Facoltà di Lettere e Filosofia
Corso di Laurea Specialistica in
Comunicazione d’impresa e pubblica

Tesi di Laurea

in

Processi decisionali e gestione della conoscenza

An information guided spidering:
a domain specific case study

Relatore
Prof. Roberto Cordeschi

Candidata
Rossella Preziosi
matr. 0320400026

Correlatore
Prof. Ernesto D’Avanzo

Correlatore
Prof. Tsvi Kuflik

Anno accademico 2007-2008
Contents

1. Motivation, introduction: the problem, the research question, research methods and tools .................................................................3
   1.1. Introduction........................................................................................................3
   1.2. The research question.....................................................................................6
   1.3 The research methods and tools......................................................................6

2. Relevant literature survey on focused crawler...........................................9
   2.1 Search engines and web crawlers...................................................................9
   2.2 Crawler infrastructure..................................................................................11
      2.2.1 Inputs.........................................................................................................12
      2.2.2 Frontier.....................................................................................................13
      2.2.3 History and Page Repository.................................................................13
      2.2.4 Networking Layer .................................................................................14
      2.2.5 Parsing and Extraction Layer..............................................................15
      2.2.6 Representation Layer.............................................................................16
      2.2.7 Intelligence Layer....................................................................................17
   2.3 Crawling algorithms......................................................................................18
      2.3.1 Naive Best-First Crawler....................................................................18
      2.3.2 Breadth-First.........................................................................................18
      2.3.3 Best First.................................................................................................19
      2.3.4 Fish Search and SharkSearch...............................................................20
      2.3.5 Focused Crawler....................................................................................21
      2.3.6 Context Focused Crawler....................................................................22
      2.3.7 InfoSpiders.............................................................................................24
      2.3.8 PageRank and its usage to guide crawling..........................................28
   2.4 Evaluation of Crawlers.................................................................................32
   2.5 Algorithms application: Myspiders.............................................................33
   2.6 Algorithms application: BINGO!.................................................................35
2.6.1 Crawler ...............................................................36
2.6.2 Document Analyzer .............................................37
2.6.3 Feature Selection ....................................................38
2.6.4 Classifier .................................................................39
2.6.5 Link Analysis .............................................................40
2.6.6 Re-Training based on Archetypes ..............................40
2.7 Algorithms application: Arachnid ...............................42
2.8 Algorithms application: SSSpider ...............................44
2.9 OWL ........................................................................46
2.9.1 DL ....................................................................46
2.10 Protégé .................................................................49
2.10.1 OWL Plugin .........................................................52
2.11 RacerPro ...............................................................59

3. The energy domain case study ........................................58

3.1 Introduction ..............................................................58
3.2 Energy Security ontology description ............................60
3.3 Experimental setup ....................................................64
3.4 Experimental results ..................................................75
3.5 Ontology population with Arachnid and Bingo ...............79

4. Discussion .........................................................................84

Conclusion ........................................................................91

Bibliography
Chapter 1
Motivation, introduction: the problem, the research question, research methods and tools

1.1 Introduction

The management of large amounts of information and knowledge is of ever increasing importance today. With the ongoing ease of supplying information online, especially in corporate intranets and knowledge bases, finding the right information becomes an increasingly difficult task [Baeza and Castillo, 2006]. Today’s search tools perform rather poorly in the sense that information access is mostly based on keyword searching or even mere browsing of topic areas. This unfocused approach often leads to undesired results.

A Web search engine takes a user need, usually stated in the form of a few keywords, and returns a list of Web pages that can be considered relevant for the given keywords [Baeza and Castillo, 2007]. These Web pages are a short list of usually few hundred items selected from a vast repository of thousands of millions of pages.

The “easy” part of Web search is to find which documents in the Web are related to a given query, because most queries are very broad, and there are thousands of pages relevant to most basic concepts [Baeza and Castillo, 2006]. The hard part of Web search is to rank those documents by relevance and select the, say, top 10, to show the first result page to the user [Baeza and Castillo, 2006].

The enormous growth of the World Wide Web has made it important to develop document discovery mechanisms based on intelligent techniques such as “intelligent” crawling. In its classical sense a crawler is a program that retrieves WebPages. This service is commonly used by search engines or Web caches to build up their repositories. Crawling is the basic technique for building huge data storages.
The large volume of the Web implies that any Web crawler can only download a fraction of the existent Web pages within a given time, so it needs to prioritize its downloads. The high rate of change of the Web implies that by the time the crawler is downloading the last pages from a site, it is very likely that new pages have been added to the site, or that pages that have already been updated or even deleted [Baeza and Castillo, 2006].

Crawling the Web, in a certain way, resembles watching the sky in a clear night: what we see reflects the state of the stars at different times, as the light travels different distances [Baeza and Castillo, 2006].

What a Web crawler gets is not a “snapshot” of the Web, because it does not represent the Web at any given instant of time: the last pages being crawled are probably very accurately represented, but the first pages that were downloaded have a high probability of have been changed [Baeza and Castillo, 2006]. A crawler requires a smart scheduling policy for downloading Web pages, in order to ensure the accuracy of the information it gets, since web pages change over time. This may become a harder problem in the future, as the amount of information on the Web can potentially grow faster than the available bandwidth for Web crawlers.

The behavior of a Web crawler is the outcome of a scheduling policy that is mostly concerned with which pages to download and in which order (selection policy) and how to re-visit pages. There are several types of Web crawlers. A Web search engine concerned with only one country or one region is called a “vertical search engine” and uses a crawler specifically designed for this purpose. In the case of a vertical crawler or an intranet crawler the problem is easy as the selection policy of pages is mostly related to selecting by a domain name [Baeza and Castillo, 2006].

On the other end, on a global Web search engine, the selection policy deals mostly with when to stop crawling, as the space of Web pages is infinite. In this regard, a usual criteria is link depth, i.e., starting from the home page, follow links up to a certain level. There is a third kind of selection policy that is used by topical crawlers. In this case, the importance of a page for a crawler can be expressed as a function of the similarity of a page to a given query. This is called “focused

5
crawling” [Chakrabarti et al., 1999]. The main problem in focused crawling is that in the context of a Web crawler, we would like to be able to predict the similarity of the text of a given page to the query before actually downloading the page. The performance of a focused crawling depends mostly on the richness of links in the specific topic being searched, and a focused crawling usually relies on a general Web search engine for providing starting points.

Focused crawling goes a step further than the classic approach. It is able to crawl particular topical (focused) portions of the World Wide Web quickly. It is therefore very attractive for scenarios where not the whole Web is of interest, but only a certain domain. These can be specialized search engines or companies trying to gain information in their field of activity. A focused crawling algorithm loads a page and extracts the links. By rating the links based on keywords, the crawler decides which page to retrieve next. Link by link the Web is traversed. In order to perform focused crawling, domain knowledge is needed. Nowadays, domain knowledge in variety of domains is represented in a form of ontology. An ontology provides a common vocabulary of an area and defines, with different levels of formality, the meaning of the terms and the relationships between them.

Ontology construction, in general, involves the selection of concepts to be included in the ontology, the specification of concepts’ attributes and relations between concepts and, finally, the population of the ontology with instances filling their concepts’ properties.

Ontology building is a complex task and requires special attention in order to build a useful and machine exploitable ontology. A general methodology for building an ontology proposes a concrete number of stages towards its development [Gruber, 1995]:

• **Specification:** the purpose of the ontology is specified in order to restrict the various conceptual models that can be used for modelling (conceptualising) the domain.
• **Conceptualisation:** terms that represent concepts, their attributes and relations are enumerated in order to form the conceptual description of the ontology.
• **Formalization:** the conceptual description of the previous stage is transformed into a formal model. Formal definition of concepts is performed through axioms.
that restrict the possible interpretations for the meaning of these concepts, as well as through relations that organize the concepts such as is-a or part-of relations.

- **Implementation**: the formalized ontology is implemented using a knowledge representation language.

- **Maintenance**: the implemented ontology is periodically corrected and updated. This is a crucial stage for the exploitation of an ontology since ontologies used in practical applications need to evolve taking into account the changes occurring in the domain knowledge they represent, i.e. new instances or variants of existing instances are introduced, new concepts may have to be added, etc.

In chapter 3 we describe the first approach to the knowledge of the energy security domain with the sample use of the web and the first four steps previously enumerated. According to Gruber [1995], the general design criteria for ontology engineering are the following:

- **Clarity**: context-independent, unambiguous, precise definitions. This is directly related to the conceptualisation stage where the conceptual model of the ontology to be built is described (i.e. ontology concepts, their attributes and relations) and is also related to the formalization stage since formal models provide the means to define concepts using necessary and sufficient conditions.

- **Coherence**: the definitions given must be consistent. As Gruber notes, at the least the formally defined part of the ontology should be logically consistent. The formal part of the ontology can be checked for consistency using an inference engine provided by the knowledge representation mechanism used in the implementation stage.

- **Extendibility**: the ontology should enable the addition of new concepts (as well as attributes and relations of existing concepts) without having to revise the existing conceptual definitions. This criterion affects not only the conceptualisation and formalization stages but also the implementation and the maintenance stages.

- **Minimal encoding bias**: representational choices must not be made for the benefit of implementation. This means that an ontology must be as independent as possible from the application which will use the ontology. This will facilitate the re-use of the ontology by other applications. A domain ontology captures an area
of the domain generically, and serves as a reusable knowledge entity that can be shared across this domain.

- *Minimal ontological commitment:* define only those knowledge units (concepts, attributes and relations) that are necessary to support the intended knowledge sharing activities. This criterion is again related to re-usability since the less committed an ontology is, the more re-usable will become in new applications.

### 1.2 The research question

Our work is in the area of focused document crawling and the general research question is:

How can knowledge directed focus crawling be applied in the domain of Energy Security for automatic search of domain related documents? How can an ontology help the formulation of a query in a focused spider? How can we evaluate focused-crawling techniques?

In chapter 2 we present a survey of focused web crawling algorithms and particularly some algorithms of different interesting applications to retrieve relevant documents for our Energy Security domain. We do not only use keywords for the crawl, but rely on high-level background knowledge with concepts and relations, which are compared with the texts of the searched page. We develop an ontology structure for the purposes of the focused crawler and an empirical evaluation which shows that crawling based on background knowledge clearly outperforms standard focused-crawling techniques.

### 1.3 The research methods and tools

In this work we present the design and implementation of knowledge directed information search system applied to the domain of energy security. Energy security is a key component of energy policy in many countries. Since all economic activity requires the use of energy resources, the continuing availability of energy at a price that protects economic growth is a major concern of governments. For this reason we have chosen to build an ontology to support the deep knowledge of the domain.
Regarding the creation of the ontology, the five design criteria were described below were satisfied. We satisfy the Clarity criterion with the use of Web Ontology Language (OWL) that is a formal language for representing ontologies in the Semantic Web [Dean et al, 2003]. OWL has features from several families of representation languages, including primarily Description Logics (DL) and shares many characteristics with RDF. DL provides high-level descriptions of the world. We use an open-source platform, called Protégé that implements a rich set of knowledge-modelling structures and actions that support the creation, visualization, and manipulation of ontologies in OWL format. In sections 3.2 and 3.3 we describe the use of OWL-DL and of Protégé to build a first version of energy security ontology. We satisfy the Coherence requirement by using RacerPro that is a server for OWL inference services and we describe in section 2.11 the results of the Racer application on our ontology. Minimal encoding bias is addressed by section 2.11.

In chapter 3 we explain the Minimal ontological commitment step: this concerns mainly the population of the ontology with new instances filling their concepts’ attributes and its enrichment with typographic variants of existing instances. The basic idea behind this step is that when we want to model a domain for an application, it is enough to have at the beginning an ontology containing only a small number of knowledge units as well as a corpus of domain specific documents. We use the different focused crawler described in chapter 2 to ontology population. Moreover we present a mechanism to estimate the relevance of a page.

Although there was an important body of information retrieval algorithms and techniques published before the invention of the World Wide Web, there are unique characteristics of this new medium that made those techniques unsuitable or insufficient for Web search. The vector space model is the standard technique for ranking documents according to a query. Under this model, both a document and a query are seen as a pair of vectors in a space with as many dimensions as terms as the vocabulary. In a space defined in this way, the similarity of a query to a document is given by a formula that transforms each vector using certain weights and then calculates the cosine of the angle between the two weighted
vectors. In a text-based information retrieval systems, documents are shown to the user in decreasing order using this similarity measure.

A weighting scheme uses statistical properties from the text and the query to give certain words more importance when doing the similarity calculation. The most used scheme is the TF-IDF weighting scheme that uses the frequency of the terms in both queries and documents to compute the similarity.

Our approach used to evaluate the results of query in different topical crawlers is to assess the similarity between the retrieved page and a gold standard that represents a relevant page. In a first approach our gold standard is represented by documents in MySql database that are relevant because we have read them and they have driven the ontology concepts formalization. In an other approach we use general framework for crawler evaluation research defined by Menczer et al [2004].
Chapter 2

Relevant literature survey on focused crawler

2.1 Search engines and web crawlers

Searching the Web is a difficult task and a lot of machine learning work is being applied to one part of this task, namely ranking indexed pages by their estimated relevance with respect to user queries [Menczer et al, 2004]. This is a crucial task because it heavily influences the perceived effectiveness of a search engine. Users often look only at a few top hits, making the precision achieved by the ranking algorithm of paramount importance [Menczer et al, 2004]. Early search engines ranked pages principally based on their lexical similarity to the query [Menczer et al, 2001]. The key strategy was to devise the best weighting algorithm to represent Web pages and queries in a vector space, such that closeness in such a space would be correlated with semantic relevance [Menczer and Pant, 2004]. More recently the structure of hypertext links has been recognized as a powerful new source of evidence for Web semantics. Many machine learning techniques have been employed for link analysis, so that a page’s linkage to other pages, together with its content, could be used to estimate its relevance [Kleinberg, 1998]. The best known example of such link analysis is the PageRank algorithm successfully employed by the Google search engine. Other machine learning techniques to extract meaning from link topology have been based on identifying hub and authority pages via eigenvalue analysis [Kleinberg, 1998] and on other graph-theoretical approaches. While purely link-based methods have been found to be effective in some cases, link analysis is most often combined with lexical pre/post-filtering [Brin and Page, 1998]. However, all this research only addresses half of the problem. No matter how sophisticated the ranking algorithm we build, the results can only be as good as the pages indexed by the search engine—a page cannot be retrieved if it has not been indexed. This brings us to the other aspect of Web searching, namely crawling the Web in search of pages to be indexed.
Web crawlers are programs that exploit the graph structure of the Web to move from page to page [Menczer et al, 2004]. In its simplest form a crawler starts from a seed page and then uses the external links within it to attend to other pages. The process repeats with the new pages offering more external links to follow, until a sufficient number of pages are identified or some higher level objective is reached. Behind this simple description lies a host of issues related to network connections, spider traps, canonicalizing URLs, parsing HTML pages, and the ethics of dealing with remote Web servers.

Search engines use Web crawlers to maintain their index databases amortizing the cost of crawling and indexing over the millions of queries received by them. These crawlers are exhaustive in their approach, with comprehensiveness as their major goal. In contrast, crawlers can be selective about the pages they fetch and are then referred to as preferential or heuristic-based crawlers. These may be used for building focused repositories, automating resource discovery, and facilitating software agents.

Preferential crawlers built to retrieve pages within a certain topic are called topical or focused crawlers. An aspect that is important to consider when studying crawlers, especially topical crawlers, is the nature of the crawl task [Menczer et al, 2004]. Crawl characteristics such as queries and/or keywords provided as input criteria to the crawler, user-profiles, and desired properties of the pages to be fetched (similar pages, popular pages, authoritative pages etc.) can lead to significant differences in crawler design and implementation.

Agents for topic driven searching (also known as topic driven crawlers and focused crawlers) respond to the particular information needs expressed by topical queries or interest profiles [Menczer et al, 2004]. These could be the needs of an individual user or those of a community with shared interests. They support decentralizing the retrieval process, which is a more scalable approach when compared with centralized multipurpose search engines. An additional benefit is that such agents can be driven by a rich context (topics, queries, user profiles) with which to interpret pages and select the links to be visited.

The current work focuses on applying focused crawlers in a specific domain.
Focused crawlers are agents that can selectively retrieve Web documents relevant to a specific domain so we studied how we can download topic-specific content from the Internet about our specific domain of energy security. We investigated how to apply some algorithms employed by focused crawlers for a task of ontology population.

In the following sections we describe a general crawler infrastructure and a variety of crawling algorithms.

2.2 Crawler infrastructure

Crawling can be viewed as a graph search problem [Menczer et al, 2004]. The Web is seen as a large graph with pages at its nodes and hyperlinks as its edges. A crawler starts at a few of the nodes (seeds) and then follows the edges to reach other nodes. The process of fetching a page and extracting the links within it is analogous to expanding a node in graph search. A topical crawler tries to follow edges that are expected to lead to portions of the graph that are relevant to a topic.

![Crawler infrastructure](image)
In a crawling loop a URL is picked from the frontier (a list of unvisited URLs), the page corresponding to the URL is fetched from the Web, the fetched page is processed through all the infrastructure layers (see Figure 2.1), and the unvisited URLs from the page are added to the frontier. The networking layer is primarily responsible for fetching and storing a page and making the process transparent to the layers above it. The parsing and extraction layer parses the page and extracts the needed data such as hyperlink URLs and their contexts (words, phrases etc.). The extracted data is then represented in a formal notation (say a vector) by the representation layer before being passed onto the intelligence layer that associates priorities or scores with unvisited URLs in the page. In a general-purpose crawler the topmost two layers (representation and intelligence layers) may not be implemented. For example, the crawler may be an exhaustive crawler. However, for a topical crawler, the unvisited URLs from the fetched page are scored to reflect their potential value before being added to the frontier. Hence, a topical crawler includes an implementation of the representation layer and the intelligence layer. The score given to an unvisited URL estimates the benefit of visiting the page corresponding to the URL. The crawling process may be terminated when a certain number of pages have been crawled. If the crawler is ready to crawl another page and the frontier is empty, the situation signals a dead-end for the crawler. The crawler has no new pages to fetch and hence it stops.

2.2.1 Inputs

The inputs to the intelligence layer initiate a crawl. In the intelligence layer the inputs may be used to suitably represent the desired information need, topic, or theme for which the crawler needs to fetch Web pages. An input can be as simple as a set of keywords or as complex as a taxonomy with representative Web pages. A set of keywords can be converted into a vector of weights where each weight represents the relative importance of the word. We’ll see in the sections dedicated to each used spider as we construct query to activate it.
2.2.2 Frontier

The frontier is the to-do list of a crawler that contains the URLs of unvisited pages. In graph search terminology the frontier is an open list of unexpanded (unvisited) nodes. Based on the available memory, one can decide on the maximum size of the frontier. Due to the large amount of memory available on PCs today, a frontier size of a 100,000 URLs or more is not exceptional. The frontier may be implemented as a FIFO (first-in-first-out) queue in which case we have a breadth-first crawler that can be used to exhaustively crawl the Web. The URL to crawl next comes from the head of the queue and the new URLs are added to the tail of the queue. If the frontier is implemented as a priority queue we have a preferential crawler which is also known as a best-first crawler. A frontier can be implemented as a priority queue using a dynamic array. The array is always kept sorted by the estimated score of unvisited URLs. The unvisited URLs that are scored the same are ordered in the frontier on a first-in basis.

At each step, the best URL is picked from the head of the queue. Once the corresponding page is fetched, the URLs are extracted from it and scored based on some heuristic. They are then added to the frontier in such a manner that the order of the priority queue is maintained. We can avoid duplicate URLs in the frontier by keeping a separate hash-table of URLs for lookup. Once the frontier’s maximum size (MAX) is exceeded, only the best MAX URLs are kept in the frontier. If the crawler finds the frontier empty when it needs the next URL to crawl, the crawling process comes to a halt. With a large value of MAX and several seed URLs the frontier will rarely reach the empty state. At times, a crawler may encounter a spider trap that leads it to a large number of different URLs that refer to the same page. One way to alleviate this problem is by limiting the number of pages that the crawler accesses from a given domain.

2.2.3 History and Page Repository

The crawl history is a time-stamped list of URLs that were fetched by the crawler. In effect, it shows the path of the crawler through the Web starting from the seed pages. A URL entry is made into the history only after fetching the corresponding
page. This history may be used for post crawl analysis and evaluations. For example, we can associate a value with each page on the crawl path and identify significant events (such as the discovery of an excellent resource). While the history may be stored occasionally to the disk, it is also maintained as an in-memory data structure. This provides for a fast lookup to check whether a page has been crawled or not. This check is important to avoid revisiting pages and also to avoid adding the URLs of crawled pages to the limited size frontier. Once a page is fetched, it may be stored/indexed for the master application (such as a search engine).

2.2.4 Networking Layer
The networking layer is responsible for getting the next (best) URL from the frontier and fetching it from the Web or the local page repository. Whether a page is obtained from the Web or from the local repository is a detail that is hidden from the higher layers. In order to fetch a Web page, the networking layer contains an HTTP client which sends an HTTP request for a page and reads the response. The client needs to have timeouts to make sure that a large amount of time is not spent on slow servers or in reading large pages. Modern programming languages such as Java and Perl provide very simple and often multiple programmatic interfaces for fetching pages from the Web. However, one must be careful when using high level interfaces where it may be harder to find lower level problems. No discussion about crawling pages from the Web can be complete without talking about the Robot Exclusion Protocol. This protocol provides a mechanism for Web server administrators to communicate their file access policies; more specifically to identify files that may not be accessed by a crawler. This is done by keeping a file named robots.txt under the root directory of the Web server. When a crawler wants to retrieve a page from a Web server, it must first fetch the appropriate robots.txt file and make sure that the URL to be fetched is not disallowed.
2.2.5 Parsing and Extraction Layer

Once a page has been fetched, we need to parse its content to extract information that will possibly guide the future path of the crawler. Parsing and extraction may imply simple hyperlink/URL extraction or it may involve the more complex process of tidying up the HTML content in order to analyze the HTML tag tree. Parsing and extraction might also involve steps to convert the extracted URL to a canonical form, remove stopwords from the page’s content and stem the remaining words.

HTML Parsers are freely available for many different languages. These parsers provide the functionality to easily identify HTML tags and associated attribute-value pairs in a given HTML document. However, we do need to convert any relative URLs to absolute URLs using the base URL of the page from which they were extracted. Different URLs that correspond to the same Web page can be mapped onto a single canonical form. This is important in order to avoid fetching the same page more than once. Here are some of the steps used in typical URL canonicalization procedures that our implementation follows:

- convert the protocol and hostname to lowercase;
- remove the ‘anchor’ or ‘reference’ part of the URL;
- perform URL encoding for some commonly used characters such as ‘˜’;
- use heuristics to recognize default Web pages. File names such as index.html or index.htm may be removed from the URL with the assumption that they are the default files. If that is true, they would be retrieved by simply using the base URL;
- remove ‘..’ and its parent directory from the URL path.
- leave the port numbers in the URL unless it is port 80.

It is important to be consistent while applying canonicalization rules. It is possible that two seemingly opposite rules work equally well (such as that for port numbers) as long as they are applied consistently across URLs.

When parsing a Web page to extract content information or in order to score new URLs suggested by the page, it is often helpful to remove commonly used words or stopwords such as “it” and “can”. In addition, one may also stem the words found in the page. The stemming process normalizes words by conflating
morphologically similar words to a single root form or stem. For example, “connect,” “connected” and “connection” are all reduced to “connect.” Implementations of the commonly used Porter stemming algorithm are easily available in many programming languages.

2.2.6 Representation Layer

The representation layer is responsible for converting the parsed pages and their extracted details into abstract mathematical representations. These representations will be used by the higher intelligence layer to appropriately score the unvisited URLs. There are two main types of representations used:

1. **Content Representation.** The content of a link context may be represented as a vector in an n-dimensional space, where each dimension corresponds to a term (word, stem, or phrase) in a given vocabulary. Link context is a neighbourhood of a hyperlink in a Web page that gives cues about the content of the destination page (page that you reach by following the link). We note that, as an extreme cases, an entire Web page may be considered as link context of all the hyperlinks within it. The hierarchical structure of the Web pages created by the placement of HTML tags within it may also be exploited for deriving link contexts. Each component of the vector representing a link context is a term weight that signifies the importance of the corresponding term in the link context.

A popular weighting scheme for text content is TF-IDF. There are many variations within the TF-IDF representation, but it always consists of two parts - term frequency (TF) and inverse document frequency (IDF). In according with Lucchese [2003] intuitively, a term that occurs many times in a document is more representative of that document than a term occurring just once. The weight of a term should be proportional to its frequency in the document and this is TF. Some terms occur very often in the whole collections while other ones occur seldomly so the weight of a term should be inversely proportional to the number of documents containing that term, that is IDF.

The TF part is a function of how often a term appears in a given link context, while the IDF weighs down the terms that commonly appear across pages. The
product of the two quantities gives tf*idf coefficient of a term, which can be used as a component of the vector representing a given document. These measures are very important in the section dedicated to evaluation of the spider used to the task of ontology population.

2. Linkage or Graph Representation. The canonical URLs extracted from Web pages may be used to build a representation of the Web as a graph. The nodes in the graph are the pages and edges are the hyperlinks between the pages. This graph may be used to find the pages that seem to be “well-connected” to relevant pages. An unvisited URL that is pointed to by many relevant pages may be given higher priority during a crawl. The graph may be represented as an adjacency list, with the aim of listing the URLs of pages that have been downloaded along with the URLs of pages they link to. This approach is largely used in one of the focused crawler used in this thesis, that is BINGO!.

2.2.7 Intelligence Layer
This layer is the “brain” of a topical crawler and is responsible for using the representations provided by the layer below to assign priorities to the unvisited URLs in the frontier. The intelligence layer may use a variety of heuristics to score the unvisited URLs and hence guide the crawler. We now list some of the general techniques that may provide the intelligence. Often a combination of these general techniques is used.

1. Classification. If the inputs are positive and negative examples (pages) of a topic, a two category classifier can be trained using these examples. The classifier can then be used to score downloaded pages based on their content. The unvisited URLs extracted from the downloaded page are assigned the scores based on contents of their parent page. Such a classifier-guided topical crawler was first suggested by Chakrabarti et. al. [1999].

2. Cosine Similarity. If the input to the intelligence layer is a set of keywords, then the cosine similarity between the keywords and a link context may be used to score the corresponding URL. The cosine similarity is computed as:

\[
\text{sim}(q, p) = \frac{(v_q \ast v_p)}{||v_q|| \ast ||v_p||}
\]
where \( v_q \) and \( v_p \) are vector representation (such as TF-IDF weights) of keywords (q) and link context (p) respectively, \( v_q \cdot v_p \) is the dot product between the vectors, and \( k_v \) is the Euclidean norm of a vector v. We’ll see this process in Infospiders’ application.

3. Link/Graph Analysis. If a link based representation for a set of downloaded pages, algorithms such as Kleinberg’s Hubs and Authorities algorithm [Kleinberg, 1998] and Brin and Pages’s PageRank algorithm [Brin and Page, 1998] to identify resourceful URLs that may be preferred by a crawler due to their centrality to a topic. We’ll see this process in BINGO!.

### 2.3 Crawling algorithms

The literature suggests different crawling algorithms and many of these algorithms are variations of the best first scheme [Menczer et al, 2005]. The difference is in the heuristics they use to score the unvisited URLs with some algorithms adapting and tuning their parameters before or during the crawl.

#### 2.3.1 Naive Best-First Crawler

This crawler represents a fetched Web page as a vector of words weighted by occurrence frequency. The crawler then computes the cosine similarity of the page to the query or description provided by the user, and scores the unvisited URLs on the page by this similarity value. The URLs are then added to a frontier that is maintained as a priority queue based on these scores. In the next iteration each crawler thread picks the best URL in the frontier to crawl, and returns with new unvisited URLs that are again inserted in the priority queue after being scored based on the cosine similarity of the parent page. The cosine similarity between the page \( p \) and a query \( q \) is computed by Equation 1.

#### 2.3.2 Breadth-First

A Breadth-First crawler is the simplest strategy for crawling. It uses the frontier as a FIFO queue, crawling links in the order in which they are encountered. Note that when the frontier is full, the crawler can add only one link from a crawled page.
The Breadth-First crawler is illustrated in Figure 2.2 and the algorithm is shown in Figure 2.3. Breadth-First is used here as a baseline crawler; since it does not use any knowledge about the topic, we expect its performance to provide a lower bound for any of the more sophisticated algorithms.

2.3.3 Best-First

The basic idea is that given a frontier of links, the best link according to some estimation criterion is selected for crawling. Different Best-First strategies of increasing complexity and (potentially) effectiveness could be designed on the bases of increasingly sophisticated link estimation criteria. In “naive” implementation, the link selection process is guided by simply computing the lexical similarity between the topic’s keywords and the source page for the link. Thus the similarity between a page \( p \) and the topic keywords is used to estimate the relevance of the pages pointed by \( p \). The URL with the best estimate is then selected for crawling. Cosine similarity is used by the crawler and the links with minimum similarity score are removed from the frontier if necessary in order to not exceed the limit size MAX BUFFER. Figure 2.4 illustrates a Best-First crawler, while Figure 2.5 offers a simplified pseudocode of the Best-First-Search (BFS) algorithm.
2.3.4 Fish Search and SharkSearch

Menczer et al [2004] describe FishSearch as one of the earliest topical crawling algorithms: it uses a similarity measure like the one used in the naive best-first crawler for estimating the relevance of an unvisited URL. Fish search also has a notion of depth-bound that prevents it from exploring paths that lead to a sequence irrelevant pages. SharkSearch, while adopting the concepts of FishSearch, has a more refined notion of potential scores for the links in the crawl frontier. The anchor text, text surrounding the links or link-context, and scores inherited from ancestors influence the potential scores of links. The ancestors of a URL are the pages that appeared on the crawl path to the URL. SharkSearch, like its predecessor FishSearch, maintains a depth bound. That is, if the crawler finds unimportant pages on a crawl path up to a certain depth or distance it stops crawling further along that path. To be able to track all the information, each URL in the frontier is associated with a depth and a potential score. The depth bound (d) is provided by the user while the potential score of an unvisited URL is computed as:

\[
\text{score(url)} = \gamma \cdot \text{inherited(url)} + (1 - \gamma) \cdot \text{neighborhood(url)}
\]

where \( \gamma < 1 \) is a parameter, the neighbourhood score signifies the contextual evidence found on the page that contains the hyperlink URL, and the inherited score is obtained from the scores of the ancestors of the URL. More precisely, the inherited score is computed as:

\[
\text{inherited(url)} = \delta \cdot \text{sim}(q, p) \quad \text{if sim}(q, p) > 0
\]
\[ \text{inherited } d(\text{url}) = \delta \cdot \text{inherited}(p) \quad \text{otherwise} \]

where \( \delta < 1 \) is again a parameter, \( q \) is the query, and \( p \) is the page from which the URL was extracted.

The neighbourhood score uses the anchor text and the text in the “vicinity” of the anchor in an attempt to refine the overall score of the URL by allowing for differentiation between links found within the same page. For that purpose, the SharkSearch crawler assigns an anchor score and a context score to each URL.

The anchor score is simply the similarity of the anchor text of the hyperlink containing the URL to the query \( q \), i.e. \( \text{sim}(q, \text{anchor text}) \). The context score on the other hand broadens the context of the link to include some nearby words. The resulting augmented context, \( \text{aug\_context} \), is used for computing the context score as follows:

\[
\begin{align*}
\text{context } (\text{url}) &= 1 \quad \text{if anchor(\text{url})} > 0 \\
\text{context } (\text{url}) &= \text{sim } (q, \text{aug\_context}) \quad \text{otherwise}
\end{align*}
\]

Finally we derive the neighbourhood score from the anchor score and the context score as:

\[
\text{neighbourhood}(\text{url}) = \beta \cdot \text{anchor(\text{url})} + (1 - \beta) \cdot \text{context(\text{url})}
\]

where \( \beta < 1 \) is another parameter. We note that the implementation of SharkSearch would need to preset four parameters \( d, \gamma, \delta, \) and \( \beta \).

### 2.3.5 Focused Crawler

The basic idea of the focused crawler is to classify crawled pages with categories in a topic taxonomy [Chakrabarti et al, 1999]. To begin, the crawler requires a topic taxonomy such as Yahoo. In addition, the user provides example URLs of interest. The example URLs get automatically classified onto various categories of the taxonomy. Through an interactive process, the user can correct the automatic classification, add new categories to the taxonomy and mark some of the categories as “good” (i.e., of interest to the user). The crawler uses the example URLs to build a Bayesian classifier that can find the probability \( (\text{Pr}(c|p)) \) that a crawled page \( p \) belongs to a category \( c \) in the taxonomy.

A relevance score is associated with each crawled page that is computed as:

\[
R(p) = \sum \text{Pr } (c|p)
\]
When the crawler is in a “soft” focused mode, it uses the relevance score of the crawled page to score the unvisited URLs extracted from it. The scored URLs are then added to the frontier. Then in a manner similar to the naive best-first crawler, the “soft” focused crawler picks the best URL to crawl next. In the “hard” focused mode, for a crawled page p, the classifier first finds the leaf node c* (in the taxonomy) with maximum probability of including p. If any of the parents (in the taxonomy) of c* are marked as “good” by the user, then the URLs from the crawled page p are extracted and added to the frontier. Another interesting element of the focused crawler is the use of a distiller. The distiller applies a modified version of Kleinberg’s algorithm [Kleinberg, 1998] to find topical hubs. These hubs are pages that provide links to many authoritative sources on the topic. The distiller is activated at various times during the crawl and top ranking hubs are added to the frontier.

2.3.6 Context Focused Crawler

Context focused crawlers use Bayesian classifiers to guide their crawl. However, unlike the focused crawler described above, these classifiers are trained to estimate the link distance between a crawled page and the relevant pages. Diligenti et al. [2000], assume that looking for papers on “numerical analysis” it means first to go to the home pages of math or computer science departments and then move to faculty pages which may then lead to the relevant papers. A math department Web site may not have the words “numerical analysis” on its home page. A crawler such as the naive best-first crawler would put such a page on low priority and may never visit it. However, if the crawler could estimate that a relevant paper on “numerical analysis” is probably two links away, we would have a way of giving the home page of the math department higher priority than for example the home page of a law school. The context focused crawler is trained using a context graph of L layers corresponding to each seed page. Seeds are the pages from where a crawler begins its crawl. As we see in the figure 2.6 the seed page forms the layer 0 of the graph. The pages corresponding to the in-links to the seed page are in layer 1. The in-links to the layer 1 pages make up the layer 2 pages and so on. Once the context graphs for all of the seeds are obtained, the
pages from the same layer from each graph are combined into a single layer. This gives a new set of layers of what is called a merged context graph. This is followed by a feature selection stage where the seed pages (or possibly even layer 1 pages) are concatenated into a single large document. Using the TF-IDF scoring scheme, the top few terms are identified from this document to represent the vocabulary (feature space) that will be used for classification. A set of Naive Bayes classifiers are built, one for each layer in the merged context graph. All the pages in a layer are used to compute Pr (t | cl), the probability of occurrence of a term t given the class cl corresponding to layer l. A prior probability, Pr(cl) = 1/L, is assigned to each class where L is the number of layers. The posterior probability of a given page p belonging to a class cl can then be computed Pr (cl | p) using the Bayes rule. Such probabilities are computed for each class. The class with highest probability is treated as the winning class (layer). However, if the probability for the winning class is still less than a threshold, the crawled page is classified into the “other” class. This “other” class represents pages that do not have a good fit with any of the classes of the context graph. If the probability of the winning class does exceed the threshold, the page is classified into the winning class. The set of classifiers corresponding to the context graph provides us with a mechanism to estimate the link distance of a crawled page from relevant pages. If the mechanism works, the math department home page will get classified into layer 2 while the law school home page will get classified to “others.” The crawler maintains a queue for each class, containing the pages that are crawled and classified into that class. Each queue is sorted by the probability scores (Pr (cl | p)). When the crawler needs a URL to crawl, it picks the top page in the non-empty queue with smallest l. So it will tend to pick pages that seem to be closer to the relevant pages first. The out-links from such pages will get explored before the out-links of pages that seem to be far away from the relevant portions of the Web. Diligenti et al. [2000], evaluated their context focused crawler over 2 topics and less than 10,000 pages were crawled on an average for each topic. Again it remains difficult to reach conclusions regarding the potential value in using such contexts for building topical crawlers, given the small number of topics tested.
2.3.7 InfoSpiders
InfoSpiders search online for information relevant to the user, by making autonomous decisions about what links to follow [Menczer and Monge, 1999]. In InfoSpiders, an adaptive population of agents search for pages relevant to the topic. Each agent is essentially following the crawling loop while using an adaptive query list and a neural net to decide which links to follow. The algorithm provides an exclusive frontier for each agent. In a multi-threaded implementation of InfoSpiders (MySpiders, that we see in the following section) each agent corresponds to a thread of execution.

The user initially provides a list of keywords (query) and a list of starting points, in the form of a bookmark file. This list could typically be obtained by consulting a search engine. First, the population is initialised by pre-fetching the starting documents. Each agent is “positioned” at one of these documents and given a random behaviour (depending on the representation of agents) and an initial reservoir of energy. The user also provides a maximum number of pages that the population of agents are allowed to visit, collectively. This would depend on how long the user is willing to wait, or how much bandwidth she is willing to consume. Finally, the user may specify whether and how often she is willing to provide the population with relevance feedback, to help focus or shift the search toward relevant areas by “replenishing” the resources in those areas.

In the innermost loop (as we see in following code lines, figure 2.7), an agent “senses” its local neighbourhood by analysing the text of the document where it is
currently situated. This way, the relevance of all neighbouring documents - those pointed to by the hyperlinks in the current document - is estimated. Based on these link relevance estimates, the agent “moves” by choosing and following one of the links from the current document.

```plaintext
InfoSpiders(query, starting_urls, MAX_PAGES, REL_FBK_FLAG) {
  for agent (1..INIT_POP) {
    initialize(agent, query);
    situate(agent, starting_urls);
    agent.energy := THETA / 2;
  }
  while (pop_size > 0 and visited < MAX_PAGES) {
    for each agent {
      pick_outlink_from_current_document(agent);
      agent.doc := fetch_new_document(agent);
      agent.energy += benefit(agent.doc) - cost(agent.doc);
      apply_Q_learning(agent, benefit(agent.doc));
      if (agent.energy >= THETA) {
        offspring := mutate(recombine(clone(agent)));
        offspring.energy := agent.energy / 2;
        agent.energy -= offspring.energy;
        birth(offspring);
      } elseif (agent.energy <= 0) death(agent);
    }
    if (REL_FBK_FLAG) process_relevance_feedback(input);
  }
}
```

Figure 2.7. Some code lines of an agent’s loop [Menczer and Monge, 1999]

The agent's energy is then updated. Energy is needed in order to survive and move, i.e., continue to visit documents on behalf of the user. Agents are rewarded with energy if the visited documents appear to be relevant. After a document has been visited, its similarity to the query is used as a learning signal to update the weights of the neural net. Further, the agent computes a fitness value to update its energy level. Since the energy dynamics of an agent determine its reproductive rate, determining an appropriate fitness function to map the quality and cost of visited pages into energy is crucial for the success of the
system. Users will only use tools that are both effective and efficient. For this reason there are two components in fitness functions: a benefit based on the similarity to the query, and a cost to account for the network resources used to download pages.

The benefit() function is used by an agent to estimate the relevance of documents. In the absence of relevance feedback, this function return a non-zero value only if the document had not been previously visited by any agent, and according to a standard measure of similarity between the query and the document. The “marking” of visited pages models the consumption of finite information resources and is implemented via a cache, which also speeds up the process by minimizing duplicate transfers of documents. Agents are charged energy for the network load incurred by transferring documents.

\[
\text{fitness}(p) = \text{sim}(p, \text{query}) - (c_t \times (t(p) / t_{\text{max}}))
\]

where \(\text{sim}()\) is the cosine similarity function, \(t(p)\) is the real time in which the page is downloaded, \(t_{\text{max}}\) is the timeout parameter for the socket connection.

The cost() function should depend on used resources, for example transfer latency or document size. For simplicity we assume a constant cost for accessing any new document, and a smaller constant cost for accessing the cache; this way stationary behaviours, such as going back and forth between a pair of documents, are naturally discouraged. Instantaneous changes of energy are used as reinforcement signals. This way agents adapt during their lifetime by Q-learning [Menczer and Belew, 1998]. This adaptive process allows an agent to modify its behaviour based on prior experience, by learning to predict the best links to follow. Local selection means that an agent is selected for reproduction based on a comparison between its current energy level and a constant that is independent of the other agents in the population. Similarly, an agent is killed when it runs out of energy.

At reproduction, agents may be recombined by the use of one of two types of crossover. Offspring are also mutated, providing the variation necessary for adapting agents by way of evolution. Energy is conserved at all reproduction events. The output of the algorithm is a flux of links to documents, ranked
according to estimated relevance. The algorithm stops when the population goes extinct for lack of relevant information resources, visits MAX PAGES documents, or is terminated by the user.

Figure 2.8 illustrates the architecture of each InfoSpiders agent. The agent interacts with the information environment, that consists of the actual networked collection (the Web) plus data kept on local disks (e.g., relevance feedback data and cache files). The user interacts with the environment by accessing data on the local client (current status of a search) and on the Web (viewing a document suggested by agents) and by making relevance assessments that are saved locally on the client and will be accessed by agents as they subsequently report to the user/client. There is no direct interaction between the user and the agents after the initial submission of the query and starting points. The InfoSpiders prototype runs on UNIX and MacOS platforms. The Web is based on the W3C library. Agents employ standard information retrieval tools such as parsing, a filter for noise words (stop words) and a stemmer based on Porter's algorithm. They store an efficient representation of visited documents in the shared cache on the client machine. Each document is represented by a list of links and stemmed keywords. If the cache reaches its size limit, the LRU (least recently used) replacement strategy is used.
2.3.8 PageRank and its usage to guide crawling

PageRank was proposed by Brin and Page [1998] as a possible model of user surfing behaviour. The PageRank of a page represents the probability that a random surfer (one who follows links randomly from page to page) will be on that page at any given time. A page’s score depends recursively upon the scores of the pages that point to it.
Source pages distribute their PageRank across all of their outlinks. Formally (Equation 2):

$$PR(p) = (1 - \gamma) + \gamma \sum_{d \in in(p)} \frac{PR(d)}{|out(d)|}$$

where $p$ is the page being scored, $in(p)$ is the set of pages pointing to $p$, $out(d)$ is the set of links out of $d$, and the constant $\gamma < 1$ is a damping factor that represents the probability that the random surfer requests another random page. As originally proposed PageRank was intended to be used in combination with content-based criteria to rank retrieved sets of documents. This is in fact how PageRank is used in the Google search engine. More recently PageRank has been used to guide crawlers and to assess page quality. Menczer et al. 2001 [Menczer et al, 2001] evaluated a crawler based on PageRank. They used an efficient algorithm to calculate PageRank. Even so, as one can see from Equation 2, PageRank requires a recursive calculation until convergence, thus its computation can be a very resource-intensive process. In the ideal situation it would recalculate PageRanks every time a URL needs to be selected from the frontier. Instead, to improve
efficiency, it needs to recompute PageRanks only at regular intervals. The PageRank crawler can be seen as a variant of Best-First, with a different link evaluation function (see Equation 2). The algorithm is illustrated in Figure 8. The \( \text{sim}() \) function returns the cosine similarity between a topic’s keywords and a page as measured according to Equation 1, and the PageRank is computed according to Equation 2. Note that the PageRank algorithm has to maintain a data structure of visited pages in addition to the frontier, in order to compute PageRank.

2.4 Evaluation of Crawlers

A crawler (especially a topical crawler) may be evaluated on its ability to retrieve “good” pages. There are some of the methods that have been used to measure page importance:

1. **Keywords in document**: A page is considered relevant if it contains some or all of the keywords in the query. Also, the frequency with which the keywords appear on the page may be considered.

2. **Similarity to a query**: Often a user specifies an information need as a short query. In some cases, a longer description of the need may be available. Similarity between the short or long description and each crawled page may be used to judge the page’s relevance.

3. **Similarity to seed pages**: The pages corresponding to the seed URLs are used to measure the relevance of each page that is crawled. The seed pages are combined together into a single document and the cosine similarity of this document and a crawled page is used as the page’s relevance score.

4. **Classifier score**: A classifier may be trained to identify the pages that are relevant to the information need or task. The training is done using the seed (or pre-specified relevant) pages as positive examples. The trained classifier will then provide boolean or continuous relevance scores to each of the crawled pages.

5. **Retrieval system rank**: \( N \) different crawlers are started from the same seeds and allowed to run till each crawler gathers \( P \) pages. All of the \( N \times P \) pages collected from the crawlers are ranked against the initiating query or description.
using a retrieval system such as SMART. The rank provided by the retrieval system for a page is used as its relevance score.

6. **Link-based popularity:** One may use algorithms, such as PageRank, that provide popularity estimates of each of the crawled pages. A simpler method would be to use just the number of in-links to the crawled page to derive similar information. Many variations of link-based methods using topical weights are choices for measuring topical popularity of pages.

Given one of these measures of page importance we can summarize the performance of the crawler with metrics that are analogous to the information retrieval (IR) measures of precision and recall. Precision is the fraction of retrieved (crawled) pages that are relevant, while recall is the fraction of relevant pages that are retrieved (crawled). In a usual IR task the notion of a relevant set for recall is restricted to a given collection or database. Considering the Web to be one large collection, the relevant set is generally unknown for most Web IR tasks. Hence, explicit recall is hard to measure. Many authors provide precision-like measures that are easier to compute in order to evaluate the crawlers [Menczer and Pant, 2004]:

a) **Acquisition rate:** In case we have boolean relevance scores we could measure the explicit rate at which “good” pages are found. Therefore, if 50 relevant pages are found in the first 500 pages crawled, then we have an acquisition rate or harvest rate of 10% at 500 pages.

b) **Average relevance:** If the relevance scores are continuous they can be averaged over the crawled pages. This is a more general form of harvest rate. The scores may be provided through simple cosine similarity or a trained classifier. Such averages may be computed over the progress of the crawl (first 100 pages, first 200 pages and so on). Sometimes running averages are calculated over a window of a few pages (e.g. last 50 pages from a current crawl point).

The measures analogous to recall are hard to compute for the Web, authors [Menczer and Pant, 2004] resort to indirect indicators for estimating recall. Some such indicators are:
a) **Target recall**: A set of known relevant URLs are split into two disjoint sets targets and seeds. The crawler is started from the seeds pages and the recall of the targets is measured. The target recall is computed as:

\[
\text{target recall} = \frac{|P_t \setminus P_c|}{|P_t|}
\]

where \(P_t\) is the set of target pages, and \(P_c\) is the set of crawled pages. The recall of the target set is used as an estimate of the recall of relevant pages.

The figure 2.10 shows a schematic justification of the measure.

![Figure 2.10](image)

b) **Robustness**: The seed URLs are split into two disjoint sets. Each set is used to initiate an instance of the same crawler. The overlap in the pages crawled starting from the two disjoint sets is measured. A large overlap is interpreted as robustness of the crawler in covering relevant portions of the Web.

### 2.5 Algorithms application: MySpiders

MySpiders is a Java applet that implements the InfoSpiders and the Naïve best-first algorithms. The applet is available online (http://myspiders.informatics.indiana.edu/). Multi-threaded crawlers are started when a user submits a query. Results are displayed dynamically as the crawler finds “good” pages. The user may browse the results while the crawling continues in the background. In line with the autonomous multi-agent nature of the InfoSpiders algorithm each thread has a separate frontier. This applies to the naive best-first algorithm as well. Hence, each thread
is more independent with non-contentious access to its frontier. The applet allows the user to specify the crawling algorithm and the maximum number of pages to fetch. In order to initiate the crawl, the system uses the Google Web API to obtain a few seeds pages. The crawler threads are started from each of the seeds and the crawling continues until the required number of pages is fetched or the frontier is empty. The implementation allows the user to change the crawling algorithm from InfoSpiders to Best-First, a crawler that has proved to be a valid alternative under certain circumstances. The shared cache is essential for the working of the multithreaded MySpiders system. Another shared object is the naming mechanism that provides a unique name to each spider. A results table is a shared object that maintains details about the pages that the spiders want to add to the results. Finally, a shared log object is used to maintain a system wide mechanism for logging major events. The shared objects other than the cache can be easily replaced by non-shared ones (as would be required for a distributed implementation), but have been kept for efficiency. Access to all shared objects is thread-safe. The utilities include an entire gamut of functions that provide facilities for retrieving documents from the Web, HTML and XML parsing, stemming, measuring similarity and other tools used by most Web crawlers. In addition to the current location, the agent stores a back-link so that it can move to the previous page and escape (some) dead-ends. It also maintains a backup of inputs used by the neural network’s learning algorithm. All the back-end storage is done in XML. This aids system portability and helps in storing and parsing hierarchical data. An example is the spider hierarchy shown in Figure 2.12.

2.6 Algorithms application: BINGO!

The BINGO! (http://www.mpiinf.mpg.de/departments/d5/software/bingo/downloads.htm) system implements an approach to focused crawling that aims to overcome the limitations of the initial training data. BINGO! identifies, among the crawled and positively classified documents of a topic, characteristic
“archetypes” and uses them for periodically re-training the classifier; this way the crawler is dynamically adapted based on the most significant documents seen so far [Sizov et al, 2002]. Two kinds of archetypes are considered: good authorities as determined by employing Kleinberg’s link analysis algorithm, and documents that have been automatically classified with high confidence using a linear SVM classifier [Sizov et al, 2002].

The BINGO! system consists of six main components:

- the crawler itself,
- an HTML document analyzer that produces a feature vector for each document,
- the SVM classifier with its training data,
- the feature selection as a "noise-reduction" filter for the classifier,
- the link analysis module for determining topic-specific authorities and hubs, and
- the training module for the classifier that is invoked for periodic re-training.

![Bingo! Architecture Diagram](image)

**Figure 2.13. Bingo! Architecture [Sizov et al, 2002]**

BINGO! is implemented as a collection of *Java servlets* that run under the control of a Web application server. All intermediate data, such as "raw documents" that the crawler fetches or feature vectors (vectors of term weights), and also the final ontology index are stored in a database using *Oracle8i*. This way the various components and their execution threads are largely decoupled with regard to timing and performance. For example, the classifier is not affected by temporary
delays in the crawler's threads as long as its input queue, a table in the database, is sufficiently filled.

2.6.1 Crawler

The initial seed for the crawler are the URLs of the documents that are referenced from the various folders of the user's bookmark file; these URLs are placed in the URL queue. The crawler then processes the links in the URL queue using multiple threads. It downloads new HTML documents and stores them in the local database which serves as a buffer for the subsequent stages of the analysis and classification pipeline. Once a crawled document has been classified, the BINGO! engine extracts all links from such a document and adds them to the URL queue for further crawling. BINGO! has several strategies for the prioritization of URLs to be visited; the simplest one of them would be depth-first traversal with a limit on the number and depth of URLs fetched from the same site.

Links from rejected documents (i.e., documents that did not pass the classification test for a given topic) are considered for further crawling, too; however, it is possible to restrict the depth of additionally traversed links from such documents to a value of two. The rationale behind this threshold is that one often has to "tunnel" through topic-unspecific welcome or table-of-content pages before again reaching a thematically relevant document. When a document is reached that passes the classification test, the limit for the allowed crawling depth along this path is dropped.

The BINGO! crawler has no global limits on crawling depth. Rather it uses the filling degrees of the ontology's various topics as a stopping criterion. When a topic holds a certain number of successfully classified documents (say 200), BINGO! suspends crawling. At this point, link analysis and re-training are performed for all topics, and then crawling is resumed.

2.6.2 Document Analyzer

Once a document has been fetched, it is stored in the database for further processing: an HTML document is parsed and analyzed to identify its outgoing hyperlinks, which are then added to the crawler's URL queue, and to produce a bag of words that occur in the document along with their frequencies. Very
common and thus insignificant words, so-called stop words such as articles ("the", "a", "an"), pronouns ("we", "you", etc.), and "universal" verbs and the auxiliary verbs ("be", "have", "take", "may", etc.) along with their flexions, are eliminated from the bag.

Each of the remaining words is reduced to its word stem using the Porter algorithm, thus mapping words with the same stem to the same feature. We refer to the resulting list of word stem frequencies in the document, normalized by dividing all values by the maximum frequency in the document, as the document's feature vector or vector of term frequencies (tf values). As an option, we can replace the tf values by tf*idf values where idf is the so-called inverse document frequency of a term, the inverse of the number of documents that contain the term. To avoid that idf values dominate the tf*idf product we use log10(idf) instead of idf for dampening. As idf values are a global measure and change with the progress of the crawl, they are re-computed lazily whenever a certain number of new documents have been crawled.

2.6.3 Feature Selection

The feature selection algorithm provides the BINGO! engine with the most characteristic features for a given topic; these are exactly the features that are used by the classifier for testing new documents [Sizov et al, 2002]. A good feature for this purpose discriminates competing topics from each other, i.e., topics that have the same parent in the ontology tree. Therefore, feature selection has to be topic-specific; it is invoked for every topic in the tree individually. Sizov et al [2002] consider for example an ontology with topics mathematics, agriculture, and arts, where mathematics has subcategories algebra and stochastics. Obviously, the term "theorem" is very characteristic for math documents and thus an excellent discriminator between mathematics, agriculture, and arts. However, it is of no use at all to discriminate algebra versus stochastics. A term such as "field", on the other hand, is a good indicator for the topic algebra when the only competing topic is stochastics; however, it is useless for a classifier that tests mathematics versus agriculture.
It is used also the Mutual Information (MI) measure for topic-specific feature selection. This technique, which is a specialized case of the notions of cross-entropy or Kullback-Leibler divergence [Sizov et al, 2002], is known as one of the most effective methods. Feature selection is a very important building block for a focused crawler.

2.6.4 Classifier

Document classification consists of a training phase for building a mathematical decision model based on intellectually preclassified documents, and a decision phase for classifying new, previously unseen, documents fetched by the crawler.

In the training phase, BINGO! builds a topic-specific classifier for each node of the ontology tree. Initially, the bookmarked documents of the topic serve as training data; these are periodically augmented by "archetypes" of the topic as the crawl proceeds. For non-leaf nodes of the ontology tree the training data is the union of the training data of all subtopics and the topic itself.

In the decision phase, BINGO! tests a new document against all topics in a top-down manner. Starting with the root, which corresponds to the union of the user's topics of interest, we invoke the classifiers for all topics with the same parent; we refer to these as "competing" topics as the document will eventually be placed in at most one of them. Each of the topic-specific classifiers returns a yes-or-no decision and also a measure of confidence for this decision. We assign the document to the tree node with the highest confidence in a positive decision. For example, if we used a Bayesian classifier, this would be the tree node with the highest likelihood that the document was "generated" from this topic given the features of the topic's training data. If none of the topics with the same parent returns yes, we place the document into a special tree node "others" under the same parent.

BINGO! uses support vector machines (SVM) as topic-specific classifiers. This method has been shown to be both efficient and very effective for text classification in general.

Sizov et al [2002] assert to use the linear form of SVM (i.e., with trivial kernel function) where training amounts to finding a hyperplane in the $m$-dimensional
feature vector space that separates a set of positive training examples for topic
from a set of negative examples (of all competing topics with the same parent)
with maximum margin.
BINGO! use an existing open-source SVM implementation that is part of the
BioJava package. The system also allows the user to interactively inspect and
possibly overwrite the classifier's decisions. So even if the classifier rejects a
document for a given topic, the user may intellectual assign it to this topic (and
may analogously drop documents that were accepted by the classifier); these
documents are then treated as if they were among the initial bookmarks (i.e.,
intellectually classified data with hundred percent confidence). This kind of
interactive user control is optional; BINGO! can also run in fully automated
mode, then relying on SVM classification only.

2.6.5 Link Analysis
The link structure between documents in each topic is an additional source of
information about how well they capture the topic. We apply Kleinberg's link
analysis method, coined HITS, to each topic of the ontology. This method aims to
identify a set of authorities, which should be Web pages with most significant
and/or comprehensive information on the topic, and a set of hubs, which should
be the best link collections with pointer to good authorities.
The algorithm considers a small part of the hyperlink-induced Web graph in the
order of a few hundred or a few thousand documents. The node set of the graph is
constructed in two steps:

1. we include all documents that have been positively classified into the topic
   under consideration, which form the "base set" in Kleinberg's terminology,
   and
2. we add all successors of these documents (i.e., documents that can be
   reached along one outgoing edge) and a reasonably sized subset of
   predecessors (i.e., documents that have a direct hyperlink to a document in
   the base set).
The predecessors are determined by asking a Web search engine that internally maintains a large fraction of the full Web graph and can thus answer "link: <...>" queries; the BINGO! system queries Google for this purpose. Unlike the set of successors, the set of predecessors of the base set can be extremely large (namely, when the base set already contains an excellent authority or hub that many personal home pages link to); so we artificially limit the number of added documents to 1000 by random selection.

Finally, the edge set for the entire set of nodes is constructed by fetching all documents and analysing their hyperlinks. The actual link analysis computes an authority score $x$ and a hub score $y$ for each document in the node set $S$, based on the following mutual reinforcement relationship:

$$x_q = \sum_{(p, q) \in E} y_p$$ $$y_p = \sum_{(q, p) \in E} x_q$$

The intuition behind these mutually recursive definitions is that a good hub is a page that points to many good authorities and a good authority is a page that is pointed to by many good hubs. The recursion in the above formulas is resolved by computing the $x$ and $y$ vectors in an iterative manner: we start with assigning initial hub and authority scores uniformly, and then compute $x(i)$ and $y(i)$ from the previous iteration's values $x(i-1)$ and $y(i-1)$; after each iteration the two vectors are normalized such that they have length (i.e., $L_2$ norm) one.

Since we are not really interested in computing the absolute values of authority and hub scores of all documents in $S$ but merely want to identify the top $N$ authorities and hubs (with $N$ in the order of 100), we interpret the notion of convergence in a somewhat loose manner and simply stop after a fixed number of iterations (the current setup of BINGO! uses 30 iterations).

2.6.6 Re-Training based on Archetypes

Whenever the filling degree of one topic exceeds a pre-specified level for example, when the topic is populated with a few hundred positively classified documents, the re-training procedure is invoked. For the sake of simplicity, we start re-training for all topics at this point, although we could invoke it for individual topics only, too.
To ensure that we have indeed made sufficient crawling progress for all topics, we additionally require that all nodes of the ontology tree have to hold at least $X$ positively classified documents. If this latter condition is not met, we can give higher crawling priority to this topic by prioritizing hyperlinks in the URL queue that have been extracted or transitively reached from positive documents of the given topic.

The purpose of the re-training procedure is to identify new training documents that promise to be better than the original ones taken from the bookmark file. Here better means more characteristic in the sense that the features of the new training data capture the topic-specific terminology and concepts and are more discriminative with regard to competing topics. It is obvious that we should consider asking the human user about suggestions for characteristic documents, and we do indeed support such human interaction as an option. At the same time, however, we should strive for providing automated support as well for scalability and versatility. Our approach thus is to identify the most characteristic "archetypes" among the documents that have been positively classified into a given topic. We aim to find at least as many good archetypes as the topic initially had bookmarks; hopefully we can identify an order of magnitude more training documents of very high relevance.

**BINGO!** draws on the following two sources of potential archetypes:

1. The link analysis provides us with good authorities for the given topic. We simply use the current set of positively classified documents as the input for Kleinberg's HITS algorithm. The result of the analysis is a ranked list of documents in descending order of authority scores; the top of these documents are considered as archetypes. The HITS algorithm also provides us with a ranked list of hubs, from which we take the top candidates for the crawl frontier; these will be placed into the URL queue of the crawler, together with the hubs for all other topics, when the re-training procedure is completed.

2. We exploit the fact that the SVM classifier yields a measure of its confidence about a positive classification, namely, the distance of the
document's feature vector from the separating hyperplane. This way we can sort the documents of a topic in descending order of confidence. We then select the top documents as archetypes.

Once the archetypes of a topic have been selected, the classifier for that topic is re-trained using the archetypes plus the original bookmarks as training data. This step in turn requires invoking the feature selection first. So the effect of re-training is twofold:

1. if the archetypes capture the terminology of the topic better than the original bookmarks (which is our basic premise) then the feature selection procedure can extract better, more discriminative, features for driving the classifier, and
2. the accuracy of the classifier's test whether a new, previously unseen, document belongs to a topic or not is improved using richer (e.g., longer but concise) and more characteristic training documents for building its decision model.

2.7 Algorithms application: Arachnid

The ARACHNID algorithm is presented in Menczer and Belew [1998]. Arachnid is a Java-based web spider framework. It includes a simple HTML parser object that parses an input stream containing HTML content. Simple Web spiders can be created by sub-classing Arachnid and adding a few lines of code called after each page of a Web site is parsed.

The user initially provides URL of a document. The population is initialized by pre-fetching the starting document. Each agent given a random behavior (depending on the representation) and an initial reservoir of energy. In step (0) each agent senses its local neighborhood by analyzing the text of the document where it is currently situated. This way, the relevance of all neighboring documents pointed to by the hyperlinks in the current document is estimated. Based on these link relevance estimates, an agent “moves” by choosing and following one of the links from the current document. In step (1) the agents
Energy is updated. Energy is needed in order to survive and continue to visit documents on behalf of the user.

Agents are charged costs for the use of network resources incurred by transferring documents. Agents are also rewarded with energy if they visited documents appear to be relevant. That is the role of the $r$ function which an agent uses to estimate the relevance of documents. Steps (2) and (3) of the algorithm embody the principle of conservation of resources. Agents visiting marked documents do not get energy from them (unless by users feedback). This mechanism is implemented via a cache, which also speeds up the process by minimizing duplicate transfers of documents. Caching documents is a form of communication, and thus a bottleneck for the parallelism of distributed algorithms. However, since network communication is the most expensive resource for the algorithm, the performance improvement warranted by the use of a cache should outweigh any such degradation. Instantaneous changes of energy can be used, in step (4) as reinforcement signals. This way agents can adapt during their lifetime, if their representation allows for it. An agent can modify its behavior based on prior experience, by learning to predict the best links to follow.

In step (5) an agent may be killed or be selected for reproduction. In the latter case the offspring is situated at the same document location as the parent, and mutated to provide evolution with the necessary variation. Evolutionary adaptation differs from learning in two important respects. First, it does not change the reproducing agents, but biases the population towards successful individuals. Second, since selection is based on energy accumulated over many steps, evolution averages out short-term energy fluctuations and can thus rely on better assessments of agent behaviors. The output of the algorithm is the flux of documents visited by the population.

In follow chapter we describe the use of Arachnid for the purpose of this work and some piece of java code that represent the way in which we implemented it.
2.8 Algorithms application: SSSpider

SSSpider™ (http://www.kryltech.com/spdownld.htm) is a personal second-generation Intelligent World Wide Web (WWW) agent that automates the retrieval and selection of useful information available from the Internet. SSSpider selects data based on questions entered in any language. SSSpider can operate in the background and when the search is complete, creates a report of its findings which can be viewed using either a default or an internal built-in browser.

SSSpider performs full-text search in several following steps.

**Step 1.** We formulate a query through different options:

- The literal search method *"All words in the same order match case"* makes distinction between capital (uppercase) and small (lowercase) characters. See also: "All words in same order ignore case".

- The literal search method *"Entire pattern match case"* selects documents which contain your phrase bodily (as you type it in) without any changes. See also, "Entire pattern ignore case" described above.

- The literal search method *"All words from the phrase match case"* is identical to the method "All words from the phrase ignore case" described above except for the fact that it makes distinction between capital and small characters.

- The literal search method *"At least one word match case"* is also similar to "At least one word ignore case", but it treats capital and small letters as absolutely different ones and does not match them to each other (even if they represent the same sound in the language of your choice).

When we have entered one or a number of Search Phrases in the Query screen, launch our queries by dragging the corresponding query icons and then dropping them onto a desired Search Set. SSSpider starts processing our queries and sends a simultaneous request to up to 20 search engines for them to find matches to query number one.

**Step 2.** Once the search engines have provided a list of locations that could contain information that match or are related to our Search Phrase, SSSpider then begins to contact the sites. SSSpider uses the existing engines (such as Alta Vista,
Google, Yahoo, etc.) to do the first level navigation within the Web and then having received this list, "drills down" to actually look into each site to see what more it can find. Since not all engines reflect the most recent changes to these sites, it’s not possible for the search engines to provide us with matched quotations extracted from the original documents, all we get is their titles and a few key words. What SSSpider has been doing up to now is rather special, but the following steps are unique to this application and where the technology starts to take effect.

**Step 3.** SSSpider having contacted the list of search engines, downloads the required number of documents mentioned in their reports directly from each location. In doing so, SSSpider obtains the most current information. Sites that are "down" are automatically bypassed, as are those that even though the Search Engine indicated a match was possible, SSSpider still didn’t find anything that was a close enough match. All this is being carried out automatically with nothing required of us. If we have performed searches using the popular search engines, we will quickly appreciate the time needed to manually search 20 sites using one search engine not 20.

**Step 4.** SSSpider parses each document using an intelligent text search engine embedded in the program. This text search engine matches text in each document with the Search Phrase you launched and selects from the document a quotation or an extract that includes matching text for the search phrase. These quotations are then sorted by relevancy (best matches on top), and SSSpider then creates a report of what it has found.

**Step 5.** The final step is for SSSpider to place the report into your library, and if user preferences are set up to do so, your default Internet browser will be loaded and the newly created query report will be displayed. SSSpider repeats the above steps until all the queries we launched have been completed. In this way we can launch multiple queries and as SSSpider works in the background continuing writing that important letter, attend a meeting or take a well deserved coffee break. The choice is yours, SSSpider can be set up to work as you want it to, limit the time for the query to be processed, restrict the number of documents to be
downloaded, create our own lists of search sites to visit; all these items are defaults that are user definable.

2.9 OWL

OWL [Dean et al, 2003] is an ontology language for the Semantic Web, developed by the World Wide Web Consortium (W3C) Web Ontology Working Group (http://www.w3.org/). OWL was primarily designed to represent information about categories of objects and how objects are interrelated.

In the context of the Semantic Web, ontologies are expected to play an important role in helping automated processes (so called “intelligent agents”) to access information. In particular, ontologies are expected to be used to provide structured vocabularies that explicate the relationships between different terms, allowing intelligent agents (and humans) to interpret their meaning flexibly yet unambiguously.

Terms whose meanings are defined in ontologies can be used in semantic mark-up that describes the content and functionality of web accessible resources [Tim Berners–Lee, 1999]

The lack of proper layering between RDFS and the less expressive species of OWL and the lack of proper layering between OWL DL and OWL Lite on the one side and OWL Full on the other, raises doubts about interoperability between ontologies written in these different languages.

Some of the most important influences on the design of OWL came from Description Logics, from the frames paradigm, and from RDF. In particular, the formal specification of the language was influenced by Description Logics, the surface structure of the language was influenced by the frames paradigm, and the RDF/XML exchange syntax was influenced by a requirement for upwards compatibility with RDF.

2.9.1 DL

Description Logics are a family of class-based (concept-based) knowledge representation formalisms. They are characterised by the use of various constructors to build complex classes from simpler ones, an emphasis on the
decidability of key reasoning problems, and by the provision of sound, complete and (empirically) tractable reasoning services [Dean et al, 2003]. Description Logics, and insights from Description Logic research, had a strong influence on the design of OWL, particularly on the formalisation of the semantics, the choice of language constructors, and the integration of datatypes and data values. In fact OWL DL and OWL Lite can be viewed as expressive Description Logics, with an ontology being equivalent to a Description Logic knowledge base.

A Description Logic knowledge base consists of two parts, namely the TBox and the ABox. The TBox consists of a number of class and property axioms; the ABox consists of a number of individual assertions.

Description Logics make the distinction between abstract and concrete properties, based on whether the range of the property is in the abstract or the concrete domain. OWL DL rejects this distinction by distinguishing object properties and datatype properties, where an object property may only have a description as its range and a datatype property may only have a datatype as its range; descriptions and datatypes are disjoint.

There exist several implementations for reasoning with Description Logics which implement different reasoning tasks in Description Logic languages (e.g. Racer Pro that will be discussed in section 2.11). Two important reasoning tasks in Description Logics are subsumption checking and checking class membership. Subsumption checking amounts to checking whether one class is a subclass of another concept, i.e. checking whether one concept is more specific than another concept. The class membership inference is used to check whether an individual is a member of a specific class.

**Semantics**

A key feature of Description Logics is that they are logics, i.e., formal languages with well defined semantics. The standard technique for specifying the meaning of a Description Logic is via a model theoretic semantics, whose purpose is to explicate the relationship between the language syntax and the intended model of the domain. A model consists of a domain and an interpretation function, where the domain is a set of objects and the interpretation function is a mapping from
individual, class and property names to elements of the domain, subsets of the
domain and binary relations on the domain, respectively.

A Description Logic knowledge base consists of a set of axioms asserting, e.g.,
that one class is a subclass of another, or that an individual is an instance of a
particular class. The meaning of these axioms is given by corresponding
constraints on models.

The meaning of a knowledge base derives from features and relationships that are
common to all possible models. If, for example, the interpretation of a class must
always be the empty set, then that class is said to be inconsistent, while if there are
no possible interpretations, the knowledge base itself is said to be inconsistent. If
the relationship specified by a given axiom must hold in all interpretations of a
knowledge base, then that axiom is said to be entailed by the knowledge base, and
if one knowledge base entails every axiom in another knowledge base, then the
first knowledge base is said to entail the second knowledge base.

OWL uses a Description Logic style model theory to formalise the meaning of the
language. This was recognised as an essential feature in all three languages, as it
allows ontologies, and information using vocabulary defined by ontologies, to be
shared and exchanged without disputes as to precise meaning. The need for this
kind of formality was reinforced by experience with early versions of the RDF
and RDFS specification, where a lack of formality soon led to arguments as to the
meaning of language constructs such as domain and range constraints [Lucchese,
2003]. Another advantage of formalising the meaning of the language in this way
is that automated reasoning techniques can be used to check the consistency of
classes and ontologies, and to check entailment relationships. This is crucial if the
full power of ontologies is to be exploited by intelligent agents, and the ability to
provide such reasoning support was a key design goal for OWL.

✔ Computational Problems

One aspect of OWL that distinguishes it from RDF and RDFS is that it supports a
rich set of inferences. Inferences supported by OWL, however, are quite complex,
requiring, e.g., reasoning by cases and following chains of properties.
Taking all the representational desires for OWL together would have created a formalism where key inference problems were undecidable. For example, allowing relationships to be asserted between property chains (such as saying that an uncle is precisely a parent’s brother) would make OWL entailment undecidable. In addition, some aspects of RDF, such as the use of classes as instances, interact with the rest of OWL to create computational difficulties, taking OWL beyond the range of practical algorithms and implemented reasoning systems. OWL has thus had to provide a solution for these computational issues while still retaining upwards compatibility with RDF and RDFS.

2.10 Protégé

Protégé (http://protege.stanford.edu) is an open-source tool for editing and managing ontologies. It is the most widely used domain-independent, freely available, platform-independent technology for developing and managing terminologies, ontologies, and knowledge bases in a broad range of application domains. Protégé is implemented in Java, and runs on a broad range of hardware platforms, including Windows, MacOS, Linux, and Unix. Protégé’s core functionality can be extended in many ways by creating plug-ins. There are over 90 Protégé plug-ins providing advanced capabilities such as reasoning and inference support and visualization of large ontologies. Protégé adopts an object way to represent concepts and links. Therefore, every concept stands for a class with a clear definition of the concept it is representing. In addition to that definition, particular attributes have to be added by the user. Those attributes are called slots and constitute the links between concepts. Thus, when a concept A is linked with another concept B, a slot is defined in the slot A which type is the class representing concept B. The name of the slot describes the link itself and a clear definition must be added. Additional features of slots exist as the cardinality or the fact that the link is an inverse one. Finally, making an ontology in protégé consists in making a class hierarchy which classes are linked.
Protégé has been used as the primary development environment for many ontologies in the life sciences and quickly become the most widely used tool in the world for use with the Ontology Web Language (OWL).

The Protégé platform supports two main ways of modelling ontologies and we chose to use the second approach:

- The Protégé-Frames editor enables users to build and populate ontologies that are frame-based, in accordance with the Open Knowledge Base Connectivity protocol (OKBC). In this model, an ontology consists of a set of classes organized in a subsumption hierarchy to represent a domain's salient concepts, a set of slots associated to classes to describe their properties and relationships, and a set of instances of those classes - individual exemplars of the concepts that hold specific values for their properties.

- The Protégé-OWL editor enables users to build ontologies for the Semantic Web, in particular in the W3C's Web Ontology Language (OWL). "An OWL ontology may include descriptions of classes, properties and their instances. Given such an ontology, the OWL formal semantics specifies how to derive its logical consequences, i.e. facts not literally present in the ontology, but entailed by the semantics. These entailments may be based on a single document or multiple distributed documents that have been combined using defined OWL mechanisms" (OWL Web Ontology Language Guide)

The basic Protégé infrastructure provides the following key features which makes it particularly suitable to the needs of developing and using ontologies in a variety of applications:

- An extensible knowledge model. The internal representational primitives in Protégé can be redefined declaratively, permitting Protégé to have representations appropriate for a variety of ontology languages. Protégé’s representational primitives - the elements of its knowledge model - provide classes, instances of these classes, slot representing attributes of classes and instances, and facets expressing additional information about slots.
• A customizable user interface. The standard Protégé user interface components for displaying and acquiring data can be replaced with new components that fit particular types of ontologies best (e.g., for OWL).

• Ability to import ontologies in different formats. There are several plug-ins available for importing ontologies in different formats into Protégé, including XML, RDF, and OWL.

• Support for data entry. Protégé provides facilities whereby the system can automatically generate data entry forms for acquiring instances of the concepts defined by the source ontology.

• Ontology authoring and management tools. The PROMPT tools are plug-ins to protégé that allow developers to merge ontologies, to track changes in ontologies over time, and to create views of ontologies. The Protégé internal knowledge representation can be translated into the various representations used in the different ontologies. Protégé has different back end storage mechanisms, including relational database, XML, and flat file.

• An extensible architecture that enables integration with other applications. Protégé can be connected directly to external programs in order to use its ontologies in intelligent applications, such as reasoning and classification services.

• A Java Application Programming Interface (API). System developers can use the Protégé API to access and programmatically manipulate Protégé ontologies. Protégé can be run as a stand-alone application or through a Protégé client in communication with a remote server. Jambalaya is a Protégé plug-in that provides an extensible, flexible, and scalable visualization environment for exploring, navigating, and understanding ontologies. Classes and instances are represented as nodes in a graph; different types may be distinguished using different colour hues. Directed edges (arcs) are used to show relationships between concepts and instances.

Protégé can be a useful tool for managing and using ontologies. Its graphical user interface components can help users visualize and browse large and complex ontologies. Protégé can help them align and compare ontologies that overlap in content as well as different versions of ontologies. The Protégé platform supports
Web-based ontology viewing and collaboration and it provides different back-end storage mechanisms.

2.10.1 OWL Plugin

The OWL Plugin is a complex Protégé extension that can be used to edit OWL files and databases. The OWL Plugin includes a collection of custom-tailored tabs and widgets for OWL, and provides access to OWL-related services such as classification, consistency checking, and ontology testing.

✓ Editing OWL Ontologies

In this section we will walk through some of the forms and tabs for editing classes and properties. A screenshot of the OWL classes tab is shown in Figure 2.14. The main class hierarchy is shown on the left, and the details of the currently selected class are shown in a form on the right. The upper section of the class form displays class metadata such as names and annotations. Annotation properties are ignored by OWL DL reasoners. Instead, they are very suitable to manage metadata about a class, such as versioning information, comments, relationships to other external resources, and labels in multiple languages.

![Figure 2.14. OWL Classes](image-url)
✓ **Editing Class Descriptions**

Traditional Protégé users are accustomed to an object-centred view to the interface that has required some effort to adapt to OWL. In the Protégé metamodel, classes are typically only related through simple superclass/subclass relationships, and therefore a simple tree view was enough to edit classes. OWL on the other hand not only distinguishes between necessary conditions (superclasses) and necessary and sufficient conditions (equivalent classes), but furthermore allows users to relate classes with arbitrary class expressions. The OWL Plugin’s class editor addresses this complexity by means of a list of conditions, which is organized into blocks of necessary & sufficient, necessary, and inherited conditions (see figure 2.15). Each of the necessary & sufficient blocks represents a single equivalent intersection class, and only those inherited conditions are listed that have not been further restricted higher up in the hierarchy.

In addition to the list of conditions, there is also a custom-tailored widget for entering disjoint classes, which has special support for typical design patterns such as making all siblings disjoint. This rather object-centred design of the OWL classes tab makes it possible to maintain the whole class definition on a single screen.

![Figure 2.15. Asserted conditions block](image)

---

54
✓ **Editing Properties**

The class form provides a listing of all properties for which instances of the selected class can take values. This includes those properties which have the class in their domain, and those that don’t have any domain restrictions. The details of the properties are edited by means of a separate form. Similar to the class form, the upper part of the property form displays name and annotation properties. The lower part contains widgets for the property’s domain, range, and characteristics such as whether a property is transitive or symmetric.

Note that Protégé and OWL support user-defined metaclasses that extend the standard system classes. For example, ontologies can define a subclass of owl:Object-Property and then add other properties to it.

✓ **Reasoning based on Description Logics**

The OWL Plugin provides direct access to DL reasoners such as Racer (we describe this tool in section 2.11). The current user interface supports two types of DL reasoning: Consistency checking and classification (subsumption).

*Consistency checking* (i.e., the test whether a class could have instances) can be invoked either for all classes with a single mouse click, or for selected classes only. Inconsistent classes are marked with a red bordered icon.

*Classification* (i.e., inferring a new subsumption tree from the asserted definitions) can be invoked with the classify button on a one-shot basis. When the classify button is pressed, the system determines the OWL species, because some reasoners are unable to handle OWL Full ontologies. This is done using the validation service from the Jena library. If the ontology is in OWL Full (e.g., because metaclasses are used) the system attempts to convert the ontology temporarily into OWL DL. The OWL Plugin supports editing some features of OWL Full (e.g., assigning ranges to annotation properties, and creating metaclasses). These are easily detected and can be removed before the data are sent to the classifier. Once the ontology has been converted into OWL DL, a full consistency check is performed, because inconsistent classes cannot be classified correctly.
Finally, the classification results are stored until the next invocation of the classifier, and can be browsed separately. Classification can be invoked either for the whole ontology, or for selected subtrees only. In the latter case, the transitive closure of all accessible classes is sent to the classifier. This may return an incomplete classification because it does not take incoming edges into account, but in many cases it provides a reasonable approximation without having to process the whole ontology. OWL files store only the subsumptions that have been asserted by the user. However, experience has shown that, in order to edit and correct their ontologies, users need to distinguish between what they have asserted and what the classifier has inferred. Many users may find it more natural to navigate the inferred hierarchy, because it displays the semantically correct position of all the classes.

The OWL Plugin addresses this need by displaying both hierarchies and making available extensive information on the inferences made during classification. After classification the OWL Plugin displays an inferred classification hierarchy beside the original asserted hierarchy. The classes that have changed their superclasses are highlighted in blue, and moving the mouse over them explains the changes. Furthermore, a complete list of all changes suggested by the classifier is shown in the upper right area, similar to a list of compiler messages. A click on an entry navigates to the affected class. Also, the conditions widget can be switched between asserted and inferred conditions. All this allows the users to analyse the changes quickly.

2.11 Racer Pro
RacerPro stands for Renamed ABox and Concept Expression Reasoner Professional. As the name suggests, the origins of RacerPro are within the area of description logics. Since description logics provide the foundation of international approaches to standardize ontology languages in the context of the so-called semantic web, RacerPro can also be used as a system for managing semantic web ontologies based on OWL, it can be used as a reasoning engine for ontology editors such as Protégé.

An important aspect of this system is the ability to process OWL documents.
Ontologies may be taken off the shelf or may be extended for domain-specific purposes (domain-specific ontologies extend core ontologies). For doing this, a reasoning system is required as part of the ontology editing system. RacerPro can process OWL Lite as well as OWL DL documents (knowledge bases).

The following services are provided for OWL ontologies and RDF data descriptions:

- Check the consistency of an OWL ontology and a set of data descriptions. Find implicit subclass relationships induced by the declaration in the ontology.
- Find synonyms for resources (either classes or instance names).
- Since extensional information from OWL documents (OWL instances and their interrelationships) needs to be queried for client applications, an OWL-QL query processing system is available as an open-source project for RacerPro.
- HTTP client for retrieving imported resources from the web. Multiple resources can be imported into one ontology.

Incremental query answering for information retrieval tasks (retrieve the next n results of a query). In addition, RacerPro supports the adaptive use of computational resource: answers which require few computational resources are delivered first, and user applications can decide whether computing all answers is worth the effort.
Figure 2.16. Connection ontology with RacerPro
Chapter 3

The energy domain case study

In order to construct a domain ontology, there is a need for domain knowledge. In the area of Energy Security we had the assistance and guidance of a domain expert, so this domain was selected for demonstrating the approach of knowledge guided information search. Database lease note that he construction of the energy security ontology was teamwork, involved several students. It is presented here as a whole, in order to put the specific research work, focused crawling, in the larger context.

3.1 Introduction

Energy security has risen to the top of the agenda among policy makers, international organizations and business. A sustained growth in demand for energy has led to serious concerns over the long-term availability of reliable and affordable supplies. Conflict and geopolitical tensions in some of the regions that are key sources of raw materials for energy production represent a short-term supply risk, as well as obstacles for much-needed investment in the sector. Transportation bottlenecks and other constraints in energy infrastructure represent additional threats to a reliable and affordable supply of energy.

Large amounts of information and knowledge is of ever increasing importance in field as this and search of the right information, especially in corporate intranets and knowledge bases, becomes an increasingly difficult task [Menczer et al, 2004].

Over the coming decades, the world faces a daunting challenge in meeting the energy needs of a growing and developing world population while mitigating the impacts of global climate change. Until recently, energy security and climate change were considered separate issues to be dealt with by different communities of experts and policymakers. The two issues are now converging, challenging the security and climate communities to develop a better understanding of how to deal with both issues simultaneously. Energy security and climate change have broad
economic, political, and societal consequences. A lack of energy security can exacerbate geopolitical tensions and impede development. The impacts of climate change and efforts to address the problem will have numerous security implications, including water scarcity, crop decline, forced migration, and changing conflict dynamics. The strict relationship underlying these two concepts are more problematic and so there is the need to systematize the knowledge about semantic information. There is a need to share meaning of terms in these domains and so, the need of a knowledge representation tool for domain knowledge like an ontology appears. This is especially relevant since today’s search tools perform rather poorly, in the sense that information access is mostly based on keyword searching or even mere browsing of topic areas. This unfocused approach often leads to undesired results. Focused search, based on domain ontology may provide an important tool for policy makers that need relevant information. This is true in general and specifically in the Energy Security domain that was selected as a case study.

Building an ontology schema is tedious and effort-intensive task, especially to find information source containing the information for the instances of the concepts. This task results more difficult if we think at the insertion of instances into the ontology from unstructured texts as web pages.

In order to initiate the generation of the Energy Security ontology, we started by a search of general documents on Google. We collected roughly 100 documents. Next we extracted manually from this first database the necessary information to classify the concepts in the ontology to add instances for these concepts and create relations among concepts and between concepts and instances. For example we created the relationship between the class of “energy security” and classes like “reliability of supply”, “affordability” and “friendliness to the environment” because energy security should be viewed as comprised of three subconcepts. To implement energy security ontology we used OWL. Specifically we adapt the OWL DL sublanguage of OWL in order to take advantage of its consistency and satisfiability checking the mechanisms and its reasoning and inference capabilities. We also used the Owl Protégé’s plugin to support ontology editing in OWL. The following sections describe the construction of the Energy Security
ontology using these tools, how this ontology has guided us in the use of different crawler specifically in the task of query formulation and the mechanism that we used to estimate the relevance of each retrieved page.

3.2 Energy Security ontology description

The first query on Google about “energy security” renders 27,300,000 links.

We decided to search in the web some information to acquire knowledge to a first approach to this domain. We used the results in Google of first 20 pages. We extracted about 100 documents into a MySql database and analysed some of them (we also used some internal links of the documents) to extract general knowledge about the domain so we can formalize some concepts and instances in the ontology. We connected our ontology, defined using Protégé, with the MySql database as demonstrated by figure 3.1. We organize every document in a database MySql (we created a connection with the database MySql as see in figure 3.1.) in which every table corresponds at the namesake class and we used four fields to represent the main information of each document: title, address, keywords and topic at which it refers (that is a class in the ontology).

Figure 3.1 Connection of database with ontology.
We created some horizontal links between the subclasses and between the individuals of the hierarchy. To create the horizontal links we have used logical properties with one or two arguments. Later we will show the classes which we have selected to create our ontology.

In the second step we constructed a hierarchy of concept classes that represent the energy domain in relation to energy security problems. We created a super-class named Energy Domain in which there are 6 sub-classes (as seen in figure 3.2) Country, Energy Security, Energy Sources, Infrastructures, Market, Environmental consequences.

![Figure 3.2. OWL Viz](image)

At the third step we inserted detailed knowledge to concepts (classes), using instances and relationships (properties).

The class *Energy Security* has two sub-classes: *Risks* and *Solutions*.

Energy security risks are different: someone report to energy market instabilities caused by unforeseen changes in geopolitical or other external factors, or compounded by fossil fuel resource concentration; other one are technical failures such as power “outages” (blackouts and brownouts) caused by grid or generation plant malfunction and there are also physical security threats such as terrorists, sabotage, theft or piracy, as well as natural disasters (earthquakes, hurricanes, volcanic eruptions, the effects of climate change etc.). So we have decided to create six instances for *Risks* class as see in figure 6: *MarketInstabilities, NaturalDisasters, Piracy, Sabotage, TechnicalFailures* and *Terrorism*. 
Instead for *Solutions* we collect different information from the analysed documents that allow only to assert that the concept of solution is linked only to the governments action to avoid problems about supplying of energy. Moreover there are several kinds of government actions addressing energy security but they aren’t arranged in a formal taxonomy. So we let open this argument for next step of the work.

**Figure 3.3. Class Browser and Instance Browser for “Risks” class**

All forms of energy are stored in different ways. There are two groups of energy sources, two classes respectively: *Non Renewable* and *Renewable*. Renewable and non renewable energy sources can be used to produce secondary energy sources like electricity.

We create a sub-class of *Energy Sources*, called *Fossil Fuel* which are considered non renewable because they cannot be replenished (made again) in a short period of time.

The instances of *FossilFuel* are Coal, Oil and Natural gas which are all considered fossil fuels because they formed from the buried remains of plants and animals that lived millions of years ago.

The five renewable sources used most often are: *biomass*, *water*, *geothermal*, *wind* and *solar* and these are some of the instances.
Another important class that results closely correlated to Energy Security is the concept of *Climate change*. It refers to any significant change in measures of climate (such as temperature, precipitation, or wind) lasting for an extended period (decades or longer).

Every type of energy and every energy source must be transported in various places. So it is necessary to represent the concept of *Infrastructure*. There are different means of transport and many instances to add at this class: pipeline, oil tanker, ship, train and there are other means to production of the energy source like turbine for wind power.

Moreover energy is used for different sector and in every field there are various information to represent. We identify four instances to describe the *Market* class: industrial sector, residential and commercial sector and transportation.

We have inserted the *Country* class because the country energy politics (and the country energy availability of the energy sources) are important aspects of the
energy domain. Furthermore, in this manner, we can create horizontal links with other classes of the ontology (e.g. Oil is mostly provided by the country X).

3.3 Experimental set up

The first version of ontology about Energy Security domain, described above, is composed by 16 classes, 4 properties and 40 instances. The ontology is partial but the information contained is useful to our next step: we investigate how to take advantage of this information and use it to guide a spidering process.

So, at this point of the work we need the use of web crawlers which have the task of collecting web pages (as we described in Chapter 2) and the Information Retrieval system which as the task of retrieving text documents that answer our query.

Our approach is using the knowledge formalized in the ontology to guide query formulation and so we use focused crawlers which start with a query. The crawlers we used were MySpiders with its relative algorithms and SSSpider.

Moreover we needed a mechanism to estimate the relevance of a page. A common approach for this is to assess the similarity between the retrieved page and a gold standard that represents a relevant page. In a first approach our gold standard is represented by documents in MySql database that are relevant because we have read them and they have driven the ontology concepts formalization. In other approach we use general framework for crawler evaluation research defined by Menczer et al 2004.

The output of a crawler $l$ for a given topic can be written as a temporal sequence $S_l = \{u_1, u_2, \ldots, u_M\}$ where $u_i$ is the URL for the $i_{th}$ page crawled and $M$ is the maximum number of pages crawled. In order to measure the performance of the crawler $l$ we need a function $f$ that maps the sequence $S_l$ to a sequence $R_l = \{r_1, r_2, \ldots, r_M\}$ where $r_i$ is a scalar that represents the relevance of the $i_{th}$ page crawled.

Once we have a sequence $R_l$ we may summarize the results at various points during the crawl. However, evaluation of topical crawlers is a challenging problem due to the unavailability of the complete relevance set for any topic or query. Also, manual relevance judgment for each of the crawled pages (manually
created function $f$ ) is extremely costly in terms of man-hours when we have millions of crawled pages spread over more than one hundred topics. Even if we sample for manual evaluation, we will have to do so at various points during a crawl to capture the temporal nature of crawl performance. So, we use the two standard information retrieval (IR) measures recall and precision.

The recall of the system is the proportion of relevant material actually retrieved in answer to a search request and the precision of the system is the proportion of retrieved material that is actually relevant. Recall and precision measure is known as the effectiveness of the retrieval system. In other words it is a measure of the ability of the system to retrieve relevant documents while at the same time holding back non-relevant one. It is assumed that the more effective the system the more it will satisfy the user. It is also assumed that precision and recall are sufficient for the measurement of effectiveness.

Before proceeding to the technical details relating to the measurement of effectiveness it is as well to explain the concept of relevance in our study. Relevance is a subjective notion. Different users may differ about the relevance or non-relevance of particular documents to given questions. So in order to be able to calculate recall as detailed below, the technical relevance of all documents in the dataset must be assessed. In this study, a document is called technically relevant for a given search term if it contains this term in the full-text and this check was done by the teamwork of this work.

The work presented here aims to evaluate the quality of the obtained results from different topical crawlers and thus to evaluate the validity and the usefulness of spidering process. Recall and precision were estimated to measure the performance of these applications.

Recall and precision are two accepted measurements to determine the utility of an information retrieval system or search strategy. They are defined as:

\[
\text{recall} = \frac{\text{number of documents retrieved and relevant}}{\text{number of relevant documents in database}}
\]

\[
\text{precision} = \frac{\text{number of documents retrieved and relevant}}{\text{number of relevant documents retrieved}}
\]
Despite the relevance evaluation from teamwork, it is necessary to know the total number of the relevant documents in a database for each query in order to estimate the recall. In the present study, a particular document in database was defined as technically relevant if the words contained in the query could be found in its full-text article.

So when a query is done we revised the recall and the precision that are:

\[
\text{recall} = \frac{\text{number of relevant documents retrieved by crawler}}{\text{number of relevant documents in database}}
\]

\[
\text{precision} = \frac{\text{number of relevant documents retrieved by a crawler}}{\text{number of documents retrieved by a crawler}}
\]

For our task, that is ontology population through information guided spidering, another important step must be described. We decided, with the help of a domain expert, Dr. Brenda Shaffer\(^1\) to use knowledge that guided the ontology building and to focus our work on topic Energy Security.

So we have been guided by the deep meaning of energy security thanks to her words:

“Energy security should be viewed as comprised of three elements: reliability of supply, affordability (supports economic growth) and friendliness to the environment. Energy security is achieved by states through: diversification of sources (different suppliers) and diversification of fuel sources. In addition, additional techniques include: energy storage and strategic petroleum reserves; redundant energy infrastructure and flexibility to shift fuels in delivering energy functions (such as producing electricity, powering mass transportation).”

We consider the concept of “security” in its broadest sense: it includes all actions aimed at physical protection of energy and transport facilities and infrastructures but

\(^1\) Dr. Brenda Shaffer: School Of Political Science, Asian Studies Department of University of Haifa
it comprises too oil and other fossil fuel depletion (peak oil, etc), reliance on foreign sources of energy, geopolitics (such as supporting dictatorships, rising terrorism, “stability” of nations that supply energy) energy needs of poorer countries, and demands from advancing developing countries such as China and India, environmental issues, in particular climate change and renewable and other alternative energy sources.

Another meaningful guideline of the domain expert is that “renewable energy sources also have environmental consequences (albeit much lower than fossil fuel use): impacts weather, bird flight patterns, large land use, and damage to views” while we initially consider them only as alternative use to fossil fuels.

We defined queries using the operator AND among two or more terms. Table 3.1 shows all queries formulated in each crawler.

|---|---------------------------------------|---|----------------------------------------|---|---------------------------------------|---|-------------------------------------|---|--------------------------------------|---|----------------------------------------|---|----------------------------------|---|---------------------------------|---|---------------------------------|---|---------------------------------|---|---------------------------------|---|---------------------------------|---|---------------------------------|---|---------------------------------|---|---------------------------------|
Table 3.1. List of queries

We used these queries in two applications described in chapter 2, MySpiders and SSSpider.

We note that SSspider has one only way to run while we can select the algorithm to use before to run MySpiders. One algorithm is Naïve best first which has a link selection process that is guided by simply computing the lexical similarity between the topic’s keyword and the source page for the link. The other is Infospiders which proceeds with the activation of various agents, everyone of which has a list of keywords (initialised with the query) and a neural net used to evaluate new links: each input unit of this net receives a count of frequency with which the keyword occurs in the vicinity of each link to be traversed, weighted to give more importance to keywords near the link (and maximum in the anchor text).

After we done all queries in each crawler we evaluate the results with the documents contained in the database. Precisely, with the help of the SQL syntax, we select documents concerning the topic of the query, then Energy Security and we consider for each query the documents that contain in the field keywords the words of the query.

Figure 3.6. Selection of documents for Energy Security topic
Figure 3.9. Spider details windows
In this figure 3.9 we can see an example of the behaviour of two infospider agents. In each spider detail there is information about the status of the agent (we see always dead, when compare “search over”, because we open the spider detail window when the spider has finished to crawl), the term that adds at the query terms and the URL that renders in the interface of the results.

Figure 3.9. Results of the different queries to Energy Security concept in MySpiders with Infospiders algorithm
We used another approach to estimate retrieved pages, that was a general framework for crawler evaluation research defined by Menczer et al 2004. Now the gold standard is represented by target set of web sites. According to Menczer et al (2004) in order to evaluate crawler algorithms, we need *topics* and some corresponding relevant *targets*. One could theoretically generate topics and target sets using frequent queries from search engines and user assessments. However this approach would make it very expensive to obtain a large number of topics and target sets, and very cumbersome to keep such a dataset up to date over time given the dynamic nature of the Web. Fortunately this data is already available from Web directories updated by human
editors and we used the Open Directory Project (ODP) that is an “open resource” (www.dmoz.org) and is maintained by a very large and diverse number of volunteer editors. We collected topics driven by our ontology classes and so we searched to identify ODP “leaves” pages that have no children category nodes.

A topic is represented by three types of information derived from the corresponding leaf pages. First, the words in the ODP hierarchy form the topic’s keywords. Second, the external links form the topic’s targets. Third, we concatenated the text descriptions and anchor text of the target URLs (written by ODP human editors) to form a topic’s description. The difference between a topic’s keywords and its description is that we give the former to the crawlers, as models of (short) query-like topics; and we use the latter, which is a much more detailed representation of the topic, to judge the relevance of the crawled pages in our post-hoc analysis. Table 3.4 shows a few sample topics.

<table>
<thead>
<tr>
<th>KEYWORDS</th>
<th>DESCRIPTION</th>
<th>TARGET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy efficiency</td>
<td>energy efficiency in buildings, industry, power, and transportation sectors;</td>
<td><a href="http://hydropower.inel.gov/">http://hydropower.inel.gov/</a></td>
</tr>
<tr>
<td>Energy program</td>
<td>renewable energy technologies; and DOE programs in these areas.</td>
<td><a href="http://www.eia.doe.gov/">http://www.eia.doe.gov/</a></td>
</tr>
<tr>
<td></td>
<td><strong>DOE Hydropower Program</strong> - Provides information on hydroelectric technology</td>
<td><a href="http://www1.eere.energy.gov/femp/">http://www1.eere.energy.gov/femp/</a></td>
</tr>
<tr>
<td></td>
<td>and tech transfer, state resource assessment, environmental concerns, and</td>
<td><a href="http://www.ofe.er.doe.gov/">http://www.ofe.er.doe.gov/</a></td>
</tr>
<tr>
<td></td>
<td><strong>Energy Information Administration (EIA)</strong> - A statistical agency of the U.</td>
<td><a href="http://www.oe.energy.gov/">http://www.oe.energy.gov/</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td><a href="http://www.rmotc.doe.gov/">http://www.rmotc.doe.gov/</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td><a href="http://www.sepa.doe.gov/">http://www.sepa.doe.gov/</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td><a href="http://www.spr.doe.gov/">http://www.spr.doe.gov/</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td><a href="http://nnsa.energy.gov/">http://nnsa.energy.gov/</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td><a href="http://www.whitehouse.gov/infocus/technology">http://www.whitehouse.gov/infocus/technology</a></td>
</tr>
<tr>
<td>Fossil fuels</td>
<td><strong>The Fossil Fuel Foundation of Africa</strong> - Organization constituted to further</td>
<td><a href="http://www.fossilfuel.co.za/index.asp">http://www.fossilfuel.co.za/index.asp</a></td>
</tr>
<tr>
<td></td>
<td>science</td>
<td><a href="http://www.iash.net/">http://www.iash.net/</a></td>
</tr>
</tbody>
</table>
Table 3.4. Representation of three types of information of different topics.

We also explored different methods for summarizing crawler performance such as plotting the mean similarity of retrieved pages to topic descriptions over time and computing the average fraction of retrieved pages that were assessed as being “relevant” by the classifiers. In particular we propose to assess both recall and precision, using target and lexical criteria for page relevance.

We employ the performance metrics focusing on the effect of different machine learning techniques on the performance of crawling algorithms over time. This view is especially appropriate when studying issues such as greediness and adaptation, whose effects are likely to vary in the course of the crawl. The dynamic measures provide a temporal characterization of the crawl strategy, by considering the pages fetched while the crawl is in progress.

The first metric is the simple precision level based on the target pages which we see in the third column of table 3.3):

$$\rho(i) = \frac{|S(i) \cap T|}{|T|}$$

where $S(i)$ is the set of pages crawled by crawler $i$ and $T$ is the target set. This measure allows us to determine how well a crawler can locate a few highly relevant pages. However, there may well be many other relevant pages that are not specified in the target set. To assess a crawler’s capability to locate these other relevant pages, we have to estimate the relevance of any crawled page. We do so using the lexical
similarity between a page and the topic description. Recall that the crawlers do not have any knowledge of topic descriptions, and that these descriptions are manually compiled by human experts as they describe relevant sites. Therefore pages similar to these descriptions have a good likelihood to be closely related to the topic.

The second metric is recall level that is like precision but it divides to $S(i)$:

$$r(i) = \frac{|S(i) \cap T|}{|S(i)|}$$

In figure 3.11 we can see how we use the directories of dmoz.org and also the windows of a topic, description and target for Fossil Fuel concept.
3.4 Experimental results
Concerning the first gold standard after we have finished the various queries with the different spiders we calculated the measure of precision and recall of the results as we described in section 3.3. Precisely we used:

- the number of relevant and retrieved documents to measure the precision of each crawler,
- the number of relevant documents in database and the number of relevant documents retrieved by each crawler.

We render the results of each query in table 3.2. Precisely the table shows in the first column a specific query, in the second column the number of relevant documents in the database for each query and in the other columns the number of relevant documents retrieved by crawler concerning a specific query.

<table>
<thead>
<tr>
<th>QUERY</th>
<th>DATABASE</th>
<th>INFOSPIDERS</th>
<th>BEST FIRST</th>
<th>SSSPIDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy infrastructures</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Energy technology</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Energy Security AND Affordability</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Energy Security AND Reliability of supply</td>
<td>8</td>
<td>7</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Energy Security AND Friendliness to environment</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Energy Security AND Oil Supply</td>
<td>15</td>
<td>10</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>Energy Security AND Natural Gas supply</td>
<td>10</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Energy Security AND Strategic petroleum reserves</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Energy Security AND Energy functions</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Energy Security AND Energy storage</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Energy Security AND Different suppliers</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Energy Security AND Different sources</td>
<td>15</td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Energy Security AND Geopolitical problems</td>
<td>12</td>
<td>10</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>Energy Security AND Renewables</td>
<td>15</td>
<td>7</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

**Table 3.2.** Relevant documents of database and relevant documents by each crawler.
Moreover table 3.3 shows the number of documents retrieved by each crawler.

<table>
<thead>
<tr>
<th>QUERY</th>
<th>INFOSPIDERS</th>
<th>BEST FIRST</th>
<th>SSSPIDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy infrastructures</td>
<td>5</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>Energy technology</td>
<td>5</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>Energy Security AND Affordability</td>
<td>3</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>Energy Security AND Reliability of supply</td>
<td>8</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Energy Security AND Friendliness to environment</td>
<td>6</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>Energy Security AND Oil Supply</td>
<td>12</td>
<td>12</td>
<td>35</td>
</tr>
<tr>
<td>Energy Security AND Natural Gas supply</td>
<td>5</td>
<td>4</td>
<td>35</td>
</tr>
<tr>
<td>Energy Security AND Strategic petroleum reserves</td>
<td>5</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Energy Security AND Energy functions</td>
<td>5</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>Energy Security AND Energy storage</td>
<td>4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Energy Security AND Different suppliers</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Energy Security AND Different sources</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Energy Security AND Geopolitical problems</td>
<td>18</td>
<td>15</td>
<td>35</td>
</tr>
<tr>
<td>Energy Security AND Renewables</td>
<td>9</td>
<td>8</td>
<td>20</td>
</tr>
</tbody>
</table>

**Table 3.3. Documents retrieved by each crawler.**

Now we shows the results of precision and recall through some graphics.
Figure 3.11. Precision and recall for each query with Infospiders algorithm.

Figure 3.12. Precision and recall for each query with Best First algorithm.
In the experiments with second gold standard we decided to formulate few queries that included broad knowledge because in ODP we can explore directory categorized through general concepts. In fact we can see in the first column of table 3.4 the sense of each keyword is less specific than concept represented. In table 3.4 we render the results of the four queries that we use to check the efficiency of three spiders. For each query done in a crawler we evaluate the presence of retrieved website url in target set of ODP.
Table 3.4. Measures of Precision and Recall.

<table>
<thead>
<tr>
<th>ENERGY SECURITY</th>
<th>PRECISION</th>
<th>RECALL</th>
<th>RENEWABLES</th>
<th>PRECISION</th>
<th>RECALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>INFOSPIDERS</td>
<td>0.45</td>
<td>0.70</td>
<td>INFOSPIDERS</td>
<td>0.50</td>
<td>0.70</td>
</tr>
<tr>
<td>BEST FIRST</td>
<td>0.40</td>
<td>0.65</td>
<td>BEST FIRST</td>
<td>0.35</td>
<td>0.60</td>
</tr>
<tr>
<td>SSSPIDER</td>
<td>0.30</td>
<td>0.50</td>
<td>SSSPIDER</td>
<td>0.25</td>
<td>0.50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OIL</th>
<th>PRECISION</th>
<th>RECALL</th>
<th>GEOPOLITICAL PROBLEMS</th>
<th>PRECISION</th>
<th>RECALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>INFOSPIDERS</td>
<td>0.60</td>
<td>0.75</td>
<td>INFOSPIDERS</td>
<td>0.40</td>
<td>0.60</td>
</tr>
<tr>
<td>BEST FIRST</td>
<td>0.50</td>
<td>0.65</td>
<td>BEST FIRST</td>
<td>0.40</td>
<td>0.50</td>
</tr>
<tr>
<td>SSSPIDER</td>
<td>0.45</td>
<td>0.30</td>
<td>SSSPIDER</td>
<td>0.50</td>
<td>0.40</td>
</tr>
</tbody>
</table>

3.5 Ontology population with Bingo! and Arachnid

Up to this point we used focused crawler that start with a query. Now we can use other type of focused crawlers having estimated the relevance of retrieved page. In fact at this phase of the work it is needed to enrich the ontology with additional classes and instances. So we turned to crawlers that start with an URL and render all its inside links. We particularly used Arachnid (described in section 2.6) and BINGO! (described in section 2.7).

We started by selecting some URLs of those documents (retrieved from MySpiders) related to concept of Energy Security and Geopolitical Problems which were questions suggested by our domain expert.

As we have seen in chapter 2 Arachnid is a Java-based web spider framework. It includes asimple HTML parser object that parses an input stream containing HTML content. Simple Web spiders can be created by sub-classing Arachnid and adding a few lines of code called after each page of a Web site is parsed. So we insert in the Test class a string that represent the URL to crawl (we see in figure 3.12 an example) and run the program. We repeat this procedure for more 10 URLs.
Figure 3.12. A piece of Java code in which we insert the name of the URL to crawl

After few seconds we had as results a list of URLs that represent external links, internal links and also not-HTML links like documents in pdf format.

<table>
<thead>
<tr>
<th>Table 3.5. Results of a Test on a URL in Arachnid. Thread[Thread-0,5,main] started</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stress Test Finished</td>
</tr>
<tr>
<td>Traversed 26 pages</td>
</tr>
<tr>
<td>Found 0 bad pages</td>
</tr>
<tr>
<td>I/O errors in 3 pages</td>
</tr>
</tbody>
</table>
Other useful tasks of Arachnid are represented by SiteMapGen class and GetGraphics class. The following tables show the results of the implementation of code that uses Arachnid to generate a site map for a Web site and a directory in which put all images met in URLs crawled.

### Table 3.6. Results of GetGraphics

<table>
<thead>
<tr>
<th>Get Graphics:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saved image from: <a href="http://www.opec.org/headers/library/images/spacer.gif">http://www.opec.org/headers/library/images/spacer.gif</a></td>
</tr>
<tr>
<td>Saved image from: <a href="http://www.opec.org/headers/library/images/logo.gif">http://www.opec.org/headers/library/images/logo.gif</a></td>
</tr>
<tr>
<td>Saved image from: <a href="http://www.opec.org/headers/library/images/homeS1.gif">http://www.opec.org/headers/library/images/homeS1.gif</a></td>
</tr>
<tr>
<td>Saved image from: <a href="http://www.opec.org/headers/library/images/aboutusS1.gif">http://www.opec.org/headers/library/images/aboutusS1.gif</a></td>
</tr>
<tr>
<td>[.....]</td>
</tr>
<tr>
<td>Finished</td>
</tr>
</tbody>
</table>

### Table 3.7. Results of SiteMapGen

```html
<html><head><title>Site Map</title></head><body><ul>
[--------------]
</ul></body></html>
```

Now we present the results of BINGO!, another focused crawler that starts its tasks with an URL. This spider is easier to use than arachnid because it offers a simple user interface. In fact there isn’t the need to implement java code but only to interact with the program. The procedure in Bingo follow three steps:

- Insert the training documents;
- Run the crawler on inserted documents;
- Update the database with new retrieved documents.

In the example (figure 3.13) we insert three documents which are the results of “energy security and reliability” query in MySpiders and send them to crawler. These
papers are very relevant because they help us (as we see in follow chapter) to make enrich the ontology particularly to add new class and new instances to Energy Security class. So we decide to extract new URLs of documents linked to those.

![Insert New Training Document](image)

**Figure 3.13 Insert new training documents through its URL.**

After few minutes we have as results about 3000 entries. We can see the title of retrieved documents and open directly in the browser.
Moreover we can some other things with each document: open in browser, show source, show features and show links. In figure 3.15 we see as we can have the frequency of each term through a feature vector that is an n-dimensional vector of numerical features that represent some object.

Figure 3.14. Database Bingo after crawling on three URLs.

Figure 3.15. Computing feature vector for a document
Chapter 4
Discussions

The purpose of the experiments described in chapter 3 was to study how an ontology help the formulation of a query in a focused spider and how can we evaluate focused-crawling techniques used.

Our experiments on three algorithms of two focused crawler allowed us to understand the efficiency of these intelligent agents with respect a simple human search. Initially, by manually using a search engine, we have had a large quantity of documents but it would take a long time to read it all and to analyze their relevance. For this reason we decided to initiate new searches with topical crawlers taking an initial database of relevant documents.

After several questions crawlers have continued calculating the various measures previously submitted (recall and precision) so we have the way to understand what algorithm performs better and to be used in a following step that is ontology population.

The analysis of the results showed that InfoSpiders agents is the most effective when measured by recall and precision even if the algorithm Best First shows a slight difference. The mechanisms behind these algorithms allow the excellent results had. With regard to InfoSpiders note benefit function based on the similarity to the query, and cost function to account for the network resources used to download pages.

Instead in Best First we note that the similarity between a page $p$ and the topic keywords is used to estimate the relevance of the pages pointed by $p$. The URL with the best estimate is then selected for crawling.

Surely InfoSpiders is more efficient if we consider the documents database, but certainly are both useful to search in the web relevant documents linked specific topic. This is even more evident if we think that for the same query a normal search
engine returns hundreds of links while crawlers retrieved ten links of specific documents that for the most part concerning the topic requested. The third crawler, SSSSpider shows behavior less efficient compared to the database but documents retrieved are mostly useful to discover new concepts related to the concepts of query subdued.

Now we present the last step of this work: ontology population using documents retrieved from MySpiders. As we have seen it retrieves a mostly of relevant documents of the database but the important thing is that can bring many other relevant documents never considered. In fact we have focused much on these new documents in order to extract new concepts and new relationships to be included in the ontology.

As the first demonstration of the advantage of using focused crawlers can note that we started from an ontology of 16 classes, 4 properties and 40 instances and after analysis of documents reported by the crawlers have been able to enrich the ontology bringing to 54 classes, 11 properties and 131 instances. Moreover we have to consider that our focus was the concept of energy security as we suggested our expert domain and so we have addressed most of the queries to the discovery of new relationships and new instances related to this vast subject. What they lacked in the first version of the ontology was the fact that it had no documents available to respond to our real needs.

The knowledge contained in the ontology allowed us to guide query formulation and the information formalized in each concept had the support of domain expert. In fact we note that in the first version of ontology the concepts have few instances and few subclasses. Only after we acquired new relevant documents through the use of crawler we extended each concept. Moreover this significant acquisition of new knowledge in retrieved documents guided the discovery of new concepts, new instances and new relationships among concepts.

For example the concept of Energy security was extended with the introduction of 4 subclasses because it appeared evident in many documents its relationship with
concepts like *Scarce Resources Dependency, Reliability of Supply, Friendliness to the environment and Resources Affordability*. These subclasses allowed to develop significant relationships between the concepts of Energy Security and Geopolitical problems. Precisely when we considered the concept *Reliability of Supply* we formalized others sub concepts that represented the problem of *cut off in supply* in various countries: *cut off in supply for Georgia, cut off in supply for Belarus* and *cut off in supply for Ukraine*. These concepts linked to the concept of *dispute* among countries contained in the class Geopolitical Problems represent significant knowledge useful to understand important current problems.

Another meaningful example of how we enriched the ontology with retrieved documents with the crawlers is represented by the extension of classes *Risk* and *Solution*. These two concepts represented in the first version of ontology two subclasses of *Energy Security*. When we analyzed different documents and we acquired new knowledge about this topics we decided to create two classes separately from Energy security. In fact we hadn’t clear this concepts in relation to the problems of security in energy supplying and so with new information we created different classes that represented better the concepts of *Risk* as *technical failures, economic problems* (like peak oil) and *geopolitical problems* and *Solution* that is a concept that different literature rendered as *short term actions* and *long term actions*.

Following we render some images that represent the evolution of the Energy Security concept and the new concept of Geopolitical Problems.

Figure 4.1 show the increase in the number of the classes of the ontology and that means how many new concepts and new relationships were introduced. This development derives from the numerous information found with the work of crawlers. In fact we were able to extend each class for more than one level of subclasses because we have had several relevant and specific information for each concept. This picture shows only links through is-a dependence even if it is meaningful that each class is extended in other subclasses so that ever more detail the concept that represents.
It is necessary to use other views because the previous figure does not make the idea of present relations between classes and between classes and individuals. Therefore, these tools employ schema visualization techniques that primarily focus on the structure of the ontology, i.e. its concepts and their relationships. The users of these
tools need to get an insight in the complexity of the ontology. For end users, instance information is often more important than the structure of the ontology that is used to describe these instances.

We use a tool of Protégé called Jambalaya that provides an extensible, flexible, and scalable visualization environment for exploring, navigating, and understanding ontologies. Classes and instances are represented as nodes in a graph. Directed edges are used to show relationships between concepts and instances.

From this view (figure 4.2) emerges as the ontology is made up concepts very interconnected with each other.

![Figure 4.2. A domain/range view of the whole ontology through Jambalaya](image)

This other view (figure 4.3) can show several properties that link different classes but especially different individuals of various classes. The use of properties is meaningful in the ontology construction because it allow to assert fundamental concepts.
Figure 4.3. A radial layout of the whole ontology through Jambalaya

The figure 4.4 present a particular view that is related at the term search task. The user can first enter an initial search term (Geopolitical problems in our example) and can browse the hierarchy. The tool presents a visual representation of requested term in a network of concepts. So we have the visualization of all relations of that term with other concept or inside its class.
Figure 4.4. A view of the search to Geopolitical Problems
Conclusions

The goal of this thesis was to show that knowledge directed focus crawling can be applied in the domain of Energy Security. More specifically, the research questions were:

How can knowledge directed focus crawling be applied in the domain of Energy Security for automatic search of domain related documents?

How can an ontology help the formulation of a query in a focused spider?

How can we evaluate focused-crawling techniques?

We investigated how we can use information to guide spidering process to ontology population. Information contained in an ontology allows more efficient discovery and extraction of resources in the web. In fact using knowledge formalized in concepts, instances and relationships among concepts we guided the discovery of meaningful documents to enrich ontology. The documents retrieved in the web without the use of crawler are a lot and their analysis request much time while a spidering process guided by basic information results more efficient.

As we have discussed in this thesis, a lot of work has already been done on the discovery of resources on the Web using different sorts of crawlers. Our goal was to show that using different sorts of crawlers allows more efficient discovery and extraction of documents in the Web for Energy security domain.

We have opted for a focused crawler as we want to discover and extract information during the crawl. To find resources relevant to a given topic, we need to model the domain of interest of this topic. In general, this is done using a simple keyword description but what distinguishes us from others approaches is that we model the domain of interest with a topic ontology instead of a keyword description. In our approach, we have used an ontology to model this domain of interest. Guiding the crawl is what distinguishes a focused crawler from a general-purpose crawler.
The Web is growing exponentially, and crawling the whole Web is an infeasible task. Topical crawling, or focused crawling, differs from standard exhaustive crawling in the way that it narrows the search down to regions on the Web which are likely to contain relevant pages. An exhaustive crawler just downloads the pages it visits, index them, and follow the hyperlinks without further analysis of the page contents. A focused crawler limits its search to regions of the web that are likely to lead to pages of interest. It uses measures to determine the goodness of a page, and skips further crawling of pages that seem to be fruitless.

The large size and the dynamic nature of the Web highlight the need for continuous support and updating of Web based information retrieval systems. Crawlers facilitate the process by following the hyperlinks in Web pages to automatically download a partial snapshot of the Web. While some systems rely on crawlers that exhaustively crawl the Web, others incorporate “focus” within their crawlers to harvest application or topic specific collections.

In this paper we discussed a review of several topical crawling and later we used different focused crawling with an approach that goes beyond existing focused crawling. In fact we built an ontology as background for focusing the search in the Web through these agents. We used this ontology to formulate various queries in different focused spiders applications and so we acquired new documents. This new information allowed us to enrich our ontology and to continue topical crawler use in a more specific way.

The spiders used have been evaluated to understand what could be more useful to ontology population task. We have provided a quality-based empirical evaluation based on a standard model of precision and recall.
Bibliography


topic specific Web resource discovery. In Proc. the 8th World Wide Web conference
WWW8, Toronto, May 1999. Available at http://www.cse.iitb.ac.in/~soumen/doc/www


Cho, J.-E. Garcia-Molina, and Lawrence Page, Efficient crawling through url
ordering. In Proc. the 7th International World Wide Web Conference, Brisbane, Australia, April 1

Cho, H. and E. Garcia-Molina. The Evolution of the Web and Implications for an
Incremental Crawler. Proc. the 26th International Conference on Very Large
papers/cho-evol.pdf

De Bo, J.-P. Spyns, and R. Meersman, Towards a methodology for semi-automatic
ww.ai.fb.unikarlsruhe.de/WBS/ywa/publications/wang06
KASO_AAAIFS.pdf

Dean, M. - D. Connolly, F. van Harmelen, J. Hendler, I. Horrocks, D. L.
McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL web ontology language


