Caio Dias Valentim

Estruturas de Dados para Séries Temporais

DISSERTAÇÃO DE MESTRADO

DEPARTAMENTO DE INFORMÁTICA
Programa de Pós–graduação em Informática

Rio de Janeiro
Julho de 2012

Pontifícia Universidade Católica
do Rio de Janeiro

Caio Dias Valentim

Estruturas de Dados para Séries Temporais

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática do Departamento de Informática da PUC-Rio

Orientador: Prof. Eduardo Sany Laber

Rio de Janeiro
Julho de 2012

Pontifícia Universidade Católica
do Rio de Janeiro

Caio Dias Valentim

Estruturas de Dados para Séries Temporais

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC-Rio. Aprovada pela comissão examinadora abaixo assinada.

Prof. Eduardo Sany Laber
Orientador
Departamento de Informática — PUC-Rio

Prof. David Sotelo Pinheiro da Silva
Departamento de Informática - PUC-Rio

Prof. Fabio Andre Machado Porto
Laboratório Nacional de Computação Científica

Prof. Raúl Pierre Rentería
Departamento de Informática - PUC-Rio

Prof. José Eugenio Leal
Coordenador Setorial do Centro Técnico Científico - PUC-Rio

Rio de Janeiro, 31 de Julho de 2012

Caio Dias Valentim

Graduou-se em Sistemas de Informação pela PUC-Rio. Trabalhou em projetos no laboratório LEARN, da PUC-Rio, onde ajudou em pesquisas nas áreas de *Web-crawling* e *Data-mining*. Trabalhou no R&D da empresa Fast - A Microsoft Subsidiary em problemas de NLP (Natural Language Processing). Foi até a China como finalista mundial da competição de programação ACM-ICPC, realizada pela ACM.

Ficha Catalográfica

Valentim, Caio

Estruturas de Dados para Séries Temporais / Caio Dias Valentim; orientador: Eduardo Sany Laber. — Rio de Janeiro : PUC-Rio, Departamento de Informática, 2012.

v., 57 f: il. ; 29,7 cm

1. Dissertação (Mestrado em Informática) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1. Informática – Tese. 2. Estrutura de Dados. 3. Séries Temporais. 4. Algoritmos. 5. Mercado Financeiro. I. Laber, Eduardo. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Agradecimentos

Ao meu orientador, Eduardo Laber, pelos seis anos de colaboração e por ter acreditado no potencial de um garoto no segundo período da faculdade que mal sabia programar. Obrigado por ter ajudado, ao longo desses anos, no meu desenvolvimento pessoal e profissional.

Ao meu co-orientador, David Sotelo, pela importante participação e contribuição no desenvolvimento desse trabalho.

Aos meus amigos da maratona de programação, em especial aos meus amigos do *Dynasty of Samba*, Eduardo Teixeira, Guilherme De Napoli e Daniel Fleischman.

A minha mãe, Josélia Dias Valentim, e minha família por sempre acreditarem em mim.

Aos meus amigos pelo apoio.

Ao CNPq e a PUC-Rio, pelos auxílios concedidos, sem os quais este trabalho não poderia ter sido realizado.

Resumo

Valentim, Caio; Laber, Eduardo. **Estruturas de Dados para Séries Temporais**. Rio de Janeiro, 2012. 57p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Séries temporais são ferramentas importantes para análise de eventos que ocorrem em diferentes domínios do conhecimento humano, como medicina, física, meteorologia e finanças. Uma tarefa comum na análise de séries temporais é a busca por eventos pouco frequentes que refletem fatos de interesse sobre o domínio de origem da série. Neste trabalho, buscamos desenvolver técnicas para detecção de eventos raros em séries temporais. Formalmente, uma série temporal $A = (a_1, a_2, \ldots, a_n)$ é uma sequência de valores reais indexados por números inteiros de 1 a n. Dados dois números, um inteiro t e um real d, dizemos que um par de índices i e j formam um evento-(t, d) em A se, e somente se, $0 < j - i \leq t$ e $a_j - a_i \geq d$. Nesse caso, i é o *início* do evento e j o *fim*. Os parâmetros t e d servem para controlar, respectivamente, a janela de tempo em que o evento pode ocorrer e a magnitude da variação na série. Assim, nos concentramos em dois tipos de perguntas relacionadas aos eventos-(t, d), são elas:

- Quais são os eventos-(t, d) em uma série A?
- Quais são os índices da série A que participam como inícios de ao menos um evento-(t, d)?

Ao longo desse trabalho estudamos, do ponto de vista prático e teórico, diversas estruturas de dados e algoritmos para responder às duas perguntas listadas.

Palavras–chave

Estrutura de Dados ; Séries Temporais ; Algoritmos ; Mercado Financeiro .

Abstract

Valentim, Caio; Laber, Eduardo (advisor). **Data Structures for Time Series**. Rio de Janeiro, 2012. 57p. MSc. Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Time series are important tools for the anaylsis of events that occur in different fields of human knowledge such as medicine, physics, meteorology and finance. A common task in analysing time series is to try to find events that happen infrequently as these events usually reflect facts of interest about the domain of the series. In this study, we develop techniques for the detection of rare events in time series. Technically, a time series $A = (a_1, a_2, \ldots, a_n)$ is a sequence of real values indexed by integer numbers from 1 to n. Given an integer t and a real number d, we say that a pair of time indexes i and j is a (t, d)-event in A, if and only if $0 < j - i \leq t$ and $a_j - a_i \geq d$. In this case, i is said to be the *beginning* of the event and j is its *end*. The parameters t and d control, respectively, the time window in which the event can occur and magnitude of the variation in the series. Thus, we focus on two types of queries related to the (t, d)-events, which are:

– What are the (t, d)-events in a series A?

– What are the indexes in the series A which are the beginning of at least one (t, d)-event?

Throughout this study we discuss, from both theoretical and practical points of view, several data structures and algorithms to answer the two queries mentioned above.

Keywords

Data Structures ; Time Series ; Algorithm ; Stock Market .

Sumário

Lista de figuras

Lista de tabelas

Amazing citation by someone.

Someone, *Someone's book.*

1
Introdução

O estudo de séries temporais é motivado por aplicações que surgem em diferentes campos do conhecimento humano tais como medicina, física, meteorologia e finanças. Para algumas aplicações envolvendo séries temporais, planílhas e bancos de dados são suficientes para a análise desejada. Contudo, existem aplicações em que as séries são grandes de mais, como na análise de dados capturados em milissegundos por sensores ou na análise de transações em mercados financeiros eletrônicos. Para essas séries, técnicas tradicionais de armazenamento de dados podem ser inadequadas devido ao consumo excessivo de tempo/espaço. Esse cenário motiva a pesquisa de estruturas de dados e algoritmos para lidar com longas séries temporais[12].

O foco de nossa pesquisa é desenvolver estruturas de dados que permitam a identificação/descoberta de certos tipos de eventos em séries temporais. Em particular, estamos interessados em mudanças significativas nas séries que ocorrem em um intervalo pequeno de tempo. Para capturar a essência dos eventos definimos um modelo onde a entrada é uma série temporal com índices inteiros de tempo e uma consulta é feita através de dois valores t e d. Esses valores buscam restringir a janela de tempo t em que o evento pode ocorrer e a magnitude d da variação. A partir desse modelo identificamos duas consultas de interesse, denominadas $AllPairs$ e $Beginnings$, as quais possuem correlação com determinados padrões comuns em séries temporais. Buscamos motivar o modelo e os dois tipos de consultas com perguntas reais comumente feitas por analistas de mercado.

Durante esse trabalho apresentamos e discutimos diferentes estruturas de dados para responder às consultas $AllPairs$ e $Beginnings$ de forma eficiente, e com espaço/tempo de criação não excessivo.

Por fim, analisamos as estruturas propostas do ponto de vista teórico e experimental.

1.1 Definição do Problema

Seja $A = (a_1, a_2, \ldots, a_n)$ uma série temporal com n elementos e seja um *índice de tempo* um inteiro do conjunto $\{1, \ldots, n\}$. Usamos dois números, um inteiro positivo t e um real d, para capturar variações na série A. De forma mais precisa, dizemos que um par de índices i e j determinam um *evento*-(t, d) em A, se e somente se:

- $0 < j - i \leq t$ e,
- $a_j - a_i \geq d$.

Por exemplo, se $A = (3, 15, 7, 16)$, $t = 2$ e $d = 3$, os eventos-(t, d) são $\{(1, 2), (1, 3), (3, 4)\}$.

Nosso objetivo é criar estruturas de dados para pré-processar/indexar séries temporais de forma a responder às seguintes consultas em tempo sublinear e com tempo/espaço de pré-processamento reduzido:

- $AllPairs(t, d)$. Esta consulta retorna todos os eventos-(t, d) na série A.
- $Beginning(t, d)$. Esta consulta retorna todos os índices i para os quais existe um índice j tal que (i, j) determina um evento-(t, d) na série A.

Na série A de exemplo, temos $AllPairs(2, 3) = \{(1, 2), (1, 3), (3, 4)\}$ e $Beginnings(2, 3) = \{1, 3\}$. Nos concentraremos nas consultas acima pois acreditamos que sejam fundamentais para o estudo de mudanças rápidas em séries temporais, tendo em vista que técnicas desenvolvidas para essas consultas podem ser estendidas para outras consultas de interesse frequente em séries temporáis, como as apresentadas na Seção 1.2.

Para facilitar a apresentação dos resultados iremos nos concentrar em variações positivas ($d > 0$) nas nossas consultas. Contudo, todas as técnicas desenvolvidas aqui também se aplicam para variações negativas.

A seguinte notação será útil ao longo do trabalho. O *valor*-Δ do par (i, j) é dado por $j - i$ e o *desvio* por $a_j - a_i$. Dizemos que i é o *início* do par (i, j) e j é o *fim*.

1.2 Aplicações

Para tornar a abordagem mais concreta, listamos aqui aplicações onde o processamento das consultas propostas é útil.

Os preços de ações com características similares (ex.: que pertencem ao mesmo segmento da indústria) normalmente têm uma forte correlação. Uma estratégia de negociação muito popular no mercado financeiro é estimar a razão entre preços de duas ações similares e apostar que a razão voltará para a estimada sempre que, por algum motivo, ela desviar muito do valor[5].

Um arquiteto de estratégias pode estudar se variações raras tendem a voltar para sua média (valor normal) e, para isso, ele pode especificar uma longa lista de pares (t, d) e analisar o comportamento das séries temporais logo após a ocorrência de eventos-(t, d), para cada par (t, d) na lista. Uma busca linear simples na série exigiria tempo $O(nt)$. Em alguns mercados de ações (ex.: NYSE), onde mais de 3000 ações são negociadas diariamente, e um número de pares muito maior pode ser formado, essa busca seria computacionalmente muito cara. Então, estratégias mais eficientes para busca de pares (t, d) podem ser de interesse.

Outra situação relacionada acontece quando são dados um intervalo de tempo t e um limite p (ex.: 0.01) e gostaríamos de saber o valor mínimo d para que tenhamos no máximo $p \times n$ eventos-(t, d), isto é, tenhamos eventos raros. Uma consulta como essa pode ser respondida através de uma busca binária no valor d, onde para cada valor d considerado o número de eventos-(t, d) são contados. Assim, um número logarítmico em d de consultas $Beginning(t, d)$ e/ou $AllPairs(t, d)$ é feito somente para uma consulta. Se quisermos processar a consulta para um número grande de intervalos t e limites p, responder eficientemente às consultas originais é mandatório.

Mais uma aplicação em potencial é o estudo de eventos globais/locais em mercados financeiros. Ações no mercado podem ser caracterizadas pelos seus preços e volumes negociados. Dada uma coleção de séries temporais, cada uma contendo o preço e volume de cada ação, medidos a cada segundo, durante 10 anos, gostaríamos de responder às seguintes consultas: em que janela de tempo de k segundos, a maior parte das ações mudaram em pelo menos $\%p$ do seu preço e $\%v$ em seu volume de negociação? Essas consultas podem ser úteis para detecção de eventos globais/locais, como alguma notícia de impacto no mercado.

1.3 Nossa Contribuição

Desenvolvemos e analisamos algoritmos e estruturas de dados para processar as consultas $AllPairs(t, d)$ e $Beginning(t, d)$. Os parâmetros relevantes em nossa análise são o tempo necessário para pré-processar/criar a estrutura, o espaço ocupado pela mesma e o tempo de consulta.

Para apreciar nossa contribuição é útil primeiro discutir formas em potencial para responder às consultas.

Um algoritmo ingênuo para a consulta $AllPairs(t, d)$ percorre a lista A e, para cada possível início i, reporta os pares (i, j) para todo j no intervalo $[i + 1, i + t]$ tal que $a_j - a_i \geq d$. Esse procedimento responde à consulta $AllPairs(t, d)$ em tempo $O(nt)$. Uma estratégia similar pode ser usada para o $Beginnings(t, d)$ e também resulta em tempo $O(nt)$.

Outra estratégia simples para a consulta $Beginnings(t, d)$ é usar uma tabela T, indexada de 1 até $n-1$; onde a posição $T[t]$, para cada t, tem um ponteiro para uma lista contendo os pares no conjunto $\{(1, m_1), (2, m_2), \ldots, (n - 1, m_{n-1})\}$, onde m_i é índice do maior valor entre $a_{i+1}, a_{i+2}, \ldots, a_{i+t}$. Cada uma dessas $n - 1$ listas é ordenada em ordem não-crescente de desvio dos pares. Para responder à consulta, percorremos a lista $T[t]$ e reportamos o início de todo par que é um evento-(t, d), parando de percorrer quando encontramos um par com desvio menor que d. Não é difícil notar que o procedimento descrito responde à consulta $Beginnings(t, d)$ em tempo $O(k+1)$[1], onde k é o número de inícios reportados. Na mesma linha, também podemos construir uma estrutura para $AllPairs(t, d)$ que responde às consultas em tempo $O(k+1)$, porém, em ambas as estruturas precisaríamos armazenar $\Theta(n^2)$ pares no pior caso.

As soluções que propomos nessa trabalho fazem uso extensivo de estruturas de dados que suportem RMQs. Seja V um vetor de números reais, indexado de 1 até n. Uma Range Minimum(Maximum) Query (RMQ) sobre V, com intervalo $[i, j]$, retorna o índice do menor(maior) elemento em $V[i, \ldots, j]$. Esse tipo de consulta é bem estudado pela comunidade de geometria computacional [1, 6]. De fato, existem estruturas de dados de tamanho $O(n)$ que respondem consultas RMQs em tempo constante.

Uma observação crucial é que as consultas $AllPairs(t, d)$ e $Beginnings(t, d)$ podem ser modeladas por múltiplas consultas RMQ (detalhes na próxima seção). De forma que, usando diretamente estruturas de RMQ, consegue-se responder cada consulta em tempo $O(n + k)$, onde k é o tamanho da resposta. Na verdade, $Beginnings(t, d)$ pode ser respondida com

[1] O $+1$ é para tratar o caso onde $k = 0$.

a mesma complexidade sem nenhum pré-processamento, como será explicado em uma seção posterior. A principal desvantagem dessa abordagem é que, independente do tamanho da saída, ambas as consultas são sempre respondidas em tempo $\Omega(n)$.

Quando a série de entrada em que estamos interessados é longa, pode ser mais interessante, do ponto de vista computacional, pré-processar a série em uma estrutura de dados e então responder às consultas. Contudo, desenvolver estruturas eficientes e compactas para nossas consultas não é uma tarefa fácil. A ideia da abordagem proposta aqui é armazenar um conjunto de *pares especiais* de índices e não todos os pares possíveis. Nós dizemos que um par de índices (i, j) é *especial* em relação à série temporal A se as seguintes condições valem:

- $i < j$ e,
- $a_i < \min\{a_{i+1}, \ldots, a_j\}$ e,
- $a_j > \max\{a_i, \ldots, a_{j-1}\}$

Mais ainda, se um evento-(t, d) é também um par especial, dizemos que é um *evento-(t, d) especial*.

Seja S o conjunto de pares especiais da série temporal A. Nossa primeira contribuição nessa linha é uma estrutura de dados de tamanho $O(|S|)$ que responde às consultas $AllPairs(t, d)$ em tempo $O(k + 1)$, onde k é o tamanho do conjunto solução. Além disso, para consultas $Beginnings(t, d)$, propomos uma estrutura de tamanho $O(|S| \log |S|)$ que apresenta tempo de consulta $O(f \times (\log f + \log t) + k)$, onde f é a quantidade de índices distintos que são fins de eventos-(t, d) especiais e k é a quantidade de inícios que são respostas à consulta $Beginnings(t, d)$.

Um possível ponto negativo em manter os pares especiais é que a estrutura pode ter espaço quadrático no pior caso. Isso acontece, por exemplo, se a série temporal for uma sequência crescente. Contudo, argumentamos que isso não é provável de ocorrer. Primeiro, provamos que o valor esperado da cardinalidade do conjunto de pares especiais, para uma permutação aleatória de $\{1, \ldots, n\}$, é $O(n)$ e, depois, que essa cardinalidade tem probabilidade baixa de assumir um valor muito maior que n. Esse resultado é interessante pois o número de pares especiais de uma série temporal com n elementos distintos é igual ao número de pares especiais de uma permutação de $\{1, \ldots, n\}$, na qual a série pode ser facilmente mapeada. Além disso, avaliamos a quantidade de pares especiais de 96 séries temporais que consistem de preços de ações, medidas a cada minuto, durante três anos, do mercado financeiro de ações

brasileiro. Nessas séries, o número médio de pares especiais é apenas 2.78 vezes maior que o tamanho da série correspondente.

1.4 Organização da Dissertação

No Capítulo 2 discutimos trabalhos relacionados e apresentamos conceitos importantes para o prosseguimento da dissertação, como as estruturas de dados RMQ e Range Tree. Em seguida, no Capítulo 3, apresentamos observações úteis para tratar as consultas $AllPairs(t, d)$ e $Beginnings(t, d)$. Nos Capítulos 4 e 5 apresentamos, respectivamente, uma série de algoritmos/estruturas para as consultas $AllPairs(t, d)$ e $Beginnings(t, d)$. Avaliamos a eficiência prática dos algoritmos no Capítulo 6. Finalmente, no Capítulo 7 conclusões são apresentadas assim como perspectivas de trabalhos futuros.

2
Trabalhos Relacionados

Na última década houve uma quantidade significativa de esforços para desenvolver técnicas de mineração de dados em séries temporais, como discutido em um recente *survey* por Fu[7]. Um importante subcampo de pesquisa apontado por esse survey questiona como identificar um dado padrão em séries temporais[10, 3, 13, 9, 11]. Apesar do problema estudado nesse trabalho ser relacionado a essa linha de pesquisa, nossas consultas não se enquadram no arcabouço proposto em [3].

Nosso trabalho também é relacionado à linha de pesquisa que tenta identificar padrões em séries financeiras[11, 9]. Padrões como *heads&shoulders*, triângulos, bandeiras, entre outros, são formas que supostamente ajudam a prever o comportamento de preços de ações. Aqui focamos em um padrão específico que ajuda no estudo de eventos raros.

A definição de pares especiais aparece de forma independente em [10] sobre o nome de *upward extended legs*. Nesse trabalho Pratt e Fink buscam comprimir e armazenar séries temporais além de permitir busca por padrões sobre as mesmas. O arcabouço usado por eles na compressão das séries consiste em encontrar pontos chamados de *important points* e armazená-los como representação da série. Dada um série $A = (a_1, a_2, \ldots, a_n)$, um ponto a_m é um *important minimum* se existem índices i e j, onde $i < m < j$, tais que:

- a_m é o menor entre $a_i, \ldots, a_j$
- $\frac{a_i}{a_m} \geq R$ e $\frac{a_j}{a_m} \geq R$

Na definição, R é um parâmetro que controla a taxa de compressão. Para buscar um padrão as séries comprimidas são indexadas por *features*, de forma que a busca é feita primeiro calculando-se as features do padrão, e, em seguida, achando as séries com features similares às do padrão, e, finalmente, fazendo uma comparação fina dentro de cada série. Uma das três features usadas é o *upward extended legs*, que representa justamente o conjunto de pares especiais na série. Apesar de usar a mesma definição, a aplicação estudada em [10]

não é diretamente relacionada à nossa e, além disso, não se aprofundam nas propriedades dos *upward extended legs* da mesma forma que fazemos aqui.

Finalmente, nosso problema tem alguma semelhança com um problema clássico em geometria computacional - *fixed radius near neighbors*. Dado um conjunto de n pontos em um espaço Euclidiano de m dimensões e uma distância d, o problema consiste em reportar todos os pares de pontos com distância de até d. Bentley et al.[2] apresentaram um algoritmo que reporta todos os k pontos que satisfazem essa propriedade em $O(3^m \times n + k)$. A mesma abordagem pode ser aplicada se um conjunto $\{d_1, d_2, \ldots, d_m\}$ é dado, um para cada dimensão. Contudo, não é claro para nós se ideias similares podem ser aplicadas para gerar todos os pares com distância *não* menor que d ou, como no caso do nosso problema, não maior que d_1 na primeira dimensão e não menor que d_2 na segunda.

2.1 RMQ

Como já citado, o problema Range Mininum (Maximum) Query (RMQ) consiste em pré-processar um dado vetor de entrada de forma que a posição do mínimo (máximo) entre dois índices possa ser obtida de forma eficiente. O primeiro algoritmo de pré-processamento linear e tempo de consulta constante para o problema RMQ foi construído por Harel e Tarjan[8]. O algoritmo, apesar de um grande passo do ponto de vista teórico, era difícil de implementar. Depois desse artigo alguns outros foram publicados propondo ideias mais simples, porém ainda complexas do ponto de vista de implementação. O trabalho de Bender et al. [1] apresentou o primeiro algoritmo realmente simples para RMQ. O algoritmo é baseado na conexão entre RMQ e o problema de Longest Common Ancestor (LCA), e tem pré-processamento linear e tempo de consulta constante além de ser relativamente simples de implementar. O trabalho de Fischer e Heun[6] melhora o algoritmo em Bender et al. reduzindo constantes na memória e tempo.

Detalharemos aqui os dois algoritmos de RMQ que serão usados durante este trabalho. O primeiro, que iremos chamar de RMQSt, cria uma estrutura de tamanho $O(n \log n)$ e responde às consultas em tempo constante. Já o segundo, que chamaremos de RMQBucket, ocupa espaço linear e responde às consultas em tempo $O(\log n)$. Para facilitar um pouco a notação, será discutido como construir uma estrutura de *Range Minimum Query* que retorna o valor do menor elemento, e não o seu índice. Os passos para retornar o índice, ou mesmo fazer uma *Range Maximum Query* são facilmente derivados do exposto. Ainda

no universo de notação, abreviaremos *Range Minimum Query* por RM(in)Q e, de forma análoga, *Range Maximum Query* por RM(ax)Q.

A RMQSt faz uso de uma função $f(i,k)$ que retorna o valor do menor elemento no intervalo $A[i,\ldots,i+2^k]$, para todos i e k aplicáveis. Com essa função em mãos, as consultas RMQ(i,j) podem ser respondidas comparando-se os valores retornados por $f(i,k)$ e $f(j-2^k,k)$, da seguinte forma: RMQ$(i,j) = \min\{f(i,k), f(j-2^k,k)\}$, onde $k = \lfloor \log_2(j-i+1) \rfloor$. Note que a função f pode ser definida recursivamente como segue:

$$f(i,k) = \min\{f(i,k-1), f(i+2^{k-1},k-1)\}$$

Nessa linha, o pré-processamento dessa estrutura consiste em usar uma estratégia de *programação dinâmica* para montar a tabela com os valores de f usando a definição recursiva dada. A consulta RMQ(i,j) é respondida comparando-se os valores adequados de f como descrito acima. Note que usando uma estratégia de programação dinâmica para calcular os valores de f teremos uma tabela de tamanho $\sum_{i=1}^{n} \log(n-i) \leq n \log n$. Como cada célula $f(i,k)$ é calculada em tempo constante, a complexidade do tempo/espaço de pré-processamento é $O(n \log n)$ e, finalmente, a consulta claramente tem complexidade $O(1)$.

No pré-processamento da segunda RMQBucket usada divide-se o vetor de entrada em blocos de tamanho $\frac{\log n}{2}$ e, em tempo linear, calcula-se o menor de cada bloco e o armazena em um vetor B, de forma que $B[i]$ contém o menor elemento no i-ésimo bloco. Em seguida o vetor B é pré-processado com o algoritmo anterior, RMQSt. Já na consulta, a RMQ criada a partir do vetor B é usada para responder consultas "entre blocos". De forma mais clara, dada uma consulta RMQ(i,j) descobrimos através de uma conta simples qual é o primeiro bloco b_i e último bloco b_j dentro do intervalo $[i,j]$. Usamos a RMQ criada sobre B para conseguir consultar RMQ(b_i,b_j) e achar o menor entre os blocos. Finalmente, se i e j não se encaixam perfeitamente dentro de um início e final de bloco, respectivamente, percorremos linearmente os valores entre i e o começo do bloco b_i e entre o final do bloco b_j e j tentando atualizar o valor do mínimo encontrado. Note que o tempo gasto iterando não é maior que $\log n$, uma vez que o tamanho de bloco é $\frac{\log n}{2}$. Juntando isso e o fato de que o menor entre os blocos pode ser achado em tempo constante, chegamos à complexidade de consulta de $O(\log n)$. Finalmente, basta mostrarmos que o pré-processamento tem complexidade linear, que é fácil, uma vez que a principal estrutura criada é uma RMQSt sobre um vetor de tamanho $\frac{2n}{\log n}$ o que garante a complexidade de $O(\frac{2n}{\log n} \times \log(\frac{2n}{\log n})) = O(n)$

Um trabalho importante a ser lembrado aqui é o de Johannes Fischer e Volker Heun[6], aonde eles apresentam uma estrutura bastante elegante e eficiente para o problema RMQ. A estrutura é uma evolução da estrutura em [1] e apresenta tanto tempo/espaço de pré-processamento linear quanto tempo de consulta constante, além de ser razoavelmente simples de implementar. Não iremos entrar em detalhes do artigo aqui, porém, um ponto de interesse é que na seção experimental eles comparam quatro estruturas, a saber: a RMQBucket com blocos de tamanho $\frac{\log n}{2}$, a RMQBucket com blocos de tamanho $\frac{\log n}{4}$, a estrutura proposta por eles e a proposta em Bender et al.[6]. Os resultados deles mostram que para aplicações práticas com vetores de tamanho até 10^8 o RMQBucket, como apresentado aqui, é o melhor do ponto de vista de tempo de consulta e de pré-processamento [1]. Assim, nossa escolha de usar o RMQBucket entre os algoritmos de pré-processamento linear durante os experimentos é justificada.

2.2 Range Tree

Uma estrutura Range Tree é uma árvore enraizada que contém uma lista de pontos como suas folhas. Essa estrutura foi introduzida por Jon Louis Bentley em 1979[2] e é normalmente usada para duas ou mais dimensões, porém os conceitos que precisamos nesse trabalho são relacionados à versão de uma dimensão.

Uma Range Tree 1D é uma árvore binária balanceada feita sobre um conjunto de pontos, onde os pontos são armazenados nas folhas da árvore. Uma vez construída, a Range Tree permite que todos os pontos dentro de um intervalo $[i, j]$ sejam reportados em tempo pseudolinear. Por exemplo, se temos um conjunto inicial de pontos $A = 3, 19, 10, 23, 30, 59, 62, 37, 49, 70, 80, 89, 93, 97$, como no exemplo da Figura 2.1, poderíamos perguntar quais estão no intervalo $[25, 90]$ e a resposta seria o conjunto $\{30, 37, 49, 59, 62, 70, 80, 89\}$. Uma Range Tree 1D sobre um conjunto de n pontos pode ser criada em tempo $O(n \log n)$ e responde a consultas em tempo $O(\log n + k)$, onde k é a quantidade de pontos reportados.

A construção da Range Tree 1D para um conjunto de pontos P de n elementos começa ordenando os pontos em $P = \{p_1 \leq p_2 \leq \ldots \leq p_n\}$ de forma não-decrescente e, depois, usa o Procedimento 2.2 passando os parâmetros $[1, n]$ para de fato construir a árvore.

[1] Por limitações de hardware eles não conseguiram testar com valores maiores

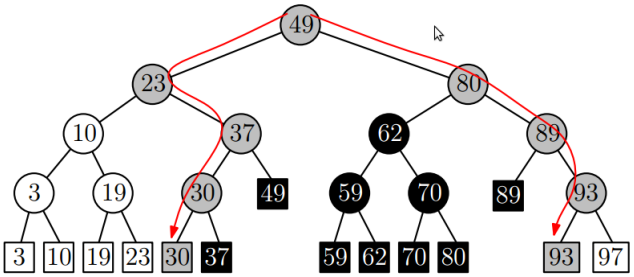

Figura 2.1: Uma Range Tree para o conjunto de $n = 12$ pontos. Figura retirada de [4].

```
ConstructRangeTree(P, low,high)
    If low = high then
        Crie um nó f (folha) com valor P[low]
        Return (P[low], f)
    Crie um nó v associado ao intervalo [low, high]
    mid ← (low + high)/2
    (valor_l, T_l) ←ConstructRangeTree(P, low, mid)
    (valor_r, T_r) ←ConstructRangeTree(P, mid + 1, high)
    Faça T_l ser o filho à esquerda de v
    Faça T_r ser o filho à direita de v
    valor_v ← valor_l
    Salve valor_v em v
    Return (valor_v, v)
```

Figura 2.2: Procedimento para criação de uma Range Tree

A construção toma tempo $O(n \log n)$ porque é necessário ordenar os pontos P. Contudo, repare que essa é a mesma árvore de chamadas recursivas feita pelo algoritmo de ordenação Merge Sort e não é difícil provar que o espaço total ocupado pela mesma é $O(n)$.

Para reportar os pontos dentro de um dado intervalo $[i, j]$ procuramos i e j na Range Tree usando os valores armazenados dentro dos nós internos como guia para a busca. Vamos nomear os caminhos de nós consultados na busca por i e j de, respectivamente, p_i e p_j. Na Figura 2.1 temos $[i, j] = [25, 90]$ e os caminhos p_i e p_j estão destacados em vermelho. Repare que tanto p_i quanto p_j tem tamanho $O(\log n)$, uma vez que a árvore é balanceada. Para reportamos todos os nós entre $[i, j]$ precisamos de um terceiro nó dá árvore, v_{split}, que é o último vértice que os caminhos p_i e p_j têm em comum. No exemplo da Figura 2.1 os dois caminhos divergem na raiz, então, nesse caso, v_{split} é a raiz da árvore.

Seguindo, para todo nó v no caminho de v_{split} (exclusive) até o final de p_i verificamos se o valor dentro dele é maior que i, e, se for, reportamos todos os

pontos nas folhas da subárvore à direita de v. Se v for uma folha, verificamos se v está no intervalo $[i, j]$ antes de reportar. De forma similar, reportamos todas as folhas nas subárvores à esquerda dos nós de v_{split} (exclusive) até o fim de p_j que têm valores menores que j. Desta forma, a complexidade do tempo de consulta é $O(\log n + k)$, já que os caminhos p_i e p_j tem tamanho $O(\log n)$ e reportar as folhas dentro das subárvore custa $O(k)$.

Um fato importante que será usado posteriormente é relacionado aos nós em preto na figura. Esses nós formam uma floresta de subárvores que contém, nas suas folhas, os pontos desejados. Por outro lado, essas subárvores particionam naturalmente o intervalo buscado $[25, 90]$ em um conjunto de $O(\log n)$ intervalos, a saber, $\{[25, 37], [38, 49], [50, 80], [81, 89], [90, 90]\}$. Esse particionamento será usado posteriormente para garantir a complexidade teórica de uma de nossas estruturas.

3
Conceitos Básicos

Nessa seção apresentamos observações/resultados que são úteis para tratar tanto a consulta $AllPairs(t, d)$ quanto a $Beginnings(t, d)$.

Nossa primeira observação é que é simples encontrar todos os eventos-(t, d) que possuem um dado índice i como início. Para isto, é necessário ter uma estrutura de dados que suporte consultas RM(ax)Q sobre A em tempo constante e que tenha tempo/espaço de pré-processamento linear. Como discutido, existem na literatura estruturas de RMQ com essas características. Usando essa estrutura de RMQ o procedimento `GenEventStart` na Figura 3, se chamado com parâmetros $(i, i + 1, i + t)$, acha todos os eventos-(t, d) com início i. De formal geral esse procedimento, se chamado com os parâmetros $(i, low, high)$, gera todos os eventos-(t, d) com início i e final no intervalo $[low, high]$. Basicamente, ele usa a estrutura de RMQ para achar o índice de maior valor em $A[low, \ldots, high]$. Se o desvio $(a_j - a_i)$ é menor que d então a busca termina, pois claramente não há evento-(t, d) que satisfaz à consulta dentro do intervalo corrente. Caso contrário, o procedimento reporta o par (i, j) e explora recursivamente os intervalos $[low, j - 1]$ e $[j + 1, high]$.

Proposição 3.0.1 *Seja k_i a quantidade de eventos-(t, d) que têm início i e fim no intervalo $[low, high]$. O procedimento* `GenEventsStart`$(i, low, high)$ *gera todos estes k_i eventos em tempo $O(k_i + 1)$.*

Prova. A corretude é trivial. O tempo de execução é proporcional ao número de chamadas recursivas executadas. O fluxo de execução pode ser visto como uma árvore binária onde cada nó corresponde a uma chamada recursiva. Note

```
GenEventsStart(i,low,high)
    If high ≥ low then
        j ← RMQ(low, high)
        If a_j − a_i ≥ d then
            Adiciona o par (i, j) à lista de eventos-(t, d)
            GenEventsStart(i,low,j-1)
            GenEventsStart(i, j+1,high)
```

que nessa árvore cada nó interno corresponde a um novo evento-(t, d). Como a quantidade de nós em uma árvore binária é no máximo duas vezes a quantidade de nós internos, o número de chamadas recursivas é no máximo $2k_i$. ■

Vamos considerar também a existência de um procedimento análogo, chamado `GenEventsEnd`, que recebe como entrada uma tupla $(j, low, high)$ e usa uma RM(in)Q para gerar todos os eventos-(t, d) que têm j como final e o início no intervalo $[low, high]$. A implementação do mesmo é análoga ao `GenEventStart` e o tempo de execução é similar.

O seguinte Lema 3.0.2 é fundamental para esse trabalho uma vez que motiva o nosso foco nos pares especias como forma de responder às consultas $AllPairs(t, d)$ e $Beginnings(t, d)$.

Lema 3.0.2 *Seja (i, j) um evento-(t, d). Então, as seguintes condições valem:*

1. *Existe um índice $i*$ tal que $(i*, j)$ é um evento-(t, d) e $i*$ é o início de um evento-(t, d) especial.*

2. *Existe um índice $j*$ tal que $(i, j*)$ é um evento-(t, d) e $j*$ é o final de um evento-(t, d) especial.*

Prova. Iremos apenas provar (i), já que a prova para a segunda condição é análoga. Seja $i*$ o índice de tempo do elemento de A com menor a_i* entre os índices no intervalo $\{i, \ldots, j-1\}$. Em caso de empates, considere o maior índice. Como $a_i* \leq a_i$, $i \leq i* < j$ e (i, j) é um evento-(t, d), temos que $(i*, j)$ é também um evento-(t, d). Seja $k*$ o índice do elemento de A com maior valor entre aqueles no conjunto $\{i*+1, \ldots, j\}$. O par $(i*, k*)$ é tanto um par especial quanto um evento-(t, d), que conclui a demonstração. ■

Na próxima seção mostramos como gerar os pares especiais de forma eficiente.

3.1 Gerando Pares Especiais

O conjunto S de pares especiais de uma série temporal A, de tamanho n, pode ser obtido em tempo $O(n + |S|)$ executando-se o pseudocódigo na Figura 3.1 com parâmetros $(1, n)$. Primeiro, o procedimento faz uma consulta à estrutura de RM(in)Q para calcular em tempo constante o índice do menor elemento em $A[low, \ldots, high]$ e outra consulta para calcular o índice j do maior elemento em $A[i+1, \ldots, high]$. O loop **While** encontra todos os pares especiais que têm como início o índice i. Finalmente, o procedimento explora recursivamente os intervalos $[low, i-1]$ e $[i+1, high]$.

```
GenSpecialPairs(low,high)
    If high ≥ low
        i ← RM(in)Q(low, high)
        j ← RM(ax)Q(i + 1, high)
        While i < j e a_j − a_i ≥ d
            Adicione o par (i, j) à lista de pares especiais
            j ← RM(ax)Q(i, j − 1)
        GenSpecialPairs(low, i − 1)
        GenSpecialPairs(i + 1, high)
```

Figura 3.1: Procedimento para gerar todos os pares especias

Para entender a corretude do procedimento basta notar que se i for o índice do menor elemento do intervalo $[low, high]$ então, pela definição de pares especiais, é impossível existir um par especial (x, y) tal que $low \leq x < i$ e $i < y \leq high$.

Por fim, note que o loop **While** só itera no máximo uma vez a mais que a quantidade de pares especiais com início i e, a cada chamada recursiva, reduzimos o tamanho do intervalo $[low, high]$ em pelo menos uma unidade. Assim, o tempo de execução é $O(n + |S|)$.

A Proposição 3.1.1 resume o discutido nessa seção.

Proposição 3.1.1 *Seja S o conjunto de pares especiais de uma série A de tamanho n. Então, S pode ser construído em tempo $O(|S| + n)$.*

3.2 Quantidade Pares Especiais

Como o tempo e espaço de pré-processamento de algumas das nossas estruturas de dados irão depender do número de pares especiais, essa seção discute alguns resultados teóricos e práticos relacionados a esse número.

Claramente, o número de pares especiais de uma série temporal é pelo menos 0 e no máximo $n(n-1)/2$, onde o limite inferior(superior) é atingido para uma sequência decrescente(crescente). A Proposição 3.2.1 mostra que o valor esperado do número de pares especiais para uma permutação de $1, \ldots, n$ escolhida de forma uniformemente aleatória é $n - H_n$[1].

Esse resultado é interessante posto que a quantidade de pares especiais em uma série de n elementos distintos é igual ao de uma permutação em que a série pode ser naturalmente mapeada. Para ficar claro o mapeamento de uma série arbitrária em uma permutação, tome a série $A = (2, 10, 7, 15)$. Podemos mapeá-la na permutação $A^p = (1, 3, 2, 4)$ que contém os mesmos

[1] H_n é a soma dos n primeiros termos da série harmônica

pares especiais. De forma geral, dada uma série temporal $A = (a_1, \ldots, a_n)$, cria-se uma permutação A^p com os mesmos pares especiais em dois passos. Ordena-se a série A resolvendo empates no valor em ordem decrescente de índice. Seja $A^{ord} = (a_{i_1}, a_{i_2}, \ldots, a_{i_n})$ a série ordenada. A permutação A^p é construída através da associação $A^p[i] = Ord(A[i])$, onde $Ord(A[i])$ é o índice do elemento $A[i]$ em A^{ord}.

Proposição 3.2.1 *Seja S o conjunto de pares especiais de uma série temporal escolhida de forma uniformemente aleatória entre as permutações de $1, \ldots, n$. Então, o valor esperado de $|S|$ é $n - H_n$, onde H_n é o n-ésimo termo da série harmônica.*

Prova. Seja $E[X]$ o valor esperado da cardinalidade de S. Além disso, seja $X_{i,j}$ uma variável aleatória indicadora que tem valor 1 se o par de índices (i, j) pertence a S e 0, caso contrário.

Pelas definições anteriores,

$$E[X] = E[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j}].$$

Pela linearidade do valor esperado, temos que:

$$E[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{i,j}].$$

Além disso, a igualdade $E[X_{i,j}] = \frac{1}{(j-i+1)(j-i)} = \frac{1}{j-i} - \frac{1}{j-i+1}$ vale. Para entender o porquê, basta ver que o menor elemento do intervalo $[i, j]$ tem probabilidade de $\frac{1}{(j-i+1)}$ de ser o elemento de índice i. E, dado que o menor é o elemento de índice i, há a probabilidade de $\frac{1}{(j-i)}$ que o maior seja o de índice j. Sendo assim, temos:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left(\frac{1}{j-i} - \frac{1}{j-i+1} \right) = \sum_{i=1}^{n-1} \left(1 - \frac{1}{n-i+1} \right) = n - \sum_{i=1}^{n} \frac{1}{i} = n - H_n.$$

■

A seguinte proposição, cuja a prova deixamos para o Apêndice, mostra que a probabilidade da cardinalidade de S ser muito maior que n é baixa.

Proposição 3.2.2 *Seja $c > 0$ um número real, então $Pr\{X \geq c.n\} \leq \frac{c' \log n}{c^2 n}$, onde c' é uma constante positiva fixada.*

4
Estruturas para a consulta $AllPairs$

Nessa seção analisamos estruturas para o problema $AllPairs(t, d)$. Primeiro mostramos um algoritmo simples baseado fortemente no conceito de RMQ. Esse algoritmo apresenta tempo de pré-processamento linear e consulta $O(n + k)$, onde n é o tamanho da série e k é o tamanho da saída. Em seguida mostramos uma estrutura de tamanho proporcional a quantidade de pares especiais e com tempo de consulta $O(k + 1)$.

4.1
AllPairs-RMQ

Como discutido anteriormente, se criamos uma estrutura de RM(in)Q para a série de entrada, é possível executar o procedimento `GenEventsStart` com parâmetros $(i, i + 1, i + t)$ para gerar todas as soluções de uma consulta (t, d) que tem como início o elemento i da série. Usando isso, a estratégia consiste em iterar em todos os possíveis inícios i e gerar as soluções para cada início. O conjunto final de soluções é a união dos pares gerados.

Note que todo pré-processamento necessário consiste em criar a estrutura de RMQ e o tempo de execução é $O(n + k)$, onde k é o tamanho da solução. Um ponto negativo dessa estratégia é que o tempo de resposta às consultas sempre depende do tamanho da série n, o que pode ser indesejável para séries muito grandes. Na próxima seção apresentamos uma estrutura baseada na ideia de pares especiais que busca melhor equilíbrio entre tempo/espaço de pré-processamento e consulta.

4.2
AllPairs-SP

Pré-processamento. Primeiro, geramos o conjunto S de pares especiais da série de entrada $A = (a_1, \ldots, a_n)$, o que pode ser feito através do Procedimento 3.1. A partir de S criamos um vetor V que contém os pares em S ordenados de forma crescente de valor-Δ, empates na ordenação são decididos arbitrariamente. Além disso, precisamos de um vetor M onde a posição

i guarda o maior índice de um par em V com valor-Δ menor ou igual a i. Repare que é possível construir tanto o vetor V quanto o vetor M, a partir de S, em tempo $O(|S|)$. Finalmente, criamos três estruturas de RMQ, a primeira, nomeada $\mathcal{D}$, é criada sobre o vetor V e tem tamanho $O(|S|)$[1], utilizamos-a para responder à seguinte consulta: Dados dois valores t^* e d^*, reporte o subconjunto de pares especiais de S com valor-Δ de até t^* e desvio de ao menos d^*. Ou seja, para encontrar os eventos-(t^*, d^*) especiais. Já as outras duas, $\mathcal{D}_{min}$ e $\mathcal{D}_{max}$ têm tamanho $O(n)$ e são usadas para podermos chamar, respectivamente, os procedimentos `GenEventsEnd` e `GenEventsStart` sobre a série A.

Abaixo detalhamos o conjunto de estruturas criadas juntamente com um exemplo onde $A = (3, 4, 5, 1)$.

1. Conjunto S de pares especiais, gerado em tempo $O(|S| + n)$. Para série A do exemplo, tem-se $S = \{(1, 2), (2, 3), (1, 3)\}$.

2. Vetor V com os pares em S ordenados por valor-Δ. Como os valores-Δ estão no intervalo $[1, n - 1]$, podemos facilmente criar o vetor V usando Count Sort em tempo $O(|V| + n)$. Na primeira contamos quantos pares em S tem valor-Δ de t, para todo $t \in [1, n]$ e armazenamos essa contagem em $C[t]$. Na segunda usamos os valores em C para inserir os pares em S diretamente na posição certa em V. No exemplo, $V = [(1, 2), (2, 3), (1, 3)]$.

3. Vetor M, de tamanho n, que pode ser facilmente criado a partir de V em tempo $O(|V| + n) = O(|S| + n)$. No exemplo, $M[1] = 2$ e $M[2] = 3$.

4. Estrutura de RMQ $\mathcal{D}$, de tamanho $O(|S|)$, criada a partir do vetor V.

5. Duas estruturas de RMQ, $\mathcal{D}_{min}$ e $\mathcal{D}_{max}$, de tamanho $O(n)$, que são criadas a partir da série original A.

Desta forma, o tempo e espaço de pré-processamento da estrutura proposta têm complexidade $O(|S| + n)$.

Consulta. Para responder à consulta $AllPairs(t, d)$ executamos duas fases. Na primeira, encontramos todos os eventos-(t, d) especiais de V usando a RMQ $\mathcal{D}$ e o vetor M. Então, na segunda fase, usamos esses eventos-(t, d) especiais e as estruturas $\mathcal{D}_{min}$ e $\mathcal{D}_{max}$ para encontrar os outros eventos-(t, d). As duas fases são detalhadas abaixo.

[1]Note que estamos assumindo na análise uma estrutura de RMQ com tempo/espaço de pré-processamento linear e consulta constante.

```
GenSpecialEvents(low, high, t, d)
    i ← D(low, high)
    If Desvio(V[i]) < d then
        Return
    Adicione o par V[i] a lista L
    GenSpecialEvents(low, i − 1, t, d)
    GenSpecialEvents(high, i + 1, t, d)
```

Figura 4.1: Gera todos os eventos-(t, d) especias (usa a RMQ $\mathcal{D}$)

Fase 1. Seja k^* o índice do último par especial em V entre aqueles com valor-Δ de até t. Usamos $\mathcal{D}$ para encontrar os eventos-(t, d) especiais em $V[1, \ldots, k^*]$. O índice k^* pode ser descoberto através do vetor M de forma direita, apenas acessando a posição $M[t]$. Seja i o índice de V retornado por uma consulta de máximo em V entre os índices $[1, k^*]$. Se o desvio do par armazenado em $V[i]$ é menor que d podemos terminar a busca, já que nenhum outro par em $V[1, \ldots, k^*]$ terá desvio de ao menos d. Caso contrário, adicionamos o par $V[i]$ à lista $\mathcal{L}$ e prosseguimos recursivamente nos subvetores $V[1, \ldots, i-1]$ e $V[i+1, \ldots, k^*]$. No final a lista $\mathcal{L}$ terá todos os eventos-(t, d) especiais. Na Figura 4.1 temos o pseudocódigo dessa fase. Note que a chamada inicial do procedimento deverá ser `GenSpecialEvents`$(1, M[t], t, d)$.

Fase 2. Esta fase pode ser dividida em duas etapas.

Fase 2.1. Nessa etapa, geramos o conjunto de inícios distintos de todos os eventos-(t, d). Primeiro, geramos a lista $\mathcal{E}_L$ de fins de eventos-(t, d) especiais em $\mathcal{L}$, o que pode ser feito em $O(|\mathcal{L}|)$ mantendo-se um vetor de 0-1 para evitar repetições de fins[2]. Então, chamamos o procedimento `GenEventsEnd`$(j, j-t, j-1)$ para todo $j \in \mathcal{E}_L$. É importante lembrar que `GenEventsEnd` utiliza a estrutura $\mathcal{D}_{min}$. Durante o processamento do `GenEventsEnd` armazenamos todos os inícios de eventos-(t, d) encontrados em uma lista $\mathcal{B}$. Novamente, utilizamos um vetor 0-1 para evitar repetições entre inícios.

Fase 2.2 Nessa etapa, geramos todos os eventos-(t, d) que têm inícios armazenados na lista $\mathcal{B}$. Para isto, chamamos o procedimento `GenEventsStart`$(i, i+1, i+t)$ para todo $i \in \mathcal{B}$. Novamente, é importante lembrar que o esse procedimento utiliza a estrutura $\mathcal{D}_{max}$. Finalmente, a lista

[2]Em detalhes, podemos criar, durante o pré-processamento, um vetor H de tamanho n onde inicialmente todas as posições têm valor 0. Então, para todo fim $f \in \mathcal{L}$ verificamos em $H[f]$ se ele já foi visto ou não. Se não foi, marcamos $H[f]$ com 1 e guardamos f em $\mathcal{E}_L$. Finalmente, ao final do processo, desmarcamos todos os índices 1 em H usando a lista $\mathcal{E}_L$, garantido assim que H poderá ser usado em uma nova consulta.

de pares retornada pelo procedimento `GenEventsStart` apresenta todos os eventos-(t, d) que estamos procurando.

A discussão dessa seção é sumarizada pelo teorema abaixo.

Teorema 4.2.1 *O procedimento de duas fases descrito anteriormente gera todas as k soluções para uma consulta $AllPairs(t, d)$ em tempo $O(k + 1)$ e com complexidade $O(|S| + n)$ de espaço e tempo de pré-processamento.*

Prova. No final da Fase 1, temos todos os eventos-(t, d) especias, isto é feito em tempo $O(k+1)$. Novamente, para justificar essa complexidade, podemos ver o fluxo de execução como uma árvore binária onde cada nó interno corresponde a um evento-(t, d) especial e as folhas a custo extra.

No começo da Fase 2.1, obtemos o conjunto $E_{\mathcal{L}}$ contendo os fins distintos dos eventos-(t, d) especias. Decorre diretamente que a complexidade de tempo dessa fase é $O(k+1)$. Pelo item (ii) do Lema 3.0.2 o conjunto $\mathcal{B}$ contendo todos os inícios de eventos-(t, d) pode ser gerado a partir dos fins em $E_{\mathcal{L}}$. Obtemos o conjunto $\mathcal{B}$ em tempo $O(k)$ chamando o `GenEventsEnd`$(j, j - t, j - 1)$ para cada $j \in E_{\mathcal{L}}$. Finalmente, gastamos mais $O(k)$ para gerar todos os eventos-(t, d) chamando `GenEventsStart`$(i, i + 1, i + t)$ para cada índice $i \in \mathcal{B}$. ■

5
Estruturas para a consulta *Beginnings*

Nessa seção exploramos estratégias para o problema $Beginnings(t, d)$. Primeiro apresentamos dois algoritmos com tempo de consulta $O(n)$, um sem pré-processamento e outro com. Em seguida, mostramos uma solução trivial para o problema porém com tempo de consulta $O(nt)$ e, logo depois, mostramos um algoritmo que se baseia no conceito de pares especiais e tem tempo de consulta $O(k + 1)$. Finalmente, construímos um algoritmo para *Beginnings* que utiliza o algoritmo AllPairs-SP da seção anterior.

5.1
Beg-Quick

Esse algoritmo percorre a série A verificando, para cada índice de tempo i, se i é o início de um evento-(t, d). Para conseguir isso usamos uma fila Q de tamanho $t + 2$ que guarda índices de tempo não dominados. Dizemos que um índice j é *dominado* por um índice de tempo k se e somente se $k > j$ e $a_k \geq a_j$. O algoritmo mantém a seguinte invariante: antes de testar se $i-1$ é o início de um evento-(t, d), a fila Q contém todos os índices do conjunto $\{i, \ldots, (i-1)+t\}$ que não são dominados por outro índice no mesmo conjunto. Além disso, a fila encontra-se naturalmente ordenada de forma crescente por índice de tempo e de forma decrescente por valor (o valor de um índice i na fila é $A[i]$).

Por causa das invariantes, o índice do maior a_j, com $j \in \{i, \ldots, (i-1)+t\}$ está no começo da fila. Iremos denotar por $Q.head$ o índice armazenado no início de Q. Podemos verificar se $i - 1$ pertence ao conjunto solução testando se o desvio do par $(i - 1, Q.head)$ é pelo menos d. Se for, $i - 1$ é adicionado à solução. Para garantir a invariante, em toda iteração, o algoritmo remove i do começo de Q (se ele ainda estiver lá); remove todos os índices dominados por $i + t$ e, finalmente, adiciona $i + t$ ao fim de Q. O algoritmo executa em tempo $O(n)$, já que ele percorre a série A somente uma vez, e, todo índice é adicionado/removido da fila exatamente uma vez. O pseudocódigo do algoritmo é apresentado na Figura 5.1.

For $k = 2, \ldots, t+1$
Percorra a fila Q do seu fim até seu começo removendo todo índice j dominado por k.
Adicione k no final de Q.
If $(1, Q.head)$ um evento-(t, d) **then**
Adicione o índice 1 ao conjunto solução.
For $i = 2, \ldots, n-1$
If $Q.head = i$ **then**
remova i do início de Q.
If $i + t \leq n$ **then**
Percorra a fila Q de seu fim até seu início removendo todo índice dominado por $i + t$.
Adicione $(i + t)$ ao fim de Q.
If $(i, Q.head)$ é um evento-(t, d) **then**
Adicione i ao conjunto solução.

Figura 5.1: Procedimento Beg-Quick

5.2 Algoritmo Beg-RMQ

Esse algoritmo é muito similar ao AllPairs-RMQ. No pré-processamento criamos uma estrutura de RM(ax)Q em tempo $O(n)$. Para responder à consulta *Beginnings* percorremos a série A e, para cada índice i, usamos a estrutura de RMQ para decidir se $\max\{a_j | i < j \leq j + t\}$ é maior ou igual a $a_i + d$. Em caso positivo, adicionamos i à lista de soluções. Note que o Beg-Quick apresentado não precisa de pré-processamento e tem a mesma complexidade teórica de resposta à consulta que o algoritmo Beg-RMQ.

5.3 Algoritmo Beg-Naive

Esse algoritmo não tem fase de pré-processamento. Para responder a uma dada consulta simplesmente percorremos a série A e, para cada possível início i, verificamos no intervalo $[i + 1, i + t]$ se existe um j tal que $a_j - a_i \geq d$. O tempo de consulta desse algoritmo é $O(nt)$.

5.4 Algoritmo Beg-SP

Aqui exploramos um algoritmo baseado no conceito de pares especias para o problema $Beginnings(t, d)$.

Para explicar essa estrutura precisamos introduzir a seguinte notação: $S_{[x,y]}$ é o conjunto dos pares especiais cujo valor-Δ está dentro do intervalo $[x, y]$, e $L_{[x,y]}$ é a lista obtida a partir de $S_{[x,y]}$ ordenando os pares de $S_{[x,y]}$ em

ordem decrescente de desvio e removendo aqueles cujo o final já aparece na lista.

Pré-processamento. Primeiro geramos o conjunto de pares especias S. Em seguida, construímos uma Range Tree onde cada nó é associado a um dado subintervalo de $[1, n-1]$. Como visto na Seção 2.2, a raiz de $\mathcal{T}$ é associada ao intervalo $[1, n-1]$. Se um nó v é associado ao intervalo $[x, \ldots, y]$ então seu filho à esquerda é associado ao intervalo $[x, \lfloor \frac{(x+y)}{2} \rfloor]$ e seu filho à direita associado a $[\lfloor \frac{(x+y)}{2} \rfloor + 1, y]$. Além disso, cada nó de $\mathcal{T}$ aponta para sua lista $L_{[x,y]}$ correspondente.

O conteúdo das listas é gerado da seguinte forma. Inicialmente, todas as listas $L_{[x,y]}$ estão vazias. Então, percorremos o conjunto S em ordem não-decrescente de desvio e, se um par especial s tem valor-Δ de t, adicionamos s ao final de todas as listas $L_{[x,y]}$ tais que $x \leq t \leq y$. No final, cada lista $L_{[x,y]}$ estará ordenada em ordem não-decrescente de desvio dos pares especiais. Finalmente, para cada lista $L_{[x,y]}$ percorremos seus pares especiais removendo aqueles que seu fim já apareceu anteriormente na lista, por exemplo, se uma dada lista $L_{[x,y]}$ é $[(1,4),(2,4),(1,2),(3,4)]$ os pares removidos serão $(2,4)$ e $(3,4)$ e a lista final será $[(1,4),(1,2)]$.

Assim, a árvore $\mathcal{T}$ é uma Range Tree sobre os pontos $[1, \ldots, n]$ estendida de forma que cada nó tem um ponteiro adicional para uma lista $L_{[x,y]}$. A Figura 5.2 mostra um exemplo com uma série de 7 elementos. Abaixo temos os passos apresentados de forma resumida.

1. Gerar o conjunto de pares especiais S.
2. Criar uma Range Tree com ponteiros adicionais para as listas $L_{[x,y]}$.
3. Popular as listas $L_{[x,y]}$ em ordem não-decrescente dos desvios dos pares especiais.
4. Remover pares especiais com fins duplicados dentro de cada lista $L_{[x,y]}$.

O tamanho da estrutura é $O(\min\{n^2, |S| \log n\})$ porque cada par especial pode aparecer em até $\log n$ listas $L_{[x,y]}$ e o tamanho de cada lista é no máximo n, uma vez que cada fim aparece no máximo uma vez.

Consulta. Para responder a consulta $Beginnings(t, d)$ o procedimento executa duas fases. Na primeira, ele obtém todos os fins de eventos-(t, d) especiais os armazena em uma lista E_f. Na segunda fase, usa os fins em E_f para obter os inícios dos eventos-(t, d).

A corretude desse procedimento é baseada no item (ii) do Lema 3.0.2, uma vez que se i é um início de um evento-(t, d) então existe um fim $j*$ de

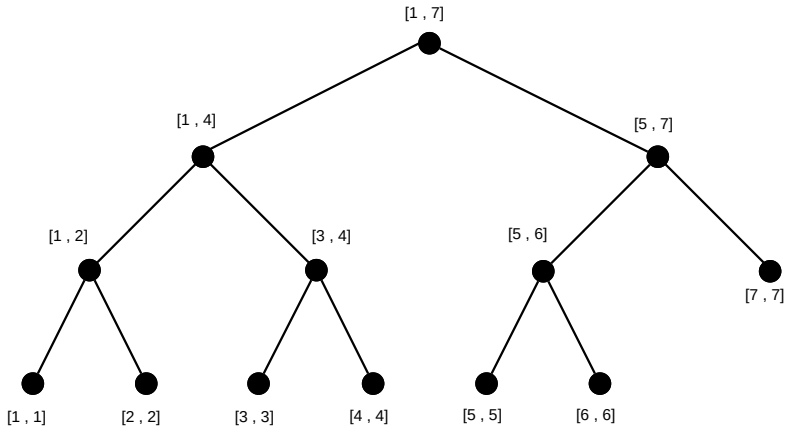

Figura 5.2: Uma árvore binária ordenada para $n = 7$. A raiz de $\mathcal{T}$ possui um ponteiro para a lista $L_{[1,7]}$ enquanto seus filhos à esquerda e a direita, respectivamente, têm ponteiros para $L_{[1,4]}$ e $L_{[5,7]}$

um evento-(t, d) especial tal que $(i, j*)$ é um evento-(t, d). E, como será visto, o fim $j*$ é achado na Fase 1 e usado para achar i na Fase 2.

Fase 1. Primeiramente, achamos o conjunto K de nós em $\mathcal{T}$ cujos intervalos associados formam uma partição do intervalo $[1, t]$. No exemplo da Figura 5.2, para $t = 6$, K consiste de três nós: o nó interno associado ao intervalo $[1, 4]$, e as folhas associadas a $[5, 5]$ e $[6, 6]$. Relembre que em uma Range Tree se tomarmos as raízes das subárvores à esquerda dos nós no caminho de busca de t achamos um conjunto K em tempo $O(\log n)$ que tem até $\log t$ nós.

Prosseguindo, inicialmente há uma lista de fins E_f vazia. Então, para cada nó $v \in K$, procedemos como segue: percorremos a lista de pares especiais associada a v; se o par especial atual tem desvio maior ou igual a d, verificamos se seu final já foi adicionado a lista de fins E_f. Se não foi, adicionamos o final a E_f; caso contrário, descartamos o par e seguimos em frente. O percurso termina quando encontramos um par especial com desvio menor que d ou chegamos ao final da lista. Se usarmos um vetor 0-1 para marcar os fins que já apareceram em E_f esse passo custará no máximo $O(|E_f| \log t)$, pois cada final aparece no máximo em $\log t$ listas. Na Figura 5.4 temos o pseudocódigo de como gerar E_f a partir de K, como descrito.

Fase 2. Nessa fase usamos a lista de fins E_f criada na etapa anterior para gerar os inícios dos eventos-(t, d). O processo pode ser dividido em três etapas. A primeira consiste em ordenar a lista de fins E_f de forma eficiente, já a segunda calcula a função *clp* (será definida posteriormente) usada na terceira etapa para finalmente encontrar os inícios. Para entender o porquê dessa fase,

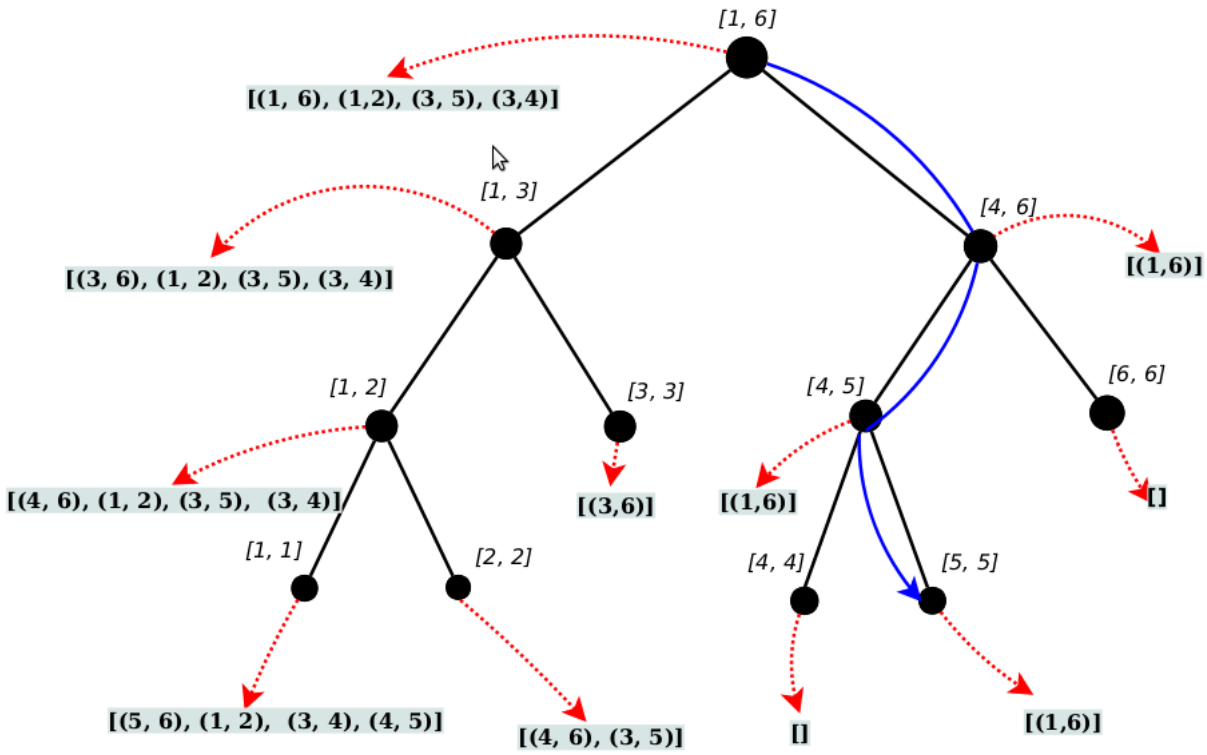

Figura 5.3: Exemplo de árvore Range Tree para consulta *Beginnings*

For $v \in K$
    $L_{[x,y]} \leftarrow$ Lista L associada ao nó v
    For $(i, j) \in L_{[x,y]}$
        If $a_j - a_i < d$ **then** pare;
        else If $j \notin E_f$ **then**
            Adicione j em E_f
    End For
End For

Figura 5.4: Procedimento para gerar a lista E_f a partir de K

de maneira geral, ser complicada vamos considerar uma solução simples para encontrar os inícios a partir E_f. Poderíamos, por exemplo, executar o algoritmo `GenEventsEnd` para cada fim $j \in E_f$ e usar um vetor 0-1 para evitar repetições dos inícios vistos. O problema dessa abordagem é que sua complexidade de tempo é $O(k|E_f|)$, pois um mesmo início pode ser encontrado por diferentes fins em E_f. De forma concreta, se tivermos a série $A = (1, 2, \ldots, n)$ e uma consulta $(t, d) = (n, 1)$, a lista E_f será $[2, 3, \ldots, n]$ e reportaremos $n-1$ inícios, já que todo início nessa lista é início de um evento-(t, d). Assim, durante as chamadas ao `GenEventsEnd`, cada início i seria encontrado por todo os fins no intervalo $[i + 1, n]$ e o tempo gasto na consulta, para este caso, seria $(n-1) + (n-2) + \ldots + 1 = O(n^2)$. Sendo assim, boa parte da dificuldade nos passos abaixo se deve a criação de uma solução sensível ao tamanho da saída, evitando fatores multiplicativos na mesma.

Fase 2.1. Nessa etapa, ordenamos os fins em E_f em tempo $O(\min\{n, |E_f| \log |E_f|\})$ usando ou um **bucket sort** ou um **heapsort** dependendo se $|E_f|$ é maior que $n/\log n$ ou não. Seja $e_1 < e_2 < \ldots < e_f$ os fins em E_f ordenados.

Fase 2.2. Primeiro, defina, para um fim $e_j \in E_f$, a função $clp(e_j) = e_{i*}$, onde $i* = \max\{i | e_i \in E_f \text{ e } e_i < e_j \text{ e } a_{e_i} \geq a_{e_j}\}$. De outra forma, $clp(e_j)$ retorna, dentre os fins de índice menor e valor maior que e_j em A, aquele de maior índice em E_f . Para garantir que todo $clp(e_j)$, para $j = 1, \ldots, |E_f|$, é válido nós assumimos que E_f contém um elemento artificial e_0 tal que $e_0 = 0$ e $a_{e_0} = \infty$. Por exemplo, seja $E_5 = \{e_1, e_2, e_3, e_4, e_5\}$ uma lista ordenada de fins, onde $(e_1, e_2, e_3, e_4, e_5) = (2, 3, 5, 8, 11)$ e $(a_{e_1}, a_{e_2}, a_{e_3}, a_{e_4}, a_{e_5}) = (8, 5, 7, 6, 4)$. Temos que $clp(e_4) = e_3$ e $clp(e_3) = e_1$. Os *clp*'s podem ser calculados em $O(|E_f|)$ como será descrito posteriormente.

Fase 2.3. Para gerar os inícios de eventos-(t, d) a partir dos fins em E_f, o procedimento na Figura 5.5 é executado.

O procedimento itera sobre todos os fins em E_f em ordem reversa e guarda na variável $CurrentHigh$ o menor índice j para qual todos os inícios maiores que j já foram gerados. Essa variável é inicializada com o valor $e_f - 1$ pois não há início maior que $e_f - 1$. Para cada fim e_i o procedimento procura por inícios no intervalo $[max\{e_i - t, clp(e_i)\}, CurrentHigh]$. Note que ele não procura inícios no intervalo $[e_i - t, clp(e_i)]$ uma vez que todo início nesse intervalo que pode ser gerado por e_i pode também ser gerado por $clp(e_i)$ (o contrário não é verdade). De fato, esta é a razão para termos calculado os *clp*'s - eles evitam que um início seja gerado mais de uma vez. Seguindo, a variável $CurrentHigh$ é atualizada e uma nova iteração começa. Essa etapa pode ser

implementada em tempo $O(|E_f|+k)$, onde k é a quantidade de inícios gerados.

Abaixo um sumário dos passos e um exemplo para a série $A = (2, 10, 3, 5, 6, 15)$ e $(t, d) = (5, 7)$. Na Figura 5.3 temos a Range Tree criada na fase de pré-processamento. A resposta esperada para essa consulta é $\{1, 3, 4, 5\}$

1. O conjunto de S de pares especiais é $\{(1,2),(1,6),(3,4),(3,5),(3,6),(4,5),(4,6),(5,6)\}$. Os desvios desses pares, são, respectivamente, $\{8, 13, 2, 3, 12, 1, 10, 9\}$.

2. Através da Range Tree encontra-se o conjunto $K = \{[1,3],[4,4],[5,5]\}$ em tempo $O(\log n)$. Na Figura, o caminho em azul representa a busca na Range Tree pelo valor $t = 5$.

3. As listas associadas ao conjunto K são $L_{[1,3]} = [(3,6),(1,2),(3,5),(3,4)]$, $L_{[4,4]} = []$ e $L_{[5,5]} = [(1,6)]$, note que elas estão ordenadas por desvio e pares com fins repetidos foram removidos. O conjunto E_f, com os fins distintos dos eventos-(t,d) especiais, é criado em tempo $O(|E_f| \log t)$. Nesse caso, $E_f = \{6, 2, 5, 4\}$.

4. E_f é ordenado em tempo $O(\min\{n, |E_f| \log |E_f|\})$. Depois da ordenação, $E_f = [2, 4, 5, 6]$.

5. Os *clp*s de todos os fins $e \in E_f$ são calculados em tempo $O(|E_f|)$. Para isso adicionamos um elemento extra na lista ficando com $E_f = [0, 2, 4, 5, 6]$ e $(a_0, a_2, a_4, a_5, a_6) = (\infty, 10, 5, 6, 15)$. Lembrando que $A = (2, 10, 3, 5, 6, 15)$. Logo, os *clp*s são:

   - $clp(2) = 0$
   - $clp(4) = 2$
   - $clp(5) = 2$
   - $clp(6) = 0$

6. Finalmente, os inícios são gerados chamando-se o `GenEventsEnd` evitando repetições de um mesmo início através dos *clp*s. Essa última etapa toma tempo $O(|E_f| + k)$. No exemplo, o Procedimento 5.5 executa apenas uma iteração. Pois, no começo, $CurrentHigh = 5$ e achamos os inícios dentro do intervalo $[\max\{6-5, 0\}, 5] = [1, 5]$ que formam fim com $a_6 = 15$, ou seja, o conjunto $\{1, 3, 5, 6\}$. Em seguida, o atualizamos $CurrentHigh$ para 1 e então o procedimento acaba, pois já geramos todos os inícios possíveis.

   A discussão pode ser resumida no seguinte teorema.

CurrentHigh $\leftarrow e_f - 1$
For $i = f$ até 1.
    `GenEventsEnd` $(e_i, \max\{e_i - t, clp(e_i)\}, CurrentHigh)$
    CurrentHigh $\leftarrow \max\{e_i - t, clp(e_i)\}$
    If $CurrentHigh = 1$ **then** Break;
End For

Figura 5.5: Procedimento para gerar inícios

Teorema 5.4.1 *Existe uma estrutura de dados de tamanho* $O(|S| \log |S|)$ *que suporta a pesquisa* $Beginnings(t, d)$ *em tempo* $O(\log n + f.(\log f + \log t) + k)$*, onde* k *é o tamanho do conjunto solução e* f *é o número de fins distintos de eventos-*(t, d) *especiais.*

É importante ressaltar que, com algum esforço, podemos reduzir o fator $\log f$ na complexidade acima. A ideia principal é criar uma estrutura de RMQ para cada lista $L_{[x,y]}$ e ordenar pares especiais em $L_{[x,y]}$ pelo seus fins e não por seu desvio. Desta forma, conseguimos trocar a ordenação na Fase 2.1 por uma operação mais rápida de Merge.

5.4.1 Calculando os clps

A Figura 5.6 mostra o pseudocódigo do procedimento para calcular *clp*'s em tempo $O(|E_f|)$. Note que E_f deve está ordenado de forma crescente de índices de tempo. O procedimento percorre os fins na lista E_f em ordem reversa. Ele mantém uma fila Q que armazena no começo da i-ésima iteração os fins em E_f com índices de tempo maiores que e_i e cujos os *clp*'s ainda não foram determinados.

A fila mantém a seguinte invariante: se um fim e foi inserido antes de e' em Q então $e > e'$ e $a_e > a_{e'}$. Na i-ésima iteração do loop externo o procedimento acha todos os finais em Q que tem *clp* igual a e_i. Para isso, percorre todos os fins em Q começando pela sua calda e, toda vez que visita um fim e tal que $a_e \leq a_{e_i}$, marca $clp(e) = e_i$ e remove e da fila Q; quando um fim e é tal que $a_e > a_{e_i}$ é encontrado, o percurso é interrompido e o final e_i é inserido na calda de Q.

O loop externo claramente é executado $O(|E_f|)$ vezes. Além disso, cada fim é adicionado à Q no máximo uma vez e a cada iteração do **While** um fim é removido de Q. Sendo assim, o loop **While** também gasta tempo $O(|E_f|)$.

```
Q ← ∅
Adicione e_f no final da fila Q
For i = f − 1 até 0.
        e ← Q.tail
        While a_e < a_{e_i}
                Remova e de Q
                clp(e) ← e_i
                e ← Q.tail
        End While
        Adicione e_i no final da fila Q
End For
```

Figura 5.6: Procedimento para gerar clp

5.5 Algoritmo Beg-Hybrid

Uma outra forma de responder à consulta $Beginnings(t, d)$ é combinar partes dos algoritmos AllPairs-SP e o Beg-SP em um novo algoritmo como segue.

Na criação da estrutura, fazemos o pré-processamento do AllPairs-SP sobre a série, porém sem a construção da RMQ $\mathcal{D}_{max}$. Para responder uma dada consulta, primeiro executamos a Fase 1 do AllPairs-SP para encontrar a lista $\mathcal{L}$ que contém todos os eventos-(t, d) especiais. Em seguida, percorremos essa lista e criamos a lista E_f de fins distintos de eventos-(t, d) especiais. Por fim, executamos a Fase 2 do Beg-SP, que parte da lista E_f e, finalmente, acha todos os ínicios dos eventos-(t, d).

A principal motivação dessa estratégia, quando comparada com o Beg-SP, é a economia do fator $\log n$ no espaço de pré-processamento. Outra vantagem é o uso da mesma estrutura do AllPairs-SP de forma que ambas consultas, $AllPairs(t, d)$ e $Beginnings(t, d)$, podem ser respondidas com a mesma estrutura de dados. A desvantagem, contudo, é que essa estrutura tem tempo de consulta $O(k \min\{t, f\})$ ao invés do $O(\log n + f.(\log f + \log t) + k)$ da Beg-SP, onde k é a cardinalidade do conjunto solução e f é o número de fins distintos de eventos-(t, d) especias.

A corretude dessa estratégia segue diretamente da corretude das duas estruturas usadas.

6
Avaliação Experimental

Nesse capítulo buscamos avaliar as estruturas propostas sobre o enfoque experimental. Primeiro descrevemos as séries usadas durante os experimentos e suas características, depois o ambiente usado e, em seguida, apresentamos os tempos de pré-processamento e consulta para as nossas implementações de RMQ. Finalmente, discutimos a eficiência das estruturas propostas, tanto do ponto de vista de pré-processamento quanto de consulta, para os problemas $AllPairs$ e $Beginnings$.

6.1
Corpus

Nosso corpus é formado de 48 séries, cada série representa o preço, de minuto a minuto, de uma dada ação do mercado financeiro brasileiro durante o período de cerca de três anos. O tamanho médio de série no corpus é 195112.50, a menor série tem tamanho 10000 e a maior 229875.

6.2
Quantidade de Pares Especiais nas Séries Financeiras

Para estimar a quantidade de pares especiais, usamos as 48 séries originais e também as séries invertidas delas, que servem para modelar o comportamento de pesquisas onde o desvio é negativo. Por exemplo, uma consulta AllPairs$(2, -3)$ em uma série $A = (4, 1, 5, 2)$ é equivalente a uma pesquisa AllPairs$(2, 3)$ na série $A^{rev} = (2, 5, 1, 4)$, o mesmo vale para consulta $Beginnings$.

A razão entre o número de pares especiais e o tamanho da série, nas 96 séries, originais mais invertidas, fica sempre entre $[0.54, 6.99]$, com valor médio de 2.78. Já nas 48 originais a razão fica entre $[0.78, 6.99]$, com valor médio de 3.42. Esses dados sugerem que estruturas de dados que dependem da quantidade de pares especiais na série devem ser práticos. De qualquer forma, é sempre possível usar uma estratégia onde calcula-se a quantidade de

Algoritmo	Menor tempo	Tempo médio	Maior tempo
RMQBucket	1.36	6.39	8.04
RMQSt	7.07	40.02	50.54

Tabela 6.1: Tempo de pré-processamento das RMQs

pares especiais na série de entrada e, se esse número passar de um certo limite, troca-se a estrutura de dados para uma que não dependa dessa variável.

6.3 Ambiente Experimental

A implementação de todas as estratégias foi feita em `C++`, o compilador utilizado foi o `g++ 4.4.3`, com flag de otimização `-O2`. O computador utilizado para os testes tem `3GB` de RAM e processador Intel(R) Core(TM)2 Duo CPU, T6600 @ 2.20GHz de 64 bits. O sistema operacional utilizado foi o Ubuntu 10.04 LTS - Lucid Lynx. Todos os tempos foram medidos em milissegundos.

6.4 RMQs

Como discutido na Seção2.1, utilizamos duas estruturas distintas de RMQ, a saber, RMQBucket e RMQSt. É interessante comparar o tempo de pré-processamento e consulta dessas duas implementações, já que elas são usadas extensivamente ao longo dos próximos experimentos. Sendo assim, na Tabela 6.1, mostramos estatísticas sobre o tempo levado para pré-processar cada uma das 48 séries do corpus, os tempos, em milissegundos, são a média de 100 execuções. Como previsto pela análise teórica, a RMQBucket tem menor tempo de pré-processamento.

Já na Figura 6.1 mostramos os tempos para consultas de mínimo nas seis primeiras séries. Os intervalos consultados foram geradas de forma aleatória porém foram restritos a ter tamanho de no máximo 100, uma vez que nos experimentos posteriores as consultas de mínimo (máximo) são feitas sobre intervalos pequenos. O tempo reportado para cada consulta é a soma de 10000 repetições da mesma e, para cada série, fizemos 1000 consultas aleatórias, atingindo um total de 6000 consultas.

Note que enquanto a RMQSt manteve um comportamento constante a RMQBucket cresce em intervalos. Isso se deve a complexidade de consulta da RMQBucket que, para intervalos pequenos, dependendo do tamanho de bloco, percorre linearmente o intervalo, que tem tamanho proporcional a $\log n$, para encontrar o mínimo.

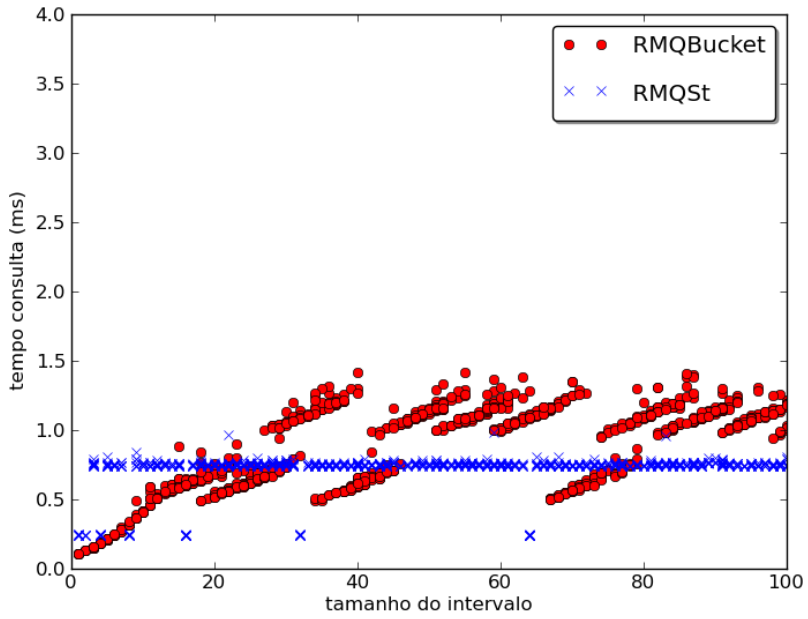

Figura 6.1: Tempos de consulta das duas estruturas de RMQ

6.5 Consulta AllPairs

Nessa seção analisamos o pré-processamento e consulta das estruturas AllPairs-RMQ e AllPairs-SP. Um ponto importante na comparação entre as duas estruturas é a implementação de RMQ usada pela estratégia. Entre as duas implementações disponíveis, RMQBucket e RMQSt, a primeira se mostrou melhor escolha para a estrutura AllPairs-SP devido ao consumo de $O(n \log n)$ de memória da segunda estrutura. Sendo assim, os experimentos com a AllPairs-SP foram feitos com essa estrutura. Já para a AllPairs-RMQ apresentamos resultados com as duas implementações: Uma que usa a RMQBucket e será chamada de AllPairs-RMQBucket e outra que usa a RMQSt e será chamada de AllPairs-RMQSt.

6.5.1 Pré-processamento

Na Tabela 6.2 estão estatísticas sobre os tempos de pré-processamento de cada estrutura, os tempos são a média de 3 execuções. A estrutura AllPairs-SP apresenta o maior tempo de pré-processamento e a AllPairs-RMQBucket o menor, como esperado pela análise teórica.

Algoritmo	Menor Tempo	Tempo Médio	Maior Tempo
AllPairs-RMQ(RMQBucket)	1.39	7.20	9.65
AllPairs-RMQ(RMQSt)	5.36	37.81	54.08
AllPairs-SP	13.23	98.85	297.22

Tabela 6.2: Tempo de pré-processamento das estruturas para consulta $AllPairs$

6.5.2 Consulta

No conjunto de teste temos 28 séries com tamanho 229875. Para cada uma dessas realizamos 10 consultas, repetidas 5 vezes, para cada um dos quatro algoritmos analisados. Relembrando os algoritmos são AllPairs-RMQBucket, AllPairs-RMQSt, AllPairs-St e NaiveScan. O tempo reportado de cada consulta é a média dos tempos em cada uma das repetições. Nas Figuras 6.2, 6.3 e 6.4 estão comparações entre os diferentes algoritmos. Nesses gráficos o tempo que cada algoritmo levou para responder a consulta está no eixo Y e no eixo X temos a razão entre a quantidade de pares no conjunto solução da consulta e o tamanho da série na qual a consulta foi feita (229875, nesse caso). As consultas foram geradas com t's no intervalo $[3, 16]$ e d's no intervalo $[0.1, 0.2]$. Estes foram escolhidos de forma que os eventos resultantes sejam raros, de fato, note que a maior parte dos pontos nos gráficos tem coordenada X entre 0 e 0.10. Como observação final, a dispersão nos resultados da AllPairs-RMQBucket é provavelmente reflexo do crescimento em blocos da estrutura RMQBucket.

Na Figura 6.2 temos a comparação da estratégia AllPairs-SP com das duas versões do AllPairs-RMQ, claramente a estratégia AllPairs-SP foi mais eficiente. No caso, os tempos da AllPairs-SP foram, em média, 23 vezes menores que o tempo da melhor versão do AllPairs-RMQ. Já no gráfico na Figura 6.3, de forma similar, temos a comparação entre o NaiveScan e as duas versões do AllPairs-RMQ, novamente, a estrutura baseada em RMQ levou mais tempo para responder as consultas. Finalmente, na Figura 6.4 comparamos as duas melhores estratégias, NaiveScan e AllPairs-SP. O AllPairs-SP apresenta melhor tempo quando a saída é pequena porém fica mais lento que o NaiveScan quando o tamanho da saída começa a crescer. É importante notar que o bom desempenho do NaiveScan nesses experimentos se deve unicamente ao fato de que os valores de t escolhidos foram muito pequenos. Para deixar claro, na Tabela 6.3 mostramos consultas com t's entre 100 e 400 feitas sobre uma das séries de maior tamanho. É claro nessa tabela que o NaiveScan rapidamente se torna a pior estratégia. Note que t's grandes são realísticos, uma vez que um

valor de t de 400, nas nossas séries, corresponde a não mais que uma janela de tempo de 7 horas dentro de uma série de 3 anos.

Apesar do custo maior de pré-processamento, como visto na seção anterior, a estratégia baseada em pares especiais apresenta melhor desempenho em consulta. Dessa forma, para o cenário comum onde a entrada é pré-processada apenas uma vez, e, a partir da estrutura respondemos uma longa sequência de pesquisas, acreditamos que essa abordagem seja a melhor escolha.

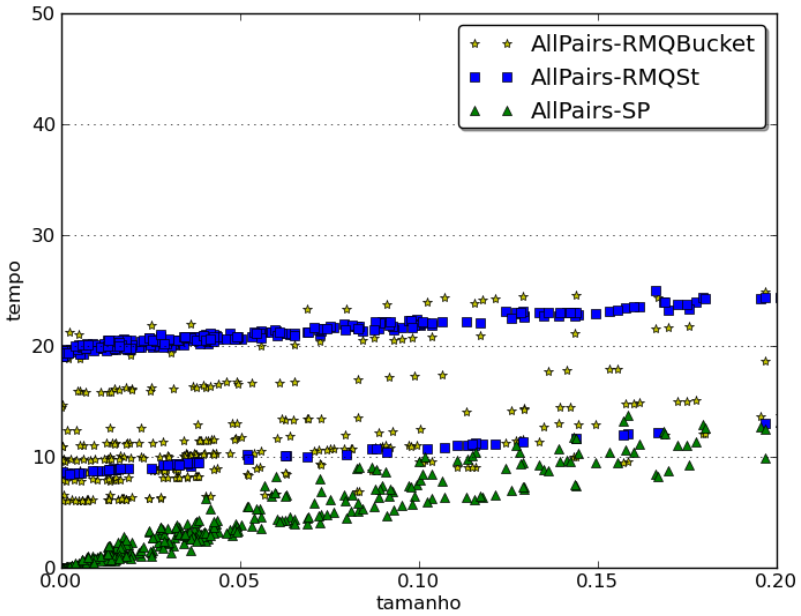

Figura 6.2: Comparação entre as estratégias AllPairs-RMQ e AllPairs-SP

6.6 Consulta Beginnings

Nessa seção analisamos os tempos das estruturas para a consulta *Beginnings*. Relembrando, são quatro algoritmos, a saber, Beg-RMQ, Beg-Hybrid, Beg-SP e Beg-Quick. Novamente o Beg-RMQ será divido em duas versões, Beg-RMQBucket e Beg-RMQSt. Para as outras duas estruturas que usam RMQ, Beg-SP e Beg-Hybrid, a implementação usada será a RMQBucket.

6.6.1 Pré-processamento

Na Tabela 6.4 estão estatísticas sobre o pré-processamento de cada estrutura, os números foram feitos a partir da média de 3 execuções. O tempo

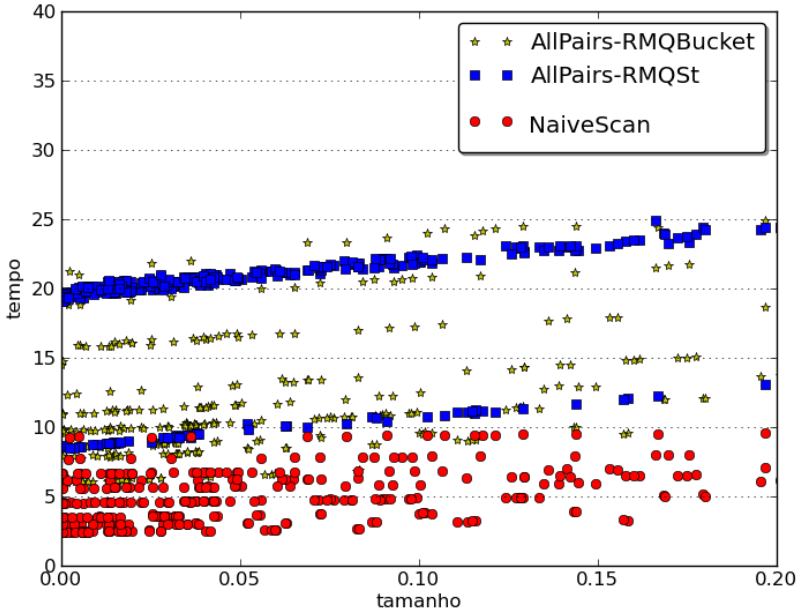

Figura 6.3: Comparação entre as estratégias AllPairs-RMQ e NaiveScan

t	d	saída	AllPairs-RMQBucket	AllPairs-RMQSt	AllPairs-SP	NaiveScan
100.0	0.5	48665	39.56	23.45	7.39	45.45
100.0	0.8	10165	36.27	20.13	1.34	45.3
100.0	1.0	886	35.58	19.37	0.15	45.26
200.0	0.5	344574	61.05	47.19	44.97	88.02
200.0	0.8	67078	39.32	24.69	8.95	87.48
200.0	1.0	7011	34.83	19.81	1.19	87.49
400.0	0.8	647017	83.09	71.01	77.31	172.19
400.0	1.0	142065	45.16	30.71	23.63	171.76
400.0	1.2	16956	35.6	20.63	2.41	171.56

Tabela 6.3: Tempos de consultas com t's maiores

de pré-processamento da estrutura Beg-RMQ é o menor, independente da estrutura de RMQ usada.

6.6.2 Consulta

As consultas reportadas aqui foram geradas com t's entre $[4, 46]$ e d's entre $[0.1, 1.5]$. Novamente, a escolha dos pares (t, d) foi feita de forma que os eventos reportados sejam raros. Os tempos foram medidos da mesma forma, isto é, cada consulta foi repetida 5 vezes e o tempo reportado é média das repetições. As consultas foram feitas sobre as 28 maiores séries do conjunto, de tamanho 229875. Para cada série foram feitas 15 consultas.

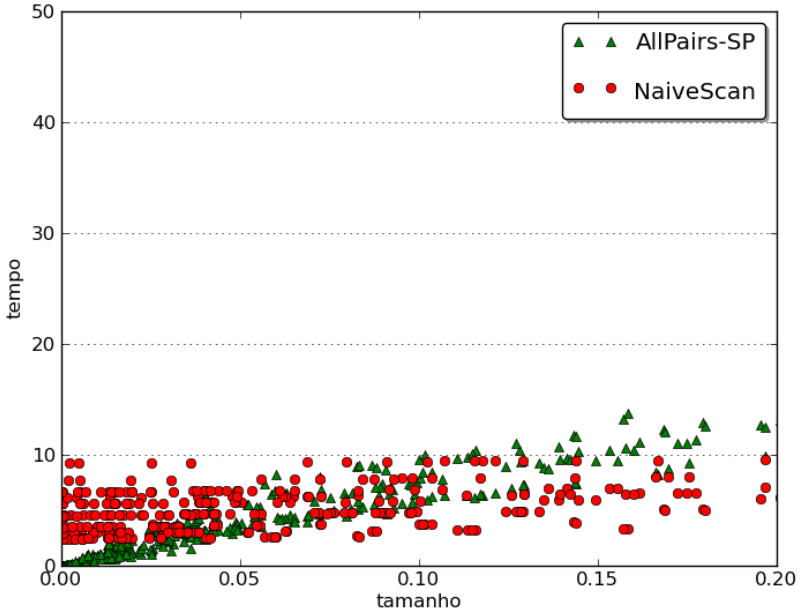

Figura 6.4: Comparação entre as estratégias AllPairs-SP e NaiveScan

Algoritmo	**Menor Tempo**	**Tempo Médio**	**Maior Tempo**
Beg-RMQBucket	1.23	7.22	10.12
Beg-RMQSt	6.00	37.36	55.12
Beg-Hybrid	15.00	122.67	302.63
Beg-SP	104.53	674.85	1601.11
Beg-Quick	N/A	N/A	N/A

Tabela 6.4: Tempo de pré-processamento das estruturas para consulta *Beginnings*

As Figuras 6.5 e 6.6 mostram que a estrutura Beg-RMQ, independente da implementação usada, é amplamente dominada pelo algoritmo Beg-Quick e Beg-SP. De fato, a Beg-SP nos experimentos foi na média 300 vezes mais rápida que a Beg-RMQSt e 448 vezes mais rápida que a Beg-RMQBucket. Sendo assim, não mostramos os resultados dessa estratégia em gráficos posteriores, também não incluímos os resultados do NaiveScan, pois esse foi inferior as estratégias Beg-SP, Beg-Hybrid e Beg-Quick, além de sofrer do mesmo problema apresentado na seção experimental do $AllPairs$, isto é, degrada consideravelmente para t's um pouco maiores.

Na Figura 6.7, temos os tempos das três estratégias com melhores resultados. A estrutura Beg-SP apresenta os melhores tempos quando a saída é até 10% do tamanho da série, seguida pela Beg-Quick e, finalmente, Beg-Hybrid. Em detalhes, a Beg-SP foi na média 14 vezes mais rápida que a Beg-

Hybrid e cerca de 80 vezes mais rápida que a Beg-Quick.

Temos aqui um cenário semelhante ao da outra consulta, uma vez que a estrutura baseada em pares especias apresenta maior tempo de pré-processamento porém ótimo tempo de consulta.

Uma parâmetro interessante de se estimar nos experimentos é o fator f na complexidade do Beg-SP. Relembrando, o tempo de consulta dessa estrutura é $O(\log n + f.(\log f + \log t) + k)$, onde k é o tamanho do conjunto solução e f é o número de fins distintos de eventos-(t, d) especiais. Na Figura 6.8 temos a proporção entre f e k nas consultas feitas que geraram pelo menos 10 saídas. No gráfico, a proporção ficou fortemente concentrada entre $[0, 1]$ com maior concentração entre $[0, 0.5]$, o que sugere que, para aplicações reais, o custo esperado por saída do Beg-SP é $O(\log k + \log t)$.

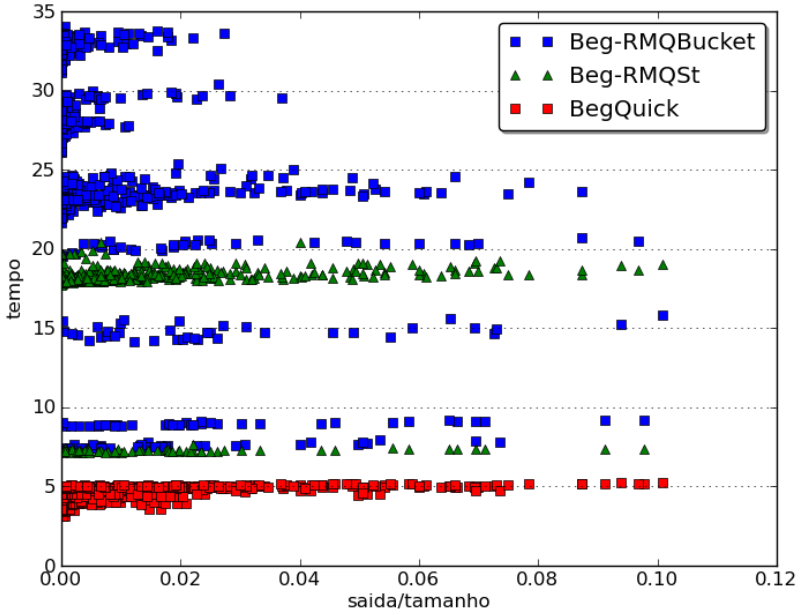

Figura 6.5: Consulta *Beginnings* para estruturas Beg-Quick e Beg-RMQ

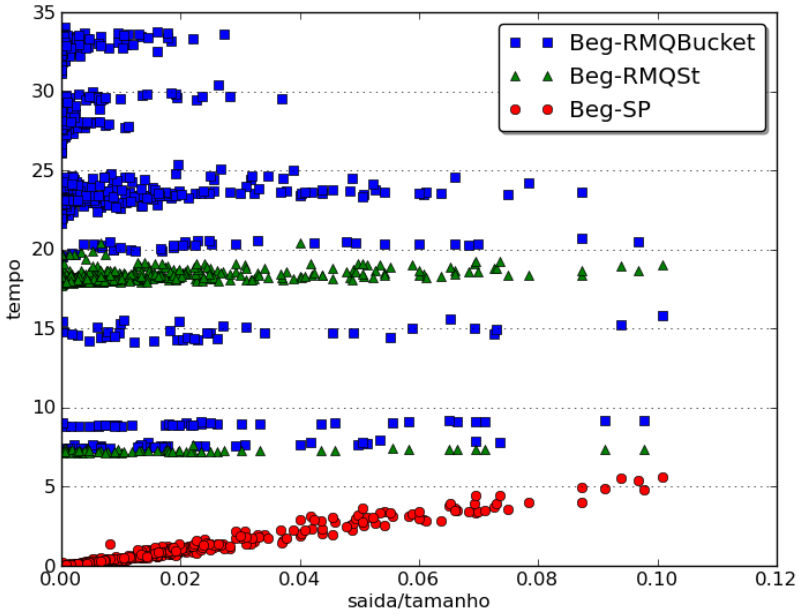

Figura 6.6: Consulta *Beginnings* para estruturas Beg-SP e Beg-RMQ

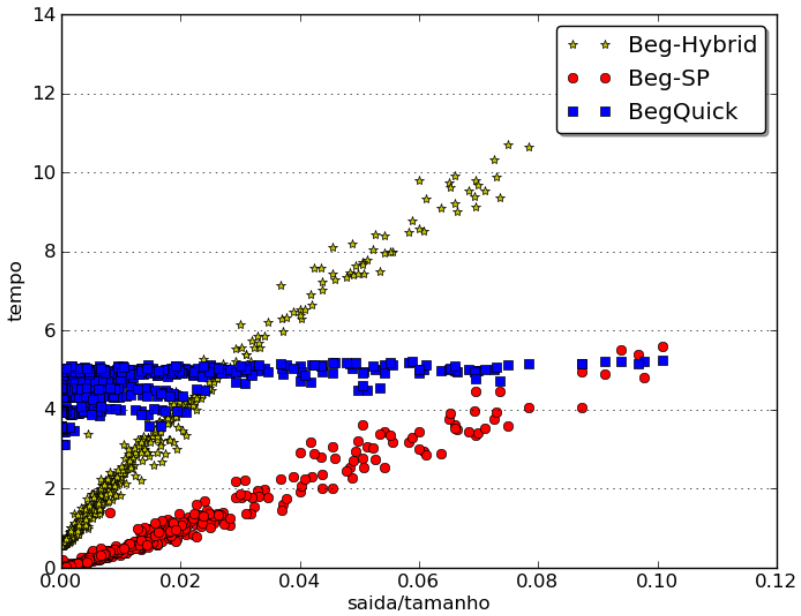

Figura 6.7: Consulta *Beginnings* para estruturas Beg-SP, Beg-Hybrid e Beg-Quick

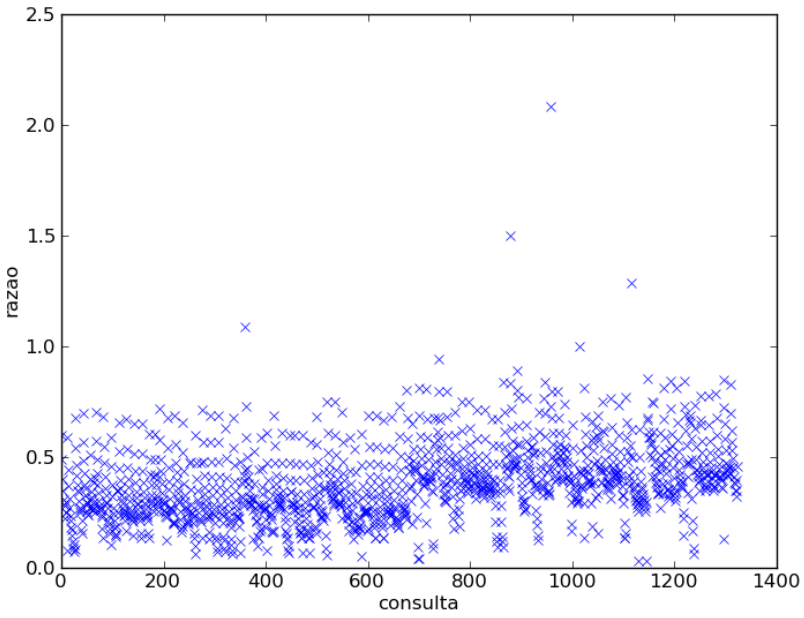

Figura 6.8: Proporção entre o inícios e fins para cada consulta.

7
Conclusão

Séries temporais estão presentes nas mais diversas áreas como medicina, física, mercado financeiro entre outras. Exemplos de séries temporais incluem, por exemplo, os valores de fechamento da Bovespa dia a dia, o volume do fluxo de água do rio amazonas anualmente, a taxa de desemprego de um país tomada ano a ano ou mesmo os valores de uma determinada ação medidos de minuto a minuto.

Ao analisar uma série temporal é comum buscarmos por variações significantes na mesma que ocorrem em um intervalo curto de tempo. Nesse trabalho estudamos formas de detectar eventos raros em séries temporais. Definimos o conceito de um evento-(t, d), que representa um evento de interesse na série parametrizado pelos valores t e d, que marcam, respectivamente, a janela de tempo do evento e a magnitude da variação do mesmo. Em seguida, apresentamos dois tipos de consultas, $AllPairs$ e $Beginnings$, que retornam, respectivamente, todos os eventos-(t, d) e os inícios dos mesmos. Discutimos três aplicações onde essas consultas podem ser úteis.

Para responder à consulta $AllPairs$ mostramos três estratégias: a solução trivial, com complexidade de consulta $O(nt)$, uma solução que cria apenas uma estrutura de RMQ na fase de pré-processamento e responde à consultas em tempo $O(n)$, e uma última, baseada nas propriedades dos pares especiais, que responde à consultas em tempo $O(k + 1)$.

Já para consulta $Beginnings$, mostramos cinco estratégias diferentes: NaiveScan, Beg-RMQ, Beg-Quick, Beg-SP e Beg-Hybrid. A primeira, NaiveScan, é uma solução trivial com complexidade de consulta $O(nt)$, já as duas seguintes apresentam tempo de consulta $O(n)$, porém a Beg-Quick não tem fase de pré-processamento. Já as duas últimas, Beg-SP e Beg-Hybrid, são baseadas no conceito de pares especiais e respondem às consultas, respectivamente, em tempo $O(k + 1)$ e $O(k \min\{t, f\})$.

Discutimos propriedades teóricas do conjunto de pares especiais, mostrando que o valor esperado da sua cardinalidade é $O(n)$, e que temos baixa probabilidade desse valor ser muito maior que $O(n)$. Apresentamos um algo-

ritmo que gera o conjunto dos pares especiais S de uma série com n elementos em tempo $O(|S| + n)$.

Finalmente, avaliamos a eficiência prática de todas as estruturas propostas usando um conjunto de 48 séries do mercado de ações brasileiro. Nos experimentos vimos que as estratégias baseadas em pares especiais, apesar do maior tempo/espaço de pré-processamento, respondem à consultas (t, d) de forma mais eficiente, tanto para consulta $AllPairs$ quanto para a $Beginnings$. Sendo assim, elas se mostram ideais para o cenário onde a série é pré-processada apenas uma vez e depois disso uma longa lista de consultas é realizada sobre a mesma. Em particular, para o cenário onde não há pré-processamento, apresentamos o algoritmo ótimo Beg-Quick para o problema Beginnings.

Em trabalhos futuros gostaríamos estender nossa analise teórica dos pares especiais para séries temporais nas quais temos mais informações sobre a distribuição de probabilidade dos elementos ao longo do tempo, como por exemplo, uma série em que os pontos seguem um processo de Markov. Além disso, existem algumas extensões interessantes das consultas $AllPairs$ e $Beginnings$, como quando restringimos o intervalo da série aonde os eventos-(t, d) podem ocorrer. Por exemplo, para nossas séries do mercado financeiro, com valores de ações medidos de minuto a minuto durante 3 anos, gostaríamos de ter a liberdade, durante a consulta, de especificar faixas de tempo de interesse para os eventos buscados, ex. no segundo ano da série, durante os meses de Abril até Setembro, etc..

A
Prova da Proposição 3.2.2

Prova.

A desigualdade de Chebyshev mostra que se X é uma variável aleatória com valor esperado $E[X]$ e variância $Var[X]$, então $Pr\{|X - E[X]| \geq c\} \leq \frac{Var[X]}{c^2}$, para qualquer constante positiva c dada. Considerando que, por definição, $Var[X] = E[X^2] - E[X]^2$, e multiplicando c por $E[x]$ na desigualdade anterior, nós temos que $Pr\{|X - E[X]| \geq cE[X]\} \leq \frac{E[X^2]-E[X]^2}{c^2 E[X]^2}$.

Para formar a desigualdade de Chebyshev é preciso determinar $E[X]^2$ e $E[X^2]$. Pela proposição 1, segue que $E[X]^2 = (n - H_n)^2$.

Por definição, $E[X^2] = E[(\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j})^2]$. Sendo assim, pela linearidade do valor esperado:

$$E[X^2] = (2\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{k=i+1}^{n-1} \sum_{l=k+1}^{n} E[X_{i,j}X_{k,l}]) + (\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[(X_{i,j})^2])$$

Agora precisamos calcular $E[X^2]$. Isso pode ser feito em seis casos disjuntos.

Em cada um dos casos abaixo nós assumimos que o produto de todo par de variáveis aleatórias $X_{i,j}$ e $X_{k,l}$ é representado pela intersecção de dois intervalos fechados $[i, j]$ e $[k, l]$, onde $[i, j] = \{p | i \leq p \leq j\}$ e $[k, l] = \{q | k \leq q \leq l\}$.

Além disso, com algum abuso de notação, nós assumimos que em cada um dos casos que a expressão $E[X_{i,j}X_{k,l}]$ denota $\sum_i \sum_{j>i} \sum_{k \geq i} \sum_{l>k} E[X_{i,j}X_{k,l}]$, onde $[i, j]$ e $[k, l]$ são pares de intervalos que pertencem ao caso em questão.

Caso 1: Intervalos $[i, j]$ e $[k, l]$ são idênticos.

$E[X_{i,j}X_{k,l}] = E[X_{i,j}X_{i,j}] = E[X_{i,j}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{(j-i+1)(j-i)} = n - H_n$

Caso 2a: Intervalos $[i, j]$ e $[k, l]$ são disjuntos:

$E[X_{i,j}X_{k,l}] = \sum_{i=1}^{n-3} \sum_{j=i+1}^{n-2} \sum_{k=j+1}^{n-1} \sum_{l=k+1}^{n} \frac{1}{(j-i+1)(j-i)(l-k+1)(l-k)} =$

$$\sum_{i=1}^{n-3}\sum_{j=i+1}^{n-2}\left[\frac{1}{(j-i+1)(j-i)}\left(\sum_{k=j+1}^{n-1}\sum_{l=k+1}^{n}\frac{1}{(l-k+1)(l-k)}\right)\right] =$$

$\sum_{i=1}^{n-3}\sum_{j=i+1}^{n-2}\frac{(n-j)-H_{(n-j)}}{(j-i+1)(j-i)} = \alpha(n) - \beta(n) - \gamma(n)$, onde

$\alpha(n) = \sum_{i=1}^{n-3}\sum_{j=i+1}^{n-2}\frac{n}{(j-i+1)(j-i)}$, $\beta(n) = \sum_{i=1}^{n-3}\sum_{j=i+1}^{n-2}\frac{j}{(j-i+1)(j-i)}$ e $\gamma(n) = \sum_{i=1}^{n-3}\sum_{j=i+1}^{n-2}\frac{H_{(n-j)}}{(j-i+1)(j-i)}$

Vamos primeiro calcular $\alpha(n)$:

$\alpha(n) = \sum_{i=1}^{n-3}\sum_{j=i+1}^{n-2}\frac{n}{(j-i+1)(j-i)} = n^2 - 2n - H_{(n-2)}$

Pela definição de $\beta(n)$, segue que:

$$\beta(n) = \sum_{i=1}^{n-3}\sum_{j=i+1}^{n-2}\frac{j}{(j-i+1)(j-i)} = \sum_{i=1}^{n-3}\left(\sum_{j=i+1}^{n-2}\frac{j}{(j-i)} - \frac{j}{(j-i+1)}\right)$$

$$\beta(n) = \sum_{i=1}^{n-3}\left[\left(\frac{i+1}{1} + \frac{i+2}{2} + \ldots + \frac{n-2}{n-i-2}\right) - \left(\frac{i+1}{2} + \frac{i+2}{3} + \ldots + \frac{n-2}{n-i-1}\right)\right]$$

$$\beta(n) = \sum_{i=1}^{n-3}\left[\frac{i+1}{1} + \left(\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n-i-2}\right) - \frac{n-2}{n-i-1}\right] = \sum_{i=1}^{n-3}\left(i + H_{(n-2)} - \frac{n-i-2}{n-i-1}\right)$$

$$\beta(n) = \frac{(n-3)(n-2)}{2} + \sum_{i=1}^{n-3} H_i - \sum_{i=2}^{n-2}\frac{n-2}{i}$$

$\beta(n) = \frac{n^2}{2} - \Theta(n\log n)$.

De forma similar, um limite justo pode ser determinado para $\gamma(n)$:

$\gamma(n) = \sum_{i=1}^{n-3}\sum_{j=i+1}^{n-2}\frac{H_{(n-j)}}{(j-i+1)(j-i)} \leq \sum_{i=1}^{n-3}\sum_{j=i+1}^{n-2}\frac{H_n}{(j-i+1)(j-i)} \leq n.H(n) = O(n\log n)$

$\gamma(n) = \sum_{i=1}^{n-3}\sum_{j=i+1}^{n-2}\frac{H_{(n-j)}}{(j-i+1)(j-i)} \geq \sum_{i=1}^{\frac{n-2}{2}-1}\sum_{j=i+1}^{\frac{n-2}{2}}\frac{H_{(n/2)}}{(j-i+1)(j-i)} \geq H(n/2).\left[\frac{n-2}{2} - H_{(n-2)/2}\right] = \Omega(n\log n)$

Assim, $\gamma(n) = \Theta(n\log n)$.

De volta a equação original:

$\sum_{i=1}^{n-3}\sum_{j=i+1}^{n-2}\frac{(n-j)-H_{(n-j)}}{(j-i+1)(j-i)} = \alpha(n) - \beta(n) - \gamma(n) \leq \frac{n^2}{2} + \Theta(n\log n)$

Caso 2b: Intervalo $[k,l]$ é subintervalo de $[i,j]$.

$E[X_{i,j}X_{k,l}] = \sum_{i=1}^{n-3}\sum_{k=i+1}^{n-2}\sum_{l=k+1}^{n-1}\sum_{j=l+1}^{n}\frac{1}{(j-i+1)(j-i)(l-k+1)(l-k)} \leq \sum_{i=1}^{n-3}\sum_{k=i+1}^{n-2}\sum_{l=k+1}^{n-1}\sum_{j=l+1}^{n}\frac{1}{(l-k+1)(l-k)(l-k+1)(l-k)}$

$E[X_{i,j}X_{k,l}] \leq \sum_{i=1}^{n-3}\sum_{k=i+1}^{n-2}\sum_{l=k+1}^{n-1}\sum_{j=l+1}^{n}\frac{1}{(l-k)^4} \leq n\sum_{i=1}^{n-3}\sum_{k=i+1}^{n-2}\sum_{l=k+1}^{n-1}\frac{1}{(l-k)^4}$

$$E[X_{i,j}X_{k,l}] \leq n\sum_{i=1}^{n-3}\sum_{k=i+1}^{n-2}\left(\int_k^{n-2}\frac{1}{(l-k)^4}\,\mathrm{d}l\right) \leq 3n\sum_{i=1}^{n-3}\sum_{k=i+1}^{n-2}\frac{1}{k^3}$$

$$E[X_{i,j}X_{k,l}] \leq 3n \sum_{i=1}^{n-3} \left(\int_{i}^{n-3} \frac{1}{k^3}\,\mathrm{d}k \right) \leq 6n \sum_{i=1}^{n-3} \tfrac{1}{i^2} \leq 6n \left(1 + \int_{1}^{n-4} \frac{1}{i^2}\,\mathrm{d}i \right) \leq$$

$18n$

Caso 3: Intervalos $[i, j]$ e $[k, l]$ se intersectam porém sem compartilhar uma extremidade.

$E[X_{i,j}X_{k,l}] = \sum_{i=1}^{n-3} \sum_{k=i+1}^{n-2} \sum_{j=k+1}^{n-1} \sum_{l=j+1}^{n} \frac{1}{(l-i+1)(l-i)(j-i)(l-k-1)}$

Essa soma pode ser escrita como:

$E[X_{i,j}X_{k,l}] = \sum_{i=1}^{n-3} \sum_{j=i+2}^{n-1} \sum_{k=i+1}^{j-1} \sum_{l=j+1}^{n} \frac{1}{(l-i+1)(l-i)(j-i)(l-k-1)}$

Desde que $j < l$ e $l - k < j - i$, nós temos:

$E[X_{i,j}X_{k,l}] < \sum_{i=1}^{n-3} \sum_{j=i+2}^{n-1} \sum_{k=i+1}^{j-1} \sum_{l=j+1}^{n} \frac{1}{(j-i+1)(j-i)(l-k)(l-k-1)}$

$E[X_{i,j}X_{k,l}] < \sum_{i=1}^{n-3} \sum_{j=i+2}^{n-1} \left[\frac{1}{(j-i+1)(j-i)} \left(\sum_{k=i+1}^{j-1} \sum_{l=j+1}^{n} \frac{1}{(l-k)(l-k-1)} \right) \right]$

$E[X_{i,j}X_{k,l}] < \sum_{i=1}^{n-3} \sum_{j=i+2}^{n-1} \left[\frac{1}{(j-i+1)(j-i)} \sum_{k=i+1}^{j-1} \sum_{l=j+1}^{n} \left(\frac{1}{l-k-1} - \frac{1}{l-k} \right) \right]$

$E[X_{i,j}X_{k,l}] < \sum_{i=1}^{n-3} \sum_{j=i+2}^{n-1} \left[\frac{1}{(j-i+1)(j-i)} \sum_{k=i+1}^{j-1} \left(\frac{1}{j-k} - \frac{1}{n-k} \right) \right]$

$E[X_{i,j}X_{k,l}] < \sum_{i=1}^{n-3} \sum_{j=i+2}^{n-1} \frac{H_n}{(j-i+1)(j-i)}$

Assim, $E[X_{i,j}X_{k,l}] = O(n \log n)$.

Caso 4: Intervalos $[i, j]$ e $[k, l]$ se intersectam com $i = k$.

$E[X_{i,j}X_{k,l}] = \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} \sum_{l=j+1}^{n} \frac{1}{(l-i+1)(l-i)(j-i)}$

$E[X_{i,j}X_{k,l}] = \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} \left[\frac{1}{(j-i)} \sum_{l=j+1}^{n} \frac{1}{(l-i+1)(l-i)} \right] \leq \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} \frac{1}{(j-i)(j-i+1)}$

Então, $E[X_{i,j}X_{k,l}] \leq n - H_n$.

Caso 5: Intervalos $[i, j]$ e $[k, l]$ se intersectam com $j = k$.

$E[X_{i,j}X_{k,l}] = \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} \sum_{l=j+1}^{n} \frac{1}{(l-i+1)(l-i)(l-i-1)} \leq \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} \sum_{l=j+1}^{n} \frac{1}{(l-i+1)(l-i)}$

$E[X_{i,j}X_{k,l}] \leq \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} \sum_{l=j+1}^{n} \left(\frac{1}{l-i} - \frac{1}{l-i+1} \right) \leq \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} \frac{1}{j-i+1}$

$E[X_{i,j}X_{k,l}] \leq \sum_{i=1}^{n-2} H_n$. Então, $E[X_{i,j}X_{k,l}] = O(n \log n)$.

Caso 6: Intervalos $[i, j]$ e $[k, l]$ se intersectam com $j = l$.

Isso é idêntico ao caso 4. Então, $E[X_{i,j}X_{k,l}] \leq n - H_n$.

Uma vez que analisamos todos os seis casos estamos prontos para estimar $E[X^2]$.

Primeiro, é importante observar que, com exceção do caso 2a, todos os outros casos são limitados superiormente por $O(n \log n)$. No caso 2a, em particular, $E[X_{i,j}X_{k,l}] = \frac{n^2}{2} + \Theta(n \log n)$. Além disso, pela definição de $E[X^2]$, com exceção do caso 1, o resultado de todos os outros casos deve ser multiplicado por 2. Assim, $E[X^2] = n^2 + \Theta(n \log n)$.

Lembrando que, da desigualdade de Chebyshev, $Pr\{|X - E[X]| \geq cE[X]\} \leq \frac{E[X^2]-E[X]^2}{c^2 E[X]^2}$.

Consequentemente, $Pr\{X \geq cn\} \leq Pr\{X \geq c(n - H_n)\} = \frac{n^2+\Theta(n \log n)-(n-H_n)^2}{c^2(n-H_n)^2}$.

Assim, $Pr\{X \geq cn\} \leq \frac{c' \log n}{c^2 n}$ onde c' é uma constante positiva. ■

Referências Bibliográficas

[1] BENDER, M. A.; FARACH-COLTON, M. **The lca problem revisited**. In: IN LATIN AMERICAN THEORETICAL INFORMATICS, p. 88–94. Springer, 2000.

[2] BENTLEY, J. **Information Processing Letters**. Decomposable search problems, journal, v.5, n.8, p. 244–201, 1979.

[3] CHRISTOS FALOUTSOS, M. R.; MANOLOPOULOS, Y. **Fast subsequence matching in time-series databases**. In: IN PROCEEDING OF ACM SIGMOD, p. 419–429. Minneapolis, MN, 1994.

[4] DE BERG, M.; VAN KREVELD, M.; OVERMARS, M. ; SCHWARZKOPF, O. **Computational Geometry: Algorithms and Applications**. Second. ed., Springer-Verlag, 2000, 367p.

[5] E GATEV, W. G.; ROUWENHORST, K. Pairs trading: Performance of a relative-value arbitrage rule, journal, v.19, n.3, p. 797–827, 2007.

[6] FISCHER, J.; HEUN, V. **Theoretical and practical improvements on the rmq-problem, with applications to lca and lce**. In: PROC. CPM. VOLUME 4009 OF LNCS, p. 36–48. Springer, 2006.

[7] FU., T. **Engineering Applications of Artificial Intelligence**. A review on time series data mining, journal, v.24, p. 164–181, 2011.

[8] HAREL, D.; TARJAN, R. E. **SIAM J. Comput.** Fast algorithms for finding nearest common ancestors, journal, v.13, p. 338–355, May 1984.

[9] HUANMEI WU, B. S.; ZHANG, D. **Online event-driven subsequence matching over financial data streams**. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, p. 23–34. ACM Press, 2004.

[10] PRATT, K. B.; FINK, E. **International Journal of Image and Graphics**. Search for patterns in compressed time series, journal, v.1, n.2, p. 89–106, 2002.

[11] SEUNG-HWAN LIM, H. P.; KIM, S.-W. **Inf. Sci**. Using multiple indexes for efficient subsequence matching in time-series databases, journal, v.24, n.177, p. 5691–5706, 2007.

[12] SHASHA, D.; ZHU., Y. **High performance discovery in time series: techniques and case studies**. Springer-Verlag, 2004.

[13] YANG-SAE MOON, K.-Y. W.; HAN, W.-S. **General match: a subsequence matching method in time-series databases based on generalized windows**. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, p. 382–393. ACM Press, 2002.