.*

Demonstração. Podemos dividir o tempo em m/n fatias, a cada n inserções. Como cada fatia inicia com no máximo n elementos e nela há apenas n inserções, então podemos aplicar o lema 3.5 em cada fatia. Assim, em cada fatia são processados $O(n\sqrt{n \lg n})$ eventos e, nas m/n fatias, são processados $O(m\sqrt{n \lg n})$ eventos. □

Quando os n elementos são funções que podem se interceptar mais de uma vez, nenhum resultado não trivial é conhecido.

Uma variação do heap cinético é, no lugar de usarmos uma árvore binária, usarmos uma árvore k-ária. Por isso enunciamos os lemas 3.2 e 3.1 usando árvores com ordenação de heap, e não heaps binários. O número máximo de eventos processados em uma árvore k-ária é $O(n \lg n / \lg k)$. O tempo para processar um envento, entretanto é $O(k + \lg(n/k))$. O tempo total é o produto dessas quantidades e é minimizado fazendo $k = O(\lg n)$. Neste caso a complexidade de tempo do heap cinético varrendo n retas é $O(n \lg^2 n / \lg \lg n)$.

CAPíTULO 4

Torneio Cinético

O torneio cinético é uma lista de prioridades cinética conceitualmente simples e eficiente. Sua maior desvantagem é não oferecer localidade $O(1)$, mas sim localidade $O(\lg n)$. Algumas referências são [**2, 4, 6**].

4.1. Versão Estática

Um algoritmo estático clássico para determinar o máximo de um conjunto de n elementos é baseado em divisão e conquista e está escrito na figura 4.1. Este algoritmo é ótimo, realizando exatamente $n-1$ comparações. Em uma máquina paralela com n (ou $n/\lg n$) processadores, o algoritmo tem complexidade de tempo $O(\lg n)$.

```
Máximo(S)
{
    Se |S| ≤ 2
        Retorne max(S[1], S[2])
    Particione S em dois conjuntos S1 e S2 com |S1| − |S2| ≤ 1
    Retorne max(Máximo(S1),Máximo(S2))
}
```

FIGURA 4.1. Algoritmo para determinar o elemento máximo de um conjunto usando divisão e conquista.

As comparações feitas pelo algoritmo formam o que é chamado de árvore de torneio. Uma árvore de torneio de um conjunto S é uma árvore estritamente binária com $|S|$ folhas, cada folha armazenando um elemento distinto de S. Cada nó interno armazena o maior elemento armazenado em um de seus filhos. Consequentemente, o elemento armazenado em um nó interno v é o elemento máximo da sub-árvore com raiz v. Uma árvore de torneio está representada na figura 4.2.

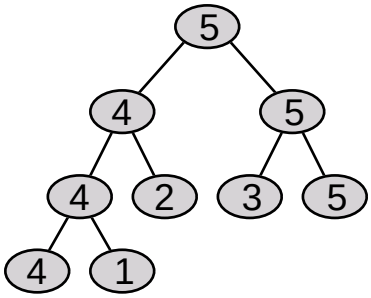

FIGURA 4.2. Árvore de torneio com 5 elementos.

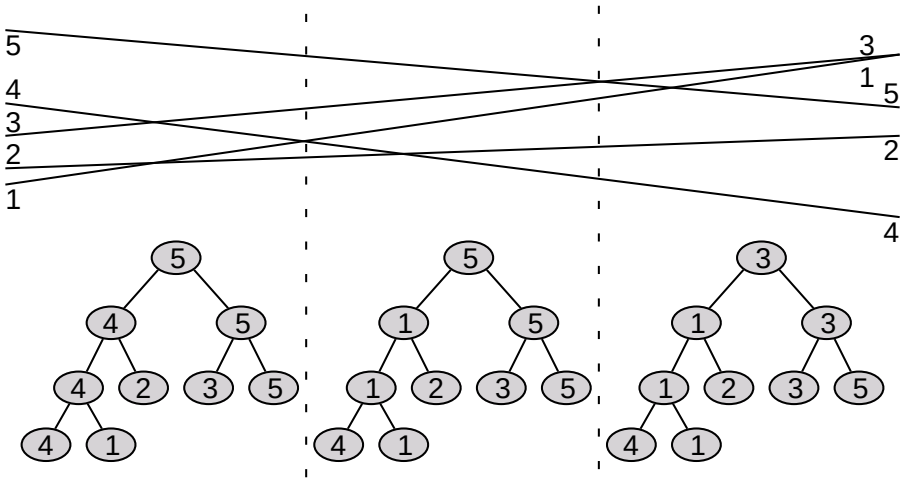

FIGURA 4.3. Torneio cinético ao longo do tempo.

4.2. Versão Cinética

É fácil transformar esta árvore de torneio no que chamamos de torneio cinético. No torneio cinético os elementos da árvore não são mais números reais, mas sim funções contínuas do tempo. Em todo instante t, a nossa fila de eventos contém um evento potencial para cada nó interno v tal que um filho f de v irá se tornar maior que v no instante $t' > t$. A chave do evento é o tempo t'. A figura 4.3 mostra as atualizações da árvore de torneio ao longo do tempo quando os elementos são retas.

O torneio cinético não tem localidade $O(1)$, mas sim localidade $O(\lg n)$, pois o processamento de um evento pode alterar todos os eventos do caminho até a raiz da árvore. A complexidade de tempo para processar um evento

depende do número de vértices do torneio que são alterados no processamento do evento. Para cada vértice alterado na árvore, um evento potencial deve ser reagendado na lista de eventos.

TEOREMA 4.1. *Em um torneio cinético com n retas, há no máximo $O(n \lg n)$ alterações na árvore. Consequentemente, a complexidade de tempo do torneio cinético em um cenário $(\delta = 1, n)$ é $O(n \lg^2 n)$.*

DEMONSTRAÇÃO. Há no máximo n alterações do elemento presente na raiz da árvore, pois cada reta só pode aparecer uma vez no envelope superior de um conjunto de retas. Para todo vértice v o número máximo de alterações do elemento contido em v é igual ao número de folhas da sub-árvore enraizada em v, pelo mesmo argumento. Somando as alterações de todos os vértices chegamos ao valor de $O(n \lg n)$. Para calcular a complexidade de tempo basta multiplicar pelo tempo de $O(\lg n)$ por alteração na árvore. □

Esta prova nos indica como fazer a análise para o caso de n curvas de grau δ. A idéia da prova está exemplificada na figura 4.4.

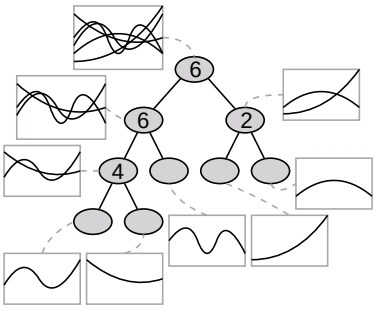

FIGURA 4.4. Envelopes superiores das sub-árvores enraizadas em cada nó da árvore de torneio, com o número de segmentos dos envelopes superiores destacados.

TEOREMA 4.2. *Em um torneio cinético com n curvas de grau δ, há no máximo $O(\lambda_\delta(n) \lg n)$ alterações na árvore. Consequentemente, a complexidade de tempo do torneio cinético em um cenário (δ, n) é $O(\lambda_\delta(n) \lg^2 n)$.*

DEMONSTRAÇÃO. O envelope superior de n funções com cada par de funções se interceptando δ vezes tem complexidade $O(\lambda_\delta(n))$. Somando todos os níveis da árvore temos:

$$\sum_{i=1}^{\lg n} i\lambda_\delta(n/i) \leq \lambda_\delta(n) \lg n.$$

□

Para analisarmos o cenário (δ, n, m), com m segmentos de curvas de grau δ tal que no máximo n segmentos coexistem no mesmo instante de tempo, primeiro provamos um lema:

LEMA 4.3. *Em um torneio cinético armazenando inicialmente n curvas de grau δ, sujeito a no máximo n inserções e n remoções, há no máximo $O(\lambda_{\delta+2}(n) \lg n)$ alterações na árvore. $O(\lambda_\delta(n) \lg^2 n)$.*

DEMONSTRAÇÃO. Consideraremos o torneio cinético varrendo um cenário com no máximo $2n$ segmentos de curvas de grau δ, pois o cenário do lema é um caso específico deste cenário. O envelope superior de $2n$ segmentos de curvas de grau δ tem complexidade $O(\lambda_{\delta+2}(n))$. Somando todos os níveis da árvore temos:

$$\sum_{i=1}^{\lg n} i\lambda_{\delta+2}(n/i) \leq \lambda_{\delta+2}(n) \lg n.$$

□

Deste lema segue:

TEOREMA 4.4. *Em um torneio cinético varrendo um cenário (δ, n, m), há no máximo $O(m/n\lambda_{\delta+2}(n) \lg n)$ alterações na árvore. Consequentemente, a complexidade de tempo do torneio cinético em um cenário (δ, n, m) é $O(m/n\lambda_{\delta+2}(n) \lg^2 n)$.*

DEMONSTRAÇÃO. Podemos dividir o cenário em m/n fatias verticais, tais que há no máximo n inserções e n remoções em cada fatia. Certamente

Cenário	Torneio Binário	Torneio $\lg n$-ário
$(\delta = 1, n)$ n retas	$O(n \lg^2 n)$	$O(n \lg^2 n / \lg \lg n)$
$(\delta = 1, n, m)$ m segmentos de reta, com n simultâneos	$O(m \lg^2 n)$	$O(m \lg^2 n / \lg \lg n)$
(δ, n) n curvas de grau δ	$O(n\lambda_\delta(n) \lg^2 n)$	$O(\lambda_\delta(n) \lg^2 n / \lg \lg n)$
(δ, n, m) m segmentos de curva de grau δ, com n simultâneos	$O(m/n\lambda_\delta(n) \lg^2 n)$	$O(m/n\lambda_\delta(n) \lg^2 n / \lg \lg n)$

TABELA 4.1. Complexidades de tempo do torneio cinético.

cada fatia inicia com no máximo n elementos armazenados. Usando o lema 4.3 em cada fatia, o teorema segue. □

Para melhorar as complexidades de tempo, quando o valor de n pode ser estimado previamente, é possível construir um torneio cinético $\lg n$-ário, conforme sugerido em [**6**]. A análise a versão $\lg n$-ária do torneio cinético é análoga e não será apresentada. Todas as complexidades de tempo são divididas por $\lg \lg(n)$. Para maior clareza, resumimos todas as complexidades de tempo na tabela 4.1.

CAPíTULO 5

Heater Cinético

Neste capítulo estudaremos a primeira estrutura a unir a localidade de $O(1)$ do heap cinético a eficiência provada do torneio cinético. Esta estrutura é randomizada, e as complexidades de tempo são esperanças, com relação a randomização interna da estrutura.

5.1. Versão Estática

Para introduzirmos o *heater*, precisamos definir o que é um *treap* [**1**]. Dado um conjunto de elementos, onde cada elemento tem uma prioridade e uma chave (ambos pertencentes a um conjunto totalmente ordenado e todos distintos), um treap é a única árvore que é ao mesmo tempo uma árvore binária de busca com relação às chaves e respeita a ordenação de heap nas prioridades. Uma árvore binária de busca é uma árvore binária enraizada onde todo vértice corresponde a um elemento de um conjunto ordenado e todo vértice é maior que seus descendentes na sub-árvore esquerda e menor que seus descendentes na sub-árvore direita. Um treap está ilustrado na figura 5.2. A prova de que, para qualquer conjunto de elementos, existe exatamente um treap é imediata a partir do algoritmo de construção da figura 5.1.

Claramente um treap com n elementos pode ter altura $n-1$, basta fazer a chave igual a prioridade. Este caso é extremamente ineficiente para qualquer aplicação razoável. Porém, em [**1**] mostrou-se que, caso as prioridades dos elementos sejam obtidas a partir de uma permutação aleatória, a altura esperada do treap é $O(\lg n)$. Este fato faz do treap uma estrutura de dados muito útil, suportando as operações de busca, inserção e remoção em tempo

```
Treap(S)
{
    Se |S| = 0
        Retorne
    x ← elemento de S com maior prioridade
    S_1 ← {e ∈ S|e.chave < x.chave}
    S_2 ← {e ∈ S|e.chave > x.chave}
    x.filhoEsquerdo ← Treap(S_1)
    x.filhoDireito ← Treap(S_2)
}
```

FIGURA 5.1. Algoritmo de construção do treap.

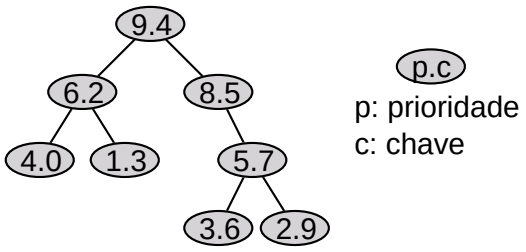

FIGURA 5.2. Exemplo de Treap.

esperado $O(\lg n)$. A busca é trivial, pois o treap é uma árvore binária de busca. Para inserirmos ou removermos elemento, entretanto, devemos usar rotações como as ilustradas na figura 5.3.

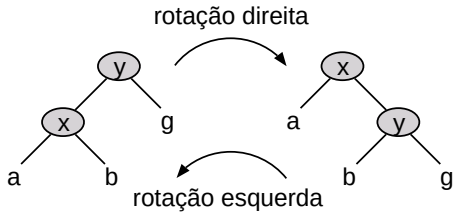

FIGURA 5.3. Rotações para a direita e esquerda.

Para inserir um elemento, fazemos a busca de sua chave e colocamos o novo elemento na respectiva folha. Enquanto a ordenação de heap for violada, fazemos a rotação apropriada, subindo o elemento em um nível. Para remover um elemento, usamos rotações para levar o elemento até uma folha e então removemos diretamente. Um resultado impressionante provado

em [**1**] é que o número esperado de rotações realizadas em uma operação de inserção ou remoção é menor ou igual a 2.

Em [**2**], foi proposta uma lista de prioridades cinética fortemente baseada no treap. No lugar das prioridades serem aleatórias e as chaves definidas, faz-se o contrário. Nesta estrutura, chamada *heater*, as chaves aleatórias servem para manter a árvore balanceada. Em princípio, isto parece pouco interessante, pois com a estratégia simples do heap, podemos manter a árvore deterministicamente balanceada. A grande vantagem do heater é que, devido a aleatoriedade da estrutura, foram obtidos resultados justos para sua complexidade.

As inserções e remoções no heater são realizadas de modo análogo ao treap. Infelizmente, o limite de $O(1)$ para o número esperado de rotações em uma inserção ou remoção não é mais válido. Isto acontece devido ao fato da prioridade do elemento inserido não ser aleatória, podendo ser sempre maior do que a de todos os demais elementos, por exemplo. Ainda assim, as complexidades da inserção e remoção são claramente limitadas pela altura da árvore. O lema a seguir (provado em [**2**]) afirma que a árvore é balanceada no caso esperado, tendo altura logarítmica.

LEMA 5.1. *O valor esperado da altura de um heater com n elementos é $O(\lg n)$.*

DEMONSTRAÇÃO. Sabe-se que o valor esperado da altura de um *treap* com n elementos é $O(\lg n)$. No heater, as chaves aleatórias definem uma bijeção entre a ordenação das prioridades e a ordenação das chaves. Esta bijeção aleatória é escolhida uniformemente dentre as $n!$ bijeções. Portanto, a estrutura probabilística da árvore é a mesma que se as chaves fossem definidas e as prioridades aleatórias, o que é o caso do treap. □

Como o heater é uma árvore com ordenação de heap, o elemento de maior prioridade está sempre na raiz, podendo ser determinado eficientemente.

5.2. Versão Cinética

Os eventos do heater são agendados da mesma maneira que no heap. Para cada nó interno, agenda-se um evento para o instante de tempo em que o primeiro filho do nó irá tornar-se maior que o pai. Processar o evento no heater, entretanto, é diferente de processar um evento no heap. No lugar de apenas trocar dois vértices pai e filho, é necessário fazer uma rotação para a direita ou para a esquerda. Ainda assim, a localidade da estrutura é claramente $O(1)$.

Para analisarmos o número de eventos processados em um heater cinético, precisamos fazer uma análise probabilística. Os resultados a seguir também são de [**2**]. Primeiro provamos um lema:

LEMA 5.2. *Em um heater aleatório com pelo menos $k \geq 2$ elementos, a probabilidade do k-ésimo elemento de maior prioridade ser filho do $k-1$-ésimo elemento de maior prioridade é igual a $2/k$.*

DEMONSTRAÇÃO. Vamos chamar de f_i o i-ésimo elemento de maior prioridade. Caso o heater tenha $n > k$ elementos, podemos desconsiderar todos os elementos de f_{k+1} até f_n, podando a árvore com a remoção desses elementos. Assim, restringimos nossa análise ao caso de um heater armazenando exatamente k elementos.

Certamente o elemento f_k é uma folha. Este elemento f_k será filho de f_{k-1} se e só se não existir nenhum elemento (de f_1 até f_{k-2}) com chave entre a chave de f_k e a chave de f_{k-1}. Como a ordenação das chaves é uma permutação aleatória, a probabilidade das chaves de f_k e f_{k-1} serem consecutivas na permutação é $2/k$. □

Então, nos baseando em um teorema de Clarkson e Shor [**8**] concluímos:

TEOREMA 5.3. *O número esperado de eventos processados por um heater cinético em um cenário (δ, n) é $O(\lambda_\delta(n) \lg n)$. Cada evento leva tempo $O(\lg n)$ para ser processado.*

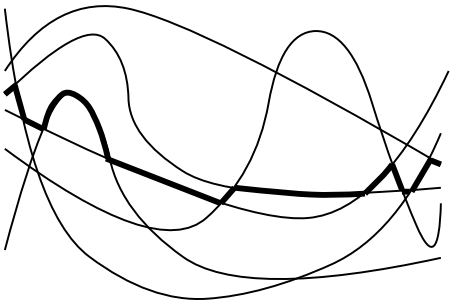

FIGURA 5.4. Ilustração do k-ésimo nível em um arranjo de curvas. O terceiro nível está destacado.

DEMONSTRAÇÃO. O tempo de processamento de cada evento é claramente $O(\lg n)$, tempo para reagendar eventos na fila de eventos. Provaremos o limite do número de eventos processados.

No capítulo 3, definimos o k-ésimo nível em um arranjo de retas. Agora precisamos extender esta definição a curvas. Definimos então o k-ésimo nível em um arranjo de funções do tempo como a seqüência de segmentos correspondentes a k-ésima função de maior valor ao longo do tempo. Um exemplo está na figura 5.4.

Denotamos por m_k o número de vértices no k-ésimo nível do arranjo Λ. Pelo lema 5.2, e usando a linearidade da esperança, o número esperado de eventos é

$$\sum_{k=1}^{n-1} m_k 2/k = \sum_{k=1}^{n-1} m_k O(1/k).$$

Usando soma por partes, substituímos m_k por sua soma parcial $M_k = \sum_{1\leq i\leq k} m_i$. Em [**8**], Clarkson e Shor provaram que $M_k = O(k\lambda_\delta(n))$. Fazendo os cálculos:

$$\begin{aligned}\sum_{k=1}^{n-1} m_k O(1/k) &= M_k O(1/k)\Big|_1^n + \sum_{k=1}^{n-1} M_{k+1} O(1/k^2) \\ &= O(\lambda_\delta(n)) + \sum_{k=1}^{n-1} O(\lambda_\delta(n)/k) \\ &= O(\lambda_\delta(n)) + O(\lambda_\delta(n)) \sum_{k=1}^{n-1} O(1/k) \\ &= O(\lambda_\delta(n) \lg n)\end{aligned}$$

□

A prova do próximo teorema é análoga, por isso omitimos:

TEOREMA 5.4. *O número esperado de eventos processados por um heater cinético em um cenário* (δ, n, m) *é* $O(m/n\lambda_{\delta+2}(n) \lg n)$. *Cada evento leva tempo* $O(\lg n)$ *para ser processado.*

CAPíTULO 6

Hanger Cinético

Vimos duas estruturas com localidade $O(1)$ até agora: o heap cinético e o heater cinético. O heap cinético é uma estrutura bastante simples de entender e implementar, porém não são conhecidos limites justos e eficientes para o número máximo de eventos processados. Em contrapartida, temos o heater cinético que, embora possua uma análise justa, é ligeiramente mais complexo e tem duas desvantagens técnicas: a necessidade de armazenar chaves e de realizar rotações, além de não possuir versão k-ária conhecida. Para unir os pontos positivos dessas duas estruturas, o autor criou uma nova estrutura randomizada chamada hanger cinético [**10**].

O hanger cinético e seus algoritmos são bastante semelhantes ao heap cinético. Primeiro apresentamos a versão estática.

6.1. Versão Estática

Um hanger é uma árvore binária com ordenação de heap. Assim como no heap e no heater, existe uma bijeção entre os vértices da árvore e os elementos. O critério usado para balanceamento é randomizado. Dado um conjunto S com n elementos, o algoritmo da figura 6.1 constrói um hanger com estes elementos. Neste processo de construção inicial, os elementos são inseridos em ordem, do elemento de maior prioridade, para o elemento de menor prioridade. Todo elemento é associado ao primeiro vértice da árvore que ainda não está associado a nenhum elemento, sempre começando da raiz. Caso o vértice já esteja associado a outro elemento, usa-se um bit aleatório para descer para o filho direito ou esquerdo do vértice da árvore, recursivamente.

```
Hanger(S)
{
    Para todo e ∈ S, em ordem decrescente de prioridade
        Hang(e, raiz);
}
Hang(elemento e, nó v)
{
    Se v.elem = ε
        v.elem ← e
        Retorne
    b ← BitAleatório()
    Hang (e, v.filho_b)
}
```

FIGURA 6.1. Algorimo para construir um hanger com um conjunto inicial de elementos.

Pode-se notar pelo algoritmo de construção que a árvore gerada não é necessariamente cheia. Entretanto, os bits aleatórios servem para balancear a árvore, como provaremos mais tarde. Os algoritmos de inserção e remoção de elementos em um hanger estão nas figuras 6.2 e 6.3.

A única diferença da inserção para o processo inicial de construção é que, quando se descobre que o nó em questão já está associado a outro elemento, deve-se verificar qual o elemento de maior prioridade. Caso o elemento que estamos inserindo tenha prioridade maior que o elemento que está previamente associado ao nó, os dois elementos devem ser trocados.

O algoritmo de deleção é extremamente simples. Iniciamos no vértice associado ao elemento que desejamos remover, sempre descendo na árvore subindo um nível com o filho de maior prioridade.

Para analisarmos a complexidade da estrutura, precisamos provar uma importante propriedade das operações de inserção e remoção: dado um um conjunto de elementos S, qualquer que seja a seqüência de inserções e remoções, a partir de um hanger com um conjunto inicial de elementos qualquer, que resulte em um hanger armazenando S, a distribuição de probabilidade do hanger obtido é a mesma. Por isto dizemos que as operações de inserção e remoção preservam a aleatoriedade da estrutura. Graças a esta

```
Inserir(elemento e, nó v)
{
      Se v.elem = ε
            v.elem ← e
            Retorne
      Se v.elem < e
            Troque os valores de e e v.elem
      b ← BitAleatório()
      hang (e, v.filho_b)
}
```

FIGURA 6.2. Algoritmo para inserir um elemento em um hanger.

```
Remover(nó v)
{
      Se v tem pelo menos um filho não nulo
            v' ← filho de v com maior prioridade
            v.elem ← v'.elem
            Remover(v')
      Senão
            v.elem ← ε
}
```

FIGURA 6.3. Algoritmo para remover um elemento em um hanger.

propriedade, ao considerarmos um hanger com um conjunto de elementos S, não precisamos nos preocupar com o modo como ele foi obtido, se foram ou não realizadas inserções e remoções.

Nos lemas abaixo, denotamos o hanger criado com o algoritmo *Hanger* a partir de um conjunto de elementos S por $Hanger(S)$, o hanger obtido pela inserção de um elemento f em um hanger h por $Inserir(h, f)$ e o hanger obtido pela remoção de um elemento f em um hanger h por $Remover(h, f)$. A prova do primeiro lema é simples e detalhes são omitidos.

LEMA 6.1. *Inserção preserva aleatoriedade, ou seja,* $Pr(Hanger(S \cup \{f\}) = h) = Pr(Inserir(hanger(S), f) = h)$.

DEMONSTRAÇÃO. Digamos que cada elemento tenha uma seqüência infinita de bits aleatórios gerados previamente. Considere que cada chamada a *BitAleatório*, tanto no procedimento *Inserir* quanto no procedimento *Hang*,

retorne o primeiro bit aleatório não usado na seqüência de bits do elemento e dos algoritmos das figuras 6.1 e 6.2 (esse bit é agora considerado usado). É fácil ver que o hanger construído por inserções sucessivas, em qualquer ordem, e o hanger construído com o procedimento *hanger* será o mesmo se e só se todos os bits aleatórios usados forem iguais. O lema segue como conseqüência. □

A prova do próximo lema é semelhante e será omitida.

LEMA 6.2. *Remoção preserva aleatoriedade, ou seja,* $Pr(Hanger(S \setminus f) = h) = Pr(Remover(hanger(S), f) = h)$.

Para analisarmos a complexidade de tempo da inserção, remoção e construção do hanger, definimos a variável aleatória L_n como a distância do n-ésimo maior elemento armazenado até a raiz do hanger. Claramente, em um hanger com n elementos, os procedimentos Hang, Inserir e Remover têm complexidade de tempo $O(E(L_n))$.

LEMA 6.3. $E(L_n) \leq \lg n$.

DEMONSTRAÇÃO. No lugar de limitarmos $E(L_n)$ diretamente, provaremos que $E(2^{L_n}) = n$. Pela desigualdade de Jensen, $2^{E(x)} \leq E(2^x)$ e o lema segue.

Seja r a raiz de um hanger com n elementos e r_e, r_d as sub-árvores enraizadas nos filhos esquerdo e direito de r, respectivamente. O número de elementos em r_e é uma variável aleatória binomial com probabilidade $1/2$, variando de 0 até $n-1$. O mesmo vale para r_d. O número de elementos na mesma sub-árvore (ou r_e ou r_d) do n-ésimo maior elemento é 1 mais uma variável aleatória binomial com probabilidade $1/2$, variando de 0 até $n-2$. Com isto, podemos construir a recorrência:

$$E\left(2^{L_n}\right) = \sum_{i=0}^{n-2} \binom{n-2}{i} \frac{1}{2^{n-2}} 2E\left(2^{L_{i+1}}\right).$$

Para provarmos que $E(2^{L_n}) = n$ usamos indução. Aplicando a hipótese de indução e a propriedade binomial de absorção [**11**], que diz

$$k\binom{n}{k} = \binom{n-1}{k-1}n, \text{ obtemos:}$$

$$\begin{aligned}
E(2^{L_n}) &= \sum_{i=0}^{n-2} \binom{n-2}{i} \frac{1}{2^{n-3}}(i+1) \\
&= \frac{1}{2^{n-3}} \left(\sum_{i=0}^{n-2} \binom{n-2}{i} + \sum_{i=1}^{n-2} \binom{n-2}{i} i \right) \\
&= \frac{1}{2^{n-3}} \left(2^{n-2} + (n-2) \sum_{i=0}^{n-3} \binom{n-3}{i} \right) \\
&= \frac{1}{2^{n-3}} \left(2^{n-2} + (n-2)2^{n-3} \right) \\
&= n.
\end{aligned}$$

□

Com isto terminamos a análise do hanger estático. A versão cinética é uma extensão natural.

6.2. Versão Cinética

Assim, como no heap cinético, mantemos uma lista de eventos agendando, para cada nó interno do hanger, o instante de tempo em que um dos filhos irá tornar-se maior que o pai. Para processarmos o evento, basta trocarmos as posições dos elementos pai e filho entre si, e reagendarmos os eventos adjacentes. Com isto, é claro que a localidade da estrutura é $O(1)$ e o tempo de processar um evento é $O(\lg n)$, tempo necessário para reagendar eventos. Antes de analisarmos o número máximo de eventos processados, enunciamos que o processamento de um evento preserva a aleatoriedade da estrutura. Este lema é semelhante aos lemas 6.1 e 6.2, e omitimos a prova. Como os elementos são funções do tempo, denotamos por $Hanger_{t-}(S)$ o hanger contendo os elementos de S logo antes do instante t e por $Hanger_{t+}(S)$ o hanger contendo os elementos de S logo após o instante t.

LEMA 6.4. *Interseção preserva aleatoriedade. Mais precisamente, seja* $h = Hanger_{t-}(S)$. *Se os dois elementos que se interceptam no tempo* t não *são pai e filho em* h, *então* $Pr(Hanger_{t-}(S) = h) = Pr(Hanger_{t+}(S) = h)$. *Se os dois elementos que se interceptam no tempo* t *são pai e filho em* h, *e* h' *é o hanger com esses dois elementos trocados entre si, então* $Pr(Hanger_{t-}(S) = h) = Pr(Hanger_{t+}(S) = h')$.

Agora podemos usar uma estratégia semelhante a usada no heater para limitarmos o número de eventos processados.

LEMA 6.5. *Seja* h *um hanger com* $m > n$ *elementos* $f_1 > f_2 > \ldots > f_m$. *A probabilidade de* f_{n+1} *ser filho de* f_n *em* h *é* $O(1/n)$.

DEMONSTRAÇÃO. Primeiro, a probabilidade de f_{n+1} ser filho de f_n é igual a probabilidade de f_j ser *descendente* de f_n, para qualquer $j > n$. Isso é claramente verdade, pois ambas as probabilidades são iguais a $1/2^{L_n}$. Definimos $p_i = 1/2^{L_i}$. Também definimos variáveis indicadoras v_i, que valem 1 se f_i é ancestral de f_n, e valem 0 caso contrário. Claramente, $E(v_i) = p_i$,

$$L_n = \sum_{i=1}^{n-1} v_i \text{ e}$$

$$E(L_n) = E\left(\sum_{i=1}^{n-1} v_i\right).$$

Usando linearidade da esperança e o Lemma 6.3,

$$\sum_{i=1}^{n-1} p_i \leq \lg n.$$

Como isto vale para todo n e p_i não depende de n, segue que $p_i = O(1/i)$. Usando calculo finito, pode-se inclusive mostrar que $p_i \leq 1/(i \ln 2)$.

□

Com este resultado, podemos provar os próximos teoremas, usando a mesma estratégia utilizada nesta tese para o heater:

TEOREMA 6.6. *O número esperado de eventos processados por um hanger cinético em um cenário* (δ, n) *é* $O(\lambda_\delta(n) \lg n)$. *Cada evento leva tempo* $O(\lg n)$ *para ser processado.*

TEOREMA 6.7. *O número esperado de eventos processados por um hanger cinético em um cenário* (δ, n, m) *é* $O(m/n\lambda_{\delta+2}(n) \lg n)$. *Cada evento leva tempo* $O(\lg n)$ *para ser processado.*

Tanto no heap cinético, quanto no torneio cinético, apresentamos uma versão k-ária das estruturas. Fazendo $k = \Theta(\lg n)$ foi possível obter melhoras na complexidade de tempo. O heater cinético não possui generalização k-ária conhecida. Já no caso do hanger cinético, é extremamente natural formular uma versão k-ária. No lugar de usarmos bits aleatórios, usamos uma variável aleatória com k valores possíveis.

A complexidade de tempo de processar um evento em um hanger k-ário com n elementos é $O(k + \lg(n/k))$. Se $k = \Theta(\lg n)$, a complexidade de tempo de processar um evento se mantém $O(\lg n)$. Temos que provar que $E(L_{k,n})$, o valor esperado da distância do n-ésimo maior elemento até a raiz do hanger k-ário, é $O(\log_k n)$.

LEMA 6.8. $E(L_{k,n}) = O(\log_k n)$.

DEMONSTRAÇÃO. A prova é muito semelhante a prova do lema 6.3. No lugar de limitarmos $E(L_{k,n})$ diretamente, provamos que $E(k^{L_{k,n}}) = (k-1)(n-1)+1$.

A recorrência probabilística fica:

$$E\left(k^{L_{k,n}}\right) = \sum_{i=0}^{n-2} \binom{n-2}{i} \left(\frac{1}{k}\right)^i \left(\frac{k-1}{k}\right)^{n-2-i} kE\left(k^{L_{k,i+1}}\right).$$

Usando novamente indução e a propriedade de absorção, como na prova do lema 6.3, provamos que $E(k^{L_{k,n}}) = (k-1)(n-1)+1$. As contas não serão apresentadas. □

A prova do lema a seguir é análoga a prova do lema 6.5.

LEMA 6.9. *Seja h um hanger k-ário com $m > n$ elementos $f_1 > f_2 > \ldots > f_m$. A probabilidade de f_{n+1} ser filho de f_n em h é $O(1/(n \lg k))$.*

E podemos analogamente provar os seguintes teoremas:

TEOREMA 6.10. *O número esperado de eventos processados por um hanger cinético k-ário em um cenário (δ, n) é $O(\lambda_\delta(n) \lg n / \lg k)$. Cada evento leva tempo $O(k + \lg(n/k))$ para ser processado. O produto do número de eventos e do tempo para processar cada evento é minimizado com $k = \Theta(\lg n)$.*

TEOREMA 6.11. *O número esperado de eventos processado por um hanger cinético k-ário em um cenário (δ, n, m) é $O(m/n\lambda_{\delta+2}(n) \lg n / \lg k)$. Cada evento leva tempo $O(k+\lg(n/k))$ para ser processado. O produto do número de eventos e do tempo para processar cada evento é minimizado com $k = \Theta(\lg n)$.*

CAPíTULO 7

Estruturas de fecho convexo dinâmico

Uma das ferramentas mais fascinantes da matemática consiste em reduzir um problema que desejamos resolver a outro problema que sabemos resolver. Esta técnica é amplamente usada na geometria computacional, principalmente graças a dualidade ponto-reta, que explicaremos neste capítulo. Com esta técnica, reduziremos o problema das listas de prioridade cinéticas (LPC) a uma outra estrutura de dados, a estrutura de fecho convexo dinâmico. Infelizmente, esta redução só funciona para o caso dos elementos serem funções lineares do tempo. Além disso, as estruturas de fecho convexo dinâmico mais eficientes conhecidas são muito complexas e de pouco interesse prático. Ainda assim, o uso dessas estruturas reduz a complexidade das LPC para funções lineares.

7.1. Fecho convexo dinâmico

Dado um conjunto S de pontos no plano, $FC(S)$, o fecho convexo de S, é o menor polígono convexo que contém todos os pontos de S (figura 7.1). Claramente os vértices do $FC(S)$ são pontos de S. Dado S, $FC(S)$ pode ser construído em tempo $O(n \lg n)$, usando vários algoritmos amplamente documentados na literatura. Mais adiante será importante para nós, decompor o fecho convexo em duas partes, chamadas de fecho convexo superior e fecho convexo inferior. A primeira consiste dos vértices do fecho convexo que se encontram acima do polígono e a segunda dos vértices que estão abaixo do polígono. Possivelmente, os vértices extremos da esquerda e direita do fecho convexo pertencem tanto ao fecho convexo superior quanto ao inferior.

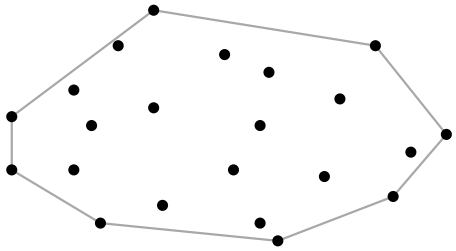

FIGURA 7.1. Fecho convexo de um conjunto de pontos.

Na versão dinâmica do problema, o conjunto de pontos S pode sofrer inserções e remoções. As estruturas de fecho convexo dinâmico atualizam $FC(S)$ para inserções e remoções em S. De fato, as estruturas mais modernas e eficientes não armazenam $FC(S)$ explicitamente, mas permitem várias consultas a $FC(S)$. Para nós, basta nos limitarmos a dois tipos de consulta: dado um vértice de $FC(S)$, determinar o próximo vértice no sentido horário; dada uma direção, determinar o vértice extremo naquela direção.

A avaliação de performance dessas estruturas é feita calculando a complexidade de tempo das operações de inserção, remoção e consultas. Uma das primeiras estruturas sugeridas (em 1981) foi a de Overmars e van Leeuwen [**12**]. Esta estrutura tem inserção e remoção em tempo $O(\lg^2 n)$, onde n é o número de elementos armazenados na estrutura. As consultas são bastante eficientes, pois $FC(S)$ é armazenado explicitemente.

Após esta estrutura, não houve nenhum progresso significativo até 2000, quando Chan publicou, em [**7**], uma estrutura com inserção e remoção em tempo amortizado de $O(\lg^{1+\varepsilon} n)$. A constante ε é arbitrariamente pequena, mas as constantes ocultas na notação O dependem de ε. As duas consultas que nos interessam levam tempo $O(\lg n)$. Esta estrutura é extremamente complexa e de pouco interesse prático, ainda que com algumas simplificações propostas em [**6**] especialmente para a aplicação em LPC.

Após a estrutura do Chan, surgiu outra estrutura de fecho convexo dinâmico, proposta por Brodal e Jacob em [**5**]. Esta nova estrutura realiza inserções e remoções em tempo $O(\lg n \lg\lg n)$.

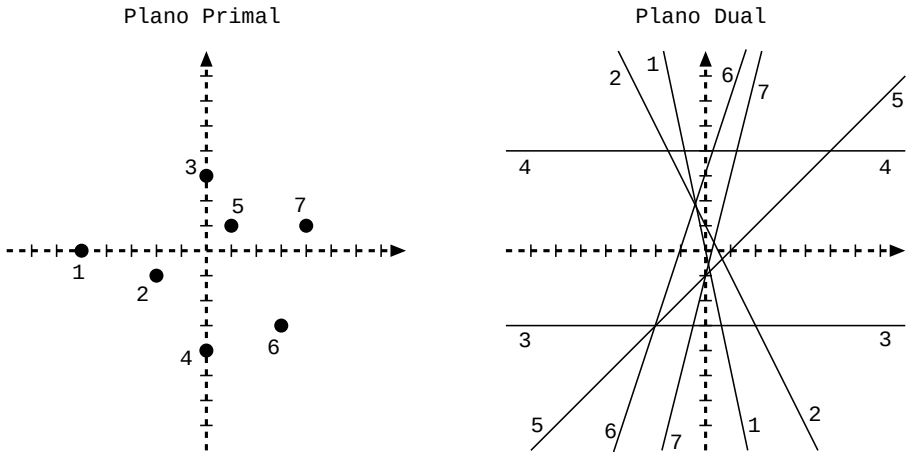

FIGURA 7.2. Pontos no plano primal e retas correspondentes no plano dual.

7.2. Dualidade ponto-reta

Cada ponto do plano cartesiano é definido por um par de coordenadas (α, β). Uma reta não vertical pode ser definida por uma única equação $y = ax - b$. Se temos um plano, chamado primal, contendo pontos e retas, podemos construir outro plano, chamado dual, da seguinte maneira: a cada ponto (α, β) do ponto primal associa-se uma reta $y = \alpha x - \beta$ no plano dual e a cada reta $y = ax - b$ no plano primal associa-se um ponto (a, b) no plano dual (figura 7.2).

Esta transformação não seria útil se não tivesse propriedades importantes (inclusive foi por causa dessas propriedades que apresentamos a equação da reta com o sinal negativo no coeficiente linear). Enunciamos algumas propriedades úteis abaixo. Reservamos as letras p para ponto e r para reta, possivelmente com índices. Definimos também $'$ como operador de dualidade. Assim, se $p = (\alpha, \beta)$, p' é a reta $y = \alpha x - \beta$, por exemplo.

Auto inverso: $(p')' = p$

Reversão de ordem: p está *acima* de r no plano primal se e só se p' está *abaixo* de r' no plano dual.

Preservar interseção: r_1 e r_2 se interceptam no ponto p se e só se p' passa por r_1' e r_2'.

Colinearidade/coincidência: Três pontos são colineares no plano primal se e só se as três retas correspondentes a eles no plano dual se interceptam em um único ponto comum.

Provar estas propriedades é um exercício simples. Uma outra propriedade, um pouco mais complexa, mas também fácil de ser demonstrada, relaciona o fecho convexo inferior com o envelope superior: Seja S um conjunto de pontos no plano primal e S' o conjunto de retas correspondente no plano dual. Os pontos do fecho convexo inferior de S, tomados no sentido horário, correspondem as retas do envelope superior de S', da esquerda para a direita. Podemos inclusive associar os segmentos de reta dos envelopes superior e inferior que interceptam uma reta vertical $x = k$ no plano dual aos pontos extremos nas direções perpendiculares a reta $y = kx$ no plano primal.

Na figura 7.2 algumas propriedades podem ser observadas. Os pontos com rótulos 3, 5 e 6 são colineares, assim como as retas correspondentes se interceptam em um único ponto. O fecho convexo inferior no plano primal é formado pelos pontos com rótulos 1, 4, 6 e 7, assim como o envelope superior no plano dual (na figura, a reta com rótulo 7 não aparece interceptando a reta com rótulo 6, mas interceptará mais acima). Os segmentos dos envelopes superior e inferior que interceptam a reta vertical $x = 1$ no plano dual pertencem as retas com rótulos 6 e 1. Consequentemente os pontos extremos nas direções $(1, -1)$ e $(-1, 1)$ tem rótulos 6 e 1.

7.3. Redução e complexidades

Com as informações que acabamos de ver, é fácil criar uma LPC que armazene funções lineares do tempo e suporte operações de inserção e remoção usando uma estrutura de fecho convexo dinâmico. As operações de inserção e remoção na LPC são apenas inserções e remoções na estrutura de fecho convexo dinâmico. Para determinar o elemento máximo em um intante de

tempo, basta determinar o ponto extremo na direção adequada. Avançar no tempo até que o máximo se altere consiste em consultar a próxima aresta.

A complexidade de tempo da LPC construída por esta redução a partir da estrutura mais eficiente conhecida (Brodal e Jacob, [**5**]) é dominada pelo tempo das operações de inserção e remoção. Com isto temos o seguinte teorema:

TEOREMA 7.1. *A complexidade de tempo de uma lista de prioridades cinética construída usando a estrutura de fecho convexo dinâmico de Brodal e Jacob varre um cenário de m segmentos de reta, onde no máximo n segmentos de reta são interceptados por uma reta vertical, em tempo $O(m \lg n \lg \lg n)$.*

A complexidade mostrada no teorema acima é mais eficiente que as demais estruturas usadas até agora. Porém, se baseia em uma estrutura de fecho convexo dinâmico extremamente complexa e poderosa. Aparentemente, a estrutura de fecho convexo dinâmico tem grande parte de seu potencial desperdiçado quando usada para construir uma LPC, que parece ser um problema significativamente mais simples. Algumas simplificações para a estrutura de fecho convexo dinâmico de Chan [**7**] foram propostas em [**6**], quando se deseja apenas construir uma LPC. Ainda assim, a estrutura é muito complexa para aplicações práticas. Uma LPC tão eficiente ou ainda mais eficiente que as obtidas através dessa redução e ao mesmo tempo mais simples é um problema de pesquisa.

CAPíTULO 8

Conclusão e Problemas Abertos

8.1. Aplicações

As estruturas de dados cinéticas tem diversas aplicações em vários problemas onde os dados variam continuamente com o tempo. Além destas aplicações, as listas de prioridades cinéticas podem ser usadas para resolver eficientemente problemas de natureza estática. Um exemplo simples que explicamos a seguir é o de computar o k-ésimo nível de um arranjo de curvas no plano. Outro exemplo está descrito em [**2**], onde dados um conjunto conexo de segmentos azuis e outro conjunto conexo de segmentos vermelhos, deseja-se computar todas as interseções entre os segmentos vermelhos e azuis.

No capítulo 5, definimos o k-ésimo nível de um arranjo de funções do tempo. Relembramos a definição aqui. O k-ésimo nível em um arranjo de funções do tempo é a seqüência de segmentos correspondentes a k-ésima função de maior valor ao longo do tempo (figura 8.1). O problema de, dado o arranjo de curvas, computar seu k-ésimo nível de modo eficiente não é nada trivial. Surpreendentemente, as LPC nos fornecem uma maneira bastante simples e eficiente de realizar esta computação [**6**].

Criamos duas LPC, K_{sup} e K_{inf}. A LPC K_{sup} armazena inicialmente os k maiores elementos no tempo inicial e armazena os elementos em ordem inversa, ou seja, permitindo a determinação do elemento mínimo, e não do máximo. A LPC K_{inf} armazena os demais elementos, e permite a determinação do elemento máximo. Ao longo do tempo, computa-se o instante em que o elemento máximo de K_{inf} interceptará o elemento mínimo de K_{sup}. Caso os elementos se interceptem realmente nessas posições, neste instante,

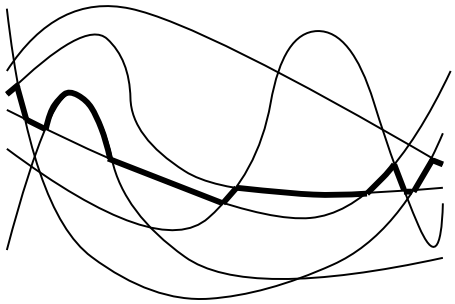

FIGURA 8.1. Ilustração do k-ésimo nível em um arranjo de curvas. O terceiro nível está destacado.

remove-se o elemento máximo de K_{inf} e insere-o em K_{sup}. Ao mesmo tempo, remove-se o elemento mínimo de K_{sup} e insere-o em K_{inf}. Desta maneira, segue-se varrendo o conjunto de curvas e computando o k-ésimo nível, que é a seqüência de elementos mínimos em K_{sup}.

8.2. Comparando as LPC Apresentadas

Apresentamos nesta tese cinco estruturas para construção de listas de prioridades cinéticas: heap cinético, torneio cinético, heater cinético, hanger cinético e redução a fecho convexo dinâmico. Apresentamos agora as complexidades de tempo das estruturas na tabela 8.1 e analisamos as vantagens e desvantagens destas estruturas em alguns quesitos: complexidade de tempo, localidade, randomização e praticidade.

1) Complexidade de tempo: O torneio cinético, heater cinético e hanger cinético têm a mesma complexidade de tempo, sendo bastante eficientes. Ainda assim a estrutura de fecho convexo dinâmico é ligeiramente mais eficiente quando os elementos são funções lineares do tempo. As complexidades de tempo do torneio cinético e do hanger cinético podem ser reduzidas usando as versões $(\lg n)$-árias das estruturas. A complexidade de tempo do heap cinético ainda é desconhecida na maior parte dos casos.
2) Localidade: O heap cinético, heater cinético e hanger cinético têm localidade ótima de $O(1)$. O torneio cinético tem localidade de

$O(\lg n)$. Esta medida não se aplica a redução a fecho convexo dinâmico.

3) Randomização: O heater cinético e o hanger cinético são estruturas randomizadas. A complexidade de tempo dessas estruturas é medida como uma esperança que independe dos dados da entrada. As demais estruturas são determinísticas.

4) Praticidade: Com exceção da redução a fecho convexo dinâmico, todas as estruturas apresentadas são de fácil implementação e baixas constantes multiplicativas na complexidade de tempo.

8.3. Problemas Abertos

Há vários problemas abertos relacionados às listas de prioridades cinéticas. A estrutura com a menor complexidade de tempo conhecida, a redução a fecho convexo dinâmico, é restrita a funções lineares do tempo e é extremamente complexa. Algumas perguntas surgem. É possível baixar ainda mais esta complexidade de tempo? É possível obter resultados similares quando as funções não são lineares? Existe uma estrutura mais simples que atinja a mesma complexidade de tempo?

#	Cenário	Heap	Heater, Hanger e Torneio	FCD
1	Retas	$O(n \lg^2 n)$	$O(n \lg^2 n)$	$O(n \lg n)$
2	Segmentos de retas	$O(m\sqrt{n} \lg^{3/2} n)$	$O(m\alpha(n) \lg^2 n)$	$O(m \lg n \lg \lg n)$
3	curvas de grau δ	$O(n^2 \lg n)$	$O(\lambda_\delta(n) \lg n)$	n/d
4	Segmentos de grau δ	$O(mn \lg n)$	$O(m/n\lambda_{\delta+2}(n) \lg n)$	n/d

TABELA 8.1. Complexidades de tempo das estruturas. Nos cenários 2 e 4, o parâmetro n denota o maior número de elementos simultaneamente armazenados na estrutura e m denota o número total de segmentos. As complexidades do heap no cenário 1 e do torneio e hanger em todos os cenários podem ser reduzidas por um fator de $O(\lg \lg n)$, usando-se versões $(\lg n)$-árias.

A complexidade de quase todas as estruturas vistas na tese encontra-se bem definida. Uma exceção é o heap cinético. Determinar o número máximo de eventos processados em um heap cinético armazenando curvas e/ou sujeito a inserções e remoções são problemas intrigantes.

Bibliografia

[1] C. R. Aragon and R. G. Seidel. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[2] J. Basch, L. J. Guibas, and G. D. Ramkumar. Reporting red-blue intersections between two sets of connected line segments. In *Proceedings of the 4th Annual European Symposium on Algorithms*, volume 1136 of *Lecture Notes Comput. Sci.*, pages 302–319. Springer-Verlag, 1996.

[3] J. Basch, L. J. Guibas, and G. D. Ramkumar. Sweeping lines and line segments with a heap. In *Proceedings of the 13th Annual Symposium on Computational Geometry*, pages 469–471, 1997.

[4] Julien Basch, Leonidas J. Guibas, and John Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31(1):1–28, April 1999.

[5] G. S. Brodal and R. Jacob. Dynamic planar convex hull with optimal query time. In *Proceedings of the 6ty Scandinavian Workshop on Algorithm Theory, SWAT2000*, pages 57–70, 2000.

[6] T. M. Chan. Remarks on k-level algorithms in the plane. http://www.math.uwaterloo.ca/~tmchan/lev2d_7_7_99.ps.gz, 1999.

[7] T. M. Chan. Dynamic planar convex hull operations in near-logarithmic amortized time. *Journal of the ACM*, 48:1–12, 2001.

[8] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, ii. *Discrete and Computational Geometry*, 4:387–421, 1989.

[9] G. D. da Fonseca and C. M. H. de Figueiredo. Kinetic heap-ordered trees: tight analysis and improved algorithms. *Information Processing Letters*, 85/3:165–169, January 2003.

[10] G. D. da Fonseca, C. M. H. de Figueiredo, and P. C. P. de Carvalho. Kinetic hanger. *submetido para Information Processing Letters*, 2002.

[11] R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics - A Foundation for Computer Science*. Addison-Wesley Publishing Company, 1994.

[12] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Sys. Sci.*, 23:166–204, 1981.

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