大規模候補リストを利用したトランスリタレーション

佐　藤　理　史

名古屋大学大学院工学研究科電子情報システム専攻
ssato@nuee.nagoya-u.ac.jp

1. はじめに

言語横断検索や機械翻訳などの多言語情報処理において、固有名を正しく翻訳することは非常に重要である。使用する文字集合が異なる 2 言語間においては、固有名は発音に基づいて翻訳されるのが普通である。これをトランスリタレーション (transliteration) と呼ぶ。そのような言語対に対して、これまで、機械的にトランスリタレーションを実現する方法が研究されてきている [1)~3)]。

これまで提案されてきた方法の基本的な枠組は、ソース側のタームに音素または字素の写像規則を適用し、ターゲット側のタームを生成するというものである。この枠組は、いままでにない新しい訳語を産み出す能力を持つ。これを生産的な枠組と名付けよう。この枠組は、他の知識 (たとえば、コーパスにおける頻度やウェブのヒット数) を用いて補強することができる [4),5)]。

これに対して、我々は、非生産的な枠組を採用する。つまり、「すでに誰かによって過去に訳されたことがある」ことを仮定し、それを見つけることに専念する。より具体的には、正解を含む巨大な候補リストが与えられることを仮定する。

このような枠組を採用する理由は、そのような候補リストが現実に入手可能となってきたからである。例えば、日本語では、トランスリタレートされた訳語は、カタカナで記述される。もし、巨大な日本語コーパス (あるいは日本語ウェブページ全体) から、そこに含まれるすべてのカタカナ語を抽出したリストを作成すれば、ほとんどのトランスリタレーションの正解がそのリストに含まれていることが期待できる。そうであれば、訳語を新たに作る必要はなく、単に、そのリストの中から正解を選ぶという選択問題を解けばよい。

理論的には、選択問題は、リストの各要素に対して何らかのスコアを定義し、そのスコアの最大のものを選ぶことによって解くことができる。このスコアに、望むべき性質—その要素が、与えられた入力の訳語としてどの程度ふさわしいか—を反映させればよい。しかしながら、巨大な候補リストに対してこのような素朴な方法を適用するのは現実的ではない。本論文では、これに代わる効率的なアルゴリズムを提案する。

2. 問題の形式化

2.1 トランスリタレーションのための文法

まず最初に、トランスリタレーション対を形式的に定義するために、2 つの言語の文字列の対を生成する単純な形式文法を導入する。

$$G = (A, B, R) \tag{1}$$

この式に示すように、文法 G は、次の 3 つの要素から構成される。

(1)　A — ソース言語の文字集合
(2)　B — ターゲット言語の文字集合
(3)　R — 規則集合

ここで、規則 $r \in R$ は、次の形式をとる。

$$r = \langle \alpha, \beta \rangle \quad \text{where } \alpha \in A^*, \beta \in B^*, \max(|\alpha|, |\beta|) \geq 1 \tag{2}$$

この文法において、規則列 $\delta \in R^*$ は、文字列の対を生成する。

$$\begin{aligned} \delta = r_1 r_2 \cdots r_n &= \langle \alpha_1, \beta_1 \rangle \langle \alpha_2, \beta_2 \rangle \cdots \langle \alpha_n, \beta_n \rangle \\ &= \langle \alpha_1 \alpha_2 \cdots \alpha_n, \beta_1 \beta_2 \cdots \beta_n \rangle \end{aligned} \tag{3}$$

それぞれの規則は、生成した文字列対における部分的な対応関係 (アライメント) を意味する。なお、以下では、規則列 δ のソース側およびターゲット側を、それぞれ、$\mathrm{src}(\delta)$、$\mathrm{tgt}(\delta)$ と書くこととする。

上記の規則集合を、2 つの言語の文字列間の許容される写像の定義と解釈すれば、文法 G は、トランスリタレーションのモデルとみなすことができる。しかしながら、実際に観察されるトランスリタレーション対を完全にカバーする規則集合を定義することは容易ではない。そこで、次に、規則集合を拡張することを考える。

2.2 文法の拡張

ある文法 G に対して、拡張された文法 $\dot{G}$ を次のように定義する。

$$\begin{aligned} &\dot{G} = (A, B, \dot{R}) \\ &\text{where } \dot{R} = R \cup \bigcup_{\forall a \in A} \{\langle a, \epsilon \rangle\} \cup \bigcup_{\forall b \in B} \{\langle \epsilon, b \rangle\} \end{aligned} \tag{4}$$

ここで ϵ は、空文字列を表す。拡張された文法では、元の規則集合 R に加えて、削除規則集合 ($\bigcup\{\langle a, \epsilon \rangle\}$)、および、挿入規則集合 ($\bigcup\{\langle \epsilon, b \rangle\}$) を追加する。すでに述べたように、元の規則集合は、トランスリタレーションにおける標準的な写像をカバーする。それらでカバーできない例外的な写像を扱うために削除規則と挿入規則を導

入するのである。

上式に示すように、ソース側の文字集合 A に含まれるすべての文字に対して削除規則を導入し、ターゲット側の文字集合 B に含まれるすべての文字に対して挿入規則を導入する。これらの規則により、任意の文字列の対 $\langle\sigma,\tau\rangle$ ($\sigma \in A^*$, $\tau \in B^*$) に対して、その対を生成する規則列 $\dot{\delta}$ が、すくなくとも 1 つは存在することになる。

2.3 ベストアライメント

与えられた規則列 $\dot{\delta} \in \dot{R}^*$ から、ソース側の文字列 $\mathrm{src}(\dot{\delta})$、および、ターゲット側の文字列 $\mathrm{tgt}(\dot{\delta})$ を求めることはたやすい。一方、与えられた文字列の対 $\langle\sigma,\tau\rangle$ に対して、それを生成する規則列 $\dot{\delta}$ を見つけることは、自明ではない。そのような規則列は、一般に複数存在する。それらの中で我々が興味があるのは、最も良く部分対応がとれたもの、つまり、使用されている挿入・削除規則の数が最も少ないものである。これをベストアライメントと呼ぶことにしよう。

ベストアライメントを求めるために、まず、規則列 $\dot{\delta} \in \dot{R}^*$ に対して、次のようなコストを定義する。

$$\mathrm{cost}(\dot{\delta}) = \sum_{i=1}^{n} \mathrm{penalty}(\dot{r}_i) \quad (\dot{\delta} = \dot{r}_1\dot{r}_2\cdots\dot{r}_n) \quad (5)$$

$$\mathrm{penalty}(\dot{r}) = \begin{cases} 0 & (\text{if } \dot{r} \in R) \\ 1 & (\text{otherwise}) \end{cases} \quad (6)$$

このコストを用いて、σ と τ のベストアライメントは、次のように表すことができる。

$$\mathrm{best\_align}(\sigma,\tau,\dot{R}) = \mathrm{argmin}\,\mathrm{cost}(\dot{\delta})$$
$$\text{satisfying } \dot{\delta} \in \dot{R}^*,\ \mathrm{src}(\dot{\delta}) = \sigma,\ \mathrm{tgt}(\dot{\delta}) = \tau \quad (7)$$

2.4 非生産型トランスリタレーション

以上の準備を経て、非生産型トランスリタレーション (non-productive machine transliteration; NPMT) の枠組を次のように定義する。

与えられるもの
(1) 文法 $G = (A, B, R)$
(2) ソース側の文字列 $\sigma \in A^*$
(3) ターゲット側の候補リスト $T \subset B^*$
求めるもの
$\dot{\delta}_{min}(\sigma, T, \dot{R}) = \dot{\delta}_{min} = \mathrm{argmin}\,\mathrm{cost}(\dot{\delta})$
satisfying $\dot{\delta} \in \dot{R}^*,\ \mathrm{src}(\dot{\delta}) = \sigma,\ \mathrm{tgt}(\dot{\delta}) \in T$
出力
(1) $\tau = \mathrm{tgt}(\dot{\delta}_{min}) \in T$
(2) $c_{min} = \mathrm{cost}(\dot{\delta}_{min})$

もし、$\dot{\delta}_{min}$ が一意に定まらない場合は、複数の τ が得られることになる。

この定義から明らかなように、この枠組で得られる出力 τ は、かならず、与えられたリスト T の要素である。つまり、この枠組は、これまで知られていなかった新たなトランスリタレーションを産み出すことはない。非生産型トランスリタレーションという名前は、この事実に由来する。

3. NPMT を解くアルゴリズム

NPMT の枠組により、トランスリタレーション問題は、ある条件を見たす規則列 $\dot{\delta}$ を探索する問題に変換される。これを実現する素朴な方法は、T のすべての要素 τ に対して式 (7) を計算し、そのなかで最小コストとなるものを見つけるという方法である。これは、式 (7) を $|T|$ 回計算することを意味し、$|T|$ が大きい場合には非現実的となる。そこで、プレフィックス・フィルタリングと反復型コスト束縛探索の 2 つの技法を用いた効率的なアルゴリズムを考える。

3.1 探索の基本戦略

文字列 σ の 1 文字目から i 文字目までの長さ i の部分文字列 (プレフィックス) を σ_1^i と表すこととする (文字列の先頭文字を 1 文字目と数える)。同様に、規則列 $\dot{\delta}$ の 1 番目の規則から k 番目の規則までの部分列を $\dot{\delta}_1^k$ と書くこととする。

求める解 $\dot{\delta}$ は、$\mathrm{src}(\dot{\delta}) = \sigma$ を満たす必要があるので、解の部分列 $\dot{\delta}_1^k$ は、$\mathrm{src}(\dot{\delta}_1^k) = \sigma_1^i$ $(i \le |\sigma|)$ を満たす必要がある (ソース側条件)。同時に、解 $\dot{\delta}$ は、$\mathrm{tgt}(\dot{\delta}) \in T$ を満たす必要がある。そのため、解の部分列 $\dot{\delta}_1^k$ のターゲット側の文字列 $\mathrm{tgt}(\dot{\delta}_1^k)$ は、かならず、T のいずれかの要素のプレフィックスとなっていなければならない (ターゲット側条件)。これらの条件を満たす $\dot{\delta}_1^k$ を部分解 と呼ぶ。

探索の基本的な戦略は、ある i に対する部分解の集合を、$i = 0, 1, \cdots, |\sigma|$ の順に計算していくというものである。ここで、値を動かす添字は k でなく i であることに注意されたい。

この探索戦略は、生成−検査法である。すなわち、ソース側部分列 σ_1^i に対して、$\mathrm{src}(\dot{\delta}_1^k) = \sigma_1^i$ を満たす規則列 $\dot{\delta}_1^k$ を列挙したのち、ターゲット側条件を満たすもののみを残す (これをプレフィックス・フィルタリングと呼ぶ) というものである。このプレフィックス・フィルタリングはかなり強力であり、探索空間を大幅に削減する。それにもかかわらず、文法の拡張の際に導入した挿入規則の影響により、上記の列挙を完全に行なうことは、現実的ではない。そこで、実際の探索では、ある閾値以下のコストをもつ解だけを探索する。これをコスト束縛探索と呼ぶ。

3.2 部分解が満たすべき条件

ここでは、与えられた閾値 c_{th} 以下のコストを持つ解をすべて探索する方法を考える。

まず、ある i に対する部分解の集合を S_i と書き、その要素を s_i と書くことにしよう。s_i が満たすべき条件は、次の通りである。

$$\text{(ソース側条件)} \quad \mathrm{src}(s_i) = \sigma_1^i \quad (8)$$
$$\text{(ターゲット側条件)} \quad \mathrm{tgt}(s_i) \in \mathrm{prefix}(T) \quad (9)$$

ここで、$\mathrm{prefix}(T)$ は、T のすべての要素のすべてのプレフィックスを重複を除いて列挙した集合を表すものとする。

さて、s_i は、以下に示すように再帰的に定義することができる。(同時に、式 (8) と (9) を満たす必要がある。)

$$s_0 = \lambda \tag{10}$$

$$s_i = s_{i-|\mathrm{src}(\dot{r})|} + \dot{r} \quad (|\mathrm{src}(\dot{r})| > 0,\ i \geq 1) \tag{11}$$

$$s_i = s'_i + \dot{r} \quad (|\mathrm{src}(\dot{r})| = 0,\ i \geq 0) \tag{12}$$

ここで、λ は長さ 0 の規則列、演算子 + は規則列の結合を意味するものとする。式 (12) の右辺の s'_i は S_i の要素であるが、左辺と s_i とは異なるのでプライムをつけて表わした。

式 (11) と (12) は、いずれも、右辺の第 1 項の部分解 (規則列) に対して、$\dot{r}$ を末尾に追加した規則列を作成する操作を表現している。式 (6) の定義より、$\mathrm{penalty}(\dot{r}) \geq 0$ が成り立つので、左辺 s_i のコストは、右辺の第 1 項のコストより小さくなることはない。つまり、コスト c_{th} 以下の解を求めるのであれば、部分解 s_i に対して、以下のような条件を設定してよい。

$$(\text{コスト条件}) \quad \mathrm{cost}(s_i) \leq c_{th} \tag{13}$$

そして、この条件を満たさない部分解は、その時点で枝刈りしてよい。

式 (12) で用いられる規則 $\dot{r}$ は、$\mathrm{src}(\dot{r}) = \epsilon$ なる規則である。このような規則は、式 (9) と (13) の条件を満たす限り何度でも適用できる。後者のコスト条件 (13) は、この適用を抑制する働きを持つ。

$c_{th} = 0$ の場合は、拡張する前の規則集合 R だけを使って、解を探索することを意味する。$c_{th} = 1$ の場合は、それぞれの解において、R に含まれない規則は高々1 つしか使えないという条件で解を探索することを意味する。c_{th} が大きくなるにつれて、探索空間は急速に増大する。そこで、まず、$c_{th} = 0$ として探索を行ない、解が見つからなかった場合のみ、c_{th} の値を 1 増加させて探索を再度試みる方法をとる。これは反復深化探索 (iterative deepening search) と同じ考え方なので、反復型コスト束縛探索と呼ぶこととする。

3.3 探索アルゴリズム

以上をまとめると、与えられた文法 G および候補リスト T に対して NPMT を解くアルゴリズムは次のようになる。(部分解は、式 (8)(9)(13) を満たす必要がある。)

(0) ソース側文字列 σ と許容される最大コスト c_{max} が与えられる。

(1) $c_{th} = 0, 1, \cdots, c_{max}$ に対して、以下を行なう
  (a) $S_0 = \{\lambda\}$
  (b) 式 (12) を満たすものを、S_0 に追加する。
  (c) $i = 1, 2, \cdots, |\sigma|$ に対して、以下を行なう
    (i) 式 (11) を満たすものを S_i とする。
    (ii) 式 (12) を満たすものを、S_i に追加する。
  (d) $S_{|\sigma|}$ の要素 $s_{|\sigma|}$ のうち、$\mathrm{tgt}(s_{|\sigma|}) \in T$ を満たすものを解集合とする。
  (e) 解集合が空でなければ、解集合と c_{th} を出力して終了。

(2) 解が見つからなかったことを報告して終了。

このアルゴリズムは、コストが c_{max} 以下の解が存在する場合、最小コストの解をすべて出力する。コストが c_{max} 以下のすべての解を出力するわけではない点に注意されたい。

4. 実験と評価

上記のアルゴリズムを Ruby を用いて実装した。本プログラムでは、ターゲット側条件をチェックする部分 (プレフィックス・フィルタリング) で、サフィックスアレイのツール sary☆を利用している。

本実験では、トランスリタレーションの対象として、外国人名 (フルネーム) を用いた。外国人名のトランスリタレーションは例外が多く、かなり難しいタスクである。まず、HeiNER[6] の日英対訳辞書から、1,161 対の外国人名トランスリタレーション対を抽出し、その半分をシステム開発 (規則集合のチューニング) に、残り半分を評価に用いた。

本実験で用いた文法 (規則集合) は、文献 7) に基づき、著者が作成した。その後、上述の実例対を用いて調整を行なった。最終的に、規則の数は 479 となった。なお、日本語側の文字集合はカタカナではなく、専用に設計したローマ字である。

非生産的トランスリタレーションの実行には、候補リスト T が必要となる。実験では、次の 3 種類の候補リストを使用した。

T_J HeiNER の日本語カタカナ見出し全て (58,083 件)
T_{E1} HeiNER の英語見出し全て (1,468,122 件)
T_{E2} 上記のうち、対応する日本語見出しを持つもの (119,453 件)

なお、HeiNER は多言語固有名辞書であり、人名以外の固有名も含まれている。

プログラムの出力結果は、次の 4 種類に分類できる。

Perfect 正解のみを出力
Ambiguous 正解とそれ以外を出力
False 正解以外を出力
Miss 出力なし

Ambiguous は、出力の数が高々数個であれば、大きな問題とはならない。最も深刻な誤りは *False* であり、これは、規則集合の不備により発生する。*Miss* は、許容される最大コスト c_{max} が、実際のトランスリタレーション対のコストを下回る場合に発生する。

表 1 に英日方向の実行例を示す。この表において、PF は、プレフィックス・フィルタリングの実行回数を示す。最後の例を除いて、いずれも正解が得られている。

表 2 に実験結果を示す。(a) は英日方向、(b) は日英方向で候補リスト T_{E2} を用いた場合、(c) は候補リスト T_{E1} を用いた場合の結果である。すべての場合において、

☆ http://sary.sourceforge.net

表 1　実行例 (英日方法)

入力 σ	c_{min}	ローマ字出力 τ	カタカナ出力	time(sec)	PF
Edmund Weiss	0	etomu0to@vaisu	エトムント・ヴァイス	0.0104	363
Ludwig Thuille	0	ru=tovihi@tuire	ルートヴィヒ・トゥイレ	0.0378	752
Javier Vázquez	0	habia=@basukesu	ハビアー・バスケス	0.0640	2152
Leonid Brezhnev	1	reoni=do@bure3inefu	レオニード・ブレジネフ	0.1821	5499
Jean-Jacques Nattiez	1	3a0-3a!ku@natie	ジャン＝ジャック・ナティエ	0.5858	19810
Amitabh Bachchan	2	amita=bu@ba!ca0	アミターブ・バッチャン	2.6999	99946
Pavlo Skoropadskyi	3	pa=veru@sukoropa=2ukii	パーヴェル・スコロパーツキイ	9.3214	285388
Miguel Bernal Jiménez			(ミゲル・ベルナル＝ヒメネス)	14.8818	433206

表 2　実験結果

(a) 英日方向 ($|T_J| = 58{,}083$)

	type					time
c_{min}	P	A	F	M	total	(sec)
0	430	8	0		438	0.023
1	85	4	2		91	0.329
2	20	5	3		28	3.071
3	7	4	2		13	20.638
–				10	10	17.495
total	542	21	7	10	580	
	93%	4%	1%	2%	100%	

(b) 日英方向 ($|T_{E2}| = 119{,}453$)

	type					time
c_{min}	P	A	F	M	total	(sec)
0	432	6	0		438	0.049
1	82	7	2		91	0.676
2	21	4	6		31	4.590
3	11	1	3		15	45.889
–				5	5	46.500
total	546	18	11	5	580	
	94%	3%	2%	1%	100%	

(c) 日英方向 ($|T_{E1}| = 1{,}468{,}122$)

	type					time
c_{min}	P	A	F	M	total	(sec)
0	385	53	11		449	0.140
1	59	20	12		91	2.163
2	14	5	7		26	16.794
3	4	4	1		9	233.131
–				5	5	253.438
total	462	82	31	5	580	
	80%	14%	5%	1%	100%	

$c_{max} = 3$ を用いた。この表より、作成したプログラムは非常に高速であることがわかる。$c_{th} = 0$ で探索が終了した場合 (つまり、$c_{min} = 0$ の場合) の平均計算時間は、23ms から 140ms である。これは、候補リストのすべての要素に対してコストを計算した後、最小コストのものを選ぶ素朴な方法と比べて、劇的に高速である。(素朴な方法の計算時間は、おおよそ $0.9 * |T|$ ms である。)

c_{th} の値が大きくなるについて、探索空間は急速に拡大し、平均計算時間も長くなる。実時間での応答が必要なアプリケーションにおいては、c_{max} を 2 以下に設定する必要があろう。これは、拡張する前の規則集合によって、実際に現れるトランスリタレーション対のほとんどをカバーする必要があるということを意味する。

表 2 は、現在の規則集合が妥当であることを示している。(a) と (b) においては、93–94%の入力に対して、正しいトランスリタレーションのみを出力 (*Perfect*) しており、*False* と *Miss* は、合わせても 3%である。

(c) においては、これほど高い性能は得られていない。これは、使用した巨大候補リストの中に、類似したスペルや音を持つ要素が数多く存在するためである。*false* の割合の増加は、規則集合に改良の余地があることを示している。しかしながら、全体としての性能は依然として高く、明らかな誤りである *False* と *Miss* の合計は、合わせても 6%に留まっている。

5. おわりに

本論文では、非生産型トランスリタレーションと名付けた新しい枠組と、その効率的なアルゴリズムを示した。この枠組は、正解を含む候補リストが与えられることを仮定する点が、これまでの枠組と大きく異なる。提案したアルゴリズムが 100 万件を越える候補リストに対しても現実的な時間で動作することを、実験的に示した。

本研究は、科学研究費補助金基盤研究 (B)「辞書自動編纂のためのテクノロジー」課題番号 21300094 の支援、および、栢森情報科学振興財団の助成（「ウェブを利用した対訳辞書の自動編纂」）を受けている。

参考文献

1) K. Knight and J. Graehl. Machine transliteration. *Computational Linguistics*, Vol. 24, No. 4, pp. 599–612, 1998.
2) I.-H. Kang, and G.-C. Kim. English-to-Korean Transliteration using Multiple Unbounded Overlapping Phoneme Chunks. In *Proc. of COLING-2000*, pp. 418–424, 2000.
3) J.-S. Kuo, H. Li, and Y.-K. Yang. Learning transliteration lexicons from the Web. In *Proc. of COLING/ACL-2006*, pp. 1129–1136, 2006.
4) L.Jiang, M.Zhou, L.-F. Chien, and C.Nui. Named entity translation with web mining and transliteration, In *Proc. of IJCAI-07*, pp. 1629–1634, 2007.
5) J. Oh and H. Isahara. Hypothesis selection in machine transliteration: A web mining approach. In *Proc. of IJCNLP-08*, pp. 233–240, 2008.
6) W. Wentland, J. Knopp, C. Silberer, and M. Hartung. Building a multilingual lexical resource for named entity disambiguation, translation and transliteration. In *Proc. of LREC-08*, 2008.
7) 国立国語研究所. 外来語の形成とその教育. 大蔵省印刷局, 1990.