Explorando a Técnica de Indexação de Conjuntos Candidatos na Mineração de Conjuntos Freqüentes

Adriana Bechara Prado

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre em Computação.

Orientador: Alexandre Plastino

Niterói, Janeiro de 2005.

Explorando a Técnica de Indexação de Conjuntos Candidatos na Mineração de Conjuntos Freqüentes

Adriana Bechara Prado

Dissertação de Mestrado submetida ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre em Computação.

Aprovada por:

Prof. Alexandre Plastino / IC-UFF (Presidente)

Profa. Simone de Lima Martins / IC-UFF

Prof. Wagner Meira Jr. / DCC-UFMG

Niterói, Janeiro de 2005.

Em memória do meu querido avô materno,

Antonio Bechara

Agradecimentos

Meu Deus,

Obrigada por ter me dado forças para cumprir mais esta tarefa em minha vida e, principalmente, por ter colocado em meu caminho pessoas maravilhosas, as quais estiveram sempre ao meu lado durante este período.

Obrigada pela presença incondicional do meu marido Rômulo, que faz de tudo para me ver feliz, mesmo que para isso tenha que abrir mão de seus desejos. Obrigada por seu carinho, apoio, torcida, por sua paciência gigantesca e pela enorme alegria em me ver crescer profissionalmente...

Obrigada pelo amor ilimitado de minha mãe Regina, que mesmo um pouco confusa e receosa com a minha opção pelo mestrado, rezou e me apoiou muito, desde o início, para que tudo fosse finalizado da melhor forma possível. Obrigada, mãe, por não ter poupado esforços quando o assunto era educação e por ter nos ensinado o quanto é importante estudar...

Obrigada pela torcida de minha irmã Renata, que, do seu jeito discreto, estava sempre torcendo por mim. Obrigada pelo apoio, mesmo que à distância, do meu pai Sylvio. Lembro-me até hoje do orgulho que teve quando me tornei universitária...

Obrigada pela presença calorosa da "família Bechara", que nos conforta sempre em todos os momentos. Conforme sua vontade, Senhor, o vô Bechara não está mais entre nós, mas tenho certeza que torceu muito para ver a neta e duplamente afilhada concluir seu trabalho. A saudade é grande, vô, mas as boas lembranças nos fortalece

e nos ajuda a seguir sempre em frente...

Obrigada pelo apoio de meus sogros Cleber e Madalena, que me receberam como filha em seus corações...

Obrigada pela presença do meu sempre amigo e terapeuta de plantão Romário. Suas palavras confortantes e seus sábios conselhos me animam e incentivam muito a continuar lutando...

Obrigada pela atenção, dedicação e paciência do professor Alexandre Plastino, que sempre acreditou em mim e cuidou para que eu não desanimasse nunca...

Obrigada pelos amigos que fiz na Universidade Federal Fluminense e por todo o apoio que recebi dos professores da Computação...

Enfim, obrigada por todos que, de uma forma ou de outra, contribuíram para a conclusão deste trabalho e, acima de tudo, obrigada por ter tanto por que e por quem agradecer...

Amém.

Resumo da Dissertação apresentada à UFF como requisito parcial para a obtenção do título de Mestre em Computação (M.Sc.)

Explorando a Técnica de Indexação de Conjuntos Candidatos na Mineração de Conjuntos Freqüentes

Adriana Bechara Prado

Janeiro/2005

Orientador: Alexandre Plastino
Programa de Pós-Graduação em Ciência da Computação

Ao longo dos últimos dez anos, várias estratégias para extração de conjuntos freqüentes têm sido propostas e aprimoradas. Na grande maioria das vezes, estas estratégias são avaliadas a partir de testes computacionais limitados. Por esta razão, a fim de compará-las de forma mais justa, foi organizado um *Workshop* de implementações de algoritmos para extração de conjuntos freqüentes (*IEEE/ICDM Workshop on Frequent Itemset Mining Implementations - FIMI'03*). De acordo com os resultados obtidos, o algoritmo *kDCI++* foi considerado um dos principais algoritmos para a extração de conjuntos freqüentes da atualidade. Entretanto, analisando seus resultados mais detalhadamente, observa-se que o algoritmo *kDCI++* não se mostra tão eficiente quando valores baixos de suporte mínimo são considerados sobre bases de dados esparsas. Neste trabalho, a fim de aprimorar o desempenho do algoritmo *kDCI++* e torná-lo ainda mais competitivo, é proposta uma adaptação deste algoritmo, denominada algoritmo *kDCI-3*. Nesta estratégia, os conjuntos candidatos são acessados diretamente não só durante as duas iterações iniciais, mas especialmente durante a terceira iteração, avaliada como sendo altamente custosa computacionalmente. Os experimentos realizados mostraram que o algoritmo *kDCI-3* reduz significativamente o tempo total de execução obtido pelo algoritmo *kDCI++*. Quando comparado a outras importantes estratégias, *kDCI-3* aumenta o número de vezes em que o algoritmo *kDCI++* apresenta o melhor desempenho.

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

Exploring Direct Counting for Frequent Itemset Mining

Adriana Bechara Prado

January/2005

Advisor: Alexandre Plastino
Department: Computer Science

During the last ten years, many algorithms have been proposed to mine frequent itemsets. In order to fairly evaluate their behavior, the *IEEE/ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)* has been recently organized. According to its analysis, kDCI++ is a state-of-the-art algorithm. However, it can be observed from the *FIMI'03* experiments that its efficient behavior does not occur for low minimum supports on sparse databases. Aiming at improving kDCI++ and making it even more competitive, we present the kDCI-3 algorithm. This proposal directly accesses candidates not only in the two initial iterations but specially in the third one, which represents, in general, the highest computational cost of kDCI++ for low minimum supports. Results have shown that kDCI-3 outperforms kDCI++ in the conducted experiments. When compared to other important algorithms, kDCI-3 enlarged the number of times kDCI++ presented the best behavior.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Regras de associação representam padrões de relacionamento entre itens de dados que ocorrem com uma determinada freqüência em uma base de dados de transações. Uma aplicação típica de mineração de regras de associação é a análise de bases de dados de transações de compras (*market basket analisys*) a qual consiste em identificar relacionamentos entre produtos comprados por seus clientes.

Uma regra de associação é formalmente definida da seguinte maneira [3]. Seja $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$ um conjunto de itens e $\mathcal{D}$ um conjunto de transações (base de dados de transações), onde cada transação t é um conjunto de itens, $t \subseteq \mathcal{I}$. Uma regra de associação $\mathcal{R}$ definida sobre $\mathcal{I}$ é uma implicação da forma $X \Rightarrow Y$ onde $X \subset \mathcal{I}$, $Y \subset \mathcal{I}$, $X \neq \emptyset$, $Y \neq \emptyset$ e $X \cap Y = \emptyset$. X é o antecedente e Y o conseqüente da regra.

A regra $X \Rightarrow Y$ tem confiança c se $c\%$ das transações em $\mathcal{D}$ que contêm X também contêm Y. A regra $X \Rightarrow Y$ tem suporte s se $s\%$ das transações em $\mathcal{D}$ contêm $X \cup Y$.

O processo de mineração de regras de associação é comumente dividido em duas fases. Seja *supmin* e *confmin* valores mínimos de suporte e confiança, respectivamente, definidos pelo usuário do processo. A primeira fase consiste em

identificar todos os conjuntos freqüentes, ou seja, todos os conjuntos de itens que aparecem em pelo menos *supmin%* das transações da base de dados. A segunda fase tem como objetivo gerar, para cada conjunto freqüente Z obtido durante a primeira fase, as regras de associação do tipo $A \Rightarrow B$ que possuem confiança maior ou igual a *confmin%*, tal que $A \subset Z$, $B \subset Z$ e $A \cup B = Z$.

A fase de extração dos conjuntos freqüentes (primeira fase) destaca-se pelo grande esforço computacional exigido, devido ao número exponencial de possíveis conjuntos freqüentes ($2^{|I|}$), que caracteriza o espaço de busca a ser avaliado. Como conseqüência, várias estratégias vêm sendo constantemente desenvolvidas e aprimoradas a fim de solucioná-la de maneira eficiente [9].

Muitas das estratégias propostas têm como base o algoritmo *Apriori* [4]. O algoritmo *Apriori* considera a seguinte propriedade para diminuir o espaço de busca a ser avaliado na extração dos conjuntos freqüentes: todo conjunto de itens que contém um subconjunto não freqüente também não é freqüente. A partir desta propriedade, os conjuntos freqüentes são extraídos, iterativamente. Primeiramente, o suporte dos conjuntos de tamanho 1 é calculado a partir de uma primeira leitura da base de dados. Os conjuntos que não possuem o suporte mínimo são desconsiderados. Em seguida, a cada iteração $k \geq 2$, os possíveis conjuntos freqüentes de tamanho k, chamados conjuntos candidatos, são gerados a partir da combinação dos conjuntos freqüentes de tamanho $k-1$, obtidos durante a iteração anterior. O suporte dos conjuntos candidatos é contado através de uma nova leitura da base de dados e aqueles que não possuem o valor de suporte mínimo são descartados. O algoritmo termina quando, em uma determinada iteração, nenhum novo conjunto candidato possui suporte maior ou igual ao suporte mínimo.

Uma outra classe de algoritmos extrai os conjuntos freqüentes sem a geração de conjuntos candidatos. O algoritmo *FP-growth* [12], por exemplo, obtém os conjuntos freqüentes a partir de uma representação compactada da base de dados, denominada *FP-tree*. Cada nó desta árvore representa um determinado item da base de dados. Os nós formam ramos que representam conjuntos de itens presentes em uma ou mais transações da base de dados. O algoritmo inicia a busca por conjuntos

freqüentes a partir dos conjuntos freqüentes de tamanho 1. Em seguida, considera um item freqüente de cada vez e gera, com o auxílio da *FP-tree*, todos os conjuntos freqüentes que contêm aquele item como último item (sufixo). A identificação dos conjuntos freqüentes de tamanho $k \geq 2$ se dá a partir de projeções da base de dados, denominadas bases de dados condicionadas (*condicional pattern base*) e suas respectivas *FP-trees* condicionadas (*condicional FP-trees*), também chamadas na literatura de projeções físicas da base de dados.

Comumente, toda nova estratégia proposta para extração de conjuntos freqüentes é comparada com estratégias propostas anteriormente, evidenciando-se sua contribuição específica. Conforme observado por Goethals e Zaki em [9], a avaliação de novas estratégias é, na grande maioria das vezes, feita a partir de poucos e limitados testes computacionais, já que muitos dos algoritmos originais não estão disponíveis para avaliação e análise de desempenho. Além disso, diferentes implementações de uma mesma estratégia podem apresentar comportamentos distintos, principalmente quando diferentes tipos de bases de dados ou valores de suporte mínimos são considerados [9].

Desta forma, com o objetivo de avaliar implementações de novas ou já existentes estratégias para extração de conjuntos freqüentes, considerando diferentes tipos de base de dados (densas ou esparsas, sintéticas ou reais) e diferentes valores de suporte mínimo, Goethals e Zaki organizaram, recentemente, um *Workshop* de implementações de algoritmos para extração de conjuntos freqüentes (*IEEE/ICDM Workshop on Frequent Itemset Mining Implementation - FIMI'03*). Todas as implementações aceitas pelo *workshop* estão, a partir de agora, disponíveis na página http://fimi.cs.helsinki.fi/ (*FIMI repository*), onde também podem ser encontrados gráficos de avaliação de desempenho em relação ao tempo total de execução obtido pelas estratégias sobre diferentes combinações de bases de dados e suportes mínimos.

Os resultados da avaliação do *Workshop FIMI'03*, em relação ao tempo total de execução obtido pelas estratégias, foram publicados pelos organizadores em [9] e classificados em dois grupos distintos: resultados para suportes altos e resultados para suportes baixos. Conforme observado em [9], nenhum algoritmo pôde

ser considerado o melhor absoluto, tanto para suportes altos quanto para suportes baixos. Entretanto, algumas estratégias apresentaram melhores desempenhos com mais freqüência que outras. Assim, de acordo com a avaliação publicada, para suportes altos, os algoritmos *kDCI++* [17] e *PatriciaMine* [22] apresentaram os melhores resultados. Para suportes baixos, embora o quadro não tenha ficado tão evidente, os algoritmos *FPgrowth** [10] e *LCMfreq* [28] apresentaram melhores desempenhos, enquanto que *kDCI++* e *PatriciaMine* obtiveram a segunda colocação. Baseado nestes resultados, os algoritmos *PatriciaMine* e *kDCI++* foram considerados, por Goethals e Zaki, os principais algoritmos para extração de conjuntos freqüentes da atualidade.

Resumidamente, o algoritmo *PatriciaMine* é uma variação do algoritmo *FP-growth* que utiliza a estrutura de dados *Patricia trie* para a extração dos conjuntos freqüentes. A estrutura *Patricia trie* representa a *FP-tree* de forma compactada. O algoritmo *kDCI++* é uma recente variação do algoritmo *DCI* [16], que tem como base o algoritmo *Apriori*. Em suas primeiras iterações, utiliza estruturas de dados eficientes para acesso aos conjuntos candidatos e procedimentos que reduzem gradativamente a base de dados durante a execução. Quando a base de dados se torna pequena o suficiente para ser carregada em memória principal, o algoritmo *kDCI++* gera uma representação verticalizada da base de dados, a partir da qual o suporte dos conjuntos candidatos é calculado.

Analisando mais detalhadamente os resultados obtidos por estes algoritmos [9], observa-se que o algoritmo *kDCI++* não se mostra tão eficiente quando valores baixos de suporte mínimo são considerados sobre bases de dados esparsas. A fim de compreender a razão de tal comportamento, neste trabalho, foram realizados vários experimentos considerando diferentes combinações de bases de dados esparsas e valores de suporte mínimo também utilizados por [5, 9, 10, 12, 13, 16, 17, 18, 22, 24, 28, 30, 32]. De acordo com os resultados obtidos, identificou-se que a terceira iteração do algoritmo *kDCI++* apresenta um custo computacional bastante elevado se comparado ao custo computacional das demais iterações. Nos experimentos realizados, a terceira iteração consome, em média, 59% do tempo total de execução.

Nesta dissertação, a fim de aprimorar o desempenho do algoritmo *kDCI++* e torná-lo também eficiente sobre bases de dados esparsas quando valores de suporte baixos são considerados, é proposta uma adaptação deste algoritmo, denominada algoritmo *kDCI-3*, em que a estratégia realizada durante a terceira iteração, avaliada como sendo altamente custosa computacionalmente, é substituída por outra que possibilita a contagem direta do suporte dos candidatos de tamanho 3. Esta estratégia é baseada em uma estrutura de dados apresentada em [26], através da qual a contagem dos candidatos de tamanho 3 torna-se mais eficiente.

Outra questão abordada refere-se à importância do gerenciamento de memória nos algoritmos para extração de conjuntos freqüentes. Experimentos computacionais realizados neste trabalho mostraram que, quando a base de dados era significativamente grande para ser compactada em memória principal, as execuções do algoritmo *FPgrowth** foram longas devido ao uso de memória virtual. O mesmo ocorreu com o algoritmo *kDCI++* quando o número de candidatos cresceu consideravelmente. Desta forma, a fim de evitar que o algoritmo *kDCI-3* apresente o mesmo problema quando a estrutura de dados adotada se torna muito grande para ser construída em memória principal, também foi incorporada ao algoritmo uma estratégia de gerenciamento de memória. Nesta estratégia, blocos de candidatos de tamanho 3 são contados separadamente e consecutivamente, de modo que a quantidade de memória necessária para a contagem de cada bloco gerado não ultrapasse a quantidade de memória principal disponível.

Os capítulos desta dissertação estão organizados da seguinte forma:

• Capítulo 2 - Principais Algoritmos para Extração de Conjuntos Freqüentes. Neste capítulo, os algoritmos *Apriori* e *FP-growth* são descritos detalhadamente. Este capítulo também é destinado à apresentação dos algoritmos *kDCI++*, *PatriciaMine*, *FPgrowth** e *LCMfreq*, sendo o *kDCI++* objeto de estudo desta dissertação. Além da descrição, também é apresentada uma análise dos seus desempenhos, baseada na avaliação publicada em [9].

• Capítulo 3 - Algoritmo *kDCI-3*. Este capítulo tem como objetivo apresentar a técnica de indexação direta de candidatos utilizada pelo algoritmo *kDCI-3*,

extensão do algoritmo *kDCI++*, proposto nesta dissertação. A estratégia de gerenciamento de memória implementada também será descrita neste capítulo.

• Capítulo 4 - Resultados Computacionais. Neste capítulo, serão apresentadas uma avaliação de desempenho da técnica proposta e uma análise comparativa entre o algoritmo *kDCI-3* e os principais algoritmos identificados no *Workshop FIMI'03*.

• Capítulo 5 - Conclusões. Finalmente, as conclusões desta dissertação serão apresentadas neste capítulo, ressaltando suas principais contribuições e trabalhos futuros relacionados.

Capítulo 2

Principais Algoritmos para Extração de Conjuntos Freqüentes

Neste capítulo, são apresentados os algoritmos para extração de conjuntos freqüentes que se destacaram no *IEEE/ICDM Workshop on Frequent Itemset Mining Implementation (FIMI'03)*, juntamente com os algoritmos que serviram como base para a proposta destes. Em cada uma das seis primeiras seções é descrito um algoritmo específico. Na Seção 2.7, são apresentadas uma análise de desempenho destes algoritmos com base nos resultados descritos em [9] e uma análise mais detalhada do algoritmo *kDCI++*, objeto de estudo desta dissertação.

2.1 Algoritmo *Apriori*

O algoritmo *Apriori* [4], proposto por Agrawal e Srikant em 1994, destaca-se por ser o primeiro a reduzir, eficientemente, o espaço de busca a ser avaliado na identificação dos conjuntos freqüentes.

O algoritmo *Apriori* é um algoritmo iterativo. Em uma primeira iteração, obtém-se o suporte dos conjuntos de itens de tamanho $k = 1$ através de uma lei-

tura da base de dados. Os conjuntos com suporte menor que o suporte mínimo são eliminados. Em seguida, em cada iteração $k \geq 2$, os possíveis conjuntos freqüentes de tamanho k, chamados conjuntos candidatos de tamanho k, são gerados a partir dos conjuntos freqüentes de tamanho $k-1$, obtidos na iteração anterior, e armazenados em uma estrutura de dados denominada árvore *hash*. A base de dados é percorrida a fim de se obter o suporte dos conjuntos candidatos de tamanho k e, conseqüentemente, obter os conjuntos de itens freqüentes de tamanho k. O algoritmo termina quando, em uma determinada iteração, nenhum novo conjunto candidato tem suporte maior ou igual ao suporte mínimo.

Durante a geração de candidatos, o algoritmo *Apriori* considera a propriedade de que todo conjunto que contém um subconjunto não freqüente, também não é freqüente, reduzindo, assim, o número de candidatos a serem avaliados. A técnica de geração de candidatos é dividida em duas fases: junção e poda. A fase de junção, na geração de conjuntos candidatos de tamanho k, combina pares de conjuntos freqüentes de tamanho $k-1$ que possuem os mesmos $k-2$ itens iniciais. Por exemplo, na terceira iteração, a combinação dos conjuntos freqüentes $\{a, b\}$ e $\{a, c\}$ gera o conjunto candidato $\{a, b, c\}$ já que possuem o mesmo prefixo a. A fase de poda elimina os candidatos gerados na fase de junção que possuem algum subconjunto não freqüente (considerando a propriedade citada anteriormente). Novamente, como exemplo, se no conjunto candidato $\{a, b, c\}$, o subconjunto $\{b, c\}$ não for um conjunto freqüente de tamanho 2, o conjunto candidato $\{a, b, c\}$ será eliminado e não será avaliado.

Vale observar que os candidatos são gerados em ordem lexicográfica e que as fases de junção e poda só são executadas a partir da geração de candidatos de tamanho 3. Os candidatos de tamanho 2 são gerados apenas combinando-se os conjuntos freqüentes de tamanho 1, dois a dois.

Para melhor ilustrar a estratégia *Apriori*, considere a base de dados da Tabela 2.1 e suporte mínimo igual a duas transações.

Inicialmente, o conjunto F_1, que contém os conjuntos freqüentes de tamanho 1, é identificado através de uma leitura da base de dados. Na base de dados

TID	Itens
1	a, b, c, d, g
2	a, b, c, e, f, o
3	d, e, h
4	a, b, d
5	a, b, c, d
6	a, b, c, e
7	a

Tabela 2.1: Exemplo de base de dados de transações.

considerada $F_1 = \{\{a : 6\}, \{b : 5\}, \{c : 4\}, \{d : 4\}, \{e : 3\}\}$ (o número após ":" representa o suporte do conjunto). Em seguida, o algoritmo gera o conjunto C_2, composto pelos conjuntos candidatos de tamanho 2, combinando os itens freqüentes de F_1, dois a dois. Após a geração de C_2, a base de dados é lida novamente e, para cada transação t, o suporte de cada um dos candidatos de tamanho 2 contidos em t é incrementado. A leitura completa da base de dados gera os conjuntos freqüentes de tamanho 2, $F_2 = \{\{a, b : 5\}, \{a, c : 4\}, \{a, d : 3\}, \{a, e : 2\}, \{b, c : 4\}, \{b, d : 3\}, \{b, e : 2\}, \{c, d : 2\}, \{c, e : 2\}\}$.

Feita a geração de F_2, o algoritmo *Apriori* gera o conjunto C_3, composto pelos conjuntos candidatos de tamanho 3, combinando os pares de conjuntos freqüentes de F_2 que possuem o mesmo prefixo de tamanho 1. Por exemplo, a combinação dos conjuntos freqüentes $\{a, b\}$ e $\{a, c\}$ gera o conjunto candidato $\{a, b, c\}$ já que possuem o mesmo prefixo a. Além disso, os conjuntos candidatos de tamanho 3 que possuem algum subconjunto não freqüente, ou seja, que não faça parte de F_2, são descartados. Observe, por exemplo, que os conjuntos candidatos $\{a, d, e\}$, $\{b, d, e\}$ e $\{c, d, e\}$ são desconsiderados, dado que o subconjunto $\{d, e\}$ não é freqüente. Desta forma, o conjunto dos candidatos de tamanho 3 é $C_3 = \{\{a, b, c\}, \{a, b, d\}, \{a, b, e\}, \{a, c, d\}, \{a, c, e\}, \{b, c, d\}, \{b, c, e\}\}$ e uma nova leitura da base de dados gera $F_3 = \{\{a, b, c : 4\}, \{a, b, d : 3\}, \{a, b, e : 2\}, \{a, c, d : 2\}, \{a, c, e : 2\}, \{b, c, d : 2\}, \{b, c, e : 2\}\}$. Neste momento, o algoritmo *Apriori* repete novamente os passos anteriores e encontra o conjunto $F_4 = \{\{a, b, c, d : 2\}, \{a, b, c, e : 2\}\}$. Como, na próxima iteração, não é gerado nenhum conjunto candidato (o subcon-

junto $\{c, d, e\}$ não é freqüente), o algoritmo atinge a condição de parada e termina.

A seguir, são descritas algumas variações do algoritmo *Apriori*.

2.1.1 Variações do algoritmo *Apriori*

Uma das primeiras variações do algoritmo *Apriori* foi o algoritmo *DHP* (*Direct Hash and Prune*) [20], proposto por Park *et. al.*. Suas principais contribuições são a redução do número de candidatos e técnicas que reduzem gradativamente a base de dados durante a execução, de forma a diminuir o tempo de sua leitura.

Outras variações tiveram como principal objetivo a redução do número de leituras da base de dados. O algoritmo *Partition* [25] baseia-se na seguinte propriedade. Considerando-se que a base de dados de transações está dividida em n partições, se um conjunto F é freqüente em relação a toda a base de dados (freqüência global), então F é freqüente em relação a pelo menos uma partição (freqüência local), ou seja, possui suporte maior ou igual ao mínimo dentro desta partição. Baseado nesta propriedade, o algoritmo *Partition* divide a base de dados em partes suficientemente pequenas para caberem em memória principal. Durante o primeiro passo do algoritmo, os conjuntos freqüentes locais (de cada partição) são extraídos utilizando as idéias do algoritmo *Apriori* com a partição inteira em memória principal. Desta forma, em um único acesso a toda a base de dados, os conjuntos freqüentes locais de cada partição são gerados. Estes conjuntos serão os candidatos a freqüentes globais. Em um segundo e último passo, é feita uma leitura completa da base de dados a fim de identificar quais conjuntos candidatos são verdadeiros conjuntos freqüentes globais.

O algoritmo *Sampling*, descrito em [27], seleciona uma amostra da base de dados a partir da qual obtém os prováveis conjuntos freqüentes globais. Em seguida, verifica no restante da base de dados se o resultado obtido é verdadeiro. Caso a amostra não contenha todos os conjuntos freqüentes globais, é feita uma segunda leitura da base de dados em busca dos conjuntos freqüentes remanescentes. O algoritmo *DIC* (*Dynamic Itemset Counting*) [7], diferentemente das demais variações

do algoritmo *Apriori*, conta candidatos de diferentes tamanhos à medida que as transações são lidas, diminuindo, assim, o número de acessos à base de dados.

No algoritmo *Tree Projection* [2], a extração dos conjuntos freqüentes é feita através de uma árvore de prefixos. O nível 0 da árvore é composto apenas pelo nó raiz, que representa o conjunto de itens vazio. Os demais nós nos níveis $k \geq 1$ representam os conjuntos freqüentes de tamanho k que foram obtidos durante a k-ésima iteração. Os nós que compõem o nível k são "pais" dos nós no nível $k+1$ que representam conjuntos freqüentes com os mesmos k itens iniciais. Os $k+1$-ésimos itens que compõem os conjuntos freqüentes representados pelos nós "filhos" são chamados de extensões de seu nó "pai".

Inicialmente, em uma primeira leitura da base de dados, obtêm-se os conjuntos de itens freqüentes de tamanho 1, os quais formarão o nível $k = 1$ da árvore. Os itens freqüentes de tamanho 1 serão incluídos na árvore em forma de nós filhos do nó raiz. Em seguida, durante as iterações $k \geq 2$, os conjuntos freqüentes de tamanho k são obtidos com o auxílio da árvore de prefixos da seguinte forma. Cada transação t da base de dados percorre a árvore de prefixos, em profundidade, até o nível $k-2$. A cada nó N visitado nos níveis anteriores ao nível $k-2$, os itens de t que não estão presentes nas extensões de N são desconsiderados (t é reduzida gradativamente a cada nó visitado). Ao atingir um nó P, no nível $k-2$, os conjuntos candidatos são gerados combinando-se dois a dois as extensões de P. Considerando-se que o nó P representa o conjunto freqüente h, cada combinação $\{i, j\}$ das extensões de P representará o conjunto candidato $h \cup \{i, j\}$. Cada conjunto $\{i, j\}$ presente em t tem seu suporte incrementado em uma matriz e após o processamento de todas as transações da base de dados, os conjuntos candidatos com suporte maior ou igual ao mínimo formarão o k-ésimo nível da árvore de prefixos.

O algoritmo *DCP* (*Direct Count & Prune transactions*) [15] utiliza estruturas de dados para acesso direto aos conjuntos candidatos e procedimentos que reduzem gradativamente a base de dados durante a execução, inspiradas nas técnicas utilizadas pelo algoritmo *DHP*.

O algoritmo *DCI* (*Direct Count & Intersect*) [16], extensão do algoritmo

DCP, é uma estratégia híbrida para extração de conjuntos freqüentes. Em suas primeiras iterações, *DCI* utiliza as idéias do algoritmo *DCP*. Entretanto, quando a base de dados, reduzida gradativamente durante a execução, se torna pequena o suficiente para ser carregada em memória principal, o algoritmo *DCI* gera uma representação verticalizada da base de dados. Uma representação verticalizada da base de dados pode ser vista como um conjunto de m listas de bits de tamanho n, onde m é o número de itens e n, o número de transações. Cada lista de bits está associada a um determinado item da base de dados e cada bit, desta lista de bits, está associado a uma determinada transação da base de dados. Desta forma, se o j-ésimo bit da lista associada ao item i, tem o valor 1, significa que o item i está presente na transação j.

Após a construção da base de dados verticalizada, em memória principal, os suportes dos conjuntos candidatos são, então, calculados a partir de interseções das listas de bits associadas aos itens que os compõe.

A fim de reduzir o custo computacional requerido pelas operações de interseção, o algoritmo *DCI* é capaz de adaptar seu comportamento através da utilização de um conjunto de otimizações que varia de acordo com a densidade da base de dados considerada.

Recentemente, uma nova variação do algoritmo *DCI*, denominada *kDCI++* [17], foi proposta pelos mesmos autores. Este algoritmo é objeto de estudo deste trabalho e será descrito a seguir.

2.2 Algoritmo *kDCI++*

O algoritmo *kDCI++* é uma recente versão do algoritmo híbrido *DCI* (*Direct Count & Intersect*) [16] proposto por Orlando *et. al.* em [17]. Conforme comentado anteriormente, o algoritmo *DCI* é um algoritmo da classe apriori.

Em suas iterações iniciais, aqui chamada de *fase de contagem*, o algoritmo *kDCI++* utiliza estruturas de dados nas quais os conjuntos candidatos são acessados

diretamente, além de procedimentos que reduzem gradativamente a base de dados, inspirados nos procedimentos utilizados pelo algoritmo *DHP* [20]. Quando a base de dados reduzida se torna pequena o suficiente para ser carregada na memória principal, o algoritmo *kDCI++* cria uma representação verticalizada da base de dados. Neste momento, os suportes dos candidatos são calculados a partir de interseções de listas de bits (*TIDlists*). Esta fase é aqui denominada *fase de interseção*.

Na Subseção 2.2.1 será apresentada a fase de contagem. Nela, as técnicas de redução da base de dados e as estruturas de dados de acesso direto aos candidatos utilizadas pelo algoritmo *kDCI++* serão descritas. A Subseção 2.2.2 será destinada à apresentação da fase de interseção em que os conjuntos freqüentes são extraídos a partir da representação verticalizada da base de dados. Finalmente, na Subseção 2.2.3, serão apresentadas algumas otimizações incorporadas ao algoritmo.

2.2.1 Fase de Contagem

Durante a fase de contagem, o algoritmo *kDCI++* faz uso de técnicas para a redução da base de dados, a fim de reduzir o tempo de sua leitura. No final de cada iteração k, uma nova base de dados com um número menor de transações e/ou itens é formada e, desta forma, reduz-se o tempo destinado à contagem dos candidatos durante a iteração $k + 1$.

Duas técnicas são utilizadas pelo algoritmo: A *poda global* da base de dados e a *poda local* da base de dados. A *poda local* é também utilizada pelo algoritmo *DHP* [20] enquanto que a *poda global* foi proposta pelos próprios autores do *kDCI++*.

As técnicas de poda global e local são aplicadas a cada transação t, respectivamente, antes e depois da contagem dos candidatos presentes em t com o objetivo principal de reduzir o número de itens presentes em t e, conseqüentemente, reduzir o número de conjuntos de itens em t que deverão ser avaliados. A seguir, as técnicas são descritas separadamente.

Poda Global

A técnica de poda global é baseada na seguinte propriedade. Um conjunto de itens I de tamanho k presente em uma transação t da base de dados poderá ser um conjunto freqüente somente se todos os seus subconjuntos de tamanho $k-1$ forem conjuntos freqüentes, ou seja, pertencerem a F_{k-1}. Observe que, de posse de tal informação, o algoritmo poderia, simplesmente, ignorar a contagem de todos os conjuntos de itens I em t, para os quais esta propriedade não é satisfeita.

Entretanto, conforme observado em [17], verificar se todos os subconjuntos de tamanho $k-1$ de todo conjunto I em t são conjuntos freqüentes de tamanho $k-1$ seria bastante custoso computacionalmente. Desta forma, os autores propõem a seguinte heurística.

Considere o exemplo da base de dados da Tabela 2.1. O conjunto de itens de tamanho 3 $\{a, b, c\}$, contido na primeira transação, será freqüente somente se todos os seus subconjuntos de tamanho 2, $\{a, b\}$, $\{a, c\}$ e $\{b, c\}$, forem conjuntos freqüentes de tamanho 2. Da mesma forma, o conjunto de itens de tamanho 4 $\{a, b, c, e\}$, contido na segunda transação, será freqüente somente se seus subconjuntos de tamanho 3 $\{a, b, c\}$, $\{a, b, e\}$, $\{a, c, e\}$ e $\{b, c, e\}$ também forem conjuntos freqüentes. Note que os subconjuntos de tamanho $k-1$ de um determinado conjunto de itens I de tamanho k são exatamente k e que cada item de I aparece em apenas $k-1$ destes.

Neste contexto, antes da contagem dos conjuntos candidatos de tamanho k presentes em uma determinada transação t, a técnica de poda global é aplicada de forma a eliminar todos os itens i de t, que pertencerem a menos de $k-1$ conjuntos freqüentes de F_{k-1}. Caso o tamanho da nova transação t' seja menor do que k, t' não é considerada durante a contagem dos conjuntos candidatos de tamanho k, já que, de forma alguma, poderá contribuir para a formação de um conjunto freqüente de tamanho k.

A poda global é aplicada a cada transação da base de dados corrente, a partir da segunda iteração.

Poda Local

A técnica de poda local é baseada na mesma propriedade da poda global. Entretanto, esta técnica é aplicada a cada transação t', após a contagem dos conjuntos candidatos de tamanho k em t', com o objetivo de eliminar os itens de t' que não poderão contribuir na formação de um conjunto freqüente de tamanho $k+1$, durante a próxima iteração.

Como, neste momento, ainda não se conhece o conjunto F_k e, como o conjunto de candidatos C_k é um superconjunto de F_k, a poda local é, então, baseada na seguinte heurística.

Após a contagem dos conjuntos candidatos de tamanho k presentes em uma determinada transação t', a técnica de poda local é aplicada de forma a eliminar todos os itens i de t' que estão presentes em menos de k conjuntos candidatos de C_k. Da mesma forma que na poda global, caso o tamanho da nova transação t'' seja menor do que $k+1$, t'' é descartada, já que não poderá contribuir para a formação de um conjunto freqüente de tamanho $k+1$.

Após a aplicação da poda local, a transação resultante t'' é incluída em uma nova base de dados a ser utilizada durante a próxima iteração.

Vale observar que, durante a segunda iteração, todos os itens presentes nas transações t' (geradas após a aplicação da poda global) estarão presentes em F_1, já que os itens não freqüentes serão eliminados pela poda. Conseqüentemente, todos os itens contidos nas transações t' também estarão presentes em C_2, já que C_2 é gerado pela combinação, dois a dois, dos itens em F_1. Desta forma, a poda local só poderá surtir efeito e só será executada a partir da terceira iteração.

Contagem indexada dos Conjuntos Candidatos

Em suas iterações iniciais, o algoritmo *kDCI++* utiliza estruturas de dados específicas nas quais os conjuntos candidatos são acessados diretamente.

Durante a segunda iteração, dado que os candidatos de tamanho 2 são

gerados pela combinação dois a dois dos itens freqüentes, o algoritmo *kDCI++* utiliza uma tabela de prefixos, aqui denominada T_2, com $\binom{|F_1|}{2}$ posições, onde cada posição é um contador que armazenará o suporte de um determinado conjunto candidato, durante a fase de contagem.

A posição em T_2 que representa o conjunto candidato de tamanho 2 $c = (c_1, c_2)$, onde c_1 e c_2 estão ordenados lexicograficamente, é obtida através da Equação 2.1. O algoritmo *kDCI++* mapeia c em um par $\{x_1,x_2\}$ onde x_1=$\mathcal{T}(c_1)$, x_2=$\mathcal{T}(c_2)$, e $\mathcal{T}$ é uma função definida por $\mathcal{T} : F_1 \rightarrow \{1, \ldots, |F_1|\}$.

$$T_2(c_1, c_2) = \sum_{i=1}^{x_1-1}(|F_1| - i) + (x_2 - x_1) = |F_1|(x_1 - 1) - \frac{x_1(x_1 - 1)}{2} + x_2 - x_1 \ . \quad (2.1)$$

Esta equação é baseada na seguinte idéia: as primeiras ($|F_1|$-1) posições são associadas aos conjuntos candidatos $\{1,x_2\}$, $2 \leq x_2 \leq |F_1|$, as ($|F_1|$-2) posições seguintes estão associadas aos candidatos $\{2,x_2\}$, $3 \leq x_2 \leq |F_1|$, e assim por diante.

Para melhor ilustrar a estrutura de dados T_2, considere, novamente, a base de dados da Tabela 2.1 e o suporte mínimo igual a duas transações. A partir da primeira leitura da base de dados, o algoritmo *kDCI++* obtém o conjunto freqüente $F_1 = \{\{a : 6\}, \{b : 5\}, \{c : 4\}, \{d : 4\}, \{e : 3\}\}$. Desta forma, como os conjuntos de candidatos de tamanho 2 são gerados através da combinação dois a dois dos itens freqüentes, a estrutura de dados T_2, construída para a contagem dos candidatos de tamanho 2, é ilustrada na Figura 2.1.

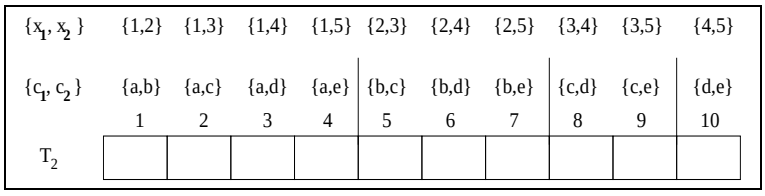


Figura 2.1: A estrutura de dados T_2 construída pelo algoritmo *kDCI++*.

Como exemplo, considere o conjunto candidato $\{b, d\}$. De acordo com a Equação 2.1, a posição de T_2 que armazenará o suporte do conjunto candidato $\{b, d\}$ é encontrada da seguinte forma:

$$T_2(b,d) = 5(2-1) - \frac{2(2-1)}{2} + 4 - 2 = 6 \tag{2.2}$$

Durante a segunda leitura da base de dados, os candidatos de tamanho 2 presentes nas transações têm seus suportes contados em T_2. No final da contagem, se o valor armazenado na posição em T_2 correspondente a um determinado conjunto candidato c é maior ou igual ao suporte mínimo, c é incluído em F_2. No exemplo considerado, o conjunto F_2 encontrado é $\{\{a,b:5\},\{a,c:4\},\{a,d:3\},\{a,e:2\},\{b,c:4\},\{b,d:3\},\{b,e:2\},\{c,d:2\},\{c,e:2\}\}$.

Para as iterações $k > 2$, o algoritmo *kDCI++* utiliza, para a contagem dos conjuntos candidatos, a tabela de prefixos T_k com $\binom{|M_k|}{2}$ posições. M_k é o conjunto de itens presentes na base de dados corrente, na iteração k, que não foram podados através das técnicas de poda descritas anteriormente. Cada posição i de T_k representa um determinado prefixo de tamanho 2 $\{c_1, c_2\}$ e aponta para uma área de memória onde estão armazenados, em ordem lexicográfica e de forma compactada, todos os candidatos de tamanho k com o mesmo prefixo.

Na Figura 2.2, é ilustrada a tabela de prefixos T_3 construída pelo algoritmo *kDCI++* para a contagem dos conjuntos candidatos de tamanho 3, a partir da base de dados da Tabela 2.1 e considerando-se o suporte mínimo igual a duas transações. No exemplo, o conjunto $C_3 = \{\{a,b,c\},\{a,b,d\},\{a,b,e\},\{a,c,d\},\{a,c,e\},\{b,c,d\},\{b,c,e\}\}$.

Já a Figura 2.3 ilustra a tabela de prefixos T_4 para a contagem dos candidatos de tamanho 4, $\{a,b,c,d\}$ e $\{a,b,c,e\}$.

A área de memória onde os conjuntos candidatos são armazenados é composta por três vetores. O vetor P armazena os diferentes prefixos de tamanho $k-1$ presentes nos conjuntos candidatos. Desta forma, cada entrada i de T_k que representa um determinado prefixo de tamanho 2 $\{c_1, c_2\}$ aponta para a primeira entrada j do vetor P que armazena prefixos de tamanho $k-1$ iniciados por $\{c_1, c_2\}$. Por exemplo, na Figura 2.3, a entrada de T_4 que representa o prefixo $\{a,b\}$ aponta para a primeira entrada do vetor P que armazena prefixos de tamanho 3 iniciados

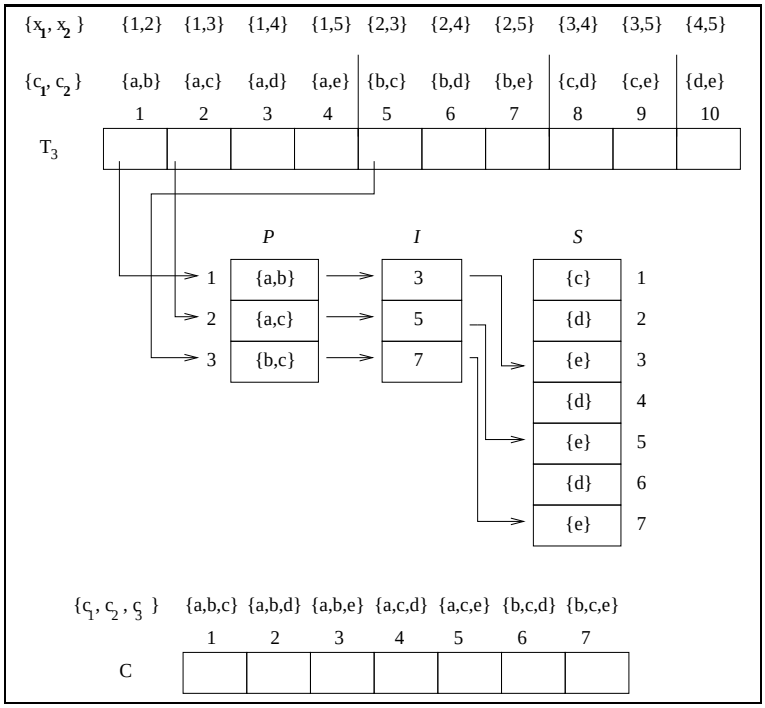


Figura 2.2: A estrutura de dados T_3 construída pelo algoritmo *kDCI++*.

por $\{a, b\}$.

O vetor S armazena os sufixos de tamanho 1 presentes nos conjuntos candidatos. O vetor I conecta cada prefixo no vetor P à seção de S cujos sufixos armazenados completam os conjuntos candidatos de mesmo prefixo (a entrada do vetor I referente a um prefixo p em P aponta para o final da lista de sufixos em S que completam os conjuntos candidatos de prefixo p).

Os suportes são contabilizados no vetor C, onde cada entrada representa um determinado conjunto candidato. Conforme ilustrado nas Figuras 2.2 e 2.3, os candidatos são representados no vetor C em ordem lexicográfica e o índice da entrada em C, que corresponde a um determinado candidato, é o mesmo índice da entrada em S, na qual seu sufixo está armazenado.

O algoritmo *kDCI++* obtém o suporte dos conjuntos candidatos de tamanho k, a partir da estrutura T_k, da seguinte forma. Para cada prefixo de tamanho 2 presente em uma transação t da base de dados corrente, *kDCI++* encontra a posição correspondente i na estrutura T_k, através da Equação 2.1. Em seguida, o algoritmo

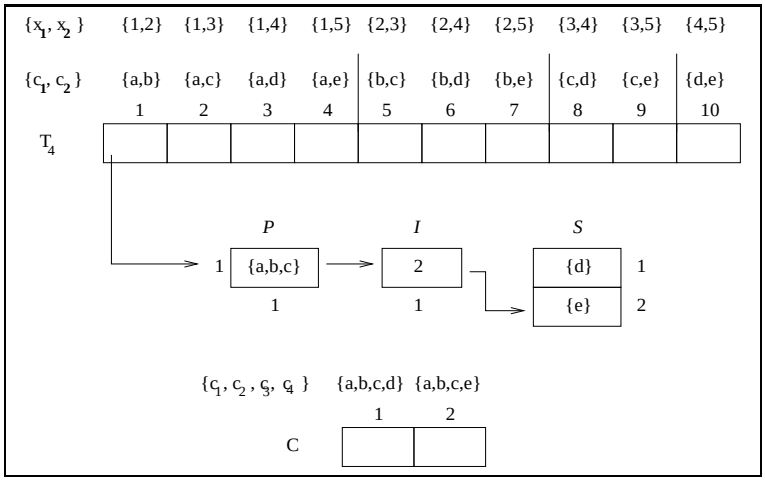


Figura 2.3: A estrutura de dados T_4 construída pelo algoritmo *kDCI++*.

verifica, seqüencialmente, quais os candidatos cujos prefixos estão armazenados em P e os sufixos correspondentes em S, posicionados entre $T_k[i]$ e $T_k[i+1]$, estão presentes em t e, então, incrementa seus respectivos contadores em C.

2.2.2 Fase de Interseção

Assim que a base de dados, reduzida gradativamente a cada iteração, torna-se suficientemente pequena para ser carregada em memória principal, o algoritmo *kDCI++* cria uma representação verticalizada da base de dados, a partir da qual os suportes dos conjuntos candidatos serão obtidos na próxima iteração.

A representação verticalizada de uma base de dados pode ser vista como um conjunto de M_k listas de bits de tamanho $|N_k|$, onde M_k é o conjunto de itens presentes na base de dados da iteração k e N_k é o conjunto de transações, identificadas por TID, presentes na base de dados da mesma iteração k. Cada lista está associada a um determinado item da base de dados corrente. Desta forma, se o j-ésimo bit da lista associada ao item i tem o valor 1, significa que o item i está presente na transação j. A Figura 2.4 ilustra a representação verticalizada da base de dados da Figura 2.1. No exemplo, $M_k = \{a, b, c, d, e, f, g, h, o\}$ e $N_k = \{1, 2, 3, 4, 5, 6, 7\}$. Note, por exemplo, que o item f só está presente na segunda transação da base de dados considerada. Desta forma, apenas o segundo bit de sua lista tem o valor 1.

		TID						
		1	2	3	4	5	6	7
Itens	a	1	1	0	1	1	1	1
	b	1	1	0	1	1	1	0
	c	1	1	0	0	1	1	0
	d	1	0	1	1	1	0	0
	e	0	1	1	0	0	1	0
	f	0	1	0	0	0	0	0
	g	1	0	0	0	0	0	0
	h	0	0	1	0	0	0	0
	o	0	1	0	0	0	0	0

Figura 2.4: Representação verticalizada da base de dados da Tabela 2.1.

No início de cada iteração $k \geq 2$, o algoritmo *kDCI++* verifica se a representação verticalizada da base de dados é suficientemente pequena para ser construída em memória principal. Esta condição é verificada através da seguinte heurística adotada pelos autores. Se o valor de $|M_k| * |N_k|$ (em termos de memória consumida) é menor que $\frac{1}{3}$ da memória principal total disponível no sistema, a representação verticalizada poderá ser construída.

Caso a condição seja verdadeira, a base de dados verticalizada é atualizada a cada transação lida durante a contagem dos conjuntos candidatos de tamanho k e é utilizada na contagem dos suportes dos conjuntos candidatos de tamanho $k+1$, na iteração seguinte.

Contagem dos Conjuntos Candidatos a partir de *TIDlists*

Durante a fase de interseção, o suporte dos conjuntos candidatos é contado a partir da base de dados verticalizada.

Observe que o suporte de um determinado conjunto de itens pode ser obtido através da interseção das listas de bits, presentes na base verticalizada, associadas aos itens que o compõe. O número de bits 1 presentes no resultado da interseção representa o suporte do conjunto. Por exemplo, a interseção das listas associadas aos itens a e b, da Figura 2.4, contém apenas cinco bits 1, o que indica que o suporte

do conjunto $\{a, b\}$ é 5.

Desta forma, durante a fase de interseção, o algoritmo *kDCI++* obtém o conjunto F_k da seguinte forma. Para cada conjunto candidato c de tamanho k, gerado através da combinação dos conjuntos freqüentes de F_{k-1}, o algoritmo encontra a interseção das k listas associadas aos itens de c e conta os bits 1 presentes no resultado. Se o número de bits 1 encontrados for maior ou igual ao suporte mínimo, c é incluído em F_k.

Note que, para se obter o suporte de um conjunto candidato de tamanho k, é preciso realizar $k - 1$ interseções. Entretanto, os autores propuseram a seguinte otimização a fim de diminuir o custo computacional deste procedimento. O algoritmo *kDCI++* utiliza uma estrutura de dados, aqui denominada $Cache$, baseada em uma matriz (*"cache" buffer*) que armazena as $k - 2$ interseções intermediárias encontradas para a determinação do suporte do último conjunto candidato avaliado. Cada linha i de $Cache$ armazena o resultado da i-ésima interseção encontrada.

Como os conjuntos candidatos são gerados em ordem lexicográfica, há uma grande probabilidade de dois conjuntos candidatos c_1 e c_2, gerados consecutivamente, terem um mesmo prefixo. Desta forma, se c_1 e c_2 têm os primeiros $i \geq 2$ itens idênticos, para determinar o suporte de c_2, as primeiras $i - 1$ interseções não precisarão ser encontradas, já que os resultados obtidos durante a contagem do suporte do candidato c_1 estarão armazenados em $Cache$.

Como exemplo, considere a base de dados da Tabela 2.1 e o suporte mínimo igual a duas transações. Suponha que a representação verticalizada da base de dados corrente tenha sido gerada durante a segunda iteração para ser utilizada na contagem dos conjuntos candidatos de tamanho 3. A Figura 2.5 ilustra a representação verticalizada construída (note que os itens f, g, h e o não estão na base verticalizada, já que foram podados durante a poda global aplicada às transações antes da contagem dos conjuntos candidatos de tamanho 2).

Lembrando que o conjunto $F_2 = \{\{a, b : 5\}, \{a, c : 4\}, \{a, d : 3\}, \{a, e : 2\}, \{b, c : 4\}, \{b, d : 3\}, \{b, e : 2\}, \{c, d : 2\}, \{c, e : 2\}\}$, o primeiro conjunto candidato

		TID						
		1	2	3	4	5	6	7
Itens	a	1	1	0	1	1	1	1
	b	1	1	0	1	1	1	0
	c	1	1	0	0	1	1	0
	d	1	0	1	1	1	0	0
	e	0	1	1	0	0	1	0

Figura 2.5: Representação verticalizada da base de dados construída durante a segunda iteração.

de tamanho 3 gerado é $c_1 = \{a, b, c\}$. Para contar o suporte de c_1, o algoritmo encontra, primeiramente, a interseção das listas de bits associadas aos itens a e b e armazena o resultado em $Cache$. Em seguida, é encontrada a interseção da lista de bits associada ao item c com a lista de bits armazenada em $Cache$. O suporte do conjunto c_1 é, então, o número de bits 1 presentes no resultado da última interseção que, no exemplo, é 4. Como o suporte do conjunto c_1 encontrado é maior do que o suporte mínimo, c_1 é incluído em F_3.

Após a inclusão de c_1 em F_3, o algoritmo gera o próximo conjunto candidato $c_2 = \{a, b, d\}$. Como c_2 tem os dois primeiros itens idênticos aos de c_1 (conjunto candidato avaliado anteriormente), não é necessário encontrar a primeira interseção, já que seu resultado foi armazenado na primeira linha de $Cache$ durante a contagem do conjunto candidato anterior, c_1. Desta forma, para determinar o suporte de c_2, basta encontrar a interseção da lista de bits associada ao item d com a lista de bits já armazenada em $Cache$. Note que apenas uma interseção foi realizada. Como o número de bits 1 encontrados no resultado da interseção é maior do que o suporte mínimo, c_2 é incluído em F_3. O procedimento se repete até que todos os conjuntos candidatos de tamanho 3 tenham sido avaliados.

Vale observar que, durante a fase de interseção, não é necessário armazenar os conjuntos candidatos em memória já que o suporte de cada candidato é obtido logo após sua geração. Além disso, não há poda de candidatos durante a geração.

2.2.3 Algumas otimizações

Com o objetivo de reduzir o custo computacional da fase de interseção, os autores do algoritmo *kDCI++* propõem algumas otimizações para serem aplicadas durante o processamento desta fase, de acordo com as características da base de dados.

Uma das contribuições do algoritmo *kDCI++* é a utilização de uma técnica, baseada nos estudos teóricos apresentados em [1], a qual permite que, em algumas situações específicas, o suporte de conjuntos candidatos possa ser deduzido sem que sejam contados na base de dados. Em [17], esta técnica é chamada pelos autores do *kDCI++* de *Pattern Counting Inference* e, neste trabalho, será referenciada como *técnica de dedução*.

A técnica utilizada é baseada no conceito de classe de equivalência e de *key pattern* [1]. Dois conjuntos de itens I e I' pertencem a mesma classe de equivalência se I e I' têm o mesmo suporte e estão presentes nas mesmas transações da base de dados. Um conjunto de itens I é considerado um *key pattern* se nenhum subconjunto de I pertence a mesma classe de equivalência de I [1]. De forma geral, a técnica permite que o suporte de um conjunto de itens I que não é um *key pattern* possa ser deduzido a partir de subconjuntos de I que estão presentes na mesma classe de equivalência de I.

Note que a aplicação da técnica é vantajosa apenas se o número de *key patterns* presentes na base de dados é pequeno. Como não é possível calcular este número antes do término da execução do algoritmo, *kDCI++* se baseia na média dos suportes dos itens freqüentes. Se o valor encontrado é alto o suficiente (parâmetro conhecido no início do processamento), significa que longos conjuntos de itens poderão ser freqüentes e que o número de *key patterns* deverá ser pequeno se comparado ao número de conjuntos freqüentes. Esta condição é verificada no início da fase de interseção.

Outras otimizações são aplicadas de acordo com a densidade da base de dados verticalizada. Conforme descrito em [19] por Orlando *et. al.*, quanto mais

densa for uma base de dados, mais parecidas serão suas transações, ou seja, apenas alguns itens irão diferir de uma transação para outra. Desta forma, o algoritmo *kDCI++* verifica a densidade da base de dados verticalizada através da seguinte heurística. Se $s\%$ dos itens estiverem presentes nas mesmas $p\%$ transações, a base de dados é considerada densa se o valor $s\% * p\%$ for maior que um determinado "índice de densidade" (*density threshold*) d, estabelecido no início do processamento. Caso contrário, a base de dados verticalizada é considerada esparsa.

A seguir, as otimizações são classificadas em dois grupos de acordo com a densidade da base de dados e apresentadas separadamente.

Otimizações para Bases Esparsas

As listas de bits de uma base de dados verticalizada esparsa é caracterizada por terem grandes seqüências de bits 0. Desta forma, ao encontrar a interseção das duas primeiras listas de bits de um determinado conjunto candidato c, o algoritmo *kDCI++* identifica as posições de início e fim das seqüências de 0 presentes no resultado da interseção e armazena o resultado na estrutura $Cache$. Assim, as demais interseções necessárias para o cálculo do suporte de c são feitas ignorando as seqüências de 0 presentes na lista de bits armazenadas em $Cache$, considerando somente as seqüências de bits onde exista pelo menos um bit 1. Os conjuntos candidatos gerados, subseqüentemente, que possuem o mesmo prefixo de tamanho $i \geq 2$, também levarão em consideração as informações das seqüências de 0 presentes em $Cache$.

Outra otimização aplicada às bases de dados consideradas esparsas é a poda de colunas, o que reduz o tamanho da base de dados verticalizada e, conseqüentemente, o custo de cada interseção. No final de cada iteração k, uma coluna pode ser podada da base de dados verticalizada caso não haja nenhum bit 1 nas posições correspondentes aos itens presentes nos conjuntos de F_k (caso a transação não contenha nenhum dos itens presentes nos conjuntos de F_k). Como a poda de colunas exige um grande esforço computacional, esta só é aplicada quando uma quantidade significativa de colunas puder ser podada.

Otimizações para Bases Densas

Nas bases de dados consideradas densas, as listas de bits, além de conterem grandes seqüências de bits 1, são muito parecidas entre si. Desta forma, a fim de reduzir o custo da fase de interseção, antes do início desta fase, o algoritmo *kDCI++* reordena as colunas da base de dados verticalizada de forma a mover seqüências idênticas de bits presentes nas listas associadas aos itens mais freqüentes, para as primeiras colunas consecutivas da base. O número n de bits 1 presentes nas seqüências idênticas, o tamanho s destas seqüências, assim como o conjunto de itens I cujas listas de bits possuem tais seqüências, são identificados. De forma geral, durante a contagem de um determinado conjunto candidato c, caso os itens presentes em c pertençam ao conjunto I, as interseções das listas de bits serão feitas a partir apenas da posição $s + 1$ e o suporte de c é obtido somando-se o número de bits 1 presentes no resultado das interseções com o valor n. Como os itens mais freqüentes têm grande probabilidade de estarem presentes nos conjuntos candidatos, esta otimização é capaz de reduzir, significativamente, o custo da fase de interseção.

2.3 Algoritmo *FP-growth*

Segundo Han *et. al.* em [12], os algoritmos da classe apriori têm como principal custo computacional a geração demasiada de conjuntos candidatos, principalmente quando o suporte mínimo considerado é baixo. Além disso, são feitas várias leituras à base de dados para a contagem do suporte destes conjuntos. Por esta razão, os autores propõem um novo algoritmo, denominado *FP-growth* [12]. Este algoritmo extrai os conjuntos freqüentes sem a geração de candidatos a partir de uma estrutura de dados denominada *FP-tree* (*Frequent Pattern Tree*). A *FP-tree* representa a base de dados de forma compactada e é acompanhada de uma estrutura auxiliar, denominada *tabela de cabeçalhos* (*header-table*), que tem como objetivo facilitar os percursos na *FP-tree*.

O algoritmo *FP-growth* inicia a busca por conjuntos freqüentes a partir dos conjuntos freqüentes de tamanho 1. A partir daí, o algoritmo considera um item

freqüente de cada vez e gera, recursivamente, todos os conjuntos freqüentes que contêm aquele item como último item (sufixo). A identificação dos conjuntos freqüentes de tamanho $k \geq 2$ se dá a partir de projeções da base de dados, denominadas bases de dados condicionadas (*condicional pattern base*) e suas respectivas *FP-trees* condicionadas (*condicional FP-trees*), também chamadas na literatura de projeções físicas da base de dados.

Esta seção está organizada da seguinte forma. Primeiramente, na Subseção 2.3.1, a estrutura *FP-tree* e os passos para a sua construção são ilustrados através de um exemplo. A Subseção 2.3.2 destina-se à apresentação do algoritmo *FP-growth*, também através de um exemplo ilustrativo e, na Subseção 2.3.3, são apresentadas variações do algoritmo *FP-growth*.

2.3.1 A Estrutura de Dados *FP-tree*

A construção da estrutura de dados *FP-tree* é baseada na seguinte idéia. Conjuntos de itens presentes em mais de uma transação da base de dados podem ser representados por um único ramo na *FP-tree* desde que o número de ocorrências dos conjuntos de itens compartilhados seja contabilizado. Para facilitar a identificação de conjuntos de itens idênticos nas transações, os itens das transações são ordenados segundo um mesmo critério. No caso do algoritmo *FP-growth*, os itens são ordenados decrescentemente pelo suporte, o que aumenta a possibilidade de compartilhamento de conjuntos de itens entre as transações.

Considere, por exemplo, a *FP-tree* da Figura 2.6, construída a partir da base de dados da Tabela 2.1. O suporte mínimo considerado é igual a duas transações.

A *FP-tree* consiste de um nó raiz, ramos compostos de nós representando os itens freqüentes e uma tabela de cabeçalhos que tem como objetivo facilitar os percursos na *FP-tree*. Cada nó n na *FP-tree* é composto de três campos distintos: *nome*, *contador* (número após ":") e um *ponteiro*. O nome do nó identifica o item da base de dados que está sendo representado. No caso da raiz, este campo é nulo. O contador do nó armazena o número de transações que compartilham o conjunto

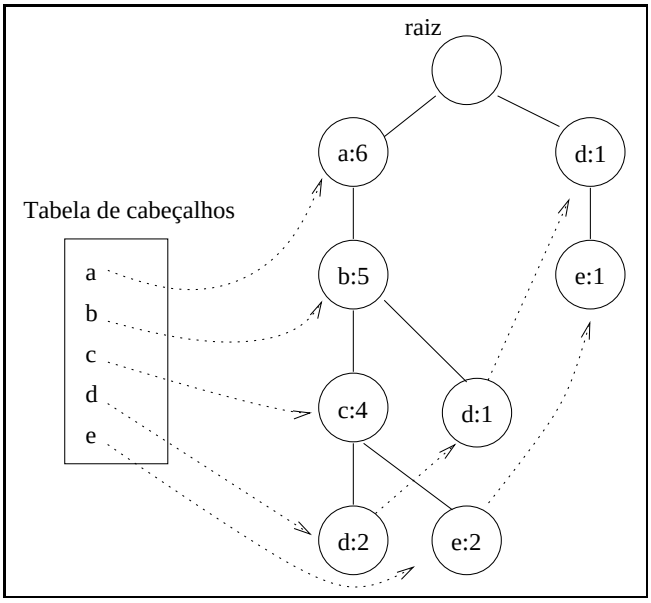


Figura 2.6: A estrutura de dados *FP-tree*.

de itens formado pelo sub-ramo composto pelos nós desde a raiz até o nó n. Por exemplo, o sub-ramo da *FP-tree* $(a : 6, b : 5, c : 4)$ indica que o conjunto de itens $\{a, b, c\}$ está presente em quatro transações da base de dados considerada. Da mesma forma, o sub-ramo $(a : 6, b : 5)$ indica que o conjunto de itens $\{a, b\}$ está presente em cinco transações e o conjunto $\{a\}$, está presente em seis transações. O ponteiro de cada nó aponta para um nó presente na *FP-tree* cujo nome é o mesmo do nó n.

Na tabela de cabeçalhos, cada entrada é composta por dois campos: *nome*, que tem a mesma função do nome de um nó, e um *ponteiro*, que aponta para um nó de mesmo nome na *FP-tree* (os nós de mesmo nome são ligados pelos seus ponteiros, formando uma lista que inicia no ponteiro correspondente na tabela de cabeçalhos).

A estrutura de dados *FP-tree* é construída a partir de duas leituras da base de dados. A primeira leitura tem como objetivo obter a lista de itens freqüentes L, ordenada decrescentemente pelo suporte. No caso da base de dados e suporte mínimo considerados, $L = \{\{a : 6\}, \{b : 5\}, \{c : 4\}, \{d : 4\}, \{e : 3\}\}$ (a ordem dos itens de mesmo suporte é arbitrária). Em seguida, a raiz da árvore e a tabela de cabeçalhos contendo uma entrada para cada item freqüente são criadas.

Após a criação da tabela de cabeçalhos, é feita uma segunda leitura da base

de dados a fim de construir a *FP-tree*. Para o caso da base de dados considerada, a primeira transação ($\{a, b, c, d, g\}$), ordenada de acordo com a lista L, é incluída na *FP-tree* em forma de um único ramo: $(a : 1, b : 1, c : 1, d : 1)$ (os itens não freqüentes não são incluídos na *FP-tree*). Cada item presente na transação lida é representado por um nó na *FP-tree* e os nós criados são ligados de acordo com a ordem em que os itens aparecem na transação após a ordenação. Como os itens estão sendo incluídos na *FP-tree* pela primeira vez, os contadores de cada novo nó têm valor 1 e os ponteiros correspondentes na tabela de cabeçalhos são atualizados.

A segunda transação ($\{a, b, c, e, f, o\}$) tem o conjunto de itens $\{a, b, c\}$ em comum com o conjunto de itens representado pelo sub-ramo $(a : 1, b : 1, c : 1)$, já incluído na *FP-tree*. Soma-se 1 ao contador de cada nó em comum e, em seguida, o nó $(e : 1)$ é incluído como filho do nó $(c : 2)$. O ponteiro correspondente ao novo nó criado é atualizado na tabela de cabeçalhos.

A terceira transação ($\{d, e, h\}$) não tem conjuntos de itens em comum com os conjuntos de itens já presentes na *FP-tree*. Desta forma, um novo ramo $(d : 1, e : 1)$ é criado partindo da raiz.

A quarta transação ($\{a, b, d\}$) tem o conjunto de itens $\{a, b\}$ em comum com o conjunto de itens representado pelo sub-ramo $(a : 2, b : 2)$, já incluído na *FP-tree*. Soma-se 1 ao contador de cada nó em comum e, em seguida, o nó $(d : 1)$ é incluído como filho do nó $(b : 2)$.

Após a leitura da quinta transação ($\{a, b, c, d\}$), soma-se 1 aos contadores dos nós que representam o conjunto de itens $\{a, b, c, d\}$ presentes no primeiro ramo incluído na *FP-tree*. Após a leitura da sexta transação ($\{a, b, c, e\}$), soma-se 1 aos contadores dos nós que representam o conjunto de itens $\{a, b, c, e\}$ presentes no segundo ramo incluído na *FP-tree* e, após a leitura da última transação ($\{a\}$), soma-se 1 apenas ao contador do nó $(a : 5)$, também já existente.

Observe que os conjuntos de itens presentes em mais de uma transação da base de dados são representados, na *FP-tree*, por apenas um único ramo. Note também que podem existir mais de um nó representando um mesmo item freqüente

i e que o suporte de i é dado pelo somatório dos contadores dos respectivos nós.

2.3.2 Extração de Conjuntos Freqüentes a partir da Estrutura de Dados *FP-tree*

O algoritmo *FP-growth* extrai os conjuntos freqüentes presentes em uma base de dados a partir da estrutura *FP-tree*, considerando um item de cada vez e gerando, recursivamente, todos os conjuntos freqüentes que contêm aquele item como sufixo. O primeiro item processado é o último item na tabela de cabeçalhos. O segundo, é o penúltimo item e, assim, sucessivamente.

O algoritmo é baseado na seguinte idéia [12]. Sejam X e Y dois conjuntos de itens. O suporte de $X \cup Y$ é igual ao suporte de Y, considerando apenas as transações da base de dados que contêm X. A essa projeção da base de dados é dado o nome de *base de dados condicionada* de X (*X's conditional pattern base*) e à *FP-tree* correspondente a essa projeção da base de dados é dado o nome de *FP-tree condicionada* de X (*X's conditional FP-tree*).

A seguir, o algoritmo *FP-growth* é ilustrado através de um exemplo. Considere a *FP-tree* da Figura 2.6, lembrando que o suporte mínimo é igual a duas transações.

Após a criação da *FP-tree*, o algoritmo *FP-growth* inicia a extração dos conjuntos freqüentes a partir do último item na tabela de cabeçalhos que, no exemplo considerado, é o item e. Primeiramente, o algoritmo tem como objetivo encontrar o conjunto de itens freqüente de tamanho 1 com sufixo e e seu respectivo suporte. Para encontrar o suporte, basta seguir os ponteiros dos nós de nome e, iniciando pelo ponteiro correspondente na tabela de cabeçalhos, e somar os respectivos contadores. Feito isso, o algoritmo obtém o primeiro conjunto freqüente $\{e : 3\}$.

O próximo passo tem o objetivo de encontrar todos os conjuntos freqüentes de tamanho 2 que tenham o sufixo e. Observe que, para encontrar os conjuntos de itens que tenham e como sufixo, basta considerar apenas os sub-ramos da *FP-tree*

desde a raiz até os nós de nome e. Os sub-ramos encontrados formarão uma nova base de dados, denominada base de dados condicionada de e, representada por BD_e.

Seguindo os ponteiros dos nós de nome e, o algoritmo encontra dois sub-ramos: $(a:6, b:5, c:4, e:2)$ e $(d:1, e:1)$. Como o primeiro sub-ramo indica que o conjunto de itens $\{a, b, c\}$ aparece duas vezes na base de dados juntamente com o item e e o segundo sub-ramo, que o conjunto de itens $\{d\}$ aparece uma vez na base de dados juntamente com o item e, os sub-ramos $(a:2, b:2, c:2)$ e $(d:1)$ formarão BD_e.

A partir deste momento, o algoritmo repete os passos iniciais, considerando BD_e como a nova base de dados. Primeiramente, cria uma nova tabela de cabeçalhos com uma entrada para cada item freqüente de BD_e: a, b, c (os suportes são obtidos através da soma dos contadores) e a raiz de uma nova *FP-tree*, *FP-tree* condicionada de e, chamada FP_e. Em seguida, um segundo percurso na *FP-tree* cria os nós de FP_e a partir de BD_e. A FP_e construída é ilustrada na Figura 2.7.

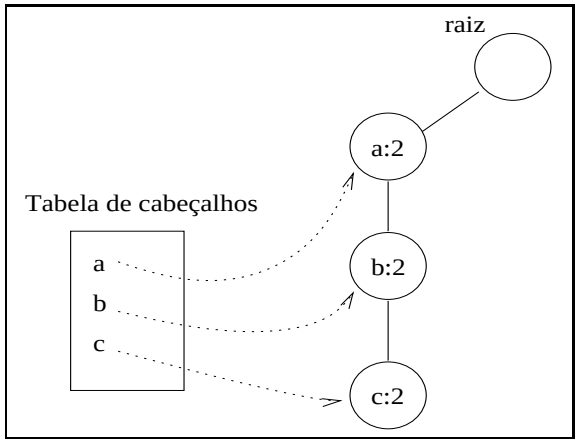


Figura 2.7: *FP-tree* condicionada de e.

Feita a construção de FP_e, seguindo os ponteiros do item c (último item na nova tabela de cabeçalhos considerada), o algoritmo encontra o conjunto freqüente $\{c, e : 2\}$ e a base de dados condicionada de ce, BD_{ce}, formada pelo sub-ramo $(a : 2, b : 2)$. Mais uma vez, o algoritmo cria uma nova tabela de cabeçalhos com uma entrada para cada item freqüente de BD_{ce} e uma nova *FP-tree*, a FP_{ce}, em busca dos itens que juntamente com o sufixo ce têm suporte mínimo. A FP_{ce}

construída é ilustrada na Figura 2.8 e o primeiro conjunto freqüente encontrado é $\{b, c, e : 2\}$.

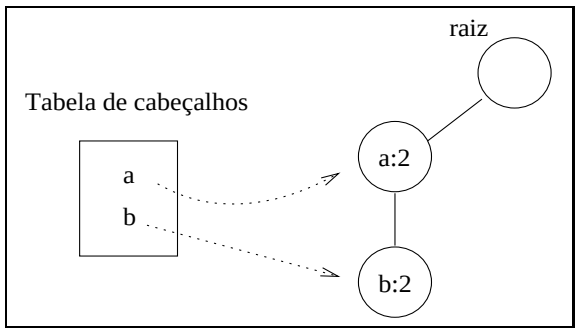


Figura 2.8: *FP-tree* condicionada de *ce*.

Após a criação de uma nova *FP-tree*, FP_{bce}, e de uma nova tabela de cabeçalhos, o algoritmo gera o conjunto freqüente $\{a, b, c, e : 2\}$. Neste ponto, não é mais possível a formação de bases condicionadas. Desta forma, o algoritmo repete os passos anteriores, iniciando a busca por conjuntos freqüentes que terminam com o conjunto *ace*, já que o item *a* é o próximo item (penúltimo) da tabela de cabeçalhos anterior (FP_{ce}).

No final do processamento do item *e*, os seguintes conjuntos freqüentes são gerados: $\{e : 3\}, \{c, e : 2\}, \{b, c, e : 2\}, \{a, b, c, e : 2\}, \{a, c, e : 2\}, \{b, e : 2\}, \{a, b, e : 2\}$ e $\{a, e : 2\}$. Observe que quando a *FP-tree* corrente tem apenas um único ramo, os conjuntos freqüentes obtidos são formados pela combinação de todos os itens presentes neste ramo, tendo como sufixo o item ou conjunto de itens que está sendo processado no momento. Desta forma, quando o algoritmo *FP-growth* identifica tal situação, gera os conjuntos freqüentes através da combinação dos itens presentes no ramo e inicia o processamento do próximo item na tabela de cabeçalhos corrente.

O algoritmo *FP-growth* repete os passos anteriores, recursivamente, até que todos os itens da tabela de cabeçalhos inicial tenham sido processados.

2.3.3 Variações do algoritmo *FP-growth*

Observe que, no algoritmo *FP-growth*, a cada nova ativação recursiva do procedimento, são geradas novas projeções da base de dados (bases de dados condicionadas) e novas *FP-trees* (*FP-trees* condicionadas). Conforme observado em [29], à medida que o tamanho dos conjuntos freqüentes que estão sendo minerados cresce, este procedimento pode se tornar bastante custoso computacionalmente.

Desta forma, em [29], os autores propõem a primeira variação do algoritmo *FP-growth*, denominada *TD-FP-Growth*, em que os itens presentes na tabela de cabeçalhos corrente são processados, um a um, a partir do primeiro item. Este novo procedimento não necessita de construções de *FP-trees* condicionadas para extração de conjuntos freqüentes de tamanho $k > 1$.

Apesar do algoritmo *FP-growth* ter se mostrado bastante eficiente quando comparado aos algoritmos *Apriori* e *Tree-Projection* (última variação do algoritmo *Apriori* até então), conforme experimentos apresentados em [32], o algoritmo não tem o mesmo bom desempenho quando a base de dados considerada é muito grande e esparsa. Isto se explica devido à pouca compactação da base de dados atingida pela *FP-tree*, nestas situações.

Neste contexto, variações mais recentes do algoritmo *FP-growth* consideram características da base de dados, como sua densidade. Os algoritmos *H-Mine* [21] e *OpportuneProject* [14] aprimoraram o algoritmo *FP-growth* alterando seu comportamento conforme a densidade da base de dados. *H-Mine* utiliza diferentes estruturas de dados de acordo com a densidade da base. Se a base de dados é considerada esparsa, a estrutura utilizada é baseada em um vetor. Caso contrário, a própria *FP-tree* é construída. O algoritmo *OpportuneProject* utiliza as principais idéias dos algoritmos *TD-FP-Growth* e *H-Mine*, adotando a melhor estratégia de acordo com às características da base de dados considerada.

Recentemente, duas novas variações do algoritmo *FP-growth* foram propostas: *PatriciaMine* [22] e *FP-growth** [10]. Ambos obtiveram bons resultados, segundo os experimentos realizados no *Workshop FIMI'03*, e serão descritos nas

Seções 2.4 e 2.5, respectivamente.

2.4 Algoritmo *PatriciaMine*

O algoritmo PatriciaMine [22] é uma recente variação do algoritmo *FP-growth* que tem como principal contribuição a utilização de uma nova estrutura de dados para representar a base de dados de forma compactada. Diferente das últimas variações, *PatriciaMine* utiliza apenas uma única estrutura para representação de bases de dados esparsas ou densas, denominada *Patricia trie*. Segundo resultados apresentados em [22], a estrutura *Patricia trie* mostrou ser bastante econômica, em termos de espaço consumido em memória, quando comparada com a estrutura *FP-tree*, principalmente em se tratando de bases de dados esparsas.

A extração dos conjuntos freqüentes é feita baseada na idéia do algoritmo *TD-FP-Growth*. Entretanto, além de outras otimizações propostas, uma característica interessante do algoritmo é ter o comportamento iterativo, ao invés de recursivo, como as demais variações do algoritmo *FP-growth*.

2.4.1 A Estrutura de Dados *Patricia Trie*

A estrutura *Patricia trie* é uma variação compactada da estrutura *FP-tree*, descrita anteriormente. Na *Patricia trie*, um sub-ramo da *FP-tree* composto por nós com o mesmo valor no contador e que, com exceção do último nó deste sub-ramo, possuem um único filho é representado por um único nó.

Para melhor ilustrar as características da estrutura, considere a *Patricia trie* da Figura 2.9, construída a partir da base de dados da Tabela 2.1. Observe que o sub-ramo $(d:1, e:1)$ da *FP-tree* ilustrada na Figura 2.6 é representado na *Patricia trie* por um único nó. O nó compactado herda o valor do contador em comum que, no exemplo, tem o valor 1 (se o nó $(e:1)$ tivesse filhos, o nó compactado herdaria seus filhos). Note que, na *Patricia trie*, o nome do nó pode representar mais de um item.

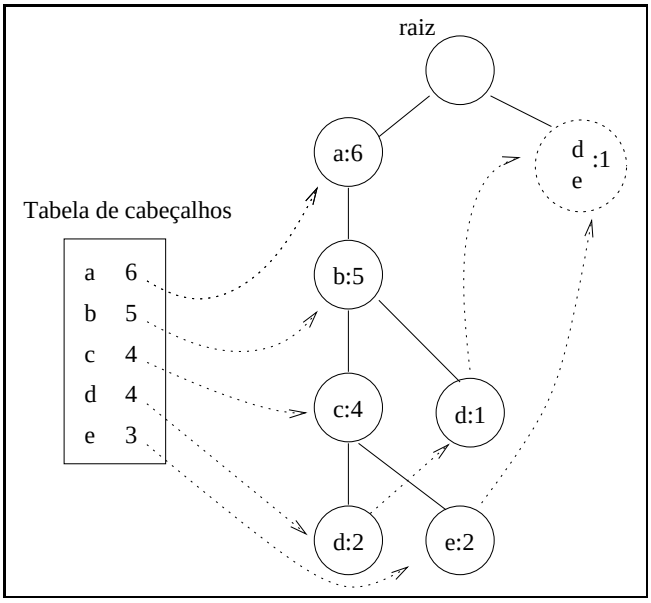


Figura 2.9: A estrutura de dados *Patricia trie*.

Assim como a *FP-tree*, a *Patricia trie* também é acompanhada de uma tabela de cabeçalhos.

2.4.2 Extração de Conjuntos Freqüentes a partir da Estrutura de Dados *Patricia Trie*

O algoritmo *PatriciaMine* extrai os conjuntos freqüentes presentes em uma base de dados a partir da estrutura *Patricia trie* correspondente.

Considere, novamente, a base de dados e suporte mínimo utilizados até o momento. No lugar da *FP-tree*, o algoritmo *PatriciaMine* constrói, no início do processamento, a *Patricia trie* correspondente à base de dados considerada. A estrutura *Patricia trie* construída é ilustrada na Figura 2.9. No algoritmo *PatriciaMine*, cada entrada da tabela de cabeçalhos, além de conter os campos *nome* e *ponteiro*, também contém o campo *contador* que armazena o suporte do item identificado pelo campo *nome*.

A extração dos conjuntos freqüentes é iniciada a partir do primeiro item da tabela de cabeçalhos. No exemplo considerado, pelo item a. Como o item a aparece

apenas no nível mais alto da *FP-tree*, do seu processamento obtém-se apenas o conjunto freqüente $\{a : 6\}$, já que o campo contador referente ao item a, na tabela de cabeçalhos, é maior do que o suporte mínimo (tem valor 6). Em seguida, o algoritmo inicia o processamento do item b, segundo item da tabela de cabeçalhos.

Da mesma forma, como o contador de b na tabela de cabeçalhos é maior do que o suporte mínimo (tem valor 5), obtém-se o conjunto freqüente $\{b : 5\}$ e, a partir deste ponto, a busca pelos demais conjuntos freqüentes de sufixo b é feita seguindo os ponteiros dos nós que contêm o nome b, na *Patricia trie*. Assim como no algoritmo *FP-growth*, basta considerar os sub-ramos da *Patricia trie* desde a raiz até os nós que contêm o nome b.

Observe que, como o algoritmo *PatriciaMine* processa os itens da tabela de cabeçalhos a partir do primeiro, os itens representados pelos nós que se encontram acima dos nós de nome b já foram anteriormente processados. Desta forma, pode-se obter a base de dados condicionada de b e uma projeção física desta base de dados, aqui chamada de *Patricia trie condicionada* de b (P_b), apenas atualizando-se os contadores dos nós dos sub-ramos encontrados e das entradas correspondentes na tabela de cabeçalhos, sem a necessidade de copiá-los para uma nova *Patricia trie*. No exemplo, apenas o nó $(a : 6)$ é atualizado para $(a : 5)$, já que $(a : 6)$ é o único nó no sub-ramo encontrado. O contador do item a, na tabela de cabeçalhos inicial, também é atualizado para 5. P_b é ilustrada na Figura 2.10. Note que P_b foi gerada na própria *Patricia trie* corrente. Nenhuma nova *Patricia trie* ou tabela de cabeçalhos foi gerada.

Após a construção de P_b, o algoritmo repete, iterativamente, os passos anteriores tendo, agora, como *Patricia trie*, a P_b. Como o contador de a na tabela de cabeçalhos é maior do que o suporte mínimo (tem valor 5), obtém-se o novo conjunto freqüente $\{ab : 5\}$. Neste ponto, como existe apenas uma única entrada atualizada na tabela de cabeçalhos, o processamento do item b é finalizado e o algoritmo inicia o processamento do item c, terceiro item na tabela de cabeçalhos.

O mesmo procedimento é repetido, iterativamente, para todos os itens da tabela de cabeçalhos, até que todos tenham sido processados.

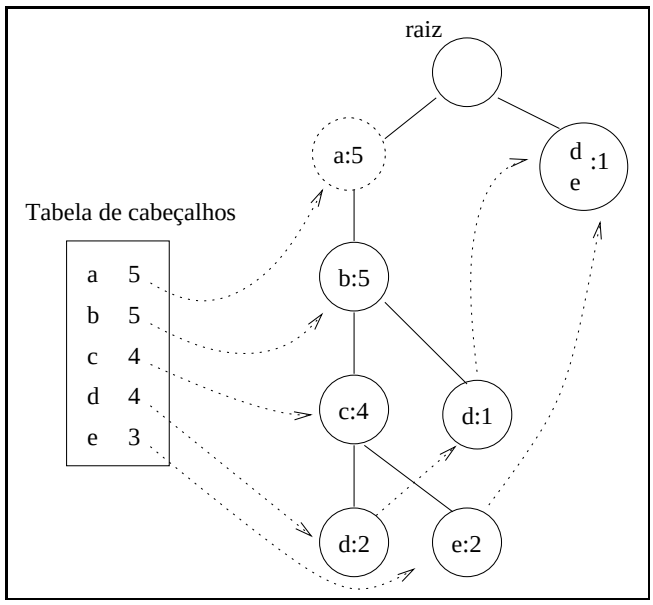


Figura 2.10: *Patricia trie* condicionada de b gerada na própria *Patricia trie* corrente.

2.4.3 Algumas Otimizações

Conforme descrito anteriormente, o algoritmo *PatriciaMine*, assim como o algoritmo *TD-FP-Growth*, não necessita da construção de projeções físicas da base de dados para a extração de conjuntos freqüentes de tamanho $k > 1$. Entretanto, de acordo com alguns experimentos realizados, os autores do *PatriciaMine* concluíram que a construção de tais estruturas pode trazer vantagens para o algoritmo nos casos em que a extração de conjuntos freqüentes a partir de uma nova projeção física se prolonga por um período suficientemente grande, compensando o custo computacional de sua construção. Desta forma, o algoritmo *PatriciaMine* utiliza uma heurística que, a partir das características da *Patricia trie* corrente, verifica se será vantajosa a construção de uma projeção física a ser utilizada no decorrer da execução. Se esta condição for verdadeira, a extração dos conjuntos freqüentes subseqüentes é, a partir de então, realizada utilizando-se uma nova *Patricia trie*.

2.5 Algoritmo *FPgrowth**

Conforme observado em [10], testes do algoritmo *FP-growth*, realizados por Grahne e Zhu, mostraram que 80% do tempo de execução do algoritmo é destinado aos percursos nas *FP-trees* construídas durante sua execução.

De fato, cada construção de uma nova *FP-tree* requer dois percursos na *FP-tree* anterior. O primeiro percurso tem como objetivo obter os itens freqüentes presentes na base condicionada do item ou conjunto de itens que está sendo analisado no momento. Cada item freqüente encontrado terá uma entrada na tabela de cabeçalhos da *FP-tree* condicionada que será construída. O segundo percurso é destinado à construção propriamente dita da *FP-tree* condicionada, a partir da base de dados condicionada identificada.

Desta forma, a fim de reduzir o número de percursos realizados nas *FP-trees*, propõe-se em [10] o algoritmo *FPgrowth**, que possibilita que *FP-trees* condicionadas sejam construídas a partir de um único percurso na *FP-tree* corrente através da utilização de uma estrutura de dados baseada em um vetor. Este vetor é denominado V e será descrito a seguir.

2.5.1 A Estrutura de Dados V

Para melhor ilustrar a utilização da estrutura de dados V pelo algoritmo *FPgrowth**, considere a mesma base de dados e suporte mínimo utilizados nas seções anteriores. Assim como no algoritmo *FP-growth*, o algoritmo *FPgrowth** gera os conjuntos freqüentes a partir da estrutura *FP-tree*, construída através de duas leituras da base de dados. A primeira leitura tem o mesmo objetivo da primeira leitura no algoritmo *FP-growth*: gerar a lista L de itens freqüentes de tamanho 1 que, no exemplo considerado, é $L = \{\{a : 6\}, \{b : 5\}, \{c : 4\}, \{d : 4\}, \{e : 3\}\}$, ordenada decrescentemente pelo suporte. Em seguida, o algoritmo *FPgrowth** cria a tabela de cabeçalhos com uma entrada para cada item freqüente de L e a raiz da *FP-tree*, como no algoritmo *FP-growth*.

Já durante a segunda leitura da base de dados, além de construir a *FP-tree*, o algoritmo *FPgrowth** constrói o vetor V, que tem como objetivo armazenar os suportes de todos os conjuntos de tamanho 2 presentes nas transações da base de dados considerada. A Figura 2.11 ilustra o vetor V populado a partir das transações da base de dados. Cada entrada do vetor armazena o suporte de um determinado conjunto de itens $c = \{c_1, c_2\}$, onde c_1 e c_2 estão ordenados lexicograficamente. Por exemplo, a entrada $V[a, b]$ armazena o suporte do conjunto de itens $\{a, b\}$ na base de dados.

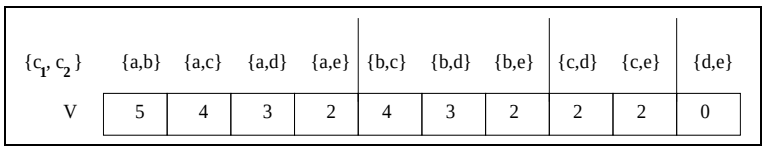


Figura 2.11: Vetor V construído pelo algoritmo *FPgrowth**.

O vetor V é populado juntamente com a construção dos ramos da *FP-tree*. A cada leitura de uma transação t da base de dados, o algoritmo *FPgrowth** inclui a transação t na *FP-tree*, seguindo os passos apresentados na Subseção 2.3.1, e, simultaneamente, soma 1 às entradas correspondentes aos conjuntos de itens de tamanho 2 presentes em t. Por exemplo, a primeira transação da base de dados considerada ($\{a, b, c, d, g\}$) é incluída em forma de um único ramo na *FP-tree* e, ao mesmo tempo, soma-se 1 às entradas $V[a, b], V[a, c], V[a, d], V[b, c], V[b, d]$ e $V[c, d]$. No exemplo considerado, a *FP-tree* e a tabela de cabeçalhos construídas serão as mesmas da Figura 2.6 e o vetor V correspondente será o ilustrado na Figura 2.11.

2.5.2 Extração de Conjuntos Freqüentes com o Auxílio da Estrutura de Dados V

Após a construção da *FP-tree* e do vetor V, o algoritmo *FPgrowth** inicia a extração dos conjuntos freqüentes a partir do último item da tabela de cabeçalhos inicial, assim como no algoritmo *FP-growth*.

Seguindo os ponteiros dos nós de nome e, partindo da tabela de cabeçalhos,

o algoritmo obtém o conjunto freqüente $\{e : 3\}$. O passo seguinte é encontrar os conjuntos freqüentes de tamanho 2 que têm o sufixo e. Observe que, com a geração do vetor V, o primeiro percurso na *FP-tree* corrente, que tem como objetivo obter os itens freqüentes de BD_e, torna-se desnecessário já que tais itens podem ser obtidos diretamente deste vetor. Desta forma, o algoritmo *FPgrowth** cria a tabela de cabeçalhos de FP_e a partir das informações do vetor V e percorre a *FP-tree* corrente apenas para identificar BD_e e construir a FP_e correspondente.

Assim como na construção da *FP-tree* inicial, durante a construção da FP_e, o algoritmo *FPgrowth** constrói um novo vetor V, V_e, que conterá o suporte dos conjuntos de itens de tamanho 2 presentes na BD_e. O vetor V_e é apresentado na Figura 2.12.

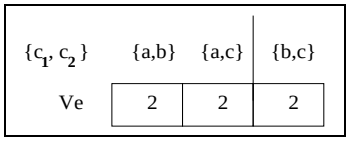


Figura 2.12: Vetor V_e construído pelo algoritmo *FPgrowth**.

O algoritmo *FPgrowth** repete os mesmos passos, recursivamente, até que todos os itens da tabela de cabeçalhos inicial tenham sido processados.

2.5.3 Algumas Otimizações

Além da proposta da geração do vetor V, algumas otimizações foram sugeridas pelos autores do algoritmo *FPgrowth**, baseadas nas características da base de dados que está sendo considerada durante a execução.

Conforme observado pelos autores em [10], *FP-trees* geradas a partir de bases de dados densas são mais compactas e, conseqüentemente, os percursos na estrutura são realizados mais rapidamente, o que torna desnecessária a geração do vetor V. Mesmo para bases esparsas, os autores observaram que os percursos realizados nas *FP-trees*, quando os primeiros itens na tabela de cabeçalhos estão sendo processados, também aparentam ser realizados mais rapidamente já que existem poucos

nós envolvidos. Desta forma, visando reduzir o custo computacional de popular o vetor V, os autores propõem que as entradas de V referentes aos primeiros itens na tabela de cabeçalhos não sejam populadas quando a base de dados considerada é esparsa.

Para determinar se a base de dados considerada é densa ou esparsa, a cada criação de uma nova *FP-tree*, o número de nós presentes em cada nível da estrutura é contabilizado. De acordo com experimentos realizados pelos autores, foi verificado que se o $\frac{1}{4}$ superior da *FP-tree* tiver menos que 15% do número total de nós, a base de dados pode ser considerada densa e, conseqüentemente, não é necessário que o vetor V seja gerado. Caso contrário, a base de dados é considerada esparsa e as entradas de V referentes aos primeiros itens na tabela de cabeçalhos (os 5 primeiros itens, como sugerido pelos autores) não são populadas.

2.6 Algoritmo *LCMfreq*

O algoritmo *LCMfreq* [28] para extração de conjuntos freqüentes é uma variação do algoritmo *LCM* (*Linear time Closed item set Miner*) [28] para extração de *closed itemsets* freqüentes. Um conjunto de itens freqüente I é um *closed itemset* freqüente caso esteja presente em pelo menos *supmin%* das transações da base de dados e não exista qualquer superconjunto de I presente nas mesmas transações em que I ocorre. Como exemplo, considere novamente a base de dados da Tabela 2.1 e o suporte mínimo igual a duas transações. O conjunto de itens freqüente $I = \{a, b, c\}$ é um *closed itemset* freqüente já que está presente em mais de duas transações e nenhum superconjunto de I está presente nas mesmas transações em que I ocorre.

O algoritmo *LCMfreq* armazena a base de dados completa em uma estrutura de dados, chamada G, a partir da qual os conjuntos freqüentes são extraídos. A seguir, a Seção 2.6.1 ilustra a estrutura G e os passos para a sua construção. A Subseção 2.6.2 é dedicada à apresentação do algoritmo *LCMfreq* através de um exemplo ilustrativo e a Subseção 2.6.3 descreve algumas otimizações adotadas pelo algoritmo.

2.6.1 A Estrutura de Dados G

Cada linha i da estrutura de dados G representa uma transação da base de dados e cada entrada $G[i]$ aponta para a lista de itens presentes nesta transação. As transações são ordenadas crescentemente pelo seu tamanho, ou seja, pela quantidade de itens que possui. Por exemplo, considere a estrutura de dados G da Figura 2.13, construída a partir da base de dados da Tabela 2.1.

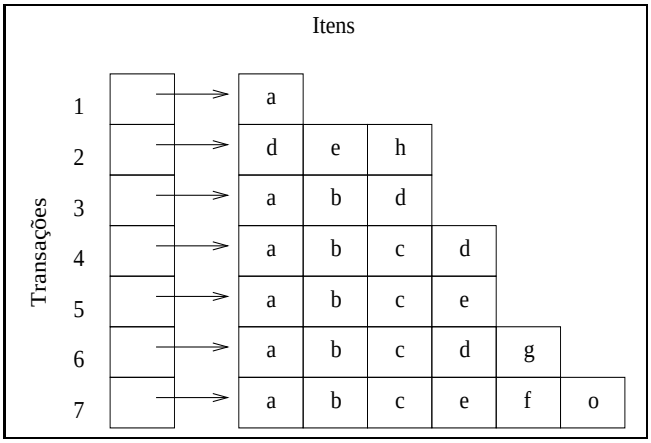


Figura 2.13: Estrutura de dados G.

Observe que a transação de $TID = 7$, na base de dados da Tabela 2.1, é a transação que possui menos itens (apenas o item a). Desta forma, ela é representada pela linha $i = 1$ de G. Já a transação de $TID = 2$, possui o maior número de itens. Por conseqüência, é representada pela última linha de G. A ordem das transações de mesmo tamanho é arbitrária.

Os itens da base de dados são ordenados crescentemente pelo valor de seu suporte e um novo identificador é associado aos itens ordenados. Por exemplo, considere a lista L dos itens presentes na base de dados da Tabela 2.1, ordenada crescentemente pelo valor de suporte: $L = \{h, g, f, o, e, d, c, b, a\}$. O item h terá o novo identificador 1, por ser o primeiro item de L. Já o item g terá o identificador 2 por ser o segundo de L e assim sucessivamente para todos os itens da lista L. Desta forma, a estrutura de dados G utilizada pelo algoritmo *LCMfreq* é ilustrada na Figura 2.14. Observe que os identificadores presentes em cada transação (em cada linha i de G), são ordenados crescentemente.

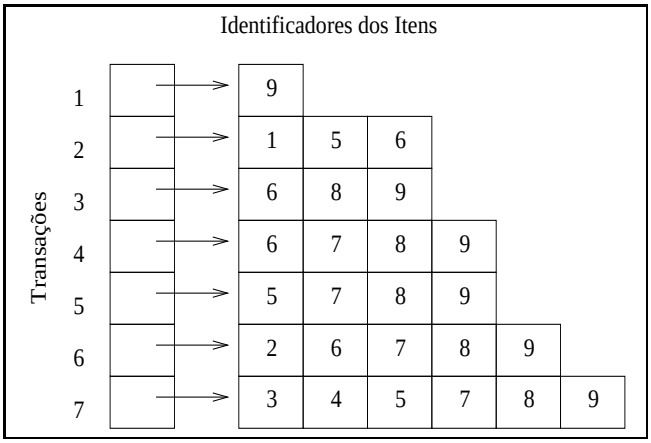


Figura 2.14: Estrutura de dados G após renomeação.

Após a construção de G o algoritmo inicia a extração dos conjuntos freqüentes a partir desta estrutura, em memória principal.

2.6.2 Extração de Conjuntos Freqüentes a partir da Estrutura de Dados G

No algoritmo *LCMfreq*, os conjuntos freqüentes são extraídos a partir da estrutura de dados G, considerando um item (identificador) de cada vez e gerando, recursivamente, todos os conjuntos freqüentes que contêm aquele item como prefixo. Os itens são processados em ordem crescente.

Considere a estrutura G ilustrada na Figura 2.14 e o suporte mínimo igual a duas transações. Primeiramente, o algoritmo armazena em um vetor, aqui chamado Q, as transações em G (identificadas pelas linhas de G) nas quais o item 1 está presente (*occurence set*). Observe que o item 1 aparece em apenas uma transação em G (linha = 2). Desta forma, o vetor Q associado ao item 1 terá apenas uma única entrada. A Figura 2.15 ilustra o vetor Q criado no início do processamento do item 1.

Em seguida, o algoritmo *LCMfreq* verifica se o item 1 tem suporte maior ou igual ao suporte mínimo, ou seja, se o número de entradas do vetor Q recém gerado é maior ou igual ao suporte mínimo. Como esta condição não se verifica, o item 1 é

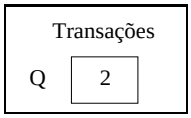


Figura 2.15: Vetor Q construído no início do processamento do item 1.

desconsiderado e o algoritmo inicia o processamento do próximo item. Observe que o mesmo ocorrerá com os itens 2, 3 e 4, pois aparecem em apenas uma transação.

Já o item 5 está presente em três transações. Desta forma, o vetor Q correspondente terá três entradas, conforme ilustrado na Figura 2.16.

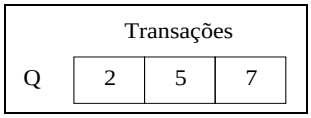


Figura 2.16: Vetor Q construído no início do processamento do item 5.

Como o número de entradas de Q é maior que o suporte mínimo, o conjunto $\{5:3\}$ ($\{e:3\}$) é gerado e, a partir deste momento, o objetivo do algoritmo é encontrar os conjuntos freqüentes de tamanho 2 que têm o item 5 como prefixo (vale observar que os conjuntos são extraídos na ordem lexicográfica dos novos identificadores). Com este objetivo, o algoritmo *LCMfreq* utiliza o vetor J, que armazena, em ordem decrescente, os itens $i > 5$ que aparecem juntamente com o item 5 em pelo menos duas transações (suporte mínimo). Para construir o vetor J, são utilizadas as estruturas Q e G.

Além de J, o algoritmo *LCMfreq* utiliza uma outra estrutura de dados, chamada JQ, cujo número de linhas é igual ao número de itens distintos na base de dados. Para todo item i em J, são armazenadas na linha i de JQ a lista dos identificadores das transações em Q, nas quais i está presente juntamente com o item 5. A Figura 2.17 ilustra as estruturas de dados J e JQ criadas pelo algoritmo.

Note que a estrutura JQ e o vetor J contêm todas as informações necessárias para a extração dos conjuntos freqüentes de tamanho 2 que contêm o item 5 como prefixo. Cada entrada de J contém um item $i > 5$ que, juntamente com o item 5, tem suporte mínimo na base de dados e para todo i em J, a linha i de JQ contêm

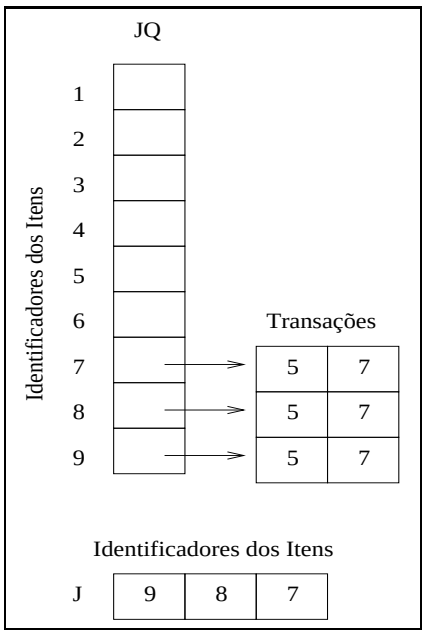


Figura 2.17: Estruturas de dados J e JQ populadas durante o processamento do item 5.

os identificadores das transações em que i está presente juntamente com o item 5, ou seja, contém o suporte mínimo do conjunto $\{5, i\}$.

Após construir as novas estruturas de dados, o algoritmo repete o processamento anterior tendo como base de dados, a estrutura JQ e como itens a serem processados, os itens em J.

O próximo conjunto freqüente a ser gerado é, então, o conjunto $\{5, 9 : 2\}$ ($\{e, a : 2\}$), já que o item 9 é o primeiro item em J, e o passo seguinte é encontrar os conjuntos freqüentes de tamanho 3 com o prefixo $\{5, 9\}$. Um novo vetor J é construído, contendo os itens $i > 9$ no vetor J corrente que apareçam juntamente com o item 9 em pelo menos duas transações em JQ. Observe que os itens presentes no vetor J corrente são todos menores que 9. Assim, o processamento do item 9 é finalizado e o algoritmo inicia o processamento do item 8, próximo item no vetor J corrente.

Da mesma forma, o algoritmo gera o conjunto freqüente $\{5, 8 : 2\}$ ($\{e, b : 2\}$) e um novo vetor J com os itens $i > 8$ no vetor J corrente que apareçam juntamente com item 8 em pelo menos duas transações em JQ. Note que apenas o item 9 atende

a estas condições. Por este motivo, o novo vetor J terá apenas uma única entrada. Além da construção de um novo vetor J, a entrada 9 de JQ é atualizada e passa a conter as transações em que o item 9 aparece juntamente com o item 8. No exemplo considerado, JQ permanecerá inalterada.

Em seguida, o algoritmo *LCMfreq* gera o conjunto $\{5, 8, 9 : 2\}$ ($\{e, b, a : 2\}$) e o processamento do item 8 é concluído, já que o item 9 é o único item no novo vetor J considerado.

A partir deste momento, o algoritmo inicia o processamento do item 7, último item do vetor J anterior (Figura 2.17), a fim de encontrar os conjuntos freqüentes de tamanho 3 com o sufixo $\{5, 7\}$. Os passos anteriormente descritos são repetidos, recursivamente, até que todos os itens freqüentes em G tenham sido processados.

2.6.3 Algumas otimizações

A fim de reduzir o número de ativações recursivas necessárias para a execução do algoritmo, a seguinte otimização é aplicada. Considere o exemplo ilustrativo da seção anterior. Antes da geração do conjunto freqüente $\{5, 8\}$ e do novo vetor J, o algoritmo *LCMfreq* identifica se algum dos itens $i > 8$, no vetor J corrente, está presente nas mesmas transações em que o item 8 está presente, em JQ. No exemplo considerado, o item 9 atende a esta condição. Desta forma, após o algoritmo identificar tal situação, a geração do conjunto $\{5, 8, 9\}$ é feita simultaneamente à geração do conjunto $\{5, 8\}$, não havendo a necessidade de uma nova ativação recursiva.

Outra otimização aplicada ao algoritmo *LCMfreq* é a utilização da técnica de *Diffsets*, proposta em [31]. De forma geral, quando o algoritmo identifica que os itens armazenados no vetor J estão presentes na base de dados juntamente com o item i, processado no momento, em um número de transações muito próximo ao número de transações em que i está presente, a estrutura JQ passa a armazenar, para cada item j em J, somente as transações em que j não aparece juntamente com o item i na base de dados. Desta forma, o suporte do conjunto $\{i, j\}$, por exemplo,

	Suportes altos		Suportes baixos	
Base de dados	1^o colocado	2^o colocado	1^o colocado	2^o colocado
Accidents	*kDCI++*	*Eclat*(Zaki)	*FPgrowth**	*PatriciaMine*
BMS-Webview1	*PatriciaMine*	*LCMfreq*	—	—
BMS-Webview2	*PatriciaMine*	*LCMfreq*	*LCMfreq*	*PatriciaMine*
BMSPOS	*kDCI++*	*PatriciaMine*	*FPgrowth**	*PatriciaMine*
Chess	*PatriciaMine*	*kDCI++*	*LCMfreq*	*PatriciaMine*
Connect	*kDCI++*	*aim*	*LCMfreq*	*kDCI++*
Kosarak	*kDCI++*	*PatriciaMine*	*PatriciaMine*	*afopt*
Mushroom	*kDCI++*	*LCMfreq*	*LCMfreq*	*kDCI++*
Pumsb	*PatriciaMine*	*FPgrowth**	*mafia*	*LCMfreq*
Pumsb*	*kDCI++*	*aim/PatriciaMine*	*PatriciaMine*	*kDCI++*
Retail	*PatriciaMine*	*afopt*	*LCMfreq*	*PatriciaMine/afopt*
T10I5N1KP5KC0.25D200k	*PatriciaMine*	*FPgrowth**	*FPgrowth**	*PatriciaMine*
T20I10N1KP5KC0.25D200k	*kDCI++*	*Apriori* (Borgelt)	*FPgrowth**	*LCMfreq*
T30I15N1KP5KC0.25D200k	*kDCI++*	*Eclat (Zaki)/Apriori (Borgelt)*	*Apriori* (Borgelt)	*FPgrowth**

Tabela 2.2: Resultados dos experimentos do *Workshop FIMI'03*.

será obtido subtraindo-se o suporte de i do tamanho da lista armazenada na linha j de JQ.

2.7 Análise de Desempenho

Com o objetivo de avaliar implementações de estratégias para extração de conjuntos freqüentes, considerando diferentes tipos de base de dados e diferentes valores de suporte mínimo, Goethals e Zaki organizaram, recentemente, um *Workshop* de implementações de algoritmos para extração de conjuntos freqüentes (*IEEE/ICDM Workshop on Frequent Itemset Mining Implementation - FIMI'03*). Gráficos de avaliação de desempenho em relação ao tempo total de execução obtido pelas estratégias sobre diferentes combinações de bases de dados e suportes mínimos estão disponíveis na página do *Workshop FIMI'03*, em http://fimi.cs.helsinki.fi/ (*FIMI repository*), e em [9].

A Tabela 2.2, retirada de [9], apresenta os algoritmos que obtiveram os dois melhores desempenhos em relação ao tempo total de execução, para valores de suporte mínimo altos e baixos, em catorze bases de dados selecionadas.[1]

De acordo com os resultados obtidos, os organizadores do *Workshop FIMI'03*

[1]Para a base de dados *BMS-Webview1*, os algoritmos que obtiveram os melhores desempenhos para valores de suporte mínimo baixos não foram indicados.

concluíram que os algoritmos *kDCI++* e *PatriciaMine* apresentaram os melhores desempenhos quando suportes altos foram avaliados. Para os suportes mais baixos, embora o quadro não tenha ficado tão evidente, os algoritmos que se destacaram foram *FPgrowth** e *LCMfreq*, enquanto que os algoritmos *kDCI++* e *PatriciaMine* obtiveram a segunda colocação. Desta forma, os algoritmos *kDCI++* e *PatriciaMine* foram considerados, por Goethals e Zaki, os principais algoritmos para extração de conjuntos freqüentes da atualidade.

Para os experimentos do *Workshop FIMI'03*, foram consideradas bases de dados sintéticas, criadas pelo gerador de bases da IBM *Almaden*[2], e bases de dados reais. Conforme descritas em [4], as bases de dados sintéticas simulam bases de dados de transações de compras reais (*market basket*) e são geradas a partir dos parâmetros $Tx.Iy.Nz.Pw.Cv.Du$, onde x corresponde ao tamanho médio das transações, y corresponde ao tamanho médio dos conjuntos de itens maximais potencialmente freqüentes[3] (*maximal potentially large itemsets*), z corresponde ao número de itens distintos, w corresponde ao número de conjuntos de itens maximais potencialmente freqüentes, v corresponde ao nível de correlação entre os conjuntos de itens maximais potencialmente freqüentes[4] (*correlation level*) e u, ao número de transações da base de dados.

As bases de dados de transações de compras são, usualmente, classificadas como bases de dados esparsas [13, 24, 30], as quais são caracterizadas por conterem, especialmente, conjuntos de itens freqüentes curtos [13, 30] e de suportes baixos [21]. Entretanto, conjuntos freqüentes longos podem ser obtidos com a utilização de suportes mínimos significativamente baixos [13]. Já as bases chamadas densas são caracterizadas por conterem muitos conjuntos freqüentes longos, mesmo para valores de suporte mínimo altos [13, 14, 21, 30].

[2]Disponível em http://www.almaden.ibm.com/cs/quest//syndata.html#assocSynData e apresentado em [4].

[3]Pode-se definir conjunto de itens maximal freqüente como sendo um conjunto de itens freqüente que não possui superconjuntos também freqüentes.

[4]O nível de correlação é definido em [4] como sendo a fração de itens em comum nos conjuntos maximais potencialmente freqüentes.

Como exemplo, as bases de dados reais *BMSPOS*, *BMS-Webview1*, *BMS-Webview2* e *Retail* [6], utilizadas nos experimentos do *Workshop FIMI'03*, são bases reais esparsas [6, 32], enquanto que as bases de dados reais *Accidents*, *Chess*, *Connect*, *Mushroom*, *Pumsb** e *Pumsb*, também utilizadas nos experimentos, são consideradas densas [8, 10, 13, 14, 16, 17, 21, 22, 30].

Apesar das bases de dados sintéticas serem, usualmente, classificadas como esparsas [1, 22, 30], para as bases de dados *T10I5N1KP5KC0.25D200k*, *T20I10N1KP5KC0.25D200k* e *T30I15N1KP5KC0.25D200k*, que se encontram na Tabela 2.2, e para as bases de dados *T20I10N1KP5KC0.25D400k*, *T20I10N1KP5KC0.25D800k* e *T20I10N1KP5KC0.25D1.6M*, cujos experimentos estão disponíveis na página do *Workshop FIMI'03*, não foram encontradas, na literatura, indicações claras de suas densidades. O mesmo ocorreu para a base de dados real *Kosarak*. Embora os valores de suporte mínimo considerados em [5, 9, 28] sejam relativamente baixos, sua densidade também não é especificada, claramente, nestes trabalhos.

Por esta razão, a fim de se obter as densidades de todas as bases utilizadas nos experimentos e, conseqüentemente, entender mais profundamente os comportamentos dos algoritmos avaliados, foram realizados experimentos com as bases de dados cujas densidades não foram identificadas, levando-se em consideração as características de uma base esparsa e densa, descritas anteriormente. As bases sintéticas foram criadas com a utilização do gerador da IBM *Almaden*, enquanto que a base real *Kosarak* está disponível na página do *Workshop FIMI'03*.

Para seis valores de suporte mínimo 10%, 5%, 1%, 0,75%, 0,5% e 0,25% foi encontrado, para cada base de dados de densidade desconhecida, o tamanho do maior conjunto freqüente gerado, utilizando o código do algoritmo *kDCI++* disponível no página do algoritmo *DCI*, em *http://miles.cnuce.cnr.it/~palmeri/datam/ DCI* (este mesmo código foi disponibilizado no site do *Workshop FIMI'03*, após a publicação dos resultados). A Tabela 2.3 apresenta o resultado dos experimentos.

Note que, mesmo considerando valores de suporte mínimo baixos, como 1% e 0,75%, os conjuntos freqüentes encontrados são significativamente pequenos,

Base de dados	10%	5%	1%	0,75%	0,5 %	0,25%
T20I10N1KP5KC0.25D200K	1	1	2	2	9	16
T20I10N1KP5KC0.25D400K	1	1	2	2	9	15
T20I10N1KP5KC0.25D800K	1	1	2	2	9	15
T20I10N1KP5KC0.25D1.6M	1	1	2	2	9	19
T10I5N1KP5KC0.25D200K	0	1	1	2	4	9
T30I15N1KP5KC0.25D200K	1	1	3	3	15	20
Kosarak	3	4	5	6	6	10

Tabela 2.3: Tamanho do maior conjunto freqüente encontrado nas bases de dados de densidade não identificada, considerando-se seis valores de suporte mínimo.

o que, de acordo com a descrição apresentada anteriormente, caracteriza uma base de dados esparsa. O tamanho dos conjuntos freqüentes aumenta a partir do suporte 0,5%, valor este significativamente baixo.

A Tabela 2.4, a seguir, apresenta os resultados do mesmo experimento, quando realizados para as bases de dados reais consideradas esparsas. Observe que o comportamento é bastante semelhante ao das bases de dados avaliadas anteriormente.

Base de dados	10%	5%	1%	0,75%	0,5 %	0,25%
BMSPOS	2	3	5	5	6	7
BMS-Webview1	0	1	2	2	3	4
BMS-Webview2	0	0	3	4	5	7
Retail	2	3	4	4	5	5

Tabela 2.4: Tamanho do maior conjunto freqüente encontrado nas bases de dados esparsas, considerando-se seis valores de suporte mínimo.

De posse da identificação da densidade de todas as bases de dados utilizadas nos experimentos do *Workshop FIMI'03*, tornou-se possível analisar, mais detalhadamente, o desempenho do algoritmo *kDCI++*, avaliado como um dos principais algoritmos para a extração de conjuntos freqüentes e objeto de estudo do presente trabalho.

Em [9] e na página do *Workshop FIMI'03*, para cada base de dados utilizada nos experimentos, é apresentado um gráfico com os tempos de execução (em segundos) alcançados pelos algoritmos avaliados, considerando-se diferentes valores de suporte mínimo. Pode-se observar, nestes gráficos, que o algoritmo *kDCI++* apresenta um bom desempenho quando suportes altos são avaliados. Entretanto, especialmente para as bases de dados esparsas, o algoritmo *kDCI++* apresenta uma acentuada queda de desempenho quando os valores mais baixos de suporte mínimo são considerados. Este comportamento pode ser observado, principalmente, nas bases de dados esparsas reais *BMSPOS*, *Retail* e *Kosarak* e para as bases esparsas sintéticas *T20I10N1KP5KC0.25D200k* e *T30I15N1KP5KC0.25D200k*.

No escopo desta dissertação, a fim de melhor compreender o comportamento do algoritmo *kDCI++* considerando tal cenário, foram realizados vários experimentos sobre diferentes combinações de bases de dados esparsas e valores de suporte mínimo também avaliados no *Workshop FIMI'03* [9] e em [5, 9, 10, 12, 13, 16, 17, 18, 22, 24, 28, 30, 32] [5]. A partir destes experimentos, observou-se que, em média, a terceira iteração apresenta um custo computacional bastante elevado se comparado à média do custo computacional das demais iterações.

A Tabela 2.5 apresenta, para diferentes combinações de bases de dados e valores de suporte mínimo, o tempo total de execução (em segundos) do algoritmo *kDCI++* e o respectivo tempo de execução da terceira iteração (em segundos). A última coluna representa a relação entre o tempo de execução da terceira iteração e o tempo total de execução.

Nestas execuções, a terceira iteração representou, em média, 59% do tempo total das execuções do algoritmo *kDCI++*, variando de 19% a 87%, com exceção da base de dados real *Kosarak* e suporte mínimo 0,1%, cuja terceira iteração é apenas 5% do tempo total de execução (a razão de tal comportamento será discutida no Capítulo 4). O alto custo computacional da terceira iteração pode ser explicado pela grande quantidade de candidatos de tamanho 3 a ser avaliada pelo algoritmo. Conforme observado em [11, 16, 17], a terceira iteração pode ser bastante custosa

[5] Em alguns destes trabalhos, estas bases de dados são, claramente, apresentadas como esparsas.

computacionalmente para algoritmos da classe apriori. Em [11], os autores estudam a escalabilidade dos algoritmos propostos avaliando apenas o custo computacional exigido na obtenção dos conjuntos freqüentes de tamanho 3, que para as bases de dados sintéticas e suportes mínimos considerados, representou 55% do tempo total de suas execuções. Já em [16] e [17], para a base de dados *T25I10D10K* e suporte mínimo 0,1%, os autores atribuem o mau desempenho dos algoritmos ao tamanho do conjunto C_3, que é significativamente maior que o conjunto F_3 encontrado.

Desta forma, com o objetivo de aprimorar o desempenho do algoritmo *kDCI++* e torná-lo também eficiente sobre bases de dados esparsas quando valores de suporte baixos são considerados, neste trabalho propõe-se uma adaptação deste algoritmo, denominada *kDCI-3*, em que a estratégia realizada durante a terceira iteração é substituída por outra que possibilita uma contagem mais eficiente do suporte dos conjuntos candidatos de tamanho 3.

Esta estratégia é baseada em uma estrutura de dados apresentada em [26], através da qual a contagem dos candidatos de tamanho 3 torna-se mais eficiente. Em [26], esta estrutura de dados foi utilizada no algoritmo *Apriori* e, segundo os resultados obtidos, aprimorou-o, significativamente.

Base de dados (*supmin* - %)	Tempo Total	Tempo 3^a iter.	%
T20I10N1KP5KC0.25D200K (0,1)	526	339	64
T20I10N1KP5KC0.25D200K (0,25)	77	56	72
T20I10N1KP5KC0.25D400K (0,1)	1.109	724	65
T20I10N1KP5KC0.25D400K (0,25)	167	127	76
T20I10N1KP5KC0.25D800K (0,1)	1000	341	34
T20I10N1KP5KC0.25D800K (0,25)	380	283	74
T20I10N1KP5KC0.25D1.6MK (0,1)	3858	733	19
T20I10N1KP5KC0.25D1.6MK (0,25)	393	186	47
T10I5N1KP5KC0.25D200K (0,01)	297	222	75
T10I5N1KP5KC0.25D200K (0,025)	123	88	71
T30I15N1KP5KC0.25D200K (0,25)	647	538	83
T30I15N1KP5KC0.25D200K (0,5)	132	115	87
T25I10D10K (0,05)	161	72	45
T25I10D10K (0,1)	24	20	83
T40I10D100K (0,5)	375	264	70
T40I10D100K (0,75)	128	100	78
T10I4D100K (0,01)	307	103	34
T10I4D100K (0,02)	74	52	70
T30I16D400K (0,4)	751	472	63
T30I16D400K (0,6)	218	156	71
BMSPOS (0,1)	415	154	37
BMSPOS (0,25)	62	31	50
Kosarak (0,1)	509	25	5
Kosarak (0,25)	51	14	27
Retail (0,01)	84	58	69
Retail (0,025)	23	15	65

Tabela 2.5: Tempo total de execução e tempo de execução da terceira iteração do algoritmo *kDCI++* para diferentes combinações de bases de dados e valores de suporte mínimo.

Capítulo 3

Algoritmo *kDCI-3*

Experimentos computacionais realizados neste trabalho mostraram que, quando valores baixos de suporte mínimo são considerados sobre bases de dados esparsas, a terceira iteração do algoritmo *kDCI++* apresenta um alto custo computacional.

Neste contexto, este capítulo tem como objetivo apresentar o algoritmo *kDCI-3*, extensão do algoritmo *kDCI++*, em que a estratégia adotada durante a terceira iteração possibilita uma contagem mais eficiente dos conjuntos candidatos de tamanho 3. Esta estratégia permite a contagem direta dos conjuntos candidatos a partir da utilização de uma estrutura de dados apresentada em [26].

Além da utilização desta nova estratégia, foi desenvolvido para o algoritmo *kDCI-3* uma técnica de gerenciamento de memória que possibilita a contagem de blocos de candidatos isoladamente, de modo que a quantidade de memória necessária para a contagem de cada bloco gerado não ultrapasse a quantidade de memória disponível.

A Seção 3.1 é dedicada à apresentação da estrutura de dados adotada. A Seção 3.2 descreve como esta estrutura é utilizada na contagem dos candidatos de tamanho 3 e discute as principais características do procedimento adotado pelo al-

goritmo *kDCI-3*, durante a terceira iteração. Na Seção 3.3, é apresentada a técnica de gerenciamento de memória e, finalmente, na Seção 3.4, são feitas algumas considerações sobre a implementação do algoritmo *kDCI-3*.

3.1 A Estrutura de Dados Utilizada

Com o objetivo de possibilitar a contagem direta dos conjuntos candidatos de tamanho 3, o algoritmo *kDCI-3* utiliza uma estrutura de dados [26] baseada em uma tabela de prefixos, chamada T_3', e um vetor C'. A Figura 3.1 ilustra as estruturas de dados T_3' e C' construídas a partir da base de dados da Tabela 2.1 e suporte mínimo igual a duas transações.

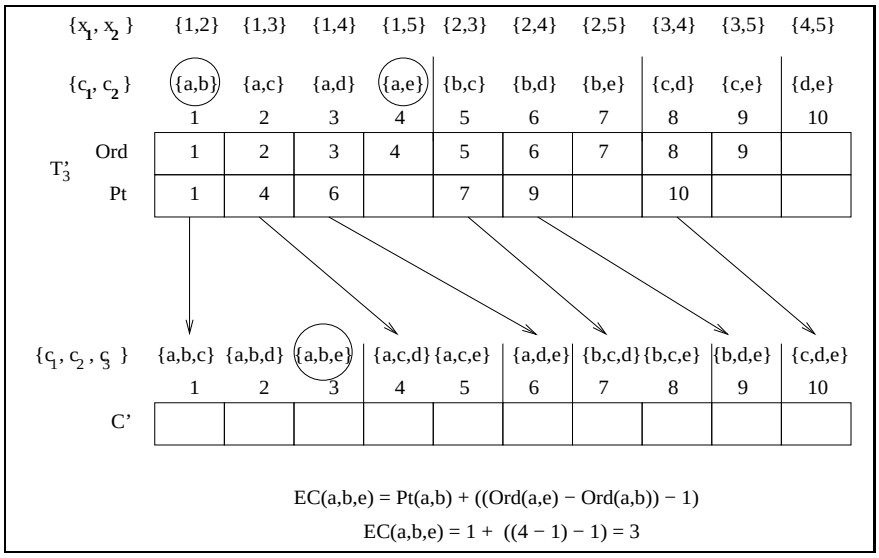


Figura 3.1: As estruturas de dados T_3' e C' construídas pelo algoritmo *kDCI-3*.

O vetor C', assim como o vetor C utilizado pelo algoritmo *kDCI++*, representa os conjuntos candidatos de tamanho 3, em ordem lexicográfica. Os candidatos são gerados combinando-se os pares de conjuntos freqüentes de tamanho 2, de F_2, que possuem o mesmo prefixo de tamanho 1. Na base de dados considerada, o conjunto F_2 encontrado é $\{\{a, b : 5\}, \{a, c : 4\}, \{a, d : 3\}, \{a, e : 2\}, \{b, c : 4\}, \{b, d : 3\}, \{b, e : 2\}, \{c, d : 2\}, \{c, e : 2\}\}$. Desta forma, a combinação dos conjuntos freqüentes $\{a, b\}$ e $\{a, e\}$, por exemplo, gera o conjunto candidato $\{a, b, e\}$ já que possuem

o mesmo prefixo a. Observe que $\{a, b, e\}$ é representado pela terceira entrada de C'.

A estrutura de dados T_3' é uma tabela de prefixos onde cada entrada representa um possível prefixo de tamanho 2. Entretanto, diferentemente do algoritmo *kDCI++*, suas entradas são compostas por dois campos. O primeiro campo armazena a ordem, representada por $Ord(c_1, c_2)$, em que o conjunto freqüente de tamanho 2 $\{c_1, c_2\}$ é localizado em T_3'. Se $\{c_1, c_2\}$ não é um conjunto freqüente, este valor é nulo. No exemplo considerado, como o conjunto freqüente $\{a, b\}$ é o primeiro conjunto freqüente representado em T_3', $Ord(a, b)$ tem valor 1. Observe que $Ord(d, e)$ é nulo já que $\{d, e\}$ não é um conjunto freqüente. Considerando-se a ordem lexicográfica dos conjuntos candidatos em C', o segundo campo de T_3', representado por $Pt(c_1, c_2)$, aponta para a primeira entrada do vetor C' que representa um conjunto candidato com o prefixo $\{c_1, c_2\}$. Este campo é nulo caso $\{c_1, c_2\}$ não tenha gerado nenhum candidato com este mesmo prefixo. Na Figura 3.1, $Pt(a, b)$ tem valor 1, indicando que a entrada 1 de C' representa o primeiro conjunto candidato com prefixo $\{a, b\}$ (candidato $\{a, b, c\}$). Da mesma forma, $Pt(a, e)$ é nulo pois $\{a, e\}$ não contribuiu na geração de nenhum candidato com este prefixo.

3.2 Contagem do Suporte dos Conjuntos Candidatos a partir da Estrutura de Dados T_3'

O suporte dos conjuntos candidatos é incrementado no vetor C' à medida que a base de dados é lida. Para a contagem do suporte de um determinado conjunto candidato $c = \{c_1, c_2, c_3\}$, é necessário, então, que o algoritmo encontre corretamente a entrada em C' que representa o conjunto candidato c. Como exemplo, considere o conjunto candidato $\{a, b, e\}$, em C'.

Lembrando que os candidatos em C' são gerados combinando-se os pares de conjuntos freqüentes de tamanho 2 que possuem o mesmo prefixo de tamanho 1, o candidato $\{a, b, e\}$ é gerado, então, a partir da combinação dos conjuntos freqüentes $\{a, b\}$ e $\{a, e\}$. Além desta combinação, observe que $\{a, b\}$ também é combinado com

os conjuntos freqüentes $\{a, c\}$ e $\{a, d\}$, já que $\{a, c\}$ e $\{a, d\}$ também são conjuntos freqüentes com o mesmo prefixo a. Estas duas combinações geram dois novos conjuntos candidatos $\{a, b, c\}$ e $\{a, b, d\}$ igualmente representados em C'. Desta forma, como $\{a, e\}$ é o terceiro conjunto freqüente após $\{a, b\}$ e considerando-se que os conjuntos candidatos estão representados em C' em ordem lexicográfica, é fácil concluir que $\{a, b, e\}$ será o terceiro candidato com o prefixo $\{a, b\}$ em C', o que significa que estará posicionado duas entradas após a entrada do primeiro candidato de mesmo prefixo. Na realidade, este deslocamento de duas entradas indica que existem dois conjuntos freqüentes entre $\{a, b\}$ e $\{a, e\}$ em T_3'. Considerando-se que o campo Ord representa a ordem em que os conjuntos freqüentes estão localizados em T_3', observe que tal deslocamento pode ser calculado pela fórmula:

$$(Ord(a, e) - Ord(a, b)) - 1 \ . \tag{3.1}$$

Assim sendo, tendo em vista que o primeiro candidato com o prefixo $\{a, b\}$ em C' é representado por $Pt(a, b)$, a entrada do conjunto candidato $\{a, b, e\}$ em C' (representada por $EC'(a, b, e)$) é dada pela Equação 3.2, conforme ilustrado na Figura 3.1.

$$\begin{aligned} EC'(a, b, e) &= Pt(a, b) + ((Ord(a, e) - Ord(a, b)) - 1) \\ &= 1 + ((4 - 1) - 1) = 3 \ . \end{aligned} \tag{3.2}$$

A entrada correspondente a um conjunto candidato genérico $c = \{c_1, c_2, c_3\}$ em C' é, então, definida pela Equação 3.3.

$$EC'(c_1, c_2, c_3) = Pt(c_1, c_2) + ((Ord(c_1, c_3) - Ord(c_1, c_2)) - 1) \ . \tag{3.3}$$

De posse da Equação 3.3, o algoritmo *kDCI-3* obtém o suporte dos conjuntos candidatos de tamanho 3 da seguinte forma. Para cada conjunto de itens de tamanho 3 $c = \{c_1, c_2, c_3\}$ presente em cada uma das transações t da base de dados corrente, o algoritmo *kDCI-3* encontra, primeiramente, a entrada de T_3' que corresponde ao conjunto $\{c_1, c_2\}$ através da Equação 2.1. Se a entrada encontrada não representar

um conjunto freqüente de tamanho 2, ou seja, se $O(c_1, c_2)$ for nulo, o conjunto c é desconsiderado, assim como todos os demais conjuntos de itens de tamanho 3 iniciados por $\{c_1, c_2\}$ presentes em t. Caso contrário, através do mesmo procedimento descrito acima, o algoritmo verifica se $\{c_1, c_3\}$ representa um conjunto freqüente de tamanho 2. Se esta condição for verdadeira, a entrada correspondente ao conjunto c em C' é obtida através da Equação 3.3 e, então, incrementada. Caso contrário, o conjunto c é desconsiderado.

Observe que, desta forma, os conjuntos candidatos são acessados diretamente, sem que um algoritmo de busca tenha que ser executado sobre um conjunto de candidatos, como ocorre por exemplo em [4, 15, 16, 17].

Para viabilizar a utilização desta estrutura, o algoritmo *kDCI-3* não poda candidatos durante sua geração. Os conjuntos candidatos de tamanho 3 $c = \{c_1, c_2, c_3\}$ em que o subconjunto $\{c_2, c_3\}$ não é freqüente são igualmente representados no vetor C'. É o caso, por exemplo, dos conjuntos candidatos $\{a, d, e\}$, $\{b, d, e\}$ e $\{c, d, e\}$ da Figura 3.1 (embora o conjunto $\{d, e\}$ não seja freqüente, estes candidatos estão representados em C'). Tal característica é necessária pelo seguinte motivo. Considere novamente a Figura 3.1. Se o conjunto $\{b, d\}$ não fosse um conjunto freqüente, a poda de candidatos eliminaria o conjunto candidato $\{a, b, d\}$ e, conseqüentemente, este não estaria representado em C' (o mesmo aconteceria com os conjuntos $\{a, d, e\}$, $\{b, c, d\}$ e $\{b, d, e\}$). Desta forma, o conjunto candidato $\{a, b, e\}$ passaria a ser o segundo candidato em C' com o prefixo $\{a, b\}$ e não mais o terceiro, como ilustrado na figura considerada. No entanto, observe que o resultado da Equação 3.2, apresentada na seção anterior, não retrataria esta nova situação. Os valores de $Pt(a, b)$, $Ord(a, e)$ e $Ord(a, b)$ em nada seriam alterados se a poda de candidatos fosse feita, razão pela qual o resultado da Equação 3.2 permaneceria o mesmo.

Durante o desenvolvimento deste trabalho, foram feitos experimentos nos quais a poda de candidatos era feita no decorrer da fase de contagem. Nestes experimentos, o suporte do conjunto candidato $c = \{c_1, c_2, c_3\}$ só era incrementado se, além dos conjuntos $\{c_1, c_2\}$ e $\{c_1, c_3\}$, $\{c_2, c_3\}$ também fosse um conjunto freqüente, o que caracteriza a poda de candidatos. No entanto, esta poda não reduziu o custo

computacional da terceira iteração em nenhum dos experimentos realizados e, desta forma, optou-se por desconsiderá-lo.

3.3 Gerenciamento de Memória

Nos experimentos computacionais realizados com o algoritmo *FPgrowth**, observou-se que, quando a base de dados é significativamente grande para ser compactada em memória, suas execuções são longas devido ao uso de memória virtual. O mesmo ocorreu com o algoritmo *kDCI++* quando o número de candidatos a ser representado pelas estruturas de dados T_3, P, I, S e C cresceu consideravelmente. Desta forma, a fim de evitar que o algoritmo *kDCI-3* apresente o mesmo problema para um número muito grande de candidatos de tamanho 3, foi desenvolvida uma estratégia de gerenciamento de memória com a seguinte idéia.

Dependendo do número de conjuntos candidatos a serem avaliados, a memória necessária para a construção completa das estruturas T_3' e C' pode ultrapassar a memória disponível no início da terceira iteração. Por esta razão, na estratégia de gerenciamento de memória adotada, blocos de candidatos de tamanho 3 são contados separadamente e consecutivamente, de modo que a quantidade de memória necessária para a contagem de cada grupo gerado não ultrapasse a quantidade de memória disponível. No entanto, de modo a facilitar a contagem isolada de um bloco de candidatos, foi estabelecido que candidatos de mesmo prefixo de tamanho 1 farão sempre parte de um mesmo bloco de candidatos a ser avaliado. Cada bloco de candidatos avaliados poderá conter um ou mais grupos de candidatos de mesmo prefixo de tamanho 1.

Inicialmente, somente a parte de T_3' que corresponde ao primeiro grupo de candidatos de mesmo prefixo (de tamanho 1) é criada (supõe-se, sem perda de generalidade, que a memória consumida por apenas um grupo de candidatos de mesmo prefixo não ultrapassa a memória total disponível). No exemplo da Figura 3.1, o primeiro grupo de candidatos tem o prefixo a. Desta forma, as entradas $\{a, b\}$, $\{a, c\}$, $\{a, d\}$ e $\{a, e\}$ da estrutura T_3' serão as primeiras entradas criadas pelo

algoritmo. Em seguida, o algoritmo calcula o número x de candidatos de tamanho 3 gerados pela combinação dos candidatos de tamanho 2 representados pela parte de T_3' recém criada. No exemplo considerado, os candidatos de tamanho 3 serão seis ($x = 6$): $\{a, b, c\}$, $\{a, b, d\}$, $\{a, b, e\}$, $\{a, c, d\}$, $\{a, c, e\}$ e $\{a, d, e\}$. Considerando-se que a memória consumida pela parte de T_3' já criada é y e a memória consumida por x entradas do vetor C' é z, após calcular o número x de candidatos, o algoritmo verifica se $y + z$ é igual ou ultrapassa 75% da memória disponível (25% de folga) ou se $y + z +$ a quantidade de memória consumida pela próxima parte de T_3' a ser criada é igual ou ultrapassa este mesmo limite (a próxima parte de T_3' será referente ao próximo grupo de candidatos de mesmo prefixo).

A partir daí, enquanto a condição descrita acima for falsa, ou seja, enquanto houver memória disponível (considerando-se o limite preestabelecido), as entradas de T_3' referentes ao próximo grupo de candidatos de mesmo prefixo são acrescentadas às partes de T_3' já criadas. O número x' de candidatos de tamanho 2 representados pela parte de T_3' recém criada é contabilizado e o número x inicial é incrementado de x' entradas.

Quando a memória total consumida for igual ou ultrapassar o limite preestabelecido, são criadas x entradas de C' em memória e inicia-se o processo de contagem dos candidatos por elas representados (para a contagem dos candidatos, é feita uma leitura da base de dados). Ao final da contagem, os conjuntos candidatos com suporte menor que o mínimo são descartados e aqueles com suporte maior ou igual ao mínimo são armazenados em memória principal. Em seguida, a memória consumida pelas estruturas T_3' e C', referentes aos conjuntos candidatos já avaliados, é desalocada e o processo de criação de novas partes de T_3', referentes aos próximos grupos de candidatos de mesmo prefixo, é reiniciado. Este procedimento é repetido até que todos os candidatos de tamanho 3 tenham sido avaliados.

3.4 Considerações sobre a Implementação

Para que fosse possível avaliar, de forma justa, os desempenhos obtidos pelos algoritmos *kDCI-3* e *kDCI++*, o algoritmo *kDCI-3* foi implementado modificando-se o mínimo possível o código do algoritmo *kDCI++*.

O algoritmo *kDCI++* foi escrito na linguagem $C++$ e, conforme as regras de submissão do *Workshop FIMI'03*, tem como parâmetros de entrada o caminho completo de onde está localizada a base de dados (arquivo texto) e o suporte mínimo (em valor absoluto) (O código do algoritmo *kDCI++* está disponível na página do algoritmo *DCI* e, também, na página do *Workshop FIMI'03*).

Para a implementação do algoritmo *kDCI++*, os autores utilizaram conceitos da programação orientada a objetos como classes e herança. Por exemplo, as funções (também chamadas de métodos) para manipulação dos conjuntos candidatos quando a base de dados verticalizada ainda não se encontra em memória principal, pertencem a uma classe específica, assim como os métodos para a manipulação da representação verticalizada da base de dados e da própria base de dados. Desta forma, a fim de obedecer o mesmo padrão adotado e alterar o mínimo possível o código já escrito, criou-se para a implementação do algoritmo *kDCI-3* uma nova classe com métodos especialmente desenvolvidos para a geração e manipulação dos candidatos de tamanho 3. Como alguns dos métodos para manipulação de conjuntos candidatos do algoritmo *kDCI++* também se aplicavam aos candidatos manipulados pelo algoritmo *kDCI-3*, estes não foram reescritos mas sim herdados da classe utilizada pelo algoritmo *kDCI++*. Dentre os métodos herdados podemos citar, por exemplo, a função que retorna o número de candidatos gerados em cada iteração e a função que verifica se o suporte de um determinado candidato é maior que o mínimo.

Capítulo 4

Resultados Computacionais

Neste capítulo, serão apresentados os resultados computacionais obtidos pelo algoritmo *kDCI-3*, proposto nesta dissertação. Parte destes resultados foram apresentados em [23]. Os experimentos foram realizados utilizando-se uma máquina de processador *Pentium* III, 600 Mhz, com 256 *megabytes* de memória principal e sistema operacional Fedora Linux 2.4.22.

O capítulo está organizado da seguinte forma. As principais características das bases de dados, assim como os valores de suporte mínimo considerados nos experimentos, são apresentados na Seção 4.1. A Seção 4.2 é destinada à avaliação de desempenho da técnica adotada pelo algoritmo *kDCI-3*, comparando-se os resultados computacionais obtidos pelos algoritmos *kDCI-3* e *kDCI++* sobre diferentes combinações de bases de dados e valores de suporte mínimo. Na Seção 4.3, é feita uma análise comparativa entre o algoritmo *kDCI-3* e os principais algoritmos para extração de conjuntos freqüentes, segundo a avaliação do *Workshop FIMI'03*. Finalmente, na Seção 4.4 é apresentado um resumo das avaliações de desempenho realizadas neste capítulo.

4.1 Bases de Dados e Suportes Mínimos

Para os experimentos computacionais deste trabalho, utilizaram-se dezoito bases de dados esparsas cujas principais características estão descritas na Tabela 4.1. Nela, são apresentados o nome da base de dados, número de itens distintos, número de transações, tamanho médio das transações (em número de itens) e alguns dos trabalhos que já realizaram experimentos sobre a base de dados correspondente. As bases de dados *T10I5N1KP5KC0.25D200k*, *T20I10N1KP5KC0.25D200k*, *T20I10N1KP5KC0.25D400k*, *T20I5N1KP5KC0.25D800K*, *T20I10N1KP5KC0.25D-1.6M* e *T30I15N1KP5KC0.25D200k*, utilizadas nos experimentos do *Workshop FIMI'03*, foram criadas com a utilização do gerador de bases de dados da IBM *Almaden*. As bases *T20I10N1KP5KC0.25D100K*, *T20I10N1KP5KC0.25D-300K*, *T15I10N1KP5KC0.25D200K*, *T25I10N1KP5KC0.25D200K* e *T30I10N1KP-5KC0.25D200K* também foram construídas a partir do gerador da IBM e são utilizadas exclusivamente para o estudo da escalabilidade da técnica utilizada pelo algoritmo *kDCI-3*. Já as bases de dados *T40I10D100K* e *T10I4D100K* estão disponíveis na página do *Workshop FIMI'03* e as bases *T25I10D10K* e *T30I16D400K* estão disponíveis na página do algoritmo *DCI*. As demais bases são bases de dados reais e estão também disponíveis na página do *Workshop FIMI'03*.

Sobre as bases de dados *T10I5N1KP5KC0.25D200k*, *T20I10N1KP5KC0.25D-200k*, *T30I15N1KP5KC0.25D200k*, *BMSPOS*, *Retail* e *Kosarak* consideraram-se os mesmos valores de suporte mínimo avaliados no *Workshop FIMI'03*. Nos experimentos sobre as bases de dados *T20I10N1KP5KC0.25D400k*, *T20I10N1KP5KC0.25D-800K* e *T20I10N1KP5KC0.25D1.6M*, como apenas um único suporte mínimo foi avaliado no *Workshop FIMI'03* (0,25%), utilizaram-se os mesmos valores de suporte mínimo considerados sobre a base de dados *T20I10N1KP5KC0.25D200k*. Finalmente, para as bases de dados *T40I10D100K*, *T10I4D100K*, *T25I10D10K* e *T30I16D400K* foram considerados valores de suporte mínimo avaliados sobre estas bases em trabalhos relacionados.

Base de dados	Itens	Trans.	Tam. méd.	Referências
T20I10N1KP5KC0.25D100K	1.000	98.729	20,1	—
T20I10N1KP5KC0.25D200K	1.000	197.437	20,1	[9]
T20I10N1KP5KC0.25D300K	1.000	296.252	20,1	—
T20I10N1KP5KC0.25D400K	1.000	394.852	20,1	[9]
T20I10N1KP5KC0.25D800K	1.000	789.655	20,1	[9]
T20I10N1KP5KC0.25D1.6M	1.000	1.579.861	20,1	[9]
T15I10N1KP5KC0.25D200K	1.000	186.024	15,9	—
T25I10N1KP5KC0.25D200K	1.000	199.659	24,8	—
T30I10N1KP5KC0.25D200K	1.000	199.970	29,7	—
T10I5N1KP5KC0.25D200K	1.000	192.889	10,3	[9]
T30I15N1KP5KC0.25D200K	1.000	199.093	29,6	[9]
T25I10D10K	1.001	9.219	27,7	[1, 12, 16, 17]
T40I10D100K	1.000	100.000	39,6	[5, 10, 13, 22, 28, 30]
T10I4D100K	1.000	100.000	10,1	[5, 13, 22, 24, 28, 30]
T30I16D400K	1.000	397.487	29,7	[16, 17, 22]
BMSPOS	1.657	515.597	6,5	[9, 14, 22, 28, 32]
Kosarak	41.271	990.002	8,1	[5, 9, 28]
Retail	16.469	88.162	10,3	[6, 9]

Tabela 4.1: Bases de dados utilizadas nos experimentos computacionais.

4.2 Avaliação de Desempenho do *kDCI-3*

Nesta seção, são feitas avaliações de desempenho da técnica adotada pelo algoritmo *kDCI-3*, considerando-se tempo de execução, memória consumida e escalabilidade.

4.2.1 Tempo de Execução

Apenas para efeito de comparação, a Tabela 4.2 apresenta o tempo de execução da terceira iteração (em segundos) e o tempo total de execução (em segundos) obtidos, respectivamente, pelos algoritmos *kDCI++* e *kDCI-3*, para as mesmas combinações de bases de dados e valores de suporte mínimo utilizados na Seção 2.7, na qual a motivação deste trabalho é apresentada. Na Tabela 4.2, a quarta coluna representa a relação entre o tempo da terceira iteração do algoritmo *kDCI-3* e o tempo da terceira iteração do algoritmo *kDCI++*, enquanto que a sétima coluna representa a relação entre o tempo total de execução do algoritmo *kDCI-3* e o tempo total de execução do algoritmo *kDCI++*.

Os resultados computacionais mostraram que a técnica de contagem de candidatos de tamanho 3 adotada pelo algoritmo *kDCI-3* reduziu significativamente o tempo da terceira iteração consumido pelo algoritmo *kDCI++*.

A primeira linha da Tabela 4.2, por exemplo, indica que, enquanto a terceira iteração do algoritmo *kDCI++* foi executada em 339 segundos sobre a base de dados *T20I10N1KP5KC0.25D200K* e suporte mínimo 0,1 %, a terceira iteração do algoritmo *kDCI-3*, para a mesma base de dados e suporte mínimo, foi executada em 38 segundos, representando uma redução de 89%. Observe que, para esta mesma base de dados e suporte mínimo, a terceira iteração representou uma grande parte do tempo total de execução. Desta forma, com a redução do tempo da terceira iteração, o tempo total de execução consumido pelo algoritmo *kDCI-3* foi 35% do tempo total de execução do algoritmo *kDCI++*, o que representou, portanto, uma redução de 65%.

	Tempo 3^a iter.			Tempo Total		
Base de dados (*supmin* - %)	*kDCI++*	*kDCI-3*	%	*kDCI++*	*kDCI-3*	%
T20I10N1KP5KC0.25D200K (0,1)	339	38	11	526	185	35
T20I10N1KP5KC0.25D200K (0,25)	56	13	23	77	35	45
T20I10N1KP5KC0.25D400K (0,1)	724	75	10	1109	381	34
T20I10N1KP5KC0.25D400K (0,25)	127	27	21	167	69	41
T20I10N1KP5KC0.25D800K (0,1)	341	165	48	1000	826	82
T20I10N1KP5KC0.25D800K (0,25)	283	59	20	380	158	41
T20I10N1KP5KC0.25D1.6MK (0,1)	733	342	46	3858	3403	88
T20I10N1KP5KC0.25D1.6MK (0,25)	186	109	58	393	320	65
T10I5N1KP5KC0.25D200K (0,01)	222	13	5	297	86	29
T10I5N1KP5KC0.25D200K (0,025)	88	8	9	123	50	41
T30I15N1KP5KC0.25D200K (0,25)	538	82	15	647	200	31
T30I15N1KP5KC0.25D200K (0,5)	115	30	26	132	47	35
T25I10D10K (0,05)	72	13	18	161	104	64
T25I10D10K (0,1)	20	5	25	24	9	37
T40I10D100K (0,5)	264	65	24	375	200	53
T40I10D100K (0,75)	100	37	37	128	66	51
T10I4D100K (0,01)	103	7	6	307	199	65
T10I4D100K (0,02)	52	5	10	74	29	39
T30I16D400K (0,4)	472	123	26	751	451	60
T30I16D400K (0,6)	156	61	39	218	138	63
BMSPOS (0,1)	154	8	5	415	211	51
BMSPOS (0,25)	31	4	13	62	30	48
Kosarak (0,1)	25	22	88	509	493	97
Kosarak (0,25)	14	5	35	51	34	66
Retail (0,01)	58	16	27	84	41	49
Retail (0,025)	15	9	60	23	16	69

Tabela 4.2: Tempo de execução da terceira iteração e tempo total de execução obtidos pelos algoritmos *kDCI++* e *kDCI-3* para diferentes combinações de bases de dados e valores de suporte mínimo.

Nestes experimentos, o tempo da terceira iteração do algoritmo *kDCI-3* foi, em média, 27% (variando de 5% a 88%) do tempo da mesma iteração do algoritmo *kDCI++*, logo alcançando uma redução média de 73%. Da mesma forma, o tempo total de execução do algoritmo *kDCI-3* variou de 29% a 97%, representando, em média, 53% do tempo total de execução do algoritmo *kDCI++*.

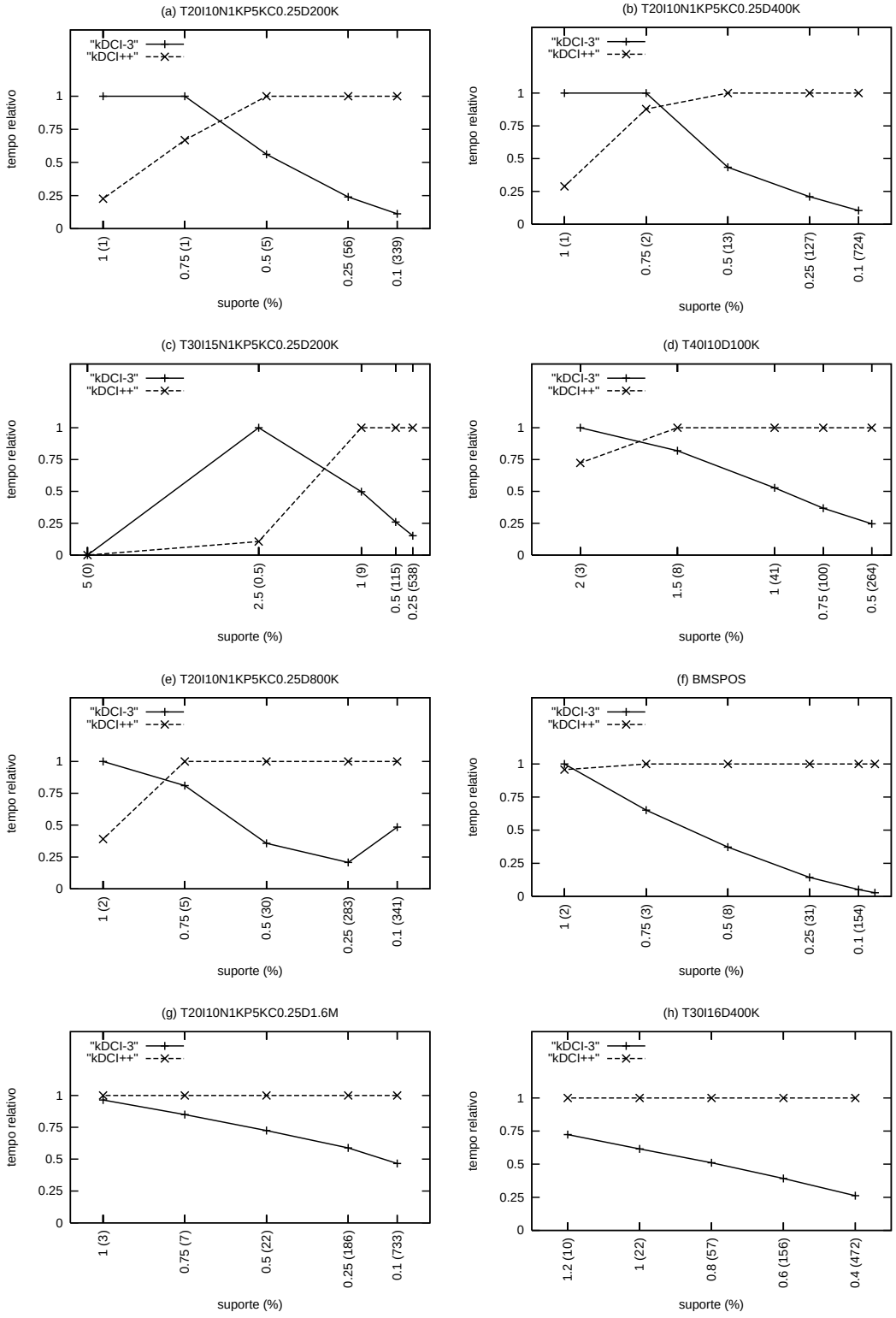
A seguir, são feitas avaliações dos resultados obtidos pelos algoritmos *kDCI++* e *kDCI-3* sobre as mesmas bases de dados, porém considerando-se novos valores de suporte mínimo.

Avaliação do Tempo de Execução da Terceira Iteração

Os experimentos realizados para a avaliação de desempenho da terceira iteração são apresentados na Figura 4.1. Para cada base de dados utilizada nos experimentos, é apresentado um gráfico com o tempo de execução relativo da terceira iteração dos algoritmos *kDCI++* e *kDCI-3*, considerando-se cinco valores de suporte mínimo. Nestes gráficos, o valor 1 no eixo y representa o tempo relativo de execução do algoritmo que obteve o pior desempenho. No eixo x, o valor entre parênteses representa o tempo absoluto (em segundos) da terceira iteração obtido pelo algoritmo de pior desempenho.

De forma a dar subsídios às avaliações de desempenho realizadas a seguir, a Tabela A.1, no Apêndice A, apresenta o número de candidatos de tamanho 3 gerados pelos algoritmos *kDCI++* e *kDCI-3* em cada uma das combinações de bases de dados e valores de suporte mínimo considerados nos experimentos.

Nas bases de dados ilustradas nos gráficos (a), (b), (c), (d) e (f), para os suportes mais altos, o algoritmo *kDCI-3* apresenta um desempenho ligeiramente inferior ao do algoritmo *kDCI++* ou reduz levemente o tempo de execução da terceira iteração (embora os tempos de execução absolutos sejam muito pouco expressivos). Conforme indicado na Tabela A.1, o número de candidatos de tamanho 3 gerados é relativamente pequeno para os suportes mais altos. Por esta razão, considerando-se que o algoritmo *kDCI++* faz uso da base verticalizada em todos os valores de suporte


(a) T20I10N1KP5KC0.25D200K
"kDCI-3"
"kDCI++"
tempo relativo
0
0.25
0.5
0.75
1
1 (1)
0.75 (1)
0.5 (5)
0.25 (56)
0.1 (339)
suporte (%)
(b) T20I10N1KP5KC0.25D400K
"kDCI-3"
"kDCI++"
tempo relativo
0
0.25
0.5
0.75
1
1 (1)
0.75 (2)
0.5 (13)
0.25 (127)
0.1 (724)
suporte (%)
(c) T30I15N1KP5KC0.25D200K
"kDCI-3"
"kDCI++"
tempo relativo
0
0.25
0.5
0.75
1
5 (0)
2.5 (0.5)
1 (9)
0.5 (115)
0.25 (538)
suporte (%)
(d) T40I10D100K
"kDCI-3"
"kDCI++"
tempo relativo
0
0.25
0.5
0.75
1
2 (3)
1.5 (8)
1 (41)
0.75 (100)
0.5 (264)
suporte (%)
(e) T20I10N1KP5KC0.25D800K
"kDCI-3"
"kDCI++"
tempo relativo
0
0.25
0.5
0.75
1
1 (2)
0.75 (5)
0.5 (30)
0.25 (283)
0.1 (341)
suporte (%)
(f) BMSPOS
"kDCI-3"
"kDCI++"
tempo relativo
0
0.25
0.5
0.75
1
1 (2)
0.75 (3)
0.5 (8)
0.25 (31)
0.1 (154)
suporte (%)
(g) T20I10N1KP5KC0.25D1.6M
"kDCI-3"
"kDCI++"
tempo relativo
0
0.25
0.5
0.75
1
1 (3)
0.75 (7)
0.5 (22)
0.25 (186)
0.1 (733)
suporte (%)
(h) T30I16D400K
"kDCI-3"
"kDCI++"
tempo relativo
0
0.25
0.5
0.75
1
1.2 (10)
1 (22)
0.8 (57)
0.6 (156)
0.4 (472)
suporte (%)

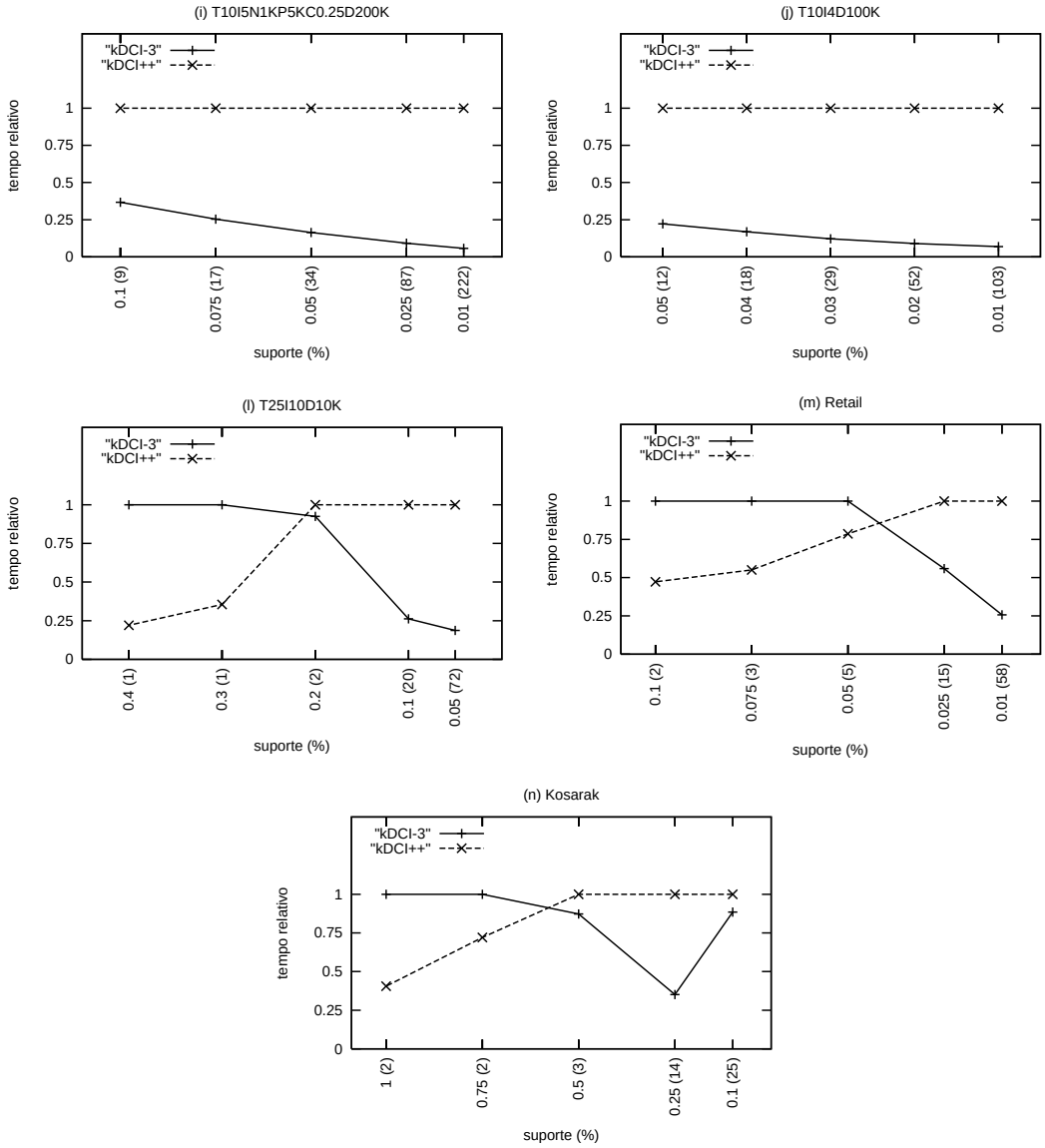


Figura 4.1: Avaliação de desempenho do tempo de execução relativo da terceira iteração dos algoritmos *kDCI++* e *kDCI-3*.

mínimo avaliados[1], este comportamento pode ser justificado pelo bom desempenho da técnica de interseções de listas de bits quando um número pequeno de candidatos é avaliado.

Para os suportes mais baixos, o número de candidatos cresce significativamente, chegando a 3.187.477 conjuntos na base de dados (a) e suporte mínimo 0,1%. Este comportamento é comum para todas as bases de dados relacionadas na Tabela A.1. À medida que o suporte mínimo diminui, o número de itens com suporte maior ou igual ao mínimo (freqüentes) aumenta. O mesmo ocorre com o número de conjuntos freqüentes de tamanho 2. Desta forma, como os candidatos de tamanho 3 são gerados a partir da combinação dos conjuntos freqüentes de tamanho 2, quanto maior o número destes conjuntos, maior o número de candidatos de tamanho 3 gerados.

Além do grande número de candidatos gerados para os suportes mais baixos, no algoritmo *kDCI++* a contagem dos candidatos de tamanho 3 é feita através de interseções de listas de bits cujo tamanho varia de acordo com o número de transações corrente. Como o tamanho de cada lista de bits, nestas execuções, variou de 3.125 inteiros na base de dados (d) a 16.113 inteiros na base de dados (f), tudo indica que a queda acentuada de desempenho do algoritmo *kDCI++* para os suportes baixos seja devido ao número de interseções de bits realizadas durante a contagem de um número significativamente alto de candidatos. Note que, para estas bases de dados, quanto maior o número de candidatos gerados, maior a redução do tempo da terceira iteração alcançada pelo algoritmo *kDCI-3*. Sobre a base de dados real *BMSPOS* (gráfico (f)), o algoritmo *kDCI++* adotou a técnica de dedução, apresentada na Seção 2.2.3, durante todas as iterações. Mesmo com o uso desta otimização, o algoritmo *kDCI-3* obteve desempenho superior para quase todos os valores de suporte.

No gráfico da base de dados *T20I10N1KP5KC0.25D800K* (gráfico (e)), observa-se que, para os suportes mais altos, o desempenho do algoritmo *kDCI-3* apresenta o mesmo comportamento analisado anteriormente. Para os valores de

[1]Esta condição foi identificada a partir de *logs* de execução.

suporte mínimo baixos, 0,5% e 0,25%, da mesma forma que nas bases de dados já avaliadas, o aumento do número de candidatos (chegando a 326.523 conjuntos para o suporte 0,5% e 3.375.811 conjuntos para o suporte 0,25%) e as grandes listas de bits geradas para estes suportes (24.677 inteiros) contribuíram para a queda de desempenho do algoritmo *kDCI++*. Para o suporte mínimo 0,1%, como a base de dados verticalizada não está em memória principal durante a terceira iteração do algoritmo *kDCI++*, a contagem dos candidatos de tamanho 3 é feita através da estrutura de dados T_3. Neste caso, a diferença entre os desempenhos alcançados pelos algoritmos *kDCI++* e *kDCI-3*, embora menos significativa que nos demais suportes baixos, pode ser explicada pela busca seqüencial adotada pelo algoritmo *kDCI++* na contagem dos candidatos. No algoritmo *kDCI-3*, esta contagem se dá de maneira mais eficiente.

Nos gráficos (g), (h), (i) e (j), é possível observar o impacto da nova técnica de contagem de candidatos a partir dos suportes mais altos. Nas bases (h), (i) e (j) para todos os valores de suporte avaliados, a representação verticalizada da base de dados coube em memória principal já na terceira iteração do algoritmo *kDCI++* e o tamanho da lista de bits foi de 12.422 inteiros para a base de dados *T30I16D400K*, 6.028 inteiros para a a base de dados *T10I5N1KP5KC0.25D200k* e de 3.125 inteiros para a base *T10I4D100K*. Observe que, além do grande número de candidatos gerados nestas bases (chegando a 11.051.970 conjuntos na base de dados (j) e suporte 0,1%), o tamanho das listas de bits geradas também poderá ter contribuído para a queda de desempenho do algoritmo *kDCI++*. Na base de dados *T20I10N1KP5KC0.25D1.6M* (gráfico (g)), a base verticalizada não estava em memória principal durante a terceira iteração em nenhum dos suportes avaliados. O baixo desempenho do algoritmo *kDCI++* pode ser justificado pela busca seqüencial na contagem dos candidatos.

Nos experimentos realizados sobre a base de dados *T25I10D10K*, ilustrada no gráfico (l), a base verticalizada do algoritmo *kDCI++* está em memória principal a partir da terceira iteração em todos os valores de suporte mínimo avaliados. Para os valores de suporte mais altos, embora o número de candidatos seja expressivo (chegando a 423.650 conjuntos para o suporte 0,2%), o tamanho da lista de bits nes-

tas execuções é de apenas 289 inteiros. O número de interseções efetuadas durante a fase de contagem é significativamente menor, se comparado com as demais bases de dados já avaliadas. Desta forma, a contagem de candidatos através de interseções de lista de bits, para esta base e para os valores de suporte mais altos, foi mais eficiente que o acesso direto utilizado pelo algoritmo *kDCI-3*. Observe porém que os tempos de execução são pouco expressivos. Entretanto, para os suportes mais baixos, embora o tamanho da lista de bits tenha se mantido em 289 inteiros, o número de candidatos cresceu consideravelmente (chegando a 44.171.824 candidatos para o suporte 0,05%), o que causou a queda de desempenho do algoritmo *kDCI++*. Conforme já observado na Seção 2.7, os autores do *kDCI++* atribuem o mau desempenho do algoritmo nesta mesma base, para o suporte mínimo 0,1%, ao tamanho do conjunto C_3, que é significativamente maior que o conjunto F_3 encontrado [17]. Neste caso, o algoritmo *kDCI++* ocupa grande parte do tempo da terceira iteração na contagem de conjuntos candidatos não freqüentes.

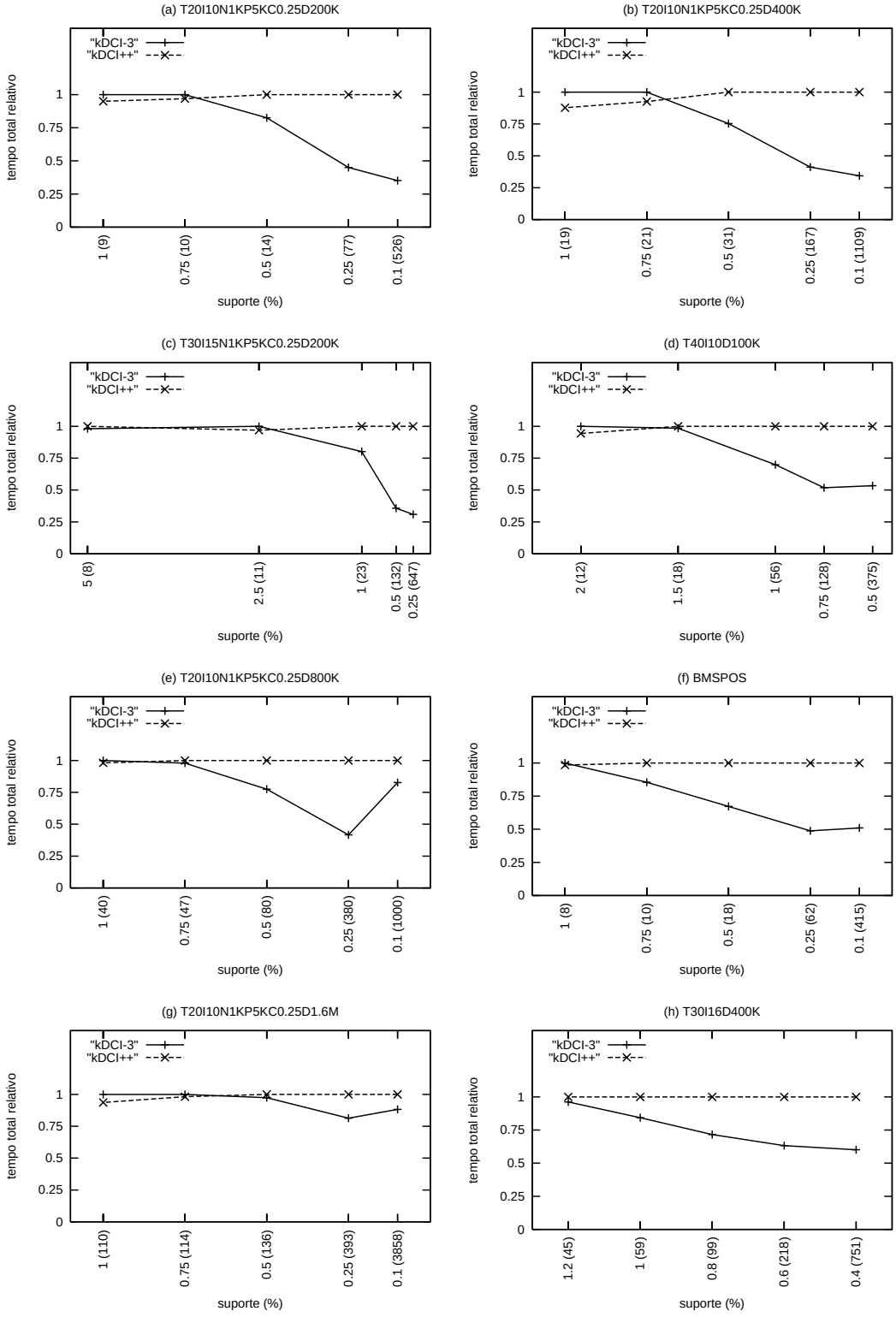
Sobre a base de dados real *Retail* (gráfico (m)), repare que o número de candidatos gerados para os suportes mais altos não é tão significativo quanto para os suportes mais baixos. Além disso, o tamanho das listas de bits geradas (2.756 inteiros) é relativamente pequeno se comparado ao tamanho destas listas nas primeiras bases de dados avaliadas. Desta forma, apesar dos tempos de execução absolutos serem pequenos, o algoritmo *kDCI++* apresentou um desempenho superior ao do algoritmo *kDCI-3* para estes valores de suporte. No entanto, observe que para o suporte 0,05%, o aumento do número de candidatos (de 7.468 conjuntos para 17.238 conjuntos) reduz a diferença de desempenho relativa entre os algoritmos. Para o suporte 0,025%, a técnica de acesso direto aos candidatos supera a utilização das interseções de listas de bits, mesmo com a utilização da técnica de dedução. Finalmente, para o suporte 0,1%, devido à grande quantidade de memória consumida pela estrutura T_3, o algoritmo *kDCI++* faz uso de memória virtual, o que torna esta fase mais lenta. Já o algoritmo *kDCI-3*, apesar de também precisar de uma quantidade significativa de memória para a construção da estrutura T_3', faz uso do gerenciamento de memória descrito na Seção 3.3, não prejudicando, assim, a execução da iteração (as vantagens alcançadas pelo gerenciamento de memória serão

discutidas na Seção 4.2.2).

Finalmente, nos testes sobre a base de dados real *Kosarak* (gráfico (n)), observa-se que, para os suportes mais altos, o algoritmo *kDCI++* se mostra um pouco mais eficiente que o algoritmo *kDCI-3*, o que, mais uma vez, pode ser explicado pelo número pouco significativo de candidatos avaliados nestas execuções. Entretanto, para o suporte 0,25%, apesar do aumento do número de candidatos não ter sido tão expressivo (de 897 para 4.631 conjuntos), as grandes listas de bits geradas (30.938 inteiros) podem ter contribuído para a reversão de tal comportamento. Durante a execução da terceira iteração, assim como das demais iterações, o algoritmo *kDCI++* utilizou a técnica de dedução. Para o suporte 0,1%, a base de dados verticalizada coube em memória principal apenas durante a quinta iteração e, por esta razão, o algoritmo *kDCI++* utilizou a estrutura de dados T_3 para a contagem dos candidatos de tamanho 3. Para este suporte, a técnica utilizada pelo algoritmo *kDCI++* foi apenas ligeiramente inferior à técnica utilizada pelo algoritmo *kDCI-3*, o que pode ser justificado pelo seguinte motivo. Inicialmente, para cada uma das transações t da base de dados, o algoritmo *kDCI++* encontra os possíveis prefixos de tamanho 2 presentes em t. Para cada prefixo i em t, o algoritmo verifica na seção do vetor S correspondente aos candidatos de prefixo i, quais daqueles candidatos estão presentes em t e, então, incrementa os respectivos suportes. Como o tamanho das transações nesta base de dados é, em média, pequeno (apenas 8.1 itens), tudo indica que os possíveis prefixos de tamanho 2 sejam poucos. Da mesma forma, como nesta execução as seções em S correspondentes aos candidatos de mesmo prefixo são formadas, em média, por apenas 4 candidatos, acredita-se que a busca pelos candidatos presentes nas transações seja feita de forma relativamente rápida.

Avaliação do Tempo Total de Execução

Nesta subseção, para cada base de dados utilizada nos experimentos, a Figura 4.2 apresenta um gráfico com o tempo total de execução relativo obtido pelos algoritmos *kDCI++* e *kDCI-3*, considerando-se os mesmos valores de suporte mínimo avaliados anteriormente.


(a) T20I10N1KP5KC0.25D200K
"kDCI-3"
"kDCI++"
tempo total relativo
0
0.25
0.5
0.75
1
1 (9)
0.75 (10)
0.5 (14)
0.25 (77)
0.1 (526)
suporte (%)
(b) T20I10N1KP5KC0.25D400K
"kDCI-3"
"kDCI++"
tempo total relativo
0
0.25
0.5
0.75
1
1 (19)
0.75 (21)
0.5 (31)
0.25 (167)
0.1 (1109)
suporte (%)
(c) T30I15N1KP5KC0.25D200K
"kDCI-3"
"kDCI++"
tempo total relativo
0
0.25
0.5
0.75
1
5 (8)
2.5 (11)
1 (23)
0.5 (132)
0.25 (647)
suporte (%)
(d) T40I10D100K
"kDCI-3"
"kDCI++"
tempo total relativo
0
0.25
0.5
0.75
1
2 (12)
1.5 (18)
1 (56)
0.75 (128)
0.5 (375)
suporte (%)
(e) T20I10N1KP5KC0.25D800K
"kDCI-3"
"kDCI++"
tempo total relativo
0
0.25
0.5
0.75
1
1 (40)
0.75 (47)
0.5 (80)
0.25 (380)
0.1 (1000)
suporte (%)
(f) BMSPOS
"kDCI-3"
"kDCI++"
tempo total relativo
0
0.25
0.5
0.75
1
1 (8)
0.75 (10)
0.5 (18)
0.25 (62)
0.1 (415)
suporte (%)
(g) T20I10N1KP5KC0.25D1.6M
"kDCI-3"
"kDCI++"
tempo total relativo
0
0.25
0.5
0.75
1
1 (110)
0.75 (114)
0.5 (136)
0.25 (393)
0.1 (3858)
suporte (%)
(h) T30I16D400K
"kDCI-3"
"kDCI++"
tempo total relativo
0
0.25
0.5
0.75
1
1.2 (45)
1 (59)
0.8 (99)
0.6 (218)
0.4 (751)
suporte (%)

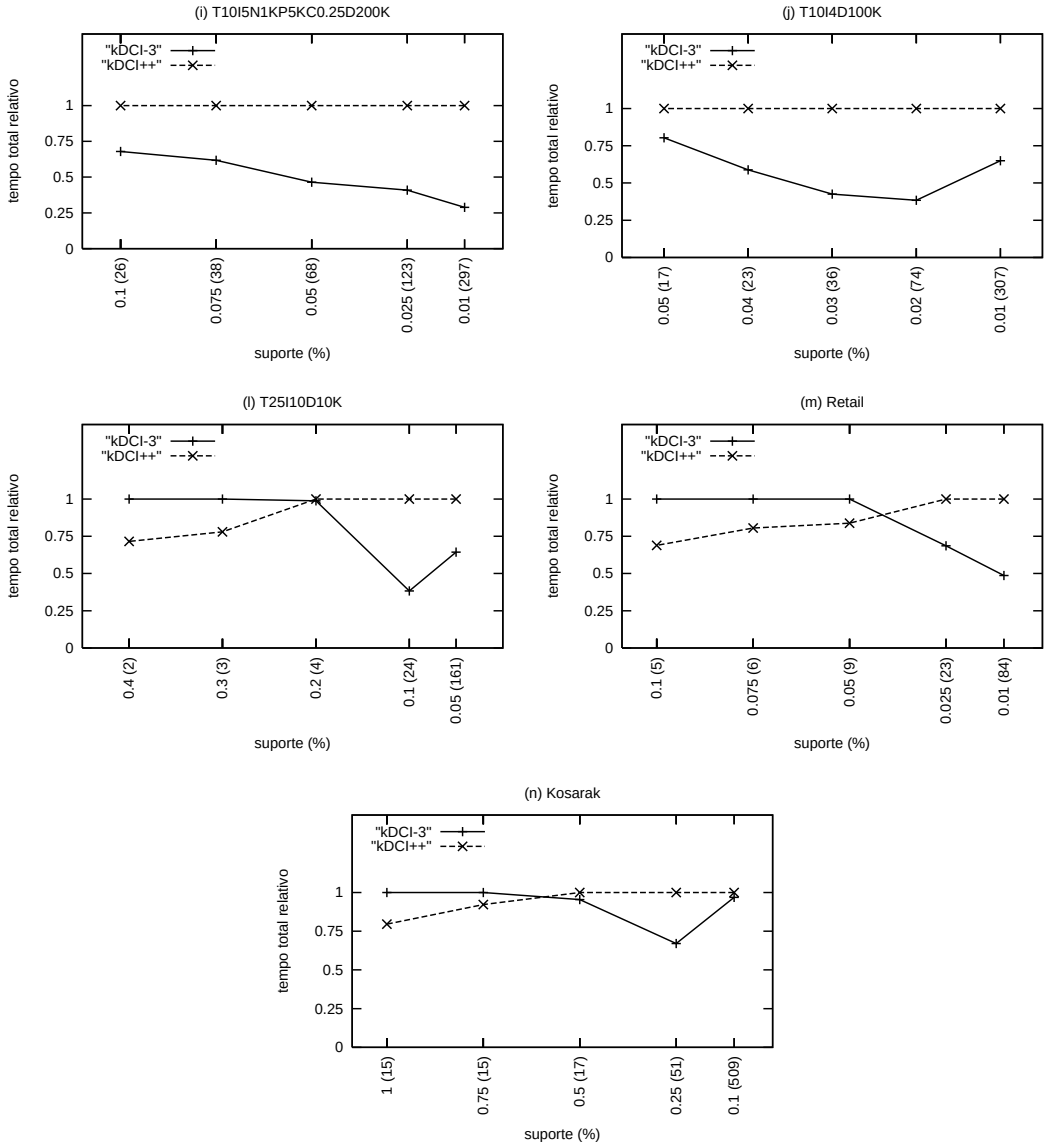


Figura 4.2: Avaliação de desempenho do tempo total de execução relativo dos algoritmos *kDCI++* e *kDCI-3*.

Os gráficos da Figura 4.2 mostram que, para todas as bases de dados utilizadas nos experimentos, a técnica de contagem de candidatos adotada pelo algoritmo *kDCI-3* reduz o tempo total de execução obtido pelo algoritmo *kDCI++* quando valores de suporte mínimo baixos são considerados. Para os suportes mais altos, com exceção das bases de dados (h), (i) e (j), devido ao número pouco expressivo de candidatos de tamanho 3, o desempenho do algoritmo *kDCI-3* foi ligeiramente inferior ou semelhante ao do algoritmo *kDCI++*. Vale observar que os tempos absolutos destas execuções foram pouco significativos. Para as base de dados ilustradas nos gráficos (h), (i) e (j), o algoritmo *kDCI-3* contribuiu para a redução do tempo total de execução já a partir dos suportes mais altos, o que evidencia o bom desempenho alcançado durante a terceira iteração nestas bases de dados.

Entretanto, da análise dos resultados obtidos sobre as bases de dados dos gráficos (e), (g) e (n) nota-se que, para o suporte mínimo mais baixo, o percentual de redução do tempo total de execução alcançado pelo algoritmo *kDCI-3* não é tão significativo quanto nas demais bases de dados (os percentuais estão apresentados na Tabela 4.2). Isto se deve ao fato de que, para estes valores de suporte mínimo, as iterações posteriores à terceira iteração também apresentam um custo computacional significativo. Como exemplo, na base de dados (e) e suporte mínimo 0,1%, a sexta iteração é 31% do tempo total de execução do algoritmo *kDCI++*. Já na base de dados (g) e suporte 0,1%, o somatório dos tempos de execução consumidos pelas iterações 8, 9 e 10 representam 45% do tempo total do algoritmo. O mesmo foi observado na base de dados (n): para o suporte 0,1%, as iterações 7, 8 e 9 representam, juntas, 46% do tempo total de execução do algoritmo *kDCI++*.

É importante ressaltar que nas bases de dados dos gráficos (j) e (l), apesar da técnica de contagem de candidatos utilizada pelo algoritmo *kDCI-3* ter reduzido de maneira significativa o tempo total de execução, nota-se que, para o suporte mais baixo considerado, existe comportamento semelhante ao apresentado para as bases de dados analisadas no parágrafo anterior. Para a base de dados do gráfico (j), com suporte mínimo 0,01%, apesar do algoritmo *kDCI-3* ter reduzido 94% o tempo da terceira iteração, a quarta iteração, equivalente a 46% do tempo total de execução do algoritmo *kDCI++*, contribuiu para a redução de apenas 45% do tempo

total. Já na base de dados do gráfico (l) e suporte mínimo 0,05%, as iterações 4, 5 e 6 representam 46% do tempo total consumidos pelo algoritmo *kDCI++*. Desta forma, devido ao custo computacional destas iterações, a redução de 82% do tempo da terceira iteração alcançada pelo algoritmo *kDCI-3* reduziu apenas 36% do tempo total de execução.

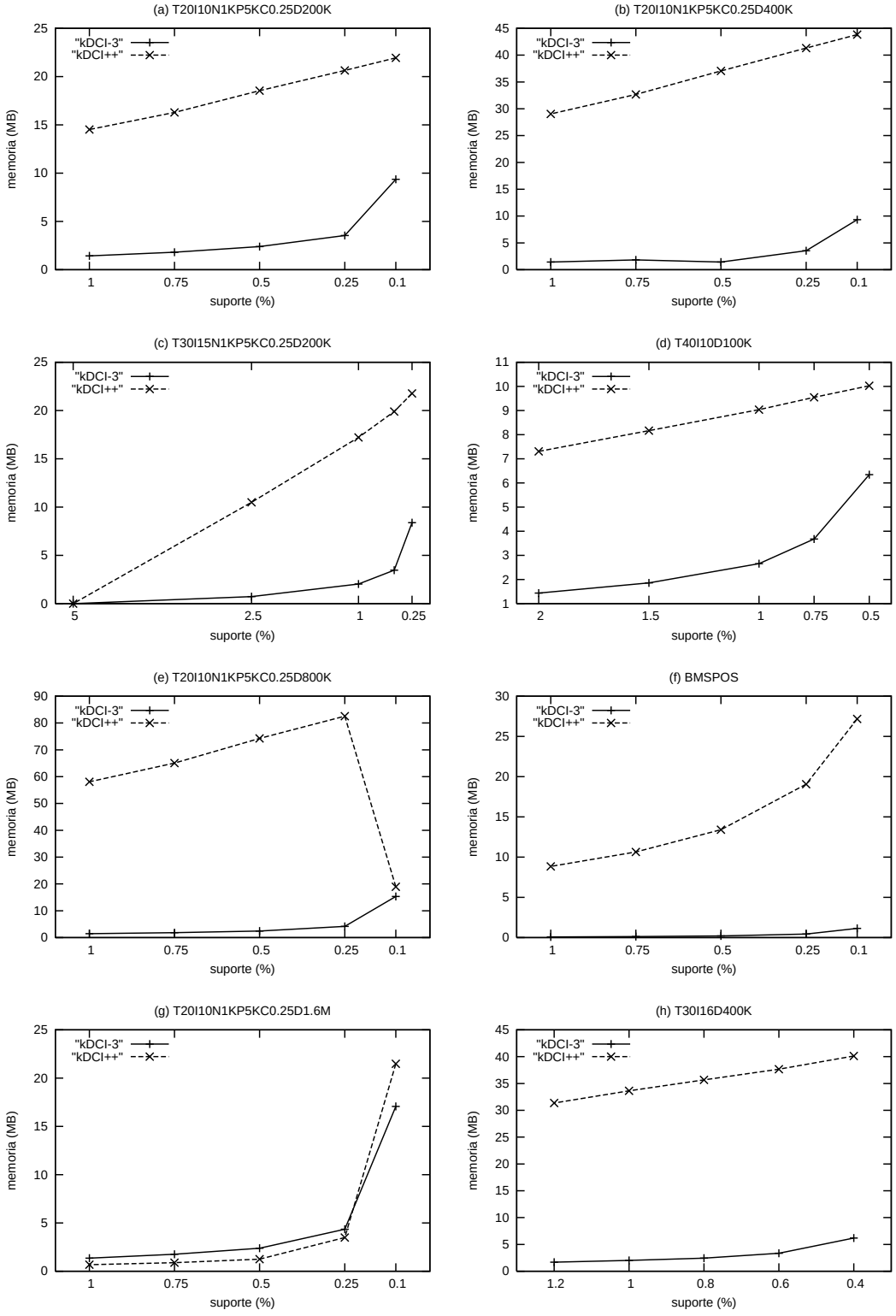
4.2.2 Memória

Esta seção é dedicada à análise da quantidade de memória consumida pelos algoritmos *kDCI++* e *kDCI-3*, durante a execução da terceira iteração.

Para a avaliação da memória utilizada pelo algoritmo *kDCI-3*, levou-se em consideração as estruturas de dados T_3' e C', apresentadas na Seção 3.1. Para o algoritmo *kDCI++*, quando a representação verticalizada da base de dados estava em memória principal, foram consideradas a memória consumida pela nova representação, além da estrutura de dados *Cache*, que auxilia a contagem dos candidatos durante toda iteração (Subseção 2.2.2). Quando a base verticalizada ainda não se encontrava em memória principal, considerou-se a memória necessária para a utilização das estruturas T_3, P, I, S e C, descritas na Subseção 2.2.1.

A Figura 4.3 apresenta, para cada base de dados utilizada nos experimentos, a quantidade de memória (em *megabytes*) consumida pelos algoritmos *kDCI++* e *kDCI-3*, em cada um dos valores de suporte mínimo considerados.

Observe que, nas bases de dados dos gráficos (a), (b), (c), (d), (f) e (h), o algoritmo *kDCI++* consome uma quantidade de memória significativamente maior do que a consumida pelo algoritmo *kDCI-3*, em todos os suportes mínimos. Isto se deve à grande quantidade de memória necessária para a construção da representação verticalizada da base de dados nestes experimentos. Como exemplo, considere a base de dados real *BMSPOS* (gráfico (f)) e o suporte mínimo 0,25%. A base de dados verticalizada, conforme descrita na Subseção 2.2.2, pode ser vista como um conjunto de $|M_k|$ listas de bits de tamanho $|N_k|$, onde M_k e N_k são, respectivamente, os conjuntos de itens e transações presentes na base de dados corrente. Vale lembrar


(a) T20I10N1KP5KC0.25D200K
(b) T20I10N1KP5KC0.25D400K
(c) T30I15N1KP5KC0.25D200K
(d) T40I10D100K
(e) T20I10N1KP5KC0.25D800K
(f) BMSPOS
(g) T20I10N1KP5KC0.25D1.6M
(h) T30I16D400K
"kDCI-3"
"kDCI++"
memoria (MB)
suporte (%)

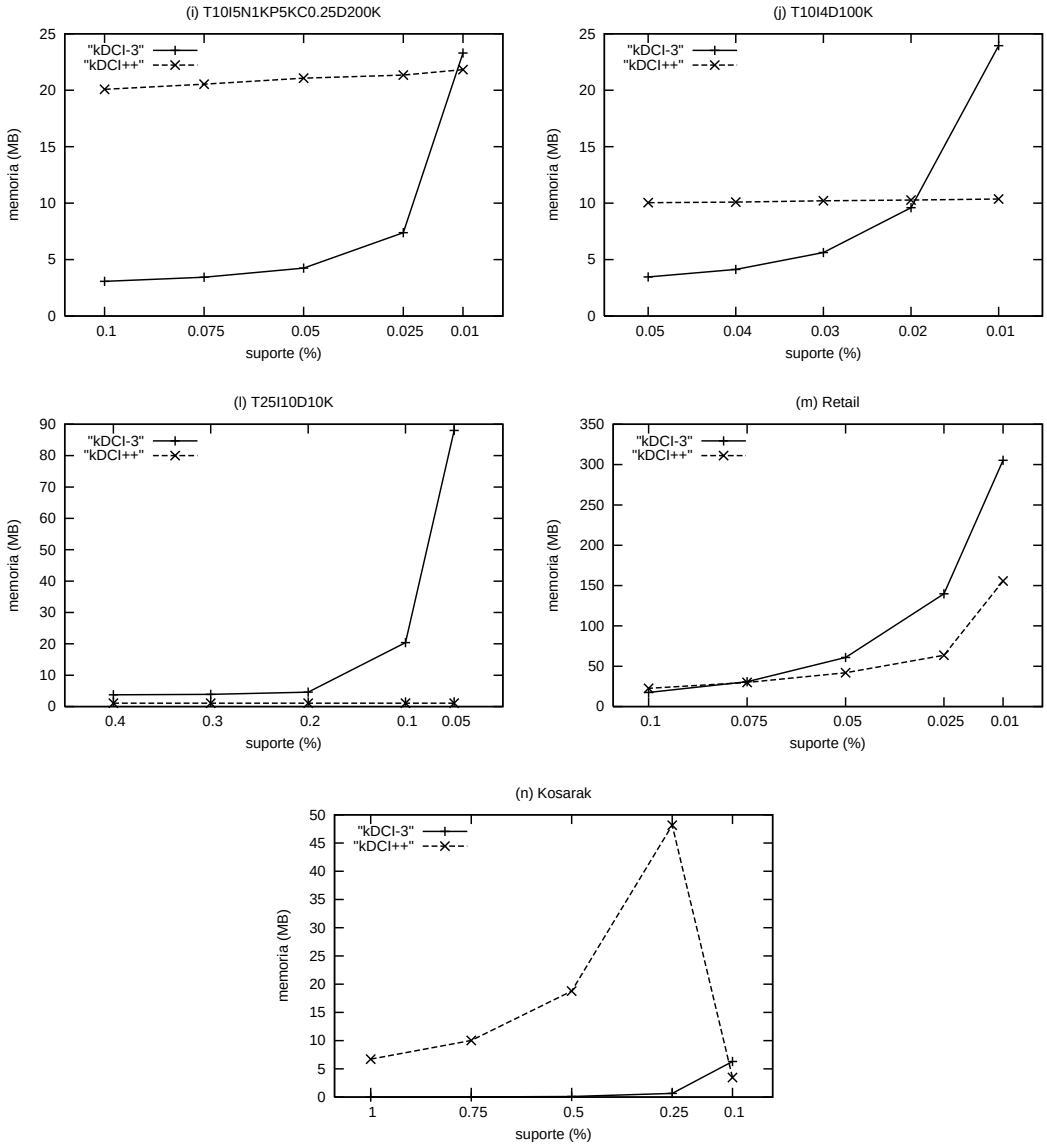


Figura 4.3: Avaliação de desempenho da memória consumida durante a terceira iteração pelos algoritmos *kDCI++* e *kDCI-3*.

que cada lista de bits está associada a um determinado item e cada bit de uma lista corresponde a uma determinada transação. Para a base de dados e suporte mínimo considerados, $|N_k|$ é igual a 307 e $|M_k|$ igual a 515.597. Desta forma, cada lista de bits associada aos 307 itens correntes deverá conter uma quantidade de bits suficiente para a representação de 515.597 transações. Como, na linguagem C, um inteiro corresponde a 4 *bytes* e como cada *byte* é composto por 8 bits, cada inteiro é capaz de representar 4*8 = 32 transações. Como o algoritmo necessita representar 515.597 transações, cada lista de bits será formada, então, por $\frac{515.597}{32}$ = 16.113 inteiros, dando um total de 16.113 * 4 *bytes* = 64.452 *bytes* consumidos por lista. Portanto, como a base verticalizada é composta por 307 listas de bits (307 itens), esta estrutura consome, ao todo, 64.452 *bytes* * 307 itens = 19.786.764 *bytes* $\simeq$ 18,87 *megabytes*.

Além da base verticalizada, o algoritmo *kDCI++* também utiliza a estrutura de dados *Cache*, que auxilia as interseções necessárias para a contagem dos candidatos. Como esta estrutura de dados consome k vezes o tamanho (em inteiros) de uma lista de bits construída, onde k é a iteração corrente, a memória consumida por *Cache* é de 3 * 16.113 inteiros = 48.339 inteiros = 193.356 *bytes* $\simeq$ 0,18 *megabytes*. Desta forma, para o exemplo considerado, a memória total consumida pelo algoritmo *kDCI++* é de 18,87 + 0,18 = 19,05 *megabytes*, conforme indicado no gráfico correspondente.

Já no algoritmo *kDCI-3*, considerou-se a memória necessária para a construção das estruturas T_3' e C'. Cada entrada da estrutura de dados T_3' é composta por dois inteiros. Desta forma, como o número de entradas de T_3' é igual ao número de candidatos de tamanho 2, que para este exemplo é 46.871, a memória ocupada por esta estrutura é de 46.871 * 8 *bytes* (dois inteiros) = 374.968 *bytes* $\simeq$ 0,36 *megabytes*. Para a estrutura de dados C', a memória consumida é calculada multiplicando-se o número de candidatos de tamanho 3, que neste caso é 21.228, pelo número de bytes correspondente a um inteiro, resultando em um total de 21.228 * 4 *bytes* = 84.912 *bytes* $\simeq$ 0,08 *megabytes*. Portanto, conforme apresentado no gráfico correspondente, a memória total consumida pelo algoritmo *kDCI-3* é de apenas 0,44 *megabytes*, aproximadamente.

Nas bases de dados dos gráficos (e) e (n), para os suportes mínimos nos quais a base verticalizada se encontra em memória principal (quatro primeiros valores de suporte), pode-se observar o mesmo comportamento das bases analisadas anteriormente. Devido à grande quantidade de memória utilizada para a construção das bases verticalizadas, o algoritmo *kDCI++* apresenta um desempenho significativamente inferior ao do algoritmo *kDCI-3*. Entretanto, para os suportes em que o algoritmo *kDCI++* utilizou as estruturas T_3, P, I, S e C para a contagem dos conjuntos candidatos, os desempenhos alcançados pelos algoritmos foram bastante semelhantes. Considere a base de dados *T20I10N1KP5KC0.25D800K* (gráfico (e)) e suporte mínimo 0,1%. A memória consumida por tais estruturas foi contabilizada da seguinte forma. T_3 possui um número de entradas igual ao número de candidatos de tamanho 2, que neste caso é de 431.056. Como cada entrada é composta por um inteiro, a quantidade de memória consumida por esta estrutura é de 431.056 * 4 *bytes* = 1.724.228 *bytes* $\simeq$ 1,64 *megabytes*. O vetor P armazenou um total de 122.398 prefixos. Como cada uma das 122.398 entradas era composta por um inteiro curto (equivalente a 2 *bytes* na linguagem C), o total consumido por este vetor foi de 122.398 * 2 *bytes* = 244.796 *bytes* $\simeq$ 0,23 *megabytes*. Da mesma forma, o vetor S, com 2.942.815 sufixos armazenados, consumiu um total de 2.942.815 * 2 *bytes* = 5.885.630 *bytes* $\simeq$ 5,61 *megabytes*. Já o vetor I armazenou um total de 61.199 inteiros, consumindo uma memória de 61.199 * 4 *bytes* = 244.796 *bytes* $\simeq$ 0,23 *megabytes*. Finalmente, a estrutura C, com um total de 2.942.815 inteiros (número de candidatos de tamanho 3), utilizou 2.942.815 * 4 *bytes* = 11.771.260 *bytes* $\simeq$ 11,22 *megabytes*. Desta forma, o total consumido pelas estruturas, conforme indicado no gráfico (e), foi de 1,64 + 0,23 + 5,61 + 0,23 + 11,22 = 18,93 *megabytes*. Para o algoritmo *kDCI-3*, foram consideradas a memória consumida pela estrutura T'_3, igual a 431.056 (número de candidatos de tamanho 2) * 8 *bytes* (dois inteiros) = 3.448.448 *bytes* $\simeq$ 3,29 *megabytes* e a memória consumida pela estrutura de dados C', igual a 3.155.473 (número de candidatos de tamanho 3) * 4 *bytes* (um inteiro) = 12.621.892 *bytes* = 12,03 *megabytes*, aproximadamente.

Observe que o número de candidatos de tamanho 3 gerados pelo algoritmo *kDCI-3* é maior que o mesmo número gerado pelo algoritmo *kDCI++*, para esta base

de dados e suporte mínimo. A diferença no número de candidatos gerados pode ser justificada pelo fato do algoritmo *kDCI++* podar candidatos durante sua geração, quando a base de dados verticalizada ainda não se encontra em memória principal.

Na base de dados ilustrada pelo gráfico (g), os desempenhos dos algoritmos apresentam-se bastante semelhantes. Para esta base de dados e para todos os valores de suporte mínimo considerados, o algoritmo *kDCI++* utilizou a estrutura T_3 na contagem dos conjuntos candidatos.

Nos gráficos (i) e (j), observa-se que, para os suportes mais baixos, a memória consumida pelo algoritmo *kDCI-3* ultrapassa a consumida pelo algoritmo *kDCI++*. Tal comportamento é devido à explosão de conjuntos candidatos gerados para estes valores de suporte, ocasionando em um aumento significativo da estrutura de dados C' utilizada pelo algoritmo *kDCI-3*.

Nas bases de dados dos gráficos (l) e (m), mesmo para os suportes mais altos, o desempenho do algoritmo *kDCI++* foi melhor que o do algoritmo *kDCI-3*. Isto se deve ao fato de que, nestas bases de dados, o tamanho das listas de bits construídas é relativamente menor que nas bases de dados dos gráficos já analisados. Por exemplo, no caso da base de dados *T25I10D10K* (gráfico (l)), a lista de bits tem o tamanho de apenas 289 inteiros em todos os suportes. Já na base de dados *Retail* (gráfico (m)), além do tamanho das listas de bits não ser tão grande quanto nas demais bases (2.756 inteiros), o número de candidatos de tamanho 2 gerados é bastante alto (variando de 2.288.730 para o suporte 0,1% e 39.769.821 para o suporte 0,01%), principalmente para os suportes mais baixos. Por esta razão, como a memória consumida pela estrutura T'_3, utilizada pelo algoritmo *kDCI-3*, é dependente do número destes candidatos, a memória total consumida pelo algoritmo torna-se significativamente alta. Por exemplo, considere o suporte 0,01%. A memória consumida pelo algoritmo *kDCI-3* é de um pouco mais de 300 *megabytes*, valor este maior que o total de memória principal disponível na máquina onde foram realizados os experimentos computacionais (256 *megabytes*). Desta forma, a contagem dos candidatos para o suporte 0,01% foi feita com a utilização da estratégia de gerenciamento de memória incorporada ao algoritmo *kDCI-3*. Vale observar porém que, apesar de

consumir mais memória, a terceira iteração do algoritmo *kDCI-3* foi mais rápida que a terceira iteração do algoritmo *kDCI++* para os suportes mais baixos.

A seguir, a Subseção 4.2.2 é dedicada à análise do desempenho alcançado pela técnica de gerenciamento de memória.

Avaliação da Técnica de Gerenciamento de Memória

Nos experimentos da base de dados *Retail*, considerando-se o suporte mínimo 0,01%, a contagem dos conjuntos candidatos foi realizada com a utilização da técnica de gerenciamento de memória desenvolvida para o algoritmo *kDCI-3*.

Apenas para efeito de comparação, criou-se uma adaptação do algoritmo *kDCI-3*, aqui chamada *kDCI-3**, em que a terceira iteração não utiliza a técnica de gerenciamento de memória. Em seguida, os experimentos sobre a base de dados *Retail*, considerando-se o suporte mínimo 0,01%, foi realizado.

A Tabela 4.3 apresenta o tempo de execução da terceira iteração (em segundos) e o tempo total de execução (em segundos) obtidos pelo algoritmo *kDCI-3** e pelo algoritmo *kDCI-3*, respectivamente. A quarta coluna representa a relação entre o tempo da terceira iteração do algoritmo *kDCI-3* e o tempo da terceira iteração do algoritmo *kDCI-3**, enquanto que a sétima coluna representa a relação entre o tempo total de execução do algoritmo *kDCI-3* e o tempo total de execução do algoritmo *kDCI-3**.

	Tempo 3^a iter.			Tempo Total		
Base de dados (*supmin* - %)	*kDCI-3**	*kDCI-3*	%	*kDCI-3**	*kDCI-3*	%
Retail (0,01)	370	16	4	413	41	10

Tabela 4.3: Tempo de execução da terceira iteração e tempo total de execução obtidos com (*kDCI-3*) e sem (*kDCI-3**) o uso da técnica de gerenciamento de memória.

Observe que o tempo de execução da terceira iteração obtido pelo algoritmo *kDCI-3* é apenas 4% do mesmo tempo obtido pelo algoritmo *kDCI-3**, alcançando uma redução de 96%. Da mesma forma, o tempo total de execução obtido pelo

algoritmo *kDCI-3* é apenas 10% do tempo total de execução obtido pelo algoritmo *kDCI-3**, o que representou, portanto, uma redução de 90%. A razão do baixo desempenho do algoritmo *kDCI-3** se deve à quantidade de memória necessária para representar os conjuntos candidatos utilizando-se as estruturas de dados T_3' e C' que, nesta execução, foi maior do que a disponível.

Vale observar que nos experimentos realizados com o algoritmo *kDCI++* sobre esta mesma base de dados (*Retail*) e para este mesmo valor de suporte mínimo (0,01%), a quantidade de memória necessária para representar os conjuntos candidatos através das estruturas de dados T_3, P, I, S e C também foi maior que a disponível. Entretanto, a falta de uma estratégia de gerenciamento de memória tornou a execução de sua terceira iteração mais longa devido ao uso de memória virtual. Conforme apresentado na penúltima linha da Tabela 2.5, enquanto a terceira iteração do algoritmo *kDCI-3* foi executada em 16 segundos, a mesma iteração foi executada pelo algoritmo *kDCI++* em 58 segundos, representando uma redução de 73%.

Para evidenciar ainda mais a necessidade de uma estratégia de gerenciamento de memória, considerou-se os suportes 0,007% e 0,002% sobre a mesma base de dados. A redução do suporte mínimo teve como objetivo aumentar o número de conjuntos candidatos avaliados pelos algoritmos durante a terceira iteração e, conseqüentemente, aumentar ainda mais a memória necessária para a representação destes conjuntos.

A Tabela 4.4 apresenta o tempo de execução da terceira iteração (em segundos) e o tempo total de execução (em segundos) obtidos pelos algoritmo *kDCI++* e *kDCI-3* sobre a base de dados *Retail* considerando-se os suportes mínimos 0,007% e 0,002%. A quarta coluna representa a relação entre o tempo da terceira iteração do algoritmo *kDCI-3* e o tempo da terceira iteração do algoritmo *kDCI++*, enquanto que a sétima coluna representa a relação entre o tempo total de execução do algoritmo *kDCI-3* e o tempo total de execução do algoritmo *kDCI++*.

Para o suporte mínimo 0,007%, o número de conjuntos candidatos aumentou de 912.402 (suporte 0,01%) para 2.910.696 no algoritmo *kDCI++* e de 1.017.833

	Tempo 3^a iter.			Tempo Total		
Base de dados (*supmin* - %)	*kDCI++*	*kDCI-3*	%	*kDCI++*	*kDCI-3*	%
Retail (0,007)	188	22	12	255	84	33
Retail (0,002)	—	70	—	—	—	—

Tabela 4.4: Tempo de execução da terceira iteração e tempo total de execução obtidos pelos algoritmos *kDCI++* e *kDCI-3* sobre a base de dados *Retail* e suportes mínimos 0,007% e 0,002%.

(suporte 0,01%) para 3.381.474 no algoritmo *kDCI-3*. A diferença no número de candidatos gerados se deve ao fato de que, nesta execução, o algoritmo *kDCI++* podou candidatos durante sua geração. Como o *kDCI-3* não poda candidatos em nenhuma situação, este algoritmo gera um número maior de candidatos. Observe que, para este valor de suporte mínimo, a terceira iteração do algoritmo *kDCI++*, executada com o auxílio das estruturas de dados T_3, P, I, S e C, foi completada em 188 segundos, enquanto que o algoritmo *kDCI-3* executou a mesma iteração em 22 segundos. Da mesma forma que para o suporte 0,01%, o algoritmo *kDCI-3* executou a terceira iteração com a utilização da estratégia de gerenciamento de memória, o que não aconteceu no algoritmo *kDCI++*. A redução do tempo total de execução alcançada pelo algoritmo *kDCI-3* foi de 67%.

Para o suporte 0,002%, devido ao aumento significativo do número de candidatos de tamanho 3, a execução do algoritmo *kDCI++* foi abortada pelo sistema operacional devido à falta de memória durante a construção das estruturas T_3, P, I, S e C. De fato, o número de conjuntos candidatos foi de 66.402.305 no algoritmo *kDCI++* e de 89.736.928 no algoritmo *kDCI-3*. Para este mesmo suporte, o algoritmo *kDCI-3* executou a terceira iteração em 70 segundos, utilizando a estratégia de gerenciamento de memória. Entretanto, o número de candidatos de tamanho 4 a ser representado pelas estruturas T_3, P, I, S e C também foi significativamente alto. Desta forma, como a quarta iteração do algoritmo *kDCI-3* é equivalente a quarta iteração do algoritmo *kDCI++* e como este último não faz uso de nenhuma estratégia de gerenciamento de memória, o *kDCI-3* foi abortado pelo sistema operacional e não conseguiu completar a execução das demais iterações. Observe, portanto,

a importância do gerenciamento de memória em todas as iterações do algoritmo *kDCI++*.

4.2.3 Escalabilidade

Esta seção é destinada ao estudo da escalabilidade dos algoritmos *kDCI++* e *kDCI-3* considerando-se o tempo de execução da terceira iteração por eles obtido.

Primeiramente, os algoritmos são avaliados de acordo com o crescimento do número de transações. Para estes experimentos são consideradas quatro bases de dados sintéticas construídas com a utilização do gerador de dados da IBM *Almaden*: *T20I10N1KP5KC0.25D100K*, *T20I10N1KP5KC0.25D200K*, *T20I10N1KP5KC0.25D300K* e *T20I10N1KP5KC0.25D400K*. Em seguida, o mesmo estudo é feito à medida que o tamanho médio das transações aumenta. Para estes experimentos as bases de dados utilizadas são *T15I10N1KP5KC0.25D200K*, *T20I10N1KP5KC0.25D200K*, *T25I10N1KP5KC0.25D200K* e *T30I10N1KP5KC0.25D200K*, também construídas a partir do gerador da IBM. Os valores de suporte mínimo avaliados são 0,25% e 0,1%.

Variação do Número de Transações

Nesta subseção é apresentada uma avaliação do comportamento dos algoritmos *kDCI++* e *kDCI-3* quando o o número de transações da base de dados aumenta. Para cada um dos suportes avaliados, a Figura 4.4 apresenta, em gráficos separados, os tempos de execução da terceira iteração, obtidos pelos algoritmos *kDCI-3* e *kDCI++* sobre diferentes bases de dados. É importante ressaltar que, nestes experimentos, o algoritmo *kDCI++* utilizou a base de dados verticalizada nos dois valores de suporte mínimo avaliados.

Considerando-se os tempos de execução, observe que o desempenho do algoritmo *kDCI-3* é superior ao desempenho do algoritmo *kDCI++* para os dois valores de suporte mínimo avaliados. No entanto, ambos os algoritmos apresentaram-se escaláveis com o aumento do número de transações da base de dados.

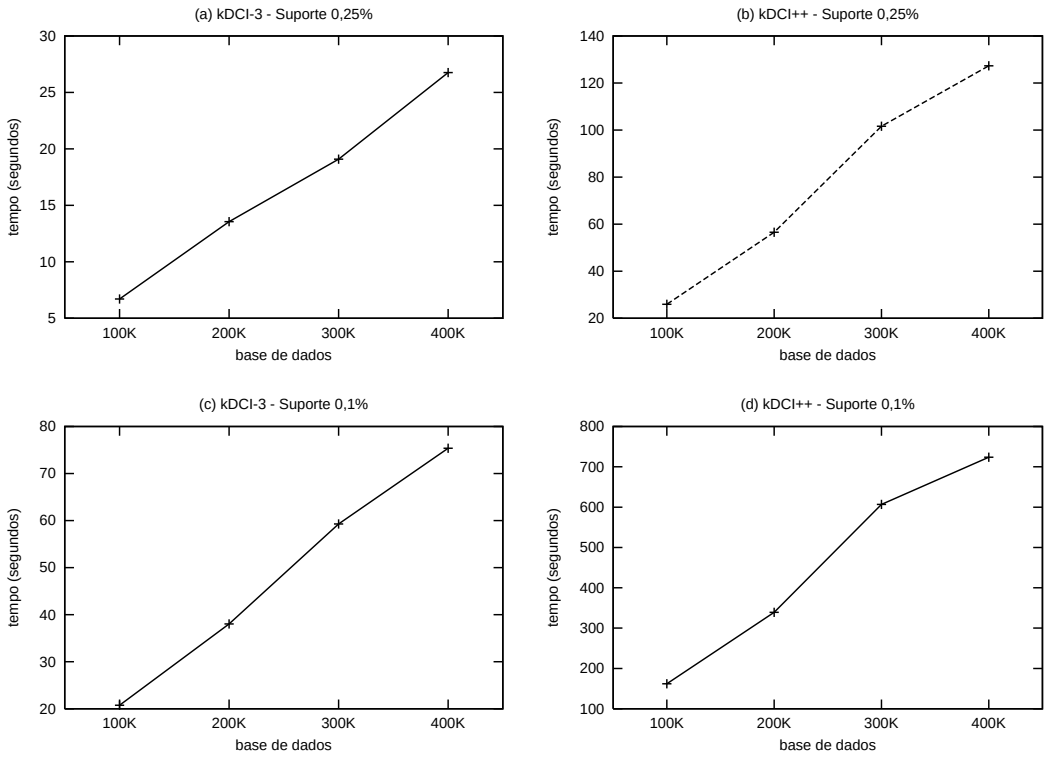


Figura 4.4: Avaliação da escalabilidade da terceira iteração dos algoritmos *kDCI++* e *kDCI-3* considerando-se o número de transações.

Variação do Tamanho Médio das Transações

Nesta subseção, a Figura 4.5 apresenta o comportamento do tempo de execução da terceira iteração obtido pelos algoritmos *kDCI-3* e *kDCI++*, em gráficos separados, de acordo com o crescimento do tamanho médio das transações presentes na base de dados. Assim como nos experimentos da subseção anterior, a base de dados verticalizada utilizada pelo algoritmo *kDCI++* já se encontra em memória principal nos dois suportes avaliados.

Da mesma forma que na subseção anterior, o desempenho do algoritmo *kDCI-3*, considerando-se os tempos de execução obtidos, foi superior ao desempenho do algoritmo *kDCI++* para os dois valores de suporte avaliados. No entanto, o comportamento dos algoritmos em relação ao crescimento do tamanho médio das transações foi bastante semelhante.

Para estes experimentos, apesar dos tempos de execução obtidos pelos al-

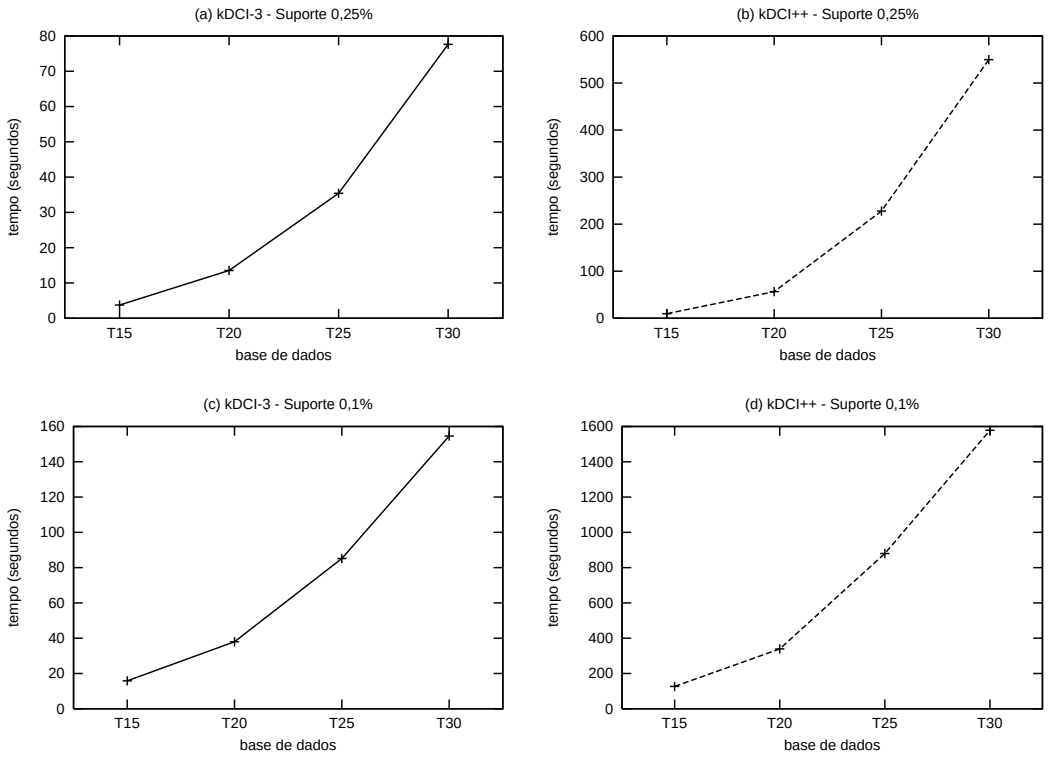


Figura 4.5: Avaliação da escalabilidade da terceira iteração dos algoritmos *kDCI++* e *kDCI-3* considerando-se o tamanho médio das transações.

goritmos crescerem continuamente com o aumento do tamanho médio das transações, este crescimento não apresenta um comportamento linear. Para o algoritmo *kDCI++*, como o custo computacional da terceira iteração é dependente do número de conjuntos candidatos avaliados, este comportamento pode ser justificado pelo crescimento também não linear do número de conjuntos candidatos à medida que o tamanho médio das transações aumenta. Considere, por exemplo, o suporte 0,25%. Conforme apresentado no Apêndice A, a base de dados *T20* gera, aproximadamente, 264.000 candidatos a mais que a base de dados *T15*, enquanto a base de dados *T25* conta 888.000 candidatos a mais que a base de dados *T20*, aproximadamente. Já a base de dados *T30* gera, aproximadamente, 1.585.000 candidatos a mais que a base de dados *T25*. Observe que no gráfico correspondente ao suporte 0,25%, o comportamento dos algoritmos acompanha o crescimento do número de candidatos em cada base de dados avaliada. O mesmo ocorre para o suporte 0,1%.

No algoritmo *kDCI-3*, conforme observado na subseção anterior, a terceira

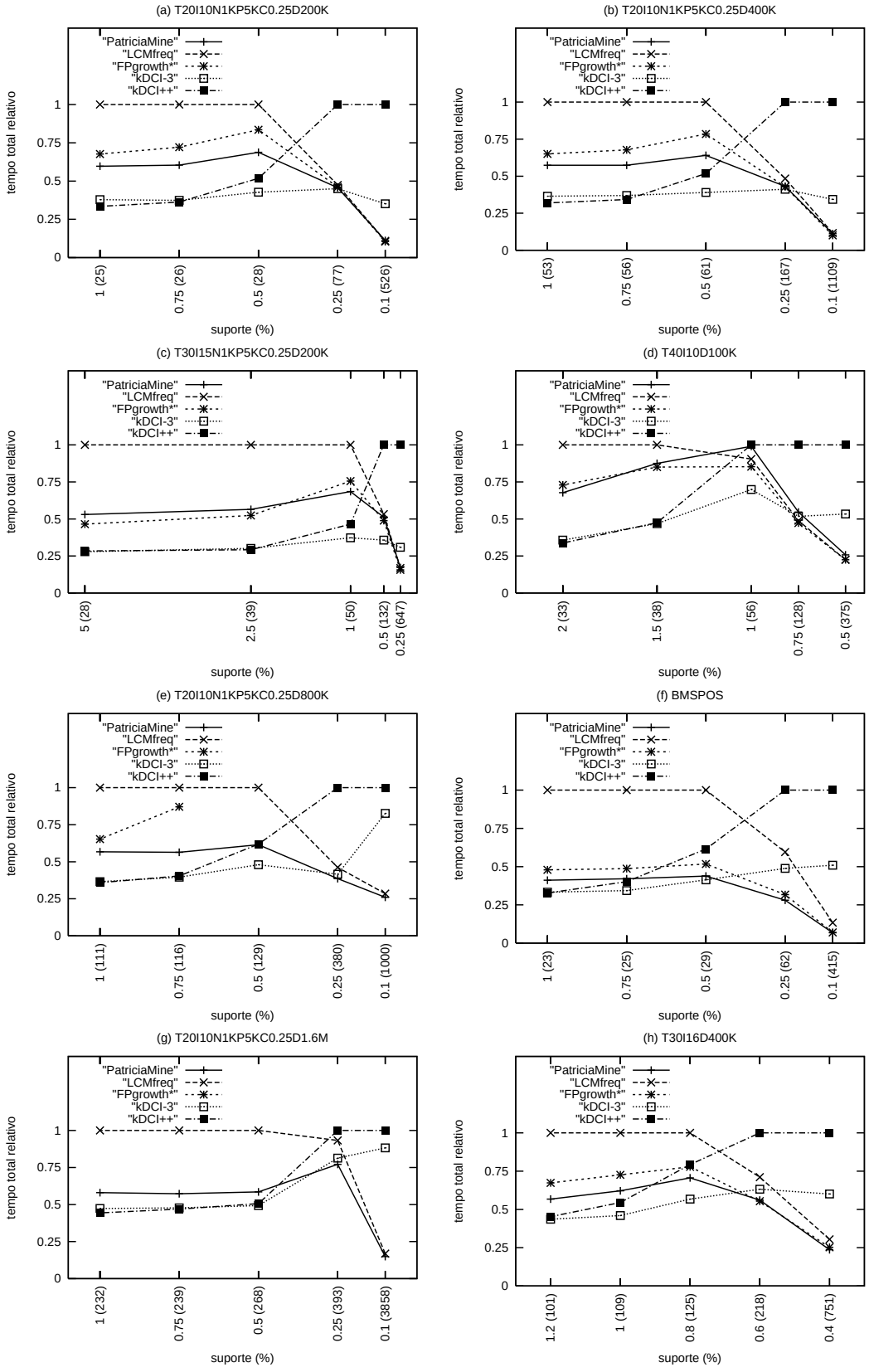
iteração é realizada a partir da contagem dos conjuntos de itens de tamanho 3 presentes nas transações da base de dados considerada. Desta forma, o comportamento não linear do algoritmo *kDCI-3* pode ser explicado pelo crescimento do número destes conjuntos à medida que aumenta o tamanho médio das transações.

4.3 Análise Comparativa entre os Principais Algoritmos

Nesta seção, são apresentados os resultados computacionais obtidos pelo *kDCI-3* e pelos principais algoritmos para extração de conjuntos freqüentes, segundo a avaliação do *Workshop FIMI'03*. São eles: *kDCI++*, *PatriciaMine*, *FPgrowth** e *LCMfreq*. Neste *workshop*, os algoritmos *kDCI++* e *PatriciaMine* obtiveram os melhores desempenhos para suportes altos, enquanto os algoritmos *FPgrowth** e *LCMfreq* apresentaram os melhores desempenhos considerando-se valores de suporte mínimo baixos. Os experimentos do algoritmo *kDCI++* foram realizados utilizando-se o código disponível na página do algoritmo *DCI*. Para os experimentos com os algoritmos *PatriciaMine*, *FPgrowth** e *LCMfreq*, foram utilizados os códigos disponíveis na página do *Workshop FIMI'03*.

Para cada uma das bases de dados utilizadas nos experimentos, a Figura 4.6 apresenta os tempos de execução relativos obtidos por cada um dos algoritmos, considerando-se os mesmos valores de suporte mínimo avaliados nas seções anteriores. Nestes gráficos, o valor 1 no eixo y representa o tempo relativo de execução do algoritmo que obteve o pior desempenho. No eixo x, o valor entre parênteses representa o tempo de execução absoluto obtido pelo algoritmo de pior desempenho.

Nas bases de dados dos gráficos (a), (b), (c), (d) e (f), os algoritmos *kDCI++* e *kDCI-3* apresentam o melhor desempenho para os suportes mais altos, com uma vantagem muito pequena para o algoritmo *kDCI++* em alguns destes valores de suporte. Tudo indica que a razão do baixo desempenho dos algoritmos *PatriciaMine*, *FPgrowth** e *LCMfreq*, para os suportes altos, seja devido ao tempo gasto


(a) T20I10N1KP5KC0.25D200K
(b) T20I10N1KP5KC0.25D400K
(c) T30I15N1KP5KC0.25D200K
(d) T40I10D100K
(e) T20I10N1KP5KC0.25D800K
(f) BMSPOS
(g) T20I10N1KP5KC0.25D1.6M
(h) T30I16D400K
"PatriciaMine"
"LCMfreq"
"FPgrowth*"
"kDCI-3"
"kDCI++"
tempo total relativo
suporte (%)
1 (25)
0.75 (26)
0.5 (28)
0.25 (77)
0.1 (526)
1 (53)
0.75 (56)
0.5 (61)
0.25 (167)
0.1 (1109)
5 (28)
2.5 (39)
1 (50)
0.5 (132)
0.25 (647)
2 (33)
1.5 (38)
1 (56)
0.75 (128)
0.5 (375)
1 (111)
0.75 (116)
0.5 (129)
0.25 (380)
0.1 (1000)
1 (23)
0.75 (25)
0.5 (29)
0.25 (62)
0.1 (415)
1 (232)
0.75 (239)
0.5 (268)
0.25 (393)
0.1 (3858)
1.2 (101)
1 (109)
0.8 (125)
0.6 (218)
0.4 (751)

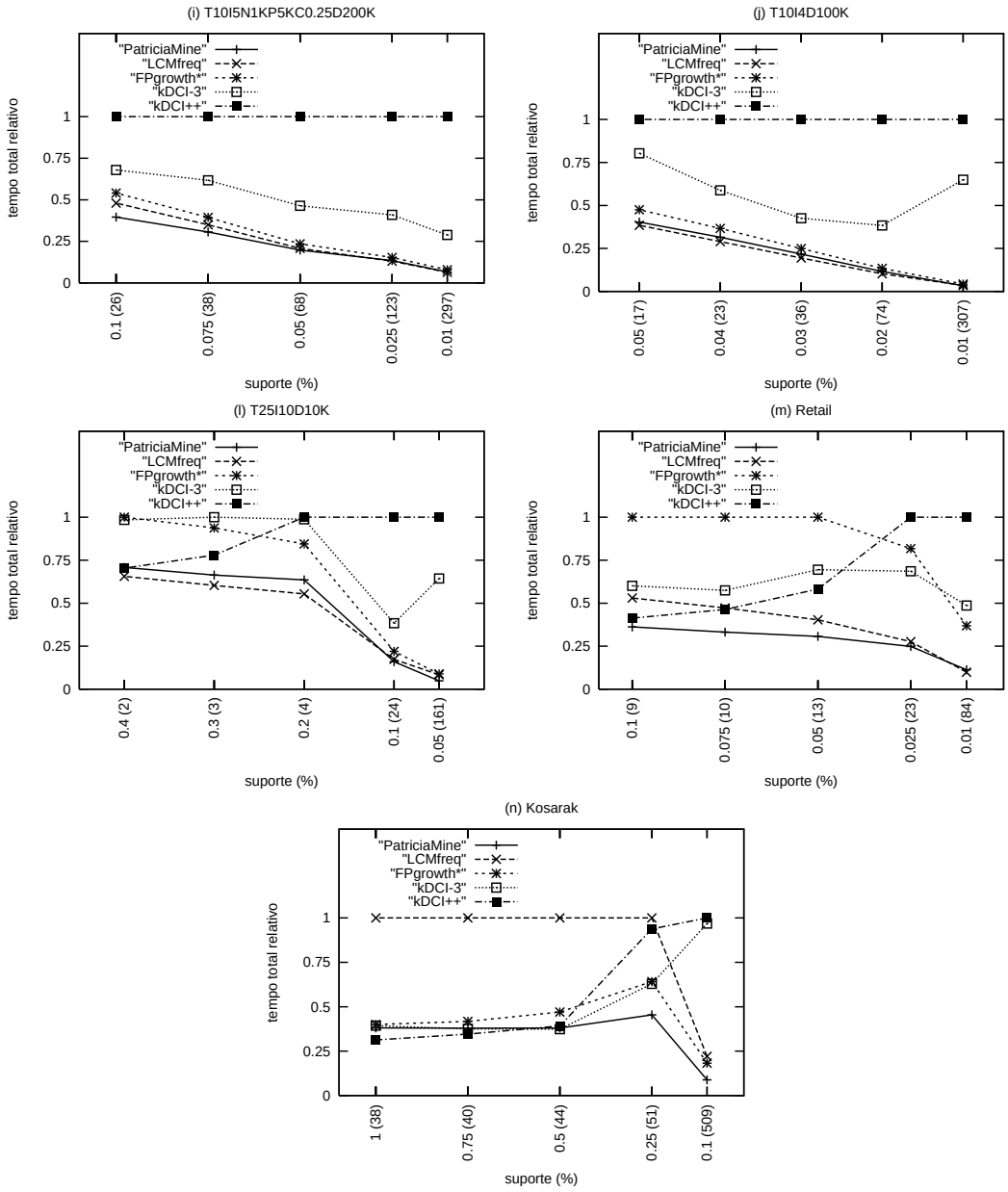


Figura 4.6: Avaliação de desempenho do tempo total de execução relativo do algoritmo *kDCI-3* e dos principais algoritmos.

na construção das estruturas *PatriciaTrie*, *FP-tree* e *G*, respectivamente, no início de suas execuções. Por exemplo, para a base de dados do gráfico (a), o algoritmo *FPgrowth** leva, em média, 89% do tempo total de execução na construção da estrutura *FP-tree*. Da mesma forma, o algoritmo *PatriciaMine* gasta, em média, 51% e o algoritmo *LCMfreq*, 32% do tempo total de execução na construção de suas estruturas iniciais. Para os suportes mínimos mais baixos, por exigirem a contagem de muitos conjuntos candidatos, os algoritmos *kDCI++* e *kDCI-3* apresentam desempenhos inferiores aos demais algoritmos. Além do fato de não gerarem nem contarem conjuntos candidatos, nos algoritmos *PatriciaMine*, *FPgrowth** e *LCMfreq* a base de dados se encontra em memória principal desde o início do processamento. Entretanto, a redução do custo computacional da terceira iteração obtida pelo algoritmo *kDCI-3* torna-o mais competitivo. Observe que para o suporte 0,25% nas bases de dados (a) e (b), 0,5% nas bases (c) e (f) e para os suportes 1% e 0,75% na base de dados (d), enquanto o algoritmo *kDCI++* representa a pior estratégia ou a segunda pior, o algoritmo *kDCI-3* apresenta o melhor desempenho ou aproxima-se, significativamente, do melhor algoritmo. Nas bases de dados (e), (g), (h) e (n), a razão do baixo desempenho dos algoritmos *PatriciaMine*, *FPgrowth** e *LCMfreq* para suportes altos e do alto desempenho para suportes baixos pode ser explicada pelos mesmos motivos descritos anteriormente.

Nas bases de dados dos gráficos (e), (g) e (n), os algoritmos *kDCI++* e *kDCI-3* apresentam desempenhos bastante semelhantes para os suportes altos. Para os suportes mais baixos 0,5% e 0,25% na base de dados (e) e 0,25% nas bases de dados (g) e (n), o algoritmo *kDCI-3* supera o baixo desempenho obtido pelo algoritmo *kDCI++*, tornado-o o melhor ou o segundo melhor algoritmo. Já para o suporte 0,1%, conforme discutido na Seção 4.2.1, os desempenhos dos algoritmos tornam-se semelhantes novamente, enquanto os demais algoritmos obtêm os melhores desempenhos. Vale observar que sobre a base de dados (e), as execuções do algoritmo *FP-growth* para os suportes 0,5%, 0,25% e 0,1% duraram, em média, três horas e sobre a base de dados (g), o algoritmo não conseguiu completar suas execuções após dez horas, com exceção para o suporte 1% cuja execução durou seis horas e meia (estas execuções não foram consideradas nos gráficos correspondentes para

não descaracterizar o comportamento dos demais algoritmos). O motivo de tal comportamento pode ser explicado pela pouca compactação da base de dados atingida pela *FP-tree* nestes experimentos. Tudo indica que a ausência de uma estratégia de gerenciamento de memória foi a razão da queda de desempenho do algoritmo.

No gráfico da base de dados (h), o algoritmo *kDCI-3* supera o bom desempenho obtido pelo algoritmo *kDCI++* para suportes altos. Observe que para o suporte 0,8%, o desempenho do algoritmo *kDCI++* cai, mas o algoritmo *kDCI-3* permanece em primeiro lugar. Para o suporte 0,6%, enquanto o algoritmo *kDCI++* apresenta o pior desempenho, o algoritmo *kDCI-3* mantém o desempenho alcançado para os suportes mais altos. Finalmente, para o suporte 0,4%, apesar do desempenho do algoritmo *kDCI-3* ter sido superado pelos desempenhos dos algoritmos *PatriciaMine*, *FPgrowth** e *LCMfreq*, este torna o algoritmo *kDCI++* mais competitivo.

Nas bases de dados nos gráficos (i) e (j), o algoritmo *kDCI++* apresenta o pior desempenho para todos os valores de suporte mínimo, enquanto o algoritmo *kDCI-3* torna-o mais competitivo. Nestes experimentos, os algoritmos *PatriciaMine*, *FPgrowth** e *LCMfreq* apresentaram os melhores desempenhos já a partir dos suportes mais altos. Como estes algoritmos extraem os conjuntos freqüentes a partir de representações da base de dados, construídas em memória principal no início de seus processamentos, a razão de tal comportamento parece se justificar pelo tamanho relativamente pequeno destas bases de dados. Enquanto as bases de dados dos gráficos (e) e (g) têm, respectivamente, 60 e 118 *megabytes*, a base de dados (i) tem 7,6 *megabytes* e a base (j) tem apenas 3,9 *megabytes*.

Finalmente, nos experimentos sobre as bases de dados (l) e (m), apesar dos tempos absolutos terem sido pequenos, pode-se observar que o algoritmo *kDCI-3* obteve um desempenho inferior ao do algoritmo *kDCI++* para os suportes mais altos. Nos experimentos sobre a base de dados (l), este comportamento pode ser justificado pelo tamanho relativamente pequeno das listas de bits que compõem a base de dados verticalizada nas execuções do algoritmo *kDCI++*. Sobre a base de dados (m), o bom desempenho do algoritmo *kDCI++* pode ser explicado pelo número pouco expressivo de candidatos de tamanho 3 avaliados nestas execuções. Além disso, o tamanho das

listas de bits utilizadas na contagem dos candidatos não foi tão significativo quanto nas primeiras bases de dados avaliadas. Entretanto, para os suportes mais baixos, enquanto o algoritmo *kDCI++* apresenta o pior desempenho, o algoritmo *kDCI-3* torna-o mais competitivo. Conforme observado na Seção 4.2.1, a razão da queda da redução alcançada pelo algoritmo *kDCI-3* para o suporte 0,05% (base de dados (l)) se deve ao fato das iterações posteriores à terceira iteração também apresentarem um custo computacional significativo para este valor de suporte.

4.4 Resumo dos Resultados

Nesta seção, é apresentado um resumo das avaliações de desempenho realizadas neste capítulo. As três primeiras subseções são dedicadas às avaliações de desempenho do algoritmo *kDCI-3*, considerando-se tempo total de execução relativo, memória consumida durante a terceira iteração e escalabilidade. Estas avaliações foram realizadas comparando-se os resultados obtidos pelos algoritmos *kDCI-3* e *kDCI++* sobre diferentes combinações de bases de dados e valores de suporte mínimo. A última subseção apresenta um resumo da análise comparativa realizada entre o algoritmo *kDCI-3* e os principais algoritmos apresentados no *Workshop FIMI'03*. Esta última análise foi realizada considerando-se o tempo total de execução relativo obtido pelos algoritmos sobre as mesmas combinações de bases de dados e valores de suporte mínimo utilizadas nas avaliações descritas anteriormente.

4.4.1 Tempo de Execução

Quando valores de suporte mínimo críticos (mais baixos) foram considerados, a técnica de contagem de candidatos de tamanho 3 adotada pelo algoritmo *kDCI-3* reduziu o tempo total de execução obtido pelo algoritmo *kDCI++* sobre todas as bases de dados utilizadas nos experimentos.

Para os suportes mais baixos, nos experimentos em que o algoritmo *kDCI++* fez uso da base de dados verticalizada para a contagem dos candidatos de tamanho

3, além do grande número de candidatos a serem avaliados, as listas de bits que formavam a base de dados verticalizada eram relativamente grandes. Desta forma, tudo indica que o mau desempenho do algoritmo *kDCI++* para os suportes baixos seja devido ao número elevado de interseções de listas de bits realizadas durante a contagem de um número significativamente alto de candidatos. Quando a representação verticalizada da base de dados não foi utilizada durante a terceira iteração, o mau desempenho alcançado pelo algoritmo *kDCI++* pode ser justificado pela busca seqüencial adotada na contagem dos candidatos.

Nos experimentos sobre as bases de dados *T20I10N1KP5KC0.25D800K*, *T20I10N1KP5KC0.25D1.6M* e *Kosarak*, notou-se que, para o menor suporte mínimo avaliado, o percentual de redução do tempo total de execução alcançado pelo algoritmo *kDCI-3* não foi tão significativo quanto para os demais suportes, devido ao fato de as iterações posteriores à terceira iteração também apresentarem um alto custo computacional.

Para os suportes mais altos, com exceção das bases de dados *T30I16D400K*, *T10I5N1KP5KC0.25D200K* e *T10I4D100K*, devido ao número pouco significativo de candidatos de tamanho 3 ou devido ao tamanho pequeno da base de dados verticalizada (listas de bits pequenas) construída pelo algoritmo *kDCI++*, o desempenho do algoritmo *kDCI-3* foi ligeiramente inferior ou semelhante ao do algoritmo *kDCI++*. Cabe ressaltar que os tempos absolutos destas execuções foram pouco expressivos.

4.4.2 Memória

Em relação à memória consumida pelos algoritmos *kDCI++* e *kDCI-3* durante a terceira iteração, observou-se que, quando a representação verticalizada da base foi utilizada na contagem de candidatos de tamanho 3, o algoritmo *kDCI++* consumiu uma quantidade de memória significativamente maior do que a consumida pelo algoritmo *kDCI-3*, com exceção das bases de dados *T10I5N1KP5KC0.25D200K*, *T10I4D100K*, *T25I10D10K* e *Retail*.

Sobre as bases de dados *T10I5N1KP5KC0.25D200K* e *T10I4D100K*, conside-

rando-se apenas o valor mais baixo de suporte mínimo, a memória consumida pelo *kDCI-3* ultrapassou a consumida pelo *kDCI++* devido ao aumento significativo de candidatos de tamanho 3 a serem representados pela estrutura de dados adotada pelo *kDCI-3*.

Nos experimentos realizados sobre as bases de dados *T25I10D10K* e *Retail* (com esta última apenas para os quatro primeiros suportes mínimos avaliados), a base de dados verticalizada construída pelo algoritmo *kDCI++* durante a terceira iteração era relativamente pequena (listas de bits pequenas) em comparação com as bases construídas nos demais experimentos. Desta forma, o *kDCI++* apresentou-se mais eficiente que o algoritmo *kDCI-3*.

Quando o algoritmo *kDCI++* não fez uso da base de dados verticalizada durante a terceira iteração, os desempenhos alcançados pelos algoritmos *kDCI++* e *kDCI-3* foram bastante semelhantes, com exceção do experimento sobre a base de dados *Retail* e suporte 0,01%. Neste caso, o algoritmo *kDCI-3* obteve pior desempenho devido à grande quantidade de candidatos de tamanho 2 a serem representados na estrutura de dados adotada durante a terceira iteração. Vale observar porém que, neste experimento, apesar de consumir mais memória, a terceira iteração do algoritmo *kDCI-3* foi mais rápida que a terceira iteração do algoritmo *kDCI++*.

4.4.3 Escalabilidade

Para o estudo da escalabilidade dos algoritmos *kDCI++* e *kDCI-3*, considerou-se o tempo de execução da terceira iteração por eles obtido de acordo com o crescimento do número de transações e de acordo com o crescimento do tamanho médio das transações da base de dados.

Em relação ao crescimento do número de transações, ambos os algoritmos apresentaram-se escaláveis. Da análise do comportamento dos algoritmos de acordo com o crescimento do tamanho médio das transações, observou-se que apesar dos tempos de execução de ambos os algoritmos crescerem continuamente, este crescimento não apresentou um comportamento linear. Para o algoritmo *kDCI++*, como

o custo computacional da terceira iteração é dependente do número de candidatos a serem avaliados, tal comportamento pode ser justificado pelo crescimento não proporcional do número de candidatos à medida que o tamanho médio das transações aumenta. Já no algoritmo *kDCI-3*, a terceira iteração é realizada a partir da contagem dos conjuntos de itens de tamanho 3 presentes nas transações. O comportamento apresentado pelo algoritmo pode ser justificado pelo crescimento não proporcional destes conjuntos em relação ao crescimento do tamanho médio das transações.

4.4.4 Comparação entre os Principais Algoritmos

Para os valores de suporte mínimo mais críticos, observou-se que o *kDCI-3* tornou o algoritmo *kDCI++* mais competitivo, com exceção do último suporte mínimo avaliado sobre as bases *T20I10N1KP5KC0.25D800K*, *T20I10N1KP5KC0.25D-1.6M* e *Kosarak*. Nestas execuções, o percentual de redução do tempo total de execução alcançado pelo *kDCI-3* não foi tão significativo quanto nas demais bases de dados devido ao custo computacional também alto das iterações posteriores a terceira iteração.

Para algumas combinações de bases de dados e suportes mínimos avaliados, enquanto o algoritmo *kDCI++* representou a pior estratégia ou a segunda pior, o algoritmo *kDCI-3* apresentou o melhor desempenho ou aproximou-se, significativamente, do melhor algoritmo.

Para os suportes mínimos mais altos, com exceção das bases de dados *T30I16D400K*, *T10I5N1KP5KC0.25D200K* e *T10I4D100K*, sobre as quais o *kDCI-3* superou o desempenho do *kDCI++* para todos os valores de suporte avaliados, o algoritmo *kDCI-3* manteve o bom desempenho alcançado pelo algoritmo *kDCI++* ou obteve desempenho ligeiramente inferior ao do mesmo. Neste último caso, apesar de os tempos de execução absolutos serem pouco expressivos, o número pouco significativo de candidatos de tamanho 3 ou o tamanho pequeno da base de dados verticalizada (listas de bits pequenas) construída durante a terceira iteração

do *kDCI++* contribuiu para seu melhor desempenho.

Capítulo 5

Conclusões

Experimentos computacionais realizados neste trabalho mostraram que quando valores baixos de suporte mínimo são considerados sobre bases de dados esparsas, a terceira iteração do algoritmo *kDCI++* (recentemente considerado um dos principais algoritmos para a extração de conjuntos freqüentes) apresenta um alto custo computacional. Este comportamento pode ser justificado pela grande quantidade de candidatos de tamanho 3 que serão avaliados pelo algoritmo, nestas condições. No algoritmo *kDCI++*, a contagem do suporte dos conjuntos candidatos de tamanho 3 é feita ou através de estruturas de dados específicas – quando a base de dados verticalizada ainda não se encontra em memória principal – ou através de interseções de listas de bits – quando é utilizada a representação verticalizada da base de dados. Neste contexto, a contribuição do presente trabalho foi a proposta de uma adaptação do algoritmo *kDCI++*, chamada algoritmo *kDCI-3*, em que a estratégia adotada durante a terceira iteração possibilita uma contagem mais eficiente dos conjuntos candidatos de tamanho 3.

Os resultados obtidos sobre treze bases de dados esparsas (sintéticas e reais) e considerando-se valores de suporte mínimo baixos mostraram que a técnica de contagem adotada pelo algoritmo *kDCI-3* reduz significativamente o tempo da terceira iteração do algoritmo *kDCI++*. Nos experimentos realizados, o tempo da

terceira iteração do algoritmo *kDCI-3* foi, em média, 27% do tempo da terceira iteração do algoritmo *kDCI++*. Da mesma forma, o tempo total de execução do algoritmo *kDCI-3* representou, em média, 53% do tempo total de execução do algoritmo *kDCI++*.

Entretanto, quando um número pouco significativo de candidatos de tamanho 3 eram avaliados (quando suportes mínimos altos foram considerados), o desempenho obtido pelo algoritmo *kDCI++* foi ligeiramente superior ao do algoritmo *kDCI-3*, embora os tempos de execução tenham sido muito pouco expressivos nestes casos. O mesmo ocorreu quando o tamanho das listas de bits da base verticalizada era relativamente pequeno. No algoritmo *kDCI++*, quando a base de dados verticalizada, composta por lista de bits, está em memória principal, o suporte dos conjuntos candidatos é calculado através de interseções de listas de bits. Assim sendo, quanto menor o número de candidatos gerados, menor o número de interseções realizadas. Da mesma forma, se estas listas de bits forem pequenas, o número de interseções realizadas também será pequeno, o que torna a iteração menos custosa computacionalmente. De fato, conforme observado pelos autores do algoritmo *DCI* (base para a proposta do algoritmo *kDCI++*) em [18], o custo computacional da fase de interseção é proporcional ao número de interseções (*and operations*) necessárias para encontrar o suporte dos conjuntos candidatos. O número de interseções dependerá do número de candidatos gerados durante a iteração e do tamanho das listas de bits que compõem a base de dados verticalizada [18].

A seguir, são discutidas algumas questões adicionais e são apresentadas sugestões para trabalhos futuros.

Um Modelo de Custo

De modo a permitir que o algoritmo *kDCI-3* decida qual a melhor estratégia a ser adotada para a contagem dos conjuntos candidatos de tamanho 3, este deveria ser capaz de estimar os custos computacionais das possíveis estratégias baseando-se nas características da base de dados.

Durante o desenvolvimento deste trabalho, foram avaliadas, preliminarmente, algumas técnicas para se estimar o custo da estratégia utilizada pelo algoritmo *kDCI-3* e da estratégia do algoritmo *kDCI++* que utiliza a base de dados verticalizada. Estas idéias preliminares são discutidas a seguir.

Algoritmo *kDCI-3* Para o algoritmo *kDCI-3*, concluiu-se que o custo computacional de sua terceira iteração é proporcional ao número total de conjuntos de tamanho 3 presentes nas transações da base de dados, já que a contagem dos candidatos de tamanho 3 é feita a partir destes conjuntos. Observe que este número é proporcional ao tamanho médio das transações da base de dados.

Desta forma, o custo computacional (tempo em segundos) da terceira iteração do algoritmo *kDCI-3* pode ser calculado, aproximadamente, pela fórmula:

$$N_c * t_c, \tag{5.1}$$

onde N_c é o número total de conjuntos de tamanho 3 presentes nas transações da base de dados e t_c é o tempo médio, em segundos, gasto no acesso às estruturas T_3' e C' durante a avaliação de um único conjunto de tamanho 3.

Neste contexto, elaborou-se uma estratégia para estimar o custo computacional da terceira iteração do algoritmo *kDCI-3* baseando-se no tempo de processamento médio, em segundos, gasto na contagem dos conjuntos de tamanho 3 presentes em 1% das transações da base de dados.

No início da fase de contagem, o algoritmo *kDCI-3* percorre a base de dados a fim de calcular N_c. À medida que a base de dados é lida, 1% das transações da base de dados é selecionado (a cada 100 transações, uma é selecionada) e o tempo gasto no acesso às estruturas T_3' e C' pelos conjuntos de itens presentes nestas transações é totalizado. Feita a leitura completa da base de dados, o tempo t_c, gasto para a contagem de um único conjunto de tamanho 3, é calculado a partir da média do tempo gasto pelos conjuntos presentes nas transações selecionadas e, em seguida, o custo total da terceira iteração é estimado utilizando-se a Fórmula 5.1.

Os resultados obtidos por esta estratégia foram muito próximos do verdadeiro custo da terceira iteração do algoritmo *kDCI-3* sobre todas as bases de dados e suportes mínimos avaliados. Apesar de t_c ter sido estimado a partir de apenas 1% das transações, N_c era o número exato de conjuntos de tamanho 3 presentes na base de dados. Entretanto, sobre bases de dados muito grandes, o custo computacional desta estratégia foi relativamente alto, o que pode ser explicado pelo fato de ser necessária a leitura completa da base de dados.

Assim sendo, a fim de diminuir o custo da primeira estratégia avaliada, tentou-se estimar N_c a partir do número médio de conjuntos de tamanho 3 presentes no primeiro 1% de transações, evitando, assim, a leitura completa da base de dados. O valor de t_c também era estimado baseando-se no tempo médio, em segundos, gasto na contagem dos conjuntos de tamanho 3 presentes nas mesmas transações. Para alguns dos experimentos realizados, os resultados obtidos, embora muito próximos do verdadeiro custo do algoritmo *kDCI-3*, não retrataram fielmente a realidade. De fato, as características das primeiras transações podem não retratar efetivamente as características das demais transações da base de dados. Por exemplo, as primeiras transações podem ser muito curtas ou muito longas, o que pode ter influenciado de maneira incorreta a estimativa de custo.

É apontado, então, como trabalho futuro o desenvolvimento de uma nova estratégia. Nela, os valores de N_c e t_c, assim como na segunda estratégia apresentada, seriam estimados a partir de um subconjunto de transações (evitando a leitura completa da base de dados). No entanto, assim como na primeira estratégia apresentada, as transações consideradas na estimativa seriam selecionadas ao longo da base de dados, não consecutivamente, o que retrataria mais fielmente as características das transações presentes na base de dados. Um outro estudo apontado é encontrar o percentual x de transações ideal a ser considerado para a estimativa. É fato que quanto maior o número x, mais próximo da realidade é a estimativa. Entretanto, deve-se considerar que o tempo gasto pela estratégia não poderá onerar o tempo total do algoritmo.

Algoritmo *kDCI++* O custo computacional da estratégia do algoritmo *kDCI++* em que a contagem dos candidatos é feita a partir da base verticalizada é proporcional ao número de interseções de listas de bits realizadas. Observe que o número destas interseções é dependente do número de candidatos que serão avaliados. Além disso, deve-se considerar que, durante a terceira iteração, se os conjuntos candidatos de tamanho 3 c_1 e c_2 têm o mesmo prefixo de tamanho 2, para determinar o suporte de c_2, a primeira interseção não precisará ser encontrada, já que o resultado obtido durante a contagem do suporte do candidato c_1 estará armazenada na estrutura de dados *Cache* (Subseção 2.2.2). Parte dos conjuntos candidatos terão, então, o suporte calculado a partir de duas interseções de listas de bits e a outra parte, a partir de apenas uma única interseção.

Desta forma, o custo computacional (tempo em segundos) da terceira iteração do algoritmo *kDCI++*, quando a base verticalizada é utilizada, pode ser calculado, aproximadamente, pela fórmula:

$$(N_{c_1} * t_{i_1}) + (N_{c_2} * t_{i_2}), \tag{5.2}$$

onde N_{c_1} é o número de conjuntos candidatos cujo suporte é calculado a partir de duas interseções, N_{c_2} é o número de conjuntos candidatos cujo suporte é calculado a partir de uma única interseção, t_{i_1} é o tempo médio, em segundos, gasto para a contagem do suporte a partir de duas interseções e t_{i_2}, tempo médio, em segundos, gasto para a contagem do suporte a partir de uma única interseção (considerando-se que o resultado da primeira interseção estará armazenado na estrutura de dados *Cache*). Vale ressaltar que t_{i_1} e t_{i_2} são dependentes do tamanho da lista de bits construída durante a execução.

No início da terceira iteração, N_{c_1} e N_{c_2} são facilmente calculados a partir da combinação dos conjuntos freqüentes de tamanho 2. Tentou-se, então, estimar t_{i_1} baseando-se no tempo médio, em segundos, gasto para encontrar o suporte do primeiro 1% de candidatos considerados no cálculo de N_{c_1}, aqui chamado de conjunto C_{t_1}. Considere $|C_{t_1}| = y$. Na estimativa de t_{i_2}, para cada conjunto candidato c de C_{t_1}, foram considerados z conjuntos candidatos com o mesmo prefixo de tamanho 2

de c, de forma que o número de conjuntos considerados $y * z$ fosse equivalente a 1% dos conjuntos considerados no cálculo de N_{c_2}. O valor de t_{i_2} foi, então, estimado baseando-se no tempo médio, em segundos, gasto para encontrar o suporte dos $y * z$ conjuntos candidatos selecionados.

O resultado obtido por esta estimativa não retratou a realidade para a maioria das bases de dados e suportes mínimos avaliados. Tal comportamento pode ser explicado pelo fato do algoritmo *kDCI++* fazer uso de otimizações ao longo da contagem de todos os conjuntos candidatos. Como as estimativas de t_{i_1} e t_{i_2} consideraram os benefícios destas otimizações sobre a contagem de apenas uma primeira fração de candidatos, estes resultados podem ter influenciado de maneira incorreta a estimativa total de custo. Na realidade, não se sabe o quanto foram vantajosas as otimizações ao longo da contagem dos demais candidatos.

Aponta-se, então, como trabalho futuro, selecionar aleatoriamente os candidatos que irão participar da estimativa, não consecutivamente, o que poderia retratar mais fielmente a distribuição dos candidatos gerados durante a execução, assim como as vantagens trazidas pelas otimizações ao longo da fase de interseção. Um outro estudo proposto é encontrar o percentual de candidatos ideal a ser considerado para a estimativa, lembrando que o custo computacional desta estimativa não deverá onerar o tempo total do algoritmo.

Apesar de o algoritmo *kDCI-3* não ter apresentado desempenho inferior ao do algoritmo *kDCI++* quando a contagem dos candidatos foi feita a partir de um algoritmo de busca (e não através de interseções de listas de bits), um outro estudo apontado como trabalho futuro é investigar sobre quais condições esta técnica pode ser mais eficiente do que a contagem direta adotada pelo algoritmo *kDCI-3*. Este estudo poderia basear-se, inicialmente, no número médio de conjuntos candidatos de tamanho 2 presentes nas transações e no tamanho médio das seções do vetor S que serão percorridas seqüencialmente. Podendo estimar o custo computacional desta estratégia e das demais discutidas acima, o algoritmo *kDCI-3* seria capaz de optar pela melhor estratégia a ser adotada na contagem dos candidatos de tamanho 3.

Iterações Posteriores à Terceira Iteração

Da análise dos resultados computacionais apresentados neste trabalho, constatou-se que sobre bases de dados esparsas e baixos valores de suporte mínimo, as iterações posteriores à terceira iteração também podem apresentar um alto custo computacional no algoritmo *kDCI++*. Por esta razão, um outro trabalho a ser realizado é a avaliação do uso da técnica adotada pelo algoritmo *kDCI-3* nas demais iterações. Uma possível extensão desta técnica é apresentada em [26].

Gerenciamento de Memória

Em alguns dos experimentos realizados neste trabalho, foi verificado que o número de conjuntos candidatos de tamanho 2 também poderá ser significativamente grande quando valores de suporte mínimo baixos são considerados. Foi o caso, por exemplo, da base de dados *Retail*, cujo número de candidatos de tamanho 2 é de 39.769.821 conjuntos para o suporte 0,01%. A fim de evitar problemas em relação à memória necessária para a contagem destes conjuntos, propõe-se, como trabalho futuro, o desenvolvimento de uma técnica de gerenciamento de memória para a segunda iteração do algoritmo *kDCI-3*.

Observou-se também que, devido à falta de uma estratégia de gerenciamento de memória sobre as estruturas de dados T_3, P, I, S e C, o algoritmo *kDCI++* teve algumas de suas iterações muito longas – devido ao uso de memória virtual – ou até mesmo abortadas – por necessitar de uma quantidade significativamente grande de memória. Desta forma, como as demais iterações do algoritmo *kDCI-3* (posteriores à terceira iteração) são equivalentes as iterações do algoritmo *kDCI++*, aponta-se, também como trabalho futuro, o desenvolvimento de uma técnica de gerenciamento de memória para as demais iterações do algoritmo *kDCI++*, quando estas forem executadas sem a utilização da base de dados verticalizada.

Apêndice A

Número de Conjuntos Candidatos de Tamanho 3 Gerados pelos Algoritmos *kDCI++* e *kDCI-3*

A Tabela A.1 apresenta o número de candidatos de tamanho 3 gerados pelos algoritmos *kDCI++* e *kDCI-3* em cada uma das combinações de bases de dados esparsas e valores de suporte mínimo utilizadas nos experimentos computacionais deste trabalho. Os dados apresentados nesta tabela têm como objetivo dar subsídios às avaliações de desempenho realizadas no Capítulo 4.

O símbolo * ao lado de alguns dos valores de suporte mínimo apresentados, indica que, para a combinação de base de dados e suporte mínimo correspondente, o número de candidatos gerados pelo algoritmo *kDCI++* e *kDCI-3* não foi o mesmo. Nestes casos, o primeiro valor apresentado se refere ao número de candidatos gerados pelo algoritmo *kDCI++* e o número após o símbolo \ representa o número de candidatos gerados pelo algoritmo *kDCI-3*. A diferença no número de candidatos gerados se deve ao fato de que, nestas execuções, o algoritmo *kDCI++* poda candidatos durante sua geração (a base verticalizada não estava em memória principal durante a contagem dos candidatos de tamanho 3). Desta forma, como o *kDCI-3*

não poda candidatos em nenhuma situação, o número de candidatos gerados por este algoritmo é maior do que o número de candidatos gerados pelo algoritmo *kDCI++*.

Base de Dados	Suportes (%)				
	Número de Candidatos				
T20I10N1KP5KC0.25D100K				**0,25**	**0,10**
				375.558	3.680.415
T20I10N1KP5KC0.25D200K	**1,0**	**0,75**	**0,5**	**0,25**	**0,10**
	364	2.555	23.182	330.317	3.187.477
T20I10N1KP5KC0.25D300K				**0,25**	**0,10**
				365.562	3.599.680
T20I10N1KP5KC0.25D400K	**1,0**	**0,75**	**0,5**	**0,25**	**0,10**
	364	2.542	22.908	326.726	3.170.943
T20I10N1KP5KC0.25D800K	**1,0**	**0,75**	**0,5**	**0,25**	**0,10***
	355	2.598	22.975	326.523	2.942.815\3.155.473
T20I10N1KP5KC0.25D1.6M	**1,0***	**0,75***	**0,5***	**0,25***	**0,10***
	37\45	826\1.068	14.283\16.458	331.193\367.588	3.375.811\3.595.217
T15I10N1KP5KC0.25D200K				**0,25**	**0,10**
				66.401	1.400.880
T25I10N1KP5KC0.25D200K				**0,25**	**0,10**
				1.218.448	7.502.092
T30I10N1KP5KC0.25D200K				**0,25**	**0,10**
				2.803.092	12.693.869
T10I5N1KP5KC0.25D200K	**0,10**	**0,075**	**0,05**	**0,025**	**0,01**
	98.892	220.518	558.140	2.158.294	10.424.407
T30I15N1KP5KC0.25D200K	**5,0**	**2,5**	**1,0**	**0,5**	**0,25**
	0	57	26.359	426.504	2.729.727
T25I10D10K	**0,40**	**0,30**	**0,20**	**0,10**	**0,05**
	27.257	63.460	423.650	8.707.152	44.171.824
T40I10D100K	**2,0**	**1,5**	**1,0**	**0,75**	**0,5**
	9.421	44.630	252.677	654.593	1.924.460
T10I4D100K	**0,05**	**0,04**	**0,03**	**0,02**	**0,01**
	411.711	743.647	1.498.335	3.559.392	11.051.970
T30I16D400K	**1,2**	**1,0**	**0,8**	**0,6**	**0,4**
	11.116	27.321	80.225	249.129	908.360
BMSPOS	**1,0**	**0,75**	**0,5**	**0,25**	**0,10**
	1.164	2.316	5.680	21.228	107.263
Kosarak	**1,0**	**0,75**	**0,5**	**0,25**	**0,10***
	194	372	897	4.631	65.027\65.797
Retail	**0,10**	**0,075**	**0,05**	**0,025**	**0,010***
	4.359	7.468	17.238	85.371	912.402\1.017.833

Tabela A.1: Número de candidatos gerados pelos algoritmos *kDCI++* e *kDCI-3* em cada uma das combinações de bases de dados e valores de suporte mínimo utilizados nos experimentos computacionais.

Referências Bibliográficas

[1] Agarwal, R., Aggarwal, C. C., e Prasad, V. Mining frequent patterns with counting inference. *ACM SIGKDD Explorations Newsletter 2*, 2 (2000), 66–75.

[2] Agarwal, R., Aggarwal, C. C., e Prasad, V. A tree projection algorithm for generation of frequent item sets. *Journal of Parallel and Distributed Computing 61*, 3 (2001), 350–371.

[3] Agrawal, R., Imielinski, T., e Srikant, R. Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1993), pp. 207–216.

[4] Agrawal, R., e Srikant, R. Fast algorithms for mining association rules. In *Proceedings of the the 20th VLDB International Conference on Very Large Databases* (1994), pp. 487–489.

[5] Bodon, F. A fast apriori implementation. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations* (2003).

[6] Brijs, T., Swinnen, G., Vanhoof, K., e Wets, G. Using association rules for products assortment decisions: A case of study. In *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (1999), pp. 254–260.

[7] Brin, S., Motwani, R., Ulman, J. D., e Tsur, S. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of the ACM*

SIGMOD International Conference on Management of Data (1997), pp. 255–264.

[8] Geurts, K., Wets, G., Brijs, T., e Vanhoof, K. Profiling high frequency accidents locations using association rules. In *Proceedings of the 82nd Annual Transportation Research Board* (2003), p. 18.

[9] Goethals, B., e Zaki, M. J. Advances in frequent itemset mining implementations: Introduction to fimi'03. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations* (2003).

[10] Grahne, G., e Zhu, J. Efficiently using prefix trees in mining frequent itemsets. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations* (2003).

[11] Han, E., Karypis, G., e Kumar., V. Scalable parallel data mining for association rules. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1997), pp. 277–288.

[12] Han, J., Pei, J., e Yin, Y. Mining frequent patterns without candidate generation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2000), pp. 1–12.

[13] Kosters, W. A., e Pijls, W. Apriori, a depth first implementation. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations* (2003).

[14] Liu, J., Pan, Y., Wang, K., e Han, J. Mining frequent item sets by opportunistic projection. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2002), pp. 229–238.

[15] Orlando, S., Palmerimi, P., e Perego., R. Enhancing the apriori algorithm for frequent set counting. In *Proceedings of the 3rd DAWAK International Conference on Data Warehousing and Knowledge Discovery* (2001), pp. 71–82.

[16] Orlando, S., Palmerimi, P., e Perego, R. Adaptive and resource–aware mining of frequent sets. In *Proceedings of the IEEE ICDM International Conference on Data Mining* (2002), pp. 338–345.

[17] Orlando, S., Palmerimi, P., Perego, R., Lucchese, C., e Silvestri, F. kdci: a multi–strategy algorithm for discovering frequent sets in large databases. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations* (2003).

[18] Orlando, S., Palmerini, P., e Perego, R. Dci: a hybrid algorithm for frequent set counting. Relatório Técnico CS-2001-9, Universitá Ca' Foscari di Venezia, 2001.

[19] Orlando, S., Palmerini, P., e Perego, R. On statistical properties of transactional datasets. Relatório Técnico CS-2003-11, Universitá di Venezia, 2003.

[20] Park, J. S., Chen, M., e Yu, P. S. An effective hash-based algorithm for mining association rules. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1995), pp. 175–186.

[21] Pei, J., Han, J., Lu, H., Nishio, S., e Yang., S. A. H–mine: Hyper–structure mining of frequent patterns in large databases. In *Proceedings of the IEEE ICDM International Conference on Data Mining* (2001), pp. 441–448.

[22] Pietracaprina, A., e Zandolin, D. Mining frequent itemsets using patricia tries. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations* (2003).

[23] Prado, A., Targa, C., e Plastino, A. Improving direct counting for frequent itemset mining. In *Proceedings of the 6th DAWAK International Conference on Data Warehousing and Knowledge Discovery* (2004), pp. 371–380.

[24] Pudi, V., e Haritsa, J. R. Armor: Association rule mining based on oracle. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations* (2003).

[25] Savasare, A., Omiecinski, E., e Navathe, S. An efficient algorithm for mining association rules in large databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1995), pp. 432–443.

[26] Targa, C. N. Mineração eficiente de regras de associação através da indexação de conjuntos candidatos. *Dissertação de Mestrado, Universidade Federal Fluminense* (2002).

[27] Toivonen, H. Sampling large databases for association rules. In *Proceedings of the 22th VLDB International Conference on Very Large Databases* (1996), pp. 134–145.

[28] Uno, T., Asai, T., Uchida, Y., e Arimura, H. Lcm: An efficient algorithm for enumerating frequent closed item sets. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations* (2003).

[29] Wang, K., Tang, L., e Liu, J. J. Top down fp-growth for association rule mining. In *Proceedings of the 6th Pacif-Asia Conference on Advances in knowledge Discovery and Data Mining* (2002), pp. 334–340.

[30] Zaki, M. J., e Gouda, K. Fast vertical mining using diffsets. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2003), pp. 326–335.

[31] Zaki, M. J., e Hsiao, C. Charm: An efficient algorithm for closed itemset mining. In *Proceedings of the 2nd SIAM International Conference on Data Mining* (2002), pp. 457–473.

[32] Zheng, Z., Kohavi, R., e Mason, L. Real world performance of association rule algorithms. In *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2001), pp. 401–406.