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 sprzedanych 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. Istnieją wskaźniki służące do oceny tychże reguł. Zostały one omówione bardziej szczegółowo w rozdziale 2.

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. Dlatego potrzebne są wydajne algorytmy, które ograniczają przestrzeń wyszukiwania i sprawdzają jedynie podzbiór wszystkich reguł. Jednym z takich algorytmów jest Apriori opracowany przez [**?**].

Podstawowy algorytm Apriori trzyma kandydatów w drzewie haszowym. W ostatnich latach zaproponowane zostały metody Common Counting ([**?**]) oraz Common Candidate Tree ([**?**]). Są one wynikiem badań 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).
W praktyce jednak lepsze okazały się implementacje Apriori gdzie drzewo haszowe zastąpiono znacznie prostszą strukturą drzewa prefiksowego. Powstało kilka rozwiązań wykorzystujących tę strukturę: Borgelt, Bodon, Goethals. Jednak do tej pory nie została zaimplementowana modyfikacja Common Counting i Common Candidate Tree dla 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.

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 działanie 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 omówiono podstawowe pojęcia i definicje wykorzystywane w pracy.

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 transakcją (ang. *itemset*). Natomiast zbiór elementów o mocy k, to taki zbiór, który posiada dokładnie k elementów (ang. *k-itemset*).

Transakcja jest zatem 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), to nie ma możliwości żeby zamodelować wszystkie możliwe transakcje (koszyki). Wynika to z tego, że elementy w zbiorze nie mogą się powtarzać. Problem ten znalazł kilka rozwiązań. Należy do nich zamodelowanie wszystkich transakcji jako multizbioru (uogólnienie pojęcia zbioru, w którym w odróżnieniu od klasycznych zbiorów jeden element może występować wiele razy) albo jako wektora (elementy na różnych pozycjach mogą być takie same, ale wyróżnia je położenie). Innym - choć podobnym do wspomnianego wyżej zastosowania wektora - rozwiązaniem jest rozszerzenie każdej transakcji o unikalny identyfikator. Kolejną możliwością jest wykorzystanie zbioru unikalnych transakcji, z tą różnicą, że do każdej transakcji przypisany jest licznik mający za zadanie zliczanie wystąpień.

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. 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 transakcji

Jeśli T oznacza jedną z transakcji w zbiorze wszystkich transakcji D, to (bezwzględne) wsparcie tej transakcji jest równe U - liczbie wystąpień T w zbiorze D. Wsparcie względne jest to z kolei procent (lub ułamek) transakcji w zbiorze D, które zawierają T. Obliczamy ze wzoru

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

. 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 poszukiwaniu zbiorów częstych interesujące są tylko te reguły, dla których $sup(T) \geq minsup$, gdzie $sup(T)$, to przyjęty w pracy sposób zapisu wsparcia transakcji T w rozważanym zbiorze transakcji D.

2.2.4 Ufność reguły asocjacyjnej

Ufność reguły asocjacyjnej jest miarą jakości danej reguły. Miara ta została przedstawiona przez autorów algorytmu Apriori [**?**]. Dla reguły asocjacyjnej postaci $R = "X \rightarrow Y"$ (gdzie X i Y to zbiory elementów) 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 $A \cup B \rightarrow C$ odpowiada wsparciu zbioru $S = \{A, B, C\}$ ([**?**]).Miara ta informuje o tym jak często dana reguła jest prawidłowa. Nieco odmienna definicja została przedstawiona i wykorzystana w [**?**]. Różnica polega na tym, że wsparcie wyrażone jest jako liczba przypadków, w których reguła jest stosowalna. Zatem dla powyżej postaci byłoby to $S = \{A, B\}$, nawet jeśli reguła może okazać się fałszywa. 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 mający dokładnie takie samo wsparcie.

2.2.8 Zbiór maksymalny

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

2.2.9 Zapytanie eksploracyjne

Zapytanie eksploracyjne jest uporządkowaną piątkę $dmq = (R, a, \Sigma, \Phi, minsup)$, gdzie R - relacja bazy danych, a - atrybut relacji R, Σ - wyrażenie warunkowe dotyczące atrybutów R nazywane

predykatem selekcji danych, Φ - 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 Φ i posiadają $wsparcie \geq minsup$ (π - relacyjna operacja projekcji, σ - 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 ([**?**]), 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 ([**?**]), Ferenca Bodona ([**?**]) oraz Barta Goethalsa ([**?**]). 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 ([**?**]) i Common Candidate Tree ([**?**]). 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.

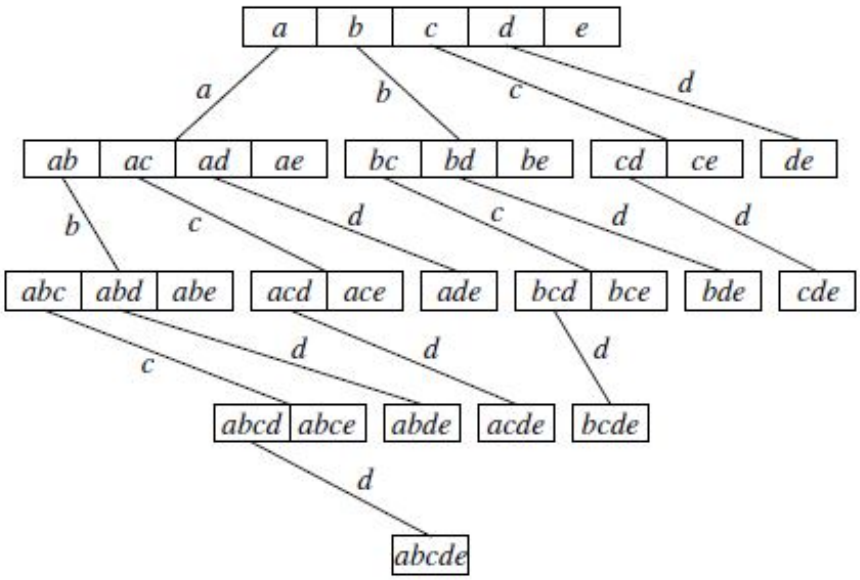
3.2 Przegląd istniejących rozwiązań

3.2.1 Algorytm Apriori [?]

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 ([**?**]). 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 [?]

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 (puste zbiory nie zostały zaprezentowane).

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 takiego samego 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.

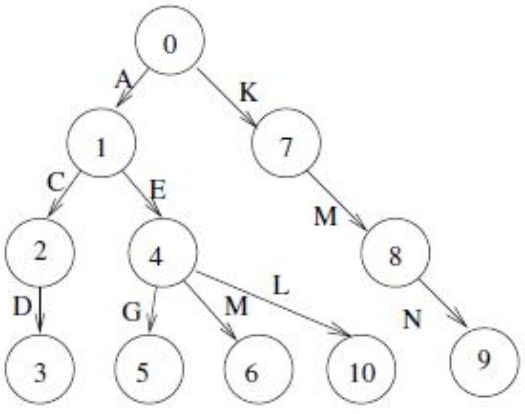
Algorytm działa szybciej niż oryginalny Apriori, a wybór struktury wierzchołka i wybranych optymalizacji zależy głównie od badanego zbioru danych.

3.2.3 Algorytm Apriori - implementacja Ferenca Bodona [?]

W tej sekcji opisana została modyfikacja algorytmu Apriori zaproponowana przez Ferenca Bodona. Podobnie jak w przypadku wyżej opisanej implementacji Borgelta ([?]) 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 wsparcie 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 czy 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 a 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 liczą 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 ([**?**]) druga metoda okazała się średnio dwukrotnie szybsza i to ona została wykorzystana do dalszych rozważań.

Modyfikacje optymalizujące czas wykonania

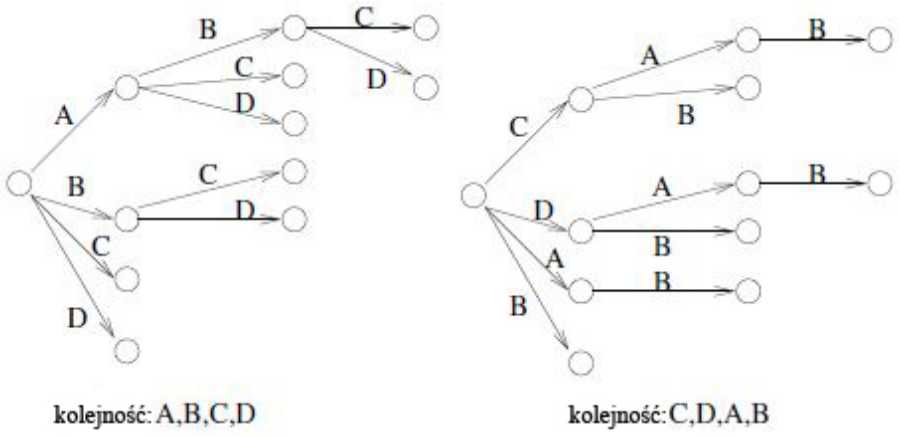
W ([**?**]) zaproponowano kilka zabiegów optymalizujących czas wykonania algorytmu. Skupiają się one przede wszystkim na skróceniu czasu potrzebnego na obliczanie wsparcia kandydatów. Z reguły wymagają więcej pamięci, aby składować dodatkowe informacje, ale mimo to w ostatecznym rozrachunku są opłacane. 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 ilość krawędzi - było szybkie, nadal będzie szybkie.

Kolejna zmiana zaproponowana w [**?**] 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 ([**?**]) 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 [?]

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 [**?**] 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) [?]

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 równoległe wykonanie zbioru zapytań eksploracyjnych algorytmem Apriori z integracją porywających się fragmentów bazy danych. Na wejściu algorytm otrzymuje zbiór elementarnych predykatów selekcji danych dla zbioru zapytań eksploracyjnych DMQ. 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 powiązane jest drzewo haszowe, w którym przechowywani są kandydaci. 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) [?]

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 [?] 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 [**?**], 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 ([**?**]) i Common Candidate Tree ([**?**]). Implementacja Apriori wykorzystywana wewnątrz tych algorytmów jest zgodna z rozwiązaniem zaproponowanym przez Borgelta ([**?**]).

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 $DMQ = \{dmq_1, dmq_2, \ldots, dmq_n\}$. 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 DMQ zawierające rozłączne zapytania eksploracyjne. Po wykonaniu wszystkich zapytań z DMQ należy w odpowiedni sposób połączyć zebrane informacje i zwrócić odpowiedzi na pierwotnie sformułowane zapytania. Tak przygotowany zbiór DMQ jest wejściem dla obu opracowanych algorytmów.

4.2.1 Implementacja algorytmu Apriori

Implementacja Apriori zastosowanego wewnątrz CC i CCT jest inspirowana propozycja Borgelta [**?**]. 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.

4.3 Common Counting z wykorzystaniem drzew prefiksowych (CCP)

4.4 Common Candidate Tree z wykorzystaniem drzew prefiksowych (CCTP)

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 GEN ([**?**]). Jako parametry przyjmuje on 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 PostgreSQL, 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.

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 ([**?**]), 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 S2

Rozdział 6

Wnioski i uwagi

Whole conclusion for one page.

Dodatek A

Zawartość płyty DVD

As an addition to this document, the DVD is attached. It provides some materials connected with the presented subject in electronic form for potential users or people, who would want to continue works on this topic.

The DVD content consists of several items:

1. Item 1
2. Item 2
3. Item 3

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",
}
```