Nyara de Araújo Silva

CobMiner – *Mineração de Padrões Arborescentes com Restrições*

Uberlândia - MG

2007

Nyara de Araújo Silva

CobMiner – *Mineração de Padrões Arborescentes com Restrições*

Dissertação apresentada à Coordenação do Mestrado em Ciência da Computação da Universidade Federal de Uberlândia para a obtenção do título de Mestre em Ciência da Computação.

Orientadora:
Profa. Dra. Sandra de Amo

Mestrado em Ciência da Computação
Faculdade de Computação
Universidade Federal de Uberlândia

Uberlândia – MG

Novembro/2007

Dados Internacionais de Catalogação na Publicação (CIP)

S586c Silva, Nyara de Araújo, 1976 -
CobMiner - Mineração de Padrões Arborescentes com Restrições / Nyara de Araújo Silva. - 2007.
87 f.: il.

Orientadora: Sandra de Amo.
Dissertação (mestrado) – Universidade Federal de Uberlândia, Programa de Pós-Graduação em Ciência da Computação.
Inclui bibliografia.

1. Mineração de dados (Computação) - Teses. I. Amo, Sandra de. II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU: 681.3.07

Elaborado pelo Sistema de Bibliotecas da UFU / Setor de Catalogação e Classificação

Os abaixo assinados, por meio deste, certificam que leram e recomendaram para a Faculdade de Ciência da Computação da Universidade Federal o aceite da dissertação intitulada ***CobMiner* – Mineração de Padrões Arborescentes com Restrições** por **Nyara de Araújo Silva** como parte dos requisitos exigidos para obtenção do título de **Mestre em Ciência da Computação**.

Profa. Dra. Sandra de Amo
Orientador

Prof. Dr. Wagner Meira Jr
Universidade Federal de Minas Gerais UFMG/MG

Prof. Dr. Ilmerio Reis da Silva
Universidade Federal de Uberlândia UFU/MG

Dedico este trabalho a Deus, e às pessoas especiais que tanto amo e que foram o meu estímulo: Lindaura, José Ricardo, Mariana, Maiza, Carlos e Cláudia, por tudo que representam em minha vida.

Agradecimentos

Agradeço à Mohammed J. Zaki, pelo despreendimento em disponibilizar o código fonte de seus trabalhos na sua página, além do apoio sempre dispensado via email. Ao meu amigo Ronaldo, que foi meu braço direito e esquerdo nessas implementações. Agradeço também o apoio da Fabíola, que foi a peça chave do sucesso dos nossos testes. Meus sinceros agradecimentos à professora Sandra de Amo, que na qualidade de orientadora, soube me direcionar com dedicação e muito apoio, além da paciência e compreensão dispensados a mim nos momentos de atropelos. Aos grandes amigos que fiz durante essa jornada, que direta ou indiretamente, compartilharam comigo o trabalho, as alegrias e as tristezas. Em especial, agradeço o carinho da Crícia.

Finalmente, agradeço a Deus por estar sempre presente na minha vida.

"Há homens que lutam um dia e são bons.
Há outros que lutam um ano e são melhores.
Porém, há os que lutam toda a vida.
Esses são os imprescindíveis."
Bertolt Brecht

Resumo

Há muito trabalho em mineração de padrões com foco em estruturas de dados simples como itemsets ou sequência de itemsets. Entretanto, recentes aplicações utilizam dados mais complexos como componentes químicos, estruturas proteicas, rede social, XML e logs da Web, exigindo estruturas de dados mais sofisticadas (árvores ou grafos) para serem especificadas. Aqui, padrões de interesse não envolvem apenas valores de objetos frequentes (*labels*) que aparecem em árvores (ou grafos), mas também topologias específicas frequentes encontradas nessas estruturas. A mineração de padrões de árvores frequentes tem sido bastante estudada, com a motivação do crescente interesse e aplicabilidade em diferentes áreas (Web Mining, Bioinformática, etc.). Porém, os sistemas convencionais de mineração de árvores permitiam ao usuário apenas definir o suporte mínimo como mecanismo de filtro dos padrões a serem minerados. Após o processo de mineração, um árduo trabalho é necessário para filtrar os padrões de interesse dos usuários.

Nessa dissertação, propomos o algoritmo ***CobMiner***, ***Co****nstrained-****b****ased* ***Miner***, um algoritmo de mineração de padrões arborescentes, incorporando ao processo de mineração os *Autômatos de Árvores*, como mecanismo para restringir o escopo da mineração e produzir padrões frequentes mais próximos do real interesse dos usuários. Comparamos dois métodos de inclusão das restrições do usuário dentro do processo de descoberta: o primeiro é o CobMiner que incorpora o autômato de árvore dentro do mecanismo de mineração, o segundo é o TreeMinerPP que consiste do conhecido algoritmo de mineração de árvores, TreeMiner (ZAKI, 2002), seguido de uma fase de pós-processamento, onde os padrões são filtrados pelo autômato de árvore.

Um grande conjunto de testes foi executado em dados sintéticos e reais (documentos XML), o que nos permite concluir que utilizar as restrições *durante* a mineração é muito mais eficiente do que filtrar os padrões frequentes após o processo de mineração.

Abstract

Most work on pattern mining focus on simple data structures like itemsets or sequences of itemsets. However, a lot of recent applications dealing with complex data like chemical compounds, protein structure, social network, XML and Web Log databases, require much more sophisticated data structures (trees or graphs) for their specification. Here, interesting patterns involve not only frequent object values (labels) appearing in the trees (or graphs) but also frequent specific topologies found in these structures. Mining frequent tree patterns have been extensively studied, motivated by the increasing interest and applicability in different areas (Web Mining, Bioinformatics, etc). However, conventional tree mining systems normally consider only minimum support criterium as a mechanism for filtering patterns to be mined. After mining process, hard work is requiring to filter patterns concerned with user interests.

In this dissertation, we propose ***CobMiner***, ***Co****nstrained-****b****ased* ***Miner***, a tree pattern mining algorithm which incorporates tree automata into the mining process in order to restrict the mining scope and to generate frequent patterns more closely related to user interests. We compare two methods for introducing user constraints into the discovery process: the first one is CobMiner which incorporates tree automata constraints as an intra-mining mechanism, the second one is TreeMinerPP which consists of a well-known tree pattern mining algorithm, TreeMiner (ZAKI, 2002), followed by a post-processing phase, where patterns are filtered using a tree automatum.

An extensive set of experiments executed over synthetic and real data (XML documents) allow us to conclude that incorporating constraints *during* the mining process is far better effective than filtering the frequent and interesting patterns *after* the mining process.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

A mineração de padrões frequentes de dados estruturados ou semi-estruturados tem sido bastante estudada, motivada pelo crescente interesse e aplicabilidade em diferentes áreas, onde a estrutura dos dados desempenha um papel relevante nas técnicas de mineração utilizadas. Com o progresso tecnológico das redes de comunicação e das tecnologias de armazenamento, muitas informações têm sido disponibilizadas e trafegadas na internet. Esse conjunto de documentos HTML e XML são construídos de forma não rigidamente estruturada, o que caracteriza uma despadronização de suas informações. Dados que não obedecem a uma estrutura de formação rígida são conhecidos como semi-estruturados, e naturalmente representados por árvores.

Recentemente, observamos um crescente interesse no desenvolvimento de técnicas para descoberta de padrões arborescentes em bancos de dados de árvores (ZAKI, 2002; MIYAHARA et al., 2001). Citamos acima a representação dos documentos XML em árvores, porém existe uma diversidade de tipos de dados que também podem ser facilmente modelados através das estruturas de árvores, como os logs de uso da Web, estruturas de RNA em Bioinformática etc. Nesse cenário, notamos um enorme potencial de aplicabilidade das técnicas de mineração em árvores em diferentes contextos, o que justifica o foco e desenvolvimento crescente de trabalhos nessa área.

A mineração de padrões frequentes de hiperlinks permite descobrir similaridades ou relacionamentos entre diferentes sites da Web, e promover a classificação de páginas por categorias de assuntos. Um exemplo de aplicação seria descobrir sites cujos links aparecem frequentemente em outros sites para avaliar estratégia de propagandas na internet ou até mesmo para descobrir a idoneidade de um site (CHAKRABARTI, 2000; KLEINBERG, 1999).

Considerando o contexto da utilização da internet, a Mineração do Uso da Web consiste na utilização de técnicas de mineração de dados para descobrir padrões que representam comportamentos dos usuários quando navegam pelas páginas de um site. Essa técnica pode ser muito útil para entender melhor as necessidades dos usuários em relação

às aplicações e serviços disponibilizados no site (COOLEY, 2000). A necessidade de uma reorganização do site também pode ser identificada através da análise de regularidades na navegação de usuários (SRIKANT; YANG, 2001). Por exemplo, dado um conjunto de logs de acessos a um site de *e-commerce*, cujo gerente está interessado em descobrir com qual frequência seus usuários acessam primeiro o link *Produtos Eletrônicos*, voltam na navegação e acessam em seguida o link *Carros* e de lá acessam o link *Acessórios*, efetivando a compra do item *CD Player*. Esse padrão de navegação pode ser um alerta de que a estrutura do site pode ser aprimorada, facilitando a navegação dos usuários e, consequentemente alavancando vendas.

Outro contexto relevante da mineração na Web é a descoberta de conhecimento no contéudo dos documentos (VIANU, 2001). Atualmente, banco de dados e páginas web estão fortemente conectados principalmente com o crescente uso do XML (reconhecida pelo W3C como linguagem padrão utilizada na troca e tráfego de informações pela internet), o que faz da internet um vasto e rico repositório de dados em constante crescimento, onde informações valiosas podem ser descobertas através de técnicas de mineração. Muitos problemas na descoberta de conhecimento em repositórios XML têm sido investigados, sendo alguns deles: (1) *Descoberta de Estrutura de Documentos* (PAPAKONSTANTINOU; VIANU, 2000; MURATA et al., 2005): Existem documentos XML encontrados na Web que não têm seus dados organizados, ou seja, eles não possuem uma DTD. É importante descobrir DTD em comum em conjuntos de documentos XML com objetivo de otimizar o armazenamento e métodos de busca desses documentos e de seus conteúdos. (2) *Descoberta de Padrões Frequentes* (ASAI et al., 2002): Dado uma coleção de documentos XML (os quais naturalmente podem ser representados por árvores ou grafos), seria do interesse de alguns usuários descobrir quais sub-documentos (sub-árvores) frequentemente aparecem entre os documentos XML. Por exemplo, poderíamos descobrir que a maioria dos documentos dessa coleção possuem tags *autores* e *titulos* ou *filmes* e *diretores*; ou que livros de ficção científica são lidos com maior frequência durante as férias escolares. (3) *Descoberta de Padrões de Consulta Frequentes* (YANG; LEE; HSU, 2003): Características básicas de linguagens de consulta para documentos XML como XPath (CLARK; DEROSE, 1999) ou XQuery (CHAMBERLIN, 2003) são expressões de caminho regular e padrões de árvores com predicados selecionados, que especificam o relacionamento entre as estruturas arborescentes (hierarquia estrutural) dos documentos. O processo de consulta em documentos XML pode ser computacionalmente caro porque envolve navegações através de árvores, que podem possuir profundidades consideráveis. Assim, a descoberta e armazenamento em *cache* de consultas frequentemente executadas podem aprimorar a per-

formance do processo de consulta em documentos XML.

Na Bioinformática, a mineração em árvores também é bastante utilizada. As pesquisas genéticas nos últimos anos contribuíram para a geração de um enorme banco de dados biológicos de DNA, RNA, aminoácidos, proteínas, etc. Por exemplo, pesquisadores têm encontrado muitas estruturas arborescentes de RNA que, através das práticas de mineração e comparação com moléculas de RNA conhecidas, permitem a descoberta de informações relevantes para um melhor entendimento funcional dessas estruturas (SHAPIRO; ZHANG, 1990; CHEVALET; MICHOT, 1992; WANG; ZHAO, 2003).

Nesses quatro diferentes contextos em que técnicas de mineração de árvores foram desenvolvidas e aplicadas, a quantidade de padrões descobertos é enorme, exigindo custo elevado de processamento e área no processo de busca em padrões arborescentes, fazendo do processo de mineração em árvores um mecanismo muito caro.

Nesse trabalho estamos propondo uma melhoria na eficiência do processo de mineração em árvores, com a redução do escopo da mineração e geração de padrões mais direcionados aos interesses do usuário através da utilização de autômatos de árvore durante a mineração.

1.1 Motivação

Recentemente, muito trabalho tem sido dedicado ao desenvolvimento de métodos que permitem aos usuários um maior controle sobre os padrões produzidos por sistemas de mineração. Esse controle pode ser oferecido como uma fase de pós-processamento à mineração ou como uma funcionalidade a mais do próprio mecanismo de mineração, através da utilização de restrições especificadas pelos usuários. Essa última idéia tem sido bastante explorada em mineração de regras de associação (PADMANABHAN; TUZHILIN, 2000) e mineração de padrões sequenciais (GAROFALAKIS; RASTOGI; SHIM, 1999; AMO; FURTADO, 2005).

O principal objetivo dessa dissertação é introduzir um método de mineração de árvores baseado em restrições (AMO et al., 2007), permitindo aos usuários definir o formato dos padrões arborescentes que eles estão interessados em minerar. Essas restrições são incluídas no processo de mineração garantindo que apenas padrões que as satisfazem sejam produzidos, reduzindo bastante o custo de uma fase de pós-processamento para filtragem e consulta dos mesmos. Muitos algoritmos para minerações em árvore foram propostos recentemente com aplicações em *Mineração do Uso da Web* e *Mineração em Documentos*

XML (ZAKI, 2002; MIYAHARA et al., 2001). O algoritmo TreeMiner proposto em (ZAKI, 2002) é um dos mais eficientes. Diferente dos demais algoritmos para mineração de árvores e grafos, baseados na técnica de mineração do Apriori (MIYAHARA et al., 2001), ele usa a estratégia de busca em profundidade na fase de geração de candidatos e a idéia de armazenamento de informações relevantes para o cálculo do suporte. De acordo com essa idéia, as duas primeiras iterações do processo de mineração ocorre sobre o banco de dados de árvores, de onde são extraídas e armazenadas poucas informações, porém essenciais para o cálculo do suporte das iterações seguintes, evitando outras varreduras ao banco de dados, atividade esta de grande custo computacional.

Nessa dissertação, apresentamos o algoritmo CobMiner que utiliza o TreeMiner como técnica de mineração e incorpora as restrições dos usuários dentro da fase de geração dos candidatos, aumentando consideravelmente a eficiência do processo de mineração. O mecanismo que utilizamos para especificação das restrições dos usuários são os autômatos de árvores (NEVEN, 2002; MURATA et al., 2005), melhor detalhado e exemplificado nos capítulos seguintes.

1.2 Contribuições

As principais contribuições desse trabalho podem ser enumeradas como se seguem:

- Estamos propondo um mecanismo de especificação de restrições sobre padrões, aplicado ao contexto da mineração de padrões arborescentes, que oferece uma maior eficiência na mineração em árvores e maior facilidade de aplicação devido ao direcionamento da mineração, reduzindo consideravelmente o conjunto de informações produzidas, através da não geração de padrões desalinhados com as expectativas dos usuários.

- Projetamos e implementamos o algoritmo CobMiner para mineração de padrões de árvores com restrições, com aplicações em *XML Mining* e *Web Mining*.

- Efetuamos testes exaustivos sobre dados sintéticos gerados através de um gerador que construímos, que nos permite diversas variações de parâmetros para análise de performance e escalabilidade em diferentes cenários.

- Um sistema de mineração de dados reais foi construído, dados estes naturalmente estruturados em árvores como os documentos XML e logs de navegação na web,

cujo algoritmo de mineração é o CobMiner, algoritmo apresentado nessa dissertação (AMO; FELICIO, 2007).

1.3 Organização da Dissertação

A apresentação dessa dissertação está organizada da seguinte maneira: no capítulo 2 temos o estado da arte, onde descrevemos os principais trabalhos relacionados a algoritmos de mineração de árvores e a utilização de restrições no processo de mineração de sequências. Ainda neste capítulo, descrevemos os principais conceitos teóricos, existentes na literatura, que estão diretamente relacionados com o trabalho desenvolvido nesta dissertação e que são necessários para sua compreensão. Dentre estes conceitos figuram os autômatos de árvore e uma classe especial de tais autômatos, os chamados autômatos locais.

No capítulo 3 formalizamos o problema da mineração de padrões arborescentes com restrições de automâto de árvore, cujo estudo constitui a principal contribuição desta dissertação. Também neste capítulo, apresentamos os conceitos principais que desenvolvemos para fundamentar teoricamente o nosso método CobMiner de mineração de padrões arborescentes, descrevendo precisamente os conceitos que criamos.

Em seguida, apresentamos o algoritmo CobMiner no capítulo 4, e discutimos sobre os exaustivos testes efetuados em dados sintéticos, e um pequeno ensaio sobre um documento XML, no capítulo 5. Finalizamos essa dissertação com uma conclusão sobre o nosso trabalho e trabalhos futuros em vista.

2 O Estado da Arte

A descoberta de conhecimento atráves de técnicas de mineração está sendo cada vez mais utilizada. Um exemplo disto, bancos de dados ricos em informações comportamentais de clientes estão favorecendo a implantação de CRM (*Customer Relationship Management*) consistentes, através do qual grandes campanhas de vendas, direcionadas para perfis distintos de potenciais compradores, são lançadas no mercado, alavancando vendas.

A característica de compra de determinados itens é um dos aspectos considerados nessa análise. Por aqui pode-se identificar por exemplo produtos que induzem a compra de outros produtos ou até mesmo compras consecutivas de itens distintos em um curto espaço de tempo. Nesse contexto, a Mineração de Padrões Sequenciais é muito empregada e, extendendo o cenário de vendas para o mundo do *e-commerce*, onde a Mineração de Padrões Arborescentes é aplicada, comportamentos de clientes podem ser identificados pela maneira com que eles navegam pelos sites e efetuam suas compras. Aqui informações relevantes existem em arquivos de logs de sevidores Web, em repositórios de documentos XML, além das tabelas relacionais nos bancos de dados convencionais.

Nesse capítulo vamos rever os conceitos da mineração de sequências e dois trabalhos interessantes propostos nesse cenário, com o objetivo de trazer à mente como o uso de restrições pode influenciar positivamente o processo de mineração. Em seguida descrevemos alguns dos principais trabalhos relacionados à mineração de árvores, com maiores detalhes o algoritmo TreeMiner (ZAKI, 2002), mecanismo de mineração *core* do algoritmo que apresentamos nessa dissertação, o CobMiner.

2.1 Mineração de Sequências

Os algoritmos de mineração de sequências obedecem à estrutura de mineração baseada na propriedade Apriori descrita em (AGRAWAL; SRIKANT, 1994). Nesse esquema, o processo de mineração é executado em iterações, e cada iteração é dividida em 3 fases bem

definidas. A fase de geração é a primeira delas, onde um conjunto de padrões candidatos de tamanho k (C_k) é gerado a partir do conjunto de padrões frequentes L_{k-1} produzido pela execução da iteração anterior. Em seguida, o conjunto C_k é submetido à fase da poda. A finalidade dessa fase é eliminar padrões candidatos irrelevantes de C_k que possuem pelo menos um sub-padrão de tamanho $k-1$ que não existe em L_{k-1}. Na última fase de execução dessa iteração da mineração, os suportes dos padrões que permaneceram em C_k são calculados e apenas aqueles de suporte igual ou superior ao σ_{min} compõem o conjunto L_k. O algoritmo de mineração é executado até que, numa determinada iteração, nenhum padrão frequente seja gerado.

Em seguida, o problema da mineração de sequências será apresentado e estudos relevantes nessa área serão revisados para melhor contextualização do cenário de mineração de árvores com restrições, detalhado no próximo capítulo.

2.1.1 O Problema da Mineração de Padrões Sequenciais

O problema da mineração de sequências foi proposto em (AGRAWAL; SRIKANT, 1995). Nesse trabalho o algoritmo AprioriALL é a solução para a mineração de padrões sequenciais sobre um banco de dados de transações de clientes. Como cenário de exemplo, cada transação corresponde a uma compra de itens de um supermercado.

Para melhor entendimento, é importante revisarmos alguns conceitos importantes. Sendo assim, seja I um conjunto de itens e It um *itemset* tal que $It \subseteq I$. Uma **sequência** s, denotada por $s =< It_1\ It_2 ...\ It_n >$, é uma lista ordenada de *itemsets*, onde It_i é o i-ésimo elemento de s e $|s|$ é o seu tamanho. No nosso cenário de exemplo, um *itemset* corresponde a um conjunto de produtos comprados por um cliente em uma única compra, e as compras de um cliente ordenadas no tempo é uma sequência. Um banco de dados de sequências de clientes é um conjunto de tuplas $\{IdCli, SeqCli\}$, onde $IdCli$ é o identificador do cliente e $SeqCli$ é a sua sequência de compras.

Dizemos que q é uma **sub-sequência** de s se q é uma projeção de s, derivada pela exclusão de elementos e/ou itens de s. A sequência s contém a sequência q se q é uma sub-sequência de s. Exemplificando, suponha o conjunto de itens $I = \{a, e, i, o, u\}$. A sequência $s =< \{a, e\}\{a, e, i\}\{e, o\}\{u\} >$ é uma sequência de comprimento 8 e tamanho $|s| = 4$, ou seja, s é uma $4-$sequência, possui 4 *itemsets*. A sequência $q =< \{a\}\{e, i\}\{o\} >$ é uma sub-sequência de s, portanto dizemos que a sequência s *suporta* a sequência q. O **suporte** de um padrão sequencial q com relação a um banco de dados de sequência de clientes $\mathcal{D}$ é a porcentagem de sequências s de $\mathcal{D}$ que suportam q.

O problema da mineração de padrões sequenciais: dado um banco de dados de sequências de clientes $\mathcal{D}$ e o suporte mínimo σ_{min}, encontrar todas as sequências frequentes de $\mathcal{D}$, ou seja, sequências cujo suporte $\geq \sigma_{min}$.

Muitos estudos têm contribuído com o aprimoramento das técnicas de mineração de sequências. Em (SRIKANT; AGRAWAL, 1996) o algoritmo GSP é apresentado com performance bem superior ao algoritmo AprioriALL. Essa superioridade é justificada no fato de que o GSP possui uma poda mais eficiente, o que diminui consideravelmente o número de candidatos testados pelo suporte. A idéia fundamental que difere o GSP do algoritmo AprioriALL é o conceito de k-sequência. Considerando a segunda iteração do processo de mineração, enquanto no AprioriALL uma 2-sequência é uma sequência de 2 *itemsets*, no GSP uma 2-sequência pode conter um único *itemset* com 2 itens frequentes, ou dois *itemsets*, cada um contendo um item frequente. O algoritmo AprioriALL poda condidatos testando se as sub-sequências obtidas através da eliminação de um *itemset* inteiro são frequentes, enquanto que o algoritmo GSP é mais refinado, porque ele verifica a frequência de sub-sequências obtidas pela eliminação de um *item* apenas, favorecendo a poda de mais candidatos indesejáveis.

É importante enfatizar que os algoritmos AprioriALL e GSP mineram *todos* os padrões sequenciais frequentes de um banco de dados. Na seção seguinte descrevemos como outros tipos de restrições foram utilizados para direcionar a mineração para melhor satisfazer os interesses do usuário e aumentar ainda mais a eficiência do processo de mineração.

2.1.2 Mineração de Padrões Sequenciais com Restrições

Os algoritmos de mineração de sequências, modestamente descritos até aqui, permitem a interferência do usuário no processo de mineração apenas com a restrição do suporte mínimo, ou seja, eles não fornecem ao usuário mecanismos que lhe possibilite restringir os padrões a serem gerados conforme seu interesse. Nessa seção, vamos ver outros tipos de restrições interessantes que podem ser utilizados para minerar padrões que melhor atendem às expectativas do usuário.

2.1.2.1 Tipos de Restrições

Restrições são premissas definidas pelo usuário com a finalidade de direcionar o processo mineração que, ao considerá-las, gera padrões frequentes mais próximos às suas necessidades. Os tipos de restrições considerados no contexto da mineração de sequências

podem ser classificadas em *restrições de geração* e *restrições de validação*. As primeiras são restrições incorporadas à fase de geração de candidatos, visando a redução do espaço de busca dos padrões o que afeta positivamente na performance de execução dos algoritmos, sem comprometer o resultado final da mineração. Já as restrições de validação são condições verificadas durante a fase de validação dos candidatos, ou seja, os padrões são gerados normalmente e somente passam pelo cálculo do suporte aqueles que satisfazem a essas imposições do usuário.

Em (SRIKANT; AGRAWAL, 1996) encontramos o exemplo de restrições de validação de tempo *min-max*. Um padrão sequencial do tipo $< It_1\ It_2 >$ é considerado interessante apenas se as transações contendo It_1 e as transações contendo It_2 acontecerem dentro do intervalo de tempo definido pelo par de inteiros (*min*,*max*). Um exemplo de restrições de geração são aquelas expressas por expressões regulares, considerada nos algoritmos SPIRIT apresentados a seguir.

2.1.2.2 Os Algoritmos da Família SPIRIT

Os algoritmos da família SPIRIT foram propostos em (GAROFALAKIS; RASTOGI; SHIM, 1999) para mineração de padrões sequenciais com restrições de geração definidas através de expressões regulares. Considerando o nosso cenário de exemplo, imagine que o gerente do supermercado esteja interessado em minerar padrões sequenciais que indicam a compra do item *leite* num momento, e terminam numa outra transação onde a compra do item *bolo* foi efetuada. Neste caso, somente serão gerados padrões que satisfazem a expressão regular $\{leite\}a^*\{bolo\}$, onde a^* representa uma sequência qualquer de *itemsets*, que representam compras intermediárias que ocorreram entre a compra do item *leite* e a compra do *bolo*.

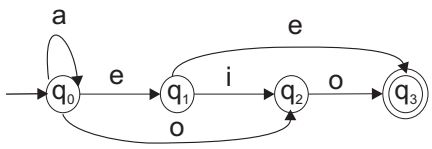


Figura 1: Autômato finito determinista associado a uma expressão regular.

A partir de uma expressão regular $\mathcal{R}$ sempre é possível construir um autômato finito determinista $\mathcal{A}_\mathcal{R}$ tal que $\mathcal{A}_\mathcal{R}$ aceita exatamente as palavras geradas por $\mathcal{R}$. Informalmente, um autômato finito determinista é uma máquina de estados finito com: um estado inicial q_0 e um ou mais estados aceitos bem definidos, e transições deterministas entre os estados dos símbolos de um dado alfabeto, no nosso caso, elementos de sequências que são itens comprados. A transição de um estado q_i para um estado q_j do elemento *bolo* é denotado

por $q_i \xrightarrow{bolo} q_j$. A sequência de transições dos elementos de uma sequência s, começando pelo estado q_i e terminando no estado q_j é representada por $q_i \overset{bolo}{\Longrightarrow} q_j$. Na Figura 1, temos o autômato finito determinista construído a partir da expressão regular $a^*(e\ e|e\ i\ o|o\ o)$, com estado inicial q_0, e final q_3 representado por um duplo círculo.

O problema da mineração de sequências com restrições: dado um banco de dados de sequências $\mathcal{D}$, o suporte mínimo σ_{min} e uma expressão regular $\mathcal{R}$ (ou o autômato equivalente $\mathcal{A}_\mathcal{R}$), encontrar todas as sequências frequentes em $\mathcal{D}$ que satisfazem $\mathcal{R}$.

Em linhas gerais, os algoritmos da família SPIRIT também são baseados na técnica de mineração Apriori. O conjunto de sequências candidatas C_k é obtido a partir de L_{k-1} porém somente candidatos que satisfazem a restrição $\mathcal{R}$ são gerados. A poda de um candidato não pode ser efetuada simplesmente eliminando as sequências de C_k que contém uma (k-1)-sequência não frequente. Porque aqui o conjunto L_{k-1} é composto por sequências *frequentes* e que *satisfazem a expressão regular* $\mathcal{R}$ e, a condição de satisfazer $\mathcal{R}$ não é *antimonotônica* como a restrição de ser frequente. Por exemplo, a sequência $< \{a\}\{b\}\{b\} >$ satisfaz a expressão regular ab^*, mas a sub-sequência $< \{b\}\{b\} >$ não a satisfaz. Entretanto, se uma sequência candidata $s \in C_k$ possui uma sub-sequência $s' \subset s$ que satisfaz $\mathcal{R}$ e $s' \notin L$, onde $L = L_1 \cup L_2 \cup ...L_{k-1}$, então neste caso s não é frequente e podemos podá-la.

As sequências de C_k que serão podadas podem ser representadas pelo conjunto P definido como: $P = \{s \in C_k \mid \exists s' \subset s,\ s' \text{ satisfaz } \mathcal{R} \text{ e } s' \notin L\}$. Observamos que quanto mais restritiva for a expressão regular $\mathcal{R}$, menor será o conjunto P, ou seja, menos sequências serão podadas. Assim, ao mesmo tempo que $\mathcal{R}$ restringe a geração de candidatos, ela também diminui a eficiência da fase da poda.

Buscando um equilíbrio na utilização de restrições, os algoritmos da família SPIRIT consideram relaxamentos para a restrição $\mathcal{R}$ com o objetivo de buscar eficiência tanto para a fase da geração quanto para a poda. Um relaxamento de uma restrição $\mathcal{R}$ corresponde a uma restrição *mais fraca* $\mathcal{R}'$ tal que toda sequência que satisfaz $\mathcal{R}$ também satisfaz $\mathcal{R}'$. Cada relaxamento $\mathcal{R}'$ de $\mathcal{R}$ corresponde a um algoritmo SPIRIT($\mathcal{R}'$), que considera a restrição $\mathcal{R}'$ ao invés de $\mathcal{R}$ no processo de mineração.

A família SPIRIT possui quatro algoritmos: SPIRIT(N), SPIRIT(L), SPIRIT(V) e SPIRIT($\mathcal{R}$), sendo o que difere eles o nível de relaxamento da restrição $\mathcal{R}$ e detalhes de tratamento dessa particularidade. O artigo (GAROFALAKIS; RASTOGI; SHIM, 1999) detalha cada um deles, verifica a performance experimentalmente, encontrando o relaxamento de

$\mathcal{R}$ mais apropriado. Numa visão simplista, a idéia principal de cada um deles é:

SPIRIT(N): esse algoritmo considera o maior de todos os relaxamentos de $\mathcal{R}$, ou seja, ele não impõe nenhuma restrição às sequências geradas. Seu mecanismo de mineração corresponde à mesma idéia do algoritmo GSP (SRIKANT; AGRAWAL, 1996) com uma simples modificação que se limita a exigir que todos os itens de uma sequência candidata apareçam na expressão regular $\mathcal{R}$.

SPIRIT(L): aqui somente são geradas sequências *legais com respeito a* algum estado do autômato $\mathcal{A}_{\mathcal{R}}$ associado à expressão regular $\mathcal{R}$. Dizemos que uma sequência $< a_1\ a_2\ ...\ a_n >$ é dita *legal com respeito a* um estado q de $\mathcal{A}_{\mathcal{R}}$ se existe um caminho em $\mathcal{A}_{\mathcal{R}}$ que começa no estado q e percorre a palavra $a_1, a_2, ..., a_n$. Considerando o autômato finito determinista da Figura 1, as sequências $< a\ e >$ e $< e\ i >$ são legais com respeito ao estado q_0.

SPIRIT(V): com esse nível de relaxamento, esse algoritmo gera sequências *válidas com respeito a* algum estado do autômato $\mathcal{A}_{\mathcal{R}}$. Dizemos que uma sequência $< a_1\ a_2\ ...\ a_n >$ é dita *válida com respeito a* um estado q de $\mathcal{A}_{\mathcal{R}}$ se existe um caminho em $\mathcal{A}_{\mathcal{R}}$ que começa no estado q e termina num estado final, percorrendo a palavra $a_1, a_2, ..., a_n$. Voltando ao autômato finito determinista da Figura 1, a sequência $< i\ o >$ é válida com respeito ao estado q_1.

SPIRIT($\mathcal{R}$): esse algoritmo não considera nenhum relaxamento da expressão regular $\mathcal{R}$, ou seja, $\mathcal{R}' \equiv \mathcal{R}$. Ele gera sequências *válidas* que são aquelas que são satisfeitas por $\mathcal{R}$, percorridas pelo autômato a partir do estado inicial até o estado final do autômato finito. A sequência $< e\ i\ o >$ é válida pelo autômato finito da Figura 1.

De uma maneira geral, esses quatro algoritmos realizam a mineração em duas etapas de processamento. Na primeira delas, etapa de relaxamento de $\mathcal{R}'$, os algoritmos encontram as sequências frequentes que satisfazem ao relaxamento $\mathcal{R}'$ da expressão regular $\mathcal{R}$. Na segunda etapa, etapa da restrição $\mathcal{R}$ também conhecido como etapa de pós-processamento da mineração, as sequências que não satisfazem a restrição original $\mathcal{R}$ são eliminadas do conjunto de sequências frequentes L. Observe que como o algoritmo SPIRIT(N) considera um relaxamento total de $\mathcal{R}$, ele não restringe a geração das sequências durante a primeira etapa de processamento, ficando a tarefa de validar em $\mathcal{R}$ todas as sequências frequentes no pós-processamento. Por outro lado, no algoritmo SPIRIT($\mathcal{R}$) não existe a etapa de pós-processamento, ou seja, todas as sequências frequentes que satisfazem $\mathcal{R}$ são obtidas na primeira etapa, durante a mineração propriamente dita.

Os quatro algoritmos foram exaustivamente testados. Na análise dos resultados ex-

perimentais apresentados em (GAROFALAKIS; RASTOGI; SHIM, 1999), concluímos que os algoritmos que incorporam a restrição $\mathcal{R}$ ou um relaxamento dela na primeira etapa de processamento são muito mais eficientes, e o algoritmo SPIRIT(V) é o que apresenta a melhor performance dentre os demais. Por essa razão, descrevemos a seguir detalhes da implementação do algoritmo SPIRIT(V).

Seja $\mathcal{A}_\mathcal{R}$ o autômato finito associado à expressão regular $\mathcal{R}$ especificada pelo usuário e $Q = \{q_0, q_1, ..., q_n\}$ o conjunto de estados de transição de $\mathcal{A}_\mathcal{R}$. Denotamos por $L'_k(q)$ o conjuntos das k-sequências frequentes e *válidas com respeito ao* estado q do autômato $\mathcal{A}_\mathcal{R}$, e $L'_k = L'_k(q_0) \cup L'_k(q_1) \cup ... \cup L'_k(q_n)$. A mesma denotação é aplicada também ao conjunto $C'_k(q)$.

Geração de Candidatos: Para que uma sequência candidata $s =< It_1\ It_2\ ...\ It_k >$ seja válida com respeito a algum estado q_x do autômato $\mathcal{A}_\mathcal{R}$, é preciso que: (1) seu sufixo $< It_2\ ...\ It_k >$ de tamanho $k-1$ seja frequente e válido com respeito a algum estado q_y de $\mathcal{A}_\mathcal{R}$ e; (2) exista uma transição $q_x \xrightarrow{It_1} q_y$ em $\mathcal{A}_\mathcal{R}$. Além disso, se queremos que $< It_1\ It_2\ ...\ It_k >$ tenha chance de ser frequente é preciso *ao menos que* o sufixo $< It_2\ ...\ It_k >$ seja frequente, ou seja, que o sufixo esteja em L'_{k-1}. Logo, o procedimento para calcular C'_k a partir de L'_{k-1} é o seguinte: para cada estado q do autômato $\mathcal{A}_\mathcal{R}$, para cada transição $q \xrightarrow{It_i} q'$ e cada sequência $< It_1\ It_2\ ...\ It_{k-1} >$ de $L'_{k-1}(q')$ é gerada a sequência $< It_i\ It_2\ ...\ It_k >$, construindo assim o conjunto $C'_k(q)$. O conjunto C'_k é a união de todos os $C'_k(q)$, para cada estado q de $\mathcal{A}_\mathcal{R}$.

Poda de Candidatos: Para cada sequência s de C'_k, o algoritmo SPIRIT(V) calcula todas as sub-sequências maximais de s que são válidas com relação a algum estado de $\mathcal{A}_\mathcal{R}$ e de tamanho inferior a k (sub-sequências maximais são aquelas que não estão contidas em nenhuma outra sub-sequência). Se uma das sub-sequências maximais válidas de s não está em L', então a sequência s é eliminada de C'_k.

Condição de Parada: O algoritmo SPIRIT(V) pára a sua execução quando em uma iteração k, nenhuma k-sequência frequente e válida com respeito a algum estado q de $\mathcal{A}_\mathcal{R}$ for encontrada, ou seja, $L'_k = \emptyset$.

Como exemplo da fase da geração de candidatos do algoritmo SPIRIT(V), considere o autômato finito $\mathcal{A}_\mathcal{R}$ da Figura 2 como sendo uma restrição imposta pelo usuário. Considere ainda um banco de dados de sequências $\mathcal{D}$ e o conjunto das 2-sequências frequentes e válidas com respeito aos estados de $\mathcal{A}_\mathcal{R}$, $L'_2 = L'_2(q_0) \cup L'_2(q_1) \cup L'_2(q_2) \cup L'_2(q_3)$, onde $L'_2(q_0) = \{< a, c >, < a, e >\}$, $L'_2(q_1) = \{< b, c >\}$, $L'_2(q_2) = \{< d, c >, < d, e >\}$ e $L'_2(q_3) = \emptyset$. Abaixo mostramos SPIRIT(V) calcula o conjunto dos candidatos válidos

com respeito a $\mathcal{A}_{\mathcal{R}}$, $C'_3 = C'_3(q_0) \cup C'_3(q_1) \cup C'_3(q_2) \cup C'_3(q_3)$, a partir de L'_2.

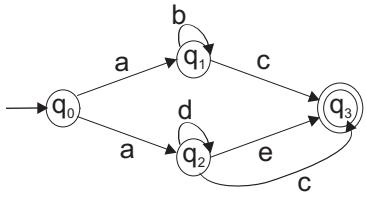


Figura 2: Autômato $\mathcal{A}_{\mathcal{R}}$ correspondente à restrição $\mathcal{R}$.

Considerando q_0: Temos duas transições partindo de q_0:

1. $q_0 \xrightarrow{a} q_1$: em $L'_2(q_1)$ temos apenas a sequência $< b, c >$, a partir da qual gera-se a 3-sequência $< a, b, c >$.

2. $q_0 \xrightarrow{a} q_2$: considerando agora $L'_2(q_2)$ temos a geração das 3-sequências $< a, d, c >$ e $< a, d, e >$.

O conjunto das sequências de tamanho 3 candidatas e válida com respeito ao estado q_0: $C'_3(q_0) = \{< a, b, c >, < a, d, c >, < a, d, e >\}$.

Considerando q_1: Também temos duas transições partindo de q_1:

1. $q_1 \xrightarrow{b} q_1$: a partir da sequência $< b, c >$ de $L'_2(q_1)$ geramos a 3-sequência $< b, b, c >$.

2. $q_1 \xrightarrow{c} q_3$: não há sequências em $L'_2(q_3)$, portanto nenhuma 3-sequência é gerada a partir dessa transição.

O conjunto das 3-sequências candidatas válida com respeito a q_1: $C'_3(q_1) = \{< b, b, c >\}$.

Considerando q_2: Transições partindo de q_2:

1. $q_2 \xrightarrow{d} q_2$: em $L'_2(q_2)$ temos as sequências $< d, c >$ e $< d, e >$, a partir das quais são geradas as 3-sequências $< d, d, c >$ e $< d, d, e >$.

2. $q_2 \xrightarrow{c} q_3$: como $L'_2(q_3) = \emptyset$, nenhuma 3-sequência é gerada.

3. $q_2 \xrightarrow{e} q_3$: como $L'_2(q_3) = \emptyset$, nenhuma 3-sequência é gerada.

Candidatas válida com respeito a q_2: $C'_3(q_2) = \{< d, d, c >, < d, d, e >\}$.

Considerando q_3: Não existe transições partindo de q_3. $C'_3(q_3) = \emptyset$.

Portanto, o conjunto de sequências candidatas de tamanho 3 com respeito ao autômato $\mathcal{A}_{\mathcal{R}}$ é $C'_3 = \{< a, b, c >, < a, d, c >, < a, d, e >, < b, b, c >, < d, d, c >, < d, d, e >\}$.

2.2 Mineração de Árvores

A mineração em árvores é um importante problema da área de mineração de dados. Muitos algoritmos para minerações de padrões arborescentes foram propostos recentemente com aplicações em *Mineração do Uso da Web* e *Mineração em Documentos XML*. Nessa seção vamos rever alguns dos principais trabalhos relacionados a esse problema, com especial atenção ao algoritmo TreeMiner (ZAKI, 2002), por ser ele o mecanismo de mineração *core* do algoritmo que apresentamos nesta dissertação, o CobMiner.

2.2.1 TreeMiner - Mineração Eficiente em Árvores

O algoritmo TreeMiner proposto em (ZAKI, 2002) é um dos mais eficientes. Diferente dos demais algoritmos para mineração de árvores e grafos, baseados na técnica de mineração do Apriori, ele usa a estratégia de busca em profundidade na fase de geração de candidatos e a idéia de armazenamento de informações relevantes para o cálculo do suporte. De acordo com essa idéia, as duas primeiras iterações do processo de mineração ocorrem sobre o banco de dados de árvores, de onde são extraídas e armazenadas poucas informações porém essenciais para o cálculo do suporte das iterações seguintes, evitando outras varreduras ao banco de dados, atividade esta de grande custo computacional. Detalhes desse mecanismo de mineração serão apresentados a sequir.

2.2.1.1 Definição do Problema

Uma árvore $T = (N, B)$ é um grafo conectado acíclico onde N é o conjunto de vértices (nós) e B o conjunto de arestas de T. Uma floresta é um grafo acíclico, que corresponde a uma coleção de árvores. Uma árvore possui raiz se um de seus vértices for distinguido dos demais. Dizemos que uma árvore é ordenada se os filhos de seus nós são ordenados, ou seja, se um nó tem k filhos, então nós podemos designá-los como primeiro filho, segundo filho, e assim por diante.

TreeMiner propõe descobrir todas as sub-árvores frequentes de uma coleção de árvores $\mathcal{D} = \{T_1, T_2..., T_n\}$ que possuem raiz, e cujos nós se encontram ordenados. Um nó de uma árvore é identificado por um número único n_i e um *label* associados a ele. Esse número representa a sua posição na profundidade da árvore e é através dele que o *label* do nó é encontrado. A árvore T_1 da Figura 3 é um exemplo da topologia das estruturas que compõem o banco de dados $\mathcal{D}$. O nó n_2 é o segundo nó encontrado durante a varredura

de T em profundidade, e o *label* a foi mapeado a ele através da função $l(n_2) = a$, onde $a \in L$: conjunto de *labels* que constituem as árvores de $\mathcal{D}$.

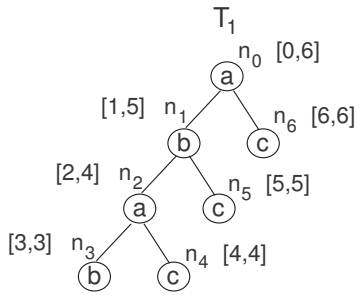


Figura 3: Um exemplo de árvore da coleção $\mathcal{D}$.

O *escopo* de um nó n_p de uma árvore T é definido pelo intervalo $[p, r]$ onde p e r são as localizações dos nós n_p e do seu nó mais a direita, n_r na profundidade de T. Essa noção de escopo possibilita a representação de uma árvore como uma lista de escopos de seus nós, estrutura esta utilizada pelo TreeMiner no cálculo da frequência de um padrão. Considerando a árvore T_1 da Figura 3, o escopo do nó n_2 é o intervalo $[2, 4]$, sendo n_4 o seu nó mais a direita.

As árvores do banco de dados são representadas por *strings* codificados pelos seus *labels* e pelo caracter -1. Numa visão geral, esse formato foi adotado por Zaki em (ZAKI, 2002) pela grande otimização de espaço de armazenamento, comparado com os métodos mais utilizados de representação de árvores (matriz ou lista de adjacência), além da facilidade de manipulação dessa estrutura. O *string* $\mathcal{T}$ = abab-1c-1-1c-1-1c-1 da Figura 3 foi obtido através do percurso em profundidade da árvore T. Cada caracter -1 representa um retorno do nó filho ao seu pai.

A eficiência do algoritmo está na mineração de padrões arborescentes indiretos, ou seja, que contenham ramos definidos não só pela relação pai-filho como também pela relação ancestral-descendente. Essa abordagem permite a descoberta de mais conhecimento relevante que não seria considerado na abordagem padrão. Na Figura 4 temos os padrões S_0 e S_1 definidas pela relação ancestral-descendente da árvore T_1 da Figura anterior. S_0 é chamada de 3-sub-árvore por ser uma sub-árvore de tamanho 3.

O suporte de uma sub-árvore candidata S corresponde ao número de árvores $T_i \in \mathcal{D}$ onde S ocorre direta ou indiretamente. Cada uma dessas ocorrências pode ser identificada através dos seus *match labels*, conjunto das posições encontradas em T_i para os nós de S. Considerando os padrões S_0 e S_1 mostrados na Figura 4, temos o conjunto dos *match labels* e o suporte calculados para cada padrão. Observe que apesar da sub-árvore S_0

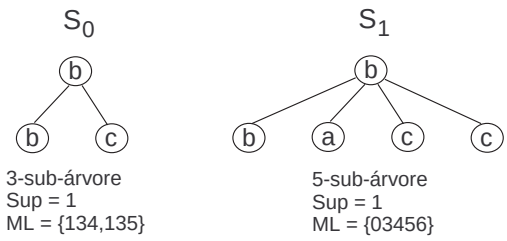


Figura 4: Sub-árvores de T_1. Exemplos de padrões a serem minerados.

ocorrer mais de uma vez em T_1, ou seja, ocorre nos nós n_1, n_3, n_4 e n_1, n_3, n_5 de T_1, no cálculo do suporte foi considerado apenas que a árvore T_1 suporta S_0, independente do quantas vezes S_0 está contida em T_1.

O problema da mineração de árvores do Algoritmo TreeMiner: dado uma coleção de árvores $\mathcal{D}$ cujos nós são ordenados, possuem *labels* e raiz definida, e o suporte mínimo σ_{min}, encontrar todas as sub-árvores frequentes de $\mathcal{D}$.

2.2.1.2 Geração das Sub-árvores Candidatas

O algoritmo TreeMiner possui uma fase de geração de candidatos baseada no método de expansão de classes. Considerando a propriedade anti-monotônica: *a frequência de um padrão é menor ou igual à frequência de um sub-padrão*, as k-sub-árvores candidatas são geradas a partir das (k-1)-sub-árvores frequentes. Esse método utiliza a estrutura *classe de equivalência* para representação das sub-árvores candidatas e mineradas em cada iteração, além de garantir a não redundância no processo de geração.

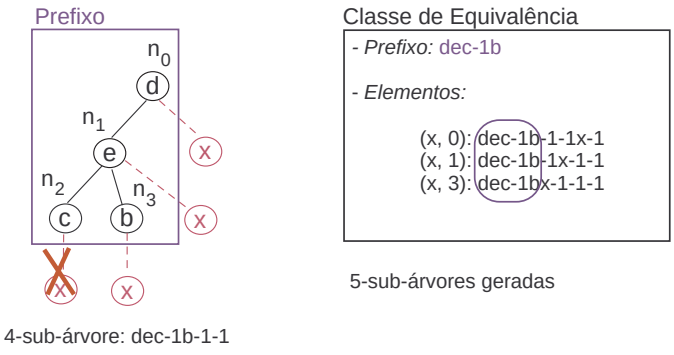


Figura 5: Uma classe de equivalência e seus elementos.

Definição (Padrões Equivalentes): Sejam T e S dois padrões arborescentes de tamanho k. Dizemos que T e S são *padrões equivalentes* se seus prefixos de tamanho $k-1$ são idênticos.

A Figura 5 mostra como o TreeMiner armazena as classes de equivalência. Consiste do *string* $\mathcal{P}$ de tamanho k-1 que representa o prefixo da classe e da sua lista de elementos. Cada elemento é uma k-sub-árvore e é representado pelo par (x, p), onde x corresponde ao *label* do último nó do padrão e p especifica a posição do nó em $\mathcal{P}$ onde x foi ligado. Por exemplo, o par $(x, 1)$ refere-se à sub-árvore gerada pela inclusão do nó x como filho do nó n_1. A classe de equivalência de prefixo $\mathcal{P}$ é denotada por $[P]_{k-1}$.

Ainda na Figura 5 temos uma classe de equivalência de sub-árvores de tamanho 5, que foram geradas a partir de um 4-padrão frequente que pertencia a outra classe de equivalência. Note que todos os elementos da classe possuem o mesmo prefixo $\mathcal{P}$ = dec-1b. Por essa razão, a sub-árvore formada com a inclusão do nó x como filho do nó n_2 não é gerada, pois o *string* desse padrão altera o prefixo da classe em formação. Essa sub-árvore não gerada pertencerá a outra classe de equivalência. As posições válidas onde o nó de *label* x pode ser incluído ao prefixo da nova classe são n_0, n_1 e n_3, assim em qualquer uma desses casos as sub-árvores obtidas pela inclusão de x a $\mathcal{P}$ têm o mesmo prefixo. Essas posições são identificadas conforme lema definido em (ZAKI, 2002): *Sejam $\mathcal{P}$ o prefixo de uma classe de sub-árvores e n_r o nó mais a direita de $\mathcal{P}$, cujo escopo é $[r, r]$. Seja $(x, i) \in [P]$, uma sub-árvore de $[P]$. O conjunto de posições válidas em $\mathcal{P}$ onde x pode ser incluído é dado por $\{i : n_i$ tem escopo $[i, r]\}$.*

Tendo uma classe de equivalência de k-sub-árvores, como obter as (k+1)-sub-árvores candidatas? O conjunto de elementos (x, p) de uma classe de equivalência é ordenado pelo *label* do nó como primeira chave e pela posição p como a segunda chave de ordenação. Sendo assim, dado uma lista de elementos ordenada, o processo de geração de candidatos através do método de expansão de classes produz novas classes de equivalências também com seus elementos ordenados, sem explicitamente submeter as listas de elementos a um processo de ordenação. A seguir descrevemos a idéia principal dessa proposta.

O Método de Expansão de Classes de Equivalência

O método de *expansão de classes* é definido em (ZAKI, 2002) pelo seguinte Teorema: Seja $\mathcal{P}$ o prefixo de uma classe de equivalência e (x, i), (y, j) dois elementos de $[P]$. Seja $[P_x]$ a classe de todas as extensões do elemento (x, i). Define-se o *operador junção* $\oplus$ sobre dois elementos, $(x, i) \oplus (y, j)$:

Caso 1 - $(i = j)$:

a) Se $\mathcal{P} \neq \emptyset$, adiciona-se (y, j) e (y, n_i) em $[P_x]$;

b) Se $\mathcal{P} = \emptyset$, adiciona-se $(y, j + 1)$ em $[P_x]$

Caso 2 - $(i > j)$: adiciona-se (y, j) em $[P_x]$

Caso 3 - $(i < j)$: nenhum candidato é gerado;

Então, todas as possíveis (k+1)-sub-árvores com prefixo de tamanho (k-1) $\mathcal{P}$, são encontradas aplicando o *operador junção* para cada par de elementos ordenados (x, i), (y, j).

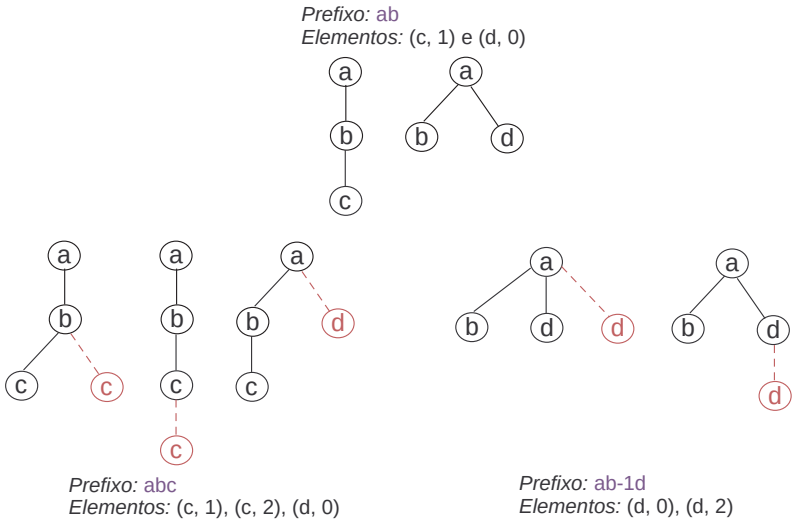


Figura 6: Geração de candidatos.

Considere a Figura 6 onde temos uma classe de equivalência com prefixo $\mathcal{P}_{ab}$ = ab, que contem os 2 elementos $(c, 1)$ e $(d, 0)$. As expansões de cada elemento de $[P_{ab}]$ darão origem a uma nova classe de equivalência. O primeiro passo é aplicar $(c, 1) \oplus (c, 1)$ que é direcionado ao caso 1a do teorema acima, produzindo os elementos candidatos $(c, 1)$ e $(c, 2)$, que correspondem à sub-árvore com a inclusão do nó c em n_1 e à sub-árvore com o nó c em n_2, ambas se encontram mais a esquerda da Figura 6. Em seguida, o elemento $(d, 0)$ é gerado através do caso 2 do teorema, pela junção $(c, 1) \oplus (d, 0)$, finalizando a expansão do elemento $(c, 1) \in [P_{ab}]$, que originou a nova classe $[P_{abc}]$, cujo prefixo $\mathcal{P}_{abc}$ = abc e seus elementos: $(c, 1)$, $(c, 2)$ e $(d, 0)$. A classe $[P_{ab-1d}]$ é criada pelo mesmo raciocínio, com todas as expansões de $(d, 0) \in [P_{ab}]$, geradas através das operações de junção $(d, 0) \oplus (c, 1)$ e $(d, 0) \oplus (d, 0)$.

Os padrões obtidos pelo método de expansão de classes, $(x_1, i_1), ..., (x_n, i_n) \in [P]$ são naturalmente ordenados pela ordem lexográfica dos pares (l, n). Por exemplo, os padrões da classe de equivalência $[P_{abc}]$, ilustrada na Figura 6, encontram-se ordenados de tal maneira, onde $(c, 1) < (c, 2) < (d, 0)$. A geração automática e ordenada de padrões do algoritmo TreeMiner é fundamentada em (ZAKI, 2002) pelo seguinte corolário: *Seja* $[P]_{k-1}$ *uma classe de equivalência com seus elementos ordenados da seguinte maneira:* $(x, i) < (y, j)$ *se* $x < y$ *ou* $(x = y$ *e* $i < j)$. *Então, o método de expansão da classe* $[P]_{k-1}$ *gera classes candidatas* $[P]_k$ *com elementos ordenados.*

O corolário descrito a seguir define a corretude do método de expansão de classes de equivalências (ZAKI, 2002): *O método de expansão de classes gera corretamente todas as possíveis sub-árvores candidatas, e cada candidata é gerada uma única vez.*

2.2.1.3 O Algoritmo TreeMiner

Nas duas primeiras iterações da execução, o algoritmo TreeMiner varre em profundidade as árvores do banco de dados em busca das sub-árvores frequentes. Nas demais iterações, o cálculo do suporte é feito sobre as estruturas *listas de escopos* e não sobre $\mathcal{D}$, tornando essa fase mais eficiente. Uma k-sub-árvore frequente X pode ser representada verticamente por uma lista de escopos. Cada elemento dessa lista é uma tupla (t, m, s), onde t é o identificador da árvore onde X ocorre, m é o *match label* que identifica o prefixo de tamanho k-1 de X na árvore t, e s corresponde ao nó mais a direita de X.

A Figura 7 mostra um banco de dados $\mathcal{D}$ com 3 árvores e a representação dessas árvores no formato horizontal de *strings* codificados. Essas árvores também podem ser representadas no formato vertical das lista de escopos das 1-sub-árvores de $\mathcal{D}$, mostradas na Figura 8. Observe que essas sub-árvores não possuem prefixo, ou seja, $m = \emptyset$. Denotamos a lista de escopos da sub-árvore a como sendo $\mathcal{L}$(a,-1).

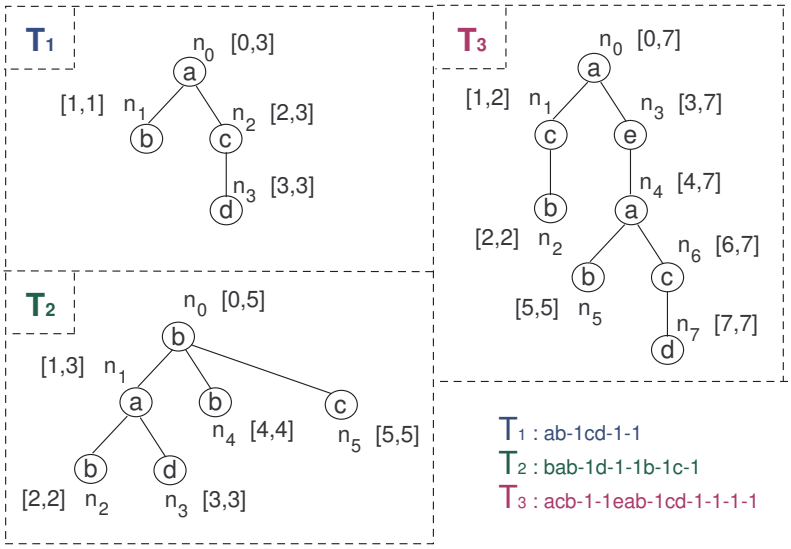


Figura 7: Banco de Dados $\mathcal{D}$ com 3 árvores.

Os principais passos da execução do TreeMiner incluem o cálculo dos itens frequentes e das 2-sub-árvores frequentes, e a procura por todas as demais sub-árvores frequentes via busca em profundidade dentro de cada classe $[P] \in F_2$, ou seja, geração recursiva de sub-árvores frequentes de tamanho k > 2. Abaixo descrevemos com mais detalhes

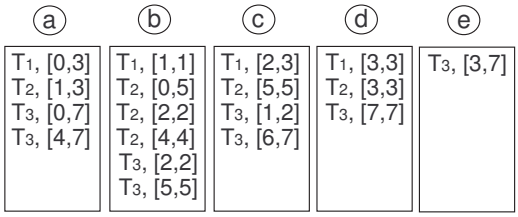


Figura 8: Representação de $\mathcal{D}$ através de listas de escopos.

cada passo de execução da mineração sobre o banco de dados $\mathcal{D}$ ilustrado na Figura 7, e $\sigma_{min} = 100\%$.

Cálculo de F_1

TreeMiner assume que as árvores de $\mathcal{D}$ estão no formato horizontal de *string* codificados, cujos *labels* pertencem ao conjunto $L = \{a, b, c, d, e\}$. Cada item $l_b \in L$ corresponde a uma 1-sub-árvore candidata. O suporte da sub-árvore candidata l_b é calculado através de um vetor 1 x L, que possui o contador de l_b incrementado na primeira ocorrência de l_b em cada árvore $T_i \in \mathcal{D}$. Na Figura 9 temos o vetor utilizado no cálculo do suporte das sub-árvores de C_1 geradas a partir de $\mathcal{D}$. Apesar da sub-árvore b ocorrer mais de uma vez em T_2, o contador de b para T_2 foi incrementado apenas 1 vez. Observe que apenas a 1-sub-árvore mais a direita não é suportada pelas 3 árvores de $\mathcal{D}$, portanto apenas ela não fará parte de F_1.

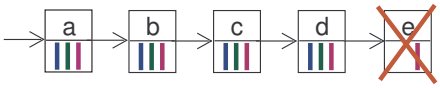


Figura 9: Cálculo do suporte das 1-sub-árvores candidatas.

Cálculo de F_2

Nessa iteração, também não existe uma fase de geração de candidatos. O cálculo do suporte de todos os candidatos é feito através de uma matriz de contadores M, de tamanho F_1 x F_1, onde cada $M[i][j]$ corresponde ao número de árvores de $\mathcal{D}$ que suportam a 2-sub-árvore candidata ij-1. Na Figura 10 temos a matriz M com o suporte dos candidatos de F_2 sobre $\mathcal{D}$. A sub-árvore ab-1 é frequente por ocorrer em todas as árvores de $\mathcal{D}$, conforme informação do contador $M[a][b]$. Ao final dessa iteração temos o conjunto de todas as 2-sub-árvores frequentes $F_2 = \{[P_a]\}$, onde $[P_a] = \{(b, 0), (d, 0)\}$. Enquanto o banco de dados é percorrido para calcular o suporte dos 2-padrões candidatos, o algoritmo TreeMiner cria as listas de escopos das sub-árvores de F_1.

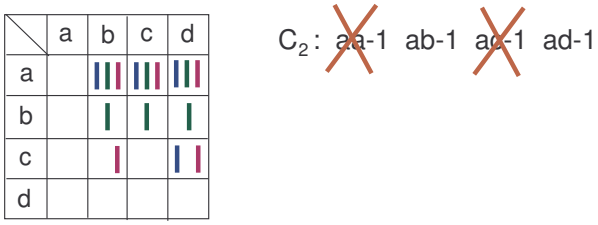


Figura 10: Cálculo do suporte das 2-sub-árvores candidatas.

A partir desse ponto, o banco de dados $\mathcal{D}$ não é mais acessado durante o restante da execução do TreeMiner. O cálculo do suporte dos k-padrões candidatos, com $k > 2$, é feito através da *Junção das Listas de Escopo*, também utilizada para construir as listas de escopo das sub-árvores frequentes de F_2.

Denotada por $\mathcal{L}(X) \cap_{\oplus} \mathcal{L}(Y)$, a junção das listas de escopos de duas sub-árvores X e Y de uma classe $[P]$, se baseia em operações entre intervalos da álgebra. Seja $s_x = [l_x, u_x]$ o escopo do nó x e $s_y = [l_y, u_y]$ o escopo de y. Dizemos que s_x *é estritamente menor que* s_y, denotado por $s_x < s_y$, see $u_x < l_y$, ou seja, não há sobreposição entre os intervalos s_x e s_y, e s_x ocorre antes de s_y. Dizemos que s_x *contém* s_y, denotado por $s_x \supset s_y$, see $l_x \leq l_y$ e $u_x \geq u_y$, ou seja, o intervalo s_y é um sub-intervalo de s_x.

Existem dois métodos de execução da junção, sendo o que direciona o algoritmo para o método adequado, é a verificação de como ocorreu a geração do padrão em questão. Recorrendo ao teorema de expansão de classes, onde $(x, i) \oplus (y, j)$ pode produzir até dois candidatos para a classe $[P_x]$. Sejam $(t_x, m_x, s_x) \in \mathcal{L}(x, i)$ e $(t_y, m_y, s_y) \in \mathcal{L}(y, j)$:

Teste In-Scope: Utilizado quando o padrão S for gerado pelo teorema como $(y, j+1)$ ou (y, n_i) em $[P_x]$, ou seja, através da adição de y como filho de x.
Se 1) $t_y = t_x = t$, isto é, ambas as tuplas ocorrem na mesma árvore $t \in \mathcal{D}$.
2) $m_y = m_x = m$, isto é, x e y são extensões sobre mesmo prefixo, com *match label* m.
3) $s_y \subset s_x$, isto é, y foi adicionado dentro do escopo de x.
Então, gera-se a tupla $(t_y, \{m_y \cup l_x\}, s_y)$ na lista de escopos de S, $\mathcal{L}(S)$.

Teste Out-Scope: Adequado para o padrão S gerado como (y, j) em $[P_x]$, ou seja, quando y é incluso como irmão do nó x.
Se 1) $t_y = t_x = t$
2) $m_y = m_x = m$
3) $s_y < s_x$; x aparece antes de y na busca em profundidade.
Então, gera-se a tupla $(t_y, \{m_y \cup l_x\}, s_y)$ na lista de escopos de S, $\mathcal{L}(S)$.

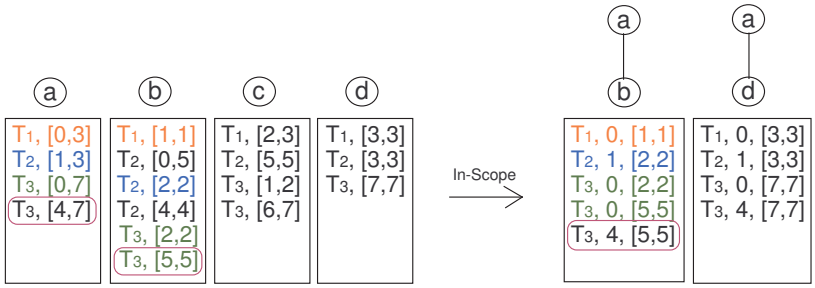


Figura 11: Construção da lista de escopos dos padrões frequentes de F_2.

Na Figura 11 temos ilustrado como as listas de escopos dos elementos de $[P_a] \in F_2$ foram construídas. Considerando que ambos foram gerados a partir da inclusão de b e d como descendentes de x, a tupla $(T_1, 0, [1,1])$ foi encontrada e incorporada à lista de escopos da sub-árvore ab-1, através da aplicação do teste *In-Scope* sobre as tuplas $(T_1, [0,3]) \in \mathcal{L}$(a,-1) e $(T_1, [1,1]) \in \mathcal{L}$(b,-1).

Cálculo de F_k, $k \geq 3$

A partir da iteração 3 do processo de mineração, o algoritmo TreeMiner gera os candidatos de tamanho k recursivamente. Na Figura 12 temos o pseudo-código da busca em profundidade das sub-árvores frequentes. Trata-se do procedimento *Enumerate-Frequent-Subtrees*, que recebe como parâmetro de entrada uma classe de equivalência $[P]$ junto com as listas de escopos de seus elementos. A expansão de cada elemento x_i de $[P]$ produz a nova classe $[P_x]$, que em seguida é enviada como parâmetro a uma nova chamada do procedimento *Enumerate-Frequent-Subtrees*. A recursão permanece até que um dos níveis da recursão não gere nenhum candidato. O próximo elemento $x_{i+1} \in [P]$ será expandido quando a pilha da recursão voltar até o k = tamanho das sub-árvores de $[P]$.

Observe que o pseudo-código da Figura 12 não explicita a fase da poda de candidatos antes do cálculo do suporte. Ao avaliar se um k-padrão candidato pode ser desprezado, verifica-se se seus (k-1)-sub-padrões são frequentes. Porém, considerando a geração recursiva, não sabemos se existem todas as informações necessárias que garantam a decisão da poda sem perda de informações, porque alguma classe de (k-1)-elementos pode não ter sido gerada ainda. O algoritmo TreeMiner usa o esquema da poda *oportunista*, que primeiro verifica se o (k-1)-sub-padrão já deveria ter sido encontrado. Caso essa informação se confirme, e o mesmo não esteja em F_{k-1}, então o k-padrão poderá ser podado com segurança. Essa verificação somente é possível pelo fato dos candidatos serem gerados ordenados. A poda é omitida da idéia central do algoritmo TreeMiner pelo fato de não

contribuir muito com a performance da execução do algoritmo, considerando que poucos padrões são desprezados por causa da geração recursiva.

Procedure TreeMiner($\mathcal{D}$,σ_{min})
  F_1 = {frequent 1-subtrees}
  F_2 = {classes $[P]$ of frequent 2-subtrees}
  For all $[P] \in F_2$ **Do** *Enumerate-Frequent-Subtrees*($[P]$)

Procedure Enumerate-Frequent-Subtrees($[P]$)
  For each $(x, i) \in [P]$ **Do**
    $[P_x] = \emptyset$
    For each $(y, j) \in [P]$ **Do**
      $R = \{(x, i) \oplus (y, j)\}$
      $\mathcal{L}(R) = \{\mathcal{L}(x) \cap_{\oplus} \mathcal{L}(y)\}$
      For each frequent $(z, w) \in R$ **Do**
        $[P_x].elem = [P_x].elem \cup (z, w)$
    Enumerate-Frequent-Subtrees($[P_x]$)

Figura 12: O Algoritmo TreeMiner

Continuando a descrever o processo de mineração do TreeMiner sobre o banco de dados $\mathcal{D}$ da Figura 7 e, considerando o término da execução da iteração 2 onde todas as 2-sub-árvores frequentes foram encontradas, o procedimento *Enumerate-Frequent-Subtrees* é chamado para a classe $[P_a] \in F_2$. Considerando a expansão do elemento (b,0) da classe $[P_a]$, encontramos os 3-candidatos que formam a nova classe de prefixo ab, $[P_{ab}]$ = {(b,0), (b,1),(d,0),(d,1)}. Os elemento (b,1) e (d,1), que correspondem respectivamente às árvores abb-1-1 e abd-1-1, são desprezados pela poda oportunista porque as (k-1)-sub-árvores bb-1 e bd-1 não são frequentes, ou seja, não existem em F_2.

A Figura 13 mostra o cálculo do suporte dos elementos (b,0) e (d,0) que permaneceram em $[P_{ab}]$ após a poda. A lista de escopos da 3-sub-árvore ab-1b-1 é calculada através da auto-junção das listas de escopo da 2-sub-árvore ab-1. Como o candidato foi gerado através da inclusão de um nó irmão, o teste *Out-Scope* é considerado, gerando apenas entradas de ocorrência da sub-árvore candidata para a árvore $T_3 \in \mathcal{D}$. A frequência da 3-sub-árvore candidata ab-1d-1 pode ser constatada pela sua lista de escopos criada conforme raciocínio descrito acima, onde encontramos pelo menos uma ocorrência do padrão em cada uma das árvores $T_i \in \mathcal{D}$.

Nesse instante $F_3 = \{[P_{ab}]\}$, $[P_{ab}] = \{(d,0)\}$. Antes de prosseguir com o cálculo de F_3 através das extensões das demais sub-árvores de F_2, ou seja, aplicar o teorema de expansão para o próximo elemento $(d, 0) \in [P_a]$, o procedimento *Enumerate-Frequent-Subtrees* é chamado para a classe $[P_{ab}]$, dando inicío ao cálculo de F_4, onde nenhuma 4-sub-árvore frequente é encontrada, voltando novamente a execução para a iteração 3.

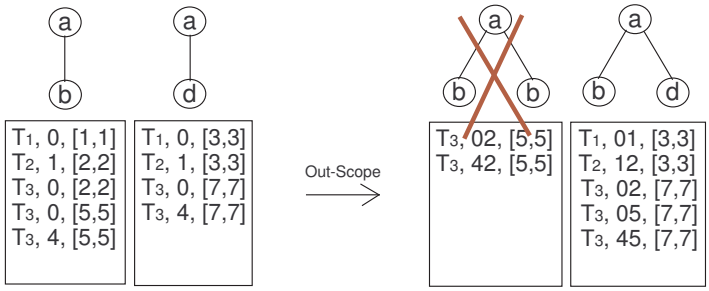


Figura 13: Construção da lista de escopos dos padrões frequentes de F_3.

Os 3-candidatos gerados pela expansão de $(d, 0) \in [P_a]$ não são frequentes, o que finaliza a execução da iteração 3 com apenas a classe $[P_{ab}]$ em F_3. O algoritmo TreeMiner encerra assim o processo de mineração sobre o banco de dados $\mathcal{D}$ e $\sigma_{min} = 100\%$, produzindo como sub-árvores frequentes: a, b, c, d, ab-1, ad-1 e ab-1d-1.

2.2.2 FREQT - Mineração em Dados Semi-estruturados

2.2.2.1 Definição do Problema

Esse algoritmo foi proposto em (ASAI et al., 2002) para extrair informações relevantes a partir de dados semi-estruturados. Nesse estudo, cada documento de dados semi-estruturado e os padrões minerados pelo algoritmo FREQT, são representados por árvores ordenadas e cujos nós possuem *labels*. A chave da idéia proposta é a noção da *expansão mais a direita*, técnica de geração de árvores através da inclusão de nós somente nos ramos mais a direita da árvore de origem. Além disso, para o cálculo eficiente da frequência de um padrão é suficiente manter apenas as ocorrências da sua folha mais a direita, o que permite uma otimização do espaço de trabalho utilizado pelo algoritmo.

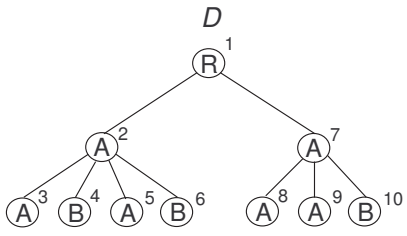


Figura 14: Uma árvore de dados D.

Uma árvore de dados D é uma 6-tupla $(V, E, \mathcal{L}, L, v_0, \preceq)$, tal que V e E são respectivamente os conjuntos de vértices e de ramos de D, $v_0 \in V$ é a raiz de D e $\mathcal{L}$ é o conjunto

de *labels* que são mapeados aos nós de V através da função L. Se $(u, v) \in E$ dizemos que o nó u é pai de v e, definimos a relação entre os irmãos v_1 e v_2 através do símbolo $v_1 \preceq v_2$, que indica v_1 como irmão mais velho que v_2. O tamanho de D, $|D|$, é definido como sendo a cardinalidade do conjunto V, ou seja, é o número de nós de $\mathcal{D}$. Na Figura 14 temos uma representação da estrutura de uma árvore D construída sobre o conjunto de *labels* $\mathcal{L} = \{A,B\}$. Observe que cada nó possui um número indentificador que corresponde à sua localização no percurso em profundidade por D, como no algoritmo TreeMiner.

Uma sub-árvore T tem a sua ocorrência encontrada em D através da função φ, que mapeia os nós de T aos nós de D de acordo com as seguintes verificações:

1) φ preserva a relação parental entre os nós: $(v_1, v_2) \in E_T$ see $(\varphi(v_1), \varphi(v_2)) \in E_D$
2) φ preserva a relação entre os nós irmãos: $v_1 \preceq_T v_2$ see $\varphi(v_1) \preceq_D \varphi(v_2)$
3) φ preserva os *labels* dos nós: $(v_1, v_2) \in E_T$ see $L_T(v) = L_D(\varphi(v))$

Considerando o exemplo de uma sub-árvore T da árvore de dados D, ilustrado na Figura 15, temos o conjunto total de ocorrências de T em D, mapeados pela função φ: $Total(\varphi_T) = \{(2,3,4),(2,3,6),(2,5,6),(7,8,10),(7,9,10)\}$. Conjunto este que nos leva à idéia do cálculo do suporte de um padrão, que em (ASAI et al., 2002) define-se como a relação entre a frequência do padrão em D e $|D|$.

A frequência de um padrão corresponde ao número de ocorrências do padrão que é encontrada a partir do conjunto $Total(\varphi_T)$. Para cada $\varphi_i \in Total(\varphi_T)$ aplica-se a função $Root(\varphi_i)$ que retorna a posição do nó em D que se mapeia à raiz de T, através de φ_i. Essas posições da raiz de T em D serão armazenadas no conjunto $Occ(T)$. Portanto, o suporte do padrão T sobre D, $\sigma_D(T) = freq_D(T) = \frac{\#Occ(T)}{|D|}$. O conjunto $Occ(T) = \{2,7\}$ foi gerado aplicando a função *Root* em todo o conjunto $Total(\varphi_T)$ do padrão T da Figura 15. O suporte de T com relação a D é 0,2 pois, $\#Occ(T) = 2$ (cardinalidade de $Occ(T)$) e $|D| = 10$.

A chave desse algoritmo é a eficiência em armazenar as informações sobre φ_i de cada padrão T. Ao invés de armazenar todas as informações de $Total(\varphi_T)$, FREQT mantém apenas as posições dos nós mais a direita de cada $\varphi_i \in Total(\varphi_T)$. Considerando o conjunto $Total(\varphi_T)$ apresentado acima temos $RMO(T) = \{4,6,10\}$ como a lista de nós mais a direita de T. A partir dela o conjunto $Occ(T) = \{2, 7\}$ também pode ser obtido.

O problema que o algoritmo FREQT propõe resolver é: seja L um conjunto de *labels*, D uma árvore de dados e $0 \leq \sigma_{min} \leq 1$, o suporte mínimo estipulado. Encontrar todas as σ-sub-árvores frequentes tal que $freq_D(T) \geq \sigma_{min}$.

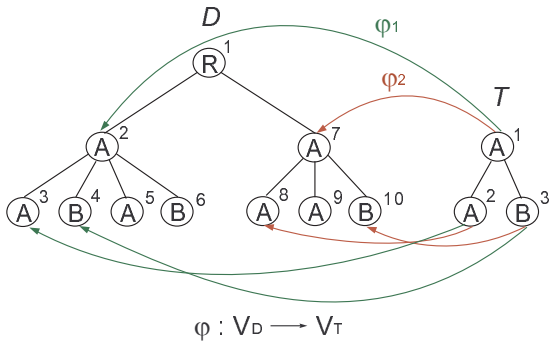
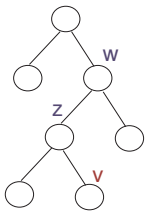


Figura 15: Padrões a serem minerados.

2.2.2.2 Geração das Sub-árvores Candidatas

O algoritmo FREQT (ASAI et al., 2002) possui a geração das árvores candidatas através do *Método* (p, l)*-Expansão*. Para explicar a funcionalidade dessa técnica precisamos introduzir a definição da função π_T^p, que aplicada a um nó r retorna um nó w, ancestral de r, tal que o cumprimento de w a r é p. Considerando a Figura abaixo, $\pi_T^0(v) = v$ (retorna o próprio nó), $\pi_T^1(v) = z$ (retorna o pai do nó) e $\pi_T^2(v) = w$ (retorna um ancestral do nó).



O *Método* (p, l)*-Expansão* é a técnica utilizada pelo algoritmo FREQT para gerar as k-sub-árvores candidatas, com a inclusão de nós na folha mais a direita de um (k-1)-padrão S. A informação (p, l) indica que um nó de *label* l será anexado ao nó w, encontrado pela aplicação da função $\pi_S^p(v)$, onde v é o nó mais a direita de S. Na Figura 16 temos as duas opções de (p, l)-expansão para a árvore S. A quantidade de expansões possíveis é definida pela profundidade de S, ou seja, pelo número de níveis que S possui. Para cada $0 \leq p \leq d$, uma k-sub-árvore candidata é gerada a partir de um (k-1)-padrão frequente. A geração pela $(0, A)$-expansão produz a 4-sub-árvore candidata ilustrada na Figura 16, com a inclusão do nó A como filho do nó v, retorno de $\pi_S^0(v)$.

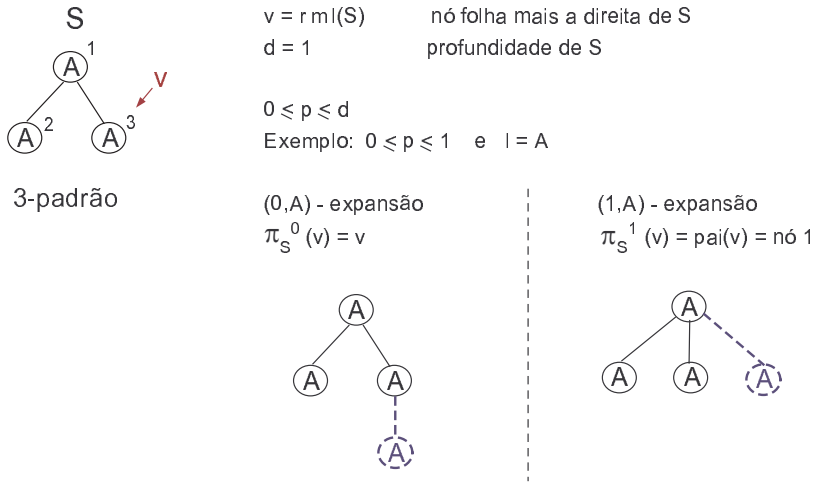


Figura 16: A (p, l)-expansão da árvore S.

2.2.2.3 O Algoritmo FREQT

Considerando a árvore D apresentada na Figura 14, onde $\mathcal{L} = \{A,B\}$, $|D| = 10$, e assumindo que o $\sigma_{min} = 0,2$, apresentaremos o entendimento de cada passo de execução do algoritmo FREQT, encontrando todos os σ-padrões frequentes.

Cálculo de F_1

As 1-sub-árvores candidatas são geradas a partir de uma varredura em D, considerando cada nó de *label* distinto uma 1-sub-árvore. Simultaneamente, o conjunto RMO_1 para cada padrão é gerado. Na Figura 17 temos as 1-sub-árvores frequentes em D. Observem que o suporte desses padrões foram calculados a partir RMO_1, uma vez que a raiz de um nó de uma árvore de tamanho 1 é o próprio nó. Portanto, $F_1 = \{T_1, T_2\}$.

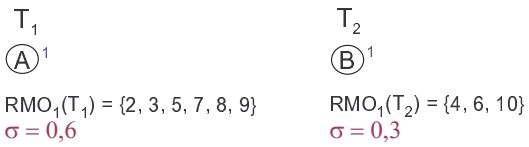


Figura 17: Cálculo de F_1.

Cálculo de F_2

Todas as k-sub-árvores candidatas, $k \geq 2$, são geradas pelo método da (p, l)-expansão, a partir de cada padrão frequente de $F_{(k-1)}$ e $(p, l) \in \{1, ..., d\} \times \mathcal{L}$. Considerando T_1, temos o nó mais a direita de $T_1 = rml(T_1) =$ nó $1 = v$, e a profundidade de $T_1 = d = p = 0$. Portanto, $\pi^0_{T_1}(v) = v$, ou seja, todas as $(0, l)$-extensões de T_1 devem ocorrer no nó 1. Na

Figura 18 temos os padrões que foram gerados com as suas respectivas RMO_2 e suportes calculados.

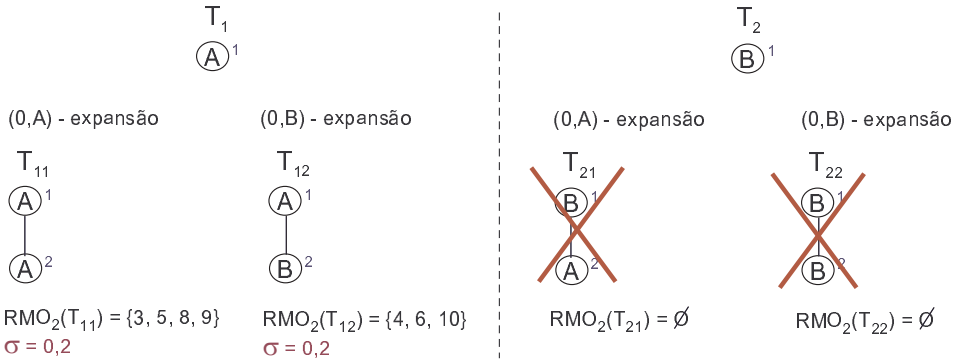


Figura 18: Cálculo de F_2.

O cálculo da lista dos nós mais a direita das ocorrências de um k-padrão S, $k \geq 2$, é feito pelo procedimento *Update-RMO* definido em (ASAI et al., 2002), cuja idéia podemos abstrair da seguinte maneira: a lista RMO_{k-1} da (k-1)-sub-árvore expandida para gerar o k-padrão candidato, é percorrida com o objetivo de buscar os filhos de cada nó i de RMO_{k-1} em D, cujo *label* seja l. A posição dos filhos encontrados dará origem à lista RMO_k, que é gerada de forma ordenada e sem duplicidade, características estas fundamentadas através de lemas definidos em (ASAI et al., 2002). Exemplificando, a RMO_2 da sub-árvore T_{11} da Figura 18 contém todas as posições em D dos filhos dos nós de RMO_1 de T_1, cujo *label* seja A.

Ao final dessa iteração, após o cálculo do suporte como foi feito na iteração anterior, temos $F_2 = \{T_{11}, T_{12}\}$.

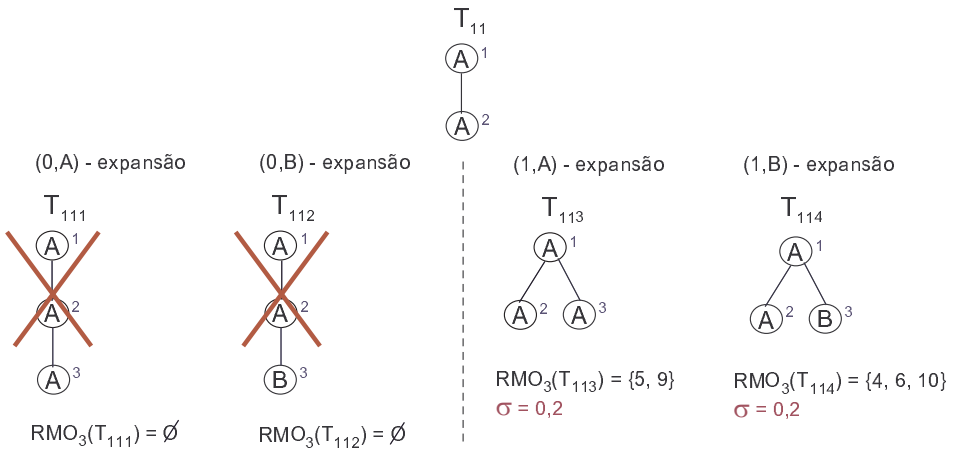


Figura 19: Cálculo de F_3.

Cálculo de F_3

Considerando a expansão a partir da 2-sub-árvore frequente T_{11}, na Figura 19 temos as sub-árvores candidatas de tamanho 3, geradas pelo método (p, l)-expansão. O nó mais a direita de T_{11} é o nó $2 = v$ e a profundidade de T_{11} é $d = 1$. Sendo assim, teremos expansão para os *labels* $l \in \mathcal{L}$ e $0 \leq p \leq 1$, a partir do nó $v = \pi^0_{T_{11}}(v)$ para para $p = 0$, e a partir do nó pai$(v) = \pi^1_{T_{11}}(v)$ para $p = 1$. Pela Figura 19 observamos que nenhuma sub-árvore gerada a partir da $(0, l)$-expansão é frequente, finalizando essa iteração com o conjunto de 3-sub-árvores frequentes, $F_3 = \{T_{113}, T_{114}\}$.

Cálculo de F_4

Seguindo o mesmo raciocínio descrito nas iterações $k \geq 2$ acima, encontramos apenas o padrão T_{1134} frequente, mostrado na Figura 20. Na iteração seguinte, onde k = 5, nenhuma sub-árvore frequente é encontrada, finalizando assim a execução do algoritmo FREQT sobre a árvore de dado D e $\sigma_{min} = 0,2$. Portanto, o conjunto das σ-sub-árvores frequentes geradas pelo algoritmo é $F = F_1 \cup F_2 \cup F_3 \cup F_4 = \{T_1, T_2, T_{11}, T_{12}, T_{113}, T_{114}, T_{1134}\}$.

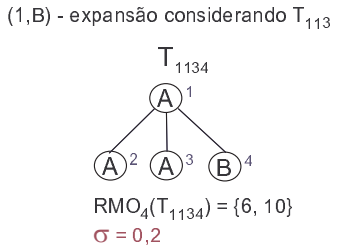


Figura 20: Cálculo de F_4.

Considerações sobre a Poda

A fase da poda de sub-árvores candidatas, fase esta executada antes do cálculo do suporte, em (ASAI et al., 2002) é citada como melhoria a ser implementada em uma nova versão do algoritmo FREQT. Em (ASAI et al., 2002), os autores sugerem duas modalidades para desenvolvimento da poda:

- *Node-Skip:* Essa técnica de poda despreza *labels* de $\mathcal{L}$ tal que as 1-sub-árvores desses *labels* $\notin F_1$, que obviamente não ocorrem nas σ-sub-árvores frequentes de F.
- *Edge-Skip:* Essa modalidade despreza ramos cujos pares de *labels* $\notin F_2$. Similar ao método *Node-Skip* acima, observamos que se o par de *labels* (l_1, l_2) não existe em F_2, então esse mesmo par não ocorre em nenhuma σ-sub-árvores frequentes de F.

2.2.3 TreeFinder - Mineração em Documentos XML

2.2.3.1 Definição do Problema

O Algoritmo TreeFinder proposto em (TERMIER; ROUSSET; SEBAG, 2002) foi desenvolvido com o objetivo de encontrar padrões frequentes em uma coleção de árvores cujos nós possuem *labels*, estrutura esta que naturalmente representa uma coleção de dados XML. Semelhante à idéia apresentada por Zaki no algoritmo TreeMiner, TreeFinder também produz padrões arborescentes frequentes indiretos, ou seja, encontrados com base na relação ancestral-descendente entre os nós. Antes de se iniciar o processo de mineração propriamente dito, a coleção de dados é trabalhada aplicando em seus dados um método de clusterização, onde árvores afins são agrupadas formando clusters de dados. Para cada cluster de árvores é gerado um conjunto de sub-árvores frequentes.

Na Figura 21 temos 2 exemplos de árvores de uma coleção de dados XML. Cada nó possui um *label* cujo contexto não é levado em consideração durante o processo da mineração, ou seja, o *label manga* será tratado da mesma forma no contexto de fruta ou de parte de uma roupa. Uma diferença importante a ser observada é que o número identificador de cada nó corresponde à sua localização no percurso da árvore em largura, e não em profundidade como nos algoritmos de mineração de árvores descritos anteriormente.

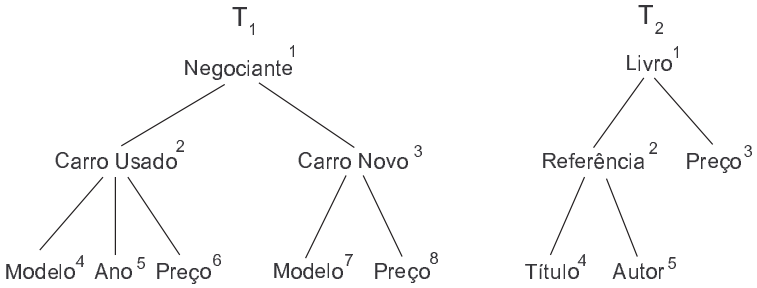


Figura 21: Exemplos de árvores da coleção de dados.

A frequência de sub-árvores indiretas também é avaliada pelo algoritmo TreeFinder. Considerando as árvores mostradas na Figura 21, temos na Figura 22 exemplos de um padrão direto e outro indireto. O padrão S_2 apresenta a relação ancestral-descendente entre os *labels* Livro, Autor e Preço da árvore T_2 da coleção.

Outra consideração sobre o banco de dados, TreeFinder abstrai a coleção de árvores através da representação relacional de cada uma delas. Essa abstração é construída com base no tipo de relação existente entre cada par de *labels* de uma árvore. Seja T uma árvore da coleção: Define-se $Rel(T)$, a conjunção de todos os átomos do tipo $ab(u, v)$ tal

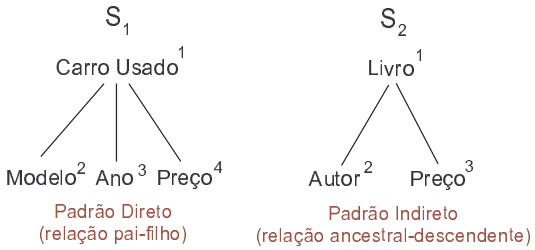


Figura 22: Padrões a serem minerados pelo TreeFinder.

que u e v são nós de T, $label(u) = a$ e $label(v) = b$, e u é pai de v. Define-se também $Rel^+(T)$, a conjunção de todos os átomos do tipo $a * b(u, v)$ tal que u e v são nós de T, $label(u) = a$ e $label(v) = b$, e u é ancestral de v. Na Figura 23 temos a representação da árvore T através das relações pai-filho e ancestral-descendente entre os seus nós. Observe que todas as relações paternas também são consideradas como ancestral-descendente.

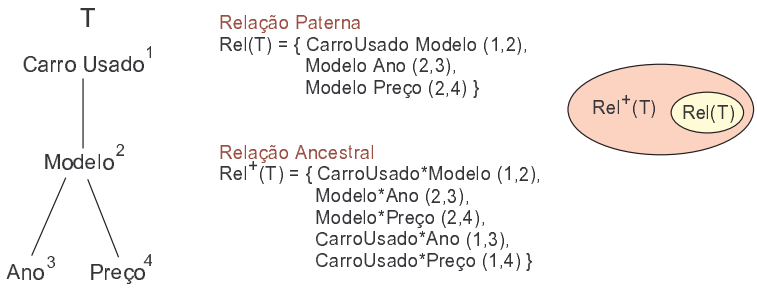
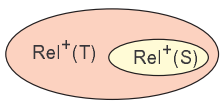


Figura 23: Representação relacional de uma árvore.

Uma árvore da coleção suporta um padrão se ela possui as mesmas relações ancestral-descendentes existentes na sub-árvore. No padrão S_1 mostrado acima, o *label* Carro Usado é ancestral dos nós Modelo, Ano e Preço. Essas mesmas relações podem ser facilmente identificadas nas árvore T_1 da Figura 21 e T da Figura 23. Portando, S_1 é suportado pelas árvores T_1 e T. Formalmente, seja S e T árvores com *labels* em seus nós. S é suportada por T se e somente se $Rel^+(T)$ θ-inclui $Rel^+(S)$. Intuitivamente, a função θ-inclui é responsável por mapear os nós de S em T. Se pelo menos 1 nó de S não for mapeado por θ-inclui em T, então T não suporta S.



O problema que o algoritmo TreeFinder propõe resolver é: seja D um conjunto de árvores com *labels* em seus nós e seja $0 \leq \sigma_{min} \leq 1$. Encontrar todas as sub-árvores

maximais frequentes T tal que $\sigma(T) \geq \sigma_{min}$. Uma árvore maximal é definida em (TERMIER; ROUSSET; SEBAG, 2002) como: seja $\{T, T_1, ..., T_n\}$ árvores cujos nós possuem *labels*. Dizemos que T é maximal com relação a $T_1, ..., T_n$ se e somente se:

1) $\forall i \in [1, n]$, T é suportada por T_i
2) T é maximal se existe $T^{'}$; T é suportada por $T^{'}$ e $\forall i \in [1, n], T^{'}$ é suportada por T_i Então $T^{'} = T$

2.2.3.2 O Algoritmo TreeFinder

Numa visão macro, o algoritmo TreeFinder divide a sua execução em duas grandes etapas de processamentos. Na *1ª Etapa* ocorre o agrupamento das árvores cujos pares de *labels* ocorrem frequentemente juntos. Essa tarefa é realizada aplicando o algoritmo padrão de mineração de *itemsets*, o Apriori (AGRAWAL; SRIKANT, 1994). Durante a *2ª Etapa*, TreeFinder identifica as árvores maximais de cada *cluster* de árvores criados na etapa anterior. Para essa atividade, TreeFinder utiliza um algoritmo de generalização da Programação Lógica Indutiva. O entendimento de como esses sub-algoritmos de TreeFinder são processados virá através do exemplo desenvolvido a seguir.

Na Figura 24 temos o banco de dados D com 5 árvores, sobre as quais iremos executar o algoritmo TreeFinder para encontrar as sub-árvores frequentes considerando o suporte mínimo $\sigma = 0,4$. Logo abaixo, temos a representação relacional de cada uma delas, que corresponde à abstração das árvores de dados de D.

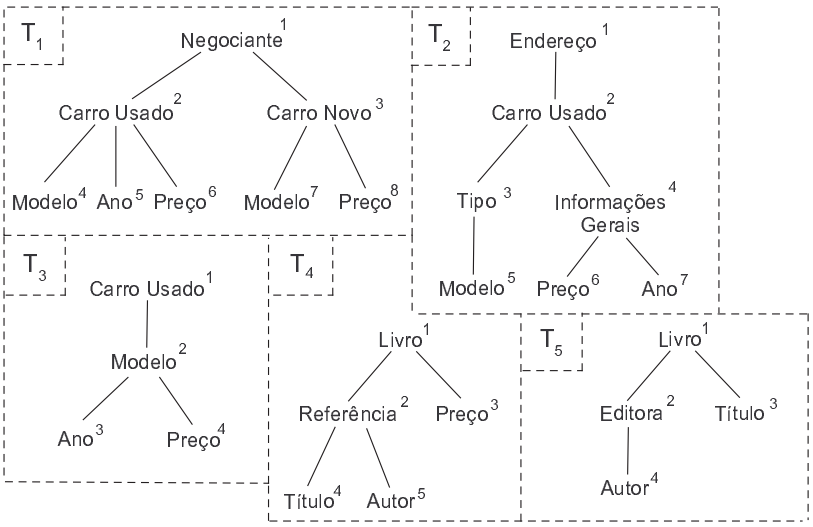
1ª Etapa: Agrupamento pela ocorrência de pares de *labels*

Essa etapa recebe como *input* a abstração do banco de dados D, onde cada T_i é visualizada como uma transação de itens do tipo $x * y$, tal que x é o *label* do ancestral do nó y em T_i. Essa divisão das árvores em itens, que correspondem a pares de *labels*, desconfigura a estrutura de árvores dos dados, mas viabiliza o uso de um algoritmo padrão de mineração de *itemsets* para descobrir pares de *labels* frequentes que ocorrem em D.

O algoritmo Apriori é aplicado sobre o conjunto de itens gerados pela abstração de D, e $\sigma = 0,4$. Apenas os maiores *itemsets* frequentes gerados pelo Apriori são considerados como *output* dessa etapa, onde cada *itemset* frequente corresponde a um *cluster* de dados. Considerando nosso exemplo, os maiores *itemsets* frequentes foram gerados na iteração 3 de Apriori, onde temos $F_3 = \{c_1, c_2\}$, tal que c_1 = {CarroUsado*Modelo, CarroUsado*Ano, CarroUsado*Preço} e c_2 = {Livro*Título, Livro*Autor}.

Ainda no contexto do algoritmo Apriori, o suporte de cada c_i é o número de transações

$T_i \in D$ que inclui c_i (operação entre conjuntos). O conjunto suporte(c_i) corresponde às transações que suportam c_i. Sendo assim, temos os *clusters* de dados gerados como *output* dessa etapa, suporte(c_1) = $\{T_1, T_2, T_3\}$ e suporte(c_2) = $\{T_4, T_5\}$.



T_1: { Negociante*Carro Usado,
Negociante*Carro Novo,
Carro Usado*Modelo,
Carro Usado*Ano,
Carro Usado*Preço,
Carro Novo*Modelo,
Carro Novo*Preço,
Negociante*Modelo,
Negociante*Ano,
Negociante*Preço }

T_2: { Endereço*Carro Usado,
Carro Usado*Tipo,
Carro Usado*Modelo,
Carro Usado*Informações Gerais
Carro Usado*Preço
Carro Usado*Ano,
Tipo*Modelo,
Informações Gerais*Preço,
Informações Gerais*Ano,
Endereço*Tipo,
Endereço*Informações Gerais,
Endereço*Modelo,
Endereço*Preço,
Endereço*Ano }

T_3: { Carro Usado* Modelo,
Modelo*Ano,
Modelo*Preço,
Carro Usado*Ano,
Carro Usado*Preço }

T_4: { Livro*Referência,
Livro*Preço,
Referência*Título,
Referência*Autor,
Livro*Título,
Livro*Autor }

T_5: { Livro*Editora,
Livro*Título,
Livro*Autor,
Editora*Autor }

Figura 24: Banco de Dados $\mathcal{D}$ com 5 árvores.

2ª Etapa: Busca pelas árvores maximais

A 2ª etapa de execução do TreeFinder recebe como parâmetro de entrada o *output* da etapa de clusterização anterior, e busca pela árvore maximal comum de cada *cluster* de dados, ou seja, busca pela árvore incluída em todas as demais árvores de um mesmo *cluster*. Para cada *cluster* $\{T_1, T_2, ..., T_n\}$, utiliza-se o método da *Menor Generalização Geral*, $LGG(Rel^+(T_1), ..., Rel^+(T_n))$, das fórmulas relacionais conjutivas que representam as árvores.

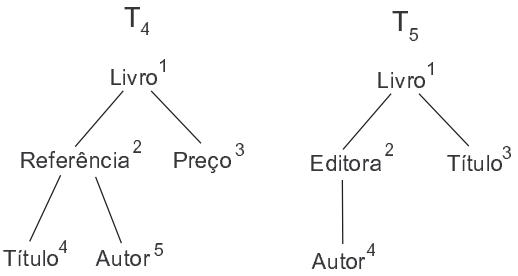


Figura 25: *Cluster* de dados suporte(c_2).

Seja C_1 e C_2 duas cláusulas de fórmulas conjuntivas. Dizemos que C_1 generaliza C_2, denotado por $C_1 \leq C_2$, se C_1 está incluso em C_2, ou seja, $\exists$ uma substituição θ tal que $C_1\theta \subseteq C_2$. Na Figura 25 temos as árvores que compõem o *cluster* suporte(c_2). As cláusulas C_1 = Livro*Título(1,4) e C_2 = Livro*Título(1,3) existem respectivamente na representação relacional de T_4 e T_5, e C_2 é a menor generalização para o par de *labels* Livro*Título. Para cada *cluster*, o algoritmo calcula a menor generalização das cláusulas do tipo a*b(u,v) através do método *LGG*.

A aplicação $LGG(Rel^+(T_4), Rel^+(T_5))$ do *cluster* de dados ilustrado na Figura 25 produz Livro*Título(1,3) $\wedge$ Livro*Autor(1,4), ou seja, as relações ancestrais Livro*Título(1,3) e Livro*Autor(1,4) correspondem à menor generalização das cláusulas de $Rel^+(T_4)$ e $Rel^+(T_5)$. Porém, a partir do resultado de $LGG(Rel^+(T_4), Rel^+(T_5))$, o algoritmo tenta identificar se alguma dessas relações são do tipo pai-filho. Essa sub-etapa é importante para melhor garantirmos a topologia dos padrões frequentes. Sendo assim, identificamos que a relação Livro Título(1,3) é paterna. Na Figura 26 ilustramos as duas sub-árvores frequentes que foram geradas pelo algoritmo TreeFinder, a primeira encontrada no *cluster* suporte(c_2) e a outra gerada a partir do *cluster* suporte(c_1). Observe que variações na estrutura são permitidas, ou seja, não há preocupação com a ordem entre os nós irmãos.

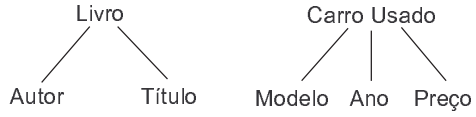


Figura 26: Padrões frequentes gerados por TreeFinder.

O próximo exemplo mostra que o TreeFinder não é completo, ou seja, ele não garante encontrar todas as σ-árvores frequentes maximais. Na Figura 27 temos um banco de dados com 4 árvores e os 2 padrões frequentes que foram gerados pelo TreeFinder, considerando um suporte mínimo $\sigma = 0,5$ e único *cluster*. Observem que o padrão mais a direita

também é frequente porém não foi encontrado pelo algoritmo.

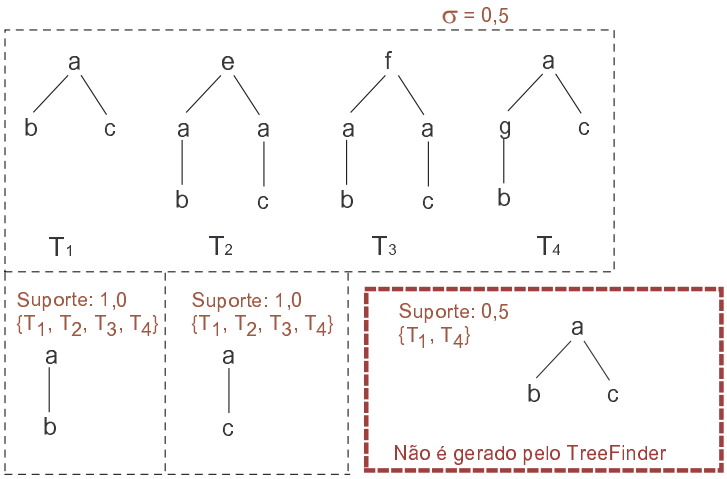


Figura 27: Exemplo de padrão frequente não encontrado pelo TreeFinder.

2.3 Autômatos de Árvores

Com o surgimento do XML, iniciou-se uma simbiose entre as áreas de conhecimento sobre pesquisa de documentos, banco de dados e linguagens formais, de onde surgiu os autômatos de árvores não *rankeadas*. Numa conceituação simples, as árvores não *rankeadas* são aquelas cujos nós podem ter um número arbitrário de filhos e, como as estruturas de documentos XML (DTD) podem ser naturalmente representadas por essas estruturas, os autômatos de árvores não *rankeadas* podem prover a pesquisa de informações nesses documentos. Em (NEVEN, 2002), encontramos um *survey* sobre a utilização dos autômatos de árvores em pesquisas envolvendo dados semi-estruturados, e em (MURATA et al., 2005) encontramos a definição de autômato de árvore local, particularidade dos autômatos de árvores que representam uma DTD.

Um autômato de árvore não *rankeada*, que convencionamos a partir desse ponto de apenas autômato de árvore, é uma tupla $\mathcal{A} = (\Sigma, Q, \delta, q_0)$, onde Σ é o alfabeto finito de símbolos, Q é um conjunto finito de estados, $q_0 \subseteq Q$ é o estado inicial, e $\delta : Q \times \Sigma \rightarrow 2^Q$ é uma função de transição que associa a cada par $(q, a) \in Q \times \Sigma$, uma expressão regular sobre Q. É importante observar que diferente dos autômatos finitos aplicados em *strings*, onde a função de transição mapeia o par (q, a) em um conjunto de estados, a função de transição nos autômatos de árvores mapeia (q, a) a uma expressão regular.

Uma árvore T é aceita pelo autômato de árvore $\mathcal{A}$ se T é percorrida por $\mathcal{A}$. Essa

validação de $\mathcal{A}$ pode ser feita sobre os nós da árvore na direção *bottow-up* ou *top-down*. Numa visão da validação *top-down*, o estado inicial é associado à raiz e novos estados são associados aos nós internos de T pela função δ, de acordo com seus *labels* e estados de seus nós pais. A árvore é aceita por $\mathcal{A}$ se a função de transição aplicada aos nós folhas de T produzir o *string* vazio.

Considere o autômato de árvore $\mathcal{A} = (Q, \Sigma, \delta, q_0)$, tal que $Q = \{q_0, q_1, q_2\}$, onde q_0 é o estado inicial, $\Sigma = \{a, b, d\}$, $\delta(q_0, a) = q_1 q_2$, $\delta(q_1, b) = \delta(q_2, d) = \epsilon$. Na Figura 28 temos o exemplo de uma árvore que é aceita por $\mathcal{A}$. Inicialmente, $\mathcal{A}$ associa o estado q_0 à raiz de T. A função δ aplicada à raiz, $\delta(q_0, a)$, associa aos seus nós filhos os estados q_1 e q_2. A aplicação de δ aos *labels* das folhas produz o *string* vazio.

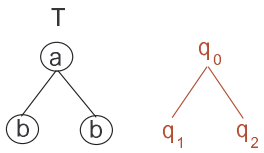


Figura 28: Uma árvore e seu percurso de validação pelo autômato $\mathcal{A}$.

2.3.1 Autômatos em Árvores Locais

Uma DTD, *Document Type Definition*, pode ser definida como um conjunto de regras que define quais tipos de dados e entidades podem existir em um documento XML, ou seja, ela esquematiza a estrutura das informações dentro de um documento. Em (MURATA et al., 2005) encontramos que uma DTD pode ser representada pela gramática de árvore local, conceito este definido em (TAKAHASHI, 1975), a partir da qual naturalmente construímos um autômato de árvore local.

Uma gramática de árvore regular é uma tupla $G = (N, T, S, P)$, onde N é o conjunto finito de nós não terminais, T é o conjunto finito de nós terminais, $S \subset N$ é o conjunto de estados iniciais, e P é o conjunto finito de regras do tipo $X \rightarrow a\ r$, onde $X \in N$, $a \in T$ e r é uma expressão regular sobre N. Voltando ao nosso autômato de árvore $\mathcal{A}$ definido acima, a gramática de árvore regular que o representa é $G_{\mathcal{A}}$, tal que $N = \{q_0, q_1, q_2\}$, $T = \{a, b, d\}$, $S = \{q_0\}$ e $P = \{q_0 \rightarrow a\ (q_1 q_2),\ q_1 \rightarrow b\ \epsilon,\ q_2 \rightarrow d\ \epsilon\}$.

Dizemos que existe uma *competição* entre os nós de uma gramática de árvore regular, se encontramos dois diferentes nós não terminais q_i e q_j tal que, existem regras r_i e r_j definidas para q_i e q_j respectivamente, que compartilham o mesmo nó terminal t. Considere a seguinte gramática de árvore regular G_2, onde $N = \{q_0, q_1, q_2, q_3\}$, $T = \{a, b, d\}$,

$S = \{q_0\}$ e $P = \{q_0 \rightarrow a\ (q_1q_2 + q_3q_2),\ q_1 \rightarrow b\ \epsilon,\ q_2 \rightarrow d\ \epsilon,\ q_3 \rightarrow b\ (q_2)\}$. Os estados q_1 e q_3 estão competindo o *label* b.

Uma gramática de árvore regular é dita *local* quando não existe competição entre seus nós não terminais. A gramatica $G_{\mathcal{A}}$ é um exemplo de gramática de árvore local, tipo de gramática de árvore que corresponde às estruturas de DTD, de acordo com (MURATA et al., 2005). Por esse motivo, as restrições consideradas pelo algoritmo CobMiner são restrições locais. A seguir mostraremos como uma restrição poder ser representada por um autômato de árvore local.

3 O Problema da Mineração de Árvores com Restrições

Como vimos até aqui, os trabalhos já realizados em mineração de árvores buscam a geração de padrões frequentes que satisfazem um suporte mínimo previamente estabelecido. No entanto, o conjunto de informações mineradas deve ser refinado com o objetivo de filtrar apenas o que é relevante para o usuário, ou seja, produzir padrões que satisfazem certas restrições que refletem melhor os interesses do usuário.

O algoritmo CobMiner utiliza a técnica de mineração do algoritmo TreeMiner (ZAKI, 2002), incorporando as restrições dos usuários dentro da fase de geração dos candidatos, o que aumenta consideravelmente a eficiência do processo de mineração. Nesse capítulo formalizamos o problema da *mineração de padrões arborescentes com restrições*. Inicialmente, mostramos como as restrições podem ser representadas por autômatos de árvores locais, em seguida apresentamos os conceitos relacionados a padrões p-válidos e ao suporte de um padrão p-válido, finalizando o capítulo descrevendo o problema de mineração em questão.

3.1 Restrições em Autômato de Árvore Local

Suponha a existência de um portal de comércio eletrônico onde os clientes navegam através das categorias de itens e eventualmente fazem uma compra. Uma dessas categorias é *Livros*, que denotaremos por LV. Suponha também que o gerente de vendas desse portal esteja interessado em descobrir se a seção de livros pode alavancar as vendas de revistas, buscando comportamentos de navegação (padrões de árvores), onde o cliente visitou o portal, percorreu diversos livros e fez ou não a compra de algum deles, além de ter percorrido, após a visita à seção de livros, a categoria *Revistas*, RS, sem necessariamente ter feito a compra de alguma revista. Observe que nesse contexto, o gerente não está interessado apenas em descobrir padrões de navegação frequentes que ocorrem no portal,

mas sim, padrões que também obedecem à particularidade descrita acima. Tais padrões são exatamente aqueles que são satisfeitos pelo seguinte autômato de árvore local $\mathcal{A}$:

- $\Sigma = \{Portal, LV, RS, Compra\} \cup L \cup R$, onde L e R são conjuntos finitos de símbolos que representam os catálogos de livros e de revistas, respectivamente.
- $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$, onde q_0 é o estado inicial.
- $\delta(q_0, Portal) = q_1 q_2$;
  $\delta(q_1, LV) = q_3^+$;
  $\delta(q_2, RS) = q_4^*$;
  $\delta(q_3, x) = \epsilon + q_5$, $\forall x \in L$;
  $\delta(q_4, y) = \epsilon + q_5$, $\forall y \in R$;
  $\delta(q_5, Compra) = \epsilon$;

Todos os padrões de navegação que são aceitos pelo autômato de árvore local $\mathcal{A}$ correspondem exatamente aos padrões de navegação que o gerente do portal deseja, sendo dois exemplos, as árvores mostradas na Figura 29. Portanto, o algoritmo CobMiner utiliza-se dos autômatos de árvores locais como ferramenta para incorporar as especificações dos usuários como restrições à mineração. Note que dado um padrão de navegação S e um autômato de árvore local $\mathcal{A}$, se existe um percurso de $\mathcal{A}$ sobre S, então esse percurso é único.

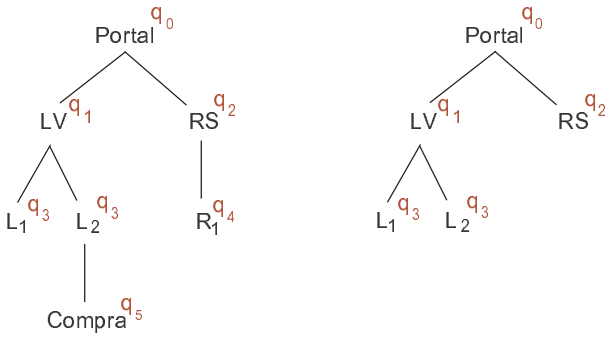


Figura 29: Exemplos de padrões de navegação aceitos por $\mathcal{A}$.

Formalizando, seja $\mathcal{A} = (Q, \Sigma, \delta, q_0)$ um autômato de árvore local e $S = (N_S, B_S)$ um padrão arborescente. Um percurso de $\mathcal{A}$ sobre S é uma função $\gamma : N_S \rightarrow Q$ tal que para cada nó $v \in N_S$ com k filhos $v_1, v_2, ..., v_k$, a palavra $\gamma(v_1)...\gamma(v_k)$ é aceita pela expressão regular $\delta(\gamma(v), l(v))$, onde $l(v) \in \Sigma$ é o *label* do nó v em S. Dizemos que $\mathcal{A}$ aceita o padrão

S, ou seja, S é *válido com respeito à* $\mathcal{A}$, se existe um percurso γ tal que para cada nó folha v de S, temos $\delta(\gamma(v), l(v)) = \epsilon$.

Dissemos anteriormente que os algoritmos CobMiner e TreeMiner possuem a mesma técnica de mineração, mas o que isso significa? Em primeiro lugar, permanece inalterada a estrutura do banco de dados a ser minerado, com as árvores representadas pelos *strings* codificados. Padrões indiretos também são gerados pelo algoritmo CobMiner, ou seja, padrões que podem ser suportados pelas árvores da coleção através da relação ancestral-descendente entre seus nós. E, por último, até o momento, o suporte dos candidatos também é calculado pelo CobMiner, através da junção das listas de escopos.

Como a principal meta do algoritmo CobMiner é a eficiência da mineração produzindo padrões realmente relevantes para os usuários, a incorporação das restrições também promove um ganho considerável no tempo de resposta de execução do algoritmo. Visualizando essas duas evoluções, o algoritmo CobMiner promove essa melhoria através de uma fase de geração de candidatos direcionada pelo autômato de árvore local, gerando sub-árvores candidatas p-válidas, conceito este formalizado a seguir.

3.2 O Padrão P-válido

Um padrão gerado pelo algoritmo CobMiner é dito *p-válido com respeito à* restrição imposta por um autômato de árvore local, se ele se enquadra na definição descrita abaixo.

Definição (Padrão P-válido): Sejam $\mathcal{A} = (Q, \Sigma, \delta, q_0)$ um autômato de árvore local, $S = (N_S, B_S)$ um padrão arborescente, e γ a função que percorre S com o autômato $\mathcal{A}$. Dizemos que S é *prefixo-válido ou p-válido com respeito à* $\mathcal{A}$, se para cada nó $v \in N_S$ com k filhos $v_1, v_2, ..., v_k$, a palavra $\gamma(v_1)...\gamma(v_k)$ é um prefixo da palavra aceita pela expressão regular $\delta(\gamma(v), l(v))$, onde $l(v) \in \Sigma$ é o *label* do nó v em S.

Considerando o autômato de árvore local $\mathcal{A}$ definido na seção anterior, na Figura 30 temos um exemplo de padrão p-válido com respeito a $\mathcal{A}$, onde $\delta(q_0, portal)$ produz o prefixo q_1 da expressão regular q_1q_2, $\delta(q_1, LV)$ produz o prefixo q_3 da expressão regular q_3^+, e $\delta(q_3, L_1)$ produz o *string* vazio, que é prefixo de toda expressão regular.

O algoritmo CobMiner gera novos candidatos de tamanho k+1 a partir da expansão de k-padrões frequentes e p-válidos. Para isso, ele utiliza de informações contidas em cada nó dos k-padrões p-válidos, armazenadas numa estrutura chamada *Traço de Percurso*, definida abaixo.

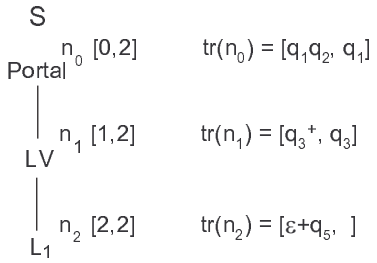


Figura 30: Um padrão p-válido com respeito à restrição $\mathcal{A}$.

Definição (Traço de Percurso): Sejam $\mathcal{A} = (Q, \Sigma, \delta, q_0)$ um autômato de árvore local, $S = (N_S, B_S)$ um padrão arborescente p-válido com respeito a $\mathcal{A}$, e γ a função que percorre S com o autômato $\mathcal{A}$. Para cada nó $v \in N_S$ com k filhos $v_1, v_2, ..., v_k$, existe uma lista $tr(v) = [e, q]$, chamada *traço de percurso* do nó v, onde $e = \delta(\gamma(v), l(v))$ corresponde à expressão regular que está validando a palavra $\gamma(v_1)...\gamma(v_k)$, e q indica até que ponto a palavra $\gamma(v_1)...\gamma(v_k)$ foi validada pela expressão regular e.

Na Figura 30 temos ilustrado o traço de percurso dos nós do padrão p-válido S. $tr(n_0) = [e_1, q_1]$, onde $e_1 = q_1q_2$, indica que os filhos de n_0 estão percorrendo a expressão regular e_1, e que esse percurso está no estado q_1. A lista de percurso do nó n_2 indica a expressão que seus filhos devem percorrer, porém, até o momento, esse nó não possui filhos.

Formalização do Problema da Mineração de Árvores com restrições:

- Dado: um banco de dados de árvores $\mathcal{D}$, um autômato de árvore local $\mathcal{A}$ e o suporte mínimo σ_{min}.
- Encontrar: todas as sub-árvores frequentes que satisfazem à restrição $\mathcal{A}$.

A seguir, apresentamos o algoritmo CobMiner como proposta de solução do problema formulado acima. A fase da geração de candidatos, a poda com restrição e a validação final do aceite dos padrões frequentes pela restrição $\mathcal{A}$, são as principais frentes abordadas nesse trabalho.

4 O Algoritmo CobMiner

Neste capítulo, apresentamos o algoritmo CobMiner como solução para o problema da mineração de árvores frequentes e válidas com respeito a uma restrição imposta pelo usuário. Em elevado nível de abstração, o CobMiner é uma especialização do algoritmo TreeMiner apresentado em (ZAKI, 2002), incluindo no processo de mineração o contexto das restrições, conforme proposta apresentada em (GAROFALAKIS; RASTOGI; SHIM, 1999). Entretanto, veremos a seguir que não é uma tarefa trivial adaptar a idéia do uso de restrição em mineração de sequências para o cenário das coleções de árvores.

Adicionando uma fase de pós-processamento ao algoritmo TreeMiner, obtemos uma primeira iniciativa de mineração de padrões arborescentes considerando restrições de um autômato de árvore local $\mathcal{A}$. Inicialmente, utiliza-se o TreeMiner para gerar o conjunto F com todos os padrões frequentes. Em seguida, cada padrão $t_i \in F$ é validado pela restrição $\mathcal{A}$, permanecendo em F apenas os padrões frequentes válidos com respeito a $\mathcal{A}$.

Basicamente, a etapa de pós-processamento que construímos para o TreeMiner é um validador de árvores, que verifica se elas são aceitas pelo autômato de árvore local $\mathcal{A}$. A função δ aplicada a um nó de uma árvore t_i indica qual expressão regular os filhos desse nó devem obedecer. Voltando às árvores ilustradas na Figura 29 e ao autômato de árvore definido na seção 3.1 do capítulo anterior, o nó n_0 possui como filhos os nós LV e RS, cujos *labels* possuem δ definidos para os estados q_1 e q_2 respectivamente, gerando a palavra q_1q_2 que deve ser reconhecida pela expressão regular $\delta(q_0, Portal)$. Aplicando o mesmo raciocínio a todos os nós de t_i, se todas as palavras forem reconhecidas, t_i é aceita por $\mathcal{A}$.

Na Figura 31 ilustramos como é o funcionamento desse reconhecedor de palavras [1], encontrado na internet e adaptado para a nossa necessidade, que recebe como parâmetro uma expressão regular ER e uma palavra P. As expressões regulares são transformadas em um DFA que tenta ser percorrido por P.

[1] http://www.codeproject.com/cpp/OwnRegExpressionsParser.asp?msg=2292318

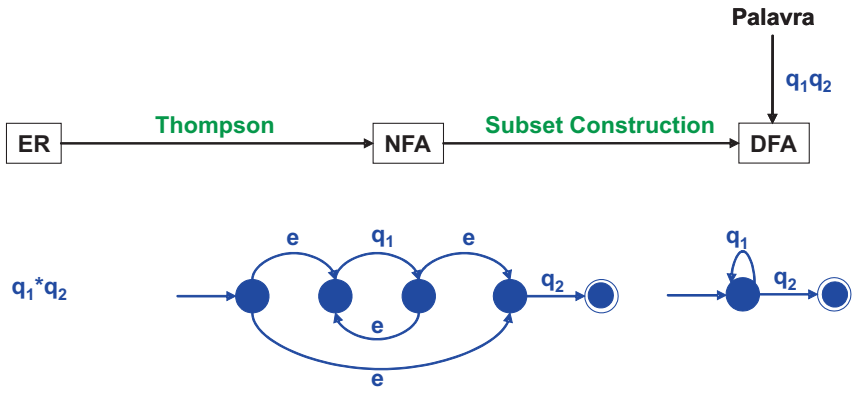


Figura 31: Reconhecedor de Palavras.

Na Figura 32 temos as fases de processamento dos algoritmos CobMiner e TreMinerPP (TreeMiner+PósProcessamento). As fases ilustradas na cor laranja referem-se às fases do processo de mineração do algoritmo TreeMiner (ZAKI, 2002). Observe que os dois algoritmos possuem o mesmo procedimento de cálculo do suporte candidatos. Na cor verde temos as etapas de pós-processamento de ambos, sendo a diferença de tonalidade indicador de que são procedimentos distintos. As restrições são consideradas nas fases de geração e poda de candidatos do CobMiner, ilustradas em azul, e nas etapas de pós-processamento do CobMiner e TreeMiner.

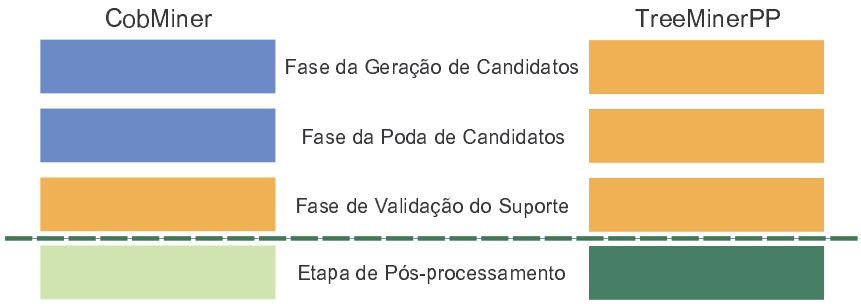


Figura 32: Etapas de processamento dos algoritmos CobMiner e TreeMinerPP.

Existem duas diferenças importantes entre os algoritmos CobMiner e TreeMinerPP: (1) CobMiner faz uso das restrições dentro do processo de mineração, nas fases de geração e poda de candidatos, enquanto que o algoritmo TreeMinerPP utiliza-se das restrições após a mineração, para filtrar os padrões frequentes que a satisfazem. (2) CobMiner considera um *relaxamento* para a restrição do autômato de árvore local $\mathcal{A}$, tornando-a menos restritiva. Ou seja, ao invés dele produzir padrões frequentes válidos, ele produz padrões frequentes p-válidos com respeito a $\mathcal{A}$. A razão do uso desse relaxamento está relacionado com a geração de candidatos, pois é mais natural pensarmos em expandir um padrão p-válido P de tamanho k para obter outro padrão p-válido P' de tamanho k+1,

do que pensarmos na expansão de um padrão válido para obtenção de outro válido.

A seguir, descrevemos com detalhes o algoritmo CobMiner. Iniciamos o capítulo mostrando as modificações implementadas na fase de geração de candidatos utilizada pelo algoritmo TreeMiner. Depois discutimos sobre a utilização da poda oportunista no contexto da mineração de árvores com restrições e, por fim, mostramos como CobMiner produz padrões válidos que satisfazem a especificação do usuário.

4.1 A Geração de Candidatos com Restrição

O mecanismo de mineração do algoritmo CobMiner permanece recursivo em profundidade, baseado no método de expansão de classes de equivalências utilizado pelo TreeMiner. Em breves linhas, e diferente dos demais algoritmos que seguem a técnica de mineração do Apriori, que a cada iteração k produz todos os padrões frequentes de tamanho k, o algoritmo CobMiner, como o TreeMiner, utiliza-se da estratégia da busca em profundidade para produzir seus padrões frequentes. A cada passo $k \geq 2$, o conjunto de k-padrões frequentes é dividido em n classes de equivalência $P_1^k, ..., P_n^k$. A geração de novos candidatos de tamanho $k+1$ se inicia pela primeira delas, P_1^k, através da combinação de seus pares de elementos, que após o cálculo do suporte, produz um novo conjunto com m classes de equivalência $P_{1,1}^{k+1}, ..., P_{1,m}^{k+1}$, sendo o processo reiniciado a partir da expansão da classe $P_{1,1}^{k+1}$. Assim, os padrões frequentes são gerados recursivamente em profundidade, até que em um nível y, nenhum padrão é encontrado, quando o CobMiner retorna ao nível anterior em busca da próxima classe de equivalência a ser expandida.

O CobMiner faz uso da *classe de equivalência p-válida*, que além do prefixo e de seus elementos, também contém o *mapa de percurso*, uma lista que armazena os traços de percurso do prefixo p-válido da classe de equivalência.

Definição (Classe de Equivalência P-válida): Sejam $\mathcal{A} = (Q, \Sigma, \delta, q_0)$ um autômato de árvore local, e C^k uma classe de equivalência de elementos de tamanho k e prefixo $\mathcal{P}$. Dizemos que C^k é uma *classe de equivalência p-válida* com respeito à restrição $\mathcal{A}$, denotada por $[P]^{pv}$, se o seu prefixo $\mathcal{P}$ de tamanho $k-1$ é p-válido com respeito a $\mathcal{A}$, ou seja, se para cada nó $n_1, ..., n_{k-1}$ do prefixo $\mathcal{P}$ de C^k, existe um traço de percurso não vazio no mapa de percurso de C^k.

Na Figura 33 temos o autômato de árvore local $\mathcal{H}$, e um exemplo de classe de equivalência p-válida com respeito a $\mathcal{H}$, gerada a partir da 2-sub-árvore $ab-1$. O mapa de percurso dessa nova classe possui 2 entradas, correspondendo respectivamente aos traços

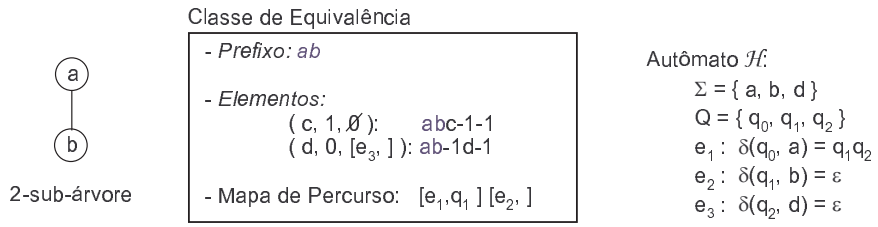


Figura 33: Uma classe de equivalência p-válida com respeito à restrição $\mathcal{H}$.

de percurso $tr(n_0)$ e $tr(n_1)$ dos nós do prefixo $\mathcal{P}$=ab. Note que não encontramos nenhum traço de percurso vazio no mapa de percurso do prefixo $\mathcal{P}$, ou seja, para cada nó de $\mathcal{P}$ existe uma expressão regular em validação ou a ser validada pelos seus filhos. Outra observação importante: os elementos de uma classe de equivalência p-válida encontram-se ordenados lexograficamente pelo par (*label*,posição), sendo então $(c, 1, \emptyset) < (d, 0, [e_3,])$.

Cada elemento $(x, p, tr) \in [P]^{pv}$ representa uma sub-árvore S, onde x e p, como nos elementos das classes do TreeMiner, correspondem respectivamente ao *label* do último nó incluso em $\mathcal{P}$ para formação do padrão S e à posição em $\mathcal{P}$ onde x foi ligado, e tr é o traço de percurso que identifica o caminho que os futuros filhos de x devem percorrer com respeito a $\mathcal{H}$, caso S seja p-válido. Por exemplo, o elemento $(d, 0, [e_3,])$ da classe de equivalência p-válida da Figura 33, refere-se ao padrão $S = ab-1d-1$, gerado através da inclusão de d ao nó n_0 de $\mathcal{P}$. S é p-válido com respeito à $\mathcal{H}$, porque a inclusão de d como filho do nó a permite percorrer a expressão regular $e_1 = q_1q_2$, do estado q_1 para o estado q_2, definindo ainda a expressão regular e_3 como caminho a ser percorrido pelos filhos d.

As posições válidas onde um nó de *label* x pode ser ligado ao prefixo para gerar novos padrões são identificadas, de maneira similar ao TreeMiner, pelo seguinte lema: *Seja $\mathcal{P}$ o prefixo de uma classe de equivalência p-válida e n_r o nó mais a direita de $\mathcal{P}$, cujo escopo é $[r, r]$. Seja $(x, i, tr) \in [P]^{pv}$, uma sub-árvore de $[P]^{pv}$. O conjunto de posições válidas em $\mathcal{P}$ onde x pode ser incluído é dado por $\{i : n_i$ tem escopo $[i, r]\}$.* Voltando à Figura 33, tanto n_0 quanto n_1 do prefixo P são posições válidas para se criar novos padrões de tamanho 3.

4.1.1 A P-Expansão das Classes de Equivalência P-válidas

O algoritmo CobMiner gera seus padrões candidatos através do método da *p-expansão de classes de equivalência p-válidas*. Numa visão macro, as novas classes de equivalência p-válidas, de candidatos de tamanho $k+1$, são criadas apenas pela expansão dos elementos

p-válidos de tamanho k.

O método da *p-expansão de classes de equivalência p-válidas* é baseado no seguinte teorema apresentado em (ZAKI, 2002): Seja $\mathcal{P}$ o prefixo de uma classe de equivalência p-válida $[P]^{pv}$, e (x, i, tr_x) e (y, j, tr_y) dois de seus elementos. Seja $[P_x]^{pv}$ a classe de todas as extensões do elemento (x, i, tr_x). Define-se o *operador junção* $\oplus$ sobre dois elementos da classe de equivalência p-válida $[P]^{pv}$, $(x, i, tr_x) \oplus (y, j, tr_y)$:
Se $\boldsymbol{tr_x = \emptyset}$: nenhum candidato é gerado;
Senão (trata-se de um padrão p-válido),

Caso 1 - $(i = j)$:

a) Se $\mathcal{P} \neq \emptyset$, adiciona-se $(y, j, \emptyset)$ e $(y, n_i, \emptyset)$ em $[P_x]$;

b) Se $\mathcal{P} = \emptyset$, adiciona-se $(y, j + 1, \emptyset)$ em $[P_x]$

Caso 2 - $(i > j)$: adiciona-se $(y, j, \emptyset)$ em $[P_x]$

Caso 3 - $(i < j)$: nenhum candidato é gerado;

Como a proposta do algoritmo CobMiner é produzir padrões frequentes que satisfazem à restrição de um autômato de árvore local, torna-se desnecessário criar novos padrões candidatos a partir de elementos frequentes, porém que já não satisfazem nem mesmo o relaxamento da restrição do autômato. Fato este que reduz bastante o universo dos padrões candidatos que devem ser validados pelo suporte, acarretando em excelentes ganhos de performance de execução do processo de mineração, sem perda de informações.

O método de expansão de classes de equivalência gera corretamente todas as sub-árvores candidatas possíveis e, cada candidata é gerada uma única vez. Essa corretude foi definida no corolário 2 encontrado em (ZAKI, 2002) e provada através do método de indução. Uma vez que o método da *p-expansão de classes de equivalência p-válidas* possui o mesmo mecanismo, ou seja, o teorema original do Zaki é aplicado apenas para a expansão dos padrões p-válidos, garantimos que todos as sub-árvores candidatas que são possíveis de serem satisfeitas pela restrição são geradas.

Na Figura 34 temos uma classe de equivalência p-válida com prefixo $\mathcal{P}_{ab} = \text{ab}$, que contem os 2 elementos $(c, 1, \emptyset)$ e $(d, 0, tr_d)$. As p-expansões de cada elemento p-válido de $[P_{ab}]^{pv}$ dará origem a uma nova classe de equivalência p-válida. Seguindo a ordem da lista dos elementos de $[P_{ab}]^{pv}$, o elemento $(c, 1, \emptyset)$ é descartado pelo método da p-expansão de classes porque esse padrão não é p-válido, ou seja $tr_c = \emptyset$. Em seguida, considerando a p-expansão do próximo elemento, a junção $(d, 0, tr_d) \oplus (c, 1, \emptyset)$ direciona a geração para o caso 3 do teorema acima, não produzindo nenhum candidato. Por fim, os candidatos $(d, 0, \emptyset)$ e $(d, 2, \emptyset)$ são gerados pela junção $(d, 0, tr_d) \oplus (d, 0, tr_d)$, dando origem à nova

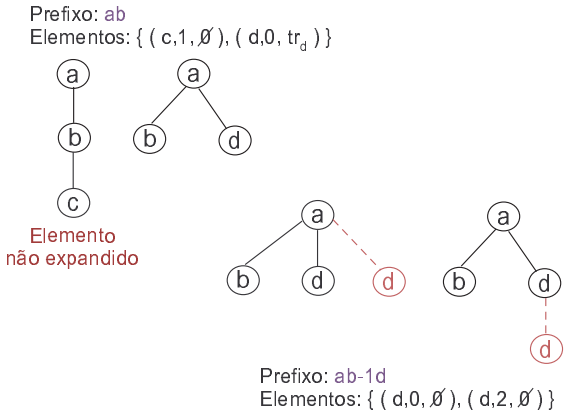


Figura 34: Geração de candidatos do CobMiner.

classe de equivalência p-válida $[P_{ab-1d}]^{pv}$.

Como vimos acima, basicamente, o processo de geração do CobMiner é similar ao utilizado pelo algoritmo TreeMiner porque a geração ocorre pela aplicação do operador junção entre seus elementos. A diferença está na redução do número de candidatos gerados pelo CobMiner, porque a sua fase de geração é direcionada pela restrição do autômato de árvore local, ou seja, o operador junção é aplicado apenas na p-expansão dos elementos p-válidos.

Observe que não mostramos o mapa de percurso dessa nova classe de equivalência ilustrada na Figura 34, porém suponha que ela seja uma classe de equivalência p-válida com respeito a uma restrição $\mathcal{H}$ de k-sub-árvores, como identificar quais desses padrões são p-válidos com relação à restrição $\mathcal{H}$? Ou seja, como é construído o mapa de percurso de uma classe de equivalência p-válida e o traço de percurso de seus elementos? As respostas a essas perguntas se encontram na seção seguinte.

4.2 Encontrando os Padrões P-válidos

Seja $[P]^{pv}$ uma classe de equivalência p-válida, criada pelo método da p-expansão de classes descrito acima. Em seguida, o algoritmo CobMiner calcula o suporte de cada candidato $(x, i, \emptyset) \in [P]^{pv}$, permanecendo em $[P]^{pv}$ somente os elementos que correspondem às sub-árvores frequentes. Antes de dar início à p-expansão dos elementos $[P]^{pv}$, o algoritmo CobMiner verifica quais dos elementos de $[P]^{pv}$ são p-válidos, criando os seus traços de percurso.

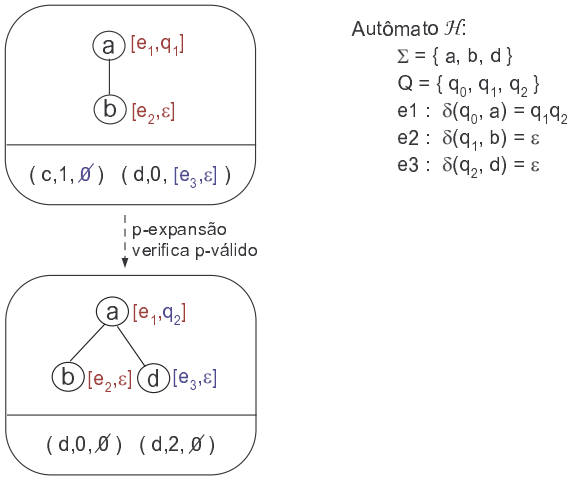
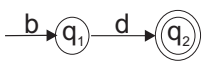


Figura 35: Mapa de Percurso de uma Classe de Equivalência P-válida.

Na Figura 35 temos uma nova representação para as classes de equivalência p-válidas mostradas na Figura 34. Aqui, os prefixos das classes estão representados como árvores e cada um dos seus nós possui explícito o seu traço de percurso.

Um traço de percurso é construído da seguinte maneira: considere que a classe de equivalência p-válida $[P_{ab}]^{pv}$, da Figura 35, foi criada e o suporte de seus candidatos calculados, sendo os elementos $(c, 1, \emptyset)$ e $(d, 0, \emptyset)$ frequentes. O algoritmo CobMiner tenta criar o traço de percurso do elemento $(c, 1, \emptyset)$ da seguinte maneira: ele percorre o mapa de percurso da classe $[P_{ab}]^{pv}$ em busca da expressão regular que esse elemento deve percorrer, ou seja, o nó c é filho de n_1 então pelo traço de percurso $tr(n_1)$, o nó c deve percorrer a expressão regular $e_2 = \epsilon$. Em seguida, ele tenta sem sucesso percorrer e_2 com o *label* c partindo de $tr(n_1).q = \epsilon$, permanecendo então $tr_c = \emptyset$. Para o elemento $(d, 0, \emptyset)$, o mesmo raciocínio é aplicado. Nesse caso, CobMiner identifica a expressão regular $e_1 = q_1q_2$ que é percorrida pelo nó d, a partir de $tr(n_0).q = q_1$ até o estado q_2. Sendo assim, os filhos de d devem percorrer a expressão regular $e_3 = \delta(q_2, d)$, sendo $tr_d = [e_3, \epsilon]$. Portanto, o elemento $(d, 0, [e_3, \epsilon])$ é p-válido. Na Figura abaixo mostramos o autômato finito que representa a expressão regular q_1q_2, que nos permite visualizar a transição $q_1 \xrightarrow{d} q_2$.



Ainda na Figura 33 temos a p-expansão dos elementos da classe de equivalência p-válida $[P_{ab}]^{pv}$. Somente o elemento $(d, 0, [e_3, \epsilon])$ deu origem a uma nova classe de equivalência p-válida, $[P_{ab-1d}]^{pv}$. O prefixo dessa nova classe corresponde ao $\mathcal{P}_{ab}$ acrescido no nó d como filho de n_0, sendo o traço de percurso $tr_d = [e_3, \epsilon]$ incorporado ao mapa de percurso do prefixo dessa nova classe. Observe que o traço de percurso de n_0 teve a

posição de percurso, $tr(n_0).q$, atualizada para q_2, somente quando d passou a fazer parte do prefixo. Suponha a situação em que $tr(n_0) = [e_j, q_1]$, onde $e_j = q_1(q_2 + q_3)$ e que $(c, 0, tr_c)$ também é um elemento p-válido de $[P_{ab}]^{pv}$, ou seja, existe a transição $q_1 \xrightarrow{c} q_3$ em e_j, além da transição $q_1 \xrightarrow{d} q_2$. Nesse caso, temos duas possibilidades para percorrer e_j com *labels* diferentes e uma nova tentativa de percurso em e_j ocorrerá apenas durante a p-expansão desses elementos, sendo por esse motivo $tr(n_0).q$ atualizado de acordo com o elemento que está sendo expandido para geração da nova classe.

Agora, podemos apresentar a estrutura do algoritmo CobMiner, discutida e exemplificada na próxima seção.

4.3 O Algoritmo CobMiner

Procedure CobMiner($\mathcal{D}$,σ_{min}, $\mathcal{A}$)
  F_1 = {padrões frequentes x_i de tamanho 1, $x \in L$}
  Repeat Until fim de F_1
    $tr(n_i) = \emptyset$
    If existe $\delta(q_0, x_i)$ em $\mathcal{A}$ **Then** $tr(n_i) = [\delta(q_0, x_i), \epsilon]$
    $V_1 = V_1 \cup tr(n_i)$
  F_2' = {padrões frequentes $x_i y_j$ de tamanho 2}
  Repeat Until fim de F_2'
    If $V_1[n_i] \neq \emptyset$ **Then**
      $[P]^{pv}.\mathcal{P} = x_i$
      $[P]^{pv}.mapa = V_1[n_i]$
      For all $y_j, x_i y_j \in F_2'$ **Do**
        $tr_y = PValid\text{-}Validate([P]^{pv}, n_j)$
        $[P]^{pv}.elem = [P]^{pv}.elem \cup (y, 0, tr_y)$
      $F_2 = F_2 \cup [P]^{pv}$
  For all $[P]^{pv} \in F_2$ **Do** *Enumerate-Frequent-PValid-Subtrees*($[P]^{pv}$)

Procedure Enumerate-Frequent-PValid-Subtrees($[P]^{pv}$)
  For each $(x, i, tr_x) \in [P]^{pv}$ **Do**
    If $tr_x \neq \emptyset$ **Then**
      New $[P_x]^{pv}$
      For each $(y, j, tr_y) \in [P]^{pv}$ **Do**
        $R = \{(x, i, tr_x) \oplus (y, j, tr_y)\}$
        $\mathcal{L}(R) = \{\mathcal{L}(x) \cap_{\oplus} \mathcal{L}(y)\}$
        For each frequent $(z, w, \emptyset) \in R$ **Do**
          $tr_z = PValid\text{-}Validate([P_x]^{pv}, (z, w, \emptyset))$
          $[P_x]^{pv}.elem = [P_x]^{pv}.elem \cup (z, w, tr_z)$
      Enumerate-Frequent-PValid-Subtrees($[P_x]^{pv}$)

Figura 36: O Algoritmo CobMiner

Na Figura 36 apresentamos o pseudo-código do algoritmo CobMiner. Como no algoritmo TreeMiner, nas duas primeiras iterações de execução, CobMiner varre em profundidade as árvores do banco de dados em busca das sub-árvores frequentes. Nas demais

iterações, o cálculo do suporte é feito através da operação de junção de listas de escopos, $\mathcal{L}(x) \cap_{\oplus} \mathcal{L}(y)$, e não sobre o banco de dados $\mathcal{D}$.

A discussão sobre o algoritmo CobMiner será conduzida sobre o mesmo banco de dados utilizado durante a apresentação do algoritmo TreeMiner. Sendo assim, considere a coleção de dados $\mathcal{D}$ apresentada na Figura 7 do capítulo 2, cujos *labels* pertencem ao conjunto $L = \{a, b, c, d, e\}$, o suporte mínimo $\sigma_{min} = 100\%$, e a restrição representada pelo autômato de árvore local $\mathcal{H} = (Q, \Sigma, \delta, q_0)$, onde $\Sigma = \{a, b, d\}$, $Q = \{q_0, q_1, q_2\}$, $\delta(q_0, a) = q_1 q_2$, $\delta(q_1, b) = \epsilon$ e $\delta(q_2, d) = \epsilon$.

Cálculo de F_1

CobMiner gera as sub-árvores frequentes de tamanho 1 como é feito pelo algoritmo TreeMiner. Nessa iteração, após a criação de F_1, existe a atividade de identificar quais padrões de F_1 são p-válidos com respeito a $\mathcal{H}$. Para isso, CobMiner utiliza-se de um vetor 1 x L, V_1, onde $V_1[i] = [\delta(q_0, label(n_i)), \epsilon]$ caso o nó n_i seja p-válido, caso contrário, $V_1[i] = \emptyset$. Pelo nosso exemplo, $F_1 = \{a, b, c, d\}$ e $V_1 = \{[e_1, \epsilon], \emptyset, \emptyset, \emptyset\}$, onde $e_1 = q_1 q_2$, ou seja, apenas a sub-árvore $a - 1$ é p-válida com respeito a $\mathcal{H}$. Na Figura 37 temos as árvores de F_1 que foram geradas pelo CobMiner.

Figura 37: Sub-árvores de F_1 geradas pelo CobMiner.

Cálculo de F_2

Nessa iteração, também não existe uma fase de geração de candidatos explícita e o cálculo do suporte permanece através da varredura do banco de dados $\mathcal{D}$ que atualiza o contador $M[i][j]$, sendo F_2' o conjunto de todas sub-árvores $x_i x_j - 1$ frequentes. As classes de equivalência p-válidas do conjunto F_2 são construídas a partir de F_2' e V_1 da seguinte maneira: se x_i é p-válido, ou seja, $V_1(i) \neq \emptyset$, então a classe $[P_x]^{pv}$ é criada em F_2. Considerando $F_2' = \{$ab-1, ad-1$\}$ gerado nessa iteração, e V_1 calculado na iteração anterior, ao final dessa iteração temos apenas a classe de equivalência p-válida $[P_a]^{pv} \in F_2$, onde $\mathcal{P}_a = a$, $[P_a]^{pv}.mapa = [e_1, \epsilon]$ e $[P_a]^{pv}.elem = \{(b, 0, [e_2, \epsilon]), (b, 0, \emptyset)\}$, onde $e_2 = \epsilon$. Na Figura 38 temos as 2-sub-árvores de F_2 geradas pelo CobMiner, observe que apenas a primeira delas é p-válida com respeito a $\mathcal{H}$.

Figura 38: Sub-árvores de F_2 geradas pelo CobMiner.

Enquanto o banco de dados é percorrido para calcular o suporte dos 2-padrões candidatos, como no TreeMiner, o algoritmo CobMiner cria as listas de escopos das sub-árvores de F_1, estruturas estas utilizadas em seguida para construir as listas de escopo das sub-árvores frequentes de F_2 através da *Junção das Listas de Escopo*, $\mathcal{L}(X) \cap_{\oplus} \mathcal{L}(Y)$, operação utilizada para o cálculo do suporte dos candidatos das iterações seguintes, e discutida no capítulo 2, onde o algoritmo TreeMiner é apresentado.

Cálculo de F_k, $k \geq 3$

A partir da iteração 3, o processo de mineração do algoritmo CobMiner ocorre recursivamente através do procedimento *Enumerate-Frequent-PValid-Subtrees*, que recebe como parâmetro de entrada uma classe de equivalência p-válida $[P]^{pv}$, junto com as listas de escopos de seus elementos. A p-expansão de cada elemento p-válido x_i de $[P]^{pv}$ produz a nova classe de equivalência p-válida, $[P_x]^{pv}$, que em seguida é enviada como parâmetro a uma nova chamada de *Enumerate-Frequent-PValid-Subtrees*. Note que essa idéia de execução recursiva é similar ao procedimento *Enumerate-Frequent-Subtrees* utilizado pelo TreeMiner. A diferença pode ser observada na Figura 36, onde verificamos que somente os elementos p-válidos de uma classe de equivalência são p-expandidos para geração de novas classes e, após a validação do suporte, os novos elementos p-válidos são identificados pelo procedimento *PValid-Validate*, cujo mecanismo foi explicado na seção 4.2 deste capítulo.

Na Figura 39 encontramos a classe $[P_{ab}]^{pv}$ de sub-árvores de tamanho 3, geradas a partir dos elementos de $[P_a]^{pv} \in F_2$. Como apenas o elemento $(b, 0, tr_b) \in [P_a]^{pv}$ é p-válido, apenas ele foi p-expandido para a geração de novos candidatos. Considerando o mesmo cenário de mineração porém aplicando o algoritmo TreeMiner, o elemento $(d, 0, \emptyset) \in [P_a]^{pv}$ também seria expandido, dando origem à classe $[P_{ad}]^{pv}$, e os novos candidatos gerados a partir dele também teriam o suporte cálculado, mesmo sendo esses padrões irrelevantes para o usuário, uma vez que nem mesmo o prefixo dessa classe, $ad - 1$, é satisfeito pelo relaxamento da restrição $\mathcal{H}$, ou seja, $ad - 1$ não é p-válido com respeito a $\mathcal{H}$.

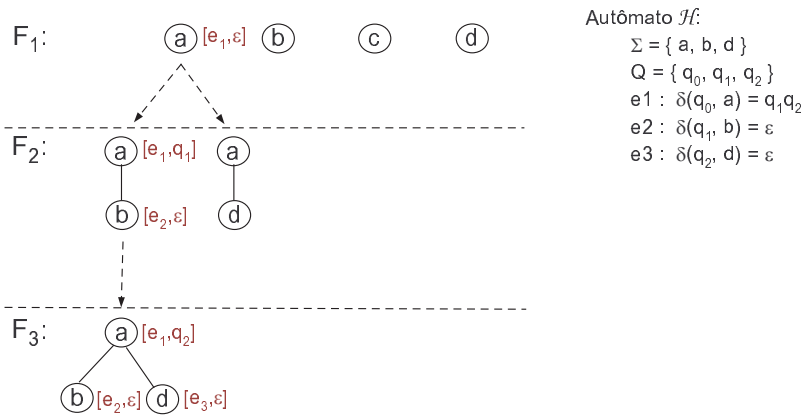


Figura 39: Classes de F_3 geradas a partir de $[P_a]^{pv} \in F_2$.

Seguindo a pilha da recursão da execução do CobMiner, F_4 é gerado vazio e, em F_3 não existe nenhum outro elemento p-válido com respeito a $\mathcal{H}$ para ser p-expandido, finalizando assim o processo da busca por padrões frequentes e p-válidos do algoritmo CobMiner. O próximo passo então é verificar quais padrões frequentes e p-válidos são também *válidos com respeito a* $\mathcal{H}$, sendo estes os padrões de real interesse do usuário.

Essa validação ocorre da seguinte maneira, para cada padrão p-válido $S_i \in F$, onde $F = F_1 \cup F_2 \cup F_3$, o traço de percurso de cada nó $n_j \in S_i$ é percorrido verificando se $tr(n_j).q$ é um estado de reconhecimento final da expressão regular $tr(n_j).e$. Considerando o nosso exemplo de mineração ilustrado na Figura 39, o padrão S_0 = a-1 é p-válido mas não é válido com respeito a $\mathcal{H}$, porque a expressão regular $tr(n_0).e = e_1 = q_1q_2$ não chegou a ser percorrida, ou seja $tr(n_0).q = \epsilon$, sendo q_2 o seu estado final de reconhecimento. Já o padrão S_2 = ab-1d-1 é válido com respeito a $\mathcal{H}$ porque a expressão regular $tr(n_0).e = e_1 = q_1q_2$ foi percorrida até o seu estado de reconhecimento final $tr(n_0).q = q_2$, e para os nós folhas n_1 e n_2, suas expressões regulares não foram percorridas, porém ϵ é o estado de reconhecimento final das expressões regulares $e_2 = e_3 = \epsilon$.

	CobMiner	**TreeMinerPP**
etapa k	***N°Cand***	***N°Cand***
1	5	5
2	16	16
3	4	8
4	2	2

Tabela 1: Quantidade de Padrões gerados - CobMiner X TreeMinerPP.

Apesar de $\mathcal{D}$ ser um banco de dados pequeno, na Tabela 1 podemos observar que na iteração 3, CobMiner gerou menos candidatos do que o TreeMiner PP. Quanto aos ganhos na performance de execução do processo de mineração recursivo, no capítulo seguinte,

comparamos as execuções dos algoritmos CobMiner e TreeMinerPP, onde discutimos o resultado dos testes efetuados sobre dados sintéticos e um pequeno ensaio de mineração sobre um documento XML.

4.4 A Poda Oportunista de Candidatos com Restrição

O algoritmo CobMiner também generalizou a poda *oportunista* implementada no TreeMiner, para podar candidatos mediante a existência de uma restrição. Ao avaliar se um k-padrão S candidato pode ser desprezado, como no TreeMiner, o algoritmo CobMiner gera todos os sub-padrões s de tamanho k-1 a partir de S, caso exista um sub-padrão x_i que deveria ter sido gerado e $x_i \notin F_{k-1}$, CobMiner poda x_i apenas se x_i for p-válido. Caso contrário o padrão S permanece como candidato. Porque a necessidade de verificar se x_i é p-válido?

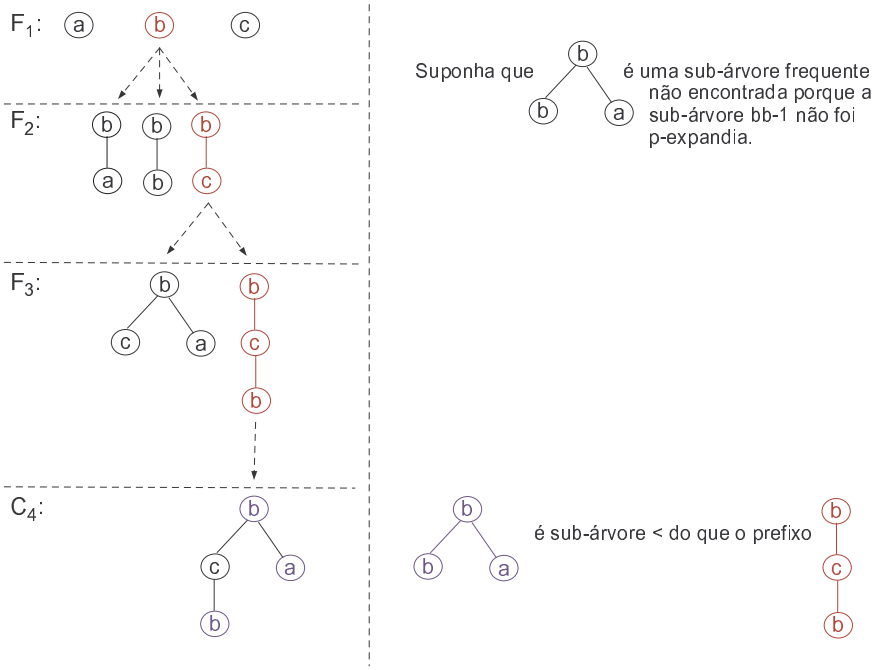


Figura 40: Poda Oportunista de Candidatos do CobMiner.

Na Figura 40, temos um processo de mineração ilustrado. Observe que a geração dos novos candidatos ocorre apenas pela p-expansão dos padrões p-válidos, destacados na Figura pela cor vermelho. Considere o 4-candidato $bcb-1-1a-1$, gerado a partir de $bcb-1-1$, sendo este candidato pertencente à nova classe de equivalência p-válida de prefixo bcb. Aplicando a poda oportunista do algoritmo TreeMiner, esse candidato seria podado porque a sub-árvore $bb-1a-1 \notin F_3$, pois apesar de ser frequente, não foi

gerada porque o elemento $bb-1$ não é p-válido. Podar esse padrão pode acarretar em perda de informações, considerando ser este padrão totalmente satisfeito pela restrição do usuário. Sendo assim, aplicando a generalização da poda oportunista utilizada pelo CobMiner, como $bb-1a-1$ não é p-válido então $bcb-1-1a-1$ não é podado.

5 Os Resultados Experimentais

Todos os testes foram executados em um computador *Pentium* 4 com 3GHz de velocidade de processamento e 1GB de mémoria RAM, com sistema operacional Suse Linux 9.2.

5.1 Gerador de Dados Sintéticos

Criamos um gerador de dados sintéticos que produz banco de dados arborescentes. Esse programa possui três etapas de processamento, sendo elas: construção da gramática de árvore local, geração dos padrões arborescentes e construção do banco de dados propriamente dito. Cada uma dessas fases é caracterizada por produzir dados de acordo com parâmetros fornecidos pelo usuário.

Na primeira etapa, o gerador constrói uma gramática de árvores local G, recebendo como entrada seis parâmetros: K, T, E, k, t, e. Na fase seguinte, ele recebe os parâmetros: F, f_m, P, p_m, além da gramática local G criada anteriormente, produzindo N padrões arborescentes. Enfim, o banco de dados sintéticos é gerado na última etapa, utilizando os parâmetros: M, H, R, μ e os N padrões gerados na etapa anterior. Em elevado nível de abstração, a seguir apresentamos a idéia geral das etapas de processamento do gerador, contextualizando cada um desses parâmetros e mostrando como eles são trabalhados durante a geração dos dados sintéticos.

Inicialmente, as expressões regulares são construídas, de acordo com os parâmetros: número máximo de blocos internos (k), número máximo de termos por bloco interno (t) e o número máximo de transações por termo interno (e), responsáveis pela definição da estrutura das mesmas. Assim, sobre o conjunto de estados $Q = \{q_0, \ldots, q_x\}$, $x > 0$, construimos o conjunto de expressões regulares $ER = \{er_1, er_2, \ldots\}$ tal que cada er_i é do tipo $B_1{*} \ldots B_y{*}$, com $y \leq k$, cada B_i é da forma $T_1 T_2 \ldots T_n$, com $n \leq t$, e cada $T_j = (q_1 + q_2 + ... + q_l)$, com $l \leq e$. O gerador de gramática de árvores locais utiliza essas

expressões regulares e o conjunto de alfabetos $L = \{a_1, \ldots, a_w\}$, $w > 0$, que representa os *labels* das árvores.

As características que a gramática local G deve assumir são definidas pelos parâmetros: número máximo de blocos externos (K), número máximo de termos para cada bloco externo (T) e o número máximo de transações por termo externo (E). As regras de uma gramática são do tipo $q \rightarrow a(er)$, com $q \in Q$, $a \in L$, e $er \in ER$. A primeira regra de G é $q_0 \rightarrow a_0(er_0)$. Apenas um estado pertencente a er_0 (didaticamente o chamaremos de q_1) terá uma regra do tipo $q_1 \rightarrow a_1(er_1)$, onde a_1 é um *label* não utilizado ainda em nenhuma regra gerada anteriormente para G, restrição esta que garante G local. A escolha do estado q_1 em er_0 é aleatória, desde que q_1 não tenha sido o estado escolhido em passos anteriores. Todos os demais estados q_i pertencentes a er_0, terão regras do tipo $q_i \rightarrow a_i\epsilon$. O mesmo processo descrito acima é aplicado para a regra $q_1 \rightarrow a_1(er_1)$ e demais regras que surgirem durante os K passos de execução, retornamos ao estado inicial q_0, fechando o ciclo de geração de K grupos de regras para G.

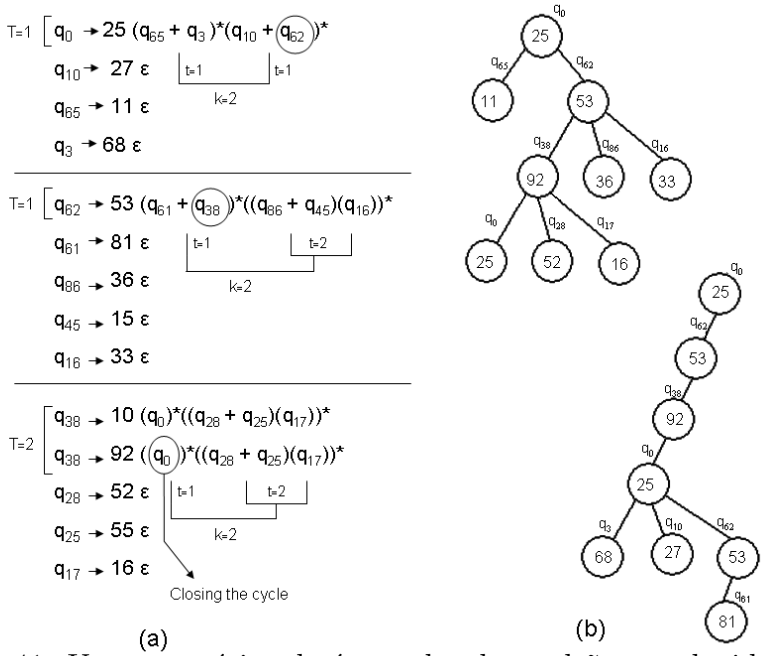


Figura 41: Uma gramática de árvore local e padrões produzidos por ela.

Na Figura 41(a) temos uma gramática de árvore local, obtida pelos parâmetros $K = 3, T = 2, E = 1, k = 2, t = 2$ e $e = 2$. Cada grupo de regras é delimitado por uma linha horizontal. O parâmetro T indica o número máximo de regras que podem ser definidas para o primeiro estado $q_i \in Q$ de cada grupo de regras. E por fim, o parâmetro E indica o número máximo de regras que podem ser geradas para os demais estados de cada grupo.

Ainda nessa primeira fase, o autômato de árvore local é construído a partir da gramática de árvore G. As funções de transição desse autômato são do tipo $\delta(q, a) = er$, deduzida a partir da regra da gramática $q \rightarrow a(er)$.

Considerando a segunda fase de processamento do gerador de dados sintéticos, a gramática local G é capaz de produzir padrões arborescentes respeitando seu conjunto de regras. Para a geração de N padrões, a gramática é percorrida N vezes respeitando os parâmetros: *fanout* máximo dos padrões (F), *fanout* médio dos padrões (f_m), profundidade máxima dos padrões (P) e profundidade média dos padrões (p_m), e utilizando conceitos estatísticos, que garantem a geração de padrões de forma distribuída e com características variadas, sustentando uma diversidade que torna os testes mais consistentes.

Na Figura 41(b) temos 2 exemplos de padrões gerados a partir da gramática G ilustrada na Figura 41(a). A partir da reqra definida para o estado inicial q_0, o *label* 25 encontrado dá origem ao nó raiz do padrão em construção. Seus filhos são os nós cujos *labels* são definidos pela expressão regular definida em $\delta(q_0, 25)$. O nível seguinte é construído aplicando o mesmo raciocínio para um único nó filho da raiz, aquele originado do estado escolhido para ser expandido durante a geração da gramática. E assim demais níveis do padrão são construídos.

Finalmente, precisamos imergir tais padrões em uma massa de dados. Para tanto, são parâmetros dessa fase: a quantidade de árvores do banco de dados (M), o *fanout* máximo das árvores (H), a profundidade máxima das árvores (R) e um fator de proporção, μ, que determina, a partir do total de árvores, o percentual delas que não contêm padrões. Com isso, tem-se um banco de dados modelado a partir de valores previamente estabelecidos, que podem ser reajustados em qualquer etapa, já que estas são independentes. Ou seja, é possível gerar bancos contendo os mesmos padrões, porém com tamanhos diferentes, ou ainda, gerar uma nova base de padrões a partir da mesma gramática, e assim por diante.

Os parâmetros que definimos como *default* para o gerador de dados encontram-se discriminados na Tabela 2. Para os testes apresentados nesse capítulo, utilizamos 13 bancos de dados sintéticos diferentes, gerados por 7 gramáticas distintas. Utilizamos a notação DB-xy para referenciar o banco de dados utilizado, onde x corresponde ao parâmetro variado durante a geração do banco de dados, e y, qual é esse novo valor. Sendo assim, DB-M200 corresponde ao banco de dados com 200(x1000) árvores e DB-K1, onde o parâmetro K teve seu valor *default* alterado para 1. DB-DEF é o banco de dados gerado com os valores *default* da tabela para todos os parâmetros do gerador. Seguindo o

mesmo raciocínio, denotamos as restrições de autômato como A-zw, onde z corresponde ao parâmentro variado e w ao seu novo valor. Novamente, A-K1 corresponde ao autômato de árvore local gerado pela gramática onde o parâmetro K teve seu valor *default* alterado para 1, e assim por diante.

Parâmetro da Gramática		**Valor *Default***
k	Nº máximo de blocos internos	1
t	Nº máximo de termos internos	4
e	Nº máximo de transações por termo interno	3
K	Nº máximo de blocos externos	2
T	Nº máximo de termos externos	2
E	Nº máximo de transações por termo externo	2
Parâmetro dos Padrões		**Valor *Default***
N	Nº de padrões	5,000
F_m	*Fanout* médio	3
F	*Fanout* máximo	5
P_m	Profundidade média	3
P	Profundidade máxima	5
Parâmetro do Banco de Dados		**Valor *Default***
M	Nº de transações (árvores de dados)	100,000
H	*Fanout* máximo	10
R	Profundidade máxima	10
μ	% de árvores do banco que não satisfazem a gramática	50

Tabela 2: Parâmetros *Default* do Gerador de Dados Sintéticos.

5.2 Dados Reais: Documentos XML

Fizemos testes de execução dos algoritmos CobMiner e TreeMinerPP em 4 banco de dados reais distintos: DB-SigMod, DB-People, DB-Mains e DB-Casts, construídos a partir de documentos XML.

O banco de dados DB-SigMod, consiste de árvores geradas a partir do documento SigmodRecord.xml (Apêndice A), obtido no site XML Data Repository[1]. Esse arquivo contém índices para artigos do site Sigmod Record. A base de dados é formada por sub-árvores desse documento, tendo como raiz a *tag* <*article*>, o que nos forneceu um banco com 1504 árvores. O autômato construído para esses testes, A-SigMod (Apêndice B), restringiu a estrutura dos padrões de forma a minerar artigos que possuam uma página inicial, uma página final e um autor, além de fazer algumas restrições com relação ao conteúdo desses itens.

DB-People é um banco de dados com 3479 árvores, extraído do documento XML people55.xml pertencente ao repositório *Movies Database* [2]. Esse documento possui uma

[1]http://www.cs.washington.edu/research/xmldatasets/data/sigmod-record/SigmodRecord.xml
[2]http://infolab.stanford.edu/pub/movies/people55.xml.

lista de produtores e diretores de filmes americanos, e algumas informações sobre os demais membros da equipe de produção desses longa metragens. A raiz dessas árvores corresponde à *tag* <*person*> e, como restrição ao processo de mineração, construimos o autômato de árvore A-People (Apêndice C). Os padrões frequentes minerados devem possuir a seguinte estrutura: pessoas que possuem no seu cadastro as sequintes informações: nome, função, ano de início do trabalho, ano de finalização trabalho, nome da família, apelido, data de nascimento, data de falecimento, local do nascimento e que tenha sido colega de trabalho de Hitchcock, Mirta Ibarra, Elizabeth Montegomery ou Bertolucci.

Outro banco de dados que utilizamos é o DB-Mains, construído a partir do documento mains243.xml também obtido do site *Movie Database* [3]. Nesse XML encontramos uma lista de filmes, sendo a raiz de cada uma das 12114 árvore do DB-Mains correspondente à *tag* <*film*>. Utilizamos as restrições do autômato de árvore A-Mains (Apêndice D) durante o processo de mineração, que representa o interesse somente pelos filmes que possuem título, diretor, produtor, processo de filmagem e que foram produzidos em 1975, 1976, 1998 ou 1999.

E por último, temos o banco de dados DB-Casts com 48937 árvores geradas a partir do arquivo casts124.xml [4]. Para esse banco, a raiz de suas árvores é a *tag* <*m*> que corresponde às informações de cada membro do grupo de atores de um filme, e o autômato A-Casts (Apêndice E) restringe a mineração aos atores que possuem as sequintes informações: identificador do filme (<*f*>), título do filme (<*t*>), nome do ator (<*a*>), descrição do papel do ator no filme (<*r*>), nome do personagem (<*rname*>) e cujo tipo do papel (<*p*>) não está definido (<*und*>).

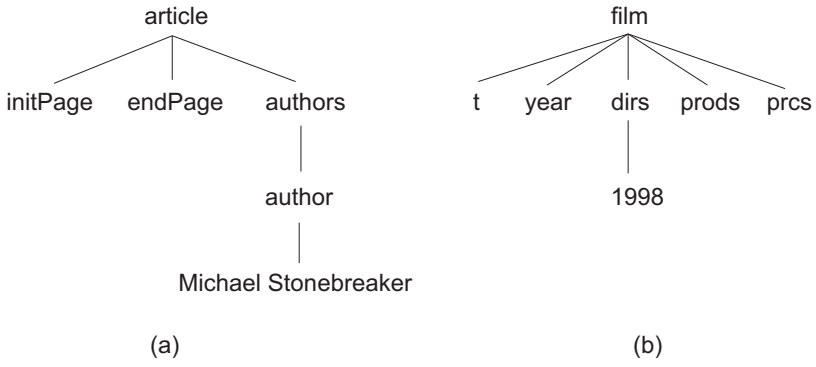


Figura 42: Padrões minerados a partir de documentos XML.

Nas Figuras 42(a) e 42(b) temos exemplos de padrões frequentes válidos, encontrados durante a minerado sobre os bancos de dados DB-SigMod e DB-Mains, respectivamente.

[3]http://infolab.stanford.edu/pub/movies/mains243.xml.
[4]http://infolab.stanford.edu/pub/movies/casts124.xml.

5.3 Dados Reais: Log de Navegação na Web

Fizemos uma pequena simulação da mineração do CobMiner e TreeMinerPP em um banco de dados reais, construído a partir de uma semana de logs de acesso e navegação no site da UFU [5]. A partir desse arquivo, geramos o banco de dados DB-UFULog com 2686 árvores. O autômato de árvore local A-UFULog (Apêndice F) representa o interesse em padrões onde as sessões de usuários começam pela página "Pesquisa e Pós-graduação", seguindo para a página "Pós-graduação Strictu Sensu"onde as disciplinas da área de educação são percorridas. Padrões onde a página "Pós-graduação Lato Sensu"foi visitada também são interessantes.

Na Figura 43 temos exemplos de padrões minerados a partir do banco de dados DB-UFULog.

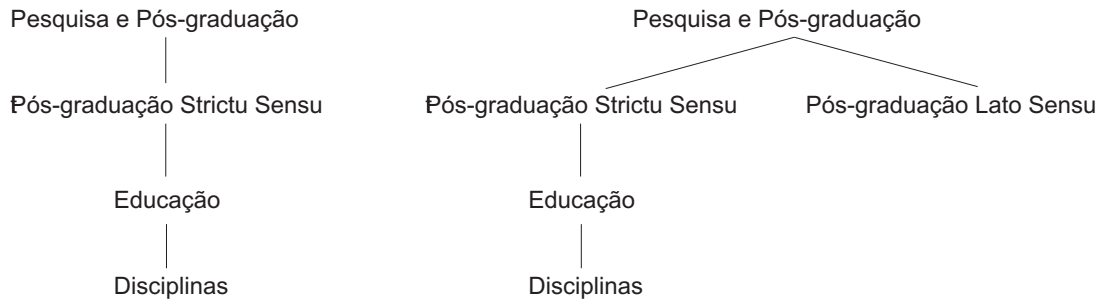


Figura 43: Padrões frequentes e válidos do banco DB-UFULog.

5.4 Análise de Performance

Na Figura 44(a) ilustramos a performance do CobMiner X TreeMinerPP. Os dois algoritmos foram executados com 4 variações de suporte mínimo, sobre o banco de dados DB-DEF e com as restrições do A-DEF. Considerando o suporte mínimo de 5%, o algoritmo CobMiner é 2 vezes mais rápido do que TreeMinerPP, sendo que essa proporção aumenta consideravelmente com o decréscimo do suporte, chegando à razão de aproximadamente 8 na execução com suporte de 1%. A explicação dessa grande diferença na performance entre os 2 algoritmos está na restrição do universo de mineração do CobMiner. O processo de mineração é direcionado pelo autômato de árvore, ou seja, somente padrões p-válidos de tamanho k são p-expandidos na geração dos padrões da etapa seguinte. Se não há nenhum $k-$padrão p-válido, o processo de mineração é finalizado. O TreeMinerPP,

[5]Universidade Federal de Uberlândia - http://www.ufu.br.

gera todos os padrões frequentes para depois filtrar esse conjunto, produzindo um novo resultado contendo apenas os padrões válidos.

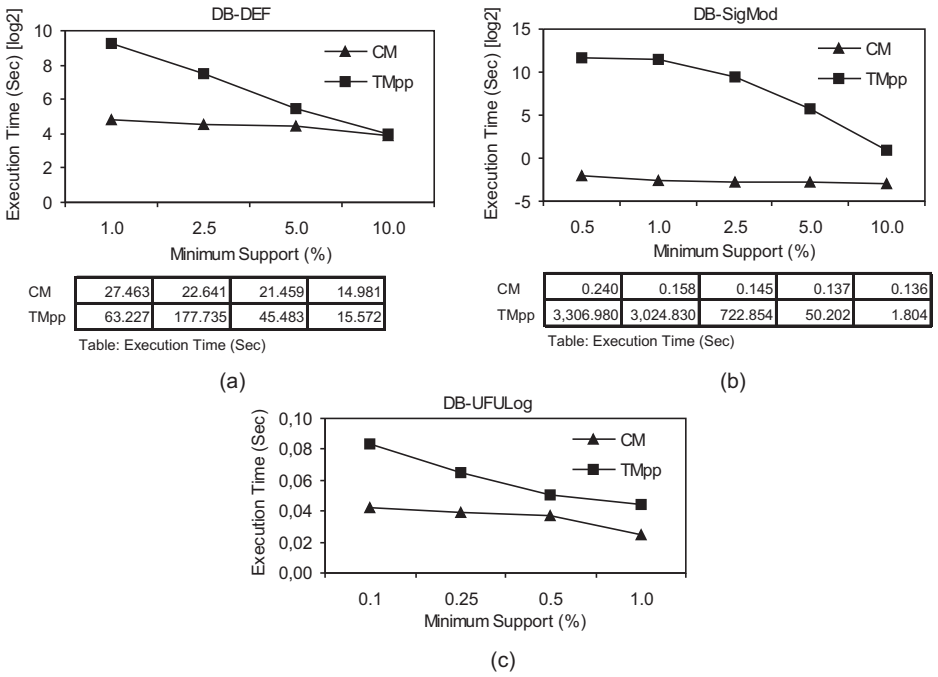


Figura 44: Análise da Performance em Tempo de Execução.

A superioridade da performance do CobMiner também pode ser constatada nas Figuras 44(b) e 44(c), que retratam as execuções dos 2 algoritmos com variações do suporte, sobre os banco de dados DB-SigMod e DB-UFULog, e restrições dos autômato A-SigMod e A-UFULog, respectivamente. Observem que o TreeMinerPP apresentou uma performance muito melhor nas execuções sobre o banco DB-DEF, com 100.000 árvores, em relação às execuções sobre o DB-SigMod, que possui 1504 árvores, comportamento justificado pelo fato das árvores de DB-SigMod possuir mais de um nó de um mesmo nível, com descendentes com *fanout* máximo encontrado de 41, enquanto que no banco de dados DB-DEF, somente um nó de cada nível possui filhos e *fanout* máximo é 10.

Uma visão da performance, considerando o número de candidatos gerados, durante a mineração sobre os bancos de dados DB-DEF, DB-SigMod e DB-UFULog, encontra-se nas Figuras 45(a), 45(b) e 45(c). Esses gráficos mostram que o número de candidatos gerados é um dos fatores que influencia diretamente no tempo de execução desses algoritmos, o que fundamenta a análise descrita acima da superioridade da performance do CobMiner, que gera muito menos candidatos do que o algoritmo TreeMinerPP.

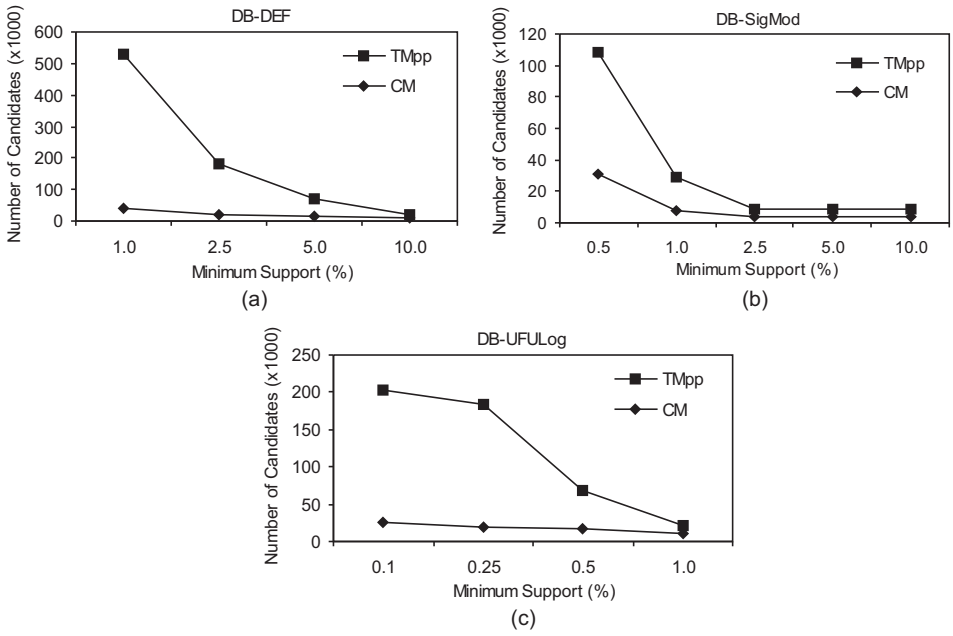


Figura 45: Análise da Performance em Número de Candidatos Gerados.

O comportamento dos algoritmos CobMiner e TreeMinerPP foram analisados também em execuções sobre bancos com 100.000 árvores, variando o μ e os parâmetros da gramática K, que está diretamente relacionado à profundidade dos padrões, e o k que influencia na largura. O resultado desses testes são mostrados nas Figuras 46 e 47, onde observamos também que em todos eles o CobMiner teve performance superior em relação ao TreeMinerPP.

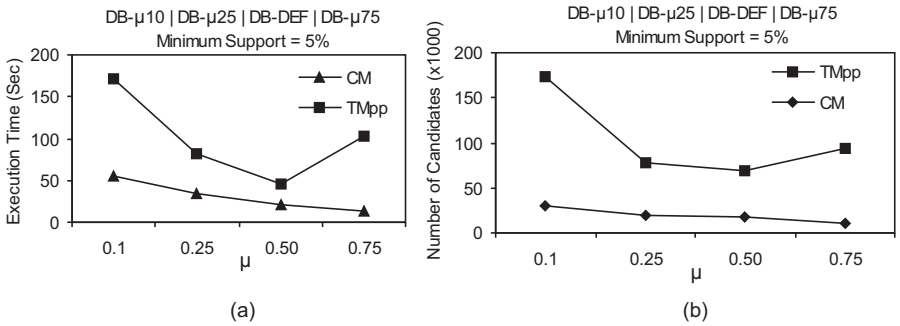


Figura 46: Análise da Performance com Variação do μ.

Na Figura 46(a) temos os tempos de execução sobre bancos de dados com variação do μ. Até $\mu = 50\%$, tanto CobMiner quanto o TreeMinerPP apresentaram queda no tempo de execução. Para o CobMiner, essa queda permanece até $\mu = 75\%$ e é justificada pela

diminuição do número de árvores que satisfazem os padrões válidos pelo autômato, o que acarreta também a diminuição do número de padrões frequentes que são gerados. No caso do TreeMinerPP, a queda no tempo de execução também está atrelada ao número de padrões frequentes gerados, porque até $\mu = 50\%$, temos 50% da base de dados com árvores que satisfazem os padrões frequentes da gramática e, o % restante corresponde a árvores que foram geradas aleatoriamente, porém mantendo o cuidado de não satisfazerem aos padrões, o que impacta na frequência de possíveis padrões frequentes que poderiam ser gerados. Com $\mu > 50\%$, essa aleatoriedade favorece a geração de mais padrões frequentes não válidos e que passam pela validação do autômato, o que justifica esse acréscimo de tempo de execução.

Essa discussão sobre a variação do μ e o impacto dele na geração de padrões pode ser verificada na Figura 46(b). O número de candidatos gerados pelo TreeMinerPP aumentou com o $\mu = 75\%$, sendo esses novos candidatos gerados não satisfeitos pelo restrição.

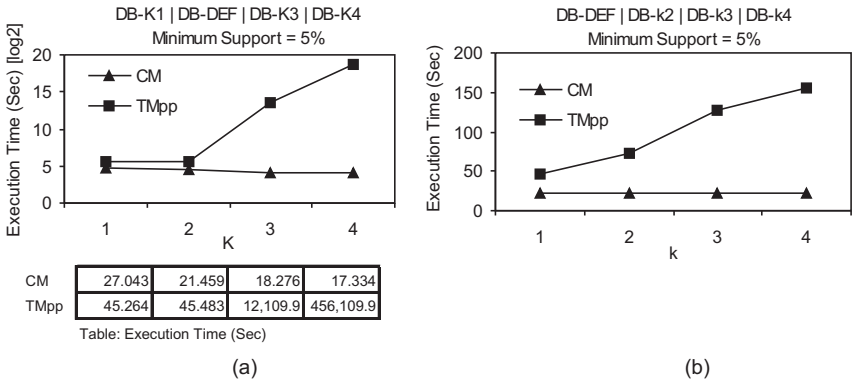


Figura 47: Análise da Performance com Variação da Gramática.

Analisando agora os testes efetuados com as variações de parâmetros da gramática, temos que o número máximo de blocos externos K influencia diretamente na profundidade dos padrões, e consequentemente na profundidade das árvores que os suportam. Com o aumento do K, identificamos uma perda de performance na execução de ambos os algoritmos porém com acentuado impacto na execução do TreeMinerPP, que se justifica no fato do processo de mineração dos 2 algoritmos ser recursivo em profundidade. Enquanto o CobMiner trabalha com uma pilha de recursão limitada pela validação do autômato, o TreeMinerPP, executa a recursão até que nenhum padrão frequente possa ser gerado. O mesmo comportamento é observado com a variação do número máximo de blocos internos da gramática que influencia na largura dos padrões e das árvores do banco, porém, o aumento da performance do TreeMinerPP nesse caso segue uma curva linear. Os tempos

de execução dos testes variando os parâmetros K e k estão respectivamente ilustrados nas Figuras 47(a) e 47(b).

Fizemos testes variando o número de padrões dos bancos de dados, onde a superioridade do CobMiner também foi evidenciada. Analisando os tempos de execução de cada um dos algoritmos separadamente, sobre suporte mínimo de 5%, o aumento do número de padrões dos bancos não impactou no tempo de execução dos algoritmos, porque, conforme N aumentava, o número de árvores do banco de dados que satisfazem cada padrão era decrescido, com o objetivo de manter o tamanho dos banco de dados em 100.000 árvores.

5.5 Análise de Escalabilidade

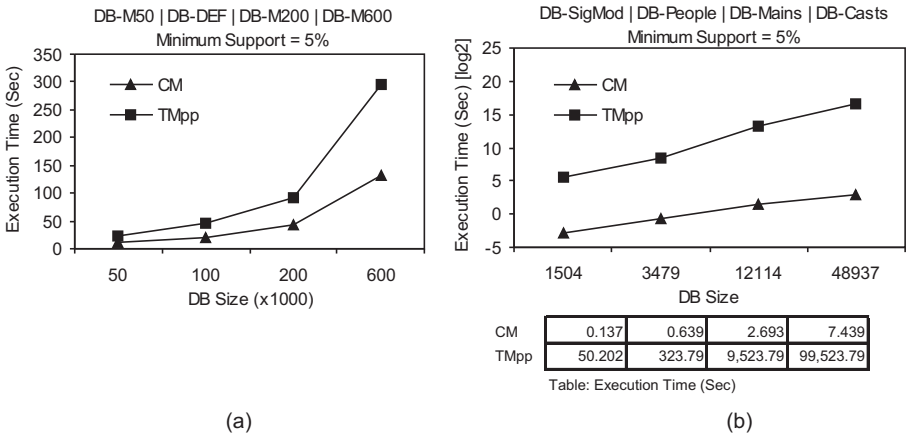


Figura 48: Análise da Escalabilidade.

A Figura 48(a) mostra como os algoritmos CobMiner e TreeMinerPP escalonaram suas execuções com o aumento do número de árvores do banco de dados de 50.000 a 600.000 árvores. Considerando o suporte mínimo de 5% e as restrições do autômato de árvore local A-DEF, notamos um comportamento exponencial no aumento do tempo de execução dos algoritmos, conforme o banco de dados cresce, embora CobMiner continua sendo 2 vezes mais rápido do que o algoritmo TreeMinerPP.

A escalabilidade também foi avaliada com os bancos de dados gerados a partir dos documentos XML. Na Figura 48(b) podemos observar que o tempo de execução dos algoritmos, nesse contexto, também está diretamente relacionado ao tamanho dos bancos de dados, sendo o CobMiner mais performático.

5.6 O efeito da Poda Oportunista

O efeito da poda no tempo de execução dos algoritmos CobMiner e TreMinerPP também foi avaliado durante os testes. Nesse cenário, CobMiner também é mais performático do que o algoritmo TreeMinerPP. Observe que TreeMinerPP sempre se beneficia da fase da poda de candidatos, enquanto que o mesmo comportamento não acontece com o algoritmo CobMiner. Existem duas razões para esse fato. Primeira delas, a geração de candidatos através da p-expansão de classes de equivalência apenas gera novos candidatos a partir de padrões p-válidos, o que já reduz consideravelmente o universo de candidatos gerados, deixando de produzir sub-árvores a partir de padrões que não satisfazem às restrições do autômato, apesar de serem frequentes. Segunda, como o número de sub-árvores candidatas possíveis de serem podadas é baixo devido à primeira razão explicada acima, existe o custo da execução do teste da poda para todos os candidatos, o que onera o tempo de execução do algoritmo CobMiner.

A tabela 3 mostra o número de candidatos gerados e os tempos de execução do algoritmo CobMiner, sobre o banco de dados DB-DEF, com e sem a fase da poda de candidatos, com $1\% \leq \sigma_{min} \leq 10\%$ e restrições do autômato de árvore local A-DEF. Analisando a execução do CobMiner considerando o $\sigma_{min} = 1\%$, somente 255 sub-árvores candidatas foram podadas, porém o teste da poda aumentou o tempo de execução do algoritmo.

	Tempo Execução (Seg)		**Nº Cand. Gen.**	
minsup	**Sem Poda**	**Com Poda**	**Sem Poda**	**Com Poda**
0,010	27,463	27,988	38681	38426
0,025	22,641	22,365	19177	18901
0,050	21,459	21,592	16911	16809
0,100	14,981	14,995	10215	10215

Tabela 3: Avaliação da Poda Oportunista do CobMiner, execuções sobre DB-DEF.

6 Conclusão e Trabalhos Futuros

A mineração de padrões arborescentes em coleções de dados semi-estruturados constitui um importante problema em mineração de dados. Excelentes trabalhos que apresentam soluções para esse contexto foram apresentados recentemente. Nessa dissertação, introduzimos o uso das restrições representadas por autômato de árvores locais, como aprimoramento da performance de execução do processo da mineração de árvores, bem como na geração de padrões frequentes mais direcionados aos anseios dos usuários.

Apresentamos o algoritmo CobMiner, que avalia as restrições durante a fase de geração de candidatos, e comparamos sua performance de execução com o algoritmo TreeMiner acrescido de uma fase de pós-processamento, para validação dos padrões frequentes com a restrição, o TreeMinerPP. Testes exaustivos foram efetuados em dados sintéticos e em todos os cenários, a superioridade do CobMiner foi incontestada, fechando o nosso trabalho com o cumprimento do objetivo proposto.

Um sistema de mineração de dados reais foi construído, dados estes naturalmente estruturados em árvores como os documentos XML e logs de navegação na web, cujo algoritmo de mineração é o CobMiner, algoritmo apresentado nessa dissertação (AMO; FELICIO, 2007).

Como trabalho futuro, propomos um terceiro algoritmo, CobMinerV, o qual consiste em modificar a fase da geração de candidatos do algoritmo CobMiner para produzir, a cada passo de execução, padrões válidos com respeito à restrição do usuário. Como identificamos nos testes que a fase da poda é pouco relevante para a performance do algoritmo CobMiner, a idéia de usar uma restrição mais forte (padrões devem ser válidos e não p-válidos) durante a geração de candidatos, e eliminando a fase de pós-processamento, onde os padrões válidos são encontrados dentre os padrões p-válidos minerados, espera-se que um ganho de performance possa acontecer.

Referências

AGRAWAL, R.; SRIKANT, R. Fast algorithms for mining association rules. In: BOCCA, J. B.; JARKE, M.; ZANIOLO, C. (Ed.). *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*. [S.l.]: Morgan Kaufmann, 1994. p. 487–499. ISBN 1-55860-153-8.

AGRAWAL, R.; SRIKANT, R. Mining sequential patterns. In: YU, P. S.; CHEN, A. S. P. (Ed.). *Eleventh International Conference on Data Engineering*. Taipei, Taiwan: IEEE Computer Society Press, 1995. p. 3–14.

AMO, S. de; FELICIO, C. Z. Using tree automata for xml mining and web mining with constraints. In: *III Workshop on DataMining Algorithms and Applications, João Pessoa, Brazil*. [S.l.: s.n.], 2007.

AMO, S. de; FURTADO, D. A. First-order temporal pattern mining with regular expression constraints. In: *20th Brazilian Symposium on Databases, Uberlândia, Brazil*. [S.l.: s.n.], 2005. p. 280–294.

AMO, S. de et al. Constraint-based tree pattern mining. In: *22th Brazilian Symposium on Databases, João Pessoa, Brazil*. [S.l.: s.n.], 2007. p. 317–331.

ASAI, T. et al. Efficient substructure discovery from large semi-structured data. In: *SDM '02: Proceedings of the Second SIAM International Conference on Data Mining 2002*. [S.l.: s.n.], 2002. p. 158–174.

CHAKRABARTI, S. Data mining for hypertext: a tutorial survey. *SIGKDD Explor. Newsl.*, ACM Press, New York, NY, USA, v. 1, n. 2, p. 1–11, 2000. ISSN 1931-0145.

CHAMBERLIN, D. Xquery: A query language for xml. In: *SIGMOD '03: Proc. ACM SIGMOD Int. Conf. on Management of Data*. New York, NY, USA: ACM Press, 2003. p. 682–682. ISBN 1-58113-634-X.

CHEVALET, C.; MICHOT, B. An algorithm for comparing rna secondary structures and searching for similar substructures. *Computer Applications in the Biosciences*, v. 8, n. 3, p. 215–225, 1992.

CLARK, J.; DEROSE, S. J. Xml path language (xpath)version 1.0. *W3C Recomedation*, 1999.

COOLEY, R. *Web Usage Mining: Discovery and Application of Interesting Patterns from Web Data*. Tese (Doutorado) — Department of Computer Science, University of Minnesota, May 2000.

GAROFALAKIS, M. N.; RASTOGI, R.; SHIM, K. SPIRIT: Sequential pattern mining with regular expression constraints. In: *The VLDB Journal*. [s.n.], 1999. p. 223–234. Disponível em: <citeseer.ist.psu.edu/garofalakis99spirit.html>.

KLEINBERG, J. M. Hubs, authorities, and communities. *ACM Comput. Surv.*, ACM Press, New York, NY, USA, v. 31, n. 4es, p. 5, 1999. ISSN 0360-0300.

MIYAHARA, T. et al. Discovery of frequent tree structured patterns in semistructured web documents. In: *PAKDD '01*. London, UK: Springer-Verlag, 2001. p. 47–52. ISBN 3-540-41910-1.

MURATA, M. et al. Taxonomy of xml schema languages using formal language theory. *ACM Trans. Inter. Tech.*, ACM Press, New York, NY, USA, v. 5, n. 4, p. 660–704, 2005. ISSN 1533-5399.

NEVEN, F. Automata theory for xml researchers. *SIGMOD Record*, ACM Press, New York, NY, USA, v. 31, n. 3, p. 39–46, 2002. ISSN 0163-5808.

PADMANABHAN, B.; TUZHILIN, A. Small is beautiful: discovering the minimal set of unexpected patterns. In: *KDD 2000:*. New York, NY, USA: ACM Press, 2000. p. 54–63. ISBN 1-58113-233-6.

PAPAKONSTANTINOU, Y.; VIANU, V. DTD inference for views of XML data. In: *PODS 2000*. New York, NY, USA: ACM Press, 2000. p. 35–46. ISBN 1-58113-214-X.

SHAPIRO, B. A.; ZHANG, K. Comparing multiple rna secondary structures using tree comparisons. *Computer Applications in the Biosciences*, v. 6, n. 4, p. 309–318, 1990.

SRIKANT, R.; AGRAWAL, R. Mining sequential patterns: Generalizations and performance improvements. In: APERS, P. M. G.; BOUZEGHOUB, M.; GARDARIN, G. (Ed.). *Proc. 5th Int. Conf. Extending Database Technology, EDBT*. [S.l.]: Springer-Verlag, 1996. v. 1057, p. 3–17. ISBN 3-540-61057-X.

SRIKANT, R.; YANG, Y. Mining web logs to improve website organization. In: *WWW '01: Proceedings of the 10th international conference on World Wide Web*. New York, NY, USA: ACM Press, 2001. p. 430–437. ISBN 1-58113-348-0.

TAKAHASHI, M. Generalizations of regular sets and their application to a study of context-free languages. *Information Control*, v. 27, n. 1, p. 1–36, Jan. 1975.

TERMIER, A.; ROUSSET, M.-C.; SEBAG, M. Treefinder: a first step towards xml data mining. In: *ICDM '02: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)*. Washington, DC, USA: IEEE Computer Society, 2002. p. 450. ISBN 0-7695-1754-4.

VIANU, V. A web odyssey: from codd to xml. In: *PODS '01: Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. New York, NY, USA: ACM Press, 2001. p. 1–15. ISBN 1-58113-361-8.

WANG, K.; LIU, H. Schema discovery for semistructured data. In: *KDD*. [S.l.: s.n.], 1997. p. 271–274.

WANG, L.; ZHAO, J. Parametric alignment of ordered trees. *Bioinformatics*, v. 19, n. 17, p. 2237–2245, 2003.

YANG, L. H.; LEE, M.-L.; HSU, W. Efficient mining of xml query patterns for caching. In: *VLDB*. [S.l.: s.n.], 2003. p. 69–80.

ZAKI, M. J. Efficiently mining frequent trees in a forest. In: *KDD 2002*. New York, NY, USA: ACM Press, 2002. p. 71–80. ISBN 1-58113-567-X.

Apêndice A: Uma parte do documento XML SigmodRecord.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<SigmodRecord>
<issue>
<volume>11</volume>
<number>1</number>
<articles>
<article>
  <title>Annotated Bibliography on Data Design.</title>
  <initPage>45</initPage>
  <endPage>77</endPage>
  <authors>
    <author position="00">Anthony I. Wasserman</author>
    <author position="01">Karen Botnich</author>
  </authors>
</article>
<article>
  <title>Architecture of Future Data Base Systems.</title>
  <initPage>30</initPage>
  <endPage>44</endPage>
  <authors>
    <author position="00">Lawrence A. Rowe</author>
    <author position="01">Michael Stonebraker</author>
  </authors>
</article>
<article>
  <title>Database Directions III Workshop Review.</title>
  <initPage>8</initPage>
  <endPage>8</endPage>
  <authors>
    <author position="00">Tom Cook</author>
  </authors>
</article>
<article>
  <title>Errors in 'Process Synchronization in Database Systems'.</title>
  <initPage>9</initPage>
  <endPage>29</endPage>
  <authors>
    <author position="00">Philip A. Bernstein</author>
    <author position="01">Marco A. Casanova</author>
    <author position="02">Nathan Goodman</author>
  </authors>
</article>
</articles>
</issue>
</SigmodRecord>
```

Apêndice B: Autômato de Árvore Local A-SigMod = (Q, q_0, Σ, δ):

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{11}, q_{12}, q_{13}, q_{14}, q_{15}, q_{16}, q_{17}, q_{18}, q_{19}, q_{20},$
$q_{21}, q_{22}, q_{23}, q_{24}, q_{25}, q_{26}, q_{27}, q_{28}, q_{29}, q_{30}\}$

Σ ={article, title, initPage, endPage, authors, author, Jeffrey D. Ullman, Oumlzguumlr Ulusoy, Arnon Rosenthal, James H. Burrows, Richard L. Nolan, Peter Thanisch, Ulrich Schiel, Ahmed K. Elmagarmid, Edwin McKenzie, Michael Stonebraker, 9, 19, 20, 30, 31, 37, 40, 55, 340, 25, 33, 36, 44, 45, 52, 355}

$\delta(q_0, article) = q_1.q_2.q_3$;
$\delta(q_1, initPage) = q_4|q_5|q_6|q_7|q_8|q_9|q_{10}|q_{11}|q_{12}|\varepsilon$;
$\delta(q_2, endPage) = q_5|q_{13}|q_7|q_8|q_{14}|q_{15}|q_9|q_{16}|q_{17}|q_{18}|q_{19}|\varepsilon$;
$\delta(q_3, authors) = q_{20}$;
$\delta(q_4, 9) = \varepsilon$;
$\delta(q_5, 19) = \varepsilon$;
$\delta(q_6, 20) = \varepsilon$;
$\delta(q_7, 25) = \varepsilon$;
$\delta(q_8, 30) = \varepsilon$;
$\delta(q_9, 31) = \varepsilon$;
$\delta(q_{10}, 33) = \varepsilon$;
$\delta(q_{11}, 36) = \varepsilon$;
$\delta(q_{12}, 37) = \varepsilon$;
$\delta(q_{13}, 40) = \varepsilon$;
$\delta(q_{14}, 44) = \varepsilon$;
$\delta(q_{15}, 45) = \varepsilon$;
$\delta(q_{16}, 52) = \varepsilon$;
$\delta(q_{17}, 55) = \varepsilon$;
$\delta(q_{18}, 340) = \varepsilon$; $\delta(q_{19}, 355) = \varepsilon$;
$\delta(q_{20}, author) = q_{21}|q_{22}|q_{23}|q_{24}|q_{25}|q_{26}|q_{27}|q_{28}|q_{29}|q_{30}$;
$\delta(q_{21}, Jeffrey\ D.\ Ullman) = \varepsilon$;
$\delta(q_{22}, Oumlzguumlr\ Ulusoy) = \varepsilon$;
$\delta(q_{23}, Arnon\ Rosenthal) = \varepsilon$;
$\delta(q_{24}, James\ H.\ Burrows) = \varepsilon$;
$\delta(q_{25}, Richard\ L.\ Nolan) = \varepsilon$;
$\delta(q_{26}, Peter\ Thanisch) = \varepsilon$;
$\delta(q_{27}, Ulrich\ Schiel) = \varepsilon$;
$\delta(q_{28}, Ahmed\ K.\ Elmagarmid) = \varepsilon$;
$\delta(q_{29}, Edwin\ McKenzie) = \varepsilon$;
$\delta(q_{30}, Michael\ Stonebraker) = \varepsilon$;

Apêndice C: Autômato de Árvore Local A-People = (Q, q_0, Σ, δ):

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{11}, q_{12}, q_{13}, q_{14}\}$

Σ ={person, pname, pcode, yearstart, yearend, familynm, givennm, dob, dod, background, rels, workedwith, colleague, name, Hitchcock, Mirta Ibarra, Elizabeth Montegomery, Bertolucci}

$\delta(q_0, person) = q_1.q_2.q_3.q_4.q_5.q_6.q_7.q_8.q_9.q_{10}$;
$\delta(q_1, pname) = \varepsilon$;
$\delta(q_2, pcode) = \varepsilon$;
$\delta(q_3, yearstart) = \varepsilon$;
$\delta(q_4, yearend) = \varepsilon$;
$\delta(q_5, familynm) = \varepsilon$;
$\delta(q_6, givennm) = \varepsilon$;
$\delta(q_7, dob) = \varepsilon$;
$\delta(q_8, dod) = \varepsilon$;
$\delta(q_9, background) = \varepsilon$;
$\delta(q_{10}, rels) = q_{11}$;
$\delta(q_{11}, workedwith) = q_{12}$;
$\delta(q_{12}, colleague) = q_{13}$;
$\delta(q_{13}, name) = q_{14}$;
$\delta(q_{14}, Hitchcock) = \varepsilon$;
$\delta(q_{14}, MirtaIbarra) = \varepsilon$;
$\delta(q_{14}, ElizabethMontegomery) = \varepsilon$;
$\delta(q_{14}, Bertolucci) = \varepsilon$;

Apêndice D: Autômato de Árvore Local A-Mains = (Q, q_0, Σ, δ):

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$

Σ ={film, t, year, dirs, prods, prcs, 1975, 1976, 1998, 1999}

$\delta(q_0, film) = q_1.q_2.q_3.q_4.q_5$;
$\delta(q_1, t) = \varepsilon$;
$\delta(q_2, year) = q_6$
$\delta(q_3, dirs) = \varepsilon$;
$\delta(q_4, prods) = \varepsilon$;
$\delta(q_5, prcs) = \varepsilon$;
$\delta(q_6, 1975) = \varepsilon$;
$\delta(q_6, 1976) = \varepsilon$;
$\delta(q_6, 1998) = \varepsilon$;
$\delta(q_6, 1999) = \varepsilon$;

Apêndice E: Autômato de Árvore Local A-Casts = (Q, q_0, Σ, δ):

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$

Σ ={m, f, t,a, p, r, rname, und}

$\delta(q_0, m) = (q_1 * .q_2 * .q_3 * .q_4 * .q_5 * .q_6*)|((q_1 * .q_2*)$;
$\delta(q_1, f) = \varepsilon$;
$\delta(q_2, t) = \varepsilon$
$\delta(q_3, a) = \varepsilon$;
$\delta(q_4, p) = q_7$;
$\delta(q_5, r) = \varepsilon$;
$\delta(q_6, rname) = \varepsilon$;
$\delta(q_7, und) = \varepsilon$;

Apêndice F: Autômato de Árvore Local A-UFULog = (Q, q_0, Σ, δ):

$Q = \{q_0, q_1, q_2, q_3, q_4\}$

Σ ={Pesquisa e Pós-graduação, Pós-graduação Strictu Sensu, Pós-graduação Lato Sensu, Educação, Disciplinas}

$\delta(q_0$,*Pesquisa e Pós-graduação*$) = (q_1|q_2)|(q_1.q_2)$;
$\delta(q_1$,*Pós-graduação Strictu Sensu*$) = q_3$;
$\delta(q_2$,*Pós-graduação Lato Sensu*$) = \varepsilon$;
$\delta(q_3$,*Educação*$) = q_4$;
$\delta(q_4$,*Disciplinas*$) = \varepsilon$;