Politechnika Poznańska
Wydział Informatyki
Instytut Informatyki

Praca dyplomowa magisterska

# INTEGRACJA DRZEW PREFIKSOWYCH W PRZETWARZANIU ZBIORÓW ZAPYTAŃ EKSPLORACYJNYCH ALGORYTMEM APRIORI

Szymon Dolata

Promotor
Dr inż. Marek Wojciechowski

Poznań, 2014 r.

Tutaj przychodzi karta pracy dyplomowej;
oryginał wstawiamy do wersji dla archiwum PP, w pozostałych kopiach wstawiamy ksero.

# Spis treści

# Rozdział 1

# Wstęp

## 1.1 Integracja drzew prefiksowych w przetwarzaniu zbiorów zapytań eksploracyjnych algorytmem Apriori

Odkrywanie zbiorów częstych i generowanie na ich podstawie reguł asocjacyjnych, to problem sformułowany w kontekście analizy koszyka zakupów. Głównym celem jest szukanie prawidłowości w zachowaniu klientów supermarketów. Szybko znalazł on również zastosowanie w wielu innych dziedzinach, takich jak chociażby analiza działalności firm wysyłkowych, sklepów internetowych etc. Z wykorzystaniem znalezionych zbiorów częstych i wygenerowanych reguł dąży się do tego, aby można było wnioskować (z dużym prawdopodobieństwem), że niektóre produkty współwystępują ze sobą. Informacje takie, zwłaszcza jeśli wyrażone w formie zasad, często mogą być stosowane w celu zwiększenia sprzedaży danych produktów - na przykład poprzez odpowiednie rozmieszczenie ich na półkach w supermarkecie lub na stronach katalogu wysyłkowego (umieszczenie obok siebie może zachęcić jeszcze więcej klientów do zakupu ich razem) lub poprzez bezpośrednie sugerowanie klientom produktów, którymi mogą być zainteresowani. Oczywistym jest, że należy szukać tylko takich reguł asocjacyjnych, które są wiarygodne i niosą ze sobą jakąś informację. Dlatego też powstały algorytmy służące do znajdywania tychże reguł oraz wskaźniki do ich oceny.

Głównym problemem indukcji reguł asocjacyjnych jest to, że istnieje bardzo wiele możliwości. Przykładowo w zakresie produktów z supermarketu, których może być nawet kilka tysięcy, istnieją miliardy możliwych reguł. Tak ogromna ilość nie może być przetwarzana sekwencyjnie. Stąd potrzeba wydajnych algorytmów, które ograniczają przestrzeń wyszukiwania i sprawdzają jedynie podzbiór wszystkich reguł. Jednym z takich algorytmów jest Apriori zaproponowany w [AS94]. Inne znane podejścia, które pojawiły się później, to m.in. algorytmy FP-growth ([HPY00]) czy Dynamic Itemset Counting ([BMUT97]). Apriori doczekało się również wielu modyfikacji. Kładły one nacisk na optymalizację czasu wykonania algorytmu przede wszystkim poprzez zmniejszenie liczby odczytów bazy danych, ulepszenie procedury generowania kandydatów lub zmiany struktury wykorzystywanej przez algorytm. Jego podstawowa wersja trzyma kandydatów w drzewie haszowym. W praktyce jednak lepsze okazały się implementacje Apriori gdzie drzewo haszowe zastąpiono znacznie prostszą strukturą drzewa prefiksowego. Powstało kilka rozwiązań wykorzystujących tę strukturę. Są to m.in. rozwiązania Borgelta ([Bor03]), Bodona ([Bod03]) czy też Goethalsa ([Goe02]).

Prowadzone są także badania w kwestii współbieżnego wykonywania algorytmów eksploracyjnych na tym samym zbiorze danych. W ostatnich latach zaproponowane zostały metody Common Counting ([WZ03]) oraz Common Candidate Tree ([GW09]). Są one wynikiem eksperymentów nad optymalizacją wykonania kilku zadań Apriori uruchomionych współbieżnie na nakładających się podzbiorach tabeli z danymi. Metody sprowadzały się do:
- Integracji odczytów współdzielonych danych z dysku;
- Integracji drzew haszowych w jedno drzewo gdzie kandydaci mają kilka liczników (po jednym dla zadania eksploracji).
Udowodniono, że im większy stopień nakładania się danych tym większa przewaga tych algorytmów nad oryginalnym, sekwencyjnym rozwiązaniem. Jednakże wykorzystują one ten sam mechanizm generowania kandydatów oraz tę samą strukturę co oryginalny Apriori ([AS94]). Do tej pory nie została zaimplementowana modyfikacja Common Counting i Common Candidate Tree dla jednej z szybszych wersji Apriori z drzewem prefiksowym i to właśnie jest celem tej pracy.

## 1.2 Cel i zakres pracy

Tak jak wspomniano, ogólnym celem pracy jest implementacja dwóch algorytmów wykonania zbioru zapytań odkrywających zbiory częste - które dotychczas implementowane były na drzewie haszowym - z wykorzystaniem drzew prefiksowych.

Na ten ogólny cel pracy składają się następujące cele szczegółowe:
- przedstawienie, analiza i porównanie istniejących rozwiązań dotyczących tematyki pracy
- implementacja modyfikacji Common Counting i Common Candidate Tree dla Apriori z drzewem prefiksowym;
- przetestowanie wydajności zaimplementowanych algorytmów i analiza uzyskanych wyników.

## 1.3 Struktura pracy

Struktura pracy jest następująca:
– w rozdziale 2 omówiono podstawowe pojęcia i definicje wykorzystywane w pracy;
– w rozdziale 3 przedstawiono istniejące rozwiązania i algorytmy, związane z tematem pracy;
– w rozdziale 4 przedstawiono ideę, opis i cechy algorytmów;
– w rozdziale 5 przeanalizowano wyniki uzyskane podczas badań działania algorytmów dla różnych parametrów i danych wejściowych;
– w rozdziale 6 przedstawiono wnioski i uwagi do pracy.

# Rozdział 2

# Podstawowe pojęcia i definicje

## 2.1 Wstęp

W poniższym rozdziale przedstawiono podstawowe pojęcia i definicje wykorzystywane w pracy. Związane są one z omawianym tematem. Na początku opisano terminy niezbędne dla sformułowania problemu. Ostatnie dwa pojęcia są natomiast kluczowe dla rozważanego w pracy problemu badawczego.

## 2.2 Lista pojęć i definicji

### 2.2.1 Transakcja i element transakcji

Danymi wejściowymi dla odkrywania zbiorów częstych i reguł asocjacyjnych jest zbiór transakcji zdefiniowanych na zbiorze elementów. Tymi elementami mogą być produkty w sklepie, usługi, książki etc. Ważne jest, aby te elementy można było w łatwy sposób od siebie odróżnić. Jeśli $I = \{i_1, i_2, \cdots, i_n\}$ (ang. *item base* ), to zbiór wszystkich możliwych elementów, to dowolny niepusty podzbiór X zbioru $X \subseteq I$ nazywamy zbiorem elementów (ang. *itemset*). Natomiast zbiór elementów o mocy $k$, to taki zbiór, który posiada dokładnie k elementów (ang. *k-itemset*).

Transakcja jest natomiast przykładowym zbiorem elementów, np. zbiorem produktów, które zostały kupione przez danego klienta. Jako że transakcje mogą się powtarzać (może istnieć kilku klientów, którzy kupili dokładnie takie same produkty) każda transakcja w bazie danych ma przypisany unikalny identyfikator $TID$. Dzięki temu nie ma problemu występowania duplikatów, pomimo tego, że mogą występować transakcje zawierające ten sam zestaw elementów.

Należy także zwrócić uwagę, że w większości rozważanych przypadków nie są znane wszystkie elementy, jakie mogą znaleźć się w zbiorze $I$. Przyjmuje się wówczas, że ten zbiór jest sumą elementów występujących we wszystkich transakcjach.

### 2.2.2 Reguła asocjacyjna

Reguła asocjacyjna jest implikacją, która daje możliwość przewidywania jednoczesnego wystąpienia dwóch zjawisk i zachowań, współzależnych od siebie. Innymi słowy jest to schemat, pozwalający - z określonym prawdopodobieństwem - założyć, że jeśli nastąpiło zdarzenie A, to nastąpi również zdarzenie B. Reguła może zostać wyrażona w postaci $R = "X \rightarrow Y"$ (gdzie $X$ i $Y$ to zbiory elementów). W kontekście problemu koszyka zakupów sprowadza się do reguł w stylu: *Jeżeli klient kupił pieluszki, to (z określonym prawdopodobieństwem) kupi też piwo.*

### 2.2.3 Wsparcie zbioru elementów

Jeśli $X$ oznacza dowolny podzbiór $I$, to (bezwzględne) wsparcie tego zbioru jest równe $U$ - liczbie transakcji $T$ w bazie danych transakcji $D$, które zawierają wszystkie elementy danego zbioru $X$. Wsparcie względne jest to z kolei procent (lub ułamek) transakcji w $D$, które zawierają $X$. Obliczamy ze wzoru

$$sup_{rel}(X) = \frac{|U|}{|D|} * 100\%$$

Dla algorytmu Apriori określa się próg minimalnego wsparcia *minsup*, który również może być wyrażony w dwojakiej postaci - jako liczba wystąpień lub procent wszystkich transakcji. W po-

szukiwaniu zbiorów częstych interesujące są tylko te reguły, dla których $sup(X) \geq minsup$, gdzie $sup(X)$, to przyjęty w pracy sposób zapisu wsparcia zbioru elementów.

### 2.2.4 Ufność reguły asocjacyjnej

Ufność reguły asocjacyjnej jest miarą jakości danej reguły. Miara ta została przedstawiona przez autorów algorytmu Apriori [AS94]. Dla reguły asocjacyjnej postaci $R = "X \rightarrow Y"$ ufność wyraża się jako stosunek wsparcia sumy wszystkich elementów występujących w regule (w tym przypadku $sup(X \cup Y)$) do wsparcia poprzednika reguły (tutaj $sup(X)$).

$$conf(R) = \frac{sup(X \cup Y)}{sup(X)}$$

Należy dodać, że nie ma znaczenia czy wykorzystywane jest wsparcie absolutne czy relatywne. Istotne jest natomiast to, aby w zarówno dla licznika i mianownika wykorzystany był ten sam typ wsparcia. Z powyższego wzoru wynika, że ufność reguły asocjacyjnej, to stosunek liczby przypadków, w których jest ona poprawna, do wszystkich przypadków gdzie mogłaby zostać zastosowana. Przykład: $R = wino \wedge chleb \rightarrow ser$ - jeśli klient kupuje wino i chleb, to ta reguła ma zastosowanie i mówi, że można oczekiwać, że dany klient kupi również ser. Jest możliwe, że ta reguła - dla danego klienta - będzie poprawna lub nie. Interesują informacją jest to jak dobra jest reguła, czyli jak często jest poprawna (jak często klient, który kupuje wino i chleb kupuje również ser). Taką właśnie informację uzyskuje się poprzez obliczenie ufności reguły asocjacyjnej. Oczywiście w przypadku gdy klient nie kupił chleba lub/i wina, to reguła nie znajduje zastosowania, a dana transakcja nie wpływa na $conf(R)$.

### 2.2.5 Wsparcie reguły asocjacyjnej

Wsparcie reguły asocjacyjnej postaci $R = "X \rightarrow Y"$ odpowiada wsparciu sumy zbiorów $X$ i $Y$ ([AS94]). Miara ta informuje o tym jak często dana reguła jest prawidłowa. Nieco odmienna definicja została przedstawiona i wykorzystana w [Bor03]. Różnica polega na tym, że wartość ta wyrażona jest jako liczba przypadków, w których reguła jest stosowalna (nawet jeśli reguła może okazać się fałszywa). Zatem dla powyżej postaci byłoby to wsparcie zbioru $X$.

Wparcie może być stosowane do filtrowania. Dla ustalonego *minsup* szuka się tylko takich reguł, których wsparcie jest nie mniejsze od *minsup*. Oznacza to, że interesujące są tylko te reguły, które wystąpiły co najmniej daną liczbę razy. W algorytmach wyszukiwania reguł asocjacyjnych stosuje się progi minimalnego wsparcia oraz minimalnej ufności. Dzięki temu w otrzymanych wynikach nie są uwzględnione mało wartościowe reguły.

### 2.2.6 Zbiór częsty

Zbiorem częstym nazywamy taki niepusty podzbiór zbioru $I$, dla którego wsparcie jest równe co najmniej wartości *minsup*.

### 2.2.7 Zbiór domknięty

Zbiorem domkniętym nazywamy taki zbiór częsty, dla którego nie istnieje żaden nadzbiór właściwy mający dokładnie takie samo wsparcie.

### 2.2.8 Zbiór maksymalny

Zbiorem maksymalnym nazywamy taki zbiór częsty, dla którego nie istnieje żaden nadzbiór właściwy, który byłby zbiorem częstym.

### 2.2.9 Zapytanie eksploracyjne

Zapytanie eksploracyjne jest uporządkowaną piątką $dmq = (R, a, \Sigma, \Phi, minsup)$, gdzie $R$ - relacja bazy danych, $a$ - atrybut relacji $R$, $\Sigma$ - wyrażenie warunkowe dotyczące atrybutów $R$ nazywane predykatem selekcji danych, $\Phi$ - wyrażenie warunkowe dotyczące odkrywanych zbiorów częstych nazywane predykatem selekcji wzorców, *minsup* - próg minimalnego wsparcia. Wynikiem zapytania eksploracyjnego są zbiory częste odkryte w $\pi_a \sigma_\Sigma R$, które spełniają predykat $\Phi$ i posiadają $wsparcie \geq minsup$ ($\pi$ - relacyjna operacja projekcji, $\sigma$ - relacyjna operacja selekcji).

### 2.2.10 Zbiór elementarnych predykatów selekcji danych

Zbiorem elementarnych predykatów selekcji danych dla zbioru zapytań eksploracyjnych $DMQ = \{dmq_1, dmq_2, \ldots, dmq_n\}$ nazywamy najmniej liczny zbiór $S = \{s_1, s_2, \ldots, s_k\}$ ($s_i$ - $i$-ty predykat selekcji danych z relacji $R$), dla którego dla każdej pary $u, v (u \neq v)$ zachodzi $\sigma s_u R \cap \sigma s_v R = \emptyset$ i dla każdego $dmq_i$ istnieją liczby całkowite $a, b, \ldots, m$, takie że $\sigma_{\Sigma_i} R = \sigma_{s_a} R \cup \sigma_{s_b} R \cup \cdots \cup \sigma_{s_m} R$. Zbiór ten jest reprezentacją podziału bazy danych na partycje, które zostały wyznaczone przez nakładające się źródłowe zbiory danych zapytań.

# Rozdział 3

# Podłoże teoretyczne

## 3.1 Wstęp

Kolejny rozdział przedstawia aktualne metody i istniejące algorytmy związane z tematem pracy. Poza podstawowym algorytmem Apriori ([AS94]), który używa drzew haszowych do przechowywania kandydatów, opisano trzy modyfikacje tego algorytmu. Główna różnica polega na tym, że wykorzystują one inną strukturę, a mianowicie drzewa prefiksowe. Są to rozwiązania zaproponowane przez Christina Borgelta ([Bor03]), Ferenca Bodona ([Bod03]) oraz Barta Goethalsa ([Goe02]). Ze względu na wykorzystanie prostszej struktury okazały się one szybsze od standardowego algorytmu.

Innym problemem jest optymalizacja wykonania kilku zadań Apriori uruchomionych współbieżnie na nakładających się podzbiorach tabeli z danymi. Metody z tym związane to Common Counting ([WZ03]) i Common Candidate Tree ([GW09]). Oparte są one o implementację Apriori z zastosowaniem drzew haszowych. Brakuje jednak adaptacji tych algorytmów, polegającej na zmianie struktury na drzewa prefiksowe. Właśnie taka modyfikacja została wprowadzona, a uzyskane efekty opisano w kolejnych rozdziałach niniejszej pracy.

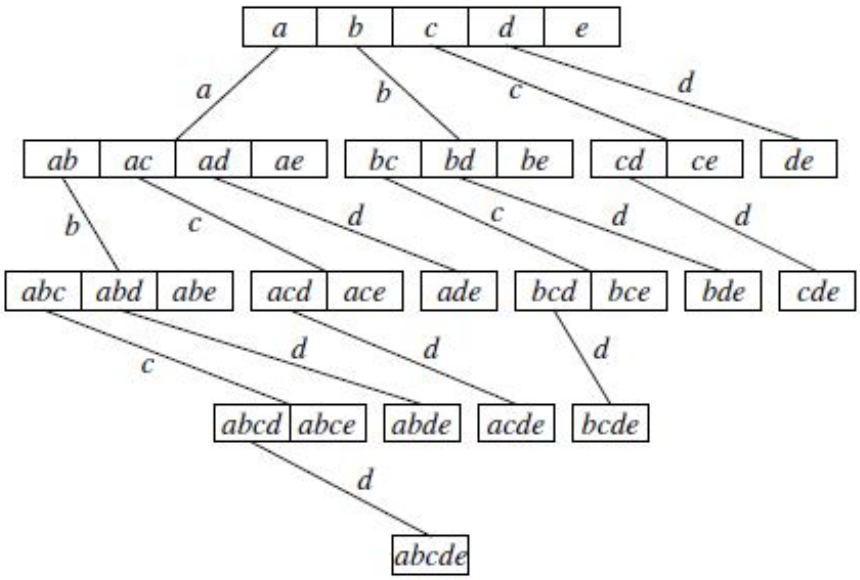
## 3.2 Przegląd istniejących rozwiązań

### 3.2.1 Algorytm Apriori [AS94]

Algorytm Apriori jest algorytmem eksploracji poziomej. Szuka zbiorów częstych o rozmiarach $1, 2, \ldots, k$. Algorytm rozpoczyna od zbiorów o rozmiarze 1 i następnie zwiększa ten rozmiar w kolejnych iteracjach. Elementy każdej transakcji są uporządkowane leksykograficznie - jeżeli nawet transakcje nie są posortowane, to krokiem wstępnym algorytmu może być leksykograficzne uporządkowanie elementów transakcji ([Mor13]). Po pierwszym kroku zebrane są zatem wszystkie elementy występujące w transakcjach (w postaci zbiorów jednoelementowych). Następnie sprawdzane jest, które z nich posiadają wsparcie nie mniejsze niż *minsup*. Elementy niespełniające tego wymagania są odrzucane. Pozostałe służą do utworzenia dwuelementowych zbiorów kandydujących (ang. *candidate itemsets*). Dla wygenerowanych zbiorów sprawdzane jest czy posiadają wsparcie równe co najmniej *minsup*. Jeśli tak, to taki zbiór jest dodawany do listy zbiorów częstych i w kolejnej iteracji jest wykorzystywany (wraz z innymi zbiorami z tejże listy) do generowania zbiorów kandydatów o rozmiarze o 1 większym. Wsparcie zbiorów sprawdzane jest na podstawie odczytu danych z bazy danych. Algorytm zatrzymuje się gdy nie ma już możliwości generowania kolejnych zbiorów. W wyniku jego działania zwracana jest suma $k$-elementowych zbiorów częstych ($k = 1, 2, \ldots$), która może zostać wykorzystana do generowania reguł asocjacyjnych.

### 3.2.2 Algorytm Apriori - implementacja Christina Borgelta [Bor03]

W tym podrozdziale opisany została adaptacja algorytmu Apriori autorstwa Borgelta. Odstępstwo od oryginału polega przede wszystkim na zmianie struktury, czyli wykorzystaniu drzew prefiksowych zamiast drzew haszowych. Rysunek 3.1 przedstawia taką właśnie strukturę. Drzewo budowane jest od korzenia do liści (ang. *top-down*), przy sprawdzaniu czy dana gałąź może zawierać zbiory częste. Jeśli ten warunek nie jest spełniony, to następuje odcięcie i ta część drzewa nie jest dalej generowana ani analizowana, gdyż nie zawiera przypadków, które powinny zostać uwzględnione w wynikach działania algorytmu.




Rysunek 3.1: Drzewo prefiksowe dla 5 elementów (pusty zbiór w korzeniu nie zostały zaprezentowany).

### Struktura wierzchołka

Wierzchołki drzewa prefiksowego reprezentowane mogą być na 3 sposoby:
– jako proste wektory liczb całkowitych (w tym przypadku następuje niejawne powiązanie każdego z możliwych elementów z pojedynczym polem wektora)
– jako wektory przechowujące licznik wystąpień wraz z identyfikatorem elementu
– jako drzewa haszowe wiążące dany element z licznikiem jego wystąpień

W pierwszym przypadku plusem jest to, że nie potrzeba pamięci na składowanie identyfikatorów elementów oraz fakt szybkiego dostępu do licznika. Minusem jest z kolei problem powstawania dziur w wektorze, tzn. wektor uwzględnia również informacje o licznikach elementów, o których wiadomo (na podstawie wcześniejszych iteracji), że nie mogą być częste. Jest to najlepszy wybór w przypadku, gdy priorytetem jest szybkość wykonania.

Druga struktura pozwala niwelować wyżej opisany problem i przechowuje jedynie te liczniki, które są nadal potrzebne. Minusem jest natomiast fakt zapotrzebowania na dodatkową pamięć na przechowywanie identyfikatorów oraz wolniejszy dostęp spowodowany koniecznością wyszukania licznika powiązanego z danym elementem. Mimo to jeśli optymalizacja pod kątem użycia pamięci jest istotniejsza od szybkości wykonania, to ta właśnie struktura powinna zostać wykorzystana.

Trzecia propozycja daje możliwość szybkiego dostępu do licznika, ale wymaga większej ilości pamięci. Jednakże to rozwiązanie nie daje łatwej możliwości sortowania elementów, dlatego też zostało odrzucona przez Borgelta.

### Reprezentacja elementu

Reprezentacja elementu ma duży wpływ na wspomniany wcześniej problem powstawania dziur w wektorze. Jeśli elementy są kodowane jako liczby całkowite, to jest to korzystne dla ograniczenia liczby i wielkości dziur. W przeciwnym razie problem ten może się nasilać. Dla dodatkowego zmniejszenia występowania tego problemu stosuje się podejście heurystyczne polegające na posortowaniu elementów malejąco względem częstości ich występowania i nadaniu elementom o podobnej liczbie wystąpień podobnego identyfikatora (przy równoczesnym odrzuceniu tych, które występują mniej razy niż wynosi *minsup*).

### Przetwarzanie transakcji

Struktura drzewa prefiksowego pozwala na proste przetwarzanie transakcji, w których elementy są posortowane. Dla poszczególnych wierzchołków odbywa się to rekurencyjnie i przebiega w następujący sposób: (1) idź do dziecka odpowiadającego pierwszemu elementowi w transakcji i

przetwarzaj kolejne elementy rekurencyjnie dla tego dziecka i (2) usuń pierwszy element z transakcji i przetwarzaj dla danego wierzchołka. Krok 1 może zostać zakończony w momencie osiągnięcia poziomu drzewa, dla którego testowane są zbiory kandydatów. Wówczas algorytm nie przechodzi dalej do dziecka, nawet jeśli istnieją kolejne elementy w transakcji.

Dzięki posortowaniu elementów można dodatkowo zoptymalizować działanie algorytmu. Po pierwsze można opuścić analizę elementów mających niższy identyfikator niż ten w przetwarzanym wierzchołku. Po drugie w przypadku gdy pierwszy element ma wyższy identyfikator niż ostatni element w wierzchołku, to można zrezygnować z rekurencyjnego przetwarzania danej transakcji dla wierzchołka. Dodatkowo jeśli elementów transakcji jest mniej niż zawierają wierzchołki na aktualnie analizowanym poziomie, to można zakończyć rekurencję, gdyż pożądany poziom drzewa nie zostanie osiągnięty.

Najłatwiej jest przetwarzać transakcje stosując wyżej opisaną metodę. Jednakże ze względu na wysoki koszt operacji rekurencyjnych możliwe są pewne ulepszenia. Jednym z nich jest pogrupowanie podobnych transakcji i umieszczenie ich w drzewie prefiksowym. Mogą być one przetwarzane wspólnie, ale należy mieć na uwadze, aby zyski były większe niż koszty stworzenia takich drzew i odpowiedniego przypisania do nich transakcji.

Kolejnym zabiegiem usprawniającym wykonanie algorytmu jest usuwanie z transakcji elementów, które nie wchodzą w skład wierzchołków na poziomie $k-1$ (gdzie $k$ to liczba elementów w wierzchołkach aktualnie analizowanego poziomu drzewa). Zmniejsza to rozmiar transakcji i liczbę wywołań rekurencyjnych, ale trzeba pamiętać, że w przypadku zastosowania drzew prefiksowych do grupowania transakcji wymagana jest kosztowna operacja przebudowania tychże drzew.

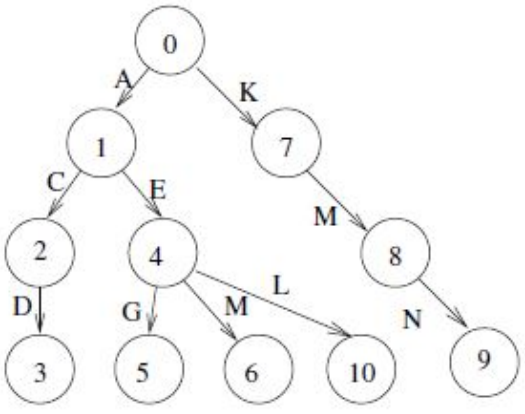
Algorytm działa szybciej niż oryginalny Apriori, a wybór struktury wierzchołka i wybranych optymalizacji zależy głównie od badanego zbioru danych.

### 3.2.3 Algorytm Apriori - implementacja Ferenca Bodona [Bod03]

W tej sekcji opisana została modyfikacja algorytmu Apriori zaproponowana przez Ferenca Bodona. Podobnie jak w przypadku wyżej opisanej implementacji Borgelta ([Bor03]) zastosowano strukturę drzewa prefiksowego oraz kilka dodatkowych optymalizacji opisanych poniżej.

#### Struktura drzewa

Drzewo budowane jest wierzchołka, który znajduje się na głębokości 0. Jest ono skierowane w dół, tzn. wierzchołki na poziomie $d$ wskazują na wierzchołki na poziomie $d+1$. Krawędzie drzewa są oznaczone, a ich etykiety odpowiadają elementom transakcji, natomiast w wierzchołkach umieszczony jest identyfikator wierzchołka. Przykładowe drzewo zostało przedstawione na rysunku 3.2. W drzewie składowani są kandydaci (wraz z licznikami wystąpień), a także zbiory częste. Pozwala




Rysunek 3.2: Przykładowe drzewo prefiksowe dla 5 kandydatów $\{A, C, D\}$, $\{A, E, G\}$, $\{A, E, L\}$, $\{A, E, M\}$, $\{K, M, N\}$.

to na łatwe i szybkie generowanie kandydatów. Transakcje przetwarzane są sekwencyjnie i dla każdej transakcji $t$ wyznaczany jest zbiór $X$ $k$-elementowych (posortowanych) podzbiorów $t$, gdzie $k$ to rozmiar aktualnie szukanych kandydatów. Jednakże dąży się do niegenerowania wszystkich

możliwych podzbiorów i jak najwcześniejszych wycofań. W momencie odnalezienia $X$ w drzewie inkrementuje się licznik dla danego kandydata i kontynuuje przetwarzanie w głąb drzewa tylko jeśli algorytm znajduje się na głębokości $d$, po przejściu krawędzią z etykietą $j$, istnieje krawędź wychodząca oznaczona etykietą $i$, taką że $i \in t$ oraz $i < |t| - k + d + 1$. Również odnajdywanie reguł asocjacyjnych jest szybsze. Wynika to bezpośrednio z faktu, że - dzięki wykorzystaniu drzewa prefiksowego - obliczanie wsparcia zbiorów elementów jest wydajniejsze. Kolejnym plusem jest łatwiejsza implementacja co przekłada się na łatwiejszy w utrzymaniu kod.

### Metody obliczania wsparcia

Wsparcie jest obliczane poprzez sekwencyjną analizę transakcji i ustalanie którzy kandydaci znajdują się w danej transakcji $t$, a którzy nie. Jest to kosztowna operacja powtarzana i determinuje ona czas wykonania całego algorytmu. Może być wykonana na dwa sposoby. Oba startują z korzenia drzewa i opierają się na rekurencji. Pierwszy z nich dla każdego elementu $t$ sprawdza czy istnieje krawędź z etykietą odpowiadającą elementowi. Sprowadza się to do wyszukiwania w posortowanym zbiorze. Druga metoda operuje na dwóch iteratorach - pierwszy z nich iteruje po elementach $t$, drugi po krawędziach wierzchołka. Oba rozpoczynają iterację w pierwszym elemencie (odpowiednio transakcji i drzewa) i jeśli wskazują na ten sam element, to oba przechodzą do dalszej analizy. W przeciwnym wypadku iterator wskazujący na niższy element jest zwiększany. Przetwarzanie kończy się gdy jeden z iteratorów osiągnie odpowiednio koniec transakcji lub ostatnią gałąź.

Obie metody są poprawne, a różnica polega w sposobie wywołań rekurencyjnych. Dla pierwszej metody liczba kroków potrzebna do wykonania z poziomu wierzchołka o $m$ krawędziach wychodzących odpowiada $min\{|t|log_2 m, mlog_2|t|\}$, dla drugiej mieści się w przedziale $< min\{m, |t|\}, m+|t| >$. W testach przeprowadzonych przez Bodona ([Bod03]) druga metoda okazała się średnio dwukrotnie szybsza i to ona została wykorzystana do dalszych rozważań.

### Modyfikacje optymalizujące czas wykonania

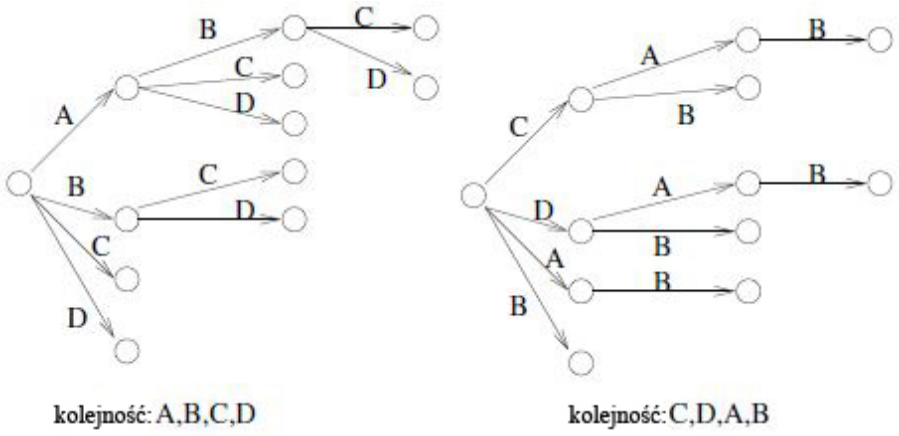
W ([Bod03]) zaproponowano kilka zabiegów optymalizujących czas wykonania algorytmu. Skupiają się one przede wszystkim na skróceniu czasu potrzebnego na obliczanie wsparcia kandydatów. Z reguły wymagają więcej pamięci, aby składować dodatkowe informacje, ale mimo to w ostatecznym rozrachunku są opłacane. Zostały one przedstawione w poniżej.

Pierwszy z nich polega na składowaniu długości maksymalnej ścieżki. Dla każdego wierzchołka pamiętana jest długość najdłużej ścieżki (począwszy od tego wierzchołka). Dzięki temu w momencie gdy wiadomo, że nie ma możliwości znalezienia w danej gałęzi kandydatów o zadanej długości jest ona pomijana. Zmniejsza to liczbę odwiedzanych wierzchołków i znacznie skraca czas poszukiwania dużych zbiorów elementów - co zostało potwierdzone doświadczalnie.

Kolejny zabieg polega na wykorzystaniu częstotliwości występowania elementów i przechowywaniu jej (oraz jej odwrotności) wraz z elementami oraz reorganizacji drzewa względem tej właśnie wartości (rysunek 3.3). Wówczas szukając wierzchołków reprezentujących zadany zbiór elementów stosowane jest wyszukiwanie liniowe a nie binarne. Dzięki wyżej wspomnianemu posortowaniu szybciej znajdowane są elementy występujące często, ale to do nich jest najwięcej odwołań, więc jest to pożądany efekt. Dodatkowo przeorganizowane drzewo nie jest zbalansowane, a to skutkuje tym, że jest mniej krawędzi do przejścia, co przekłada się bezpośrednio na mniej kosztownych wywołań rekurencyjnych. Zastosowanie tej modyfikacji skutkuje zwiększonym czasem generowania kandydatów, ale pozwala na szybsze obliczenie wsparcia kandydatów. Jako, że druga operacja jest bardziej kosztowna, to jest to zabieg opłacalny.

Wsparcie zbiorów jedno- i dwuelementowych może być obliczane w szybszy sposób niż za pomocą opisywanego drzewa. Dla przyspieszenia wykonania zalecane jest wykorzystanie wektora (dla jednoelementowych) lub tablicy dwuwymiarowej (dla dwuelementowych). Daje to możliwość łatwego dostępu do poszczególnych kandydatów i ich licznika, a także jest mniej wymagające pamięciowo.

Następna modyfikacja polega na zastosowaniu technik haszujących. Sprowadza się to do zmiany reprezentacji krawędzi wychodzących z danego wierzchołka z posortowanej listy na tablicę haszującą. Pozwala to przyspieszyć przeszukiwanie drzewa, a co za tym idzie liczenie wsparcia. Jednakże drzewa haszowe są znacznie bardziej wymagające pamięciowo. Dlatego też powinno stosować się je gdy liczba krawędzi wychodzących przekracza pewną ustaloną wartość. W wyniku tego założenia otrzymane drzewo będzie miało tablice haszowe jedynie do pewnego momentu, potem struktura




Rysunek 3.3: Zmieniona organizacja drzewa prefiksowego dla dwóch 3-elementowych kandydatów $\{A, B, C\}$, $\{A, B, D\}$

wierzchołka pozostanie niezmieniona. Dzięki temu tam gdzie przeszukiwanie było wolne zostanie ono przyspieszone, a tam gdzie - ze względu na małą liczbę krawędzi - było szybkie, nadal będzie szybkie.

Kolejna zmiana zaproponowana w [Bod03] odnosi się do Dzielnego Apriori (ang. *Apriori-Brave*). Jest to heurystyka wykorzystująca śledzenie użycia pamięci. Polega na generowaniu kandydatów o rozmiarach $k+1, k+2, \ldots$ ($k$ - rozmiar największych znalezionych zbiorów częstych) bez sprawdzania wsparcia tych kandydatów do momentu aż wymagana będzie cała dostępna pamięć. Prowadzi to do redukcji odczytów wszystkich transakcji z bazy danych, ale skutkuje także generowaniem fałszywych kandydatów, dla których również obliczane musi być wsparcie. Dlatego też metoda ta nie gwarantuje zysków względem oryginalnego Apriori, ale - jak wykazały testy - heurystyka ta sprawdza się w praktyce.

Ostatnią modyfikacją jest usunięcie z transakcji niepotrzebnych elementów. Po pierwszym odczycie całej bazy wiadomo, które elementy są częste, a które nie. Mając na uwadze własność anty-monotoniczności miary wsparcia ([Mor13]) można ze wszystkich transakcji usunąć elementy nieczęste, gdyż nie wpłynie to na wynik działania algorytmu, a pozwoli zmniejszyć liczbę kroków wykonania.

### 3.2.4 Algorytm Apriori - implementacja Barta Goethalsa [Goe02]

W poniżej sekcji omówiono implementację algorytmu Apriori przedstawioną przez Goethalsa. Podobnie jak w uprzednio opisanych adaptacjach tego algorytmu, w miejsce drzewa haszowego, wykorzystana została struktura drzewa prefiksowego. Głównym elementem, który został poprawiony jest liczba odczytów bazy danych. Jej zredukowanie przekłada się bezpośrednio na czas wykonania algorytmu. Do osiągnięcia takiego efektu wykorzystana została idea górnych ograniczeń (ang. *upper bound*), które zostały opisane poniżej.

#### Górne ograniczenie na liczbę zbiorów kandydatów

Zastosowanie górnych ograniczeń wymaga znalezienia odpowiedzi na pytanie (przy założeniu, że znane są numer obecnej iteracji oraz kolekcja zbiorów częstych na tym poziomie wykonania algorytmu): ile maksymalnie kandydatów może zostać wygenerowanych w pozostałych iteracjach? Taka informacja pozwala na podjęcie dalszych odpowiednich kroków optymalizacyjnych. Goethals [Goe02] zaproponował obliczanie górnej granicy na koniec każdej iteracji Apriori. Do ustalenia podstawowej górnej granicy wykorzystywany jest zbiór $L$ zawierający ze wszystkie możliwe $k$-elementowe podzbiory. Zadanie to nie jest trywialne i zostało wsparte wieloma teoriami, które zostały potem potwierdzone eksperymentalnie. Wyniki tych eksperymentów pokazały, że możliwe jest ustalenie górnych granic w czasie zależnym liniowo od liczby obecnie odkrytych zbiorów częstych. Zredukowało to liczbę odczytów bazy danych przy jednoczesnym zysku pamięciowym związanym ze zmniejszoną liczbą generowanych kandydatów.

### 3.2.5 Common Counting (CC) [WZ03]

Często przeszukiwane zbiory danych służące do generowania reguł asocjacyjnych są bardzo liczne. Niejednokrotnie poszukiwane są reguły odnoszące się tylko do podzbioru tych danych. Wówczas należy sformułować odpowiednie zapytania eksploracyjne filtrujące dane w zadany sposób. Do optymalizacji zbioru takich zapytań służy algorytm Common Counting. W metodzie chodzi o współbieżne wykonanie zbioru zapytań eksploracyjnych algorytmem Apriori z integracją pokrywających się fragmentów bazy danych. Na wejściu algorytm otrzymuje zbiór zapytań eksploracyjnych $DMQ$, dla którego generowany jest zbiór elementarnych predykatów selekcji danych $S$. Początkowo algorytm ustala zbiór wszystkich elementów, czyli takich, które wystąpiły w co najmniej jednej transakcji. W kolejnych krokach generowane są zbiory częste oddzielnie dla każdego z zapytań. Przebiega to w taki sam sposób jak w przypadku standardowego algorytmu Apriori. Z każdym zapytaniem $dmq_i$ powiązane jest drzewo haszowe, w którym przechowywani są kandydaci dla tego zapytania. Warunek zatrzymania algorytmu jest taki jak w standardowym Apriori (brak możliwości wygenerowania kandydatów w kolejnej iteracji), z tą różnicą, że musi być spełniony dla wszystkich zapytań ze zbioru. Zliczenie wystąpień kandydatów jest realizowane dla wszystkich zapytań jednocześnie. Partycje bazy danych są odczytywane sekwencyjnie dla poszczególnych elementarnych predykatów selekcji danych. Powiększeniu ulegają liczniki kandydatów zawartych w analizowanej transakcji dla zapytań posiadających odwołania do danej partycji. Lista kandydatów zawierających się w danej transakcji ustalana jest poprzez testowanie transakcji względem drzew haszowych. Należy tutaj zaznaczyć, że w przypadku gdy kilka zapytań współdzieli dany elementarny predykat selekcji danych, to podczas zliczeń wystąpień kandydatów odczyt właściwej mu partycji jest wykonywany tylko raz. Zatem optymalizowane są odczyty współdzielonych przez zapytania fragmentów bazy danych, przy czym pozostałe kroki algorytmu Apriori pozostają niezmienione i są wykonywane oddzielnie dla każdego zapytania.

### 3.2.6 Common Candidate Tree (CCT) [GW09]

Common Candidate Tree jest podobnym algorytmem do Common Counting. Tak jak w przypadku CC algorytm przetwarza zbiór $DMQ$ i zwraca zbiory częste dla poszczególnych jego elementów, a także korzysta z oryginalnego Apriori i wykorzystuje strukturę drzewa haszowego. Różnica polega na tym, że zwiększony został stopień współbieżności przetwarzania. Uzyskano to dzięki współdzieleniu pamięciowej struktury drzewa składującego kandydatów. Jest to duża zaleta w porównaniu z Common Counting, gdyż - zamiast wielu - tworzone jest jedno drzewo haszowe o niezmiennej strukturze. Poza zachowaniem integracji odczytów współdzielonych możliwa jest integracja testowania czy w danej transakcji zawierają się kandydaci z poszczególnych zapytań. Realizacja tego algorytmu wymagała rozszerzenia struktury kandydatów. W jej wyniku z każdym kandydatem związany został wektor liczników (jeden licznik dla jednego zapytania), a nie pojedynczy licznik. Dodatkowo - dla rozróżnienia zapytań, które wygenerowały danego kandydata - dołączony został wektor flag logicznych przechowujący taką właśnie informację. Po wyłonieniu kandydatów są oni umieszczani w jednym zbiorze. Zbiór ten trafia do wspólnego drzewa haszowego. W tym kroku modyfikowane są również odpowiednie flagi. Samo generowanie kandydatów i selekcja zbiorów częstych nadal realizowane są odrębnie dla poszczególnych zapytań. Zliczany jest natomiast zintegrowany zbiór kandydatów. Podczas tej fazy brani są pod uwagę tylko kandydaci wygenerowani przez zapytania odwołujące się do aktualnie odczytywanej partycji bazy danych i w przypadku gdy kandydat zawiera się w przetwarzanej transakcji, to zwiększa się liczniki kandydatów związane z tymi zapytaniami. Algorytm kończy się gdy nie można wygenerować kandydatów dla kolejnego poziomu drzewa.

Eksperymenty [GW09] pokazały, że Common Candidate Tree jest wydajniejszy i lepiej skalowany od Common Counting.

### 3.2.7 Podsumowanie

Przytoczone algorytmy są przykładami optymalizacji wykonania operacji znajdowania zbiorów częstych. Pierwsze trzy wykorzystują drzewa prefiksowe, a więc strukturę szybszą, prostszą i mniej wymagającą pamięciowo. Poza zmianą struktury stosowane jest sortowanie elementów transakcji według określonego dla danego algorytmu sposobu. Dwa ostatnie algorytmy (Common Counting i Common Candidate Tree) dokonują partycjonowania danych w celu zmniejszenia liczby operacji i sprawdzają się w sytuacji gdy należy wykonać wiele (przynajmniej częściowo nakładających się)

zapytań eksploracyjnych odnoszących się do tego samego zbioru danych. Jednakże korzystają one z oryginalnej wersji algorytmu Apriori [AS94], a co za tym idzie ze struktury drzew haszowych, co skutkuje wolniejszym wykonaniem operacji generowania kandydatów i obliczania wsparcia, a także większych wymagań pamięciowych. W kolejnym rozdziale przedstawiono modyfikacje tych algorytmów. Zmiana polega właśnie na użyciu drzew prefiksowych i jednej z wcześniej opisanych adaptacji Apriori.

# Rozdział 4

# Opracowane algorytmy

## 4.1 Wstęp

Poniższy rozdział poświęcono przedstawieniu idei, opisu i cech zaprojektowanych algorytmów. Oba oparte są na opisanych w poprzednim rozdziale algorytmach przetwarzania zbiorów zapytań eksploracyjnych, tj. Common Counting ([WZ03]) i Common Candidate Tree ([GW09]). Implementacja Apriori wykorzystywana wewnątrz tych algorytmów jest zgodna z rozwiązaniem zaproponowanym przez Borgelta ([Bor03]).

## 4.2 Przygotowanie danych

Zbiór danych, w którym poszukiwane są zbiory częste z użyciem Common Counting lub Common Candidate Tree musi spełniać kilka założeń. Krokiem wstępnym wykonania obu algorytmów jest zatem odpowiednie przygotowanie danych. Po pierwsze elementy każdej transakcji są sortowane leksykograficznie. Następnie niezbędne jest wyznaczenie zbioru elementarnych predykatów selekcji danych dla zbioru zapytań eksploracyjnych $S = \{s_1, s_2, \ldots, s_n\}$. Przykładowo jeżeli relacja $R$ posiada atrybut całkowitoliczbowy $a$ oraz do wykonania są trzy zapytania eksploracyjne $dmq_1 = (R, 0 \leq a < 10, \emptyset, 4\%)$, $dmq_2 = (R, 5 \leq a < 20, \emptyset, 2\%)$, $dmq_3 = (R, 0 \leq a < 5 \; or \; 25 \leq a < 30, \emptyset, 3\%)$, to w tym wypadku zbiór elementarnych predykatów selekcji danych będzie równy $S = \{0 \leq a < 5, 5 \leq a < 10, 10 \leq a < 20, 25 \leq a < 30\}$. Znając ten zbiór można zdefiniować podzielić zbiór danych na rozłączne partycje i minimalizować ich odczyt. Dane odczytane z partycji są przetwarzane przez te zapytania, które się do niej odnoszą. Sprawia to, że części wspólne dla wielu zapytań są odczytywane tylko raz. Wejściem dla obu opracowanych algorytmów jest zbiór $DMQ$.

## 4.3 Struktura wierzchołka

Wierzchołek drzewa prefiksowego zbudowany w podobny sposób do tego zaproponowanego dla CC ([WZ03]) i CCT ([GW09]). Tzn. w przypadku Common Counting wierzchołek zawiera licznik wystąpień oraz listę elementów w zbiorze reprezentowanym przez ten wierzchołek, natomiast w przypadku Common Candidate Tree jest to wektor liczników (jego pola odpowiadają kolejnym zapytaniom eksploracyjnym, tj. $i$-ty element wektora, to licznik wystąpień danego zbioru dla $dmq_i$), wektor flag logicznych informujących, które zapytania odnoszą się do danego wierzchołka (zależność jak w przypadku liczników - $i$-ty element wektora informuje czy wierzchołek zawiera się w $dmq_i$). Dodatkowo w obu przypadkach dołożona została lista etykiet dla każdego wierzchołka. Etykiety odpowiadają wszystkim krawędziom wychodzącym z danego wierzchołka. Pozwala to na łatwiejsze i szybsze poruszanie się po drzewie podczas wykonywania na nim operacji. Struktura wierzchołka wykorzystywana w CCTP zaprezentowana została na rysunku 4.1.

| elements | fromQuery[] | | | counters[] | | | labels |
|---|---|---|---|---|---|---|---|
| {1, 4, 7} | 1 | 1 | 0 | 17 | 6 | 0 | {8, 12} |

Rysunek 4.1: Struktura wierzchołka wykorzystywana w CCTP.

## 4.4 Implementacja algorytmu Apriori

Implementacja Apriori zastosowanego wewnątrz CC i CCT jest inspirowana propozycją Borgelta [Bor03]. Zastosowane zostało drzewo prefiksowe jako struktura przechowująca kandydatów, dla których ustalane jest wsparcie. Drzewo generowane jest od korzenia. Na pierwszym poziomie znajdują się wszystkie możliwe zbiory jednoelementowe. Ich liczność odpowiada liczbie różnych elementów w zbiorze wszystkich elementów występujących w przetwarzanych transakcjach. Następnie wykonywane jest ustalanie wsparcia dla każdego wierzchołka. Kolejny poziom w drzewie generowany jest tylko z wierzchołków zawierających zbiory częste. Dzięki temu znacząco zmniejsza się rozmiar wygenerowanego drzewa. Dla pierwszego poziomu wsparcie jest po prostu sumą wystąpień poszczególnych elementów w transakcjach. Dla pozostałych poziomów Wsparcie wyznaczane jest metodą rekurencyjną liczenia rekurencyjnego (RC) (ang. *recursive counting*). Metoda ta działa dla każdego wierzchołka w następujący sposób: (1) przejdź do dziecka wskazywanego przez krawędź z etykietą odpowiadającą pierwszemu elementowi transakcji i przetwarzaj dla niego pozostałe elementy transakcji w ten sam sposób oraz (2) pomiń pierwszy element transakcji i przetwarzaj dla danego wierzchołka pozostałe elementy. Gdy metoda znajdzie się na poziomie odpowiadającym aktualnie dodanym kandydatom, to zwiększany jest licznik wystąpień w danym wierzchołku i nie następuje dalsze przechodzenie w głąb drzewa. Procedura ta jest sekwencyjnie powtarzana dla każdej transakcji. Dodatkowo, w celu zmniejszenia liczby analizowanych transakcji, sprawdzane jest czy liczba elementów transakcji jest wystarczająca do osiągnięcia rozważanej głębokości drzewa - jeśli nie, to taka transakcja jest pomijana. Algorytm Apriori kończy się w momencie gdy nie udało się wygenerować kandydatów dla kolejnego poziomu drzewa.

## 4.5 Common Counting z wykorzystaniem drzew prefiksowych (CCP)

Pseudokod tej metody przedstawiono na Rysunku. 4.2. Odpowiada on temu co zostało zaproponowane w [WZ03] i opisane w punkcie 3.2.5. Zmiana polega zastosowaniu adaptacji Apriori opisanej w sekcji 4.4. W liniach 1-2 ustalane są wszystkie możliwe elementy i tworzoną one pierwszy poziom drzewa. Następnie ze zbiorów częstych o rozmiarze $k-1$ generowane są $k$-elementowe zbiory kandydatów (dla $k > 1$). Jest to realizowane osobno dla każdego z zapytań $dmq_i$ i przebiega w sposób opisany w 4.4. Ostatnim krokiem jest zebranie odpowiedzi na zapytania eksploracyjne ze zbioru $DMQ$ (linie 12-13).

**Input:** $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$,
where $dmq_i = (\mathcal{R}, a, \Sigma_i, \Phi_i, minfreq_i)$

(1) **for** ($i$=1; $i \leq n$; $i$++) **do**
(2) $\quad \mathcal{C}_1^i$ = all possible 1-itemsets
(3) **for** ($k$=1; $\mathcal{C}_k^1 \cup \mathcal{C}_k^2 \cup .. \cup \mathcal{C}_k^n \neq \emptyset$; $k$++) **do begin**
(4) $\quad$ **for each** $s_j \in S$ **do begin**
(5) $\quad\quad \mathcal{CC} = \{\mathcal{C}_k^i : \sigma_{s_j}\mathcal{R} \subseteq \sigma_{\Sigma_i}\mathcal{R}\}$
(6) $\quad\quad$ **if** $\mathcal{CC} \neq \emptyset$ **then** $count(\mathcal{CC}, \sigma_{s_j}\mathcal{R})$ **end**
(7) $\quad$ **for** ($i$=1; $i \leq n$; $i$++) **do begin**
(8) $\quad\quad minsup_i = \lceil minfreq_i * |\sigma_{\Sigma_i}\mathcal{R}| \rceil$
(9) $\quad\quad \mathcal{F}_k^i = \{C \in \mathcal{C}_k^i : C.counter \geq minsup_i\}$
(10) $\quad\quad \mathcal{C}_{k+1}^i = apriori\_gen(\mathcal{F}_k^i)$ **end**
(11) **end**
(12) **for** ($i$=1; $i \leq n$; $i$++) **do**
(13) $\quad Answer_i = \sigma_{\Phi_i} \bigcup_k \mathcal{F}_k^i$

Rysunek 4.2: Pseudokod metody Common Counting

## 4.6 Common Candidate Tree z wykorzystaniem drzew prefiksowych (CCTP)

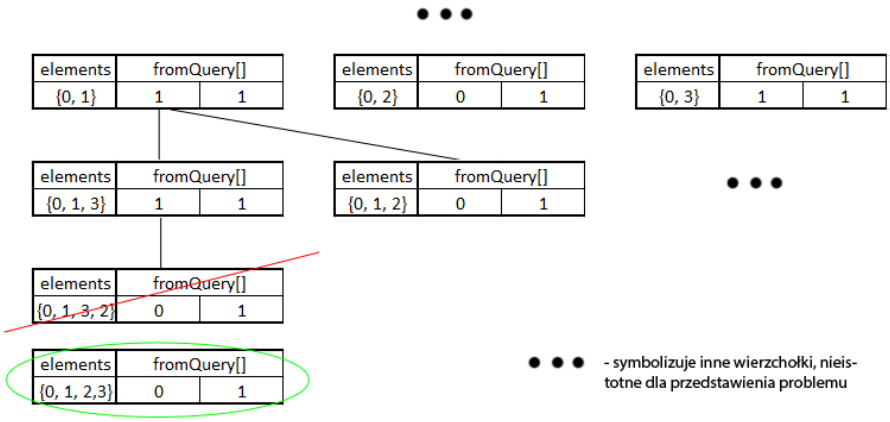
Pseudokod tej metody przedstawiono na Rysunku. 4.3. Pokrywa się on z tym co zostało zaproponowane w [GW09] i opisane w punkcie 3.2.6. Zmiana polega zastosowaniu adaptacji Apriori opisanej w sekcji 4.4, a co za tym idzie zmianie wykorzystywanej struktury z drzewa haszowego

na prefiksowe, innym sposobie generowania kandydatów (linie 7-9) oraz wyznaczania ich wsparcia (linie 4-6) (zgodnie z tym co opisano w 4.4). Podobnie jak w przypadku CCP ostatnim krokiem jest zebranie odpowiedzi na zapytania eksploracyjne ze zbioru $DMQ$ (linie 12-13). Jednakże wykorzystywanie drzew prefiksowych wymaga utrzymania elementów wewnątrz wierzchołków drzewa w kolejności rosnącej. Ma to odzwierciedlenie w kroku generowania kandydatów, a dokładniej łączenia zbiorów kandydatów wygenerowanych przez wszystkie zapytania. Ostatecznie zdecydowano się na sprawdzanie kolejności elementów wewnątrz wierzchołka-kandydata podczas dodawania go do drzewa i ewentualne sortowanie w przypadku gdy jest to konieczne.

**Input:** $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$,
where $dmq_i = (\mathcal{R}, a, \Sigma_i, \Phi_i, minfreq_i)$
(1) $\mathcal{C}_1 =$ all possible 1-itemsets
(2) **for** ($k$=1; $\mathcal{C}_k \neq \emptyset$; $k$++) **do begin**
(3) **for each** $s_j \in S$ **do begin**
(4) $\mathcal{CC} = \{C \in \mathcal{C}_k : \exists i\ C.fromQuery[i] = true \wedge \sigma_{s_j}\mathcal{R} \subseteq \sigma_{\Sigma_i}\mathcal{R}\}$
(5) **if** $\mathcal{CC} \neq \emptyset$ **then** $count(\mathcal{CC}, \sigma_{s_j}\mathcal{R})$ **end**
(6) **for** ($i$=1; $i \leq n$; $i$++) **do begin**
(7) $minsup_i = \lceil minfreq_i * |\sigma_{\Sigma_i}\mathcal{R}| \rceil$
(8) $\mathcal{F}_k^i = \{C \in \mathcal{C}_k : C.counters[i] \geq minsup_i\}$
(9) $\mathcal{C}_{k+1}^i = apriori\_gen(\mathcal{F}_k^i)$ **end**
(10) $\mathcal{C}_{k+1} = \mathcal{C}_{k+1}^1 \cup \mathcal{C}_{k+1}^2 \cup .. \cup \mathcal{C}_{k+1}^n$
(11) **end**
(12) **for** ($i$=1; $i \leq n$; $i$++) **do**
(13) $Answer_i = \sigma_{\Phi_i} \bigcup_k \mathcal{F}_k^i$

Rysunek 4.3: Pseudokod metody Common Candidate Tree

Przykład uzasadniający dodatkowy krok algorytmu został zaprezentowany na rysunku 4.4. Jak widać w wyniku generowania kandydatów dla $dmq_1$ powstał wierzchołek $\{0, 1, 3\}$, następnie dla $dmq_2$ utworzono $\{0, 1, 2\}$ oraz ustawiono odpowiednią flagę $fromQuery_2$ w wektorze flag wierzchołka $\{0, 1, 3\}$. W wyniku generowania kolejnych kandydatów bez sortowania wartości zapytanie $dmq_2$ wygenerowałoby wierzchołek $\{0, 1, 3, 2\}$, co byłoby niezgodne z założeniami na temat drzewa prefiksowego. Dlatego też dla każdego dodawania sprawdzane są ostatnie dwa elementy i jeśli jest taka konieczność, to są one zamieniane miejscami. Pozostała część listy elementów jest wspólnym prefiksem i nie musi być weryfikowana.




Rysunek 4.4: Przykład uzasadniający konieczność sprawdzania elementów podczas dodawania wierzchołka.

# Rozdział 5

# Wyniki eksperymentów

## 5.1 Wstęp

W poniższym rozdziale opisano przebieg przeprowadzonych eksperymentów oraz przedstawiono uzyskane wyniki wraz z ich interpretacją. Algorytmy testowane były na tych samych zbiorach danych. Tworzenie tych zbiorów odbywało się z wykorzystaniem generatora przyjmującego jako parametry liczbę transakcji (np. 10 tys., 100 tys.), średnią liczbę elementów w transakcji (np. 6), liczbę wzorców do odkrycia (np. 500), średni rozmiar wzorców częstych do odkrycia (np. 3), liczbę różnych elementów występujących w transakcjach (np. 1000, 10000) oraz nazwę pliku wyjściowego. Wygenerowany plik jest importowany do bazy danych, do której odwołuje się aplikacja podczas wykonywania algorytmów. Dane w bazie składowane są w jednej tabeli w postaci par $(idtransakcji, idelementu)$. Dlatego też po odczycie tej tabeli składane są transakcje wykorzystywane w dalszym przetwarzaniu. Zbiór zapytań $DMQ$ generowany był przed rozpoczęciem przetwarzania. Liczba transakcji obejmująca każde zapytanie $dmq_i$ wynosi $|T|/|DMQ|$, gdzie $|T|$ - liczba wszystkich transakcji, $|DMQ|$ - liczba wszystkich zapytań eksploracyjnych.

## 5.2 Opis infrastruktury

Algorytmy napisane zostały w języku Java, z wykorzystaniem narzędzia Maven oraz środowiska programistycznego Eclipse. Dane testowe generowane były za pomocą generatora GEN ([AMS+96]), a następnie wczytywane do bazy PostgreSQL, z której korzystała aplikacja. Testy przeprowadzone zostały na komputerze HP Envy 14 Notebook PC, z procesorem Intel Core i5-2410M 2x2.30GHz oraz 8GB pamięci RAM, pracującym pod kontrola systemu operacyjnego Microsoft Windows 7.

## 5.3 Wyniki

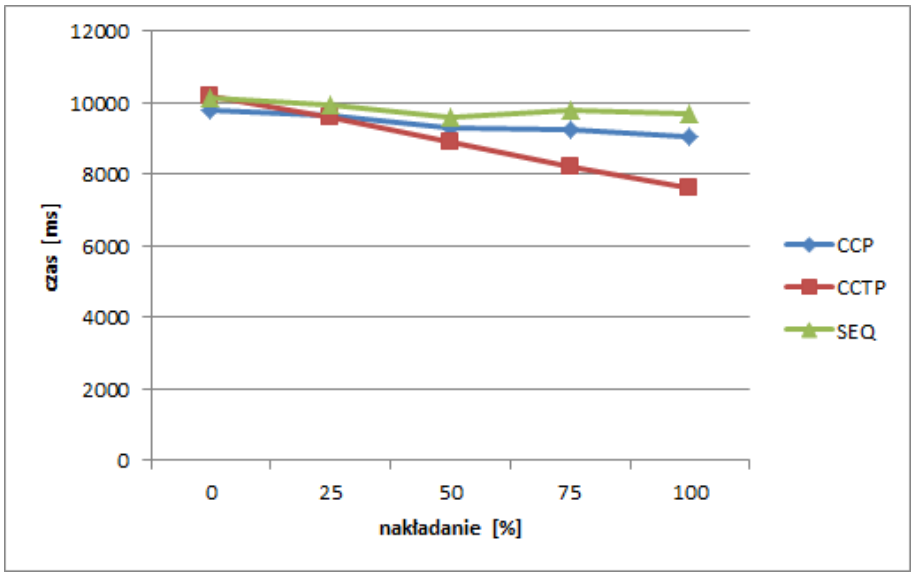
Poniżej zaprezentowano wyniki eksperymentów. Wszystkie uwzględnione czasy są średnią 4 dokonanych pomiarów dla dokładnie tych samych parametrów. Badania przeprowadzono dla dwóch zapytań eksploracyjnych z różnym poziomem nakładania się danych. Oba zapytania wykorzystywały ten sam próg wsparcia, mimo że nie jest to wymaganiem algorytmu. Jest to podyktowane chęcią możliwości lepszego zaobserwowania różnic w działaniu algorytmów niezależnie od wykorzystywanych progów dla poszczególnych zapytań. Algorytmy uwzględniane w wynikach to Common Counting z wykorzystaniem drzew prefiksowych (CCP), Common Candidate z wykorzystaniem drzew prefiksowych (CCTP) oraz sekwencyjnie wykonywana adaptacja algorytmu Apriori (SEQ) zgodna z tą zastosowaną wewnątrz CCP i CCTP.

Zbiór danych 1. Parametry generatora dla drugiego zbioru danych przedstawia tabela 5.1.

Na wykresie 5.1 zestawione czasy trwania poszczególnych algorytmów wykonywanych na pierwszym zbiorze danych z różnymi stopniami nakładania się. Przyjęta wartość *minsup* dla wszystkich zapytań wynosiła 2%. Jak widać na wykresie w przypadku braku nakładania się zapytań uzyskane czasy były do siebie bardzo zbliżone. Wraz ze wzrostem stopnia nakładania się uwidaczniały się różnice w czasach wykonania algorytmów. Wykonanie sekwencyjne cały czas utrzymywało się na niemalże takim samym poziomie. Jest to związane z podobną liczbą operacji wykonywaną w trakcie działania algorytmu. W przypadku CCP i CCTP wraz ze wzrostem nakładania zwiększała się współbieżność. W przypadku CCTP skrócenie czasu następuje szybciej niż ma to miejsce

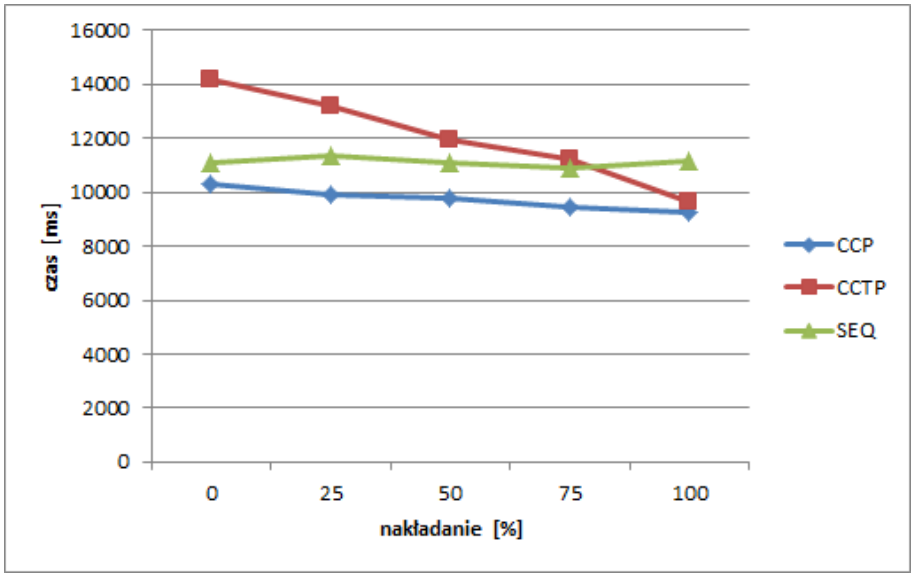
| | |
|---|---|
| liczba transakcji | 100000 |
| średnia liczba elementów w transakcji | 8 |
| liczba różnych elementów | 1000 |
| liczba wzorców (zbiorów częstych) | 1500 |
| średnia liczba elementów we wzorcu | 4 |

Tablica 5.1: Parametry generatora dla pierwszego testowego zbioru transakcji.




Rysunek 5.1: Wykres dla pierwszego zbioru danych dla dwóch zapytań *dmq* z różnym stopniem nakładania się i $minsup = 2\%$

dla CCP. Jednakże dla CCP także zaobserwowany został stały spadek czasu wykonania. Można powiedzieć, że dla tego przypadku algorytm CCTP okazał się zdecydowanie najlepszy, a także zastosowanie CCP zamiast SEQ skutkowało krótszym czasem udzielenie odpowiedzi na zadane zapytania eksploracyjne.




Rysunek 5.2: Wykres dla pierwszego zbioru danych dla dwóch zapytań *dmq* z różnym stopniem nakładania się i $minsup = 1\%$

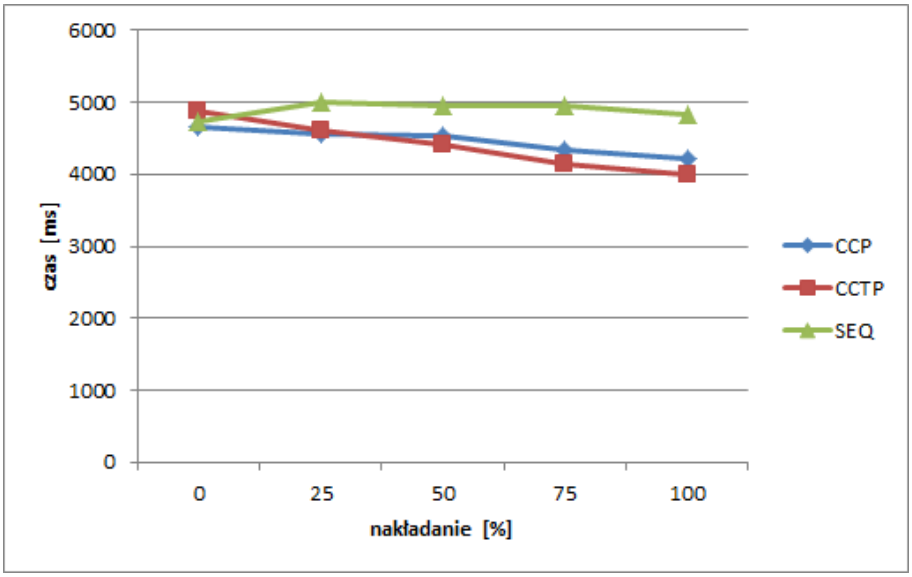
Drugi rozważany przypadek na pierwszym zbiorze danych został przedstawiony na wykresie 5.2.

Przyjęto *minsup* ma poziomie 1%. Mniejsza wartość progu minimalnego wsparcia przekłada się na większą liczbę zbiorów częstych odkrywanych w trakcie wykonań algorytmów, stąd też wydłużenie czasów wykonania. W przypadku wykonania sekwencyjnego po raz kolejny widoczny jest brak wpływu nakładania się zapytań eksploracyjnych. Podobnie jak w pierwszym analizowanym przykładzie zachowuje się algorytm Common Counting. Widoczne jest skracanie czasu wykonania wraz ze wzrostem nakładania się zapytań. Jest to zależność liniowa i mimo, że przyspieszenie nie jest bardzo znaczące, to po raz kolejny CCP działa szybciej niż SEQ. Ciekawa sytuacja została zaobserwowana dla Common Candidate Tree. Znowu widoczny jest stały, liniowy spadek czasu wykonania, ale tym razem algorytm dla większości rozważanych przypadków nakładania się *dmq* działa wolniej. Jest to zapewne spowodowane koniecznością sprawdzania kolejności (i sortowania jeśli jest to niezbędne) elementów w danym wierzchołku (problem ten opisany został w 4.6). Nie było to widoczne dla $minsup = 2\%$ ze względu na mniejszą liczbę zbiorów częstych na każdym poziomie drzewa, co ma bezpośrednie przełożenie na liczbę kandydatów na kolejnym poziomie, a więc liczbę dodawanych do drzewa wierzchołków. Mimo to dla znaczącego nakładania się obserwowany jest zysk względem SEQ.

Zbiór danych 2. Parametry generatora dla drugiego zbioru danych przedstawia tabela 5.2.

| | |
|---|---|
| liczba transakcji | 50000 |
| średnia liczba elementów w transakcji | 8 |
| liczba różnych elementów | 1000 |
| liczba wzorców (zbiorów częstych) | 1000 |
| średnia liczba elementów we wzorcu | 4 |

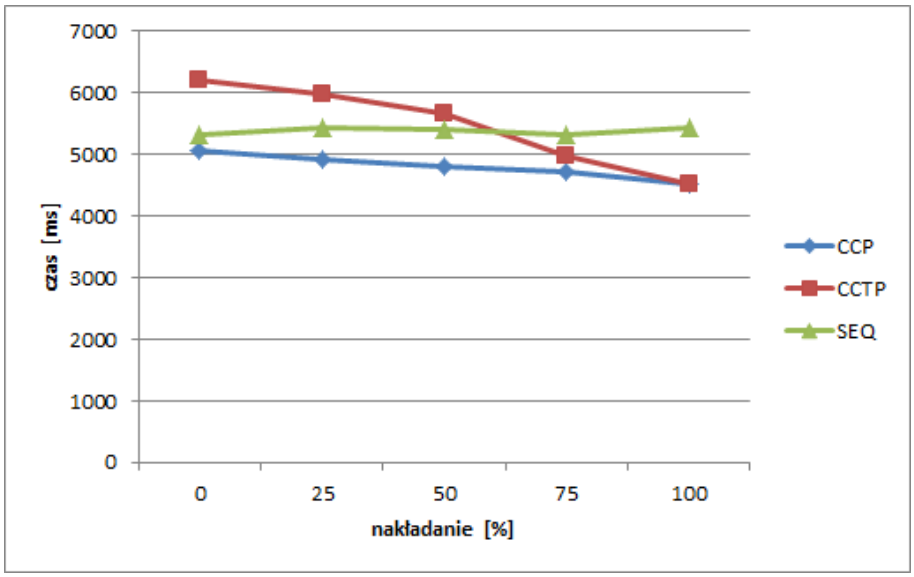
Tablica 5.2: Parametry generatora dla drugiego testowego zbioru transakcji.




Rysunek 5.3: Wykres dla drugiego zbioru danych dla dwóch zapytań *dmq* z różnym stopnieniem nakładania się i $minsup = 2\%$

Wykres 5.3 obrazuje czasy wykonania algorytmów dla mniejszego z wygenerowanych zbiorów danych, na których przeprowadzane były testy. Podobnie jak miało to miejsce w uprzednio opisanych przypadkach czasy wykonania SEQ są niemalże liniowe. Zaobserwowane drobne różnice wynikają prawdopodobnie z tego, że zapytania odnoszą się do różnych podzbiorów zbioru danych, a także wpływem chwilowych różnic w działaniu środowiska testowego. Patrząc na wyniki uzyskane przez CCP o CCTP okazuje się, że są one bardzo zbliżone. Po raz kolejny czas wykonania maleje szybciej dla Common Candidate Tree niż dla Common Counting. Jednakże dla przyjętych dla tej obserwacji parametrów ciężko jednoznacznie wskazać algorytm lepszy, aczkolwiek oba są szybsze od algorytmu sekwencyjnego.

Na wykresie 5.4 zaprezentowano wyniki uzyskane dla drugiego zbioru danych przy założeniu, że próg minimalnego wsparcia wynosi 1%. Potwierdza on niezależność wykonania sekwencyjnego




Rysunek 5.4: Wykres dla drugiego zbioru danych dla dwóch zapytań *dmq* z różnym stopnieniem nakładania się i $minsup = 1\%$

względem stopnia nakładania się zapytań $dmq_i$. Obserwowane rezultaty dla CCP i CCTP przypominają te uzyskane dla pierwszego zbioru danych i $minsup = 1\%$. Widoczny jest dłuższy czas wykonania CCTP, w stosunku do SEQ i CCP, dla zapytań nie nakładających się lub małego stopnia nakładania. Powodem jest tak jak poprzednio konieczność wykonywania większej liczby sprawdzeń kolejności elementów w wierzchołku (i ewentualnych sortowań). Nie zmienia to faktu, że widoczny jest stały spadek czasu wykonania wraz ze wzrostem współbieżności. Z kolei Common Counting potwierdza swoją przewagę nad wykonaniem sekwencyjnym oraz fakt, że zwiększanie stopnia nakładania się zapytań działa na rzecz skrócenia (proporcjonalnie do zwiększania się nakładania) przetwarzania $DMQ$ tym algorytmem.

## 5.4 Porównanie CCP i CCTP z CC i CCT

Powołując się na wyniki uzyskane w [WZ03] oraz [GW09] można stwierdzić, że zastosowanie Common Counting wykorzystującego struktury prefiksowe daje podobny efekt jak w przypadku drzew haszowych. Efektem tym jest stały wzrost szybkości przetwarzania proporcjonalny do stopnia nakładania się danych. Nieco inaczej rzecz ma się dla Common Candidate Tree. Po zmianie drzewa haszowego na prefiksowe dla algorytmu nadal obserwowany jest liniowy (szybszy niż dla CC) spadek czasu wykonania. Niestety w przypadkach gdy na każdym poziomie drzewa generowanych jest dużo kandydatów (np. stosunkowo małe *minsup*) i nie występuje duże nakładanie się zapytań, to algorytm ten traci swoją przewagę. Jest to przewaga CCT ([GW09]), gdyż dla drzewa haszowego kolejność dodawania wierzchołków nie ma wpływu na przetwarzanie.

# Rozdział 6

# Wnioski i uwagi

Efektem pracy są implementacje dwóch algorytmów wykonania zbioru zapytań odkrywających zbiory częste - które dotychczas implementowane były na drzewie haszowym - z wykorzystaniem drzew prefiksowych. Opisana została problematyka zagadnienia oraz dokonano przeglądu istniejących rozwiązań związanych z tematem pracy. Przeprowadzone zostały również testy tych algorytmów oraz przedstawiona została analiza uzyskanych wyników.

Przeprowadzone eksperymenty pokazują, że możliwa jest adaptacja algorytmów przetwarzania zbiorów zapytań eksploracyjnych (Common Counting [WZ03] i Common Candidate Tree [GW09]) opartych na oryginalnym Apriori ([AS94]) do rozwiązań korzystających z szybszych wersji algorytmu Apriori wykorzystujących drzewa prefiksowe. Dla CCT nie było to zadanie tak trywialne jak w wypadku CC, ale jednak po rozwiązaniu problemów implementacyjnych osiągnięto efekt podobny jak raportowany wcześniej dla drzew haszowych. Utrudnieniem w przypadku CCTP okazał się sposób analizy i przeglądania drzewa prefiksowego przez użyty algorytm Apriori. Z testów wynika, że stosowanie CCP jest wskazane dla przetwarzania zbioru zapytań eksploracyjnych niezależnie stopnia nakładania się danych czy też innych parametrów przetwarzania. W najgorszym przypadku uzyskany czas będzie porównywalny z czasem algorytmu sekwencyjnego. Należy jedynie pamiętać o ograniczeniach pamięciowych, które wymagają od CC przechowywania wielu drzew w pamięci, podczas gdy dla (SEQ) analizowane jest tyko jedno drzewo w danym momencie. Natomiast wykorzystanie CCTP przynosi najwięcej korzyści w momencie gdy partycje danych, do których odwołują się zapytania mają wysoki współczynnik nakładania się, w przeciwnym wypadku istnieje ryzyko wydłużenia czasu wykonania.

Adaptacje algorytmu Apriori oraz algorytmy przetwarzające zbiory zapytań eksploracyjnych nadal mogą być ulepszane i optymalizowane. Jednym z możliwych rozwiązań jest próba wykorzystania Common Counting w taki sposób, że na podstawie zbioru elementarnych predykatów selekcji danych dla zbioru zapytań eksploracyjnych $S$ generowane są rozłączne zapytania $dmq_i$, dla których algorytm wykonywany jest w ten sam sposób jaki opisano w 4.3. Ostatnim krokiem byłaby wówczas integracja znalezionych zbiorów częstych w celu uzyskania odpowiedzi na oryginalne zapytania ze zbioru $DMQ$. Takie podejście zmniejszyłoby rozmiary drzew przechowujących kandydatów podczas przetwarzania. Należałoby sprawdzić czy zysk ten jest większy niż koszt operacji generowania zbioru rozłącznych zapytań $DMQ$.

Innym zagadnieniem jest problem optymalizacji generowania kandydatów dla CCTP. Sortowanie wartości podczas dodawania jest czasochłonne, mimo że porównywane są jedynie dwie wartości zbioru elementów, a nie cała lista wartości. Mimo prób zastosowania innych struktur (m.in. zbioru czy też drzew haszowych) podczas tej operacji nie udało się uzyskać lepszego wyniku niż dla ostatecznie zastosowanego rozwiązania.

Pewne jest, że badany obszar pozostawia bardzo dużo możliwości i perspektyw do prowadzenia dalszych prac nad optymalizacją wykonania zarówno zbioru zapytań eksploracyjnych, jak i samego algorytmu Apriori.

# Dodatek A

# Zawartość płyty DVD

Jako dodatek do tego dokumentu dołączona jest płyta DVD. Zawiera ona materiały związane z prezentowanym tematem w formacie elektronicznym dla osób, które mogą być zainteresowane kontynuacją prac w tym temacie.

Płyta DVD zawiera następujące elementy:

1. Elektroniczna wersja pracy magisterskiej w formacie PDF jak i w formacie do edycji (TEX).
2. Aplikacja konsolowa w języku Java z zaimplementowanymi algorytmami.
3. Generator danych testowych.

# Literatura


[AMS+96] Rakesh Agrawal, Manish Mehta, John Shafer, Ramakrishnan Srikant, Andreas Arning, and Toni Bollinger. The quest data mining system. In *In Proc. of the 2nd Int'l Conference on Knowledge Discovery in Databases and Data Mining*, pages 244–249, 1996.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 487–499, 1994.

[BMUT97] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. *SIGMOD Rec.*, 26(2):255–264, June 1997.

[Bod03] Ferenc Bodon. A fast APRIORI implementation. In *FIMI '03, Frequent Itemset Mining Implementations, Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations, 19 December 2003, Melbourne, Florida, USA*, 2003.

[Bor03] Christian Borgelt. Efficient implementations of apriori and eclat. In *Proc. 1st IEEE ICDM Workshop on Frequent Item Set Mining Implementations (FIMI 2003, Melbourne, FL). CEUR Workshop Proceedings 90*, page 90, 2003.

[Goe02] Bart Goethals. *Efficient Frequent Pattern Mining*. PhD thesis, transnationale Universiteit Limburg, 2002.

[GW09] Przemyslaw Grudzinski and Marek Wojciechowski. Integration of candidate hash trees in concurrent processing of frequent itemset queries using apriori. *Control and Cybernetics*, 38(1):47–64, 2009.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29(2):1–12, May 2000.

[Mor13] Tadeusz Morzy. *Eksploracja danych*. Wydawnictwo Naukowe PWN, Warszawa, 2013.

[WZ03] Marek Wojciechowski and Maciej Zakrzewicz. Evaluation of common counting method for concurrent data mining queries. In *Advances in Databases and Information Systems, 7th East European Conference, ADBIS 2003, Dresden, Germany, September 3-6, 2003, Proceedings*, pages 76–87, 2003.



Instytut Informatyki, Wydział Informatyki
Politechnika Poznańska

Skład przy użyciu systemu LaTeX.

BibTeX:

```
@mastersthesis{ key,
    author = "Szymon Dolata",
    title = "{Integracja drzew prefiksowych w przetwarzaniu zbiorów zapytań eksploracyjnych
algorytmem Apriori}",
    school = "Poznan University of Technology",
    address = "Pozna{\'n}, Poland",
    year = "2014",
}
```