Seminar report

Set Intersection Problem

Marco Bellò

April 3, 2025

Faculty of Science

University of Helsinki

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki, Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	

Tekijä — Författare — Author

Marco Bellò

Työn nimi — Arbetets titel — Title

Set Intersection Problem

Ohjaajat — Handledare — Supervisors

Prof. J. Kärkkäinen, Prof. V. Mäkinen, Dr. D. Diaz

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Seminar report	April 3, 2025	8 pages, 1 appendix pages

Tiivistelmä — Referat — Abstract

ACM Computing Classification System (CCS)

General and reference → Document types → Surveys and overviews

Software and its engineering → Software creation and management → Designing software

Applied computing → Document management and text processing → Document searching

Avainsanat — Nyckelord — Keywords

algorithms, data structures, set intersection, inverted index

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Seminar in Algorithm Engineering

Contents

1 Introduction

The act of *searching* has become so deeply ingrained in the modern society that we tend to take it for granted, not only assuming it normal to have immediate and easily accessible information on the tip of our thumbs, but expecting it: a study from 2004 showed that users were not willing to wait more than ten seconds for a page to load (Nah, 2004). Fast forward twenty years and nowadays even a couple seconds holdup would be unacceptable, thus query retrieval needs to be fast. Blazingly fast in fact, since we need to account for all the delays typical of a gargantuan structure as big as the modern web, and, as the reader probably knows, it is *not* a good idea to rely on memory's performances increasing over time: the smart way to tackle this problem is via research and development of efficient algorithms, and exactly which type should be self-evident from the title of this document. The problem of the set intersection constitutes the backbone of every query resolver in a (web) search engine, since every word in a query is interpreted as a collection of documents' IDs which contains it.
In this survey-style paper we will first explain what searching (i.e., querying) entails, show how a document (e.g., a web page) can be transformed into word tokens which are then further processed into inverted indexes, and, finally, we will see a collection of algorithms that concern themselves with intersecting sets, meaning finding common elements between two or more comparable collections.

1.1 How Do We Search?

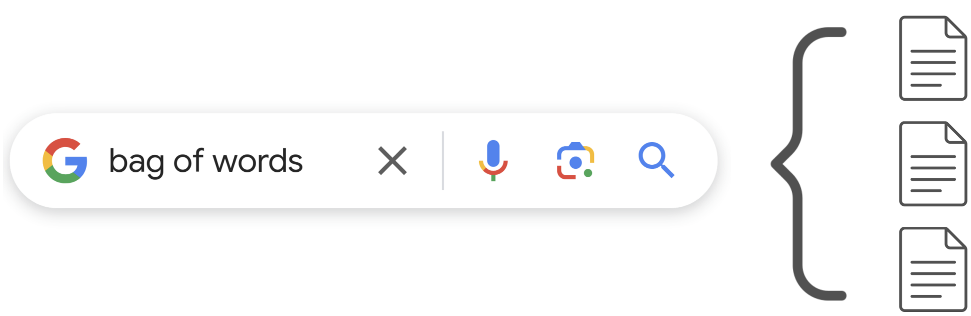

Figure 1.1: From a bag of words to a set of documents

Generally speaking, a query is called a *bag of words*, and finding its result means computing which documents contain all word tokens that are being searched for [1.1]. Let's make and example: word `abiura` is contained in documents number `[31, 42, 127]`, while word `bitonto` is contained in documents number `[20, 42, 72]`.
Thus `query = (abiura,bitonto)` will return the result `42`.

Dictionary	**Posting List (ASC)**	**Relevance**
abaco	1, 7, 136	0.6, 0.3, 0.8
abiura	31, 42, 127	0.12, 0.5, 0.77
bitonto	20, 42, 72	0.8, 0.1, 0.03

Figure 1.2: Table of word tokens

Both The example above [1.2] and all the algorithms we will see in this survey consider the problem of searching as the problem of complete intersection, but modern search engine (e.g., Google) leverage input relevance and filter unneeded outputs to obtain faster and better results. Unfortunately finding information about how they do it is near impossible, since everything is covered by trade secret.
Let's now see what inverted indexes are and how we can obtain them starting from a document corpus.

2 Inverted Indexes

Most of the information present in this chapter is thanks to Mahapatra and Biswas "Inverted indexes: Tyoes and techniques" (Mahapatra and Biswas, 2011).

What we will need for the algorithms presented in the rest of this documents are inverted indexes (also called posting lists). To get them we first need to process documents into lists of words (called *word tokens*), then for each token compute a list of `IDs` that refer to the documents which contain that specific token. Let's see each step in order.

2.1 Document Pre-Processing

Documents go trough a series of processing steps before being indexed: they get converted into token in the lexing phase, which are then possibly normalized, stemmed or even pruned (removed) entirely.

2.1.1 Lexing

The process of transforming a document into a list of tokens, each of which is a single word, si called *lexing* [2.1]. There often is a maximum length for a single token, as to prevent unbounded index growth in edge cases, and all input is generally first converted into lower-case to normalize it. The all non-punctuation characters are added to the list of tokens one by one, and those that exceed a certain size are often pruned (removed from the corpus). It is not entirely clear how Google and other big companies do this step, and it certainly feels strange to think they employ a simple *brute force*, single scan approach, but

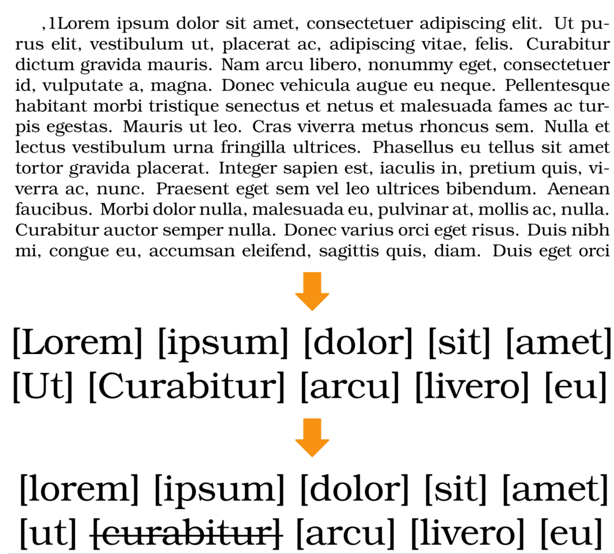

Figure 2.1: Lexing: from text to word tokens

as mentioned before it is not easy to find information about it.
All of the above works only with alphabetic languages, ideographic ones (e.g., Chinese) need specialized search techniques.

2.1.2 Stemming

We can consider this step deprecated, since nowadays memory, especially for things like text and arrays (which inverted index basically are), is cheap and bountiful.
The idea is to find a sort of *root* (stem) of the words, and indexing that instead. To make an example: fishnet, fishery, fishing, fishy, fishmonger, can all be boiled down to their stem *fish* [2.2].

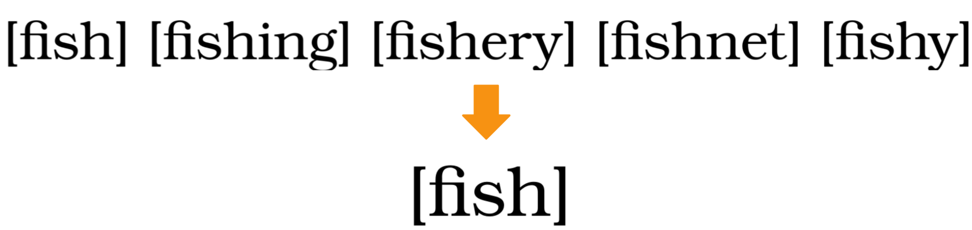

Figure 2.2: Stemming to stem "fish"

In the example above should be clear already that stemming carry some problems: a user searching for "fishnet" is likely not shopping for fishing equipment, thus most modern search engine skip this normalizing step, and most stemming algorithms (most famous of which is Porter's) are complex, full of exceptions and exceptions to the exceptions, while still failing to unite together the correct words. This step basically reduce query precision while providing very little in return.

2.1.3 Stop Words

3 Results

4 Discussion

5 Conclusions

Bibliography

Mahapatra, A. K. and Biswas, S. (July 2011). “Inverted indexes: Types and techniques”. In: *International Journal of Computer Science Issues*, 8.

Nah, F. (Jan. 2004). “A study on tolerable waiting time: how long are Web users willing to wait? Citation: Nah, F. (2004), A study on tolerable waiting time: how long are Web users willing to wait? Behaviour & Information Technology, forthcoming”. In: *Behaviour & IT*, 23.

Appendix A Sample Appendix

You can add one or more appendices to your thesis.