CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Expressive corpus search in a modern framework

Developing expressiveness for Korpsearch, a more efficient tool by which to query a corpus

Master's thesis in Computer science and engineering

VICTOR SALOMONSSON

MIJO THORESSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Expressive corpus search in a modern framework

Developing expressiveness for Korpsearch, a more efficient tool by which to query a corpus

VICTOR SALOMONSSON & MIJO THORESSON

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Developing expressiveness for Korpsearch, a more efficient tool by which to query a corpus
Victor Salomonsson Mijo Thoresson

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Developing expressiveness for Korpsearch, a more efficient tool by which to query a corpus

Victor Salomonsson
Mijo Thoresson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In this thesis we have developed a new corpus querying program, whose purpose is to be fast but also expressive. We achieved this by implementing the ability to query on prefix, suffix, contains, Python regular expressions, as well as with disjunctions whilst maintaining high speeds. Using our program, execution times for queries were on average half those of an established corpus querying program, Corpus Workbench. Our program's time taken to execute queries on disjunction was a 4% of what Corpus Workbench required, for instance. This thesis shows that one can implement disjunction by dividing disjunctive queries into all the possible permutations of subqueries. Then one can find their result through a quick intersection finding program. This extends to being able to find all words containing a certain string, and general Python regex matches. Our program also shows that one can depart from the disjunction solution path, the path that is to find all matching words and then form a disjuntive query between them, in special cases. Special solutions have been made for prefix and suffix which have managed to have shorter execution time for those kinds of queries.

Keywords: corpus, corpora, Corpus Workbench, Korpsearch, efficiently, query, Computer, science, computer science, engineering, project, thesis.

Acknowledgements

We want to especially thank our wonderful supervisor Nick Smallbone who has been a great help, giving invaluable feedback and insights on everything from the operations of the program, to the structure and language of the thesis. No doubt neither the program nor this thesis would have been able to be as efficient or eloquent without your guidance. As we would say in our parlance, Nick is an M.V.P. of this project, Most Viktig Person.

Additionally we want to give thanks to Graham Kemp for agreeing to examine our project. Your contribution to making our thesis better, and enabling our graduation is greatly cherished.

We also owe thanks to Peter Ljunglöf for giving us the chance to work on this project, as well as for volunteering support for our work on the project. The help in getting access to the logs and interface of Språkbanken was also very important in shaping the project to be as useful as it hopefully is now.

We would also like to thank our opponents Selma Moqvist and Weilong Chen, for agreeing to oppose us, and for providing valuable feedback, which helped us improve our thesis.

We'd also like to give thanks to some inanimate things.

We also want to thank Go-morgon for getting us up on drowsy Monday mornings.

We also want to thank Laplace Fika for bringing a well needed break Tuesday afternoons.

We would also like to thank Froffice for providing a good environment to write our thesis in relative peace, with a good view.

We also want to thank Seterra for teaching us all Chinese provinces and their capitals.

Do you feel left out? Do not hesitate to send us an email at mijot@student.chalmers.se, with your contributions.

Victor Salomonsson & Mijo Thoresson, Gothenburg, 2024-06-20

Contents

List of Figures

List of Tables

1

Introduction

1.1 Introduction

The raison d'etre of this project is to develop a program capable of efficiently querying a corpus. It seems pertinent to explain what a corpus is in order to understand how such a program works. A corpus is a database that stores a large amount of texts [1]. These texts are not stored as plain text and thus quick string searching tools like suffix arrays cannot be used [2]. A corpus instead comprises the texts as tokens consisting of the texts' words, part of speech, lemmas, and possibly more info. An illustration of this can be seen in Figure 1.1.

	The Corpus
Word	The horse runs . We caught the trespassing horse . The horse escaped . It is a tame horse . The black horse .
pos	DET SUBST VERB PUN PRO VERB DET ADJ SUBST PUN DET SUBST VERB PUN PRO VERB DET ADJ SUBST PUN DET ADJ SUBST PUN
lemma	The horse run . We catch the trespassing horse . The horse escape . It is a tame horse . The black horse .

Figure 1.1: Example Corpus
This is a representation of a corpus for illustrative purposes. The abbreviation 'pos' stands for part of speech.

When querying a corpus, the query could for example be

```
[word="the"][pos="ADJ"][word="horse"],
```

with each square bracket and content being called a literal. This query returns all occurrences of the word 'the' being followed by an adjective, which in turn is followed by the word 'horse'. For the previously mentioned corpus they would be: {The black horse}, and {the trespassing horse}.

Right now a corpus querying program that is commonly used is Corpus Workbench [3]. Whilst it is very expressive, it is not particularly fast in executing bigger queries. Korpsearch was developed to fix the problem of Corpus Workbench being too slow [4]. It achieved that for some queries, but is not able to execute all queries that Corpus Workbench can execute.

Let us look at two queries:

```
[word="the"][pos="ADJ"][word="horse"],
```

and

```
[word="the.*"][pos="ADJ"|pos="ART"][word=".*horse.*"].
```

The first query searches for all occurrences of 'the' followed by an adjective followed by 'horse'. The second query searches for all occurrences of a word beginning with 'the', followed by either an adjective or an article, in turn followed by a word containing the string 'horse'. The signs '.*' symbolise any string of characters. For instance 'the.*' indicates that any word starting with 'the' is a match regardless of what follows. Corpus Workbench is able to execute both of the above queries, though it does this relatively slowly.

Korpsearch is able to quickly execute the first query, but cannot execute the second.

It is to this background we developed Korpsearch into our program. Our program retains the ability to quickly execute the queries Korpsearch can execute quickly. It does this whilst also adding the ability to execute second query in the above paragraph. It is able to execute larger queries more quickly than Corpus Workbench, but small queries on only prefix, suffix, or contains are slower. To the average linguist currently using Corpus Workbench we believe this would be an improvement to switch to.

We had access to logs of queries performed by linguists at Gothenburg University using their current corpus querying program, Korp [5]. This list informed what we decided to implement as new features in Korpsearch. Some example queries taken from these logs can be seen in Figure 1.2. Let us now define some features before presenting statistics of what we saw in the logs.

Disjunctive queries are queries containing an or-statement, a disjunction. For instance `[word="cat"|word="dog"]` is a disjunctive query finding occurrences of the word being 'dog' **or** 'cat'. The 'or' is indicated by '|'.

Prefix queries are queries where the value of the queried feature has to begin with a given string, rather than be that string. For instance `[word="cat.*"]` is a prefix

```
[word="balalajka"]
[word="inklination"]
[word="deviation"]
[word="galvanometer"]
[word="i"][word="underkant"]
[word=".*isera"]
[word="det.*"][word="var.*"][word="han.*"][word="som.*"]
[word="denna"][word=".*en"&pos="NN"]
[word="tavl.*"][(word="sitt.*"|word="satt"|word="suttit")]
[(word=".*att.*"|word="att")][(word=".*det.*"|word="det")]
```

Figure 1.2: Example queries taken from the logs of linguists at Gothenburg University

query finding occurrences of the words **beginning with** 'cat'. The fact that the word can end with anything is indicated by '.*' being at the end.

Suffix queries are queries where the value of the queried feature has to end with a given string, rather than be that string. For instance `[word=".*cat"]` is a suffix query finding occurrences of the words **ending with** 'cat'. The fact that the word can begin with anything is indicated by '.*' being at the beginning.

Contains queries are queries where the value of the queried feature has to contain a given string, rather than be that string. For instance `[word=".*cat.*"]` is a contains query finding occurrences of the words **containing** 'cat'. The fact that the word can begin or end with anything is indicated by '.*' being at the end and the beginning.

Regular expression queries are queries where the value of the queried feature has to match a regular expression[6]. For instance `[word="[cr]at"]` is a regular expression query finding occurrences of the words 'cat' and 'rat'. A regular expression query is indicated by the query containing any special characters not already handled by the previous query types.

In the logs from Gothenburg University the two most common searches that we saw were some version of contains, and disjunctions. The prevalences were 23.1% and 4.9% respectively. The proportion of the queries in the log that contained none of those query types was 75% almost all of which being the types of queries the original Korpsearch can already handle. These numbers in combination mean that about 3% of queries contain both disjunction and some version of contains. Prefix and suffix are versions of contains for the purposes of this paragraph. It was these numbers that led us to implement disjunction, contains, prefix, and regex. Let us now define what we mean by them.

Another type of query which is interesting, but which we have not implemented is repeating literals. They do not appear in the logs very often, only 0.7 ‰ of queries contain them. It would, however, be good to implement them in the future since they can make queries quite expressive, so we will present them here anyway.

Repeating literals are ones where the literal can be any number of tokens matching the literal in a row. For example they might look like `[pos="NN"]*` meaning that one is searching for any number of tokens representing nouns that are in a row in the corpus. That the literal is repeating is indicated by the '*'. This does include zero nouns, meaning that the repetition returns occurrences where there are no nouns. This kind of query is mainly useful if there are other literals in the query as well.

1.2 Goals and Challenges

The goal of the project was to enable more expressive querying akin to what can be done with Corpus Workbench, but with the speed of Korpsearch. To say that this has been been accomplished two things would need to be true. First our program would need to be able to execute the types of queries that linguists are currently using. Based on the log of queries performed at Gothenburg University, this means that our program should be able to execute disjunctive, prefix, suffix, and contains queries.

Second our program must be quicker than Corpus Workbench. It should also be able to execute queries quickly enough that big queries are executed in a reasonable time. It is primarily for big queries that Corpus Workbench falters, and thus it is for big queries that execution times must be kept similar to Korpsearch. At the same time one must consider that adding all of the features that we are adding will cause some slow down of query execution. A small slowdown should not be seen as a failure to accomplish this project's goals.

At the end of this project we are able to handle queries containing any of these features such as

```
[word="love"][word="hey.*"],
[lemma="horse"|lemma="fool"][word="around"]
```

and somewhat general combinations of all kinds.

After this project, we will add our contributions to Korpsearch. This will make it so that there is a quick solution that has more capabilities to use, and develop in future master's theses.

2

Background

In this chapter we present some information on Corpus Workbench and Korpsearch to build some background knowledge of the world of corpus search programs that our program is based on and will be compared to.

2.1 Corpus Workbench

Corpus Workbench will in this report serve as a representative of well established corpus query tools. Korpsearch and our program were created specifically to serve as a faster corpus querying tool [4], so it seems pertinent to present how it works. To explain how Corpus Workbench executes a query we will again use the example query

```
[word="the"][pos="ADJ"][word="horse"],
```

on the example corpus in Figure 1.1 in the Introduction chapter. Corpus Workbench finds all the results by finding all occurrences of the word 'the', through an index look-up. Having found those, it filters by the condition that the word after is an adjective, and then filters what remains by the condition that the following word to those is 'horse'. In our example corpus this can be done quickly since it is so small. In the British National Corpus [7], however, the word 'the' appears 5 405 646 times. Filtering such a large number of occurrences understandably takes a long time. One could improve the execution time of the query by having it find occurrences of the literal corresponding to the fewest occurrences. Depending on the corpus used different literals would be picked. In the BNC it would be the 'horse'-literal. In the example corpus however, it would be the adjective literal. This would be quicker, because the program would then filter a smaller set in executing the rest of the query. Corpus Workbench has not implemented this sort of optimisation. One can easily theorise of two reasons why this is the case.

Firstly corpora used to be a lot smaller when Corpus Workbench was created in the 90's. Back then corpora would typically be of the size 10 to 100 million tokens, whereas now they get up to the size of billions of tokens, so optimisation work that wasn't necessary before is now [8].

Secondly Corpus Workbench has many features available when querying such as: suffixes, prefixes, optional tokens, repetitions, that add up to basically any conceivable kind of query [3]. This variety in possible literals makes it very easy to err

when trying to create efficient indexing. The risk is great, in such an endeavour that the program does not manage to return precisely the set of occurrences the query should return in all cases. Should the query be

```
[word="the"][pos="ADJ"]*[word="horse"],
```

one cannot use the index on adjectives, as zero adjectives also permits a match, and one must make a decision which index to look up instead of adjectives. Even with so small a change to the query it is not obvious how one should make Corpus Workbench's strategy more efficient.

2.1.1 Corpus Workbench as a Baseline

Corpus Workbench is as previously mentioned, a commonly used querying tool. For instance, it is used by linguists at Gothenburg University when they query corpora using Korp, their corpus tool [5]. Corpus Workbench is mainly interesting for this paper due to its wide-spread use making it a reasonable baseline. It is both a baseline for how much quicker one could query using our program, and a baseline in the sense of it showing what features linguists demand from a corpus querying tool.

Viewing Corpus Workbench as a baseline in the first sense, one finds that it ought to be achievable to improve the time taken to execute a query, as Corpus Workbench simply finds the results matching the first literal and then filters on the rest from left to right. One could improve the time taken to execute simple queries by starting with the literal with the fewest matches instead. When disjunctions, repetitions, optional literals, and other complexities add up, however, there will still be some work necessary to improve upon Corpus Workbench's performance.

This brings us to the second sense in which it provides a baseline. Corpus Workbench also lets us see what linguists want to query for. Since they have access to a query program that can perform most any query, one can see which queries they want to perform by what queries they do perform[1]. In this sense Corpus Workbench also lets us see what functionalities are more important than others to linguists. Looking at the data we have on linguists' queries at Gothenburg University, the most common types of queries seem to be those already implemented in Korpsearch, and prefix/suffix searches which we implemented. Disjunctive queries are also quite common. Additionally there are some 'contains' queries, i.e. querying for tokens whose words contain a certain string. All other types of queries are way less common to the extent that examples are hard to find, whereas the noted types of queries are observable at a cursory glance.

[1]Due to CWB's long execution time of complex queries and the difficulty in writing complex queries, one could argue that this list is not representative of all the queries that researchers might want to make. However, the list shows what queries are used currently and thus shows what one has to be able to do with a corpus querying tool to supplant CWB.

2.2 Korpsearch

Recently, in the year 2022, Ljunglöf and Smallbone have optimised querying, addressing both of the aforementioned reasons for slowness. Their program, Korpsearch, executes queries faster. It is, however, not as expressive as Corpus Workbench, only able to execute queries which consist of a sequence of tokens, each of which identifies a specific feature, e.g. word or part of speech. That is to say that it can handle neither disjunctions nor regular expressions. The program they developed has reduced the time taken to compute a query significantly through two things mainly: firstly it enables merge-based intersecting of two subquery results, secondly it utilises binary indexes. A subquery is a query that is part of a bigger query. Merge-based intersecting two queries' results works as an alternative to search-based intersection, which means searching for the first query's result in the second query's result with binary search. Merge-based intersection utilises that the result returned by a query will be of the form [2 3 7 14], where the numbers correspond to where the tokens matching the query are to be found in the corpus. Since these query results will be in ascending order, it is possible to find what numbers are in common in linear time by running through the lists of results. This merge-based intersection is primarily quicker than search-based intersection when subqueries results are similarly sized. Then there is also the second reason why Korpsearch is quick [4].

2.2.1 Binary indexes

Binary indexes are indexes on attributes of two tokens, as opposed to the unary indexes otherwise used, which are on one attribute of one token. A binary index might for instance indicate all positions of `[word="the"][][word="horse"]`, that is to say all occurrences of the word 'the' being separated from the word 'horse' by one word. The unary indexes would here instead be `[word="the"]`, and `[word="horse"]`. Using only unary indexes a program would find all occurrences of horse and all occurrences of the by index look-ups, and would then have to combine them through one of the intersection methods. Using binary indexes instead allows finding the occurrences through one index look-up, turning what would otherwise be two look-ups and one intersection into one look-up.

2.2.2 The Mechanics of Korpsearch

Korpsearch is used as a base for our program so it is good to explain how it works before writing about how it was changed into our program. When one passes a query of the form `[word="the"][pos="ADJ"][word="horse"]` to the program, the program divides it up into three subqueries `[word="the"]`, `[pos="ADJ"]`, and `[word="horse"]`, one for each literal. It then finds the results of these subqueries. Lastly it finds the intersection of these results. This intersection is the result of the original query. There is some more detail to this process. To determine in what order subqueries' results should be intersected the program looks at their sizes. The subqueries are placed in a queue in ascending order by result size. In the example query in Figure 2.1 there are 5 occurrences of 'the', 4 of adjectives, and 6 occur-

rences of horse. The queue would then be 1. `[pos="ADJ"]` 2. `[word="the"]` 3. `[word="horse"]`'. The intersections can then be found through use of search-based or merge-based intersection. In this case since they are all so similar in size Korpsearch would use merge-based intersection. Fewer intersections could be made if there is a binary index between pairs of literals. Should for instance '[the][][horse]' exist as a binary index, one could immediately use the index to look up 'the + horse' and use that in the intersections. The subqueries 'the' and 'horse' would in that case be subsumed, as they would no longer be necessary.

In the above paragraph some concepts are introduced meriting a more thorough explanation. Let us start by elaborating on how the result of a subquery is procured. When one looks for all the tokens which have the word being 'the', what the program actually finds is the word index of that word, as can be seen in steps 1 and 2 in Figure 2.1. After this, as can be seen in step 3, the occurrences of that word are found in the corpus index list, which is sorted lexicographically. Since they are stored in order one can find the first and last occurrence using binary searches, and then know that all rows between those constitute the result, which is returned as all the corpus indexes, i.e. places in the corpus in step 4.

All values that the corpus contains are stored in a number of string files, one for each feature. Each word is stored only once, in lexicographical order. For the example query in Figure 2.1 this string file would, for the word feature, look like: [**a**c**a**ught**e**scaped**e**scorted**h**orse**i**s**i**t**o**ut**r**uns**s**wims**t**ame**t**he**t**respassing**w**as**w**e**.**]. The information about where new words start in this file is kept in another file, the position file.

2.2.3 Usage

Korpsearch can return its results in two specified formats, KWIC (keywords in context) or a json format. In theory any program that can handle these formats can be used to interact with Korpsearch. This means that whoever wants to use Korpsearch as a back-end can have their own tailor-made front-end. This means that anyone who already has one of these to interact with another corpus querying software, that uses the same output formats, in theory should be able to simply swap it out with Korpsearch.

For people who do not already have a front-end program, the original authors of Korpsearch have created a simple demonstration program which lets you input queries and look through the results in context. The program has a quite simple interface which can be seen together with an example search result in Figure 2.2.

2.3 Query optimisation

We will briefly discuss other work that has been done in this project's field. David [9] describes an idea to make a quick yet expressive corpus querying program through storing the corpus as a relational database. He also identified a problem with existing corpus querying tools in them being quite slow. The relational database proposed

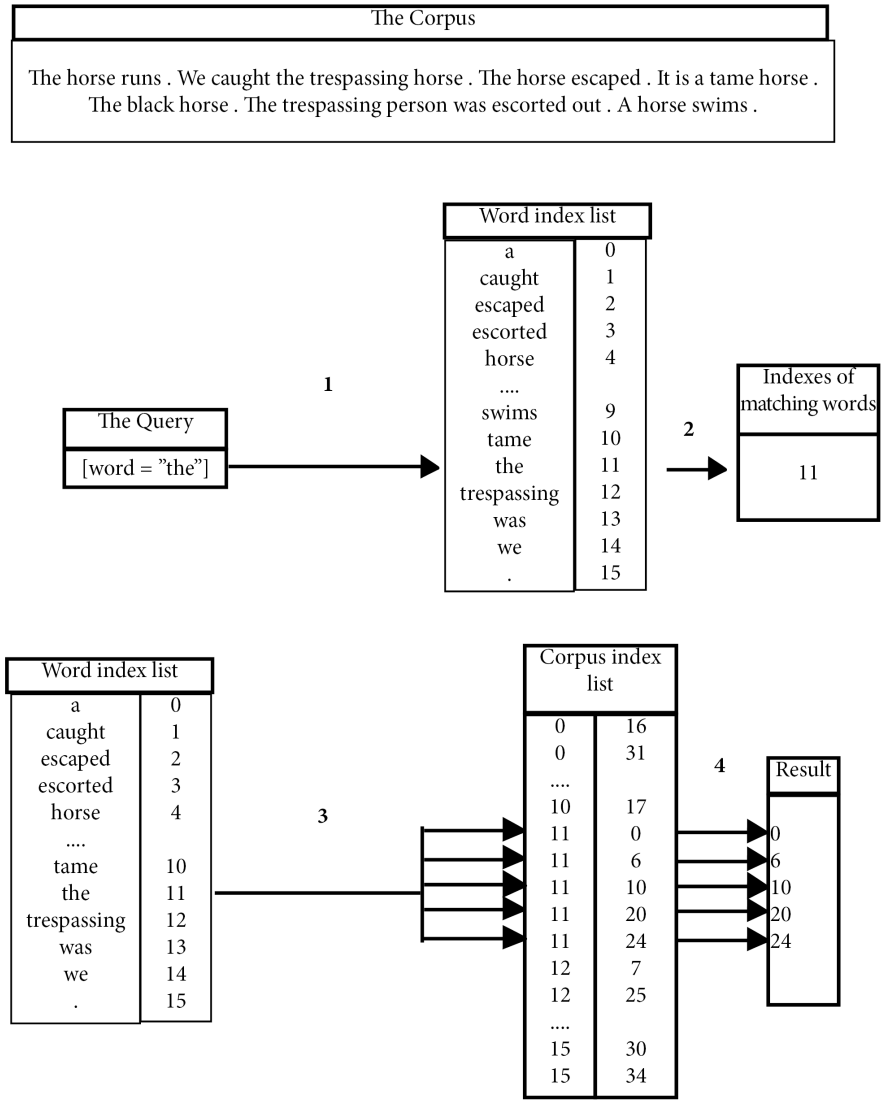
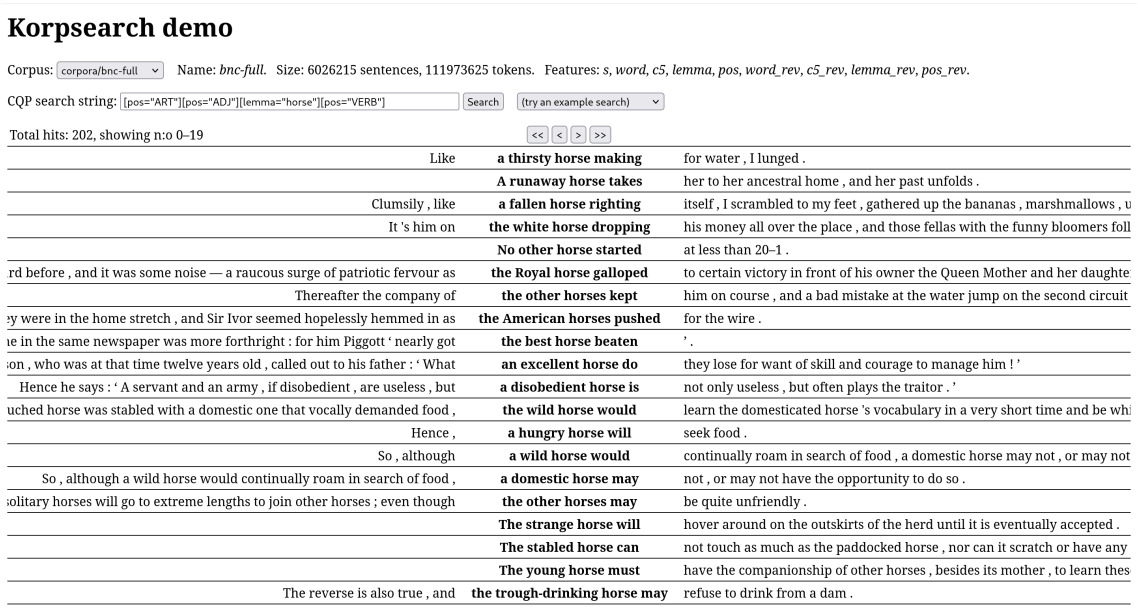


Figure 2.1: Flowchart delineating Korpsearch's query execution.
This is an illustration of how Korpsearch finds the results of a one token query. Step 1 is finding that we want word to be 'the', step 2 is finding that 'the' has index '11' in the corpus. Step 3 is finding the first and last 11, between which all 11's are. Having found all 11's their positions are returned as a result.



Korpsearch demo

Corpus: corpora/bnc-full Name: *bnc-full.* Size: 6026215 sentences, 111973625 tokens. Features: *s, word, c5, lemma, pos, word_rev, c5_rev, lemma_rev, pos_rev.*

CQP search string: [pos="ART"][pos="ADJ"][lemma="horse"][pos="VERB"] Search (try an example search)

Total hits: 202, showing n:o 0–19 << < > >>

Like	**a thirsty horse making**	for water , I lunged .
	A runaway horse takes	her to her ancestral home , and her past unfolds .
Clumsily , like	**a fallen horse righting**	itself , I scrambled to my feet , gathered up the bananas , marshmallows , u
It 's him on	**the white horse dropping**	his money all over the place , and those fellas with the funny bloomers foll
	No other horse started	at less than 20–1 .
rd before , and it was some noise — a raucous surge of patriotic fervour as	**the Royal horse galloped**	to certain victory in front of his owner the Queen Mother and her daughte
Thereafter the company of	**the other horses kept**	him on course , and a bad mistake at the water jump on the second circuit
y were in the home stretch , and Sir Ivor seemed hopelessly hemmed in as	**the American horses pushed**	for the wire .
e in the same newspaper was more forthright : for him Piggott ' nearly got	**the best horse beaten**	' .
on , who was at that time twelve years old , called out to his father : ' What	**an excellent horse do**	they lose for want of skill and courage to manage him ! '
Hence he says : ' A servant and an army , if disobedient , are useless , but	**a disobedient horse is**	not only useless , but often plays the traitor . '
uched horse was stabled with a domestic one that vocally demanded food ,	**the wild horse would**	learn the domesticated horse 's vocabulary in a very short time and be whi
Hence ,	**a hungry horse will**	seek food .
So , although	**a wild horse would**	continually roam in search of food , a domestic horse may not , or may not
So , although a wild horse would continually roam in search of food ,	**a domestic horse may**	not , or may not have the opportunity to do so .
olitary horses will go to extreme lengths to join other horses ; even though	**the other horses may**	be quite unfriendly .
	The strange horse will	hover around on the outskirts of the herd until it is eventually accepted .
	The stabled horse can	not touch as much as the paddocked horse , nor can it scratch or have any
	The young horse must	have the companionship of other horses , besides its mother , to learn thes
The reverse is also true , and	**the trough-drinking horse may**	refuse to drink from a dam .

Figure 2.2: Korpsearches demonstration front-end
Korpsearches demonstration front-end querying the BNC for `[pos="ART"][pos="ADJ"][lemma="horse"][pos="VERB"]`

to amend this utilised n-grams. The program created tables containing all 1-gram, 2-gram, 3-grams, and 4-grams and their frequency in different categories of text. Davies presents that with this all queries tested took less than five seconds to execute, and that most took less than a second [9].

Another idea to enable quicker query execution is using inverted indexes, as has been done by Diewald and Margareta. Their program KRILL, like Korpsearch, finds the results of the subqueries as lists, and checks the list against each other. KRILL was intended to be used for a specific German corpus [10].

Yet another approach is taken by Meurer in the program Corpuscle. This program has optimisations both for their regular expression evaluation and also in the subquery merging strategy. They also employ inverted indexes. To do faster regex matching, they utilise suffix arrays. Furthermore, they use a graph cutting algorithm to find the optimal way to merge results [11].

Our program still has purpose with this background, as it combines expressiveness and binary indexes, and manages to execute queries very quickly. The other programs also do not have public code. The other projects briefly presented in this section show that this is an area where research and developments are made. Therefore, our program and Korpsearch has a real purpose.

3

Methods

Korpsearch works under the philosophy that queries can be executed as an intersection of a number of sets corresponding to the results of a query's subqueries. When implementing disjunction we had to break from that philosophy. Suddenly subqueries had varying relations, requiring that the program record the relationship between different subqueries in some way since not all subqueries' results should have their intersection found. Building on this we have also enabled more expressive searching using regular expressions and implemented some special cases to speed up commonly queried patterns. What is commonly queried has been identified from studying logs of queries made by linguists at Gothenburg University using their current querying tool.

3.1 Disjunction

The largest addition that we made was to add the ability to use disjunctive queries. Disjunctive queries' optimal execution plan cannot be found by Korpsearch's priority algorithm. An algorithm working with disjunctive queries has to take into account that some pairs of subqueries' results should be unioned with each other, have their union found, whilst other pairs of results should be have their intersection found. This is necessary to ensure that the resulting set is correct.

Our program executes disjunctive queries by expanding them into all possible combinations, whose results are all then intersected. We will explain through the use of the example query:

```
[word="the"|word="a"][word="horse"|word="cat"].
```

All the possible combinations would then be:

```
[word="a"][word="cat"],
[word="the"][word="cat"],
[word="a"][word="horse"],
[word="the"][word="horse"].
```

First the results to the subqueries are found. This is step 1 in Figure 3.1. After this the intersection of two subquery results are found. In this example the occurrences of 'the cat' are found through intersection, as step 2. The occurrences of 'the horse' are also found as step 3 in the figure. As step 4 the union of their results is found. Then

the occurrences of 'a horse' are found and added to the union, then the occurrences of 'a cat' are treated the same way. After the last combination has been added to the union the query has been executed to completion.

This approach extends Korpsearch naturally since it performs the same task as in the non-disjunctive case only multiple times. Additionally it is the most efficient approach given the assumption that intersections of a and b have a size of $\frac{ab}{tot}$. 'tot' being the size of the entire corpus. This assumption and an alternative assumption we could have made is discussed in Section 3.3.

We have now written about the strategy consisting of expanding the disjunction, called the 'intersection first approach'. The other strategy to consider is not expanding the union, and is called the 'union first approach'. This strategy consists of finding the union of the disjunctive subqueries, and then once no disjunctive elements remain, the intersection of all the sets can be calculated to find the result of the query. For the example query above this would entail unioning the results of `[word="the"]` and `[word="a"]`, and unioning the results of `[word="horse"]` and `[word="cat"]`. One would then have two sets, one of all the occurrences of 'a' and 'the', and one set of all occurrences of 'horse' and 'cat'. The intersection of these two sets is the result of the query. As mentioned in the previous paragraph this strategy is not as efficient under our assumptions on the size of intersections. Why it isn't as efficient is explained in Section 3.3.1.

The execution of disjunctive queries has to take into account that disjunctive literals of the same token should maybe be handled during different times of the query execution. To elucidate we will look at the example query:

```
[word="brown"|word="swift"][word="horse"],
```

where `|` means disjunction. Let us say that when checking the corpus, *brown* occurs more often than *horse*, which in turn occurs more often than *swift*. In this case *brown horse* should be sought by finding all occurrences of *horse* and checking for the preceding word being *brown*, whereas *swift horse* should be sought by finding all occurrences of *swift* and checking if the following word is *horse*. This thought experiment indicates that one shouldn't necessarily treat the different conditions making up a disjunctive literal the same. The main challenge then lies in finding a way that holds true for all possible cases of finding the union necessary to evaluate a disjunctive query, in a way which is optimised.

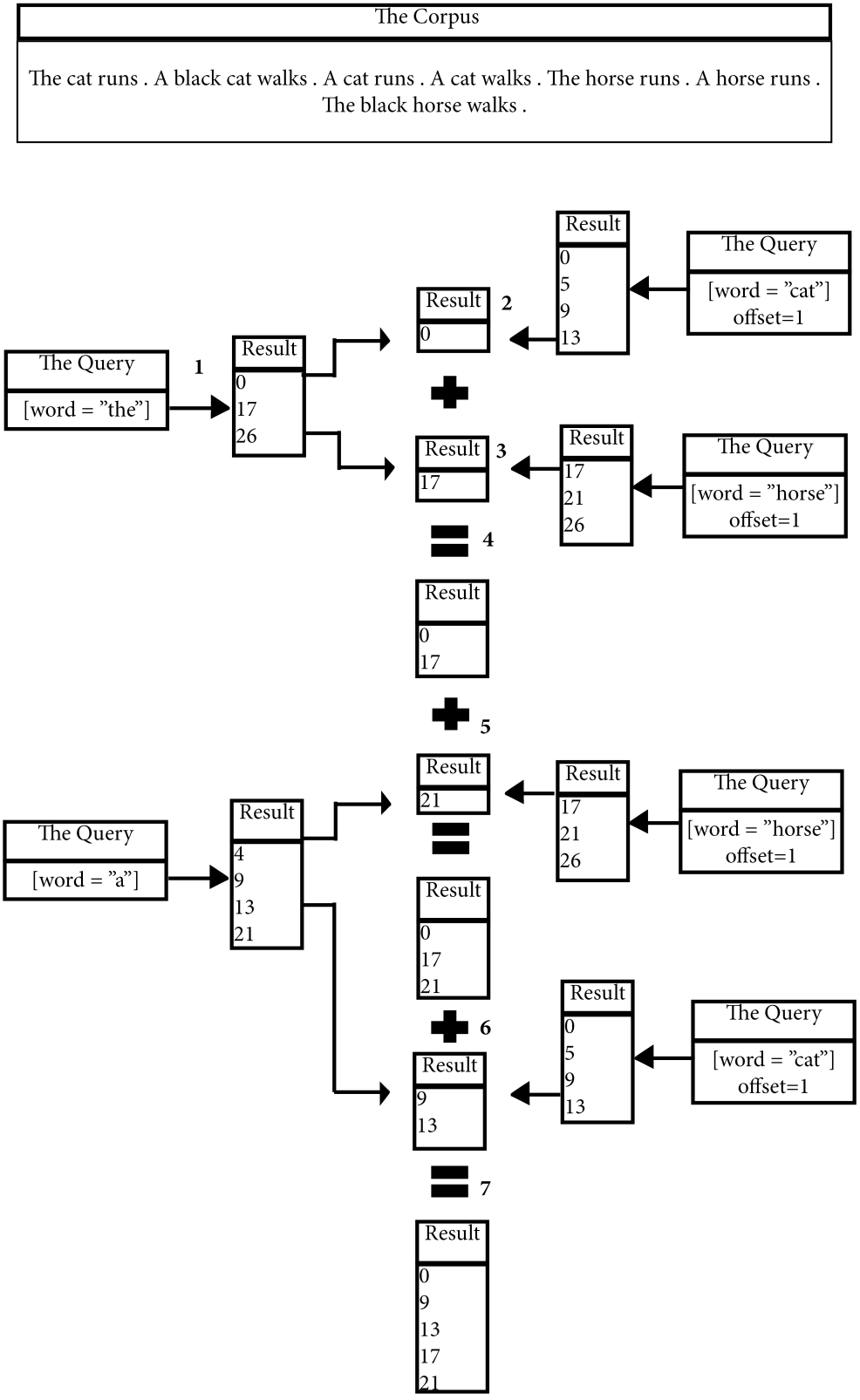


Figure 3.1: Illustration of how union is handled in disjunctive queries.
This is an illustration of how the union is taken of the results of all possible queries in a disjunctive query. Results are found and sequentially added in order of size.

3.2 Regular Expression Querying

To get some of the expressivity that CWB has one natural thing to enable in Korpsearch was the ability to use regular expressions when searching. This was implemented using disjunctive queries. All words matching the expression are found, after which a disjunctive query is formed around them. All words are found in the corpus's word list where each word is only included once, not in the corpus. This implementation is reasonable because there are a lot fewer words in a given corpus than there are tokens. We will not describe regular expressions in here. Interested readers are instead directed to Python's documentation of their regular expression engine[6].

Let us describe the process of performing a simple regex search when searching for `t.*[eg]`, which evaluates to all words starting with t and ending in either e or g. This is visualized in Figure 3.2. The example corpus and the array containing all the words can be seen in the figure. The first step is to search for all matches of the regular expression in the file containing all words. This gives us the start and end positions of all words containing strings matching our criteria. We now know for example that the first instance starts at position 28 and ends at position 29. However, looking at the words array we can see that this is not a complete match, but simply part of the word *escorted*. Therefore, this is not a true match. To remove these false findings in step 3, we cross reference the found matches with a list containing all the positions where new words start, to make sure that our matches line up. Finally in step 4 we find what words correspond to the found positions and use that as our result. These words, *tame*, *the*, and *trespassing*, will be converted into a disjunctive query for the purposes of executing the query. The query `[word='t.*[eg]']` would, in this corpus, be turned into the disjunctive query `[word='tame'|word='the'|word='trespassing']` before execution.

3.2.1 Prefix

Now the approach used in the previous example is possible to use to find prefixes and suffixes. Its implementation based in disjunction, however, makes it somewhat slow. Therefore, more specific prefix and suffix routines were created. Instead of building on disjunction they build on hacking the result finding function found in Korpsearch to accept multiple values utilising that all words matching a prefix are in the same place of a lexicographically sorted list. This 'hack' is best understood by comparing of Figure 2.1 with Figure 3.3.

The philosophy behind our approach was to use the relationship the prefix has to the full feature. Let's say that you are looking for the tokens that have the prefix 'th' on their word feature. As mentioned in Section 2.2, were it not a prefix search, Korpsearch, and thus our program too, would be conducting a binary search utilising that the tokens are stored lexicographically, to find the first and the last position that the word can be found at. To find all words matching the prefix, the binary search instead tries to find the first and last occurrence of the prefix, similar to how it works in the standard case, with the following difference. Neither of these searches

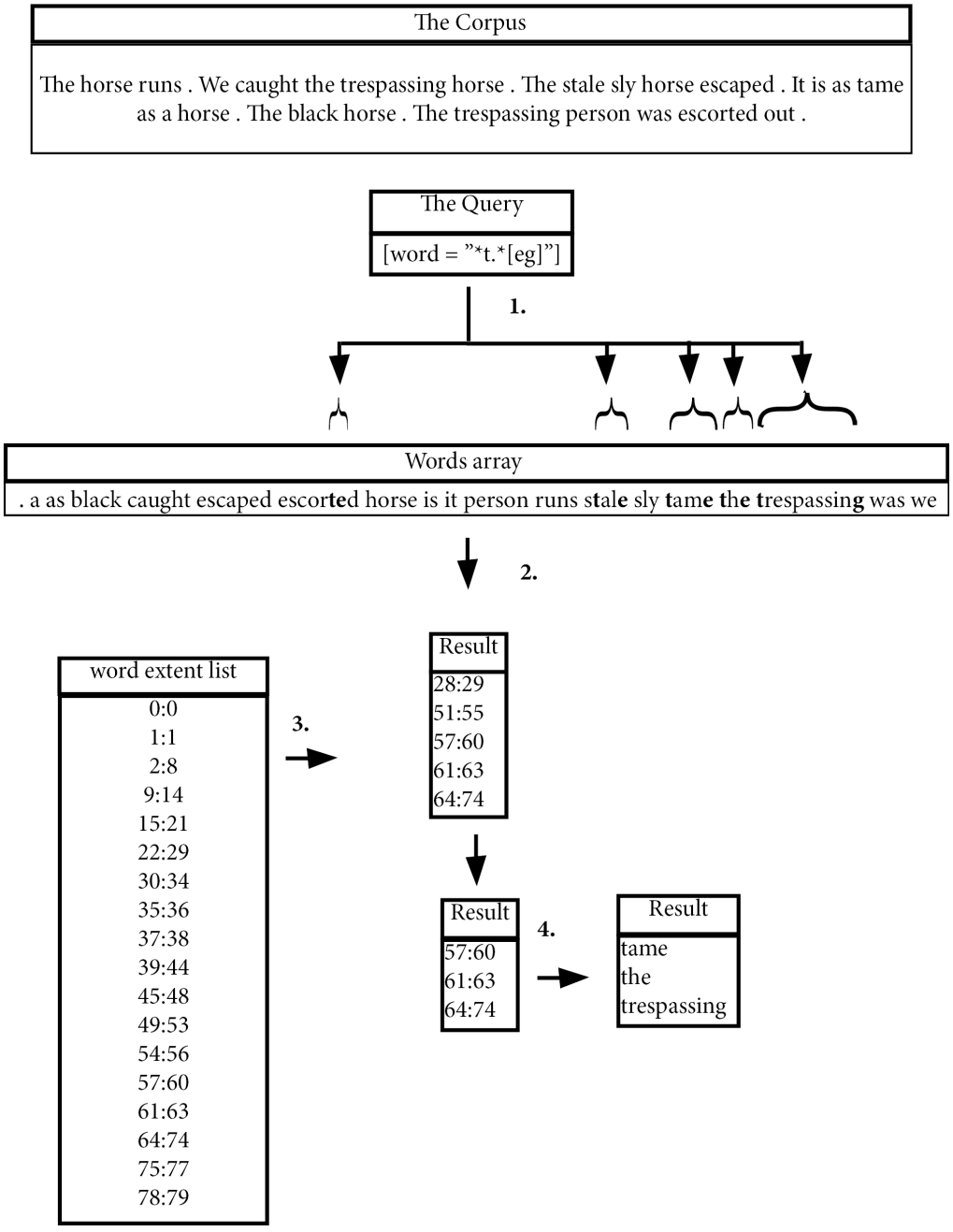


Figure 3.2: Flowchart of processing regular expressions.
Flowchart showing how queries containing regular expressions are processed by finding the list of all words matching the expression in the corpus. The query can then be executed as a disjunctive query between, in this example, word being 'tame', 'the', and 'trespassing'.

will return a match in the case of searching for the prefix 'th'. Instead, when both fail to find anything and terminate we know that the token just after the first search spot's termination is the first token matching the prefix, and the token just before where the search for the last token terminates is the last result. Using this strategy, we can use the already existing search algorithm with some augmentation to fetch the results matching a prefix instead.

A prefix search might return multiple tokens of different values, e.g. different words or different parts of speech. Because of this, some false positives could be included in the result when using binary indexes. For instance one could have binary indexes for [tame horse; timid cat, trespassing horse] lexicographically sorted. Performing our prefix algorithm on this list would mean finding that 't.* horse' starts at *tame horse* and ends at *trespassing horse*. Finding all the indexes between those two will include the index for *timid cat*. This is a false positive, and its corresponding occurrences of 'timid cat' must not appear in the final result. To remedy this one needs to filter the results, and thereby cancelling any time saved by the utilisation of binary indexes. We therefore chose not to use binary indexes when searching for prefixes.

Having construed a way to find all tokens matching a prefix on a feature, we also need to sort the results. This is because the intersection function relies on both the results being intersected to be sorted in ascending order. Since prefix results can span multiple different words, and the corpus positions are only sorted on a word-by-word basis, this condition is not met. Having sorted the results of a prefix search one can treat it as any other result, intersecting as normal.

If the prefix matches enough tokens, sorting and intersecting might not be beneficial. Therefore another strategy was developed to deal with such cases. Dubbed 'the throwing out'-method it consists of simply ignoring the prefix literal, and filtering the final result on the condition given by the prefix at the end. This throwing out method is utilised in all cases where the prefix result is not so small that it should be search-based intersected with. When the prefix result is so big that merge-based intersection will be used when finding its intersection with a different set, that other set is close enough in size to the prefix set. In that case the cost of sorting the prefix result is deemed not worth it and the other set is used instead as the first set in the intersection queue mentioned in Section 2.2.2.

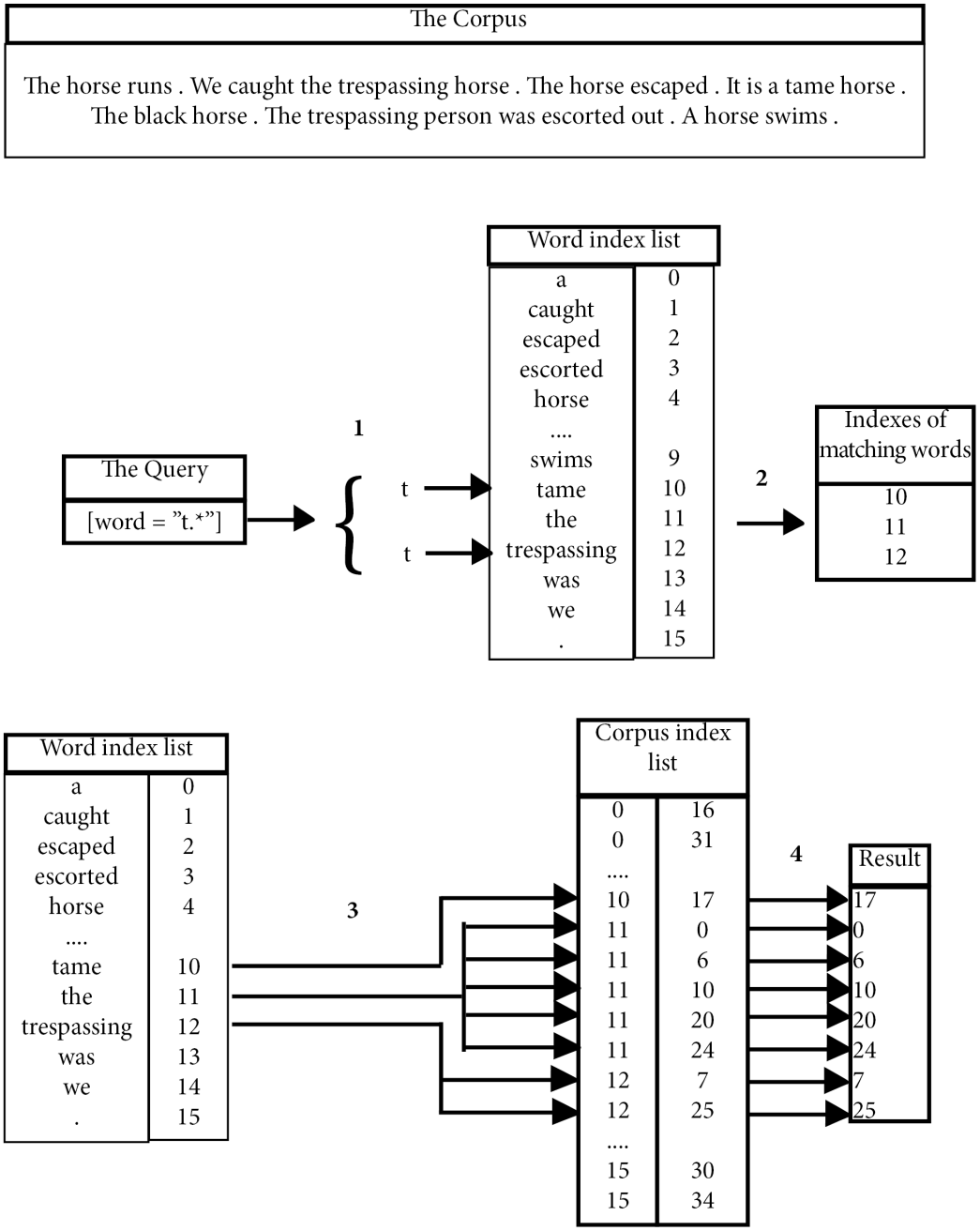


Figure 3.3: Flowchart delineating query execution for prefix queries.
This is an illustration of the change made to Korpsearch's result finding algorithm to account for the value being a prefix. The changes are in step 2 where, a range of indexes 10, 11, 12 is returned, and in step 3 where we find the first 10, and the last 12, with all the results being between those two. The final result is also not sorted in this scenario.

3.2.2 Suffix

	The Corpus
word	The horse runs . We caught the trespassing horse . The horse escaped . It is a tame horse . The black horse .
drow	eht esroh snur . ew thguac eht gnissapsert esroh . eht esroh depacse . ti si a emat esroh . eht kcalb esroh .
ammel	eht esroh nur . ew hctac eht gnissapsert esroh . eht esroh epacse . ti eb a emat esroh . eht kcalb esroh .

Figure 3.4: Example Corpus of reversed features.

Adding suffixes onto the program was done according to the philosophy that reversing a string turns suffixes into prefixes. For example the word 'the' has suffix 'he', and the reversed word 'eht' has prefix 'eh'. Reversing the strings in the previous example corpus in Figure 1.1 we obtain the features shown in Figure 3.4. This meant that the solution revolved around creating a reversed feature for every feature, and to search for the reversed suffix the same way prefix was sought before on the standard feature.

Not all features are interesting to do a suffix search on, such as the part of speech for example. These will not be generated if not desired by the user. This means that the increase in indexes only takes place as it helps the user.

3.3 Approximation of intersections

When optimising the order in which queries will be intersected or have their union found, not only their sizes matter, but also the size of the intersection of two sets. This is relevant as it is the intersection that will be used when performing future intersections and unions. Different strategies of executing a query are optimal for different assumptions one makes on the size of the intersections, so it does indeed matter which assumption one makes. We will go into more detail on this result after a minor detour. The size of the result of unions also matters for disjunctive queries, but the assumption that $a \cup b$ has a size of $a + b$ seems reasonable, as one typically would have a disjunction between values on the same feature leading to there being no overlap. Word cannot very well be 'horse' and 'pig' at the same place in the sentence. Since there is no rival to this approximation on the size of the result of a union, there is no reason which merits another strategy. The execution strategies are therefore based in the assumption that unions result in a set that is as big as the sum of the two sets unioned. There are two assumptions that we have considered as regards the size of an intersection however.

The size of the intersection of two sets a and b must be between 0 and $\min(a, b)$. This insight leads to the first assumption. The first assumption one can make is that the intersection of two sets a and b, will have the size $\min(a, b)$ times some constant factor. We chose not to use this assumption, for reasons which we now describe.

The previously discussed assumption entails that the same percentage of a will be in $a \cap b$ no matter the size of b. But it is reasonable to assume that the size of b matters too, which is the second possible assumption we present. One way to take this into account would be to say that if b contains 1% of the whole corpus, a's intersection on b will be $0.01 \cdot a$, whereas if b contains half of the corpus, a's intersection on b will be $0.5 \cdot a$. This assumption feels more reasonable since $a \cap b$ ought to be much smaller than $a \cap c$, if b is much smaller than c, even if a is the smallest of the three. This assumption still doesn't take into account any relationship between the two literals intersecting, as for instance `[word="the", offset=0]` ought to have a bigger intersection with `[pos="SUBST", offset=1]`, than with `[pos="VERB", offset=1]`, due to how the English language works. Finding a way to take these kind of relationships into consideration without actually computing the intersection is very difficult though. Made even more so by the fact that the corpus querying algorithm ought not be language dependent, meaning that insight into English word order does not constitute optimising information.

3.3.1 Motivation of approach

Intersections of a and b are assumed to have a size of $\frac{a \cdot b}{tot}$. Why does this mean that it is cheaper to perform intersections first and unions later? Let us start by looking at a small query $(a \cup b) \cap c$, i.e. (a or b) and c, and compare the two strategies of either finding the union of a and b first, written $(a \cup b) \cap c$, or the strategy of finding the intersections ac and bc first, written $(a \cap c) \cup (b \cap c)$.. We are in this section going to use parentheses in this manner to show what set actions are completed in what order. Before delving into the motivation it is also important to note that using merge-based intersection on a and b has a time complexity of $a+b$, that is the sum of the sizes of a and b, whereas the time complexity of using search-based intersection with a on b is $a \cdot \log(b)$, with the italicised letters once again referring to the set's sizes. The costs of using the two strategies vary depending on the relative sizes of a, b, and c.

Let us start with when c is so much larger than a and b that one should use search-based intersection, $a, b \ll c$, when finding the intersection with c, while a and b are about equal in size, $a \approx b$. The signs '$\ll$' and '$\approx$' are going to be used to mean those relationships throughout this section. Then the union first strategy costs $a+b+(a+b) \cdot \log(c)$, whereas the intersection first costs $a \cdot \log(c)+b \cdot \log(c)+(a+b)\frac{c}{tot}$. The latter expression will always be smaller since c must be a subset of the corpus.

In the scenario $a \approx b \approx c$, then the union first strategy costs $a + b + a + b + c$, whereas the intersection first costs $a + c + b + c + (a + b)\frac{c}{tot}$. Since the sets are of about equal size in this scenario, the difference is an extra a and b in the union first strategy, and an extra $c + (a + b)\frac{c}{tot}$ in the intersection first strategy. Using the fact that they are all of about equal size, we get that $\frac{2c}{tot}$ should be greater than 1, for

the union first strategy to be superior, which happens when c contains more than half the corpus. This is not going to be the case when running real queries, why intersection first wins this scenario as well.

What if $c \ll a \approx b$? Then the union first strategy costs $a+b+c \cdot \log(a+b)$, whereas the intersection first costs $c \cdot \log(a) + c \cdot \log(b) + c \cdot \frac{a+b}{tot} = c \cdot \log(a \cdot b) + c \cdot \frac{a+b}{tot}$. If we say that a and b are about the same size, we get $c \cdot 2 \cdot log(a) + c \cdot \frac{a+b}{tot}$ in the intersection first case compared to $a+b+c \cdot \log(a)+c$ in the union first case. Where the cost for the intersection first case is smaller since c is much smaller than a and b.

Though this is not a complete proof by any means it strongly indicates that intersection first is the better strategy in most any scenario. These scenarios have all been with but one set outside of the disjunctive subquery. How does this extend to when there are more queries to intersect with, or multiple disjunctive subqueries? One can intuitively see that stacking more sets to intersect with unto the scenario wouldn't change the strategy as the first intersection should come before the union and then, you have the same choice with the added set, whether to intersect with that first, or to union first, and one will come to the same conclusion. With another disjunction it isn't as clear, but one could imagine the query (a or b)(c or d). Given c we would have the same intersection or union first with (a or b) as before, and the same is true given d. Since the intersection first strategy is better for both it ought to be good for (c or d). This is in no way proven, but it makes sense enough to develop a strategy from.

This is the mathematical algorithm analysis that has led us to make a query into disjunctive normal form, DNF [1], and find intersections first before unioning. In practice, this has led to the program finding all permutations when executing a disjunctive query. The individual permutations will correspond to non disjunctive queries executed, and are executed the ordinary way. After this all of the results are unioned. This is the method described in Section 3.4, which is in agreement with the time complexity findings.

The motivation in this section has not taken binary indexes into account. Binary indexes can only be used before any unioning has occurred. The intersection first strategy unions at the last possible step. Thus the strategy found best without taking binary indexes into account, intersection first, is the one that would benefit the most from them. The program's use of binary indexes, therefore, does not change the fact that finding intersections first, and unioning results after that is superior in regards to the time complexity.

[1] To read more about DNF:s see `https://en.wikipedia.org/wiki/Disjunctive_normal_form`

4

Results

In this chapter we present the result in two main ways. We first show some examples of queries that are now executable, that weren't executable in Korpsearch before our project. That is to say we show some prefix, disjunctive, and regex queries rendering the correct results. After this we show how the time taken to execute some example queries differs between our program and Corpus Workbench.

4.1 Extension of querying abilities

The first part of presenting the result is presenting the types of queries that are now executable, that were not previously.

4.1.1 Disjunction

Our program has added the ability to include disjunctive literals in queries as was the goal with this part of the project. Whilst before one could perform the four queries:

```
[word="a"][word="dog"],
[word="the"][word="dog"],
[word="a"][word="horse"],
[word="the"][word="horse"],
```

one can now query:

```
[word="the"|word="a"][word="dog"|word="horse"],
```

gathering all those queries into one. This type of query is also used by linguists so ought to help them in their work. We have not implemented disjunction between subqueries. Because of this queries such as:

```
([word="the"][word="scruffy"][word="dog"])|([word="a"][word="horse"])
```

are not queriable at present. Some thoughts on how one might implement them can be found in Section 5.2.

4.1.2 Regular expression search

We have enabled searching corpora using regular expression strings which allows users to be more expressive in their searches. The program utilises Python's regular expression engine and therefore has inherited its expressability.

With these capabilities, something that can now be done is to do disjunction from within a literal, i.e. to search for `[word="cat|dog"]`. We can also search for things such as words in singular and plural, `[word="cats?"]`. The question mark here signals that we are looking for cat followed by at most one 's'.

4.1.3 Prefix and suffix

The program is able to handle all forms of prefix as well as all forms of suffix. The program that ours is based on would have been able to handle a query such as `[word="the"][pos="ADJ"][word="horse"]`. In addition to this, our new program can handle queries such as `[word="t.*"][pos="ADJ"][word=".*se"]`, and `[word="t.*"][word=".*se"]`. That is to say it manages to handle prefixes and suffixes correctly, whether both are used, and whether all literals being prefix or suffix.

4.2 Time comparison to corpus workbench

Corpus Workbench is as previously mentioned another commonly used querying tool for searching a corpus. To get a comparison for the relative improvement in execution speed we will therefore compare time taken by our program and Corpus Workbench to execute queries containing the relevant literal types, i.e. containing prefix searches, and disjunction. The queries are made and timed on the British National Corpus [7] on a laptop with an i7 processor. The comparisons are presented in tables containing examples of both programs performing poorly. The times that are being compared are gathered from 'internal' timings, meaning that the programs where already started when the timing commenced. Therefore, start up times for the programs are not included, but since these are static and small this should not detract from the conclusions.

4.2.1 Prefix and suffix

Query	Korpsearch (s)	CWB (s)
`[word="t.*"]`	8.19	2.17
`[word="ho.*"]`	0.22	0.17
`[word="ho.*"][word="a.*"][word="p.*"]`	1.35	2.97
`[word="t.*"][word="a.*"][word="p.*"]`	3.80	10.91
`[word="t.*"][word="ho.*"][word="a.*"][word="p.*"]`	1.59	10.27

Table 4.1: Timing results of Korpsearch and CWB for prefix searches.

Our program is mostly quicker than Corpus Workbench except for the case discussed in the paragraph below. The time taken for individual queries can be seen in table 4.1.

The search for `[word="t.*"]` took 8 seconds for our program and 2 seconds for Corpus Workbench. Our program is quicker until it sorts the list of results. Sorting is unnecessary here, since the prefix is not followed or preceded by another term, and here Corpus Workbench skips the sorting step. Though if it was part of a disjunction it does need to be sorted, or one will have to accept there being duplicates. Given the time difference, figuring out if sorting is needed seems like the better solution.

Something else worth noting is that Korpsearch can get a significant speed up when adding a literal which yields few matches anywhere in the query. In Table 4.1 this effect can be seen when adding `[word="ho.*"]`, as it has fewer matches than the other subqueries that match all words beginning with a certain letter. The same is not true for Corpus Workbench. It only speeds up if the smaller set is the first in the query. We can see the effects of this for both programs in the last two queries in Table 4.1.

4.2.2 Disjunction

Query	Korpsearch (s)	CWB (s)
`[word="the"|word="a"][word="cat"|word="dog"]`	0.075	9.14
`[word="the"|word="a"|word="one"]` `[word="cat"|word="dog"|word="horse"]`	0.13	9.59
`[word="the"|word="a"|word="one"]` `[word="cat"|word="dog"|word="horse"][pos="VERB"]`	0.28	9.70

Table 4.2: Timing results of Korpsearch and CWB for disjunctive searches. Timing results of Korpsearch and CWB for disjunctive searches. Note that the second and third query are both two lines long.

Our version of Korpsearch is consistently quicker at executing disjunctive queries than Corpus Workbench. The results for the individual queries are in Table 4.2.

4.2.3 Regular expressions

Query	Korpsearch (s)	CWB (s)
`[word="he|she|they"]`	0.17	0.19
`[word="(t(?=hey)|s)?he((?<=the)y)?"]`	0.35	0.32
`[word="(t(?=hey)|s)?he((?<=the)y)?"][pos="ADJ"]`	0.29	1.30
`[word="a|the"][word="cat|dog"]`	0.37	9.12

Table 4.3: Timing results of Korpsearch and CWB for regular expression searches.

Korpsearch can now be used to query for any query consisting of terms describable by Python's flavour of regular expressions. We can see in Table 4.3 that CWB outperforms Korpsearch on single literal queries but Korpsearch takes over when more elements are added. We can also see that though Corpus Workbench is quicker for the first two queries it is slower for the latter two. This is because Corpus Workbench can quickly find the result of a subquery with its many indexes, but is quite slow at finding intersections.

Two queries in this table also bear explaining. Namely the second and the third one. These queries, as the first one, describe the words: he, she and they. Here, so called 'look-aheads' and 'look-behinds' are used to make sure that when the 't' or the 'y' is matched, they are actually followed or preceded by the parts needed to describe the wanted words. We also use grouping which allows us to put a conditional '?' for the 't' or the 's'.

Lastly it is interesting to note the time that the program spends finding all words matching the regular expressions. This differs with how complex the regular expression is and how common your search string is, but for the cases presented in Table 4.3 the times are: 0.041, 0.18, 0.18, 0.32 seconds.

5

Conclusion

There are some conclusions one can draw from the results of this project. The primary, we would say, is that disjunction can be implemented by dividing a disjunctive query into all the possible permutations of subqueries. Then one can find their result through a quick intersection finding program. This extends to being able to find all words containing a certain string, and general Python regex matches. All these kinds of queries can be executed in a matter of seconds even on big corpora.

Our program also shows that one can depart from the disjunction solution path, the path that is to find all matching words and then form a disjunctive query between them, in special cases. Special solutions have been made for prefix and suffix which have managed to have shorter execution time for those kinds of queries.

5.1 Discussion

In this section we will discuss the different parts of the project. The discussion will cover: the results, execution time, enabling of query types, how they fulfil the goal, as well as in what ways they have failed to fulfil the goal. We will also discuss what future work could be performed within the area, and how one could build on our program to achieve some of that future work.

5.1.1 Discussion of time

Corpus Workbench required nearly ten seconds to execute the rather short query `[word="a"|word="the"][word="dog"|word="cat"]`. Our program required only 0.35 seconds. This is a large improvement, that ought to be bigger for more complex queries. This improvement will enable researchers to execute queries in a reasonable time, that with Corpus Workbench would take too long to consider executing perhaps. Corpora can be bigger by a factor 10 than the corpus we've used when gathering results, so Korpsearch's improvement can be even bigger on those colossal corpora.

5.1.2 Query types

Our program is able to execute the types of queries found to be common in the log of linguists at Gothenburg University's queries. It has therefore achieved meaningful

progress towards being a valid substitute to Corpus Workbench, since it is able to do the same job but more quickly.

5.1.3 Failings

As one can see in the Results chapter, our program was much slower at executing the prefix query `[word="t.*"]`, than Corpus Workbench. As was previously stated this was due to us sorting the results of the prefix subquery. The sorting is done so that it is possible to intersect or union it with other queries, but in this case there are no others. This slower speed could therefore be fixed by checking whether the prefix subquery contains the only literal, and in that case does not require sorting. This sort of query of having only one literal, which also is a prefix literal, is quite common in the dataset we have of linguists searches, so it ought to be fixed.

In its current form our program is incapable of handling disjunctions paired with another requirement in a token, e.g. `[word="the"|word="a" pos="SUBST"]`. Fixing this would require an augmentation of how disjunctions are handled currently and define how to interpret the extra requirements. This type of query does not seem to occur in our list of queries linguists make, so it should not be too damning for the program not to be able to execute them.

5.2 Future work

There is work still to be done to improve this software. This work can be put into two categories: work that can be built on our solution and which should therefore be easier to implement in the future; and work that would improve the solution, allowing for more expressive queries, though they are not necessarily easier to implement now than they were before we did any work.

One feature which would be good to implement in the future is multi-token disjunction. What we mean by that is disjunction between several tokens. For instance, this could be:

```
[word="the"][word="beautiful"][word="horse"]|[word="a"][word="cat"]
```

where we have two sets of tokens, with different amount of tokens. This cannot be executed with the current version of disjunction as the first token affects the latter ones. If the first token has the word being ‘a’, the second token must have the word being ‘cat’, ‘beautiful’ no longer being an option since the first word is ‘a’. One could implement it as we have implemented single-token disjunction forming all possible combinations as queries, ‘the beautiful horse’ and ‘a cat’, and unioning their results. This has, however, not been done since the actual implementation requires time that has been prioritised elsewhere. It is also due to the fact that the solution one implements ideally should hold to our principle of keeping execution time similar to Korpsearch, which requires some thought put into optimisation.

If one implements multi-token disjunction one could use that to implement repetition queries. Imagine that one, for instance, had the query:

```
[word="the"][word="ADJ"]*[word="horse"],
```

searching for 'the horse' with any number of adjectives between 'the' and 'horse'. One could in that case look up the number of possible adjectives that can appear in a row in the corpus one is working in. Let us say that up to three adjective can come in a row in the corpus. In that case one can split the query using multi-token disjunction into the queries:

```
[word="the"][word="horse"]
[word="the"][word="ADJ"][word="horse"]
[word="the"][word="ADJ"][word="ADJ"][word="horse"]
[word="the"][word="ADJ"][word="ADJ"][word="ADJ"][word="horse"]
```

which cover all the possible number of adjectives, and therefore will return all results to the repetition query. As with all the disjunction implementations we implement one would then union the results to get the result of the repetition query. If one implements multi-token disjunction, and a good way of knowing the number of times the repeating token can repeat, then repetition implemented this way is a good step in extending expressiveness further.

One thing that ought to be introduced is the ability to have optional subqueries, or to allow for a certain or arbitrary number of repetitions of tokens matching a literal. This all falls under repetition, and could be implemented using a version of disjunction between groups of literals. Our current solution to disjunction cannot find the proper result set when the disjunctive branches are of varying lengths as this leads to there being different offsets between the various branches.

Regarding disjunctive queries one could also have 'or'-statements on groups of literals, e.g.

```
([word="the"][word="horse"]|[word="an"][word="owl"]).
```

We have not aimed to solve this case, but it would add expressiveness, and thus is a reasonable continuation of our work.

The implementation of contains and general regex matches increases the types of queries our program is capable of by quite a lot. They are, however, not optimised in other ways than relying on our solution for disjunction. In order to in the future have the ability to use them for bigger corpora as parts of larger more complex query optimisation would be beneficial. Since they are converted to the quick disjunction implementation they still are quite quick for the contemporary size of corpora. Regex could however be improved by implementing something similar to what Corpuscle does.

One could save information between disjunctive strands in a lot of cases especially with repetition. This is good future work, since a lot of the program builds on disjunction. This is best explained through an example. Imagine that you have `[word="cat"|word="dog"][pos="VERB"][word="a"]`. As it currently stands our program would first execute `[word="cat"][pos="VERB"][word="a"]` and then execute `[word="dog"][pos="VERB"][word="a"]`. In both those queries we have the part `[pos="VERB"][word="a"]`, which will have the same result. If one executes

that part of the first query first, one ought to be able to save the result from it and use it for the second query. Figuring out when this is worth it, and how it should be implemented is beneficial future work.

Another improvement one can make in the future is to have a maximum number of disjunctions arising from a regex literal before it is dropped. If we, for instance have that a regular expression matches 5 words in the corpus that is fine. If the regular expression instead matches all words containing the letter 'p' the program will not be very quick. This is because a disjunction between that many sets will take a long time to execute. It seems plausible that there is some cutoff point where it would be better to throw out the regular expression literal, and to instead filter the results at the end for the literal's condition. This is similar to what has been done for the specialised prefix implementation. This change would not change the execution time of most queries, but where our program currently performs worst it might lead to big improvements in execution time.

5.2.1 Repetition queries

Repetition queries allow for any number of tokens matching the repeating literal to be in its part of the query. Keeping repetition to one literal, as in allowing `[pos="ADJ"]*[word="horse"]`, but `([pos="ADJ"][word="horse"])*` not being allowed, there are two main cases. Either a repeating literal is at one end of the query or not. If it is at one end of the query, then the rest of the query is executable without taking the repeating literal into account. Since zero tokens matching the repeating literal is also a match of the query, the results found ignoring the repeating literal are then the actual results, and it can be optimised away. The real work, therefore, comes in when the repeating literal isn't at one end but rather has literals to either side. Without any information on how sentences are structured in the corpus, any distance between literals to either side of the repeating literal is possible. The best way to account for this is hard to say without having developed repetition queries, but one can consider some different ways. One tool is the binary indexes inherited from Korpsearch, allowing finding different amounts of tokens matching the repeating literal in a row more quickly, assuming that two of the repeating literal next to each other is common enough to have warranted a binary index. One could also have a solution based in the maximum number of tokens in a row that match the literal in the entire corpus act as an upper limit to how many there can be, and then turn the repeating literal into a disjunctive query. This is assuming that such a limit is feasible to find. A third option would be to search for the other literals with any distance possible between the literals to either side of the repeating literal, and then filtering on that distance consisting only of tokens following the condition imposed by the repeating literal.

Bibliography

[1] *What is a corpus?* [Online]. Available: https://www.awelu.lu.se/language/corpora-resources-for-writer-autonomy/what-is-a-corpus/.

[2] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," *siam Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[3] S. Evert and A. Hardie, "Twenty-first century Corpus Workbench: Updating a query architecture for the new millennium," 2011.

[4] P. Ljunglöf and N. Smallbone, "Efficient corpus search using unary and binary indexes," 2022.

[5] L. Borin, M. Forsberg, and J. Roxendal, "Korp the corpus infrastructure of Språkbanken," in *Proceedings of LREC 2012. Istanbul: ELRA*, vol. Accepted, 2012, pp. 474–478.

[6] *re Regular expression operations*. [Online]. Available: https://docs.python.org/3/library/re.html.

[7] *The British National Corpus*. [Online]. Available: https://www.english-corpora.org/bnc/.

[8] L. Anthony, "A critical look at software tools in corpus linguistics," *Liguistics Research*, vol. 30, no. 2, p. 145, 2013.

[9] M. Davies, "The advantage of using relational databases for large corpora: Speed, advanced queries, and unlimited annotation," *International Journal of Corpus Linguistics*, vol. 10, no. 3, pp. 307–334, 2005.

[10] N. Diewald and E. Margaretha, "Krill: KorAP search and analysis engine," *Journal for Language Technology and Computational Linguistics*, vol. 31, no. 1, pp. 63–80, 2016.

[11] P. Meurer, "Designing efficient algorithms for querying large corpora," *Oslo Studies in Language*, vol. 11, no. 2, pp. 283–302, 2020.