Gustavo Menezes Siqueira

Algoritmos de Mineração de Dados Eficientes Quanto ao Consumo de Memória

Dissertação apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Belo Horizonte

16 de julho de 2004

Resumo

A quantidade de dados submetida às aplicações de mineração de dados tem crescido consideravelmente como conseqüência indireta das reduções dos custos de coleta, transmissão e armazenamento de dados. Portanto, as aplicações de mineração de dados devem ser escaláveis, isto é, as perdas em desempenho devem ser pequenas com o aumento do tamanho da entrada. A mineração de conjuntos freqüentes é uma aplicação popular de mineração de dados para a qual há diversos algoritmos e implementações. O EClaT está entre os algoritmos mais bem-sucedidos e conhecidos. Seu tipo abstrato de dados que mais consome memória é o conjunto de números naturais. Nesse trabalho, substituímos a implementação desse tipo abstrato de dados por outra, comumente empregada por algoritmos de recuperação de informação mas nunca antes empregada por algoritmos de mineração de dados, que economiza memória. Também adaptamos para o novo contexto e/ou implementamos outras estratégias de economia de memória. Obtivemos economia do consumo máximo de memória de até uma ordem de magnitude em relação à implementação original.

Abstract

The volume of data input to data mining applications has grown considerably as an indirect consequence of the price reductions for data aquisition, transmission and storage. Thus, data mining applications must be scalable, that is, the losses in performance should be small when the size of the input is increased. Frequent itemset mining is a popular data mining application for which there are several algorithms and implementations. EClaT is among the most successful and well-known algorithms. Its most memory consuming abstract data type is the natural number set. In this work, we replaced the implementation for this abstract data type for another, commonly employed by information retrieval algorithms but never before employed by data mining algorithms, that saves memory. We adapted to the new context and/or implemented other memory saving techniques as well. We achived an economy in maximum memory consumption of up to an order of magnitude compared to the original implementation.

Sumário

Lista de Algoritmos

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O conhecimento é um elemento fundamental para a tomada de decisões racionais. Tanto a quantidade quanto a variedade de dados disponíveis para a sua aquisição podem ser enormes, dificultando ou impossibilitando análise humana não-assistida.

Knowledge Discovery in Databases, KDD, ou Descoberta de Conhecimento em Bancos de Dados, é o processo não-trivial de identificação de padrões válidos, novos, potencialmente úteis e, finalmente, inteligíveis. O processo de *KDD*, ilustrado pela Figura 1.1, é interativo e iterativo, com várias decisões tomadas pelo usuário [6], envolvendo numerosos passos, enumerados a seguir.

1. Entendimento do domínio de aplicação e do conhecimento prévio relevante e identificação do objetivo do processo do ponto de vista do destinatário;

2. Criação de um conjunto de dados alvo, selecionando um conjunto de dados ou focalizando em um subconjunto de variáveis ou amostras, sobre o qual a descoberta será realizada;

3. Limpeza e pré-processamento dos dados, o que inclui a remoção de ruído, o tratamento de valores ausentes e a consideração de variações temporais;

4. Redução e projeção dos dados, que consiste em encontrar características úteis para representar os dados dependendo do objetivo da tarefa. Com redução de dimensionalidade ou métodos de transformação, o número efetivo de variáveis sob consideração pode ser reduzido, ou representações invariantes para os dados podem ser encontradas;

5. Casamento dos objetivos do processo de *KDD* (primeiro passo) a um método particular de mineração de dados, por exemplo, sumarização, classificação, regressão ou agrupamento;

6. Análise exploratória e seleção de modelos e hipóteses, que envolve a escolha dos algoritmos de mineração de dados e métodos de seleção a serem utilizados para a busca por padrões. Este processo inclui a decisão de quais modelos e parâmetros poderiam ser apropriados e o casamento de um método particular de mineração de dados ao critério global do processo de *KDD*;

7. Mineração de dados, a busca por padrões de interesse em uma forma representacional particular ou um conjunto de tais representações, incluindo regras, árvores de classificação, regressão ou agrupamento. O usuário provê dados adequados ao método de mineração de dados realizando corretamente os passos precedentes;

8. Interpretação dos padrões minerados, possivelmente retornando a qualquer dos passos 1 a 7 para iteração adicional. Este passo pode também envolver visualização dos dados ou dos padrões e modelos extraídos;

9. Atuação sobre o conhecimento descoberto, utilizando-o diretamente, incorporando-o a outro sistema para ação adicional, documentando-o ou reportando-o às partes interessadas. Este passo também inclui o confronto e a resolução de potenciais conflitos com conhecimento previamente aceito ou extraído.

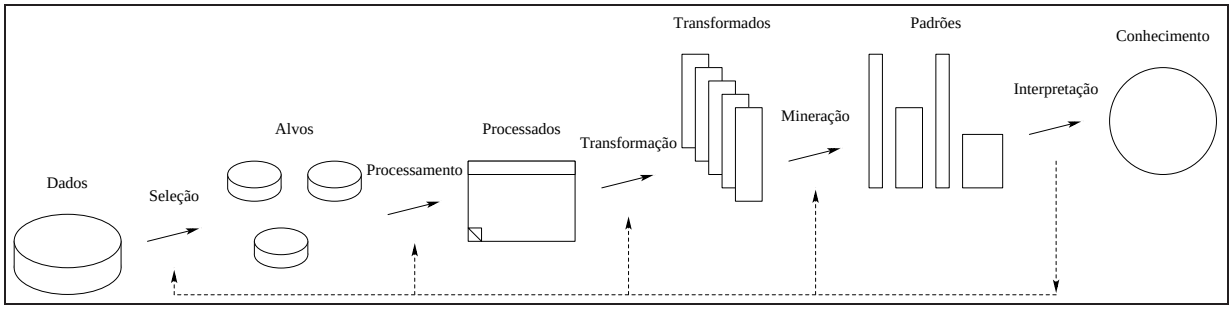


Figura 1.1: Uma visão geral dos passos que compõem o *KDD*

Os conjuntos freqüentes e as regras de associação [1] são dois exemplos de padrões de interesse da mineração de dados. A entrada para os problemas de mineração de conjuntos freqüentes e de mineração de regras de associação é um multiconjunto de conjuntos de entidades. Um conjunto freqüente é um conjunto de entidades cujo suporte, freqüência relativa que o conjunto é subconjunto de um elemento pertencente à entrada, é maior ou igual ao parâmetro de limite inferior de suporte, definido pelo usuário. Uma regra de associação é uma implicação entre dois conjuntos de entidades, denominados antecedente e conseqüente, onde a união entre antecedente e conseqüente é um conjunto freqüente e a confiança, freqüência relativa que o conseqüente é subconjunto de um superconjunto do antecedente pertencente à entrada, é maior ou igual ao parâmetro de limite inferior de confiança. A definição formal do problema e um exemplo simples serão apresentados no Capítulo 2. O problema de mineração de regras de associação pode ser decomposto em dois subproblemas, o primeiro, a mineração de conjuntos freqüentes, e o segundo, relativamente simples, o cálculo das regras de associação em função da solução do primeiro subproblema. Fica assim estabelecida a importância dos conjuntos freqüentes ao processo de *KDD* e à mineração de regras de associação.

Os algoritmos de mineração de conjuntos freqüentes enfrentam problemas de desempenho e escalabilidade porque são intensivos em acessos a dispositivos de entrada e saída, armazenamento e processamento e a quantidade de informações resultante é exponencial em função do parâmetro de entrada.

1.1 Motivação

O problema de escalabilidade dos algoritmos de mineração de conjuntos freqüentes é muito grave. O consumo de memória por esses algoritmos eventualmente ultrapassa a capacidade de memória primária e o sistema operacional é obrigado a recorrer à memória secundária. Acessos de pouca localidade de referência espacial à memória secundária são ordens de grandeza mais lentos que à memória primária. Quando nem a memória secundária é suficiente o sistema operacional é obrigado a abortar o processo.

O consumo de memória por algoritmos de mineração de conjuntos freqüentes cresce invariavelmente à medida em que o limite inferior de suporte decresce. O crescimento é sempre observável, mesmo que os valores absolutos do consumo sejam menores ou maiores dependendo de características das entradas, e quase sempre exponencial. O gráfico da Figura 1.2 exibe o consumo de memória máximo de uma implementação de algoritmo de mineração de conjuntos freqüentes em função do limite inferior de suporte para entradas de diferentes características. Não apresentamos as curvas completas para todas bases de dados por termos limitado o tempo de execução dos processos. Observamos que ao reduzir o limite inferior de suporte, o consumo máximo de memória para algumas bases de dados cresce rapidamente, comprometendo a escalabilidade quando o valor for maior que a quantidade de memória primária disponível.

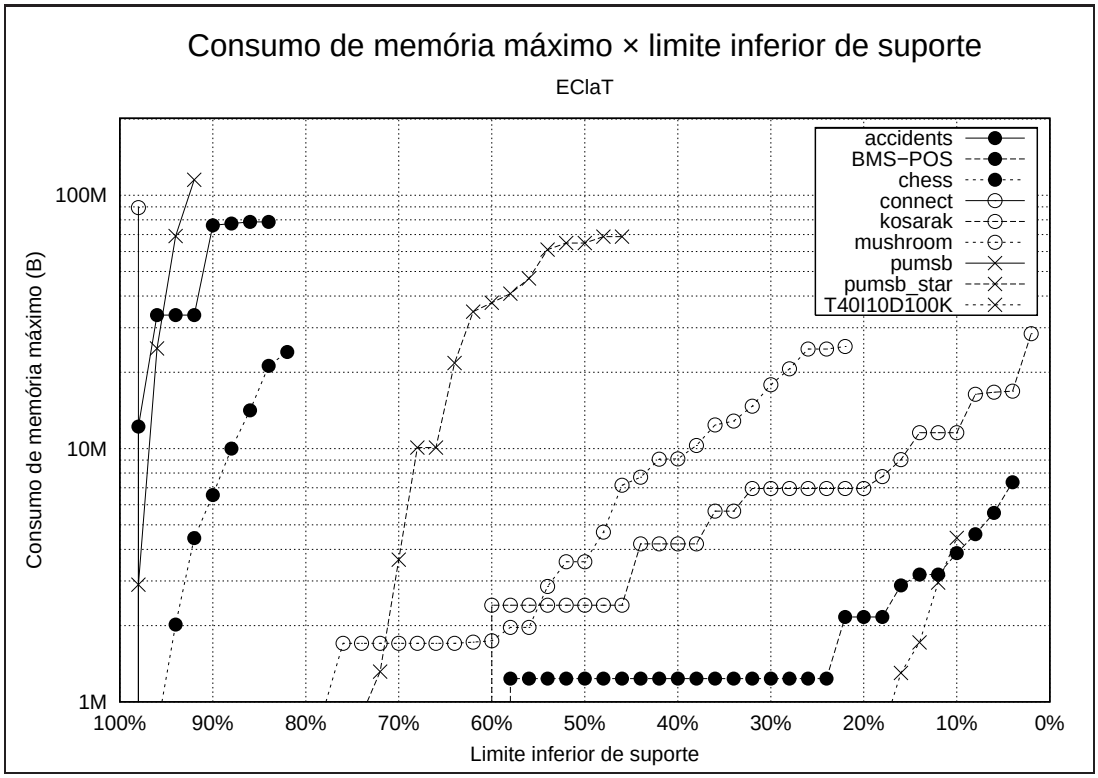


Figura 1.2: Consumo de memória máximo × limite inferior de suporte

1.2 Objetivo

O objetivo deste trabalho é reduzir o consumo de memória sem afetar significativamente o desempenho de algoritmos de mineração de conjuntos freqüentes, pela aplicação inédita de uma técnica de compressão de conjuntos totalmente ordenados finitos de números inteiros positivos à principal estrutura de dados desses algoritmos.

1.3 Metodologia

A metodologia de desenvolvimento do trabalho envolveu a revisão bibliográfica das estratégias de redução de memória empregadas pelos algoritmos de mineração de conjuntos freqüentes e das técnicas de compressão aplicáveis a conjuntos totalmente ordenados finitos de números inteiros positivos, a definição de algoritmos de mineração de conjuntos freqüentes em função de operadores sobre conjuntos totalmente ordenados finitos de números inteiros positivos, a implementação e a análise empírica de algoritmos de mineração de conjuntos freqüentes.

1.4 Trabalhos relacionados

As implementações de algoritmos de mineração de conjuntos freqüentes baseados no EClaT [20] armazenam, para cada conjunto de itens, o conjunto dos identificadores das transações que o contêm. Usualmente, esses conjuntos são representados por um vetor ordenado de números inteiros positivos. Apresentamos nesse trabalho uma representação alternativa para esses conjuntos. Várias outras representações alternativas já foram propostas para atenuar o problema de escalabilidade dos algoritmos de mineração de conjuntos freqüentes. As mais bem-sucedidas são os *bitmap*s [7], os *bitmap*s comprimidos por *skinning* [14] e os *diffset*s [19]. Uma outra representação, os *d-gap*s [16], igualmente aplicável à mineração de conjuntos freqüentes, foi proposta para a indexação de documentos.

Um *bitmap* é uma representação de um conjunto por uma seqüência de valores booleanos na qual cada valor representa a pertinência de um determinado elemento ao conjunto. Os *bitmap*s são utilizados pelos algoritmos de mineração de conjuntos freqüentes DCI [13] e EClaT [20], implementado por Borgelt [2], e pelo algoritmo de mineração de conjuntos maximais (um problema relacionado) MAFIA [9]. A técnica de compressão *skinning* reduz o comprimento de um *bitmap* quando as ausências são muito mais freqüentes que as presenças de elementos no conjunto. Um *diffset* representa um conjunto pela diferença a um superconjunto de referência e a cardinalidade da diferença normalmente é menor que a cardinalidade do conjunto. A ausência de elementos é desnecessariamente representada pelos *bitmap*s, mesmo comprimidos por *skinning*, e nenhuma técnica de compressão é aplicada aos *diffset*s.

Um *d-gap* representa um conjunto totalmente ordenado finito de números inteiros positivos por uma seqüência de diferenças entre elementos consecutivos codificadas por um código para o

qual números menores ocupam menos memória. Normalmente, as diferenças são muito menores que os próprios elementos e os *d-gap*s economizam memória.

Alguns refinamentos relacionados ao consumo de memória ou ao desempenho dos algoritmos de mineração de conjuntos freqüentes são o curto-circuito, empregado pelo algoritmo EClaT [20], a projeção, empregada pelo algoritmo (de mineração de conjuntos maximais) MAFIA [9] e pelo algoritmo FP-growth [10] e a reordenação dinâmica, empregada pelo algoritmo (de mineração de conjuntos maximais) MAFIA [9].

Os trabalhos a serem utilizados para a solução proposta são detalhados no Capítulo 3.

1.5 Contribuições

As contribuições desse trabalho são:

- Aplicação inédita à mineração de conjuntos freqüentes de técnica de compressão baseada em codificação de diferenças adjacentes de elementos de conjuntos totalmente ordenados;
- Revisão bibliográfica das estratégias de redução de consumo de memória aplicáveis à mineração de conjuntos freqüentes;
- Definição dos operadores sobre conjuntos necessários a algoritmos de mineração de conjuntos freqüentes;
- Implementação e análise empírica do algoritmo zEClaT e dos refinamentos de curto-circuito, *diffset*s, projeção e reordenação dinâmica.
- Evidência de que houve redução do consumo de memória em relação a algoritmos existentes de até uma ordem de magnitude.

1.6 Organização

Apresentamos formalmente o problema de mineração de conjuntos freqüentes, os operadores sobre conjuntos e o algoritmo EClaT, que os utiliza, no Capítulo 2. Algumas estratégias de redução de consumo de memória aplicáveis à mineração de conjuntos freqüentes são abordadas no Capítulo 3. A adaptação do algoritmo EClaT, denominada zEClaT, os experimentos e a análise dos resultados são os assuntos do Capítulo 4. Finalmente concluímos e vislumbramos trabalhos futuros no Capítulo 5.

Capítulo 2

Mineração de Conjuntos Freqüentes

O problema de mineração de conjuntos freqüentes consiste em calcular, a partir da entrada $\mathcal{D}$, um arquivo que representa um multiconjunto de conjuntos de entidades, e do parâmetro $\sigma_{\inf}$, um número real pertencente ao intervalo $(0, 1]$, os conjuntos de entidades cujos suportes, probabilidade de que o conjunto de entidades seja subconjunto de um elemento pertencente a $\mathcal{D}$, são maiores ou iguais a $\sigma_{\inf}$ e os respectivos suportes. A mineração de conjuntos freqüentes é simultaneamente um problema de mineração de dados e uma etapa do problema de mineração de regras de associação.

A instância do problema na qual a entrada representa os cupons de vendas de um supermercado é denominada análise de cestas de compra. Os conjuntos de produtos vendidos juntos freqüentemente pelo supermercado e a freqüência relativa da venda dos conjuntos de produtos são a solução dessa instância. Exemplos de pares ordenados de conjunto freqüente e suporte para a análise de cestas de compra são $(\{\text{tomate}\}, 1\%)$, $(\{\text{cebola}\}, 0,8\%)$ e $(\{\text{tomate}, \text{cebola}\}, 0,5\%)$. O conhecimento desses pares ordenados ajudaria o gerente do departamento de hortifrutigranjeiros do supermercado a decidir, de acordo com seu objetivo, por exemplo, aumentar os preços das cebolas e dos tomates, iniciar uma promoção das cebolas, finalizar uma promoção dos tomates ou aproximar ou distanciar as gôndolas das cebolas e dos tomates. Outras instâncias do problema incluem a análise de compras com cartão de crédito, a análise de padrões de chamadas telefônicas, a identificação de solicitações fraudulentas de seguro médico e a análise de compras de serviços de telecomunicações.

Nesse capítulo apresentamos o problema de mineração de conjuntos freqüentes, a terminologia e os conceitos matemáticos envolvidos, os operadores sobre conjuntos empregados pelos algoritmos de mineração de conjuntos freqüentes e seus refinamentos, e o algoritmo de mineração de conjuntos freqüentes EClaT.

2.1 Definição Formal, Terminologia e Conceitos

Nessa seção apresentamos a definição formal do problema de mineração de conjuntos freqüentes e introduzimos a terminologia e os conceitos auxiliares às descrições do problema, dos algoritmos e dos refinamentos.

Sejam a entrada $\mathcal{D}$, um multiconjunto de conjuntos de entidades, e o parâmetro $\sigma_{\inf}$, um número real pertencente ao intervalo $(0, 1]$. O objetivo da mineração de conjuntos freqüentes é calcular o conjunto $\mathcal{F}$, expresso abaixo.

$$\mathcal{F} = \left\{ \left(I, \frac{|\{T : T \in \mathcal{D} \wedge I \subseteq T\}|}{|\mathcal{D}|} \right) : I \in \mathcal{P}\left(\bigcup_{T \in \mathcal{D}} T \right) \wedge \frac{|\{T : T \in \mathcal{D} \wedge I \subseteq T\}|}{|\mathcal{D}|} \geq \sigma_{\inf} \right\}$$

Os elementos pertencentes a $\mathcal{D}$ são chamados transações. Os elementos pertencentes às transações são chamados itens. O conjunto de itens que pertencem ao menos a uma transação é expresso por $\mathcal{I} = \bigcup_{T \in \mathcal{D}} T$ e o conjunto de subconjuntos de $\mathcal{I}$, o conjunto potência de $\mathcal{I}$ (Conceito 2.20) é expresso por $\mathcal{P}(\mathcal{I})$. Quando o item i pertence à transação T, dizemos que o item i ocorre na transação T. Um conjunto de itens é um *itemset*. Quando o *itemset* I é subconjunto da transação T, dizemos que o *itemset* I ocorre na transação T. Um *itemset* de cardinalidade k é um k-*itemset*. A quantidade de transações nas quais o *itemset* I ocorre, a freqüência do *itemset* I em $\mathcal{D}$, é expressa por $\Sigma(I) = |\{T : T \in \mathcal{D} \wedge I \subseteq T\}|$ e a freqüência relativa do *itemset* I em $\mathcal{D}$, o suporte de I, é expresso por $\sigma(I) = {}^{\Sigma(I)}/_{|\mathcal{D}|}$. $\mathcal{F}$ é sintetizado pela expressão abaixo.

$$\mathcal{F} = \{(I, \sigma(I)) : I \in \mathcal{P}(\mathcal{I}) \wedge \sigma(I) \geq \sigma_{\inf}\}$$

O identificador de uma transação é o *tid* da mesma. O conjunto totalmente ordenado de *tid*s das transações nas quais o item i ocorre é o *tidset* de i, simbolizado por $\mathcal{L}(i)$. O conjunto totalmente ordenado de *tid*s das transações nas quais o *itemset* I ocorre é o *tidset* de I, simbolizado por $\mathcal{L}(I)$. O *tidset* do *itemset* I é igual à interseção dos *tidset*s dos itens pertencentes a I, formalmente, $\mathcal{L}(I) = \bigcap_{i \in I} \mathcal{L}(i)$.

Transação
$\{A, B, D, E\}$
$\{B, C, E\}$
$\{A, B, D, E\}$
$\{A, B, C, E\}$
$\{A, B, C, D, E\}$
$\{B, C, D\}$

Tabela 2.1: Multiconjunto de transações

itemset	Suporte
$\{\}$	100%
$\{A\}$	67%
$\{A, B\}$	67%
$\{A, B, D\}$	50%
$\{A, B, D, E\}$	50%
$\{A, B, E\}$	67%
$\{A, D\}$	50%

itemset	Suporte
$\{A, D, E\}$	50%
$\{A, E\}$	67%
$\{B\}$	100%
$\{B, C\}$	67%
$\{B, C, E\}$	50%
$\{B, D\}$	67%
$\{B, D, E\}$	50%

itemset	Suporte
$\{B, E\}$	83%
$\{C\}$	67%
$\{C, E\}$	50%
$\{D\}$	67%
$\{D, E\}$	50%
$\{E\}$	83%

Tabela 2.2: *itemset*s freqüentes

Considere o multiconjunto de transações da Tabela 2.1, os itens são A, B, C, D e E e as transações são $\{A, B, D, E\}$, $\{B, C, E\}$, $\{A, B, D, E\}$, $\{A, B, C, E\}$, $\{A, B, C, D, E\}$ e $\{B, C, E\}$. $\mathcal{I}$ é $\{A, B, C, D, E\}$. Alguns *itemset*s são $\{\}$, $\{A\}$, $\{A, B\}$ e $\{A, B, C\}$, respectivamente, o 0-*itemset*, um 1-*itemset*, um 2-*itemset* e um 3-*itemset*. O item A ocorre e o item C não ocorre na transação $\{A, B, D, E\}$ porque $A \in \{A, B, D, E\}$ e $C \notin \{A, B, D, E\}$. O *itemset* $\{A, B\}$ ocorre e o *itemset* $\{A, C\}$ não ocorre na transação $\{A, B, D, E\}$ porque $\{A, B\} \subseteq \{A, B, D, E\}$ e $\{A, C\} \nsubseteq \{A, B, D, E\}$. As freqüências e os suportes dos *itemset*s $\{\}$, $\{A\}$, $\{A, B\}$ e $\{A, B, C\}$ são, respectivamente, 6 e 100%, 4 e 67%, 4 e 67% e 2 e 33%. Os *itemset*s freqüentes para o limite inferior de suporte 50% são exibidos pela Tabela 2.2.

Apresentamos agora diversos conceitos [15, 18] úteis à explicação do algoritmo EClaT na Seção 2.3 e dos refinamentos no Capítulo 3.

Conceito 2.1 *Ordem Parcial*

Uma relação $\leq$ é uma ordem parcial sobre um conjunto S se são válidas as propriedades de reflexividade ($a \leq a, \forall a \in S$), anti-simetria ($a \leq b \wedge b \leq a \Rightarrow a = b, \forall a \in S \wedge b \in S$) e transitividade ($a \leq b \wedge b \leq c \Rightarrow a \leq c, \forall a \in S \wedge b \in S \wedge c \in S$).

Conceito 2.2 *Conjunto Parcialmente Ordenado*

Um conjunto parcialmente ordenado é um par ordenado $P = (S, \leq)$, onde S é um conjunto e $\leq$ é a ordem parcial de P (Conceito 2.1). O par ordenado $(\mathbb{Z}^+, |)$, onde $\mathbb{Z}^+$ é o conjunto de números inteiros positivos e $|$ é a relação de divisibilidade, é um conjunto parcialmente ordenado.

Conceito 2.3 *Ordem Total*

Uma relação $\leq$ é uma ordem total sobre um conjunto S se ela é uma ordem parcial (Conceito 2.1) e é válida a propriedade $a \leq b \vee b \leq a, \forall a \in S \wedge b \in S$.

Conceito 2.4 *Conjunto Totalmente Ordenado*

Um conjunto totalmente ordenado é um par ordenado $P = (S, \leq)$, onde S é um conjunto e $\leq$ é a ordem total de P (Conceito 2.3). O par ordenado $(\mathbb{Z}^+, <)$, onde $\mathbb{Z}^+$ é conjunto de números inteiros positivos e $<$ é a relação menor que, é um conjunto totalmente ordenado.

Conceito 2.5 *Conjuntos Isomórficos*

Dois conjuntos totalmente ordenados $(A, \leq)$ e $(B, \leq)$ (Conceito 2.4) são isomórficos se, e somente se, existe uma bijeção $f : A \rightarrow B \mid a_1 \leq a_2 \Leftrightarrow f(a_1) \leq f(a_2), \forall a_1 \in A \wedge a_2 \in A$. Em outras palavras, as cardinalidades dos conjuntos são iguais e existe um mapeamento entre eles que preserva a ordem. Quaisquer dois conjuntos totalmente ordenados finitos de cardinalidades iguais são isomórficos.

Conceito 2.6 *Relação de Equivalência*

Uma relação de equivalência sobre um conjunto S é um subconjunto de $S \times S$, isto é, uma coleção $\equiv$ de pares ordenados de elementos de S, satisfazendo as propriedades reflexiva ($s_1 \equiv s_1, \forall s_1 \in S$), simétrica ($s_1 \equiv s_2 \Rightarrow s_2 \equiv s_1, \forall s_1 \in S \wedge s_2 \in S$) e transitiva ($s_1 \equiv s_2 \wedge s_2 \equiv s_3 \Rightarrow s_1 \equiv s_3, \forall s_1 \in S \wedge s_2 \in S \wedge s_3 \in S$).

Conceito 2.7 *Classe de Equivalência*

Uma classe de equivalência é definida como um subconjunto da forma $\{s_1 : s_1 \in S \wedge s_2 \in S \wedge s_1 \equiv s_2\}$. A notação $s_1 \equiv s_2$ significa que existe uma relação de equivalência entre s_1 e s_2. Pode ser mostrado que quaisquer duas classes de equivalência são ou iguais ou disjuntas, então a coleção de classes de equivalência forma uma partição de S.

$s_1 \equiv s_2 \Leftrightarrow s_1 \in [S] \wedge s_2 \in [S], \forall s_1 \in S \wedge s_2 \in S$

Para todos $s_1 \in S \wedge s_2 \in S$, temos $s_1 \equiv s_2$ se, e somente se, s_1 e s_2 pertencem à mesma classe de equivalência.

Conceito 2.8 *Relação de Cobertura*

Sejam $(P, \leq)$ um conjunto parcialmente ordenado (Conceito 2.2) e p_1 e p_2 dois elementos pertencentes a P. p_1 é coberto por p_2, simbolizado por $p_1 \sqsubset p_2$, se $p_1 \leq p_2 \wedge \nexists p \in P \mid p_1 \leq p \wedge p \leq p_2$. Considere o conjunto parcialmente ordenado $(\mathbb{Z}^+, |)$. 4 é coberto por 12 porque $\nexists z \in \mathbb{Z}^+ \mid 4 \mid z \wedge z \mid 12$.

Conceito 2.9 *Limites inferior e superior*

Sejam $(P, \leq)$ um conjunto parcialmente ordenado (Conceito 2.2), p um elemento pertencente a P e S um subconjunto de P. p é um limite inferior de S se $p \leq s, \forall s \in S$. p é um limite superior de S se $s \leq p, \forall s \in S$. Considere $(\mathbb{Z}^+, |)$. Os limites inferiores são os divisores comuns e os limites superiores são os múltiplos comuns.

Conceito 2.10 *Encontro e junção*

Sejam $(P, \leq)$ um conjunto parcialmente ordenado (Conceito 2.2) e S um subconjunto de P. A junção de S, simbolizada por $\bigvee S$, é o menor limite superior de S e o encontro de S, simbolizado por $\bigwedge S$, é o maior limite inferior de S (Conceito 2.9). Considere $(\mathbb{Z}^+, |)$. O encontro é o máximo divisor comum e a junção é o mínimo múltiplo comum.

Conceito 2.11 *Fundo e topo*

Seja $(P, \leq)$ um conjunto parcialmente ordenado (Conceito 2.2). O fundo de P, simbolizado por $\perp(P)$, é o menor elemento de P. O topo de P, simbolizado por $\top(P)$, é o maior elemento de P. Considere $(\mathbb{Z}^+, |)$. O fundo é 1 e não existe topo.

Conceito 2.12 *Semitreliças encontro e junção*

Seja $(P, \leq)$ um conjunto parcialmente ordenado (Conceito 2.2). P é uma semitreliça encontro se existe o encontro (Conceito 2.10) de qualquer par de elementos pertencentes a P ($\exists p_1 \curlywedge p_2, \forall p_1 \in P \wedge p_2 \in P$). P é uma semitreliça junção se existe a junção (Conceito 2.10) de qualquer par de elementos pertencentes a P ($\exists p_1 \curlyvee p_2, \forall p_1 \in P \wedge p_2 \in P$).

Conceito 2.13 *Treliça*

Seja $(P, \leq)$ um conjunto parcialmente ordenado (Conceito 2.2). P é uma treliça se P é simultaneamente semitreliça encontro e semitreliça junção (Conceito 2.12) ($\exists p_1 \curlyvee p_2 \wedge \exists p_1 \curlywedge p_2, \forall p_1 \in P \wedge p_2 \in P$).

Conceito 2.14 *Treliça completa*

Seja $(P, \leq)$ um conjunto parcialmente ordenado (Conceito 2.2). P é uma treliça completa se existem o encontro e a junção (Conceito 2.10) de qualquer subconjunto de P ($\exists \curlyvee S \wedge \exists \curlywedge S, \forall S \subseteq P$).

Conceito 2.15 *Subtreliça*

Sejam $(P, \leq)$ um conjunto parcialmente ordenado (Conceito 2.2) e S um subconjunto próprio de P. S é uma subtreliça de P se pertencem a S o encontro e a junção (Conceito 2.10) de qualquer par de elementos pertencentes a S ($s_1 \curlyvee s_2 \in S \wedge s_1 \curlywedge s_2 \in S, \forall s_1 \in S \wedge s_2 \in S$).

Conceito 2.16 *Treliça distributiva*

Uma treliça P (Conceito 2.13) é distributiva se $p_1 \curlywedge (p_2 \curlyvee p_3) = (p_1 \curlywedge p_2) \curlyvee (p_1 \curlywedge p_3), \forall p_1 \in P \wedge p_2 \in P \wedge p_3 \in P$.

Conceito 2.17 *Átomo*

Seja P uma treliça (Conceito 2.13) com um elemento fundo $\perp$ (Conceito 2.11). Então $p \in P$ é chamado um átomo se $\perp \sqsubset p$, isto é, p cobre $\perp$ (Conceito 2.8). O conjunto de átomos de P é denotado por $\mathcal{A}(P)$.

Conceito 2.18 *Treliça booleana*

Uma treliça P é uma treliça booleana se:

- É uma treliça distributiva (Conceito 2.16)

- $\exists\top(P) \in P$ e $\exists\bot(P) \in P$ (Conceito 2.11)
- Qualquer elemento p pertencente a P tem um complemento

Conceito 2.19 *Treliça de conjuntos*

Para o conjunto S, qualquer $P \subseteq \mathcal{P}(S)$ é uma treliça de conjuntos se é fechado sob finitas uniões e interseções, isto é, $(P;\ \subseteq)$ é uma treliça com a ordem parcial especificada pela relação de subconjunto $\subseteq$, $X \curlyvee Y = X \cup Y$, e $X \curlywedge Y = X \cap Y$.

Conceito 2.20 *Conjunto potência*

O conjunto potência $\mathcal{P}(S)$ do conjunto S é o conjunto de todos os subconjuntos de S.

Conceito 2.21 *Treliça potência*

Seja P um conjunto. O conjunto ordenado $\mathcal{P}(P)$, o conjunto potência de P (Conceito 2.20), é uma treliça completa para a qual junção e encontro são união e interseção, respectivamente, ($\bigvee_{i\in I} A_i = \bigcup_{i\in I} A_i$, $\bigwedge_{i\in I} A_i = \bigcap_{i\in I} A_i$). O topo de $\mathcal{P}(P)$ é $\top = P$, e o fundo de $\mathcal{P}(P)$ é $\bot = \emptyset$.

Os diversos conceitos definidos nessa seção são subsídios para a apresentação do algoritmo EClaT, na Seção 2.3, e o algoritmo zEClaT, no Capítulo 4.

2.2 Operadores sobre Conjuntos

Nessa seção são apresentados os operadores de cardinalidade, diferença, diferença simétrica, interseção e união de conjuntos. O algoritmo zEClaT será descrito no Capítulo 4 em função dos operadores de cardinalidade, diferença e interseção. A cardinalidade e a interseção são empregadas pelo algoritmo EClaT e a cardinalidade e a diferença são empregadas pelo algoritmo dEClaT. Nenhum algoritmo de mineração de conjuntos freqüentes conhecido emprega a diferença simétrica ou a união, que são apresentadas apenas para evitar perda de generalidade.

Apresentamos uma definição intuitiva, a simbologia, uma definição formal em função da pertinência de elemento a conjunto, exemplos, a expressão análoga da álgebra booleana e o diagrama de Venn para cada operador.

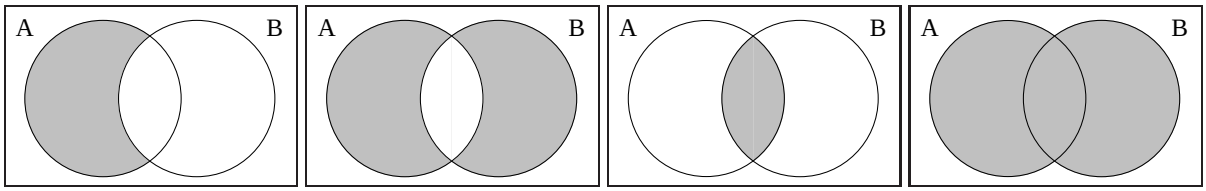
Cardinalidade A cardinalidade do conjunto A, a quantidade de elementos pertencentes a A, é simbolizada por $|A|$ e definida por $\sum_{e\in A} 1$. $|\{2,3,5,7\}| = 4$ e $|\{1,3,5,7,9\}| = 5$. A expressão análoga da álgebra booleana e o diagrama de Venn não se aplicam.

Diferença A diferença entre os conjuntos A e B, o conjunto de elementos pertencentes a A e não pertencentes a B, é simbolizada por $A - B$ e definida por $\{e : e \in A \wedge \neg e \in B\}$. $\{2,3,5,7\} - \{1,3,5,7,9\} = \{2\}$ e $\{1,3,5,7,9\} - \{2,3,5,7\} = \{1,9\}$. A expressão análoga da álgebra booleana é $A \cdot \overline{B}$ e o diagrama de Venn é ilustrado pela Figura 2.1(a).

Diferença Simétrica A diferença simétrica entre os conjuntos A e B, o conjunto de elementos não pertencentes a A e pertencentes a B ou pertencentes a A e não pertencentes a B, é simbolizada por $A \ominus B$ e definida por $\{e : \neg e \in A \wedge e \in B \vee e \in A \wedge \neg e \in B\}$. $\{2,3,5,7\} \ominus \{1,3,5,7,9\} = \{1,2,9\}$ e $\{1,3,5,7,9\} \ominus \{2,3,5,7\} = \{1,2,9\}$. A expressão análoga da álgebra booleana é $\overline{A} \cdot B + A \cdot \overline{B}$ e o diagrama de Venn é ilustrado pela Figura 2.1(b).

Interseção A interseção entre os conjuntos A e B, o conjunto de elementos pertencentes a A e pertencentes a B, é simbolizada por $A \cap B$ e definida por $\{e : e \in A \wedge e \in B\}$. $\{2,3,5,7\} \cap \{1,3,5,7,9\} = \{3,5,7\}$ e $\{1,3,5,7,9\} \cap \{2,3,5,7\} = \{3,5,7\}$. A expressão análoga da álgebra booleana é $A \cdot B$ e o diagrama de Venn é ilustrado pela Figura 2.1(c).

União A união entre os conjuntos A e B, o conjunto de elementos pertencentes a A ou pertencentes a B, é simbolizada por $A \cup B$ e definida por $\{e : e \in A \vee e \in B\}$. $\{2,3,5,7\} \cup \{1,3,5,7,9\} = \{1,3,5,7,9\}$ e $\{1,3,5,7,9\} \cap \{2,3,5,7\} = \{1,3,5,7,9\}$. A expressão análoga da álgebra booleana é $A + B$ e o diagrama de Venn é ilustrado pela Figura 2.1(d).



(a) $A - B$ (b) $A \ominus B$ (c) $A \cap B$ (d) $A \cup B$

Figura 2.1: Diagramas de Venn $A - B$, $A \ominus B$, $A \cap B$ e $A \cup B$

Os operadores sobre conjuntos são os operadores fundamentais dos algoritmos de mineração de conjuntos freqüentes. Eles são responsáveis por grande parte do tempo de execução. Portanto, sua implementação eficiente é essencial para a rapidez dos algoritmos. A complexidade do operador de cardinalidade é $O(1)$ quando um inteiro não-negativo adicional, que representa a cardinalidade do conjunto, é atualizado após a adição de elementos ao conjunto e após a remoção de elementos do conjunto. A complexidade dos operadores de diferença, diferença simétrica, interseção e união é $O(|A| + |B|)$ quando os conjuntos são totalmente ordenados.

O Algoritmo 2.1 detalha a implementação original dos operadores de diferença, diferença simétrica, interseção e união sobre conjuntos totalmente ordenados. Os parâmetros A e B são os conjuntos totalmente ordenados e o parâmetro o é o operador. a e b são os índices dos elementos correntes de A e B, respectivamente. R é o resultado e r é o índice do elemento corrente de R. O algoritmo itera sobre os elementos de A e B segundo a ordem total e adiciona, conforme o operador, os elementos não pertencentes a A e pertencentes a B (linhas 9 e 20), os elementos pertencentes a A e não pertencentes a B (linhas 5 e 17) e os elementos pertencentes a A e pertencentes a B (linha 13) ao resultado, preservando a ordem.

```
1  funct set_operation(A, o, B) ≡
2    while (a < |A| ∧ b < |B|) do
3          if (A[a] < B[b])
4            if (o ∈ {−, ⊖, ∪})
5              R[r] ← A[a]; r ← r + 1;
6            a ← a + 1;
7          else if (B[b] < A[a])
8                 if (o ∈ {⊖, ∪})
9                   R[r] ← B[b]; r ← r + 1;
10                b ← b + 1;
11         else
12               if (o ∈ {∩, ∪})
13                 R[r] ← A[a]; r ← r + 1;
14               a ← a + 1; b ← b + 1;
15   if (o ∈ {−, ∪, ⊖})
16     while (a < |A|) do
17           R[r] ← A[a]; r ← r + 1; a ← a + 1;
18   if (o ∈ {∪, ⊖})
19     while (b < |B|) do
20           R[r] ← B[b]; r ← r + 1; b ← b + 1;
21   return R;
22 .
```

Algoritmo 2.1: Operadores de diferença, diferença simétrica, interseção e união sobre conjuntos totalmente ordenados

2.3 O Algoritmo EClaT

A Figura 2.2 mostra a treliça potência para o banco de dados da Tabela 2.1 e limite inferior de suporte de 50%. Um vértice representa um *itemset*. O primeiro conjunto representa os itens do *itemset* e o segundo conjunto representa o *tidset* do *itemset*. Vértices preenchidos representam *itemset*s freqüentes. As arestas representam a relação de cobertura subconjunto. Para ilustrar a descrição, considere o *itemset* $\{A\}$. O *tidset* de $\{A\}$ é $\{1, 3, 4, 5\}$, $\{A\}$ é freqüente e $\{A\}$ é subconjunto de $\{A, B\}$, $\{A, C\}$, $\{A, D\}$ e $\{A, E\}$,

Observe que o conjunto de todos os *itemset*s freqüentes forma uma treliça encontro porque é fechado sob a operação de encontro, isto é, para quaisquer *itemset*s freqüentes X, e Y, $X \cap Y$ é também freqüente. Por outro lado, não forma uma semitreliça junção, porque X e Y freqüentes, não implica que $X \cup Y$ é freqüente. Os *itemset*s infreqüentes formam uma semitreliça junção.

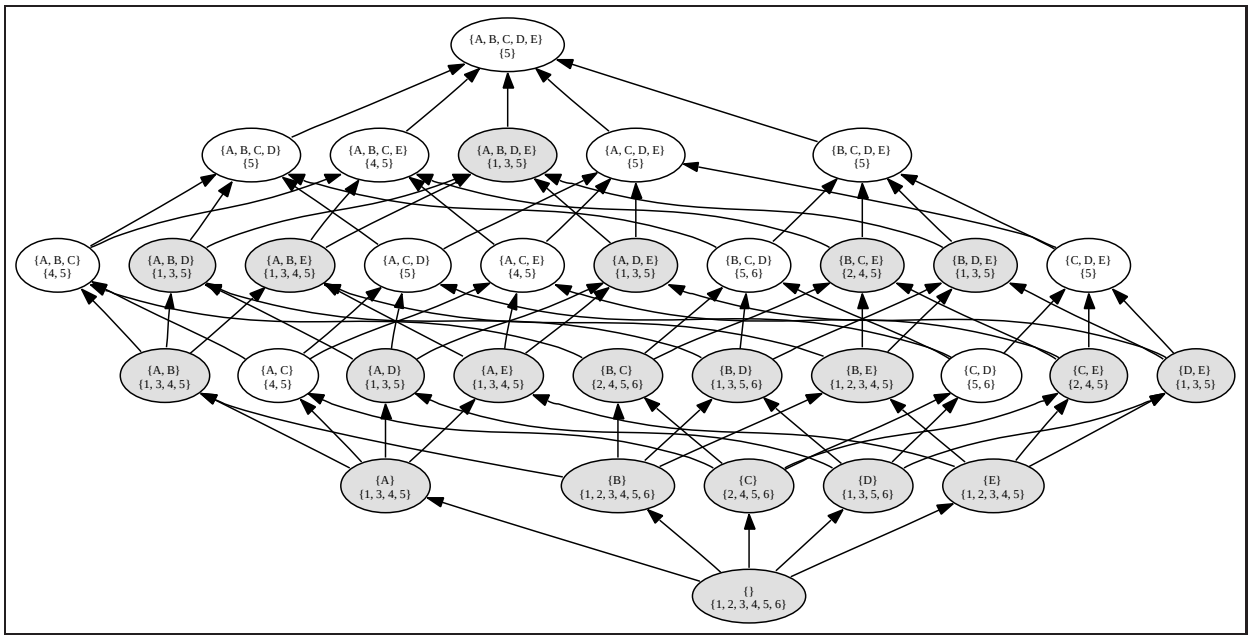


Figura 2.2: Treliça $\mathcal{P}(\{A, B, C, D, E\})$

Lema 2.1 *Todos subconjuntos de um itemset freqüente são freqüentes.*

O lema acima é uma conseqüência do fechamento sob a operação de encontro para o conjunto de *itemset*s freqüentes. Como um corolário, nós temos que todo superconjunto de um *itemset* infreqüente é infreqüente. Essa observação forma a base de uma estratégia de poda bem poderosa em uma busca *bottom-up* por *itemset*s freqüentes, que foi aproveitada em muitos algoritmos de mineração de dados. Nominalmente, somente os *itemset*s freqüentes do nível anterior precisam ser estendidos como candidatos para o nível corrente. Entretanto, a formulação de treliça deixa aparente que não precisamos nos restringir a busca simplesmente *bottom-up*, ou seja, que são igualmente viáveis buscas *top-down*, que parte do *itemset* maximal e identifica *itemset*s infreqüentes, e híbridas, que é alternam *bottom-up* com *top-down*.

Começamos notando que a treliça potência $\mathcal{P}(\mathcal{I})$ no conjunto de itens $\mathcal{I}$ é uma treliça booleana, com o complemento de $X \in \mathcal{P}(\mathcal{I})$ dado por $\mathcal{I} - X$. O conjunto de átomos da treliça potência corresponde ao conjunto de itens, isto é, $\mathcal{A}(\mathcal{P}(\mathcal{I})) = \mathcal{I}$. Associamos a cada átomo (item) X sua *tidset*, denotada $\mathcal{L}(X)$, que é a lista de todos os identificadores de transação que contém o átomo. A *tidset* do átomo A é $\{1, 3, 4, 5\}$.

Lema 2.2 *Para uma treliça booleana finita L, com $X \in L$, $X = \curlyvee\{Y \in \mathcal{A}(L) \mid Y \leq X\}$.*

Em outras palavras cada elemento de uma treliça booleana é dado como uma junção de um subconjunto do conjunto de átomos. Como a treliça potência é uma treliça booleana, com a operação junção correspondendo à união de conjuntos, nós temos.

Lema 2.3 *Para qualquer $X \in \mathcal{P}(\mathcal{I})$, seja $J = \{Y \in \mathcal{A}(\mathcal{P}(\mathcal{I})) \mid Y \leq X\}$. Então $X = \bigcup_{Y \in J} Y$, e $\sigma(X) = |\bigcap_{Y \in J} \mathcal{L}(Y)|$.*

O lema acima diz que se um *itemset* é dado como a união de um conjunto de itens em J, então seu suporte é dado como a interseção das *tidset*s dos elementos em J. Em particular nós podemos determinar o suporte de qualquer k-*itemset* por simples interseção das *tidset*s de quaisquer dois de seus subconjuntos de tamanho $(k-1)$. A verificação da cardinalidade da *tidset* resultante nos diz se o novo *itemset* é freqüente ou não. A Figura 2.2 mostra esse processo. Ela mostra o banco de dados inicial com a *tidset* para cada item (isto é, átomos). A *tidset* intermediária para $\{B,C\}$ é obtida pela interseção das listas de B e C, isto é, $\mathcal{L}(\{B,C\}) = \mathcal{L}(\{B\}) \cap \mathcal{L}(\{C\})$. Similarmente, $\mathcal{L}(\{B,C,E\}) = \mathcal{L}(\{B,C\}) \cap \mathcal{L}(\{B,E\})$, e assim por diante. Assim, somente dois subconjuntos no nível anterior são necessários para computar o suporte de um *itemset* em qualquer nível.

Lema 2.4 *Sejam X e Y dois itemsets, com $X \subseteq Y$. Então $\mathcal{L}(X) \supseteq \mathcal{L}(Y)$.*

Esse lema diz que se X é um subconjunto de Y, então a cardinalidade da *tidset* de Y (isto é, seu suporte) deve ser menor que ou igual à cardinalidade da *tidset* de X. Uma conseqüência importante e prática do lema acima é que cardinalidades de *tidset*s intermediárias encolhem à medida em que movemos para cima na treliça. Isso resulta em interseções e contagem de suporte mais rápidos.

Se tivéssemos memória suficiente poderíamos enumerar todos os *itemset*s freqüentes caminhando pela treliça potência, e realizando interseções para obter o suporte dos *itemset*s. Na prática, entretanto, temos somente uma quantidade limitada de memória primária, e todas as *tidset*s intermediárias não caberiam na memória. Isto levanta uma questão natural: podemos decompor a treliça original em peças menores tal que cada porção possa ser resolvida independentemente na memória principal? Endereçamos essa pergunta abaixo.

O prefixo de X de tamanho k em respeito à relação $\leq$, $p(X,k,\leq)$, é definido pela expressão $p(X,k,\leq) \subseteq X \wedge |p(X,k,\leq)| = k \wedge x_1 \in p(X,k,\leq) \leq x_2 \in X - p(X,k,\leq)$, ou seja, o subconjunto de X de tamanho k cuja relação $\leq$ é verdadeira entre seus elementos e os elementos restantes de X. Por exemplo, $p(\{2,3,5,7\},2,\leq) = \{2,3\}$. Defina uma relação de equivalência $\theta_{k,\leq}$ na treliça $\mathcal{P}(\mathcal{I})$ como seguinte: $\forall X,Y \in \mathcal{P}(\mathcal{I})$, $X \equiv_{\theta_{k,\leq}} Y \Leftrightarrow p(X,k,\leq) = p(Y,k,\leq)$. Isto é, dois *itemset*s estão na mesma classe de equivalência (Conceito 2.7) se eles compartilham um prefixo comum de tamanho k em respeito à relação $\leq$. Nós chamamos $\theta_{k,\leq}$ uma relação de equivalência baseada em prefixo.

Lema 2.5 *Cada classe de equivalência $[X]_{\theta_{k,\leq}}$ induzida pela relação de equivalência $\theta_{k,\leq}$ é uma subtreliça de $\mathcal{P}(\mathcal{I})$.*

Cada $[X]_{\theta_{k,\leq}}$ é por si só uma treliça booleana com seus próprios átomos (Conceito 2.17). Por exemplo, os átomos de $[\{A\}]_{\theta_{k,\leq}}$ são $\{\{A,B\},\{A,C\},\{A,D\},\{A,E\}\}$, e os elementos topo e fundo são $\top = \{A,B,C,D,E\}$, e $\bot = \{A\}$. Pela aplicação dos lemas 2.3, e 2.4, podemos gerar todos os suportes dos *itemset*s em cada classe (subtreliça) pela interseção das *tidset*s de átomos ou quaisquer dois subconjuntos no nível anterior. Se há memória principal suficiente para acomodar *tidset*s temporárias para cada classe, então resolvemos $[X]_{\theta_{k,\leq}}$ independentemente. Outra

característica interessante das classes de equivalência é que os elos entre classes denotam dependências. Isso quer dizer, se quisermos podar um *itemset* se existir pelo menos um subconjunto infreqüente, então temos que processar as classes em uma ordem específica. Em particular nós temos que processar as classes do fundo para o topo, que corresponde a uma ordem lexicográfica reversa, isto é, processamos $[\{E\}]$, depois $[\{D\}]$, seguido por $[\{C\}]$, então $[\{B\}]$ e finalmente $[\{A\}]$. Isso garante que toda informação de subconjunto está disponível para a poda.

Na prática a decomposição induzida por $\theta_{1,\leq}$ é suficiente. Entretanto, em alguns casos, uma classe pode ser muito grande para ser solucionada em memória principal. Nesse cenário, aplicamos a decomposição recursiva. Assumamos que $[\{A\}]$ é muito grande para caber em memória principal. Como $[\{A\}]$ é por si só uma treliça booleana, pode ser decomposto usando $\theta_{2,\leq}$. O conjunto de classes resultante é $\{[\{A,B\}],[\{A,C\}],[\{A,D\}],[\{A,E\}]\}$. Como antes, cada classe pode ser resolvida independentemente, e nós podemos resolvê-las em ordem lexicográfica reversa para habilitar poda de subconjunto. Como antes, os elos mostram as dependências de poda que existem entre as classes. Dependendo da quantidade de memória principal disponível nós podemos particionar recursivamente classes grandes em classes menores, até que cada classe seja pequena o suficiente para ser resolvida independentemente em memória principal.

A busca *bottom-up* é baseada na decomposição recursiva de cada classe em classes menores induzidas pela relação de equivalência $\theta_{k,\leq}$. A treliça de classes de equivalência é atravessada em profundidade. Isto é, primeiro é processada a classe $[\{A\}]$, seguida pela classe $[\{A,B\}]$ e suas derivadas e depois são processadas as classes $[\{B\}]$, $[\{C\}]$, $[\{D\}]$ e $[\{E\}]$. Para calcular o suporte de qualquer *itemset*, simplesmente realizamos a interseção dos *tidset*s de dois dos subconjuntos do nível anterior.

Uma versão não-recursiva do EClaT é detalhada pelo Algoritmo 2.2. Os parâmetros de entrada são um, o conjunto de 1-*itemset*s, e Σ_{inf}, o inteiro positivo $\lceil \sigma_{\text{inf}} \times |\mathcal{D}| \rceil$, e a saída é $\mathcal{F}$, o conjunto de *itemset*s freqüentes.

Os 1-*itemset*s freqüentes são minerados nas linhas 2 a 4 e os demais *itemset*s freqüentes são minerados nas linhas 5 a 22. A mineração dos 1-*itemset*s freqüentes é trivial. Quando a freqüência de um 1-*itemset* pertencente a um é maior que Σ_{inf}, ele é adicionado a $\mathcal{F}$ na linha 4. A mineração dos demais *itemset*s é iterativa. As classes de equivalência corrente e anterior são, respectivamente, as classes de equivalência ao topo e imediatamente inferior ao topo da pilha de classes de equivalência a processar, denominada $classes$. A classe de equivalência representada pelo conjunto vazio é empilhada na linha 5 e a cada iteração o maior *itemset* da classe de equivalência corrente é processado nas linhas 6 a 22.

Quando mais de um *itemset* freqüente pertence à classe de equivalência corrente, os *itemset*s correspondentes às junções entre o maior e os demais *itemset*s pertencentes à classe de equivalência corrente são calculados e eles são adicionados a $\mathcal{F}$ quando suas freqüências são maiores que Σ_{inf} nas linhas 10 a 14. Quando apenas um *itemset* freqüente pertence à classe de equivalência corrente, a classe de equivalência corrente é desempilhada e o maior *itemset* é removido da classe de equivalência anterior nas linhas 20 a 22.

Quando ao menos um dos *itemset*s calculados é freqüente, a classe de equivalência correspondente ao maior *itemset* pertencente à classe de equivalência corrente é empilhada na linha

```
1  funct EClaT (um, Σ_inf) ≡
2    foreach itemset ∈ um do
3      if Σ(itemset) ≥ Σ_inf
4        itemsets.push (itemset) ; F.push (itemset) ;
5    classes.push ([1, |itemsets|]) ;
6    while true do
7          menor ← classes.back () .menor; maior ← classes.back () .maior;
8          if menor < maior
9            adicionar ← false;
10           while menor < maior do
11                 itemset ← itemsets [menor] ϒ itemsets [maior] ;
12                 if Σ(itemset) ≥ Σ_inf
13                   adicionar ← true; itemsets.push (itemset) ; F.push (itemset) ;
14                 menor ← menor + 1;
15           if adicionar
16             classes.push ([classes.back () .maior + 1, |itemsets|]) ;
17           else
18                classes.back () .maior ← classes.back () .maior − 1; itemsets.pop () ;
19         else
20               classes.pop () ; itemsets.pop () ;
21               if classes.empty () return F;
22               classes.back () .maior ← classes.back () .maior − 1; itemsets.pop () ;
23 .
```

Algoritmo 2.2: EClaT não-recursivo

16. Quando não é calculado nenhum *itemset* freqüente, o maior *itemset* é removido da classe de equivalência corrente na linha 18.

Um exemplo passo a passo da execução do algoritmo EClaT é mostrado no Apêndice A.

O próximo capítulo apresenta as técnicas de redução de consumo de memória aplicadas ao algoritmo zEClaT, descrito no Capítulo 4, ou às demais especializações do algoritmo EClaT.

Capítulo 3

Estratégias de Redução de Consumo de Memória

Nesse capítulo apresentamos algumas estratégias de redução de consumo de memória aplicáveis aos algoritmos de mineração de conjuntos freqüentes. Nominalmente, *bitmap*s, curto-circuito, *diffset*s, projeção, reordenação dinâmica, *skinning* e *d-gap*s. Algumas dessas estratégias já foram empregadas por algoritmos de mineração de conjuntos freqüentes e outras apenas por outros algoritmos.

3.1 *Bitmap*s

Os *bitmap*s são um refinamento que procura reduzir o consumo de memória representando os *tidset*s não por vetores de números inteiros positivos mas por vetores de dígitos binários.

Um *bitmap* [7] é uma representação completa de um conjunto de números inteiros em que a ausência ou presença de um número é sinalizada por um dígito binário falso ou verdadeiro na posição correspondente a esse número. A representação esparsa de conjunto demanda um inteiro para cada número pertencente ao conjunto. Nas máquinas atuais, isso corresponde a 32 *bit*s. Quando a razão entre dígitos binários verdadeiros e totais é maior que o inverso do comprimento do inteiro, a representação *bitmap* ocupa menos que a representação esparsa.

Por exemplo, o conjunto $\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31\}$ demanda 32 *bit*s em *bitmap* e 55 *bit*s na representação esparsa. As operações entre conjuntos são realizadas em *bitmap* palavra por palavra. Cada uma dessas operações palavra por palavra substitui uma quantidade de operações igual ao comprimento da palavra em relação à representação tradicional. Isso significa que essas operações são mais rápidas que as operações na representação tradicional em proporção igual a essa redução. Os *bitmap*s são alinhados e as operações *bit* a *bit* são executadas sobre os *bitmap*s. A correspondência entre conjuntos e álgebra booleana permite que isso seja feito. A diferença $A - B$ é dada por $A \cdot \overline{B}$, a diferença simétrica $A \ominus B$ por $\overline{A} \cdot B + A \cdot \overline{B} = A \oplus B$, a interseção $A \cap B$ por $A \cdot B$ e a união $A \cup B$ por $A + B$. Somente os dígitos binários que representam

elementos do universo podem ser verdadeiros. Assim, qualquer operação deve ser seguida pela interseção com o conjunto universo. A cardinalidade pode ser calculada rapidamente por uma tabela pré-computada da quantidade de dígitos verdadeiros para cada divisão da palavra. Se a palavra tem 32 *bit*s e a tabela 16 *bit*s, a contagem dos elementos é potencialmente 16 vezes mais rápida.

Os *bitmap*s são aplicados nos algoritmos de mineração de conjuntos freqüentes EClaT implementado por Borgelt [2] e DCI [13] e no algoritmo de mineração de conjuntos maximais GenMax [9] mas não são aplicados ao algoritmo zEClaT porque, apesar da possibilidade de aplicação das duas abordagens, de *bitmap*s e de *d-gap*s, para o propósito de representação de conjuntos de números naturais, o foco desse trabalho é apenas a abordagem de *d-gap*s.

3.2 Curto-circuito

O curto-circuito é um refinamento aplicado pelo algoritmo EClaT [20] que aborta um operador entre conjuntos quando antecipa que a cardinalidade do resultado será menor que um limite inferior ou maior que um limite superior.

O *diffset* ou *tidset* de um *itemset* é relevante somente se esse *itemset* é freqüente. A informação de infreqüência de um *itemset* é suficiente para que ele seja podado. A interseção entre os *tidset*s de dois *itemset*s é relevante quando a cardinalidade é maior ou igual ao suporte mínimo e a diferença entre os *diffset*s de dois *itemset*s é relevante quando a cardinalidade é menor que a diferença entre o suporte do *itemset* correspondente ao prefixo da classe de equivalência e o suporte mínimo. Quanto antes a infreqüência de um *itemset* for descoberta melhor, porque poupa processamento e armazenamento desnecessários. A verificação das condições de curto-circuito é relativamente simples em relação à recuperação de um elemento de um conjunto e ao armazenamento de um elemento a um conjunto, que podem incluir codificação e decodificação de elementos e ampliação de conjunto.

Suponha que a *tidset* do *itemset* $\{A\}$ ($\mathcal{L}(\{A\})$) seja $\{1, 3, 5, 7, 9\}$, a *tidset* do *itemset* $\{B\}$ ($\mathcal{L}(\{B\})$) seja $\{2, 3, 5, 7\}$ e o suporte mínimo seja 4. A *tidset* do *itemset* $\{A, B\}$ ($\mathcal{L}(\{A, B\}) = \mathcal{L}(\{A\}) \cap \mathcal{L}(\{B\})$) seria relevante para a mineração de conjuntos freqüentes somente se esse *itemset* fosse freqüente. Quando o *itemset* é infreqüente, apenas a informação de infreqüência é relevante. A interseção é realizada percorrendo o elemento 1, que pertence a $\mathcal{L}(\{A\})$ e não pertence a $\mathcal{L}(\{B\})$ e não é acrescentado ao resultado $\mathcal{L}(\{A, B\})$. A seguir percorrendo o elemento 2, que não pertence a $\mathcal{L}(\{A\})$ e pertence a $\mathcal{L}(\{B\})$ e não é acrescentado ao resultado. Nesse instante é possível afirmar que a cardinalidade de $\mathcal{L}(\{A, B\})$ será menor ou igual a 3 porque os elementos do resultado devem pertencer simultaneamente aos dois conjuntos e restam apenas três elementos no segundo conjunto e abortar o operador.

As condições de curto-circuito para as operações $A - B$, $A \ominus B$, $A \cap B$ e $A \cup B$ são expressas na Tabela 3.1, onde a, b e r são as quantidades de elementos avaliados dos conjuntos A e B e adicionados ao resultado e $\inf$ e $\sup$ são os limites inferior e superior da cardinalidade do resultado. A implementação dos operadores sobre conjuntos totalmente ordenados refinada por

curto-circuito é detalhada no Algoritmo 3.1.

```
1  funct set_operation(A, o, B, inf, sup) ≡
2    while (a < |A| ∧ b < |B|) do
3          if (A[a] < B[b])
4            if (o ∈ {−, ⊖, ∪})
5              R[r] ← A[a]; r ← r + 1;
6            a ← a + 1;
7            if (o ∈ {−, ⊖} ∧ r + |A| − a − |B| + b > sup) then exit;
8            if (o ∈ {∩} ∧ r + |B| − b < inf) then exit;
9            if (o ∈ {∪} ∧ r + |A| − a > sup) then exit;
10         else if (B[b] < A[a])
11                 if (o ∈ {⊖, ∪})
12                   R[r] ← B[b]; r ← r + 1;
13                 b ← b + 1;
14                 if (o ∈ {−} ∧ r > sup) then exit;
15                 if (o ∈ {∩} ∧ r + |A| − a < inf) then exit;
16                 if (o ∈ {⊖} ∧ r + |B| − b − |A| + a > sup) then exit;
17                 if (o ∈ {∪} ∧ r + |B| − b > sup) then exit;
18         else
19               if (o ∈ {∩, ∪})
20                 R[r] ← A[a]; r ← r + 1;
21               a ← a + 1; b ← b + 1;
22               if (o ∈ {∩} ∧ r > sup) then exit;
23               if (o ∈ {−} ∧ r + |A| − a < inf) then exit;
24               if (o ∈ {⊖, ∪} ∧ r + |A| − a + |B| − b < inf) then exit;
25   if (o ∈ {−, ∪, ⊖})
26     while (a < |A|) do
27           R[r] ← A[a]; r ← r + 1; a ← a + 1;
28           if (o ∈ {−, ⊖} ∧ r + |A| − a − |B| + b > sup) then exit;
29           if (o ∈ {∪} ∧ r + |A| − a > sup) then exit;
30   if (o ∈ {∪, ⊖})
31     while (b < |B|) do
32           R[r] ← B[b]; r ← r + 1; b ← b + 1;
33           if (o ∈ {⊖} ∧ r + |B| − b − |A| + a > sup) then exit;
34           if (o ∈ {∪} ∧ r + |B| − b > sup) then exit;
35   return R;
36 .
```

Algoritmo 3.1: Operadores de diferença, diferença simétrica, interseção e união sobre conjuntos totalmente ordenados, refinados por curto-circuito

Operação	Condição de curto-circuito
$A - B$	$r + \|A\| - a - \|B\| + b > \sup \vee r > \sup \vee r + \|A\| - a < \inf$
$A \ominus B$	$r + \|A\| - a - \|B\| + b > \sup \vee r + \|B\| - b - \|A\| + a > \sup \vee r + \|A\| - a + \|B\| - b < \inf$
$A \cap B$	$r > \sup \vee r + \|A\| - a < \inf \vee r + \|B\| - b < \inf$
$A \cup B$	$r + \|A\| - a > \sup \vee r + \|B\| - b > \sup \vee r + \|A\| - a + \|B\| - b < \inf$

Tabela 3.1: Condições de curto-circuito para as operações $A - B$, $A \ominus B$, $A \cap B$ e $A \cup B$

O curto-circuito é um refinamento incorporado pelo algoritmo zEClaT, que é avaliado no próximo capítulo.

3.3 *Diffset*s

Os *diffset*s são um refinamento que procura reduzir o consumo de memória representando um conjunto não pelos seus elementos mas pelos elementos da diferença entre esse conjunto e um determinado superconjunto muito parecido.

Os *diffset*s [19] armazenam as diferenças entre a lista de identificadores de transação de um conjunto candidato e a lista de identificadores de transação de seu gerador. Os *diffset*s reduzem dramaticamente o consumo de memória para armazenar resultados intermediários e aumentam significativamente o desempenho.

O armazenamento do *tidset* de cada membro de uma classe é evitado pelos *diffset*s. Ao invés disso, eles armazenam apenas as diferenças dos *tid*s entre cada membro da classe e o *itemset* do prefixo. Essas diferenças dos *tid*s são armazenadas nos *diffset*s, que são as diferenças entre dois *tidset*s (nominalmente, o *tidset* prefixo e o *tidset* do membro da classe), e são propagadas pelo caminho de um nó para seus filhos começando pela raiz. Os membros do nó raiz podem usar *tidset*s ou diferenças do prefixo vazio (que por definição aparece em todos os *tid*s).

Mais formalmente, considere uma classe com prefixo P. Seja $\mathcal{L}(X)$ o *tidset* do elemento X, e seja $d(X)$ o *diffset* de X, com respeito ao *tidset* prefixo, que é o universo de *tid*s corrente. Em métodos verticais normais temos disponível para uma dada classe o *tidset* para o prefixo $\mathcal{L}(P)$ assim como os *tidset*s de todos membros da classe $\mathcal{L}(P \cup X)$. Assuma que $P \cup X$ e $P \cup Y$ são dois membros quaisquer de P. Por definição de suporte é verdade que $\mathcal{L}(P \cup X) \subseteq \mathcal{L}(P)$ e $\mathcal{L}(P \cup Y) \subseteq \mathcal{L}(P)$. Além disso, obtemos o suporte de $P \cup X \cup Y$ pela checagem da cardinalidade de $\mathcal{L}(P \cup X) \cap \mathcal{L}(P \cup Y) = \mathcal{L}(P \cup X \cup Y)$.

Agora suponha que tenhamos disponível não $\mathcal{L}(P \cup X)$ mas $d(P \cup X)$, dado por $\mathcal{L}(P) - \mathcal{L}(X)$, isto é, as diferenças nos *tid*s de X a partir de P. Similarmente, temos disponível $d(P \cup Y)$. A primeira coisa a notar é que o suporte de um *itemset* não é mais a cardinalidade do *diffset* mas deve ser armazenada separadamente e é dado por $\sigma(P \cup X) = \sigma(P) - |d(P \cup X)|$. Então, dados $d(P \cup X)$ e $d(P \cup Y)$, como podemos computar se $P \cup X \cup Y$ é freqüente?

Usamos *diffset*s recursivamente como mencionado acima, isto é, $\sigma(P \cup X \cup Y) = \sigma(P \cup X) - |d(P \cup X \cup Y)|$. Então temos que computar $d(P \cup X \cup Y)$. Pela nossa definição $d(P \cup X \cup Y) = \mathcal{L}(P \cup X) - \mathcal{L}(P \cup Y)$. Mas temos somente *diffset*s, e não *tidset*s como a expressão requer. Isso

é fácil de consertar, porque $d(P \cup X \cup Y) = \mathcal{L}(P \cup X) - \mathcal{L}(P \cup Y) = \mathcal{L}(P \cup X) - \mathcal{L}(P \cup Y) + \mathcal{L}(P) - \mathcal{L}(P) = (\mathcal{L}(p) - \mathcal{L}(P \cup Y)) - (\mathcal{L}(P) - \mathcal{L}(P \cup X)) = d(P \cup Y) - d(P \cup X)$. Em outras palavras, ao invés de computar $d(X \cup Y)$ como uma diferença dos *tidset*s $\mathcal{L}(P \cup X) - \mathcal{L}(P \cup Y)$, computamos como a diferença dos *diffset*s $d(P \cup Y) - d(P \cup X)$. O *tidset* de P é o universo de *tid*s relevantes. Note também que ambos $\mathcal{L}(P \cup X \cup Y)$ e $d(P \cup X \cup Y)$ são subconjuntos do *tidset* do novo prefixo $P \cup X$.

Podemos escolher começar com o conjunto original de *tidset*s para os itens freqüentes, ou poderíamos converter da representação *tidset* para representação *diffset* no início. Podemos observar claramente que, para bases de dados densas como a mostrada, uma grande redução no tamanho do banco de dados é alcançada usando esta transformação.

Se começarmos com *tidset*s, então para computar o suporte de um 2-*itemset* como $\{A, C\}$, acharíamos $d(\{A, C\}) = \mathcal{L}(A) - \mathcal{L}(C) = \{1, 3\}$. Para descobrir se $\{A, C\}$ é freqüente nós checamos $\sigma(\{A\}) - |d(\{A, C\})| = 4 - 2 = 2$, então $\{A, C\}$ não é freqüente. Se tivéssemos começado com os *diffset*s, então nós teríamos $d(\{A, C\}) = d(\{C\}) - d(\{A\}) = \{1, 3\} - \{2, 6\} = \{1, 3\}$, o mesmo resultado de antes. Mesmo o exemplo simples apresentado no Capítulo 2 ilustra o poder dos *diffset*s. As representações dos 1-*itemset*s consomem 23 elementos na abordagem baseada em *tidset*s, enquanto que somente 7 elementos na abordagem baseada em *diffset*s (3 vezes melhor). Se olharmos para o tamanho dos resultados, descobriremos que a abordagem baseada em *tidset* leva 76 *tid*s no total, enquanto a abordagem *diffset* (com dados iniciais *diffset*) armazena apenas 22 *tid*s. Se compararmos por comprimento, descobriremos que o tamanho médio de *tidset* para os 2-*itemset*s freqüentes é 3,8, enquanto o tamanho do *diffset* é 1. Para os 3-*itemset*s o tamanho do *tidset* é 3,2 mas o tamanho médio do *diffset* é 0,6. Finalmente, para 4-*tidset*s o tamanho do *tidset* é 3 e o tamanho do *diffset* é 0! O fato que a representação inicial é menor e os *diffset*s encolhem quando os *itemset*s mais longos são encontrados. Isso permite que métodos baseados em *diffset*s sejam extremamente escaláveis, e ordens de magnitude melhores que outras abordagens de mineração de associações existentes.

Os *diffset*s são empregados pelo algoritmo dEClaT [19] e pelo algoritmo zEClaT, descrito no próximo capítulo.

3.4 Projeção

A projeção é um refinamento aplicado no algoritmo FP-Growth [10], de mineração de conjuntos freqüentes, e no algoritmo MAFIA [4], de mineração de conjuntos freqüentes maximais, não aplicado no algoritmo EClaT por não apresentar nenhuma vantagem para esse algoritmo, que procura reduzir o consumo de memória trabalhando apenas sobre transações relevantes ao contexto. As transações para as quais o prefixo de uma classe de equivalência não ocorre são irrelevantes para essa classe de equivalência. As transações restantes são renumeradas para que sua representação codificada seja menor.

Os identificadores associados a cada transação são irrelevantes contanto que as transações relevantes a um contexto sejam identificadas univocamente. As transações relevantes a uma

classe de equivalência $[C]$ são as transações que contêm os átomos freqüentes de $[C]$ mas essas transações são conhecidas pelos algoritmos somente após a realização de operações.

A aplicação de qualquer função injetora p cujo domínio é um superconjunto de $\mathcal{L}(\perp([C])) \cap \left(\bigcup_{I \in \mathcal{A}([C]) \wedge \frac{\sigma(I)}{|\mathcal{D}|} \geq \frac{\sigma_{\inf}}{|\mathcal{D}|}} I\right)$ atende aos requisitos. Em particular, $\mathcal{L}(\perp([C]))$ é conhecido ao início do processamento da classe $[C]$. Se a propriedade $a \leq b \Rightarrow p(a) \leq p(b)$ é válida para p, a ordem é preservada pelo mapeamento. Seja $\iota(A) = \{1, 2, \ldots, |A|\}$ e p a função injetora de A para $\iota(A)$ que preserva a ordem. p tem uma propriedade interessante, que é $p(b) - p(a) = 1$ quando $a \sqsubset b$.

Tomando como exemplo a base de dados apresentada no Capítulo 2, o conjunto das transações que contêm o *itemset* $\{A\}$, $\mathcal{L}(\{A\})$, é $\{1, 3, 4, 5\}$. Todas as transações que contêm *itemset*s da classe de equivalência $[\{A\}]$ pertencem a esse conjunto. É possível, portanto, renumerá-las durante o processamento dessa classe de equivalência. Aplicando uma função injetora de $\{1, 3, 4, 5\}$ para $\{1, 2, 3, 4\}$ que preserve a ordem, temos $\mathcal{L}(\{A, B\})$, originalmente igual a $\{1, 3, 4, 5\}$, agora igual a $\{1, 2, 3, 4\}$ e $\mathcal{L}(\{A, C\})$, originalmente igual a $\{3, 4\}$, agora igual a $\{2, 3\}$. A interseção $\mathcal{L}(\{A, B\}) \cap \mathcal{L}(\{A, C\})$, originalmente igual a $\{3, 4\}$, agora é igual a $\{2, 3\}$. A projeção pode ser aplicada a qualquer classe de equivalência, independente de sua aplicação a outras classe de equivalência. Isso quer dizer que sua aplicação à classe de equivalência $[\{A, B\}]$ independe de sua aplicação à classe de equivalência $[\{A\}]$, transformando $\{3, 4\}$ ou $\{2, 3\}$ em $\{1, 2\}$. A corretude da projeção é garantida pela propriedade que a aplicação da função antes ou depois do cálculo das operações produz o mesmo resultado, que tem a mesma cardinalidade que teria se a projeção não fosse empregada.

O Algoritmo 3.2 detalha o refinamento de projeção. O resultado do operador refinado é o resultado do operador original projetado em U, o conjunto universo. Por exemplo, $A \cup B$ é $p(A \cup B)$, onde p é a função que projeta os elementos em $\iota(U)$. As linhas cinzas indicam as mudanças.

```
1  funct set_operation(A, o, B, U) ≡
2    while (a < |A| ∧ b < |B|) do
3          if (A[a] < B[b])
4            if (o ∈ {−, ⊖, ∪})
5              while (U[u] < A[a]) do u ← u + 1;
6              R[r] ← u;  r ← r + 1;
7            a ← a + 1;
8          else if (B[b] < A[a])
9                if (o ∈ {⊖, ∪})
10                 while (U[u] < B[b]) do u ← u + 1;
11                 R[r] ← u;  r ← r + 1;
12               b ← b + 1;
13         else
14               if (o ∈ {∩, ∪})
15                 while (U[u] < A[a]) do u ← u + 1;
16                 R[r] ← u;  r ← r + 1;
17               a ← a + 1;  b ← b + 1;
18   if (o ∈ {−, ∪, ⊖})
19     while (a < |A|) do
20           if (o ∈ {−, ⊖, ∪})
21             while (U[u] < A[a]) do u ← u + 1;
22             R[r] ← u;  r ← r + 1;
23           a ← a + 1;
24   if (o ∈ {∪, ⊖})
25     while (b < |B|) do
26           if (o ∈ {⊖, ∪})
27             while (U[u] < B[b]) do u ← u + 1;
28             R[r] ← u;  r ← r + 1;
29           b ← b + 1;
30   return R;
31 .
```

Algoritmo 3.2: Operadores de diferença, diferença simétrica, interseção e união sobre conjuntos totalmente ordenados, refinados por projeção

O algoritmo zEClaT emprega a projeção renumerando os *tid*s relevantes a uma classe de equivalência. O zEClaT é detalhado no próximo capítulo.

3.5 Reordenação dinâmica

A reordenação dinâmica é um refinamento, introduzido pelo algoritmo de mineração de conjuntos freqüentes maximais Max-Miner [11], que procura reduzir o consumo de memória de resultados intermediários alterando a ordem segundo a qual as classes de equivalência são pro-

cessadas e as operações (de diferença ou interseção) são executadas.

O resultado de uma interseção independe da ordem segundo a qual as interseções intermediárias são realizadas porque a interseção é uma operação associativa ($(A \cap B) \cap C = A \cap (B \cap C)$), mas as cardinalidades das interseções intermediárias dependem dessa ordem e a velocidade de processamento da interseção é relacionada a essas cardinalidades. Assim, a escolha adequada da ordem de realização das interseções promove maior velocidade de processamento sem invalidar o resultado. Espera-se que, realizando primeiro as interseções entre os conjuntos de menor cardinalidade, a cardinalidade dos resultados intermediários e a velocidade de processamento sejam, respectivamente, menores e maiores que para alguma outra ordem. A situação é análoga à multiplicação de matrizes.

Por exemplo, $A = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, $B = \{1, 3, 5, 7, 9\}$ e $C = \{2, 3, 5, 7\}$. Para obter $(A \cap B) \cap C = (\{1, 2, 3, 4, 5, 6, 7, 8, 9\} \cap \{1, 3, 5, 7, 9\}) \cap \{2, 3, 5, 7\} = \{1, 3, 5, 7, 9\} \cap \{2, 3, 5, 7\} = \{3, 5, 7\}$, interseções entre conjuntos de cardinalidades 9 e 5 e entre conjuntos de cardinalidade 5 e 4 são calculadas e para obter $A \cap (B \cap C) = \{1, 2, 3, 4, 5, 6, 7, 8, 9\} \cap (\{1, 3, 5, 7, 9\} \cap \{2, 3, 5, 7\}) = \{1, 2, 3, 4, 5, 6, 7, 8, 9\} \cap \{2, 3, 5, 7\} = \{3, 5, 7\}$, interseções entre conjuntos de cardinalidades 5 e 4 e entre conjuntos de cardinalidade 9 e 4 são calculadas. $A \cap (B \cap C)$ é calculada mais rapidamente que $(A \cap B) \cap C$.

O algoritmo EClaT original processa as classes de equivalência em ordem lexicográfica reversa. O processamento das classes de equivalência segundo a ordem crescente de freqüência dos *itemset*s promove maior velocidade de processamento por alterar a ordem das interseções intermediárias da *tidset* de um *itemset* e a utilização mais racional da memória porque quando as maiores classes de equivalência forem processadas, as menores classes de equivalência já o foram e a memória por elas ocupada já foi liberada.

A reordenação dinâmica é um dos refinamentos aplicados ao algoritmo zEClaT, detalhado no próximo capítulo.

3.6 *Skinning*

Skinning é uma técnica de compressão, ajustada à mineração de conjuntos freqüentes, para reduzir ainda mais o consumo de memória quando aplicada aos *bitmap*s.

À primeira vista, poderia parecer que o clássico, e simples de implementar, *Run Length Encoding (RLE)* seria a escolha apropriada para comprimir os vetores de *bit*s. Entretanto, esperamos que enquanto podem existir longas seqüências de 0's, seqüências de 1's que implicam uma seqüência de consumidores consecutivos comprando o mesmo item pode ser incomum em bancos de dados transacionais. No pior caso, onde todos os 1's ocorrem de maneira isolada, o vetor RLE conteria duas palavras para cada ocorrência de um 1 - uma palavra para a seqüência precedente de 0's e uma para o próprio 1. Isso significa que o tamanho do vetor resultante seria o dobro do original, que teria somente uma palavra associada a cada 1 (porque os 0's não são explicitamente representados). Resumindo, resultaria em expansão e não em compressão.

Shenoy et al. [14] desenvolveram uma técnica alternativa de compressão chamada *Skinning*,

baseada no esquema de codificação Golomb clássico. Seqüências de 0's e 1's são divididas em grupos de tamanho W_0 e W_1, respectivamente - os W's são referenciados por "pesos". Cada grupo completo é representado no vetor codificado por um único *bit* "peso" igual a 1. O último grupo parcial (de tamanho $R \mod W_i$, onde R é o comprimento total da seqüência) é representado por um campo de contagem que armazena o equivalente binário do comprimento restante, expresso em $\log_2 W_i$ *bit*s - por razões explicadas abaixo, este campo é armazenado mesmo se o comprimento do último grupo parcial for zero. Finalmente um *bit* 0 "separador de campo" é posicionado entre o último *bit* peso e o campo de contagem para indicar a transição do primeiro para o último. Note que um "separador de seqüência" para distinguir entre uma seqüência de 0's e uma seqüência de 1's, não é requerido porque é implicitamente conhecido que o símbolo da seqüência muda depois do campo de contagem e o número de *bit*s usado para o campo de contagem ($\log_2 W_i$) é fixo.

Para ilustrar a técnica de *Skinning*, considere o vetor de 30 *bit*s $(1)^A\ (000)^B\ (1)^C\ (0000)^D\ (0000)^E\ (0)^F\ (1)^G\ (0000)^H\ (0000)^I\ (0)^J\ (1)^K\ (0000)^L\ (0)^M$ a ser codificado usando os pesos $W_0 = 4$ e $W_1 = 1$. Os superescritos alfabéticos não são parte do vetor de *bit*s mas são incluídos para indicar os grupos associados a esta configuração de pesos. Após o *Skinning*, o vetor comprimido resultante é de 25 *bit*s $(1)^A\ 00\ (11)^B\ (1)^C\ 0\ (1)^D\ (1)^E\ 0\ (01)^F\ (1)^G\ 0\ (1)^H\ (1)^I\ 0\ (01)^J\ (1)^K\ 0\ (1)^L\ 0\ (01)^M$ onde os superescritos alfabéticos indicam correspondência entre o grupo no vetor de *bit*s original e sua versão codificada, e os *bit*s 0 não classificados são os separadores de campos.

No exemplo acima, a compressão é somente de 30 para 25 *bit*s - entretanto, para valores práticos de W_0 e W_1, taxas de compressão muito maiores são obtidas por *Skinning*. Na realidade, com escolha apropriada de W_0 e W_1, *Skinning* resulta em perto de uma ordem de magnitude de compressão do formato original para os bancos de dados considerados nos experimentos. Este alto grau de compressão é suficiente para assegurar que, embora a representação completa ocupe muito mais espaço que a esparsa, as representações obtidas através da técnica de *Skinning* são por volta de um terço do tamanho obtido com estes formatos.

A técnica de *Skinning* é aplicada pelo algoritmo VIPER [14] mas não pelo algoritmo zEClaT porque o último não emprega os *bitmap*s.

3.7 *d-gap*s

O refinamento de *d-gap*s procura reduzir o consumo de memória através de uma representação alternativa de conjuntos de números inteiros positivos. O conjunto a ser representado é ordenado e as diferenças entre os elementos consecutivos são representadas por um código para o qual números menores requerem menor espaço de armazenamento. A redução de consumo de memória é alcançada quando os números menores são suficientemente mais freqüentes que os números maiores. Apesar do uso de *d-gap*s ser amplamente difundido na área de recuperação de informação, apresentando excelentes resultados, não conhecemos nenhum trabalho que aplique essa técnica para mineração de conjuntos freqüentes, sendo essa nossa principal contribuição.

A ordem dos elementos na representação de um conjunto não importa, ou seja, o conjunto $\{1, 20, 15, 27, 5, 7\}$, representado pela seqüência $\langle 1, 20, 15, 27, 5, 7\rangle$, é igual ao conjunto totalmente ordenado $(\{1, 20, 15, 27, 5, 7\}, \leq)$, representado pela seqüência ordenada $\langle 1, 5, 7, 15, 20, 27\rangle$. As operações sobre os conjuntos A e B, representados por seqüências ordenadas, podem ser implementadas em $O(|A| + |B|)$ quanto ao processamento e $O(1)$ quanto ao armazenamento transiente. A representação de um conjunto totalmente ordenado pela seqüência ordenada correspondente requer que os elementos estejam ordenados. Os *tid*s são inseridos em ordem nos *tidset*s dos itens, portanto, tal requisito é atendido sem custo de processamento adicional.

Quando as seqüências ordenadas são processadas seqüencialmente a partir do primeiro elemento, elas podem ser representadas pelas diferenças adjacentes, chamadas de *d-gap*s, e recuperadas pelas somas parciais. Apresentamos agora, formalmente, as relações entre os elementos p_i da seqüência ordenada e os elementos a_i da seqüência de *d-gap*s.

$$a_i = p_i - p_{i-1}$$

$$p_i = \sum_{j=1}^{i} a_j$$

A seqüência ordenada $\langle 1, 5, 7, 15, 20, 27\rangle$, por exemplo, pode ser representada pela seqüência de *d-gap*s $\langle 1, 4, 2, 8, 5, 7\rangle$. As duas formas são equivalentes.

Nenhuma compressão foi atingida ainda. O maior *d-gap* na segunda representação ainda é potencialmente igual ao maior elemento na primeira representação. Se uma codificação binária é usada para representar os *d-gap*s e existem N transações, ambos métodos requerem $\lceil \log N \rceil$ dígitos binários por *tid* armazenado. A compressão é atingida apenas quando utilizamos um código cuja redução do comprimento dos *d-gap*s mais freqüentes compensa o aumento do comprimento dos *d-gap*s menos freqüentes.

As probabilidades $p_{n,p}(x)$ de *d-gap*s iguais a x para a uma seqüência ordenada de p inteiros positivos escolhidos aleatoriamente com distribuição uniforme no intervalo $[1, n]$ são iguais a:

$$p_{n,p}(x) = \frac{\begin{pmatrix} n-x \\ p-1 \end{pmatrix}}{\begin{pmatrix} n \\ p \end{pmatrix}}$$

A Figura 3.1 mostra as probabilidades $p_{16,y}(x)$ (eixo probabilidade) de *d-gap*s iguais a x (eixo diferença adjacente) para uma seqüência ordenada de y (eixo tamanho da seqüência) inteiros positivos escolhidos aleatoriamente com distribuição uniforme no intervalo $[1, 16]$. Observe que, à medida em que o tamanho da seqüência aumenta, as diferenças adjacentes menores tornam-se mais prováveis e as maiores tornam-se menos prováveis.

Assim, assumindo que os *tid*s de um *tidset* seguem uma distribuição uniforme, podemos dizer que, para os *itemset*s mais freqüentes, são mais adequados os códigos que privilegiam os números menores em detrimento aos números maiores. A premissa pode parecer muito forte, mas, como

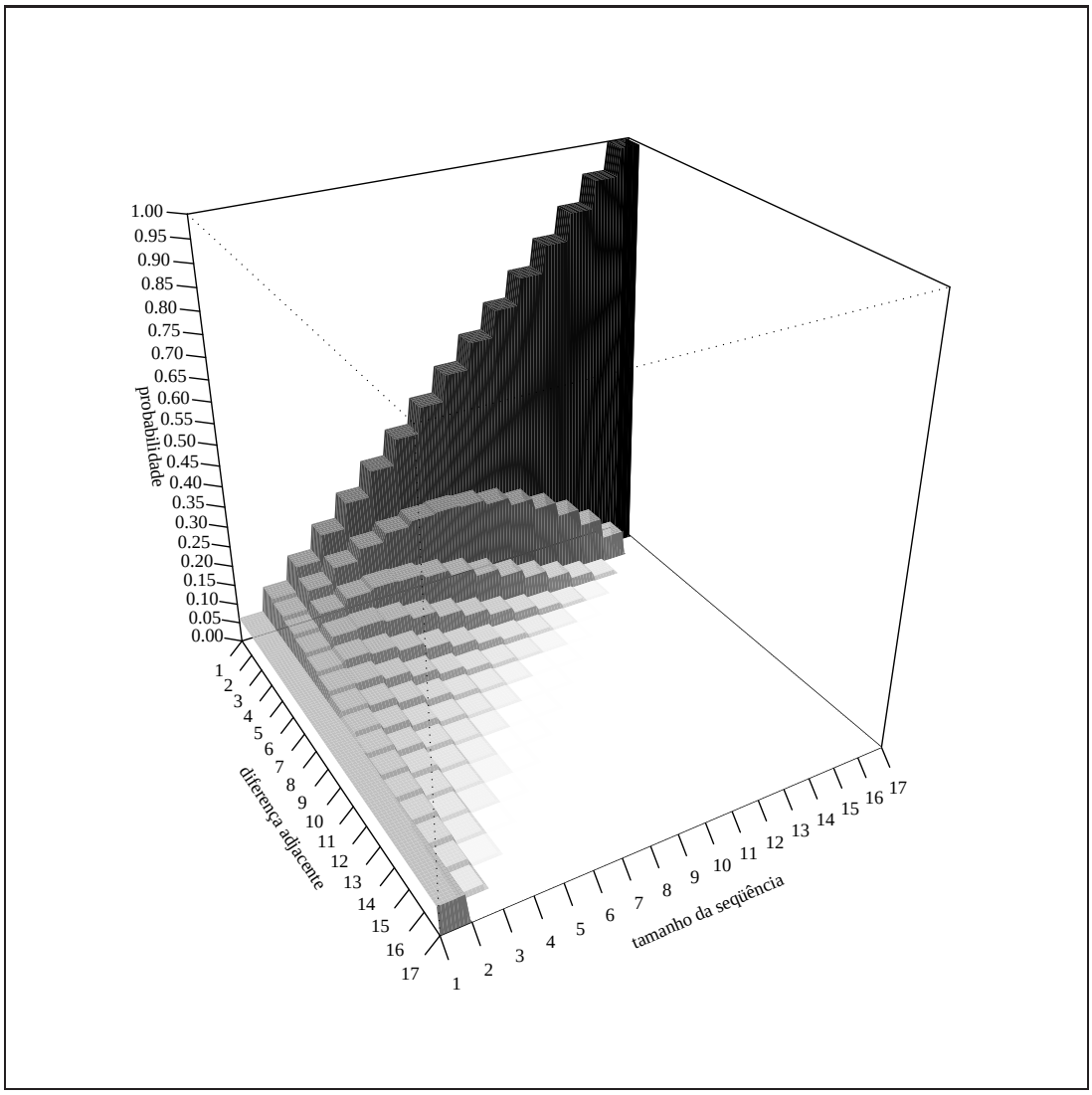
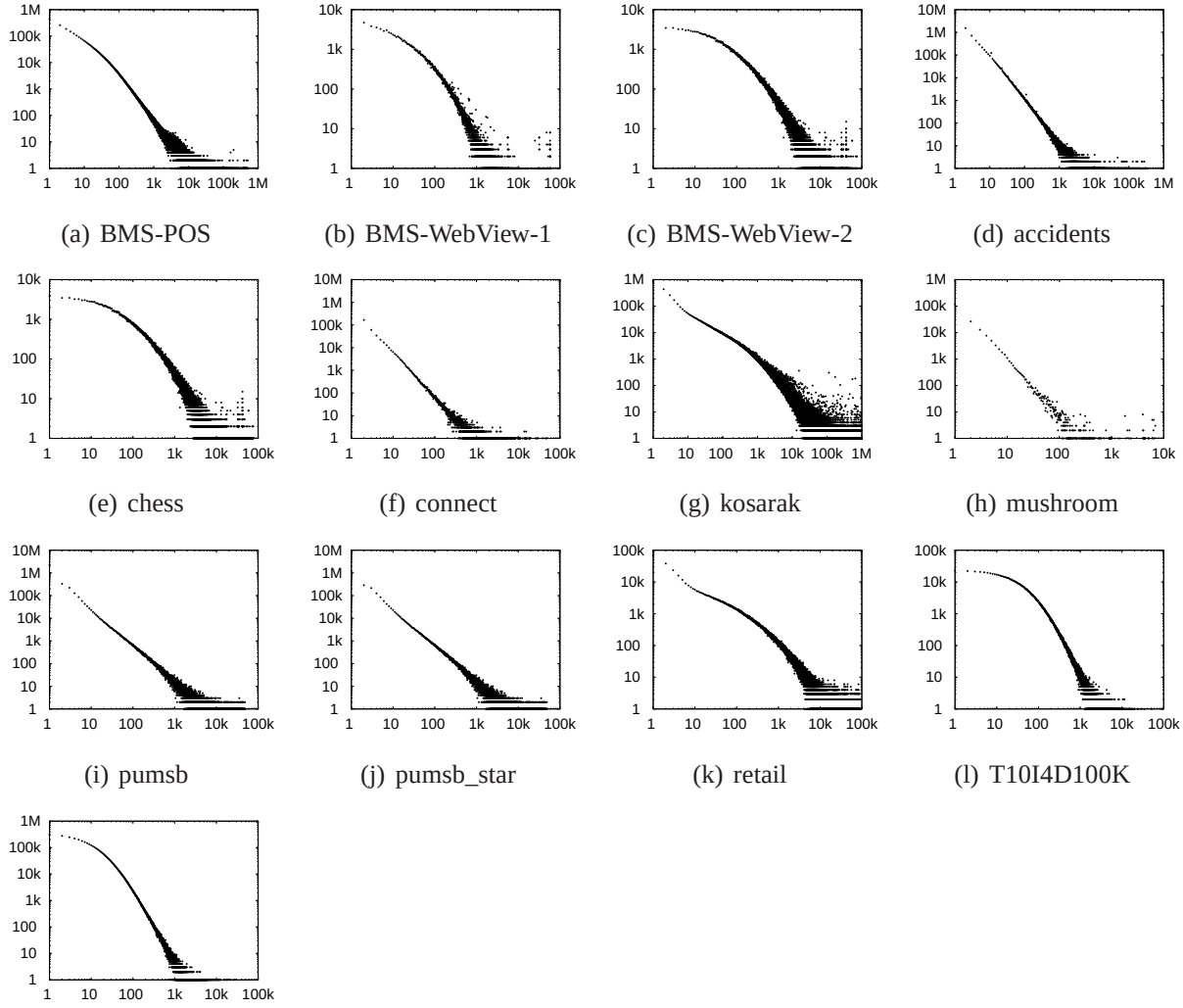


Figura 3.1: Probabilidades dos *d-gap*s para uma seqüência ordenada de inteiros positivos no intervalo $[1, 16]$



(a) BMS-POS (b) BMS-WebView-1 (c) BMS-WebView-2 (d) accidents

(e) chess (f) connect (g) kosarak (h) mushroom

(i) pumsb (j) pumsb_star (k) retail (l) T10I4D100K

(m) T40I10D100K

Figura 3.2: Freqüência dos *d-gap*s dos itens de algumas bases de dados

a ordem das transações não importa para o problema, mediante a uma permutação aleatória das transações, ela é razoável.

Os gráficos da Figura 3.2 apresentam para os valores dos *d-gap*s dos *tidset*s dos itens, no eixo das abscissas, as freqüências correspondentes, no eixo das ordenadas, para diversas bases de dados. Observe que *d-gap*s pequenos são freqüentes e *d-gap*s grandes são raros, o que valida a estratégia de codificação de *d-gap*s para redução de consumo de memória, mesmo que a distribuição dos *tid*s nos *tidset*s não seja uniforme.

Sumário Nesse capítulo apresentamos várias estratégias de redução de consumo de memória aplicáveis aos algoritmos de mineração de conjuntos freqüentes. Algumas delas serão empregadas pelo algoritmo zEClaT e avaliadas nesse novo contexto no próximo capítulo.

Capítulo 4

O Algoritmo zEClaT

Nesse capítulo apresentamos o algoritmo zEClaT e os parâmetros que alteram seu comportamento e ainda os experimentos e a análise dos resultados. O algoritmo zEClaT, a especialização do algoritmo EClaT que representa um conjunto por uma seqüência de diferenças entre elementos consecutivos, é uma das contribuições deste trabalho.

Cinco parâmetros alteram o comportamento do algoritmo zEClaT. Os parâmetros especificam o código com o qual as diferenças entre elementos consecutivos serão codificadas e quais refinamentos, entre curto-circuito, *diffset*s, projeção e reordenação dinâmica, serão utilizados. Observe que o $\text{zEClaT}_{\text{binário}}$ é equivalente ao EClaT e o $\text{zEClaT}_{\text{binário},\textit{diffsets}}$ é equivalente ao dEClaT (EClaT com *diffset*s). As combinações de parâmetros que incluem os refinamentos de *diffset*s e projeção não foram implementadas porque esses refinamentos em suas formas originais não são ortogonais.

As Seções 4.1, 4.2, 4.3, 4.4 e 4.5 explicam os parâmetros e refinamentos implementados para o algoritmo zEClaT, respectivamente, os códigos para representação das diferenças entre elementos consecutivos dos conjuntos de números inteiros, o curto-circuito, refinamento que melhora o desempenho abortando antecipadamente operações quando seus resultados são irrelevantes, os *diffset*s, refinamento que armazena diferenças entre conjuntos similares, a projeção, refinamento que renumera os identificadores das transações para que as diferenças entre elementos consecutivos sejam menores e ocupem menos espaço, e a reordenação dinâmica, refinamento que reordena as operações para uma utilização mais racional da memória. Na Seção 4.6, explicamos que a complexidade computacional do problema é preservada por nosso algoritmo mesmo com os diversos refinamentos. Finalmente, as descrições das bases de dados, do ambiente de execução, dos experimentos e os resultados experimentais são os assuntos da Seção 4.7.

4.1 Código

Um código é um sistema de símbolos com que se representam dados para serem processados por computador. Na implementação do zEClaT, os códigos disponíveis são o Binário, o Elias δ,

o Elias γ, o Fibonacci e o Unário, que são explicados abaixo e exemplificados na Tabela 4.1.

Binário O código Binário é um código posicional para representar números inteiros não-negativos. O valor do i-ésimo dígito binário é igual a 2^i, ou seja, o primeiro dígito binário vale 2^1, o segundo, 2^2, e assim sucessivamente. A delimitação do código Binário é implícita porque todas as representações ocupam a mesma quantidade de dígitos binários. O número 81, por exemplo, é representado no código Binário pela seqüência 1010001 porque $81 = 1 \times 2^0 + 0 \times 2^1 + 0 \times 2^2 + 0 \times 2^3 + 1 \times 2^4 + 0 \times 2^5 + 1 \times 2^6$.

A quantidade de dígitos binários necessária para representar o número inteiro não-negativo $n \in [0, n_{\max}]$ no código Binário é igual a:

$$\lceil \log_2 (n_{\max}) \rceil$$

Unário Para ler um número natural positivo n em código Unário, leia a dígitos binários iguais a zero e 1 dígito binário igual a um, $n = a + 1$. Para escrever um número natural positivo n em código Unário, escreva $n - 1$ dígitos binários iguais a zero e 1 dígito binário igual a um.

A quantidade de dígitos binários necessária para representar o número inteiro positivo n no código Unário é igual a:

$$n$$

Elias γ [5] Para ler um número natural positivo n em código Elias γ, leia a em código Unário e b em código Binário de $a - 1$ dígitos, $n = 2^a + b$. Para escrever um número natural positivo n em código Elias γ, escreva $\lfloor \log_2 n \rfloor + 1$ em código Unário e $n - 2^{\lfloor \log_2 n \rfloor}$ em código Binário de $\lfloor \log_2 n \rfloor$ dígitos.

A quantidade de dígitos binários necessária para representar o número inteiro positivo n no código Elias γ é igual a:

$$2 \lfloor \log_2 n \rfloor + 1$$

Elias δ [5] Para ler um número natural positivo n em código Elias δ, leia a em código Elias γ e b em código Binário de $a - 1$ dígitos, $n = 2^a + b$. Para escrever um número natural positivo n em código Elias δ, escreva $\lfloor \log_2 n \rfloor + 1$ em código Elias γ e $n - 2^{\lfloor \log_2 n \rfloor}$ em código Binário de $\lfloor \log_2 n \rfloor$ dígitos.

A quantidade de dígitos binários necessária para representar o número inteiro positivo n no código Elias δ é igual a:

$$\lfloor \log_2 n \rfloor + 2 \lfloor \log_2 (\log_2 n + 1) \rfloor + 1$$

Fibonacci Os números de Fibonacci formam uma seqüência definida recursivamente pelas seguintes equações:

$$F_n = \begin{cases} 1 & , n = 1 \\ 1 & , n = 2 \\ F_{n-1} + F_{n-2} & , n \geq 3 \end{cases}$$

O código de Fibonacci é um código posicional onde o valor do i-ésimo dígito binário é igual ao $(i+1)$-ésimo número de Fibonacci, ou seja, o primeiro dígito binário vale $F_2 = 1$, o segundo, $F_3 = 2$, e assim sucessivamente. Nenhuma representação possui dois dígitos binários consecutivos iguais a 1. Essa característica permite que o delimitador do código de Fibonacci seja um dígito binário igual a 1 seguindo o dígito binário mais significativo. O número 12, por exemplo, é representado no código de Fibonacci pela seqüência $10101\ 1$ porque $12 = 1 \times F_2 + 0 \times F_3 + 1 \times F_4 + 0 \times F_5 + 1 \times F_6$.

A quantidade de dígitos binários necessária para representar o número inteiro positivo n no código de Fibonacci é igual a:

$$\left\lfloor \log_{\frac{1+\sqrt{5}}{2}} \text{round}\left[(n + 1/2)\sqrt{5} \right] \right\rfloor$$

A Tabela 4.1 mostra os códigos Binário, Elias δ, Elias γ, Fibonacci e Unário para os inteiros positivos n no intervalo $[0, 16]$. Os índices representam a quantidade de vezes que uma seqüência delimitada por parênteses se repete. O símbolo $\nexists$ indica que uma representação não existe. Não existe representação para o número 0 nos códigos Elias δ, Elias γ, Fibonacci e Unário, assim como não existe representação para números maiores ou iguais a $2^{\lceil \log_2(n_{\max}) \rceil}$ no código Binário.

n	Binário	Elias δ	Elias γ	Fibonacci	Unário
0	$(0)_{\lceil \log_2(n_{\max}) \rceil}$	$\nexists$	$\nexists$	$\nexists$	$\nexists$
1	$(0)_{\lceil \log_2(n_{\max}) \rceil - 1}\,1$	1	1	11	1
2	$(0)_{\lceil \log_2(n_{\max}) \rceil - 2}\,10$	0100	010	011	01
3	$(0)_{\lceil \log_2(n_{\max}) \rceil - 2}\,11$	0101	011	0011	$(0)_2\,1$
4	$(0)_{\lceil \log_2(n_{\max}) \rceil - 3}\,100$	01100	00100	1011	$(0)_3\,1$
5	$(0)_{\lceil \log_2(n_{\max}) \rceil - 3}\,101$	01101	00101	00011	$(0)_4\,1$
6	$(0)_{\lceil \log_2(n_{\max}) \rceil - 3}\,110$	01110	00110	10011	$(0)_5\,1$
7	$(0)_{\lceil \log_2(n_{\max}) \rceil - 3}\,111$	01111	00111	01011	$(0)_6\,1$
8	$(0)_{\lceil \log_2(n_{\max}) \rceil - 4}\,1000$	00100000	0001000	000011	$(0)_7\,1$
9	$(0)_{\lceil \log_2(n_{\max}) \rceil - 4}\,1001$	00100001	0001001	100011	$(0)_8\,1$
10	$(0)_{\lceil \log_2(n_{\max}) \rceil - 4}\,1010$	00100010	0001010	010011	$(0)_9\,1$
11	$(0)_{\lceil \log_2(n_{\max}) \rceil - 4}\,1011$	00100011	0001011	001011	$(0)_{10}\,1$
12	$(0)_{\lceil \log_2(n_{\max}) \rceil - 4}\,1100$	00100100	0001100	101011	$(0)_{11}\,1$
13	$(0)_{\lceil \log_2(n_{\max}) \rceil - 4}\,1101$	00100101	0001101	0000011	$(0)_{12}\,1$
14	$(0)_{\lceil \log_2(n_{\max}) \rceil - 4}\,1110$	00100110	0001110	1000011	$(0)_{13}\,1$
15	$(0)_{\lceil \log_2(n_{\max}) \rceil - 4}\,1111$	00100111	0001111	0100011	$(0)_{14}\,1$
16	$(0)_{\lceil \log_2(n_{\max}) \rceil - 5}\,10000$	001010000	000010000	0010011	$(0)_{15}\,1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\geq 2^{\lceil \log_2(n_{\max}) \rceil}$	$\nexists$	$\exists$	$\exists$	$\exists$	$\exists$

Tabela 4.1: Exemplos dos códigos Binário, Elias δ, Elias γ, Fibonacci e Unário

O Algoritmo 2.2 é alterado da seguinte forma. A linha 11, correspondente ao cálculo das junções, agora se torna a linha 12 do Algoritmo 4.1, armazenando nos conjuntos, em lugar dos elementos em forma nativa, as diferenças adjacentes dos elementos, codificadas com o código escolhido através de um parâmetro do algoritmo zEClaT.

4.2 Curto-circuito

O curto-circuito, detalhado na Seção 4.2, é um refinamento que aborta um operador sobre conjuntos quando antecipa que o resultado será irrelevante, procurando evitar processamento desnecessário. O resultado de um operador é irrelevante para um algoritmo de mineração de conjuntos freqüentes quando o *itemset* é infreqüente, ou seja, a cardinalidade da interseção entre os *tidset*s é menor que o limite inferior de freqüência ou a soma da freqüência do *itemset* prefixo e a cardinalidade da diferença é maior que o limite inferior de freqüência.

O Algoritmo 2.2 é alterado da seguinte forma. A linha 11 passa a utilizar os operadores refinados por curto-circuito detalhados no Algoritmo 3.1 e se torna a linha 12 do Algoritmo 4.1. O operador utilizado é o de interseção, no cálculo dos *tidset*s, ou o de diferença, no cálculo dos *diffset*s. O parâmetro $\inf$ para a interseção é o limite inferior de freqüência. O parâmetro $\sup$ para a diferença é o limite inferior de suporte menos a freqüência do *itemset* prefixo.

4.3 *diffset*s

Os *diffset*s, detalhados na Seção 4.3, são um refinamento que armazena para um *itemset* a sua freqüência e o seu *diffset*, a diferença entre o seu *tidset* e o *tidset* do *itemset* prefixo, em detrimento ao seu *tidset*.

O Algoritmo 2.2 é alterado da seguinte forma. A linha 11, que calcula as junções por meio de interseções, agora passa a calculá-las por meio de diferenças, se tornando a linha 12 do Algoritmo 4.1. As junções entre 1-*itemset*s são calculadas pelas diferenças entre os *tidset*s. As demais junções são calculadas pelas diferenças entre *diffset*s. A ordem dos argumentos da diferença é importante. Quanto ao *itemset* prefixo, seu *tidset* deve ser o primeiro argumento da diferença entre *tidset*s e seu *diffset* deve ser o segundo argumento da diferença entre *diffset*s.

4.4 Projeção

A projeção, detalhada na Seção 4.4, é um refinamento que renumera os *tid*s relevantes a uma classe de equivalência de forma que as diferenças entre os valores consecutivos dos elementos dos *tidset*s dos *itemset*s dessa classe sejam menores. Quando as diferenças são menores o consumo de memória é menor. Os *tid*s relevantes a uma classe de equivalência de prefixo P são os *tid*s das transações que contêm P, que são renumeradas em seqüência por 1, 2, ...

O Algoritmo 2.2 é alterado da seguinte forma. A linha 11, que calcula as junções utilizando interseções, agora passa a utilizar o operador refinado por projeção detalhado no Algoritmo 3.2 com o parâmetro U igual a *itemsets*$[$*maior*$]$, se tornando a linha 12 do Algoritmo 4.1.

4.5 Reordenação dinâmica

A reordenação dinâmica, detalhada na Seção 4.5, é um refinamento que altera a ordem em que as junções entre *itemset*s são realizadas. Os *itemset*s menos freqüentes são processados pelo algoritmo antes dos *itemset*s mais freqüentes. A tendência é que os resultados intermediários sejam menores e que quando os *itemset*s mais freqüentes forem processados a memória alocada pelos *itemset*s menos freqüentes já tenha sido liberada.

O Algoritmo 2.2 é alterado da seguinte forma. Um comando que ordena os 1-*itemset*s freqüentes é acrescentado ao final da linha 5, se tornando a linha 6 do Algoritmo 4.1, e outro comando que ordena os demais *itemset*s freqüentes de cada classe de equivalência ao final da linha 16, se tornando a linha 18 do Algoritmo 4.1.

4.6 Manutenção da Complexidade Computacional

Nesta seção apresentamos o problema de mineração de conjuntos freqüentes visto por outra perspectiva e sua complexidade computacional e explicitamos a manutenção da complexidade

```
1  funct zEClaT (um, Σ_inf) ≡
2    foreach itemset ∈ um do
3      if Σ(itemset) ≥ Σ_inf
4        itemsets.push (itemset) ; F.push (itemset) ;
5    classes.push ([1, |itemsets|]) ;
6    sort_rd(itemsets, 1, |itemsets|);
7    while true do
8          menor ← classes.back () .menor; maior ← classes.back () .maior;
9          if menor < maior
10           adicionar ← false;
11           while menor < maior do
12                 itemset ← itemsets [menor] ϒ_{c,c−c,d,p} itemsets [maior] ;
13                 if Σ(itemset) ≥ Σ_inf
14                   adicionar ← true; itemsets.push (itemset) ; F.push (itemset) ;
15                 menor ← menor + 1;
16           if adicionar
17             classes.push ([classes.back () .maior + 1, |itemsets|]) ;
18             sort_rd(itemsets, classes.back () .maior + 1, |itemsets|);
19           else
20               classes.back () .maior ← classes.back () .maior − 1; itemsets.pop () ;
21         else
22             classes.pop () ; itemsets.pop () ;
23             if classes.empty () return F;
24             classes.back () .maior ← classes.back () .maior − 1; itemsets.pop () ;
25 .
```

Algoritmo 4.1: zEClaT não-recursivo

computacional do problema apesar dos diversos refinamentos dos algoritmos.

Um grafo bipartido $G = (V_1, V_2, E)$ tem dois conjuntos disjuntos de vértices V_i e V_t, e um conjunto de arestas $E = \{(v_i, v_t) \,|v_i \in V_i \wedge v_t \in V_t\}$. Um subgrafo bipartido completo $I \times T$, chamado clique bipartido, é denotado por $K_{i,t}$, onde $|I| = i$, $|T| = t$, $I \subseteq V_i$ e $T \subseteq V_t$.

A entrada para o problema de mineração de conjuntos freqüentes é essencialmente um grafo bipartido, com V_i como o conjunto de itens, V_t como o conjunto de transações, e cada par ordenado (item, transação) como uma aresta. O problema de mineração de conjuntos freqüentes corresponde à enumeração de cliques bipartidos restritos, $K_{i,t}$, com $\frac{t}{|\mathcal{D}|} \geq \sigma_{\text{inf}}$ [18].

O espaço de busca para enumeração de todos os *itemset*s é $2^{|\mathcal{I}|}$, que é exponencial em $\mathcal{I}$. Entretanto, se assumirmos que existe um limite l no tamanho da transação, a tarefa de encontrar todos os *itemset*s freqüentes é essencialmente linear no tamanho do banco de dados, pois a complexidade global neste caso é dada por $O(m \times |\mathcal{D}| \times 2^l)$, onde m é o número de itemsets freqüentes maximais [17].

Nenhum dos refinamentos altera a complexidade computacional do problema de mineração de conjuntos freqüentes. São apenas estratégias que, na prática, dependendo de características da entrada, podem reduzir por um fator polinomial o tempo de execução.

4.7 Resultados Experimentais

Nesta seção, avaliamos empiricamente as efetividades dos refinamentos para diversos parâmetros e bases de dados.

4.7.1 Bases de dados

As bases de dados utilizadas foram accidents, BMS-POS, BMS-WebView-1, BMS-WebView-2, chess, connect, kosarak, mushroom, pumsb, pumsb_star, retail, T10I4D100K e T40I10D100K, disponíveis publicamente em `http://fimi.cs.helsinki.fi/data/`.

Accidents [8] A base de dados accidents foi doada por Karolien Geurts e contém dados anônimos de acidentes automobilísticos;

BMS-POS, BMS-WebView-1 e BMS-WebView-2 [12] As bases de dados BMS-POS, BMS-WebView-1 e BMS-WebView-2 contêm dados de seqüências de cliques e compras do varejista virtual de vestuário e medicamentos para pernas Gazzele.com, que fechou em 18/08/2000;

Chess A base de dados chess descreve o fim-de-jogo de uma partida de xadrez entre rei e torre brancos contra rei e peão em a7 pretos (usualmente abreviado KRKPA7). O peão em a7 significa que a promoção de peão para rainha está a um movimento. O jogador com as brancas deve movimentar uma peça. O formato para instâncias nessa base de dados é uma seqüência de 37 valores de atributos. Os 36 primeiros atributos descrevem o tabuleiro e trigésimo sétimo a classificação “branco pode vencer” ou “branco não pode vencer”;

Connect A base de dados connect contém posições legais do jogo de *connect-4* nas quais nenhum jogador ganhou ainda, e nas quais o próximo movimento é não-forçado;

Mushroom A base de dados mushroom inclui descrições de exemplos hipotéticos correspondentes a 23 espécies de cogumelos com hifas dos gêneros *Agaricus* e *Lepiota*. Cada espécie é identificada como definitivamente comestível, definitivamente venenosa, ou desconhecida e não recomendada. A última classe foi combinada com a venenosa;

Kosarak A base de dados kosarak foi provida por Ferenc Bodon e contém dados anônimos de seqüências de cliques de um portal de notícias *on-line* húngaro;

Pumsb e pumsb_star As bases de dados pumsb e pumsb_star contém informações pessoais e habitacionais dos questionários distribuídos para uma amostra da população do censo norte-americano de 1980. As variáveis habitacionais incluem propriedade, ano de construção, número e tipos de quartos, encanamento, equipamentos de aquecimento, custos de taxas e hipoteca,

número de crianças e renda familiar. As variáveis pessoais incluem sexo, idade, estado civil, origem espanhola, renda, profissão, meio de transporte ao trabalho e escolaridade;

Retail [3] A base de dados retail foi doada por Tom Brijs e contém dados anônimos de cestas de compra de uma loja de varejo belga;

T10I4D100K e T40I10D100K As bases de dados T10I4D100K e T40I10D100K foram geradas usando o gerador do grupo de pesquisa IBM Almaden Quest.

As características dessas bases de dados são mostradas pela Tabela 4.2.

Arquivo	*Bytes*	Média de itens por transação	Itens	Transações
accidents	35.509.823	33,8	468	340.183
BMS-POS	10.869.129	6,5	1.657	515.597
BMS-WebView-1	897.834	2,5	497	59.602
BMS-WebView-2	2.239.752	4,6	3.340	77.512
chess	339.098	37	75	3.196
connect	9.187.752	43	129	67.557
kosarak	32.029.467	8,1	41.270	990.002
mushroom	562.284	23	119	8.124
pumsb	16.640.715	74	2.113	49.046
pumsb_star	11.242.868	50,5	2.088	49.046
retail	4.167.490	10,3	16.470	88.162
T10I4D100K	3.922.055	10,1	870	100.000
T40I10D100K	15.378.113	39,6	942	100.000

Tabela 4.2: Características das bases de dados

4.7.2 Ambiente de execução

O algoritmo zEClaT foi implementado na linguagem C++ e utiliza classes da STL (*Standard Template Library*). Foram utilizados o compilador g++, versão 3.4.3, do projeto GNU, e os parâmetros de compilação `-march=i686`, que gera instruções ajustadas para os processadores Intel Pentium II, III e IV, e `-O3`, que habilita otimizações, e `-Wall`, que habilita todos os avisos sobre construções questionáveis fáceis de evitar.

O microcomputador utilizado possui processador Intel Celeron (Coppermine), de 565MHz de freqüência com 128kB de *cache*, 255.708kB de memória primária, 10,24GB de memória secundária com 512B de *cache* em um disco rígido Seagate ST310212A, 748.432kB de *swap* e Linux *kernel* 2.4.27.

Um programa executa a implementação do algoritmo zEClaT para combinações de entradas e parâmetros. Qualquer execução que ultrapasse 5 minutos de tempo de execução é abortada para que a coleção de execuções seja concluída em tempo hábil. A entrada é lida de um arquivo, a saída é escrita para outro arquivo. O consumo de memória, o tempo de execução e outras medidas são registrados pelo próprio algoritmo em um terceiro arquivo.

4.7.3 Experimentos

Realizamos um conjunto de experimentos com o objetivo de avaliar empiricamente a efetividade do zEClaT e seus refinamentos.

Medimos o consumo de memória e o tempo de execução após a realização de cada junção de *itemset*s. Para algumas bases de dados e limites inferiores de suporte, escolhidos para cada base de dados de forma que o tempo de execução fosse próximo do limite de tempo estipulado para os processos de cinco minutos, o consumo de memória máximo é mostrado nas Tabelas 4.3 e 4.4 e o tempo de execução total é mostrado nas Tabelas B.1 e B.2. A primeira coluna exibe a configuração, na ordem, código, *diffset*s, projeção, curto-circuito e reordenação dinâmica. Os símbolos d, p, c e r representam, respectivamente, os refinamentos *diffset*, projeção, curto-circuito e reordenação dinâmica. Uma linha horizontal sobre o símbolo representa refinamento desabilitado. A primeira linha exibe a base de dados e o limite inferior de suporte. As demais células exibem o consumo de memória máximo (Tabelas 4.3 e 4.4), em *byte*s, ou o tempo de execução total (Tabelas B.1 e B.2), em segundos. Os gráficos das Figuras 4.1, 4.2, 4.3, 4.4 e 4.5 mostram, para a base de dados chess e limite inferior de suporte de 80%, o efeito, respectivamente, do código, dos *diffset*s, da projeção, do curto-circuito e da reordenação dinâmica, no consumo de memória e no tempo de execução. Os gráficos à esquerda mostram o consumo de memória em função do tempo de execução e os gráficos à direita mostram o consumo de memória em função da quantidade de junções de *itemset*s.

Para avaliar a efetividade da alteração de um parâmetro de configuração p, do valor v_0 para o valor v_1, quanto à medida M, calculamos a média geométrica das razões entre as medidas M válidas das configurações $c_{i,p=v_1}$ e $c_{i,p=v_0}$. Os parâmetros das configurações $c_{i,p=v_1}$ e $c_{i,p=v_0}$ são iguais, à exceção do parâmetro p, que nas configurações $c_{i,p=v_1}$ assume o valor v_1 e nas configurações $c_{i,p=v_0}$ assume o valor v_0.

$$m = \sqrt[n]{\prod_{i=1}^{n} \frac{M\left(c_{i,p=v_1}\right)}{M\left(c_{i,p=v_0}\right)}}$$

Sobre a alteração do parâmetro p, do valor v_0 para o valor v_1, quando $m > 1$, dizemos que ela aumenta a medida M em $\lfloor (m-1) \rfloor$ vezes ou $\lfloor 100 \times (m-1) \rfloor$% e quando $m < 1$, dizemos que ela reduz a medida M em $\lfloor 100 \times (1-m) \rfloor$%.

Em relação ao consumo de memória máximo, para a base de dados accidents e limite inferior de suporte 90%, os códigos Elias γ, Elias δ, de Fibonacci e Unário o reduziram em, respectivamente, 89%, 88%, 87% e 55%, em relação ao código Binário. A habilitação dos *diffset*s o reduziu em 10%. As habilitações da projeção e da reordenação dinâmica o reduziram em menos de 1%.

Já para a base de dados BMS-POS e limite inferior de suporte 1%, foi pequena a quantidade de configurações que respeitaram o limite de trezentos segundos de tempo de execução. O consumo de memória máximo foi reduzido em 16% pelas habilitações da reordenação dinâmica e da projeção e em 9% pelo código de Fibonacci em relação ao código Elias δ.

Para a base de dados BMS-WebView-1 e limite inferior de suporte 0,1%, a habilitação da

reordenação dinâmica e a desabilitação dos *diffset*s reduziram o consumo de memória máximo em, respectivamente 85% e 80%, e os códigos de Fibonacci, Elias δ e Elias γ o reduziram em, respectivamente, 49%, 43% e 40%, em relação ao código Binário. Finalmente, a habilitação da projeção o reduziu em 17%.

Quanto à base de dados BMS-WebView-2 e limite inferior de suporte 0,2%, os códigos de Fibonacci, Elias δ e Elias γ reduziram o consumo de memória máximo em, respectivamente, 41%, 35% e 28%, em relação ao código Binário. A habilitação da reordenação dinâmica, a desabilitação dos *diffset*s e a habilitação da projeção o reduziram em, respectivamente, 26%, 15% e 10%.

O consumo de memória máximo, para a base de dados chess e limite inferior de suporte 80%, foi reduzido em, respectivamente, 87%, 87%, 80% e 75%, pelos códigos Elias γ, Elias δ, de Fibonacci e Unário, em relação ao código Binário. A habilitação dos *diffset*s, da reordenação dinâmica e da projeção o reduziram em, respectivamente, 86%, 68% e 6%.

A maior redução do consumo de memória máximo foi obtida para a base de dados connect e limite inferior de suporte 98%. Os códigos Elias γ, Elias δ, Unário e de Fibonacci o reduziram em, respectivamente, 93%, 93%, 89% e 87%. A habilitação dos *diffset*s e da reordenação dinâmica o reduziram em, respectivamente, 61% e 56%.

Apenas os códigos fizeram diferença para a base de dados kosarak e limite inferior de suporte 1%. As configurações de código Binário e Unário não respeitaram o limite de trezentos segundos de tempo de execução. Os códigos de Fibonacci e Elias δ reduziram o consumo de memória máximo em, respectivamente, 19% e 13%, em relação ao código Elias γ.

Configuração					accidents	BMS-POS	BMS-WebView-1	BMS-WebView-2	chess	connect	kosarak
					90%	1%	0,1%	0,2%	80%	98%	1%
Binário	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	43,1M	-	1,08M	-	12,3M	45,3M	-
Binário	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	r	42,9M	-	292k	-	4,76M	13,3M	-
Binário	$\overline{\text{d}}$	$\overline{\text{p}}$	c	$\overline{\text{r}}$	43,1M	-	1,08M	1,05M	12,3M	45,3M	-
Binário	$\overline{\text{d}}$	$\overline{\text{p}}$	c	r	42,9M	-	292k	743k	4,76M	13,3M	-
Binário	$\overline{\text{d}}$	p	$\overline{\text{c}}$	$\overline{\text{r}}$	43,1M	-	1,08M	-	12,3M	45,3M	-
Binário	$\overline{\text{d}}$	p	$\overline{\text{c}}$	r	42,9M	-	292k	-	4,76M	13,3M	-
Binário	$\overline{\text{d}}$	p	c	$\overline{\text{r}}$	43,1M	-	1,08M	1,05M	12,3M	45,3M	-
Binário	$\overline{\text{d}}$	p	c	r	42,9M	-	292k	743k	4,76M	13,3M	-
Binário	d	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	26,0M	-	21,6M	-	767k	5,88M	-
Binário	d	$\overline{\text{p}}$	$\overline{\text{c}}$	r	26,0M	-	626k	-	173k	5,88M	-
Binário	d	$\overline{\text{p}}$	c	$\overline{\text{r}}$	26,0M	-	21,6M	1,80M	767k	5,88M	-
Binário	d	$\overline{\text{p}}$	c	r	26,0M	-	626k	743k	173k	5,88M	-
Elias δ	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	4,04M	-	838k	731k	1,21M	2,74M	-
Elias δ	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	r	4,04M	-	179k	541k	485k	831k	-
Elias δ	$\overline{\text{d}}$	$\overline{\text{p}}$	c	$\overline{\text{r}}$	4,04M	4,34M	838k	731k	1,21M	2,74M	11,1M
Elias δ	$\overline{\text{d}}$	$\overline{\text{p}}$	c	r	4,04M	3,06M	179k	541k	485k	831k	11,1M
Elias δ	$\overline{\text{d}}$	p	$\overline{\text{c}}$	$\overline{\text{r}}$	4,04M	-	521k	-	1,10M	2,70M	11,1M
Elias δ	$\overline{\text{d}}$	p	$\overline{\text{c}}$	r	4,04M	-	179k	-	416k	811k	11,1M
Elias δ	$\overline{\text{d}}$	p	c	$\overline{\text{r}}$	4,04M	3,06M	521k	541k	1,10M	2,70M	11,1M
Elias δ	$\overline{\text{d}}$	p	c	r	4,04M	3,06M	179k	541k	416k	811k	11,1M

Configuração					accidents	BMS-POS	BMS-WebView-1	BMS-WebView-2	chess	connect	kosarak
					90%	1%	0,1%	0,2%	80%	98%	1%
Elias δ	d	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	4,04M	-	8,39M	-	216k	564k	-
Elias δ	d	$\overline{\text{p}}$	$\overline{\text{c}}$	r	4,04M	-	387k	-	37,3k	564k	-
Elias δ	d	$\overline{\text{p}}$	c	$\overline{\text{r}}$	4,04M	-	8,39M	973k	216k	564k	11,1M
Elias δ	d	$\overline{\text{p}}$	c	r	4,04M	-	387k	541k	37,3k	564k	11,1M
Elias γ	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	3,75M	-	941k	-	1,17M	2,71M	12,9M
Elias γ	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	r	3,75M	-	193k	-	467k	823k	12,9M
Elias γ	$\overline{\text{d}}$	$\overline{\text{p}}$	c	$\overline{\text{r}}$	3,75M	-	941k	807k	1,17M	2,71M	12,9M
Elias γ	$\overline{\text{d}}$	$\overline{\text{p}}$	c	r	3,75M	-	193k	613k	467k	823k	12,9M
Elias γ	$\overline{\text{d}}$	p	$\overline{\text{c}}$	$\overline{\text{r}}$	3,75M	-	530k	613k	1,08M	2,69M	12,9M
Elias γ	$\overline{\text{d}}$	p	$\overline{\text{c}}$	r	3,75M	-	193k	613k	414k	809k	12,9M
Elias γ	$\overline{\text{d}}$	p	c	$\overline{\text{r}}$	3,75M	-	530k	613k	1,08M	2,69M	12,9M
Elias γ	$\overline{\text{d}}$	p	c	r	3,75M	-	193k	613k	414k	809k	12,9M
Elias γ	d	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	3,75M	-	8,11M	-	213k	535k	12,9M
Elias γ	d	$\overline{\text{p}}$	$\overline{\text{c}}$	r	3,75M	-	416k	-	37,9k	535k	12,9M
Elias γ	d	$\overline{\text{p}}$	c	$\overline{\text{r}}$	3,75M	-	8,11M	1,02M	213k	535k	12,9M
Elias γ	d	$\overline{\text{p}}$	c	r	3,75M	-	416k	613k	37,9k	535k	12,9M
Fibonacci	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	4,69M	-	759k	656k	2,15M	5,36M	10,4M
Fibonacci	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	r	4,69M	-	160k	493k	852k	1,58M	10,4M
Fibonacci	$\overline{\text{d}}$	$\overline{\text{p}}$	c	$\overline{\text{r}}$	4,69M	-	759k	656k	2,15M	5,36M	10,4M
Fibonacci	$\overline{\text{d}}$	$\overline{\text{p}}$	c	r	4,69M	-	160k	493k	852k	1,58M	10,4M
Fibonacci	$\overline{\text{d}}$	p	$\overline{\text{c}}$	$\overline{\text{r}}$	4,55M	-	463k	493k	2,09M	5,34M	10,4M
Fibonacci	$\overline{\text{d}}$	p	$\overline{\text{c}}$	r	4,53M	-	160k	493k	817k	1,57M	10,4M
Fibonacci	$\overline{\text{d}}$	p	c	$\overline{\text{r}}$	4,55M	-	463k	493k	2,09M	5,34M	10,4M
Fibonacci	$\overline{\text{d}}$	p	c	r	4,53M	2,76M	160k	493k	817k	1,57M	10,4M
Fibonacci	d	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	4,36M	-	7,44M	865k	232k	829k	10,4M
Fibonacci	d	$\overline{\text{p}}$	$\overline{\text{c}}$	r	4,36M	-	345k	493k	40,6k	829k	10,4M
Fibonacci	d	$\overline{\text{p}}$	c	$\overline{\text{r}}$	4,36M	-	7,44M	865k	232k	829k	10,4M
Fibonacci	d	$\overline{\text{p}}$	c	r	4,36M	2,76M	345k	493k	40,6k	829k	10,4M
Unário	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	16,0M	-	-	-	1,16M	2,69M	-
Unário	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	r	16,0M	-	-	-	460k	1,03M	-
Unário	$\overline{\text{d}}$	$\overline{\text{p}}$	c	$\overline{\text{r}}$	16,0M	-	-	-	1,16M	2,69M	-
Unário	$\overline{\text{d}}$	$\overline{\text{p}}$	c	r	16,0M	-	-	-	460k	1,03M	-
Unário	$\overline{\text{d}}$	p	$\overline{\text{c}}$	$\overline{\text{r}}$	16,0M	-	-	-	1,08M	2,68M	-
Unário	$\overline{\text{d}}$	p	$\overline{\text{c}}$	r	16,0M	-	-	-	412k	1,03M	-
Unário	$\overline{\text{d}}$	p	c	$\overline{\text{r}}$	16,0M	-	-	-	1,08M	2,68M	-
Unário	$\overline{\text{d}}$	p	c	r	16,0M	-	-	-	412k	1,03M	-
Unário	d	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	16,0M	-	-	-	1,05M	2,66M	-
Unário	d	$\overline{\text{p}}$	$\overline{\text{c}}$	r	16,0M	-	-	-	405k	1,03M	-
Unário	d	$\overline{\text{p}}$	c	$\overline{\text{r}}$	16,0M	-	-	-	1,05M	2,66M	-
Unário	d	$\overline{\text{p}}$	c	r	16,0M	-	-	-	405k	1,03M	-

Tabela 4.3: Consumo de memória máximo ($^1/_2$)

Para a base de dados mushroom e limite inferior de suporte 30%, os códigos Elias γ, Elias

δ, de Fibonacci e Unário reduziram o consumo de memória máximo em, respectivamente, 85%, 84%, 79% e 79%, em relação ao código Binário. As habilitações da reordenação dinâmica, dos *diffset*s e da projeção o reduziram em, respectivamente, 63%, 58% e 27%.

A redução do consumo de memória máximo para a base de dados pumsb e limite inferior de suporte 90% pelos códigos Elias γ, Elias δ e de Fibonacci foi de, respectivamente, 79%, 79% e 78%, em relação ao código Binário. As habilitações dos *diffset*s, da reordenação dinâmica e da projeção o reduziram em, respectivamente, 65%, 36% e 3%.

Um resultado parecido foi alcançado para a base de dados pumsb_star e limite inferior de suporte 50%. A redução do consumo de memória máximo pelos códigos Elias γ, Elias δ e de Fibonacci foi de, respectivamente, 87%, 87% e 83%, em relação ao código Binário. As habilitações dos *diffset*s, da projeção e da reordenação dinâmica o reduziram em, respectivamente, 63%, 21% e 15%. Algumas configurações de código Binário não respeitaram o limite de trezentos segundos de tempo de execução.

Já para a base de dados retail e limite inferior de suporte 0,5%, os códigos de Fibonacci, Elias δ e Elias γ reduziram o consumo de memória máximo em, respectivamente, 48%, 45% e 40%. A desabilitação dos *diffset*s e a habilitação da reordenação dinâmica o reduziram em, respectivamente, 29% e 20%.

Finalmente, para as bases de dados sintéticas T10I4D100K e T40I10D100K, a redução do consumo de memória máximo foi obtido apenas pelo código. A redução em relação ao código Binário, para a base de dados T10I4D100K e limite inferior de suporte 20%, para os códigos de Fibonacci, Elias δ e Elias γ, foi de, respectivamente, 49%, 43% e 39%, e, para a base de dados T40I10D100K e limite inferior de suporte 2%, para os códigos de Fibonacci, Elias γ e Elias δ, foi de, respectivamente, 63%, 59% e 58%.

Configuração					mushroom	pumsb	pumsb_star	retail	T10I4D100K	T40I10D100K
					30%	90%	50%	0,5%	20%	2%
Binário	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	6,92M	-	30,8M	1,84M	2,04M	8,02M
Binário	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	r	3,69M	-	23,1M	1,84M	2,04M	8,02M
Binário	$\overline{\text{d}}$	$\overline{\text{p}}$	c	$\overline{\text{r}}$	6,92M	-	30,8M	1,84M	2,04M	8,02M
Binário	$\overline{\text{d}}$	$\overline{\text{p}}$	c	r	3,69M	-	23,1M	1,84M	2,04M	8,02M
Binário	$\overline{\text{d}}$	p	$\overline{\text{c}}$	$\overline{\text{r}}$	6,92M	-	30,8M	1,84M	2,04M	8,02M
Binário	$\overline{\text{d}}$	p	$\overline{\text{c}}$	r	3,69M	-	23,1M	1,84M	2,04M	8,02M
Binário	$\overline{\text{d}}$	p	c	$\overline{\text{r}}$	6,92M	-	30,8M	1,84M	2,04M	8,02M
Binário	$\overline{\text{d}}$	p	c	r	3,69M	-	23,1M	1,84M	2,04M	8,02M
Binário	d	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	4,36M	6,92M	4,72M	10,7M	2,04M	-
Binário	d	$\overline{\text{p}}$	$\overline{\text{c}}$	r	481k	6,92M	4,72M	1,84M	2,04M	-
Binário	d	$\overline{\text{p}}$	c	$\overline{\text{r}}$	4,36M	6,92M	4,72M	10,7M	2,04M	8,02M
Binário	d	$\overline{\text{p}}$	c	r	481k	6,92M	4,72M	1,84M	2,04M	8,02M
Elias δ	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	1,05M	7,43M	3,65M	1,27M	1,15M	3,33M
Elias δ	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	r	686k	3,21M	3,03M	1,27M	1,15M	-
Elias δ	$\overline{\text{d}}$	$\overline{\text{p}}$	c	$\overline{\text{r}}$	1,05M	7,43M	3,65M	1,27M	1,15M	3,33M
Elias δ	$\overline{\text{d}}$	$\overline{\text{p}}$	c	r	686k	3,21M	3,03M	1,27M	1,15M	3,33M
Elias δ	$\overline{\text{d}}$	p	$\overline{\text{c}}$	$\overline{\text{r}}$	921k	7,14M	2,41M	1,27M	1,15M	-
Elias δ	$\overline{\text{d}}$	p	$\overline{\text{c}}$	r	321k	2,85M	1,54M	1,27M	1,15M	3,33M

Configuração					mushroom 30%	pumsb 90%	pumsb_star 50%	retail 0,5%	T10I4D100K 20%	T40I10D100K 2%
Elias δ	$\overline{\text{d}}$	p	c	$\overline{\text{r}}$	921k	7,14M	2,41M	1,27M	1,15M	3,33M
Elias δ	$\overline{\text{d}}$	p	c	r	321k	2,85M	1,54M	1,27M	1,15M	3,33M
Elias δ	d	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	796k	1,57M	1,20M	1,87M	1,15M	3,33M
Elias δ	d	$\overline{\text{p}}$	$\overline{\text{c}}$	r	144k	1,26M	1,10M	1,27M	1,15M	3,33M
Elias δ	d	$\overline{\text{p}}$	c	$\overline{\text{r}}$	796k	1,57M	1,20M	1,87M	1,15M	3,33M
Elias δ	d	$\overline{\text{p}}$	c	r	144k	1,26M	1,10M	1,27M	1,15M	3,33M
Elias γ	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	962k	7,18M	3,24M	1,45M	1,23M	3,28M
Elias γ	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	r	611k	3,07M	2,68M	1,45M	1,23M	3,28M
Elias γ	$\overline{\text{d}}$	$\overline{\text{p}}$	c	$\overline{\text{r}}$	962k	7,18M	3,24M	1,45M	1,23M	3,28M
Elias γ	$\overline{\text{d}}$	$\overline{\text{p}}$	c	r	611k	3,07M	2,68M	1,45M	1,23M	3,28M
Elias γ	$\overline{\text{d}}$	p	$\overline{\text{c}}$	$\overline{\text{r}}$	841k	6,99M	2,26M	1,45M	1,23M	3,28M
Elias γ	$\overline{\text{d}}$	p	$\overline{\text{c}}$	r	313k	2,83M	1,51M	1,45M	1,23M	3,28M
Elias γ	$\overline{\text{d}}$	p	c	$\overline{\text{r}}$	841k	6,99M	2,26M	1,45M	1,23M	3,28M
Elias γ	$\overline{\text{d}}$	p	c	r	313k	2,83M	1,51M	1,45M	1,23M	3,28M
Elias γ	d	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	706k	1,59M	1,16M	1,62M	1,23M	3,28M
Elias γ	d	$\overline{\text{p}}$	$\overline{\text{c}}$	r	134k	1,22M	1,07M	1,45M	1,23M	3,28M
Elias γ	d	$\overline{\text{p}}$	c	$\overline{\text{r}}$	706k	1,59M	1,16M	1,62M	1,23M	3,28M
Elias γ	d	$\overline{\text{p}}$	c	r	134k	1,22M	1,07M	1,45M	1,23M	3,28M
Fibonacci	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	1,31M	13,5M	4,73M	1,17M	1,03M	2,91M
Fibonacci	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	r	793k	5,72M	3,71M	1,17M	1,03M	2,91M
Fibonacci	$\overline{\text{d}}$	$\overline{\text{p}}$	c	$\overline{\text{r}}$	1,31M	13,5M	4,73M	1,17M	1,03M	2,91M
Fibonacci	$\overline{\text{d}}$	$\overline{\text{p}}$	c	r	793k	5,72M	3,71M	1,17M	1,03M	2,91M
Fibonacci	$\overline{\text{d}}$	p	$\overline{\text{c}}$	$\overline{\text{r}}$	1,25M	13,4M	4,05M	1,17M	1,03M	2,91M
Fibonacci	$\overline{\text{d}}$	p	$\overline{\text{c}}$	r	596k	5,59M	2,93M	1,17M	1,03M	2,91M
Fibonacci	$\overline{\text{d}}$	p	c	$\overline{\text{r}}$	1,25M	13,4M	4,05M	1,17M	1,03M	2,91M
Fibonacci	$\overline{\text{d}}$	p	c	r	596k	5,59M	2,93M	1,17M	1,03M	2,91M
Fibonacci	d	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	925k	1,57M	1,13M	1,92M	1,03M	2,91M
Fibonacci	d	$\overline{\text{p}}$	$\overline{\text{c}}$	r	141k	1,41M	1,13M	1,17M	1,03M	2,91M
Fibonacci	d	$\overline{\text{p}}$	c	$\overline{\text{r}}$	925k	1,57M	1,13M	1,92M	1,03M	2,91M
Fibonacci	d	$\overline{\text{p}}$	c	r	141k	1,41M	1,13M	1,17M	1,03M	2,91M
Unário	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	1,10M	11,0M	10,9M	-	10,3M	11,2M
Unário	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	r	791k	11,0M	10,9M	-	10,3M	11,2M
Unário	$\overline{\text{d}}$	$\overline{\text{p}}$	c	$\overline{\text{r}}$	1,10M	11,0M	10,9M	-	10,3M	11,2M
Unário	$\overline{\text{d}}$	$\overline{\text{p}}$	c	r	791k	11,0M	10,9M	-	10,3M	11,2M
Unário	$\overline{\text{d}}$	p	$\overline{\text{c}}$	$\overline{\text{r}}$	859k	11,0M	10,9M	-	10,3M	11,2M
Unário	$\overline{\text{d}}$	p	$\overline{\text{c}}$	r	309k	11,0M	10,9M	-	10,3M	11,2M
Unário	$\overline{\text{d}}$	p	c	$\overline{\text{r}}$	859k	11,0M	10,9M	-	10,3M	11,2M
Unário	$\overline{\text{d}}$	p	c	r	309k	11,0M	10,9M	-	10,3M	11,2M
Unário	d	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	993k	11,0M	10,9M	-	10,3M	11,2M
Unário	d	$\overline{\text{p}}$	$\overline{\text{c}}$	r	500k	11,0M	10,9M	-	10,3M	11,2M
Unário	d	$\overline{\text{p}}$	c	$\overline{\text{r}}$	993k	11,0M	10,9M	-	10,3M	11,2M
Unário	d	$\overline{\text{p}}$	c	r	500k	11,0M	10,9M	-	10,3M	11,2M

Tabela 4.4: Consumo de memória máximo ($^2/_2$)

Os resultados em função do tempo de execução podem ser encontrados nas Tabelas B.1 e B.2 do Apêndice B.

Em resumo, para as instâncias avaliadas, os melhores códigos quanto à redução do consumo máximo de memória em relação ao código Binário, foram o Elias δ, redução de 76%, seguido pelo Elias γ, redução de 76%, depois o de Fibonacci, redução de 73%, e, por último, o Unário, redução de 45%. A redução do consumo máximo de memória para a habilitação da reordenação dinâmica é de 40%, para a habilitação dos *diffset*s é de 33% e para a habilitação da projeção é de 8%. Quanto ao tempo de execução, os códigos Elias γ, Elias δ, de Fibonacci e Unário, o reduzem em, respectivamente, 57%, 54%, 48% e 39%, em relação ao código Binário. A habilitação dos *diffset*s o reduz em 41%, a habilitação do curto-circuito o reduz em 25%, a habilitação da reordenação dinâmica o reduz em 15% e a habilitação da projeção o aumenta em 11%. Na maioria dos casos as reduções conjugadas são ainda maiores que as individuais.

A diferença de consumo máximo de memória entre a melhor configuração do zEClaT e a configuração equivalente ao EClaT ou ao dEClaT chega a mais de uma ordem de magnitude em alguns. Esse é o caso para a base de dados chess, com limite inferior de suporte igual a 80%, código Elias δ e curto-circuito, *diffset*s e reordenação dinâmica habilitados, mais de 20 vezes melhor; para a base de dados connect, com limite inferior de suporte igual a 30%, código Elias γ e curto-circuito e *diffset*s habilitados, mais de 10 vezes melhor; e a base de dados mushroom, com limite inferior de suporte igual a 30%, código Elias γ e curto-circuito, *diffset*s e reordenação dinâmica habilitados, mais de 30 vezes melhor.

Nos gráficos da Figura 4.1 podemos observar que o padrão do consumo de memória do algoritmo independe do código e que, para a base de dados chess e limite inferior de suporte igual a 80%, os códigos Elias δ, Elias γ, Fibonacci e Unário foram responsáveis por reduções entre 82% e 90% do consumo de memória e entre 68% e 80% do tempo de execução.

Nos gráficos da Figura 4.2 podemos observar que o padrão do consumo de memória do algoritmo independe da habilitação de *diffset*s e que, para a base de dados chess e limite inferior de suporte igual a 80%, a habilitação dos *diffset*s foi responsável por reduções de 93% do consumo de memória e 92% do tempo de execução.

Nos gráficos da Figura 4.3 podemos observar que o padrão do consumo de memória do algoritmo independe da habilitação da projeção e que, para a base de dados chess e limite inferior de suporte igual a 80%, a habilitação da projeção, além de não reduzir o consumo de memória, aumentou o tempo de execução em 58%.

Nos gráficos da Figura 4.4 podemos observar que o padrão do consumo de memória do algoritmo independe da habilitação do curto-circuito e que, para a base de dados chess e limite inferior de suporte igual a 80%, a habilitação do curto-circuito não reduz o consumo de memória mas reduz o tempo de execução em 17%.

Nos gráficos da Figura 4.5 podemos observar que o padrão do consumo de memória do algoritmo difere dependendo da habilitação da reordenação dinâmica e que, para a base de dados chess e limite inferior de suporte igual a 80%, a habilitação da reordenação dinâmica foi responsável por reduções de 61% do consumo de memória e de 40% do tempo de execução.

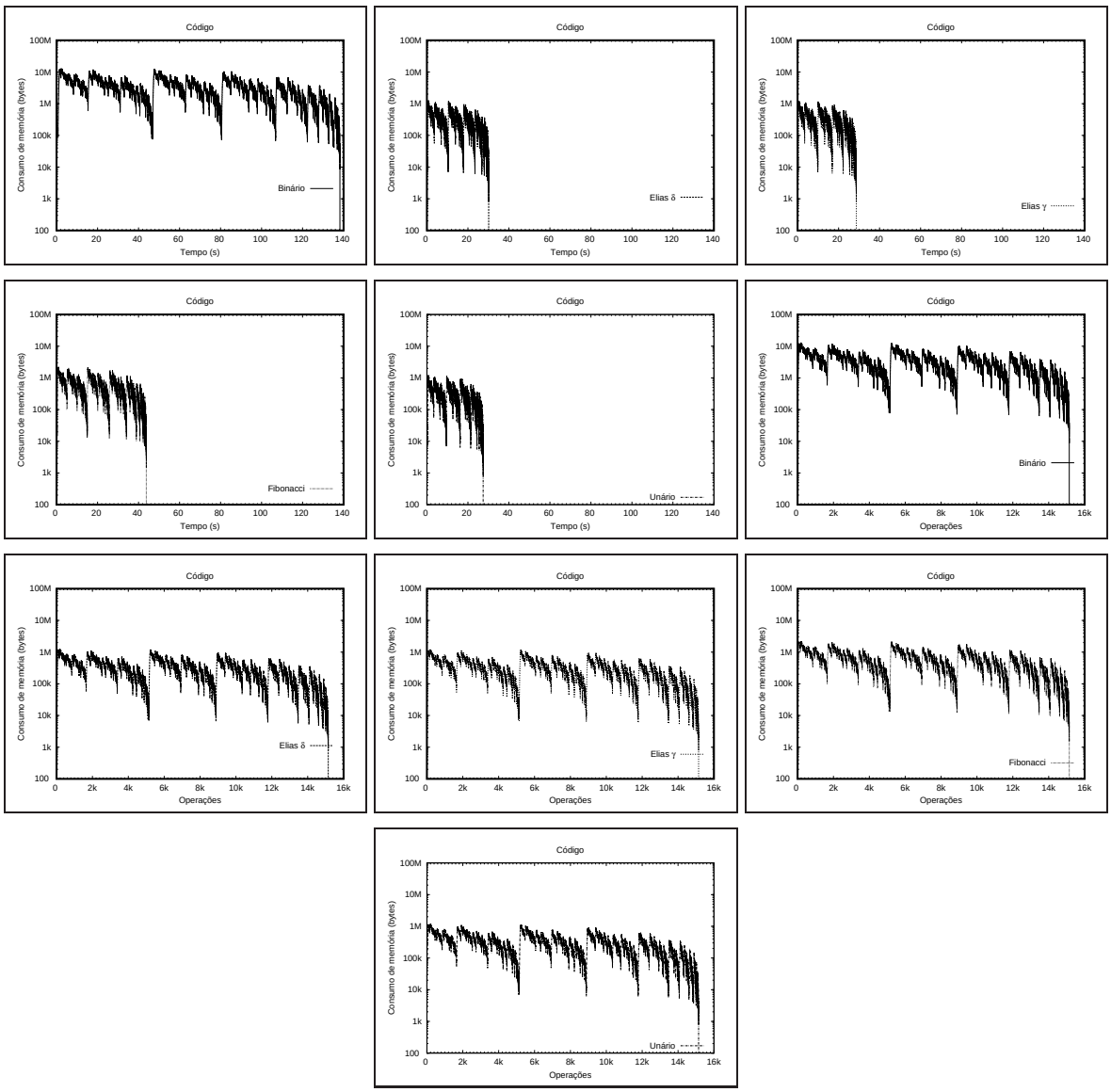


Figura 4.1: Código

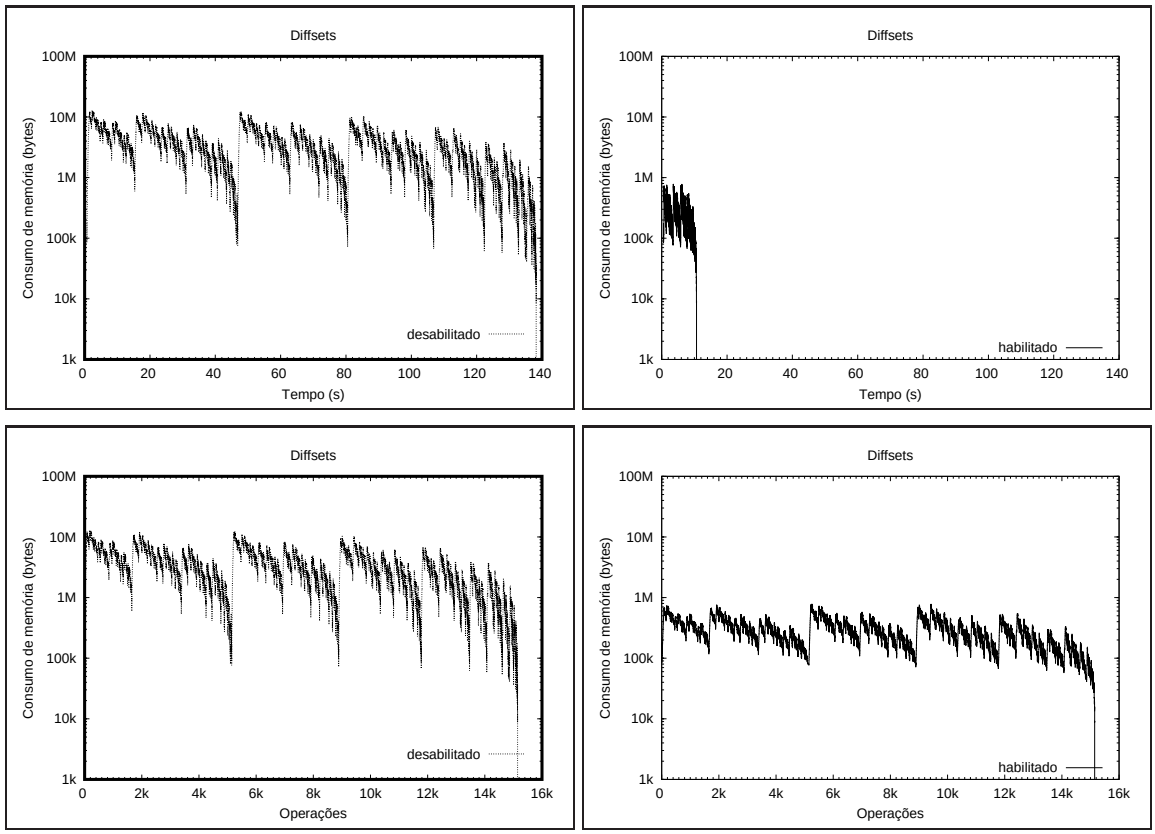


Figura 4.2: *Diffsets*

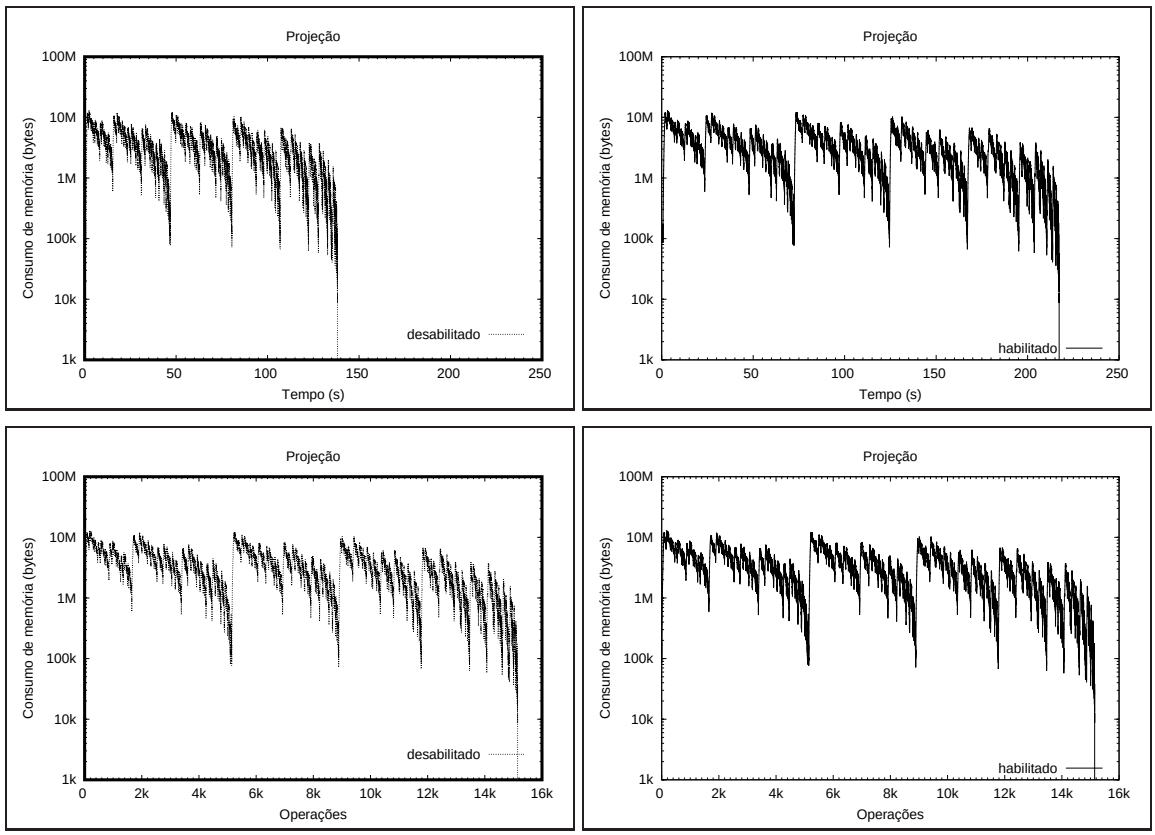


Figura 4.3: Projeção

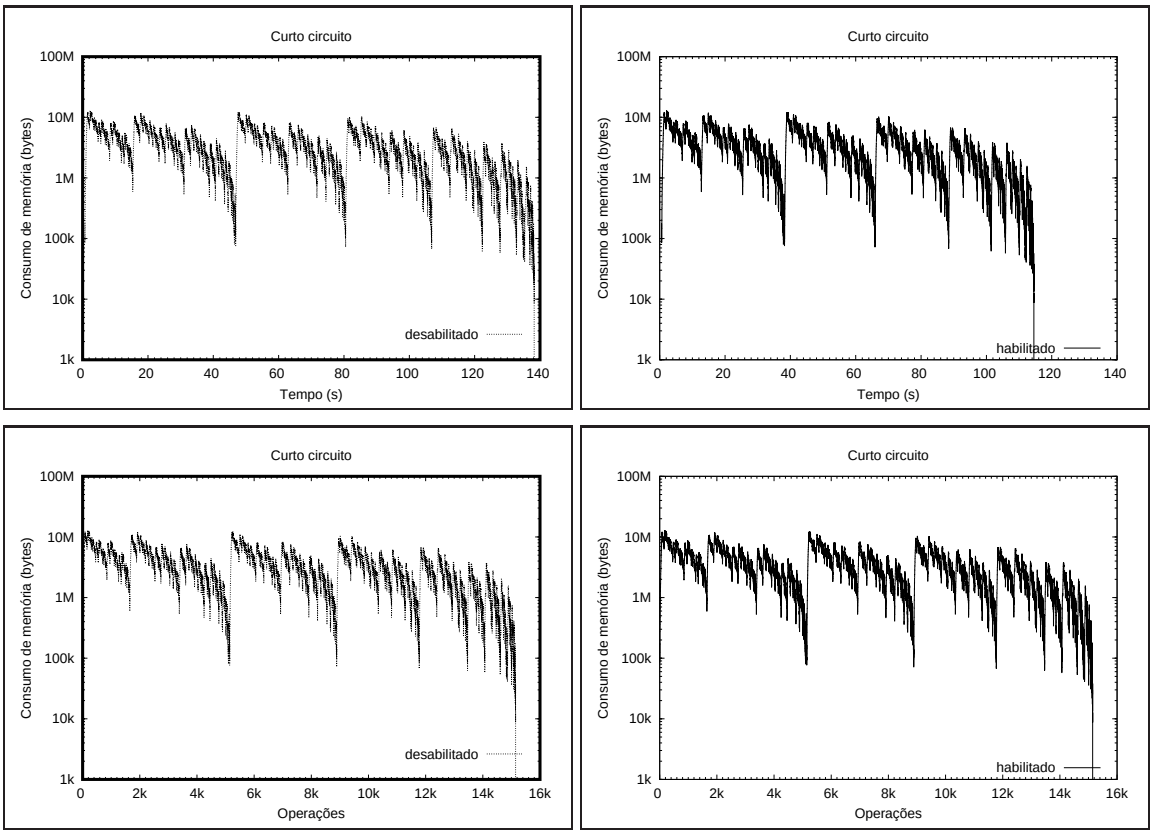


Figura 4.4: Curto-circuito

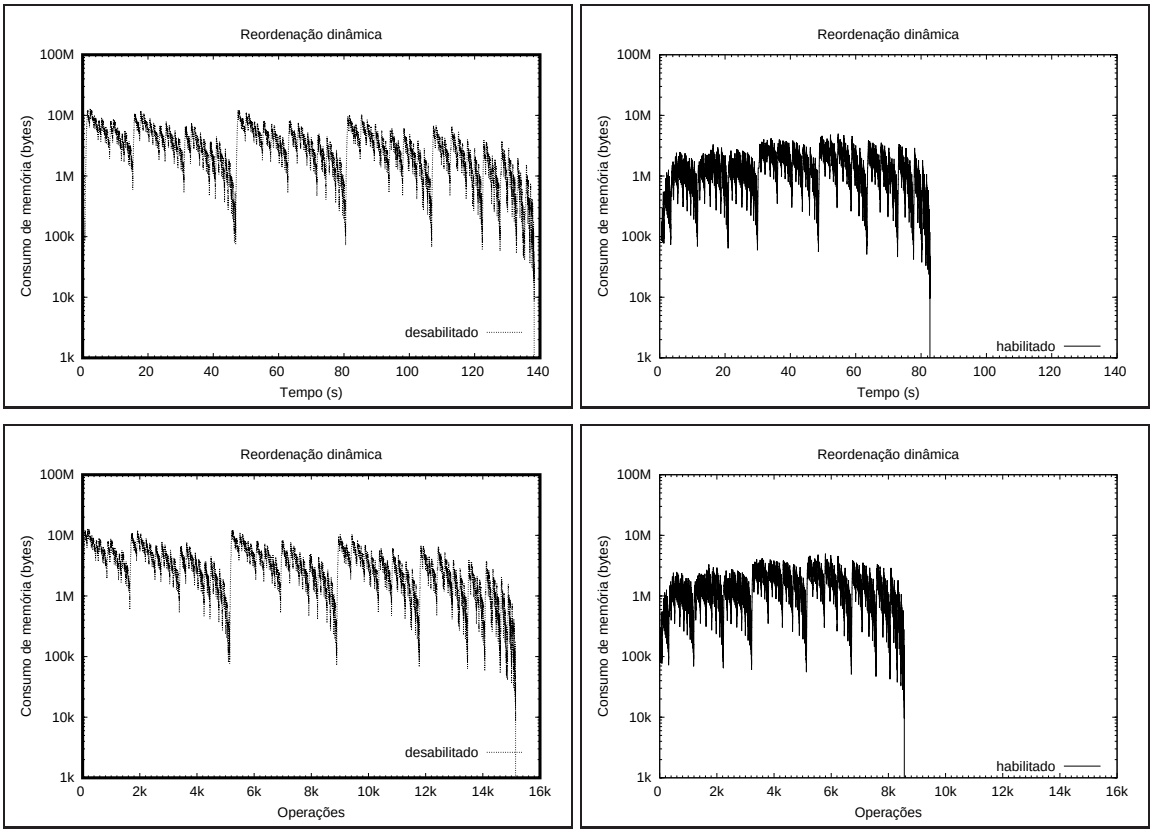


Figura 4.5: Reordenação dinâmica

Capítulo 5

Conclusões e Trabalhos Futuros

Neste trabalho realizamos a revisão bibliográfica de estratégias de redução de consumo de memória aplicáveis aos algoritmos de mineração de conjuntos freqüentes, implementamos e avaliamos um novo algoritmo que aplica algumas das estratégias, denominado zEClaT. O algoritmo zEClaT efetivamente reduz o consumo de memória em relação ao algoritmo EClaT e ao algoritmo dEClaT. A redução é superior a uma ordem de grandeza para algumas bases de dados e limites inferiores de suporte. Um código cuja distribuição de probabilidade implícita é mais próxima ao decaimento exponencial é adequado a limites inferiores de suporte altos e um código cuja distribuição de probabilidade implícita é mais próxima à distribuição uniforme é adequado a limites inferiores de suporte baixos. As características das bases de dados e o parâmetro de limite inferior de suporte influenciam diretamente a efetividade das estratégias de redução de consumo de memória e tempo de execução.

O código Unário é efetivo apenas para a redução de consumo de memória para bases de dados densas e limites inferiores de suporte grandes. À medida que a densidade da base de dados ou o limite inferior de suporte diminui, os códigos Elias δ, Elias γ e Fibonacci, não necessariamente nessa ordem, tornam-se mais interessantes, até que o código Binário seja o mais econômico, para bases de dados esparsas ou limites inferiores de suporte pequenos. Para um dado código, o consumo de memória do algoritmo com *d-gap*s nunca é maior que o do algoritmo sem *d-gap*s. O curto-circuito é efetivo somente para a redução do tempo de execução, quando a quantidade de *itemset*s infreqüentes verificados é grande em relação à quantidade de *itemset*s freqüentes. O consumo de memória do algoritmo com curto-circuito nunca é maior que o do algoritmo sem curto-circuito. Os *diffset*s reduzem tanto o consumo de memória quanto o tempo de execução, especialmente quando as bases de dados são densas e os limites inferiores de suporte são grandes. Há possibilidade que o consumo de memória do algoritmo com *diffset*s seja maior que o do algoritmo sem *diffset*s. A projeção reduz marginalmente o consumo de memória e aumenta marginalmente o tempo de execução. A redução é maior quando as bases de dados são esparsas e quando o limite inferior de suporte é pequeno. O consumo de memória do algoritmo com projeção nunca é maior que o do algoritmo sem projeção. A reordenação dinâmica altera o comportamento do consumo de memória tanto pela redução do consumo pelos resul-

tados intermediários quanto pelo deslocamento do consumo dos picos provendo uma utilização mais racional da memória. Quando a base de dados é longa o custo da reordenação dinâmica é compensado pelo pela redução de custo das operações. Há possibilidade que o consumo de memória do algoritmo com reordenação dinâmica seja maior que o do algoritmo sem reordenação dinâmica.

Como trabalhos futuros podemos implementar e avaliar outros refinamentos, em particular, calcular os 2-*itemset*s freqüentes a partir do arquivo de entrada e não a partir dos 1-*itemset*s freqüentes. Avaliar o algoritmo quando há memória secundária envolvida. Inventar e avaliar métricas para escolha automática de configuração. Caracterizar os *tidset*s mais propícios à economia de consumo de memória e aplicar compactação seletiva. A princípio, os *itemset*s mais rasos na treliça são os que mais consomem memória e os mais propícios à economia e sua compactação já reduziria significativamente o consumo afetando menos o desempenho.

Apêndice A

Exemplo do algoritmo EClaT

Agora mostramos um exemplo passo a passo do funcionamento do algoritmo EClaT. Os vértices representam os *itemset*s. O primeiro conjunto é o *itemset* e o segundo conjunto é o *tidset*. Vértices de borda preta representam *itemset*s na memória. Vértices de fundo cinza representam *itemset*s freqüentes. As arestas representam a relação de cobertura. Arestas pretas representam junções.

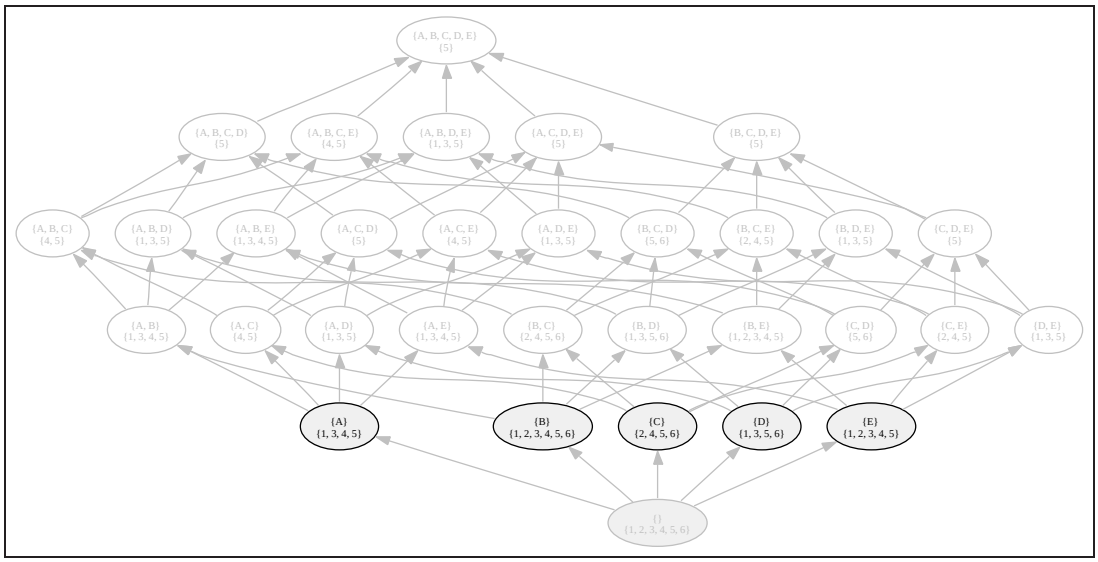


Figura A.1: Classe de equivalência $[\{\}]$, *itemset*s $\{A\}$, $\{B\}$, $\{C\}$, $\{D\}$ e $\{E\}$

Mineramos primeiro os 1-*itemset*s freqüentes (Figura A.1).

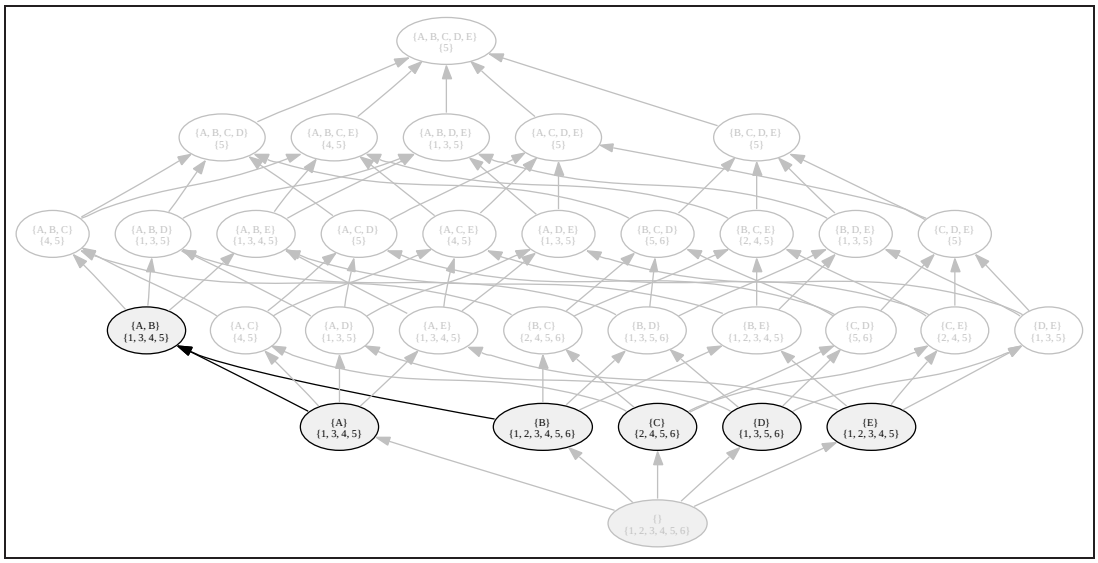


Figura A.2: Classe de equivalência $[\{A\}]$, *itemset* $\{A, B\}$

A primeira classe é $[\{\}]$. O primeiro *itemset* é $\{A\}$. A primeira junção da classe $[\{\}]$ e do *itemset* $\{A\}$, $\{A\} \curlyvee \{B\}$, é freqüente (Figura A.2).

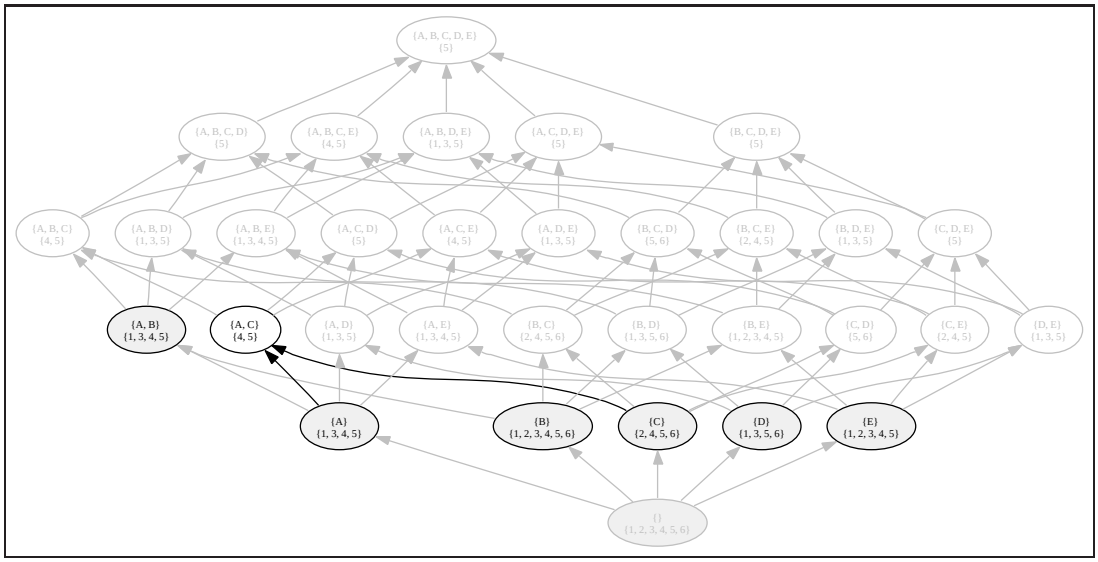


Figura A.3: Classe de equivalência $[\{A\}]$, *itemset* $\{A, C\}$

A próxima junção da classe $[\{\}]$ e do *itemset* $\{A\}$, $\{A\} \curlyvee \{C\}$, é infreqüente e é liberada (Figura A.3).

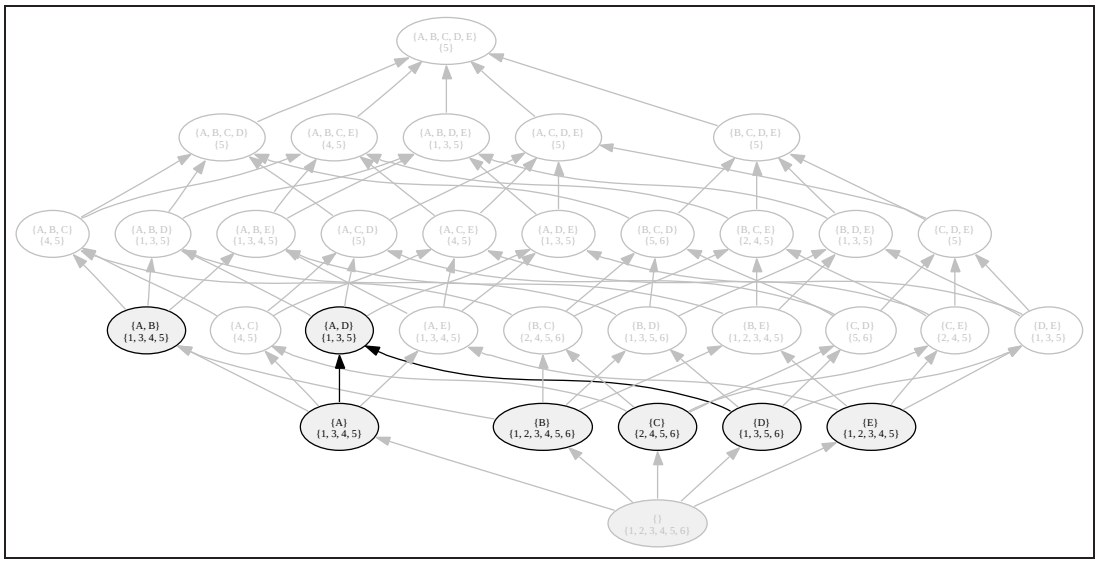


Figura A.4: Classe de equivalência $[\{A\}]$, *itemset* $\{A, D\}$

A próxima junção da classe $[\{\}]$ e do *itemset* $\{A\}$, $\{A\} \curlyvee \{D\}$, é freqüente (Figura A.4).

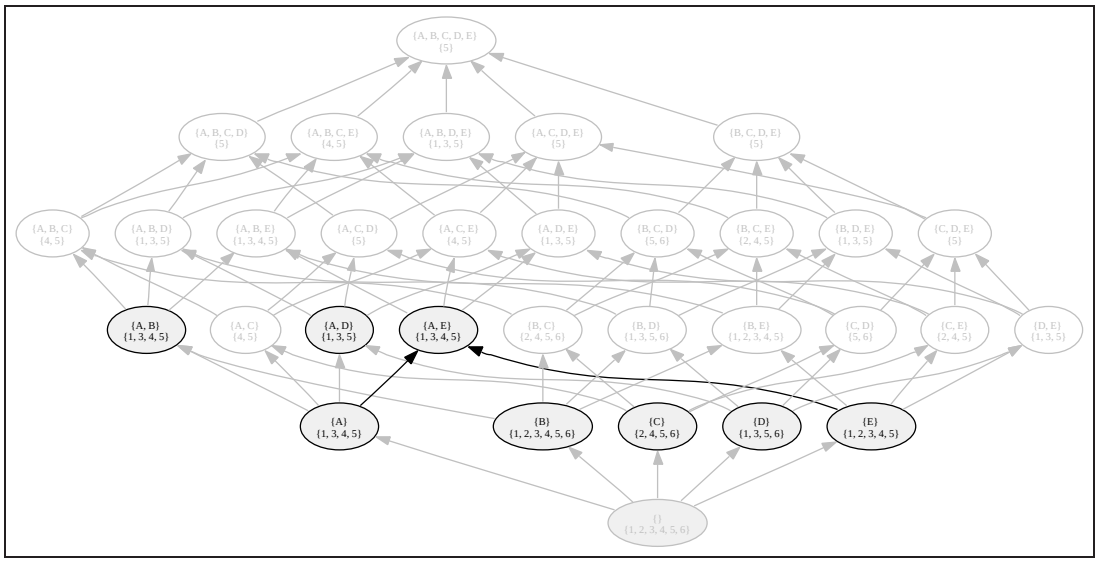


Figura A.5: Classe de equivalência $[\{A\}]$, *itemset* $\{A, E\}$

A próxima junção da classe $[\{\}]$ e do *itemset* $\{A\}$, $\{A\} \curlyvee \{E\}$, é freqüente (Figura A.5).

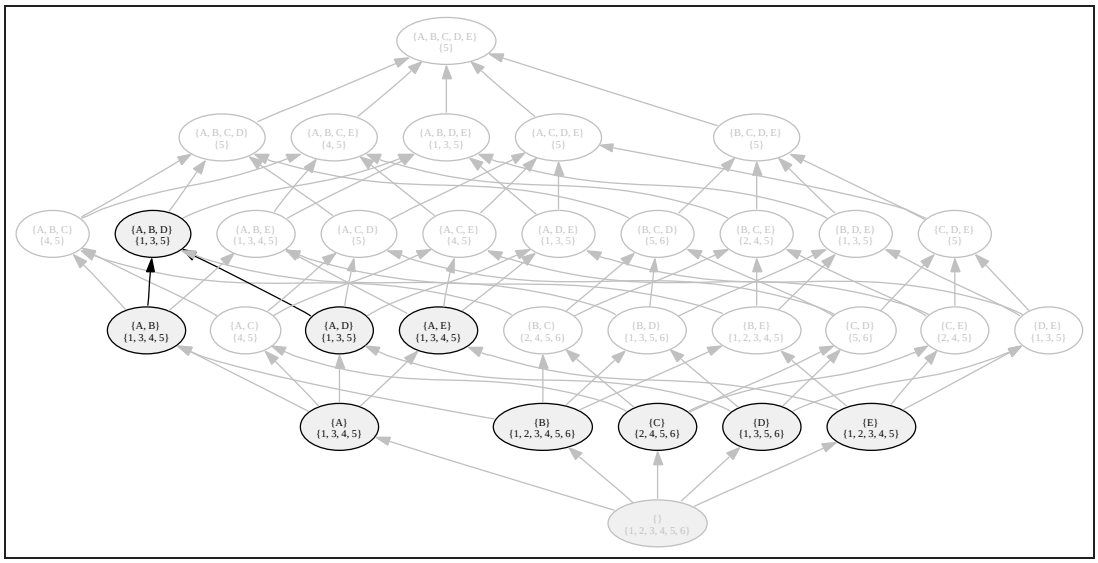


Figura A.6: Classe de equivalência $[\{A, B\}]$, *itemset* $\{A, B, D\}$

A próxima classe é $[\{A\}]$. O primeiro *itemset* é $\{A, B\}$. A primeira junção da classe $[\{A\}]$ e do *itemset* $\{A, B\}$, $\{A, B\} \curlyvee \{A, D\}$, é freqüente (Figura A.6).

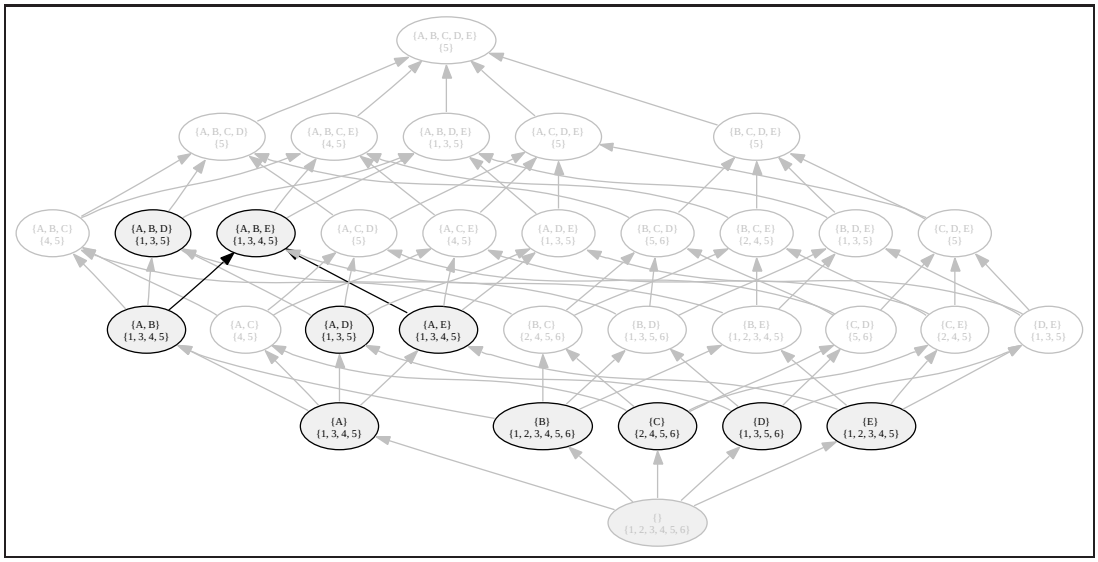


Figura A.7: Classe de equivalência $[\{A, B\}]$, *itemset* $\{A, B, E\}$

A próxima junção da classe $[\{A\}]$ e do *itemset* $\{A, B\}$, $\{A, B\} \curlyvee \{A, E\}$, é freqüente (Figura A.7).

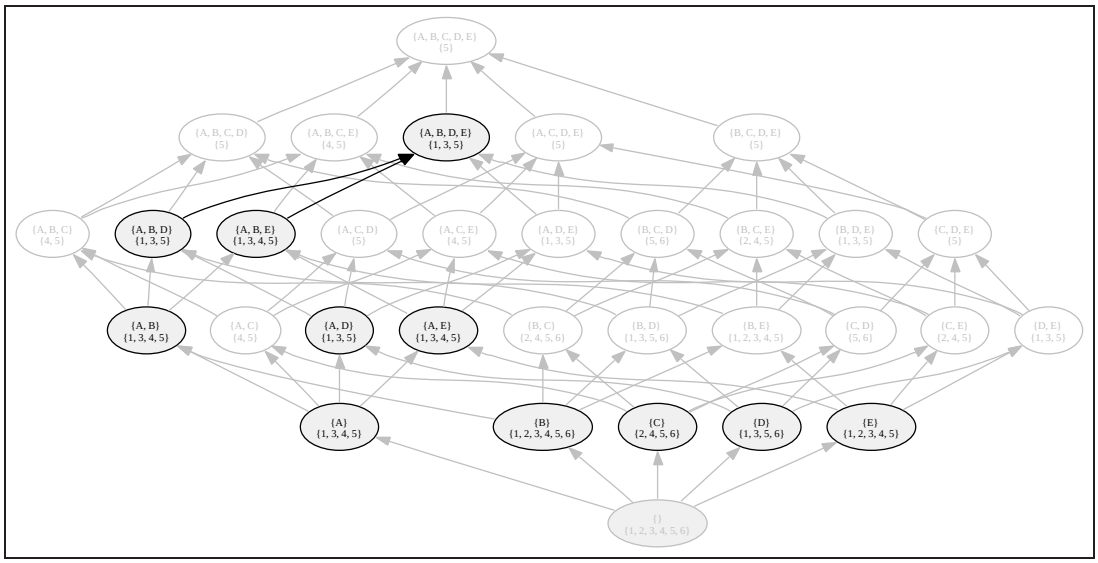


Figura A.8: Classe de equivalência $[\{A, B, D\}]$, *itemset* $\{A, B, D, E\}$

A próxima classe é $[\{A, B\}]$. O primeiro *itemset* é $\{A, B, D\}$. A primeira junção da classe $[\{A, B\}]$ e do *itemset* $\{A, B, D\}$, $\{A, B, D\} \curlyvee \{A, B, E\}$, é freqüente (Figura A.8). A próxima classe é $[\{A, B, D\}]$ e o único *itemset* é $\{A, B, D, E\}$, que é liberado. A classe $[\{A, B\}]$ é retomada e o *itemset* $\{A, B, D\}$ é liberado. O único *itemset* restante é $\{A, B, E\}$, que é liberado. A classe $[\{A\}]$ é retomada e o *itemset* $\{A, B\}$ é liberado.

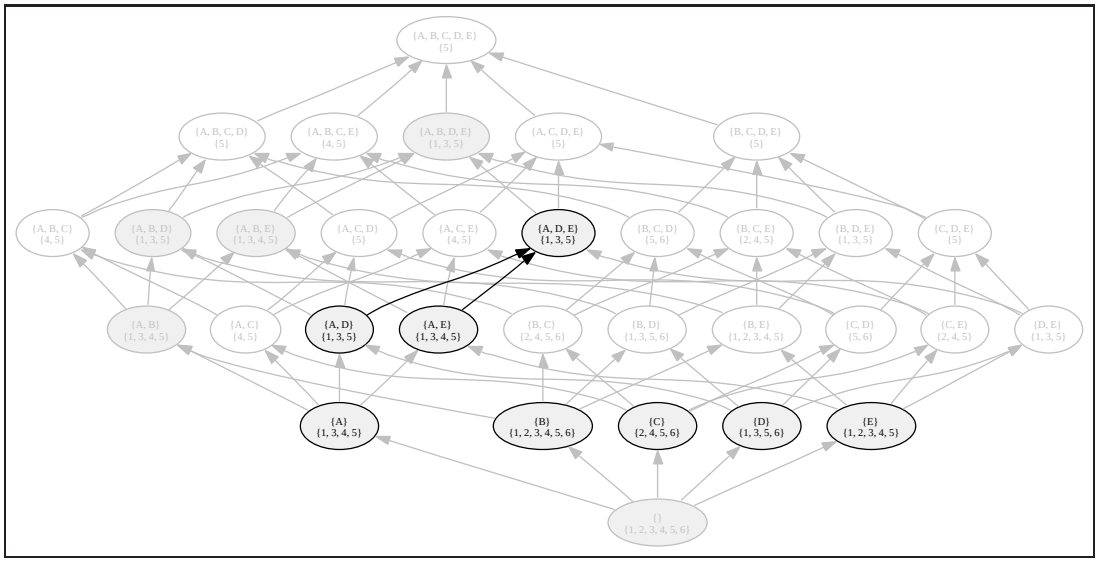


Figura A.9: Classe de equivalência $[\{A, D\}]$, *itemset* $\{A, D, E\}$

O próximo *itemset* é $\{A, D\}$. A primeira junção da classe $[\{A\}]$ e do *itemset* $\{A, D\}$, $\{A, D\} \curlyvee \{A, E\}$, é freqüente (Figura A.8). A próxima classe é $[\{A, D\}]$ e o único *itemset* é $\{A, D, E\}$, que é liberado. A classe $[\{A\}]$ é retomada e o *itemset* $\{A, D\}$ é liberado. O único *itemset* restante é $\{A, E\}$, que é liberado. A classe $[\{\}]$ é retomada e o *itemset* $\{A\}$ é liberado.

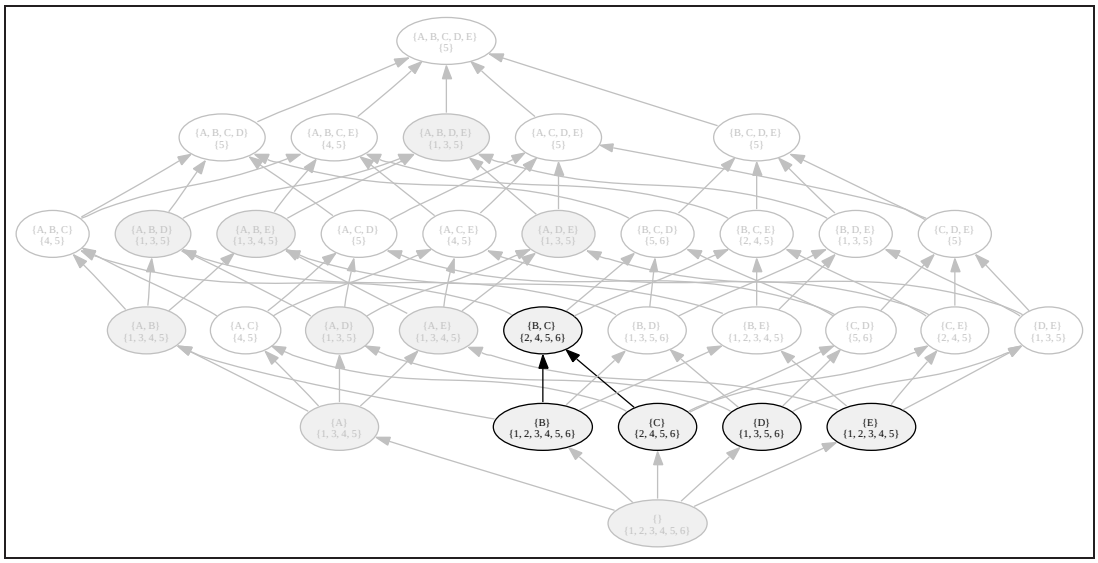


Figura A.10: Classe de equivalência $[\{B\}]$, *itemset* $\{B, C\}$

O próximo *itemset* é $\{B\}$. A primeira junção da classe $[\{\}]$ e do *itemset* $\{B\}$, $\{B\} \curlyvee \{C\}$, é freqüente (Figura A.10).

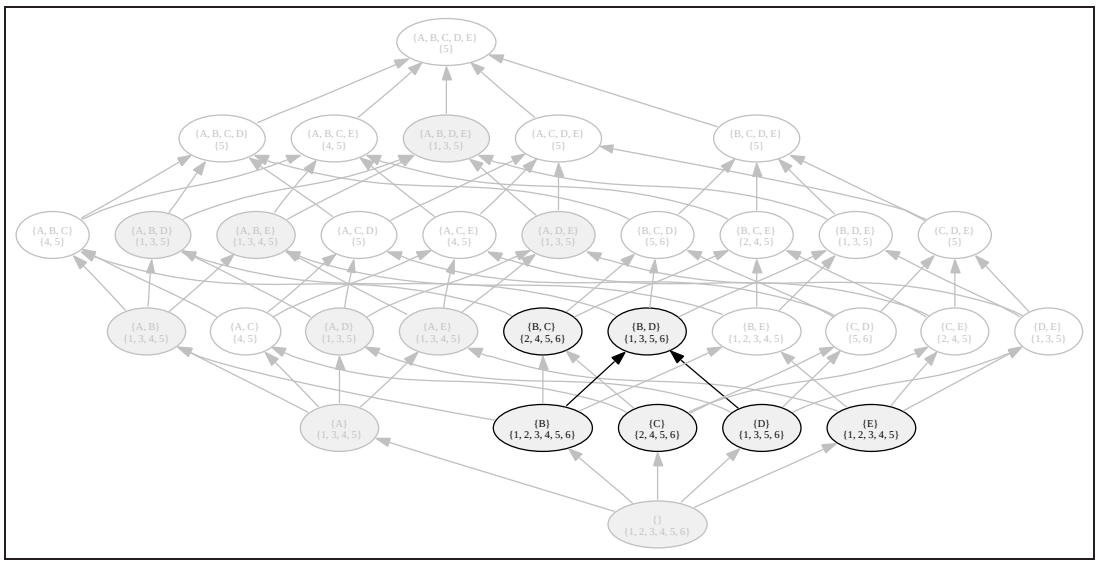


Figura A.11: Classe de equivalência $[\{B\}]$, *itemset* $\{B, D\}$

A próxima junção da classe $[\{\}]$ e do *itemset* $\{B\}$, $\{B\} \curlyvee \{D\}$, é freqüente (Figura A.11).

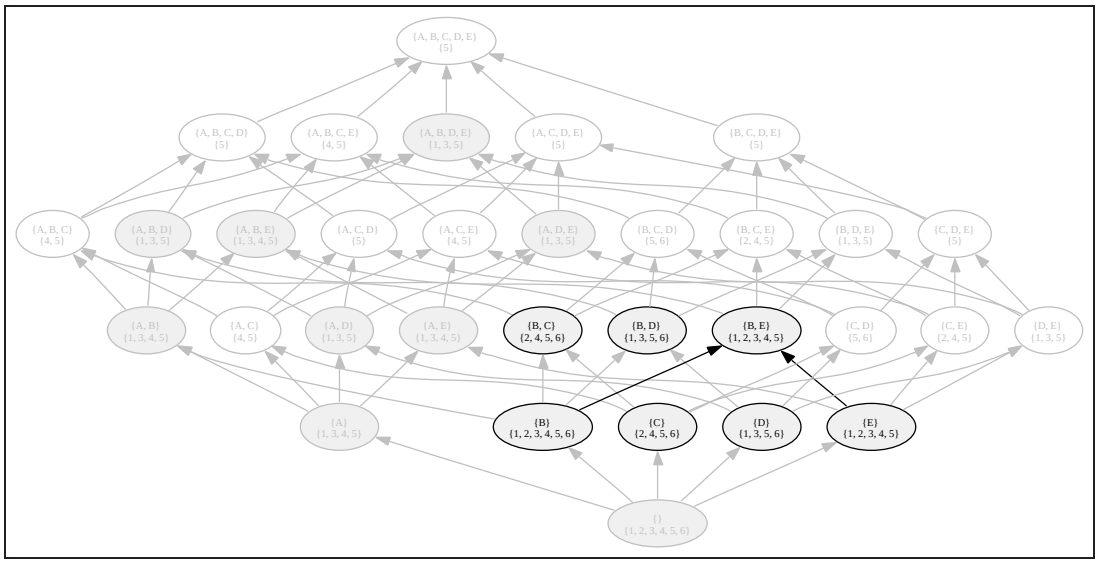


Figura A.12: Classe de equivalência $[\{B\}]$, *itemset* $\{B, E\}$

A próxima junção da classe $[\{\}]$ e do *itemset* $\{B\}$, $\{B\} \curlyvee \{E\}$, é freqüente (Figura A.12).

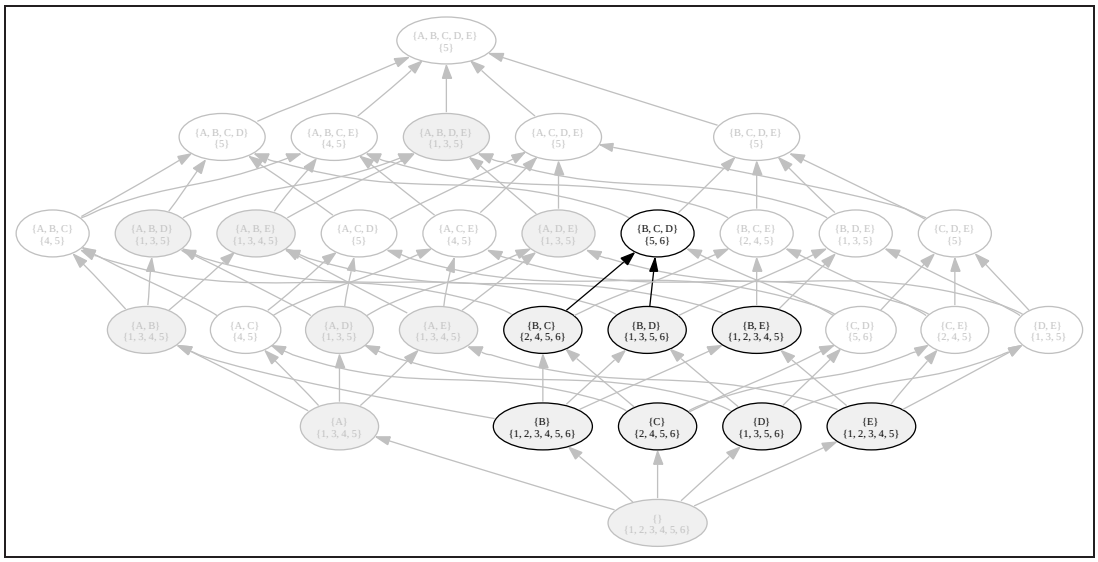


Figura A.13: Classe de equivalência $[\{B, C\}]$, *itemset* $\{B, C, D\}$

A próxima classe é $[\{B\}]$. O primeiro *itemset* é $\{B, C\}$. A primeira junção da classe $[\{B\}]$ e do *itemset* $\{B, C\}$, $\{B, C\} \curlyvee \{B, D\}$, é infreqüente e é liberada (Figura A.13).

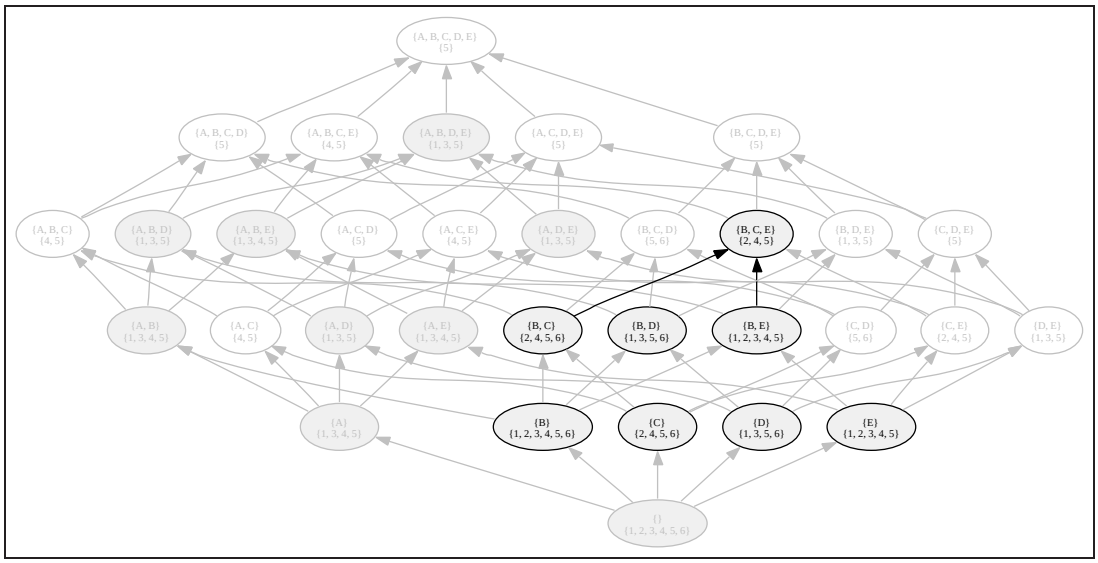


Figura A.14: Classe de equivalência $[\{B, C\}]$, *itemset* $\{B, C, E\}$

A próxima junção da classe $[\{B\}]$ e do *itemset* $\{B, C\}$, $\{B, C\} \curlyvee \{B, E\}$, é freqüente (Figura A.14). A próxima classe é $[\{B, C\}]$ e o único *itemset* é $\{B, C, E\}$, que é liberado. A classe $[\{B\}]$ é retomada e o *itemset* $\{B, C\}$ é liberado.

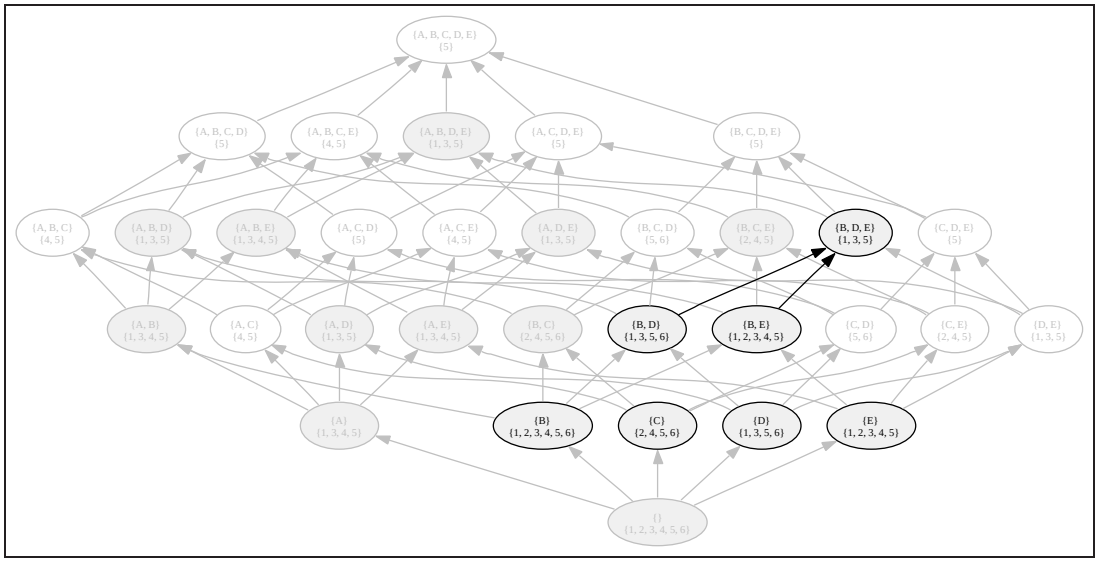


Figura A.15: Classe de equivalência $[\{B, D\}]$, *itemset* $\{B, D, E\}$

O próximo *itemset* é $\{B, D\}$. A primeira junção da classe $[\{B\}]$ e do *itemset* $\{B, D\}$, $\{B, D\} \curlyvee \{B, E\}$, é freqüente (Figura A.15). A próxima classe é $[\{B, D\}]$ e o único *itemset* é $\{B, D, E\}$, que é liberado. A classe $[\{B\}]$ é retomada e o *itemset* $\{B, D\}$ é liberado. O único *itemset* é $\{B, E\}$, que é liberado. A classe $[\{\}]$ é retomada e o *itemset* $\{B\}$ é liberado.

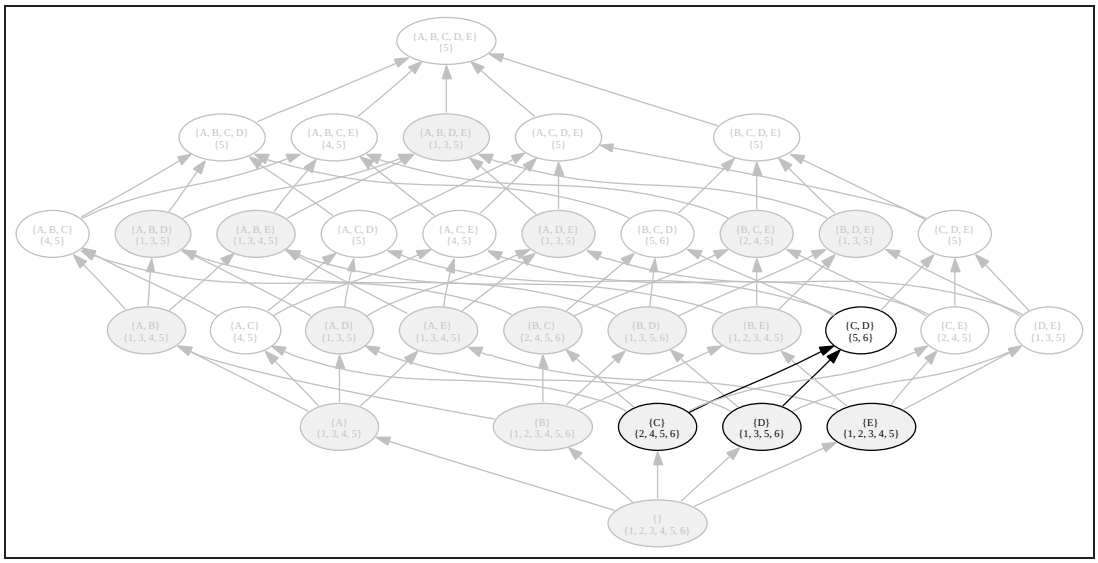


Figura A.16: Classe de equivalência $[\{C\}]$, *itemset* $\{C, D\}$

O próximo *itemset* é $\{C\}$. A primeira junção da classe $[\{\}]$ e do *itemset* $\{C\}$, $\{C\} \curlyvee \{D\}$, é infreqüente e é liberada (Figura A.16).

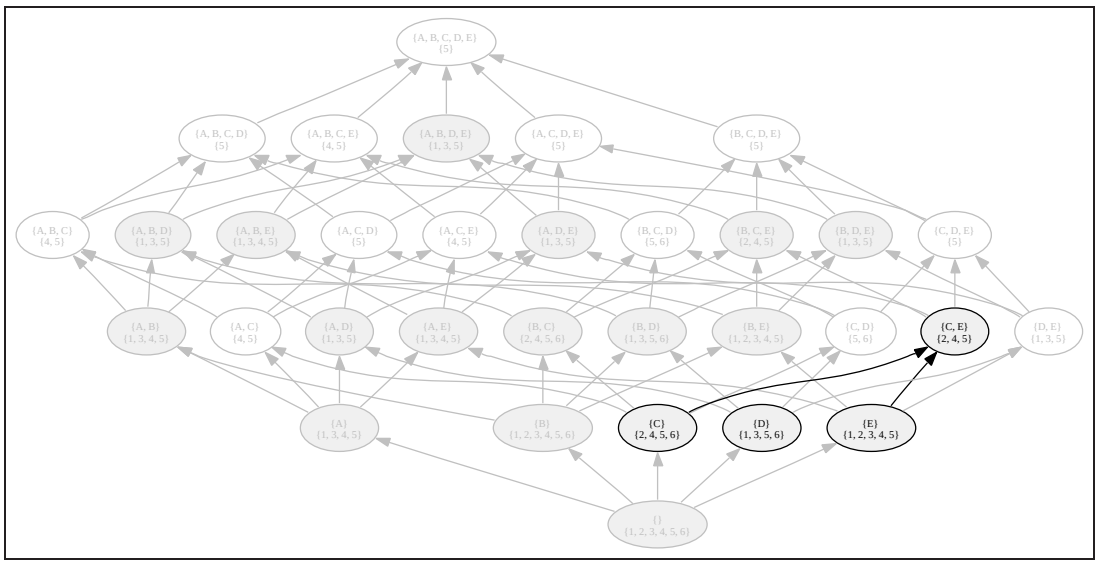


Figura A.17: Classe de equivalência $[\{C\}]$, *itemset* $\{C, E\}$

A próxima junção da classe $[\{\}]$ e do *itemset* $\{C\}$, $\{C\} \curlyvee \{E\}$, é freqüente (Figura A.17). A próxima classe é $[\{C\}]$ e o único *itemset* é $\{C, E\}$, que é liberado. A classe $[\{\}]$ é retomada e o *itemset* $\{C\}$ é liberado.

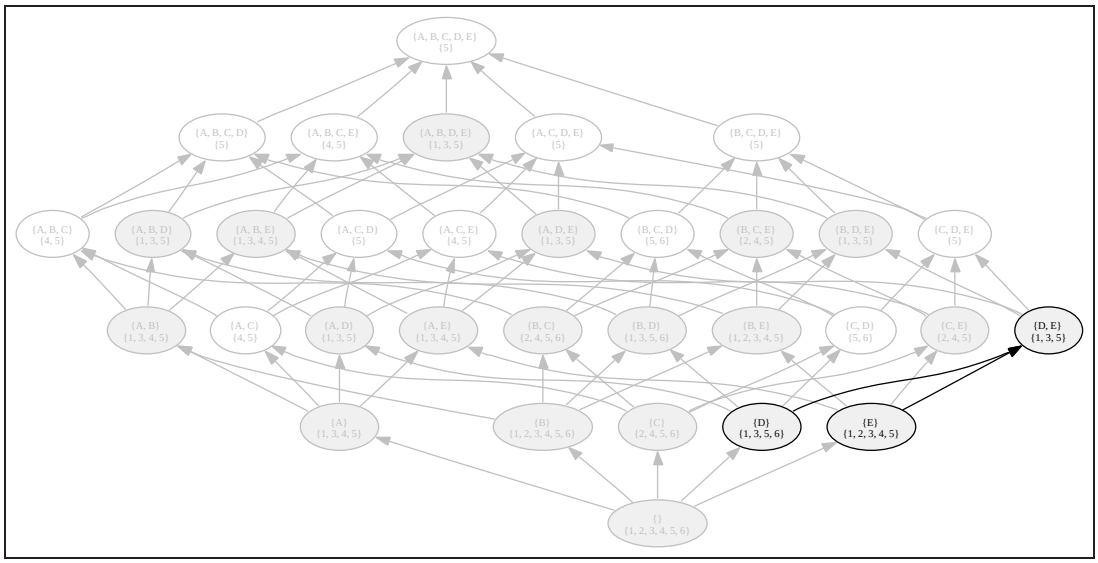


Figura A.18: Classe de equivalência $[\{D\}]$, *itemset* $\{D, E\}$

O próximo *itemset* é $\{D\}$. A primeira junção da classe $[\{\}]$ e do *itemset* $\{D\}$, $\{D\} \curlyvee \{E\}$, é freqüente (Figura A.16). A próxima classe é $[\{D\}]$ e o único *itemset* é $\{D, E\}$, que é liberado. A classe $[\{\}]$ é retomada e o *itemset* $\{D\}$ é liberado. O único *itemset* é $\{E\}$, que é liberado. O algoritmo chega ao fim.

Apêndice B

Tempos de execução

Configuração	accidents 90%	BMS-POS 1%	BMS-WebView-1 0,1%	BMS-WebView-2 0,2%	chess 80%	connect 98%	kosarak 1%
Binário $\overline{d}$ $\overline{p}$ $\overline{c}$ $\overline{r}$	125	≥300	145	≥300	138	128	≥300
Binário $\overline{d}$ $\overline{p}$ $\overline{c}$ r	127	≥300	141	≥300	82,7	91,8	≥300
Binário $\overline{d}$ $\overline{p}$ c $\overline{r}$	125	≥300	102	144	114	104	≥300
Binário $\overline{d}$ $\overline{p}$ c r	129	≥300	100	143	90,1	82,1	≥300
Binário $\overline{d}$ p $\overline{c}$ $\overline{r}$	139	≥300	184	≥300	217	160	≥300
Binário $\overline{d}$ p $\overline{c}$ r	140	≥300	164	≥300	104	119	≥300
Binário $\overline{d}$ p c $\overline{r}$	152	≥300	130	170	150	148	≥300
Binário $\overline{d}$ p c r	158	≥300	117	164	102	107	≥300
Binário d $\overline{p}$ $\overline{c}$ $\overline{r}$	111	≥300	290	≥300	10,6	27,2	≥300
Binário d $\overline{p}$ $\overline{c}$ r	81,0	≥300	193	≥300	2,30	32,3	≥300
Binário d $\overline{p}$ c $\overline{r}$	78,0	≥300	182	189	7,69	26,0	≥300
Binário d $\overline{p}$ c r	77,9	≥300	134	180	2,20	32,5	≥300
Elias δ $\overline{d}$ $\overline{p}$ $\overline{c}$ $\overline{r}$	48,9	≥300	97,4	256	30,1	27,1	≥300
Elias δ $\overline{d}$ $\overline{p}$ $\overline{c}$ r	49,1	≥300	93,0	254	17,1	20,2	≥300
Elias δ $\overline{d}$ $\overline{p}$ c $\overline{r}$	48,8	178	67,8	97,7	24,5	47,6	214
Elias δ $\overline{d}$ $\overline{p}$ c r	49,2	176	65,7	97,2	17,0	38,8	213
Elias δ $\overline{d}$ p $\overline{c}$ $\overline{r}$	50,8	≥300	114	≥300	38,1	36,5	247
Elias δ $\overline{d}$ p $\overline{c}$ r	50,7	≥300	104	≥300	20,7	31,4	250
Elias δ $\overline{d}$ p c $\overline{r}$	50,4	211	81,3	108	30,7	30,8	185
Elias δ $\overline{d}$ p c r	50,9	200	75,1	106	20,1	21,5	210
Elias δ d $\overline{p}$ $\overline{c}$ $\overline{r}$	44,8	≥300	182	≥300	4,79	12,0	≥300
Elias δ d $\overline{p}$ $\overline{c}$ r	44,7	≥300	132	≥300	1,05	16,5	≥300
Elias δ d $\overline{p}$ c $\overline{r}$	44,7	≥300	115	131	3,56	12,0	239
Elias δ d $\overline{p}$ c r	44,7	≥300	91,1	150	0,99	12,9	151
Elias γ $\overline{d}$ $\overline{p}$ $\overline{c}$ $\overline{r}$	48,6	≥300	102	≥300	28,6	34,3	193
Elias γ $\overline{d}$ $\overline{p}$ $\overline{c}$ r	48,8	≥300	98,0	≥300	16,1	20,3	185
Elias γ $\overline{d}$ $\overline{p}$ c $\overline{r}$	48,7	≥300	71,0	87,7	23,6	26,3	150
Elias γ $\overline{d}$ $\overline{p}$ c r	50,1	≥300	69,3	86,6	16,2	22,1	143
Elias γ $\overline{d}$ p $\overline{c}$ $\overline{r}$	52,0	≥300	119	268	36,4	45,0	209
Elias γ $\overline{d}$ p $\overline{c}$ r	49,1	≥300	109	260	20,0	38,4	197

Configuração					accidents	BMS-POS	BMS-WebView-1	BMS-WebView-2	chess	connect	kosarak
					90%	1%	0,1%	0,2%	80%	98%	1%
Elias γ	$\overline{\text{d}}$	p	c	$\overline{\text{r}}$	52,5	≥300	84,6	96,8	31,6	29,6	155
Elias γ	$\overline{\text{d}}$	p	c	r	55,2	≥300	77,6	94,8	20,2	24,4	147
Elias γ	d	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	48,7	≥300	187	≥300	4,55	13,4	217
Elias γ	d	$\overline{\text{p}}$	$\overline{\text{c}}$	r	48,7	≥300	138	≥300	1,03	11,8	198
Elias γ	d	$\overline{\text{p}}$	c	$\overline{\text{r}}$	48,5	≥300	116	112	3,45	14,4	161
Elias γ	d	$\overline{\text{p}}$	c	r	57,9	≥300	93,2	109	0,98	13,9	146
Fibonacci	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	108	≥300	109	244	43,7	37,7	205
Fibonacci	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	r	80,3	≥300	105	242	24,9	27,7	199
Fibonacci	$\overline{\text{d}}$	$\overline{\text{p}}$	c	$\overline{\text{r}}$	87,4	≥300	75,6	91,8	36,3	33,0	161
Fibonacci	$\overline{\text{d}}$	$\overline{\text{p}}$	c	r	79,5	≥300	74,0	90,9	25,0	27,0	152
Fibonacci	$\overline{\text{d}}$	p	$\overline{\text{c}}$	$\overline{\text{r}}$	68,5	≥300	128	285	55,6	41,3	229
Fibonacci	$\overline{\text{d}}$	p	$\overline{\text{c}}$	r	64,6	≥300	118	278	31,1	33,0	214
Fibonacci	$\overline{\text{d}}$	p	c	$\overline{\text{r}}$	82,7	≥300	90,7	102	46,2	37,2	168
Fibonacci	$\overline{\text{d}}$	p	c	r	69,3	267	83,6	100	31,1	28,2	156
Fibonacci	d	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	65,6	≥300	204	288	5,72	16,2	237
Fibonacci	d	$\overline{\text{p}}$	$\overline{\text{c}}$	r	75,9	≥300	149	214	1,19	13,2	212
Fibonacci	d	$\overline{\text{p}}$	c	$\overline{\text{r}}$	64,1	≥300	127	79,2	4,13	13,9	168
Fibonacci	d	$\overline{\text{p}}$	c	r	66,2	278	100	77,6	1,08	14,3	154
Unário	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	68,1	≥300	≥300	≥300	27,4	25,6	≥300
Unário	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	r	67,7	≥300	≥300	≥300	16,3	19,4	≥300
Unário	$\overline{\text{d}}$	$\overline{\text{p}}$	c	$\overline{\text{r}}$	66,7	≥300	≥300	≥300	24,3	21,6	≥300
Unário	$\overline{\text{d}}$	$\overline{\text{p}}$	c	r	67,1	≥300	≥300	≥300	17,0	33,9	≥300
Unário	$\overline{\text{d}}$	p	$\overline{\text{c}}$	$\overline{\text{r}}$	70,9	≥300	≥300	≥300	45,6	59,9	≥300
Unário	$\overline{\text{d}}$	p	$\overline{\text{c}}$	r	67,9	≥300	≥300	≥300	20,6	52,4	≥300
Unário	$\overline{\text{d}}$	p	c	$\overline{\text{r}}$	82,8	≥300	≥300	≥300	30,5	73,5	≥300
Unário	$\overline{\text{d}}$	p	c	r	70,5	≥300	≥300	≥300	20,4	44,3	≥300
Unário	d	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	77,0	≥300	≥300	≥300	21,2	16,4	≥300
Unário	d	$\overline{\text{p}}$	$\overline{\text{c}}$	r	67,6	≥300	≥300	≥300	8,23	17,8	≥300
Unário	d	$\overline{\text{p}}$	c	$\overline{\text{r}}$	69,0	≥300	≥300	≥300	11,2	19,2	≥300
Unário	d	$\overline{\text{p}}$	c	r	64,8	≥300	≥300	≥300	5,94	14,5	≥300

Tabela B.1: Tempo de execução total ($1/2$)

Configuração					mushroom	pumsb	pumsb_star	retail	T10I4D100K	T40I10D100K
					30%	90%	50%	0,5%	20%	10%
Binário	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	44,7	≥300	139	188	7,64	172
Binário	$\overline{\text{d}}$	$\overline{\text{p}}$	$\overline{\text{c}}$	r	36,2	≥300	129	178	6,47	172
Binário	$\overline{\text{d}}$	$\overline{\text{p}}$	c	$\overline{\text{r}}$	47,2	≥300	110	108	7,01	51,3
Binário	$\overline{\text{d}}$	$\overline{\text{p}}$	c	r	35,6	≥300	106	75,7	6,81	51,2
Binário	$\overline{\text{d}}$	p	$\overline{\text{c}}$	$\overline{\text{r}}$	63,6	≥300	187	194	6,51	238
Binário	$\overline{\text{d}}$	p	$\overline{\text{c}}$	r	60,3	≥300	170	175	6,52	228
Binário	$\overline{\text{d}}$	p	c	$\overline{\text{r}}$	53,4	≥300	144	77,0	6,68	65,5
Binário	$\overline{\text{d}}$	p	c	r	46,9	≥300	138	76,4	6,74	73,1
Binário	d	$\overline{\text{p}}$	$\overline{\text{c}}$	$\overline{\text{r}}$	18,3	65,7	59,4	198	6,41	≥300

Configuração	mushroom	pumsb	pumsb_star	retail	T10I4D100K	T40I10D100K
	30%	90%	50%	0,5%	20%	10%
Binário d $\overline{p}$ $\overline{c}$ r	7,26	55,3	54,1	175	6,65	≥300
Binário d $\overline{p}$ c $\overline{r}$	11,8	59,2	37,7	84,3	7,38	110
Binário d $\overline{p}$ c r	5,46	51,6	34,7	77,3	6,94	100
Elias δ $\overline{d}$ $\overline{p}$ $\overline{c}$ $\overline{r}$	13,9	144	41,6	71,9	5,67	225
Elias δ $\overline{d}$ $\overline{p}$ $\overline{c}$ r	11,3	119	38,9	70,8	5,60	≥300
Elias δ $\overline{d}$ $\overline{p}$ c $\overline{r}$	17,8	123	34,5	35,2	5,45	103
Elias δ $\overline{d}$ $\overline{p}$ c r	12,5	110	33,2	35,3	5,17	80,5
Elias δ $\overline{d}$ p $\overline{c}$ $\overline{r}$	14,3	178	42,9	90,7	5,55	≥300
Elias δ $\overline{d}$ p $\overline{c}$ r	10,0	146	39,5	89,4	5,03	229
Elias δ $\overline{d}$ p c $\overline{r}$	11,5	148	33,8	37,2	4,98	86,6
Elias δ $\overline{d}$ p c r	9,18	131	32,0	40,2	5,10	78,4
Elias δ d $\overline{p}$ $\overline{c}$ $\overline{r}$	6,39	28,0	23,4	101	5,07	236
Elias δ d $\overline{p}$ $\overline{c}$ r	2,76	23,4	21,2	92,4	6,45	183
Elias δ d $\overline{p}$ c $\overline{r}$	4,57	25,9	17,3	43,6	5,28	75,3
Elias δ d $\overline{p}$ c r	2,21	22,4	16,1	39,9	5,23	74,4
Elias γ $\overline{d}$ $\overline{p}$ $\overline{c}$ $\overline{r}$	12,6	137	38,3	66,2	4,99	182
Elias γ $\overline{d}$ $\overline{p}$ $\overline{c}$ r	17,9	113	36,3	65,2	5,11	160
Elias γ $\overline{d}$ $\overline{p}$ c $\overline{r}$	12,7	117	32,5	29,8	5,00	58,4
Elias γ $\overline{d}$ $\overline{p}$ c r	11,3	105	31,5	32,0	5,88	44,1
Elias γ $\overline{d}$ p $\overline{c}$ $\overline{r}$	14,1	171	40,7	79,1	5,12	132
Elias γ $\overline{d}$ p $\overline{c}$ r	10,3	141	37,8	78,5	5,45	134
Elias γ $\overline{d}$ p c $\overline{r}$	12,0	146	32,7	33,7	5,17	46,0
Elias γ $\overline{d}$ p c r	9,71	131	31,1	31,6	5,32	44,1
Elias γ d $\overline{p}$ $\overline{c}$ $\overline{r}$	6,95	28,0	22,4	90,3	5,29	126
Elias γ d $\overline{p}$ $\overline{c}$ r	2,87	23,2	20,2	83,1	5,22	124
Elias γ d $\overline{p}$ c $\overline{r}$	4,64	25,6	16,7	37,6	5,16	44,4
Elias γ d $\overline{p}$ c r	2,14	22,4	15,6	34,7	5,27	45,8
Fibonacci $\overline{d}$ $\overline{p}$ $\overline{c}$ $\overline{r}$	18,8	204	49,1	72,1	5,19	168
Fibonacci $\overline{d}$ $\overline{p}$ $\overline{c}$ r	14,7	169	45,9	71,9	5,27	173
Fibonacci $\overline{d}$ $\overline{p}$ c $\overline{r}$	15,9	173	40,3	31,4	5,23	65,5
Fibonacci $\overline{d}$ $\overline{p}$ c r	13,9	156	39,1	31,2	5,30	68,5
Fibonacci $\overline{d}$ p $\overline{c}$ $\overline{r}$	20,2	258	54,0	90,5	5,42	228
Fibonacci $\overline{d}$ p $\overline{c}$ r	16,0	208	50,3	84,6	5,48	221
Fibonacci $\overline{d}$ p c $\overline{r}$	17,8	215	43,1	37,0	5,30	74,9
Fibonacci $\overline{d}$ p c r	15,5	191	41,3	36,8	5,53	77,6
Fibonacci d $\overline{p}$ $\overline{c}$ $\overline{r}$	8,67	30,8	25,5	99,4	5,32	225
Fibonacci d $\overline{p}$ $\overline{c}$ r	3,46	25,4	23,0	88,6	5,50	220
Fibonacci d $\overline{p}$ c $\overline{r}$	5,88	27,8	18,7	40,6	5,40	81,0
Fibonacci d $\overline{p}$ c r	2,67	24,3	17,1	37,1	5,42	73,3
Unário $\overline{d}$ $\overline{p}$ $\overline{c}$ $\overline{r}$	14,7	136	42,7	≥300	12,3	239
Unário $\overline{d}$ $\overline{p}$ $\overline{c}$ r	12,0	114	40,6	≥300	12,4	235
Unário $\overline{d}$ $\overline{p}$ c $\overline{r}$	12,4	115	36,3	≥300	13,1	78,1
Unário $\overline{d}$ $\overline{p}$ c r	16,9	105	35,6	≥300	11,5	77,6
Unário $\overline{d}$ p $\overline{c}$ $\overline{r}$	14,8	172	45,9	≥300	11,6	246
Unário $\overline{d}$ p $\overline{c}$ r	11,7	141	43,4	≥300	11,8	241
Unário $\overline{d}$ p c $\overline{r}$	12,8	145	38,1	≥300	11,9	80,4

Configuração	mushroom	pumsb	pumsb_star	retail	T10I4D100K	T40I10D100K
	30%	90%	50%	0,5%	20%	10%
Unário $\overline{d}$ p c r	11,5	131	37,0	$\geq$300	11,5	80,5
Unário d $\overline{p}$ $\overline{c}$ $\overline{r}$	10,2	68,2	32,6	$\geq$300	11,7	236
Unário d $\overline{p}$ $\overline{c}$ r	4,39	55,7	29,6	$\geq$300	12,3	236
Unário d $\overline{p}$ c $\overline{r}$	6,95	58,4	27,2	$\geq$300	11,8	78,8
Unário d $\overline{p}$ c r	3,70	51,7	25,0	$\geq$300	11,3	80,7

Tabela B.2: Tempo de execução total ($^2/_2$)

Bibliografia

[1] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, D.C., Apr. 1993.

[2] C. Borgelt. Efficient implementations of apriori and eclat. In *Workshop on Frequent Item Set Mining Implementations*, Melbourne, FL, USA, 2003.

[3] T. Brijs, G. Swinnen, K. Vanhoof, and G. Wets. Using association rules for product assortment decisions: A case study. In *Knowledge Discovery and Data Mining*, pages 254–260, 1999.

[4] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *Proceedings of the 17th International Conference on Data Engineering*, pages 443–452, Heidelberg, Germany, Apr. 2001.

[5] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, Mar. 1975.

[6] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From data mining to knowledge discovery in databases. *Ai Magazine*, 17:37–54, 1996.

[7] G. Gardarin, P. Pucheral, and F. Wu. Bitmap based algorithms for mining association rules. In *Actes des journes Bases de Donnes Avances (BDA'98)*, Hammamet, Tunisia, Oct. 1998.

[8] K. Geurts, G. Wets, T. Brijs, and K. Vanhoof. Profiling high-frequency accident locations using association rules. In *Transportation Research Record*, volume 1840, pages 123–130, Jan. 2003.

[9] K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. In *1st IEEE International Conference on Data Mining*, San Jose, Nov. 2001.

[10] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In W. Chen, J. Naughton, and P. A. Bernstein, editors, *2000 ACM SIGMOD Intl. Conference on Management of Data*, pages 1–12. ACM Press, May 2000.

[11] R. J. B. Jr. Efficiently mining long patterns from databases. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 85–93, 1998.

[12] R. Kohavi, C. Brodley, B. Frasca, L. Mason, and Z. Zheng. KDD-Cup 2000 organizers' report: Peeling the onion. *SIGKDD Explorations*, 2(2):86–98, 2000. http://www.ecn.purdue.edu/KDDCUP.

[13] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. Adaptive and resource-aware mining of frequent sets. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)*, page 338. IEEE Computer Society, 2002.

[14] P. Shenoy, J. R. Haritsa, S. Sundarshan, G. Bhalotia, M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. In *ACM SIGMOD Intl. Conference Management of Data*, pages 22–33, Dallas, Texas, May 2000.

[15] E. W. Weisstein. Mathworld. `http://mathworld.wolfram.com/`.

[16] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[17] M. J. Zaki. Generating non-redundant association rules. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 34–43, Boston, Massachusetts, 2000.

[18] M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390, May 2000.

[19] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. In *9th International Conference on Knowledge Discovery and Data Mining*, Washington, DC, Aug. 2003.

[20] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *3rd International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 283–286, Newport, CA, Aug. 1997.