Spis treści

Rozdział 1

Opis problemu

Praca ma na celu zbadanie problemu wyliczania rankingu dokumentów w sieciach społecznościowych w czasie wyszukiwania wyników zapytania. Badanie w tej pracy sieci społecznościowe charakteryzują się tym,że zasoby przechowywane przez nie zostały przez różnych użytkowników wybranymi etykietami (zwanymi również annotacjami czy tagami). Jako dodatkowy element, w czasie wyszukiwania dokumentów zostaną użyte dane mówiące o popularności danych wyników, pozyskane z rożnych serwisów społecznościowych, takich jak twitter, facebook czy digg. Dane te powinny polepszyć jakoś wyszukiwania i przyśpieszyć obliczenia potrzebne do wyliczenia statycznych rankingów dokumentów.

Rozdział 2

State of the art

Obecnie prowadzonych jest wiele badań nad sieciami społecznymi używającymi etykiet (tzw. tagów, czy adnotacji) do porządkowania zasobów użytkownika. Część z nich za cel stawia sobie ulepszenie algorytmów wyszukiwania dokumentów, inne używają tych zasobów dla personalizacji wyników dla użytkowników, rekomendacji dokumentów, użytkowników, czy tagów w czasie procesu etykietowania.

Autorzy prac [9] i [5] analizują użyteczność etykietowania dokumentów przy wyszukiwaniu. Ich analizy bazują głównie na danych systemu delicious. Wnioski z tych prac są pozytywne. Zwracają one uwagę na korelacje miedzy popularnością stron w serwisie, a ich rankingiem według algorytmu PageRank. Dodatkowo wskazują na problemy z nowymi dokumentami dodanymi do sieci, z którymi nie radzą sobie algorytmy bazujących na odnośnikach pomiędzy dokumentami, a które mogą być rozwiązane przez strony takie jak delicous. Również rozwiązanie problemu nowych danych jest zaproponowane w pracy [2]. W tym artykule, jako rozwiązania, autorzy proponują użycie informacji pobranych z systemu twitter, który służącego do mikro-blogowania. Pod uwagę brane są krótkie informacje umieszczane przez użytkowników zawierające odnośniki do dokumentów jak również inne informacje dostępne w profilu użytkownika.

Autorzy pracy [3] zaprezentowali formalny model folksonomi i algorytmy AdaptedPageRank i jego bardziej spersonalizowana wersję algorytm FolkRank. Oba te algorytmy bazują na metodzie PageRank. Algorytm ten pozwala na rekomendacje tagów dla użytkowników jak również na ustalenie rankingu w wyszukiwanych elementach.

W pracy [1] przedstawione zostały algorytmy SocialSimRank i SocialPageRank. Algorytm SocialPageRank jest statycznym rankingiem zasobów, opartym również o ideę algorytmy PageRank. Algorytm ten bierze pod uwagę ilość tagów wskazujących na dany dokument jak również różną wagę opisujących dokument etykiet. Drugim proponowanym w tej pracy algorytmem jest SocialSimRank, który estymuje podobieństwo między używanymi

tagami, a następnie wyniki te są używane dla wyliczanie podobieństwa między zapytaniem użytkownika a tagami przypisanymi do danego zasobu.

W pracach [8] i [6] autorzy biorą pod uwagę całą sieć społecznościową użytkownika, dodając dodatkowo znajomości użytkowników i wykorzystują te dane dla przedstawienia spersonalizowanych wyników. W artykule [8] opisywany jest algorytm ContextMerge, który wykonuje wyszukiwanie biorąc pod uwagę zależności między użytkownikami. Dodatkowo pozwala na dynamiczne dodawanie nowych wyników do odpowiedzi dla uzyskania k. W pracy [6] autorzy zaproponowali system, który łączy wyniki wyszukiwarki z danymi użytkownikami aplikacji takich jak delicous. Jako bazowa wyszukiwarka może być użyta dowolna aplikacja, która zwraca dokumenty wraz z przypisanym do nich rankingiem. Wyniki te są następnie przeliczane za pomocą danych uzyskanych od użytkownika, czyli ze zbiorem dokumentów i tagów opisanych przez niego i innych użytkowników. Jako rezultat otrzymujemy nowy, bardziej spersonalizowany ranking dokumentów.

W pracy [7] celem autorów było spersonalizowanie wyników wyszukiwania w dokumentach z tagami. Do tego celu stworzone zostały dwa modele: użytkownik-tag, który ukazuje jak użytkownik użył tagi podobne do wybranego tagu. Drugim modelem jest model tag-zasób opisujący w jaki sposób dokumenty podobne do danego zasobu zostały opisane etykietami.

Probabilistyczne podejście do rankingu dokumentów przy użyciu tagów zostało zaprezentowane w pracy [10] i [4]. W [10] autorzy zaprezentowali model probabilistyczny dla generowania etykiet i zależnych im dokumentów i wyszukiwania ich tematów. W [4] autorzy prezentują metodę rankingu tagów w zależności od ich tematu i konstruują graf przejścia między tagami należącymi do różnych tematów. Metoda ta jest wykorzystana następnie dla rekomendacji tagów dla użytkownika w czasie opisywania dokumentów.

Rozdział 3

Opis algorytmów

3.1 Folksonomia

Folksonomia nazywamy społeczne klasyfikowanie czy tagowanie różnych zasobów. Formalnie model folksonomii został przedstawiony w pracy [3] i zdefiniowany został jako:

Definicja 1. *Folksonomia jest to krotka* $F := (U, T, R, Y,)$*, gdzie:*

- U*,*T *i* R *są to zbiory skończone, których elementy nazywają się odpowiednio użytkownicy, tagi i dokumenty*
- Y *jest relacją między tymi elementami:* $Y \subseteq U \times T \times T$*, nazywamy ją relacją przypisania tagu.*

Użytkownicy i tagi identyfikowani są na podstawie ich unikalnych nazw własnych i identyfikatorów. Dokumenty mogą być różnymi danymi: stronami www, zdjęciami, plikami np: pliki pdf. Ta praca bazuje na danych pobranych z witryny delicous, które w zdecydowanej większości są stronami www. Dane, które nie są stroną www nie są brane pod uwagę w tej pracy.

3.2 Opis metod rankingu dokumentów

Do wyliczenia rankingu dokumentów zostały użyte kilka metod.

AdaptedPageRank i SocialPageRank, są to algorytmy którę biorą pod uwagę popularność tagów, użytkowników, dokumentów i relację między nimi.

Kolejnym użytym w pracy rankingiem użytym w pracy jest wersja algorytmu tf-idf użyta we frameworku Lucene.

Kolejny ranking stanowią dane z serwisów takich jak facebook, twitter czy digg mówiące o ilości udostępnień danego zasobu przez użytkowników tych serwisów.

3.3 SocialPageRank

3.3.1 Opis

Ilość tagów przypisanych do danej strony może świadczyć o jej popularności i jakości. Jednak, tradycyjne algorytmy wyliczające statyczny ranking strony nie są w stanie poprawnie wyliczyć popularności strony oparciu o posiadane dane. Na przykład, istnieją strony, posiadające bardzo dużą liczbę opisanych tagów, jednak posiadają ranking PageRank równy zero. Dodatkowo nie można tylko brać pod uwagę ilość tagów, różne tagi mogą mieć różną wagę na wyliczanie popularności danej strony.

Można zauważyć następującą relację między aktywnymi użytkownikami, popularnymi tagami i stronami o wysokiej jakości:

Obserwacja 1. *Popularne strony są dodawane przez wielu aktywnych użytkowników i opisywane popularnymi tagami. Aktywni użytkownicy lubią dodawać strony o wysokiej jakości i opisywać je popularnymi tagami. Popularne tagi są używane do opisywania popularnych stron przez aktywnych użytkowników.*

Bazując na powyższej obserwacji stworzony został algorytm SocialPageRank, służący do wyliczania rankingu strony w systemie opartym na socjalnych annotacjach. Algorytm działa na zasadzie wzmacnia wzajemnych relacji między użytkownikami, tagami i opisanymi zasobami.

Załóżmy że posiadamy N_T tagów, N_D stron internetowych i N_U użytkowników. Niech M_{DU} będzie $N_D \times N_U$ macierzą asocjacyjną między zasobami i użytkownikami, M_{UT} będzie macierzą $N_U \times N_T$ asocjacyjną między użytkownikami i tagami i niech M_{TD} będzie macierzą $N_T \times N_D$ asocjacyjną miedzy tagami a dokumentami. Macierze te zostały wypełnione w następujący sposób:

- W komórce macierzy$M_{DU}(d_n, u_k)$ znajduje się wartość będąca ilością tagów przypisanych do dokumentu d_n przez użytkownika u_k.
- elementy $M_{UT}(u_k, t_n)$ to ilość dokumentów opisanych tagiem t_n przez użytkownika u_k,
- elementy $M_{TD}(t_n, d_k)$: ile użytkowników dodawało dokument d_k i oznaczyło go tagiem t_n.

Niech P_0 będzie wektorem zainicjalizowanym losowymi wartościami z przedziału $[0, 1]$. Jest on pierwszym przybliżeniem rankingu dokumentów.

3.3.2 Algorytm

Dane wejściowe:

Macierze asocjacyjne M_{DU}, M_{UT} ,M_{TD} i zainicjalizowany wektor P_0.

repeat
$U_i = M_{DU}^T * P_i$
$T_i = M_{UT}^T * U_i$
$P'_i = M_{TD}^T * T_i$
$T'_i = M_{TD} * P'_i$
$U'_i = M_{UT} * T'_i$
$P_{i+1} = M_{DU} * U'_i$
until wartości wektora P_n nie zbiegną

Wektory U_i, T_i i P_i opisują popularność użytkowników, tagów i dokumentów w i-tej iteracji. U'_i, T'_i i P'_i są wartościami pośrednimi. Z algorytmu powyżej wynika, że popularność użytkownika może zostać wyznaczona z popularności stron, które zostały przez nich opisane. Popularność adnotacji wyliczana jest z popularności użytkowników. Następnie w algorytmie 'popularność' jest przenoszona między tagami, do stron, od stron do użytkowników użytkowników i na koniec od wektor 'popularności' przenosi się od użytkowników do dokumentów. Wartości wektora P_n zbiegają do wektora P^*, który jest wynikiem działania algorytmu SocialPageRank.

Złożność

Złożoność czasowa każdej iteracji wynosi $O(N_u * N_d + N_t * N_d + N_t * N_u)$. Ponieważ macierze utworzone z posiadanych danych charakteryzują się tym, że są bardzo rzadkie, czas wykonywania, jest mniejszy. Należy również pamiętać, że dane w sieci internetowej rosną bardzo szybko i czas trwania całych obliczeń będzie również szybko rosną wraz ze zwiększającym się zbiorem danych,

Dowód zbieżności algorytmu

[TODO]

3.4 Adapted PageRank

3.4.1 Opis

Algorytm Adapted PageRank podobnie jak algorytm Social PageRank został zainspirowany algorytmem PageRank. Algorytm ten od Algorytmu Social PageRank różni się sposobem tworzenia macierzy. Zamiast używania trzech macierzy, używana jest jedna, utworzona z grafu $G_f = (V, E)$.

Ponieważ algorytm PageRank nie może być bezpośrednio zastosowany do zebranych danych autorzy algorytmu Adapted PageRank zmienili strukturę danych na nieskierowany graf trzydzielny $G_f = (V, E)$.

Proces tworzenia grafu G_f:

1. zbiór wierzchołków V powstaje z sumy rozłącznej zbioru użytkowników U, tagów T, i dokumentów D: $V = U \cup T \cup D$

2. wszystkie wystąpienia łącznie tagów, użytkowników, dokumentów stają się krawędziami grafu G_f: $E = \{\{u,t\},\{t,d\},\{d,u\}|(u,t,d) \in Y\}$. Wagi tych krawędzi są przydzielane w następujący sposób: każda krawędź $\{u,t\}$ ma wagę $|\{d \in D : (u,t,d) \in Y\}|$, czyli jest to ilość dokumentów, którym użytkownik u nadał annotacje t. Analogicznie dla krawędzi $\{t,d\}$: $waga = |\{u \in U : \{(u,t,d) \in Y\}\}|$ i krawędzi $\{d,u\}$, gdzie $waga = |\{t \in T : \{(u,t,d) \in Y\}\}|$. Wagi te są analogiczne do wartości w macierzach w algorytmie Social PageRank.

Oryginalny algorytm PageRank bazuje na idei, że strona jest ważna jeśli dużo stron ma do niej odnośniki, i te strony są również ważne. Rozłożenie wag może zostać przedstawione jako punkt stały procesu przekazywania wag grafu sieci www. W tym algorytmie został zaimplementowana podobno analogia przeniesiona na folksonomie: strony, które zostały opisane ważnymi tagami, przez ważnych użytkowników stają się ważne. Analogiczna zasada odnosi się tagów i użytkowników. Dzięki temu otrzymujemy graf, w którym zależności między wierzchołkami wzajemnie się wzmacniają w czasie rozprzestrzenienia się wag.

Jak w algorytmie PageRank, tutaj również został użyty model losowego internauty (random surfer model), czyli definicje ważności strony opierającą się o idealnego losowego internautę, który najczęściej przechodzi przez strony internetowe używając odnośników. Jednak od czasu do czasu, internauta przeskoczy losowo do nowej strony, nie przechodząc po żadnym z odnośników. Używając tego założenia dla folksonomii otrzymujemy:

$$\vec{w} = \alpha\vec{w} + \beta A\vec{w} + \gamma\vec{p} \tag{3.1}$$

gdzie A jest prawo stochastyczna macierz sąsiedztwa grafu G_f, $\vec{p}$ jest wektorem preferencji, odpowiadającym za losowego internautę. Wektor preferencji używany jest do obliczeń spersonalizowanych wyników. W przypadku algorytmu Adapted PageRank $\vec{p} = 1$. $\alpha, \beta i \gamma$ to stale, gdzie: $\alpha, \beta, \gamma \in [0,1]$ i $\alpha + \beta + \gamma = 1$. Stała α wpływa na prędkość zbieżności algorytmu, $\beta i \gamma$ regulują wpływ wektora preferencji na obliczenia.

3.4.2 Algorytm

Dane wejściowe:

A – prawo stochastyczna macierz sąsiedztwa grafu G_f
p – wektor preferencji
w – losowo zainicjalizowany wektor

repeat
$w_{n+1} = \alpha * w_n + \beta * A * w_n + \gamma * p$
until wartości wektora w_n nie zbiegną

wynikiem algorytmu jest wektor $\vec{w}$.

Dane zostały uzyskane dla paremetrów: $\alpha = 0.35, \beta = 0.65, \gamma = 0$ i pochodzą z pracy [3].

W algorytmie adapted page rank wektor preferencj i $p = 1$.

3.4.3 Algorytm FolkRank

Algorytm FolkRank jest wersją algorytmu Adapted PageRank, który daje różne wyniki w zależności od tematu, czyli wektora preferencji p. W poprzednim algorytmie wszystkie pola były ustawione na 1. Wektor ten może być ustalony na wybrany zbiór tagów, użytkowników, dokumentów, albo na pojedynczy element. Wybrany temat będzie propagowany na resztę dokumentów, tagów i użytkowników. Można dzięki temu, ustając wagę na zapytanie albo konto użytkownika systemu uzyskać wyniki bardziej zbliżone do zainteresowań danego użytkownika.

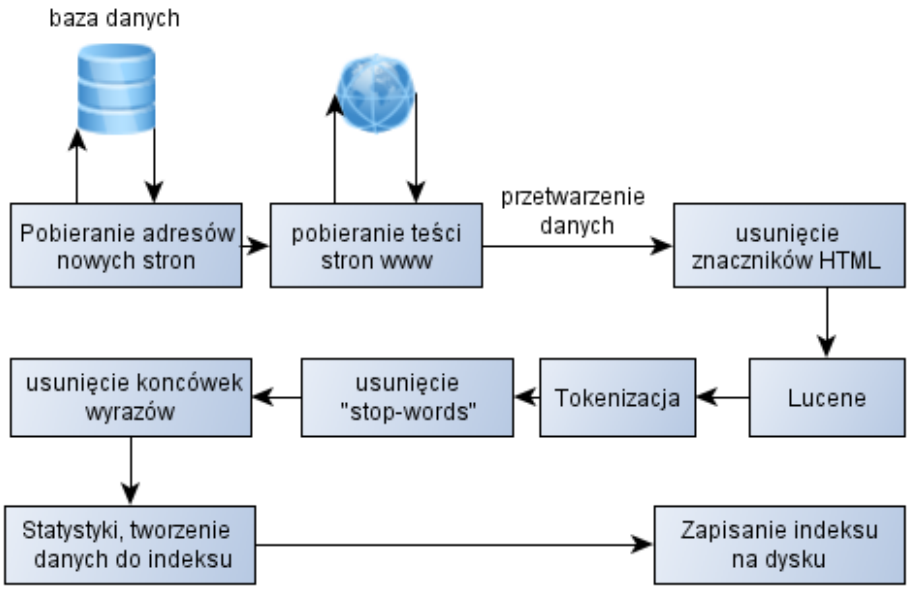
Algorytm FolkRank może pomóc w analizie zbioru danych, albo w systemach nie działających w czasie rzeczywistym, ale nie jest użyteczny w zaprezentowanym systemie. Nie jest możliwy do wykorzystania z powodu długiego czasu obliczania wag dokumentów.

3.5 Lucene

Lucene jest biblioteką napisaną w Javie. Biblioteka ta jest w stanie indeksować dużą ilość dokumentów z różnych źródeł i przeprowadzać szybkie wyszukiwania w tych tekstach. W opisywanej aplikacji, framework Lucene przechowuje źródła stron. Strony te zostały pobrane z serwisu delicous i zapisane w bazie danych.

3.5.1 Pobieranie stron

W pewnych odstępach czasu, wątek odpowiedzialny za indeksowanie stron sprawdza czy w bazie danych tabela DOCUMENT nie ma informacji o nowych wpisach. Z bazy danych pobrane są informacje o adresach tych stron. Następnie dla każdego adresu URL zostaje pobrana treść strony na którą wskazuje. Strona WWW następnie zostaje oczyszczona ze znaczników HTML, i przekazane do frameworku lucene do zindeksowania. Jeśli wszystkie czynności zakończą się powodzeniem, w bazie danych zostaje odnotowana informacja o posiadaniu na dysku danego dokumentu. Rysunek 3.1 przedstawiony jest cały proces pobierania i przetwarzania danej strony.



Rysunek 3.1: Lucene: pobieranie danych i indeksowanie

Do Lucene zapisywane są następujące informacje: identyfikator *id* dokumentu z bazy danych, oraz przetworzony tekst strony WWW. Przechowywanie identyfikatora dokumentu w danych Lucene pozwala późniejsze powiązanie wyników wyszukiwania z odpowiednim rekordem w bazie danych.

W czasie indeksowania biblioteka Lucene wykonuje wiele czynności które pozwalają jej później szybko wyszukiwać informację. Główne z nich to:

- Tekst zostanie przetworzony na ciąg termów,
- usunięcie końcówek wyrazów,
- usunięcie 'stop-words' z tekstu, czyli słów nie mających dużego znaczenia przy wynikach wyszukiwania, takich jak: i, lub, ...
- obliczenie statystyk, np: wystąpienia słów w dokumencie, odległości od siebie

Czas wyszukiwania zapytania w dokumentach przechowywanych Lucene jest szybkie. Przy małej, poniżej 1GB danych, wyszukiwanie następuje praktycznie w czasie rzeczywistym. Zapytanie jest przekazywane do frameworku, w którym jest one przekształcane na termy. Dla zapytania q i dla każdego dokumentu d wyliczana jest wartość funkcji $score(q, d)$. Wynikiem są dokumenty posortowane wg. wyniku tej funkcji.

$score(q,d) = coord(q,d) * queryNorm(q) * \sum_{t \text{ in } q} \Big(tf(t \text{ in } d) * idf(t)^2 * getBoost(t) * norm(t,d))$

gdzie:

- coord(q,d): funkcja zwraca wartości zależne od miejsca występowania i odległości od siebie szukanych termów w dokumencie.
- queryNorm(q): funkcja normalizująca wyniki zapytania
- tf(t in d): funkcja wyliczająca częstość występowania danego termu w dokumencie
- idf(t) : funkcja wyliczająca częstość występowania termu we wszystkich dokumentach.
- getBoost(t) - Lucene pozwala na zwiększenie wagi niektórych termów. Nieużywane w tej aplikacji.

Lucene ocenia dokumenty głównie na podstawie funkcji TF-IDF. Każdy dokument reprezentowany jest przez wektor, składający się z wag słów występujących w tym dokumencie. TFIDF informuje o częstości wystąpienia termów uwzględniając jednocześnie odpowiednie wyważenie znaczenia lokalnego termu i jego znaczenia w kontekście pełnej kolekcji dokumentów.

3.6 Inne rankingi

Rankin używający danych z digg'a, twittera i facebooka Propozycja rankingu (prosta):

- d_i - ilość użytkowników udostepniających dokument i w serwisie digg
- f_i - ilośc użytkowników udostępniających dokument i w serwisie facebook
- t_i - ilość użytkowników udostępniających dokument i w serwisie twitter
- $rank_i$ - ranking dokumentu i

powyższe wartości zostały znormalizowane.
$rank_i = d_i + f_i + t_i$

Rozdział 4

Implementacja

4.1 Opis Aplikacji

Aplikację można podzielić na 3 główne części:

- iterface użytkownika
- główna część, odpowiedzialna za wyliczanie algorytmów,
- crawlery zbierające nowe dane

W pierwszej części odbywa się wykonywanie zapytań, wyliczenie ostatecznego rankingu dokumentów i wyświetlanie odpowiedzi użytkownikowi. Część algorytmiczna zawiera głównie implementacje algorytmów Social PageRank, Adpated PageRank jak również odpowiada za wykonywanie wcześniejszych obliczeń (preprocessing), które przyśpieszają późniejsze wykonanie algorytmów.

Ostatni fragment aplikacji, czyli crawlery można również podzielić na:

- pobierające nowe dane z serwis delicous i odświeżające zapisane poprzednio dane,
- crawler odpowiedzialny za pobieranie treści dokumentów i zapisywanie ich przy użyciu frameworku Lucene.

Wszystkie wymienione poprzednio fragmenty aplikacji korzystają z bazy danych, która bardziej szczegółowo zostanie opisana w kolejnych rozdziałach.

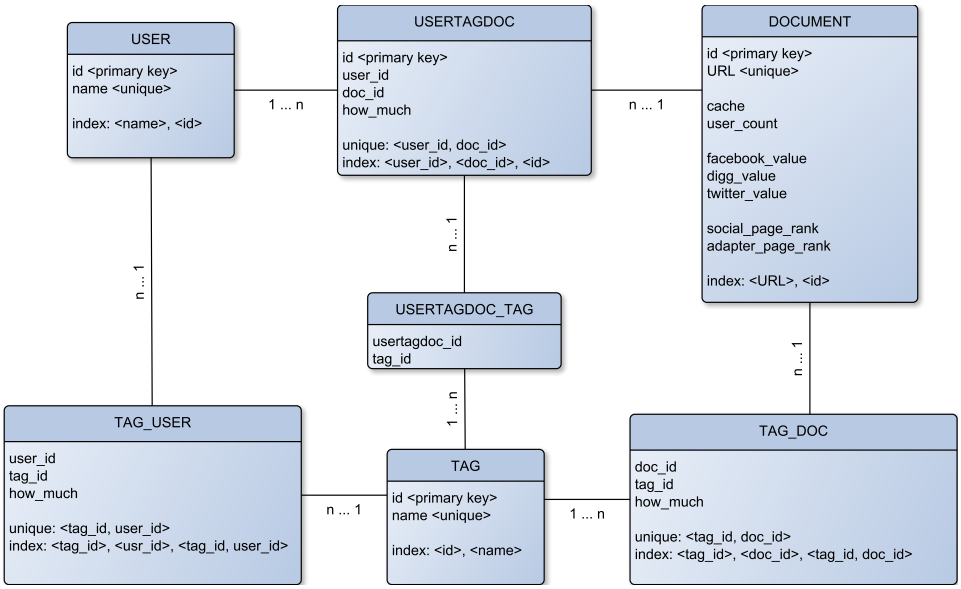
4.2 Baza danych

4.2.1 System zarządzania baza danych

Jako serwer bazy danych został użyty MySQL serwer z silnikiem bazy danych InnoDB. MySQL został wybrany głównie z powodu swojej szybkości

działania jak również powodu prostoty używania. Dodatkowe funkcjonalności bazy danych rozbudowane w innych systemach zarządzania takich jak: PostgreSQL i Oracle, czyli na przykład: funkcje, triggery, nie są używane w tej aplikacji.

4.2.2 Tabele



Rysunek 4.1: Schemat bazy danych

Baza danych składa się z trzech głównych tabel: USER, DOCUMENT i TAG. Zawierają one informację na temat, odpowiednio, użytkowników, dokumentów i adnotacji pobrane z serwisu delicous. W bazie danych znajdują się też tabele: USERTAGDOC i USERTAGDOC_TAG, które służą do zapisania relacji między użytkownikami a dokumentami (USERTAGDOC) i adnotacjami ich opisującymi (USERTAGDOC_TAG). W tych tabelach zapisane są informacje na temat tego czy dany użytkownik dodał dokument do serwisu i jakimi tagami została dana strona opisana.

4.2.3 Pomocnicze tabele i pola. Indeksy

Dodatkowo w bazie danych (rys 4.1)znajdują się dwie tabele: TAG_DOC i TAG_USR. W tabelach tych zapisywane są dane wyliczone z pozostałych tabel. W tabeli TAG_DOC znajdują się informację na temat tego ilu użytkowników dodało dany dokument doc_i i opisało go tagiem tag_k. Odpowiednia w tabeli TAG_USR znajdują się informację na temat ilości różnych dokumentów dodanych przez użytkownika usr_n i opisanych tagiem tag_m. Dane

ilości różnych tagów którymi użytkownik usr_l opisał dokument doc_j przechowywane są w już istniejącej tabeli USERTAGDOC.

Tabele TAG_DOC, TAG_USR i pole how_much w tabeli USERTAGDOC zostają wypełnione w czasie preprocessingu. Dane te posłużą później do utworzenia macierzy, używanych przez algorytmy Social PageRank i Adapted PageRank.

W tabelach na różnych polach zostały dodane indeksy. Przyśpieszają one działanie aplikacji, pozwalają na szybsze operacja przy często używanych polach. Główne indeksy zostały zaznaczone na rysunku 4.1

4.2.4 Listing

Poniżej znajduje się listing zapytania SQL tworzącego tabele w bazie danych.

Listing 4.1: Skrypt tworzący tabele w bazie danych

```
CREATE TABLE `DOCUMENT` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `url` varchar(255) NOT NULL,
  `digg_value` int(11) DEFAULT '0',
  `facebook_value` int(11) DEFAULT '0',
  `twitter_value` int(11) DEFAULT '0',
  `page_fetch` tinyint(1) DEFAULT '0',
  `tag_count` bigint(20) DEFAULT NULL,
  `user_count` bigint(20) DEFAULT NULL,
  `cache` text,
  `adapted_page_rank` double DEFAULT NULL,
  `social_page_rank` double DEFAULT NULL,
 PRIMARY KEY (`id`),
 UNIQUE KEY `url` (`url`)
)

CREATE TABLE `TAG` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `doc_count` bigint(20) DEFAULT NULL,
  `doc_dist_count` bigint(20) DEFAULT NULL,
  `tag` varchar(255) NOT NULL,
  `user_count` bigint(20) DEFAULT NULL,
  `adapted_page_rank` double DEFAULT NULL,
 PRIMARY KEY (`id`),
 UNIQUE KEY `tag` (`tag`)
)
```

```
CREATE TABLE `TAG_DOC` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `doc_id` bigint(20) DEFAULT NULL,
  `tag_id` bigint(20) DEFAULT NULL,
  `how_much` int(11) DEFAULT '1',
 PRIMARY KEY (`id`),
 UNIQUE KEY `tag_doc` (`doc_id`,`tag_id`),
 KEY `doc_id` (`doc_id`,`tag_id`),
 KEY `tag_doc_doc` (`doc_id`),
 KEY `tag_doc_tag` (`tag_id`)
)

CREATE TABLE `TAG_USR` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `user_id` bigint(20) DEFAULT NULL,
  `tag_id` bigint(20) DEFAULT NULL,
  `how_much` int(11) DEFAULT '1',
 PRIMARY KEY (`id`),
 UNIQUE KEY `tag_user` (`user_id`,`tag_id`),
 KEY `user_id` (`user_id`,`tag_id`),
 KEY `tag_usr_doc` (`tag_id`),
 KEY `tag_usr_usr` (`user_id`)
)

CREATE TABLE `USER` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `doc_count` bigint(20) DEFAULT NULL,
  `name` varchar(255) DEFAULT NULL,
  `new_data` tinyint(1) DEFAULT '1',
  `tag_count` bigint(20) DEFAULT NULL,
  `tag_dist_count` bigint(20) DEFAULT NULL,
  `adapted_page_rank` double DEFAULT NULL,
 PRIMARY KEY (`id`),
 UNIQUE KEY `name` (`name`)
)


CREATE TABLE `USERTAGDOC` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `doc_id` bigint(20) DEFAULT NULL,
  `user_id` bigint(20) DEFAULT NULL,
  `how_much` int(11) DEFAULT NULL,
 PRIMARY KEY (`id`),
```

```
 UNIQUE KEY `user_id` (`user_id`,`doc_id`),
 KEY `FKB30EFAE96714BC07` (`doc_id`),
 KEY `FKB30EFAE9520DD4E4` (`user_id`),
 CONSTRAINT `FKB30EFAE9520DD4E4` FOREIGN KEY (`user_id`) REFERENCES
`user` (`id`),
 CONSTRAINT `FKB30EFAE96714BC07` FOREIGN KEY (`doc_id`) REFERENCES
`document` (`id`)
)


CREATE TABLE `USERTAGDOC_TAG` (
  `USERTAGDOC_id` bigint(20) NOT NULL,
  `tags_id` bigint(20) NOT NULL,
 KEY `FK4C01D124CA702D03` (`tags_id`),
 KEY `FK4C01D1243D83CB28` (`USERTAGDOC_id`),
 CONSTRAINT `FK4C01D1243D83CB28` FOREIGN KEY (`USERTAGDOC_id`)
REFERENCES `usertagdoc` (`id`),
 CONSTRAINT `FK4C01D124CA702D03` FOREIGN KEY (`tags_id`) REFERENCES
`tag` (`id`)
)
```

4.3 Zbieranie danych

[TODO - OPIS CRAWLERÓW]

4.3.1 Czyszczenie tagów przed zapisem

Tagi pobierane z serwisu delicous nie zawsze są w postaci wymaganej przez aplikacje. Spowodowane to jest błędami użytkowników, czy też specyficznym stylem zapisywania tagów.

Niektóre tagi mają różne znaczenie w zależności od kontekstu, na przykład tag 'design' ma inne znaczenie w kontekście strony o programowaniu, a inne w kontekście strony o sztuce. Część użytkowników żeby poradzić sobie z tym problemem dodają do tagów informacje mówiące o ich domenie. Często domena ma wygląd 'programming@design' czy 'art#design'.

Z powodu tego specyficznego zapisu każda etykieta, przed zapisaniem do bazy danych jest dzielona na 2 lub więcej słów w miejscach występowania popularnych znaków specjalnych. Dlatego też

Dodane kontekstu do tagu mogłoby być przydatne w aplikacji, ale z powodu tego że każdy użytkownik ma swój specyficzny sposób opisywania dokumentów np: design@art i art#design, trudno jest je zunifikować. Dodatkowym problemem jest to, że nie jest to sposób opisu używany przez wszystkich użytkowników.

Adnotacje przypisywane przez użytkowników często kończą się lub zaczynają od znaków specjalnych. Jest to spowodowane np: błędami (dodatkowe przecinki) albo specyficznym stylem opisywania danych przez użytkownika. Wszystkie znaki specjalne z końca i początku dokumentu są usuwane przed dodaniem do bazy danych.

Przykłady danych przed i po ich oczyszczeniu:

- '@java' : 'java'
- '@@java' : 'java'
- '#java6@' : 'java6'
- 'design!$%@art' : 'design', 'art'
- 'art!#,': 'art'

4.4 Implementacja algorytmów Social PageRank i Adapted PageRank

Zarówno algorytm Social PageRank i Adapted PageRank wykorzystują w trakcie każdej iteracji szcześć macierzy. Sa to macierze opisane poniżej i ich traspozycje:

- M_{DU}: macierz $N_D \times N_D$ asocjacyjna między dokumentami a użytkownikami, która w każdej komórce $M_{DU}(d_m, u_k)$ zawiera ilość tagów, którymi dany użytkownik u_k opisał wybrany dokument d_m.
- M_{UT}: macierz $N_U \times N_T$ asocjacyjna między użytkownikami a tagami, zawerająca w komórce ilość dokumentów opisanych wybranym tagiem przez danego użytkownika
- M_{TD}: macierz $N_T \times N_D$ asocjacyjna między tagami a dokumentami, zawierająca w komórach liczbę użytkowników, którzy dany dokument opisali wybranym tagiem.

Algorytmy te działały na około danych składających sie z około 1,000,000 użytkowników, 600,000 dokumentów i 80,000 tagów. Macierze utworzone z tych danych są duże. Dodatkowo, w algorytmie Adapted PageRank używamy macierzy, która jest stworzona ze wszystkich 6-ciu opisanych powyżej:

$$G_f = \begin{pmatrix} 0 & M_{DU} & M_{TD}^T \\ M_{DU}^T & 0 & M_{UT} \\ M_{TD} & M_{UT}^T & 0 \end{pmatrix}$$

Problemem jest nie tylko sama wielkośc macierzy, które nie mogą zmieścić sie jednocześnie w pamięci, ale również ich zawartość. Wyliczanie danych przy każdej iteracji jest bardzo czasochłonne. Dlatego, żeby nie wyliczać zawartości macierzy przy każdej iteracji, potrzebujemy zapamiętać ich zawartość. Dodatkowo, wyliczanie choćby jednarozowe danych, jest bardzo kosztowne.

Żeby rozwiązać te problemy, napocżątku wykonywany jest preprocessing w bazie danych, wyliczający wymagane dane. Następnie, dane te są zapisywane na dysku w łatwą do użycia później strukturę danych, która zawiera tylko określoną ilość wierszy macierzy.

Pliki te później są używane jako fragmenty macierzy. Na tych fragmentach wykonywane są wymagane obliczenia. Kolejne wyniki obliczeń są następnie łączone i przekazywane do kolejnej iteracji, albo zwracane jako wynik algorytmu.

4.5 Preprocessing w bazie danych

W bazie danych wyliczone zostają informacje potrzebne do późniejszego utworzenia macierzy używanych w algorytmach Adapted PageRank i Social PageRank. Algorytmy te przy każdym przebiegu korzystają z tych samych macierzy, obliczanie ich przy każdej iteracji wymagałoby zbyt dużego nakładu czasu. Dane te są wyliczane w dwóch częściach. Na początku wyliczane są w bazie danych i zapisywane w pomocniczych tabelach. Następnie zapisywane są one do do struktury i serializowane w oddzielnych plikach. Pliki te są później używane do tworzenia macierzy.

Informacje te zostaną zapisane w tabeli USERTAGDOC w polu how_much i w tabelach TAG_USR i TAG_DOC. Wyliczenie tych informacji pozwoli później na szybszy dostęp do nich.

Czas wykonania preprocessingu w bazie danych jest różny. Jak widać w tabeli poniżej (4.1) najwięcej czasu trwało tworzenie tabeli TAG_USR i powstało w niej najwięcej nowych rekordów. Wykonywanie tych obliczeń przy każdej iteracji algorytmu spowodowałoby znacznie zwiększenie czasu działania aplikacji.

Tabela	czas wykonania polecenia	ilość powstałych rekordów
TAG_USR	FIXME	FIXME
TAG_DOC	ok 3h	FIXME
USERTAGDOC	FIXME	FIXME

Tablica 4.1: Czas wykonania wypełnienia odpowiednich tabel w bazie danych !FIXME!

Poniżej znajdują się listingi zapytań SQL obliczających pola tabeli USER-

TAGDOC 4.4 i wypełniające tabele TAG_USR (4.2) i TAG_DOC (4.3).

Listing 4.2: Skrypt dodający dane do tabeli tag_doc

```
insert into tag_doc (doc_id, tag_id, how_much)
select d.new_id, tag.new_id, count(utd.user_id)
from
tag,
usertagdoc_tag as utd_t,
usertagdoc as utd, document as d
where
d.id = utd.doc_id
and utd_t.usertagdoc_id = utd.id
and utd_t.tags_id = tag.id
group by utd.doc_id, tag.id;
```

Listing 4.3: Skrypt dodający dane do tabeli tag_usr

```
insert into tag_usr (tag_id, user_id, how_much)
select utd.user_id, tag.new_id, count(utd.doc_id)
from
tag,
usertagdoc_tag as utd_t,
usertagdoc as utd
where
utd_t.usertagdoc_id = utd.id and
utd_t.tags_id = tag.id
```

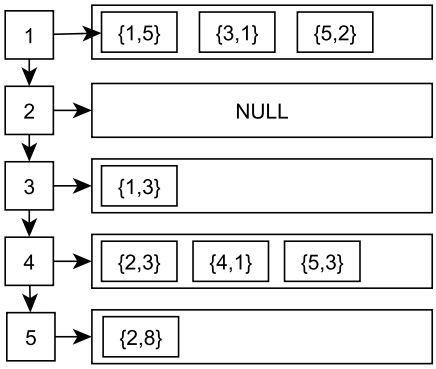
Listing 4.4: Skrypt updatujący pole how_much w tabeli usertagdoc

```
update usertagdoc utd
set how_much = (select count(distinct tags.tags_id)
from usertagdoc_tag tags
where utd.id = tags.usertagdoc_id);
```

4.5.1 Preprocessing: zapis do plików

Po wyliczeniu w bazie danych pomocniczych tabel, wyniki są pobierane przez program, a następnie zapisywane do pliku. Ponieważ pamięć maszyny ogranicza wielkość macierzy na której jesteśmy w stanie operować, tworzone pliki zawierają tylko wyznaczoną ilość wierszy. Ilość wierszy macierzy zapisanych w pliku jest konfigurowalna i zależna od przydzielonej pamięci aplikacji.

W czasie działania algorytmów wymagane są również traspozycje wybranych macierzy. Ponieważ ograniczenia pamięciowe uniemożliwiają obrócenie macierzy w pamięci, w plikach zapisane zostaną również traspozycje macierzy.



Rysunek 4.2: Struktura zapisana w pliku

Wynikiem działania tej części preprocesingu jest struktura będąca HashMapą. Zawierająca ona w komórce i listę wszystkich niezerowych elementów znajdujących się w i-tym wierszu macierzy wraz z ich wartościami. Figura 4.2 pokazuje fragment macierzy, który zostanie zapisany do pliku. Poniżej znajduje sie macierz $M_{5,5}$, która powstanie ze struktury przedstawionej na 4.2

$$M_{5,5} = \begin{pmatrix} 5 & 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 1 & 3 \\ 0 & 8 & 0 & 0 & 0 \end{pmatrix}$$

4.5.2 Inne metody przyśpieszenia wykonywanych preprocesingów

Jednym z głównym problemów jest to, że w czasie wykonywania wymienionych powyżej procedur, tabele USER, TAG i DOCUMENT są zablokowane na zmiany. W działającej aplikacji byłoby to poważnym problemem.

Można poradzić sobie z tym problemem poprzez posiadanie kopi bazy danych i uzupełnianie obydwu jednocześnie. Dzięki czemu operacje wymagające dużej ilości czasu, mogłyby być wykonane na innej bazie. Po dokonaniu wszystkich obliczeń wymagane jest zsynchronizowanie obydwu baz danych.

Inną możliwością jest zrównoleglenie wykonywanych obliczeń. Możliwe jest na przykład: podzielenie obliczenia tabel TAG_DOC, TAG_USER i USERTAGDOC na różne maszyny. Dodatkową możliwością jest podzielenie samych tabel na różne procesy. Na przykład wyliczanie tabeli TAG_DOC można podzielić na niezależne procesy ze względu na pole ID obiektów z tabeli TAG. Analogicznie można przeprowadzić taki podział dla innych tabel.

Kolejną możliwością są zmiany danych w tabelach TAG_DOC, TAG_USER

i USERTAGDOC w czasie kiedy dodawane są nowe rekordy do tabel USER, DOCUMENT i TAG. Wyeliminowało by to potrzebę wykonywania opisane powyżej preprocesingu, ale stanowiło by problem przy tworzeniu macierzy. Tworzenie macierzy (dokładnie: plików z których następnie tworzona jest macierz) jest czasochłonne i wymaga zatrzymania zmian dokonywanych na tabelach TAG_DOC, TAG_USER i USERTAGDOC. Problem ten dałoby się rozwiązać przez np: posiadanie kopi wymienionych wcześniej tabel. Synchronizacja kopi tabeli i głównej tabeli nie zajmowała by aż tak wiele czasu. Pomysł ten spowodował by jeszcze jeden potencjalnie groźny problem. Ilość operacji, które trzeba by wykonać w czasie gdy dodawane są nowe rekordy do tabel, znacznie by się zwiększyła. Mogłoby to spowodować większą awaryjność np: problemy z transakcjami.

4.6 Tworzenie macierzy na potrzeby algorytmów

W algorytmach Social PageRank i Adapted PageRank potrzebne są 3 rodzaje macierzy i ich transpozycje. Macierze te są tworzone z danych zebranych z serwisu delicous:

- M_{UT} macierz ta zawiera w komórce $m_{n,m}$ informacje o ilości dokumentów dodanych przez użytkownika u_n i opisanych tagiem t_m. Dane, z których zostaje utworzona ta macierz znajdują się w tabeli TAG_USR. Tabela ta została wyliczona w czasie preprocessingu.

- M_{TD} macierz ta zawiera w komórce $m_{n,m}$ informacje o ilości użytkowników którzy opisali tagiem t_n dokument d_m. Źródłem danych dla taj macierzy jest tabela TAG_DOC

- M_{DU} analogicznie, ta macierz zawiera dane na temat ilości tagów. Informacje pobierane są z tabeli USERTAGDOC

Wykorzystywane są one w kolejnych iteracjach algorytmu Social PageRank. Przy algorytmie Adapted PageRank są one używane pośrednio. Struktura na której operuje algorytm Adapted pagerank jest macierzą złozoną z macierzy M_{UD}, M_{TD}, M_{UT} i ich transpozycji. Macierzy używana w algorytmie wygląda następująco:

$$G_f = \begin{pmatrix} 0 & M_{DU} & M_{TD}^T \\ M_{DU}^T & 0 & M_{UT} \\ M_{TD} & M_{UT}^T & 0 \end{pmatrix}$$

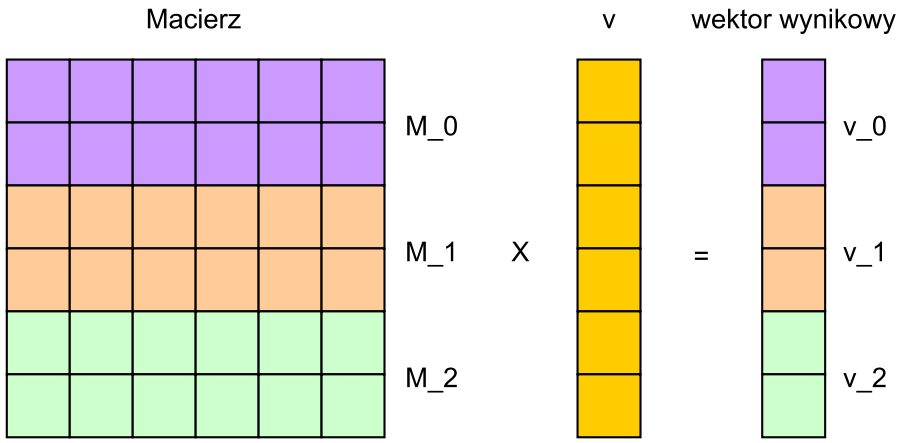
Ważną cechą macierzy na których przeprowadzane są operacje jest to, że są to macierze rzadkie. Liczba niezerowych komórek w macierzy wynosi: [TODO, dane się zmieniły, uzupełnić po testach]

4.7 Operacje przeprowadzane na macierzach

W każdej iteracji algorytmów, główna operacją przeprowadzaną jest mnożenie wymienionych wcześniej macierzy przez wektor. Operacja ta jest przeprowadzana do czasu uzyskania zbieżności wartości wektora wynikowego.

Z powodu wielkości macierzy w aplikacji nie możemy wczytać bezpośrednio całych macierzy do pamięci i na nich operować. Dodatkowo używana biblioteka stawia ograniczenie na iloczyn kolumn i wierszy takie że: $ilosc\_kolumn * ilosc\_wierszy <= 2^{31} - 1$. Gdzie wartość $2^{31} - 1$ jest to maksymalna liczba jaką można przypisać zmiennej typu integer w języku Java.

Implementacja



Rysunek 4.3: Mnożenie macierzy przez wektor v

Mnożenie macierzy odbywa się częściami. Bierzemy fragment macierzy: M_i i mnożymy go przez wektor v. Wynikiem jest wektor p_i, który stanowi i-ty fragment wynikowego wektora. Powstałe fragmenty wektora łączymy razem w odpowiedniej kolejności otrzymując w ten sposób w wektor wynikowy (4.3)

Listing 4.5: Mnożenie macierzy przez wektor v

```
def matrix_multiply(v, matrix_source):
        v_return = empty_vector()  #wektor zwracany
        for matrix_part in matrix_source.get_part_matrixes():
                #mnozenie macierzy i wektora
                v_part = multiply(matrix_part, v)
                # dokladamy wyliczony fragment wektora na koniec
```

```
                v_return.append(v_part)
        return v_return
```

Mnożenie odbywa się poprzez funkcje z biblioteki Colt. Biblioteka ta uwzględnia to, że macierz na której odbywają się operacje jest macierzą rzadką. Oszczędzana jest pamięć przez zapisywanie tylko niezerowych elementów. W czasie mnożenia przez wektor pomijane są wszystkie zerowe komórki, przez co sama operacja mnożenia jest krótka. Najwięcej czasu zajmuje samo tworzenie częściowych macierzy i ładowanie plików zawierających kolejne fragmenty macierzy.

4.7.1 Inne metody przyśpieszenia obliczeń na macierzach

Opisana powyżej metoda nie mogłaby zostać wykorzystana w działającej. Przy tylko 1 milionie dokumentów czas wykonania algorytmu Social PageRank wynosi około 36 godzin[FIXME, inne dane, inne czasy]. Prawdziwy system zawierałby zdecydowanie więcej danych. Poniżej opisanych jest kilka pomysłów na ulepszenie i przyśpieszenie działania aplikacji

Zmiana języka w którym została zaimplementowana aplikacja

Maszyna wirtualna Javy nie jest bardzo wydajna jeśli chodzi o szybkość obliczeń, dodatkowo dochodzą ograniczenia związane np: z pamięcią. Możliwe jest również przepisanie tylko kluczowych dla obliczeń fragmentów, a następnie ich uruchamianie z aplikacji.

Proste zrównoleglenie obliczeń

Wykorzystując opisaną wcześniej metodę dzielenia macierzy na kawałki, możliwe jest podzielenie macierzy na różne procesory/maszyny. Każdy proces po zakończeniu obliczania swojej części zapisałby wynikowy wektor do np: bazy danych, w której następowałaby synchronizacja procesów.

Obliczanie przy użyciu GPU/CUDA

Możliwe jest wykorzystanie GPU do wykonania obliczeń na macierzach. Obecne jednostki graficzne zawierają dodatkowe technologie wspomagające operacje na macierzach rzadkich.

Wykorzystanie innych aplikacji

Obliczanie kolejnych iteracji mogłoby również być przerzucone na zewnętrzną aplikacji np: MATLAB.

Rozdział 5

Dane [TODO]

[statystyki danych, czyszczenie danych, wyszukiwanie outlierów, ...]

Rozdział 6

Wyniki

6.1 SocialPageRank: wyniki

6.1.1 Wyniki algorytmu dla prostych danych

W tabelce 6.1 znajdują się dane, dla których zostało sprawdzone działanie algorytmu Social PageRank. Dane są nie duże i składają się z trzech różnych dokumentów, dwóch użytkowników i trzech tagów.

Dla takich danych macierz $M_{d,u}$ mówiąca o zależności dokumentów z użytkownikami ma postać:

$$M_{d,u} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 2 \end{pmatrix}$$

Macierz użytkowników i tagów, $M_{u,t}$:

$$M_{u,t} = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 0 & 1 \end{pmatrix}$$

Macierz tagów i dokumentów, $M_{t,d}$:

$$M_{t,d} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix}$$

	Użytkownicy	
	użytkownik 1	użytkownik 2
http://www.ted.com/	inspiration	
http://www.colourlovers.com/	design	inspiration
http://www.behance.net/	portfolio, design	portfolio, inspiration

Rysunek 6.1: Dane do prostych testów

wyniki:

Dla powyższych danych wyniki algorytmu zbiegają po czterech iteracjach z dokładnością $|P_3 - P_4| < 10^{-10}$. Wyniki zostały przedstawione w tabelce 6.1

	Social PageRank
www.ted.com	0.2381373691295440
www.colourlovers.com	0.4343479235414989
www.behance.net	0.8686958470829979

Tablica 6.1: Wyniki działania algorytmu Social PageRank

Można zauważyć, że największy ranking ma strona behance.net, która została dodana przez dwóch użytkowników i oznaczonych najpopularniejszymi tagami - 2 razy tagiem portfolio, użytym tylko dla tej strony, raz tagiem design, który użyty był 2 razy w powyższych danych i również raz tagiem inspiration, który jest najpopularniejszym tagiem, użytym w przykładzie aż 3 razy.

6.2 Adapted PageRank: wyniki

6.2.1 Wyniki algorytmu dla prostych danych

Algorytm Adapted PageRank przetestowano na tych samych danych co algorytm Social PageRank (tabelka 6.1). Macierz asocjacyjna powstała z tych danych ma wymiary 8×8 i wygląd:

$$G_f = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 2 & 2 & 1 & 1 & 2 \\ 1 & 1 & 2 & 0 & 0 & 1 & 2 & 1 \\ 0 & 1 & 2 & 0 & 0 & 2 & 0 & 1 \\ 1 & 1 & 1 & 1 & 2 & 0 & 0 & 0 \\ 0 & 1 & 1 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Zbierzność wektora została uzyskana po 22 iteracjach. Jest to zdecydowanie dłuższy czas w porównaniu z czterema wymaganymi iteracjami przy poprzenim algorytmie

Analizując wyniki z tabeli 6.1 można zauważyć, że najwyższy ranking wśród dokumentów ma strona begence.net: została ona dodana przez 2 użytkowników i przypisane jej zostały 4 tagi. Niewiele niższy ranking ma witryna colourlovers dodana przez 2 uzytkowników i opisana 2 różnymi annotacjami. Można po tym wywnioskować ze nadanie większej ilości tagów nie ma dużego wpływu na rank strony. Za to zmniejszenie liczby użytkowników którzy

	Adapted PageRank
doc: http://www.ted.com/	0.280676409730572
doc: http://www.colourlovers.com/	0.369220441236174
doc: http://www.behance.net/	0.384551423972747
usr: użytkownik A	0.402473669662513
usr: użytkownik B	0.354578057974890
tag: inspiration	0.383291185401107
tag: design	0.321455285546253
tag: portfolio	0.314739469615726

Tablica 6.2: Wyniki działania algorytmu Adapted PageRank

tą strone dodali, ma duze: przykład strona ted.com i colourlovers.com gdzie widoczny jest dość duzy skok wartości wyniku.

6.3 Wyniki testów [TODO!!]

Bibliografia

[1] Shenghua Bao, Guirong Xue, Xiaoyuan Wu, Yong Yu, Ben Fei, and Zhong Su. Optimizing web search using social annotations. In *Proc. WWW '07*, pages 501–510, Banff, Canada, 2007.

[2] Anlei Dong, Ruiqiang Zhang, Pranam Kolari, Jing Bai, Fernando Diaz, Yi Chang, Zhaohui Zheng, and Hongyuan Zha. Time is of the essence: improving recency ranking using twitter data. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 331–340, New York, NY, USA, 2010. ACM.

[3] Andreas Hotho, Robert Jäschke, Christoph Schmitz, and Gerd Stumme. Information retrieval in folksonomies: Search and ranking. In York Sure and John Domingue, editors, *The Semantic Web: Research and Applications*, volume 4011 of *Lecture Notes in Computer Science*, pages 411–426, Berlin/Heidelberg, June 2006. Springer.

[4] Yan'an Jin, Ruixuan Li, Kunmei Wen, Xiwu Gu, and Fei Xiao. Topic-based ranking in folksonomy via probabilistic model. *Artificial Intelligence Review*, pages 1–13, February 2011.

[5] Makoto Kato, Hiroaki Ohshima, Satoshi Oyama, and Katsumi Tanaka. Can social tagging improve web image search? In James Bailey, David Maier, Klaus-Dieter Schewe, Bernhard Thalheim, and Xiaoyang Wang, editors, *Web Information Systems Engineering - WISE 2008*, volume 5175 of *Lecture Notes in Computer Science*, chapter 19, pages 235–249. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2008.

[6] Michael Noll and Christoph Meinel. Web search personalization via social bookmarking and tagging. pages 367–380. 2008.

[7] Majdi Rawashdeh, Heung-Nam Kim, and Abdulmotaleb El-Saddik. Folksonomy-boosted social media search and ranking. In Francesco G. B. De Natale, Alberto Del Bimbo, Alan Hanjalic, B. S. Manjunath, and Shin'ichi Satoh, editors, *ICMR*, page 27. ACM, 2011.

[8] Ralf Schenkel, Tom Crecelius, Mouna Kacimi, Sebastian Michel, Thomas Neumann, Josiane X. Parreira, and Gerhard Weikum. Efficient

top-k querying over social-tagging networks. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '08, pages 523–530, New York, NY, USA, 2008. ACM.

[9] Y. Yanbe, A. Jatowt, S. Nakamura, and K. Tanaka. Can social bookmarking enhance search in the web? In *JCDL '07: Proceedings of the 7th ACM/IEEE-CS joint conference on Digital libraries*, pages 107–116, New York, NY, USA, 2007. ACM.

[10] Ding Zhou, Jiang Bian, Shuyi Zheng, Hongyuan Zha, and C. Lee Giles. Exploring social annotations for information retrieval. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 715–724, New York, NY, USA, 2008. ACM.