Resolution of analogies between strings in the case of several solutions

DENG, Xulin

EBMT/NLP Lab, IPS, Waseda University

October 6th, 2022

早稲田大学 情報生産システム研究科
Graduate School of Information, Production and Systems, Waseda University

Table of Contents

Research topic

Our research topic is to design a **non-deterministic** algorithm based on an existing deterministic algorithm (Lepage, 1998). Our algorithm should output **all solutions** of a formal analogy in the case there are several solutions:

$$A : B :: C : x \Rightarrow x = ?$$

Baseline algorithm

The algorithm on which this research is based **only outputs one solution, even if there are several solutions for the analogy.**

Baseline algorithm

The algorithm on which this research is based **only outputs one solution, even if there are several solutions for the analogy.** For example:

$$a : abc :: aa : x \quad \Rightarrow \quad x = ?$$

The answers should be *abca*, *aabc* and *abac*, but the program outputs only one of the answers, *aabc*, because it is the first one the algorithm finds.

Background on solving analogies between strings

There are three methods:

- **Shuffle** method (Langlais et al., 2009)
  Definition: $D \in (B \bullet C) \backslash A$
- **Kolmogorov Complexity** method (Murena et al., 2020)
  Definition: $D = (argmin_{prog}\{prog/B = prog(A)\})(C)$
- **Distance** method. (Lepage, 1998)

Background of the algorithm

Accuracy of the three methods on the the data set **Sigmorphon Analogy** (Lepage, 2017), which is the largest data set of analogies we have.

	Number of analogies	Method		
		Complexity	Distance	Shuffle
Total	9,181,112	**96.41**	94.34	87.93

Table: Accuracy of the three methods. Table copied from (Murena et al., 2020).

We suppose that the distance algorithm performs worse because some **solutions are missed**.

Justification for the missing solutions

The Sigmorphon Analogy dataset is a practical and large data set built from data from the Sigmorphon campaign data. But in this data set, **the given answer does not include all solutions, i.e., although theoretically valid, the answers delivered by the program are regarded as incorrect if not equal to the dataset answer.**

Justification for the missing solutions

The Sigmorphon Analogy dataset is a practical and large data set built from data from the Sigmorphon campaign data. But in this data set, **the given answer does not include all solutions, i.e., although theoretically valid, the answers delivered by the program are regarded as incorrect if not equal to the dataset answer.** For example:

$$\begin{aligned} &\mathit{asked} : \mathit{ask} :: \mathit{seemed} : x \\ &\qquad \Rightarrow \quad x = \mathit{seem} \text{ or } \mathit{seme} \end{aligned}$$

The answer of this example is *seem*. Another answer, *seme*, **satisfies the definition of our algorithm.** If an algorithm output is the second answer, it will be regarded as incorrect.

Goal

To develop a non-deterministic version of the existing algorithm.
To evaluate whether we achieve our task:

- Do experiments checking **whether all solutions are output**.

Contributions

- If there are several solutions to an analogy problem, the existing program can output **only one solution** but ours can output **all of them**.
- We **generate a data set** of analogies which consists of analogies with **no solution, one solution and several solutions** for evaluation.

Table of Contents

Edit distance: Pseudo distance

The pseudo edit distance between two strings is the minimal number of operations required to transform one string into another. The **operations** include:

- Insertion
- Deletion
- Substitution

Different from Levenshtein distance, the **cost of insertion is 0**, and the costs of deletion and substitution of a character are 1. One substitution can be also decomposed into one deletion and one insertion.

Baseline Algorithm

Definition of this algorithm

To solve analogy problems, the algorithm is based on a definition of analogy, which is:

$$A : B :: C : D \quad \Rightarrow \begin{cases} pdist(A, B) & = pdist(C, D) \\ pdist(A, C) & = pdist(B, D) \\ |A|_c + |D|_c & = |B|_c + |C|_c, \forall c \end{cases} \tag{1}$$

$|A|_c$ is the number of occurrences of character c in string A.

Example

For the analogy:

$$\textit{arsala} : \textit{mursil} :: \textit{aslama} : x$$
$$\Rightarrow \quad x = ?$$

We compute the matrices and traces (marked by red) exploiting a result presented by Ukkonen (1985):

$$\begin{array}{cccccccccccccc}
\mathrm{l} & \mathrm{i} & \mathrm{s} & \mathrm{r} & \mathrm{u} & \mathrm{m} & & \mathrm{a} & \mathrm{s} & \mathrm{l} & \mathrm{a} & \mathrm{m} & \mathrm{a} \\
\cdot & \cdot & \cdot & \cdot & \textcolor{red}{1} & \textcolor{red}{1} & \mathrm{a} & \textcolor{red}{0} & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \textcolor{red}{1} & 2 & \cdot & \mathrm{r} & \textcolor{red}{1} & 1 & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \textcolor{red}{1} & 2 & \cdot & \cdot & \mathrm{s} & \cdot & \textcolor{red}{1} & \textcolor{red}{1} & \cdot & \cdot & \cdot \\
\cdot & \textcolor{red}{2} & 2 & \cdot & \cdot & \cdot & \mathrm{a} & \cdot & \cdot & 2 & \textcolor{red}{1} & \cdot & \cdot \\
\textcolor{red}{2} & 3 & \cdot & \cdot & \cdot & \cdot & \mathrm{l} & \cdot & \cdot & \cdot & 2 & \textcolor{red}{2} & \cdot \\
\textcolor{red}{3} & \cdot & \cdot & \cdot & \cdot & \cdot & \mathrm{a} & \cdot & \cdot & \cdot & \cdot & 3 & \textcolor{red}{2}
\end{array}$$

Example

We denote the direction of the trace of the current cell in the matrix of strings A and B as dir_{AB} (resp. dir_{AC}). And the operation *Copy* is to copy a character into D at the beginning of the D.

Example

We denote the direction of the trace of the current cell in the matrix of strings A and B as dir_{AB} (resp. dir_{AC}). And the operation *Copy* is to copy a character into D at the beginning of the D.

dir_{AB}	dir_{AC}	*Copy*	move
vertical	diagonal		A and C
diagonal	diagonal	m	A, B and C
diagonal	diagonal	i	A, B and C
diagonal	horizontal	l	C
diagonal	diagonal	s	A, B and C
diagonal	vertical		A and B
horizontal	diagonal	u	B
diagonal	diagonal	m	A, B and C

Then we can get the answer 'muslim'.

Table of Contents

Several solutions

If an analogy problem has **at least two solutions, there should be at least two traces** in either the distance matrix between A and B or the matrix between A and C.

The reason **why the algorithm only gives one solution is that the algorithm does not search for all the traces** in the matrices, but follows the first one it finds.

Iterative version and Recursive version

- The existing algorithm uses an iterative way and **ends as soon as it finds one of the solutions**.
- Our proposal is to use a recursive way to improve the realization of the algorithm. Using a recursive way, the algorithm **not only goes through the trace it first finds but it backtracks to the beginning of the trace and seeks if there are other traces**.

Example

For example:

$$aa : ab :: aaa : x$$
$$\Rightarrow \quad x = ?$$

We can get 2 traces (marked by red) in matrix *aa : aaa*.

$$
\begin{array}{cccccc}
b & a & & a & a & a \\
\cdot & 0 & a & {\color{red}0} & {\color{red}1} & \cdot \\
1 & \cdot & a & \cdot & 0 & {\color{red}1}
\end{array}
\qquad
\begin{array}{cccccc}
b & a & & a & a & a \\
\cdot & 0 & a & {\color{red}0} & 1 & \cdot \\
1 & \cdot & a & \cdot & {\color{red}0} & {\color{red}1}
\end{array}
$$

The result will be 'aab' and 'aba'.

Table of Contents

Data set we generate

To evaluate our proposal, we generate a data set consists of analogy problems which have **no solution, only one solution and several solutions**.

Generation of analogy with No solution

To generate an analogy $A : B :: C : \textit{no solution}$.

- Randomly generate a string as A:
- Randomly select a character in A which will not appear in B or C:
- Randomly generate strings without the character selected in step 2 get B and C:

Generation of analogy with No solution

Example:

- Generate string ‘*do*’.
- Select character ‘*d*’.
- Generate ‘*two*’ and ‘*to*’ as B and C.

Then we get analogy: $do : two :: to : no\ solution$

Generation of analogy with One solution

To generate an analogy $A : B :: C : D$.

- Randomly generate a string A and select a position to divide A into prefix B' and suffix C'.
- Create any number of sub-strings randomly, each without any of the characters in A (Character constraint).
- Insert the sub-strings into B' and C' and get B and C, respecting the position constraints:
  - no sub-string is inserted as a suffix of the prefix B'.
  - no sub-string is inserted as a prefix of the suffix C'.

Generation of analogy with One solution

Example:

- Generate a string $A = abcd$. Select the position of a character, for instance, ”c” to divide A into prefix abc and suffix d.
- Create three sub-strings mn, op and xyz.
- Insert the sub-strings respecting the constraints. For instance, get $B = amnbopc$ and $C = dxyz$.

Then we get analogy: $abcd : amnbopc :: dopmn : mnopxyz$

Generation of analogy with Several solutions

Without the position constraint:

$$\begin{aligned} & abcd : abM :: Ncd : x \\ & \quad \Rightarrow \quad x = \textit{any string in } M \bullet N \end{aligned}$$

Generation of analogy with Several solutions

Without the character constraint:

$$abcd : ab :: cd\textcolor{red}{NcMdN} : x$$
$$\Rightarrow \quad x = any\ string\ in\ cdNcMdN \setminus cd$$

Table of Contents

Data set

We use the **Sigmophon analogy** data set (Lepage, 2017). This data set contains 9,181,112 analogy problems from 10 languages.

The data generated by ourselves consists of 3,000,000 analogy problems.

Sketch of dataset

	Size	Average length of solutions	Number of solutions
Zero	1,000,000	-	$= 0.0$
One	1,000,000	5.25	$= 1.0$
Several	1,000,000	6.21	avg. 3.4
Sigmorphon	9,181,112	8.74	$= 1.0$

Format

The data file contains one analogy per line:

- Arabic:
  fanniyyayn : *ḥammār* :: *al-fanniyyayn* : *al-ḥammār*
  kubbāya : *ḡarāiz* :: *al-kubbāya* : *al-ḡarāiz*
- Finnish:
  katko : *kakomaisillaan* :: *dekoodata* : *dekoodaamaisillaan*
  karahteerata : *katko* :: *karahteeraamaisillaan* : *kakomaisillaan*
- Spanish:
  muten : *mutaban* :: *loquee* : *loqueaba*
  muten : *mutaban* :: *derrumbe* : *derrumbaba*

Format

- Analogy with no solution:
  undo : *walk* :: *do* : *no solution*
  aa : *bb* :: *cc* : *no solution*
- Analogy with one solution:
  do : *read* :: *open* : *reapen*
  see : *case* :: *equal* : *caqual*
- Analogy with several solutions:
  do : *d* :: *openopen* : *penopen*, *openpen*
  at : *a* :: *tagtag* : *tagag*, *agtag*

Experiment results: overview

Dataset		Average time (μs)	Precision (%)	Recall (%)	F-measure
Sig	baseline	1.40	92.0	92.2	92.0
	proposed	565.00	34.8	**98.5**	51.2
Zero	baseline	0.17	100.0	100.0	100.0
	proposed	0.26	100.0	**100.0**	100.0
One	baseline	0.63	97.2	81.9	88.9
	proposed	1.38	99.7	**100.0**	99.8
Sev	baseline	1.26	96.7	30.7	46.6
	proposed	1.83	99.4	**100.0**	99.7

Table: Assessment of baseline and proposed methods on the data sets

Resluts on the Sigmorphon Analogy Dataset

		Method			
Language	Data size	Complexity	Distance	Shuffle	**Proposal**
Arabic	165,113	87.18%	93.33%	81.91%	98.91%
Finnish	313,011	93.69%	92.76%	78.75%	97.13%
Georgian	3,066,273	99.35%	97.54%	88.42%	99.85%
German	730,427	98.84%	96.21%	95.42%	99.81%
Hungarian	2,912,310	95.71%	92.61%	86.02%	98.62%
Maltese	28,365	96.38%	84.72%	91.84%	98.17%
Navajo	321,473	81.21%	86.87%	78.95%	97.45%
Russian	552,423	96.41%	97.26%	95.46%	99.48%
Spanish	845,996	96.73%	96.13%	94.42%	96.18%
Turkish	245,721	89.45%	69.97%	70.06%	98.63%
Total	9,181,112	96.41%	94.34%	87.93%	**98.50%**

Table: Results with the methods.

Table of Contents

Conclusion

By introducing a recursive approach we could expand the scope in solutions and could increase the performance of the Distance algorithm on the dataset of analogy puzzles extracted from the Sigmorphon Analogy Dataset.

Thank you for listening.

Langlais, P., Zweigenbaum, P., and Yvon, F. (2009). Improvements in analogical learning: application to translating multi-terms of the medical domain. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics (EACL 2009)*, pages 487–495, Athens, Greece. Association for Computational Linguistics.

Lepage, Y. (1998). Solving analogies on words: an algorithm. In *Proceedings of the 17th International Conference on Computational Linguistics and 36th Annual Meeting of the Association for Computational Linguistics (COLING-ACL'98)*, volume I, pages 728–735, Montréal.

Lepage, Y. (2017). Character-position arithmetic for analogy questions between word forms. In *ICCBR (Workshops)*, pages 23–32.

Murena, P.-A., Al-Ghossein, M., Dessalles, J.-L., Cornuéjols, A., et al. (2020). Solving analogies on words based on minimal complexity transformation. In *IJCAI*, pages 1848–1854.

Ukkonen, E. (1985). Algorithms for approximate string matching. volume 64, pages 100–118. Elsevier.

Wagner, R. A. and Fischer, M. J. (1974). The string-to-string correction problem. volume 21, pages 168–173. ACM New York, NY, USA.

Edit distance: Pseudo distance

The **length of the longest common subsequence** between A and B is also called the **similitude** between A and B. The similitude between A and B is equal to the length of A minus the pseudo distance between A and B.
We denote $sim(A, B)$ is the similitude between A and B, and $|A|$ is the length of A. Then:

$$sim(A, B) = |A| - pdist(A, B) \tag{2}$$

Example

If we transform 'arsala' to 'mursil':

- The longest common subsequence for these two strings 'arsala' and 'mursil' is 'rsl'.
- Insertion: arsala $\rightarrow$ aursala (1 op.)
- Deletion: aursala $\rightarrow$ aursal (1 op.)
- Substitution: aursal $\rightarrow$ mursil (2 ops.)

We denote pseudo edit distance as $pdist(A, B)$. Then:

$$pdist(A, B) = 3$$

$$sim(A, B) = |A| - pdist(A, B) = 6 - 3 = 3$$

Appendix: operation to get solution

- Case: $dir_{AB} = dir_{AC} =$ diagonal:
  Copy $B[i_B] + C[i_C] - A[i_A]$, i.e., copy the character in B or C which does not belong to A, then move in three words at the same time.
- Case: $dir_{AB} = dir_{AC} =$ horizontal:
  Copy the character belongs to the one with less similitude with A.
- Case: $dir_{AB} = dir_{AC} =$ vertical:
  Move in A.
- Case: dir_{AB} (resp. dir_{AC}) $=$ horizontal and $dir_{AB} \neq dir_{AC}$:
  Copy $B[i_B]$. (resp. *Copy* $C[i_C]$.)
- Case: dir_{AB} (resp. dir_{AC}) $=$ vertical and $dir_{AB} \neq dir_{AC}$:
  Move in A and C. (resp. Move in A and B.)

Pseudo Distance and Coverage Constraint

If an analogy problem has **at least one solution**, all the letters in A must appear either in B or C, i.e.,

$$sim(A, B) + sim(A, C) \geq |A|$$

Considering formula (1):

$$|A| \geq pdist(A, B) + pdist(A, C)$$

Pseudo Distance and Coverage Constraint

When $|A| > pdist(A, B) + pdist(A, C)$, some characters belong to A are common to B and C. Such characters are also common to the solution D. We denote the number of such characters as $com(A, B, C, D)$, then:

$$|A| = pdist(A, B) + pdist(A, C) + com(A, B, C, D)$$

So, **every time** before each operation of solving analogy problems, we check the constraint. This helps the algorithm to end **as soon as it finds a failure**. This is also applied in our propasal.

Example

For analogy:

$$read : do :: readable : doable$$

We have:

$$|read| = 4, pdist(read, do) = 3, pdist(read, readable) = 0$$

$$com(read, do, readable, doable) = 1$$

Then:

$$|read| = pdist(read, do) + pdist(read, readable) +$$

$$com(read, do, readable, doable) = 3 + 1 = 4$$

Compute Matrices

The formula definition of the pseudo distance is:
if $min(i,j) = 0$,

$$pdist_{A,B}(i,j) = max(i,j) \tag{3}$$

otherwise,

$$pdist_{A,B}(i,j) = min \begin{cases} pdist_{A,B}(i-1,j) + 0 \\ pdist_{A,B}(i,j-1) + 1 \\ pdist_{A,B}(i-1,j-1) + 1_{(A_i \neq B_j)} \end{cases} \tag{4}$$

$1_{(A_i \neq B_j)}$ is an indicator function. When $A_i \neq B_j$, it is 1, otherwise it is 0.
$pdist_{A,B}(i,j)$ denotes the pseudo distance between the first i characters of string A and the first j characters of string B.

Compute Matrices

There are three cases to determine the trace, which are three cases of the direction of the cells (coordinate in the matrix is [i, j]) in the pseudo distance matrix to find the traces from the end of the matrix to the beginning of matrix. This is a method proposed by Wagner and Fischer (1974).

- Case horizontal (Insertion): $pdist_{A,B}(i,j) = pdist_{A,B}(i-1,j) + 0$
- Case vertical (Deletion): $pdist_{A,B}(i,j) = pdist_{A,B}(i,j-1) + 1$
- Case diagonal (Substitution): $pdist_{A,B}(i,j) = pdist_{A,B}(i-1,j-1) + 1_{(A_i \neq B_j)}$

Generation of analogy with No solution

If we generate a analogy $A : B :: C : \textit{no solution}$.

- Randomly generate a string as A:
- Randomly select a character in A which will not appear in B or C:
- Randomly generate strings without the character selected in step 2 get B and C:

Generation of analogy with No solution

Example:

- Generate string '*do*'.
- Select character '*d*'.
- Generate '*two*' and '*to*' as B and C.

Then we get analogy: *do* : *two* :: *to* : *no solution*

Generation of analogy with Several solutions

If we generate a analogy $A : B :: C : Ds$.

- Steps for generating A, B and C are the same as the generation of analogy with one solution, but there is only one insertion of sub-string in C and no insertion in B. C should consist of only one character before the insertion.
- Repeat C for random times. Then get several answers according to C.

Generation of analogy with Several solutions

Example:

- Generate string '*abc*'.
- Select character '*b*' and divide '*abc*' into '*ab*' and '*c*' as *B* and *C*.
- Generate '*xy*'. Insert it to get *C*: '*cxy*'.
- Repeat *C* for 3 times and get *C* '*cxycxycxy*'. Then we get *D*: '*xycxycxy*', '*cxyxycxy*', and '*cxycxyxy*'.

Then we get analogy:
$abc : ab :: cxycxycxy : xycxycxy \text{ or } cxyxycxy \text{ or } cxycxyxy$

Sketch of Iterative version

```
def solve_nlg_i():
    result_D = ''
    while halting_constraint():
        if constraint(com(A, B, C, D)):
            if dir_AB = dir_AC = diagonal:
                #do the moves in this case
            elif ...
            else:
                break
        else:
            break
    return result_D
```

Sketch of Recursive version

```
def solve_nlg_r(parameters):
    if constraint(com(A, B, C, D)):
        if dir_AB = dir_AC = diagonal:
            #do the moves in this case
            #get new parameters, store old parameters
            solve_nlg_r(new_parameters)
        if ...

    if constraint_to_halt():
        if constraint(com(A, B, C, D)):
            set_result.add(result)
def solve_nlg():
    set_result = set()
    solve_nlg_r(parameters)
    return set_result
```