Diplomarbeit

Using Integer Linear Programming for Search Results Optimization

Einsatz ganzzahliger linearer Programmierung
zum Optimieren von Suchergebnissen

Jérôme Kunegis
March 15th 2006

Diplomarbeit
an der Fakultät IV
der Technischen Universität Berlin

Betreuer:
Şahin Albayrak
Stefan Fricke

Abstract

Document search is often implemented by assigning a score to each document and then searching the documents with the highest scores. Sometimes, constraints are added to the search, such as searching documents with a maximal average age, or needing the resulting documents to be as diverse a possible. Several such constraints are formulated as a mixed integer linear program. The resulting problem is then solved by using well-known algorithms from linear programming. Various possible constraints are researched, combined, and then compared to existing algorithms and implementations.

Eidesstattliche Erklärung

Die selbständige und eigenhändige Anfertigung versichere ich an Eides Statt.

Berlin, den 15.3.2006

_____
Jérôme Kunegis

Acknowledgements

First of all I would like thank my professor and advisor Prof. Dr. Şahin Albayrak for supporting and advising me in the choosing and redaction of my thesis.

I thank Dragan Milošević for helping me with his experience, and for enforcing important rules at the beginning of the thesis writing.

I thank the other members of project PIA for helping me with their knowledge and experience, both on the topic of information retrieval and in their experience of thesis redaction, especially in solving technical problems.

Deutschsprachige Zusammenfassung

Beim Information-Retrieval (z.B. eine Suchmaschine) gibt der Benutzer Suchbegriffe ein. Daraufhin wird für jedes bekannte Dokument eine Bewertung berechnet. Die n besten Dokumente werden dann dem Benutzer präsentiert. In diesem Modell kann die Bewertungsfunktion beliebig komplex sein. Jedoch ist es nicht möglich, Beziehungen zwischen Dokumenten Auszudrücken, z.B. dass mindestens die Hälfte der Dokumente in einer bestimmten Sprache sein sollen, oder dass die Dokumente sich nicht zu sehr ähneln sollen. In dieser Diplomarbeit formuliere ich die Dokumentensuche als ein gemischt ganzzahliges lineares Programm. Damit ist es möglich, solche Querbeziehung auszudrücken, solange sie linear sind. Das ganze wird dann von einem Solver gelöst. Dabei ist zu bemerken:

- Die Anzahl der Zusatzbedingungen (constraints) ist nicht begrenzt.
- Alle Nebenbedingungen sind mit einem Faktor versehen, den man stufenlos anpassen kann.
- Das Endergebnis ist garantiert optimal.
- Es können Zwischenergebnisse ausgegeben werden, deren Abstand zur Optimallösung nach oben abgeschätzt werden kann.

Unter anderem werden folgende Nebenbedingungen untersucht:

- Ein Mindestanteil der Ergebnisse muss in einer bestimmten Sprache sein.
- Die Diversität der Ergebnismenge soll maximiert werden. Diese kann zum Beispiel als der Mindestabstand zwischen zwei Dokumenten definiert werden, wobei eine Abstandsfunktion gegeben sein muss.
- Das Durchschnittsalter der Dokumente soll eine bestimmte Grenze nicht überschreiten.

Zusätzlich kann jede dieser Nebenbedingungen auch als schwach deklariert werden (soft constraint). Dann sind Überschreitungen der Grenzen erlaubt, werden aber mit einem Abzug in der Zielfunktion mitberechnet.

Contents

1 Introduction

In this thesis, a method for implementing user constraints in information retrieval is developed. This method is based on integer linear programming.

In traditional information retrieval, documents are searched which best match a user's query. Usually, users want additional constraints to be fulfilled by the results, such as documents being in a specific language or the absence of duplicate results. This thesis is based on the idea that several of these constraints can be combined.

Taken in isolation, methods are known for implementing each constraint. The novelty of this method lies in the fact that constraints can be combined arbitrarily, and that they can be weighted individually.

To achieve this goal, a method known as integer linear programming is used. The problem of finding an optimal result set is formulated as an integer linear program, for which good solvers exist.

To assist this thesis, an implementation of these methods was made. In this text, "the implementation" will always denote this implementation.

The rest of this introduction describes some scenarios which highlight weaknesses or just interesting situations in conjunction with traditional search engines. In the chapter *Related Work*, several approaches from the literature are presented. The chapter *Integer Linear Programming* introduces integer linear programming and how it can be used to implement a standard document search. The next chapter, *Constraints*, describes how various functionality found in related publications can be formulated as linear constraints, and thus integrated into the problem. In the chapter *Implementation*, the implementation made in conjunction with this thesis is presented. Then, in the chapter *Evaluation*, the implementation is evaluated by various tests, and then the results compared to other methods described in related work. In the *Conclusion*, results are summarized, and tasks beyond the scope of this thesis are shortly described.

1.1 Problem Description

In this section, several scenarios are described which highlight weaknesses in existing information retrieval implementations.

1.1.1 "Apple"

The word "apple" may denote different concepts: a fruit, a corporation and other less often used names.

Typing in "apple" into Google returns the following results[1]:

[1] The search was made using `http://www.google.com/.` on 2005-7-11

- place 1–13: pages about Apple Computers
- place 14: Apple Vacations
- places 15–20: again pages about Apple Computers
- ...
- place 22: the first fruit-related result

This example shows that:

- The first page[2] contains only links to Apple Computers.
- The fruit is mentioned for the first time on the third page.
- The results have absolutely no diversity: The results on the first page are similar, and results different from these are not on the first page.

Assuming that the apple fruit is about as important as the corporation, if not more so, results from both domains should be equally represented on the first result page. To achieve this, the search space should be divided into several classes: one for each meaning of the word. Requiring that about as many results are returned from each class will thus increase diversity in this case.

Figure 1.1 shows the situation graphically: Each document is represented by a point. The X axis represents the score of the document, the Y axis the domain. Documents about the fruit are many but do not have very good scores. Documents about the corporation are less but with higher scores. The graphics show global search in categories. Results chosen by each method are encircled.

1.1.2 "Agent"

The word "agent" has several meanings. Searching for this word should take all possible acceptions into account. The word is also part of some proper names. As an example, let's take a user searching encyclopedic documents about "agent". Search engines that divide the documents space into categories are useful for this purpose. Each document known to the search engine would have to be in one category: scientific documents, news, commercial information, etc. In such a search engine, our user would choose the category "science", and would find only scientific documents. But what if some document from another category had a good score when applied to this query? This document could not be returned, because it is in the wrong category.

This example shows that categories should not always be strict: Allowing a small proportion of documents from other categories can be useful. So how should this be implemented? One simple answer is to search in all documents, and adjust the score of each document: If the user wanted scientific documents, add a bonus to the score of

[2] Google displays ten results per page.

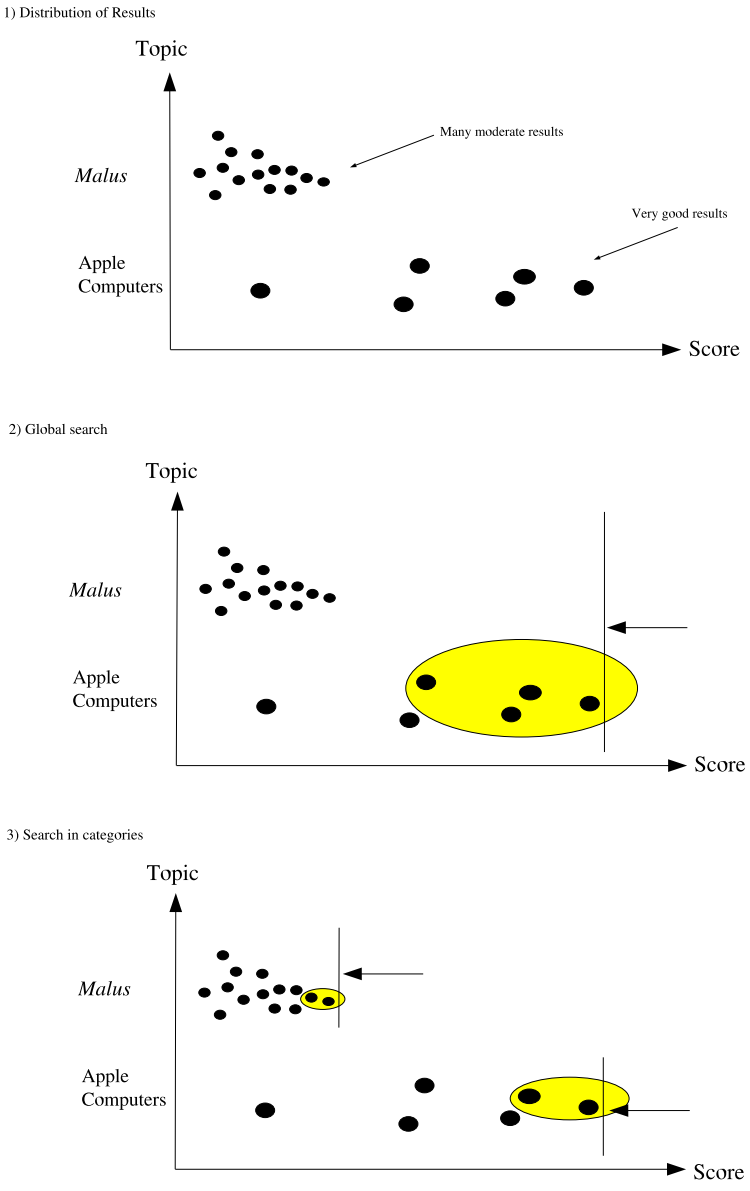

Figure 1.1: Searching in categories: Using global search and search in categories

each scientific document. But is this good? Maybe the user can tolerate a small amount of documents from another category.

This kind of constraints, namely that a certain proportion of returned documents should have a certain property, can be formulated by linear programming. This thesis will show how several constraints can be combined.

1.1.3 Encyclopedias and Dictionaries

When a user enters a word in a search engine, there are usually two cases to distinguish: First, the user may enter a word he knows the meaning of, and wants more information about. In the second case, the user has read a word he does not understand, and passes it to a search engine to find out its meaning. As the user has not seen the word before, he does not know whether the word is a name, a word in his own language, or a word into another language. In the first case, users typically need encyclopedic information. In the second case, they need information from a dictionary.

Usually, search engines do not distinguish the two information types. Still, results of search engines are mostly relevant because for a given word, most information is either encyclopedic or lexical. A problem arises however when the user has set an option in the search engine to only search one type of information. If the documents found are all of one type and the user searched for the other type, no results will be displayed. As a solution, the restriction to one type of document should not be strict. Even if the user searches for definitions, other, non-definition documents could be returned when no definitions were available.

This kind of functionality can be implemented by using mixed integer programming, and by setting appropriate soft constraints.

1.1.4 User-Defined Constraints

Sometimes, when users read the list of results from a search engine, they encounter results they cannot or do not want to read. If this happens, users would like to change the results, such that these results are not returned again for the same query. To achieve this, meta-information about documents can be used. Meta-information can include the language of the document, or the file format of the document, or simply the length of the text.

A simple implementation of such a feature would be to allow the user to block certain documents. For instance, if the user encounters a document in Spanish, but does not understand Spanish, then the user may like to block all results in Spanish. Such blocks can be part of the user's settings, and could be reused for later searches. However, these blocks can have negative consequences. If, for a certain query, all results are in Spanish, the user would not see any results. In this case, returning documents in a language the user does not understand is still better than returning no results at all.

Using the approach developed in this thesis, such a block can be formulated as a *soft constraint*. Soft constraints, which will be described in detail in the section *Soft*

Constraints of chapter *Constraints*, allow a constraint to be honored most of the time, but still be violated when the alternative is even worse.

1.1.5 Document Types

Searching information about the city of Paris can give several types of resulting documents, all being relevant:

- Encyclopedic information, such as a document about the history of Paris
- News, for instance about sports events
- Papers and scientific publications, for instance about conferences in Paris
- Commercial information, for instance web sites of companies located in Paris
- Official sites such as *www.paris.fr*

Search engines such as Google do not distinguish these different types of information. It is however possible, using classificators, to find out programmatically the document type. This information can then be used for providing some additional search functionality:

- Users may want to refine their search by excluding a certain type of document (i.e. commercial sites), or by restricting their search to one specific type.
- When users do not refine their search, it might be preferable to return about as many results from each type.

Both these constraints can be implemented by using mixed integer programming. These constraints can be either hard, to force a given number of results from each type, or soft to allow documents from other types to be returned if they are very good.

2 Related Work

This chapter gives an overview of existing search methods described in the literature. These methods are compared to the method developed in this work. References are given that describe the methods in detail. When no reference is given for a method, the method is trivial, and used implicitely in many applications.

Most publications presented here try to solve one specific problem, or approach the problem in one specific way. As the integer linear optimization approach taken here is general and not specific, these methods are analyzed from an angle of generalization: How can they be integrated into the global search using integer linear programming?

The first section, *Independent Search*, describes information retrieval in general. Further sections add other features which should make the search better in some specific way.

2.1 Independent Search

Mathematically, a user search can be described as follows: A certain number of documents are known to the system. When a user enters a query, the system calculates, for each document known, a score indicating how it matches the user query. Then, the k documents with the highest scores are returned to the user. Documents with a high score shall be called "good" and those with a low score "bad".

One must first note that the previous description is functional; that is, it does not describe an algorithm. In fact, algorithms for information retrieval will unconditionally use an index. Searching in all documents sequentially would have a runtime proportional to the number of documents known. This runtime is probably too long for but the simplest cases.

Second, the term "document" should not imply that the user is searching for text. He could as well be searching for sound or another form of information. However, because text search is the prevalent form of search, the following chapters will assume a textual search. The implementation also uses text search.

The simple method described shall be called "independent search". The word "independent" describes the property that each document is considered separately by this method. For instance, if two good documents are identical, they will both be returned by independent search, although it is generally thought to be better to only return one and instead return a different document with a lower score.

The main difficulty in implementing independent search lies in the fact that the overall number of documents is usually huge. By default, documents will be indexed, and only documents from the index of the user keywords are checked.

2.2 Eliminating Duplicate Entries

To overcome the problem of similar results common in simple independent search, a technique of eliminating duplicate entries as described in [25] and [26] is often used. Using a predefined metric, documents too similar are put into groups, and only one document is kept from each group. The others are discarded.

This elimination of duplicates can be implemented before any search is done, and is thus efficient. It is not optimal because the similarity function cannot depend on the user query. For instance, if one document is the translation of another document, then a user accepting both languages will probably not want both documents. For a user talking just one language, the documents should not be taken as equal.

Also, when eliminating duplicate entries, the similarity must be very strict, or perfectly valid documents would never be returned. It is not possible to mark documents as similar but not equal.

As described in the chapter *Constraints*, eliminating duplicate entries will be covered in this thesis by maximizing diversity, and thus no separate duplicate elimination will be necessary.

2.3 Strict Categories

Sometimes, the documents are already classified. For instance their language is known, or they are marked as being a news article, or commercial information. In any case, increasing the diversity of the results by returning documents from several classes will make the result set more diverse.

The simplest implementation makes a search in each category separately, and then combines the result to give overall results. The proportion of which documents are taken from each category must be decided. If the proportion is fixed in advance, for instance by taking the overall number of documents, the results will not depend on the query itself. For instance, if the search phrase is the name of a product, then most good results will be from commercial sites. Searching in various categories, however, and aggregating results will give as much results from encyclopedic sites as from commercial sites, thus forcing bad encyclopedic results to replace good commercial results.

When using integer linear search, maximizing the document diversity globally will cover categories in this implementation. Documents from the same category will be more similar by nature, thus diversity maximization as described in section 4.9. To go even further, information about the category can be integrated into the distance function, thus making the results even more diverse category-wise.

2.4 Combinatorial Diversity

The paper [1] describes a greedy algorithm for improving the diversity of results by setting a minimum threshold for the document score. All documents better than this

threshold are taken, and among these, k are searched such that the diversity is maximized. The diversity here is meant as the minimum "distance" between two documents, where the distance is calculated for each document pair separately.

In the general case, this algorithm will return documents with a lower average score than pure independent search. However, a lower bound can be set for the average score by choosing it as the threshold.

What this algorithm cannot do is making a trade-off between similarity of documents to the query and diversity of the result set. Once documents from above the threshold are chosen, their score is forgotten.

It will be shown in the evaluation of the integer linear programming approach that the combinatorial diversity approach can be considered a special case of the integer linear programming approach.

2.5 Diversity Using Similarity Layers

In [2] David McSherry describes a post-processing technique for increasing the diversity of the result set without decreasing its similarity to the search phrase. In essence, the most dissimilar results in the result set may be interchanged with other as dissimilar results.

Similarity is thus never affected, and only few results can be changed. This technique is mostly useful when many documents have the same score. If scores are all different, diversity cannot be increased in this way. No tradeoff between similarity and diversity takes places. This method is thus not useful in conjunction with complex score function, where the probability of equal scores is low.

If the diversity is included in the objective function using appropriate constraints, it will be weighted against the score of better results. In the chapter *Evaluation* a test is made modifying the weight of the diversity constraints and measuring the similarity.

2.6 Multi-Objective Optimization

While the main goal of a document search is to return documents similar to the query, users may also want other goals: As much as possible of the documents should be in their language, and as few as possible should be in a format they can't process. In the general case, such goals cannot be fulfilled at once. If the documents most similar to the query are not in the user's language, maximizing similarity would return these documents, while maximizing the number of documents in the correct language would return other, less similar, documents in the correct language. Definining even more goals, such as maximizing the diversity of the result set, only makes the problem more complicated.

How then to achieve multiple goals? If each goal function is represented as a number, just adding all these functions with appropriate factors will give a new function that can be maximized. However, another approach exists: The set of Pareto optimal solutions.

If several goals are known, optimal solutions can be calculated for each goal individually. Each of these results has the property that it cannot be modified without decreasing

at least one of the goal functions, in this case the one for which the result was optimal. But this property also holds for other solutions: the solutions that cannot be modified in such a way that no goal function gets worse. The set of these solutions is called the Pareto optimal set. The set is also called the Pareto optimal front since geometrically it lies at the edge of the overall result set.

[4] describes a method whereas multiple objective functions can be defined, and only solutions in the Pareto optimal front are considered. A combinatorial approach is used to optimize diversity among all solutions in the front.

[9] also describes an efficient algorithm for finding the Pareto optimal set integrated into database queries. The paper calls the new operator the "skyline operator".

[10] describes a similar algorithm, but without integration into database queries. [12] gives more detailed algorithm descriptions. Several papers describe how to use a genetic algorithm search for finding a set of solutions in the Pareto optimal set.

In runtime, all these algorithms are generally not worse than $O(n^2)$, and not better than $O(n \log n)$. The algorithms are highly dependent on the number of combinations. That is, if the objective functions are all themselves diverse, the front is big and the algorithm slow, if the front is small the different objectives must be few and similar. The more objectives are defined, the greater is the front, and the slower is the algorithm.

A general inconvenience of combining objectives using the Pareto set is that the objective functions cannot be weighted. Thus, only important ones should be used.

The approach taken in this thesis consists of allowing a multitude of constraints, and each of these constraints can be regarded as a goal. Thus, the set of Pareto optimal solutions can serve as another approach for implementing constraints.

2.7 Genetic Algorithms

[16], [17], [18] and [19] describe how to use genetic algorithms to find a set of solution to search with multiple objectives. Since all results in the Pareto optimal front may be interesting, these algorithms try to spread out all results on this front.

This method may give diversity and good intermediary results. However, it has some drawbacks:

- The pertinence of the result set is not measured in any way. Thus, no guaranty about the results can be made in any way.
- As usual with genetic algorithms, no runtime estimation can be made.

Using integer linear optimization, the optimality of the result is always guaranteed. As to the runtime estimation, it cannot be made either with integer linear search.

2.8 Multiple Global Objectives

The Pareto optimal set can also be used in another way: Each possible result set is considered a point, and then objectives are defined for sets, such as diversity. Then,

the Pareto optimal set is considered. Since there are 2^n points, this is hard to calculate explicitely. Heuristics such as evolutionary algorithms have to be used. These are not described in the papers describing genetic algorithms, which only use the simple document set.

2.9 Online Algorithms

[11] and [15] describe online or conversional systems. In a conversational system, instead of entering a query directly, giving all information about the query at once, the computer asks questions to the user. When the user answers, the computer processes the response to generate new questions. The result of these iterated questions and answers is a complete query. Instead of entering a great amount of information about what a user is searching, a user just has to answer simple questions.

Note that constraints in general can be regarded as a kind of feedback. They can be saved in the user profile and reused in other queries. Similarly, feedback collected about a user can be turned into constraints. For instance, if it was detected that a user prefers documents in a specific language, this information can be turned into a constraint concerning the language of the results.

The goal of this thesis is to give the user a way to add constraints to a document search. Instead of giving all constraints at once, online algorithms are provided with constraints sequentially. As will be described in the chapter *Implementation*, constraints can also be added to a previously solved problem to generate a new problem. Thus, an implementation of the integer linear programming approach can be made into an online algorithm.

2.10 Constraint Programming

[13] and [27] describe constraint programming, another general approach to satisfying several constraints at once. This approach is combinatorial, not necessarily efficient, and constraint satisfaction problems necessitate specific algorithms. In constraint programming, constraints are usually hard, whereas in this integer linear programming approach, soft constraints are preferred.

In this implementation, no methods of constraints programming will be used. Constraints will be formulated as inequalities in an integer linear program.

2.11 The General Economic Approach

In economy, the objective function is called the utility function; it is always maximized. [14] describes a general framework for solving the general utility maximization problem were:

- The utility function is not necessarily linear,
- and several utility functions may exist.

The general problem is very hard to solve. All constraints considered can be reformulated as linear (when considered soft). [14] uses a custom depth first search, which is suitable for non-linear objective functions. This approach is similar in a way to binary integer programming (using variables whose values can only be zero and one), because some integer program solvers use branch cutting and depth first search, known there as branch and bound.

[14] mentions that *unfortunately, 0-1 integer programming is proved NP-complete, and no [existing] polynomial time [algorithm] is able to solve it.* Of course, integer programming solvers are very efficient, and have the advantage that they apply to general problems. Problems can be formulated in a general way and solved automatically. No custom algorithm has to be written like the one described in [14]. Using integer programming, adding new constraints or soft constraints is thus easier.

2.12 Sorting Objective Functions

This idea comes from [14] again: Formulate several objective functions, for instance from soft constraints, order them by importance, solve the problem for the first objective function. Take the results, which are now less numerous than the those in the initial set, and continue with the second most important objective function. Repeat until all objective functions were considered or until the set of results cannot be made smaller.

In this method, the weighting of objectives is qualitative and not quantitative. Secondary objectives are possibly ignored, for instance if only few results are found after the first iteration.

The several objective functions must be independent from each other. For instance, the diversity cannot be used as a goal. If it is chosen as the most important goal, many documents must remain after the first iteration maximizing diversity, and the diversity of the final result set will not be guaranteed. Again, if diversity is done last in the chain of goals, only few documents will remain, which will probably not be as diverse as possible.

Using integer linear programming, there is just one objective function, which contains all these criteria either as a weighted sum, or as penalty variables from soft constraints. Thus, constraints which are not violated anyway have no influence of the results. However, constraints with small weights may still influence the objective function if they are violated, because they have non-zero penalty variables.

2.13 The Greedy Algorithm

In [14], a greedy algorithm is described for implementing various constraints. These constraints cover the result set as a whole. They can be of the form: At least N% of the results shall have this and this property. The algorithm works by beginning with an empty set of results, and adding documents one by one to this result set. At each step, take the document which lets the objective function increase the most. The aggregation stops when the objective function cannot be further increased by adding documents.

This method allows the constraints to be complex. For instance, maximizing the average diversity, which, as will be seen in this thesis is fairly expensive, can be done cheaply. The running time of this greedy algorithm will always be polynomial.

However, the resulting set is not guaranteed to be optimal. In fact, an upper bound on its optimality cannot be given for this method. This method also has the disadvantage that not all constraints can be used sensibly: Maximizing the diversity will not work, since it is hard to compare the diversity of sets of different size, which would be necessary. In the objective function, penalties can be combined using any complex formula. The objective function needs not even be linear.

In contrast, using integer linear optimization will be able to guarantee the optimality of the final result and the bounds given for intermediary results. As to the runtime, since the typical integer linear program solver uses the branch and bound algorithm, it can be compared to the greedy algorithm: The branch and bound algorithm will start with a valid solution, and branch on the value of a variable. If the branch and bound is done using the depth first method, more and more variables will be fixed until all variables are fixed. Then, the results will be given. This first part of the branch and bound algorithm can be compared to the greedy algorithm in that more and more variables are taken (or excluded) until a result is reached. The difference is that the branch and bound algorithm will continue after this first search, and not stop after this intermediary result, thus making the results better.

2.14 Cost and Time As a Goal

The publications [23] and [24] (both from the same authors) define the cost of retrieval as something to be minimized, e.g the time it takes for retrieval. Also, a retrieval cost is considered in these papers. This could be for instance the price to pay for viewing results, or the bandwidth to view the documents.

Although each of these values are defined independently for each document, they cannot be just added together with the score. Example: If cost was just substracted from the score, some document would have a positive score and some a negative score. Documents with a negative score would never be shown, and showing more documents with a positive score would never be restricted. Thus, the score and the cost must be considered separately.

When using integer linear programming, one could define a maximum price and a penalty variable only non-zero if this price is surpassed. This can be implemented as a soft constraint, as will be seen in the chapter *Constraints*.

2.15 Elaborate Scores

[21] describes a method for finding the score function itself. This is an interesting optimization because it can be done offline before searching for the documents themselves.

[22] describes how to search inside big datasets without indexing using linear algebra. Essentially, it is interesting because it defines a similarity function more complex than

usual search algorithms. However, the search is still independent.

[20] lists some constraints that may be useful for users, but without an algorithm for solving them. These constraints include the following three:

- The term frequency
- Term discrimination: term frequency in relation to global term frequency
- Length normalization: term frequency divided by the document length. When two documents mention a search word as often, the shorter is better.

The paper also describes how to combine these constraints. However, all these constraints concern just one document each. Implementing them would thus give an independent search.

Note that all scoring functions work by measuring the similarity between a document and a query. Some of these algorithms may be adapted to make comparisons between documents, and may therefore be suited for modeling diversity, which will play a role in the chapter *Constraints*, when the diversity constraints will be defined.

Because using elaborate score functions does not change the basic algorithm used in independent search, all these methods can be applied in combination with more complicated algorithms. Thus, they can also be integrated into this integer linear programming approach.

2.16 Summary

As these related methods and algorithms show, many extensions to simple independent search exist, such as using categories, finding and eliminating duplicates by hand, heuristics, using multiple goals, and genetic and online algorithms. Some of these features can be implemented by specialized algorithms.

3 Integer Linear Programming

Solutions as elaborated in various publications have in common that they solve specific problems by using specific algorithms. Since these methods are specific, it is hard to combine multiple of these methods to implement several features at once.

To overcome these problems, this thesis uses a mathematical problem type known for its flexibility: linear programming. The search for a result set of documents pertinent to a given query and fulfulling a series of constraints is transformed into an equivalent linear form. The resulting problem can then be solved by conventional methods, as implemented abundantly by so-called integer linear program solvers such as GLPK[5]. Since the problem of selecting results is inherently discrete, integer linear programming has to be used.

Section 3.1 and 3.2 contain an overview of integer linear programming. Then, in section 3.3, basic independent search is formulated as an integer linear program.

3.1 Linear Programming

In the context of linear programming, the word "program" denotes a problem, and not an implementation, which would be the meaning of "program" in computer science. Similarly, "linear programming" does not refer to a programming style, but to the act of formulating problems as linear programs or solving linear programs.

A linear program is an optimization problem: A value must be found for a variable x such that a certain objective function is maximized. In linear programming, the variable x is actually a vector, and therefore can be considered as multiple variables.

The objective function that is to be maximized (or minimized) must be linear, In other words, it must be a linear combination of the components of the variable. Mathematically, the factors of each component in the objective function can be seen as a vector. The objective function then becomes the scalar product of this vector and the variable.

Furthermore, each linear program contains a set of constraints: inequalities that must not be violated by the result. The objective function must be optimized for all values of the variable that are valid, that is which do not violate any of the inequalities. Inequalities must be linear too, and can always be formulated as $a^T x \leq b$, where a is a vector of the dimension of x and b is a number[1].

When the dimension of x is small, a linear program is equivalent of finding an extreme point in a certain polyhedron. For instance, if $|x| = 2$, the set of x not violating an inequality forms a half-plane. The set of x not violating any constraint thus is the

[1] The notation used here is from linear algebra. $a^T b$ corresponds to the scalar product of the vectors a and b

intersection of several half-planes, and thus a polygon. The solution of the linear program can then be found by search through all corners of this polygon.

In pratice, x can have a dimension in the hundreds or thousands. To solve such linear programs, several algorithms have been developed, of which the best known is the simplex algorithm. Other algorithms known are the ellipsoid method and the interior point method. Implementations of the simplex algorithms are many, and these implementations are used to solve problems in many domains. In this thesis, linear programming will be applied to information retrieval.

3.2 Integer Linear Programming

In addition to the linear constraints, one may require certain variables to be whole numbers. In these cases, special algorithms have to be used such as cutting plane methods or branch and bound methods. Integer linear programming, or mixed integer linear programming if not all variables are integers, is an area of active research, and existing implementations are still evolving, and often of very heterogeneous performance.

Note that this thesis is not concerned with implementing an algorithm for solving linear programs. Instead, different kinds of problems are converted to a form that can be processed by a mixed integer programming solver, which is then solved by third party solvers.

3.3 Description

This section will now explain how independent search can be formulated as an integer linear program.

Each document is represented by a variable, retricted to the values 0 and 1, denoting the absence or the presence of the document in the result set respectively. The score of each document as given by the search engine is used to define the objective function: the sum of the scores of all documents whose corresponding variables is 1.

Constraints can then be added at will, provided they are linear. For instance, if at least half of all results should be in English, the sum of the variables of English documents must be not less than half the number of wanted results. Constraints that are not naturally linear, such as diversity, must be transformed into a linear form.

The core of this thesis thus analyzes the various constraints that can be formulated, how they are transformed into a linear mathematical form, and how the results thus obtained compare to other methods of information retrieval.

3.4 Documents

Since the number of documents typically known to an information retrieval system is normally huge, defining variables for each document would produce too big an integer linear program to be solvable. Therefore, the number documents first has to be reduced to a workable number. To this end, a search engine will be used. The search terms are

passed to the search engine, which returns a number $n > k$ of documents. The number n will be called the initial document count. It can be adjusted: If it is higher, the results will be of better quality, but the optimization will also take longer.

Note that reducing the search space before doing more complex computation is a common pattern in information retrieval, as examplified by [14] and [1].

The set of all documents shall be called Ω. $n = |\Omega|$ shall be the number of documents considered by the linear optimization. This number is known when contstructing the linear program, and is thus considered a constant mathematically, although it can be adjusted by the user. The documents are supposed to be numbered from 1 to n, and are called $s_i \in \Omega$ for $1 \leq i \leq n$. Note that this order of documents does not have to represent decreasing scores. For instance, documents may be retrieved from multiple search engines and not be ordered.

In practice, the number of initial documents will be about 75. Depending on the constraints enabled, it can also get as high as 10,000.

3.5 Objective Function

The objective function must represent the score of the returned document set. In simple independent search without constraints, the score function to be maximized is simply the sum of the scores of documents returned. This same definition is also used here. The score vector $S \in \mathbb{R}^{\Omega}$ is taken as given by the search engine. The objective of the problem thus becomes:

$$\max \quad S^T x \tag{3.1}$$

3.6 Number of Results

In the simplest case, the number of results is a constant. This number shall be called k. To define that exactly k documents should be chosen, the following constraint must be given:

$$\mathbf{1}^T x \quad = \quad k \tag{3.2}$$

If this constraint was omitted, the result would naturally consist of all documents, giving the highest possible score sum. Such search results are of course not useful to the user.

3.7 Results

The result set shall be called $R \subseteq \Omega$, and contains all s_i for which $x_i = 1$.

The previously defined objective function and equality taken together implement independent search, without any constraints.

$$\begin{aligned} \max \quad & S^T x && (3.3) \\ \text{s.t.} \quad & \mathbf{1}^T x = k && (3.4) \\ & x \in \{0,1\}^\Omega && (3.5) \end{aligned}$$

This is the base form of the integer linear program. Other inequalities and variables have to be added to implement various user-level constraints.

3.8 Summary

In this chapter, simple independent search was formulated as an integer linear program. This formulation will be the base for defining additional constraints next.

4 Constraints

The bulk of the information contained in a linear program lies in the constraints. In a linear program, constraints are all linear inequalities of the form $a^T x \leq b$, where x is the document vector, a the document coefficients and b a value. Non-linear inequalities must be transformed into linear ones, sometimes also changing the objective function.

The following sections describe possible constraints based on the document language, categories, document types, document formats, document sources, the document age, length of documents, and diversity.

4.1 Classes

Many constraints are of the form: at least N% of the results should have attribute X. Therefore, it is useful to define classes: a class $C \subseteq \Omega$ is a subset of the document space having a certain attribute.

Classes can be defined in various ways. Many of these classes correspond to some attribute (for instance, being in PDF); other can be specified by the user, such as documents in a specific language. These attributes of documents are meta-information not returned by all search engines. It is therefore necessary to compute this information. Chapter 5 gives details about such meta-information.

For a class C, the class vector $c \in \{0,1\}^\Omega$ shall represent all documents in C with the number 1. A typical constraint can then be formulated as:

$$c^T x \quad \geq \quad pk \tag{4.1}$$

Where c is the class vector, and $0 < p \leq 1$ the proportion of documents that the user wants to be in the class. Note that it doesn't make sense for p to be 0, since the constraint would always be fulfilled.

The classes described in the following subsections give rise to so-called flat constraints. These flat constraints are the simplest constraints to define. They need only one inequality, and can all be turned into soft constraints, as will be described in the section *flat constraints*.

4.1.1 Language

The language of a document is usually well-defined and, as as meta-information, easy to understand for users. Because a language detection algorithm was available during

the implementation of this algorithm, the language will serve as the primary meta-information for constraints and examples.

The general language constraint will look as follows:

$$c_{\text{lang}}^{T} x \geq p_{\text{lang}} k \tag{4.2}$$

Here, c_{lang} represents the documents that are in the "correct" language, as defined by the user. p_{lang} represents the minimal proportion of results that should be in a correct language.

Correct languages can be defined as being a single language, for users wanting results in just one language, or several languages. For instance, a user speaking multiple languages may want results in several of his languages for the most diverse results. In any case, a proportion p_{lang} of the documents must be readable. There can be several reasons for wanting results in a language the user does not understand: documents could still be interesting because the information only exists in this language, or the document is an original text in a foreign language and the user did not know the nature of the results.

As defined above, a proportion p_{lang} of documents must be in correct languages. If not enough documents of the correct languages are in the initial document set, then the problem has no solutions. Such problem instances are called infeasible. Later, when defining *soft constraints*, a way will be introduced for allowing such constraints to be violated, but at a price.

The next subsections describe other flat constraints. They all result in analogous inequalities as the language constraint. Thus they are only described and no formulas are given.

4.1.2 Categories

Some search engines include the concept of categories. Categories form a tree of domains relevant for searching. For instance, "culture" may be a category containing the subcategories "cinema" and "theater".

Categories may be useful as classes when e.g. a search for "Africa" should find documents both about politics and sports.

One method for defining constraints based on categories is described as strict constraints in *Related Work*: Find the number of results in each category, and try to return proportionally as many results from each category. Example: For "Africa", if the Category "Sports" has 30 documents about Africa and the category "Politics" has 50 documents about Africa, then try to find about 37% of documents from the "Sports" category.

In isolation, these constraints are not so useful, since they correspond exactly to search in strict categories. However, in combination with other constraints they can be weighted, and results will be found that cannot be the result of using additional constraints on strict categories.

4.1.3 Document Types

Generally information can be split into different types, such as academic information, news, journal entries and literature. A search for "Paris" should return both information about the history of the city and information about current events.

Types that can be used include:

- Encyclopedic documents. These also include dictionaries and other research tools. Information here is not necessarily current.
- News. Information must be recent and about a recent event.
- Papers. They are of academical nature, often in formats such as PDF or PS.
- Commercial information. Much commercial information exists. This kind of information is sometimes filtered out in information retrieval systems.
- Official sites of organizations and entities. They contain official information and press releases.

The determination of type can be done automatically, or can be given by the search engine.

4.1.4 Format

Documents come in numerous formats, ranging from text to HTML and PDF. A good search engine will ensure the diversity of formats, especially since certain formats may present accessibility problems in some situations. Some users may exclude some formats, such as PDF files.

Format can be discriminated by filename, or by analyzing the content. Since at least a filename will be known in information retrieval systems, implementing a format constraint is thus always possible.

Usually, users will expect to not get results in formats they excluded. Thus, this constraint could be set as strict, that is, not soft, as described in the next section, or with a very low weight.

4.1.5 Sources

Documents from the same source, that is, documents retrieved from the same website or the same domain may form a class. These classes may be useful for limiting the number of documents retrieved from a same source.

In the implementation, sources are determined by analyzing the filenames. In general, the domain name is extracted from the URL and the top level domain is read. It represents the country of origin and thus the source of the document. If the top level domain is not a country top level domain, it can be used to classify the documents into categories such are commercial, organizations, etc. These categories represent sources too.

4.1.6 The Pareto Optimal Set

According to [4], the Pareto optimal set according to predefined objective contains good candidates for the result set. Requesting that documents come from this set, or at least partly from this set will increase the diversity of the results. The Pareto optimal set can be calculated in not more than $O(n^2)$ time, where n is the initial number of documents.

Thus, one could want a certain proportion of results to come from this set. The diversity, which is inherent in many algorithms finding solutions in the Pareto optimal set, would not be given here. Thus, this type of constraint would have to be combined with the diversity constraint.

4.1.7 Partitions

Classes may be generated by partitionning Ω itself. In this case, constraints may be formulated for all partitions. Example: If the user speaks English, French and German, he may be interested in as many results for each language. This may be combined with pseudo-variables (introduced in the section *Soft Constraints*). Partitions may be thought of as diversifying the results. To reach diversity, the number of partitions should be about 3 or 4.

4.1.8 The Totality of Documents

In some cases, constraints can be used which use all documents in their formulation. In this case, we set $C = \Omega$. The restriction to the number of documents is also of this form, but since it is necessary in all cases, it is not described here.

Some useful constraints defined on the whole document set include the age constraint and the total length constraints, in which additional weights are used. They are described after the weights have been defined.

4.2 Weights

By giving a coefficient c whose values can only be 0 and 1, constraints in the previous sections gives a bound on the number of results from a specific class. These constraints shall be called “unweighted”.

The factor given to each variable can also have more diverse values. In this case, the constraint shall be called “weighted”.

4.3 Age

When the age of the documents is known, it can be used to define an additional constraint. The age is usually given in days, and the users will want newer documents over older documents.

In the simplest case, a threshold is set for the age, and a simple flat constraint is defined for it. In this model, no difference is made between document just barely above

the threshold age or much older. To consider the difference between all age values, another approach can be taken: limit the average age.

When A_i is defined as the age of document i and a is the maximal average age, the inequality

$$A^T x \leq ka \tag{4.3}$$

will limit the average age of the results. If this inequality is used on conjunction with a varying number of results, k must be replaced by the effective number of results, $\mathbf{1}^T x$, thus giving

$$\begin{aligned} & A^T x && \leq \mathbf{1}^T xa && (4.4)\\ \Leftrightarrow \quad & A^T x - \mathbf{1}^T xa && \leq 0 && (4.5)\\ \Leftrightarrow \quad & (A^T - \mathbf{1}^T a)x && \leq 0 && (4.6) \end{aligned}$$

The last form corresponds to the standard form of iniequalities in linear programs.

The age constraint can thus be set to “hard” or to “average”.

4.4 Length

The length of documents represents meta-information usually available to information retrieval systems, and meaningful to the user: Documents should not be too large generally, but some large documents are of course useful. Also, some mediums such as portable computers may have a limited bandwidth where huge documents are not wanted. Thus, an upper bound may be defined on the total length.

When the length of the documents is given by

$$l \in \mathbb{R}^{\Omega} \tag{4.7}$$

then the total length of results given takes the form of $l^T x$. An upper bound on this total length can thus be given by

$$l^T x \leq L \tag{4.8}$$

The measure of length can be chosen arbitrarily, and may be one of the following:

- The number of bytes, characters, words or lines in the document. This corresponds roughly to the length visible by the user. An upper bound would limit the amount of text to read for the user.
- The number of images.

- The number of links.

For the last two length measures, one could envisage to use a lower bound: Providing results with a guaranteed number of images could make for more real websites as results, instead of just text.

4.5 Soft Constraints

Constraints are by definition strict and must always be fulfilled by results. It is however possible to generate "soft constraints" by assigning a penalty for violating the constraint, or for deviating from the ideal value. This method is sometimes called "goal programming".

The constraint

$$a_i^T x \leq b_i \tag{4.9}$$

Can be made into a soft constraint by adding a new variable y_i into the left side, which can take values greater than zero when the constraint can otherwise not be met. This variable must never be negative. This variable is then added to the objective function with a negative coefficient to serve as a penalty. This coefficient is fixed but can be chosen by the user. It is called the penalty of the constraint, and noted λ_i. The variable y_i is called the penalty variable.

The simple constraint is thus changed into:

$$\begin{aligned} a_i x + y_i &\leq b_i && (4.10) \\ y_i &\geq 0 && (4.11) \end{aligned}$$

And from the objective function is substracted the penalty:

$$\max \ \ldots - \lambda_i y_i \tag{4.12}$$

In fact all constraints can be made soft, and this is the default in the implementation.

4.6 Soft Goals

A goal in a linear program is an equality. For instance, the number of results can be fixed by using a goal. The general goal takes the form of

$$a_i x = b_i \tag{4.13}$$

Goals can be formulated as two identical constraints up the direction of the inequality:

$$a_i x \leq b_i \tag{4.14}$$
$$a_i x \geq b_i \tag{4.15}$$

A goal is thus regarded as a special kind of constraint. To make the equality soft, a penalty variable is added as before:

$$a_i x + y_i = b_i \tag{4.16}$$

Now y_i can be negative, and we must use its absolute value in the objective function:

$$\max \ldots - \lambda_i |y_i| \tag{4.17}$$

But since using the absolute value violates linearity, the following equivalent form is used:

$$\max \ldots - \lambda_i y_i \tag{4.18}$$
$$y_i \geq a_i x - b_i \tag{4.19}$$
$$y_i \geq b_i - a_i x \tag{4.20}$$

4.7 Soft Flat Constraints

All flat constraints can be turned into soft constraints as just described. In the general form, an equation of

$$c_i^T x \geq p_i k \tag{4.21}$$

will become

$$c_i^T x + y_i \geq p_i k \tag{4.22}$$

Thus, each flat constraint will have the following options for the user:

- The constraint can be disabled, set as a soft constraint, or set as a hard constraint.
- The properties can be chosen which are covered by this constraint, for instance the languages for the language constraint. This choice will set c_i.
- The proportion p_i of documents the user wants having this property.

- The weight λ_i of the constraint, realized as the negated weight of the penalty variable in the objective function.

4.8 Soft Number of Results

By default, the number of results is fixed using the inequality $\mathbf{1}^T x = k$. As other goals, it can be made soft. There are two options for defining the penalty variables:

- Define the penalty to be given when the number of results deviates in any direction. This is the simplest option.
- Give only a penalty when the number is too high. Since the score sum in the objective function will make the number of results tend to more than wanted, the penalty for having less results than wanted may seem to be superfluous. However, some cases may arise were the solver would return zero or one result without it: when the diversity is to be maximized, it could be simpler to just remove documents from the results to get more diverse results. Thus, a penalty for not having enough results is necessary.

When the number of results is fixed, this constraint is usually said to be disabled, since for the user, no special features were enabled beyond independent search.

4.9 Diversity

A set of results is called diverse when its documents differ from each other. The diversity of the result set is thus a measure based on the similarity of chosen documents. Thus, a similarity measure is needed, or, equivalently, a distance measure. The goal of defining the diversity is to maximize it.

First a distance measure has to be found that gives a distance between each pair of documents in the initial document set. Then, a mapping has to be defined from the distance matrix to a diversity measure of the result set. Two such diversity measures will be presented in equations 4.23 and 4.24.

4.9.1 Distance Measures

Measuring the distance between two documents is a big topic in itself. In this implementation, several measures are taken and combined. First, the meta-information of two documents is compared, and for each piece of meta-information, a number is calculated: zero if they differ, one if they are equal. Meta-information used in this way may be the language, the source, the length, the category, or anything else that is available.

The other distance measure used is a value based on what is called the cosine similarity. To calculate it, the frequency of all words in the documents are calculated. These frequencies are taken as two vectors, one for each document. These vectors are then multplied, and the result is higher when the documents contain the same words. This

value is substracted from an upper bound of the cosine similarity to give a distance measure.

All these distance measures are then combined using custom weights to give an overall distance measure. As an additional feature, the weights of each measure can depend on the constraints enabled. This does not always make sense: Even when the user did not care about the languages of the result, providing results in diverse languages is probably better than providing results in all the same language.

The distance between the documents i and j will be denoted $\Delta_{ij} = \Delta_{ji}$ for $i \neq j$.

4.9.2 Modeling Diversity

Once the distance between documents is defined and calculated, it must be used to define the diversity. Between the k documents chosen, $\binom{k}{2}$ distances are known and must be combined to give a diversity measure. Two different kinds of diversity are considered here: the average distance and the minimum distance.

$$\theta_{\text{avg}} = \binom{k}{2}^{-1} \sum_{i<j} \Delta_{ij} \tag{4.23}$$

$$\theta_{\text{min}} = \min_{i,j \in R} \Delta_{ij} \tag{4.24}$$

[1] Takes into account for the diversity only those features specified in the query. This approach is not taken here.

Average Distance

Since the diversity should represent the distance between results, it can be simply defined by taking the average over the distances between all $\binom{k}{2}$ pairs of documents. Thus,

$$\theta_{\text{avg}} = \binom{k}{2}^{-1} \sum_{i<j} \Delta_{ij} \tag{4.25}$$

$$= \binom{n}{2}^{-1} \sum_{i<j} x_i x_j \Delta_{ij} \tag{4.26}$$

This variable θ_{avg} will have to be integrated into the objective function. Unfortunately, this is not linear, since the product of the two variables x_i and x_j is taken.

It can be transformed into an equivalent linear form by introducing pair variables. For all $i \neq j$, e_{ij} shall be defined as follows:

$$e_{ij} = e_{ji} = x_i x_j \tag{4.27}$$

$$\tag{4.28}$$

To define e_{ij} linearly, the equations

$$e_{ij} \leq x_i \tag{4.29}$$
$$e_{ij} \leq x_j \tag{4.30}$$

can be used. The average distance is then defined as:

$$\theta = \binom{n}{2}^{-1} \sum_{i<j} \Delta_{ij} e_{ij} \tag{4.31}$$

which is only linear if n is constant. Thus, maximizing the diversity using the average distance necessitates quadratically many inequalities in the number of initial documents, and can only be enabled when the number of results is fixed.

Minimum Distance

To overcome the problem of the number of results having to be constant and the huge number of additional inequalities needed when using the average distance in diversity, the minimum distance can be used. Inserting the minimum function into the objective function is of course not linear, but the same result can be achieved by defining the following inequality for each pair $i < j$:

$$\theta_{\min} \leq \Delta_{ij} x_i x_j \tag{4.32}$$

Still not linear, we can use an equivalent form which needs an upper bound $\Delta_{\max}$ for the distance to be defined:

$$\theta_{\min} + \Delta_{\max}(x_i + x_j) \leq 2\Delta_{\max} + \Delta_{ij} \tag{4.33}$$

Now $\theta_{\min}$ can be inserted into the objective function with a factor of λ_{dist}:

$$\max \quad \ldots + \lambda_{\text{dist}} \theta_{\min} \tag{4.34}$$

4.10 Summary

In this chapter, constraints have been presented that implement various extensions to the traditional independent search. Each constraint was formulated as additional inequalities and variables to the basic integer linear program. Taking all these inequalities and variables together will provide a big complex mixed integer linear program, which must be solved by an integer linear programming solver.

5 Implementation

The constraints presented in the previous sections were implemented partially in the course of this thesis. The implementation is in form of an agent platform called the Post Optimization Platform. This chapter will describe this implementation.

5.1 Framework

The implementation takes the form of an agent-oriented application using the JIAC platform[7]. The optimization itself is provided as a service by an agent called the manager. This service is used by another agent, the GUI agent, which displays a graphical user interface. The services are described in JIAC's own ontology and knowledge description language Jadl. The agents are implemented in the programming language Java.

A session of using the Post Optimization Platform is depicted in figure 5.1. In the graphic, the GUI and the Manager are part of the implementation. The search engine is an external tool.

Figure 5.2 shows the post optimization platform with the GUI agent and the manager agent in action. As usual when using the JIAC framework, messages sent between agents are represented graphically.

5.2 Search Engines

The initial set of documents represented by Ω must be acquired from a search engine. The search phrase entered by the user is given to the search engine, which provides a list of documents used as the initial set. The search should not be restricted to a certain type of document; the documents returned by the search engine should be as diverse as possible, or the different constraints would not make much sense. Also, the search engine must be able to provide a big number of results. This number must be much greater than what is usually shown as results in search engines. For instance, having just 20 documents as an initial set to choose from would be pointless. Thus, several hundred are needed.

For the constraints to be effective, as much meta-information on the documents is needed. For instance, wanting a certain proportion of documents in a certain language does not make any sense if we don't know the language of documents. Meta-information can come from two sources: Either directly from the search engine, or by producing it locally. Some search engines provide meta-information such as the creation date of a document. Other meta-information can be inferred easily, such as the country of

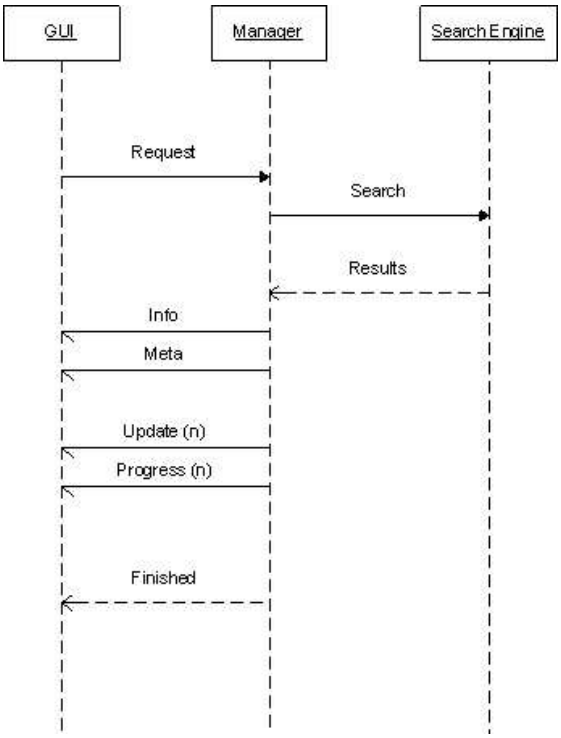

Figure 5.1: Sequence diagram of one query being processed

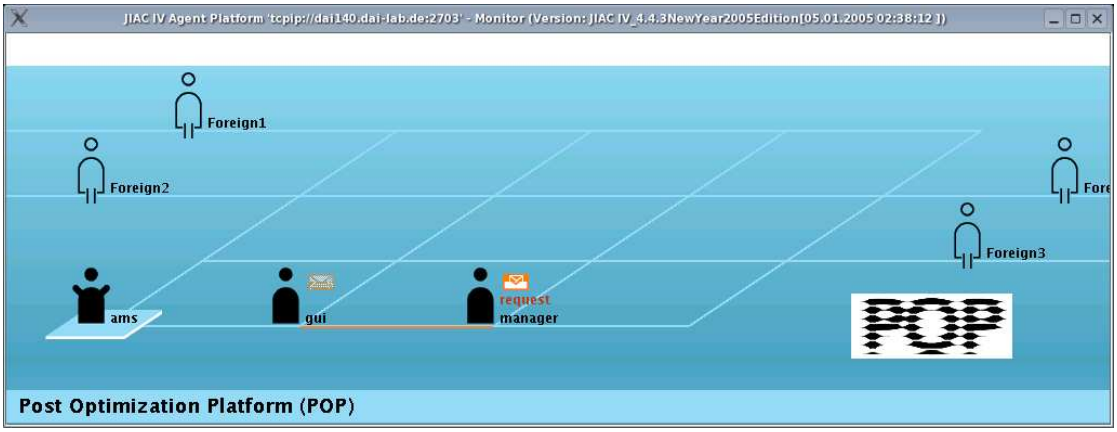

Figure 5.2: The Post Optimization Platform

origin: Knowning the URL of a document is enough for extracting the top level domain which indicates the origin. Some meta-information can be extracted from the content of the document itself. As an example, the length of the document can be important information. Also, the language, if not given by the search engine, can be detected. But note that the language can also be detected by analyzing the summary, which was used in this implementation and works in practice.

The implementation is written to support several search engines, of which two are used at the moment:

- A search engine developed at the DAI laboratory, implemented as an agent service and using self organizing maps (SOM)[6]. This search engine itself aggregates results from conventional search engines.
- Instead of a search engine, the second option is a random generator of results. The URL, the title and the summary of each document is taken from a previously defined list of texts, and the search phrase in inserted to make the results look relevant. Texts are in English, German and French.

Documents are returned as URLs and associated basic meta-information: the title, the summary, and a score. Depending on the search engine used, the communication between the post optimization platform and the search engine can be through JIAC services (for instance when using self organizing maps), or simply through a Java API.

5.3 Attribute Collection

Since this approach needs several document attributes such as the language which are not usually returned by search engines, these have to be collected.

The language of each document was detected by tools written as part of the PIA project. The detection thus made is based on identifying stopwords for each language. The languages recognized were English, French and German.

The format of the document is inferred from the URL. URLs that have no known ending (such as ".ps") were supposed to be in HTML.

What is called the source in this work is the country of origin of the document. Countries were identified by their two-letter code.

5.3.1 Random Attribute Generation

Other meta-information, such as the document type and age were not available. For testing purposes, these were generated using a random generator. This technique was tuned to ressemble actual data from search engine results. The attributes concerned are the title and the summary, the age, the URL, the format and the source.

5.3.2 Bonuses and Maluses

Since not all meta-information is known by the search engine used, the meta-information generated is used to adjust the score of documents. This adjustment takes the form of

bonuses and maluses given to documents that match or do not match each constraint. These adjustments are only made when the corresponding constraint is enabled.

5.4 Options

The following constraints are implemented:

The number of results can be a hard constraint, or a soft constraint. The default is the soft constraint. For the soft constraint, a penalty is given when there are too much documents, or when there are not enough documents.

In addition to being disabled, diversity can be set to average and minimum. When "average" is selected, the average distance between chosen documents is maximized, using a weight set by the user. Average distance can only be used when the number of results is fixed, i.e. when the number constraint is hard. When the minimum distance is chosen, the minimum distance between two chosen articles is maximized. This works even when the number of results is not fixed.

The age can be set to act like a hard flat constraint, described in the next paragraph, or as the a soft constraint concerning the average age. When "average" is chosen, the average age of documents chosen is calculated, and a penalty is given when is exceeds the user setting.

All other constraints are "flat" constraints, in that they prescribe that a certain proportion of results shall have a certain property, and give a penalty when the proportion is less. The factor used for the penalty is choosable in the user interface and is called the "weight". All flat constraints can be disabled individually. When enabled, each flat constraint can be set as a hard or as a soft constraint, with a configurable weight.

The flat constraints are the language of documents, their source, the document format and the type.

5.5 Integer Optimization

Programs that solve linear programs are generally called *solvers*. They come in two forms: standalone programs that take a file containing the problem as input, or programming libraries. In the implementation, the GNU Linear Programming Kit (GLPK)[5] was used. GLPK implements both a standalone and library interface. The standalone interface was used. GLPK supports several types of input files. The format chosen was the Cplex format, used by the well known solver Cplex.

GLPK uses the branch and bound algorithm for solving integer linear programs. GLPK was modified to output intermediary values of variables, which it normally does not. As all solvers, GLPK outputs upper and lower bounds for the optimal value of the goal function. These come in regular intervals, and their values are displayed in realtime in the graphical user interface.

The following code section shows a sample of a linear program read by GLPK in Cplex format.

```
Maximize
  +109 x_0+111 x_1+90 x_2+89 x_3+89 x_4+93 x_5+93 x_6+102 x_7+94 x_8+90 x_9+100
  x_10+89 x_11+86 x_12+97 x_13+88 x_14+103 x_15+92 x_16+85 x_17+82 x_18+78
  x_19+80 x_20+81 x_21+83 x_22+73 x_23+82 x_24+80 x_25+75 x_26+76 x_27+58
  x_28+65 x_29+70 x_30+65 x_31+71 x_32+54 x_33 - 40 y_language  - 40 y_source  -
  40 y_format  - 40 y_type  - 120 y_number - 40 y_age

Subject to
  y_language >= 0
  y_source   >= 0
  y_format   >= 0
  y_type     >= 0
  y_number   >= 0
  y_age      >= 0
   + x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 + x_10 + x_11 +
  x_12 + x_13 + x_14 + x_15 + x_16 + x_17 + x_18 + x_19 + x_20 + x_21 + x_22 +
  x_23 + x_24 + x_25 + x_26 + x_27 + x_28 + x_29 + x_30 + x_31 + x_32 + x_33 -
  y_number <= 7
   + x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 + x_10 + x_11 +
  x_12 + x_13 + x_14 + x_15 + x_16 + x_17 + x_18 + x_19 + x_20 + x_21 + x_22 +
  x_23 + x_24 + x_25 + x_26 + x_27 + x_28 + x_29 + x_30 + x_31 + x_32 + x_33 +
  y_number >= 7
   +x_0+x_1+x_7+x_8+x_13+x_22+x_25+x_26+x_32 + y_language >= 5
   +x_5+x_12+x_15+x_16+x_19+x_21+x_30 + y_source >= 3
    +375 x_0 -28 x_1 +100 x_2 +265 x_3 +107 x_4 +469 x_5 +297 x_6 +874 x_7 +475
  x_8 +189 x_9 -15 x_10 +10 x_11 +19 x_12 +138 x_13 +113 x_14 -28 x_15 +153 x_16
  +235 x_17 +1084 x_18 +69 x_19 +879 x_20 +187 x_21 +73 x_22 +202 x_23 -26 x_24
  -2 x_25 +396 x_26 -21 x_27 +239 x_28 +34 x_29 +235 x_30 +641 x_31 +133 x_32
  +610 x_33 -y_age <= 0
   +x_0+x_6+x_7+x_9+x_10+x_11+x_13+x_14+x_15+x_16+x_17+x_18+x_20+x_21+x_22+x_23+
   x_24+x_26+x_27+x_29+x_30+x_31+x_32+y_format >= 6
   +x_0+x_1+x_2+x_6+x_8+x_9+x_12+x_14+x_17+x_18+x_21+x_22+x_23+x_24+x_27+x_29+
   x_30+x_31+x_32+x_33+y_type >= 7


Binary
  x_0
  x_1
  x_2
  x_3
  x_4
  x_5
  x_6
  x_7
```

```
  x_8
  x_9
  x_10
  x_11
  x_12
  x_13
  x_14
  x_15
End
```

5.6 Diversity

Diversity can be set to use the minimum distance or the average distance. As dictated by how the average distance works, this setting can only be used when the number of results is fixed. Depending on the setting, one of the equations 4.23 and 4.24 is used.

5.7 Results Ordering

When the search results are presented to the user, an order must be chosen. Usually, they are sorted by decreasing similarity to the query. To the user, this gives the impression of seeing the beginning of the long list of *all* documents ordered by score.

For displaying the results of a search with constraints, the order is not so simple to define. The score can be used as before, but it must be decided whether the bonuses are taken into account or not. Also, the constraints could be used to provide an order. For instance, first display the documents that fulfill the most important constraint, then those that don't. However, it is not clear which constraint should be considered more important than others.

To use information about diversity in the sorting of results, one could try to minimize to sum of distances of documents adjacent in the displayed results. This corresponds to solving the travelling salesman problem.

In this implementation, the results are simply ordered by decreasing score, without considering bonuses.

5.8 Intermediary Results

Results are displayed as soon as they are found. Typically, the solver will output intermediary values, along with an upper and lower bound on the optimal result. These results are displayed in the GUI. Thus, the list of documents chosen changes during the search until an optimal solution is found. The value of the penalty variables is also diplayed while running. These include the average or minimum distance distance, the average age, the number of results, and for each flat constraint, the percentage of documents chosen that fullfil the condition setup for the constraint.

5.9 User Interface

The graphical user interface, as shown in figure 5.3, is divided into two parts: the search area and the results area. In the search area, the user enters search terms along with constraints. Each constraint is represented on a single line. Radio buttons allow each constraint to be set to hard, soft, or disabled. The options such as the language and the document formats can be chosen by the user. Clicking on "Search" will start the search.

When results are available, they are displayed in the results area. The result area is a big table, containing information about one document in each line. For each document, the title, the summary and the URL are displayed. Also all known meta-information is shown. During the search, the documents currently in the "chosen" set are highlighted. This happens synchronously with the solver running in the background. In the header line of the table, a summary of the constraints enabled by the user is shown in the appropriate column of meta-information. Meta-information that is "correct" according to the corresponding constraint is highlighted in bold. In the current implementation, all documents from the initial set are displayed. In a system meant for users instead of testers, only chosen documents would have to be displayed.

For instance, if the user selected that 80% of his results shall be in German using a soft constraint, the header line will display "($\sim$ 80%)", where the tilde indicates that the constraint is soft. When intermediary results are available, the proportion of actually chosen documents is also displayed. For instance, if in the search described previously, only 71% of results are in German, this number will be indicated just below the line with "($\sim$ 80%)".

While the search progresses, the progress in indicated by showing the percentage difference between the current upper and lower bounds for the optimal solution. When this percentage reaches 0%, the search is finished. The user can then change some options, add or remove constraints, and start the search again. New searches can also be started when a search is running. The search running is then simply interrupted.

When results arrive, information about constraints and variables is displayed. This information includes:

- The score of each document. Both the raw score as returned by the search engine and the score adjusted using bonuses is displayed.
- For soft constraints, the proportion of documents chosen having the requested property is shown. For instance, if the user wanted 80 percent of results in his language, the number of results in the user's language will be displayed alongside the requested percentage. The user can then assess how good the results fulfill his request.
- The diversity is displayed as a numeric value.
- The degree to which the optimization has progressed. The progress is measured in the percentage difference between the upper and the lower bound on the optimal value of the objective function. The lower bound corresponds to the value of the objective function using the current results. This number is given by the solver.

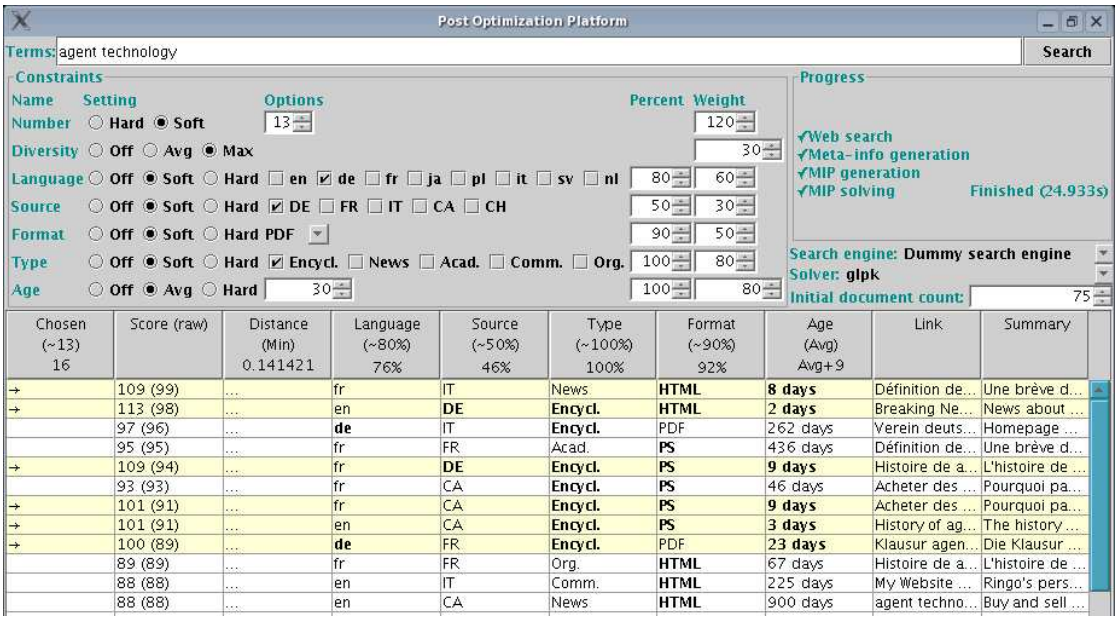

Figure 5.3: Screenshot of the graphical user interface, showing the default settings

Figure 5.3 shows the realization of the constraints in the graphical user interface that was implemented.

5.10 Reusing Previous Results

When the results of the optimization are displayed in the GUI, the query and the constraints that led to these results are not removed. They can be further changed by the user, and the new problem is solved. Thus, it is easy to experiment with various settings. The default settings and especially weights are all the result of such experimentation.

5.11 Summary

A graphical user interface and a constraint generator were implemented. These cover some of the constraints presented in chapter 4, but not all. The next step is now to test the implementation, evaluate the results, and compare integer linear programming search with other information retrieval methods.

6 Evaluation

In this chapter, tests are made to evaluate the implementation and the algorithms described. Then, comparisons are made to other methods described as related work. Results given in this chapter will be summarized in the conclusion.

6.1 Tests

In the following, the implementation is tested, and the results are compared to the expected outcome.

Except when noted otherwise, the default settings as described in the previous chapter are used. A phrase is entered into the search bar, the constraints are adjusted as described or kept at the default values, and the search is started. Tests were made using the random text generator as a search engine, so the phrase used for searching is not relevant. The tests were made on a PC with a 900 megahertz processor and half a gigabyte of operational memory.

6.1.1 Scalability

How many initial documents can be used? The more documents are present at the beginning, the better the quality of the final result will be. First, taking not enough documents in the initial set could exclude some documents that fulfill a constraint. Such documents may not always have a top score, but are still relevant for fulfilling the user's constraints. Also, constraints such as diversity will be more meaningful.

In an implementation, one should always try to use the highest number of results possible. Because using too many initial documents may make the search too slow, this test explores the scalability of the optimization. All settings remain in the default state. That is, all constraints are enabled, the flat constraints are set to be soft, and the diversity is set to maximize the minimum distance. The search phrase used is "agent technology". In the test, the runtime is measured against the number of initial documents. Since changing the initial number of documents will change the results themselves, each test was done four times and the average runtime is shown. Figure 6.1 shows the results of the test.

Search using just 20 or 30 initial documents takes a few seconds, but this runtime is probably more due to the overhead of the implementation than to the optimization itself. With over a hundred initial documents, the runtime already increases to over a minute. One can expect the runtime to increase further exponentially if more initial results were used. However, such huge runtimes would already be too big in practice.

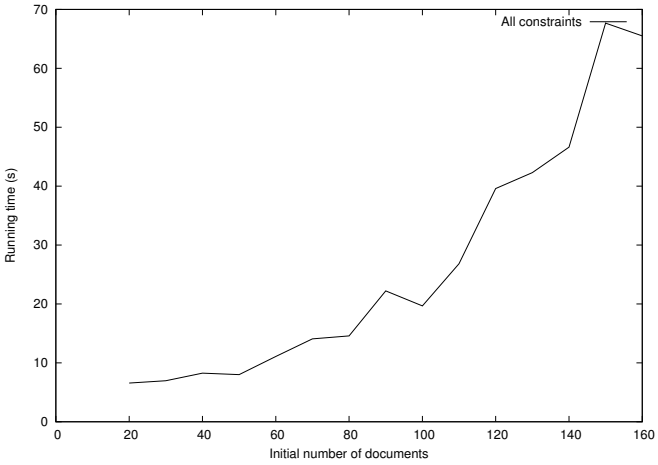

Figure 6.1: Running time of optimization in function of the number of initial documents, using the default settings

As will be shown in the next tests, the runtime can also be influenced by the weights of the constraints, especially by the weight of the diversity constraint.

6.1.2 Scalability without Diversity

As just has been seen, the diversity constraint accounts for the greatest runtime penalty. In this test, diversity is disabled, and all remaining constraints are enabled. The runtime is measured against the number if initial documents. The initial number of documents is varied from 20 and increasing until it took too long or until the solver ran out of memory.

Figure 6.2 gives the results of the test. The most initial documents that could be used was 10,000. After this, runs took too long to be useful (and were aborted), or the computer ran out of memory. Using 10,000 initial documents thus seems to be a good number when diversity is not wanted: Results will be fast and many initial documents can be used, even when for instance results from multiple search engines are aggregated.

Note that the small variations are due to the fact that only one run was made for each test. These fluctuations are however not significant.

6.1.3 Scalability with Average Distance Diversity

Optimizing the average distance between chosen documents is expected to take longer than optimizing the minimum distance. When the diversity is set to use the average distance, the number of documents must be fixed. All other constraints can be enabled however. In this test, all these other constraints are enabled. Starting with 20, the initial number of documents is increased until the runtime is too high to be useful, or until no memory is left on the computer. For each initial count, three runs were made with different sets of randomly generated documents.

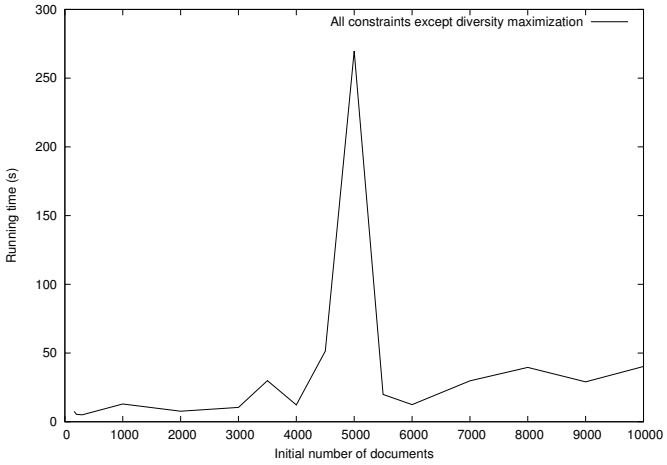

Figure 6.2: Running time of optimization in function of initial document count, without diversity maximization

17.2%	13.8%	10.3%	8.5%	5.9%	4.8%	2.3%	0.7%	0.0%

Table 6.1: Progress of optimization using the default settings

Figure 6.3 shows the average runtime of these three runs in function of the initial number of documents. Even using a few hundred documents, the runtime gets high quickly. It is uncertain whether the average distance should even be used in practice.

6.1.4 Intermediary Results

This test shows intermediary results in sample runs. The solver outputs the precision of the intermediary results in regular intervals. The following tests were made in the default settings with varying diversity settings and initial document count.

With Diversity Based on Minimum Distance

Figures 6.1, 6.2, and 6.3 show the progress of the quality of intermediary results first in the default settings, then doubling the initial number of documents to 150, and then doubling the diversity weight to 60.

Results show that even when optimization is fast intermediary results are usually available. Also, the precision of the first results available varies. It can be as low as under ten percent, at 25%.

8.6%	8.6%	7.5%	5.8%	5.3%	3.9%	3.3%

Table 6.2: Progress of optimization using 150 initial documents instead of the default 75

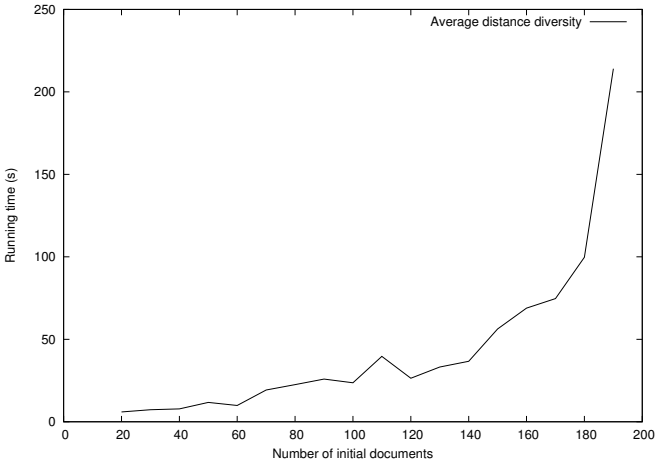

Figure 6.3: Running time of optimization when using diversity based on average distance and all other constraints are enabled

25.5%	25.5%	19.6%	16.1%	12.6%	12.6%	10.2%	6.0%	0.0%

Table 6.3: Progress of optimization using 150 initial documents and a diversity weight of 60, the default being 30

Without Diversity

In this run, the diversity constraints were disabled and 5000 initial documents were used. Table 6.4 shows the results. The results do not differ by much from the pattern observed when using diversity based on minimum distance.

With Average Diversity

This test shows how intermediary results behave when diversity is set to average distance. 40 initial documents were taken, and the weight of the diversity constraint was set to 200. Figure 6.4 shows the results graphically. The single run took several hours, and was run over the night. It shows that initially, the quality gets better repidly. Then, the quality only gets better in small steps for a long time. Near the end, the last fifty percentage points are gained fast. This behavior, first fast, then slow, then fast again can be observed in other tests too. No explanation for this behavior is given here.

This example also shows that the precision of intermediary results can be as high as several hundred percent.

17.9	10.1	5.8	3.5	3.1	1.6	1.2	1.2	0.1	0.0

Table 6.4: Intermediary results of optimization using 5000 initial documents and no diversity

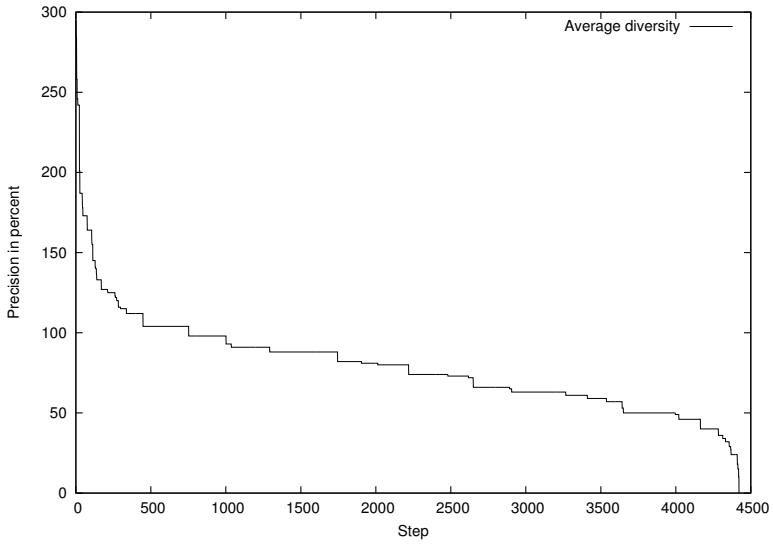

Figure 6.4: Approach to the optimal result with time in a very long run

6.1.5 No Constraints

If no constraints are enabled, with the exception of the number constraint, then integer linear search behaves like independent search as implemented by conventional search engines. Thus, integer linear search could be used by search engines, with all constraints disabled by default, and allowing users to enable constraints one by one. In this case, it is unnecessary that n be greater than k, Setting $n = k$ will give the same results. In fact, just returning the k documents will give the same results.

6.1.6 Languages

In these tests, only the language constraint was enabled, first as a hard constraint, then as a soft constraint.

Language as a Hard Constraint

When only the language is enabled as a hard constraint, the result will be the $p_{\text{lang}}k$ first documents in the correct language, and the rest the $1-p_{\text{lang}}k$ best documents that were not taken, which may or may not be in the correct language.

Languages as a Soft Constraint

Setting the language constraint as hard will of course give the simplest results. Behavior of optimization gets interesting when the constraint is made soft.

As a general rule, the following holds: If the penalty is high, the proportion of documents in the correct language tends to 1. If the penalty is low, the proportion will tend

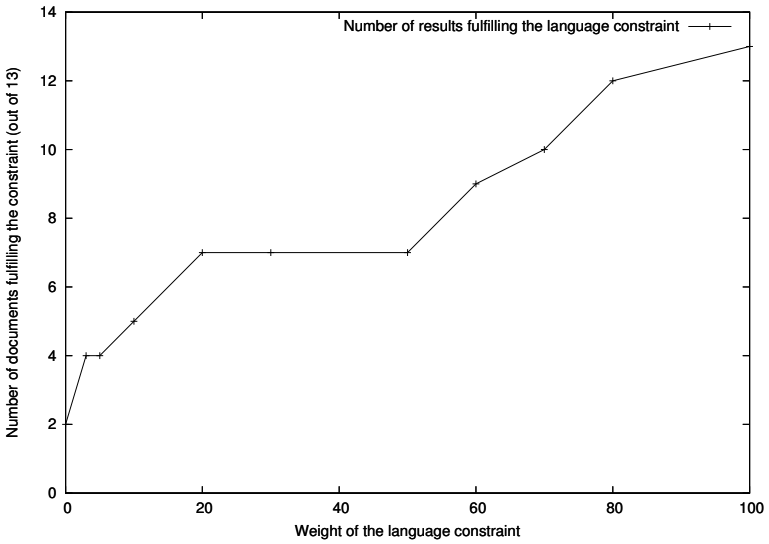

Figure 6.5: Proportion of results fulfilling the language constraint using varying weights

to the proportion of documents in the correct language in the search result without a language constraint.

To test this case, the same query was made with a weight varying for the language constraint. The proportion of documents wanted in the correct language is set to 100 percent. Only one language is chosen. Figure 6.5 shows the results. As expected, high penalties give the same result as a hard constraint: 100% of articles on the correct language. Lowering the weight allows more and more documents in other languages until, when the weight reaches 0, only two documents are in the correct languages. A weight of zero is not equivalent to disabling the language constraint, because even when the weight is zero the bonuses are still enabled. Choosing a weight between these two extremes will give various amounts of documents in the desired language. This case shows that it absolutely makes sense to try to adjust weights to find good values such as to find a balance between documents with high scores and documents in the correct language.

The source, format and type constraints, being flat constraints, behave similarly to the language constraint and are thus not tested explicitely.

6.1.7 Diversity Using Average Distance

Setting the diversity to maximize the average distance between documents produces by far the longest runs of the optimization. Thus, this test will try to show how the weight of this constraint influences the resulting average distance between documents, and how the runtime is affected.

All constraints except the diversity constraint were disabled. To avoid very long runtimes, only 30 documents are used as the initial set, and only seven documents are

Weight	Diversity	Time (s)	First
30	1.776137	8.468	7
40	1.776137	13.596	7
50	1.776137	24.558	7
60	1.776137	53.098	7
70	1.776137	87.257	7
80	1.776137	184.317	7
90	1.837337	317.085	5
100	1.837337	642.165	5

Table 6.5: Using average distance diversity with increasing weights

searched.

Table 6.5 shows the average distance (as "Diversity") and the runtime. The column "First" denotes how many of the documents were continguously from the top. Disabling diversity would thus give a value of seven for this column.

The results show that the runtime increases with increasing weight. This is to be expected since with a low weight, taking low-score documents will not even be considered, since the penalty for including them would be lower than could be gained by increasing the diversity.

As far as the diversity itself is concerned, the results show not much change, even with high weights.

6.1.8 Diversity Using Minimum Distance

This test is identical as the one before, except that the diversity was set to maximize the minimum distance between any two chosen documents. Empirically, the runtime should be much less.

Again, the only constraint enabled is the diversity, this time set to "minimum". As before, seven from 30 documents are searched. The weight of the diversity constraint was incremented from 30 to 110 in steps of 10. Table 6.6 shows the number of contiguous best results, the diversity achieved (which here represents the minimum distance between any two chosen documents), and the time taken.

The runtime increases as expected, and is indeed less than when using the average distance as a diversity measure. The diversity of the result increases by a big step when the weight of 80 is passed. The behavior of the results changing abruptly at one weight, and not changing before or after that weight are typical and can be observed in other tests described here.

6.1.9 Runtime of Diversity Maximization

This test compares the runtime of both distance maximization modes, in function of:

- The initial number of articles

Weight	First	Minimum Distance	Time (s)
30	13	0.14142	22.728
40	13	0.14142	34.245
50	13	0.14142	45.241
60	13	0.14142	89.976
70	13	0.14142	156.921
80	6	2.39749	424.313
90	6	2.39749	320.187
100	6	2.39749	272.614
110	6	2.39749	730.745

Table 6.6: Minimum distance diversity with increasing weight

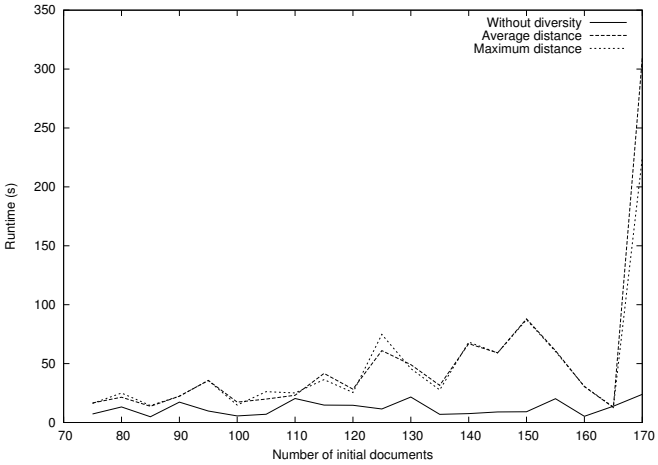

Figure 6.6: Runtime of optimization using all three diversity modes and a varying number of initial documents

- The penalty of the distance constraint
- The relative penalty of the distance constraint and a flat constraint

6.1.10 Runtime of Diversity in Function of Initial Document Count

The runtime is measured against the initial number of documents. Runtimes are in seconds. Figure 6.6 shows the results.

6.1.11 Two Flat Constraints

In this test only the two flat constraints language and source are enabled as soft constraints. The weight of the language is constant (70), and the weight of the source is changed. The proportion of good documents for each constraint is shown.

Both constraints are set to a proportion of 100%. The system was configured to return 13 from 100 initial documents. Results can be seen in figure 6.7.

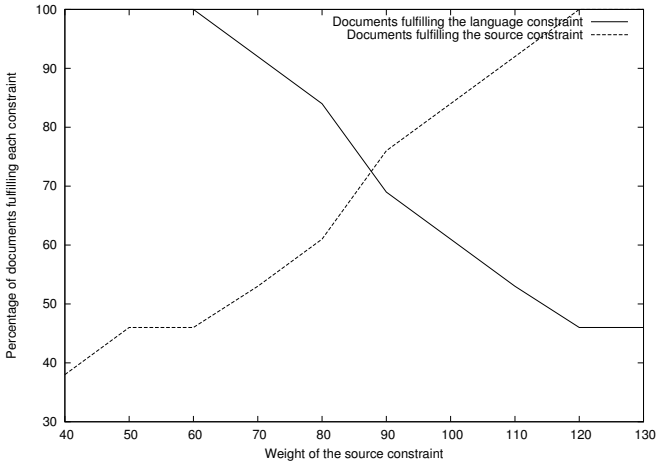

Figure 6.7: Trading off two flat constraints

Constraint	Weight	Wanted Proportion	Proportion in results
Language	60	80%	76%
Source	30	50%	30%
Type	80	100%	69%
Format	50	90%	84%

Table 6.7: Results of a search using the default settings

6.1.12 All Constraints, Qualitative Analysis

In the implementation's default settings, all constraints are enabled. The flat constraints are set to be soft and the diversity is set to maximize the minimum distance.

Figure 6.8 shows a search for "agent technology" after the results are found. The flat constraints are honored. The results show two patterns:

- None of the constraints were fulfilled. However, the language and format constraints were almost fulfilled. Taking one document more would make the proportion of documents fulfilling these constraints more than what was needed. Therefore it is acceptable to return less.

- The source and type constraints were violated much more. These can be explained by the low weight for the source constraint. As to the type constraint, the result of only 69% correct documents can be explained by the great number of documents with a high score but with a wrong type. This example shows that soft constraints may be violated if fulfilling them would bring other, worse, penalties.

These two patterns were encountered several timed during tests.

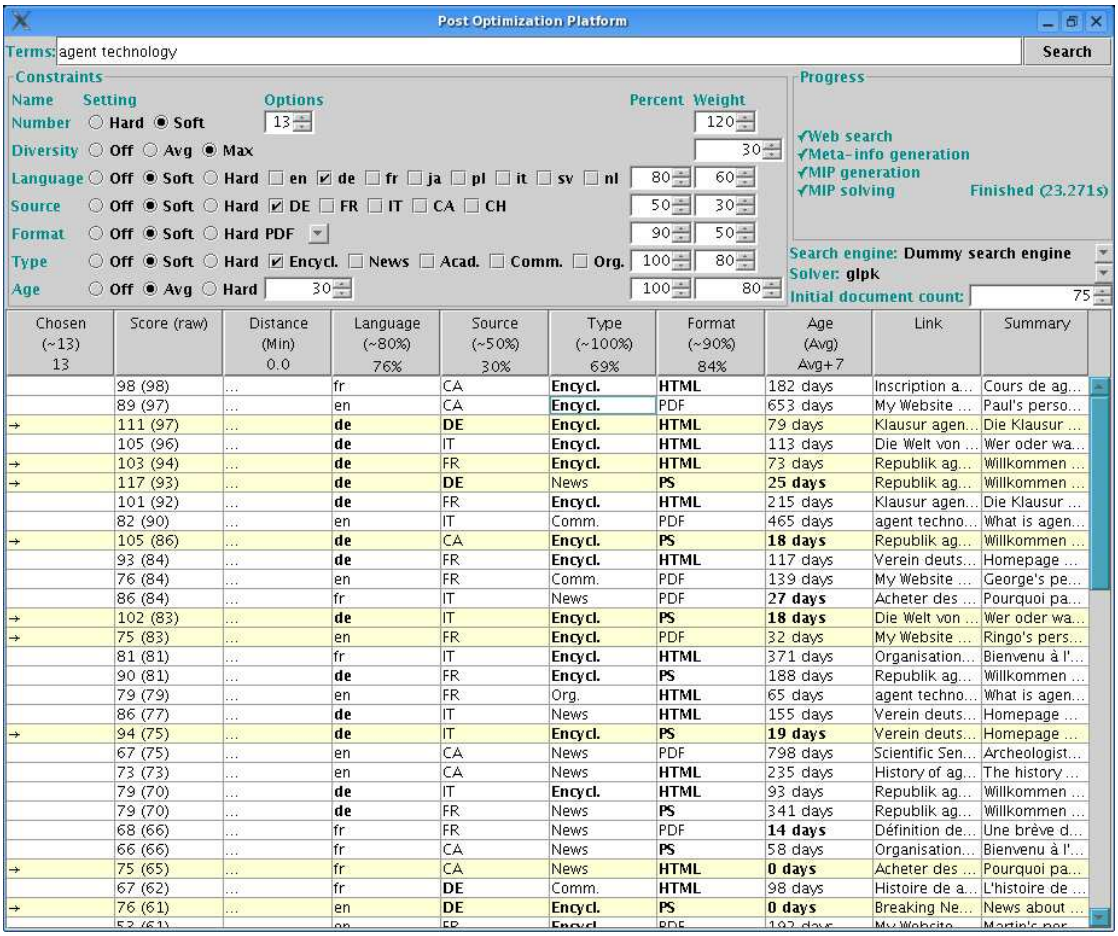

Figure 6.8: Performing a search using the default settings in the implementation

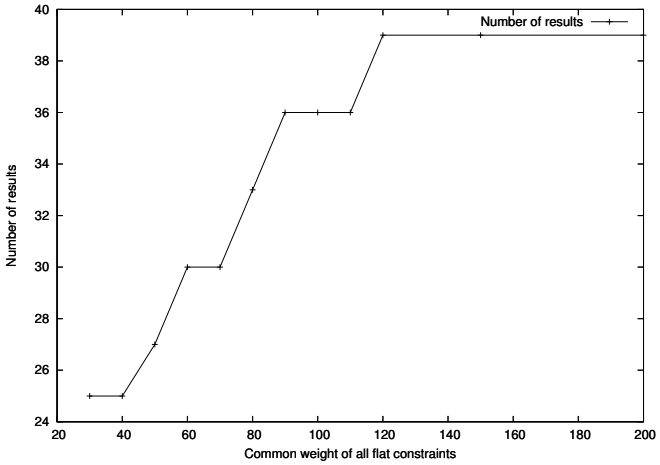

Figure 6.9: Forcing more results than wanted using high weights

6.1.13 Many Hard to Fulfill Soft Constraints

When many soft constraints are defined with high proportions and hard to fulfill values, the number of results constraint should be violated.

Diversity is off, the number constraint is soft with a penalty of 120 (default), wanting 13 results. The other penalties are all soft with all the same penalty. The number of results is shown in function of the weight of the flat constraints. 165 initial documents were used. The results are in figure 6.9.

6.1.14 Infeasible Constraints

When, in a linear program, the constraints are such that no variables exist that fulfill all constraints, then the problem is called infeasible. Generally, the integer linear program generated for a search can be infeasible. This is the case when, for instance, the user has set as a hard constraint that he wants 100% of documents in one language, but no documents in this language are contained in the initial document set. No subset of the initial document count can fulfill this constraint, and thus the problem is infeasible.

Realistically, the search is most of the time infeasible when all constraints are set to be hard. For this reason, constraints are soft by default, and should only be set to hard when really needed. Hard constraints should therefore be used sparingly, and if they are used, then only as few as possible constraints should be hard.

When only soft constraints are used, the problem always has a solution. In fact, every subset of the initial document set represents a valid solution when appropriate values for the penalty variables are used.

Number of hard constraints	Time (s)	Number of results
0	13.309	15
1	4.784	30
2	4.793	30
3	4.486	30
4	4.316	30
5	infeasible	

Table 6.8: Using increasingly many hard constraints

6.1.15 Soft Constraints vs. Hard Constraints

In this test, all constraints except diversity are enabled. Diversity is not enabled because it does not have the distinction between hard and soft. Because it uses a penalty variable, it can be considered as inherently soft.

The default settings of the implementation were kept, to emulate a typical user. This test tries to answer the question, whether constraints should better be hard, soft, or a mixture of both. These cases are tested:

- All five constraints are soft. This is the default setting in the implementation.
- One constraint is hard, the other four are soft.
- Two constraints are hard, three soft.
- Three constraints are hard, two soft.
- All constraints are hard except one.
- All constraints are hard.

The percentages and penalties were kept on the default settings.

The test results show the proportion of matching results for each constraint, using the same document set every time.

Table 6.8 shows that the more hard constraints are used, the more results have to be returned. Using too many hard constraints makes the problem infeasible.

6.2 Comparison with Other Algorithms

6.2.1 Comparison with Independent Search

When all constraints are disabled, the k first documents are always chosen, and the result set thus corresponds to the result of an independent search. Independent search is therefore a special case of post optimization.

Also setting all enabled constraints as hard is equivalent to independent search in only "good" documents.

6.2.2 Comparison with Strict Categories

When searching using strict categories, the search space is divided into several categories which are searched independently. The results from all searches are then combined to produce the final result. Search in strict categories can be modeled using integer linear optimization by defining one hard constraint for each category and defining the amount of documents that should come from each category. Search using strict categories is thus a special case of search using integer linear optimization.

6.2.3 Comparison with Elimination of Duplicate Results

The diversity maximization constraints have the effect of recognizing duplicates. Thus, a preliminatory step that searches and eliminates duplicate documents is not necessary. However, both are not completely equivalent.

Documents that are duplicates would introduce a penalty if both chosen. However, they may still both be present in the results if the objetive function is increased in another way.

Because it is the similarity between documents that is calculated, not only perfect duplicates, but also near duplicates are recognized. This alleviates the problem of near duplicate detection: if the threshold is too low, very similar documents are included. If the threshold is too high, some similar documents may be excluded that would both be interesting to the user. Using linear optimization, the similarity is added to the objective function and is thus automatically weighted against other constraints.

6.2.4 Comparison with Combinatorial Diversity

Combinatorial diversity as described in [1] cannot be combined with other constraints. For instance, combining it with categories would reduce diversity because the number of documents from each category is predetermined. Using linear optimization, all possible linear constraints can be combined at will with arbitrary weight.

Also, combinatorial diversity may not be optimal: There is no guaranty that the greedy algorithm finds an optimal solution. In fact, the maximal distance from the optimum cannot be given by this algorithm. The simplex method, even if not run until the end, will give an upper bound on the distance to the optimal solution by considering the dual problem.

6.2.5 Comparison with Genetic Algorithms

The optimum found by the solver is guaranteed to fulfill all constraints and to be optimal for the objective function given. Because the simplex and other algorithms return an optimal solution, the solution found is known to be the best. Genetic algorithms, on the other hand, cannot proove any of their solutions optimal, and cannot even give a bound on the correctness. Even intermediary results output by a mixed integer program solver have a bound.

As do genetic algorithms, integer programs provide intermediary results that get better at each iteration. The interval in which better solutions are found in integer programming is however not fixed, and it is often the case that the ratio between upper and lower bound to the optimum remains high during a long time and jumps to a small value later. In contrast to this, genetic algorithms give intermediary results continuously, and would thus be better suited if the runtime available was fixed from the beginning. In such a case, it could be that solving a linear program would not give any results.

Genetic algorithms also differ from solving a linear program in the way diversity maximization is handled. In the linear programming case, diversity is modeled as one constraint among others, and can weighted or disabled at will. In the genetic case, diversity is assumed to be inherent to the algorithm.

The main inconvenience of genetic algorithms lies in the fact that objectives cannot be weighted, which is a main advantage of the integer linear optimization approach. As was seen before, using hard constraints should only be limited to the most important constraints.

6.3 Case Studies

6.3.1 Multilingual Search

This example shows that using soft constraints may produce a better result that separate searches on subsets of the search space.

The scenario is the following: The user speaks German and French. By default, he wants to receive the same amount of German and French results. However, for some requests, both languages don't provide the same level of results. In this case, it is acceptable to return most documents in one language.

If bounds were defined for the number of documents returned in each language, the results may be wrong, since half of them will be from the language that has not much interesting results. Using pseudo-variable in this case will ensure that other criteria may trump the language contingency in a linear way. Without pseudo-variables, the search would be reduced to two separate independent searches.

6.3.2 Category Clustering

A search term usually has several meanings. Ideally, information retrieval systems would be able to discern them and propose two search refinements to the user. Such a functionality would require a database of words and their relationships, much like WordNet[28] provides. However, WordNet does not include proper nouns. This represents a limitation, since many users will make such requests. For instance, a user searching for "apple" as described in 1.1.1 may want to find information about the fruit, the flowering plant and the company. In this case, WordNet only provides the first two meanings. By searching for "apple", many documents will be found. The documents are then analyzed for their categories. Categories containing many found documents are remembered and a search is made using pseudo-variables. The search values are adjusted to approximately

find a number of documents from each category proportional to the number of matching documents in each category. Thus if the database does only contain very few documents about the apple plant, the few documents will still be returned, preventing all computer related documents to be returned.

6.4 Summary

Tests were executed to measure the power of integer linear programming search. It was found out that the different constraint options have a great influence on the results. Some tests uncovered negative results, especially that maximizing the average distance between documents is very expensive in terms of runtime. Other features implemented, such as combining multiple flat constraints in section 6.1.11, worked as expected.

7 Conclusion

Constraints in information retrieval were implemented using mixed integer programming. In conclusion, this approach can be seen as a generalization of several known approaches, which can be combined as needed. Other constraints can be mixed in at will. Diversity is hard to compute, but problems are easier to solve when there are more constraints. Hard constraints should be used with caution.

Some known approaches to information retrieval can be formulated as mixed integer programming. These include simple independent search which does not require any constraints. Search in fixed categories can be modeled by using hard constraints. Maximizing diversity in a set of documents of maximal score can be achieved by setting an appropriate weight for the diversity constraint. Eliminating duplicates is done automatically when enabling diversity, and the weight of the diversity constraint decides how strict duplicate elimination should be.

All these constraints can be combined using any weight. Other constraints, such as the minimum average age, can be enabled additionally.

As a general rule, this method is scalable to tens of thousands of documents when diversity is disabled. With diversity, the initial number of documents must be kept low. Tests showed that the worst case happens when the average distance between documents is used as the base for the diversity. In fact, enabling average distance diversity maximizing is probably too slow for current systems.

Interstingly, problems containing more constraints are generally easier to solve than problems with few constraints. This anti-intuitive result can be explained by the lower number of possible solutions when more constraints are enabled. In the same way, the runtime usually is shorter when constraints have a high weight, and they are even shorter when constraints are hard. The diversity constraint, however, behaves the other way around: The runtime increases when the constraint has a higher weight.

Hard constraints can easily make the problem infeasible. As a general rule, they should be used with caution. Implementations should make all constraints soft by default, and users should only need hard constraints in rare cases. It is also soft constraints that enable a trade-off between different constraints, and which thus give the most interesting use cases.

Changing a weight incrementally does not change the results incrementally. Sometimes, the results will get better when the weight of a constraint is beyond a certain threshold, and will not get better when the weight is increased. The tests show that these thresholds vary from case to case, and that it is difficult to find good default values for the constraints.

7.1 Future Work

In this section, aspects not covered in this thesis are briefly described. They may be the subject of future research.

7.1.1 Constraint Generation from Implicit Feedback

The implementation used here contains a huge array of options, and the users has to provide them all. While the default settings are meant to be useful without further modification, an inherent advantage of the integer programming approach is the high degree of customizability. To provide more options than a user wants to enter, data from implicit feedback could be used. How that data can be collected and how it can be adapted to work as options for constraints would have to be seen.

7.1.2 Search Engine Combination

One possible enhancement of the current system is to use more than one search engine. The best results from at least two different search engines would be combined, and the whole set of documents then used for optimization.

Scores would have to be available from all engines, and would have to be comparable. For instance, if one engine gave a score of over 90% to the twenty first results, and another engine only gave scores under 90%, the first engine would be unfairly advantaged. In this case, scores are not comparable, and information fusion would be needed for the combination to be useful.

Since the probability of having to deal with duplicate results would be high, the distance constraint would have to be enabled in most cases.

7.1.3 More Meta-Information

Meta-Information used in this implementation is either based on basic criteria such as the URL, or generated randomly for testing. For providing more interesting results, meta-information could be extracted from the document itself.

7.1.4 Custom Score and Similarity Functions

Custom score functions could be implemented, based on existing algorithms. For the similarity algorithm, more elaborate measures could be used.

7.1.5 Explanation of Results

When users enter many options into the GUI, they want to see the results accordingly. Results have to be explained, and be connected to the constraints. Currently, the degree to which constraints are fulfilled is displayed, and users can compare this value to what they entered into the GUI. Also, when a property of a document matches the user's options, the corresponding meta-information is highlighted in the list of results.

To provide more information, one could envision a system where the user could ask: "Why was this document not included?" The system would then make a search where the document in question must not be in the results, and present the difference between the new results and the old. For instance, if a user wonders why a certain document neither in the wanted language nor from the wanted source is included in the results, the system would indicate that if this document was not chosen, the document chosen instead would have a greater age, and would thus make the age penalty bigger. Similarly, a user might want to know why a document was *not* included. Such a request could be met in the same way, by starting a new search where this document is excluded.

7.1.6 Application on a Corpus

In the implementation of this work, two sources of search results were used for the initial set of documents: Results from the self organizing maps filtering community[6] and a random algorithm. None of these two is by nature deterministic, and the tests used for evaluation thus cannot show concrete examples.

To remedy this problem, a corpus of fixed documents could be used. Meta-information would have to be available for each document. Also, settings for the constraints should be given with useful results to compare against the results of this implementation. Of course, searching in the corpus would not be covered by this implementation and would be delegated to another program. This program would have to already conform to the expectations of the corpus user to not falsify the results.

7.1.7 Parallel Solving

Solvers for linear programs always hold a current best solution, and can be programmed to begin the search at a specific solution. Thus, multiple solvers could be run in parallel, and intermediary solutions be exchanged between them at regular intervals. Such an approach was not tried in the implementation because only one solver was used. It is not known whether this tactic would bring a benefit.

7.1.8 Search Refinement

If a user is not satisfied with search results, he usually modifies the query, for instance by excluding certain keywords. When using search with constraints, users may activate or modify constraints in reaction to unwanted results. To assist users in their selection of new constraints, the GUI could, for each document returned, provide a list of constraints that could be enabled which would exclude this result, thus including other results. Such a system would result in an interaction between the user and the computer comparable to conversational systems: The user starts by entering a simple query without any constraints, and incrementally refines it to make the results correspond to what he wanted.

List of Tables

List of Figures

Bibliography

[1] Barry Smyth and Paul McClave, *Similarity vs. Diversity*. Lecture Notes In Computer Science, Springer-Verlag, Germany, 1999.

[2] David McSherry, *Diversity-Conscious Retrieval*. Proceedings of the 6th European Conference on Advances in Case-Based Reasoning, pp. 219–233, Springer-Verlag, London, 2002.

[3] Martin Grötschel, *Skript zu Lineare Optimierung*, 2004.

[4] J. D. Landa Silva and E. K. Burke, *Using Diversity to Guide the Search in Multi-Objective Optimization*. Applications of Multi-Objective Evolutionary Algorithms, Advances in Natural Computation, Vol. 1, World Scientific, pp. 727–751, 2004.

[5] Andrew Makhorin, *GNU Linear Programming Kit, Reference Manual, Version 4.8*.

[6] Christian Scheel, *Managing Strategies for a Self-Organising Map based Filtering Agent Community*. Diploma thesis at the Technical University of Berlin, 2005.

[7] Şahin Albayrak, Dirk Wieczcorek, *JIAC - A Toolkit for Telecommunication Applications*. Proceedings of the Third International Workshop on Intelligent Agents for Telecommunication Applications, 1999.

[8] Fourer, Gay, Kernighan, *AMPL: A Mathematical Programming Language*. Duxbury Press / Brooks/Cole Publishing Company, 2002.

[9] S. Borzsonyi, D. Kossmann, and K. Stocker, *The Skyline Operator*. In Proc. IEEE Conf. on Data Engineering, pp. 421–430, Heidelberg, Germany, 2001.

[10] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, *Skyline with Presorting*. Technical Report, Computer Science, York University, Toronto, ON, Canada, Oct. 2002.

[11] L. McGinty, B. Smyth, *On the Role of Diversity in Conversational Recommender Systems*. In Proceedings of the Fifth International Conference on Case-Based Reasoning (ICCBR'03), pp. 276–290, 2003.

[12] D. Kossmann, F. Ramsak, and S. Rost, *Shooting Stars in the Sky, an Online Algorithm for Skyline Queries*. In VLDB, Aug. 2002.

[13] Emmanuel Hebrard, Brahim Hnich, Barry O'Sullivan and Toby Walsh, *Finding Diverse and Similar Solutions in Constraint Programming*. Proceedings of AAAI'05, pp. 372–377, Pittsburgh, Pennsylvania, July 2005.

[14] Tao Tao and ChengXiang Zhai, *Optimal Information Retrieval with Complex Utility Functions*. Proceedings of ACM SIGIR 2004 Workshop on Mathematical/Formal methods in Information Retrieval.

[15] Barry Smyth, Lorraine McGinty, *The Power of Suggestion*. IJCAI 2003, pp. 127–132.

[16] C. M. Fonseca and P. J. Fleming, *An Overview of Evolutionary Algorithms in Multi-Objective Optimization*. Evolutionary Computation, 3(1): pp. 1–16, 1995.

[17] J. Horn, N. Nafpliotis, and D. E. Goldberg, *A Niched Pareto Genetic Algorithm for Multiobjective Optimization*. in IEEE Symposium on Circuits and Systems, pp. 2264–2267, 1991.

[18] N. Srinivas, K. Deb, *Multi-Objective Optimization Using Nondominated Sorting in Genetic Algorithms*. 1994.

[19] Eckart Zitzler and Lothar Thiele, *An Evolutionary Algorithm for Multiobjective Optimization: The Strength Pareto Approach*. Technical Report 43, Computer Engineering and Communication Networks Lab (TIK), Swiss Federal Institute of Technology (ETH), Zürich, Switzerland, May 1998.

[20] Hui Fang, Tao Tao, ChengXiang Zhai, *A Formal Study of Information Retrieval Heuristics*. Proceedings of ACM SIGIR 2004.

[21] Hui Fang, ChengXiang Zhai, *An Exploration of Axiomatic Approaches to Information Retrieval*. Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval, 2005.

[22] Michael W. Berry, Zlatko Drmač, Elizabeth R. Jessup, *Matrices, Vector Spaces, and Information Retrieval*. SIAM Review, Volume 41, Issue 2, June 1999.

[23] Shengli Wu, Fabio Crestani, *Multi-Objective Resource Selection in Distributed Information Retrieval*. International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems, Volume 11, Issue Supplement, September 2003.

[24] Shengli Wu, Fabio Crestani, *Distributed Information Retrieval: A Multi-Objective Resource Selection Approach*. International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems, Volume 11, Issue Supplement, September 2003.

[25] Nick Craswell, Francis Crimmins, David Hawking, Alistair Moffat, *Performance and Cost Tradeoffs in Web Search*. Proceedings of the fifteenth conference on Australasian database – Volume 27, 2004.

[26] Abdur Chowdhury, Ophir Frieder, David Grossman, and Mary Catherine McCabe, *Collecting Statistics for Fast Duplicate Document Detection*. ACM Transactions on Information Systems (TOIS), Volume 20, Issue 2, pp. 171–191, 2002.

[27] Irvin J. Lustig, Jean-François Puget, *Program Does Not Equal Program: Constraint Programming And Its Relationship To Mathematical Programming*. 2001.

[28] C. Fellbaum, *WordNet, an electronic lexical database*. MIT Press, 1998.