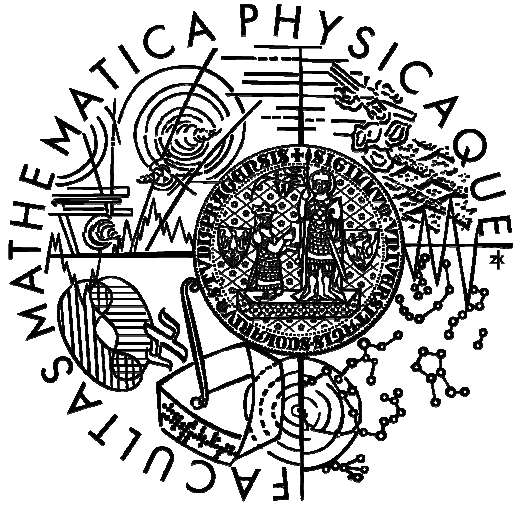
Charles University in Prague
Faculty of mathematics and physics

MASTER THESIS



Peter Irikovský

Graph algorithms in text retrieval

Department of Software Engineering

Supervisor: RNDr. Michal Kopecký, Ph.D.

Field of study: Computer Science

Hereby, I would like to thank my supervisor RNDr. Michal Kopecký, Ph.D. for his willingness, professional advises and comments and for prompt answers to all my questions. For me, it was a pleasant cooperation. I would also like to thank RNDr. Leo Galamboš, PhD. for supplying the test collection and for valuable comments on the implementation.

I declare that I wrote this thesis myself and with use of only the explicitly cited sources. I agree with lending the thesis.

20th April 2007, Prague Peter Irikovský

Název práce: Grafové algoritmy ve vyhledávání textových dokumentů

Autor: Peter Irikovský

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Michal Kopecký, Ph.D.

e-mail vedoucího: michal.kopecky@mff.cuni.cz

Abstrakt: Tato diplomová práce zkoumá možnosti využití grafových algoritmů v oblasti information retrieval (vyhledávání informací). Na začátku je poskytnut přehled základních pojmů z oblasti dokumentografických informačních systémů a základů teorie grafů. Zbytek práce se pak zabývá průnikem těchto dvou oblastí. Mezi příklady z tohoto průniku patří například klastrování a kategorizace dokumentů, či hledání komunit. Nejvíc pozornosti je však soustředěno na algoritmy hodnotící důležitost dokumentů s pomocí využití grafů. Tyto algoritmy vylepšují nejdůležitější vlastnost informačních systémů, jejich přesnost. Práce poskytuje přehled různých hodnotících algoritmů založených na grafech a uvádí komentáře k jejich praktičnosti, časovým a paměťovým nárokům. V práci je taky detailně popsaná implementace algoritmů na počítaní PageRanku stránek navržená pro využití ve vyhledávači Egothor. Popis také obsahuje výsledky měření časové a paměťové náročnosti a uvádí návrhy na další zlepšení.

Klíčová slova: grafové algoritmy, vyhledávání dokumentů, PageRank

Title: Graph algorithms in text retrieval

Author: Peter Irikovský

Department: Department of software engineering

Supervisor: RNDr. Michal Kopecký, Ph.D.

Supervisor's e-mail address: michal.kopecky@mff.cuni.cz

Abstract: This thesis surveys use of graph theory and algorithms in information retrieval. It provides an introduction to graph and information retrieval theories and an overview of the overlap between these disciplines. We show application of the graph theory in clustering, document classification, finding communities etc. The most stress is, however, put on ranking algorithms as they aim to improve the most critical property of the information retrieval systems, their precision. The paper presents different graph-based ranking algorithms, provides comments to their time and memory requirements and to realistic usage of these rankings. It also contains a description and test results of our implementation of algorithms for computing the PageRank distribution designed for the Egothor search engine.

Keywords: graph-based retrieval, information retrieval, graph algorithms, PageRank

Content

List of pictures

List of tables

1. Introduction

Graph theory is a very well studied science discipline. So is the field of information retrieval (IR). These two disciplines are often perceived as being totally different, but as we show in this paper, there is a large overlap between them.

In the last century, graph theory and graph algorithms have enjoyed a great deal of attention. Many new graph-based techniques have been proposed and utilized in various sectors. One of the best known theoretical results from the mid 20th century is the max-flow min-cut theorem which was proved by Ford and Fulkerson in 1956 [1]. The consequences of this theorem have helped researchers in multiple fields. Even now, more than 50 years after proving the theorem, new applications for it are being discovered. As we show in this paper, algorithms for computing maximum flow and their modifications are now starting to be used also in the field of information retrieval. We show two interesting applications of this group of algorithms. In one case it helps us to discover communities (web pages relevant to a given topic). In the other case a modification of maximum flow algorithm is utilized to obtain a relevance ranking of web pages to a given query.

In this paper we show that network flow algorithms are not the only part of graph theory that is or can be used in the process of information retrieval. Graphs are now base elements of many information retrieval applications.

In contrast to the graph theory, the history of information retrieval is much shorter. It relates to the amount of information that needed to be stored. As the amount of information that was being stored boosted, new techniques for storing information were needed. People first started to use paper for storing information. Soon they had many papers and books, but when they needed to find some information quickly, they ran into trouble. Hence, they started to develop structures for organizing documents that would help them to find the right information faster. This time point is considered as the birth time of information (text) retrieval systems. As the evolution has continued, data storage techniques have been improved. Data are now mostly stored on hard drives, CD and DVD discs, and also on magnetic tapes etc. However, the amount of data that we keep at disposition is enormous and surpasses one's imagination. This presents the same problem people had many years ago and that is: *How can we find what we want quickly*?

Currently, the most popular source of information is the World Wide Web (WWW or the Internet). It comprises several billion web pages containing information about almost everything we can imagine. To find information without a search engine might be an extremely difficult quest. That is why people started to develop search engines for the web. It probably started out as research aimed at helping people to find what they want, but it turned out to be a great business. Today companies like Google [2] and Yahoo! [3] are known around the world not only as very useful tools for searching the web, but also as companies traded on the stock exchange markets. The outstanding, and for many people unbelievable performance of these systems in the process of information retrieval got people thinking these companies would achieve similar performance in business, so we now see people paying enormous sums of money for their shares. To better explain this we can use the following example. If you bought the entire Google Company at the current share price at the beginning of the year 2006 times the number of shares issued (i.e. the market capitalization) and the company performed the same way it did then for the next 100 years, the profit for the 100 years would roughly equal the price you paid for the company. That is, if Google was not a publicly traded company and you invested money into it, you would get it back after 100 years and only then you would start to make money. Therefore, the return of such an investment would be about 1% per annual while the average is more than 5 times higher.

The success of these companies on the stock exchange market has brought these companies a considerable amount of money that is invested in research. This is probably one of the main reasons why so many top-class scientists are concentrating their research on IR. Thanks to this, there are numerous papers about the latest research in IR. The amount of available papers on IR would be even greater if most of the research done by these commercial companies was not confidential. There are two main reasons why companies such as Google or Microsoft do not publicize the results of their latest research nor the exact principles of their search engines. The obvious one is the competitors' battle between these companies. The second more important one is that many people are commercially interested in increasing the ranking of their web pages for certain queries. Uncovering the exact manner in which these engines work would make it easier to find out new tricks how to manipulate the ranking.

In this thesis we first provide an introduction to the fields of information retrieval and graph theory and then provide an overview of approaches that use the

knowledge from the graph theory to improve some properties of the IR systems. We will also present our implementation of a program for computing the PageRank distribution, by our thoughts the most important utilization of the graph theory in information retrieval.

The rest of this paper is organized as follows. Chapter 2 provides an introduction to information retrieval, presents the common structure of the IR systems and points out the most critical issues of the current IR systems. Chapter 3 contains insights to graph and complexity theories and describes algorithms for solving graph problems that are applicable in the field of information retrieval. A more detailed introduction to the use of graphs and graph theory in information retrieval is provided in chapter 4 which also premises the content of the following chapters. Chapter 5 then provides a detailed description of graph-based ranking algorithms provides different perspectives to the ranking process and reviews the results and applicability of different approaches. This chapter also contains a description of our implementation of the PageRank algorithm and provides comments, to the run-time and memory efficiency of different versions of this algorithm. An overview of approaches to finding communities – sets of web pages characterized by page content similarity and cohesive link structure is present in chapter 6. In chapter 7 we briefly describe several other applications of the graph theory in information retrieval. We conclude with chapter 8,

2. Information retrieval

With the increasing amount of documents available in electronic form the need for efficient storing and searching these documents arose. For this purpose information retrieval systems were developed.

Text documents are generally stored in text databases and the IR systems provide a framework that enables searching for documents. A problem is that users mostly do not have extensive knowledge of these systems nor about the documents stored. Consequently most of user queries are vague and users often get either a lot of irrelevant documents or no documents as a result of their queries. However, the relevancy of a document to a query is very difficult to generalize. Different people expect different documents containing different information as a result of a single query ([4]). This is the main difference between information retrieval and database retrieval, where the relevance of a record in the database to a query is explicit.

In this chapter we provide an introduction to information retrieval and to the most common structure of the IR systems. We also point out most common issues of the current systems. Chapters 4 and 7 then present graph-based approaches to resolving some of these issues.

2.1 Information retrieval systems

The IR systems store some information about a set of documents and compare this information with user queries. Choosing the right information about the documents to store is a common issue in IR and is handled in a process called *indexation*.

However, the main goal of IR systems is to find relevant documents to a given user request (query). There are two main forms of search – listing a directory and formulating a query. In this thesis we concentrate on the latter.

In response to a given query, IR system outputs a list of documents. Generally, these are sorted by estimated relevancy in descending order. Since users often only take a look at the first 10-50 documents (the *maximum criteria*), the systems aim to place the most relevant documents at the top of the list. Documents actually relevant for the user are called *hits*.

In order to compare the quality of the information retrieval systems, we define two basic measures - the *precision* and the *recall* of a system. Both of these measures assume binary relevancy. A document is either relevant, or is completely irrelevant.

Precision (*P*) states the proportion of retrieved and relevant documents to all documents retrieved.

$$P = \frac{N_{rr}}{N_r} = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|\{retrieved\ documents\}|}$$

Recall (*R*) indicates the proportion of retrieved and relevant documents to all the relevant documents available.

$$R = \frac{N_{rr}}{N_{re}} = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|\{relevant\ documents\}|}$$

Where N_{rr} = number of retrieved relevant documents

N_r = number of retrieved documents

N_{re} = number of relevant documents

In an ideal IR system both of these metrics are equal to one. However in the real world, we can observe a reciprocal proportion between these two metrics. It is also important to note that the relevancy of a document depends on an individual opinion. The retrieved documents depend on the quality of the IR system, but also on the user-formulated query.

Many systems also provide a *query tuning* functionality. This can be either handled by simply allowing users to reformulate their queries or by allowing users to input some kind of feedback and using this information to re-rank the results.

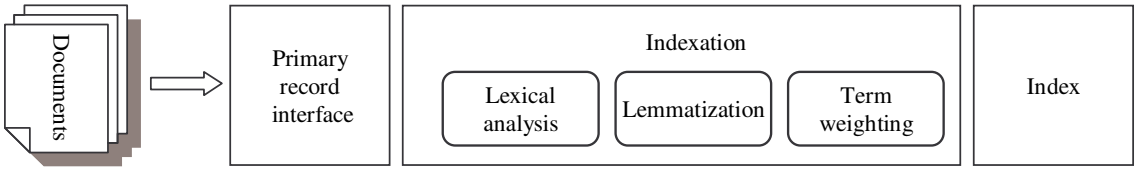
2.2 Architecture of IR systems

Generally, the IR systems have two basic functions – *inserting* (*indexing*) documents and the *searching* function. Each of these functions can be broken down into modules. The *indexation* units usually have modules for *lexical analysis*, *lemmatization*, *thesauri usage* and *term weighting*. The *search* unit usually has *user interface*, *lexical* and *syntactic analyzer* and *search engine* modules.

2.2.1 Indexation

The main goal of the *indexation unit* is to create a structure that properly characterizes documents so that they can be efficiently retrieved when they match a query. This structure can contain information about billions of documents (e.g. web), so it has to be space (memory) efficient, but also quick to work with. In this structure each document is represented by a *record* that contains a formal description of the document. The record should be composed of properly specified attributes and well chosen (key)

terms. The procedure of choosing terms that characterize the document is called *document indexation*. Documents can be indexed automatically, semi-automatically or manually.



Picture 1 Indexation

The first phase of indexation in general IR systems is the *lexical analysis*. In this phase the elimination of improper (not influential) terms takes place. The list of such terms is commonly called *stop-list*.

In the *lemmatization* phase, the morphological forms of words are eliminated to avoid excessive size increase of the index.

The *term weighting* phase depends on the model of IR system. It the basic – *Boolean* model no term weighting is done and the index only contains the information about terms present in a document.

2.2.2 Vector Model

In the vector model ([5]), we assume using n various terms $t_1,..,t_n$ for the indexation of all documents in the set. Then each document from the collection is represented by a vector

$$d_i = (w_{i1}, w_{i2}, ..., w_{in})$$

where $w_{ij} \in <0,1>$ expresses the importance of term t_j for the identification of document d_i, where higher values of w_{ij} represent higher importance. In the *vector* model the document collection is represented by a matrix $M=(w_{i,j})$ with size m x n where the i-th row represents the i-th document and the j-th column represents the j-th term.

The query is in the vector model represented by an n dimensional vector:

$$q = (q_1, q_2, ..., q_n)$$

where $q_j \in <0,1>$ or optionally $q_j \in <-1,1>$. The *similarity coefficient* between the query and each document can be thought of as the *distance* between the vector of the document and the query vector and can be calculated as:

$$Sim(q, d_i) = \sum_{k=1}^{n} q_k . w_{ik}$$

To ensure good precision of the system, the weights of terms in matrix M have to be set properly. Most of the automated indexation methods are based on the

observation that the importance of terms for the indexation is in direct proportion with the frequency of term occurrence in the document. The basic weight setting is the *term frequency* that is defined as:

$$TF_{ij} = \frac{m_{i,j}}{m_i}$$

As the *TF* values are usually very low, the normalized term frequency *NTF* measure that ignores very low values is used more often.

$$NTF_{i,j} = \begin{cases} 0 & \text{if } TFi,j \leq \varepsilon \\ \frac{1}{2} + \frac{1}{2}\frac{TF_{i,j}}{\max(TF_{i,k})} & \text{otherwise} \end{cases}$$

The terms that occur in most of the documents are not very good key terms for the indexation process. Because of this we also incorporate a measure that characterizes the terms depending on the document collection. One such measure is the *inverse term frequency* that can be calculated as:

$$ITF_j = \log(m / O_k)$$

where O_k is the number of document where term k occurs. The elements w_{ij} of matrix M can be calculated as:

$$w_{ij} = NTF_{ij}.ITF_j$$

To achieve uniform size of the document vectors, the term weights of a document are usually normalized to one.

2.2.2.1 Vector model with the term similarity matrix

As the basic vector model does not consider the substitutability of key terms, the results of the query significantly depend on choosing the right terms in both index and the query. This problem can be partly solved by using the *thesauri*, by *lemmatization* or by using a *term similarity matrix*.

The *term similarity matrix* is a square matrix S, where $S_{ij} \in \langle 0,1 \rangle$ represents the substitutability of term t_i by term t_j. This matrix can then be used to process the user query by incorporating the term similarity as:

$$q' = q.S$$

This transformation ensures that terms that are similar to the query terms but are not part of the query are also considered in the result. The matrix S is usually symmetric and is computed by statistical methods.

2.3 Web IR systems

Before the expansion of web, IR systems were typically installed in libraries and were mostly used by the skilled librarians. The retrieval algorithms of these systems were usually based exclusively on word analysis.

The web changed all this. Users have now access to various search engines whose algorithms consider many more factors when computing the results of the query. The major difference between the traditional and the web IR systems is that the web IR systems have to solve one additional issue – obtaining the document collection. The process of collecting web pages is called *crawling* and is described in chapter 7.1.1.

In addition the web IR systems also have to face the issue ([6]) that the web lacks any imperative structure as e.g. centralized obligatory structure of the documents; control of reliability of information; information categorization; standard terminology; or separation of advertisement from other documents, etc.

As we show in chapters 5 to 7 a lot of these issues can be partly resolved using the hyper-link analysis. However, all of these issues still present open problems.

2.4 Main drawbacks and issues of the current IR systems

Although current web IR systems use more and more sophisticated and complex methods, users of these systems still face several issues. The results provided by these systems often contain a lot of irrelevant documents and the documents actually matching user's needs are stashed in the rest of the results.

This is mainly caused by the subjectivity of human understanding. The user queries are often vague and the notion of what should actually be returned as relevant results is very indefinite.

To improve this condition, different approaches to ranking the results were proposed. In this paper we describe different graph-based rankings that try to incorporate available information to provide the current users with better results.

3. Graphs theory and algorithms

In this chapter we would like to introduce you to graph theory and graph algorithms and briefly describe algorithms that we use later on in the process of information retrieval. We will describe general terms and explain principles of algorithms that we show have application in IR systems. As graph theory is very broad, we only define terms that we use further on. In this thesis we would use the terminology used in the largest free web encyclopedia – Wikipedia [7].

3.1 Graphs – general terms

In mathematics and computer science graph theory is the study of mathematical structures called *graphs*. *Graphs* are used to model pairwise relations between objects from a collection. In this context a graph $G = (V, E)$ usually means a set V of *vertices* and a set E of pairs of vertices called *edges*. *Edges* connect two *vertices*. We say a graph is *undirected* if there is no distinction between the two vertices associated with each edge. If all edges of a graph are directed from one vertex to another, the graph is *directed*. Each edge can also have parameter called weight (w: $E \rightarrow R$) associated with it. In that case we say the graph is weighted, otherwise it is unweighted. (Pictures 2 and 3 present examples of an unweighted and undirected graph and a weighted and directed graph respectively).

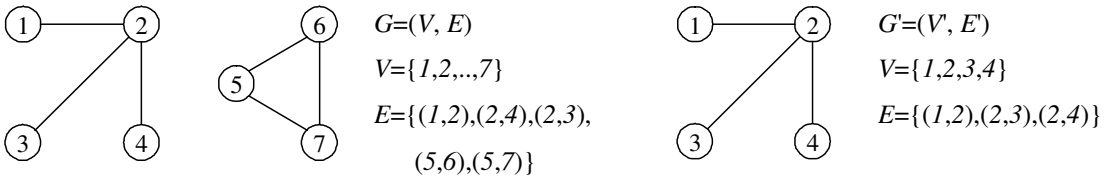
In an undirected graph, we say that v_0 and v_1 are *adjacent* if and only if there is an edge $(v_0, v_1) \in E$. The edge is then *incident* on vertices v_0 and v_1. In a directed graph, v_0 is *adjacent to* v_1 and v_1 is *adjacent from* v_0 if there is an edge $<v_0, v_1> \in E$ and the edge is then incident on vertices v_0 and v_1.

Each vertex in a graph has its *degree*. The *degree* of a vertex is the number of edges incident on that vertex. In directed graphs, we consider both the *in-degree* and the *out-degree* of a vertex v meaning the number of edges adjacent from v and adjacent to v respectively and denote them by $ind(v)$ and $outd(v)$ respectively.

Another commonly used term is a *path*. Existence of a *path* from v_0 to v_k is equal to the existence of a sequence $v_0, v_1, \ldots, v_k$ such that for every $i \in \{1, \ldots, k\}$, vertices v_{i-1} and v_i are adjacent. A path is called *simple* if no vertex is repeated in the sequence and is called a *cycle* if it is a *simple* path except that $v_0 = v_k$. The shortest *distance* between two vertices v_0 and v_1 in an unweighted graph is defined as the number of edges of the shortest path between v_0 and v_1 in the graph and is denoted by $dist(v_0, v_1)$. In a weighted

graph the $dist(v_0, v_1)$ is defined as the sum of edge weights of a path between v_0 and v_1 such that sum is minimal among all paths connecting v_0 and v_1.

A graph is said to be *connected* if for any two vertices there exists a path that connects them. A *subgraph* $G' = (V', E')$ of graph $G = (V, E)$ consists of a subset of vertices $V' \subseteq V$ and a subset of edges $E' \subseteq E$ such that they form a graph. A *connected component* of a graph is a maximal connected subgraph. I.e. a *connected component* is a subgraph of a graph that can not be enlarged by any vertex or edge from the original graph without losing the property of being connected. A directed graph is *strongly connected* if for any two vertices there exist a directed path between them in both directions. A graph is a called a *tree* when it is a connected graph without cycles and is called a *forest* when it is a collection of trees. A graph is *complete* if all pairs of vertices are connected by an edge.



Picture 2 Graph examples

Notes: G is an undirected unweighted graph. G' is a subgraph of G. G' is also a connected component of G and also a tree.

3.2 Graph representation

Graphs can be represented by geometrical means as shown above, but for mathematical and computer science purposes we generally use other representations of which most commonly used is the *adjacency matrix*. For a graph with n vertices the *adjacency matrix A* has size n x n. The element $a_{i,j}$ in the i-th row and j-th column is equal to *1* if there is an edge between v_i and v_j. In the case of an undirected graph, the matrix is *symmetric*, i.e. $a_{i,j} = a_{j,i}$ for each i and j. If the graph is weighted, then $a_{i,j}$ is equal to the weight of edge (v_i, v_j). The degree of any vertex of an undirected and unweighted graph represented by an adjacency matrix equals to the sum of the i-th line (or row). In a *directed* graph the sum of the i-th line and the sum of the i-th column are equal to the *out-degree* and the *in-degree* of v_i respectively.

Another possible, commonly used representation is an *adjacency list*, where each row of the *adjacency matrix* is represented as a list. This representation is more

suitable for sparse graphs because of its memory efficiency, but is more difficult to work with and much slower when changes in graph are made.

There are lots of other representations that can be used for various purposes, but those are not of much importance for this paper.

3.3 Graph algorithms

Graph algorithms are used to compute some properties of a given graph. If we can imagine a map represented by a graph where towns are represented by vertices and highways correspond to edges. Then we might want to know the distance between certain cities. We might want to find out names of all the towns where we can get from a given town just by using highways. We might be interested in which town is the most *central* one in our map or in many others aspects. To answer these questions, one might utilize an algorithm for solving the shortest-path problem and a graph-crawling algorithm.

A different realistic scenario where graph algorithms may be used to solve a real world problem is if we imagine a water pipe network. In such a network each pipe (modeled by an edge) has a certain width and so can transport only a limited amount of water in a given time unit. The junctions of these pipes (modeled by vertices) can be considered not being able to hold any water; that is the amount of water that flows into the junction equals the amount of water flowing out. Furthermore we can imagine we have an inlet to this network called *source* and an outlet called *sink*. Intuitively the total flow of this network is the rate at which the water comes out of the outlet.

If we model this network by a weighted directed graph, we can use a maximum flow problem solving algorithm to find out the total flow of the network, or we use a minimum cut algorithm to find out the least number of pipes that need to be blocked to prevent the possibility of transporting any flow from the inlet to the outlet.

These examples illustrate some applications of graph algorithms. Further on in this paper we show how these algorithms are used in IR systems.

3.4 Complexity

Before going any further in explaining various algorithms, we need to define some measures to be able to compare them.

To describe the *asymptotic* behavior of a function, we use the big *O* notation. The *time complexity* of graph-based algorithm can be asymptotically represented by a

function of two variables m, n meaning the number of edges and vertices respectively. Thus, if we say an algorithm has asymptotic time complexity $O(m + n)$ we are saying that the time complexity can be bounded from above by $a(m + n)$ for some constant a. A similar definition holds for the space complexity. For a more detailed explanation of asymptotical complexities see [8]. The asymptotic complexity is mostly used for worst case complexity analysis. This means that if $f(x) = O(g(x))$, then for "big" x, f performs better than or equal to $a.g$. However, this does not say anything about neither the constant a nor the minimum size of x. They both can be small, but they also can be enormous numbers.

As a result we also use other measures of algorithms. One of these measures is the *average complexity* which attempts to measure the average performance of an algorithm. Again it is a theoretical approach, but for practical use, it is much more relevant than the worst-case asymptotical complexity. Probably the best example where the worst-case asymptotic complexity is not a reasonable ranking is the case of sorting algorithms. These algorithms get a list of numbers on input and their task is to sort these numbers. The best achieved and proved to be best achievable worst-case asymptotic result is $O(n.log(n))$ which is for example achieved by *merge-sort*. In practice, however, the most used algorithm is quick-sort because it is simple to implement, but primarily because of its performance. Empirical tests showed that the algorithm is faster than other sorting algorithms and that the average asymptotic complexity is $O(n.log(n))$ with a small constant. The main drawback of these theoretical approaches is that they treat all basic operations (multiplying, square root etc.) as being equally time demanding, which obviously is not true. However, they can provide a good heuristic for the real time complexity.

Another important measure is counting the *I/O* (input / output) operations of a given algorithm. This measure is generally important in algorithms on huge graphs that can-not be loaded into the main memory (random access memory - RAM). This measure only counts the number of I/O operations because they are much more expensive (in terms of time) than other operations. Current disks are several orders of magnitude slower (disk seek time) than the random access memories. Therefore if an algorithm works with a graph that is not stored in the main memory, it is reasonable not to choose an algorithm with the lowest asymptotical nor average complexity but rather the one using the least I/O operations.

All of these heuristics help us to choose the right algorithm for given problem, but the real time complexity may sometimes surprise us at the end. For example different graph representations used by one algorithm can drastically change the performance.

3.5 Graph traversal

The goal of graph traversal algorithms can either be to find out if a certain vertex is reachable from a given start vertex, or to provide a list of all vertices connected (by a path) to a given start vertex.

There are two commonly used approaches – the *BFS* and *DFS*. The basic idea of *breadth first search* (BFS) is to start several paths at a time and advance in each one step at a time. I.e. it starts from a given start vertex and in the *i-th* iteration it finds vertices that are reachable from the start vertex by a path of length *i*. It terminates when no previously undiscovered vertices are found in the iteration. On the other hand the *depth first search* (*DFS*) propagates the idea of continuing the search until the end of the path once a possible path is found. Both of these algorithms have their pros and cons and both are widely used. Good examples of uses of these algorithms are *web crawlers* – programs that browse the World Wide Web (WWW) in an automated predefined manner.

3.6 Shortest path finding

The objective of the shortest path algorithms is either to find the shortest path between two given vertices, or to find the minimum distance between a given vertex and all other vertices.

3.6.1 BFS based approach

On unweighted graphs (we consider all edges having length 1) the problem can be solved by a slight modification of the above-mentioned BFS algorithm. In the *i-th* iteration the distance $d[v]$ of the newly discovered vertices is set to *i*. After running the algorithm all reachable vertices have been assigned a distance that is also the shortest distance. The average and worst case time complexity of this algorithm is $O(n+m)$ because every vertex and every edge is only used once and then it is marked as discovered and never used again.

3.6.2 Dijkstra

Dijkstra's algorithm is a *greedy* algorithm (more about *greedy* algorithm technique can be found in [9]) that solves the shortest path problem for weighted directed graphs with nonnegative edge weights. The algorithm works in iterations on two sets *S* and *Q*. Initially S is empty and Q contains all vertices. For all vertices except the start vertex *s*, distance $d[v]$ of vertex *v* from vertex *s* is set to infinity. In each iteration the algorithm extracts a vertex *u* with minimum $d[u]$ from *Q* and *relaxes* all edges (u,v). By relaxation we mean that we compare if $d[v] > d[u] + w(u,v)$ (weight of the edge (u,v)) and if the condition is fulfilled we set $d[v] = d[u] + w(u,v)$. The vertex u is then deleted from *Q* and added to *S*. The intuition behind this algorithm is that in each iteration the distance of the vertex from Q with the minimum distance cannot be improved any further and so we say it is the definite distance. After *n* iterations, all vertices have the definite distance set.

The time complexity of this algorithm depends primarily on the data-structure we use. The complexity can be described as *n* times *Extract Min from Q* + *m* times *Relax edge*. For the simplest case – Q represented as an unsorted list the time complexity is $O(n^2)$. If Q is represented as a binary heap (BH), the *Extract Min* and *Relax edge* operations require $O(1)$ and $O(log(n))$ respectively which implies time complexity $O((m+n).log(n))$ which in the worst case (for dense graphs) means $O(n^2 log(n))$. To lower the worst-case time complexity we can use the Fibonacci heap (FH) (for description see [10]) where *Extract Min* and *Relax edge* operations require $O(log(n))$ and $O(1)$ respectively. So the time complexity is $O(m + n.log(n))$ implying worst-case time complexity $O(n^2)$. For dense graphs the complexity of the simple and FH implementations ($O(n^2)$) outperform the binary-heap implementation ($O(n^2 log(n))$) and for sparse graphs the heap implementations ($O(n.log(n))$) perform better than the simple implementation ($O(n^2)$).

From what is stated above, one might very probably think, the Fibonacci heap implementation of the Dijkstra algorithm would be generally the most efficient and most commonly used. But as noticed from the tests done in [11] the Fibonacci heap only rarely outperforms both other implementations. For dense and sparse graphs the simple and the binary heap implementations performed the best respectively. The binary heap implementation was generally the best performing, although it was seldom slightly outperformed by the FH implementation on the dense graphs. The price this

implementation pays for achieving the best worst-case asymptotical complexity is too high and does not pay off in the real world.

By this example we illustrated that a better worst-case or average complexity does not guarantee, that the algorithms would also perform better in empirical tests.

3.6.3 Other shortest path algorithms

There are several other approaches to the shortest path problem. Another that is worth mentioning is the Bellman-Ford algorithm that is very simple to implement and works also on graphs with negative edge weights. Initially $d[v]$ is set to infinity for all vertices except the start one, which is set to zero. The algorithm then works in iterations. In each iteration it relaxes all edges. By relaxation we mean that it checks if $d[v] > d[u] + w(u,v)$ and if the condition is fulfilled, then it sets $d[v] = d[u] + w(u,v)$. It repeats the iterations until no $d[v]$ is changed in the last iteration. For graphs with positive edge weights the time complexity is $O(m.n)$ because in each iteration we process all edges and after each iteration at least one vertex has its final distance value set.

As already stated, the most commonly used algorithms for solving the shortest path problem are the Dijkstra algorithm and the BFS-based algorithm. In this paper we show an interesting utilization of the shortest path algorithm that helps us to re-rank web pages to a given query using the user-supplied relevance feedback.

3.7 Network flow

Maximum *s-t flow* problem is defined on a weighted oriented graph G with two special vertices: source (s) and sink (t). For illustration we may suppose that edges are (oriented) pipes with a given diameter and vertices are the only places where edges can cross. We can then ask how many liters of water can flow from the *source* to the *sink* in a given time unit.

As we will demonstrate the network flow algorithms have a wide variety of applications. In [12] the following examples are mentioned:

Examples of network flow utilization in real life			
Network	**Vertices**	**Edges**	**Flow**
hydraulic	reservoirs, pumping stations, lakes	pipelines	fluid, oil
circuits	gates, registers, processors	wires	byte flow
mechanical	joints	rods, beams, springs	heat, energy
communication	telephone exchanges, computers, satellites	cables, fiber optics, microwave relays	voice, video, packets
financial	stocks, currency	transactions	money
transportation	airports, rail yards, street intersections	highways, rail beds, airway routes	freight vehicles, passengers
chemical	sites	bonds	energy

Table 1 Examples of network flow utilizations

In this paper we show two IR applications of the network flow computations. As already mentioned it helps us to discover communities and it provides us with a relevancy ranking to a given query.

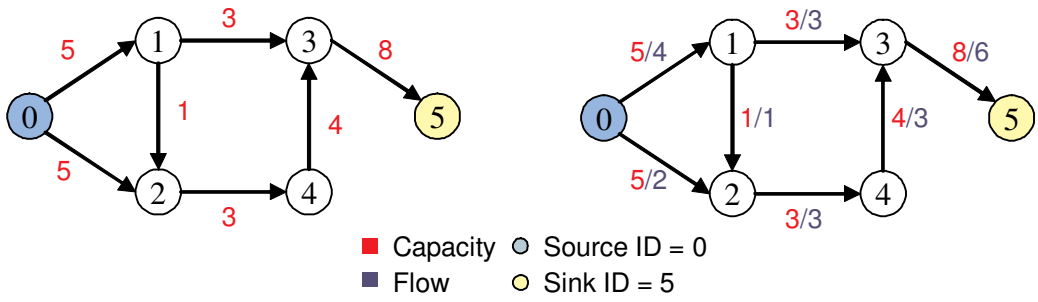
3.7.1 Network flow problem definition

The input of the problem is of the following form $G = (V, E, s, t, c)$ where (V, E) is a directed graph, s (*source*) and t (*sink*) are two distinguished vertices, $c(e)$ stands for the capacity function c: $E \rightarrow R^+$.

In graph theory a network flow is an assignment of *flow* f to the edges of a digraph that satisfies the following constraints:

- *flow on each edge is less than or equal to the capacity of that edge* ($f(e) \leq c(e)$)

- for every vertex (except source and sink) the amount of flow into that vertex equals to the amount of flow out of it ($f^+(v) = f^-(v)$)



Picture 3 Example of a network and a flow in this network

3.7.2 Ford – Fulkerson

To explain this algorithm, we need a few more definitions. Suppose that we have a correct flow *f*. Then the *residual graph* $G[f]$ for graph G and flow f can be defined as: $G[f] = (V, E')$ where edge $\langle x,y \rangle \in E'$ if $f(u,v) < c(u,v)$ or $f(v,u) > 0$, i.e. the residual graph contains unsaturated edges and backwards oriented edges that contain a flow in the original graph. An *augmenting path* is a path from source to sink in the residual graph. The above definitions imply that f is a maximum flow if there is no augmenting path between source and sink in the residual graph.

The basic idea of the algorithm is very straightforward. Iteration: find an augmenting path and augment the flow along this path. Repeat iteration until there is no augmenting path. It has been proved, that if we use the shortest (in terms of number of edges used) augmenting path (Edmonds-Karp enhancement) in each iteration, the algorithm converges to the maximum flow. As *attachment 1* you can find illustration to the steps of Ford-Fulkerson algorithm on network flow from Picture 3.

The worst-case asymptotic time complexity of the Ford-Fulkerson algorithm is not bounded. When using irrational edge capacities, the algorithm might never achieve the exact solution. When using integer capacities, the run-time is bounded by $O(m.f)$ where m means the number of edges and f is the value of the maximum flow. The modified version (Edmonds-Karp) of Ford-Fulkerson algorithm has asymptotic worst-case time complexity $O(m^2.n)$.

3.7.3 Pre-flow push (Push re-label) algorithm

The pre-flow push algorithm uses a different approach than augmenting algorithms as for example Ford-Fulkerson. *Augmentation paradigm* says: "start with zero flow and augment, maintain a feasible flow and aim for optimality". *Pre-flow push paradigm* says: "start with super-optimality first, then aim for feasibility"

This means that we first try to push the flow through the network without checking that for each vertex the *conservation rule* applies (*conservation rule* – for each

vertex other than source and sink, the *inflow* equals *outflow*). And in the second phase we make the flow feasible.

In the pre-flow push algorithm we use two further definitions. We define: *Excess* (*pre-flow*) of a vertex v as: e: $V \rightarrow R^+_0$ where $e(v) = f^+(v) - f^-(v)$; *Height* of a vertex h: $V \rightarrow N_0$

Initially we set the *height* of each vertex v as the shortest path distance (number of edges on the path) between v and t (*sink*) and set the *excess* of source to infinity. Then we iterate the following principle: Find a vertex with non-zero *excess* and try to *push* the excess further. If we are not able to *push* the whole *excess* out of the vertex (other then *s*), we increase its *height*. By "*pushing*" we mean moving the excess (or part of it) from vertex u to its neighbours v such that $h(u) = h(v) + 1$ and there is an unsaturated edge $<u,v>$ or there is an edge $<v,u>$ with a non-zero flow in the network.

The iterations converge because the biggest *height* a vertex can achieve is $2.n$ because after that, the vertex is surely able to push the excess back to *s*, *which* can have the maximum height of $n-1$. Thus the worst-case (and also average) time complexity is $O(n^2)$. This is also the best-known theoretical result.

For a better illustration of the principles of the Pre-flow push algorithm see attachment 2 that illustrates the steps of this algorithm on the graph from picture 3.

3.8 Minimum s-t cut

The goal of algorithms that solve the minimum s-t cut problem is to partition vertices of a network into two sets A and B such that $s \in A$, $t \in B$ and the sum of weights of edges between A and B is minimized.

There are two ways how to compute it. The naive way is to try to partition V in all possible ways and check the sum of edge weights between A and B. However, this algorithm requires exponential time depending on the size of the graph.

A more useful way comes from the max-flow min-cut theorem proved by Ford and Fulkerson in [1]. This theorem enables us to compute the maximum flow in the network and find the minimum cut using the computed flow. The idea is very simple. We start with a set A that contains only s and then run a breadth-first search and add all vertices reachable from A through unsaturated edges or reversely oriented edges containing a flow. When no vertices are reachable from A through such edges, we set $B = V \setminus A$ and the edges between A and B form the cut and the total weight of the cut is equal to the value of the flow. For proof see [13].

As the minimum cut algorithms have application in various fields, multiple studies have tried to improve the performance of these algorithms or proposed faster approximate methods. In this paper we mostly use the method explained above and use the Pre-flow push algorithm for the flow computation. For alternative approaches see [13].

4. Graphs and graph theory in information retrieval

As already noted, graph theory has many applications in various fields. In the following chapters we provide an overview of graph theory usage in the field of information retrieval. We provide a detailed overview of graph-based ranking algorithms, algorithms for finding communities and also provide a brief overview of other interesting applications of graph theory in IR.

A significant part of the algorithms described bellow takes advantage of the hyperlink-induced web-graph. In this graph, each web page is represented by a node and each link is denoted by a directed edge. Even though the web has no obligatory structure and every user can create and add pages with any content we show that the web has a self-organizing structure. This structure is shown to contain valuable information and can be utilized to help to solve various IR problems as e.g. relevancy ranking or finding communities.

The most stress in this thesis is put on the ranking algorithms because these are trying to improve the most critical property of the IR systems, the *relevancy* (see chapter 2.1). We provide a very detailed description of the most important of these rankings – the PageRank, and also present various perspectives to its optimisation and improvements. We also provide a description of our implementation of the PageRank algorithm designed and optimized for the Egothor 2.0 search engine developed by RNDr. Leo Galambos from the Charles University [14].

In chapter 6 we describe several max-flow min-cut based approaches to finding communities. These approaches provide an application of min-cut algorithms that is not straightforward but is shown to be very effective.

In the last chapter we provide an overview of some other graph-based algorithms that try to improve some other characteristics of the IR systems. Again some of these algorithms take advantage of the web-graph, but we also present a few algorithms that create and work with a totally different graph structure.

5. Ranking Algorithms

In this chapter we describe graph-based relevancy ranking algorithms, which can be divided using several criteria. In this paper we describe three main categories. We start with the query independent web-graph based ranking – PageRank, then we describe some query dependent web-graph based ranking as e.g. HITS and in the last subchapter we describe some graph-based approaches that do not use web-graph structure at all.

The query independent rankings provide a general ranking of documents based on some inter-document structure. One such ranking is citation ranking that assigns each document a ranking directly proportional to the number of documents that cite this document. In this paper we provide a detailed description of query independent web-graph based page ranking algorithm - PageRank. We also describe our implementation of PageRank algorithm designed and optimized for the current version of the Egothor search engine.

In the following subchapter we describe some web-graph based query dependent ranking algorithms, point out their major pros and cons and give comments to realistic usage of these algorithms. As we show, the most critical property of the query dependent ranking is the time consumption in the query time.

The last subchapter describes some other graph-based rankings. In this chapter we show that the web-graph is not the only graph used by ranking algorithms.

5.1 Query independent ranking – PageRank

In the last several years, The World Wide Web (WWW) has witnessed an exponential growth in size. The number of pages on the web has grown from a few thousand in 1993 ([15]) to more than 25 billion ([16]) in 2006. Due to this boom, search engines are becoming ever more important tools for locating relevant information. The amount of available information on most topics has given rise to the importance of not only finding documents relevant to a given query, but also sorting them in descending order by estimated relevancy. The relevancy is influenced by many factors, some of which have been mentioned earlier. In the WWW there is one more instrument that helps us to better estimate the relevancy - the hyperlink structure.

PageRank, probably the most important example of graph theory usage in web IR systems, uses the information of the hyperlink structure of the web to rank web

pages. Hyperlinks are a useful instrument for simplifying navigation on the web however the important information comes from the reason why authors create these links. From existence of a hyperlink between pages *p* and *q* we can assume that author of *p* thinks *q* is related to *p*.

The PageRank of a page is an estimation of general relevancy of that page. It is calculated solely from the link structure of the web and its main power comes from the fact that it uses the content of other pages to rank the current page. It was developed at Stanford University by Larry Page and Sergey Brin as part of a research project held between 1995 and 1998 that led to a functional prototype of Google [17]. The name PageRank is usually explained either as web page ranking or as Larry Page's ranking. According to [18] the latter is the right one. Even though PageRank is only one of several factors that determine Google's relevancy ranking of documents to a given query, it is one of the main reasons why Google became so popular. While nowadays Google claims considering more than one hundred factors [19] when calculating the results of a query, PageRank still provides the basis for it.

In this chapter we provide an introduction to the PageRank computation and usage and also describe several approaches to improving the run-time and the precision of this ranking. In the subchapter 5.2, we also describe our implementation of the PageRank algorithm designed for the Egothor search engine.

5.1.1 Intuitive Background

PageRank can be intuitively explained as a model of user behavior. Imagine that there is a *random surfer* who selects a random page as the start of a web journey and then follows the links on the current page with uniform probability, but can eventually get bored, and instead of following a link, moves (jumps) to a random page and then continues the journey. The PageRank of a page is then the probability that the random surfer visits the page. This intuition is often thought of as a random walk and can be modeled by a *Markov chain* [20]. PageRank is then the stationary distribution of the Markov Chain.

Another intuitive justification comes from the citation ranking, but refines it. The citation ranking sets the number of citations of a document as its rank. This ranking, however, does not work well on the web, because it is vulnerable to manipulation as it is relatively easy to create lots of pages with no informative value pointing to a given page and hence improving its ranking. PageRank uses the idea of distributing the ranking through links, i.e. a page can have a high PageRank, if there are

a lot of pages pointing to it, or if important pages point to it. The idea is intuitively clear because pages that are linked by many other pages and pages that are linked from pages like Yahoo's homepage might be worth looking at. It is improbable that a low quality page would be linked from a high quality page (e.g. Yahoo's homepage), or that lots of pages with at least moderate quality would link to it.

Picture 4 PageRank intuition

5.1.2 Existence of PageRank distribution

In order to provide an explanation to existence of the PageRank distribution, we first need to define several terms. We use the notation $P(p)$ for PageRank of page p; D for *damping factor* – probability that the random surfer continues his journey by clicking on a link on the current page – usually set around 0.85; n for number of pages in our collection; and $d(p)$, $ind(p)$ and $outd(p)$ for total degree, in-degree and out-degree of page p respectively. With these definitions, we can write the formula for computing a PageRank of a page p as:

$$P(a) = \frac{(1-D)}{n} + D. \sum_{(b,a)\in E} \frac{P(b)}{outd(b)}$$

This formula shows that PageRank of a given page is derived from PageRank of pages that point to it. The problem is that such an equation is created for every page, so to compute PageRank of all pages in a collection, a huge set of linear equations needs to be solved.

However there is also another important point – the question of whether, there even is a solution to all of these equations. To be able to give a positive answer to this question, we need to handle a few special cases and use some knowledge from linear algebra.

We define the adjacency matrix $H = [H_{i,j}]$ *for* $i,j = 1$ to n as:

$$H_{i,j} = \begin{cases} 1/outd(a_j) & \text{if } (a_i, a_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Notice that H has some special properties. All entries in H are nonnegative and sum of each column is one unless the page is a sink (page with no links on it). To improve this, we set all values in the i-th column to $1/n$ if p_i is a sink. Now sum of each column of H is 1 and all values are nonnegative, i.e. H is *stochastic*.

In this matrix representation the PageRank of all pages can be represented as a vector P where $P_i = P(p_i)$ *i.e.* P_i is the page rank of i-th page in the collection. Now we can express the equation for computing PageRank as:

$$P = H.P$$

I.e. P is the eigenvector of matrix H with the eigenvalue 1 or P is the *stationary vector* of H. The existence of such a vector implies from the stochastic property of H (for proof see [21] or [22]).

5.1.3 Computation of PageRank

When we know that PageRank assignment exists we might want to know how to compute it. The "naive" way is to create an adjacency matrix H and compute its stationary vector using standard eigenvector computation techniques. However, computation on a matrix having n = 25 billion (which was the estimated size of web in 2006 [16]) rows and columns is prohibitively memory expensive. To store such a matrix we would need 2.5 Zettabytes (Zetta = 10^{21}) when using 32-bit floating-point numbers for storing the value of its elements. This is far more than our current resources allow and given the fact that web grows much faster than the computational capacities, future use is also improbable.

A good improvement of the memory effectiveness comes from the observation that (according to [23]) the average number of links on a page is 10 (in our mff.cuni.cz collection it is 22). This means that in average in each column all but 10 entries are zero. This fact motivates us to use another graph representation – the adjacency list, mentioned in chapter 3. The adjacency list can in this case be thought of as an array of size n where the *i-th* record consists of an integer number m, meaning the number of links on page i and a list of IDs of pages linked from page i. For sinks, we only store $m=0$. Given the average number of links contained on a page is 10 the memory needed for storing this structure for n = 25 billion is 2.05 Terabytes (we need to use the 64-bit

integer numbers for storing page IDs (the range of 32-bit unsigned integer is 0 to approximately 4.3 billions) and we need to store a 16-bit unsigned "short" integer – the link count, for each page). This is a fairly better result however it is still quite over current (top computers) possibilities. It is, however, the most widely used representation structure of the web graph. It is also important to note that the web-graph is not the only structure we need to store.

Having the web graph represented by the adjacency list structure we can use another approach to computing the PageRank distribution – the power method. The general power method can be thought of as an iterative algorithm, where in each iteration we use the results of the foregoing iteration to get a new result that is closer to the definite solution. In our PageRank computation, we use the power method in the following way. We start with a vector P^0 (initial PageRank) of size n where each element is equal to 1/n. In the i-th iteration (for i = 1, 2...), we initialize elements in P^i to $(1\text{-}D)/n$ and then we distribute rank from all elements of P^{i-1} in the following way. If page j contains m links, P^i values of pages linked to by $j\text{-}th$ page are increased by $D.(P^{i-1})_j/m$. If page j is a sink, all elements of P^i are increased by $D.(P^{i-1})_j/n$. I.e. in each iteration each page distributes D % of its PageRank uniformly to all its neighbours and in case the page is a sink, to all pages from the collection.

$$P_j^i = \frac{(1-D)}{n} + D\sum_{k\to j}\frac{P_k^{i-1}}{outd(k)}$$

It is proved (for proof see [23]) that when $D \in (0,1)$ P^i converges to the stationary vector. The *dumping factor D* is usually set to 0.85 and empirical tests (see [24]) showed that in average after 20 – 50 iterations the relative ordering of pages is close to the converged state. The power method and adjacency list structure are also used in our implementation of PageRank computation.

Algorithm 1 PageRank computation

```
// Initialization
for i = 1 to n do
    NewP[i] = 1/n

// Main iteration
while ( |NewP – OldP| > ε ) do
begin
    OldP = NewP
    // Initialization of the the rank vector
    for i = 1 to n do
          NewP[i] = (1-D)/n

    for i = 1  to n do
          for all j such that page j is linked from page i do
                NewP[j] += D.OldP[i]/outd(i)

end while
```

5.1.4 PageRank I/O optimisation

One of the main advantages of PageRank is that it is a query independent measure, so it can be precomputed and then used for optimizing the structure of the inverted index file. It is believed that Google re-computes the PageRank once in every 3-4 weeks. These facts might imply that the PageRank calculation is not that time critical. However the slowness of Google in adding new pages into his index is one of the main drawbacks that Google is upbraid for. It is also one of the main reasons why a lot of people find Live search (from Microsoft) [25] more useful when searching for more fresh information. According to [26] the Google's PageRank computation takes 2-3 days, which results in impossibility to rank the newest pages. This information gives us a notion about how long might it take for a non optimized version of PageRank algorithm run on the whole web to converge.

Some people argue that the PageRank distribution does not change too much in time and so if we use the last computed PageRank as the initial vector in the new PageRank computation we can expect that the number of iterations till getting to converged state decreases considerably. However this is not exactly true. There are two main reasons contradicting this idea. First one is that the number of newly added pages is still increasing. The second argument is that according to [27] the average half-life of pages on the web is 10 days (in 10 days half of the pages on the web are gone; i.e. their URL's are no longer valid). This brings us to a conclusion that using the last computed PageRank when creating the initial vector might decrease the number of iterations needed, but only slightly.

There are scenarios where the time needed for PageRank computation plays a major role because it is computed multiple times. Extensions of PageRank like, topic-biased or personalized PageRank (both to be described later) are good examples of such scenarios. Thus a highly optimized PageRank computation is needed in order to make it possible to use these extensions on the web scale.

There are several approaches to PageRank optimisation. Some studies try to minimize the number of iterations needed, some try to speed up the time needed in each iteration, some examine the possibility to approximate PageRank using various heuristics etc.

In this paper we survey some of these approaches and give some comments on realistic usage of some recent algorithms.

5.1.4.1 File structure definition

For further use we describe the structures needed for the computation in more details. We assume the web graph is stored in an input file with the following structure. The first record in the file has the meaning of *number of pages* indexed through the crawl process. Then in the rest of the file, there is a record for each crawled page having the structure: *ID* of the crawled page; *Number of links* on this page; and the list of *IDs* these links link to, sorted by ID. As already stated, the average number of links on a page is a small constant, so the sorting of the ID's would not present a time critical issue.

The result of the algorithms is written to a file that is sorted by the PageRank values in descending order and has the form: *ID* and the *PageRank* of this page.

5.1.4.2 Memory non-critical scenario

In a lot of realistic scenarios web search engines work on just a fragment of the web, e.g. pages in the domain cuni.cz or mff.cuni.cz. In such cases the whole adjacency list and also the vectors for computing PageRank can be fit into main memory. In this case the easiest way how to compute PageRank of these pages is to load the input file into an adjacency list in the memory, create two arrays (vectors) of size n for rank computation and use the power method for the computation. A good time saving improvement can be achieved in treating sinks in a smart way. Instead of distributing a portion of PageRank to each page every time we process a sink, we can store the portion of PageRank by which this page (sink) contributes to the new rank of all pages to a

variable. At the end of the iteration we just add the sink contribution to new rank of all pages.

In such scenarios no further optimisation is needed. The only I/O operations performed are input file loading and saving the final PageRank file. The run time of our implementation of the PageRank algorithm, run on a collection of about 1.4 million pages crawled by the Egothor crawler, is approximately one minute.

5.1.4.3 Memory critical scenarios

In many common scenarios the size of the collection would, however, not allow us to store the whole web graph and the PageRank vectors in the main memory.

However, if the graph is stored as proposed earlier, the only data we need to store in the main memory (when using the above mentioned graph representation) to ensure an acceptable speed of the computation is the currently computed PageRank vector or at least a part of it. The reason for this is that in each step of the iteration any element of the newly computed PageRank vector can be changed. To write this information to disk every time would be prohibitively time expensive, because of the disk seek time. We review the options of computation we have with decreasing ratio between the size of main memory and the size of a given collection.

First we can imagine that we are able to store two PageRank vectors. One for the current computation and the other one containing results from the previous iteration. In this case the only change in the computation process is that instead of loading the input file into the memory, we read the information directly from the disk in each iteration. However if we store the graph in the above proposed way, we can read the whole file sequentially. The rest of the algorithm stays the same.

In an even more memory critical scenario, we might not even have enough memory for storing the two PageRank vectors. In this case we only keep the currently computed vector in the main memory and in each iteration we read the vector from last iteration and the graph from disk. Thus it would be beneficial to have the file containing the graph sorted by ID, so that both files can be read sequentially. This idea is also used in of the program versions implemented as a part of this thesis. Detailed description of the implemented algorithm can be found in chapter 5.2.4.

5.1.4.4 Haveliwala's algorithm

There are scenarios, where the size the PageRank vector exceeds the size of the main memory. A practical example of such a scenario is the PageRank computation for the whole web. Even if we only assume that the size of the web is 25 billion pages we would need more than 200 Gigabytes of main memory (when using a 64-bit floating point number) to store the currently computed PageRank vector. In such cases we can use an approach proposed by Haveliwala in [28]. The main idea of this approach is that if we can not store the whole currently computed PageRank vector into the memory, we only store a part of it that fits there leaving some space for other computations. We then process the input file in a way that would enable us to only process pages from the collection that contain links to pages in the currently stored part of the vector. To achieve this, we would first estimate the size *s* that we would be able to fit in the main memory and then we preprocess the input file in the following way. We create $q = n/s$ files. The *i*-th file contains information about all pages that have links (at least one) to any page in the range $i.s$ to $(i+1).s$ (for illustration see Table 2).

Original link file

Source page ID (8 bytes)	Out Degree (2 bytes)	Destination pages ID (8 bytes each)
0	50	1, 2, 4, 66, 122, 146, 222, 223,387..
1	8	102, 109, 199, 212, 333, 568..
2	14	4, 5, 6, 33, 98, 318, 355, 514..
3	3	14, 249,298
4	14	1, 2, 186, 313..

Link file divided into blocks (q = 100)

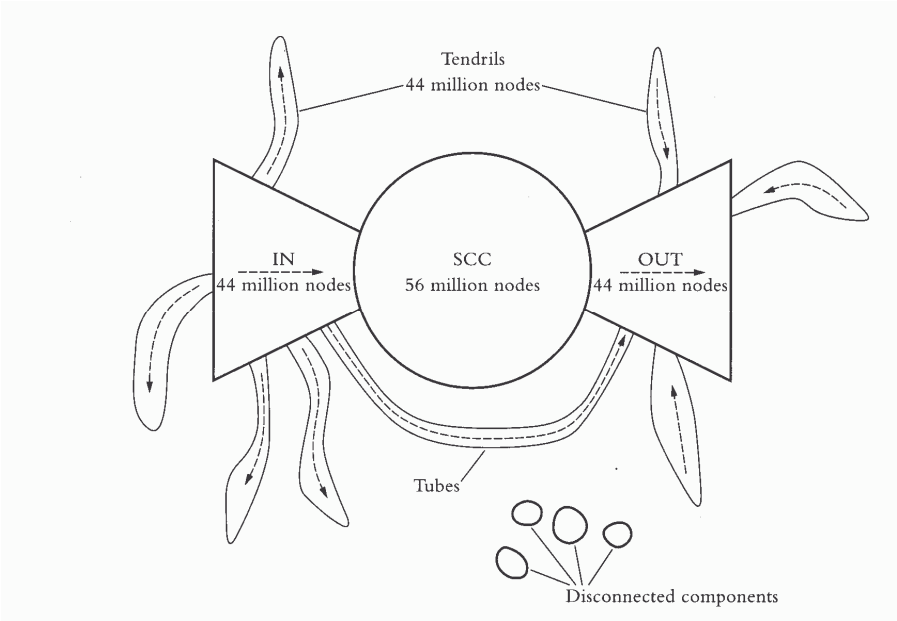
Source page ID (8 bytes)	Out Degree (2 bytes)	Number of links in block (2 bytes)	Destination pages ID (8 bytes each)
Block of links: 0 -99			
0	50	4	1,2,4, 66
2	14	5	4, 5, 6, 33, 98
3	3	1	14
4	4	2	1,2
Block of links: 100-199			
0	50	2	122, 146
1	8	3	102,109,199
4	4	1	186
Block of links: 200-299			
0	50	2	222,223
1	8	1	212
3	3	1	249, 298

Table 2 Example of link file partitioning in Haveliwala's algorithm

Having this structure of the input file we are able to compute the part of the PageRank vector stored in the main memory without the need of many disk seek operations. We then save this part of the vector for the use in the next iteration and free the memory for the next part of the vector. Empirical tests showed that this algorithm performs very well for moderate values of q. We present several improvements of the algorithm that would enlarge the scope of its usage, but that can also be used in other algorithms.

5.1.4.5 Use of the web graph structure

First we can use the results of web graph structure studies as for example [29, f30, f31]. These papers present a good overview of the web's organization. The study by Broder et al. describes the web having the shape of something like a bow tie. That is, about 28% of the pages form a strongly connected core (the center of the bow-tie). About 22% of pages that can be reached from the core, but cannot reach it themselves, form one of the tie's loops. The other loop consists of approximately 22% of the pages that can reach the core, but cannot be reached from it. The remaining nodes (pages) can neither be reached from the core nor reach it.



Picture 5 Bow tie structure of the web

Some papers (e.g. [32]) also present the idea of hyperlink locality. These studies proved that if the ID file is sorted in a smart way then the most of the links on a page link to pages with a close ID.

Results of these studies can be used to improve Haveliwala's algorithm by using an intelligent crawler (e.g. [33]). Such crawlers can provide a structure that could be processed to get an efficient block structure, i.e. majority of links on pages in a block would link to pages in the same block. This would result in a very good improvement in the run time of the algorithm, because it would minimize (or at least decrease) the number of pages stored in each block of the input file.

5.1.4.6 Compression techniques

A further improvement of Haveliwala's algorithm (but also all the others) can be achieved by using some kind of compression technique. A simple compression technique that is also used in our implementation is the Pack7 number format. It tries to avoid storing zero bytes in the memory by not using a uniform byte length for storing numbers. I.e. in each byte, first seven bits contain the value information and the last bit signalizes, if this is the end of the number, or if it continues in the next byte. This technique is intended for use on integer numbers, but can easily be transformed for use on the floating-point numbers, when we know an approximate range of these numbers. More about the Pack7 compression technique can be found in chapter 5.2.

In this paper we do not concentrate too much on the compression techniques, however for an illustration we present a concept of a compression technique that might decrease the memory requirements for storing the PageRank vector by a factor of more than 2.

Our compression technique is based on the following observations. As shown in [34] the distribution of the logarithmically scaled PageRank follows a power law distribution with exponent approximately *2.1*. I.e. the portion n_r of pages with rank r is approximately $c/(r.n)^{2.1}$ where c is a constant set so that the sum of n_r for all $r \in \{0,..9\}$ is equal to *1*. In this case the constant *c* is roughly equal to *1*.6. This fact implies that about *2*/3 of pages have PageRank very close to minimum and that the highest rank of a single page would mostly be close to $10^{-log(n)/2}$.

The second observation that lead to the concept of our compression technique comes from the fact that some papers assume computing PageRank not in the scale (*0*,*1*), but in the range (*0*, *n*). We assume that rounding (truncating) the floating-point numbers for storing the PageRank values 4-th decimal places bellow the first non zero number of the minimum rank would not have any major influence on neither the number of iterations nor the final PageRank values.

If we summarize these facts we come to the conclusion that the range of PageRank values we need to be able to store is approximately (10^{-4}.0.15/*n*, $10^{-log(n)/2}$), which is $\{0,..., 10^{log(n).5/2}\}$ in integer numbers. For *n = 25 billion* this is approximately the range $\{0, .., 10^9\}$ which is less than the range of a 32-bit unsigned integer which has range {*0*,.. *4*,*294*,*967*,*295*}.

It is now important to note that we actually know the minimum PageRank of all pages. So instead of storing this number, we can only store the difference between the PageRank of the current page and the minimum rank. This improvement causes that for a lot of pages we would only store *0* as its page rank.

Realizing all these facts we can use the Pack7 format for storing the PageRank value for all pages. Due to the power-law distribution of the PageRank values at least *2/3* of all pages would only need 1 byte for storing their rank and for more than half of the rest of the pages 2 bytes are sufficient for storing the rank which results in compression factor of more than 2 as presented earlier. It is also important to note that the upper bound of the highest PageRank we need to store is not crucial, because it would only be achieved by a few pages and so it would not change the final compression factor considerably.

Even though this compression technique is only based on a few simple observations, it can decrease the memory requirements for storing the PageRank vector by more than half, which implies that better results should be achievable.

5.1.4.7 Improvements summary

To summarize this we can come to the following conclusion. If the size of the web was currently 25 billion of pages and we used our simple compression technique we would need approximately 50 Gigabyte of main memory for storing the PageRank vector. This is achieved by using the observation about the range of PageRank values, which enables us to use 32-bit unsigned integer numbers for storing the values and the use of our compression technique.

This is already a size that might fit into the main memory of some top computers. So we might not even need the Haveliwala's algorithm, but can use the one of the algorithms mentioned earlier. However taking into account that it is highly effective for moderate values of q this algorithm can be used to efficiently compute PageRank for all pages on the web on computers with at least 20 GB of main memory. However, if we take into account that most common main memory size of the present computers is between *0.5* and *2* GB, further optimisation is deserved.

5.1.4.8 Split-Accumulate algorithm

Several other papers presented approaches that tried to minimize the number of I/O operations needed to compute the PageRank distribution. One of the best

performing algorithms in this category is the Split-Accumulate algorithm presented by Chen et al. in [15].

The main idea of this algorithm is derived from Haveliwala's algorithm, but uses it reversely. This algorithm again splits the vector for PageRank computation into q blocks V_i, such that each block fits into main memory leaving again some space for further computations. These blocks however only exist in the main memory and are written to disk only after the last iteration.

Further, this algorithm works with three sets of files L_i, P_i, O_i each containing q files. L_i contains for each page all pages from the i-th block that contain a link to it and also the number of such pages (see Table 3). O_i contains out-degree of each page from the i-th block. P_i is defined as containing all packets of rank values with destination in V_i in arbitrary order, i.e. P_i contains all rank values that influence PageRank of pages in the i-the block. E.g. For the example from Table 3 P_0 contains packets (*0*, (*1-D*)/*8n*+..), (*1*, (*1-D*)/*14n*), *etc*. in the first iteration. The packet (*0*, (*1-D*)/*8n* +..) contains information, that PageRank of page *0* should be increased by (*1-D*)/*8n* (initial rank divided by the out-degree of page *1*) + .. (initial PageRank divided by the out-degree of other pages in this block that contain a link to page *0*).

Original Link file

Source page ID (8 bytes)	Out Degree (2 bytes)	Destination pages ID (8 bytes each)
0	50	1, 2, 4, 66, 122, 146..
1	8	0, 9, 199, 212, 568..
2	14	0, 1, 5, 6, 33, 114..
100	12	0, 2, 4, 20..
101	23	3, 4, 21..
200	51	0, 1, 2, 3, 4..
201	3	0, 2, 202

Link files - L_i

Destination page (8 bytes)	In-Degree (2 bytes)	Destination pages ID (8 bytes each)
Block of links 0: Source 0 -99		
0	6	1,2,4, 66, 77, 98
1	4	0, 2, 4, 21
2	3	0, 4, 87
4	4	1,2
Block of links 1: Source 100 -199		
0	40	100, 102, 120, 122..
2	14	100, 102, 104..
3	1	101
4	3	100, 101, 103
Block of links 2: Source 200 -299		
0	50	200, 201, 203..
1	14	200, 223, 228..
2	2	200, 201
3	1	200
4	4	200, 209, 212..

Table 3 Example of link file partitioning in the Split-Accumulate algorithm

The algorithm works in iterations where each iteration has q phases. In each phase we first initialize all values in vector V_i in the memory to (*1-D*)/*n*. Then we run a scan through the file P_i (that contains all packets with destination in V_i) and add rank from each packet to the appropriate entry in V_i. After finishing the scan through P_i we load the file O_i (out-degrees) and divide each rank value in V_i by its out-degree. Then

we run a scan through L_i and for each record in L_i that contains some (at least one) sources in V_i and a destination in V_j we write one packet with this destination node and the total amount of rank to be transmitted to it from these sources and output it into file P'_j (that is used as P_j in the next iteration).

The idea of this algorithm is not as straight forward as the one of the algorithm proposed by Haveliwala. Empirical tests showed that for moderate values of q (number of blocks the PageRank vector has to be divided to, so that the blocks fit into the main memory) the Split-Accumulate algorithm performs similar to Haveliwala's algorithm. However with increasing values of q the time consumption of the Split-Accumulate approach if significantly lower than the one of Hawelivala's approach.

If well implemented, this algorithm already enables us to compute PageRank on the web-scale on common home PC's without need for any significant compression. The paper [15] also shows how this algorithm can be slightly modified to efficiently compute the Topic-Sensitive PageRank. We return to this application later.

In this chapter we presented algorithms that allow us to compute PageRank even for massive graphs without a need of the best high-tech hardware. The computation would however take quite a long time and the potential of usage on common computers is still limited.

5.1.4.9 I/O efficient approaches summary

In the previous subchapters we showed that there are several ways how to speed up the power-method computation of the PageRank distribution. The Split-Accumulate algorithm is shown to be able to efficiently compute the ranking even on computers with common memory size. However we also noted that the computation on the web-scale would take a few days even on above average computers, which is still very limiting. These approaches only attempted to speed up the computation using various ways how to distribute the data in a way that would minimize the number of I/O operations needed. In the next subchapter we describe some approaches that use alternative ideas.

5.1.5 Other approaches to PageRank optimisation

As we showed in the previous chapter, even though the size of the web is huge, it is still possible to compute PageRank of all pages on it. However, the computation takes a really long time even when using a highly (I/O) optimized approach. Hence, people examined other approaches how to speed it up. One approach that can easily

come to our minds is not to compute the PageRank as it is defined, but to try to approximate it in a way that would enable a quicker computation. This might be a reasonable approach because PageRank already is an approximation of user behavior, so a further approximation might still have the required property of providing a general ranking of web pages while requiring less computation time.

A naive way of utilizing this idea would be to try to use a bigger (than usual) epsilon in the convergence check in the basic PageRank computation. However this approach reduces the run time only slightly and the trade off between the lost precision and the saved run time is not reasonable.

Several papers (e.g. [35]) presented algorithms that try to approximate PageRank computation using the sites (for simplicity mostly modeled by hostnames) as nodes of the web graph and then distribute the rank to pages in the site using only the intra-site links. These approaches take into account that a lot of documents (e.g. PowerPoint documents) consist of multiple web pages and that pages on one site are often topically related. So they consider the site level to be the right level of granularity for the relevancy analysis. Further on we use the terminology illustrated by Table 4.

URL Terminology	
Term	Example: en.wikipedia.org/wiki/
top level domain	org
domain	wikipedia.org
hostname	en
host	en.wikipedia.org
path	/wiki/
web page	en.wikipedia.org/wiki/

Table 4 URL Terminology

Another fact that speaks for using the site approximation is that the amount of web sites (hosts) is considerably smaller than the amount of web pages. According to [36] there were less than 100 million of registered hosts and about 25 billions of web pages in October 2006. Another important fact is that according to [37] more than 75% of all hyperlinks on the web are intra-host links. These facts imply that the PageRank computation on the host graph should be substantially more time efficient. The following two subchapters show that this assumption actually holds.

5.1.5.1 PageRank approximation via Graph Aggregation

An algorithm utilizing this idea was proposed by Broder et al. in [35]. The main idea of this algorithm is to compute PageRank of all sites and then to distribute this rank to pages in each site. In their approach they were able to achieve a (Spearman) rank-order correlation of 0.95 in respect to PageRank while requiring less than half of the running time of a highly optimized PageRank implementation.

To be able to explain the principles of this algorithm, we need the following definitions. Let the n nodes be partitioned into m classes $H_1,..,H_m$ by their hostnames. Let $P = [p_{i,j}]$ denote the stochastic matrix (of size n x n) defining the web graph on the page level. Then we define an alternative random walk T derived from P, whose stationary distribution can be computed more efficiently. In T the random walk consists of two basic steps. First we move to some node $y \in H_i$ with respect to the distribution Π_i, (where Π_i only depends on the class of y - H_i) then we perform a step from y according to P.

The algorithm then works as follows:

We define (compute) an m x m stochastic matrix R = [ri,j] (for calculating the stationary distribution of T) as follows:

$$r_{H_i, H_j} = \sum_{q \in H_i} \pi_i(q) \sum_{p \in H_j} p_{q,p}$$

Then we calculate the stationary vector of R, i.e. we compute a vector A satisfying $AT = A$.

We then compute the vector P' of size n where for each page p, $P'(p)$ would be computed as follows ($h(p)$ denotes the class (host) to which page p belongs):

$$P'(p) = A_{h(p)}.\pi_{h(p)}(p)$$

The stationary distribution of T is then defined as the vector $B = P'T$.

The main advantage of this approach is that we only run the iterative power method on a graph that is substantially smaller than the web page collection. As stated above, the average number of web pages on a host is approximately 250. Compared to other algorithms for computing PageRank, this might be a source of substantial time saving.

It is also important to note that it is very difficult to describe how the differences between PageRank and this approximation would influence the overall search quality of

search engines using it. It might even turn out that this might be a more appropriate model.

5.1.5.2 BlockRank algorithm

Golub et al. in [32] proposed an algorithm very similar to the one described above. Actually they use an algorithm very similar to the one mentioned above, to get an initial ranking for all pages and then try to run standard PageRank algorithm expecting only a few iterations needed till getting to the converged state.

In this algorithm they use the "naive" host graph. This graph structure uses hosts as nodes and there is an edge between two hosts *i*, *j* if and only if there exists a page on *i* that contains a link to a page on *j* (the weighted host graph has the same structure, but allows multiple (or weighted) edges between hosts based on the real number of links between pages on these hosts). They also described a simple method how to create this structure. They propose sorting the link file (containing the url index) lexicographically, but with reversed order of URL components before the first slash (e.g. the sort key for www.ms.mff.cuni.cz/~kopecky/ would be cz.cuni.mff.ms.www/~kopecky/). Then each host is assigned an ID and the adjacency list structure is created.

In the proposed algorithm they use this structure as input for a standard PageRank algorithm, which assigns a rank to each of the hosts. This rank is then distributed to the pages in each host using a local PageRank algorithm (standard PageRank algorithm that computes PageRank on the graph of this host and then weights it by the rank of the host). This part of algorithm results in a vector very similar to the one computed by the algorithm proposed by Broder et al. in [35]. However in this algorithm this vector is used as the initial vector for the standard PageRank computation on the page-induced graph.

According to time measures provided in their paper, the last step takes more than 70% of the time computation. This time can probably be improved by using a weighted host graph, which is believed to produce more precise host ranking.

Another improvement of this algorithm can be achieved by performing the Local PageRank algorithm during the crawl process as soon as the whole host has been crawled. The precomputed Local PageRank would then just be weighted by the rank of its host. Also it is important to note that the Local PageRank computation is highly parallelizable because the block PageRank values are completely independent.

The proposed algorithm is (again) described to be approximately two times faster than the standard PageRank algorithm. However it is only slightly better when compared to PageRank algorithm run on a graph with sorted URL list. This could be improved by Local PageRank computation parallelization and using the weighted host graph in the host rank computation.

The main advantages of this algorithm are that by using the block structure and the sorted link structure the number of I/O operations is decreased significantly. Also the Local PageRank computation converges quickly because the number of pages in a site is mostly a small number (they showed that approximately 75% of the local rank computations converge in less than 10 iterations). All these pros apply to both approaches (Golub et al. and Broder et al.). However this algorithm has one more significant advantage. It enables very quick (approximate) rank re-computation after node update. This can be achieved by storing the sum-rank (sum of rank of all pages on a host) of all hosts and after the node update only compute the Local PageRank of that host and weight it by the sum rank of this host. This approach might help to keep the index of web search engines more up-to-date.

5.1.5.3 Other site-based algorithms

Recently several other papers proposed using the site (host) structure, although we think that these brought only minor improvements or proposed a more appropriate host rank calculation ([38]) that however requires much more time than the weighted host graph computation and so it looses the required time saving property.

Though we think the host-based approach to PageRank computation has a lot of advantages (e.g. time efficiency) and because of its simple idea we think it will very probably become widely used. It is also probable that it is already utilized in various search engines.

5.1.6 PageRank extensions

The major success of PageRank very much influenced the web search engines. The idea of pre-computing some measures that would help better estimate the relevancy to a given query is however much older. It is already the basis of indexation process. The first indexes only stored key terms contained in the documents. Nowadays however, they also try to store the importance of the term to identification of the document. So the evolution of indexes tried to improve the information richness of the stored data.

The same idea applied to PageRank resulted in two PageRank extensions. One combines the topical distribution and the link structure and other one tries to personalize the PageRank distribution.

5.1.6.1 Topic-sensitive PageRank

The topic-sensitive PageRank (proposed by Haveliwala in [39].) is a very simple extension of the standard PageRank algorithm. It performs a separate PageRank calculation, for each topic. In his approach Haveliwala chose the top class topics from the ODP (Open Directory Project [40]) for the topic biasing.

The topic-sensitive PageRank for topic j and page i ($TSPR_j(i)$) is then defined as the standard PageRank random surfer model except that when the random surfer gets bored instead of jumping to a random page, he randomly jumps to one of the t_j (number of pages within the *j-th* topic) pages within j. So the TSPR is defined as follows:

$$TSPR_j(i) = D.\sum_{k:k\to i}\frac{TSPR_j(k)}{O(k)} + \begin{cases}(1-D)/|t_j| & if\ i \in t_j \\ 0 & if\ i \notin t_j\end{cases}$$

To rank the results for a particular query q we compute $C(q,j)$, the relevancy of the query q to topic j. Then for page i the query-sensitive importance ranking is

$$S_q(i) = \sum_j TSPR_j(i).C(q,j)$$

The algorithm based on this approach first computes PageRank for each topic and then provides a query-time efficient document ranking. Its main advantage is its simplicity and parallelization (PageRank for each topic can be computed separately). The topic-biasing is however relatively weak and the PageRank for each topic is in fact a basic PageRank just partly skewed towards pages contained in the page list for the current topic in a human processed directory.

5.1.6.2 Topical PageRank

A much more complex approach to topic-biased PageRank was proposed by Davison et al. in [41]. They proposed a ranking that can also be used in other link based rankings such as HITS (described in chapter 5).

This approach tries to improve the ranking by incorporating the page content into the ranking process. As the PageRank already is a very memory critical application we need to use a memory efficient abstraction of the page content. For this purpose a set of topics T is chosen (usually top-level topics of a directory like ODP [40] or Yahoo

directory [42]). A measure $C(p_i)$ meaning the relevancy of page p to topic i (in percentage) is then stored for each document and each topic. The measure $C(p_i)$ has the property that the sum through all topics is equal to one. It can be computed by textual classifiers such as the one proposed in [43]

In this approach the *topical random surfer* model is used. This model is very similar to the standard random surfer except that the topic sensitivity is added. In this algorithm we assume that if the surfer is interested in topic k and is on page p then in the next move he might do one of the following steps. He might either follow an outgoing link on the current page with probability D or teleport to a random page with probability $(1\text{-}D)$. The only difference (comparing to classical PageRank) is that when following a link, the surfer is with probability A likely to stay on the same topic (action - *follow-stay* F_S), however with a probability $(1\text{-}A)$ he may jump to any topic i in the target page (action - *follow-jump* F_j). Though, the topic bias is only present when the random surfer follows a link. When he is taking the teleport (*jump-jump* J_j) action he is always considered to take a random topic. The constant D is usually set the same way as in standard PageRank algorithms, i.e. $D = 0.85$. The probabilities of taking certain action when browsing page v while interested in topic k can be described as:

$$P(u_i \mid v_i, F_S) = \frac{1}{outd(v)}$$

$$P(u_i \mid v_k, F_J) = \frac{1}{outd(v)} C(v_i)$$

$$P(u_i \mid v_i, J_J) = \frac{1}{n}.C(u_i)$$

The probability to arrive at topic i in target page u by the defined actions is:

$$P(F_S \mid v_k) = D.A$$
$$P(F_J \mid v_k) = D.(1-A)$$
$$P(J_J \mid v_k) = (1-D)$$

The probability that the surfer is on page u for topic i can in this model be computed as:

$$\begin{aligned} R(u_i) = & \sum_{v:v\to u} P((u_i \mid v_i, F_S)P(F_S \mid v_i)R(v_i)) + \\ & \sum_{v:v\to u}\sum_{k\in T}(P(u_i \mid v_k, F_J)P(F_J \mid v_k)R(v_k)) + \\ & \sum_{v\in G}\sum_{k\in T}(P(u_i \mid v_k, J_J)P(J_J \mid v_k)R(v_k)) \\ = \quad & D.A\sum_{v:v\to u}\frac{1}{outd(v)}R(v_i) + \\ & D(1-A)\sum_{v:v\to u}\frac{1}{O(v)}C(v_i)\sum_{k\in T}R(v_k) + \\ & \frac{(1-D)}{n}C(u_i)\sum_{v\in G}\sum_{k\in T}R(v_k) \\ = \quad & D\sum_{v:v\to u}\frac{AR(v_i)+(1-A)C(v_i)R(v)}{outd(v)} + \frac{(1-D)}{n}C(u_i) \end{aligned}$$

These equations look very scary on the first sight. However, it is not difficult to understand them with a few comments. The part between the first two equal signs can be explained as follows: Each line contains a rank contribution of each action multiplied by the probability of taking this action. So for example the first line (after the first equals sign) indicates the sum of rank contribution of action FS to rank of page u to topic i, i.e. it is the sum (through all vertices v containing link to u) of the probability of moving from v to u when taking the action FS multiplied by the probability of choosing action F_S (i.e. probability that the random surfer follows the link between v and u while keeping the same topic interest) multiplied by the rank of page v on topic i. The second and third lines describe similar probabilities for the other two actions.

In the second part we only substitute the probabilities with actual constants using the equations above. The last part (after the last equals sign) only substitutes the following two sums by their equivalents and groups sums with identical ranges.

$$R(v) = \sum_{i\in T} R(v_i)$$

$$\sum_{v\in G} R(v) = 1$$

Such an equation is created for each page and each topic. However the definition allows us to compute all of the ranks (for all topics) simultaneously. To compute the solution of such a set of equations we can again use the power-method described earlier. All the conditions essential for the distribution convergence (see chapter 4.2) hold if $A\in(0,1)$ so the power method usage is feasible.

The best results of this algorithm were achieved by not setting A constant, but deriving it from the relevance of the current page to the current topic (Such use of A still satisfies the power method convergence conditions.). This improvement does not require any further computation, because we have to compute relevance of all pages to all topics before the start of this algorithm. However, it turned out to deliver a good improvement in precision. The intuition behind this idea is that when a topic-biased browser gets to page p when interested in topic i and the relevance of a page to the topic is low, he might be more likely reset his current topical interest.

After the computation converges, each component $R(u_i)$ of the vector R_u represents the ranking of page u to topic i. $R(u)$ is the overall PageRank of the given page and it is identical to rank produced by standard PageRank algorithms. I.e. the topical PageRank provides us with both, the PageRank for each topic as well as the global PageRank for each page.

This algorithm provides a very complex approach to integrating topics into the PageRank computation. However, the possibility of using it on the web-scale is a big question. The main drawbacks of this approach are its very high memory and computational requirements. In the standard approaches we only work with the currently computed PageRank vector (or its part) stored in the main memory and with the adjacency list and the last computed PageRank vector stored on the disk and we already need to use sophisticated algorithms to be able to calculate the PageRank on the web-scale in a reasonable time.

However, in order to compute the Topical PageRank, the stored structures have to be enlarged. The main memory requirements are more than $|T|$ times higher because of the need to store |T| distinct PageRank values and the need to store the content vector C for each processed page. The structures stored on the hard disk require more space because of the need to store the topical rank vector (instead of the simple PageRank vector) and the need to store the content vectors C of all pages. Hence, each computational step is more complex compared to standard PageRank.

These facts imply that a highly sophisticated algorithm would need to be used, when computing the Topical-PageRank on the web-scale. The best (time) performing algorithms are considered those that utilize the host graph structure. The idea of approximating multiple pages by hosts might also work for the Topical-PageRank. It is very common that pages on a single host are highly topically related which implies that the topical relevancy generalized to the site (host) level might provide a good

approximation. However, the topical relevancy generalization would have to be done properly otherwise the following bad case could happen.

Imagine a host containing a lot of pages highly related to topic *i* also contains a few related to topic *j* where the latter pages are highly relevant (highly rated) to this topic. So it might happen that the topic relevancy vector of the site would only contain a very little or no portion of relevancy to the *j-th* topic. This might result in requiring a lot of iterations to get to the converged state after using the precomputed Topical-PageRank as the initial vector. However this might not be a common case and given the fact that amount of host is substantially smaller than the amount of web pages, we might be able to use a very precise topic relevancy approximation at the site level, which significantly decreases the risk of such cases.

As this approach is relatively fresh (publicized in August 2006) we might expect various studies trying both to improve the run-time and the precision of this algorithm. We think that this is a very perspective approach and it would be very interesting to follow its evolution.

5.1.6.3 Personalized PageRank

In this subchapter we just provide a very brief introduction to the principles and perspectives of Personalized PageRank.

The main idea of this PageRank extension is very similar to that used in topic sensitive approaches. The personalization can again take place in various parts of the computation. A simple approach might consider that the user has a certain set of favorite pages *F*. The random surfer then prefers these pages when doing the *jump* (teleport) action. Another approach might use the idea that the user has certain preferred topics and when the random surfer chooses a link to follow, links leading to pages related to topics of his interest are favored.

Such approaches might be very useful. It might not be reasonable to compute personalized PageRank for each user, however it might make sense to create user categories and after gathering some information about the user, to try to assign him to one of these categories. The current search engines (e.g. Google) already try to gather a lot of information about its users. They have access to details about queries submitted by certain user and they can often incorporate it with a few important profile details got from providing the user with a free-mail service and from the time zone and regional settings of users browser etc.

This information may be extensive enough to enable us to well categorize the users. It is believed that Google has been gathering such data for several years. It is however questionable if this way of data gathering is ethical and how it can be utilized. A utilization that might be convenient for the user is the personalization of query results. However it is also believed that Google uses this information (mainly) for improving the "accuracy" of its advertisements, which might not be in accordance with user's desire. Multiple discussion about this topic are held on the Internet (see e.g. [44])

5.2 Our implementation

As part of this thesis, we have also implemented two versions of algorithm computing the PageRank distribution in C++. The *M* (as memory non-critical) version is derived from the concept presented in chapter 5.1.4.2 and the *D* (memory critical – using *d*isk) version implements the concept from chapter 5.1.4.3. Both versions are multiplatform and were tested on Windows and Gentoo (UNIX).

5.2.1 File and Pack7 formats

Both these programs work with binary input files where all page ID's are stored in the Pack7 format. This format uses variable byte length for storing integer numbers thus provides a simple compression. In the Pack7 format in each byte the lowest seven bits contain the information about the number and the highest bit signalizes if the information continues in the next byte. In Table 5 we can see the representation of number 517 first in normal format and then in the Pack7 format. If the number were represented as a 16-bit integer, using the Pack7 format would not bring us any benefit. However, is the number is stored as 32-bit or 64-bit integer the Pack7 format we would still need only 2 bytes.

0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	1

1	0	0	0	0	1	0	0	0	0	0	0	0	1	0	1

Table 5 Normal and Pack7 representation of number 517

This format is very suitable for the PageRank computation where ID's of pages have a wide range and so have to be saved in at least 32-bit integer numbers. Nevertheless, if the indexing is done properly then the most frequently used pages (i.e. pages with the highest in-degrees) will have the lowest ID's and so the Pack7 format

would provide a comprehensive compression ratio with only a slight processing overhead (see Table 6).

Both these programs use two input files and write the computed PageRank distribution into an output file. All of the files use 64b integer numbers in the Pack7 format.

The input files contain information about the link structure of the pages in the collection and about redirections between pages in this collection. The link file has the following form:

source	out-links	dest[1]	...	dest[out-links]

source: 64b ID of the source page

out-links: number of links on the source page

dest[]: array of 64b ID's of target pages

and the redirect file has the form:

source	destination

source: 64b ID of page from which we are redirected to page dst

destination: 64b ID of page to which we are redirected from page src

The *D* versions executed without the –p parameter produces the file containing the computed PageRank in the following format:

ID	rank

ID: 64b ID of a page

rank: 64b double precision floating point number representing rank of the page

All other versions generate the final rank in a text format to allow easier reading.

5.2.2 Program principles

Both program versions work on the same principles. We first describe the common principles and then explain the differences and improvements made in both versions.

Both programs first load the link graph into a dynamically allocated structure where for each page the number of its neighbours and ID's of its neighbours are stored. Then the redirects file is loaded to a structure, where for each page we store the ID of the page where we are redirected to, or zero if no redirection is done. This structure is then used to modify the link structure. All pages that contain a redirection are replaced

by the pages where the redirection links. The link structure is then processed further. All links to not-crawled pages (pages with no record in the link file), duplicate links and links to zero are erased and the new link count is calculated. This structure is then stored for the rank computation.

The rank is computed using the power-method algorithm described in chapter 5.1.3. When checking the rank change we compute the change of rank of each page in percentage and then compare the maximum change with ε, which we set to 0.1, i.e. the computation ends as soon as no rank of a page changed by more than a 0.1 percent in the current iteration. The pages are then sorted in descending order by their rank and are written to the output file in the form mentioned above.

5.2.3 *M* version

The *M* version program was created only for testing and comparison purposes. It contains not only the procedures for computing PageRank distribution, but has also other functionalities as rewriting the graph file in Pack7 format into a text format, creating sub-graphs of the input graph etc. These functionalities are, however, present only for further research and are not discussed further in this paper.

To compute the PageRank distribution the program has to be run with the following parameters: -e "link file" –r "redirect file" "output file". As already mentioned this program is based on the non-memory critical scenario, so all structures needed for the computation are kept in memory. I.e. for each page we keep not only information about the link graph but also the values of new and old PageRank in the memory. The memory requirements are discussed in the chapter 5.2.6.

In the computation we used one optimisation that was based on the fact that after preprocessing the graph, we erased almost half of the indexed pages. So for all pages that were not crawled, instead of saving the number of their neighbours, we save the number of not crawled pages that directly follow this page in the ID list. This enables us to jump over these blocks during the computation.

5.2.4 *D* versions

The *D64p7* version is the version intended for use in the Egothor search engine. This version contains several differences comparing to the *M* version. It is intended to be used on larger subsets of the web and is implemented in such a way that should allow computation on the web-scale. In the tests bellow we present several variations of *D* version – *D64*, *D64p7*, *D32* and *D32p7*. The numbers *32* and *64* denote how many bites

are used for storing the page ID's. The only difference between the *p7* and the "standard" version is that after preprocessing, the *p7* version saves the graph using the Pack7 format while the "standard" version stores in a standard binary format The run-times of all these versions are very similar and the program is written in a way that requires very few modifications for switching between the versions.

Because of the intention to use this program for the Egothor search engine, the program reads directly gzipped input files using `popen` procedure if compiled in UNIX. As this is not possible in Windows, the Windows compilation works directly with unpacked files. The time consumption of the unzipping operation is discussed in chapter 5.2.5.

The graph loading part is then the same as in the *M* version. Because of the link file format we need to be able to randomly access any part of the graph during the loading and also during preprocessing part, so we load the whole graph into the memory and if it does not fit there the system will create a swap file for it. After the preprocessing we will have the file in a form that would enable us to process it sequentially. So we store it to disk. However, to enable caching of the file on systems with sufficient memory size, we used "`ST`" parameters in the `fopen` procedure that ensure optimization for sequential caching and not flushing the file to disk if not necessary.

In the PageRank distribution computation part we create a file for storing the old rank values. To save some I/O operations this file only contains the information about the rank of the crawled pages. In each iteration we then read one record from the link file and the rank value of this page from rank file and distribute this rank to pages linked from this page. The newly computed rank values are stored in the main memory. Again this is necessary because of the need for random access. The convergence test is the same as in the *M* version. However, to be able to compare the old and new rank we need to perform additional scan through the rank file.

After the computation converges, we create and array of size equal to the number of crawled pages, where each record contains the ID of a crawled page and its computed rank. This array is then sorted using the quick-sort [45] procedure. The resulting structure is than written to disk in the above-mentioned form.

5.2.5 Time consumption

All program versions were tested on three computers and two different systems *1*. Intel Pentium M 1.6 GHz, with 512 MB RAM, 60 GB 5400 rpm disk with 8 MB cache running Windows XP Professional, *2*. AMD Athlon 2500+, 512 MB RAM, 120 GB 7200 rpm disk with 8 MB cache, running Gentoo with kernel version 2.6.16. and *3*. 64-bit AMD Athlon 3000+, 1 GB RAM, 160 GB 7200 rpm disk with 16 MB cache, running both Windows XP and Gentoo with kernel version 2.6.19 compiled in 64-bit mode. The processors were running on 1.28 GHz, 1.84 GHz and 1980 GHz respectively. All computers had more than 200 MB free main memory for the computation.

The tests were done on an “mff.cuni.cz” test collection. The crawl of this collection was done in April 2006 and was focused on pages containing the string "mff.cuni.cz" in their URL. The index file of this collection contains 1 415 790 pages, out of which, however, only 779 780 pages have a record in the link file, i.e. were crawled. In the link file the average number of links of the crawled pages is approximately 26. This number incorporates also links to not crawled pages, duplicate links, zero links etc. After processing the link structure using the redirect file and erasing of worthless links the average number of links of a crawled pages decreases to 17.2.

To test the run times and memory consumption of the program procedures, the *M* version automatically prints out control texts. To display similar texts in the *D* versions it has to be run with parameter –p.

Time comparison of different versions of programs for computing PageRank (in seconds)								
Computer	Version	Graph Loading	Redirect Processing	Graph Processing	Graph Saving	Computing PageRank	Sorting and Saving rank	**Total**
1 - Win	M	13	<1	4	X	9	4	**30**
	D32	13	<1	4	16	183	5	**222**
	D32p7	13	<1	3	10	153	5	**188**
	D64	14	<1	4	17	192	7	**235**
	D64p7	14	<1	5	10	163	8	**201**
2 - UNIX	M	2	<1	3	X	7	2	**14**
	D32	2	<1	4	1	28	2	**38**
	D32p7	2	<1	4	4	28	2	**41**
	D64	3	<1	5	2	28	16	**55**
	D64p7	3	<1	5	4	28	16	**57**
3 – Win	D32	10	<1	3	4	46	3	**67**
	D32p7	10	<1	3	9	115	3	**140**
	D64	10	<1	3	4	52	4	**74**
	D64p7	10	<1	3	9	120	4	**148**
3 – UNIX 64 bit	D32	2	<1	2	2	17	1	**24**
	D32p7	2	<1	2	2	18	1	**25**
	D64	2	<1	2	2	17	1	**24**
	D64p7	2	<1	2	2	17	1	**24**

Table 6 Time comparison of procedures of different program versions

In this table we can notice that if the *M* version is run on the first computer than more time is spent loading the graph than computing the PageRank although the PageRank computation requires 17 iterations. This confirms the fact stated in chapter 3.4 that the most time expensive operations are the I/O operations. It also shows that when we have enough memory, the PageRank computation can be done in very reasonable time. It is important to note that the *D* versions executed on UNIX directly read the gzipped input files, which, however does not have any negative effect on the run time. The time spent unpacking the compressed file is balanced by reading less data from disk

From Table 6 we can also notice several other interesting facts. First, the use of the Pack7 format for storing the graph file presents an advantage only on computers with slower disks (in terms of rotations per minute) running Windows. If the program is run on UNIX the compression of the numbers brings no advantage because the files are not directly written to disk, but thanks to caching are only stored to the main memory. It might, however, be beneficial if the computation is run on larger datasets where the full caching would not be possible.

Other important fact that can be noticed is that the UNIX caching is much more effective than caching in Windows. In fact we observed that Windows does not do any caching at all even when there is enough memory for it. This results in much worse run times of the *D* versions when run in Windows.

Another important observation is that using the 64 bit instead of 32 bit numbers for storing the IDs does not result in any overhead when the program is run in the 64 bit system. In fact the *D64* version is even slightly quicker than *D32*.

5.2.6 Memory requirements

We have also measured the amount of dynamically allocated memory used by different versions of the program. The following table provides the results.

Memory consumption of procedures in different versions of programs for computing PageRank (in KB)								
System	Version	Graph Loading	Redirect Processing	Graph Processing	Graph Saving	Computing PageRank	Sorting and Saving rank	**Max**
32 bit	M	119 738	5 663	8	0	0	9 358 + *q*	129 096
	D32	94 158	5663	8	0	11 326	9 358 + *q*	99 821
	D64	175 477	11 326	16	0	11 326	12 476 + *q*	186 803
64 bit	D32	106 938	5 663	8	0	11 326	9 358 + *q*	112 601
	D64	188 277	11 326	16	0	11 326	12 476 + *q*	199 603

Table 7 Memory consuption comparison

This table, however, requires a few comments. In the *M* version the graph loading consumes more memory because we already allocate the structure for the PageRank computation that is we allocate two floating-point numbers for the PageRank computation for each page. In this version this memory is kept throughout the whole computation process. In contrast, the *D* versions only allocate memory for the adjacency list structure (see chapter 3.2), which is then preprocessed and stored to disk. For the rank computation and array for computing new ranks is allocated and the link structure and old ranks are read from disk. This means that if the link graph was preprocessed before the rank computation, the total memory requirements of the *D* versions would in

this case be 21.6 MB for the *D32* and 23.8 MB for the D64 versions (we assume that the memory for allocated for the PageRank computation is deallocated before running the quick sort procedure and that *q*, the memory needed in the quick sort procedure, is less than the memory needed for the rank computation).

In order to give some more detailed comments to figures in Table 7 we have to provide a few more details about the exact way of memory allocation and about the data types used.

In the graph-loading phase, we start with an array of initial size, in all programs set to hundred thousand, and load records about the graph from the input file. If we load an ID of a page that is higher than the current size of the array, we double the size of the array using the `realloc` function. So in our case the final array has size 1.6 million.

The structure for storing the link structure in the *M* version contains a 32-bit integer for storing the number of links on the current page, a 32-bit pointer to array of ID's of pages linked from this page and also contains two 64-bit double precision floating point numbers for storing rank. So each page needs at least 24 bytes of memory. Given the fact that we allocated an array of size 1.6 million for this structure gives us 38.4 MB memory. As already stated the average number of links on a crawled page in the input file is 26 and the number of crawled pages is 779 780, then when using 32-bit integer number for storing the IDs of pages in the neighbour list, we need approximately 81 MB of memory. When we sum these two numbers we get 119.4 MB, which is approximately the number present in Table 7.

On the other hand, the *D* versions do not allocate any memory for the numbers for computing PageRank in the load phase. So each page requires only 8 bytes of memory - 4 bytes for link count and 4 bytes for pointer to the neighbour list. So the size required for storing the array of pages is 8 times 1.6 million, which is 12.8 MB. For the *D32* version the memory requirements for the link structure stay the same so the total is 93.8 MB, which is again very close to the figure, presented in Table 7. For the *D64* version, the memory requirements for storing the neighbour lists are doubled. So the total consumed memory size is 174.2.

In the redirect-processing phase we need to create an array of size of the current collection containing page ID's. So the *M* and *D32* version need 1 415 790 times 4 bytes which is 5 663 KB and the *D64* requires double this size, which is 11 326 KB. This memory is, however, freed as soon as the redirects are processed.

The *D* versions then further require an array of double precision floating-point numbers of size equal to the size of the collection for the rank computation which is again 11 326 KB. The *M* version does not require any memory for the rank computation. For the sorting purposes we allocate a structure of size equal to the number of crawled pages, where we store the ID and the rank of each crawled page. This array is then used for sorting. The *M* and *D32* versions require 9 358 KB and the *D64* version requires 1.5 times as much, which is 12 476 KB for the array allocation. Further memory is also required for the quick sort procedure.

In the 64-bit system the additional memory is consumed by the use of 64 bit pointers.

5.2.7 Results of the PageRank computation

In this subchapter we provide a few comments to the PageRank distribution produced by our program.

Surprisingly, the page with the highest rank is not the root page – mff.cuni.cz, but the personal homepage of RNDr. Libor Forst (www.ms.mff.cuni.cz/~forst) containing only a few links and personal photos. The high rank is mainly caused by the fact that our collection contains multiple faculty dictionary pages which all contain a copyright logo with the link to this page. Also, the Google Toolbar PageRank [46] of this page is 6/10, which is still a very high ranking that confirms our results. This web page is a very good example that shows that the PageRank is completely independent of the page content.

The second best page in our ranking competition was the www.mff.cuni.cz page which has the Google Toolbar Rank 8/10 and the third highest ranked page was the homepage of the faculty magazine called M&M – mam.mff.cuni.cz with Google Toolbar Rank 5/10. It is important to note that the Google Toolbar Rank considers the links from the whole web, while our ranking only takes into account links within the mff.cuni.cz site.

The highest achieved rank was approximately 0.005 which confirms our assumptions from chapter 5.1.4.6. We have also tested the power-law distribution of the logarithmically scaled PageRank values and were able to observe that the collection roughly follows the power-law. We assume that the small deviations are mainly caused by the low size of our collection.

Distribution of PageRank values											
Log scale	0	1	2	3	4	5	6	7	8	9	10
Nr. of pages	558 817	181 688	27 942	7 053	2 745	1 096	329	76	26	7	1

Table 8 PageRank distribution

5.2.8 Our implementation – summary

In this chapter we have presented our implementation of program for computing the PageRank distribution and provided figures describing its run time and memory requirements. We have also pointed out some issues connected with the PageRank computation.

However, we believe that the *D64p7* program version is designed in such a way that would also enable its use on the web-scale. In such a case a computer with a huge main memory and three fast disks and a good preprocessing would be needed. The disks would be used for following purposes: swapping the main memory content; reading graph structure; reading and writing the PageRank values. The preprocessing would ensure that the main memory would be used "almost" sequentially so the total run-time of the program might be reasonable.

For further optimisation it might be interesting to test computing the PageRank in scale (0, .., *n*) and use Pack7 format for storing the values to disk as proposed in chapter 5.1.4.6.

For further research, the *M* version program might be found useful as it contains a lot of procedures for processing the input files. The *D* version contains detailed comments in the code and so can be used as a framework for different rank computations as for example computing the rank using the site approximation as proposed in chapter 5.1.5.2.

5.3 Query dependent web-graph based rankings

In chapter 5.1 we have described PageRank, in our opinion the most important utilization of graph theory in information retrieval. Another well-known approach utilizing the web-graph structure to provide web page relevancy estimation is the HITS algorithm proposed by Kleinberg in [47]. In this chapter we describe the idea of HITS and also a simple way how to compute it. However, we do not describe the algorithm in detail nor provide overview of possible optimisation, because as we explain in the following subchapter, although the algorithm is very well known, it is not widely used.

We also present a few alternative approaches that utilize the web-graph structure to provide a relevancy ranking. One of the presented approaches also analyses the use of the combination of the use of user-supplied feedback with the distance of the pages in the web-graph to provide a customized ranking.

5.3.1 HITS

At around the same time as Brin and Page presented PageRank, Jon Kleinberg proposed an alternative link-based ranking scheme called Hyperlink Induced Topic Search (HITS) [47]. The main difference between these two schemes is the different query dependency. While PageRank provides a query independent ranking, the HITS algorithm incorporates the query information into the link-based ranking.

The basic idea is to build a query-specific (*neighbourhood*) graph and perform the link analysis on this graph. In the ideal case, this graph would only contain pages relevant to the query and the link analysis helps us to find the most informative ones.

In his approach Kleinberg used *neighbourhood* graph as proposed by Carrere and Kazman in [48]. This graph is constructed in the following way. We create a *start set* of n (e.g. 200) vertices where the vertices represent the first n pages retrieved by a search engine when searching for results of query q. The *start set* is then augmented by its *neighbourhood*, which is the set of pages that either contain a link to a page in the *start set* or are linked by a page from this set. Since the number of pages linking to a page (like google.com) can be huge, the number of pages added to the *neighbourhood* graph because of linking to a page in the start set is limited by a constant i (e.g. 50). Similarly to the case of web-graph used in PageRank, each page in these sets is modeled by a node in the graph. However there is link from page a to page b if and only if a contains link to b and a and b are pages on different sites (hosts). The second condition tries to prevent the possibility of manipulation.

In his approach Kleinberg tries to determine pages with good content on the topic of the query called *authorities* and directory-like pages with many hyperlinks to pages on the topic called *hubs*. The algorithm then assumes that a page that links to many other pages is a good *hub* and a page that is linked by many pages is a good *authority*.

These two scores can be calculated by the following recursive algorithm. We first create the query-specific *neighbourhood* graph *N* and initialize *hub* scores of all pages to *1*.Then till *H* and *A* have not converged, calculate the new authority and hubness score for all $p \in N$.

$$A(p) = \sum_{q:q \to p} H(q)$$

$$H(p) = \sum_{q:p \to q} A(q)$$

Again elementary linear algebra ensures that both these vectors converge even though it does not provide an upper bound to the number of iterations. Empirically the vectors converge quickly again.

Several improvements of HITS were proposed recently. The CLEVER HITS [49] expands the *start* set up to two links away and weights links by the similarity between the query and the text surrounding the hyperlink (*anchor* text).

Bharat and Henzinger [50] proposed several improvements to HITS. The most important is derived from Topical PageRank. In order to reduce *topic drift* they calculate similarity of each document to the query and use this measure in the mutual reinforcement process so that the pages most relevant to the query have the most influence on the calculation.

Even though these changes improved the HITS algorithm, it is still not that widely used. As already premised, according to Monika Henzinger, the director of research in Google Inc, the HITS algorithm is currently not used in any commercial search product [51]. The main reasons for this are that the HITS algorithm is query-sensitive and the creation of the query-dependent graph has to be done for every single query. This requires significant computational capacities when taking into account that the major search engines process thousands of queries every second [26]. Important point also is that the results of this ranking algorithm are not much better comparing to PageRank when the query is related to a specific topic. In such cases the *topic drift* may still occur.

5.3.2 Flow based Rank

Recently Chitrapura and Kashyap [52] proposed a query dependent flow-based ranking. In their approach they use the network flows in the web-graph as a measure of page relevance. Even though their intuition is on the first glance totally different from PageRank, their work is closely related to topically biased PageRank algorithms (see chapter 5.1.6). In their model, the volume of flow indicates the relevancy of the pages to associated labels (topics) and can be used to compute both the query dependent and independent rankings.

In their model they use a flow that moves through one edge at a time. At the beginning they start with one vertex which is assigned all the flow. Then in each time point all vertices containing a flow with label *l* move this flow along the outgoing edges containing label *l* to vertices pointed to by these edges. If no such edge is present, the flow is lost. However when moving the flow a small portion of it is lost. All vertices containing label *l* get a small volume of flow belonging to label *l* in each time point. The labels of vertices (pages) are derived from the content relevance of the current page to the given labels. The edge (link) labels are derived from *anchor* text and the content of the page that they link to.

From another point of view this flow-based model is only a slightly modified different perspective to PageRank. However these modifications are shown to bring results better than the topic-sensitive PageRank. However the comparison to the Topical PageRank which is much more similar to this flow based approach was not done. Both Topical PageRank and the flow-based ranking were shown to provide better *fine-grained* ranking than the Topic-sensitive PageRank, however to the best of our knowledge no comparison of these two rankings was publicized.

Another important point is of course the time complexity of this approach. It depends mainly on the type of algorithm used in the computation. The approach presented in paper [52] used an algorithm similar to the Pre-flow push (Push-relabel) algorithm, which in fact uses very similar principles, and so no major modifications were needed. However no indication of the run-time of this algorithm was mentioned, so the web-scale usage of this approach is questionable.

5.3.3 Re-ranking pages using user supplied feedback

This approach is fundamentally different from the ranking schemes mentioned earlier, because it is intended for use in a different phase of the retrieval process. It

assumes that the user submitted a query however is not totally satisfied with the retrieved pages. So rather than using query refinement, Vassilvitskii and Brill [53] propose using user-supplied feedback to re-rank the initially retrieved set of documents.

This algorithm presents a novel approach to using the relevancy feedback. While previous studies used mostly term re-weighting schemes or tried to automatically refine the query, Vassilvitskii and Brill propose using web-graph distance as a measure of relative document relevancy.

Their approach is based on the hypothesis that relevant pages tend to contain links to other relevant pages while irrelevant pages are mostly linked from other irrelevant pages. The results of this algorithm showed that this hypothesis holds.

The intended scenario for this system is as follows. Upon receiving the list of retrieved pages, the user provides a feedback about the relevancy of *1* to *5* random pages from this set, where each of these pages is ranked on the scale *1* to *3*. In this scale *1* stands for relevant, *2* for neutral and *3* for irrelevant. The system then uses this information to re-rank the retrieved documents and provide a new list, which better matches user's needs. What is however important is that this approach is shown to be effective even when the user provides feedback to only one page in retrieved set.

The re-ranking algorithm is relatively simple. For all *rated* pages a set of pages with distance (both ways) less than or equal to *4* is discovered. To avoid exponential growth of this graph the maximum considered number of links was set to *35*. Ranking of all pages in these graphs is then refined by the ranking of the rated pages multiplied by inversed distance between these pages.

5.4 Other query dependant rankings

This chapter provides an overview of some graph-based ranking algorithms also suitable for traditional IR systems. Non of these algorithms explicitly uses the web-graph structure but all work with a graph-based framework.

5.4.1 FlowRank - Collaborative Ranking

A recently presented paper [54] describes a very unique approach to document ranking. It does not explicitly use the links of the web-graph structure, but works on a graph where vertices present the collaborating users, their queries and the documents, so is also suitable for traditional IR systems. This network is derived from search engine logs and its edge weights model the relationship between the relevant entities.

This approach tries to use gathered collective knowledge and use it to enhance outputs for current users. It can be interpreted as using some *Artificial Intelligence* techniques to help gather and use the information present in logs. In contrast to most other collaborative ranking algorithms this algorithm models the relationships between relevant queries, documents and collaborators by a graph. The paper presents a framework, where the final ranking is derived from the maximum flow in a graph.

The intuitive background of this approach comes from the study [55] which showed that user click-through (record of clicks on retrieved documents for a query) is an accurate reflection of user-related relevancy (preference) of the documents to the given query. This measure provides very valuable information even for similar queries and documents [56]. This information is utilized to re-rank pages based on the gathered knowledge.

The FlowRank algorithm is a query and user specific ranking that performs the creation of the above described network and computation of the network flow on this network in the query time. This might be considered a drawback of this technique nevertheless it presents a very prospective approach that was shown to provide very good results mainly for the naive queries. Although it is important to note that the use of such techniques might improve the competition advantage of the biggest web search engines because of availability of plenty of fresh log information which might enable these engines to adjust their rankings very quickly. Time will show if the advantages resulting from such approaches will be valuable enough for the search engines to offset the slow-down of the response and if some engines will encapsulate such an algorithm into their rankings.

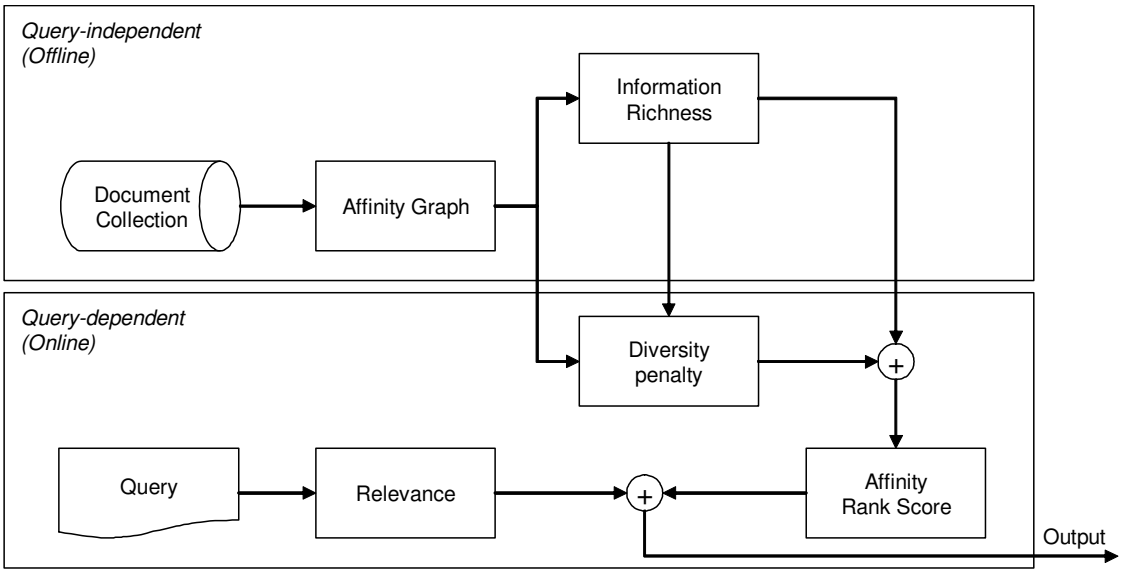
5.4.2 Ranking retrieved pages using Affinity Graph

The paper of Zhang et al. [57] presents a novel query dependent ranking scheme called Affinity Ranking suitable also for classic IR systems. In this approach the ranking is focused not only on optimizing the *precision* and *recall* (see chapter 2.1) of the systems but also aims to also optimize two novel metrics – *diversity* and *information richness*. The *diversity* of a set of documents indicates the variance of topics within the set. The *information richness* measures the coverage of a single document to its topic. Both of these metrics are calculated from a directed weighted graph called *Affinity Graph* (AG), which models the content similarity between all pairs of documents in a set.

The idea of the re-ranking algorithm is derived from the observation that most user queries are ambiguous [58] and the exact needs of the users are unknown. The result of such queries contain a vast of documents relevant to some popular topic that are however not very relevant to user's needs.

To prevent such situations Zhang et al. propose an algorithm that aims to improve the information richness and diversity within the top results. Increasing these metrics would increase the probabilities that some page that represents actual user's needs are present in the top rated retrieved documents and also that the document is one of the most informative to his actual needs.

The Affinity Ranking is a query-time re-ranking that does not use the information about the web-graph link structure at all. It only assumes that the retrieved results have already been rated using a full-text analyzing algorithms. The goal of this ranking is to re-rank the retrieved set of documents in order to achieve higher *diversity* and *information richness* values while keeping the *precision* and *recall* of the system at the same level.



Picture 6 Affinity Ranking Framework

For a set of documents $D = \{d_1, d_2, .., d_n\}$ the *diversity* $Div(D)$ denotes the number of different topics contained in D, i.e. the number of unique topics present in D. For a document d_i *information richness* $IR_D(d_i) \in \langle 0,1 \rangle$ denotes the informative degree of the document d_i with respect to the entire collection D. For each topic k, N_k denotes the number of documents in D associated with k and d_i^k denotes the i-th document associated with the k-th topic.

The average *information richness* of a set of documents D can be calculated as:

$$IR(D) = \frac{1}{Div(D)} \sum_{k=1}^{Div(D)} \frac{1}{N_k} \sum_{i=1}^{N_k} IR_D(d_i^k)$$

The *Affinity Graph* $AG=(V, E)$ is a directed weighted graph where V represents documents and E represents the similarity between documents. The similarity is not computed using the common *cosine* measure (see chapter 0) but with similar measure called *affinity* defined as:

$$Aff(d_i, d_j) = \frac{\vec{d}_i . \vec{d}_j}{\left\| \vec{d}_i \right\|}$$

To save some space we $Aff(d_i, d_j) = 0$ if $Aff(d_i, d_j) < Aff_t$ where (Aff_t is a threshold). To represent this graph we define a normalized adjacency matrix $M = (M_{i,j})$ of size n x n defined as:

$$M_{i,j} = \begin{cases} aff(d_i, dj) / \sum_{k=1}^{n} aff(d_i, d_k) & \text{if } \sum_{k=1}^{n} aff(d_i, d_k) > 0 \\ 0 \quad \text{otherwise} \end{cases}$$

On such a matrix we can define a model very similar to PageRank. Assume there is a *random reader* who selects a random document as the start of his reading journey and when looking for another document, he chooses one of the documents similar to the current document with probability D and chooses a random document with probability ($1-D$). The stationary distribution of this journey can be computed using any of the algorithms for computing PageRank described in chapter 4.

The computed distribution helps us to choose the most informative documents, however some of them can still be very similar. To increase the topic coverage of the top retrieved documents, Zhang et al. propose imposing different *diversity penalty* to the *information richness* score of each of the retrieved documents. The penalty can be calculated by a simple iterative greedy algorithm. The algorithm starts with the set of documents A returned by the full-text search. For each document we initialize $AR_i = IR(d_i)$ and sort A by AR in descending order. Then for $i = 1$ to q (where $q=|A|$ or q is the desired number of returned documents) it extracts the most *informative* document d_a from A, places it to the output (as i-th top ranked result) and for each document $d_j \in A$ with $M_{j,j} > 0$ it sets $AR_j = AR_j - M_{j,i}.AR_i$ and resorts the set A using the new AR values.

This procedure ensures that for each topic relevant to the query only the most informative document becomes distinctive in the ranking process.

To combine the *Affinity Ranking* with ranking with *full-text* ranking we can use a simple linear combination:

$$Score(q, d_i) = a.Rank_{Sim(q,\, d_i)} + b.Rank_{AR_i}$$

where a and b are tunable constants. We think that setting of a and b derived from the length of the query could improve this ranking further. We assume that the shortest queries are more likely ambiguous than the longer ones and so the variety of topics should be more desirable for the shorter queries.

The *Affinity Ranking* is a ranking that does not use the link structure at all and is fully applicable to basic IR systems. However it would be possible to integrate this technique in the web IR systems. The main problem of this integration would be the creation of the *Affinity Graph* and computation of the *Affinity Ranking*. Nevertheless the construction of the *AG* could be done during the crawl process, when for each new crawled page, we would calculate the *affinity* between this page and all the pages that have already been crawled. The ranking computation would require similar time as the PageRank computation (even the site approximation might be possible) so we would only need to have more available resources to be able to take advantage of the *Affinity Ranking* in any web IR system.

It is very probable that Google already uses a similar kind of ranking. We have tested this assumption by a few experiments. E.g. when we searched for "puma" on the Czech version of Google, the results covered three topics incorporated with "puma" – the sport brand, the animal and the car brand (Ford Puma). However when we checked the first page relevant to "Ford Puma" the Google toolbar shows *0* as its PageRank while several pages having considerably higher PageRank and containing similar occurrences of word puma (e.g. pumastore.com) can be found below this site in the results. It is probable that this can be caused by various other factors like the localization of the "Ford Puma" page. Nevertheless similar result structure was observed experimenting with a few other queries (e.g. jaguar). As already stated the principles and algorithms of Google are confidential, so we can only guess which techniques does it use and how. However when we take into account that Google claims considering more than hundred factors when computing results to a query, it is highly probable that the diversity of the top results also plays a role in the procedure.

5.4.3 Conceptual graph

A conceptual graph is a good example of using the advantages of graph theory and processing of text based on semantics to improve the effectiveness of IR. This concept is intended for use in classical information systems and improves the performance of these systems by incorporating limited semantic knowledge into an improved representation of documents. The semantic analysis is not widely used in IR systems because of the complexity of such analysis. This approach tries to bring benefits of the semantic analysis into IR systems by modeling the documents in a conceptual graph that contains basic semantic information about the documents. To avoid the high time complexity, the graph is constructed in the query time using only part of the top results. The semantic knowledge is then used to improve the ranking of this set.

5.4.4 HITS and PageRank without hyperlinks

The major success of the web-graph based rankings, motivated researches that tried to use similar algorithms also on documents lacking the hyper-link structure. In [59] Kurland and Lee utilized a PageRank-like algorithm to re-rank the retrieved set of documents. The graph for the rank computation was constructed using language models induced from the document set. In their approach the documents are represented by vertices and edges represent similarity between the documents. The PageRank-like distribution is then computed on a graph that contains first *i* (e.g. 200) documents retrieved by text-based search engine.

As this approach was not showed to be effective, Kurland and Lee proposed incorporating the cluster information into the ranking process [60]. The intuition of their approach is that (1) the documents within the clusters that do best represent user's information needs, are likely to be relevant and that (2) the most representative clusters should be those that contain many relevant documents. As this intuition is very similar to intuition behind the HITS algorithm (see chapter 5.3.1) they used this algorithm to compute the authority and hubness score on a bipartite graph composed of documents and clusters. The graph was again created in the query time from the first *i* results. They showed, the cluster-document graphs provide a very efficient framework for the graph-based ranking algorithms.

6. Finding communities

A lot of current search engines also provide an alternative to searching by queries, they enable us to input a web page and want the engines to find similar pages to the input page.

To achieve the desired result, the search engines often use the web-graph structure. The main idea is that highly similar pages would very likely be close to each other in the web graph.

For this purpose it is possible to employ the HITS algorithm. In this case the *start set* is the set of input pages. We then just run the algorithm and sort the pages by *authority* score in descending order and output the first k pages that were not part of the input set (k is the desired number of output pages). However usage of HITS algorithm for this purpose is even less feasible than the query-sensitive ranking. The input set would often be relatively small so the *topic drift* occurs more probably.

The problem of finding communities is similar to the above mentioned approach. The goal in finding communities is to find pages that are thematically highly related to the given input set of pages. It has been shown such pages are often highly connected and so this task can be efficiently solved using the web-graph structure.

6.1 Flake's max-flow based algorithm

In an approach proposed by Flake et al. in [61] they assume that the input is a set of *seed* pages and the result of their algorithm is the approximate community composed of the seed pages and some of their neighbour pages. Their method uses a maximum flow, minimum s-t cut (see chapter 3.7) based approach to find a dense subgraph containing the seed pages and regards it as the approximate community.

In their approach they use a graph $G=(V, E)$ such that the set V is composed of sets S,P and Q, where S is the set of *seed* (input) pages; P is the set of all pages that contain a link to or are linked from any page in S; and Q is the set of all pages linked from pages in P. The set of edges E contains all links from pages in S to pages in $S \cup P$ and all the links from pages in P to pages in $S \cup P \cup Q$.

We then add virtual source s and sink t to G. For each $x \in S$ we add a virtual edge (s,x) to E' and for each $x \in V$ we add a virtual edge (x,t) to E. Let $G'=(V',E')$ be the

resulting graph, formally $V' = V \cup \{s\} \cup \{t\}$ and $E' = E \cup \{(s,x) \mid x \in S\} \cup \{(x,t) \mid x \in V\}$.

The capacity $c(e)$ of each edge $e \in E'$ is set as follows: $c(e) = \infty$ for $e = (s,x)$; $c(e) = 1$ for $e = (x,t)$; and $c(e) = k$ for each $e \in E$, where $k = |S|$ i.e. k equals to the number of seed pages.

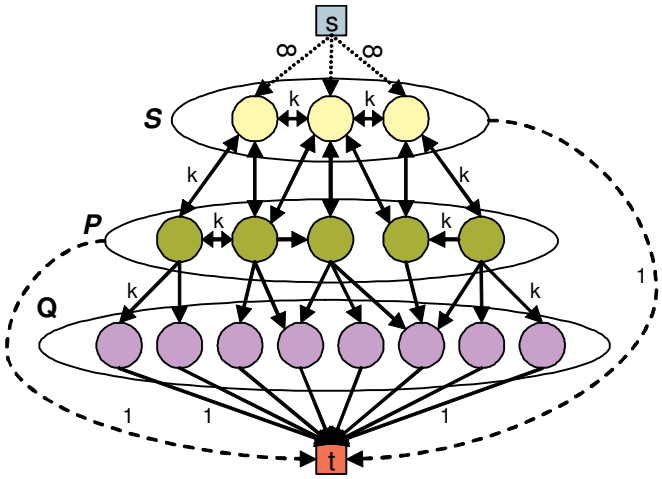
The result of the *construction* phase is the graph $G'=(V',E')$ and capacity function c.

The algorithm then works as follows. For $i=1$ to l (where $l = 4$ in the [61]) we do the following procedures.

1. We compute the *minimum s-t cut* C of network G_i' (where $G_0' = G'$) such that C is nearest to s. Let X denote the connected component of $G_i' \backslash C$ containing s.

2. if $i<l$, then we find a vertex u of maximum degree in $X \setminus (S_i \cup \{s\})$. I.e. vertex u with maximum $d(u)$ in X such that u is not one of the seed vertices, where $d(u)$ denotes the sum of the in and out degree. We set $S_{i+1} = S_i + \{u\}$, *construct* G'_{i+1} and increment i.

3. if $i=l$ we output $X \setminus \{s\}$as an approximate community.



Picture 7 Example of *G'* constructed for finding communities

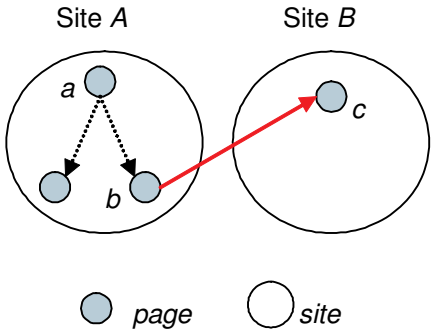
In their experiments Flake et al. pointed a few problems and also easy solutions to these problems. First to avoid *topic drift* (adding pages with different topics) they adopted the assumption that if a page has more than q (e.g. 50) links they regard it as a portal and totally ignore it. Second, to avoid only adding pages reachable from S_0 to the final community they regard all edges between vertices in P and S a bi-directed even if there exist only an edge in one direction. Third, to avoid adding all pages of all sites (hosts) they ignore all intra-site (intra-host) links.

This approach was found very effective for finding communities even though it does not consider the content of the pages at all. However, several papers noted several problems and improvements to this algorithm.

6.2 Improvements

Asano et al. [62] presented a site-based approach that tries to improve the Flake's algorithm by pointing out several problems and providing solutions to these problems.

They consider the *ignored link* problem to be the major weakness of the algorithm proposed by Flake and demonstrate how this can be avoided by using the site as unit of information. The following picture illustrates the *ignored link problem* as defined in [62].



Picture 8 Ignored link problem illustration

This picture demonstrates that ignoring the intra-site links can lead to loosing valuable inter-site links because of inability to get to different page in a site that contains this link. In the picture above we can see that if we get to page *a* in site *A* by ignoring the intra-site links in *A* we loose the information about the link from page *b* in *A* to page *c* in *B*.

Asano et al. showed that the *ignored link* problem occurs relatively often and propose solving this problem by using the site as the element in the graph. In their approach they propose constructing the inter-site graph $G'=(V',E')$ in a similar way as the host-graph in chapter 5.1.5.1 I.e. the vertices represent hosts (sites) and there is an edge between site S_1 and S_2 if there exists a page $a \in S_1$ containing a link to page $b \in S_2$. Intuitively this framework disposes of the *ignored link* problem.

They also showed that a major loss of precision is caused by the *capacity* problem. They showed that because of the uniform edge capacities setting to |S| the following two situations may occur if the graph G' does not contain "too many" links between vertices within sets P and Q. First a vertex $v \in Q$ that only has only one

incoming edge (u,v) might become a member of the community if $u \in X \cap P$. Second if a vertex v has incoming edges from all vertices in S, v would not become a member of the community when $outd(v) > |S|$. These two cases are intuitively wrong, because a vertex with only one link from the community is highly unlikely to be a good member of the community, while a vertex connected to all pages in the seed set would be expected to belong to the community.

To solve such special cases Asano et al. tested setting the parameter k (see algorithm definition in chapter 6.1) to various values. The best results were achieved when k was set in the range $\{10, .., 15\}$.

Such an assignment can remind us of an idea of not setting the edge capacities constant, but choosing another criteria. This idea was explored in detail by Imafuji et al. in [63]. As the result of their study their proposed improving Flake's algorithm by using *HITS* (see chapter 5.3.1) score based edge capacities. In their approach they proposed to set edge capacity c for edge (u,v) as follows:

$$c(u,v) = \frac{h(u) + a(v)}{2^{dist(u)}} + 1$$

Where h and a mean the hub and authority score of a vertex respectively and $dist(u)$ represents the distance of page u from any of the seed pages. Their results showed a considerable improvement in the precision of the community pages assignment.

To the best of our knowledge, no study has explored incorporating the site-based approach with the HITS score based capacity assignment, which might surely be an interesting study.

6.3 Summary

In the subchapters above we presented algorithms for finding web communities. All of the experiments above only focused on improving the precision of the resulting communities. However to be able to facilitate these algorithms we need to be able to compute the results in a reasonable time. The run-time of these algorithms is influenced by two main factors, the construction of graph G' and the computation of the minimum s-t cut. The most of time needed for the graph construction phase is spent loading the information from the disk (if we assume web-scale use). It might be interesting to find out how different storage techniques would influence the run-time however we did not concentrate on this part.

The second part of the algorithm is influenced by the choice of algorithm for finding the maximum-flow and the minimum cut. Even though it might not look to be important to try to optimize this part of computation because of the graph size it might not be absolutely true. Given the fact that average number of out-links on a page is *10-20* gives us approximately 30 edges to other pages for each page. So if we consider size of seed approximately 30 we need to compute minimum s-t cut on a dense graph with about *5-20* thousand vertices (when considering..). This is significantly less than in the case of PageRank computation however if we have to take into account that algorithms for finding communities are also intended for instant use then each saved mili-second is valuable. The size of this graph enables us to store it into the main memory, so we should only concentrate on minimizing the number of operations.

For such graph sizes the most common implementation – the Ford-Fulkerson algorithm would probably not be the best performing because these graphs contain lots of paths between source and sink and improving one at a time would require a lot of run-time. Hence, we would emphasize using the Pre-flow Push (push re-label) algorithm which would very probably work very well in this kind of graph. We assume that it would be able to compute the maximum flow in very few iteration because of the small distance between the source (or sink) and all other vertices. The shortest distance between source and sink is *3* and so the algorithm would need at most 6 iterations till returning the flow value. In each iteration we would run a scan through (at most) all edges so the resulting time complexity of this algorithm would be less than 6*m*, *which* is a fairly good result.

7. Other examples of graph theory in IR

7.1 Other uses of Web-graph Analysis

As we showed the analysis of the link structure of the web can be a valuable source of information. In the last two chapters we have shown using it to provide us with document ranking and for helping us to find communities. However there are several other applications.

7.1.1 Web Crawling

The web crawler (also known as Web spider or Web robot) is a program that browses the web in a predefined manner. The crawlers are mostly used to provide a snapshot of the web, which can be used for generating index. The basic crawlers are usually based on the BFS algorithm. Common crawlers generate three files, the web page archive, web-page index file and the web-graph file. They can work in the following way.

The algorithm gets a set of start pages Q as the input. It then repeats the following iteration until an end condition is reached.

Choose a page p from Q, download it, create its page index record and store it to the archive. Check if pages linked from p have already been indexed. If not add these pages to Q. Remove p from Q.

As the crawlers are an important part of all search engines they were in focus of various studies. Some of these studies [64] tried to improve the crawlers in a way that would ensure that they crawl most of the important pages while avoiding the irrelevant ones. Some [65] aimed to create a topic-driven crawler that would be able to download web pages relevant to a given topic and thus provide complex information about the topic.

7.1.2 Mirrored hosts

Two hosts H_1 and H_2 are *mirrors* if and only if for every document on H_1 there is a highly similar document on H_2. Mirrors have very similar hyperlink structure both within and out of the host. Mirrors waste space in the index data structure and can lead to duplicate results. Combining the IP address analysis, URL analysis, the text analysis and the link structure analysis can help us to detect many near-mirrors [66]. It is however important to note that even though the link structure can provide us with a good heuristic in finding mirrors the present state-of-the-art algorithms for near-

duplicate web-page finding do not use it at all. The reason for this is that the problem can be explicitly solved by text analyzing tools and computing the heuristic provided by exploiting the link structure consumes more time than it saves.

7.1.3 Web sampling

Web-graph analysis can also be useful tool for computing statistics about groups of Web pages, like their average length, average number of links, the percentage of web pages that are in Slovak etc. PageRank based random walks can provide us with almost uniformly distributed samples of the web. These almost random samples can then be used to measure various properties of the web pages but also to compare the number of the pages indexed by various search engines. More about the use of web sampling can be found in [67]

7.1.4 Geographical scope

Whether a page is globally interesting, or is only of interest of a small region or nation might be interesting information about the page content (e.g. the official site of the Czech Hydro meteorological Institute chmu.cz is mostly interesting only for Czech citizens). This information can be very useful for search engines providing personalized rankings. The information about the geographical position of the user might be detected from the regional settings of his browser, or it can be supposed that if user is using regional version of the search engine rather than the global one, he is located in that region (E.g. use of google.cz instead of google.com implies being located in the Czech Republic).

The information about the range of interest of a page can again be computed using the hyperlink analysis. The intuition is that local pages are mostly linked from pages from the same region, while globally interesting pages are assumed to have a regionally uniform distribution of pages linking to them. More details about this subject can be found in [68]

7.2 Other graphs in IR

7.2.1 Clustering

A very important part of the information retrieval starts a long time before a user queries the system. We first need to store the documents. There are lots of possibilities in organizing the stored documents. As already mentioned in Chapter 3, the most expensive (measured in time spent) operations are the I/O operations. A significant part

of an I/O operation is the disks seek time. So when storing the documents we need to think of a structure that would minimize the average number of seeks for a query. This problem is often handled by the document (text) clustering algorithms. Most of these algorithms are graph-based and use the maximum flow – minimum cut theory for clustering the documents in such a way, that the documents in one cluster are highly similar, so there is a high probability that when one document is relevant to a given query, all the others would be. The common definition of a good clustering requires that nodes assigned to the same cluster should be highly similar while points in different clusters should be highly dissimilar.

A simple example of a clustering approach is the Spectral Clustering algorithm. It uses a similarity graph where documents are modeled by vertices and edges (edge weights) represent the similarity between the documents connected by this edge. The bi-partitioning (dividing the document into two disjoint sets) problem can be solved by finding a minimum cut in this graph, however the eigenvector-based approach is more commonly used. The goal of k-way clustering algorithms is to partition the given documents to k clusters. One way how to achieve this is using the bi-partitioning algorithms repeatedly however the eigenvector based approaches are used more commonly, because they are empirically performing better.

The eigenvector-based algorithm for solving the k-way spectral clustering problem can be very briefly described to work as follows. It first builds a reduced space from multiple eigenvectors of the adjacency matrix of the similarity graph. Then it uses the 3^{rd} eigenvector to further partition the clustering output produced using the 2^{nd} eigenvector.

Lots of studies have focused on solving the clustering problem, however, we do not concentrate on the clustering in this paper any further.

7.2.2 Graphs in document classification

Another interesting example that uses graph theory to improve some functionality of the IR systems is document classification. The goal of the text document classification algorithms is to assign (predefined) labels to texts and so to create directories similar to ODP [40] or Yahoo! directory [41].

There are two main types of classification. In one we do not have any predefined labels and only know the approximate number of categories we want to create. In such

case the clustering and communities finding algorithms (see chapters 7.2.1 and 6 respectively) might be utilized.

However, the more common case is that we are given a category structure and want to assign documents to these categories. This task is usually solved by text-based algorithms for document classification that use some kind of artificial intelligence as e.g. [43]. In [69] Angelova and Weikum proposed enhancing these algorithms by using additional information about the documents gathered from neighbours of the documents. In the hyperlink environment this is a very suitable approach because the neighbours can easily be found in the web-graph. Information about the neighbour documents and the *anchor* texts surrounding the links can provide valuable additional information for the classification.

As the human made directories are very difficult to create and maintain the demand for the automated tools for this task is obvious. The automatically created directories would probably not achieve as high quality as the human generated ones in the next few years, but are much cheaper to create and much easier to enhance and so we think that the text classification algorithms will surely find a broad utilization in the upcoming years.

8. Conclusion

The goal of this thesis was to explore and provide an overview of application of graph theory in information retrieval, choose an interesting application, explore it in detail and provide a test implementation.

Chapters 4 to 7 provide an exhaustive overview of graph theory usage in information retrieval. To the best of our knowledge this is a unique overview and was not presented in any previous paper.

For the detailed inquiry we have chosen the graph-based ranking algorithms as they aim to improve the most critical property of the IR systems - their precision. In this part we have presented various perspectives to computation of the best known graph-based ranking - PageRank. We have also described several approaches to both optimising the computation, and improving the precision of the ranking. In this chapter we have also described our implementation of algorithms for computing the PageRank distribution and provided facts and comments about the run time and memory consumption of different versions of the algorithm.

Further we have also presented rankings that are based on a graph structure fully independent of the web link structure and so can also be used in traditional IR systems. We have also provided ideas, how these rankings can be incorporated into web IR systems and gave comments on their realistic usage.

Another contribution of this thesis is that it provides several subjects for further research. E.g. a very interesting study (described in chapter 5.1.6.2) might be to try to compute the Topical PageRank described in chapter 5.1.6.2 using the BlockRank algorithm described in chapter 5.1.5.2. Another interesting idea for further research might be to implement several other PageRank algorithms described in chapters 5.1.4 and 5.1.5 and compare their performance to our implementation. Chapter 5.2 also contains several concepts that might further improve the efficiency of the presented implementation.

9. References

[1] L.R. Ford, Jr. and D.R. Fulkerson, "Maximal Flow Through a Network". *Canadian Journal of Mathematics*, 8:399 404, 1956.

[2] Google – Searching. *Google*. [online], 2007
WWW: http://www.google.com

[3] Yahoo search – Searching. *Yahoo*. [online], 2007
WWW: http://search.yahoo.com

[4] R. Baeza-Yates, B. Ribeiro-Neto, "Modern Information Retrieval", Addison Welsey, 1999

[5] M. Kopecky, "Information Retrieval Systems", Slides and Lecture Notes, 2005
WWW: http://www.ms.mff.cuni.cz/~kopecky/vyuka/dis

[6] L.Galambos, "Web data mining", Slides and Lecture Notes, 2006
WWW: http://kocour.ms.mff.cuni.cz/~galambos/swi107/slides.pdf

[7] Wikipedia the free encyclopedia. *Wikipedia* [online] 2007
WWW: http://www.wikipedia.org

[8] H.W. Lang, "Asymptotic Complexity". Retrieved 12 March 2007
WWW: http://www.inf.fh-flensburg.de/lang/algorithmen/asympen.htm

[9] Paul E. Black, "greedy algorithm", in Dictionary of Algorithms and Data Structures, 2005. Retrieved March 2007
WWW: http://www.nist.gov/dads/HTML/greedyalgo.html

[10] M. L. Fredman, R. E.Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms". *Journal of the ACM* 34(3), 596-615, 1987

[11] J. Cibulka, T. Dzetkulic et. al., "Peklo project". Retrieved March 2007
WWW:http://sourceforge.net/projects/peklo/

[12] R. Mihalcea, "Graph-based Algorithms for Information Retrieval and Natural Language processing". *Tutorials on RANLP 2005 Conference*, 2004

[13] S. C. Chekuri, A. V. Goldberg, D. R. Karger, M. S. Levine, C. Stein, "Experimental Study of Minimum Cut Algorithms". In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, 1997

[14] L. Galambos, "Egothor" – Searching [online] 2007
WWW: http://www.egothor.org

[15] Y.Y. Chen, Q. Gan. T. Suel, "I/O efficient techniques. for computing *PageRank"*, *In Proceedings of the 11. th. International Conference on Information and. Knowledge Management*, 2002

[16] D. Austin, "How Google Finds Your Needle in the Web's Haystack". Retrieved March 2007
WWW: http://www.ams.org/featurecolumn/archive/pagerank.html

[17] S. Brin and L. Page, "The Anatomy of a Large- Scale Hypertextual Web Search Engine;", In *Proceedings of the 7th International WWW Conference*, 1998
WWW: http://www-db.stanford.edu/pub/papers/google.pdf

[18] D. Vise, M. Malseed , "The Google Story", *37 ISBN 0-553-80457-X*. 2005

[19] Google, Inc. "Google information for webmasters". Retrieved March 2007
WWW: http://www.google.com/webmasters/4.html

[20] Wikipedia contributors, "Markov chain", *Wikipedia*, *The Free Encyclopedia*. Retrieved January 2007
WWW: http://en.wikipedia.org/wiki/Markov_Chain

[21] D. L. Isaacson and R. W. Madsen. "Markov Chains: Theory and Applications", *chapter IV*, *pages 126–127*. *John Wiley and Sons*, *Inc*., New York, 1976.

[22] J. H.Wilkinson, "The Algebraic Eigenvalue Problem", Oxford University Press, 1965

[23] K. P. Chitrapura, "Node ranking in Labeled Directed Graphs", In *Proceedings of the 13th ACM CIKM*, 2004

[24] I. Rogers, "The Google PageRank Algorithm and How it works", IPR Computing Ltd. 2005

[25] Microsoft Inc – Searching, "Live search", [online] 2007
WWW:http://www.live.com

[26] N. Blachman and J. Peek, "The Google Guide". Retrieved April 2007
WWW: http://www.GoogleGuide.com

[27] A. Arvind, Ch. Junghoo, G. M. Hector, P. Andreas, R. Sriram, "Searching the Web", *ACM Transactions on Internet Technology*, *Vol1*, *No. 1*, 2001

[28] T.H. Haveliwala, "Efficient computation of pagerank", *Technical report*, *Stanford University*, 1999
WWW: http://dbpubs.stanford.edu:8090/pub/1999-31.

[29] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, J. Wiener, A. Borodin, G. O. Roberts, J. S. Rosenthal, P. Tsaparas. "Graph structure of the web". In *Proceedings of the ninth international conference on World Wide Web*, pages 309–320. Foretec Seminars, Inc., 2000.

[30] J. Leskovec, J. Kleinberg, C. Faloutsos "Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations", In *Proceedings of the KDD conference*, 2005.

[31] J. Kleinberg, S. Lawrence, "The structure of the Web",*Science's compass VOL 294*, 30 November 2001

[32] S. D. Kamvar, T. H. Haveliwala, C. D. Manning, and G. H. Golub, "Exploiting the block structure of the Web for computing PageRank", Technical report, Stanford University, 2003.

[33] C. C. Aggarwal, "Intelligent Crawling on the World Wide Web with Arbitrary. Predicates", *IBM T. J. Watson Research. Center*. 2003

[34] G. Pandurangan, P. Raghavan and E. Upfal, "Using PageRank to characterize Web structure", In *Proc. of 6. th. COCOON*, *vol. 2387*, 2002

[35] A. Z. Broder and T.J. Watson, "Efficient PageRank Approximation via Graph Aggregation", 2004

[36] Netcraft, Internet services and statistics company. [online] 2007
WWW: http://www.netcraft.com

[37] M. R. Henzinger, R. Motwani, and C. Silverstein, "Challenges in web search engines", *SIGIR Forum*, *36*(*2*):*11--22*, 2002

[38] Bao Guang Feng, Tie-Yan Liu, et al, "AggregateRank: Bringing Order to Web Sites", In *Proceeding of SIGIR*, 2006.

[39] T.H.Haveliwala, "Topic-sensitive PageRank". In *Proceedings of the Eleventh International World Wide Web Conference*, Honolulu, Hawaii, 2002

[40] R. Skrenta, B. Truel. "Open Directory Project", [online] 2007
WWW: http://www.dmoz.org

[41] B.D. Davison, L. Nie, X. Qi, "Topical Link Analysis for Web Search", In *Proceedings of the ACM SIGIR conference*, 2006

[42] Yahoo Inc., "The Yahoo dictionary".
WWW: http://dir.yahoo.com/

[43] X. Zhu; X. Yin, "A new textual/non-textual classifier for document skew correction Pattern Recognition", In *Proceedings of the 16th International SIGIR Conference*, Volume 1, Issue , 2002

[44] Leapteg blog, "Google the Spy?"
WWW: http://blog.leaptag.com/2007/02/google_the_spy.html

[45] Wikipedia contributors, "Quicksort," *Wikipedia*, *The Free Encyclopedia*, Retrieved February 2007.
WWW: http://en.wikipedia.org/wiki/Quicksort

[46] Google, Inc., "Google Toolbar", *Take the power of Google with you anywhere on the Web*.
WWW: http:// toolbar.google.com

[47] M. J. Kleinberg. "Authoritative Sources in a Hyperlinked Environment", In *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms*, 1998

[48] J. Carroere and R. Kazman "Webquery: Searching and Visualizing the Web through connectivity", *Computer Networks and ISDN Systems*, Volume 29, 1997.

[49] S. Chakrabarti, B. Dom, P. Raghavan, S. Rajagopalan, D. Gibson, J. Kleinberg, "Automatic resource compilation by analyzing hyperlink structure and associated text", In *Proceedings of World-Wide Web '98* (*WWW7*), 1998

[50] K. Bharat and M.R. Henzinger, "Improved algorithms for topic distillation in a hyperlinked environment", In *Proceedings of SIGIR'98*, *pages 104--111*, 1998.

[51] M. Henzinger, "Hyperlink Analysis for the Web", volume 23, In *Proceedings of the IEEE 2000 Conference*, 2000

[52] K. T. Chitrapura, S. R. Kashyap, "Node Ranking in Labeled Directed Graphs", In *Proceeding of the SIKD 2004 Conference*, 2004

[53] E. Brill, S. Vassilvitskii, "Using Web-Graph Distance for Relevance Feedback in Web Search", In *Proc. of Annual ACM Conference on Research and Development in Information Retrieval* (*SIGIR*) *2006*, pp. 147-153, 2006

[54] S. Cucerzan, C. L. Giles, Z. Zhuang, "Network Flow for Collaborative Ranking", In *Proceeding of the PKDD Conference* 2006

[55] T. Joachims, L. Granka, B. Pan, H. Hembrooke, G. Gay, "Accurately Interpreting Clickthrough Data as Implicit Feedback", In *Proc. of Annual ACM Conference on Research and Development in Information Retrieval* (*SIGIR*) *2005*, pp. 154-161, 2005.

[56] J. Wen, J. Nie and H. Zhang, "Clustering user queries of a search engine", *In Proc. of the 10th International World Wide Web Conference*, pp. 162-168. 2001

[57] B. Zhang B et al., "Improving Web Search Results Using Affinity Graph", In *Proceedings of Annual ACM Conference on Research and Development in Information Retrieval* (*SIGIR*) *2005*, pp. 504-511, 2005.

[58] W. B. Croft, S. Cronen-Townsend, V. Larvenko, "Relevance feedback and personalization: A language modeling perspective", In *Proceedings of DELOS Network of Excellence Workshop*, 2001

[59] O. Kurland and L. Lee, "PageRank without hyperlinks: Structural re-ranking using links induced by language models", In *Proceedings of SIGIR 2005*, pages 306-313, 2005

[60] O. Kurland and L. Lee, "Respect my authority! HITS without hyperlinks, Utilizing Cluster-Based language models", In *Proceedings of SIGIR 2006*, pages 83-90, 2006

[61] G. W. Flake, S. Lawrence and C. L. Giles, "Efficient identification of Web communities", In *Proceeding of the 6th SIGKDD KDD2000*, pages 150-160, 2000

[62] Y. Asano, T. Nishizeki, M. Toyoda, M. Kitsuregawa, "Mining Communities on the Web Using a Max-Flow and a Site-Oriented Framework", In *Proceedings of the 6th International Conference on Web InformationRetrieval*, 2006

[63] N. Imafuji and M. Kitsuregawa, "Finding a Web Community by Maximum Flow Algorithm with HITS Score Based Capacity", In *Proceedings of the 8th International Conference on Database Systems for Advanced Applications*, *Proc of DASFAA 2003*, pp.101-106, 2003

[64] M. Diligenti, F.M. Coetzee, S.Lawrence, C.L.Giles, M.Gori, "Focused Crawling Using Context Crawling", In *Proceeding of the 26th VLDB Conference*, 2000 WWW: http://clgiles.ist.psu.edu/papers/VLDB-2000-focused-crawling.pdf

[65] F. Menczer, G. Pant, P. Srinivasan, M. Ruiz, "Evaluating Topic-Driven Web Crawlers", In *Proceeding of the 24th SIGIR Conference*, 2001

[66] K. Bharat, A. Broder, J. Dean, and M. R. Henzinger, "A comparison of Techniques to Find Mirrored Hosts on the WWW", *JASIS* (*Journal of the American Society for Information Science*) 51, 2000

[67] Henzinger M. et al., On Near-Uniform URL Sampling. *Proceedings of the Ninthe International World Wide Web Conference*. Elsevier Science, Amsterdam 2000

[68] O. Buyukkokten et al., "Exploiting Geographical Location Information of Web Pages". In *Proccedings of the ACM SIGMOD Workshop on the Web and Databases* 1998.

[69] R. Angelova, G. Weikum, "Graph-based Text Classification: :Learn from your neighbours", In *Proceedings of Annual ACM Conference on Research and Development in Information Retrieval* (*SIGIR*) *2006*, pp. 485-492, 2006