TR-IT-0287

Analysis by Analogy:
A Set of Experiments and Some Preliminary Results

Jean-François Morreeuw　　Yves Lepage

February 1999

Abstract

Analogy, analysis by analogy, experiments.

Table of contents

1 Introduction **4**

A - What: experiments with analogy 4

B - Why: using a syntax without rules 4

C - How: data, experiments, and results 5

2 Preparing the data **6**

A - Listing the data that we have 6

A1 - Different types of data 6

A2 - Problems coming from the different formats 7

B - Tools to prepare the data 8

B1 - Checking and formatting the data 8

B2 - Tool to cut the data 9

C - Computing some properties 10

C1 - Program to compute the properties 10

C2 - Properties of the data 10

3 Experimentation methods **15**

A - Presentation of the experiments 15

A1 - The maps: a way to see numerical results 15

A2 - Open-set and closure: how to get some literal results 17

A3 - Table of the experiments 18

B - Defining a common engine for all the experiments 18

B1 - Reasons for such an engine 18

B2 - A new view of our experiments 19

B3 - Output of the common engine: scheduling file 19

C - Detailed description of all experiments 20

C1 - Computing and exploiting the numerical maps 20

C2 - Open-set: computing a base 22

C3 - Closure of a corpus 23

4 Experiments' engine: the scheduler **24**

A - A didactic overlook of the algorithm 24

A1 - A naive algorithm 24

A2 - Verify = solve + search 25

A3 - Getting an incremental algorithm 25

A4 - Using the properties of analogy 25

B - Example of results 28

5 An experiment: analysis by analogy **29**

A - Presentation of the experiment 29

A1 - Description of the experiment 29

A2 - Description of the data used 30

B - Results 31

B1 - Number of calculated analysis 31

B2 - Quality of the analysis 32

C - Conclusion 32

Appendix A **34**

Appendix B **36**

Chapter 1

Introduction

A - What: experiments with analogy

Analogy is a relation between four terms. If we consider the first three terms, we can say that the fourth term can be obtained by analogy when the relation of analogy is verified. Here is an example of analogy:

" *Thegreenlampsturnsoff*" : " *Thelampturnson*" = " *Thegreensignalisoff*" : " *Thesignalison*"

As we can see, analogy can creates new sentences, using a correct syntax. Our goal is to experiment the properties of analogy with real corpora to see how this relation can be used to create some new tools, or to improve some methods like sentence analysis.

B - Why: using a syntax without rules

The main point with analogy is the possibility to create new sentences with a correct syntax, without describing this syntax. This can be seen in the following example. The rule is involved by the two first term, and the three terms are used as a complete description of this rule.

Thetimeislong : *Thereisalongtime* = *Thecatisblue* : *Thereisabluecat*

As we can imagine with syntactic categories, we can describe any LR1 grammar with such analogies, without writing directly the rules. This is a very useful property, as a software using analogy should be able to work with any language if you give to it the proper analogies that could involve automatically the needed rules.

To do so, we need a corpus of examples. As we can use very large corpora, coming for example from internet, or directly from the ones used in ATR, we hope that a big corpus could act as an example corpus, containing enough involved rules for our use.

C - How: data, experiments, and results

As we need some corpora, the first thing we have to do is to filter and transform some various files, coming from different origins, to be able to experiment with analogy. This will be the first step of our study.

Then, we need to characterise the different corpora with analogy, to choose the ones that could be use with analogy. As we want to be able to do so with any new corpus, we have to realise a complete set of tools, and this will be the second part of our job.

Our goal during this internship is not to experiment with analogy, but to realise the tools used to experiment. But as we find interesting to give a proof of the possibilities that can be afford with analogy, we will give some preliminary results with a first real use of analogy: sentence analysis by analogy.

Chapter 2

Preparing the data

As we have told before, we want some results dealing with analogy. To do so, we have to realise some experiments with data coming from different horizons, from different types and formats. The preliminary works to do are first to collect and select the useful data that we could find, then to uniform them to be able to realise generic programs to experiment. After showing the data we want to use, we will present some tools we have done to format them, and then we will list a few properties of those data.

A - Listing the data that we have

A1 - Different types of data

From about 14 000 files, making an amount of approximately 38 megabytes of information, we can find data in various languages, using different ways of encoding, or simply different types of data. We have files containing some text, some syntactic analysis, and some sentences of syntactic categories. Also, we have text in French, English, Japanese with kanji and kana or with roman alphabet.

The following tables are summing-up the different information about the files, after a first selection. The sizes are given in a number of lines, a board is a couple of (sentence, tree).

File	Size	Language	Type	Remark
`ATRLancaster` *English*				
`baa.board`	23 055	English	boards	
`baa.text`	23 055	English	sentences	
`baa.tree`	23 055	English	trees	contains some attributes
`baa.tag.board`	23 055	English	boards	same trees in `baa.board` and `baa.tree`
`baa.tag.text`	23 055	English	codes	
`words`	17 727	English	words	words from `baa.text` in alphabetic order

File	Size	Language	Type	Remark
`NEC` *japonais*				
`nec.tree-bank`	6 553	Japanese	boards	the trees contain the segmentation of the sentences
`nec.text`	6 553	Japanese	sentences	not segmented sentences
`nec.tree`	6 553	Japanese	trees	trees from `nec.tree-bank`

File	Size	Language	Type	Remark
English-Malays *English/Malays*				
win.eng	247	English	sentences	translation of win.mel
win.mel	247	Malays	sentences	

File	Size	Language	Type	Remark
JEK *Japanese/English/Korean*				
E-sentence	12 320	English	sentences	no punctuation, segmented sentences
J-sentence	12 373	Japanese	sentences	translation of E-sentence

File	Size	Language	Type	Remark
TDMT *Japanese*				
good.tree-bank	1 448	Japanese	sentences and trees	each sentence is associated with two trees containing the segmentation
tdmt.text	724	Japanese	sentences	sentences segmented, no punctuation
tdmt.tree	724	Japanese	trees	contain some attributes and words with Hepburn transcription
tdmt.pattern	724	Japanese	codes	trees and syntactic categories, words with Hepburn transcription

File	Size	Language	Type	Remark
UPENN *English*				
all.org	787	English	trees	contain words
text	787	English	codes	codes can be associated to words from all.org with the pos2char file

Directory	Size	Language	Type	Remark
atr:tb/LMD/dialogue *Japanese/French/English*				
atr:tb	32	jp/fr/en	trees	one tree in each file
LMD	59	French	texts	unformatted texts
dialogue	9	English	boards	9 dialogues for 187 boards

From these first selection, we have chosen data from which we were able to have a couple of information, like both trees and sentences, or sentences and syntactic categories, or categories and trees. We call such a couple a corpus. In the following, the data we are using are coming from ATRLancaster, TDMT, UPENN and dialogue.

A2 - Problems coming from the different formats

The first thing we have to do is to choose a unique format to convert all the data, for sentences, trees and syntactic categories. Our choice is to have only one information for each line of a file. Texts are double quoted, words and punctuation are separated with empty characters, and if possible written using roman alphabet. Trees are in the format used by existing tools dealing with tree operations.

Extracts from `tdmt.jap.txt`:

"そちらに フィットネス 宿泊 パック というのが 有 ると 聞 いたんですけれども"
"1 泊 は 確 か 200 ドル 程度 でしたよね"
"来週 の 土曜日 に 予約 をしたいんですけれども"

Extracts from `tdmt.jap.tree`:

```
SE(SM(SP(NP(sochira,NP(ND(N\+N(N\+N(fittonesu,shukuhaku),pakku)),aru)),PM(kii))))
SM(SM(NP(1haku,ND(N\+N(tashika,200doru)))))
SE(SM(NP(N\+N(raishuu,doyoubi),PM(yoyakusuru))))
```

We can separate the tools we have done into two parts. The first part is a toolkit used to verify and transform the data, written in C shell. The second part is a tool, realised to correct a problem encountered because of a limitation of the unix tools, written in C language. Then, we will describe a third tool, used to compute some properties to characterise our different files.

B - Tools to prepare the data

B1 - Checking and formatting the data

Mainly two tools have been realised to format the data. Those two tools are written in C shell and are self documented.

The first one, `checktree`, is a program to verify the syntax of trees, able to find and correct the errors that could be in a corpus. The program takes each line of the file, and uses `2Forest` to verify the syntax of this line. If the syntax is okay, the program output the lines verified in stdout, and the errors in stderr. We can use this to redirect the errors in a file to have a list of the errors we have to correct.

```
Usage: checktree [-h] [-v] FILENAME

Check if a file is at the 2Forest format and print errors
* Errors are printed to /dev/stderr
* Working informations to /dev/stdout
/home/as49/lepage/bin/2Forest & /home/xmorreeu/bin/-l is needed

  -h --help     display this help
  -v --version  output version information
```

The second tool, `cleantxt`, allow us to filter the texts. For example, we want the sentences to begin and end with double quotes, and each word and punctuation to be separated with empty characters. Result of `cleantxt` is on stdout. The use of `cleantxt` is the following:

```
Usage: cleantxt [-h] [-v] FILENAME

Clean a text file

  -h --help     display this help
  -v --version  output version information
```

Another tool made is `listtr`, which allow us to use `tr` with a list of multiple operations to do, all transformation written in a file. The file we want to change is inputted in stdin, the result is on stdout, the name of the operation file is given as an argument.

```
Usage: listtr [-h] [-v] REPLACEFILE

  -h --help     display this help
  -v --version  output version information
```

Example of file `replacefile`:

```
France  <--     Japan
ENST    <--     ATR
```

B2 - Tool to cut the data

While we were verifying the trees, we found a few problems with `head` and `tail` that were not able to deal with very long lines. So, we had to realise another tool, to do the same thing, but without this limitation. We have made this program in C language:

```
Usage: -l [-h] [-v] [-BEGIN / -l BEGIN] [+DEPTH / -d DEPTH]

EXECNAME [OPTIONS ... [--counter] ...]
Send DEPTH lines at line BEGIN from stdin to EXECNAME
EXECNAME is launched with optional OPTIONS for each line
In OPTIONS --counter is replaced by the current line number

  -l --line  -...  start at line BEGIN (default is 1)
  -d --depth +...  send DEPTH lines from BEGIN (default is -1 = all)
  -h --help        display this help
  -v --version     output version information
```

The program, named `-l`, reads the data from stdin, select a number of lines depending of the arguments given, and execute a full command for each line. To do so, stdin is duplicated and read. Then, each line is written in a new stdin, which is given to our command. This command is executed the number of lines we have to deal with. Output of the executed command is the standard output of the command. The following extract shows the way used to redirect stdin and create some new temporary stdin for each executed command.

```
void
fonction_principale(void)
{
  char *buffer = (char *) malloc( (BUFFER_SIZE+1)*sizeof(*buffer));
  int new_stdin = dup(STDIN_FILENO);
  ...
  read(new_stdin, buffer, BUFFER_SIZE);
  fill_stdin(buffer);
  ...
}
void
fill_stdin(char *buffer)
{
  int fd[2];
  pipe(fd);
  dup2(fd[0], STDIN_FILENO);
  close(fd[0]);
  write(fd[1], buffer, strlen(buffer));
  write(fd[1], "\n", 1);
  close(fd[1]);
  return;
}
```

We can note that, moreover, it is possible to send the number of the line in the file, which allow a script to print some information during the computations.

C - Computing some properties

C1 - Program to compute the properties

The formatted files are named, using a unique syntax. First we have the name of the corpus, then the language, and at least the type of the data. Reading this last information allow us to know which kind of properties have to be computed for each file.

The program `compute` is able to compute some properties about the data. It uses the extension of the files to choose the different operations to do. The results we want are for example the number of lines, of different lines, the number of words or of nodes for each line. The syntax of `compute` is the following:

```
Usage: compute [-h] [-v] [-q] FILENAME ...

  -q --quiet    print only data
  -h --help     display this help
  -v --version  output version information
```

C2 - Properties of the data

a - Table of the properties

All the properties, obtained with the program `compute`, are in the following table:

name of file	lines			words			words/lines	
name.lang.type	total	≠	%	total	≠	%	total	≠
baa.eng.txt	23 055	18 796	81,5	281 772	26 671	9,4	12,2 $\pm 12,2$	1,42
baa.eng.cat	23 055	15 632	67,8	281 772	1 356	0,4	12,2 $\pm 12,2$	0,09
baa.eng.tree	23 055	15 755	68,3	801 535	2 094	0,2	34,8 $\pm 33,6$	0,13
dialogue.eng.txt	187	126	67,3	1 606	326	20,2	8,6 $\pm 5,4$	2,59
dialogue.eng.cat	187	119	63,6	1 213	15	1,2	6,5 $\pm 4,4$	0,13
dialogue.eng.tree	187	121	64,7	2 280	22	0,9	12,2 $\pm 7,9$	0,18
nec.jap.txt	6 553	6 296	96,0	41 838	7 976	19,0	6,4 $\pm 3,0$	1,27
nec.jap.tree	6 553	4 571	69,7	71 878	224	0,3	11,0 $\pm 5,9$	0,05
tdmt.jap.txt	724	723	99,8	4 234	1 563	36,9	5,8 $\pm 3,6$	2,16
tdmt.jap.tree	724	712	98,3	5 233	1 018	19,4	7,2 $\pm 4,3$	1,43
tdmtpat.jap.txt	724	723	99,8	4 234	1 563	36,9	5,8 $\pm 3,6$	2,16
tdmtpat.jap.tree	724	722	99,7	5 233	1 346	25,7	7,2 $\pm 4,3$	1,86
upenn.eng.txt	787	776	98,6	9 466	778	8,2	12,0 $\pm 5,3$	1,00
upenn.eng.cat	787	576	73,1	9 457	18	0,1	12,0 $\pm 5,3$	0,03
upenn.eng.tree	787	647	82,2	17 291	38	0,2	22,0 $\pm 9,6$	0,06

b - Calculation and analysis of the properties

Results about the lines

First colon

- The first colon contains the number of lines of the file. Each line of the file contains one and only one entry, so this number is the number of elements in our file.
- The values are calculated by using `wc -l`.

Second colon

- The second colon is obtained by sorting the lines, and removing all duplicated lines.
- Those operations can easily be done with the unix programs like `sort`, `uniq` and `wc`.

Third colon

The last colon is the division of the first two values. This is the rate of non-duplication, or hapax rate. This value is useful to characterise the different files.

Results about the words

First colon

- The first colon is the number of words, of syntactic categories, or of nodes, respectively for a text, a syntax description, or a tree bank of analysis. These values are used to know the length of the elements in a file.
- The values are obtained with the `wc -l` program. Values for trees are obtained by first replacing all parenthesis and commas with empty spaces, allowing us to use then the `wc` command.

Second colon

- The second colon contains the number of distinct words, syntactic categories or nodes. Those values tell us about the vocabulary used in each file. For a tree bank or a syntactic categories text, the values give us some information about the precision used to classify the original words in the text.
- The values are computed in the same way than the first colon, but then they are sorted and the duplicated words, syntactic categories or nodes are removed.

Last colon

The last colon is the division of the two first values.

Division between lines values and words values

First colon

- The first colon is the mean and the square mean of the number of words, syntactic categories or nodes in our file. The mean tells us about the length of each element, giving us some information of the complexity of the syntax used in the file.
- The values are computed with the values obtained about the lines and the words in the two first colons.

Second colon

The second colon is the division of the number of different words, syntactic categories or nodes, by the number of different lines. A value tells us about the information really contained in one line of a file. It is very useful to compare the diversity of the vocabulary used: a big value means that each element adds an important amount of new vocabulary.

c - Analyse of the results

Duplication of elements, of words

For the text files, we can separate our files into two parts, first `baa.eng` and `dialogue.eng`, second `nec.jap`, `tdmt.jap` and `tdmt.jap`. In the first group, we have some dialogues, or some extracts from reports, containing a lot of duplicated elements, like the name of the towns, or salutations. The second group contains some extracts of very used elements, still cleaned, which explains the low value of repetitions.

About the syntactic categories and the trees, `nec.jap` and `upeen.eng` contain a very high rate of repetition compared to the values for the text files. The reason comes from the poor vocabulary used to analyse the sentences. For example, `upeen.eng` uses only 18 categories, and `baa.eng` 1356 !

Moreover, we can notice that such corpora like `tdmt.jap`, `tdmtpat.jap` and `nec.jap` seem to have a very rich vocabulary compared to the other files. In fact, Japanese writing doesn't separate words, so we have to use a tool to do it. Alas, the program used gives very poor results, so the values can't be compared to other languages.

Vocabulary and length of the elements

The rate of words gives us a very useful information about the quality of a corpus. Except for `baa.eng`, we can see that all the files contain some elements having some very close length. The difference for `baa.eng` is coming from the quantity of dates, flight numbers in all the file. All those are considered as new words, even if it doesn't make sense. Such a property modifies the results about the number of different words used.

We can also show two points about `dialogue.eng` and `uppen.eng`. First, `dialogue.eng` seems to use a very rich vocabulary. In fact, the reason comes from the very small size of the corpus. Second, on the opposite, `upeen.eng` seems to use a very poor vocabulary if we compare it to `baa.eng` for example. The reason comes from the way used to realised the corpus: some people have been asked to talk to a machine, and in a way to be easily understood they used a limited vocabulary !

d - Conclusion about the choice of a corpus

Reading the previews results, we decide to use `upenn.eng`. More preciously, our reasons are the followings:

- As we will see later, the complexity of our experiments to be done on a corpus is very high. We do want some first and fast preliminary results, so we want a short corpus.
- With a too short corpus, we could not be able to have enough results.
- Our preliminary experiments deal with characters, computing directly with string of characters. In a way to reduce the number of different words, categories or nodes, we want to use a poor vocabulary.

We can also add that `upeen.eng` has been used before to get some experiments about analysis before, so this choice will allow us to compare the results obtained with analogy.

Chapter 3

Experimentation methods

First, we have realized some tools to format and unify the data coming from different origins. With all these treated data, we want to make some experiments with analogy. These experiments will answer to questions that we can have about analogy, in a way to know the possibilities that are allowed.

We will begin with a description of the experiments we want to realize, which is the first step of our study. Then, we will see how we can unify all the experiments into a unique engine, that will be optimized. To finish, we will describe all the experiments as some simple operations with the result of the common engine.

A - Presentation of the experiments

A1 - The maps: a way to see numerical results

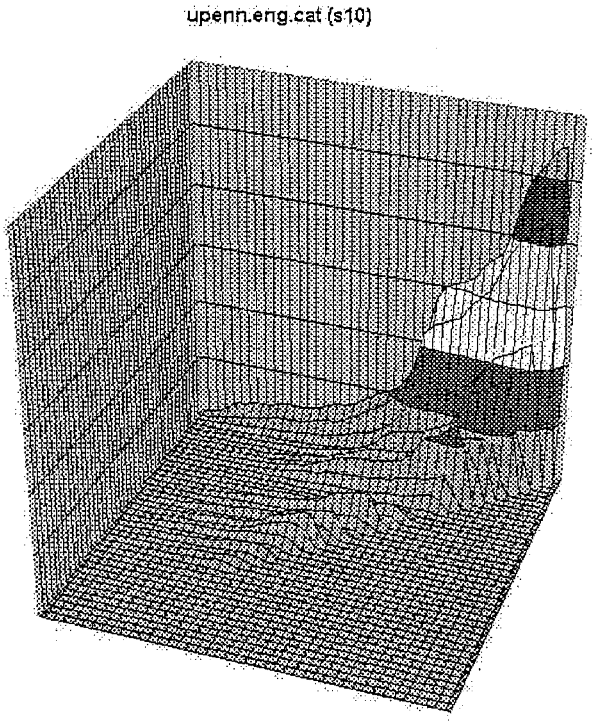
a - Definition of two maps in a 3D model

Let's consider a three dimensions model:

- The first dimension p is the position in the corpus.
- The second dimension is the number n of lines, from the current position p to the beginning of the corpus.
- The third dimension is the number of analogies.

We have two possibilities, defining two different maps:

- map of verified analogies,
- map of created analogies.



b - Frontier of the map of verified analogies: number of past analogies

The first question we can have is about the evolution of the number of verified analogies with the number of data read. This information is shown by the bisecting plan $n = p$. In our graph, this is the limit of the map: we can not have a number of lines that is greater than the position in the corpus.

c - View of the verified analogies: immediate memory

Horizontal section

When we cut the map, we can have some information about the number of elements needed to verify a given number of analogies: this allows us to know the minimal number of elements, for each position in the corpus, that is used for this number of analogies. These values are the immediate memory, and are shown by a section of the map for a constant number of analogies.

Vertical section

A second section of the same map, but for a constant position in the corpus, shows us the evolution of the number of analogies as a function of the number of elements used.

d - Meaning of the map by all possible 4-tuples: density

We can modify our graph, in a way to show a rate between the number of analogies and the number of all possible 4-tuples. This new graph is a representation of the density, and we can use the previously detailed sections. We can notice that this modification can be seen as a meaning of the dimension n, and allows us to measure the local homogeneity of a text.

e - Map of the created analogies: pregnancy

We can now consider the map of the created analogies. The frontier $p = n$ described previously shows the pregnancy, which is a representation of the possibility of a corpus to generate some new elements.

f - Some common points from all experiments

As we have seen, we can realize a unique program which could compute the two maps, using two parameters to let us decide:

- if we want to normalize the values with all the possible 4-tuples,
- if we want to normalize with the number of elements,

and let us decide to consider either:

- the verified analogies,
- the created analogies.

Moreover, this method can allow us to imagine some new possible computations.

A2 - Open-set and closure: how to get some literal results

During the description of the numerical experiments, we have seen that we can work with verified analogies and created analogies. The two different types are linked to a set notion of open-set computation for verified analogies, and closure computation for the created analogies.

a - Generating the open-set of a corpus

The first experiment is the generation of the open-set of our corpora. We want a generating free base of a corpus, and the comparison with the original corpus will give us some information about the representation of each element of this corpus. This experiment is done by using the numerical result from the verified analogies.

b - Generating the closure of a corpus

The second experiment is the reversed computation which consists in isolating the new analogies created. This generation allows us to get the closure of our corpus, and informs us about the property of generation for a given corpus. This computation gives us some literal results where the pregnancy was just giving us some numerical results about the same point.

A3 - Table of the experiments

	Verified analogies		Created analogies
	normalized	not normalized	
Numerical results	density	immediate memory for a give number of analogies	pregnancy
		immediate memory for a given number of elements	
		number of past analogies	
Literal results	closure		open-set

B - Defining a common engine for all the experiments

B1 - Reasons for such an engine

a - Some very high consuming experiments

Computing analogy is a very time consuming operation with three terms for verification, and four terms when solving. We can imagine that the complexity will be very important, which will be shown in the next chapter.

Our goal is to use a method by examples to analyze sentences. This means that we will certainly have to cope with very big corpora, containing a very large number of elements.

Because of the very large number of data we have to deal with, and the important complexity, we can conclude that programming each experiment as an independent one would cost too much time of computation.

b - Some very closed experiments

Moreover, we can see that all experiments needs to verify or to solve analogies. As we will see later, we can describe verification as a computation of a solution and the search of this solution. According to this, we can say that all experiments contain a solving computation part.

B2 - A new view of our experiments

a - A centralized method

Showing the solving part of all experiments allows us to separate our experiments into two steps:

- first step is the computation of the solutions of analogy;
- second step is a computation using the results of the first one.

Separating these two steps gives us a very important property: we can unify the first common step into a unique common program that will be used by all experiments, and named the scheduler.

b - Moving the complexity

The first step is the most time consuming part of the experiments. Any experiment can now be shown as a few computations to use the result of the common engine. The next idea is to continue to move all the possible complexity of any experiment into this unique engine.

This method has the following advantages:

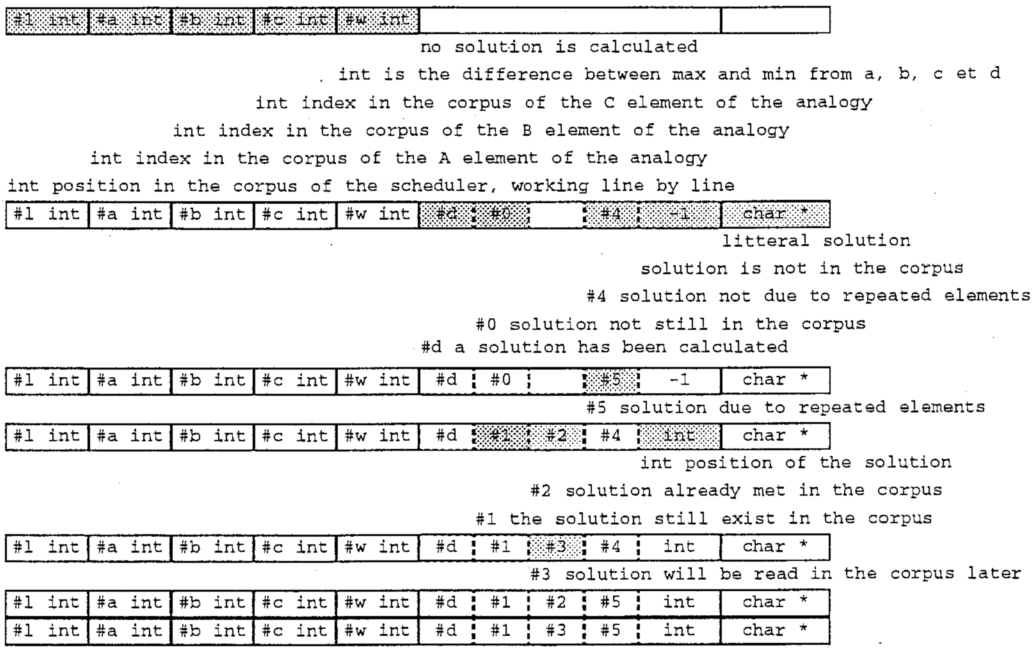
- the engine done, all experiments can be done very quickly;
- it becomes very easy and fast to realize a new experiment;
- we can optimize the common engine without changing our experiments.

B3 - Output of the common engine: scheduling file

The scheduling file contains all the computed information done by the scheduler. In a matter of using the results quickly with the existing shell tools, the scheduling file format is the following:

- all information are in ASCII, with lines and colons, allowing us to use tools like `cut`, `tail` and `head`;
- in each line, any information about the elements begins with a special symbol: #;
- in each line, any result begins with the same symbol, and when it is possible a mark must have its opposite mark, allowing us to filter for a property or the opposite property with `grep`.

Finally, the output of the scheduler is given in the following picture:



```
|#1 int|#a int|#b int|#c int|#w int|                         |        |
                                    no solution is calculated
                              . int is the difference between max and min from a, b, c et d
                          int index in the corpus of the C element of the analogy
                   int index in the corpus of the B element of the analogy
         int index in the corpus of the A element of the analogy
int position in the corpus of the scheduler, working line by line
|#1 int|#a int|#b int|#c int|#w int|#d ¦ #0 ¦     ¦ #4 ¦  -1  | char * |
                                                              litteral solution
                                                        solution is not in the corpus
                                                    #4 solution not due to repeated elements
                                             #0 solution not still in the corpus
                                         #d a solution has been calculated
|#1 int|#a int|#b int|#c int|#w int| #d ¦ #0 ¦     ¦ #5 ¦  -1  | char * |
                                                    #5 solution due to repeated elements
|#1 int|#a int|#b int|#c int|#w int| #d ¦ #1 ¦ #2 ¦ #4 ¦ int  | char * |
                                                        int position of the solution
                                                #2 solution already met in the corpus
                                             #1 the solution still exist in the corpus
|#1 int|#a int|#b int|#c int|#w int| #d ¦ #1 ¦ #3 ¦ #4 ¦ int  | char * |
                                                #3 solution will be read in the corpus later
|#1 int|#a int|#b int|#c int|#w int| #d ¦ #1 ¦ #2 ¦ #5 ¦ int  | char * |
|#1 int|#a int|#b int|#c int|#w int| #d ¦ #1 ¦ #3 ¦ #5 ¦ int  | char * |
```

C - Detailed description of all experiments

C1 - Computing and exploiting the numerical maps

a - Arguments used and returned by the map computation

Viewing all the results as a numerical map allows us to see the evolution of the verified and the computed analogies in a corpus.

- This computation is based on the output of the scheduler and exploits its numerical results. Lines of the scheduling file kept for the first map are the ones where an element in the used part of the file is generated by three other elements of the file. Lines kept for the second map are the ones where an element is generated.
- Result of the computation of the map is a file that can be used with a program like GNUplot.

b - Algorithm to compute a map

The computation falls into two parts:

- The first step transforms the file into a binary file counting the data for the two following parameters:
  - the position in the corpus;
  - the number of elements used from this position.

  This step allows us to used a cumulated sum also, to be able to have global results instead of local results.
- The second step extracts the computed values. This step allows the user to choose to compute a local mean of the data to remove the local variations. This mean can be computed for the position in the corpus, and for the number of elements used. The user can realize a down sampling of the results, and gives the step for this down sampling. In this case, the value computed is the mean of the neighborhood. We can remark that the two means have a different meaning: to give an image, we could imagine the first one as a low frequencies filter, and the second one as a Nyquist filter followed by a down sampling algorithm.

c - Normalizing the maps

The map can be normalized to allow the user to see the results as a function of the memory used. The map can be a direct view of the scheduling computation, or a cumulated view. We have two normalization methods for the two possibilities. In fact, the real difference is for the not cumulated mode, where one of the four terms is fixed.

We can note that the algorithm uses the value 3 in the normalization part, multiplied with the number of possible parts in the given section of the corpus. We have to give some details about this value.

Map of the created analogies

Verified analogies are done using three terms to generate a fourth term. We have 6 possible triples, but the program only uses 3 triples, because of the equivalent analogies as we will see in the next chapter. According to this, the value 3 represents the maximal number of analogies that can be computed.

Map of the verified analogies

Verified analogies are done using four terms, and one of the four terms is fixed according to the mode used: cumulated or not. In a set of four elements, we have 24 possible 4-tuples. Because of the property of exchange of the means, only 12 4-typles are computed. The 12 4-tuples fall into 3 groups of 4 equivalent analogies, and in each group one analogy contains a term D that is greater than the three others. This analogy doesn't have to be considered in our case. According to this, we have 3 groups of 3 equivalent analogies, and we should consider 9 possibilities instead of 3. But because the 3 groups can not be observed at a same time, we can only find 3 analogies. This justifies in this second case the value 3.

C2 - Open-set: computing a base

a - Arguments used and returned by the computation

The computation of the open-set gives us some information about the representation of each element in the corpus.

- This computation, again, is based on the result of the scheduler. The studied corpus must not have repeated elements, and lines from the scheduling file where an element is generated by three other elements are kept.
- The result of the open-set computation is a text file containing the number of the lines we must keep for the base.

b - Two equivalent approaches

A direct method would be a constructive method. Our method uses some computed results from the scheduler, so this is a destructive method. This method constructs the complement of the open-set of the corpus.

c - Algorithm of the open-set computation

Used tags for this experiment are #a, #b, #c and #d. The #d tag is filtered with the #1 tag, to consider only verified analogies, and #5 to be sure that the element is unique in the corpus. Repeating some elements does not seem to modify the results, but the user should avoid such a situation.

```
while(some more lines in the file)
 {
  for(begin of file; more lines to read; next line)
   {
    d =generated element;
    increase the number of time d is generated;
    if(generated for the first time)
      memorize the generating triple;
   }
  d =position of the latest generated element;
  a =position of the term A of the memorized triple generating d;
  b =...;
  c =...;
  mark d as not being in the open-set;
  remove the lines generating d;
  remove the lines where d appears as a term of an analogy;
  remove the lines generating a, b or c;
 }
```

This algorithm marks the elements that are not in the open-set. To show this open-set, we just have to output the elements that are not marked. By the way, in our case, it is more interesting to now the triples from the open-set generated by the elements and that are not in this open-set:

```
for(d = 1; d≤n; d++)
 if(d not in the open-set)
  print the 4-tuple;
```

We can notice that we can destroy and compute the number of generating triples at the same time at the step $k+1$ for the computation and k for the destruction. To make the computation going faster, the scheduling file is first converted into an appropriated binary format, allowing us to read it into a constant size buffer.

C3 - Closure of a corpus

a - Arguments used and returned by the closure computation

Generating the closure gives us some information about the global property of a corpus to create new elements.

- This computation is directly based on the scheduling file. Lines that are kept are the ones where a new element is created with three elements from the corpus.
- Result of this computation is a file containing the new elements created and the elements of the corpus.

b - Algorithm of the closure computation

The tag needed is only the seventh one. The very simple computation falls into three parts:

- we only keep the seventh colon of the lines containing an analogy,
- we add this to the original corpus,
- we filter the obtained file to remove all repeated elements.

Chapter 4

Experiments' engine: the scheduler

A - A didactic overlook of the algorithm

In the following, we will call element an object used as one of the four terms of analogy. We will suppose that the element are the lines of a file, and the number of an element is the number of the line in the file of this element. A term is one of the four elements used in an analogy. Terms will be represented by upper letters, and their position by a lower letter corresponding.

A1 - A naive algorithm

We write $A : B = C : D$ a generic analogy. In the naive algorithm, we take all 4-tuples possible and we verify the analogy. The complexity of such an algorithm is n^4 verifications:

```
for($a = 1$; $a \leq n$; $a++$)
  for($b = 1$; $b \leq n$; $b++$)
    for($c = 1$; $c \leq n$; $c++$)
      for($d = 1$; $d \leq n$; $d++$)
        if(analogy verified for the 4-tuple)
          show the 4-tuple;
```

We have three properties of analogy, and each property allows, with one analogy, to get another equivalent analogy. For one verified analogy $A : B = C : D$ the three properties allow us to get seven other properties. According to this, an ideal algorithm of verification must have a complexity of $n^4/8$ verifications.

$A : C = B : D$	exchange of the means
$D : B = C : A$	exchange of the extremes
$D : C = B : A$	exchange of the means and extremes
$C : D = A : B$	symmetry of the equality
$C : A = D : B$	symmetry of the equality and exchange of the means
$B : D = A : C$	symmetry of the equality and exchange of the extremes
$B : A = D : C$	symmetry of the equality and excahnge of the means and the extremes

A2 - Verify = solve + search

When three terms are chosen, instead of verifying analogy for all elements from 1 to n, we can solve the analogy for the three terms, and then search the solution in the file. As analogy gives only one solution, and as the complexity for a search in a file of n elements can be realised with a complexity of $\log_2(n)$, this new algorithm allows us to change our n^4 algorithm for a $n^3.\log_2(n)$ algorithm in the worst case, when a solution does exist, or n^3 when the analogy doesn't have a solution.

```
for ($a = 1; a \leq n; a++$)
  for ($b = 1; b \leq n; b++$)
    for ($c = 1; c \leq n; c++$)
      {
        $D$ =result of the analogy from three terms;
        if($D$ exists and is in the file)
          show the 4-tuple;
      }
```

In an ideal case, if we consider all equivalent analogies, we can wait for a $n^3 \log_2(n)/8$. In the following, we will use this property only at the last step, in a way to be able to use as best as possible some other properties that could increase the speed of our algorithm.

A3 - Getting an incremental algorithm

We want to be able to follow the whole evolution of the computations element by element. To do so, we have to change our global algorithm for an incremental algorithm. The modification consists in treating a loop of k, with the following two properties for all 4-tuples:

- numbers of the four terms must be lesser or equal to k;
- one of the four terms' positions at least must be k.

According to this, we have the complexity of the new algorithm:

$$\sum_{k=1}^{n} \left(\begin{array}{l} \sum_{a=1}^{k}\sum_{b=1}^{k}\sum_{c=1}^{k} 1_{d=k} + \sum_{a=1}^{k}\sum_{b=1}^{k}\sum_{d=1}^{k} 1_{c=k} \\ + \sum_{a=1}^{k}\sum_{c=1}^{k}\sum_{d=1}^{k} 1_{b=k} + \sum_{b=1}^{k}\sum_{c=1}^{k}\sum_{d=1}^{k} 1_{a=k} \end{array} \right) = \sum_{k=1}^{n} 4.k^3 = n^4 + o(n^4)$$

A4 - Using the properties of analogy

a - A partial use of the properties

We want the two conditions $d \leq c \leq b$ and $a = k$. Expect for some situations, we still have a complexity of n^4 and all 4-tuples if we consider the 24 possible combinations:

$1 \leq a \leq n$ $1 \leq d \leq c \leq b \leq a$		
$A : B = C : D$	$B : A = C : D$	$C : A = B : D$
$A : C = B : D$	$B : C = A : D$	$C : B = A : D$
$D : B = C : A$	$D : A = C : B$	$D : A = B : C$
$D : C = B : A$	$D : C = A : B$	$D : B = A : C$
$C : D = A : B$	$C : D = B : A$	$B : D = C : A$
$C : A = D : B$	$C : B = D : A$	$B : C = D : A$
$B : D = A : C$	$A : D = B : C$	$A : D = C : B$
$B : A = D : C$	$A : B = D : C$	$A : C = D : B$

In the table of the 24 possibilities, the fifth line can be obtained with the property of symmetry of analogy. The third and seventh lines can be obtained from the first and the fifth lines with the property of exchange of the extremes. Lines 2, 4, 6 and 8 are obtained from 1, 3, 5 and 7, with the property of exchange of the means. According to this, we can remove 21 analogies that are equivalent to 3 analogies. Complexity of the new algorithm is n^8:

$1 \leq a \leq n$ $1 \leq d \leq c \leq b \leq a$		
$A : B = C : D$	$B : A = C : D$	$C : A = B : D$

Now, we can use the solve and search method. We can overestimate the complexity of the new algorithm, the proof is done as an appendix:

$$\sum_{a=1}^{n}\sum_{b=1}^{a-1}\sum_{c=1}^{b-1} E\left(\log_2(c)+\frac{1}{2}\right) \leq \frac{(n+1)^3}{6}\left(\log_2(n+1)+1-\frac{11}{6.\ln(2)}\right) + \underbrace{\cdots}_{O(n^2)}$$

$1 \leq a \leq n$ $1 \leq c \leq b \leq a$ $x \in [1;c]$		
$A : B = C : X$	$B : A = C : X$	$C : A = B : X$

We have three trivial analogies:

- $A : A = C : C$ from $A = B \Leftrightarrow C = D$;
- $A : B = A : B$ from $A = C \Leftrightarrow B = D$.

In our case, the trivial analogies are not interesting and are forgotten. This is done by changing the beginning of the A loop from 1 to 2. Moreover, we have the following equations that allow us to simplify our algorithm:

- $\{c \leq b \leq a, a \neq b\} \Rightarrow c \leq b < a \Rightarrow a \neq c$
- $\{c \leq b \leq a, c \neq b\} \Rightarrow c < b \leq a \Rightarrow c \neq a$

$2 \leq a \leq n$ $1 \leq c \leq b \leq a$ $x \in [1; c]$		
$a \neq b$	$b \neq a$ $b \neq c$	$c \neq b$
$A : B = C : X$	$B : A = C : X$	$C : A = B : X$

Finally, the new algorithm is:

```
for($a = 2$; $a \leq n$; $a++$)
  for($b = 1$; $b \leq a$; $b++$)
    for($c = 1$; $c \leq b$; $c++$)
     {
      if($a \neq b$ && $b \neq c$)
        {
         $D$ =result of analogy between $B$, $A$ and $C$;
         if($D$ exists and is in the $c$ first elements of the file)
           print the 4-tuple;
        }
      if($a \neq b$)
        {
         $D$ =result of analogy between $A$, $B$ and $C$;
         if($D$ exists and is in the $c$ first elements of the file)
           print the 4-tuple;
        }
      if($b \neq c$)
        {
         $D$ =result of analogy between $C$, $A$ and $B$;
         if($D$ exists and is in the $c$ first elements of the file)
           print the 4-tuple;
        }
     }
```

b - Full use of the properties: sparse matrices

We have seen that we could wait for a $n^3 \log_2(n)/8$ algorithm, and for the moment we only have a $n^3 \log_2(n)/6$ algorithm. The reason is coming from the fact that we are not searching in the whole file, but in the n first elements. When a solution does exist, but in a position greeter than n, we can not use it, and this value will be computed again later.

- $C : D = A : B$ done again for (D, A, C)
- $B : D = A : C$ done again for (D, A, B)
- $D : B = C : A$ done again for (D, B, C)

When such a case appears, we have three analogies that we should not compute again. The use of this property for a position d needs two steps, first we must save the 3-tuple to use it later at the position d, then we must recover the three analogies in a matrix. Position in the matrix is (b, c) and the value stored a. As a real matrix would have been two big, we had to realise a sparse matrices algorithm to implement this improvement.

For more information about the sparse matrices algorithm, the reader could check in the appendix at the end of this report.

B - Example of results

We have the file containing the three following lines:

- Element 1 : c
- Element 2 : ab
- Element 3 : a

Result of the algorithm is given in the table:

current	positionA	positionB	positionC	Analogy
2	2	1	1	
	1	2	2	
3	3	1	1	
	2	3	1	
	3	2	1	cb
	1	3	2	
	3	2	2	abb
	1	3	3	
	2	3	3	

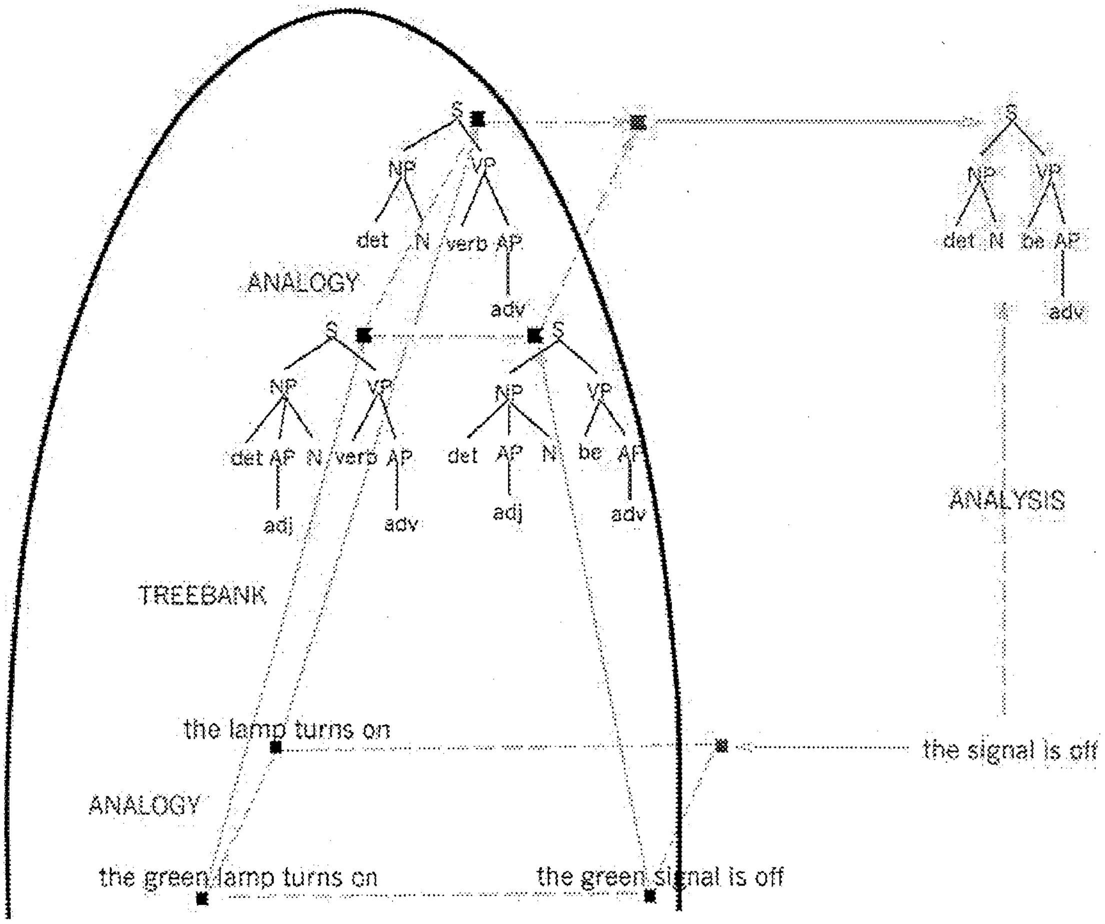
Chapter 5

An experiment: analysis by analogy

A - Presentation of the experiment

A1 - Description of the experiment

The experiment we want to process is shown in the following picture:



With some given analysis, we want to try to analyse automatically some new sentences: our entries, using the properties of analogy. The steps for this experiment are:

- a triple is searched in our corpus, the three terms and our entry verifying the analogy;
- when such a triple is available, we extract the three analysis;
- if this three analysis can produce a fourth term by analogy, we associate this new analyse to the entry.

This method is an ideal one. Our goal is to verify if this idea can really be used, and we want to elude the following points:

- for one entry, we can have several triples and several analysis, so we want to have some information about the quantity of results;
- analogy is very time-consuming, and we want to know the minimal useful size of a corpus to have a good analyse;
- of course we want to compare the results with the real analysis.

What we have to do is to compare the distance between the results and the real analysis for a set of sentences. The distance between the trees we use is a generalised distance of edition for the strings, that can be applied to trees.

In the following experiment, we simulate a real situation: the analyse of sentences to create a corpus. For memory, a tree bank is a set of sentences associated to the trees. To create such a bank, the sentences are analysed one by one, and added to the tree bank.

According to this, when the tree bank has n data, the entry must be analysed by analogy using the n available elements, elements that have been verified and corrected before being added.

A2 - Description of the data used

Our goal is to justify the use of this method, with a short corpus, in a way to have some fast preliminary results. Because of the short time we have, the requested properties for the corpus are:

- because of the $n^3.\log_2(n)$ complexity, the corpus has to be short;
- to have non trivial results, the structure of the sentences must be long enough;
- to increase the number of analogies, we use the syntactic categories sentences, with a small vocabulary.

According to these remarks, the corpus we will use is `upenn.eng`. Because of the rich vocabulary used by `baa.eng`, results would have been better, but this experiment will take to much computing time, and can not be done for a preliminary study. Our experiment is done here with some English sentences, where words are replaced by the syntactic categories.

B - Results

As we want to show a real use of analysis by analogy, we simulate a man analysing the sentences one by one, and making a preliminary analyse by analogy with the previous realised analysis. At the line n, the entry, of our corpus:

- we search all the triples in the $n-1$ first elements that can create our entry by analogy;
- we compute analysis by analogy, and we store the obtained results;
- then we can compare the results with the real analyse.

B1 - Number of calculated analysis

For memory, our corpus has 787 sentences. The number of analysis realised as described before is 296, a little bit more than one for three.

Analysis are done with the exact analysis of the previous sentences. We can imagine that the number of analysis must grow with the number of element used. This can be verified in the graph:

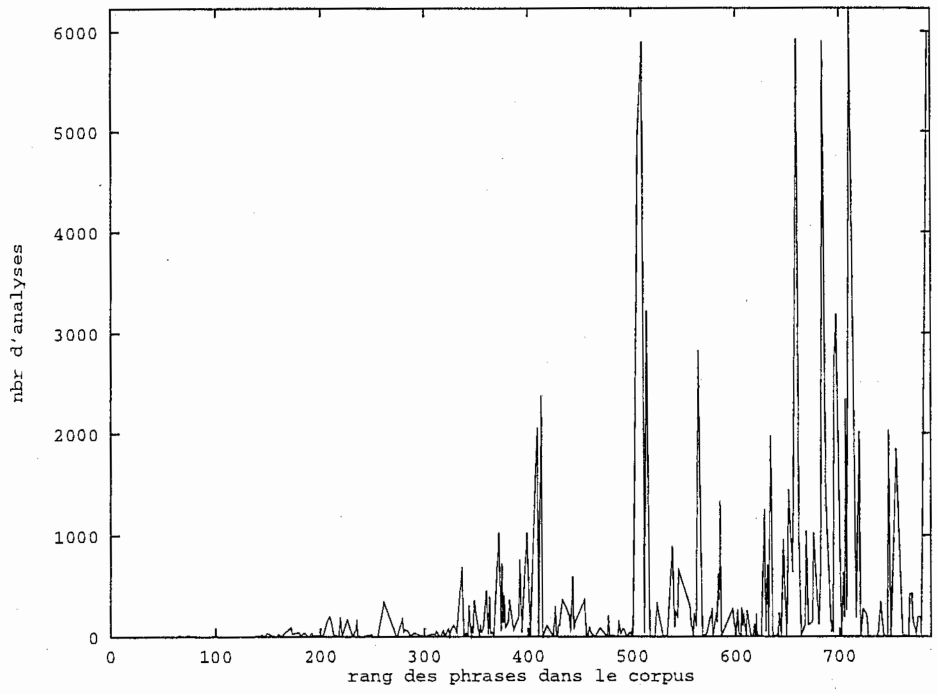


Figure 5.1: Number of analysis as a function of the position in the corpus (smoothed).

The number of analysis that we can get for one sentence can be very important, the mean is 374 analysis, with an atypical maximum of 6 231. The density of this number is shown in the graph 5.2.

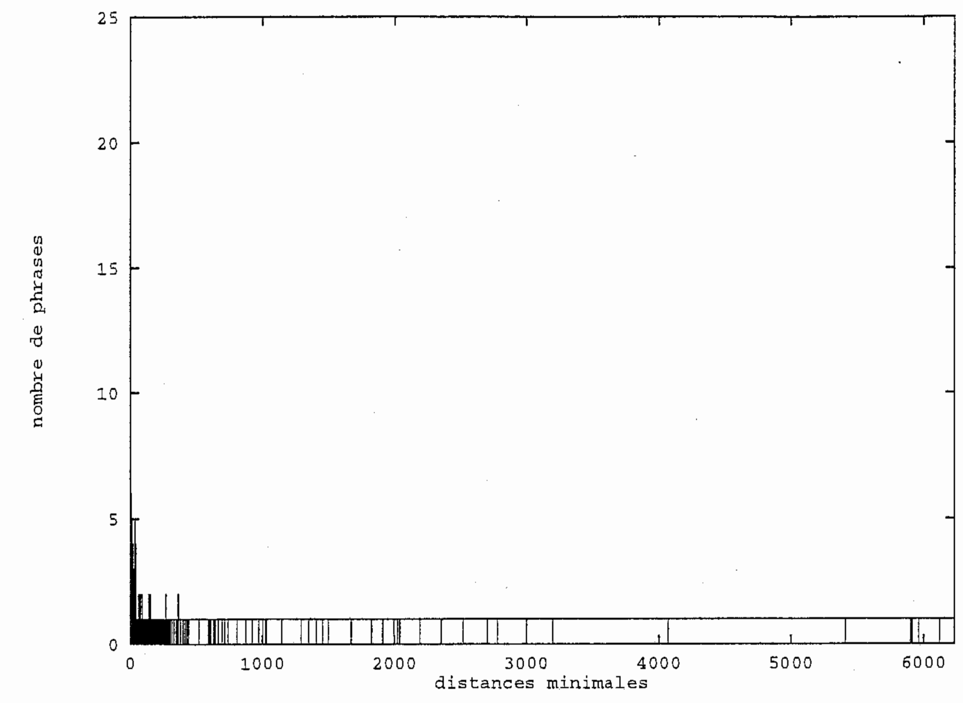


Figure 5.2: Density of analysis.

B2 - Quality of the analysis

We can now compare the quality of the analysis with the real analysis. To do so, we consider the closest analyse and we measure the distance between this analyse and the exact one. The results of this measurement is shown in the graphs 5.3 and 5.4.

For all our corpus, we have one exact analyse. 170 analysis from the 296 have a distance lower than 9, and 81 analysis, so more than one for four, have a distance lower than 4 to the exact result. We should remember that the number of node of a sentence is $22 \pm 9,6$ nodes.

C - Conclusion

As we have seen, the number of analysis increases with the size of the corpus, so we can expect to get some better results. We could find that the quality of the results is not so good, but we must remember that our example is a very short corpus, lesser than 800 entries. On the opposite, these results can be considered as very promising, and we should make some further experiments, with a real corpus.

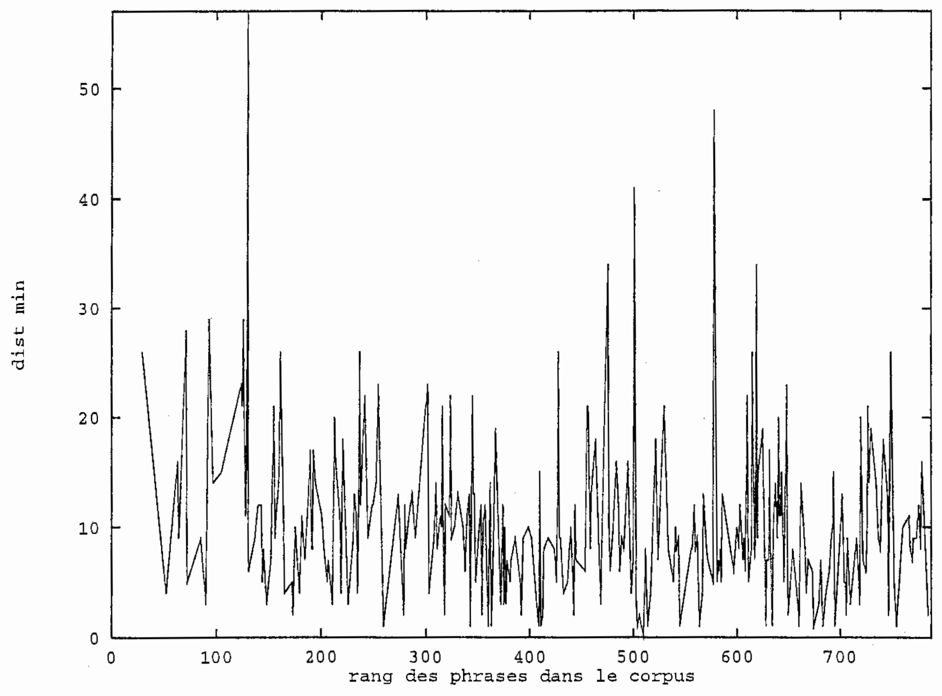


Figure 5.3: Minimal distance.

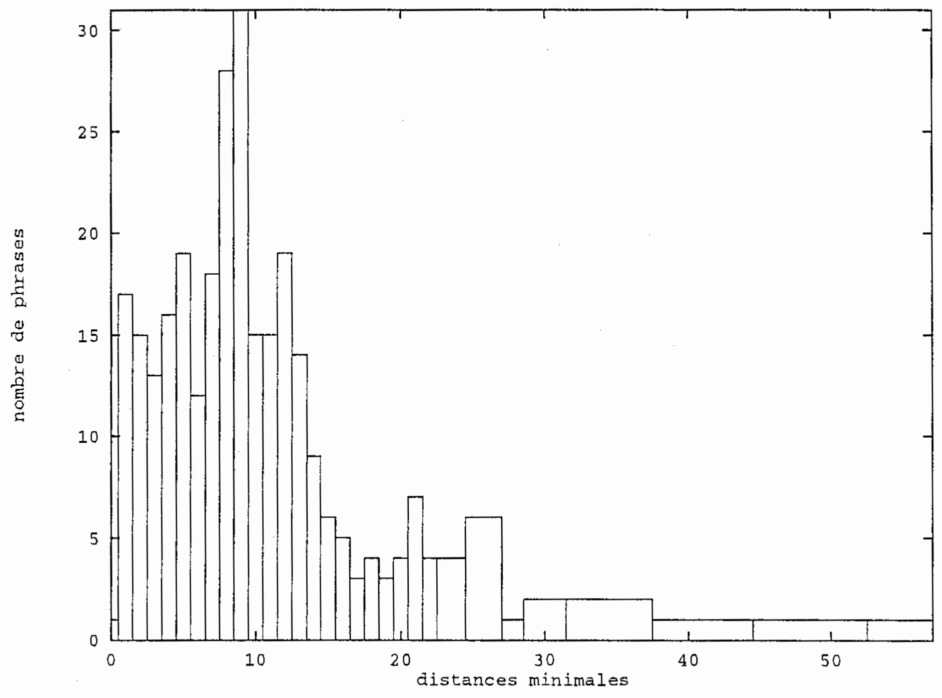


Figure 5.4: Number of sentences for each distance.

Appendix A

This appendix details a overestimated approximation of the complexity of the scheduler. This approximation is done by the calculation of three integrals:

$$\sum_{a=1}^{n}\sum_{b=1}^{a-1}\sum_{c=1}^{b-1} E\left(\log_2(c)+\frac{1}{2}\right) \leq \int_1^{n+1}\int_1^{x}\int_1^{y} \left(\log_2(z)+1\right)\ dz.dy.dx$$

A.1 - First integral

$$\begin{aligned}
\int_1^{n+1}\int_1^{x}\int_1^{y} \left(\log_2(z)+1\right)\ dz.dy.dx &= \frac{1}{\ln(2)}\int_1^{n+1}\int_1^{x}\int_1^{y} \left(\ln(z)+\ln(2)\right)\ dy.dx \\
&= \frac{1}{\ln(2)}\int_1^{n+1}\int_1^{x}\left[\frac{z}{1}\left(\ln(z)-\frac{1}{1}\right)+z.\ln(2)\right]_1^y\ dy.dx \\
&= \frac{1}{\ln(2)}\int_1^{n+1}\int_1^{x}\left[z.\ln(z)+z\left(\ln(2)-1\right)\right]_1^y dy.dx \\
&= \frac{1}{\ln(2)}\int_1^{n+1}\int_1^{x}\left(y.\ln(y)+y\left(\ln(2)-1\right)-\left(\ln(2)-1\right)\right)\ dy.dx
\end{aligned}$$

A.2 - Second integral

$$\begin{aligned}
\int_1^{n+1}\int_1^{x}\int_1^{y} \left(\log_2(z)+1\right)\ dz.dy.dx &= \frac{1}{\ln(2)}\int_1^{n+1}\left[\frac{y^2}{2}\left(\ln(y)-\frac{1}{2}\right)+\frac{y^2}{2}\left(\ln(2)-1\right)-y\left(\ln(2)-1\right)\right]_1^x dx \\
&= \frac{1}{\ln(2)}\int_1^{n+1}\left[\frac{y^2}{2}\ln(y)+\frac{y^2}{2}\left(\ln(2)-\frac{3}{2}\right)-y\left(\ln(2)-1\right)\right]_1^x dx \\
&= \frac{1}{\ln(2)}\int_1^{n+1}\left(\begin{array}{c}\frac{x^2}{2}\ln(x)+\frac{x^2}{2}\left(\ln(2)-\frac{3}{2}\right)-x\left(\ln(2)-1\right)\\ +\left(\frac{\ln(2)}{2}-\frac{1}{4}\right)\end{array}\right) dx
\end{aligned}$$

A.3 - Third integral

$$\begin{aligned}
\int_1^{n+1}\int_1^x\int_1^y (\log_2(z)+1)\ dz.dy.dx &= \frac{1}{\ln(2)}\left[\begin{array}{c} \frac{1}{2}\frac{x^3}{3}\left(\ln(x)-\frac{1}{3}\right)+\frac{1}{2}\frac{x^3}{3}\left(\ln(2)-\frac{3}{2}\right) \\ -\frac{x^2}{2}(\ln(2)-1)+x\left(\frac{\ln(2)}{2}-\frac{1}{4}\right)\end{array}\right]_1^{n+1} \\
&= \frac{1}{\ln(2)}\left[\begin{array}{c} \frac{x^3}{6}\ln(x)+\frac{x^3}{6}\left(\ln(2)-\frac{11}{6}\right) \\ -\frac{x^2}{2}(\ln(2)-1)+x\left(\frac{\ln(2)}{2}-\frac{1}{4}\right)\end{array}\right]_1^{n+1} \\
&= \frac{1}{\ln(2)}\left(\begin{array}{c} \frac{(n+1)^3}{6}\ln(n+1)+\frac{(n+1)^3}{6}\left(\ln(2)-\frac{11}{6}\right) \\ -\frac{(n+1)^2}{2}(\ln(2)-1)+(n+1)\left(\frac{\ln(2)}{2}-\frac{1}{4}\right) \\ -\frac{1}{6}\left(\ln(2)-\frac{11}{6}\right) \\ +\frac{1}{2}(\ln(2)-1)-\left(\frac{\ln(2)}{2}-\frac{1}{4}\right)\end{array}\right)
\end{aligned}$$

A.4 - Simplifying the expression

$$\int_1^{n+1}\int_1^x\int_1^y (\log_2(z)+1)\ dz.dy.dx = \frac{1}{\ln(2)}\left(\begin{array}{c} \frac{(n+1)^3}{6}\left(\ln(n+1)+\ln(2)-\frac{11}{6}\right) \\ -\frac{(n+1)^2}{2}(\ln(2)-1)+(n+1)\left(\frac{\ln(2)}{2}-\frac{1}{4}\right) \\ -\frac{1}{6}\left(\ln(2)-\frac{11}{6}\right) \\ +\frac{1}{2}(\ln(2)-1)-\left(\frac{\ln(2)}{2}-\frac{1}{4}\right)\end{array}\right)$$

A.5 - Main term of the expression

$$\int_1^{n+1}\int_1^x\int_1^y (\log_2(z)+1)\ dz.dy.dx = \frac{(n+1)^3}{6.\ln(2)}\left(\ln(n+1)+\ln(2)-\frac{11}{6}\right)+\ldots$$

A.6 - Asymptotic limit of the expression

$$\int_1^{n+1}\int_1^x\int_1^y (\log_2(z)+1)\ dz.dy.dx \underset{+\infty}{\sim} \frac{n^3\log_2(n)}{6}+\circ\left(n^3\log_2(n)\right)$$

Appendix B

B.1 - Reasons for using sparse matrices

While we were studying the scheduler, we first realised a non optimum algorithm. Then we remarked that, when verifying an analogy for a fourth element whose position in the file is greater than the current position, this analogy will be calculated again later three other times for three equivalent analogies. Our goal is to store the first analogy to avoid the computation of the three equivalent analogies.

Alas, we have four terms to remember, so we must store one term as a value depending to three parameters, the three other terms. While we are working at a fixed position in the corpus, one of the three parameters can be fixed to this value. According to this, if N is the length of the corpus, we have to push three terms in an array of N lists. Then, when we are at the line n, we pop the values from the list n, and we have to consider one of the three terms as a value depending on the two others.

Again, we have one problem left: this means that we have to use a matrix. As our corpus can be very long, with N greater than 10 000, such a matrix can not be used. By chance, we have very few elements to store, and it makes possible to us to use a sparse matrices method.

B.2 - Description of the algorithm

Our algorithm extends the method used for trees, and instead of having p nodes we now have $p * p$ nodes:

- each node is a p matrix of pointers to $p*p$ nodes, and if one node is not used, the pointer is NULL;
- this structure is used as we could to with trees, except that we must consider two directions at the same time.

The advantage of this method is that, when a part of the matrix does not contain any element, this part does not have to be allocated. All sub-matrices have the same structure, and we can put the non-used sub-matrices in a reservoir. This allow us to allocate a wide number of sub-matrices at a same time to increase the size of the reservoir, and it avoids us loosing time with allocations and deletions.

Two parameters characterise our sparse matrice:

- the size 2^m of each sub-matrix;
- the number l of level of sub-matrices.

Concerning the dimension 2^m, we can remark that the sparse matrix is fully used if we have more than one analogy for 2^{m^2} computations. The quality of the sparse matrix depends directly on this value, so it must be chosen carefully.

As we have seen, from m depends the size of the structure. On the opposite, if m is too little, we have to increase the number of levels l of our sparse matrix to have a $N * N$ global matrix. The exact size of the matrix is the following:

$$\sqrt{\left((2^m)^2\right)^{l+1}} = 2^{m(l+1)}$$

The choice of m and l is done according to:

- the memory that we have;
- the global size that we want.

The following table gives the global size of the sparse matrix, depending on the values m and l:

	levels of sub-matrices l					
bits m	2	3	4	5	6	7
2		256^2	1024^2	4096^2	16384^2	65536^2
3		4096^2	32768^2			
4	4096^2	65536^2				

For a given corpus, we must choose the best solution. Using a too big solution does cost any memory, as the non-used nodes are not allocated, but this will mean that the use of the levels is not optimal. When we have more than one dimension, the choice must be done according to the memory we have and the speed we want: a greater value of l will slow down the sparse matrix, but a greater value of m will decrease the quality of the sparse matrix.

B.3 - Some more remarks

If the memory is fully used, and we can not allocate any new sub-matrix, analogies are not stored and will be computed again later. It means that this is not a problem as it will just slow down our algorithm, and the algorithm will be able again to store the values later when some no more used sub-matrix will be in the reservoir.

The method used avoid us repeating analogies. As we were able to verify that all pushed values have been successfully popped during our experiments, we can be sure that the new algorithm using sparse matrices is optimal.

The cost of this method depends on the searches done in our sparse matrix. In fact, we have a low number of elements stored, and a search is often stopped very quickly. Our experiments showed us that this cost was insignificant compared to the gain, because of the high cost of a successful computation of analogy.

Our algorithm does not estimate the needed dimension of our sparse matrix. It would be a better idea to choose the values l and m according to the size of the corpus. This remark is important, as a too small dimension will cause an error for a big corpus, so the user should verify this dimension before using the scheduler. By the way, the choose of the good parameters should not be a big problem to program.