Universidade Federal do Rio de Janeiro
Instituto de Matemática
Departamento de Ciência da Computação
Rio de Janeiro, RJ - Brasil

Aplicação do algoritmo *Perceptually Important Points* em séries temporais de datacenters

Ronald Andreu Kaiser
Orientadora:
Adriana Santarosa Vivacqua

Dezembro de 2010

Aplicação do algoritmo *Perceptually Important Points* em séries temporais de datacenters

Ronald Andreu Kaiser

Projeto Final de Curso submetido ao Departamento de Ciência da Computação do Instituto de Matemática da Universidade Federal do Rio de Janeiro como parte dos requisitos necessários para obtenção do grau de Bacharel em Informática.

Apresentado por:

_____

Ronald Andreu Kaiser

Aprovado por:

_____

Adriana Santarosa Vivacqua
D.Sc. COPPE/UFRJ

_____

João Carlos Pereira da Silva
D. Sc. UFRJ

_____

Marcello Goulart Teixeira
D. Sc. PUC-Rio

RIO DE JANEIRO, RJ - BRASIL
Dezembro de 2010

Agradecimentos

Agradeço a todo o Universo, por ter se organizado de tal maneira que fosse possível a realização desse projeto. Obrigado por sua complexidade, que torna nossas vidas menos enfadonhas e previsíveis.

Agradeço a minha mãe, Monique E. C. D. Nitsche, por ter me ajudado a experimentar a vida e por sua luta em dar o melhor de si para que eu pudesse sobreviver e alcançar meus objetivos.

Agradeço a minha tia Vera Sarres, por ter aturado minhas excentricidades por esses anos e me dado abrigo e conforto.

Agradecimento especial aos amigos da casa Intelie, Ricardo Clemente, Pedro Teixeira, Hubert Fonseca e o grande Falcão pelo apoio e incentivo que vocês me deram durante todo o tempo que nos conhecemos. Obrigado pelas grandes lições de vida e experiências que vocês me proporcionaram. Obrigado pela oportunidade que foi fundamental para realização deste trabalho e a compreensão nos momentos que precisei me ausentar para terminá-lo. Serei eternamente grato a vocês.

Agradeço a Anna Cruz pelas conversas e por ter me ajudado a tirar alguns screenshots para a versão final e ao Rodolfo Carvalho, pelas coisas que aprendi com ele.

Agradeço a minha orientadora, Adriana S. Vivacqua, por ter aceitado esse desafio, seu incentivo e indispensável ajuda nesse trabalho.

Agradeço a UFRJ por, apesar de todos os seus defeitos, prezar pelo ensino de qualidade e excelência de computação.

Agradeço ao Tiago S. da Silva, por sua amizade e pelos papos filosóficos que tivemos durante o curso.

Agradeço a Adriana Pinho pelas prazerosas conversas sobre ciência e sociedade, além de conselhos para essa monografia.

Agradeço também a todas as inúmeras pessoas e situações que me ajudaram direta ou indiretamente a ser uma pessoa melhor.

Resumo

Kaiser, R. A. Aplicação do algoritmo *Perceptually Important Points* em séries temporais de datacenters.

O propósito deste trabalho é auxiliar operadores de datacenters na visualização de gráficos de séries temporais, que representam a evolução do comportamento de seus equipamentos e serviços. Para tanto, foi buscada inspiração em um algoritmo utilizado para o reconhecimento de padrões em séries temporais do mercado financeiro. O algoritmo *Perceptually Important Points* ao mesmo tempo que contribui para a redução de dimensionalidade, fornece um mecanismo automatizado de eleição dos pontos mais importantes de uma série temporal para um observador humano, favorecendo, deste modo, sua visualização e tráfego pela rede. A implementação do algoritmo e sua implantação em um sistema de monitoração de datacenters já existente, hoje utilizado por grandes datacenters do Brasil, constituem o cerne deste trabalho.

Palavras-Chave: Séries Temporais, Redução de dimensionalidade, Perceptually Important Points, Datacenters, Clojure.

Abstract

Kaiser, R. A. Aplicação do algoritmo *Perceptually Important Points* em séries temporais de datacenters.

The main purpose of this work is helping operators of datacenters in the task of visualizing the behaviour of their devices and services through time, represented by large time series. In order to accomplish that, a technique used in pattern recognition from the financial market context was choosed. The "Perceptually Important Points" algorithm gives a method for dimensionality reduction and a mechanism to automatically extract the most important points from a human observer perspective, favouring compression and a good visualization of time series with high dimensionality. The implementation of the algorithm and its integration in an existing monitoring system was explored and encompasses the content of this work.

Keywords: Time series, Dimensionality Reduction, Perceptually Important Points, Datacenters, Clojure.

Lista de Figuras

2.1 Preço de fechamento de ações da IBM - 2010. 11
2.2 Padrões de tendência e sazonalidade em séries temporais 12
2.3 Piecewise Linear Approximation 13
2.4 Comparação entre algoritmos de representação. 14
2.5 Família de Wavelets. 15
2.6 Gráficos típicos no dia a dia de operação de um datacenter 17

3.1 Evolução do algoritmo PIP 19
3.2 Distância vertical. 20
3.3 Variância influencia número de PIPs 23

4.1 Comparativo entre as 3 versões implementadas 29

5.1 Arquitetura do sistema HOLMES 31
5.2 Gráficos gerados a partir do algoritmo PIP no sistema HOLMES 33
5.3 Gráficos gerados a partir do algoritmo PIP no sistema HOLMES 34

D.1 Série original com 8582 pontos reduzida com PIP para 300 pontos 48

Lista de Tabelas

4.1 Tempos de execução 28
4.2 Tempos de execução para as duas melhores versões 28

Lista de pseudo-algoritmos

1 Pseudo-algoritmo Perceptually Important Points 21
2 Pseudo-algoritmo Perceptually Important Points 2 22
3 Pseudo-algoritmo Perceptually Important Points 3 23

Sumário

1 Introdução **6**
- 1.1 Motivação . . . 7
- 1.2 Objetivos . . . 8
- 1.3 Metodologia . . . 8
- 1.4 Organização da Monografia . . . 8

2 Séries temporais **10**
- 2.1 Características . . . 11
  - 2.1.1 Dependência de observações . . . 11
  - 2.1.2 Alta dimensionalidade . . . 11
  - 2.1.3 Sazonalidade . . . 12
  - 2.1.4 Tendência . . . 12
- 2.2 Representação e redução de dimensionalidade . . . 13
  - 2.2.1 Algoritmos utilizados . . . 13
- 2.3 Séries temporais de datacenters . . . 16

3 Perceptually Important Points **18**
- 3.1 Ideia do algoritmo . . . 18
- 3.2 Cálculo de distâncias . . . 20
- 3.3 Pseudo-algoritmo . . . 21
  - 3.3.1 Análise de Complexidade . . . 21
- 3.4 Cálculo de erro . . . 22
- 3.5 Limitações do algoritmo . . . 23

4 Implementação do algoritmo **24**
- 4.1 Clojure . . . 24
- 4.2 Desenvolvimento . . . 24
- 4.3 Desempenho . . . 25
  - 4.3.1 Otimizações . . . 25

5 Aplicação em um sistema real **30**
- 5.1 HOLMES . . . 30
- 5.2 Integração HOLMES e PIP . . . 30
  - 5.2.1 Arquitetura . . . 31
  - 5.2.2 Visualização . . . 32
- 5.3 Caso de Uso . . . 32

6 Conclusão **35**

Apêndices **36**

A Primeira Versão **37**

B Segunda Versão **39**

C Terceira Versão **42**

D Aproximações **48**

Capítulo 1

Introdução

Manter sistemas complexos em pleno funcionamento e harmonia requer constante monitoração de suas partes. Datacenters, por menores que sejam, precisam ser constantemente monitorados para garantir que seus equipamentos e serviços estejam disponíveis e em correto funcionamento. Nesses ambientes, subutilização de recursos, períodos de inatividade ou falhas em dispositivos são situações indesejáveis que podem levar a perdas de faturamento significativas por parte de seus mantenedores.

Para evitar tais riscos, medições sobre equipamentos e serviços precisam ser coletadas continuamente com o objetivo de apoiar a tomada de decisão, tanto em níveis micro, onde simples mudanças de dispositivos são agendadas, até reestruturações em arquiteturas de serviços.

Datacenters podem ser vistos como verdadeiras "máquinas" de produção de séries temporais. Como medições são realizadas com frequência, é natural observar nesse contexto séries temporais com alta dimensionalidade representadas por *datasets* de muitos gibibytes de aglomerados numéricos semi-estruturados. É desejável, portanto, que se tenha um meio apropriado de visualização de dados e entre outras formas de leitura, a visualização de gráficos desempenha um papel importante.

O propósito deste trabalho é auxiliar operadores de datacenters na recuperação e visualização de gráficos, que representam a evolução do comportamento de seus equipamentos e serviços, e para tanto foi buscada inspiração em um algoritmo utilizado para o reconhecimento de padrões em séries temporais do mercado financeiro.

O algoritmo *Perceptually Important Points*[1] ao mesmo tempo que contribui para a redução de dimensionalidade, fornece um mecanismo automatizado de eleição dos pontos mais importantes de uma série temporal para um observador humano, favorecendo, deste modo, sua visualização e tráfego pela rede.

A implementação do algoritmo e sua implantação em um sistema de monitoração de

[1] Preferiu-se o termo *Perceptually Important Points* em sua língua inglesa por ser mais conhecido dessa forma na literatura, além de oferecer o mneumônico e acrônimo PIP.

datacenters já existente[2], hoje utilizado por grandes datacenters do Brasil, constituem o cerne deste texto.

Nas próximas seções são apresentadas as motivações que fomentaram a elaboração deste trabalho, os objetivos pretendidos, a metodologia aplicada e um resumo de cada capítulo foi acrescentado para que o leitor possa obter uma visão geral da organização deste texto.

1.1 Motivação

Softwares atuais largamente utilizados no contexto de datacenters, como RRDtool [1] se baseiam em funções agregadoras simples de média, máximo ou mínimo, que podem deturpar a interpretação de seus gráficos.

A motivação deste trabalho é a de solucionar a carência de um mecanismo eficaz para a representação e visualização de séries temporais em um software de monitoração de datacenters, a saber, HOLMES.

O sistema HOLMES conta com uma interface web, onde o usuário final, em geral, um operador de datacenter, tem a capacidade de visualizar o comportamento de um de seus equipamentos ou serviços ao longo do tempo. Para tanto, o usuário faz requisições HTTP a séries temporais via REST[3].

Responder a essas requisições com os dados originais pode provocar alguns problemas, tais como:

- Perdas de desempenho causadas pelo tráfego de milhares de pontos pela rede;
- Gráficos com muitos pontos em uma área reduzida pode favorecer representações muito densas, visualmente poluídas e pouco informativas;
- Consumo de memória excessivo para o usuário final, dado que os pontos são armazenados no web browser do cliente.

Para se ter uma noção do ambiente atual do sistema, existem 2000 entidades (produtores de eventos) gerando 20 tipos de eventos distintos, com 5 variáveis numéricas em média cada, totalizando 200.000 séries temporais. Com uma resolução de 1 ponto a cada 5 minutos, temos aproximadamente 288 pontos por dia para cada uma das séries, que em um ano, totalizam 105.120 pontos. Para todos as séries, em um ano, temos 21.024.000.000 pontos para armazenar em apenas duas máquinas.

[2]O software citado recebe o nome de HOLMES e será apresentado no capítulo 5.

[3]REST, Representational State Transfer, técnica de engenharia de software para sistemas hipermídia distribuídos. Comumente usada no contexto da World Wide Web.

1.2 Objetivos

A fim de solucionar os problemas mencionados na seção anterior, os objetivos traçados para a realização deste trabalho incluem:

1. Implementar um algoritmo que reduza a dimensionalidade de séries temporais e represente visualmente de modo mais fiel possível as séries originais;

2. Integração em um software de monitoração de datacenters.

1.3 Metodologia

Para fins de execução dos desafios propostos, fez-se necessário o estudo e pesquisa de artigos e livros na área de séries temporais. Fundamentado nesse estudo, optou-se por implementar o algoritmo *Perceptually Important Points* por ser conceitualmente simples e atender de forma satisfatória no domínio aplicado.

Como o algoritmo escolhido apresentava complexidade de tempo $O(n^2)$, questões de desempenho foram trabalhadas e viabilizar a implementação do código-fonte requereu um estudo mais aprofundado sobre a linguagem de programação Clojure.

Vale ressaltar também que desde o princípio deste projeto foi vislumbrada a implantação do algoritmo em um software de monitoração de datacenters e sua arquitetura e integração com o algoritmo escolhido foi pensada e analisada durante todo o processo.

1.4 Organização da Monografia

- O capítulo 2 contribui com um apanhado sobre o que existe na literatura sobre séries temporais objetivando fundamentar as bases teóricas para o desenvolvimento dos próximos capítulos. Modelos de representação e características de séries temporais são apresentados, além de contextualizações com a realidade de datacenters típicos.

- O capítulo 3 apresenta o algoritmo *Perceptually Important Points*, escolhido para implementação neste trabalho. A intuição por trás do método é revelada, junto com seu pseudo-algoritmo, análise de complexidade, limitações e comparações com métodos existentes.

- O capítulo 4 expõe uma instância do algoritmo *Perceptually Important Points* implementada na linguagem Clojure. Detalhes de implementação, características do código gerado, dificuldades e testes de desempenho realizados são apresentados.

- O capítulo 5 demonstra como se realizou a integração do algoritmo descrito no capítulo 4 em um software de monitoração de datacenters. Seu propósito e uma visão

geral da arquitetura do sistema envolvida na integração do algoritmo é discutida, seguida de um caso de uso de sucesso.

- Por fim, no capítulo 6 seguem as conclusões sobre o trabalho realizado e propostas de trabalhos futuros são delineadas.

Capítulo 2

Séries temporais

Na área médica, é comum a monitoração da evolução de funções vitais de pacientes através de aparelhos como eletroencefalogramas e eletrocardiogramas, além do acompanhamento da incidência de padrões sazonais de epidemias [2]. Na ciência econômica, muitos livros [3] e artigos [4] já foram escritos sobre análises e previsões do mercado financeiro, políticas fiscais e monetárias, baseadas em observações no tempo. O que a Medicina e a Economia e outras áreas supostamente não correlatas possuem em comum é o fato de permitirem a aplicação de estudos e métodos de séries temporais.

Séries temporais podem ser construídas sobre praticamente quaisquer medições possíveis no tempo e esse caráter genérico é o que torna a utilidade do seu estudo indiscutível em diversas áreas do conhecimento humano. O tema é usado com sutis diferenças entre matemáticos, estatísticos e outros profissionais. Por essa razão, segue a definição que será utilizada[1]:

Definição 1. Série temporal: sequência de observações[2] ordenadas cronologicamente (n-tupla):

$$S = (s_1, s_2, s_3, ..., s_n) \tag{2.1}$$

Quando tais observações são coletadas no tempo continuamente, diz-se que a série temporal é *contínua* e quando as medições se fazem em instantes de tempo específicos, geralmente equidistantes, são conhecidas como séries temporais *discretas*.

Na figura 2.1, podemos visualizar um exemplo de série temporal discreta, descrevendo o preço de fechamento das ações da IBM ao longo do ano de 2010.

Em geral, o estudo de séries temporais comporta objetivos dos mais diversos, como identificação de tendências e outliers[3], predição, similaridade entre séries temporais por clusterização, existência de sazonalidades – endereçado na seção 2.1.3 – entre outros.

[1] Séries temporais também podem ser definidas como um conjunto de observações $\{X(t), t \in T\}$, com X representando a variável aleatória observada e T um conjunto de índices.

[2] Tais observações podem ser coletadas em períodos regulares ou não.

[3] Observações distantes ou discrepantes quando comparadas globalmente com outras observações.

Figura 2.1: Preço de fechamento de ações da IBM no ano de 2010 [5].

A seguir, serão apresentadas características e técnicas envolvidas na visualização, representação e análise de séries temporais. Ao final deste capítulo, contextualizaremos o papel de séries temporais com a realidade de grandes datacenters.

2.1 Características

2.1.1 Dependência de observações

O valor das ações da IBM ao final de um dia se torna naturalmente dependente do seu desempenho em dias anteriores. Da mesma forma, o número de pessoas que habitavam o planeta Terra em anos passados tem influência direta sobre a população mundial de hoje.

É comum encontrar na literatura [6], ressalvas sobre a aplicação de muitos métodos estatísticos tradicionalmente baseados na suposição de que observações adjacentes são independentes e identicamente distribuídas. Sendo assim, identificar a existência de correlações entre medições no tempo é essencial para que possamos aplicar análises estatísticas apropriadas.

2.1.2 Alta dimensionalidade

É comum observar na literatura a interpretação da n-tupla descrita na definição 2.1 como um ponto em uma espaço de dimensão $\Re^n$. Como novas observações são coletadas constantemente, séries temporais assumem, portanto, caráter de alta dimensionalidade.

Como possuem centenas de milhares a milhões de pontos e tratá-los em sua forma bruta se torna computacionalmente custoso, é desejável, em aplicações práticas, um mecanismo que favoreça a redução de dimensionalidade.

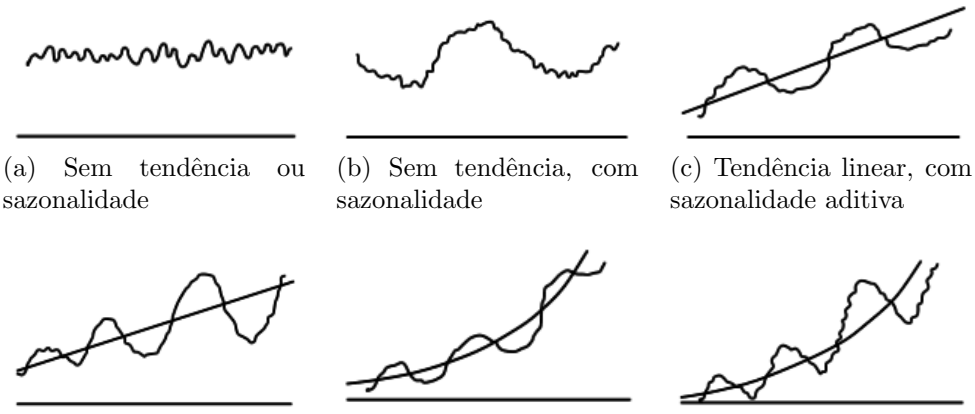
2.1.3 Sazonalidade

É possível identificar na Natureza diversos fenômenos que possuem algum comportamento sazonal. A produção de arroz na China, o número de passagens vendidas por uma companhia aérea ao longo de um ano, a migração das aves com a chegada do inverno e a quantidade de visitas em uma página na Internet possuem no mínimo um comportamento em comum: sazonalidade.

No contexto de séries temporais, o termo *sazonalidade* diz respeito a padrões que se repetem em períodos no tempo. Sazonalidades podem ser classificadas em dois tipos: aditivas e multiplicativas. A primeira retrata flutuações que não sofrem mudanças muito bruscas quando considerada a série globalmente. Sazonalidades multiplicativas dependem de um fator em cada período em que é observada e possuem uma maior variabilidade quando comparadas com a série original como um todo.

2.1.4 Tendência

A quantidade de água doce no mundo vem diminuindo ao longo dos anos. Ao mesmo tempo, o número de artigos científicos publicados no meio acadêmico cresce monotonicamente. Tendências de crescimento ou decrescimento podem ser identificadas em séries temporais e podem ser caracterizadas por sua taxa de variação. Se sua taxa de crescimento se aproximar de uma reta, diz-se que a série possui tendência linear. Caso contrário, tendências podem ser classificadas como não lineares, exponenciais ou quadráticas.



(a) Sem tendência ou sazonalidade

(b) Sem tendência, com sazonalidade

(c) Tendência linear, com sazonalidade aditiva

(d) Tendência linear e sazonalidade multiplicativa

(e) Tendência não linear e sazonalidade aditiva

(f) Tendência não linear e sazonalidade multiplicativa

Figura 2.2: Padrões de tendência e sazonalidade em séries temporais

Na figura 2.2, padrões de comportamento relacionando sazonalidade e tendência podem ser observados. Além das características apresentadas, outras bastante utilizadas na literatura e não menos importantes são estacionariedade, aleatoriedade e ciclidade. A

seguir, uma breve descrição de técnicas largamente utilizadas para análise e compressão de séries temporais.

2.2 Representação e redução de dimensionalidade

A representação de séries temporais em sua forma original pode ser computacionalmente custosa, dada sua alta dimensionalidade – discutida na seção 2.1.2. A escolha do método de representação deve se basear na aplicabilidade desejada. Existem atualmente diversas abordagens para compressão e representação de séries temporais. Algumas se baseiam em transformações de domínios, como *Discrete Fourier Transform*, que mapeiam para o domínio de frequências uma série em seu domínio temporal, outras podem preservar dois domínios, como é o caso de *Discrete Wavelet Transform*. O objetivo dessa seção é apresentar as ideias das abordagens mais conhecidas e utilizadas.

2.2.1 Algoritmos utilizados

Séries temporais podem ser vistas como funções matemáticas e como tais podem se favorecer de técnicas básicas da Matemática Aplicada, como aproximações de funções complexas por funções mais simples, como retas. O método conhecido como PLA (Piecewise Linear Approximation) segmenta uma série temporal em partes e para cada uma delas define a reta que mais se aproxima dos pontos no segmento considerado. Na figura 2.3 podemos ver um exemplo de como uma série pode ser aproximada por esse método.

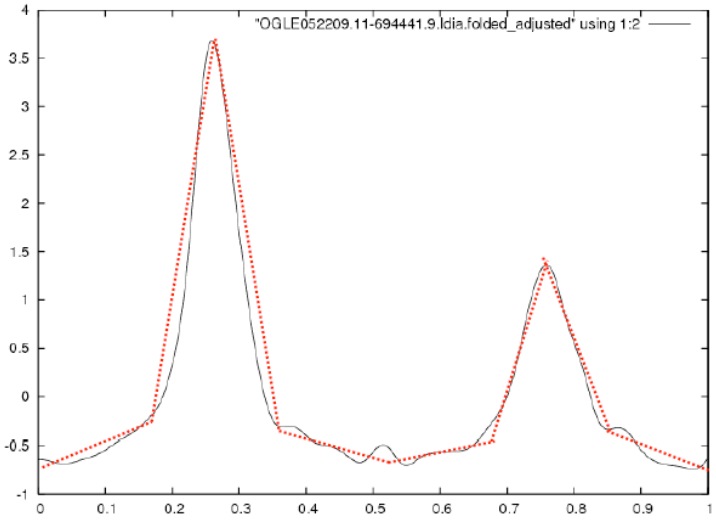


Figura 2.3: Piecewise Linear Approximation

Abordagens com teor matemático mais apurados também são utilizadas para representar séries temporais. Os resultados de Jean-Baptiste Joseph Fourier, que demonstrou que todo sinal periódico poderia ser aproximado por somas de senos e cossenos, podem

ser mapeados para o campo dos números complexos, dando origem a transformada de Fourier.

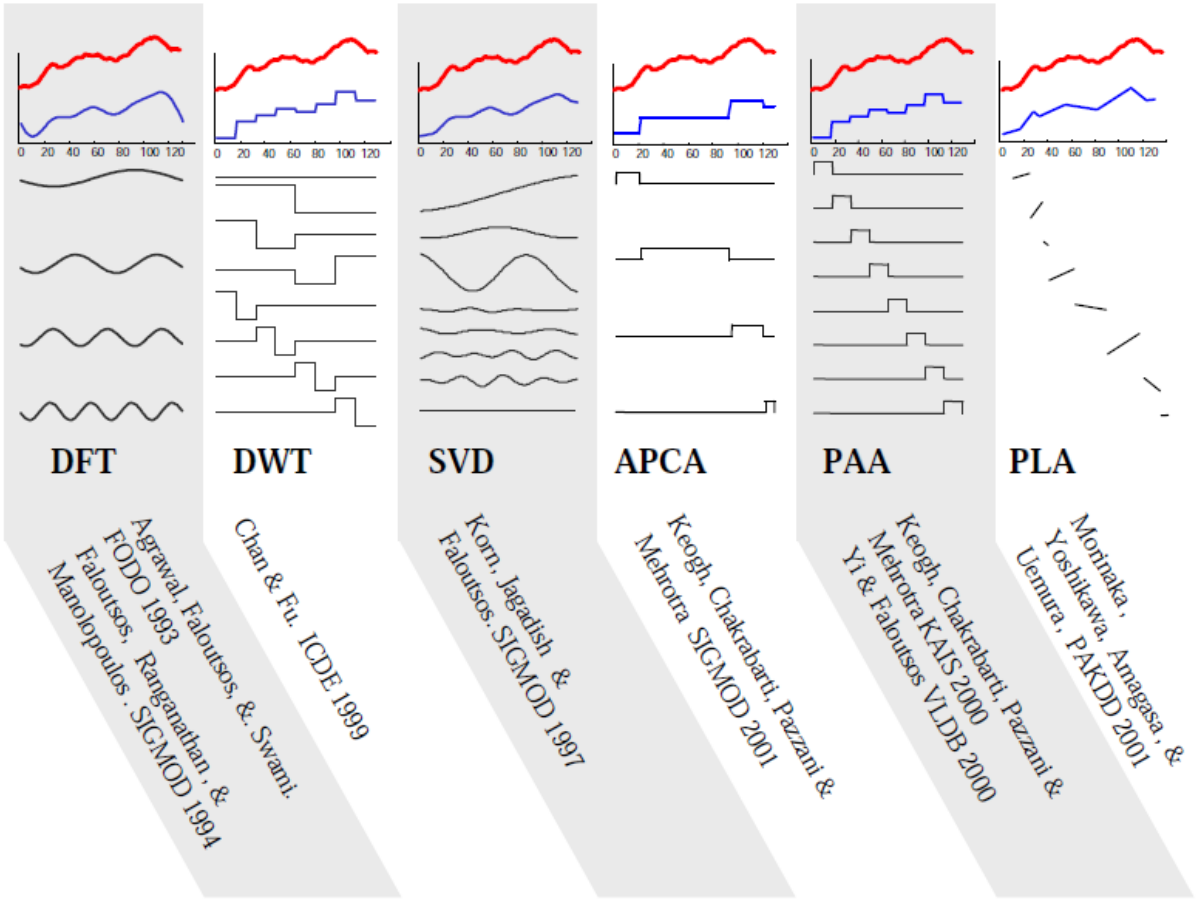


Figura 2.4: Comparação entre algoritmos de representação e redução de dimensionalidade de séries temporais.

A partir dessa transformada, uma série temporal pode ser decomposta por suas componentes no domínio das frequências. Na primeira coluna da figura 2.4 podemos ver como uma aproximação pode ser feita com o algoritmo FFT (Fast Fourier Transform), uma das variantes de uma classe maior de algoritmos baseados na transformada de Fourier, conhecida como DFT (Discrete Fourier Transform). Uma desvantagem deste método é que se perde o referencial de eventos no tempo, tendo que se recorrer a transformada inversa de Fourier [7].

Outro algoritmo largamente utilizado para compressão de imagens e que pode ser utilizado para redução de dimensionalidade de séries temporais é o DWT (Discrete Wavelet Transform) – implementado durante o desenvolvimento deste projeto, mas omitido por objetividade –, mais vantajoso do que o DFT em alguns aspectos, pois preserva componentes no domínio do tempo [8]. Uma "wavelet" ou ondaleta é uma pequena onda que pode ser usada como base para a reconstrução da série original e diferente dos senos e cossenos utilizados no DFT não podem ser funções periódicas. Para a utilização desse al-

goritmo é necessário que se escolha previamente duas ondaletas, normalmente conhecidas como ondas mãe ψ e pai ϕ que devem satisfazer alguns critérios de ortogonalidade.

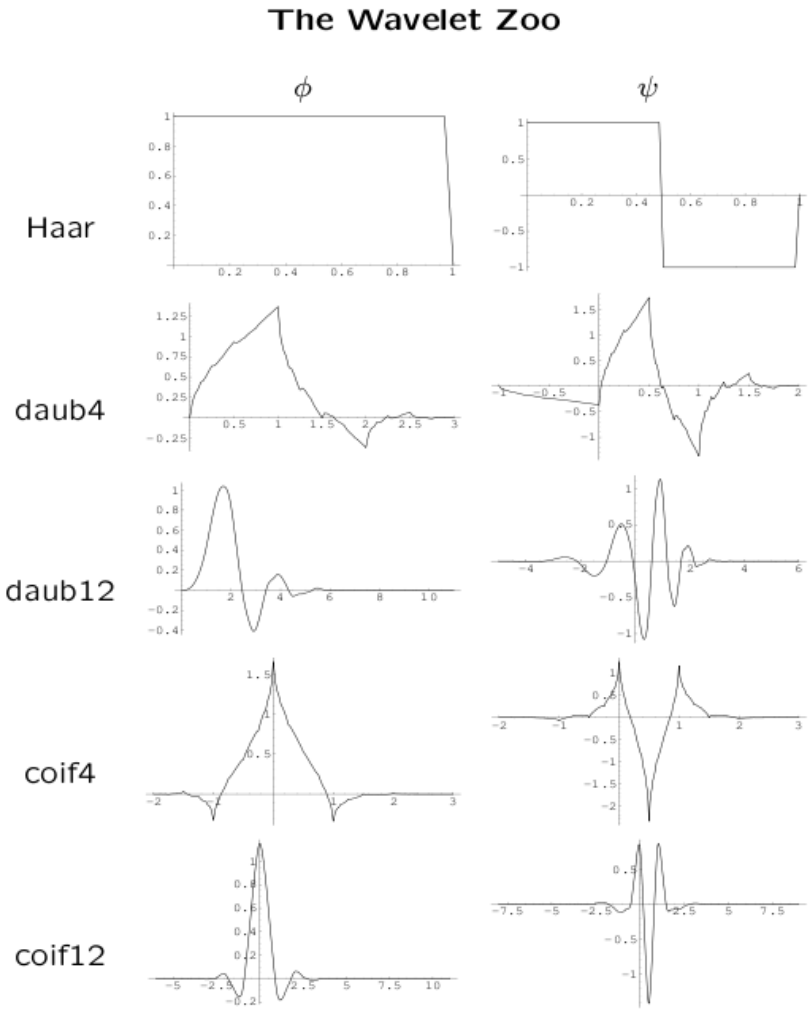


Figura 2.5: Família de Wavelets.

Na figura 2.5 podemos ver pares de funções mãe e pai que satisfazem esses critérios e podem ser utilizadas como wavelets. Repare na segunda coluna da figura 2.4 a aproximação de uma série utilizando a ondaleta Haar, reconhecida como a primeira ondaleta utilizada.

Ainda na figura 2.4, pode-se observar como os métodos apresentados e outros se comportam para a mesma série original de entrada. Em cada coluna, a série original de entrada aparece em vermelho e logo abaixo, em azul, a série aproximada pelo algoritmo correspondente. Comparações mais detalhadas sobre DFT e DWT podem ser encontradas em [7].

Além dos apresentados, existem na literatura diversos algoritmos de representação e redução de dimensionalidade de séries temporais. Alguns deles são: Piecewise Constant Approximation (PCA), Symbolic Approximation (SA), Symbolic Aggregate ApproXima-

tion (SAX), Singular Value Decomposition (SVD), CHEByshev polynomials (CHEB) e Self-Organizing Maps (SOM).

A partir dessa visão geral sobre o atual cenário no que diz respeito a representação e compressão de séries temporais, na próxima seção uma contextualização com o ambiente de datacenters será apresentada.

2.3 Séries temporais de datacenters

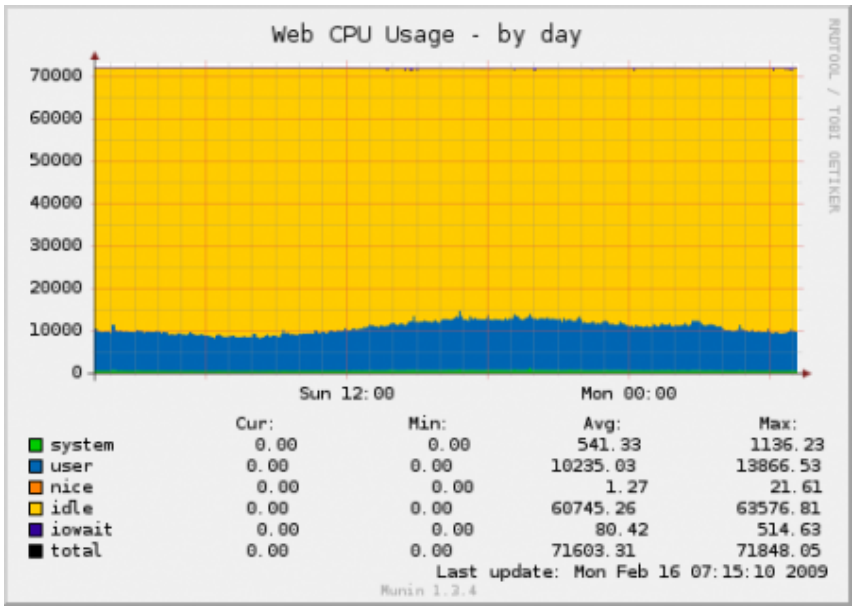
Datacenters são ambientes que concentram equipamentos de processamento e armazenamento de dados de empresas ou organizações. Podem ser considerados como legítimas "máquinas" de produção de séries temporais. Em uma instalação típica, são encontrados diversos racks de servidores centralizadores de inúmeras unidades de processamento e armazenamento.

O foco deste trabalho se concentra em séries temporais de observações feitas sobre os equipamentos e dispositivos desses datacenters. O operador de um grande datacenter, em geral, precisa monitorar dados de consumo de CPU, memória, sessões TCP abertas, requisições HTTP, taxa de hits em caches, número de consultas lentas a bancos de dados, entre outras métricas. Em datacenters possuímos um conjunto bem heterogêneo de padrões de séries temporais. Na figura 2.6 podemos ver alguns exemplos de gráficos que fazem parte do dia a dia de operação de um datacenter.

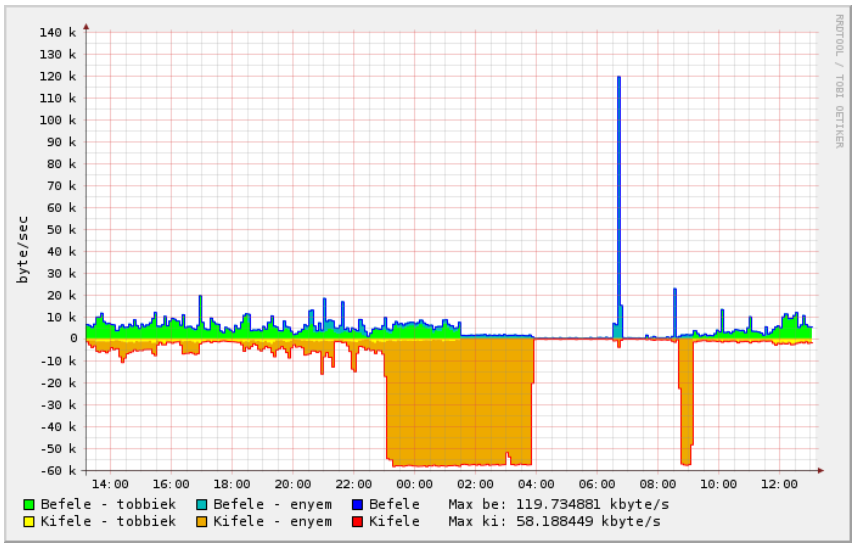
Esses gráficos foram gerados através de uma ferramenta bem conhecida, RRDtool. Com essa ferramenta é possível escrever scripts para coletar informações de dispositivos de tempos em tempos. O software é livre e utiliza o conceito de Round Robin Database, em que é mantido sempre uma quantidade fixa de pontos de cada série. Quando o limite é ultrapassado, algumas estratégias podem ser definidas. A mais simples de todas é descartar sempre os pontos mais antigos. Pode-se também configurar cálculos de agregados, resumindo pontos por sua média, mínimo ou máximo.

Essa abordagem nem sempre é satisfatória na busca por uma causa para uma possível falha em um desses equipamentos, pois médias em dados com variância muito alta podem mascarar falhas graves. É desejável, portanto, que se tenha uma forma eficiente de visualização que permita capturar características apresentadas nesse capítulo, como sazonalidades e tendências. Essas observações é que permitirão coletar fatos e reorganizar a estrutura do datacenter para que o mesmo se mantenha em pleno funcionamento. Para tanto, algum dos algoritmos de representação e redução de dimensionalidade se faz necessário.

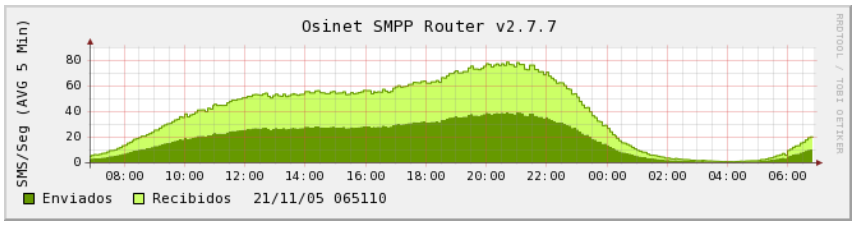
No próximo capítulo veremos as ideias que estão por trás do algoritmo *Perceptually Important Points*, seguido de sua implementação e a contextualização com datacenters será retomada no capítulo 5, que retratará a aplicação do algoritmo em um sistema real de monitoração.



(a) Consumo de CPU



(b) Tráfego de rede



(c) Uso do protocolo SMPP em um roteador

Figura 2.6: Gráficos típicos no dia a dia de operação de um datacenter

Capítulo 3

Perceptually Important Points

Para atingir o primeiro objetivo – proposto na seção 1.2 –, escolheu-se o algoritmo *Perceptually Important Points*, pois se mostrou conceitualmente simples em relação aos outros métodos apresentados no capítulo anterior, além de favorecer uma forma eficaz na redução de dimensionalidade de séries temporais.

3.1 Ideia do algoritmo

A primeira proposta formal do processo de identificação de *Perceptually Important Points* foi introduzida por [9] em uma análise técnica de padrões para aplicações financeiras, com ideias parecidas utilizadas em trabalhos independentes de [10] e [11]. Entretanto, não é conhecida a utilização dessa técnica para o auxílio da representação de séries temporais de datacenters.

O algoritmo expressa um método para se definir que pontos são de fato imprescindíveis e que devem ser considerados para que a plotagem do gráfico preserve sua forma original para um observador humano.

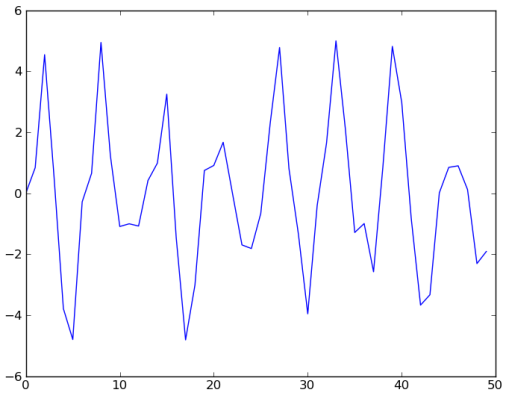
Para ilustrar a ideia do algoritmo, vamos utilizar como série base, uma amostra de 50 pontos de uma função seno multiplicada por um fator aleatório entre 0 e 5 – para incluir algum ruído nos dados, ilustrada na figura 3.1a.

Para podermos identificar os pontos mais importantes de uma série, é preciso considerá-la como um todo. Alguns pontos podem ser importantes em um período, mas completamente descartáveis quando se considera toda a sua história. Por essa razão, faz parte da inicialização do algoritmo incluir o primeiro e último pontos da série original, veja figura 3.1b.

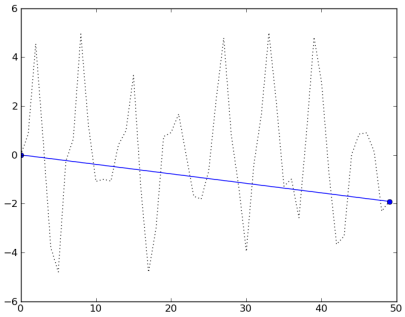
Em sua primeira iteração, calculamos a distância – descrita na seção 3.2 –, de todos os pontos no intervalo com relação aos primeiros pontos escolhidos na inicialização. O ponto que possuir maior distância dos dois primeiros será o próximo ponto a ser promovido a PIP.

Após a primeira iteração, processamos todas as distâncias, agora para os dois intervalos implícitos pelos 3 pontos já escolhidos. Escolhemos nessa iteração o ponto que possui a maior distância desses intervalos e o promovemos a PIP. O processo se repete até que todos os pontos sejam escolhidos e estejam devidamente ordenados.

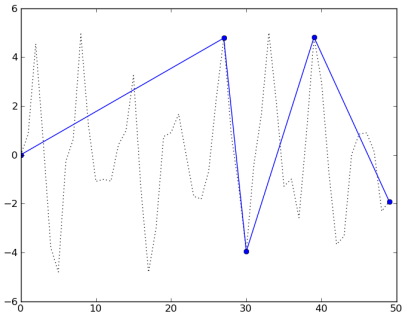
Outros procedimentos e condições de parada podem ser definidos, como por exemplo, um valor fixo para o número de pontos a serem escolhidos, ou a atribuição de um erro tolerável. Tais condições serão discutidas ainda neste capítulo na seção 3.3.



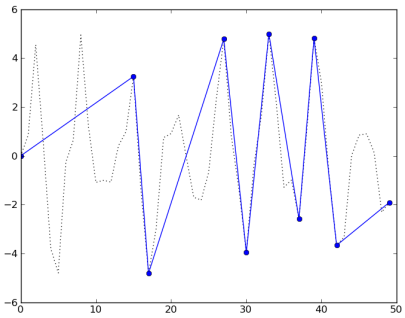
(a) Série original com 50 pontos.



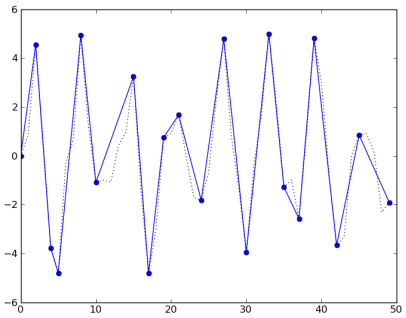
(b) Inicialização.



(c) 5 PIPs.



(d) 10 PIPs.



(e) 20 PIPs.

Figura 3.1: Evolução do algoritmo PIP

Na figura 3.1, conseguimos representar com 20 dos 50 pontos originais (descarte de 60 % dos dados originais) a série com boa precisão visual, quase se confundindo com a série original para um observador humano. Compressões ainda melhores podem ser alcançadas dependendo da forma geral da série considerada. No apêndice D, pode-se visualizar a representação de uma série original de um mês de dados, com 8582 pontos sendo reduzida para apenas 300 pontos (descarte de aproximadamente 99.97% dos pontos originais), com uma boa aproximação visual.

3.2 Cálculo de distâncias

Em [12] são apresentados testes comparativos com três abordagens diferentes para o cálculo de distâncias entre pontos: distância vertical, distância perpendicular e distância euclidiana. Constatou-se que a distância vertical apresentava os melhores resultados, e portanto, foi a escolhida para este trabalho.

Na figura 3.2 podemos capturar a ideia por trás do cálculo da distância vertical. Os pontos $p1$ e $p2$ representam PIPs previamente escolhidos e o ponto $P3$ um candidato a ser promovido a PIP. Este método incorpora a intuição de que flutuações verticais são mais importantes para um observador humano.

Em termos práticos, é utilizada a equação 3.1 para calcular a distância entre pontos no algoritmo. Podem ser utilizadas técnicas de geometria analítica para se chegar na equação 3.1 para o cálculo de distância vertical.

$$VD(p_3, p_c) = \mid y_3 - y_c \mid = \mid \left(y_1 + (y_2 - y_1)\left(\frac{x_3 - x_1}{x_2 - x_1}\right)\right) - y_3 \mid \tag{3.1}$$

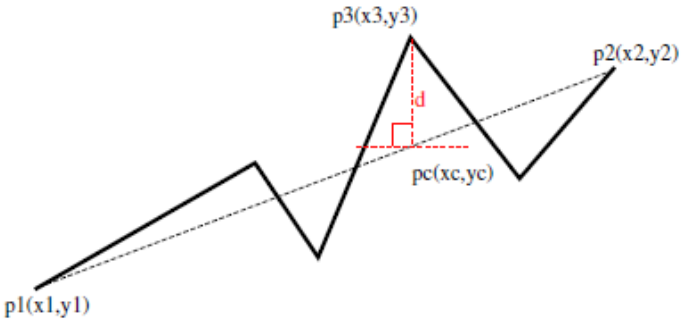


Figura 3.2: Distância vertical.

3.3 Pseudo-algoritmo

Pseudo-algoritmos tem seu papel fundamental em estabelecer as macro operações que devem ser realizadas para a resolução de um problema, independente de linguagem de programação, facilitando o entendimento do leitor e o endereçamento de algumas observações sobre o método.

A entrada do algoritmo esperada é uma lista de pontos da série temporal original. Sua saída é um subconjunto desses pontos, que supostamente deveriam ser os mais significativos para um observador humano.

Nas linha 1 e 2 inicializamos a lista de PIPs com o primeiro e último pontos da série. O bloco delimitado pelas linhas 3 a 6 itera sobre os pontos de entrada procurando pelo próximo ponto mais significativo a cada iteração. É importante perceber que a cada iteração são promovidos a PIP apenas um ponto, pois devemos considerar séries temporais por inteiro para não cair em ótimos locais.

Algorithm 1 Pseudo-algoritmo Perceptually Important Points

Require: Lista de pontos da série original $INPUT[1..n]$ ordenada
Ensure: Lista de PIPs $PIP[1..n]$
1: $PIP[1] \Leftarrow INPUT[1]$
2: $PIP[2] \Leftarrow INPUT[n]$
3: **repeat**
4: Selecionar ponto $P - MAX$ de $INPUT$ com maior distância a pontos adjacentes da lista de PIPs.
5: Inserir $P - MAX$ em PIP
6: **until** tenham sido escolhidos n pontos de $INPUT[1..n]$ PIP

O pseudo-algoritmo 1 retorna sempre a mesma lista com todos os pontos originais, ainda que a ordem com que sejam escolhidos os pontos revele a importância de cada ponto, a resposta é sempre a mesma, a série original. Pequenos ajustes podem ser considerados para deixar o algoritmo mais útil em termos práticos. Um deles é definir uma condição de parada para retornar apenas os $k <= n$ pontos mais importantes, exemplificado pelo pseudo-algoritmo 2.

3.3.1 Análise de Complexidade

Pelo pseudo-algoritmo 1, podemos observar que as linhas 1, 2 e 7 são executadas em tempo constante. Para fins de análise de complexidade podemos portanto, eliminá-las do cálculo. O bloco delimitado pelas linhas 3 a 6 representam o maior tempo de computação e por isso devemos analisá-lo com mais cuidado.

Na linha 4 são selecionados pontos com maior distância a pontos adjancentes da lista atual de PIPs. Como a cada iteração são promovidos a PIP apenas um ponto e na inicialização já escolhemos dois destes – primeiro e último da série original – são selecionados a

Algorithm 2 Pseudo-algoritmo Perceptually Important Points 2

Require: Lista de pontos da série original $INPUT[1..n]$ ordenada
Require: k pontos a serem escolhidos
Ensure: Lista de PIPs $PIP[1..k]$
1: $PIP[1] \Leftarrow INPUT[1]$
2: $PIP[2] \Leftarrow INPUT[n]$
3: **repeat**
4: Selecionar ponto $P - MAX$ de $INPUT$ com maior distância a pontos adjacentes da lista de PIPs.
5: Inserir $P - MAX$ em PIP
6: **until** tenham sido escolhidos k pontos de $INPUT[1..n]$ PIP

cada iteração, $(n-2, n-3, n-4, ..., 3, 2, 1)$ pontos para comparação de distâncias.

Inserir o ponto promovido a PIP na iteração corrente, linha 5, pode ser considerada constante, dependendo da estrutura de dados que se utiliza[1].

Sendo assim, a complexidade do algoritmo é dada pelo número de cálculos de distâncias que fazemos no total, e pode ser calculado como:

$$\sum_{i=1}^{n-2} i = \frac{n^2 - 3n + 2}{2}$$

Deste modo, a complexidade do algoritmo é dada por $O(n^2)$. Já para o pseudo-algoritmo 2, o mesmo raciocínio pode ser traçado e sua complexidade de tempo é $O(nk)$.

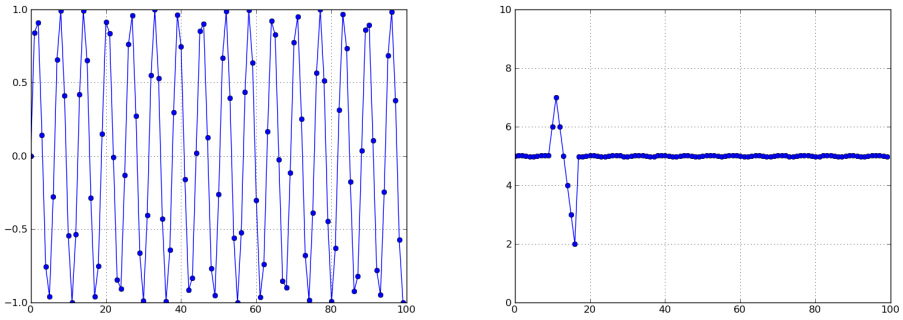
3.4 Cálculo de erro

Como destacado na seção 3.3, pequenas variantes do pseudo-algoritmo apresentado podem ser implementadas para solucionar algumas dificuldades. Para o pseudo-algoritmo 2 é preciso escolher como entrada o número k de pontos que se deseja resgatar da série original. Normalmente a escolha desse valor não gera bons resultados sem o conhecimento *a priori* da série original.

Em geral, o valor de k que representa razoavelmente bem[2] a série original não depende do valor de n. Para ilustrar tal afirmação, podemos visualizar a figura 3.3. Na série da figura 3.3a, um número maior de pontos são necessários para representá-la quando comparamos com a figura 3.3b, uma série mais bem comportada; ainda que as duas possuam a mesma quantidade de pontos no total, no caso 100.

Sendo assim, como não é razoável que a entrada do valor de k seja feita manualmente, é desejável um mecanismo automatizado de inferência sobre o menor valor de k necessário para preservar a forma geral da série original. Para tanto, podem ser utilizadas diferença

[1]Para a implementação, explicada com mais detalhes no próximo capítulo, as inserções são na verdade $O(log_{32}n)$, pois foi preferida a utilização de *hashsets*, mas que na prática pode ser considerado constante.
[2]Como razoavelmente bem, busca-se o conceito subjetivo de similaridade entre duas séries temporais.



(a) Série com 100 pontos, variância muito alta (b) Série com 100 pontos, baixa variância

Figura 3.3: Séries com o mesmo número de pontos podem necessitar de valores bem distintos de PIPs para serem representadas visualmente de modo satisfatório.

Algorithm 3 Pseudo-algoritmo Perceptually Important Points 3

Require: Lista de pontos da série original $INPUT[1..n]$ ordenada
Require: Limite máximo m de pontos a serem escolhidos
Require: Erro ϵ aceitável
Ensure: Lista de PIPs $PIP[1..s]$, com $s <= m$

1: $PIP[1] \Leftarrow INPUT[1]$
2: $PIP[2] \Leftarrow INPUT[n]$
3: **repeat**
4: Selecionar ponto $P - MAX$ de $INPUT$ com maior distância a pontos adjacentes da lista de PIPs.
5: Inserir $P - MAX$ em PIP
6: **until** m pontos de $INPUT[1..n]$ já foram escolhidos ou tenha sido alcançada uma diferença menor que ϵ PIP

de mínimos quadrados – para mais detalhes veja [12] –, a cada iteração do algoritmo, dando origem ao pseudo-algoritmo 3.

3.5 Limitações do algoritmo

O algoritmo dá mais importância para as grandes flutuações em detrimento de variações mais suaves. O número mínimo de pontos necessários para representar razoavelmente bem uma série está diretamente ligado ao número de picos e vales que a série possui. Para séries com muitos picos e vales com alta amplitude, o algoritmo não consegue uma redução de dimensionalidade significativa.

Capítulo 4

Implementação do algoritmo

Neste capítulo são apresentadas as ferramentas e técnicas utilizadas para a implementação do algoritmo, testes de desempenho realizados, dificuldades encontradas e a evolução da implementação durante seu processo de desenvolvimento.

4.1 Clojure

Clojure é uma linguagem de propósitos gerais, dinâmica e funcional, lançada em 2007, por Rich Hickey. Por se tratar de um dialeto de Lisp[1], Clojure foi criada com o intuito de preservar o caráter funcional e poderoso de Lisp para resolver problemas atuais como concorrência e paralelismo numa roupagem mais moderna.

Como se trata de uma implementação do paradigma funcional, a computação dos dados é encarada como avaliações de funções matemáticas, que favorecem o fraco acoplamento entre módulos e incentiva o uso de recursões evitando efeitos colaterais, uma vez que dados são considerados imutáveis nessa linguagem.

Para este trabalho foi utilizada a última implementação estável[2] da linguagem Clojure no momento da escrita deste texto.

4.2 Desenvolvimento

Preocupações como desempenho e corretude permearam o processo e técnicas de desenvolvimento ágil como TDD (Test Driven Development)[3] foram utilizadas durante sua

[1]Lisp (LISt Processing) teve sua primeira especificação em 1958 e foi originalmente pensada como uma notação matemática prática para programas de computador, influenciada pelo Cálculo-λ de Alonzo Church. A linguagem ainda é uma das linguagens mais antigas ainda utilizadas largamente no momento da escrita deste texto.

[2]Foi utilizada a versão 1.2, disponível em http://clojure.org/downloads. Último acesso em 20 de novembro de 2010.

[3]Test Driven Development consiste em um processo de desenvolvimento de software em que testes são escritos antes do desenvolvimento do código em si, facilitando o entendimento e condução do processo de raciocínio sobre o problema, além de contribuir como documentação do código.

execução.

Todas as funções criadas acompanham testes unitários. Tais testes não garantem totalmente que o código é correto, mas oferecem um meio de assegurar que alguns casos previamente pensados em que o algoritmo poderia falhar estão cobertos e uma garantia maior pode ser dada para futuros usuários e mantenedores do código.

Nenhuma particularidade do sistema em que foi incorporado este algoritmo – descrito no capítulo 5 – foi incluída na implementação. Deste modo, o mesmo código pode ser encarado como uma caixa preta e pode ser incorporado em outro sistema com facilidade. Como a linguagem Clojure é compilada para a JVM (Java Virtual Machine), o código também pode ser executado na maioria dos sistemas operacionais existentes atualmente. O potencial de execução em uma JVM e o fato de ser uma linguagem funcional – favorecendo o fraco acoplamento –, foram os fatores decisivos para a escolha da linguagem na execução do projeto, além de ter facilitado a implantação no software descrito no capítulo 5, escrito originalmente em Java.

4.3 Desempenho

Apesar do algoritmo apresentado no capítulo 3 não representar muitas dificuldades de entendimento, implementá-lo de uma forma eficiente requereu pesquisa sobre as estruturas de dados adequadas e detalhes da linguagem para a realização da tarefa.

A implementação do código passou por diversas modificações para atender aos requisitos de desempenho exigidos pelo projeto. Foram desenvolvidas três versões – incluídas no Apêndice – e suas evoluções serão brevemente descritas na próxima seção.

4.3.1 Otimizações

Versão 1

A primeira versão do algoritmo – anexo A – foi desenvolvida sem grandes otimizações. Não foi utilizado nenhum mecanismo de cache de resultados intermediários, nem testes muito robustos.

Nesta versão foram definidas as estruturas de dados a serem utilizadas. Para a lista de entrada, se preferiu utilizar *vectors*, que possuem complexidade $O(1)$ para acesso aleatório. Durante o processamento, mantém-se um *vector* que cresce a cada iteração, com os índices na série original para os PIPS escolhidos. Como durante o processamento, são armazenados apenas os índices – ao invés da combinação (x, y) de cada ponto – ao se calcular as distâncias são necessários acessos aleatórios a série original para se buscar os pontos. A cada iteração, o *vector* de PIPs é ordenado.

Como os tempos de execução apresentados ficaram bem aquém do desejado, a implantação em um software real – um dos objetivos pretendidos com esse trabalho, apresentados

na seção 1.2 – se tornou impeditiva. Foi necessário, portanto, o desenvolvimento de uma versão de melhor desempenho.

Versão 2

Para a segunda versão foram implementadas algumas melhorias. A mais importante de todas se deveu a percepção de que cálculos de distância eram repetidos desnecessariamente. Foi buscada, deste modo, uma solução de cacheamento dos resultados e para tanto, utilizou-se a função *memoize* da linguagem Clojure, que contribuiu para a melhora significativa no tempo de execução.

A cada iteração são armazenados os intervalos entre pips em um *hashset*. A vantagem de se armazenar os intervalos é que os mesmos podem ser avaliados em paralelo – dependendo de uma sincronização no término para se analisar a melhor distância – apesar de não ter se utilizado nenhuma espécie de paralelismo nessa versão. Com essa alteração, a necessidade de se ordenar a lista de PIPs a cada iteração passou a inexistir.

A escolha de um *hashset* se deve a dois motivos. O primeiro deles é que se pode garantir que se a entrada possuir pontos duplicados, estes não aparecerão na saída do algoritmo. O segundo motivo é que como os intervalos são divididos a cada iteração, precisamos redefinir a lista de intervalos, retirando o intervalo anterior, criando dois novos intervalos, no ponto em que o novo PIP foi encontrado. Acessos aleatórios em *hashsets* ocorrem em tempo $O(log_{32}n)$, que na prática pode ser considerado constante.

Como são processadas listas muito extensas e sua avaliação pode ser custosa, optou-se também nesta versão por se utilizar *lazy sequences*[4]. Como muitos programas escritos em Clojure envolvem computação de listas, a linguagem fornece esse mecanismo para se construir expressões complexas e extensas que só são avaliadas quando realmente são necessárias. Listas infinitas podem ser definidas com esse recurso.

Outra melhoria surgida nessa versão se deu ainda com relação ao cálculo de distâncias. Como são efeituados muitos cálculos de distâncias, foi buscada nessa versão uma implementação mais eficiente. Para isso, substituiu-se o uso da função *abs* do pacote `clojure.contrib.generic.math-functions` pela versão de maior desempenho da função *abs*, do pacote `clojure.contrib.math`.

Como Clojure é uma linguagem construída por cima da linguagem Java, casting para tipos primitivos para esta linguagem – discutido em [14], onde podem ser encontrados outras técnicas para se aumentar o desempenho de programas em Clojure – também foram adicionados.

[4]Para maiores detalhes sobre o conceito de *lazy sequences*, veja [13]

Versão 3

A versão 3, última desenvolvida, é a que oferece o menor tempo de execução dentre todas as anteriores. Ao invés de definirmos nossas *lazy sequences*, foi preferida a utilização da função *iterate*, disponível no *core* da linguagem. A função *iterate* recebe como parâmetro uma função e um parâmetro adicional. No momento de sua chamada, a função passada como argumento é executada com o argumento adicional e a sua saída é aplicada a mesma função. O comportamento pode ser continuado indefinidamente. Isso evita muitas trocas de contexto e avaliações de listas muito extensas previamente.

Fez-se uso de mais *lazy sequences* onde necessário e ganhos de desempenho consideráveis foram conseguidos quando casting para tipos primitivos foram acrescentados no código.

Testes de desempenho

Para comparar a evolução das implementações foram executadas as três versões em ambiente controlado para séries aleatórias e comparados seus tempos de execução.

A máquina utilizada possui as seguintes configurações:

- Processador: Intel Celeron processor 560, 2.13 GHz, 1 MiB L2 cache
- Memória Principal: 1 GiB
- Sistema Operacional: Ubuntu Linux versão 10.04

Na tabela 4.1 é possível ver um resumo dos tempos de execução para testes realizados com séries aleatórias – para não privilegiar nenhuma das versões – de 100 pontos e 1000 pontos. Foram executadas 500 rodadas para cada valor de PIPs e tirada sua média para atender a Lei dos Grandes Números e garantir uma baixa variância nos resultados. Pode-se observar que a Versão 1 destoa bastante das outras duas versões.

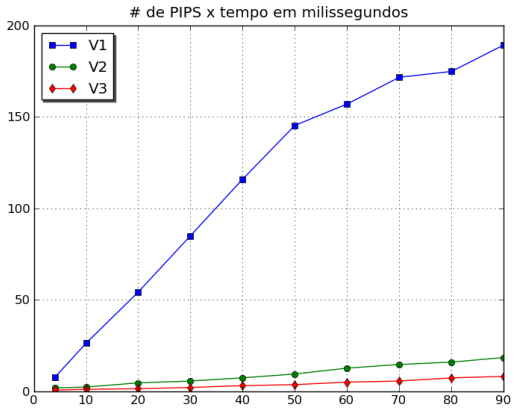
É possível observar também que a Versão 3 é duas vezes mais rápida que a Versão 2 analisando os valores da tabela 4.2. Um ganho considerável de desempenho quando séries com alta dimensionalidade devem ser processadas. Nas figuras 4.1a e 4.1b, podem ser vistos os mesmos dados de forma gráfica.

Tempos de execução			
Número de PIPs	Versão 1	Versão 2	Versão 3
4	7.49943	1.8605996	0.6156479
10	26.238653	2.321418	1.0887681
20	54.226994	4.5783596	1.3933954
30	84.99333	5.6005797	2.06694
40	115.92436	7.35721	3.0876982
50	145.32526	9.438611	3.6283286
60	156.99683	12.641098	4.9837728
70	171.61131	14.659222	5.6078625
80	174.81184	15.915979	7.315132
90	189.28699	18.447191	8.141004

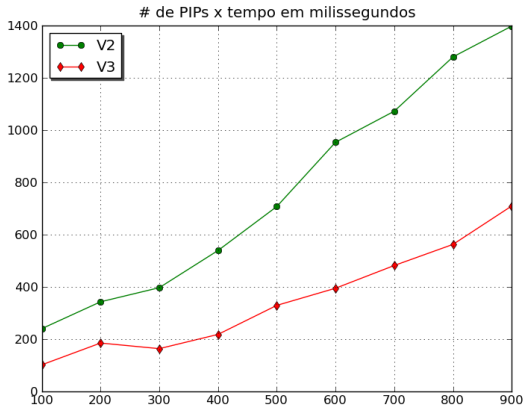
Tabela 4.1: Tempos de execução para as três versões implementadas do algoritmo PIP para séries com 100 pontos. Os tempos estão em milissegundos.

Tempos de execução		
Número de PIPs	Versão 2	Versão 3
100	239.5926	101.20084
200	342.2604	184.52263
300	396.32697	163.0423
400	538.5814	217.53302
500	706.6794	328.58374
600	952.6192	394.18314
700	1071.5186	481.18253
800	1279.3595	561.9523
900	1397.1849	708.9733

Tabela 4.2: Tempos de execução para as duas melhores versões implementadas do algoritmo PIP para séries com 1000 pontos. Os tempos estão em milissegundos.



(a) Comparação entre os tempos de execução das três versões implementadas. Série original possuía 100 pontos.



(b) Comparação entre os tempos de execução das duas melhores versões implementadas. Série original possuía 1000 pontos.

Figura 4.1: Tempos de execução para as implementações do algoritmo PIP

Capítulo 5

Aplicação em um sistema real

O objetivo da implementação do algoritmo apresentado no capítulo 4 foi permitir sua implantação no sistema de monitoração de datacenters HOLMES, brevemente descrito neste capítulo.

A integração com o algoritmo PIP e a arquitetura do software envolvida também são discutidas. Ademais, o resultado final é apresentado e em seguida um caso de uso de sucesso é apresentado.

5.1 HOLMES

O dia a dia da equipe de operação de um datacenter envolve entre outras atividades, o acompanhamento em tempo real de informações provenientes de diferentes máquinas e dispositivos. Centenas de métricas precisam ser monitoradas para garantir que o sistema não apresente quedas de serviço ou *downtime*, o que na maioria dos casos pode provocar perdas significativas de faturamento para o mantenedor do datacenter e seu negócio.

O software HOLMES foi criado com o propósito de fornecer aos operadores de datacenters ferramentas para análise e monitoramento de informações em tempo real, favorecendo o aumento de sua consciência situacional, fator crítico em um ambiente em que muitos equipamentos precisam ser monitorados e a correlação entre eventos para a resolução de um dado problema pode ser difícil, mesmo para operadores experientes.

5.2 Integração HOLMES e PIP

Dentre outras funcionalidades, o software possui uma interface de visualização de gráficos. Nessa interface é possível inspecionar gráficos de métricas coletadas ao longo do tempo provenientes dos itens de configuração cadastrados pelos usuários do sistema.

O termo item de configuração é originário da terminologia ITIL[1] e pode representar

[1]ITIL, Information Technology Infrastructure Library.

um dispositivo físico como roteador, disco rígido, unidade de processamento, ou qualquer outra métrica lógica como protocolo TCP/IP, HTTP, etc.

Itens de configuração são, em geral, cadastrados no sistema pela equipe de operadores do datacenter. Uma vez cadastrados e configurados, os itens de configuração passam a emitir eventos para um barramento central, de onde são coletados e armazenados. Com base nesses eventos, podemos construir suas séries temporais em sua forma mais bruta.

Devido a propriedade de alta dimensionalidade, essas séries requerem um processamento adicional antes de serem entregues efetivamente para o usuário final, na interface de visualização. É nesta fase que o algoritmo PIP é empregado, com o papel de reduzir a dimensionalidade dos dados originais e oferecer uma visualização que preserva o comportamento e a forma do gráfico da série original.

5.2.1 Arquitetura

O software de monitoração HOLMES possui um barramento central – módulo Message broker na figura 5.1 – destino dos eventos provenientes de diversos equipamentos. Esses dados são coletados e enviados para um módulo de armazenamento denominado *storage*, ilustrado na figura 5.1.

Na figura 5.1 é demonstrada a arquitetura geral do sistema HOLMES, com destaque para os módulos envolvidos na integração.

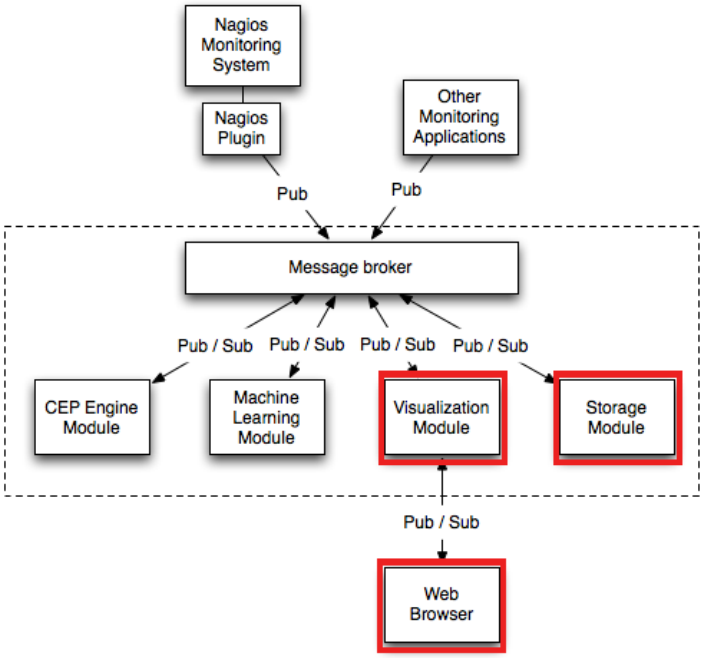


Figura 5.1: Arquitetura do sistema HOLMES e módulos participantes da integração com o algoritmo PIP

Quando um usuário final requisita uma série temporal através da interface web do sis-

tema HOLMES, o módulo storage é ativado. Os dados originais são coletados e entregues para processamento pelo algoritmo PIP, devolvidos em seu término[2] para o web browser do cliente.

A partir dos pontos perceptivelmente mais importantes no web browser do cliente, seus pontos são entregues finalmente a uma biblioteca Javascript para a plotagem final. Essa estratégia evita que pela rede sejam trafegados todos os pontos originais das séries temporais requisitadas.

Além disso, ainda que não existissem problemas de banda no tráfego de todos os pontos, outra dificuldade surgiria na etapa de plotagem dos gráficos, no que diz respeito a densidade de pontos. O algoritmo PIP resolve de forma satisfatória essas duas dificuldades.

5.2.2 Visualização

Para ilustrar o resultado final do que foi exposto até aqui, alguns screenshots foram tirados do sistema HOLMES, que é apenas um codinome para o sistema real: Intelie Event Manager.

As séries temporais que são exibidas para o usuário final estão organizadas na área Histórico do sistema. Na figura 5.2, é possível observar alguns gráficos utilizando o algoritmo PIP. Como o sistema é comercial, alguns nomes foram borrados para manter privacidade de seus clientes. Repare que segundo descrito na seção 3.5, o algoritmo PIP não se comporta muito bem com muitos picos e vales, mas ainda assim apresenta resultados satisfatórios quando o número de PIPs é suficiente, como pode ser observado na figura 5.2b.

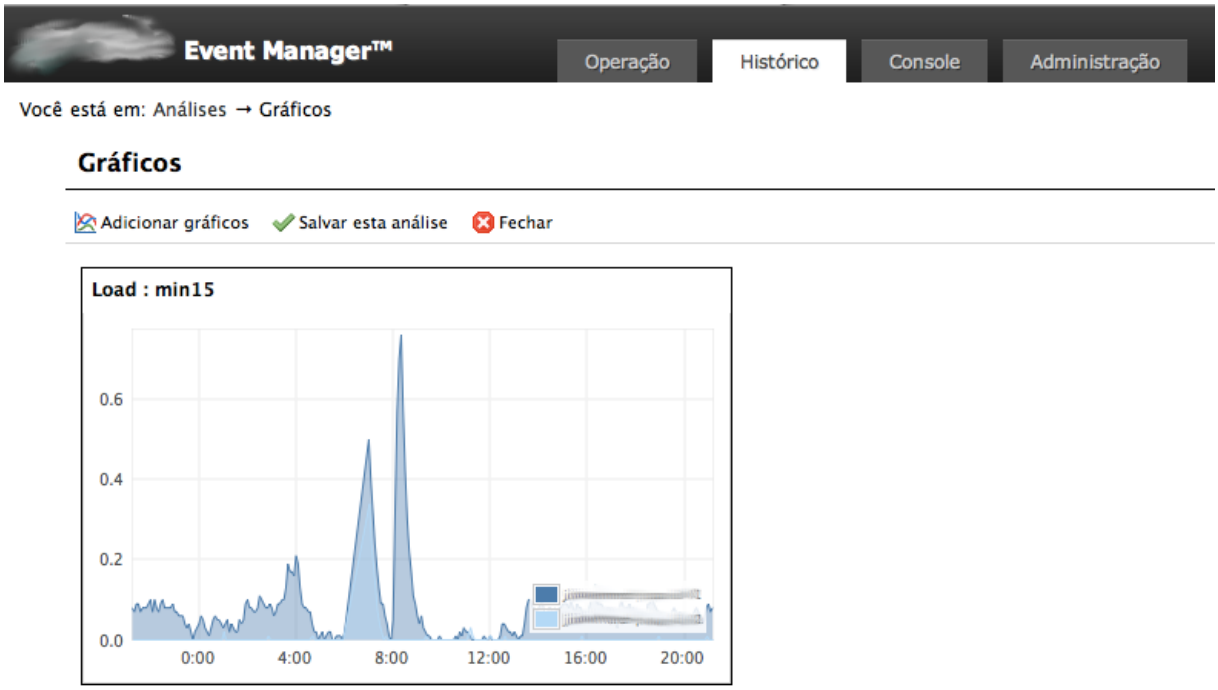
5.3 Caso de Uso

O sistema HOLMES está em uso atualmente em dois grandes datacenters do Brasil: Globo.com e iG.

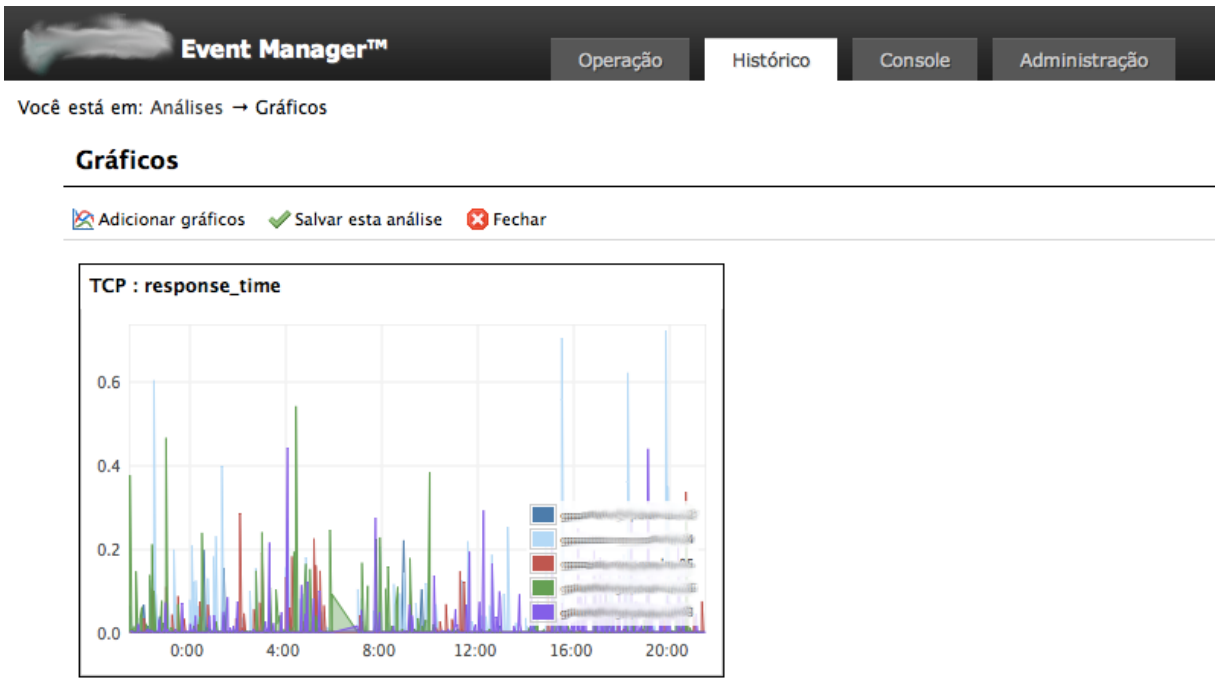
Versões anteriores do sistema HOLMES contavam com técnicas de média sobre média, brevemente discutidas na seção 2.3, que desfiguravam a forma geral do gráfico, confundindo usuários finais em sua interpretação dos dados.

Operadores destes datacenters manifestaram diminuição no tempo de detecção de falhas de problemas em equipamentos de seus servidores após a implementação da interface de visualização utilizando o algoritmo PIP.

[2]Os dados são devolvidos em formato JSON (JavaScript Object Notation), formato de serialização de dados largamente utilizado na Internet.

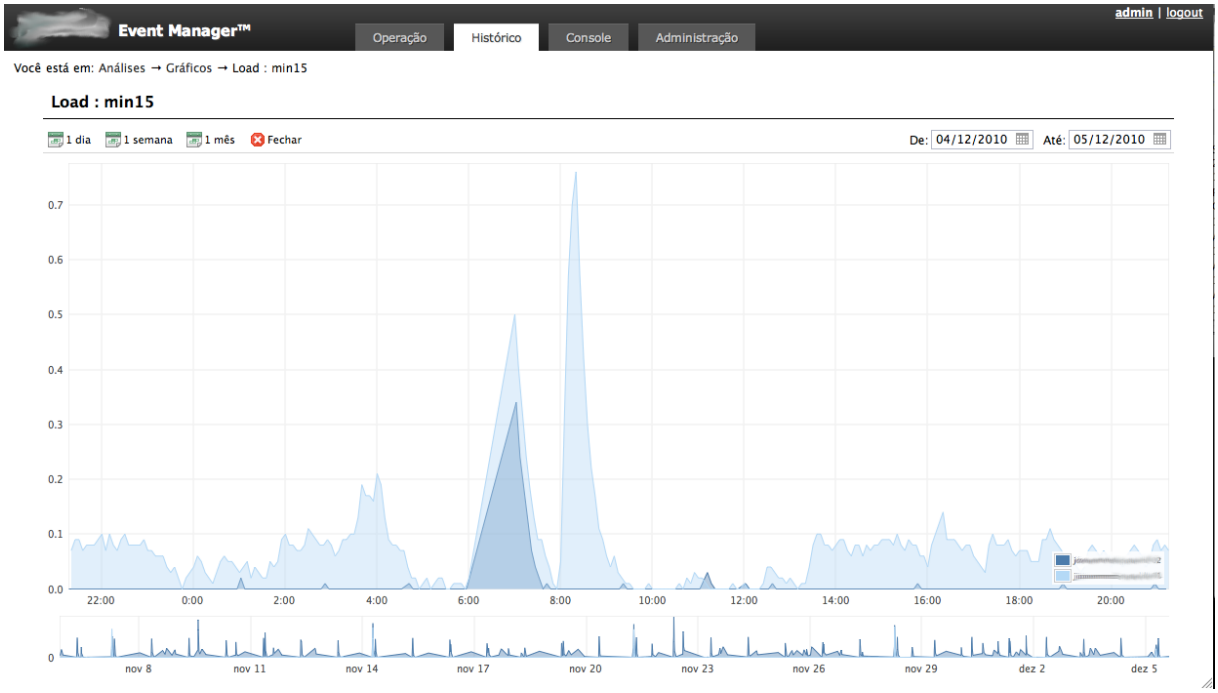


(a) Load mínimo nos últimos 15 minutos

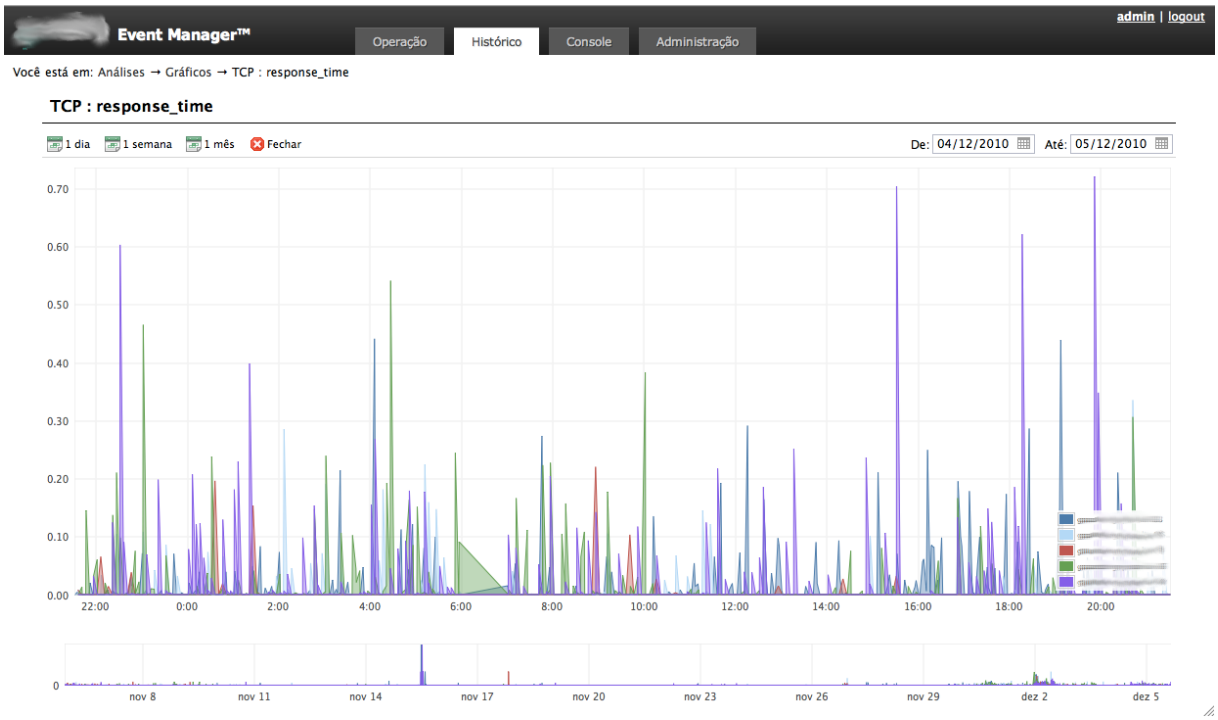


(b) Tempo de resposta do protocolo TCP

Figura 5.2: Gráficos gerados a partir do algoritmo PIP no sistema HOLMES



(a) Load mínimo nos últimos 15 minutos



(b) Tempo de resposta do protocolo TCP

Figura 5.3: Gráficos gerados a partir do algoritmo PIP no sistema HOLMES

Capítulo 6

Conclusão

Com o aumento da capacidade de processamento e armazenamento das máquinas seguindo a Lei de Moore, a possibilidade de se lidar com séries temporais cada vez maiores vem crescendo ao longo dos anos. A literatura existente sobre o assunto é extremamente vasta e rica, com aplicações nos mais variados campos do conhecimento humano.

O algoritmo escolhido para esse trabalho tem sido usado em séries temporais do mercado financeiro e aplicações no domínio de datacenters é nova até onde foi possível se investigar.

A relevância deste trabalho se manifesta a partir da aprovação por parte dos usuários do sistema HOLMES, com relatos confirmando a diminuição no tempo de identificação e solução de problemas em ambientes de operação de datacenters.

Apesar dos objetivos definidos para este trabalho – expostos na seção 1.2 – terem sido alcançados com êxito, muitas melhorias no software ainda se fazem necessárias; como um mecanismo para o cálculo de erro e escolha automatizada do número mínimo de pontos – descritos na seção 3.4 – para representar a série temporal original, não incluída na versão final da implementação devido o fator tempo.

Por fim, pode-se endereçar a trabalhos futuros, a integração com outros algoritmos de representação e redução de dimensionalidade na tentativa de se reduzir as limitações do algoritmo *Perceptually Important Points*, além de testes comparativos mais extensos no domínio de datacenters.

Apêndices

Apêndice A

Primeira Versão

```
1  ; Implementation of Perceptually Important Points algorithm
2  ; Author: Ronald Andreu Kaiser
3  ; First version
4  ; Last Updated: Sat 20 Nov 2010 05:04:05 PM BRST
5
6
7  (ns pip
8    (:require [clojure.contrib.generic.math-functions :as math]))
9
10
11 (defn vertical-distance
12   [[x1 y1] [x3 y3] [x2 y2]]
13   (math/abs (- (+ y1 (* (- y2 y1) (/ (- x3 x1) (- x2  x1)))) y3)))
14
15
16 (defn max-distance-in-interval
17   [original-serie first-pip-index last-pip-index]
18   (let [p1 (original-serie first-pip-index)
19         p2 (original-serie last-pip-index)]
20     (reduce
21       (fn [current-best candidate-point-index]
22         (let [current-distance (vertical-distance p1 (original-serie
               candidate-point-index) p2)]
23         (if (> current-distance (first current-best))
24           [current-distance candidate-point-index]
25           current-best)))
26     [-1 -1] (range (+ first-pip-index 1) last-pip-index))))
27
28
29 (defn next-pip-index
30   [original-serie current-pips]
31   (second (first
32       (reduce
```

```
33          (fn [[[last-best-distance last-best-index] left-pip-index]
                 right-pip-index]
34            (if (> (- right-pip-index left-pip-index) 1)
35                (let [best-current (max-distance-in-interval original-serie
                       left-pip-index right-pip-index)]
36                  (if (> (first best-current) last-best-distance)
37                    [best-current right-pip-index]
38                    [[last-best-distance last-best-index] right-pip-index]))
39                [[last-best-distance last-best-index] right-pip-index]))
40          [[-1 0] 0] (vec (rest current-pips))))))
41
42
43  (defn new-pip-seq
44    [original-serie current-pips]
45    (vec (sort (conj current-pips (next-pip-index original-serie current-pips
          )))))
46
47
48  (defn format-output
49    [original-serie pips]
50    (vec (map (fn [pip] (original-serie pip)) pips)))
51
52
53  (defn pip
54    [original-serie k]
55      (let [original-length (count original-serie)]
56      (if (or (< k 3) (< original-length 3) (>= k original-length))
           original-serie
57           (let [current-pips [0 (- original-length 1)]]
58           (loop [current-pips current-pips]
59             (if (< (count current-pips) k)
60               (recur (new-pip-seq original-serie current-pips))
61               (format-output original-serie current-pips))))))
```

Apêndice B

Segunda Versão

```
1  ; Implementation of Perceptually Important Points algorithm
2  ; Authors: Pedro Henriques dos Santos Teixeira & Ronald Andreu Kaiser
3  ; Second version
4  ; Last Updated: Sat 22 Nov 2010 05:04:05 PM BRST
5
6
7  (ns pip
8    (:require [clojure.contrib.math :as math]))
9
10
11 (defn max-key-index
12   "Returns [key, index] where (f (k item)) is the greatest. Most
13     applicable to vectors."
14   [f k & coll]
15   (apply max-key f (map #(vec [(k %1) %2]) coll (range (count coll)))))
16
17
18 (defn max-key-value
19   "Returns [key, value] where (f (k item)) is the greatest. Most applicable
       to vectors."
20   [f k & coll]
21   (apply max-key f (map #(vec [(k %) %]) coll)))
22
23
24 (defn max-distance
25   "Returns [distance, index of point] where distance is greatest. The
       extremes are excluded from candidates so the index returned is
       offseted by 1. pre: to > from + 1."
26   [series from to]
27   (apply max-key-index first
28     (fn [[x y]]
29       (let [[xi yi] (series from) [xf yf] (series to)]
```

```
30          (math/abs (- (+ (double yi) (* (- (double yf) (double yi)) (/ (- (
                double x) (double xi)) (- (double xf) (double xi))))) (double y)
                ))))
31      (subvec series (+ 1 from) to)))
32
33
34  (defn max-distance-over-intervals
35    "Find new candidate pip for each interval, and returns [[distance
          index-point-in-interval] interval] for the interval where distance of
          the pip is the greatest."
36    [series pips]
37    (let [candidate-intervals (filter (fn [[p1 p2]] (> p2 (+ 1 p1))) pips)]
38      (when-not (empty? candidate-intervals)
39        (apply max-key-value ffirst #(apply max-distance series %)
              candidate-intervals))))
40
41
42  (defn pip*
43    "Expect series as vector of points. pips is a set of [start end]
          intervals which are defined by the important points extracted so far.
44  Given the current intervals, this function chooses a new important point
        with max distance across all segments.
45  The new point is used to split one of the current segments."
46    [series pips]
47    (if-let [[[_ relative-index-pip] [p1 p2 :as interval-to-split]] (
          max-distance-over-intervals series pips)]
48      (let [index-new-pip (+ 1 p1 relative-index-pip)]
49        (conj (disj pips interval-to-split)
50          [p1 index-new-pip] [index-new-pip p2]))))
51
52
53  (defn pip-seq
54    "Lazy strategy using a recursive definition."
55    [series pips]
56    (if-let [new-pips (pip* series pips)]
57      (cons new-pips
58        (lazy-seq (pip-seq series new-pips)))))
59
60
61  (defn find-pips
62    "Returns lazy sequence of segments defined by the indices of PIPs. The
          extreme points are given as the first set of PIPs."
63    ([series]
64      (let [extremes [0 (- (count series) 1)]]
65        (conj (pip-seq series #{extremes}) #{extremes})))
66
67    ([k series]
```

```
68      (distinct (flatten (vec (nth (find-pips series) (- k 1)))))))
69
70
71  (defn pip
72    "Returns k points more importants from series."
73    [series k]
74    (binding [max-distance (memoize max-distance)]
75      (if (<= (count series) k) series
76        (sort-by first (map #(series %) (find-pips k series))))))
```

Apêndice C

Terceira Versão

```
1  ; Implementation of Perceptually Important Points Algorithm
2  ; Author: Ronald Andreu Kaiser
3  ; Third Version - Optimized
4  ; Last Updated: Mon 29 Nov 2010 01:01:45 AM BRST
5
6
7  (ns pip
8    (:use clojure.test)
9    (:require
10     [clojure.contrib.math :as math]
11     [clojure.contrib.generic.math-functions :as math-functions]))
12
13
14 (defn vertical-distance
15   "Returns the vertical distance between three points.
16    P1 [x1 y1] and P2 [x2 y2] represent the extreme left and right PIPs
          respectively.
17    P3 [x3 y3] is the candidate."
18   [[x1 y1] [x3 y3] [x2 y2]]
19   (math/abs
20     (- (+ (double y1)
21           (* (- (double y2) (double y1))
22             (/ (- (double x3) (double x1))
23               (- (double x2) (double x1)))))
24        (double y3))))
25
26
27 (defn max-in-interval
28   "Returns a vector with the best distance thorough the interval with the
29    corresponding index in the original-serie.
30    Assumes that there is at least one point between P1 and P2."
31   [original-serie [p1-index p2-index :as interval]]
32   (let [p1 (original-serie p1-index)
```

```
33          p2 (original-serie p2-index)]
34      (reduce
35        (fn [[last-best-distance last-best-index :as last-best]
              candidate-index]
36          (let [current-distance (vertical-distance p1 (original-serie
                candidate-index) p2)]
37            (if (> (double current-distance) (double last-best-distance))
38              [current-distance candidate-index]
39              last-best)))
40        [-1 -1]
41        (lazy-seq (range (inc p1-index) p2-index)))))
42
43
44  (defn find-max-from-intervals
45    "Returns the index of the point in original-serie that is most
46     distant and the subjacent interval."
47    [original-serie intervals]
48    (rest
49      (reduce
50        (fn [[last-best-distance last-best-index last-interval :as last-best]
              current-interval]
51          (let [current-best
52                (if (> (- (int (current-interval 1)) (int (current-interval
                    0))) 1)
53                  (max-in-interval original-serie current-interval)
54                  last-best)]
55            (if (> (double (current-best 0)) (double last-best-distance))
56              [(current-best 0) (current-best 1) current-interval]
57              last-best)))
58        [-1 -1 [-1 -1]]
59        intervals)))
60
61
62  (defn format-output
63    "Returns a vector, which is a transformation of the
64     current-intervals to points in the original-serie
65     representing the final PIPs."
66    [original-serie current-intervals]
67    (let [sorted-intervals (sort-by first (vec current-intervals))]
68    (reduce
69      (fn [last-interval current-interval]
70        (conj last-interval (original-serie (current-interval 1))))
71      [(original-serie (ffirst sorted-intervals))]
72      sorted-intervals)))
73
74
75  (defn pip
```

```
76    "Returns a vector with the best k PIPs from original-serie.
77     If k is greater or equal the length of original-serie,
78     returns the same original-serie."
79    [original-serie k]
80      (binding [max-in-interval (memoize max-in-interval)]
81      (let [original-length (int (count original-serie))
82            k (int k)]
83      (if (or (< k (int 3))
84          (< original-length (int 3))
85          (>= k original-length))
86            original-serie
87          (let [current-intervals #{[0 (- original-length 1)]}]
88            (format-output
89              original-serie
90              (nth
91                (iterate
92                  (fn [current-intervals]
93                    (let [new-pip-interval
94                            (find-max-from-intervals original-serie (lazy-seq
                                current-intervals))
95                          point-to-split (first new-pip-interval)
96                          interval-to-split (second new-pip-interval)]
97                      (conj (disj current-intervals interval-to-split)
98                        [(first interval-to-split) point-to-split]
99                        [point-to-split (second interval-to-split)])))
100                 current-intervals)
101               (int (- k 2)))))))))
102
103 ;=================================
104 ; find-max-from-intervals tests
105 ;=================================
106 (deftest find-max-from-intervals-test
107   (let [input [[[1 2] [3 4] [5 30] [6 5] [9 7]] #{[0 3] [3 4]}]
108         expected [2 [0 3]]
109         result (apply find-max-from-intervals input)]
110   (is (= expected result))))
111
112 ;=================================
113 ; max-in-interval tests
114 ;=================================
115
116 (deftest max-in-interval-simple-case
117   (let [input [[[1 2] [3 4] [5 30] [9 7]] [0 3]]
118         expected [25.5 2]
119         result (apply max-in-interval input)]
120   (is (math-functions/approx= (first expected) (first result) 0.00001))))
121
```

```
122 (deftest max-in-interval-simple-case-2
123   (let [input [[[1 2] [3 1] [5 30] [9 2]] [0 3]]
124         expected [28 2]
125         result (apply max-in-interval input)]
126   (is (math-functions/approx= (first expected) (first result) 0.00001))))
127
128 ;=================================
129 ; pip tests
130 ;=================================
131
132 (deftest pip-empty-input
133   (let [input []
134         expected []
135         k 2
136         result (pip input k)]
137   (is (= expected result))))
138
139 (deftest pip-one-observation-input
140   (let [input [[1 2]]
141         expected [[1 2]]
142         k 3
143         result (pip input k)]
144   (is (= expected result))))
145
146 (deftest pip-two-observations-input
147   (let [input [[1 2] [2 3]]
148         expected [[1 2] [2 3]]
149         k 3
150         result (pip input k)]
151   (is (= expected result))))
152
153 (deftest pip-simple-test-case
154   (let [input [[1 2] [2 3] [3 50] [4 10]]
155         expected [[1 2] [3 50] [4 10]]
156         k 3
157         result (pip input k)]
158   (is (= expected result))))
159
160 (deftest pip-required-more-points-than-series-length
161   (let [input [[1 2] [2 3] [3 50] [4 10]]
162         expected input
163         k 10
164         result (pip input k)]
165   (is (= expected result))))
166
167 (deftest pip-testing-if-pip-is-evaluating-intervals-correctly
168   (let [input [[0 1] [1 2] [2 4] [3 10] [4 20] [5 1] [6 3]
```

```
169                 [7 1] [8 10] [9 1] [10 10] [11 5] [12 30] [13 5]
170                 [14 1] [15  9] [16 13] [17 1] [18 0] [19 20] [20 1]
171                 [21 2] [22 4] [23 10] [24 5] [25 2] [26 50] [27 0]
172                 [28 7] [29 8] [30 5] [31 4] [32 3] [33 2] [34 20]
173                 [35 1] [36 3] [37 1] [38 10] [39 1] [40 10] [41 5]
174                 [42 30] [43 5] [44  1] [45 9] [46 13] [47 1] [48 0]
175                 [49 20] [50 1] [51 2] [52 4] [53 10] [54 20] [55 1]
176                 [56 3] [57 1] [58 10]  [59 1] [60 10] [61 5] [62 30]
177                 [63 5] [64 1] [65 9] [66 13] [67 1] [68 0] [69 20]
178                 [70 1] [71 2] [72 4] [73  10] [74 5] [75 2] [76 50]
179                 [77 0] [78 7] [79 8] [80 5] [81 4] [82 3] [83 2]
180                 [84 20] [85 1] [86 3] [87 1][88 10] [89 1] [90 10]
181                 [91 5] [92 30] [93 5] [94 1] [95 9] [96 13] [97 1]
182                 [98 0] [99 20]]
183         expected [[0 1] [12 30] [25 2] [26 50] [27 0]
184                   [42 30] [75 2] [76 50] [77 0] [99 20]]
185         k 10
186         result (pip input k)]
187   (is (= expected result))))
188
189 ;=================================
190 ; vertical-distance tests
191 ;=================================
192 (deftest vertical-distance-colinearity
193   (let [p1 [1, 2]
194         p3 [2, 3]
195         p2 [3, 4]
196         expected 0
197         result (vertical-distance p1 p3 p2)]
198   (is (math-functions/approx= expected result 0.00001))))
199
200 (deftest vertical-distance-positive
201   (let [p1 [1, 2]
202         p3 [2, 5]
203         p2 [3, 2]
204         expected 3
205         result (vertical-distance p1 p3 p2)]
206   (is (math-functions/approx= expected result 0.00001))))
207
208 (deftest vertical-distance-negative
209   (let [p1 [1, 2]
210         p3 [2, -1]
211         p2 [3, 2]
212         expected 3
213         result (vertical-distance p1 p3 p2)]
214   (is (math-functions/approx= expected result 0.00001))))
215
```

```
(deftest vertical-distance-float-positive
  (let [p1 [1, 2]
        p3 [2, 4.8]
        p2 [3, 5]
        expected 1.3
        result (vertical-distance p1 p3 p2)]
  (is (math-functions/approx= expected result 0.00001))))

(deftest vertical-distance-float-negative
  (let [p1 [1, 2]
        p3 [2, -10.0]
        p2 [3, 5]
        expected 13.5
        result (vertical-distance p1 p3 p2)]
  (is (math-functions/approx= expected result 0.00001))))


(run-tests)
```

Apêndice D

Aproximações

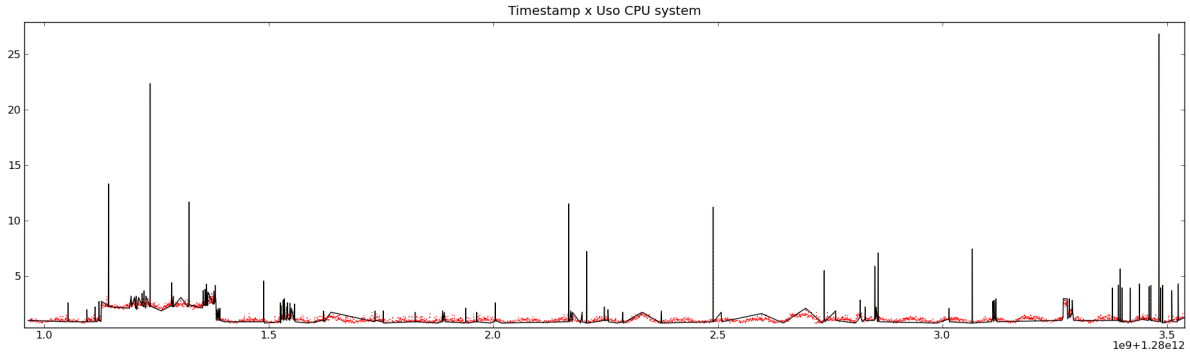


Figura D.1: Série original – linha pontilhada em vermelho – com 8582 pontos representando 1 mês de observações, reduzida com PIP para 300 pontos. Descarte de 99.976% dos pontos originais.

Referências Bibliográficas

[1] Rrdtool web site. http://www.mrtg.org/rrdtool/. Acesso em 7 de dezembro de 2010.

[2] Eduardo Massad, Reneé X. de Menezes, Paulo S. P. Silveira, and Neli R. S. Ortega. *Métodos Quantitativos em Medicina*, pages 365–387. Manole, 1 edition, 2004.

[3] C. W. J. Granger and Paul Newbold. *Forecasting Economic Time Series (Economic Theory and Mathematical Economics)*. Academic Press, 1977.

[4] Andreas Dietrich and Jens J. Krüger. Long-run sectoral development: Time-series evidence for the german economy. *Structural Change and Economic Dynamics*, 21(2):111–122, 2010.

[5] Wolfram alpha web site. http://wolframalpha.com. Acesso em 30 de novembro de 2010.

[6] Robert H. Shumway and David S. Stoffer. *Time Series Analysis and Its Applications: With R Examples (Springer Texts in Statistics)*. Springer, 2006.

[7] S. Allen Broughton and Kurt M. Bryan. *Discrete Fourier Analysis and Wavelets: Applications to Signal and Image Processing*. Wiley-Interscience, 2008.

[8] K. P. Soman, N. G. Resmi, and K. I. Ramachandran. *Insight Into Wavelets : From Theory to Practice*, pages 48–72. Prentice-Hall of India Pvt.Ltd, 2010.

[9] F.L. Chung, T.C. Fu, R. Luk, and V. Ng. Flexible time series pattern matching based on perceptually important points. *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1–7, 2001.

[10] C. S. Perng, H. Wang, S. R. Zhang, and D. S. Parker. Landmarks: a new model for similarity-based pattern querying in time series databases. In *Proceedings of 16th International Conference on Data Engineering*, pages 33–42, 2000.

[11] Eugene Fink and Kevin B. Pratt. Indexing of compressed time series. In *Data Mining in Time Series Databases*, pages 51–78. World Scientific, 2003.

[12] Tak-chung Fu, Fu-lai Chung, Robert Luk, and Chak-man Ng. Representing financial time series based on data point importance. *Engineering Applications of Artificial Intelligence*, 21(2):277–300, 2008.

[13] Stuart Halloway. *Programming Clojure (Pragmatic Programmers)*, page 125. Pragmatic Bookshelf, 2009.

[14] Luke VanderHart and Stuart Sierra. *Practical Clojure (Expert's Voice in Open Source)*, pages 189–198. Apress, 2010.