Wspomaganie tworzenia analizatorów stron wyników z internetowych systemów wyszukujących

Paweł Kowalik

Wspomaganie tworzenia analizatorów stron wyników z internetowych systemów wyszukujących

Paweł Kowalik

Spis treści

Spis ilustracji

Spis tabel

Spis równań

1

Wstęp

W ciągu ostatnich kilkunastu lat nastąpił nagły i bardzo szybki wzrost dostępności informacji. Postęp ten zawdzięczamy bardzo dynamicznemu rozwojowi sieci Internet, a w szczególności tzw. ogólnoświatowej pajęczyny World Wide Web. Internet jest jednak bardzo „niezorganizowanym, nieustrukturalizowanym i zdecentralizowanym miejscem" [Lawrence & Giles, 98] i to, co miało być początkowo jego zaletą, stało się zmorą. Ilość dostępnych stron wzrasta w zastraszającym tempie. W 2001 roku serwis Google [Google, 03] podawał 1,35 miliarda jako liczbę zaindeksowanych stron, dziś liczba ta wynosi ponad 3 miliardy. W tej sytuacji problemem nie jest sama dostępność informacji, lecz jej skuteczne wyszukiwanie.

Podobnie, jak w latach siedemdziesiątych dwudziestego wieku został zdefiniowany **Information Retrieval Problem**: *mając zestaw dokumentów i zapytanie, określ podzbiór dokumentów odpowiadających zapytaniu*, dla zastosowań w sieci Internet został sformułowany **Web Search Problem** [Selberg, 99]: *jak użytkownik może znaleźć wszystkie informacje dostępne w sieci odpowiadające pewnemu tematowi*. Odpowiedzią na ten problem stały się licznie obecne wyszukiwarki internetowe obejmujące swoimi indeksami bardzo dużą część dostępnych zasobów sieci. Wymienić tutaj należy przede wszystkim te największe jak Google [Google, 03], alltheweb [AllTheWeb, 03], Yahoo [Yahoo, 03], czy Altavista [Altavista, 03].

Paradygmat zapytanie-lista wyników

Ogromna większość serwisów wyszukiwawczych opiera się na paradygmacie *zapytanie-lista wyników*. Użytkownik podaje listę słów, a system odpowiada *uporządkowaną listą dokumentów*, które pasują do zapytania. Uporządkowanie dokumentów ma kluczową rolę z punktu widzenia przydatności otrzymanych wyników. Różne serwisy stosują różne podejścia, jednak wydaje się, iż żadne podejście nie może zapewnić idealnego dostosowania pierwszych wyników na liście do potrzeb użytkownika. Problemem jest fakt, iż użytkownicy nie potrafią sformułować dostatecznie precyzyjnych zapytań i podają zazwyczaj krótkie listy 2-3 słów, które najczęściej dają w wyniku olbrzymią liczbę odnalezionych dokumentów. W efekcie nie podejmują próby uściślenia zapytania, lecz rezygnują przytłoczeni ilością informacji.

Jasna stała się więc potrzeba dalszego przetwarzania listy wyników otrzymanych z serwisu wyszukującego w celu jej lepszej prezentacji dla użytkownika. Najbardziej intuicyjną i przystępną formą wizualizacji rezultatów, znaną chyba każdemu jest *katalog*. Jako pierwsi odkryli tę formę twórcy serwisu Yahoo [Yahoo, 03] tworząc początkowo ręcznie pielęgnowany rejestr stron internetowych. Jednak utrzymanie takiego katalogu przy gwałtownie wzrastającej liczbie stron internetowych stało się niemożliwe. Alternatywą

stało się automatyczne grupowanie[1] listy referencji do dokumentów [Zamir & Etzioni, 99], łączące ze sobą dokumenty o podobnej treści, tworzące w ten sposób swego rodzaju wirtualny katalog.

Rzadko kiedy zdarza się, aby dobry serwis wyszukiwawczy zapewniał także jakiekolwiek rozszerzenie prezentacji poza zwykłą listą wyników. Zazwyczaj systemy wzbogacające wizualizację wyników mają charakter wtórny i korzystają z innych serwisów wyszukiwawczych w celu pozyskania listy wyników do przetworzenia. Jednym z nielicznych systemów komercyjnych tego typu jest serwis Vivisimo [Vivisimo, 03], który także pobiera dane z innych dostępnych wyszukiwarek jak Altavista [Altavista, 03], MSN [MSN, 03], Netscape [Netscape, 03], Lycos [Lycos, 03] czy Looksmart [LookSmart, 03]. Dodatkowo umożliwia także wyszukiwanie w serwisach wiadomości m.in. CNN, BBC i Reuters. Pozostałe systemy takie jak Grouper [Zamir & Etzioni, 99], czy Carrot [Weiss, 01] mają charakter naukowy i także nie posiadają własnego indeksu stron, lecz wykorzystują zasoby istniejących.

Pojęcie wrappera

Serwisy wyszukiwawcze stworzone zostały do wykorzystania przez ludzi, a nie do automatycznego pobierania z nich informacji w celu ich dalszego przetwarzania. Stąd lista wyników jest dostępna jedynie w postaci dokumentu HTML, głównie zorientowanego na jej wizualizację, a nie na możliwość łatwej ekstrakcji danych. Do tego celu potrzebne są procedury zdolne do wydobycia potrzebnej informacji z pobranego dokumentu HTML [Kushmerick, 97]. W literaturze takie specjalizowane procedury są powszechnie nazywane *wrapperami* [Papakonstantinou et al., 95] [Childovskii et al., 97] [Roth&Schwartz, 97].

Oczywiście można sobie wyobrazić ręczne tworzenie takich modułów programowych dostosowanych do poszczególnych wyszukiwarek. Utrzymanie takiego systemu jest jednak bardzo pracochłonne, głównie ze względu na bardzo częste zmiany w wewnętrznej strukturze stron, które dla normalnego użytkownika objawiają się co najwyżej zmienionym wyglądem, jednak mogą sprawić, iż *wrapper* przestanie pobierać jakiekolwiek dane. Dlatego bardzo potrzebnym narzędziem w takich zastosowaniach jest automatyczny lub pół-automatyczny generator *wrapperów*, który będzie potrafił utworzyć procedurę ekstrahującą potrzebne dane.

Dotychczasowe prace

Do tej pory naukowcy zajmowali się tym problemem w znacznie szerszym ujęciu. Poszukiwali metod indukcji *wrapperów* dla dowolnych źródeł danych o dowolnym formacie i prezentacji. Nicholas Kushmerick dokonał przeglądu podstawowych klas *wrapperów* i zaproponował metody ich automatycznej indukcji [Kushmerick, 97]. Metody te opierały się na dokumentach z ręcznie oznaczonymi fragmentami, które *wrapper* powinien wyekstrahować. Podobne założenia przyjęli Ashish i Knoblock [Ashish & Knoblock, 97] do indukcji hierarchicznych *wrapperów* jako automatów skończonych. Wszystkie te prace koncentrowały się jednak na bardzo ogólnym zadaniu indukcji *wrapperów* dowolnych stron i wymagały wstępnego przygotowania źródeł, stąd wszystkie te podejścia można określić jako pół-automatyczne.

[1] ang. clustering

Przyglądając się stronom z rezultatami wielu przeglądarek można łatwo zauważyć, iż w zasadzie ich wygląd jak i elementy na nich zawarte są bardzo do siebie podobne. Na uniwersytecie Stanford został zainicjowany projekt STARTS [STARTS, 97] opisujący podstawowe elementy funkcjonalne, wizualne i semantyczne, które powinien zawierać system udostępniania i wyszukiwania dokumentów. Wymienia się tam kilka podstawowych części, które powinny być podane dla każdego dokumentu na liście wyników: tytuł, tekst dokumentu (lub fragment), czas modyfikacji, adres URL, pod którym dokument jest dostępny. Wyszukiwarki internetowe w większości stosują się do tych zaleceń. Stąd też powstaje możliwość w pełni automatycznego tworzenia wrapperów dla takich stron, korzystając z ich wysokiej strukturalności i wiedzy na temat informacji, które tak utworzony *wrapper* miałby pozyskiwać.

1.1 Cel i zakres pracy

Cel pracy Celem tej pracy jest utworzenie w pełni automatycznego generatora wrapperów dla stron wyników wyszukiwania bazujących na paradygmacie zapytanie-lista wyników. Algorytm generujący wrappery, (zwany dalej *WrapIT*) opierając się na analizie wewnętrznej struktury strony HTML, wyszuka powtarzające się regularności, aby odnaleźć wzorzec określający pojedynczy wynik wyszukiwania. Praca obejmuje także implementację i ocenę skuteczności działania algorytmu.

Zadania szczegółowe Zadania ogólne przedstawione w poprzednim akapicie zostaną zrealizowane przez następujące zadania szczegółowe:

1. Analiza problematyki wyszukiwania w sieci i sposobów prezentacji wyników wyszukiwania.
2. Przegląd literatury dotyczącej wrapperów i ich automatycznej generacji.
3. Konstrukcja algorytmu WrapIT - odkrywającego lokalizację odnośników do dokumentów na stronie wyników wyszukiwania na podstawie analizy struktury wewnętrznej dokumentu HTML.
4. Implementacja algorytmu WrapIT, oraz komponentu wejściowego wykorzystującego wygenerowane wrappery w środowisku Carrot^2.
5. Ocena i analiza działania algorytmu w praktycznych zastosowaniach dla istniejących serwisów wyszukujących.

1.2 Struktura pracy

Układ pozostałej części pracy będzie następujący. W rozdziale 2 przedstawimy ogólne zagadnienia związane z wyszukiwaniem w sieci Internet. Rozdział 3 zostanie poświęcony pozyskiwaniu informacji z sieci przy wykorzystaniu wrapperów. Te swa rozdziały będą miały charakter odtwórczy i przedstawią aktualny stan nauki w tych dziedzinach.

Rozdział 4 wprowadzi algorytm FREQT wykorzystywany do wyszukiwania regularności w dokumentach semi-strukturalnych. Przedstawione także zostaną zagadnienia analizy częstościowej dokumentów tekstowych wykorzystywane później w analizie semantyki elementów wzorca w algorytmie WrapIT.

Rozdział 5 zostanie w całości poświęcony opisowi algorytmu WrapIT. Zostaną tu także przedstawione szczegóły wykorzystania algorytmu w systemie Carrot^2. W rozdziale 6 postaramy się przedstawić na przykładzie dokładną analizę działania poszczególnych faz działania algorytmu. W drugiej części tego rozdziału postaramy się dokonać oceny działania tego algorytmu przy wykorzystaniu obiektywnych miar jakości działania w praktyce.

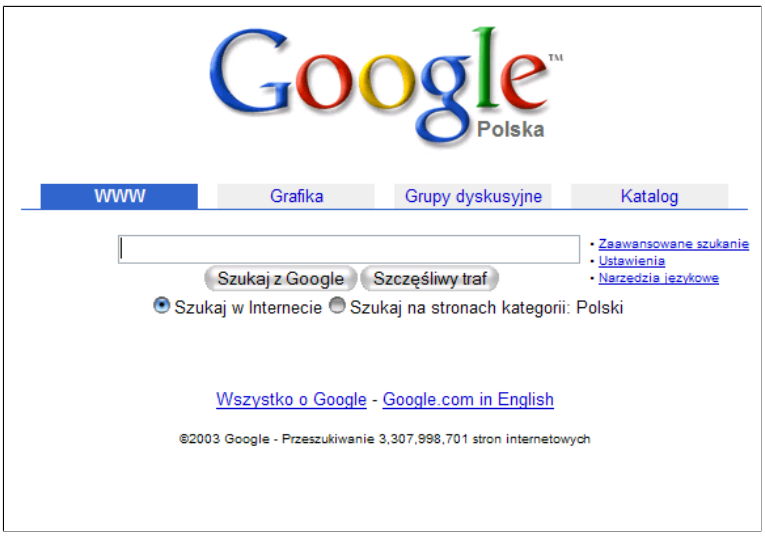
2 Wyszukiwanie dokumentów w sieci

W tym rozdziale przedstawimy problem wyszukiwania w sieci Internet. Przedstawimy także cele i zastosowania, a także wewnętrzną konstrukcję serwisów wyszukujących. Dokonamy przeglądu metod prezentacji wyników stosowanych w istniejących wyszukiwarkach internetowych.

2.1 Serwisy wyszukujące

Podstawowym celem istnienia **serwisów wyszukujących**, zwanych także potocznie **wyszukiwarkami internetowymi** jest ułatwienie użytkownikowi odnalezienia interesujących go informacji znajdujących się na różnych dokumentach w sieci, takich jak strony (dokumenty) HTML[2], dokumenty PDF[3], czy inne.

Ilustracja 2.1
Strona startowa wyszukiwarki Google



Ogromna większość wyszukiwarek internetowych jest dostępna poprzez serwisy WWW[4] oparte na protokole HTTP[5]. Użytkownik chcący skorzystać z usług jakiegoś serwisu musi wpisać jego adres w swojej przeglądarce internetowej. Ilustracja 2.1 przedstawia przykładową stronę startową wyszukiwarki Google [Google, 03]. Na stronie startowej zazwyczaj znajduje się formularz umożliwiający użytkownikowi wprowadzenie **zapytania**,

---

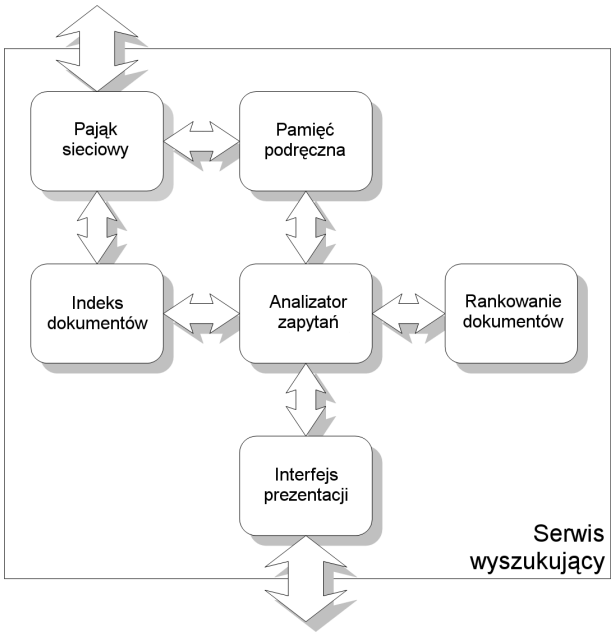
2 HyperText Markup Language
3 Portable Document File
4 World Wide Web - ogólnoświatowa pajęczyna
5 HyperText Transfer Protocol

czyli zazwyczaj listy słów kluczowych odpowiadających informacjom przez niego poszukiwanym. W wyniku serwis wyszukiwawczy wyświetla listę odsyłaczy do stron lub dokumentów najbardziej wg. niego związanych z zadanym zapytaniem. Twórcy serwisów wyszukiwawczych muszą jednak wybrać pomiędzy ogólnością a specyfiką. Stąd wykształciły się dwa kierunki: wyszukiwarki ogólne - wyszukujące dokumenty w sieci, nie dostosowane jednak do pewnych specyficznych zastosowań, oraz wyszukiwarki profilowane, czyli dostosowane do pewnych specyficznych zastosowań. Przykładami takich specyficznych serwisów jest wyszukiwarka grup dyskusyjnych Google (Google-Groups [Google Groups, 03]), serwis przeszukujący zasoby serwerów FTP[6] - AllTheWeb-FTP [AllTheWeb FTP, 03], serwis CiteSeer [CiteSeer, 03] czy też wyszukiwarka części elektronicznych Questlink [Questlink, 03].

2.2 Struktura wewnętrzna wyszukiwarki

Ilustracja 2.2 przedstawia elementy typowego serwisu wyszukującego, takich jak: pająk internetowy[7], pamięć dokumentów, indeks dokumentów, funkcja rankująca dokumenty, analizator zapytań oraz interfejs prezentujący wyniki użytkownikowi. Ilustracja przedstawia jedynie zależności logiczne w architekturze wyszukiwarki a nie implementacyjne.

Ilustracja 2.2 *Elementy typowego serwisu wyszukującego*



2.2.1 Pająk internetowy

Pająk internetowy, często nazywany także robotem, jest elementem odpowiedzialnym za pozyskiwanie nowych zasobów z sieci Internet aby umożliwić ich późniejsze wyszukanie.

---

[6] File Transfer Protocol

[7] ang. crawler

Proces ten opisany jest przez **Problem Odkrywania w Sieci** (ang. Web Discovery Problem) [Selberg, 99]: *Jak można zlokalizować wszystkie strony WWW?*. W najprostszej postaci pająki realizują swoje zadanie ekstrahując adresy URL z wcześniej pobranych dokumentów.

Ilustracja 2.3
Pseudo-kod pająka internetowego [Selberg, 99]

```
WebCrawler(UrlPool urlPool, DocumentIndex documentIndex)
{
  tak długo jak urlPool nie jest pusty
  {
    url = pobierz URL z urlPool;
    doc = sciagnij url;
    newUrls = wyszukaj nowe URL-e z doc;
    wstaw doc do documentIndex;
    wstaw url do indexedUrls;
    dla kazdego u w newUrls
    {
      jezeli (u nie jest w indexedUrls)
      {
        dodaj u do urlPool
      }
    }
  }
}
```

Poza podstawowym zadaniem pająka jakim jest pozyskiwanie nowych dokumentów, drugą bardzo istotną rolą jaką ma do spełnienia jest utrzymanie bazy już pobranych dokumentów. Robot musi w określonych odstępach czasowych odwiedzać zindeksowane strony, sprawdzając czy wciąż istnieją, a także uaktualniając swoją bazę w przypadku zmiany zawartości. Odstęp czasowy weryfikacji zależy od tego jak często dana strona zmienia swoją zawartość. Przykładowo serwisy informacyjne powinny być odwiedzane przez pająka codziennie, lub nawet co kilka godzin. Problem utrzymania bazy pobranych dokumentów został określony jako **Problem Spójności w Sieci** [8] [Selberg, 99].

Wiele serwisów internetowych generuje swoją zawartość dynamicznie, zmieniając ją przy każdym odwołaniu i powodując iż indeksowanie ich jest kompletnie bezcelowe. Stąd administratorzy mogą oznaczać takie serwisy lub ich fragmenty przy pomocy **Protokołu Wykluczania dla Robotów** (ang. Robot Exclusion Protocol) [REP, 03].

2.2.2 Indeks dokumentów

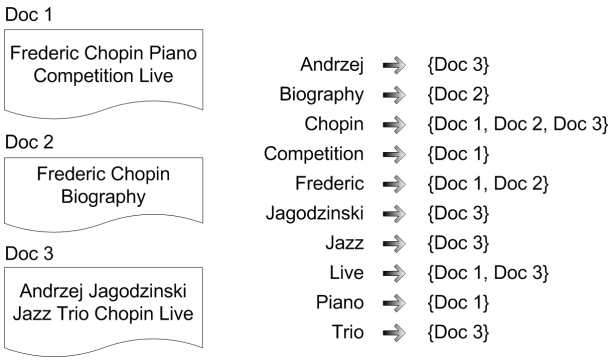
Podstawowym celem indeksu dokumentów jest wspomaganie rozwiązania problemu: *Które dokumenty zawierają podane słowo*. Aby odpowiedzieć na to pytanie większość serwisów wykorzystuje struktury danych oparte na idei **indeksu odwróconego**. Indeks taki jest skonstruowany podobnie jak spis terminów na końcu książki - dla każdego słowa przechowywana jest lista dokumentów je zawierającego (patrz: Ilustracja 2.4).

Możliwym rozszerzeniem indeksu odwróconego jest wersja, gdzie przechowywana jest dodatkowo informacja o lokalizacji słowa w dokumencie, dając możliwość wyszukiwania

[8] ang. Web Coherence Problem

fraz oraz wyszukiwania przybliżone wykorzystują informację o słowach które znajdują się blisko siebie.

Ilustracja 2.4 Przykład odwróconego indeksu



W prawdziwych zastosowaniach sumaryczny rozmiar indeksowanych dokumentów może sięgać setki terabajtów. To może prowadzić do wielu problemów zarówno związanych z przechowywaniem indeksu jak i samym wyszukiwaniem. Dodatkowe techniki indeksowania umożliwiają zredukowanie rozmiaru indeksu do 30% umożliwiając dodatkową redukcję aż do 10% przy wykorzystaniu kompresji [Baeza-Yates et al., 99].

2.2.3 Pamięć dokumentów

Wiele serwisów przechowuje nie tylko indeks stron internetowy ale także całe pobrane dokumenty. W przypadku wyszukiwarki Google umożliwia to nie tylko pobieranie snippetów, ale także udostępnianie wcześniej pobranych dokumentów z pamięci. Jest to szczególnie przydatne w przypadku gdy oryginalny dokument nie jest już dostępny.

2.2.4 Funkcja rankująca dokumenty

Nawet najbardziej precyzyjne zapytanie zadane wyszukiwarce internetowej może dać w wyniku tysiące a nawet miliony dokumentów. Konieczne staje się więc zastosowanie dodatkowych technik umożliwiających takie uporządkowanie dokumentów aby na początku znalazły się te najbardziej odpowiadające wprowadzonemu zapytaniu i tym samym potencjalnie najbardziej interesujące dla użytkownika. Technika taka jest zwana **rankowaniem wyników** wyszukiwania.

Historycznie serwisy wyszukujące do rankowania wykorzystywały miary oparte na **modelu wektorowym** reprezentacji dokumentów. Zgodnie z modelem, zarówno dokumenty jak i zapytanie reprezentowane są jako wielowymiarowe wektory, w których każdy wymiar odpowiada jednemu słowu występującemu w indeksowanych dokumentach. Odpowiedniość danego dokumentu jest obliczana na podstawie iloczynu wektorowego wektorowej reprezentacji tego dokumentu i wektorowej reprezentacji zapytania. Mimo tego, że taka funkcja rankująca działa dobrze dla tradycyjnych zastosowań pozyskiwania danych, to w przypadku zastosowań w sieci Internet możliwa jest łatwa manipulacja umieszczając w dokumencie wszystkie możliwe słowa kluczowe. Nowoczesne

wyszukiwarki wykorzystują techniki oparte na połączeniach hiperlinkowych pomiędzy dokumentami, które są dużo bardziej odporne na manipulacje.

Algorytm rankujący oparty na hiperlinkach wykorzystuje informację o połączeniach między dokumentami. Opiera się on na założeniu, iż liczba odnośników prowadzących do danego dokumentu z innych dokumentów jest miarą jego jakości i przydatności. Następujący opis algorytmu Page Rank - bazującego na hiperlinkach algorytmu wykorzystywanego w przeglądarce Google został podany w [Baeza-Yates et al., 99]:

> *Algorytm symuluje użytkownika nawigującego losowo po sieci, przechodzącego na losową stronę z prawdopodobieństwem q lub podążającego za losowym hiperlinkiem (na kolejnej stronie) z prawdopodobieństwem 1-q. Dalej zakłada się, iż użytkownik nigdy nie powraca do raz już odwiedzonej strony podążając spowrotem po raz już eksplorowanym linku. Taki proces może być zamodelowany przez łańcuch Markowa, skąd można obliczyć prawdopodobieństwo przebywania dla każdej strony. Ta wartość jest następnie wykorzystywana jako część funkcji rankującej.*

2.2.5 Analizator zapytań

Funkcją analizatora zapytań jest koordynacja przetwarzania zapylania użytkownika. Podczas przetwarzania zapytania użytkownika analizator komunikuje się z indeksem dokumentów, pamięcią podręczną i komponentem rankującym w celu uzyskania listy dokumentów najlepiej odpowiadających zapytaniu zadanemu przez użytkownika. Taka lista jest następnie przekazywana do komponentu prezentacyjnego, który przekazuje odpowiedź do użytkownika.

2.2.6 Interfejs prezentacji wyników

Funkcją interfejsu prezentacji wyników jest pokazanie użytkownikowi listy wyników w taki sposób, aby z łatwością mógł zidentyfikować dokumenty których tak naprawdę poszukuje. Stąd wynika proste założenie, iż należy zaprezentować wyniki w taki sposób, aby użytkownik nie musiał przeszukiwać długiej listy wyników, z których większość jest dla niego nieprzydatna. Wiąże się to bezpośrednio z obserwacją, iż użytkownicy najczęściej nie potrafią sformułować dostatecznie szczegółowych zapytań charakteryzujących dokładnie obszar ich zainteresowania, a jedynie dość ogólne zapytania pokrywające bardzo dużą przestrzeń dokumentów.

Poniżej prezentujemy przegląd podstawowych sposobów prezentacji wyników wyszukiwania.

Lista rankowana

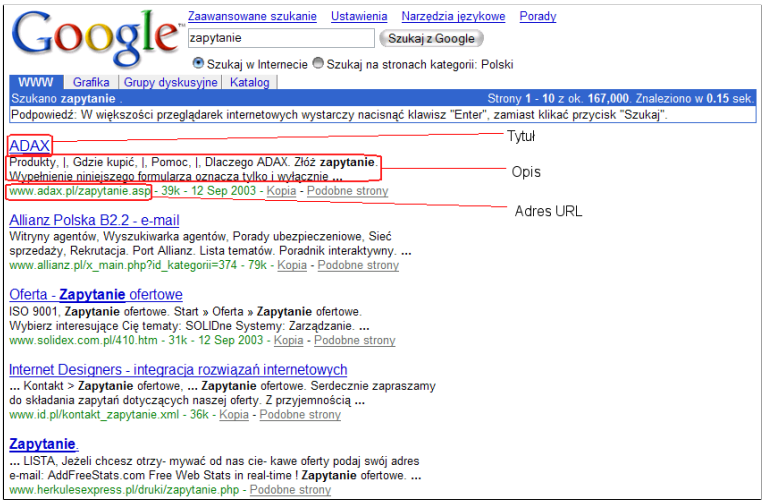
Lista rankowana jest dziś najpopularniejszą i najczęściej stosowaną formą prezentacji wyników wyszukiwania. W modelu takim wyniki prezentowane są w postaci listy

posortowanej wg. dopasowania do zapytania. Najbardziej pasujące dokumenty pojawiają się na początku listy. Pojedynczy wynik zazwyczaj zawiera najczęściej odsyłacz do dokumentu, jego tytuł i fragment zawartości odpowiadający najtrafniej zapytaniu.

Definicja snippetu

Dodatkowo bardziej zaawansowane wyszukiwarki wyświetlają krótki opis/streszczenie wyszukanej pozycji. Takie informacje dotyczące pojedynczego wyniku wyszukiwania, tj. adres URL[9] pod którym znajduje się dokument, tytuł dokumentu oraz jego fragment lub opis, będziemy dalej nazywać **snippetem** [10]. Przykładową stronę z wynikami wyszukiwania przedstawia Ilustracja 2.5.

Ilustracja 2.5 *Przykładowe wyniki wyszukiwania - Google*



Mimo tego, że model taki jest bardzo szeroko stosowany posiada liczne wady [Zamir & Etzioni, 99] [Weiss, 01]:

a. Użytkownik musi przeglądać długą listę dokumentów, z których bardzo liczne nie odpowiadają jego zapytaniu.
b. Nie jest możliwe uzyskanie wyjaśnienia dlaczego dany dokument znalazł się w zbiorze wynikowym.
c. Nie ma bezpośredniej widocznej zależności między wyświetlonym dokumentem a zapytaniem.
d. Nie ma żadnej informacji na temat zależności istniejących między dokumentami zbioru wynikowego.
e. Wyniki muszą być przedstawione w określonym porządku, mimo tego że niektóre dokumenty są między sobą nieporównywalne.

Grupowanie wyników

Grupowanie wyników próbuje zorganizować otrzymaną listę rezultatów w pewne grupy tematyczne podobne do tych istniejących w katalogach internetowych. Jednakże jest dość duża różnica między systemami grupującymi a katalogami. Po pierwsze przy budowie grup tematycznych brane pod uwagę są jedynie dokumenty pasujące do zapytania.

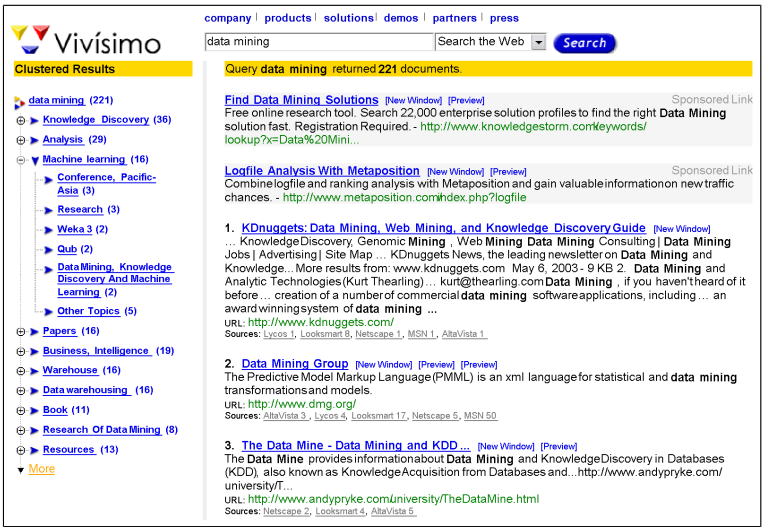
[9] Uniform Resource Locator

[10] snippet

Dokumenty nie są dopasowywane do istniejących predefiniowanych kategorii, lecz kategorie są tworzone on-line z uzyskanych rezultatów. Stąd też samo przyporządkowanie dokumentów do poszczególnych kategorii odbywa się dopiero w momencie wydania zapytania. Ponieważ grupy tworzone są ad-hoc istotne jest, aby każda grupa miała pewne uzasadnienie swojego istnienia. W tym celu każda grupa powinna zawierać odnośniki do dokumentów, których opisy są jak najbardziej do siebie zbliżone. Z drugiej strony każda z grup powinna się jak najbardziej odróżniać pod względem swojej zawartości od każdej innej grupy. Oczywiście w zależności od podejścia można stosować różne miary oceny podobieństwa między opisami dokumentów. Pod uwagę można brać zarówno podobieństwo występujących słów lub fraz, jak i próbować oceny znaczeniowej poszczególnych opisów. Zagadnienia grupowania zostały poruszone w [Zamir & Etzioni, 99] i definiują warunki które musi spełniać dobry system grupujący:

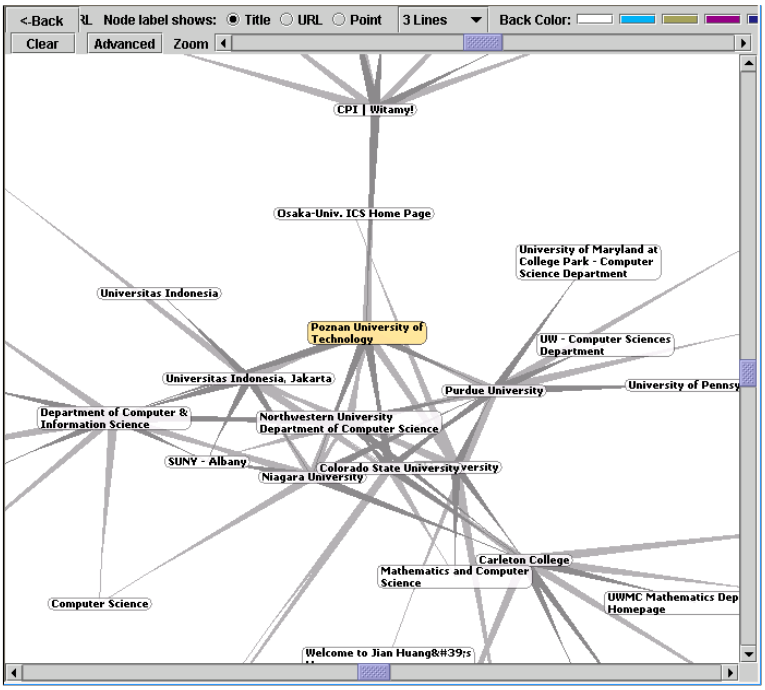
a. Odpowiedniość - dokumenty znajdujące się w zgrupowaniach muszą odpowiadać zapytaniu wydanemu przez użytkownika.
b. Dobre opisy - użytkownik na podstawie opisu zgrupowania powinien szybko i bez problemu stwierdzić, czy to zgrupowanie zawiera interesujące go dokumenty.
c. Działanie jedynie na snippetach - użytkownik nie jest skłonny czekać, aż system grupujący ściągnie całe zawartości stron. System musi polegać jedynie na wycinkach tekstu w celu utworzenia dobrych zgrupowań.
d. Szybkość - grupowanie on-line nie powinno wprowadzać dodatkowych opóźnień przy oczekiwaniu na wyniki wyszukania.
e. Przetwarzanie przyrostowe - algorytm powinien przetwarzać snippety tak szybko, jak przybywają do systemu grupującego bez potrzeby oczekiwania na wszystkie wyniki.

Ilustracja 2.6 *Interfejs metawyszukiwarki grupującej Vivisimo*



Historycznie jednym z pierwszych systemów stosujących się do tych zasad był Grouper wykorzystujący algorytm STC. Jednym z najbardziej zaawansowanych systemów tego typu, łączących w sobie zarówno metawyszukiwarkę jak i grupowanie tworzące hierarchie tematyczne jest Vivisimo [Vivisimo, 03].

Ilustracja 2.7
Interfejs graficzny z siecią powiązań TouchGraph GoogleBrowser



Poza takimi, możnaby nazwać "klasycznymi" sposobami prezentacji wyników wyszukiwania poszukuje się także metod bardziej wysublimowanych. Najczęściej spotykane są podejścia grafowe, gdzie wyniki prezentowane są w postaci graficznych węzłów połączonych siatką krawędzi. Krawędzie w takim grafie reprezentują różnego typu powiązania - mogą to być powiązania znaczenia/treści zawartych na tych stronach, jak również powiązania odnośników odnalezionych na badanych stronach. Dobrym przykładem będzie tutaj TouchGraph GoogleBrowser [TouchGraph, 03] wykorzystujący wyniki z wyszukiwarki Google do prezentacji grafu powiązań między witrynami.

3 Automatyczne pozyskiwanie informacji z sieci

W poprzednim rozdziale opisaliśmy zagadnienia związane z wyszukiwaniem informacji i dokumentów w sieci Internet. Przedstawiliśmy także podstawy działania systemów ułatwiających prezentację wyników takich jak metawyszukiwarki i systemy grupujące. Nie każdy taki system jest bezpośrednio związany z indeksem dokumentów. Metawyszukiwarki z samej swojej natury korzystają z zewnętrznych źródeł danych, natomiast prezentacja przy wykorzystaniu grup jest spotykana zarówno jako integralny element systemów wyszukujących lub też jako oddzielny serwis wykorzystujący zasoby klasycznych wyszukiwarek. Bardzo licznie spotykane są również systemy łączące obie te funkcjonalności - czyli grupujące metawyszukiwarki. Dla wszystkich systemów pobierających dane z zewnętrznych źródeł bardzo istotny jest komponent umożliwiający pobieranie danych z innych systemów wyszukujących - tzw. wrapper.

3.1 Definicja wrappera

Interfejs wyszukiwarek internetowych doskonalony jest pod kątem wykorzystania go przez ludzi. Stąd pobieranie z nich danych przez inne systemy informatyczne nie jest zadaniem prostym, gdyż jedynym dostępnym dla nich interfejsem jest standardowe zadanie zapytania przez przesłanie formularza HTML i rozkodowanie otrzymanej strony wynikowej również w języku HTML. Strony wynikowe są jednak także generowane automatycznie, wynika stąd ich bardzo wysoka strukturalność wewnętrzna. Umożliwia ona konstruowanie specjalizowanych procedur lub podprogramów potrafiących wydobyć z kodu źródłowego HTML strony potrzebnych danych. Takie procedury lub podprogramy będziemy dalej nazywać **wrapperami**. Ponieważ wrappery są zazwyczaj wykorzystywane do ekstrakcji danych powtarzających się, często pochodzących z baz danych, pozwolimy sobie tutaj zaczerpnąć terminologię z tej dziedziny informatyki i nazwiemy **krotką danych** wszystkie te fragmenty, które w całości stanowią pojedynczy wynik (np. w przypadku danych adresowych osób krotką będzie każde wystąpienie ulicy, kodu i miasta dotyczące konkretnej osoby).

Ręcznie kodowane wrappery

Wrappery stron wyników wyszukiwania rzadko kiedy są skomplikowane. Każdy poszukiwany element danych jest w jednoznaczny sposób określony przez poprzedzające go i następujące po nim znaczniki HTML. Można z łatwością wyobrazić sobie prosty wrapper, którego podstawą będą informacje o znacznikach okalających poszukiwaną informację. Tego typu wrappery zazwyczaj są ręcznie kodowane przez administratorów

systemów i są przystosowane do przetwarzania konkretnych stron. Tak naprawdę większość systemów pobierających w ten sposób dane zewnętrzne opiera się właśnie na ręcznie pisanych wrapperach.

Zaletą ręcznie wpisywanych wrapperów jest z pewnością ich dokładność i prostota. Mają jednak bardzo poważną wadę, a mianowicie nie są odporne na zmianę wewnętrznej konstrukcji przetwarzanego serwisu. Bardzo często zdarza się, iż administratorzy serwisów wyszukujących dokonują kosmetycznych zmian w wyglądzie strony wynikowej, dodają nową funkcjonalność, powodując czasem nawet bardzo nieznaczne zmiany w kodzie HTML strony. Mała zmiana w kodzie może jednak doprowadzić do błędów we wrapperze. W takim przypadku niezbędna jest kolejna ingerencja człowieka, który musi dostosować wrapper do nowej architektury. Widać więc wyraźnie iż nakład na utrzymanie ręcznie zakodowanego wrappera jest duży gdyż wymaga stałego nadzoru człowieka.

Automatyczna generacja wrapperów

Alternatywą dla takiej sytuacji mogą być automatycznie generowane wrappery. Najszerzej ta idea została przedstawiona w [Kushmerick, 97]. Autor przedstawia tam metodę indukcyjnego generowania wrapperów dla dowolnych semistrukturalnych źródeł danych. Generowane wrappery zostały podzielone na klasy, dla których zostały zaproponowane modele indukcyjne ich automatycznej generacji:

a. **Wrapper HRLT** - model ten składa się z 4 komponentów, h wykorzystywany jest do opuszczenia części nagłówkowej strony oraz t oznaczający część końcową strony. W korpusie dokumentu każda para lewego i prawego ogranicznika (odpowiednio l_k i r_k) jest wykorzystywana do wyekstrahowania k-tego atrybutu każdej kolejnej krotki danych. Zachowanie wrappera HRLT dla domeny o K atrybutach można opisać za pomocą wektora 2K + 2 ciągów znaków <h, t, l_1, r_1, ..., l_K, r_K>. Pseudokod wrappera HRLT jest przedstawiony poniżej:

```
ExecHRLT(wrapper <h, t, l1, r1, ..., lK, rK>, strona P)
    • pomin pierwsze wystapienie h w P
    • tak dlugo, jak pierwsze wystapienie l1
    jest przed kolejnym wytapieniem t w P powtarzaj:
        • dla kazdego <lk, rk>   {<l 1, r1>, ..., <lK, rK>}
            • ekstrahuj z P wartosc k-tego atrybutu kolejnej
            krotki danych spomiedzy natepnego
            wystapienia lk i nastepujacego po nim wystapienia rk
    • zwroc wszystki wyekstrahowane krotki
```

Wrapper HRLT jest podstawowym typem wrappera i dla niego zostanie przedstawiona metoda automatycznej indukcji, opisana szczegółowo w kolejnej sekcji.

b. **Wrapper LR** - jest to uproszczony model HRLT bez wyróżnionej części nagłówkowej h i części zakończenia t.
c. **Wrapper OCLR** - wrapper HLRT wykorzystuje komponenty h i t aby zapobiec niepoprawnej ekstrakcji danych w części nagłówkowej i końcowej strony. Wrapper OCLR wykorzystuje inną technikę radzenia sobie z tym problemem.

Zamiast traktować stronę jako "nagłówek + korpus + końcowka" ORLT traktuje stronę jako sekwencja krotek danych oddzielonych nieznaczącym tekstem. OCLR jest zaprojektowany na opuszczanie niepotrzebnego tekstu w miejscach pomiędzy krotkami (podobnie jak HRTL jest zaprojektowany na opuszczanie niepotrzebnego tekstu nagłówka i końca dokumentu).

OCLR wykorzystuje dwa komponenty o i c. Komponent o jest znacznikiem otwierającym krotki danych (zwany jest też rozpoczęciem ktorki), natomiast komponent c jest zamknięciem krotki danych (zakończeniem krotki). OCLR używa o i c do oznaczenia zakresu każdej krotki danych podobnie jak HRLT wykorzystuje znaczniki l_k i r_k do oznaczenia k-tego atrybutu.

Wrapper OCLR może być opisany poprzez wektor <o, c, l_1, r_1, ..., l_K, r_K> 2K + 2 ciągów znaków. Aby opisać dokładnie znaczenie poszczególnych komponentów przedstawiamy poniżej pseudokod procedury ExecOCLR opisującej działanie wrappera:

```
ExecOCLR(wrapper <o, c, l1, r1, ..., lK, rK>, strona P)
    • tak dlugo, jak istnieje kolejne wystapienie o w P
     • przejdz do nat pnego wystapienia o w P
        • dla kazdego <lk, rk>   {<l 1, r1>, ..., <lK, rK>}
            • ekstrahuj z P wartosc k-tego atrybutu kolejnej
            krotki danych spomiedzy natepnego
            wystapienia lk i nastepujacego po nim wystapienia rk
        • przejdz do nastepnego wystapienia c w P
    • zwroc wszystki wyekstrahowane krotki
```

d. **Wrapper HOCLRT** - ta klasa stanowi połączenie klas OCLR i HLRT. Wykorzystuje zarówno oznaczoną część początkową i końcową strony h i t, jak i oddzielne oznaczenie granic każdej krotki danych o i c. Jakkolwiek ta klasa jest w stanie opisać najwięcej występujących rodzajów stron, automatyczne wygenerowanie takiego wrappera jest najbardziej złożone obliczeniowo.

3.2 Automatyczne generowanie wrapperów

Automatyczne generowanie wrapperów oparte zostało na uczeniu indukcyjnym. Wejściem dla takiego systemu jest zbiór przykładów zachowania wyjściowego wrappera. Konstrukcja wrappera jest więc procesem odtworzenia wrappera dającego wyniki identyczne z przykładami.

Uczenie indukcyjne jest zadaniem, którego celem jest generalizacja danych zawartych w przykładach, tak aby na podstawie wyniku generalizacji móc w dość precyzyjny sposób oceniać nowe, wcześniej nie widziane przykłady. Biorąc za przykład trzy zdania: "Bush kłamał", "Blair kłamał" i "Einstein nie kłamał" dobrym uogólnieniem może być ogólne zdanie "Politycy kłamią". Zdanie takie jest sensowne, ponieważ jest zgodne z dostępnymi

przykładami. Na pytanie: "Czy Nixon kłamał?" wyidukowany system odpowiedziałby: "Tak".

Zadanie indukcji składa się z trzech podstawowych elementów:

- zbiór I = {..., I, ...} wystąpień
- zbiór Λ = {..., L, ...} etykiet
- zbiór Θ = {..., H, ...} hipotez

Każda hipoteza $H \in \Theta$ przypisuje etykietę $L \in \Lambda$ do wystąpienia $I \in I$. Dla przykładu z politykami zbiorem wystąpień jest zbiór ocenianych ludzi, natomiast etykieta oznacza, czy dany człowiek kłamie czy też nie. Hipotezy są dowolnymi funkcjami z $I \rightarrow \Lambda$. Stąd prawidłową hipotezą może być funkcja "Jeśli osoba jest mężczyzną, to jest kłamcą" przypisująca do dowolnego wystąpienia płci męskiej etykietę "kłamca".

Proces nauki polega na poszukiwaniu pewnej nieznanej hipotezy docelowej $T \in \Theta$. Dostępny jest na wejściu jedynie zbiór przykładów $E = \{..., <I, T(I)>, ...\} \subset 2^{I \times \Lambda}$ będących przykładami wystąpień wraz z etykietami przypisanymi zgodnie z T. Zadaniem indukcyjnym jest rekonstrukcja T na podstawie zbioru przykładów uczących E.

Do prawidłowego oznaczenia przykładowego zbioru wystąpień wykorzystywana jest **wyrocznia**. Wyrocznia jest to funkcja odpowiedzialna za dostarczenie oznaczonych zgodnie z T przykładów <I, T(I)>, gdzie $I \in I$ i $T(I) \in \Lambda$. Funkcję taką będziemy oznaczać jako $Oracle_T$, gdzie indeks dolny *T* sugeruje, iż funkcja wyroczni jest zgodna z poszukiwaną hipotezą T. Proces indukji będzie wielokrotnie odwoływał się do wyroczni w celu pozyskania zbioru przykładówE.

Ogólny algorytm indukcji

Algorytm indukcji można scharakteryzować następująco: otrzymując na wejściu funkcję wyroczni $Oracle_T$, daje w wyniku hipotezę $H \in \Theta$.

Ilustracja 3.1 *Podstawowy algorytm indukcji*

```
Procedura Indukuj(Oracle_T, Generalizuj_H)
  E ← {}
  powtarzaj
    <I, L> ← Oracle_T()
    E ← E ∑ {<I, L>}
  az zbior przykladow bedzie wystarczajacy
  zwroc wynik Generalizuj_H(E)
```

Algorytm *Indukuj* (patrz Ilustracja 3.1) działa następująco. Najpierw używa funkcji $Oracle_T$ do konstrukcji zbioru przykładów E = {..., <I, L>, ...}. W tym miejscu pominiemy element obliczania wystarczającej liczności zbioru przykładowego. Dokładna analiza znajduje się w [Kushmerick, 97]. Po zebraniu zbioru przykładowego E, *Indukuj* przekazuje E do funkcji $Generalizuj_H$. Funkcja ta jest specyficzna dla domeny analizowanego problemu, wyspecjalizowana do generalizacji klasy hipotez H. Ten schemat przetwarzania jest bardzo ogólny. Główne zadanie indukcyjne jest przeprowadzane w specyficznej dla danego zastosowania funkcji $Generalizuj_H$.

Generalizacja wrappera HRLT

Przedstawimy teraz funkcję generalizującą dedykowaną do generacji opisanych wcześniej wrapperów klasy HRLT. Za zbiór wystąpień przykładowych I przyjmiemy w tym przypadku zbiór odpowiedzi źródła danych na zadane zapytanie. Dla źródła HTML (np. wyszukiwarki internetowej) przykładem będzie pojedyncza strona wynikowa serwisu. Etykietami E dla takiej strony będą oznaczone fragmenty dla których chcielibyśmy wygenerować wrapper, czyli posługując się przykładem strony wynikowej wyszukiwania oznaczymy etykietami każde miejsce wystąpienia tytułu, opisu i adresu URL dla każdego pojedynczego wyniku. Zbiór hipotez Θ jest w tym przypadkiem zbiorem wszystkich możliwych wrapperów HRLT. Funkcja wyroczni $Oracle_T$ w tym przypadku będzie wykorzystywała użytkownika-eksperta do prawidłowego oznaczenia strony wynikowej.

Ilustracja 3.2 *Algorytm* $Generalizuj_{HRLT}$

```
procedura Generalizuj_HRLT(przyklady E = {<P_1, L_1>, ..., <P_N, L_N>})
    • dla r_1 <- kazdy prefiks przerwy miedzy krotkami dla S_1,1
    .
     .
      • dla r_K <- kazdy prefiks przerwy miedzy krotkami dla S_1,K
       • dla l_1 <-kazdy sufiks naglowka P_1
        • dla l_2 <- kazdy sufiks przerwy miedzy krotkami dla S_1,1
         .
          .
           • dla l_K <- kazdy sufiks przerwy miedzy krotkami dla S_1,K-1
            • dla h <- kazdy podciag naglowka P_1
             • dla h <- kazdy podciag zakonczenia P_1
              w <- <h, t, l_1, r_1, ..., l_K, r_K>
               • Jesli C_HRLT(w, E)
               to podaj w jako wynik
```

Ilustracja 3.2 przedstawia funkcję indukującą wrapper HRLT. Algorytm ten wykorzystuje proste podejście "generuj i testuj". Mając wejście E, $Generalizuj_{HRLT}$ przeszukuje przestrzeń wrapperów HRLT w poszukiwaniu wrappera *w*, takiego, który zaspokaja warunek C_{HRLT} . Warunek ten jest warunkiem poprawności wrappera i sprowadza się do sprawdzenia, czy wrapper prawidłowo oznacza wszystkie fragmenty danych dla wszystkich stron danych przykładowych. $Generalizuj_{HRLT}$ wykorzystuje przeszukiwanie wgłąb: algorytm rozpatruje wszystkich kandydatów na r_1; dla każdego takiego r_1 algorytm rozpatruje wszystkich kandydatów na r_2; dla takiej pary r_1 i r_2 rozpatruje wszystkich kandydatów na r_3 itd.. Cała funkcja składa się z zagnieżdżonych 2K + 2 pętli. Lista kandydatów na każdy poziom jest ograniczona i skończona (zdefiniowana przez zawartość zbioru przykładowego), stąd algorytm jest także skończony.

W pracy [Kushmerick, 97] zostały zaprezentowane również analogiczne funkcje dla pozostałych klas wrapperów. Została także przeprowadzona analiza skuteczności i wydajności automatycznej generacji wrapperów. Należy jednak zauważyć, iż to podejście nadal wymaga czynności wykonywanych ręcznie przez użytkownika. W tym przypadku nie jest to już samo definiowanie wrappera, lecz prostsza czynność oznaczenia danych przykładowych dla zaspokojenia potrzeb algorytmu *Indukuj*. Autorzy proponują jedynie zautomatyzowanie tego procesu jednie poprzez automatyczne oznaczanie charakterystycznych danych, które można łatwo i jednokrotnie zdefiniować (np. przy

użyciu wyrażeń regularnych), takich jak elementy daty, godziny, adresy URL. Dla zastosowań generacji wrapperów stron wyników wyszukiwania nie da się niestety ustalić uniwersalnych reguł dla oznaczenia wystąpień opisów i tytułów poszczególnych wyników.

4

WrapIT - Pojęcia podstawowe

W tym rozdziale przedstawimy algorytmy i techniki wykorzystane przez Nas do konstrukcji algorytmu WrapIT. Na początku przedstawimy algorytm analizy wzorców w semistrukturalnych dokumentach FREQT [Asai et al., 02] wykorzystany jako podstawa algorytmu odnajdującego wzorzec snippetu na stronie wyników wyszukiwania. Algorytm ten zostanie w Naszej pracy przedstawiony w postaci ogólnej - możliwej do wykorzystania w znacznie większym spektrum zastosowań niż Nasze. Z drugiej strony w kolejnym rozdziale pokażemy, jakie modyfikacje byliśmy zmuszeni wprowadzić do tego oryginalnego algorytmu w celu przystosowania go do naszego zadania - wyszukiwania wzorców na stronie wynikowej. W drugiej części przedstawimy zagadnienia związane z wyszukiwaniem informacji w dokumentach tekstowych wykorzystane przez Nas do analizy semantyki informacji zawartej w wyszukanym wzorcu.

4.1 Analiza drzewa HTML

4.1.1 Drzewo rozkładu HTML

Język HTML na początku był jedynie sieciową częścią standardu SGML[11]. Przez lata zmieniał się bardzo dynamicznie głównie pod wpływem wyścigu o dominację na rynku przeglądarek internetowych. Obecnie obowiązująca jest wersja 4 tego standardu i ta wersja, lub też nieco starsza 3.2 jest wykorzystywana przez twórców stron WWW jak i wszystkie przeglądarki internetowe. Obecnie wkracza także nowy standard XHTML stosujący zasady języka XML do dokumentów HTML-owych.

Język XML z samej swojej natury opisuje dokument w sposób strukturalny tworząc pewną hierarchię. Próbę takiego opisu można także podjąć dla dokumentów HTML. Poniżej przedstawimy przykład ilustrujący rozkład prostej struktury HTML.

```
<H1><B>Tekst</B><I>Tekst</I><H1>
```

Dużo łatwiej zanalizować ten fragment zapisując go z hierarchicznymi wcięciami zgodnie z zawieraniem się poszczególnych znaczników HTML:

[11] ang. Standard Generalized Markup Language

```
<H1>
 <B>
  Tekst
 </B>
 <I>
  Tekst
 </I>
<H1>
```

Każde wcięcie oznacza zawieranie się znacznika wewnątrz innego znacznika. Stąd można już łatwo zbudować drzewiastą strukturę tego fragmentu:

```
    H1
   /  \
  B    I
 /      \
Tekst   Tekst
```

Utworzenie drzewa dokładnie odzwierciedlającego strukturę dokumentu HTML nie zawsze jest możliwe, głównie ze względu na to, że SGML-owa definicja standardu dopuszcza istnienie znaczników których otwarcie i domknięcie nie jest zawarte w jednym i tym samym poddrzewie (przykładem takiego znacznika jest <*FORM*>). Dodatkowym utrudnieniem przy konstrukcji drzewa rozkładu są znaczniki, które standard dopuszcza jako niedomknięte. W większości przypadków jednak wydedukowanie miejsca prawidłowego domknięcia takiego znacznika. XHTML stanowi duży postęp w tej dziedzinie. Oparcie tego standardu na XML-u gwarantuje, iż konstrukcja drzewa rozkładu jest prosta i jednoznaczna. Nie ma potrzeby wtedy również stosować żadnych wyszukanych parserów HTML, lecz wystarczy użyć istniejących dla XML-a parserów DOM. Mimo tego, iż XHTML jeszcze długo nie będzie jedynym obowiązującym standardem, twórcy stron już teraz starają się stosować do jego zasad, unikając chociażby konstrukcji zakłócających strukturę drzewiastą znaczników HTML. Stąd wygenerowanie drzewa rozkładu jest możliwe praktycznie dla każdej strony wyników wyszukiwania.

Mając możliwości utworzenia z dokumentu HTML-owego drzewiastej struktury znaczników, możemy z powodzeniem stosować algorytmy zaprojektowane do analizy danych semistrukturalnych (takich jak XML). Jednym z takich algorytmów jest FREQT, który przedstawimy w kolejnym podrozdziale.

4.1.2 Algorytm FREQT

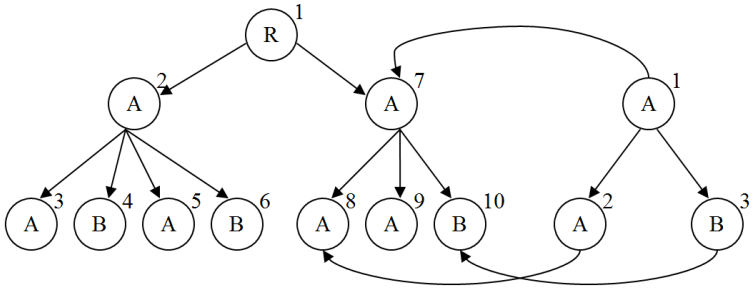
FREQT jest algorytmem efektywnego odkrywania podstruktur z dużych zbiorów danych semi-strukturalnych [Asai et al., 02]. Analizie zgodnie z intencją autorów mogą zostać poddane różne typy dokumentów, takich jak XML [XML, 00] i OEM [Abiteboul et al., 97]. Jak pokażemy w dalszej części pracy algorytm można także z powodzeniem wykorzystywać do analizy dokumentów HTML, jeśli zbuduje się dla nich odpowiednie drzewo rozkładu i traktuje odtąd jako dokument ustrukturalizowany.

Algorytm ten modeluje dane semi-strukturalne jako **etykietowane uporządkowane drzewa** i rozpatruje problem odkrywania wszystkich **drzewiastych wzorców**, które mają conajmniej **minsup** poparcie w danym zbiorze danych semi-strukturalnych. Podstawą metody jest tzw. **prawostronne rozszerzenie** - technika rozbudowy drzewa przez dołączanie nowych węzłów tylko do **prawej skrajnej ścieżki** drzewa. Algorytm skaluje się prawie liniowo wraz ze wzrostem maksymalnego rozmiaru wzorca, zależnie od jego maksymalnej głębokości.

Pojęcia podstawowe

Jako model dla danych oraz wzorców semi-strukturalnych takich jak modele XML i OEM, wykorzystana zostanie klasa etykietowanych drzew uporządkowanych zdefiniowanych poniżej. Dla zbioru A, $\#A$ oznacza liczność zbioru A. Niech $\Lambda = \{l, l_0, l_1, ...\}$ będzie skończonym alfabetem etykiet, które będą oznaczać atrybuty danych semistrukturalnych, lub też tagi.

Ilustracja 4.1 *Drzewo danych* D *i drzewo wzorca* T *na zbiorze etykiet* $\Lambda = \{A, B\}$



Etykietowane drzewo uporządkowane na Λ (w skrócie drzewo uporządkowane) jest 6-stką $T = (V, E, \Lambda, L, \upsilon_0, \lesssim)$. Odpowiednie symbole mają tutaj następujące znaczenie: V - zbiór węzłów drzewa, E - zbiór krawędzi łączących węzły, Λ - alfabet etykiet, którymi oznaczane są węzły przy wykorzystaniu funkcji etykietującej L. Wyróżniony jest korzeń drzewa υ_0 oraz relacja uporządkowania węzłów w drzewie $\lesssim$. Teraz zostaną omówione szczegółowe własności tej szóstki: $G = (V, E, \upsilon_0)$ jest drzewem z korzeniem $\upsilon_0 \in V$. Jeżeli $(u, v) \in E$ wtedy mówimy, że u jest **rodzicem** v lub że v jest **dzieckiem** u. Przypisanie $L : V \rightarrow \Lambda$, zwana **funkcją etykietującą**, przypisuje etykietę $L(\upsilon)$ dla każdego węzła $\upsilon \in V$. Relacja binarna $\lesssim \in V^2$ reprezentuje **relacje sąsiedztwa** dla uporządkowanego drzewa T taką, że jeśli v i u są dziećmi tego samego rodzica i $v \lesssim u$ wtedy i tylko wtedy gdy v jest starszym bratem u. W dalszej części mając drzewo $T = (V, E, \Lambda, L, \upsilon_0, \lesssim)$, odnosimy się do V, E, L i $\lesssim$ odpowiednio jako V_T, E_T, L_T i $\lesssim_T$, jeśli nie jest jasne z kontekstu jakiego drzewa dotyczą oznaczenia.

Niech T będzie etykietowanym drzewem uporządkowanym. Rozmiar T jest zdefiniowany przez liczbę jego węzłów $T = \#V$. **Długością** ścieżki w T określamy liczbę węzłów należących do tej ścieżki. Jeśli istnieje ścieżka z węzła u do węzła v to mówimy, że u jest **przodkiem** v lub że v jest **rodzicem** u. Dla każdego $p \geq 0$ i węzła v, p-ty rodzic v, oznaczany jako $\pi^p(v)$, jest jednoznacznie określonym przodkiem u węzła v takim, że długość ścieżki z u do v wynosi p + 1. Z definicji $\pi^0(v)$ jest samo v, a $\pi^1(v)$ oznacza rodzica v. **Głębokość** węzła υ w T, oznaczaną jako $depth(\upsilon)$ jest zdefiniowana przez długość ścieżki od korzenia υ_0 drzewa

T do węzła υ. Głębokość T jest długością najdłuższej ścieżki od korzenia do któregoś z liści T.

Ilustracja 4.1 pokazuje przykłady etykietowanych drzew uporządkowanych D i T, opartych na alfabecie $\Lambda = \{A, B\}$, gdzie okrąg z liczbą z prawej strony oznacza węzeł υ, a symbol wewnątrz okręgu oznacza jego etykietę l(υ). Węzły w drzewach są także ponumerowane w porządku wgłąb.

Dopasowanie drzew

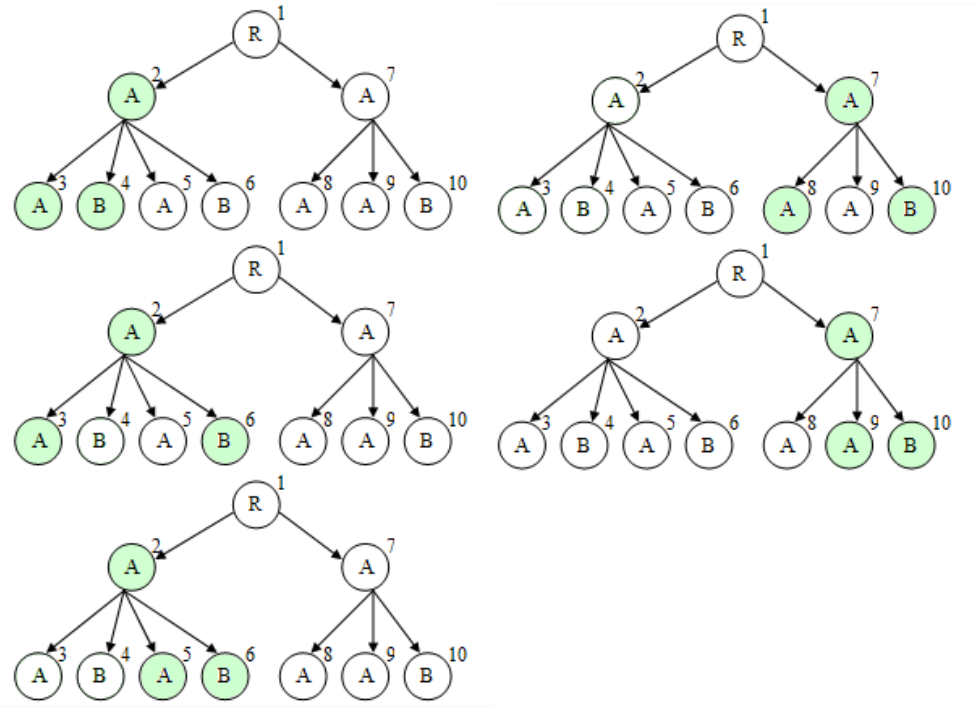
Niech T i D będą uporządkowanymi drzewami na alfabecie Λ i nazwanymi odpowiednio **drzewem wzorca** i **drzewem danych**. Drzewo T jest także ponumerowane i posiada $k \geq 0$ węzłów. Takie drzewo T nazywamy **k-wzorcem** na alfabecie Λ (w skrócie k-wzorzec). Zbiór wszystkich wzorców o rozmiarze k oznaczamy jako T_k . W takim razie definiujemy zbiór wzorców na alfabecie Λ przez $T = \cup_{k\geq 0} T_k$.

Najpierw zdefiniujemy **funkcję dopasowującą**. Funkcja jeden do jednego $\varphi: V_T \rightarrow V_D$ z węzłów drzewa T do drzewa D, zwana *funkcją dopasowującą* T *na* D , spełnia następujące warunki dla każdego $\upsilon, \upsilon_1, \upsilon_2 \in V_T$:

- φ zachowuje relację rodzicielstwa, tj. $(\upsilon_1, \upsilon_2) \in E_T$ wtedy i tylko wtedy, gdy $(\varphi(\upsilon_1), \varphi(\upsilon_2)) \in E_D$.
- φ zachowuje relację sąsiedztwa, tj. $\upsilon_1 \lesssim_T \upsilon_2$ wtedy i tylko wtedy, gdy $\varphi(\upsilon_1), \lesssim_D \varphi(\upsilon_2)$.
- φ zachowuje etykiety, tj. $L_T(\upsilon) = \varphi(L_D(\upsilon))$.

Drzewo wzorca T **pasuje** do drzewa D, lub drzewo T **występuje w** D , jeśli istnieje jakaś funkcja dopasowująca φ z T do D. W takim przypadku **całkowite wystąpienie** T **w** D przy użyciu φ jeśli listą $Total(\varphi) = (\varphi(1),...,\varphi(k)) \in (V_D)^k$ wszystkich węzłów na które dopasowują się węzły T a **wystąpieniem korzenia T w D** przy użyciu φ jest węzeł $Root(\varphi) = \varphi(1) \in V_D$ na który dopasowuje się korzeń drzewa T, gdzie k = T. Dla wzorca T definiujemy Occ(T) = {Root(φ) φ jest funkcją dopasowującą T w D}, to jest, zbiór wszystkich wystąpień korzenia T w D. Dalej, **częstość** (lub **poparcie**) dla wzorca T w D, oznaczana jako $freq_D(T)$, jest zdefiniowana jako iloraz liczby wystąpień korzenia T do całkowitej liczby węzłów w D, to jest $freq_D(T) = \#Occ(T)/D$. Dla dodatniej liczby $0 \leq \sigma \leq 1$, wzorzec T jest σ-częsty w D, jeśli $freq_D(T) \geq \sigma$.

Ilustracja 4.2 *Wszystkie możliwe dopasowania przykładowego wzorca* T *do drzewa danych* D *pokazanych na Ilustracja 4.1*



Dla przykładu rozpatrzmy drzewa, które przedstawia Ilustracja 4.1. Na tym rysunku pewna funkcja dopasowująca φ_1 , dla wzorca T zawierającego 3 węzły na drzewo D zawierające 10 węzłów jest zaznaczona przez zestaw strzałek. Wystąpienia całkowite i korzenia w odniesieniu do φ_1 wynoszą $\text{Total}(\varphi_1) = (7, 8, 10)$ i $\text{Root}(\varphi_1) = 7$. Jak pokazuje Ilustracja 4.2 istnieją dwa wystąpienia korzenia T w D, dokładnie 2 i 7, podczas, gdy można znaleźć 5 wystąpień całkowitych T w D, czyli (2, 3, 4), (2, 3, 6), (2, 5, 6), (7,8,10), (7,9,10). Stąd poparcie T w D wynosi $\text{freq}_D(T) = \#\text{Occ}(T)/D = 2/10.t$

Sformuowanie problemu

Teraz sformułujemy nasz problem eksploracji danych, zwany **problemem odkrywania częstych wzorców** *(ang. frequent pattern discovery problem*, który jest generalizacją problemu **odkrywania zbiorów częstych wystąpień dla reguł asocjacyjnych** *(ang. frequent itemset discovery problem in associacion rules)* [Agrawal&Sirkant, 94]

Problemem odkrywania częstych wzorców.

> Mając zestaw etykiet Λ, drzewo danych D i dodatnią liczbę $0 \leq \sigma \leq 1$, zwaną **minimalnym poparciem**, znajdź wszystkie σ-częste uporządkowane drzewa $T \in \mathcal{T}$ takich, że $\text{freq}_D(T) \geq \sigma$.

Algorytmy poszukujące

W tej sekcji, opisany zostanie algorytm rozwiązujący problem odkrywania częstych wzorców dla drzew uporządkowanych, który skaluje się prawi liniowo w odniesieniu do całkowitego rozmiaru maksymalnych częstych wzorców.

Ilustracja 4.3 *Algorytm odkrywania wszystkich częstych wzorców w drzewach uporządkowanych*

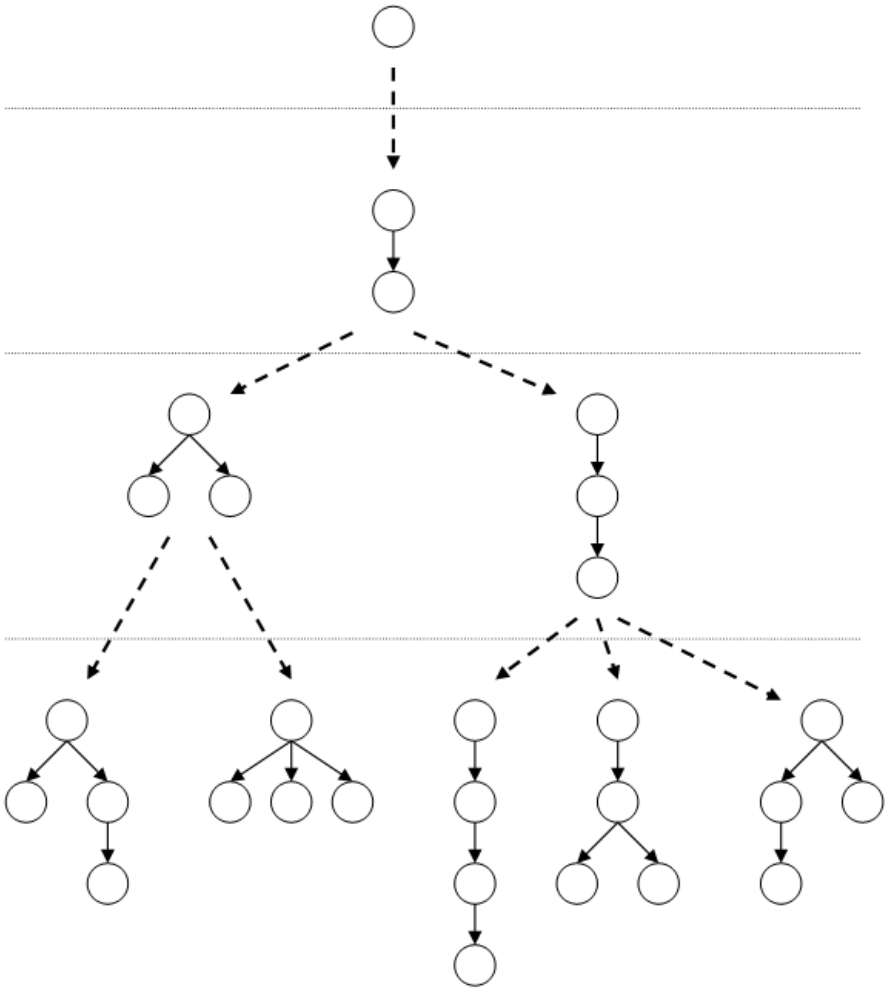
Algorytm FREQT

1. Oblicz zbiór $C_1 := F_1$ σ-częstych 1-wzorców i zbiór RMO_1 ich prawostronnych rozszerzeń poprzez przeskanowanie D; Ustaw $k := 2$;
2. Tak długo jak $F_{k-1} \neq \varnothing$ wykonaj:
    a. $<C_k, RMO_k> := \text{Expand-Trees}(F_{k-1}, RMO_{k-1})$; Ustaw $F_k := \varnothing$.
    b. Dla każdego wzorca $T \in C_k$ wykonaj: Oblicz $freq_D(T)$ na podstawie $RMO_k(T)$ i następnie jeśli $freq_D(T) \geq \sigma$ to $F_k := F_k \cup \{T\}$
3. Podaj wynik: $F = F_1 \cup ... \cup F_{k-1}$.

Na Ilustracja 4.3 prezentujemy algorytm **FREQT** odkrywający wszystkie częste uporządkowane wzorce drzewowe z częstością conajmniej równym danemu minimalnemu poparciu $0 > \sigma \geq 1$ w drzewie danych D. Podstawą projektu algorytmu jest strategia poszukiwania poziomowego [Agrawal&Sirkant, 94].

W pierwszym przebiegu FREQT po prostu tworzy zbiór F_1 wszystkich 1-wzorców i zapisuje ich wystąpienia w RMO_1 przeszukując drzewo D. W kolejnym przebiegu $k \geq 2$ FREQT przyrostowo oblicza zbiór C_k wszystkich **kandydackich** k-wzorców i zbiór RMO_k prawostronnych list wystąpień dla drzew C_k na podstawie zbiorów F_{k-1} i RMO_{k-1} obliczonych w poprzednim przebiegu przy użyciu prawostronnego rozszerzenia w podprocedurze *Expand-Trees*. Powtarzając ten proces tak długo, aż nie zostanie wygenerowany żaden nowy wzorzec, algorytm oblicza wszystkie σ-częste wzorce w D.

Ilustracja 4.4
Graf poszukiwania drzew uporządkowanych



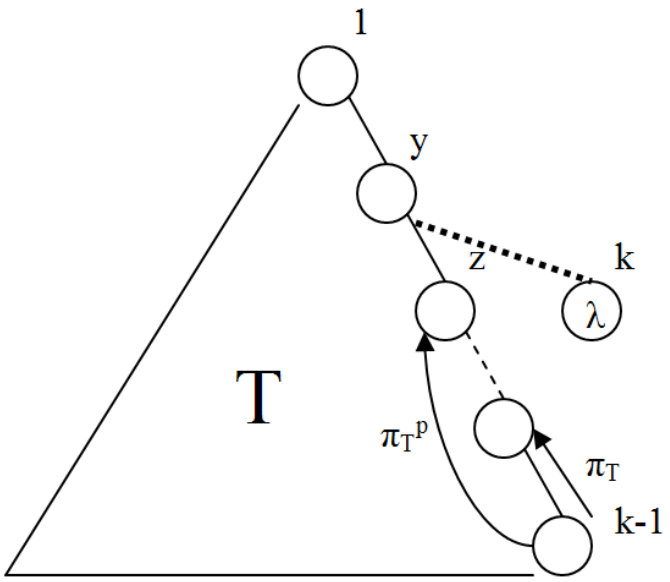
Efektywne enumerowanie drzew uporządkowanych

W tej podsekcji przedstawimy techinikę enumeracji generującą wszystkie drzewa uporządkowane bez powtórzeń przez ikrementalne rozszerzanie mniejszych drzew w wieksze.

Prawostronne rozszerzenie. Podstawy tego algorytmu enumeracji przedstawia Ilustracja 4.4. W poszukiwaniu, rozpoczynając od zbioru drzew składających się z pojedynczych węzłów, dla każdego $k \geq 2$ algorytm enumeracji rozszerza dane drzewo uporządkowane o rozmiarze k - 1 przez dołączenie nowego węzła tylko do węzła w skrajnie prawej gałęzi danego drzewa uzyskując w ten sposób wieksze drzewo o rozmiarze k.

Niech $k \geq 0$ będzie liczbą całkowitą i Λ będzie alfabetem. Niech T będzie dowolnym (k-1)-wzorcem na Λ, a rml(T) będzie skrajnym prawym liściem T. Zakładając, że drzewo T jest ponumerowane, to rml(T) = k. Stąd **skrajnie prawa gałąź** T jest ścieżką prowadzącą od rml(T) do korzenia drzewa T.

Ilustracja 4.5 (p, λ)-rozszerzenie *drzewa* T



Prawostronnym rozszerzenim T nazywamy dowolny k-wzorzec uzyskanym z T przez rozszerzenie węzła υ na prawej skrajnej gałęzidrzewa T poprzez dołączenie nowego skrajnie prawego dziecka do υ. Teraz przedstawimy dokładny formalny opis algorytmu zilustrowanego na Ilustracja 4.5. Niech $p \geq 0$ będzie nieujemną liczbą całkowitą, nie większą niż głębokość rml(T) a $\lambda \in \Lambda$ będzie dowolną etykietą, wtedy (p, λ)-rozszerzeniem T nazywamy uporządkowane drzewo S uzyskane z T przez dołączenie noewgo węzła k do węzła y = $\pi_T{}^p(x)$ jako prawego skrajnego dziecka y. Etykietą k jest λ. *Prawostronnym rozszerzeniem* drzewa uporządkowanego T jest (p, λ)-rozszerzenie T dla dowolnej liczby całkowitej $p \geq 0$ i dowolnej etykiety $\lambda \in \Lambda$. W takim przypadku mówimy, że T jest **poprzednikiem** S, lub też inaczej S jest **następnikiem** T.Wzorzec T jest **maksymalny pod względem prawostronnego rozszerzenia** jeśli T nie ma następników.

Utrzymywanie listy wystąpień

Kluczem algorytmu jest efektywne przechowywanie informacji o dopasowaniach φ dla każdego wzorca T w drzewie danych D. Zamiast przechowywać pełną informację <φ(1), ..., φ(k)> dla φ, algorywm utrzymuje tylko częściową informację zwaną prawostronnym wystąpieniem zdefiniowanymi poniżej.

Prawostronne wystąpienie. Niech $k \geq 0$ będzie dowolną liczbą całkowitą. Niech T będzie dowolnym k-wzorcem i $\varphi: V_T \rightarrow V_D$ będzie dowolną funkcją dopasowującą z T do D. Wtedy *prawostronnym wystąpieniem* T w D pod względem φ jest węzeł Rmo(φ) = φ(k) w D na który mapuje się prawy skrajny liść k z T. Dla każdego wzorca T definiujemy RMO(T) = { Rmo(φ) φ jest funkcją dopasowującą T w D }, czyli zbiór wszystkich wystąpiń T w D. Rozpatrzmy przykładowe drzewo danych na Ilustracja 4.1. W takim przypadku wzorzec T ma 3 prawostronne wystąpienia: 4, 6 i 10 w D. Korzenie 2 i 7 mogą być bardzo łatwo obliczone przez odwołanie do rodziców węzłów 4, 6 i 10 w D.

Ilustracja 4.6 *Algorytm obliczania prawostronnych rozszerzeń i ich list wystąpień*

Algorytm Update-RMO(RMO, p, λ)

1. Przypisz do RMO_{new} listę pustą i ustaw check:= null
2. Dla każdego elemetu x ∈ RMO powtarzaj:

    a. jeżeli p = 0, przypisz do y skrajnie lewe dziecko x.

    b. W przeciwnym wypadku, gdy p ≥ 1:

    - jeśli check = $\pi_D{}^p(x)$ to pomiń x i wykonaj kontynuuj iterację po elementach RMO (wykrywanie duplikatów)
    - W przeciwnym wypadku przypisz do y kolejnego sąsiada wezła $\pi_D{}^{p-1}(x)$ ((p-1)-wszy rodzic x w D) i ustaw check:= $\pi_D{}^p(x)$.

    c. Tak długo jak y ≠ null wykonuj:

    - jeśli $L_D(y) = \lambda$, to RMO_{new}:= $RMO_{new} \cup y$;
    - y:= y.nextSibling();

3. Wynik: RMO_{new}

Ilustracja 4.6 przedstawia dokładny przebieg algorytmu obliczania kolejnych prawostronnych wystąpień. Algorytm ten jest wywoływany kolejno dla każdej etykiety λ ∈ Λ i każdego poziomu p. Celem jest obliczenie nowego prawostronnego k-rozszerzenia na podstawie (k-1) rozszerzenia. W pkt 1 inicjalizowana wartością pustą jest lista wynikowa RMO_{new} . Inicjalizowana jest także zmienna check, którma odpowiada za unikanie powtórzeń. W pkt 2 wykonywana jest pętla po wszystkich elementach x ∈ RMO. Tam w zależności od wartości p podejmowane są odpowiednio akcje. Jeśli p = 0, oznacza to rozbudowanie wzorca wgłąb w pkt 2a, a więc zmienna y odpowiedzialna za poszukiwanie wystąpień λ w D ustawiana jest na pierwsze z lewej dziecko x. Jeśli p ≥ 0 w pkt 2b następuje sprawdzenie, czy p-ty rodzic nie został już przetworzony i stąd nie jest przypisany do zmiennej check. Jeśli taka sytuacja ma miejsce, żadne nowe wystąpienia nie zostaną dodane do listy, więc dalesze przetwarzanie tego węzła nie ma sensu. Jeśli taka sytuacja nie ma miejsca to zmienna y oznaczająca początek poszukiwań nowych wystąpień będzie zainicjalizowana pierwszym sąsiaden (p-1)-wszego rodzica x. W końcu w pkt 2c poszukiwane są wystąpienia λ poczynając od węzła y przeglądając w prawo jego wszystkich sąsiadów.

Analiza algorytmu

Wracając do obliczania zbioru kandydatów C_k Ilustracja 4.7 przedstawia algorytm Expand-Trees, który oblicza zbiór C i powiązany z nim zbiór RMO_k prawostronnych wystąpień.

Ilustracja 4.7 *Algorytm obliczania zbioru prawostronnych rozszerzeń*

Algorytm Expand-Trees(F, RMO)

1. C:= ∅; RMO_{new}:= ∅;

2. Dla każdego drzewa S ∈ F powtarzaj:

- Dla każdego (p, λ) ∈ {1, ..., d} × Λ wykonaj następujące czynności (d oznacza głębokość prawego skrajnego liścia S):
  - Oblicz (p, λ)-rozszerzenie T z S
  - $RMO_{new}(T) :=$ Update-RMO(RMO(S), p, λ);
  - C = C ∪ {T};

3. Wynik: <C, RMO_{new}>;

Czas działania algorytmu jest ograniczony przez $O(k^2bLN)$, gdzie k jest maksymalnym rozmiarem częstego wzorca, b jest maksymalnym współczynnikiem rozgałęzienia drzewa D, L = #Λ, a N jest sumą długości list prawostronnych wystąpień częstych wzorców. Dalej, *FREQT* generuje co najwyżej O(kLM) wzorców podczas obliczeń, gdzie M jest sumą rozmiarów maksymalnych σ-częstych wzorców.

Obcinanie algorytmu przez pomijanie węzłów i krawędzi

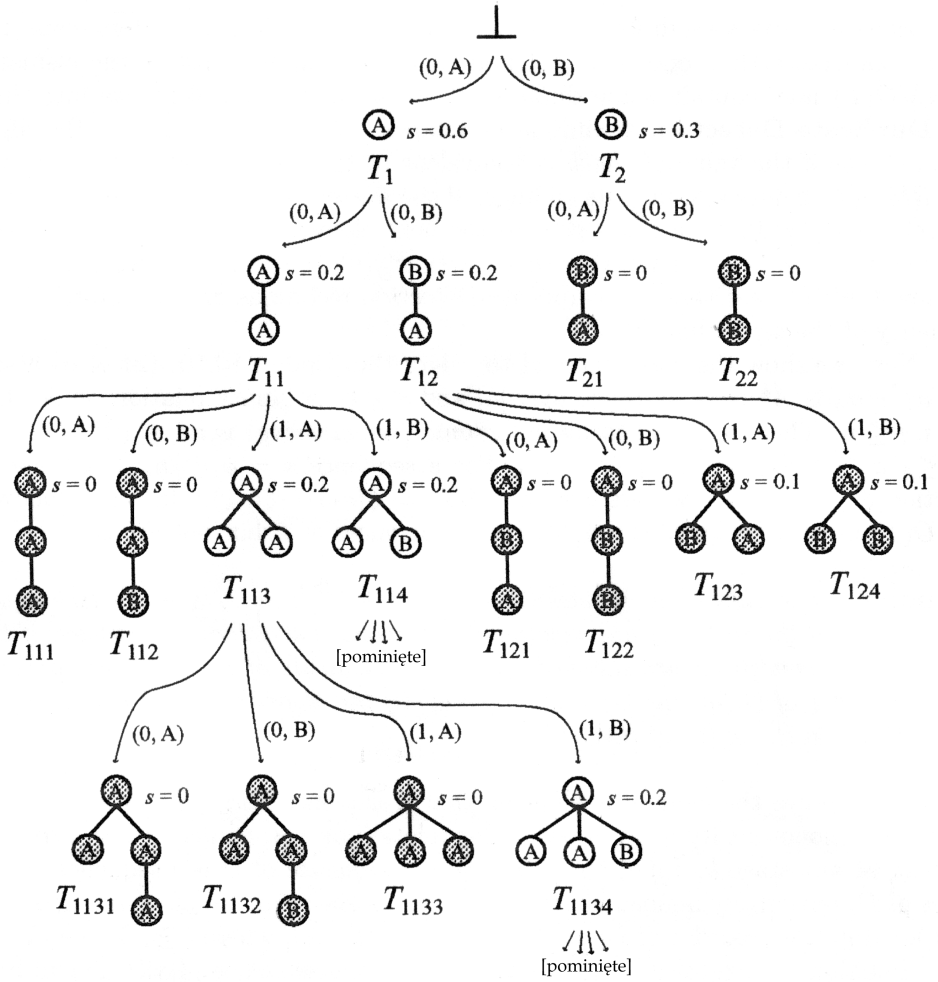
Pomijanie węzłów. Ta technika obcinania pomija bezużyteczne węzły, które mają etykiety o częstości mniejszej niż σ, przy użyciu informacji zawartej w zbiorze F_1 . Zakładając, że 1-wzorzec z etykietą λ ∈ Λ nie jest σ-częsty w D, można z całą pewnością stwierdzić, że etykieta λ nie pojawi się w żadnym σ-częstym wzorcu w F. Stąd, można pominąć wywołanie Update-RMO(RMO(T), p, λ) dla każdego wzorca T gdzie λ nie jest częstą etykietą.

Pomijanie krawędzi. Ta technika usuwa bezużyteczne krawędzie z nieczęstymi parami etykiet wykorzystując informacje z F_2 . Podobnie do pomijania węzłów, opisanego powyżej, można zauważyć, iż jesli (λ_1, λ_2) jest parą etykiet z nieczęstego 2-wzorca $E \notin F_2$, to para (λ_1, λ_2) nie wystąpi w żadnym σ-częstym wzorcu w F. Stąd można pominąć wywołanie Update-RMO(RMO(T), p, l_2), gdzie para (λ_1, λ_2) jest nieczęsta a λ_1 jest etykietą węzła $\pi_T{}^p(rml(T))$, który jest p-tym rodzicem skrajnie prawego liścia w T.

Przykład

Będziemy w tym miejscu analizować przetwarzania drzewa D widocznego na Ilustracja 4.1 o rozmiarze D = 10 przy założeniu minimalnego poparcia σ = 0,2 i Λ = {A, B}. Z wartości σ wynika, iż minimalną liczbą wystąpień korzenia dla każdego σ-częstego wzorca wynosi 2. Ilustracja 4.8 przedstawia wzorce wygenerowane przez FREQT podczas obliczeń.

Ilustracja 4.8
Drzewo poszukiwania wzorców na drzewie danych D *(Ilustracja 4.1) z minimalnym poparciem* $\sigma = 0{,}2$



Najpierw algorytm oblicza zbiór F_1 częstych 1-wzorców T_1 i T_2 przeglądając drzewo D i zapisuje ich prawe skrajne wystąpienia w RMO_1. W fazie drugiej, wywoływana jest procedura Expand-Trees od C_1 i RMO_1, dając w wyniku zbiór kandydatów C_2 i zbiór prawostronnych wystąpień RMO_2. Ilustracja 4.8 pokazuje $C_2 = \{T_{11}, T_{12}, T_{21}, T_{22}\}$, oraz to, że są one wyprowadzone poprzedników T_1 i T_2 poprzed dołączenie nowego liścia z etykietą A lub B. Lista prawostronnych wystapień dla T_{11} zawiera węzły {3, 5, 8, 9} a dla T_{12} {4, 6, 10}. Lista wystąpień korzenia jest dla obu taka sama i zawiera węzły {2, 7}, stąd T_{11} i T_{12} są σ-częste. Z drugiej strony wzorce T_{21} i T_{22} mają częstość $\sigma \leq 0{,}2$, stąd nie są zawarte w F_2.

Powtarzając ten proces w kolejnych etapach algorytm zatrzymuje się na etapie 5 i zwraca wynik $F = \{ T_1, T_2, T_{11}, T_{12}, T_{113}, T_{114}, T_{1134}\}$.

4.2 Analiza semantyczna wzorca

W tym podrozdziale omówimy pewne techniki z zakresu wyszukiwania informacji[12], które zostały przez nas wykorzystane w procesie analizy semantycznej wzorca - czyli drugiej fazy Naszego algorytmu WrapIT (patrz Sekcja 5.2.4). Przedstawimy **Model wektorowy** jako najbardziej popularny i dobrze przebadany model reprezentacji dokumentów w problematyce wyszukiwania informacji [Osiński, 03]. Następnie przedstawimy wykorzystanie miary entropii do oznaczania zestawów dokumentów zawierających przydatne informacje.

4.2.1 Model wektorowy

W systemach i algorytmach Wyszukiwania Informacji, a w szczególności tych związanych z przetwarzaniem tekstów. Model Boolowski wykorzystywał Boolowską reprezentację zapytania porównywaną z zestawem termów do identyfikacji interesującej informacji. Model probabilistyczny jest oparty na obliczaniu prawdopodobieństwa przynależności dokumentu ze zbioru wejściowego do zbioru wyjściowego zgodnego z zapytaniem. Najbardziej popularny jest jednak Model Wektorowy wykorzystujący listę termów do reprezentacji zarówno dokumentów jak i zapytania i używający algebry liniowej do obliczania podobieństwa pomiędzy nimi. Model ten jest najprostszy w użyciu spośród wszystkich tu wymienionych, jest również najwydajniejszy [Salton, 89].

W modelu wektorowym każdy dokument z kolekcji jest reprezentowany przez wielowymiarowy wektor. Każdy wymiar odpowiada w jakiś sposób powiązaniu danego dokumentu z jednym termem (pojedynczym słowem lub sentencją). Wartość w każdym wymiarze odpowiada stopniowi powiązania dokumentu z termem reprezentowanym przez ten wymiar. Stopień powiązania dokumentu z termem, zwany także częściej wagą termu w dokumencie, może być obliczany na kilka różnych sposobów. Poniżej przedstawimy najpopularniejsze wykorzystywane sposoby ważenia termów.

Ważenie termów

Ważenie termów jest procesem obliczania wag, czyli stopnia przynależności termu do dokumentu. W Modelu Wektorowym wartość taka jest pojedynczą liczbą. Niech $a_{i,j}$ oznacza wagę termu i w dokumencie j.

Ważenie binarne

Najprostszą możliwą wagą jest waga binarna. $a_{i,j} = 1$ w przypadku gdy term i występuje w dokumencie j, podczas gdy $a_{i,j} = 0$ w przeciwnym wypadku. Taka miara informuje jednak tylko o samym istnieniu jakiegoś związku między dokumentem a termem, jednak nie niesie żadnej informacji na temat siły tego związku.

[12] ang. Information Retrieval

Ważenie częstością termów

Bardziej złożonym modelem jest model wykorzystujący **częstość termów**. W tej technice $a_{i,j} = Tf_{i,j}$, gdzie $Tf_{i,j}$ określa ile razy term i występuje w dokumencie j. Łatwo zauważyć, iż częstość występowania termów niesie więcej informacji, niż proste ważenie binarne. Podejście takie ma jednak pewną wadę. Wyobraźmy sobie sytuację, że prawie wszystkie dokumenty z kolekcji zawierają pewne słowo, np. "zeszyt". Jakkolwiek słowo "zeszyt" jest z pewnością bardzo istotnym składnikiem każdego dokumentu, staje się bezużyteczne jeśli chcielibyśmy go użyć do rozróżnienia poszczególnych dokumentów. Stąd wniosek, że term "zeszyt" w ujęciu globalnym jest mało wartościowy, gdyż występuję w prawie każdym dokumencie. Nasuwa się stąd wniosek, iż ważenie częstością termów skupia się jedynie na lokalnym znaczeniu poszczególnych termów w odniesieniu do dokumentów.

Ważenie Tf-idf

Ważenie Tf-idf (Częstość termów - Odwrotna częstość w dokumentach) ma na celu wyeliminowanie wad ważenia częstością termów przez odpowiednie wyważenie znaczenia lokalnego termu i jego znaczenia w kontekście pełnej kolekcji dokumentów. W tym schemacie $a_{i,j} = tf_{i,j} \times \log(N/df_i)$, gdzie $tf_{i,j}$ jest częstością termu, df_i jest liczbą dokumentów w których term i występuje natomiast N oznacza całkowitą liczbę dokumentów w kolekcji. Czynnik $\log(N/df_i)$, oznaczany często idf (odwrotna częstość w dokumentach[13]) koryguje wartość częstości termu i w odniesieniu do jego globalnego znaczenia w całej kolekcji dokumentów. W przypadku gdy jakiś term występuje we wszystkich dokumentach, czyli $df_i = N$ cały czynnik idf przyjmie wartość 0 odzwierciedlając tym samym brak wartości danego termu przy rozróżnianiu dokumentów z kolekcji.

Ilustracja 4.9
Przykład konstrukcji macierzy term-dokument

Kolekcja dokumentów:
D1: Zautomatyzowane pozyskiwanie danych
D2: Pozyskiwanie wiedzy z baz danych
D3: Eksploracja baz danych
D4: Zautomatyzowane pozyskiwanie wiedzy

Termy:
T1: zautomatyzowane
T2: pozyskiwanie
T3: danych
T4: wiedzy
T5: baz
T6: eksploracja

Macierz term – dokument dla 4 dokumentów i 6 termów przy ważeniu Boolowskim

$$A = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 1.0 \\ 1.0 & 1.0 & 0.0 & 1.0 \\ 1.0 & 1.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 1.0 \\ 0.0 & 1.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \end{bmatrix}$$

Macierz term – dokument dla 4 dokumentów i 6 termów przy ważeniu Tf-idf

$$A = \begin{bmatrix} 0.3 & 0.0 & 0.0 & 0.3 \\ 0.13 & 0.13 & 0.0 & 0.13 \\ 0.13 & 0.13 & 0.13 & 0.0 \\ 0.0 & 0.6 & 0.0 & 1.0 \\ 0.0 & 0.3 & 0.3 & 0.0 \\ 0.0 & 0.0 & 0.6 & 0.0 \end{bmatrix}$$

Macierz term-dokument

W modelu wektorowym kolekcja d dokumentów opisanych przy pomocy t termów może być opisana przy pomocy macierzy d×t, zwanej **macierzą term-dokument**. Każdy z elementów $a_{i,j}$ macierzy określa przynależność termu i do dokumentu j zgodnie z przyjętym

[13] ang. Inverse Term Frequency

schematem ważenia termów. Kolumny macierzy, zwane **wektorami dokumentów**, reprezentują dokumenty z kolekcji, podczas gdy wiersze, zwane **wektorami termów**, wykorzystywane są do indeksowania kolekcji dokumentów. Przykład konstrukcji takiej macierzy przedstawia Ilustracja 4.9.Taka reprezentacja jest bardzo pomocna przy zapytaniach, gdyż traktując zapytanie tak samo jak dokument można bardzo łatwo obliczyć podobieństwo dokumentu do zapytania przy wykorzystaniu zwykłych zależności geometrycznych między dwoma wektorami.

4.2.2 Wykrywanie informatywnych bloków danych

W analizie wzorca algorytmu WrapIT bardzo istotnym elementem jest określenie fragmentów wzorca zawierających istotne informacje. Dla potrzeb naszego algorytmu interesujące są bloki o dużej zmienności zawartości. Bloki, które we wszystkich wystąpieniach mają taką samą lub podobną zawartość z pewnością nie będą interesujące. W [Lin & Ho, 02] zostało zaprezentowane podejście oparte na entropii poszczególnych bloków danych. W tym podrozdziale postaramy się przedstawić to podejście w kontekście wykorzystania go w algorytmie WrapIT.

Obliczanie entropii dla termów

Blokiem danych będziemy nazywać pewien logiczny fragment dokumentu, występujący w każdym dokumencie z kolekcji (np. abstrakt artykułu). **Wystąpienie bloku danych** będziemy natomiast nazywać zawartość danego bloku w konkretnym dokumencie. Wartość entropii termów jest określana zgodnie z rozkładem wag termu w kolekcji dokumentów. Dla łatwego obliczenia entropii każdego termu, termy poszczególnych bloków danych mogą być zgrupowane i reprezentowane jako macierz term-dokument z metodą ważenia częstością termów lub też przy wykorzystaniu miary Tf-idf. Dla każdego bloku danych należy wygenerować jego własną macierz term-dokument na podstawie wystąpień tego bloku w kolekcji dokumentów.

Bazując na macierzy term-dokument obliczenie wartości entropii dla termu odpowiada obliczeniu rozkładu prawdopodobieństwa w wierszu macierzy. Poniżej przedstawiony jest miara nieokreśloności wg. Shannona [Shannon, 48].

Równanie 4.1 *Miara nieokreśloności wg. Shannona*

$$0 - \sum_{i=1}^{n} \log p_i \le \log n \text{, gdzie } p_i \text{ jest prawdopodobieństwem } zdarzenia_i$$

Po znormalizowaniu wag termu do wartości [0, 1], entropia termu ma wartość następującą:

$$H(F_i) = -\sum_{j=1}^{n} w_{ij} \log_2 w_{ij} \text{ , gdzie } w_{ij} \text{ jest wagą } F_i \text{ w dokumencie } D_i$$

Aby znormalizować wartość entropii do przedziału [0, 1], zmieniamy podstawę logarytmu na wartość równą liczbie dokumentów w kolekcji.

Równanie 4.2 *Entropia termu t_i*

$$0 \le H(F_i) = -\sum_{j=1}^{n} w_{ij} \log_d w_{ij} \le 1 \text{, gdzie } d \text{ jest liczbą dokumentów } (d = |D|)$$

Rozpatrzymy teraz przykład dwóch dokumentów D_1 i D_2, oraz dwóch bloków danych B_1 i B_2. Poszczególne dokumenty zawierają następujące termy:

$B_1(D_1) = T_1$, $B_2(D_1) = T_2$

$B_1(D_2) = T_1$, $B_2(D_2) = T_3$

Wartości entropii dla poszczególnych termów:

$H(T_1) = -½ \log_2 ½ = 1$

$H(T_2) = H(T_3) = -1 \log_2 1 - 0 \log_2 0 = 0$

Obliczanie entropii dla bloków danych

Entropia całego bloku danych naturalnie oparta jest na wartościach entropii wszytkich termów występujących w tym bloku danych. Entropia bloku danych jest więc sumą entropii termów tego bloku danych:

$$H(B_i) = \sum_{j=1}^{n} H(T_j)$$, gdzie T_j jest i-tym termem bloku o n termach

Ponieważ różne bloki danych składają sie z różnej liczby termów, należy podane powyżej równanie znormalizować, tak aby entropia bloku danych zawierała się w zakresie [0, 1].

Równanie 4.3 *Znormalizowana wartość entropii dla bloku danych B_i*

$$H(B_i) = \frac{\sum_{j=1}^{n} H(T_j)}{n}$$

Klasyfikacja bloków danych

Bazując na wartości $H(B_i)$ bloki danych możemy podzielić na dwie kategorie - informatywne i redundantne:

- Jeśli wartość $H(B_i)$ jest wyższa od pewnego założonego progu, lub też bliska 1, oznacza to iż blok B_i jest redundantny, co oznacza, iż większość jego tokenów pojawia się w każdym dokumencie i nie niesie żadnej istotnej informacji.

  Jeśli wartość $H(B_i)$ jest niższa od pewnego założonego progu blok B_i jest informatywny i poszczególne wystąpienia tego bloku różnią się znacząco między sobą. Dokładniej tokeny występujące w tym bloku pojawiają się rzadko w wystąpieniach tego bloku.

Istotnym czynnikiem wpływającym na dobre rozróżnienie bloków danych jest dobry dobór progu odcięcia. Wyższa wartość spowoduje uznanie większej liczby bloków za informatywne, mogą się wśród nich pojawić jednak te nie informatywne. Natomiast niższa wartość spowoduje większą precyzję (mniejszą liczbę błędów), jednak istnieje ryzyko, iż informatywny blok zostanie niepoprawnie zakwalifikowany jako redundantny. Dobrą wartość tego progu należy wyznaczyć eksperymentalnie aby dostosować go do konkretnego zastosowania.

5

WrapIT - Opis algorytmu

5.1 Założenia

WrapIT jest algorytmem, którego celem jest przeprowadzenie analizy strony wynikowej wyszukiwarki internetowej i wydobycie informacji o strukturze tesj strony niezbędnej do późniejszego szybkiego odpytywania tej wyszukiwarki i pobieraniu zestawu snippetów.

Wejściem dla algorytmu są podstawowe parametry potrzebne do odpytania wyszukiwarki w celu uzyskania wyników (patrz Ilustracja 5.1). Algorytm sam nie wykrywa formularzy na stronie wyszukiwarki internetowej, stąd wszystkie parametry wykorzystywane przy zapytaniu muszą być podane jako wejście algorytmu. Ponieważ praktycznie wszystkie serwisy przy zatwierdzaniu formularza zapytania wykorzystują metodę HTTP GET, stąd też parametry są widoczne w adresie URL przeglądarki internetowej. Dodatkowo parametry te niezwykle rzadko ulegają zmianie, stąd nie wymagają kosztownego utrzymywania. Algorytm swoje funkcjonowanie, co zostanie opisane w dalszej części, opiera na wiedzy na temat liczby snippetów na pojedynczej stronie wyszukiwawczej. W dużej liczbie serwisów wyszukujących liczbę snippetów można kontrolować poprzez parametry HTTP GET, stąd też ustalenie tego parametru nie wymaga żadnych dodatkowych czynności.

Ilustracja 5.1 *Parametry wejściowe algorytmu WrapIT*

- Adres URL serwisu wyszukiwawczego
- Parametry potrzebne do prawidłowego odpytania serwisu wyszukującego
    - Parametr HTTP GET zawierający zapytanie do wyszukiwarki
    - Parametr HTTP GET pozwalający określić kolejne strony zapytania
    - Inne parametry HTTP GET wymagane przez wyszukiwarkę
- Liczba snippetów na pojedynczej stronie wynikowej

Podstawowym założeniem poczynionym przy projektowaniu tego algorytmu jest fakt, iż strona wynikowa serwisu wyszukującego w swojej głównej części zawiera listę następujących po sobie wyników. Snippety wynikowe stanowią w większości wyszukiwarek od 50-80% powierzchni strony. Ponieważ wyniki generowane są automatycznie, naturalne jest, iż każdy pojedynczy snippet jest oparty na podobnym wzorcu HTML. Stąd też wniosek,

iż 50-80% strony wynikowej zawierają powtarzające się wzorce HTML z wynikami wyszukiwania. Pozostałą część źródła strony stanową w większości elementy statyczne:

- Nagłówek HTML
- Nagłówek strony zawierający zazwyczaj logo wyszukiwarki
- Formularz wyszukiwania umożliwiający zadawanie kolejnych zapytań
- Opis aktualnie wyświetlonego zapytania
- Nawigator umożliwiający poruszanie się między stronami wynikowymi
- Stopkę strony zawierającą zazwyczaj informacje o prawach autorskich itp.

Dodatkowo coraz częściej w wyszukiwarkach pojawiają się także generowane automatycznie reklamy odpowiadające zadanemu zapytaniu. Nie są one zazwyczaj zbyt liczne, i nie występują nigdy w liczbie porównywalnej z liczbą wyników zapytania. Stąd też widać wyraźnie, iż na stronie wynikowej należy poszukiwać powtarzających się często struktur HTML w liczbie równej spodziewanej (określonej w parametrach wejściowych) liczbie snippetów.

Ilustracja 5.2 *Przykładowy deskryptor snippetu dla wyszukiwarki Google*

```
<snippet>
 <p>
  <a address="attribute:href" title="inside"></a>
  <br description="beginafter"/>
  <font>
   <br />
   <br optional="true" />
   <br optional="true" />
   <font description="endbefore" />
   <a />
  </font>
 </p>
</snippet>
```

Oczekiwanym wyjściem algorytmu jest *deskryptor snippetu*, czyli struktura XML opisująca pojedynczy snippet przetwarzanego serwisu wyszukującego. Przykład takiego deskryptora ilustruje Ilustracja 5.2. Struktura tagów wewnątrz znacznika <*snippet*> przedstawia strukturę tagów HTML w poszukiwanym snippecie. Obejmuje także struktury opcjonalne, czyli takie, które nie muszą występować w każdym snippecie, ale zawierają cenną i pożądaną informację. W przykładowym deskryptorze takimi znacznikami są znaczniki <*BR*> posiadające atrybut *optional* o wartości *true*. Wszystkie pozostałe znaczniki są wymagane, tzn. muszą wystąpić w każdym pojedynczym snippecie. Deskryptor opisuje także oczywiście zawartość semantyczną snippetu, czyli oznaczenia elementów, które zawierają poszukiwaną informację. Wyróżnione elementy informacyjne to *tytuł*, *opis* oraz *adres URL*. Poszczególne elementy mogą znajdować się w różnych miejscach snippetu. Pierwsza możliwość jest taka, iż poszukiwane dane znajdują się w całości wewnątrz pojedynczego znacznika. W przykładowym deskryptorze takim znacznikiem jest <*a title*="*inside*">, co oznacza, iż tytuł

snippetu znajduje się w całości wewnątrz tagu <*a*>. Poszukiwana wartość może także się znajdować w jednym z atrybutów znacznika. Najczęściej taka sytuacja dotyczy adresu URL, który zazwyczaj jest aktrybutem *href* tagu <*a*>. Na naszym przykładzie przedstawiona jest dokładnie taka sytuacja: <*a address*="*attribute:href*">. Ostatni dopuszczalny przez nas przypadek jest taki, iż wartość znajduje się pomiędzy dwoma wyszczególnionymi znacznikami. W takiej sytuacji tag początkowy jest oznaczony atrybutem o wartości "*beginAfter*", natomiast tag końcowy zawiera taki sam atrybut o wartości "*endBefore*". Na przykładzie w ten sposób jest wydobywany opis snippetu (atrybut *description*). Deskryptor w takiej postaci może być bezpośrednio użyty do implementacji wrappera danego serwisu wyszukującego.

5.2 Algorytm

Algorytm WrapIT składa się z kilku elementarnych faz, które przekształcają pobraną stronę wyników wyszukiwania w formacie HTML w pełny deskryptor XML opisujący strukturę HTML jak i semantykę pojedynczego snippetu. Krótka charakterystyka poszczególnych faz wygląda następująco:

- *Budowa drzewa rozkładu HTML* - w tej fazie czysto tekstowy dokument HTML zostaje przekształcony do reprezentacji drzewa rozkładu HTML opisanego wcześniej (Zobacz: Sekcja 4.1.1). Reprezentacja ta pozwala na wygodną analizę struktury wewnętrznej dokumentu oraz na wykorzystanie algorytmów operujących na drzewach do późniejszej analizy. Jest to faza przygotowująca dokument do dalszego przetwarzania.

- *Odnalezienie wzorca podstawowego* - na tym etapie rozpoczyna się właściwe przetwarzanie strony wynikowej. Przy wykorzystaniu zmodyfikowanej wersji algorytmu *FREQT* (Zobacz: Sekcja 4.1.2) w strukturze drzewa HTML zostaje odnaleziony maksymalny powtarzający się w każdym snippecie wzorzec, który najprawdopodobniej odpowiada pojedynczemu snippetowi. Taki wzorzec jest nazywany **wzorcem podstawowym**.

- *Rozszerzenie wzorca* - w tej fazie wzorzec podstawowy zostaje rozszerzony o węzły opcjonalne. Zrealizowane jest to przez zmniejszenie wymaganego poparcia dla wzorca przy jednoczesnym znacznym ograniczeniu zakresu poszukiwań wewnątrz wzorca podstawowego.

- *Analiza semantyczna wzorca* - Uzyskany w poprzednich etapach wzorzec zostaje przeanalizowany pod kątem potencjalnych miejsc wystąpień poszukiwanych danych. W tym celu dla wszystkich "podejrzanych" miejsc zostają wyliczone odpowiednie miary pozwalające ocenić znaczenie poszczególnych fragmentów znalezionego wzorca.

- *Czyszczenie wzorca* - Niestety wydedukowany w poprzednich fazach wzorzec może zawierać dość liczne elementy formatujące, które nie mają wiele wspólnego z poszukiwanymi danymi, a jedynie zaburzają spójność struktury snippetu. Ponieważ jednak nie wszystkie węzły formatujące są nieistotne, eliminacja wykorzystuje miary informacji dla zawartości poszczególnych znaczników, umożliwiając ocenę przydatności danego węzłą we wzorcu.

5.2.1 Budowa drzewa rozkładu HTML

Strona wynikowa serwisu wyszukującego jest dokumentem HTML. Dokument taki jest dokumentem tekstowym, posiada jednak wewnętrzną strukturę drzewa zdefiniowaną przez znaczniki (patrz Sekcja 4.1). Dla algorytmu budującego drzewo rozkładu HTML bardzo istotną rzeczą jest odporność na błędy, które bardzo licznie pojawiają się w analizowanych stronach. Przyczyna takiego stanu rzeczy leży niestety po stronie producentów przeglądarek internetowych, którzy uczynili swoje produkty bardzo "pobłażliwe" w kwestii zgodności wyświetlanych stron ze standardem. Stąd też bardzo duża liczba stron internetowych nie jest w pełni zgodna ze specyfikacją języka HTML. Sam format dokumentu dopuszcza tzw. niedomknięte znaczniki, które tak naprawdę utrudniają automatyczną budowę drzewa rozkładu, gdyż tak naprawdę czasem trudno jest jednoznacznie stwierdzić gdzie dany niedomknięty tag powinien zostać sztucznie domknięty. Dodatkowo istnieje pewna grupa znaczników (jak np. <*BR*>, czy <*HR*>), które mogą występować jedynie jako liście drzewa rozkładu i także nie należy oczekiwać ich domknięcia. Algorytm wykorzystuje dwie listy StandAlone i NotClosed zawierające odpowiednio znaczniki występujące jako liście i znaczniki, których domknięcie jest opcjonalne.

Ilustracja 5.3 *Algorytm budowy drzewa rozkładu HTML*

Algorytm DrzewoHTML(D)

1. Dla dokumentu D zbuduj listę występujących w nim kolejno tagów L.
2. Utwórz sztuczny węzeł R, którego dziećmi będą wszystkie węzły analizowanego dokumentu.
3. Wywołaj BudujWęzełHTML(R, L)
4. R jest drzewem rozkładu HTML dla D.

Funkcja BudujWęzełHTML(W, L)

1. Inicjuj listę niedomkniętych tagów NCT = ∅.
2. Tak długo jak L ≠ ∅ powtarzaj:
    - Pobierz i usuń z L aktualny znacznik do analizy: T = L[0]; L = L - T;.
    - Jeśli T jest domknięciem i Label(T) = Label(W) to zakończ funkcję.
    - Jeśli T jest domknięciem Lebel(T) ≠ Label(W) to zignoruj.
    - Jeśli jest znacznikiem otwierającym oraz T ∈ StandAlone lub jest ciągiem tekstowym to dodaj węzeł T do dzieci W.

- Jeśli T jest znacznikiem otwierającym i T ∈ NotClosed to:
  - Jeśli Label(T) ∈ NCT to N = NCT(Label(T)), NCT = NCT - N

    Przypisz wszystkie dzieci W starsze od N jako dzieci N.
  - NCT = NCT + T

    Dodaj T do dzieci W
- Jeśli T jest znacznikiem otwierającym oraz T ∉ NotClosed i T ∉ StandAlone dodajemy T jako dziecko W, a następnie wywołujemy BudujWęzełHTML(T, L).

Główną częścią algorytmu jest funkcja rekurencyjna BudujWęzełHTML(W, L) , która ma za zadanie odczytanie z listy znaczników L wszystkich swoich dzieci. Funkcja kończy się, gdy cała lista L zostanie wykorzystana, lub też gdy zostanie napotkany odpowiedni znacznik zamykający W. Analizując kolejne znaczniki są one traktowane na 3 sposoby. Znaczniki należące do listy StandAlone są od razu dodawane do listy dzieci W. Znaczniki należące do listy NotClosed są także dodawane jako dzieci W, lecz ich wystąpienie jest zapamiętywane, i w przypadku kolejnego wystąpienia wszystkie dzieci W od poprzedniego wystąpienia są przenoszone jako dzieci poprzedniego wystąpienia znacznika. Trzecia grupa znaczników, to są znaczniki strukturalne, które są także dodawane jako dzieci W, a następnie jest dla nich rekurencyjnie wywoływana funkcja BudujWęzełHTML.

Algorytm ten buduje drzewo HTML na podstawie dokumentu w czasie O(L). Każdy węzeł jest tylko jednokrotnie wyjmowany z listy L i analizowany przez funkcję BudujWęzełHTML. Dodatkowo każdy węzeł może jednokrotnie zmienić rodzica w procesie przetwarzania znaczników należących do listy NotClosed. Widać więc, że zbudowanie całego drzewa odbywa się w czasie liniowym.

5.2.2 Poszukiwanie wzorca podstawowego

W celu Odnalezienia wzorca opisującego snippet algorytm WrapIT musi odnaleźć struktury HTML tworzone dla każdego snippetu przez serwis wyszukiwawczy. Każdy snippet zawiera w ogólności te same elementy i ma podobne formatowanie. Wynika stąd, że wyszukiwarka używa do generacji każdego snippetu podobnych struktur HTML. Podstawowym celem algorytmu jest więc odnalezienie takich powtarzających się struktur w drzewie rozkładu HTML. Snippety na stronie wynikowej są zazwyczaj w pewien sposób zorganizowane poprzez użycie znaczników formatujących HTML. Najczęściej wykorzystywane są znaczniki listy (<*li*>), tabele (<*TD*>), regiony (<*DIV*>, <*SPAN*>), paragrafy (<*P*>). Stąd wynika wniosek, iż każdy snippet jest zawarty w jednym poddrzewie drzewa rozkładu strony i posiada tylko jeden korzeń. Oczywiście może się zdarzyć, iż administrator serwisu wyszukującego będzie w zupełnie inny sposób formatował snippety i nie będą one zawarte w pojedynczych poddrzewach, lecz każdy snippet będzie zawarty w więcej niż jednym poddrzewie. W większości przypadków jednak nawet wtedy można

wyróżnić dominujące powtarzające się poddrzewo w snippecie, tzn. takie, które zawiera większą liczbę znaczników i powtarza się w dokładnie każdym snippecie. Takie właśnie poddrzewo (lub też całe drzewo w przypadku wyszukiwarek organizujących snippety przez tagi formatujące) będziemy nazywać **wzorcem podstawowym**.

Ilustracja 5.4 *Przykład powtarzającej się struktury z zaburzeniami*

```
<p>
 <a><b>linia</b></a>
 <i><font>linia2</font></i>
</p>
<p>
 <a><b>linia</b></a>
 <font>linia2</font>
</p>
```

Dopasowanie wzorca do drzewa zostało opisane już wcześniej (zobacz Sekcja 4.1.2.2). W naszym zastosowaniu musimy jednak zmodyfikować funkcję dopasowującą φ. Patrząc na Ilustracja 5.4 można od razu zauważyć, iż powtarzający się wzorcem jest:

```
<p>
 <a><b>#tekst</b></a>
 <font>#tekst</font>
</p>
```

Wzorzec ten jednak jest niezgodny z definicją funkcji dopasowującej φ dla algorytmu FREQT, ponieważ znacznik <font> nie jest w pierwszym poddrzewie przykładu dzieckiem <p>, gdyż występuje po drodze znacznik <i>. Znacznik <i> w tym przypadku nazwiemy **zaburzeniem**, gdyż jest to element który nie powtarza się w każdym wystąpieniu wzorca, lecz zaburza relację rodzicielstwa między elementami z drzewa przykładowego. W analizie drzewa HTML takie zaburzenia będą niezwykle częste (np. bardzo licznie zdarzają się dodatkowe formatowania słów kluczowych), więc algorytm poszukiwania, jak i sama definicja wzorca powinny być odporne na takie zaburzenia.

Aby zdefiniować nową funkcję dopasowującą φ' musimy posłużyć się nową relacją rodzicielstwa i sąsiedztwa przechodniego. Węzeł v jest w sensie przechodnim dzieckiem u, jeśli istnieje ścieżka z węzła u do węzła v. Węzeł v jest w sensie przechodnim sąsiadem węzła u wtedy i tylko wtedy, gdy istnieją tacy przodkowie odpowiednio przodek(v) i przodek(v), tacy, że przodek(v) jest sąsiadem przodek(u). Nowa funkcja dopasowująca φ' musi spełniać następujące warunki:

- φ' zachowuje relację rodzicielstwa przechodniego.
- φ' zachowuje relację sąsiedztwa przechodniego.
- φ' zachowuje etykiety.

Dzięki takiej definicji funkcji dopasowującej możliwa jest konstrukcja wzorca prawidłowo opisującego maksymalne powtarzające się struktury w drzewie z zaburzeniami.

Odnalezienie wzorca podstawowego snippetu jest równoznaczne z odnalezieniem maksymalnego wzorca dopasowującego się do wejściowego drzewa rozkładu HTML przy pomocy pewnej funkcji φ'. Poszukiwać będziemy maksymalnego wzorca, ponieważ interesuje nas tylko wzorzec pokrywający cały snippet (lub jego dominujące poddrzewo), a nie interesują nas żadne mniejsze wzorce, które zawierają się we wzorcu maksymalnym. W naszym zastosowaniu zmodyfikujemy także definicję poparcia. Znając na wejściu liczbę otrzymywanych snippetów możemy zdefiniować poparcie jako stosunek wystąpień korzenia analizowanego wzorca do liczby snippetów na stronie: freq'(T) = Root(T) / #Snippets. Mając tak zdefiniowaną miarę poparcia i założenie, iż wzorzec podstawowy powtarza się w *każdym* snippecie, możemy ustawić minimalny próg poparcia na 100%, czyli σ' = 1.

Jak widać różnice w założeniach pomiędzy algorytmem FREQT i poszukiwaniem wzorca podstawowego wykluczają bezpośrednie wykorzystanie tego algorytmu. Stanie się on jednak podstawą do stworzenia nowego algorytmu działającego zgodnie z przyjętymi założeniami. W tym celu należy zlokalizować elementy algorytmu FREQT, które są w sprzeczności z nowymi założeniami i zaproponować nowe rozwiązania. Najważniejsza zmiana dokonała się w definicji funkcji dopasowującej. Funkcja ta jest podstawą algorytmu Update-RMO (Zobacz Ilustracja 4.6. Algorytm ten jest niestety silnie związany z funkcją dopasowującą φ i nie ma możliwości jego bezpośredniego wykorzystania. Zaproponujemy w takim razie nowy algorytm Update-RMO'(RMO, p, λ), który będzie odpowiedzialny za generację listy prawostronnych wystąpień dla danego prawostronnego rozszerzenia wzorca T, jego listy prawostronnych wystąpień RMO oraz etykiety $\lambda \in \Lambda$.

Ilustracja 5.5 *Algorytm obliczania prawostronnych rozszerzeń i ich list wystąpień w algorytmie WrapIT*

Algorytm Update-RMO'(RMO, p, λ)

1. Przypisz do RMO_{new} listę pustą
2. Dla każdego elementu $x \in RMO$ powtarzaj:
   a. jeżeli p = 0, przypisz do y skrajnie lewe dziecko x.
   b. W przeciwnym wypadku, gdy $p \geq 1$ przypisz do y kolejnego sąsiada wezła $\pi_D{}^{p-1}(x)$ ((p-1)-wszy rodzic x w D).
   c. Tak długo jak $y \neq null$ wykonuj:
      - jeśli $L_D(y) = \lambda$ oraz $y \notin RMO_{new}$, to $RMO_{new} := RMO_{new} \cup y$;
      - y:= y.nextExtendedSibling();
3. Wynik: RMO_{new}

Ilustracja 5.5 przedstawia nowy algorytm generowania listy prawostronnych wystąpień dostosowany do założeń algorytmu WrapIT. Porównując ten algorytm z wersją oryginalną FREQT można zauważyć, że niestety trzeba było zrezygnować z rozwiązania automatycznie wykrywającego duplikaty w pkt 2b (zmienna *check*), ponieważ mogło ono doprowadzić do pominięcia niektórych sąsiadów w sensie rozszerzonym, gdyby (jak w oryginalnym algorytmie Update-RMO) listę dzieci danego węzła przeglądać tylko jednokrotnie. Wynika

stąd bezpośrednio druga zmiana w pkt 2c. W wersji oryginalnej mając zagwarantowane tylko jednokrotne przejrzenie listy sąsiadów można było pominąć sprawdzanie, czy analizowany element y nie znajduje się już na liście wynikowej RMO_{new} . W naszym dostosowanym algorytmie Update-RMO' takie sprawdzenie jest niestety niezbędne i powoduje zwiększenie złożoności obliczeniowej algorytmu.

Ilustracja 5.6 *Algorytm obliczania zbioru prawostronnych rozszerzeń dla algorytmu WrapIT*

Algorytm Expand-Trees'(F, RMO)

1. $C := \varnothing$; $RMO_{new} := \varnothing$;
2. Dla każdego drzewa $S \in F$ powtarzaj:
   - Dla każdego $(p, \lambda) \in \{1, ..., d\} \times \Lambda$ wykonaj następujące czynności (d oznacza głębokość prawego skrajnego liścia S):
     - Jeżeli $\lambda_S \geq \lambda_D$ / #Snippets to pomiń λ.
     - Oblicz (p, λ)-rozszerzenie T z S
     - $RMO_{new}(T) :=$ Update-RMO(RMO(S), p, λ);
     - $C = C \cup \{T\}$;
3. Wynik: $<C, RMO_{new}>$;

W naszym algorytmie zostały wprowadzone także pewne optymalizacje do funkcji Expand-Trees. Ilustracja 5.6 przedstawia zmodyfikowaną funkcję Expand-Trees'. Dodane zostało kolejne ograniczenie na przetwarzanie konkretnych etykiet $\lambda \in \Lambda$. Wynika ono z założenia o 100% poparciu dla wzorca snippetu. Implikuje ono wystąpienie każdej etykiety co najwyżej λ_D / #Snippets razy we wzorcu. Modyfikacja ta pozwala na pominięcie w analizie etykiet, które nie mogą już więcej wystąpić. Jest to pewne rozszerzenie obcinania nieczęstych węzłów.

Algorytm FREQT poszukiwał wszystkich wzorców spełniających kryterium częstości w drzewie danych D. W przypadku algorytmu WrapIT interesuje nas jedynie maksymalny wzorzec, czyli taki, który zawiera największą liczbę węzłów. Nie ma więc potrzeby przechowywania w czasie całego przetwarzania całego zbioru wzorców spełniających kryterium 100% częstości, a jedynie bieżącego maksymalnego wzorca. W momencie wygenerowania nowego wzorca o odpowiednim poparciu zostaje on automatycznie przypisany jako wzorzec maksymalny, ponieważ algorytm generuje wzorce o wzrastających rozmiarach bez nawrotów.

W tym podrozdziale przedstawiliśmy metodę adaptacji algorytmu FREQT poszukującego częste wzorce w uporządkowanym drzewie etykietowanym do poszukiwania wzorca podstawowego snippetu w drzewie rozkładu HTML. Algorytm ten zachowuje właściwości

dotyczące wydajności algorytmu FREQT, jedyne zwiększenie złożoności obliczeniowej nastąpiło w funkcji Update-RMO'.

5.2.3 Rozszerzenie wzorca

Jak wspomnieliśmy w poprzedniej sekcji niektóre serwisy wyszukiwawcze konstruują snippety bez wykorzystania znaczników formatujących, które stanowiłyby korzeń dla poddrzewa zawierającego cały snippet. W takim przypadku algorytm poszukujący wzorca podstawowego odnajdzie jedynie poddrzewo dominujące w snippecie, a nie snippet w całości. W takim przypadku niezbędne jest dalsze przetwarzanie mające na celu odnalezienie kolejnych fragmentów wzorca snippetu.

Ilustracja 5.7 *Przykładowy dokument HTML z rozszerzonym wzorcem*

```
<p>test</p>
<a>
 <font>test</font>
 <b>tekst</b>
</a>
<i>test</i>
<p>test</p>
<a>
 <font>test</font>
 <b><i>tekst</i></b>
</a>
<i>test</i>
```

Do wygenerowania rozszerzonego wzorca wykorzystywany jest wynik poprzedniej fazy przetwarzania - wzorzec maksymalny. Do wzorca dodawany jest sztuczny węzeł-korzeń reprezentujący cały dokument. W ten sposób możliwe jest dodawanie do wzorca wezłów sąsiednich do korzenia wzorca podstawowego. Następnie wykorzystywany jest ten sam zmodyfikowany algorytm FREQT opisany w poprzedniej sekcji. Pozwala to na wygenerowanie wzorca składającego się z wielu sąsiednich poddrzew. Ilustracja 5.7 przedstawia przykładowy dokument HTML z wzorcem składającym się z wielu poddrzew. W pierwszej fazie zostanie wygenerowany następujący wzorzec podstawowy:

```
<a>
 <font>#tekst</font>
 <b>#tekst</b>
</a>
```

Przyglądając się uważnie danemu dokumentowi widzimy, iż każdy snippet powinien także zawierać dodatkowe znaczniki #tekst. Po dodaniu sztucznego węzła-korzenia i ponowieniu przetwarzania z fazy pierwszej otrzymujemy w wyniku wzorzec:

```
<SZTUCZNY KORZEN>
 <a>
  <font>#tekst</font>
  <b>#tekst</b>
 </a>
```

```
 <i>#tekst</i>
</SZTUCZNY KORZEN>
```

We wzorcu brakuje jednak fragmentu #tekst , który pojawia się przed każdym jego wystąpieniem. Niestety technika prawostronnego rozszerzania wzorca użyta w algorytmie FREQT, a więc także w algorytmie WrapIT, nie umożliwia rozszerzenia wzorca w lewo. Aby jednak umożliwić takie rozszerzenie algorytm WrapIT został wyposażony w kolejne rozszerzenie. Aktualnie wyznaczony wzorzec oraz drzewo danych są odwracane (wszystkie węzły są ustawiane w odwrotnej kolejności). Po odwróceniu wzorca lewa skrajna gałąź staje się prawą skrajną gałęzią, natomiast lewy skrajny element staje się prawym skrajnym elementem. Dla wszystkich elementów z nowej prawoskrajnej gałęzi drzewa jest następnie obliczana lista wystąpień RMO. Po takiej konwersji otrzymujemy wzorzec, który możemy dalej rozwijać poprzez kolejne uruchomienie zmodyfikowanego algorytmu FREQT. Dla naszego przykładowego drzewa wynikowy wzorzec będzie wyglądał następująco:

```
<SZTUCZNY KORZEN>
 <i>#tekst</i>
 <a>
  <b>#tekst</b>
  <font>#tekst</font>
 </a>
 <p>#tekst</p>
</SZTUCZNY KORZEN>
```

Oczywiście, aby uzyskać wzorzec działający na wejściowym drzewie danych D należy po raz kolejny go odwrócić, tak żeby uzyskać rozszerzony wzorzec o 100% poparciu w danych:

```
<SZTUCZNY KORZEN>
 <p>#tekst</p>
 <a>
  <font>#tekst</font>
  <b>#tekst</b>
 </a>
 <i>#tekst</i>
</SZTUCZNY KORZEN>
```

Kolejnym etapem rozszerzania wzorca jest odnalezienie elementów opcjonalnych. Jak już zaznaczyliśmy w założeniach algorytmu każdy wzorzec może zawierać pewne fragmenty opcjonalne zawierające potencjalnie wartościowe dane. Aby odnaleźć takie elementy należy przekształcić dotychczasowy maksymalny wzorzec próbując między 2 kolejne występujące w nim węzły "włożyć" nowe węzły, które w danych mają poparcie conajmniej $0 < \sigma < 1$. Do realizacji tego celu można po raz kolejny wykorzystać zmodyfikowany algorytm FREQT. Niezbędne będzie jednak pewne rozszerzenie polegające na wprowadzeniu prawego ograniczenia dla każdego prawostronnego wystąpienia w RMO. Ograniczenie takie określa jak daleko algorytm Update-RMO' może szukać kolejnych wystąpień etykiety λ. Po takiej zmianie warukek w pkt 2c algorytmu Update-RMO' przyjmie postać:

Tak długo jak $y \neq null$ i $y \leq Rbound(x)$ wykonuj:

- jeśli $L_D(y) = \lambda$ oraz $y \notin RMO_{new}$, to $RMO_{new} := RMO_{new} \cup y$;
- $y := y.nextExtendedSibling()$;

gdzie Rbound(x) oznacza prawe ograniczenie na wystąpienie węzła x.

Wyznaczenie elementów opcjonalnych wzorca odbywa się przez podzielenie wzorca T na T - 1 wzorców T_i o rozmiarach kolejno 1, 2... T - 1. Każdy wzorzec T_i będzie zawierał i pierwszych elementów wzorca T. Dodatkowo każde prawe skrajne wystąpienie wzorca T_i będzie miało ograniczenie równe odpowiadającemu mu prawemu skrajnemu wystąpieniu wzorca T_{i+1} . Dla każdego takiego wzorca T_i zostanie następnie wykonany zmodyfikowany algorytm FREQT z progiem minimalnego poparcia σ. Dzięki temu zostaną odnalezione wszystkie węzły lub też poddrzewa występujące pomiędzy istniejącymi węzłami wzorca o poparciu conajmniej równym σ. Wyniki z rozszerzeń poszczególnych wzorców T_i należy następnie połączyć w jeden wzorzec z opcjonalnymi elementami. Przykładowy wzorzec cząstkowy T_6 dla naszego przykładowego drzewa danych wygląda następująco:

```
<SZTUCZNY KORZEN>
 <p>#tekst</p>
 <a>
  <font>#tekst</font>
  <b></b>
 </a>
</SZTUCZNY KORZEN>
```

Prawymi ograniczeniami dla wystąpień znacznika <b> są wszystkie wystąpienia ciągu tekstowego "tekst". Po wykonaniu algorytmu przy poziomie minimalnego poparcia σ = 0,5 otrzymamy następujący wzorzec:

```
<SZTUCZNY KORZEN>
 <p>#tekst</p>
 <a>
  <font>#tekst</font>
  <b><i></i></b>
 </a>
</SZTUCZNY KORZEN>
```

Po połączeniu z wzorcem oryginalnym T otrzymujemy końcowy wzorzec rozszerzony:

```
<SZTUCZNY KORZEN>
 <p>#tekst</p>
 <a>
  <font>#tekst</font>
  <b><i optional="true">#tekst</i></b>
 </a>
 <i>#tekst</i>
</SZTUCZNY KORZEN>
```

Jest to końcowy wzorzec otrzymany z naszego przykładowego drzewa danych.

W sekcji tej zostały opisane techniki przekształcenia wzorca podstawowego w pełny (rozszerzony) wzorzec zawierający zarówno sąsiednie poddrzewa wchodzące w skład snippetu, jak i opcjonalne znaczniki, które mogą, ale nie muszą znajdować się pomiędzy znacznikami obowiązkowymi.

5.2.4 Analiza semantyczna wzorca

W poprzedniej sekcji został opisany algorytm wyznaczający strukturę drzewa HTML zawierającą pojedynczy snippet. Kolejnym etapem przetwarzania jest **analiza semantyczna wzorca**, czyli określenie znaczników, zakresów znaczników, poddrzew lub też atrybutów zawierających poszukiwane informacje.

Ilustracja 5.8 Algorytm oznaczania tokenów we wzorcu: znajdzTokeny

```
procedura znajdzTokeny(wezel W, biezaca strefa S, zbior E)
( 1)   • dodaj Token_wezel(W) do E
( 2)   • jesli etykieta W nalezy do znacznikow strukturalnych
     i W nie jest w złem opcjonalnym
( 3)    • jesli S jest otwarta
( 4)      • zamknij S
( 5)      • dodaj Token_strefa(S) do E
( 6)      • otworz nowa strefe S
( 7) • jesli etykieta W jest znacznikiem <A>
( 8)    • dodaj Token_atrybut: href(W) do E
( 9)  • dla wszytkich dzieci W: Dziecko_i(W)
(10)    • wykonaj znajdzTokeny(Dziecko_i(W), S, E)
```

Na początku analizy należy odnaleźć elementy strukturalne wzorca, które potencjalnie mogłyby zawierać poszukiwane informacje. Każdy taki element będziemy nazywać **tokenem**. Do oznaczenie tokenów we wzorcu T wykorzystaliśmy algorytm przedstawiony na Ilustracja 5.8. Jest on uruchamiany dla korzenia wygenerowanego wzorca. W wyniku otrzymujemy wypełniony zbiór tokenów E. Tokeny uzyskiwane z tej procedury mogą być trzech rodzajów: tokeny węzłowe, czyli obejmujące swoim zasięgiem węzeł i wszystkie jego dzieci, tokeny strefowe, czyli takie zawierające całą przestrzeń między dwoma węzłami wzorca oraz tokeny atrybutowe zawierające wartość konkretnego atrybutu węzła.

Algorytm znajdzTokeny na wejściu przyjmuje węzeł do analizy W, bierzącą strefę S oraz wynikowy zbiór tokenów E. Algorytm analizuje wejściowy węzeł W a następnie rekurencyjnie wykonuje się dla jego wszystkich dzieci (9-10). Dla każdego analizowanego węzła tworzony jest token węzłowy $Token_{węzeł}(W)$ (1). Umożliwia to oznaczenie pojedynczych węzłow i całych poddrzew, które zawierają istotne informacje. Oprócz tego pomiędzy znacznikami strukturalnymi oznaczane są tokeny strefowe $Token_{strefa}(S)$. Takie tokeny umożliwiają analizę fragmentów wzorca zawierające więcej niż jeden węzeł lub poddrzewo wzorca. Tokeny strefowe umożliwiają także analizę fragmentów, których wzorzec nie obejmuje, ale takich, które znajdują się pomiędzy dwoma znanymi węzłami wzorca. W naszej implementacji wybraliśmy następujący zbiór znaczników strukturalnych HTML wykorzystywanych do konstrukcji stref:

- <DIV>
- <A>
- <OL>
- <UL>
- <LI>
- <DL>
- <DT>
- <DD>
- <TR>
- <TD>
- <TABLE>
- <SPAN>
-
- <P>

Znaczniki te należą do tych najczęściej wykorzystywanych przez twórców stron do strukturalizacji dokumentu. Znajdują się tutaj więc znaczniki oznaczające rozpoczęcie nowej linii (BR, P), znaczniki list (OL, UL, LI, DL, DD, DT) i tabel (TABLE, TR, TD) a także znaczniki regionów (DIV, SPAN). Dodatkowo uwzględnione zostały wszystkie znaczniki hiperlinków (A) jako takie które logicznie rozdzielają części wzorca. Dodatkowo dla węzłów znacznika *A* tworzony jest dodatkowy token atrybutowy dla atrybutu *href* umożliwiając w ten sposów uwzględnienie w poszukiwaniu adresu URL także wszystkich docelowych adresów linków.

Dla każdego oznaczonego tokenu obliczane są wartości pewnych atrybutów tego tokenu, umożliwiając rozróżnienie czterech klas: *tytuł, opis, url i nieistotne*. Listę obliczanych atrybutów wraz z ich znaczeniem prezentujemy poniżej:

- Miara informatywności - określenie czy token zawiera istotne informacje, czy też jest nieistotnym stałym elementem wzorca.
- Średnia liczba słów w tokenie.
- Procent prawidłowych adresów URL.
- Liczba różnych serwerów w adresach URL.
- Procent wystąpień zawartych w tagu <A>, czyli linków.

Miara informatywności

Sekcja 4.2.2 opisauje metodę określania informatywnych bloków danych. Taka technika została zastosowana do rozpoznania informatywnych tokenów. Każdy token wzorca reprezentuje pojedynczy blok danych, a każdy snippet stanowi oddzielny dokument dla algorytmu rozpoznawania informatywnych bloków. Jako schemat ważenia termów została wykorzystana znormalizowana miara Tf (częstości termów). Za tym wyborem stoi fakt, iż sama miara entropii uwzględnia termy o dużej częstości występowania w kolekcji dokumentów. Nie ma więc potrzeby wykorzystywania miary Tf-idf, która jest napewno lepsza w zastosowaniu do wyszukiwania dokumentów, jednak w naszym przypadku jest bezcelowa. Tak odnalezione tokeny o wysokiej informatywności mogą być potencjalnymi kandydatami na tokeny zawierające tytuł lub też opis dokumentu.

Procent prawidłowych adresów URL

W celu sprawdzenia czy token zawiera prawidłowy adrs URL stosowany jest kilkustopniowy proces weryfikacji. Pierwszym krokiem jest normalizacja adresu, tj. uzupełnienie jego brakujących elementów (np. bardzo powszechnie pomijanego nagłówka *http://* w przypadku adresów wypisanych na stronie). Następnie adres jest sprawdzany pod względem poprawności składniowej. W przypadku poprawnego przejścia tych dwóch testów następuje próba pobrania dokumentu spod wyekstrahowanego adresu URL. Operacja taka jest wymagana ze względu na stosowany bardzo powszechnie przez serwisy wyszukujące mechanizmu przekierowania. Jako adres docelowy hiperlinka do odnalezionej strony wyszukiwarki często podają adres swojego serwisu, który przekierowuje dopiero przeglądarkę internetową pod właściwy adres. W takim przypadku należy odnaleźć i użyć adresu URL, który jest zazwyczaj wypisany w formie tekstowej w treści snippetu.

Liczba różnych serwerów w adresach URL

Obliczanie liczby różnych serwerów w adresach URL ma stanowić drugie zabezpieczenie przed niewłaściwymi adresami przekierowującymi. Atrybut ten zlicza liczbę różnych serwerów w adresach URL w stosunku do ich całkowitej liczby. W przypadku adresu odnoszącego się do serwisu przekierowującego różnorodność tych serwerów będzie bardzo mała. Z kolei przy prawidłowym adresie URL spodziewać się należy dość dużej różnorodności serwerów.

Procent wystąpień zawartych w tagu <A>

Ponieważ zarówno tytuł dokumentu jak i jego opis są elementami wysoce informatywnymi, nie istnieje sposób rozpoznania tylko na podstawie informatywności pomiędzy opisem a tytułem. Jedyną obserwacją prowadzącą do rozróżnienia tych elementów jest fakt, iż standardem dla wyszukiwarek internetowych jest fakt, iż tytuł dokumentu jest zawsze aktywny, tj. stanowi hiperlink do znalezionej strony. Stąd niezbędne jest obliczenie dla każdego tokenu procenta zawartości zawartej wewnątrz jakiegoś tagu <A>.

Po obliczeniu wartości atrybutów, dla każdego tokenu stosowana jesli lista reguł umożliwiająca przypożądkowanie tokenu do danej grupy:

```
• dla kazdego tokenu powtarzaj:
    • jesli token jest informatywny oraz jego srednia ilosc slow > 2
     • jesli token jest wewn trz tagu <A>
         • przypisz token do klasy: tytul
     • w przeciwnym przypadku przypisz token do klasy: opis
 • w przeciwnym przypadku jesli token jest prawidlowym URL
          oraz ma roznorodne adresy serwerow w URL
       • przypisz token do klasy: URL
    • w przeciwnym przypadku przypisz token do klasy: nieistotny
```

Należy zauważyć, iż algorytm gwarantuje przynależność jednego tokenu do jednej i tylko jednej klasy.

5.2.5 Uspójnienie i oczyszczenie wzorca

Wygenerowany wzorzec będziemy nazywali **spójnym**, jeśli żadem z oznaczonych elementów wynikowych nie zawiera w swoim poddrzewie innego elementu. Wymaganie takie zapobiega sytuacji aby w opisie snippetu został ujęty URL lub tytuł, lub też aby

fragmenty opisu zostały powielone. Z przedstawionego powyżej algorytmu taka sytuacja jest jednak możliwa. Wynika to głównie z własności informatywności, takiej że jeśli token oparty na zawartości jednego węzła jest informatywny, również token oparty na jego rodzicu we wzorcu będzie prawdopodobnie informatywny. Dodatkowo spójny wzorzec może zawierać jedynie jeden element *URL* i jeden element *tytuł*. Jest to o tyle sensowne, że URL może być tylko jeden a tytuł nigdy nie jest rozbijany na fragmenty i dlatego spodziewamy się, iż będzie ujęty w jeden token.

Rozstrzyganie konfliktów we wzorcu

Wzorzec wygenerowany przez przedstawiony powyżej algorytm może nie być spójny. Aby go uspójnić należy zastosować specjalne procedury rozstrzygania konfliktów. Dla różnych elementów wzorca stosowane są różne reguły rozwiązywania problemów.

W przypadku adresu URL istnieje wymaganie, iż we wzorcu adres może występować tylko raz. W przypadku liczniejszych wystąpień niezbędne jest rozstrzygnięcie który należy wybrać. Preferencje ma token o wyższej wartości prawidłowych adresów URL. W drugiej kolejności brana jest pod uwagę różnorodność serwerów.

Jeśli chodzi o tytuł, w przypadku pojawienia się więcej niż jednego tokenu z taką etykietą, stosowana jest intuicja, iż tytuł jest zazwyczaj pierwszym elementem wyswietlonego na stronie wyniku. Stąd po prostu brany pod uwagę jest token który występuje wcześniej we wzorcu.

W przypadku opisu usuwane są wszystkie tokeny które zawierają w swoim wnętrzu dowolny inny token z przypisaną wcześniej klasą różną od *nieistotny*.

Oczyszczenie wzorca

Ostatnią fazą algorytmu jest końcowe oczyszczenie wzorca. Celem jest usunięcie z niego pewnych elementów, które nie są istotne pod względem struktury wzorca jak i nie zostały przypisane do żadnej klasy różnej od *nieistotny*.

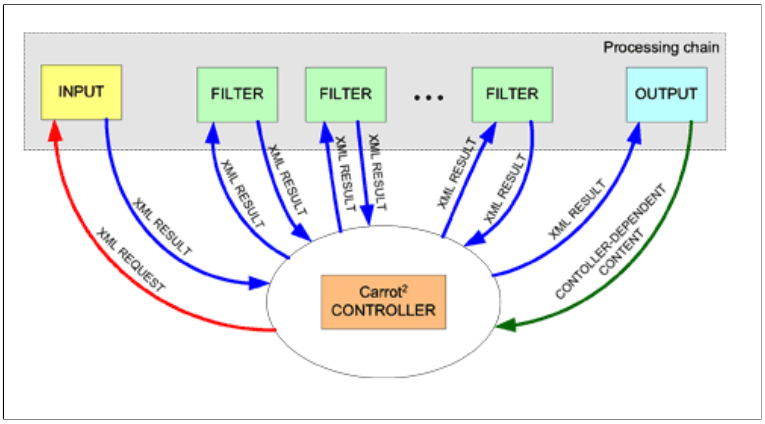
5.3 Zastosowanie wzorca do ekstrakcji

Elementem tej pracy jest także praktyczna implementacja algorytmu WrapIT, jak i zastosowanie jego wyników do ekstrakcji snippetów z różnych serwisów wyszukujących. Wrapper wykorzystujący wzorce WrapIT został zaimplementowany z myślą o wykorzystaniu jako komponent wejściowy architektury Carrot^2.

5.3.1 Architektura Carrot^2

Carrot^2 jest systemem autorstwa Dawida Weissa [Weiss, 03]. System ten jest zaprojektowany z myślą o eksperymentach z dziedziny przetwarzania i wizualizacji wyników zapytań do różnorodnych źródeł danych (w tym do wyszukiwarek internetowych). Projekt architektury zakłada jak największą modularność i elastyczność dając możliwości dowolnego tworzenia i eksperymentowania na nowych komponentach.

Ilustracja 5.9
Architektura systemy Carrot²



Carrot² jest oparty na koncepcji oddzielnych komponentów, które komunikują się ze sobą poprzez wymianę danych semistrukturalnych XML. Sama komunikacja odbywa się przy wykorzystaniu protokołu HTTP. Daje to wielką elastyczność dodawania nowych komponentów niezwiązanych z implementacją reszty systemu zarówno pod względem języka programowania jak i lokalizacji. Architektura Carrot² przewiduje wystąpienie komponentów czterech typów:

- **Wejściowy** - ten typ komponentu przyjmuje jako wejście zapytanie użytkownika (w odpowiedniej strukturze XML) i powinien na wyjściu dać listę dokumentów zgodnych z zapytaniem. Komponent może zarówno sam posiadać funkcjonalność wyszukiwarki lub też może jedynie odpytywać inny serwis w celu pozyskania danych.
- **Filtrujący** - komponent taki powinien przyjąć na wejściu listę dokumentów (dostarczoną przez komponent wejściowy lub też inny komponent filtrujący) i dać w wyniku tą samą niezmienioną listę wraz z dołączoną własną informacją. Komponent taki może np. generować własny ranking dokumentów, realizować grupowanie wyników.
- **Wyjściowy** - tego typu komponenty są odpowiedzialne za prezentację wyniku do użytkownika. Komponent taki przyjmuje na wejściu listę dokumentów. Wyjście takiego komponentu nie jest określone. Może to być zarówno wynikowa strona HTML jak i poprostu zapis wyników na dysk.
- **Kontroler** - zadaniem takiego komponentu jest spinanie pozostałych komponentów w całość tworząc **proces przetwarzania**. Kontroler jest odpowiedzialny za kompunikację pomiędzy modułami, które (co pokazuje Ilustracja 5.9) nie są świadome ani swoich poprzedników jak i następników w procesie przetwarzania.

Umieściliśmy tutaj jedynie podstawowe informacje na temat architektury Carrot². Dokładniejsze informacje są dostępne na stronie domowej projektu [Carrot2, 03].

5.3.2 Implementacja komponentu wejściowego

W tym podrozdziale przedstawimy szczegóły wykorzystania wzorca snippetu wygenerowanego przez algorytm WrapIT do konstrukcji komponentu wejściowego Carrot².

Zastosowanie wrappera jako komponentu wejściowego dla systemu przetwarzania zapytań jest dość naturalne. Dotychczas Carrot^2 stosowany jest do grupowania wyników wyszukiwania, stąd wykorzystywany w nim jest komponent pobierający dane z innych serwisów wyszukiwawczych. Stworzyliśmy właśnie taki komponent, którego zadaniem jest pobranie listy snippetów na podstawie deskryptora wrappera, a następnie przekształcenie jej na listę dokumentów Carrot^2.

Deskryptor dla komponentu wejściowego - wrappera stanowi połączenie deskryptora wejściowego algorytmu WrapIT i wzorca snippetu będącego wynikiem tego algorytmu. Ilustracja 5.10 przedstawia pseudokod podstawowego przetwarzania wrappera. Dwie funkcje *pobierzStroneWynikowa* i *pobierzKolejnaStroneWynikowa* realizują zapytania do wyszukiwarki internetowej na podstawie opisu serwisu a następnie konstruują drzewo rozkładu HTML dla takiej strony wynikowej.

Ilustracja 5.10 *Algorytm pobierania snippetów ze strony wynikowej na podstawie deskryptora*

```
pobierzSnippety(wzorzec, liczbaSnippetow)
  • Strona := pobierzStroneWynikowa;
  • Wyniki := ∅
  • powtarzaj
    • Element := znajdzWzorzec(korzen(strona), korzen(wzorzec));
    • jesli znaleziony(Element) to
      • Wyniki := Wyniki + Element;
      • Strona := Strona - Element;
      • jesli #Wyniki = liczbaSnippetow
        • Zakoncz
    • jesli nie znaleziony(Element) to
      • Strona := pobierzKolejnaStroneWynikowa
```

Najistotniejszym elementem algorytmu jest funkcja *znajdzWzorzec* odpowiedziana za odnalezienie konkretnego poddrzewa wzorca poczynając od konkretnego węzła drzewa HTML. Pseudokod tej funkcji przedstawia Ilustracja 5.11.

Ilustracja 5.11 *Algorytm wyszukiwania węzła wzorca snippetu*

```
znajdzWzorzec(wezelHTML, wezelWzorca)
  • Element := znajdzWezel(wezelHTML, wezelWzorca)
  • jesli nie znaleziony(Element) to
    • zwroc wynik: nie znaleziony
  • dla kazdego dziecka wezlaWzorca powtarzaj
    • Dziecko := znajdzWzorzec(Element, dziecko wezlaWzorca)
    • jesli nie zneleziony(Dziecko) to
   • zwroc wynik: nie znaleziony
 • jesli znaleziony(Dziecko) to
   • wezelHTML := wezelHMTL - Dziecko
```

Istotnym elementem w tego algorytmu jest funkcja *znajdzWezel*. Realizuje ona proste wyszukanie w poddrzewie HTML węzła podanego jako pierwszy parametr elementu HTML o takiej samej etykiecie jak element wzorca podany jako drugi parametr. Ten element można zrealizować w czasie liniwym poprzez przegląd zakresu znaczników w poddrzewie analizowanego węzła.

Komponent wejściowy realizuje prosty algorytm wyszukania wzorca w stronie wynikowej dla pozyskania fragmentów drzewa HTML odpowiadających wzorcowi. Na podstawie

oznaczeń semanycznych we wzorcu dla każdego odnalezionego wystąpienia ekstrahowane są wszystkie elementy tekstowe i składane by dać w wyniku cały snippet (czyli tytuł, opis i adres URL).

5.4 Wady i zalety algorytmu

Przedstawiony w tym rozdziale algorytm WrapIT jest generatorem wrapperów dla stron wyników wyszukiwania z powtarzającą, aczkolwiek nieregularną strukturą. Przedstawimy teraz w sposób skumulowany zalety tego algorymu:

a. **Analiza dowolnej strony wyników wyszukiwania** - algorytm nie wymaga żadnej wiedzy na temat serwisu dla którego generowany jest wrapper poza parametrami niezbędnymi do wykonania zapytania i liczby zwracanych wyników. Nie wymaga oznaczania stron przykładowych dla systemu uczącego ani innej interakcji z użytkownikiem.
b. **Szybkie działanie wygenerowanego wrappera** - dzięki rozdzieleniu generatora wrapperów i samego komponentu wrappera możliwa była optymalizacja jego kodu, tak aby pobieranie stron w trybie on-line było maksymalnie szybkie. Sam proces generacji może być używany w trybie off-line i nie stanowi problemu fakt, iż generacja może pochłonąć dłuższy okres czasu.
c. **Możliwość ujęcia we wzorcu elementów opcjonalnych** - wiele serwisów wyszukiwawczych zawiera elementy, które nie są zawarte w każdym wyniku wyszukiwania. Niektóre z tych elementów mogą być bardzo interesujące z punktu widzenia dalszego przetwarzania. Algorytm WtapIT umożliwia odnalezienie takich elementów i oznaczenie ich jako interesujących na równi z elementami występującymi w każdym snippecie.
d. **Możliwość stałej aktualizacji deskryptora wrappera** - automatyczny proces generacji wrappera umożliwia regularne uaktualnianie wrappera bez wymaganej ingerencji człowieka. Stwarza to możliwość uodpownienia serwisu wykorzysującego wrapper na zmiany wyglądu lub też konstrukcji wewnętrznej snippetu.

Algorytm oczywiście ma także pewne wady, których wyeliminowanie należałoby uwzględnić w dalszych pracach nad rozwojem tego algorytmu:

a. **Bazowanie na jednym zapytaniu** - algorytm analizuje stronę wynikową dla pojedynczego zapytania. Stąd wynika wymaganie, aby to zapytanie było reprezentatywne, czyli zwracało wyniki kompletne, zawierające wszystkie poszukiwane elementy.
b. **Słabe działanie na stronach o małym ustrukturalizowaniu wyników** - algorytm WrapIT bazuje na wyszukiwaniu struktur HTML. Jeśli strona wynikowa jest takich struktu pozbawiona algorytm będzie spisywał się dużo słabiej.

Rozwiązanie niektórych problemów związanych z tym algorytmem proponujemy w kolejnym podrozdziale.

5.5 Perspektywy rozwoju algorytmu

Jakkolwiek WrapIT daje całkiem niezłe rezultaty, dla niektórych serwisów wyszukujących nie jest w stanie wygenerować prawidłowego wrappera. Dzieje się tak głównie z powodów opisanych w poprzednim podrozdziale jako wady algorytmu.

Opieranie się na jednym zapytaniu

Aby umożliwić dokładniejszy przegląd struktur występujących w wynikach wyszukiwania algorytm mógłby stosować słownik zapytań stosowanych do analizy. Dla każdego zapytania oddzielnie algorytm generowałby wrapper, a następnie dokonywał próby połączenia wzorców uzyskanych dla różnych zapytań w jeden wzorzec pasujący do wszystkich zapytań. Konieczna byłaby w tym podejściu iteracyjna weryfikacja połączonego wzorca, tak aby wzorzec pasował do każdego wyniku, a jednocześnie nie zawierał zbyt wielu elementów opcjonalnych prowadząc do błędnych dopasowań wzorca.

Problem słabej struktury HTML wyników

Dla stron o słabej strukturze wyników problemem zazwyczaj jest błędne określenie wzorca podstawowego na stronie. W takim przypadku nie da się skutecznie określić lokalizacji snippetów przy użyciu jedynie podejścia zastosowanego w WrapIT. Jednym z możliwych podejść do rozwiązania tego problemu mogłaby być analiza informatywności dla całego dokumentu. Odnajdując pewne powtarzające sie informatywne struktury możliwe byłoby prawidłowe określenie lokalizacji snippetów i konstrukcja wzorca dla wrappera.

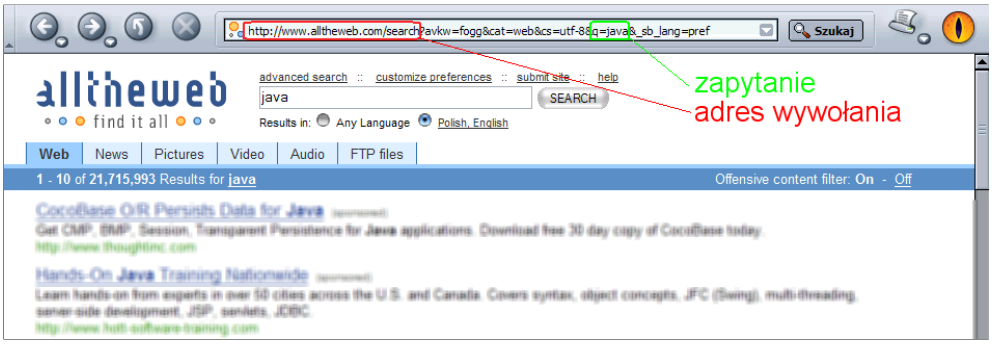
6 WrapIT - Wykorzystanie i ocena algorytmu

W rozdziale tym zostanie przedstawione praktyczne wykorzystanie algorytmu WrapIT do konstrukcji wrappera dla wyszukiwarki AllTheWeb [AllTheWeb, 03]. Sformułujemy także dwie obiektywne miary oceny działania tego algorytmu i dokonamy analizy jego skuteczności dla różnych serwisów wyszukujących. Na końcu przedstawimy szczegółowo pewne specyficzne sytuacje, w których algorytm nie zachowuje się zgodnie z oczekiwaniami.

6.1 Przykładowa generacja wrappera

6.1.1 Przygotowanie danych wejściowych

Aby wygenerować wrapper dla dowolnej wyszukiwarki, należy w pierwszej kolejności określić sposób wykonania zapytania do niej. W tym celu najczęściej wystarczy przeprowadzić kilka prostych kroków zupełnie wystarczających do określenia niezbędnych parametrów. W pierwszej kolejności należy urochomić przeglądarkę internetową i wpisać dowolne zapytanie.



Ilustracja 6.1 Wyszukiwarka AllTheWeb dla zapytania "java"

Ilustracja 6.1 przedstawia stronę wyszukiwarki dla zapytania "java". Aby określić sposób wywołania zapytania należy dokonać prostej analizy adresu URL dla strony wynikowej. W powyższym przykładzie ten adres jest następujący: *http://www.alltheweb.com/search?avkw=fogg&cat=web&cs=utf-8&q=java&_sb_lang=pref*. Należy zwrócić uwagę na dwa istotne elementy: adres wywołania i parametr określający zapytanie. W tym wypadku wywoływany jest adres: *http://www.alltheweb.com/search*, a zapytanie jest widoczne w parametrze *q* co widać we fragmencie adresu: *q=java*. W tym momencie mamy określone dwa najważniejsze parametry. Kolejny potrzebny parametr to ten odpowiedzialny za odwołanie do kolejnych stron wynikowych wyszukanego zapytania. W tym celu

nawigujemy do kolejnej strony i po raz kolejny sprawdzamy adres. W wyszukiwarce AllTheWeb kolejna odwołanie do kolejnej strony daje nam adres: *http://www.alltheweb.com/search?_sb_lang=pref&cs=utf-8&cat=web&q=java&avkw=fogg&o=10*. Jak widać przybył tutaj dodatkowy parametr *o* o wartości *10*. Nawigując do kolejnej strony możemy się łatwo przekonać, iż ten parametr będzie zawierał zawsze liczbę oznaczającą numer rekordu zaczynającego stronę. Ostatni parametr konieczny do określenia to liczba pojedynczych wyników na stronie. Można to policzyć ręcznie uzyskując wartość *10* dla naszej przykładowej wyszukiwarki. Mamy już wszystkie niezbędne informacje konieczne do zbudowania *deskryptora wejściowego*[14] dla algorytmu.

Ilustracja 6.2
Deskryptor wejściowy dla wyszukiwarki AllTheWeb

```
<?xml version="1.0"?>
<!-- Opis wyszukiwarki AllTheWeb-->
<service>
    <request>
        <!-- Adres wywołania -->
        <service url="http://www.alltheweb.com/search" method="get" />
        <parameters>
            <!-- Zapytanie -->
            <parameter name="q" mapto="query.string" />
            <!-- Kolejne strony -->
            <parameter name="o" mapto="query.startFrom" />
            <!-- Dodatkowe parametry -->
            <parameter name="cat" value="web" />
            <parameter name="cs" value="utf-8" />
            <parameter name="avkw" value="fogg" />
        </parameters>
    </request>
    <response>
        <pageinfo>
            <!-- Liczba wyników na stronie -->
            <expected-results-per-page>10</expected-results-per-page>
        </pageinfo>
    </response>
</service>
```

W deskryptorze można odnaleźć wszystkie określone wcześniej elementy. Adres wywołania znajduje się w atrybucie *url* znacznika *service*. Po tym znaczniku występuje sekcja parameters, w której są wyszczególnione wszystkie parametry występujące w URL zapytania. Parametry o specjalnym znaczeniu, takie jak treść zapytania i odwołanie do kolejnych stron posiadają zamiast atrybutu o stałej wartości *value*, atrybut oznaczający mapowanie konkretnego znaczenia. Stąd parametr oznaczający zapytanie ma atrybut *mapto*="*query.string*", natomiast parametr odpowiedzialny za odwołania do kolejnych stron *mapto*="*query.startFrom*". Ostatnim znaczącym elementem jest wartość w znaczniku *expected-results-per-page* sekcji *response*. To tutaj określona jest liczba wyników na pojedynczej stronie produkowanej przez wyszukiwarkę.

[14] Plik XML określający sposób wywołania serwisu wyszukującego

W ten sposób na przykładzie wyszukiwarki AllTheWeb przedstawiliśmy wszystkie niezbędne kroki wymagane, aby utworzyć deskryptor wejściowy dla algorytmu WrapIT.

6.1.2 Uruchomienie generatora

Mając gotowy deskryptor wyszukiwarki możemy wygenerować dla niej wrapper. W tym celu uruchamiamy skrypt *run.bat* podając plik deskryptora jako pierwszy parametr. Dodatkowo możemy określić takie parametry jak zapytanie, dla którego będzie generowany wrapper (domyślnie "java"), maksymalny próg entropii zawartości, dla którego elementy wzorca są traktowane jako istotne (domyślnie 0,35), oraz minimalne poparcie na stronie przykładowej dla elementów opcjonalnych (domyślnie 0,7). Najistotniejszym elementem, który może mieć wpływ na skuteczność generacji jest zapytanie. Aby sprawdzić, czy zapytanie będzie dobre do generacji, warto uruchomić ręcznie wyszukiwarkę i sprawdzić, czy zwraca dostatecznie dużo wyników na to zapytanie (minimalnie dwie strony), oraz czy zapytanie nie zwraca zbytnio specyficznych wyników (tzn. czy nie występuje sytuacja, gdy większość wyników na stronie jest z jednego serwera, lub też nie zawiera jakiegoś istotnego elementu, np. opisu). Pozostałe parametry można pozostawić przy wartościach domyślnych i wykorzystać jedynie w przypadku niedokładności generacji.

Wywołanie generatora spowoduje wyświetlenie na ekranie komunikatów poszczególnych faz generacji. Na początku pojawią się komunikaty związane z generacją wzorca podstawowego, między innymi wypis słowników poziomu 1 i 2. Następnie pojawia się wygenerowany wzorzec podstawowy.

```
*** MAX non-optional pattern ***

<__ROOT__ [1.0]>
  <p class=result [1.0]>
    <span class=resTitle [1.0]>
      <a [1.0]/>
    </span class=resTitle>
    <br [1.0]/>
    <span class=resDescrLabel [1.0]/>
    <span class=resDescr [1.0]/>
    <br [1.0]/>
    <span class=resURL [1.0]/>
    <span class=resSize [1.0]/>
  </p class=result>
</__ROOT__>
```

Po tym następuje poszukiwanie elementów opcjonalnych. Wyszukany wzorzec jest wypisywany na ekranie. W nawiasach kwadratowych podane jest wsparcie poszczególnych elementów wzorca. Znaczniki o wsparciu mniejszym niż *1.0* są opcjonalne.

```
*** MAX optional pattern ***

<__ROOT__ [1.0]>
  <p class=result [1.0]>
    <span class=resTitle [1.0]>
      <a [1.0]>
        <span class=hlight [0.7]/>
```

```
      </a>
    </span class=resTitle>
    <br [1.0]/>
    <span class=resTeaser [0.85]>
      <span class=hlight [0.85]/>
      <span class=hlight [0.75]/>
      <span class=hlight [0.75]/>
    </span class=resTeaser>
    <br [0.75]/>
    <span class=resDescrLabel [1.0]/>
    <span class=resDescr [1.0]>
      <span class=hlight [0.7]/>
    </span class=resDescr>
    <br [1.0]/>
    <span class=resURL [1.0]/>
    <span class=resSize [1.0]/>
  </p class=result>
</__ROOT__>
```

W tym momencie kończy się faza generowania wzorca, a rozpoczyna się analiza semantyki tego wzorca. Pierwszym elementem tej analizy jest wydzielenie elementów, które zostaną poddane analizie. Wydzielone są elementy 3 typów. Strefy pomiędzy dwoma znacznikami (*zone*), całkowite zawartości znaczników (*node*), oraz zawartości atrybutów (*attribute*). Strefy pomiędzy znacznikami są oznaczane przy pomocy pary atrybutów z nazwą elementu i wartością *beginOn* na początku strefy i *endBefore* na końcu strefy. Zawartość pierwszego znacznika strefy wchodzi do tej strefy, natomiast zawartość ostatniego znacznika nie wchodzi w skład strefy. W ten sposób kolejne strefy mogą zaczynać się i kończyć na tym samym znaczniku nie posiadając jednocześnie nakładających się zawartości.

```
<extractor>
  <snippet>
    <p class="result">
      <span class="resTitle" node_1="inside" zone_1="beginOn">
        <a node_2="inside" zone_1="endBefore" zone_2="beginOn"
                attr_1="attribute:href">
          <span class="hlight" optional="true" node_3="inside" />
        </a>
      </span>
      <br node_4="inside" zone_2="endBefore" zone_3="beginOn" />
      <span class="resTeaser" optional="true" node_5="inside">
        <span class="hlight" optional="true" node_6="inside" />
        <span class="hlight" optional="true" node_7="inside" />
        <span class="hlight" optional="true" node_8="inside" />
      </span>
      <br optional="true" node_9="inside" zone_3="endBefore"
              zone_4="beginOn" />
      <span class="resDescrLabel" node_10="inside"
              zone_4="endBefore" zone_5="beginOn" />
      <span class="resDescr" node_11="inside"
              zone_5="endBefore" zone_6="beginOn">
        <span class="hlight" optional="true" node_12="inside" />
      </span>
      <br node_13="inside" zone_6="endBefore" zone_7="beginOn" />
      <span class="resURL" node_14="inside" zone_7="endBefore"
              zone_8="beginOn" />
      <span class="resSize" node_15="inside" zone_8="endBefore"
              zone_9="beginOn" />
    </p>
  </snippet>
</extractor>
```

Kolejnym krokiem jest prosta eliminacja elementów bez zawartości. Algorytm wypisuje kolejne komunikaty oznaczające takie elementy:

```
Removing: zone_1
Removing: zone_9
Removing: node_13
Removing: node_4
Removing: zone_4
```

Dla pozostałych elementów następuje ocena ich funkcji semantycznych. W tym celu obliczane są różne statystyki dla zawartości każdego elementu.

```
node_2[true]: Srednia dlugosc, value: 43.0
node_2[true]:  rednia liczba słów, value: 5.0
Checking URLs:
.... done
node_2[true]: Is URL, value: 0.2
node_2[true]: Entropia zawarto ci, value: 0.1613810213181383
node_2[true]: URL TfIdf, value: 0.25
.
.
.
node_10[false]: Srednia dlugosc, value: 12.0
node_10[false]:  rednia liczba słów, value: 1.0
Checking URLs:
 done
node_10[false]: Is URL, value: 0.0
node_10[false]: Entropia zawarto ci, value: 0.0
node_10[false]: URL TfIdf, value: 0.05
```

Elementy, które są kandydatami na poszukiwane fragmenty wzorca są następnie wypisywane. Elementy oznaczone jako *Desc* stanowią podstawę do lokalizacji opisu. Elementy *Title* najprawdopodobniej zawierają tytuł, natomiast elementy *URL* są przypadkami występowania adresu dokumentu.

```
Desc: zone_3
Desc: node_1
Desc: zone_6
Desc: node_5
Desc: node_11
Title: node_2
URL: node_14
```

Można jednak łatwo zauważyć, iż znalezione elementy nie są rozłączne. Przykładowo *node_1* jest kandydatem na opis, podczas gdy zawarta w nim *node_2* stanowi kandydata na tytuł. Takie sprzeczności są rozwiązywane w fazie przetwarzania końcowego.

```
Removing: node_1 cause it has node_2 inside
Removing: node_11 cause it has zone_6 inside
Removing: zone_3 cause it has node_5 inside

Desc: zone_6
Desc: node_5
```

```
Title: node_2
URL: node_14
```

Konflikty między dwoma elementami są rozwiązywane poprzez zignorowanie jednego z nich. Zazwyczaj wybierany jest element bardziej wewnętrzny. W takim przypadku algorytm wypisuje komunikat (np. "*Removing: node_1...*"). Następnie wypisane zostają końcowe wyniki rozpoznania elementów. W tym przypadku opis został zidentyfikowany w elementach *zone_6* i *node_5*, natomiast tytuł w *node_2* a adres URL w *node_14*. W tym momencie z wzorca można usunąć wszystkie elementy opcjonalne, oraz znaczniki HTML oznaczające tylko formatowanie, które nie wchodzą w skład żadnego elementu wyniku. Ostatnim już elementem przetwarzania jest wypisanie ostatecznego oznaczonego juz wzorca.

```
*** Final tree: ***
<extractor>
  <snippet>
    <p class="result">
      <span class="resTitle">
        <a title="inside" />
      </span>
      <br />
      <span class="resTeaser" optional="true" description="inside" />
      <br optional="true" />
      <span class="resDescr" description="beginOn" />
      <br description="endBefore" />
      <span class="resURL" url="inside" />
    </p>
  </snippet>
</extractor>
```

W katalogu *output* jest również zapisywany pełny deskryptor wrappera gotowy do użycia w systemie Carrot[2]. W katalogu tym jest także zapisywana oryginalna postać ściągniętej strony z wynikami wyszukiwania oraz plik XML z wynikami testowego wykorzystania wygenerowanego wrappera dla tej właśnie strony. Umożliwia to szybkie sprawdzenie, czy stworzony wrapper działa poprawnie dla strony, na której podstawie został wygenerowany.

6.2 Ocena działania algorytmu

W tym podrozdziale sformułujemy dwie obiektywne miary oceny skuteczności działania algorytmu WrapIT oraz przedstawimy wyniki przeprowadzonych eksperymentów.

6.2.1 Ocena błędów wzorca

Wygenerowany przez algorytm WrapIT wzorzec, a dokładniej jego elementy semantyczne mogą w pewnych przypadkach nie być dostatecznie precyzyjne. Możliwa jest sytuacja, gdy element wzorca pokrywa zbyt dużo danych (np. elementy które nie są opisem są do

tego opisu dołączane) lub też jest zbyt wąska i nie bierze pod uwagę kompletnych danych (np. wzorzec zakłada istnienie tylko dwóch linijek tekstu, podczas gdy w szczególnych przypadkach występują trzy linijki - stąd wzorzec może tą ostatnią pomijać). Mogą również pojawić się sytuacje, gdy wzorzec całkowicie niepoprawnie zinterpretuje jakąś daną (np. adres URL włączony do wzorca nie będzie adresem odnoszącym się do dokumentu, lecz innym występującym w snippecie), lub też nie uda się wogóle zidentyfikować miejsca występowania któregoś elementu (np. nie zostanie zlokalizowane miejsce występowania tytułu dokumentu).

Równanie 6.1
Procent błędnej zawartości pobieranej przez wrapper

$$E = \frac{\sum_{j=1}^{d} \frac{\sum_{i=1}^{n} \frac{epos_{j,i} + eneg_{j,i}}{|res_{j,i}|}}{n}}{d} \cdot 100\%$$

Równanie 6.1 przedstawia wzór na miarę oceny błędu wzorca E przy pobieraniu elementu wyniku wyszukiwania. Wartość jest obliczana dla d różnych zapytań, z których w każdym sprawdzaliśmy n pierwszych wyników. Brana pod uwagę była liczba słów nadmiarowych pobieranych przez wrapper *epos*, jak i liczba słów brakujących w wyniku *eneg*. Wszystko to było obliczane w odniesieniu do całkowitej liczby słów w spodziewanym wyniku $|res|$. Wynik *0%* oznacza więc całkowitą zgodność pomiędzy danymi pobranymi przez wrapper a danymi spodziewanymi, natomiast wynik wynoszący 100% oznacza całkowity błąd pobrania danych (albo w ogóle nie ma prawidłowych danych, albo pobranych jest conajmniej tyle samo danych nadmiarowych). W przypadku pobierania całkowicie niepoprawnych danych (czyli brak danych właściwych a w zamian dane nadmiarowe) wynik może nawet przekroczyć teoretycznie 100%.

Testy przeprowadziliśmy dla 8 różnych serwisów wyszukujących: Google - niezależnie dla wersji angielskiej i polskiej [Google, 03], AllTheWeb [AllTheWeb, 03], HotBot [HotBot, 03], Altavista [Altavista, 03], Yahoo [Yahoo, 03], Netoskop [Netoskop, 03] oraz Szukacz.pl [Szukacz, 03]. Eksperyment wykorzystywał następujące zapytania: "*data mining*", "*java*", "*sport onet*", "*sport*", "*letnie igrzyska olimpijskie*".

Tabela 6.1
Wyniki oceny błędu wygenerowanego wzorca

Wyszukiwarka	Parametry	% błędow - tytuł	% błędow - URL	% błędow - opis
HotBot	domyślne	0%	0%	0%
Netoskop	domyślne	100%	0%	0%
Netoskop	Próg entropii = 0,45	0%	0%	0%
Google (PL)	domyślne	0%	0%	23%
Google	domyślne	0%	0%	7%
Altavista	domyślne	0%	0%	0%

Wyszukiwarka	Parametry	% błędow - tytuł	% błędow - URL	% błędow - opis
AllTheWeb	domyślne	0%	0%	0%
Yahoo	domyślne	0%	0%	0%
Szukacz.pl	domyślne	100%	100%	100%

Przedstawione powyżej wyniki wskazują na dużą skuteczność działania algorytmu WrapIT. W pięciu przypadkach wrapper wygenerowany przy wartościach domyślnych był dobry i pobierał poprawne dane ze stron wyników wyszukiwania. W przypadku wyszukiwarki Netoskop dla wartości domyślnych nie została zidentyfikowane lokalizacja tytułu. Jest to związane z pewną specyfiką tej wyszukiwarki, która ma tendencje do podawania wiele razy dokumentów z tego samego serwisu, czyli często posiadających taki sam tytuł. W takim przypadku naturalne jest, iż entropia takiej zawartości automatycznie będzie większa i dla domyślnego zapytania "java" właśnie taka sytuacja wystąpiła. Podniesienie progu akceptacji do wartości 0.45 dało w wyniku prawidłową lokalizację tytułu i w sumie całkowicie poprawnie funkcjonujący wrapper. Jedynie w jednym badanym przypadku dla serwisu Szukacz.pl nie został wygenerowany żaden wrapper. Odnaleziona struktura danych zawierała jedynie fragment snippetu i nie został w nim zidentyfikowany żaden z elementów funkcjonalnych. Pozostałe dwa przypadki to wyszukiwarka Google w wersji angielskiej i polskiej. W obu przypadkach wygenerowany wrapper spełniał swoją funkcję, jednak zawierał pewne drobne nieprawidłowości. W wersji angielskiej wzorzec objął swoim zasięgiem znacznik spoza opisu, stąd w niektórych opisach dokumentów pojawiała się zbędna etykieta "Category", a w niektórych do opisu była także dołączana data powstania dokumentu. Z kolei w wersji polskiej wyszukiwarki google wystąpił innej natury błąd, a mianowicie do opisu został zakwalifikowany fragment oznaczający przynależność do kategorii w katalogu tematycznym google. Stąd w opisie pojawiły się dodatkowe słowa.

6.2.2 Ocena dokładności wzorca podstawowego

Wzorzec podstawowy jest odpowiedzialny za pobranie kolejnych rezultatów ze strony wyników wyszukiwania. Jeśli wzorzec podstawowy zawiera pewne elementy obowiązkowe, które jednak nie muszą występować w każdym pojedynczym wyniku wyszukiwania danego serwisu, możliwe jest, iż wzorzec pominie taki wynik w czasie aktywnego wykorzystywania wrappera. Taka sytuacja występuje, gdy generator dostał stronę wzorcową, na której wszystkie wyniki zawierały element, który tak naprawdę jest opcjonalny. W celu przetestowania algorytmu na taką okoliczność wykorzystaliśmy drugą miarę jakości algorytmu określającą ilość "zagubionych" wyników dla poszczególnych serwisów wyszukujących.

Równanie 6.2 *Procent pominiętych wyników przez wrapper*

$$E = \frac{\sum_{j=1}^{d} \frac{|\ pobranych\ |}{|\ wyników\ |}}{d} \cdot 100\%$$

W mierze tej uwzględniamy wyniki dla *d* różnych zapytań. W każdym przypadku liczymy stosunek liczby wyników pobranych ze strony przez wrapper (|*pobranych*|) do liczby wyników faktycznie na tej stronie występujących (|*wyników*|). Wynik równy 100% oznacza w tym przypadku pobranie wszystkich wyników podawanych przez wyszukiwarkę, natomiast im wartość niższa, tym więcej wyników zostało zagubionych przez wrapper.

Tabela 6.2 *Wyniki oceny dokładności wygenerowanego wzorca podstawowego*

Wyszukiwarka	**Parametry**	**% pobranych wyników**
HotBot	domyślne	100%
Netoskop	domyślne	70%
Netoskop	Próg entropii = 0,45	100%
Google (PL)	domyślne	100%
Google	domyślne	100%
Altavista	domyślne	75%
AllTheWeb	domyślne	100%
Yahoo	domyślne	100%
Szukacz.pl	domyślne	0%

Jak ukazują wyniki (Tabela 6.2) tego eksperymentu większość wraperów nie miało skłonności do gubienia wyników. Jedynie Netoskop wymagał drobnego "dostrojenia" parametrów w celu uzyskania dobrego wrappera. Z kolei w przypadku Altavisty nie udało się znaleźć zapytania które uwzględniłoby wszystkie elementy opcjonalnie występujące w tej wyszukiwarce.

6.3 Podsumowanie

W rozdziale tym ukazaliśmy możliwości algorytmu WrapIT. Jak pokazaliśmy w części pierwszej tego rozdziału wygenerowanie wrappera dla serwisu wyszukującego jest mało skomplikowaną czynnością. Największego wysiłku od użytkownika wymaga konstrukcja deskryptora serwisu wyszukującego. Czynność tą zazwyczaj wystarczy wykonać tylko raz dla danego serwisu, gdyż administratorzy takich witryn rzadko zmieniają sposób wywołania ich wyszukiwarki. Możliwe jest zatem wykorzystanie generatora w systemach wymagających stałego utrzymywania i aktualizowania wrapperów.

W drugiej części rozdziału przeprowadziliśmy testy jakościowe generowanych automatycznie wrapperów. Pokazaliśmy jak skuteczny potrafi być algorytm i to iż w większości przypadków udaje mu się wygenerować dobry i kompletny wrapper. Ukazaliśmy także przypadki w których uzyskanie dobrych rezultatów wymaga dobrania dobrych parametrów algorytmu WrapIT. Podsumowując można powiedzieć, iż jak na

algorytm działający w pełni automatycznie WrapIT daje dobre wyniki i całkowicie funkcjonalne wrappery dla dużej liczby serwisów wyszukujących.

7

Podsumowanie

Znaczenie naukowe rezultatów

W pracy tej przedstawiliśmy nowe podejście do generacji wrapperów. Mimo tego, iż wszystkie rozwiązania z literatury zawierały metody pół automatycznego uczenia nadzorowanego lub też indukcji z przykładów, przedstawiony przez Nas algorytm WrapIT funkcjonuje bez takich dodatkowych informacji. Dzięki temu, iż strony wyników wyszukiwania są skonstruowane w pewien specyficzny sposób, możliwe stało się stworzenie algorytmu, który potrafi zrealizować zadanie generacji wrappera w sposób automatyczny. Rozwiązanie to może okazać się bardzo przydatne we wszystkich systemach pozyskujących dane z serwisów wyszukujących, gdzie niewygodne jest lub wręcz niedopuszczalne utrzymywanie ręcznie lub półautomatycznie generowanych wrapperów.

Postaramy się podsumować cechy naszego rozwiązania, które stanowią o jego oryginalności i przydatności w realnych zastosowaniach:

a. **Możliwość zastosowania dla dowolnego serwisu wyszukującego** - algorytm ma wbudowane jedynie bardzo ogólne założenia dotyczące konstrukcji strony wyników wyszukiwania. Założenia te nie są dotyczą w żaden sposób konstrukcji wewnętrznej analizowanego dokumentu, a jedynie informacji, jakich algorytm spodziewa się na takiej stronie odnaleźć. Dzięki temu nie jest wymagane praktycznie żadne przetwarzanie wstępne typu oznaczanie przykładów.
b. **Pełna automatyka działania** - wrapper dla serwisu wyszukiwawczego może być wygenerowany całkowicie automatycznie, bez potrzeby ingerencji użytkownika w czasie działania. Tym samym uaktualnianie wrappera dla wyszukiwarki może odbywać się automatycznie, nawet w regularnych odstępach czasu, zapewniając zawsze aktualny wrapper dla serwisu.
c. **Możliwość dostosowania do specyfiki serwisu** - dla serwisów, dla których wartości domyślne nie dają dobrych rezultatów i ze względu na ich specyfikę (np. tematyczną) wymagają doprecyzowania parametrów generacji (np. zmiany zapytania) wystarczy zazwyczaj to zrobić tylko za pierwszym razem. Przy zmianie prezentacji takiego serwisu dostosowane do niego parametry nadal pozostaną aktualne i pozwolą wygenerować nowy dobry wrapper.
d. **Szybkie działanie wrappera on-line** - dzięki rozdzieleniu generatora od samego wrappera samo przetwarzanie w produkcyjnym systemie on-line jest szybkie i niczym nie zakłócone. Czasochłonny proces generacji opisu wrappera może odbywać się zupełnie niezależnie od działania całej aplikacji.

Przy opracowywaniu tego podejścia wykorzystaliśmy istniejący algorytm FREQT, który jednak w oryginalnej postaci nie mógł być zastosowany. W rozdziale poświęconym szczegółowemu opisowi algorytmu WrapIT zaprezentowaliśmy liczne modyfikacje tego algorytmu wymagane przy użyciu go do przetwarzania dokumentów HTML. Dodatkowo opracowaliśmy metodę analizy powtarzającego się wzorca HTML w celu odnalezienia fragmentów o określonym znaczeniu z punktu widzenia poszukiwanego wyniku.

Implementacja

Algorytm WrapIT został także zaimplementowany w języku Java dając możliwość dokładnego przetestowania jego działania w praktyce i obiektywnej oceny jego skuteczności. Dodatkowo powstał komponent uruchomieniowy dla wygenerowanych wrapperów możliwy do wykorzystania w systemie Carrot^2 jako element wejściowy dostarczający snippety dokumentów do dalszego przetwarzania.

Możliwości dalszego rozwoju

W trakcie pracy nad algorytmem zauważyliśmy możliwości zwiększenia jego funkcjonalności, jednak ze względu na objętość oraz ograniczenia czasowe nie mieliśmy możliwości ich rozwinięcia. Przedstawimy teraz podstawowe problemy, które mogą być interesujące z punktu widzenia dalszego rozwoju algorytmu:

a. **Automatyczna analiza sposobu odpytywania serwisu** - w obecnej wersji wejściem dla algorytmu jest pełny opis serwisu wyszukującego, wraz z pełnym opisem parametrów wymaganych do odpytania takiego serwisu. Idealne rozwiązanie byłoby takie, kiedy użytkownik wpisywałby tylko adres serwisu, a algorytm sam wykrywałby formularz, który należy wypełnić w cesu uaktywnienia wyszukiwania, jak i zbadałby wymagane parametry. Umożliwiłoby to zachowanie automatyki przetwarzania także w przypadkach zmiany sposobu wywołania serwisu.
b. **Możliwość działania na wielu zapytaniach** - obecnie algorytm generuje wrapper na podstawie pojedynczej strony wynikowej. Większe możliwości dawałaby generacja dla szerszego spektrum zapytań wraz ze specjalnym algorytmem inteligentnie łączącym uzyskane wyniki.

Mimo tych nadal otwartych problemów możemy stwierdzić, iż założone w naszej pracy cele zostały uzyskane.

Bibliografia

[HotBot, 03]

Serwis wyszukujący HotBot. http://www.hotbot.com.

[Netoskop, 03]

Serwis wyszukujący Netoskop. http://www.netoskop.pl.

[Szukacz, 03]

Serwis wyszukujący Szukacz.pl. http://www.szukacz.pl.

[Google, 03]

Serwis wyszukujący Google. http://www.google.com.

[Google Groups, 03]

serwis wyszukujący grup dyskusyjnych Google. http://www.google.com.

[Yahoo, 03]

Wyszukiwarka i katalog Yahoo. http://www.yahoo.com.

[AllTheWeb, 03]

AllTheWeb search engine. http://www.alltheweb.com.

[AllTheWeb FTP, 03]

wyszukiwarka zasobów FTP. http://www.alltheweb.com/?cat=ftp.

[CiteSeer, 03]

Scientific Literature Digital Library. http://citeseer.nj.nec.com.

[TouchGraph, 03]

TouchGraph GoogleBrowser. http://www.touchgraph.com/TGGoogleBrowser.html.

[Questlink, 03]

Wyszukiwarka części elektronicznych. http://www.questlink.com.

[Altavista, 03]

Altavista search engine. http://www.altavista.com.

[Netscape, 03]

Netscape search engine and dictionary. http://search.netscape.com.

[MSN, 03]

Microsoft Network search engine. http://www.msn.com.

[Lycos, 03]

Lycos search engine. http://www.lycos.com.

[LookSmart, 03]

LookSmart search engine. http://www.looksmart.com.

[AltaVista Prisma, 03]

Serwis wyszukujący AltaVista Prisma. http://www.altavista.com/prisma.

[Teoma, 03]

Serwis wyszukujący Teoma. http://www.teoma.com.

[Vivisimo, 03]

Vivisimo document clustering engine. http://www.vivisimo.com.

[Carrot2, 03]

Strona domowa projektu Carrot^2 .
http://www.cs.put.poznan.pl/dweiss/carrot/index.php/index.xml.

[REP, 03]

Protokół Wykluczania dla Robotów. http://www.robotstxt.org.

[Baeza-Yates et al., 99]

Richardo Baeza-Yates i Berthier Ribeiro-Neto. *Modern Information Retrieval*. ACM Press. 0-201-39829-X.

[Selberg, 99]

Eric W. Selberg. *Towards Comprehensive Web Search*. Doctoral Dissertation, University of Washington, 1999.

[Lawrence & Giles, 98]

Lawrence S., Giles C. L.: Searching the World Wide Web. Science, 280:98-100, April 1998.

[Zamir & Etzioni, 99]

Oren Zamir i Oren Etzioni. *Grouper: A Dynamic Clustering Interface to Web Search Results*. WWW8/Computer Networks, Amsterdam, Netherlands, 1999.

[Kushmerick, 97]

Nicholas Kushmerick. *Wrapper Induction for Information Extraction*. PhD dissertation, University of Washington, 1997..

[XML, 00]

W3C. . *Extensible markup language (XML) 1.0 (Second edition), W3C Recomendation*. http://www.w3.org/TR/REC-xml.

[Weiss, 01]

Dawid Weiss. *A Clustering Interface for Web Search Results in Polish and English*. Master Thesis, Poznan University of Technology, 2001.

[Papakonstantinou et al., 95]

Y. Papakonstantinou, H. Garcia-Monlina, i J. Widom. *Object exchange across heterogenous information sources*. Proc. 11th Int. Conf. Data Engineering, strony 251-60, 1995.

[Roth&Schwartz, 97]

M. Roth i P. Schwartz. *Don't scrap it, wrap it! A wrapper architecture for legacy data sources*.. Proc. 22nd VLDB Conf., strony 266-75, 1997.

[Agrawal&Sirkant, 94]

R. Agrawal i R. Sirkant. *Fast algorithms for mining association rules*. Proc. 20nd VLDB Conf., strony 487-499, 1994.

[Childovskii et al., 97]

B. Childovskii, U. Borghoff, i P. Chevalier. *Towards sophisticated Wrapping of Web-based Information Repositories*. Proc. Conf. Computer-Assisted Information Retrieval, strony 123-35, 1997.

[Ashish & Knoblock, 97]

N. Ashish i C. Knoblock. *Semi-automatic wrapper generation for internet information sources*. Proc. CoopIS, 1997.

[STARTS, 97]

Luis Gravano, Hector Garcia-Molina, Carl Lagoze, i Andreas Paepcke. *STARTS - Stanford Protocol Proposal for Internet Retrieval and Search*. http://www-db.stanford.edu/~gravano/starts_home.html.

[Asai et al., 02]

Tatsuya Asai, Kenji Abe, Hiroki Arimura, Hiroshi Sakamoto, i Setsuo Arikawa. *Efficient substructure discovery from large semi-structured data*. SIAM SDM'02, April 2002.

[Abiteboul et al., 97]

S. Abiteboul, D. Quass, J. McHugh, J. Widom, i J. Wiener. *The Lorel query language for semistructured data*. Intl. J. on Digital Libraries, 1997.

[Lin & Ho, 02]

Shian-Hua Lin i Jan-Ming Ho. *Discovering Informative Content Blocks from Web Documents*. SIGKDD'02, July 2002.

[Shannon, 48]

C. Shannon. *A Mathematical Theory of Communication*. Bell Systems Technical Journal, July and October 1948.

[Salton, 89]

Gerald Salton. *Automatic Text Processing*. Addison-Wesley. 0-201-12227-8.

[Weiss, 03]

Dawid Weiss. *Carrot^2 Developers Guide*. http://www.cs.put.poznan.pl/dweiss/carrot/site/developers/manual/manual.pdf.

[Osiński, 03]

Stanisław Osiński. *An Algorithm for Clustering of Web Search Results*. Praca magisterska, Politechnika Poznańska, 2003.

A Zawartość dołączonej płyty CD

Na dołączonej do tej pracy płycie CD zamieściliśmy aktualne kody źródłowe Naszej implementacji algorytmu WrapIT. Źródła te są dostawrczone wraz z całym bieżącym kodem systemu Carrot^2. Dodatkowo zamieściliśmy wersję skompilowaną systemu Carrot^2, gotową do natychmiastowego wykorzystania i przetestowania. Na płycie znajdują się również elektroniczne wersje wybranych artykułów, prac i dokumentów wspominanych w tej pracy. Umieszczona została oczywiście także wersja elektroniczna naszej pracy.

Szczegółowa zawartość płyty CD:

- **/dokumenty/** - wersje elektroniczne dokumentów wykorzystanych w pracy
    - *ashish97-coopis.pdf* - Naveen Ashish and Craig Knoblock: Semi-automaticWrapper Generation for Internet Information Sources
    - *developers guide.pdf* - Dawid Weiss: Carrot^2 Developers guide
    - *grouper.pdf* - Oren Zamir and Oren Etzioni: Grouper: A Dynamic Clustering Interface to Web Search Results
    - *infodisc.pdf* - Shian-Hua Lin and Jan-Ming Ho: Discovering Informative Content Blocks from Web Documents
    - *kushmerick-aij2000_skrocony.pdf* - Nicholas Kushmerick: Wrapper induction: Efficiency and expressiveness
    - *kushmerick-phd.pdf* - Nicholas Kushmerick: Wrapper Induction for Information Extraction
    - *muslea00.pdf* - Ion Muslea, Steven Minton, Craig A. Knoblock: Hierarchical Wrapper Induction for Semistructured Information Sources
    - *sdm02-10.pdf* - Tatsuya Asai, Kenji Abe,Shinji Kawasoe,Hiroki Arimura, Hiroshi Sakamoto,and Setsuo Arikawa: Efficient Substructure Discovery from Large Semi-structured Data
    - *thesis_print.pdf* - Dawid Weiss: A CLUSTERING INTERFACE FOR WEB SEARCH RESULTS IN POLISH AND ENGLISH

- **/prezentacja/** - prezentacja PowerPoint dotycząca algorytmu WrapIT (w j. angielskim)
- **/tekst/** - elektroniczna wersja tej pracy
    - *praca.pdf* - pełny tekst pracy (pdf)
    - **src/** - źródła tekstu pracy (DOCBOOK XML)
- **/uruchomienie/** - gotowa do uruchomienia skompilowana wersja systemu Carrot^2
    - *run_carrot.bat* - skrypt uruchamiający serwer systemu Carrot^2
    - *run_wrapit.bat* - skrypt uruchamiający algorytm *WrapIT*
    - **j2sdk1.4.1_02/** - Sun Java SDK - wymagany do uruchomienia systemu Carrot^2
    - **runtime/** - Serwer aplikacyjny TomCat z zainstalowanymi komponentami Carrot^2
- **/zrodla/** - kody źródłowe całego systemu Carrot^2 - aktualny na dzień tworzenia tej płyty obraz CVS
    - *build.xml* - skrypt **ant** umożliwiający kompilację całego systemu Carrot^2
    - **carrot2/components/** - źródła komponentów Carrot^2
        - **inputs/treeSnippetMiner/src/** - źródła implementacji algorytmu WrapIT oraz komponentu go wykorzystującego w systemie Carrot^2
            - **com/paulodev/carrot/treeExtractor** - źródła komponentu wejściowego WrapIT odpowiedzialnego za dostarczanie online wyników do systemu Carrot^2
            - **com/paulodev/carrot/treeSnippetMiner** - źródła implementacji algorytmu WrapIT
                - *MainApp.java* - klasa uruchomieniowa aplikacji generującej wrappery
                - *TestApp.java* - klasa uruchomieniowa aplikacji testującej działanie wrappera
                - **frequentTreeMiner** - część generatora odpowiedzialna za pierwszą fazę algorytmu WrapIT - odnalezienie częstego wzorca HTML

- **treeAnalyser** - część generatora odpowiedzialna za drugą fazę algorytmu WrapIT - analizę częstego wzorca HTML pod kątem znaczenia poszczególnych jego części

- **carrot2/descriptors/** - deskryptory komponentów i procesów przetwarzania Carrot^2
- **carrot2/lib/** - zewnętrzne biblioteki wykorzystywane przez Carrot^2