Technische Universität Berlin
Fakultät IV - Fakultät Elektrotechnik und Informatik
Fachgebiet Datenbanksysteme und Informationsmanagement

Project Report
Generating Acrostics via Paraphrasing and Heuristic Search
DBPRO - Database Projects (WS 2014/2015)

Supervisor:
Johannes Kirschnick

Authors:
Bruno Soares Fillmann ()
Fernando Bombardelli da Silva (bombardelli.f@mailbox.tu-berlin.de)
Jürgen Bauer (jbauer@mailbox.tu-berlin.de)
William Bombardelli da Silva (wbombardellis@mailbox.tu-berlin.de)

February 2^{nd}, 2015

Contents

Abstract

ascasdas

1 Introduction and Motivation

2 Generating Acrostics via Paraphrasing and Heuristic Search

2.1 Problem Definition

2.2 Modeling as Search Problem

In this section we show how to model ACROSTIC GENERATION as a search problem. Let s be a text (for which we intend to generate an acrostic) and let $\mathcal{T}$ be the universe of all paraphrased versions of s. The idea is to represent the elements of $\mathcal{T}$ as states or equivalently as nodes of a tree, where the tree is constructed as follows:

- s represents the root node
- if $n \in \mathcal{T}$ is a node, then its successor nodes are given by all nodes, which are produced by applying an paraphrasing operator ϕ to n.

A node $\gamma \in \mathcal{T}$ that already encodes the acrostic is called a goal node. The set of all goal nodes is denoted by Γ.
Solving an instance of ACROSTIC GENERATION under the so-called state-space representation then means to find a path from s to some goal node $\gamma \in \Gamma$. According to the ACROSTIC GENERATION problem several question might arise, such as:

- how to control the exploration of the search space?
- which solution candidate should be chosen?
- can we avoid that states are revisited?

To overcome these issues our strategy is to employ an A^*-algorithm. This algorithm performs a best-first search leading to a result with the best possible quality.

2.3 Cost Measure

In order to apply the A^*-algorithm it is essential to define cost measures for the paraphrasing operators. These measures enable us to quantify the cost of each path and to define the heuristic. To define the cost measures, we first assign a constant value $q \in [0, 1]$, named local quality, to each paraphrasing operation. The local quality values are on the one hand defined based on our experiences and on the other hand are inspired by the values given in [1]. The local qualities are summarized in the following table:

Operator	Local quality q
LineBreak	0.9
Hyphenation	0.8
WordInsertionDeletion	0.7
Synonym	0.3
WrongHyphenation	0.4
WrongSpelling	0.7

The cost of an operator ϕ is then defined as $1/q$ where q denotes the local quality of ϕ. In the following we will write $q(n_0, n_1)$ for the local quality of the operator, which transforms node n_0 into n_1.

In order to apply the A^*-algorithm we need to define a cost estimation heuristic $f(n) = g(n)+h(n)$. Here, the function g calculates the true cost from the start node s to the node n, while h underestimates the cost of the path from node n to a goal node $\gamma \in \Gamma$. It is known that the A^*-algorithm finds an optimal (least cost) solution. If moreover, the function h is monotonic, no node needs to be processed more than once (cf. [9]). In this case the algorithm can be described as follows:

First, we insert the start node into a priority queue. (The lower $f(n)$ for a given node n, the higher its priority.) As long as the queue is not empty we dequeue the element n with the highest priority. If this element already encodes the acrostic, we are done! Otherwise we put n in the visitedSet and generate its neighbors. For each neighbor of n we check that it was not already visited. When this is true, and the neighbor node is not contained in the priority queue, we add it to the priority queue; if the neighbor node it contained in the priority queue, we update the f(n) value if necessary.

A more precise formulation of the A^*-algorithm is given in the following pseudocode:

Algorithm 1 A^*-algorithm

```
function A*(start, acrostic)
    stateQueue ← {start}        ▷ Priority queue of nodes preferring nodes with lower cost
    visitedSet ← ∅              ▷ Hash Set of nodes already visited
    while stateQueue is not empty do
        current ← stateQueue.pop()        ▷ get first element and remove it
        if current encodes the acrostic then
            return current
        add current to visitedSet
        for all successor in successor_nodes(current) do
            if successor in visitedSet then
                continue

            tentative_g_value ← g(current) + 1/q(current, successor)

            if successor not in stateQueue or tentative_g_value < g(successor) then
                g(successor) ← tentative_g_value
                f(successor) ← g(successor) + h(successor)
                if successor not in stateQueue then
                    add successor to stateQueue
  return failure
```

After we have explained how the A^*-algorithm functions, we proceed with describing the semantics of the cost estimation function $f(n) = g(n) + h(n)$.

Here, the function g accumulates the true cost for the partial acrostic via a concrete path $s = n_0, n_1, \ldots, n_k = n$, while $h(n)$ gives an underestimation of the cost for the remaining part of the acrostic. The function f can be stated as:

$$f(n) = \underbrace{\sum_{i=1}^{k} 1/q(n_{i-1}, n_i)}_{g(n)} + \underbrace{log_K(E(\tau(n))) \cdot \frac{1}{q_{\max}}}_{h(n)},$$

where $\tau(n)$ denotes the remaining acrostic y that is associated with $n \in \mathcal{T}$. (If x' is the part of the acrostic x that has already been constructed, then the remaining acrostic y is given by $x = x'y$.)

The mapping E is a measure for the difficulty or *effort* for generating the acrostic y. It is defined by $E(y) = e(y_1) \cdots e(y_n)$, where $e(u)$, for a letter u, denotes the multiplicative inverse of the occurrence probability for u as a first letter. The first letter probabilities can be found in [8].

The logarithm base K serves for normalization purposes. We define K as the multiplicative inverse of the occurrence probability of the least frequent first letter in the remaining acrostic $y = \tau(n)$. In our example, we have $E(y) = K^{l_1} \cdots K^{l_n}$ with $0 < l_i \leq 1$, this yields $log_K(E(y)) = \sum_{i=1}^{n} l_i \leq n = |y|$, where $|y|$ means the number of letters in y.

Finally, $q_{\max}$ is the maximum of the local operator qualities.

Our choice of $h(n)$ entails two properties:

(1) it underestimates the length of the remaining acrostic and therefore ensures the admissibility characteristic. Here, we assume that the remaining acrostic y could be solved in the best case within a number of n steps. In each step an operator with a cost of at least $1/q_{\max}$ is applied.

(2) it is monotonic, i.e. for each two nodes $n_1, n_2 \in \mathcal{T}$, such that n_2 is a successor of n_1, we have $h(n_1) - h(n_2) \leq 1/q(n_1, n_2)$.

To show the monotonicity of h we consider two cases:
1. If $h(n_1) - h(n_2) = 0$ then the inequality $h(n_1) - h(n_2) \leq 1/q(n_1, n_2)$ is obviously satisfied.
2. If $h(n_1) - h(n_2) > 0$ then one letter more, say u, of the acrostic has been generated. We can therefore write $\tau(n_1) = u\tau(n_2)$. Let K_1 and K_2 be the logarithm bases for computing $h(n_1)$ and $h(n_2)$, respectively. Then we have $K_1 \geq K_2$, which yields the inequality

$$\begin{aligned}
h(n_1) - h(n_2) &= log_{K_1}(\tau(n_1)) - log_{K_2}(\tau(n_2)) \\
&= log_{K_1}(u\tau(n_2)) - log_{K_2}(\tau(n_2)) \\
&\leq log_{K_1}(u\tau(n_2)) - log_{K_1}(\tau(n_2)) \\
&= (log_{K_1}(u) + log_{K_1}(\tau(n_2)) - log_{K_1}(\tau(n_2)) \\
&= log_{K_1}(u) \\
&\leq 1 \leq 1/q(n_1, n_2).
\end{aligned}$$

Hence we have verified that h is indeed monotonic.

Remark: From the implementational point of view it might be possible that one operator generates more than one letter of the acrostic at a time. The reason for this is that an adjustment of the whole text is conducted after each operation, which could lead to formal difficulties with both admissibility as well as monotonicity. As an easy way out we suggest to modify the operator cost by adding it one up for each additional letter the operator generates.

The above two properties guarantee that the A^*-algorithm finds an optimal (least cost) solution without processing any node twice (cf.[9]).
But the guarantee of optimality during best-first search might not be the ultimate goal. When running the algorithm we observed that often, say the first 2-3 letters of the acrostic were generated one after the other, but then the algorithm proceeds with a node in higher layers. The reason for this is that the heuristic h underestimate the cost too rigorously. As a consequence, the best-first search degenerates to a breadth-first search. Hence, especially when computing power is a scarce resource one should better be off with a depth-preferring strategy.

2.4 Operators

most common according to the author

2.4.1 Word Insertion or Deletion

The idea around this operator is to insert words in the text or delete words from it, in order to insert new letters and accomplish the goal acrostic or to remove words and change the position of

words inside the text.
To illustrate the execution, consider the following text[1]:

> `Ah ja, ich heisse Frederik Hoske und ich bin 13 Jahre.` *`Ich kann nicht vorstellen`*`, weil ich kaum Deutsch sprechen kann. Trotzdem versuche ich es. Ich habe zwei Geschwister Mein Bruder der 16 Jahre alt ist und meine Schweseter ist elf.`

Figure 2.1: Example of word insertion application − Original Text

After inserting the word *"mir"* in the sentence *"Ich kann nicht vorstellen"* in the first line and after breaking a line right before *"Trotzdem"* the algorithm can reach the acrostic *amt*. Note that the insertion of *"mir"* was crucial for the result, once that the letter m was not there.

> `Ah ja, ich heisse Frederik Hoske und ich bin 13 Jahre.` *`Ich kann`*
> *`mir nicht vorstellen`*`, weil ich kaum Deutsch sprechen kann.`
> `Trotzdem versuche ich es. Ich habe zwei Geschwister Mein Bruder der 16 Jahre alt ist und meine Schweseter ist elf.`

Figure 2.2: Example of word insertion application − Paraphrased Text

The Word Insertion or Deletion operator takes as input a text. Then first it tries to insert a new word in each space and second tries to remove each word of the text. The condition to insert a new word w in the *i-th* space of the text is that w has to fit the context around the *i-th* space. It means that from the set of all possible words of the language, only a restricted subset can be inserted in this place. More specifically, the algorithm starts by taking for each space in the text n words around it as context − In our implementation in this context $n = 4$. This is a so called n-gram, an array of words. After this, the n-gram just taken is sent to the context database (which is in this implementation the Netspeak API [2]), that returns the possible words that could be inserted in the required space. For each of these possible words a new version of the text is created with the word inside.
Analogously, for each word w in the text a n-gram including the words around it is created − In our implementation we take two words from each side, so here $n = 5$. w is then taken out of the n-gram, which is tested against the context database to check whether this n-gram is frequent enough in the language. If the answer is positive a new version of the text without w is created. Our implementation allows the adjustment of the minimum frequency cited above, but we set it to zero, so a broader set of deletions is executed.
The queries to the context database are made in form of HTTP requests to the Netspeak web service using the Netspeak API.

[1] This text was adapted for didactic purposes from `http://cornelia.siteware.ch/blog/wordpress/2008/11/03/sich-vorstellen-horverstehen`. Access in January, 2015

2.4.2 Synonyms

The synonym operator has the goal of changing words in the text for other words, which have similar meaning. In general the operator takes a text as input and generates a set of new texts, in which each text has a word replaced by a synonym.
In order to perform the replacements it is required a synonym dictionary, which is know as thesaurus. In our implementation we used Open Thesaurus [3], which is available for download for free. This data source is available as a plain text file, but as the dictionary is accessed many times during the execution of the algorithm, it soon becomes intractable to handle a text file as a database.
To solve this problem we decided to use a NoSQL database server [4], namely, Redis. Redis is an open source advanced key-value pair cache and store [5]. Into the database server we load once the data from the thesaurus in a structured way where, every word is added as a key that points to a set of synonyms. Thus the application can easily and efficiently find similar terms for a given word only by accessing this key.
Naturally it is then required that the Redis server is running and listening to requests when the application runs, and that it has been once loaded by our script with the data from the dictionary.

```
[MY EXAMPLE]
```

Figure 2.3: Example of synonym application − Original Text

```
[MY EXAMPLE]
```

Figure 2.4: Example of synonym application − Paraphrased Text

The application of the Synonym Operation brings much more possibilities for new paraphrased versions of the original text. It happens because, when comparing with Word Insertion and Deletion, the probability of changing a word in the text with this operation is higher once it does not check the context around the changed word, therefore allowing many words to be replaced.
Consequently, the drawback is the considerable loss of quality in the results, in function of the fact that synonyms are strongly related to context, and some replacements may change substantially the meanings of the resulting texts.

2.4.3 Line break

The fastest and most basic operation is the line break. A line break can be applied in two cases:

- After a word when the line length lies in the $[l_{min}, l_{max}]$-window, given by the line length constraints.
- After the end of a sentence.

After performing a line break, the lines following the line break have to be aligned again to satisfy the line length constraints.

For this task we apply a greedy word wrap algorithm, which works as follows: we split the text into words, put the words on the line as long as there is free space, if there is no free space left, we continue with the next line.
When applying the greedy word wrap algorithm we have to ensure that there is no word of length > 20 in the initial text. Otherwise it might happen, that the minimal line length constraint is not fulfilled.
Identifying the end of a sentence in general is a difficult problem. One reason for this is that a period might occur in several contexts, e.g.

- abbreviations (Prof., Dr., d.h., z.B., ...)
- ordinal numbers (der 26. April, Joseph II., 2. Auflage, ...)
- numbers (10.1312, 192.11301, ...)

Another issue is that there is a wide variety of punctuations which could mark the end of a sentence. These punctuations include question marks, exclamation marks, ellipses, semi-cola, cola.
To overcome these problems we make use of a sentence-splitter library, called Sentrick (cf. [6]).

2.4.4 Hyphenation

Related to the line break are hyphenations. A hyphenation is applicable if the line after hyphenating (and line breaking) has a length of at least $l_{min} = 50$. For hyphenating a word we employ a re-implementation of Knuth's hyphenation algorithm in TEX (cf. [7]).
After the hyphenation, the text following the hyphen has to be aligned again to satisfy the line length constraints.
Analog to the line break operation, in order to rearrange the lines we apply a greedy word wrap algorithm.

3 Evaluation of the Results

The goal of the evaluation of the developed application is to show that it is able to generate results of acrostics for texts written in the German language. So we analyze the success rate, the operation application rate and the influence of the several variations in the test cases. For that we split the evaluation in four battery of tests, basically it is the combination of two possible configuration: The search algorithm, which can be A* ($f(x) = g(x) + h(x)$) or best first A* ($f(x) = h(x)$); and the chosen acrostic, which can then be self-referential or the most common word that starts with the first letter of the text. This list of words was obtained from Wortschatz Universität Leipzig[1].
The configuration on the search algorithm allows the program to be switched between caring, or not, about the quality of the applied operations. That means, when running it the mode best first A* all operations have maximal quality (minimal cost). That leads to a goal oriented computation, once the choice of the operation does not influence the choice of the solution path in the search space. In the other mode, it is possible to have an optimization oriented computation, which is achieved by running with the regular A*-algorithm [11].
In order to vary the input domain, we ran test cases in two different possibilities: the so called, self-referential, where the acrostic is the first word in the input text; and the most common word, in which the acrostic is the word that has the highest occurrence probability and starts with the first letter of the input text. Note that, in every test case the first letter of the text and of the acrostic are equal, this is a crucial decision for the general success of the application, once that by [1], employing the same algorithm, less than 1% of the test cases could generate results, whenever that condition was not held.

3.1 Experiment Setup

FROM WHERE DO THE TEXTS COME? AND THE MOST COMMON ACROSTICS? Limas-Korpus[2] Gutenberg Project[3]
We let the Redis database started in the same system where the tests run. The initialization of data into the key-value store server is realized once by the script that process the thesaurus and it took no more than a couple minutes to be successfully completed. For the consume of the NetSpeak API web service is required an internet connection. As advised by [1], we set line length constraints of $l_{min} = 50$ and $l_{max} = 70$ because of the flexibility it brings to operators like line break and hyphenation. As the application may run for a long time, we set a timeout for its execution, for

[1] `http://wortschatz.uni-leipzig.de/html/wliste.html`
[2] `http://www.korpora.org/Limas/`
[3] `http://www.gutenberg.org/browse/languages/de`

Configuration	Success Rate	Runtime	Nodes	Timeout Rate
A* + Self-reference	S	S	S	S
A* + Most Common	S	S	S	S
Best First A* + Self-reference	S	S	S	S
Best First A* + Most Common	S	S	S	S

Table 3.1: Experimental results

that we chose arbitrarily 15 minutes, but it can eventually be increased or decreased according to the availability of resources and time from the user.

3.2 Experiment Discussion

4 Summary of Findings

The algorithm described in this report is able to build acrostics with short words (we have achieved acrostics with 6 letters) in reasonable time. These acrostics have regular quality, in the sense that some paraphrasing operations are easily perceived by the reader – the wrong hyphenation is one of them. The use of synonyms without checking the context lowers the quality as well. Additionally, the algorithm can generate result in a few minutes for a good number of cases, what makes possible its use in some practical situations, aiding a process that was totally human so far. The review of the result by a human is although necessary.
//comment result here??
In despite of these results, we did not develop all the operators described in the reference paper ([1]) mainly for project time reasons. A broader experimental discussion with a higher number of tests and a discussion about quality metrics on the text would be interesting as well, although it would be out of the scope of the project.
What should be noted in the problem of generating acrostics, is that the success depends highly on certain conditions of the text. When the target acrostic has uncommon letters, it becomes naturally more difficult to accomplish the result. This yields the need for more powerful operators, operators able to change more the structure of the text or insert new elements that aids the generation of the desired letters in the beginning of the lines. In this vein, a grammatical operator that could change swap the position of grammatical elements would be helpful, like illustrated in the example below.

```
Wenn Sie wissen, was Sie wollen, wird es einfacher.
Wenn Sie, was Sie wollen, wissen, wird es einfacher.
Es wird einfacher, wenn Sie wissen, was Sie wollen.
Einfacher wird es, wenn Sie wissen, was Sie wollen.
```

Figure 4.1: Example of possible additional operator

Other possible improvement is the development of a non-empirical method for defining costs for each operation. The choice of such parameters seems to be vital for the success of the search method. According to [1, p. 2023], a bad choice may lead to a breadth-first search, while if the computing resources are scarce the desired is a depth-first search. The time consumed by requests to the n-gram web service are also sources of problems for such environments. But in this case the implementation of a n-gram database locally may smooth the problem.
The use of the Netspeak API seems to be another issue, once it does not produce many options for inserting or removing words. A solution would be the implementation of the complete Google n-gram database locally. Even though the use of context in form of n-gram are sometimes too

restrictive. This could be therefore a discussion for future work.

References

[1] Benno Stein, Matthias Hagen, and Christof Bräutigam. *Generating Acrostics via Paraphrasing and Heuristic Search.*
In Junichi Tsujii and Jan Hajic, editors, 25th International Conference on Computational Linguistics (COLING 14), pages 2018-2029, August 2014. Association for Computational Linguistics.

[2] Martin Potthast, Martin Trenkmann, and Benno Stein. *Netspeak: Assisting Writers in Choosing Words.*
In Cathal Gurrin et al, editors, Advances in Information Retrieval. 32nd European Conference on Information Retrieval (ECIR 10) volume 5993 of Lecture Notes in Computer Science, pages 672, Berlin Heidelberg New York, March 2010. Springer.

[3] *Synonyme - OpenThesaurus - Deutscher Thesaurus.* Available on: <`https://www.openthesaurus.de`>. Accessed in: January 2015.

[4] Jing Han; Haihong, E.; Guan Le; Jian Du. *Survey on NoSQL database.* Pervasive Computing and Applications (ICPCA), 2011 6th International Conference on, pp.363,366, 26-28 October 2011.

[5] *Redis.* Available on: <`http://redis.io`>. Accessed in: January 2015.

[6] *Sentrick.* Available on: `http://sourceforge.net/projects/sentrick/` Accessed in: January 2015.

[7] *TexHyphJ.* Availabe on: `http://sourceforge.net/projects/texhyphj/` Accessed in: January 2015.

[8] *Private Communication* with Rainer Perkuhn, Institut für Deutsche Sprache Programmbereich Korpuslinguistik, via Email, in Dezember 2014. He send us two files of an old study of Cyril Belica, including the first letter frequencies.

[9] `http://de.wikipedia.org/wiki/A*-Algorithmus`

[10] Quasthoff, U.; M. Richter; C. Biemann. *Corpus Portal for Search in Monolingual Corpora.* Proceedings of the fifth international conference on Language Resources and Evaluation, LREC 2006, Genoa, pp. 1799-1802.

[11] Dechter, Rina; Pearl, Judea. *Generalized best-first search strategies and the optimality of A*.* Journal of the ACM (JACM), Volume 32 Issue 3, Pages 505-536, New York, July 1985.