Lattice Path Edit Distance: A Romanization-aware Edit Distance for Extracting Misspelling-Correction Pairs from Japanese Search Query Logs

Nobuhiro Kaji
LY Corporation*
nkaji@lycorp.co.jp

Abstract

Edit distance has been successfully used to extract training data, *i.e.*, misspelling-correction pairs, of spelling correction models from search query logs in languages including English. However, the success does not readily apply to Japanese, where misspellings are often dissimilar to correct spellings due to the romanization-based input methods. To address this problem, we introduce *lattice path edit distance*, which utilizes romanization lattices to efficiently consider all possible romanized forms of input strings. Empirical experiments using Japanese search query logs demonstrated that the lattice path edit distance outperformed baseline methods including the standard edit distance combined with an existing transliterator and morphological analyzer. A training data collection pipeline that uses the lattice path edit distance has been deployed in production at our search engine for over a year.

1 Introduction

Edit distance (Levenshtein, 1966; Damerau, 1964) is indispensable for query spelling error correction, which is an essential component in modern search engines. Training spelling correction models requires a huge amount of training data, *i.e.*, misspelling-correction pairs, but creating such data by hand is costly. To address this problem, previous studies have automatically extracted misspelling-correction pairs from query logs by using edit distance between two queries as a clue (Zhang et al., 2006; Hasan et al., 2015; Zhou et al., 2019; Kuznetsov and Urdiales, 2021).

In Japanese, however, edit distance is not effective for detecting misspelling-correction pairs due to the unique input methods (IMs) (*c.f.*, Section 2.1). In typical Japanese IMs, users first enter romanized text and then convert it into Japanese characters. In the latter step, a typo can produce misspellings that are dissimilar to the correct spellings (Table 1). Note that the misspellings and correct spellings in Table 1 do not share many characters in common. Such misspellings are abundant in query logs, but difficult to detect by using edit distance.

Misspelling	Correct spelling
きめつのやいば (*kimetunoyaiba*)	鬼滅の刃 ‘*Demon Slayer*[1]’ (*kimetunoyaiba*)
いんさt (*insat*)	印刷 ‘*printing*’ (*insatu*)

Table 1: Difficult-to-detect misspellings and their correct spellings. The romanized forms that are entered by IMs are presented in the parentheses. English translations are assigned to the correct spellings.

To deal with such difficult-to-detect misspellings, we explore using romanized forms that are entered by IMs. As shown in Table 1, even if the correct spellings and misspellings are dissimilar, their romanized forms that are entered by IMs are often similar or even identical[2]. This observation reasonably leads us to the idea of computing edit distance between romanized forms rather than surface strings.

Using romanized forms is simple in theory but in actuality it is difficult to implement in Japanese (*c.f.*, Section 2.2). Estimating romanized forms that are entered by IMs from surface strings is challenging because it requires sense disambiguation and Japanese has multiple romanization systems.

To bypass the difficulty in estimating romanized forms, we introduce *lattice path edit distance*, an edit distance that uses all possible romanized forms of input strings rather than uniquely determined romanized forms. Because Japanese characters generally have many possible romanized forms, it is inefficient to simply consider every possible com-

*Yahoo Japan Corporation at the time of submission.

[1] https://en.wikipedia.org/wiki/Demon_Slayer:_Kimetsu_no_Yaiba

[2] Different spellings can have the same romanized forms in Japanese.

Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: Industry Track, pages 233–242
December 6-10, 2023 ©2023 Association for Computational Linguistics

Intended Text	**IM Input**	**Candidate List**	**IM Output**
鬼滅の刃 ‘*Demon Slayer*’ (*kimetsunoyaiba*)	*kimetunoyaiba*	鬼滅の刃 毀滅の刃 きめつのやいば	きめつのやいば
印刷 ‘*printing*’ (*insatu*)	*insat*	いんさt 蔭佐t 院さt	いんさt

Table 2: How difficult-to-detect misspellings are caused by Japanese IMs. The first column shows the text the users intended to enter and its correct romanized form. The second column shows the (possibly misspelled) romanized text that the users actually entered, and the third column shows the automatically generated candidate list. The last column is the candidate selected by the user.

bination. This problem is addressed by a dynamic programming (DP) algorithm that makes use of the lattice structure.

Experiments using Japanese search query logs compared the qualities of the misspelling-correction pairs extracted by using five types of edit distances, one of which is the lattice path edit distance. The results demonstrated that the lattice path edit distance outperformed the others including the standard edit distance combined with an existing transliterator and morphological analyzer. It was also demonstrated that the standard edit distance and the lattice path edit distance were complementary, and their combination improved the extraction results.

2 Problems

This section provides in-depth discussions on the problems to be addressed in this work.

2.1 Ineffectiveness of edit distance

Japanese text is written using a combination of four scripts: the Latin alphabet, Chinese characters, and two Japanese syllabic scripts (*i.e.*, *hiragana* and *katakana*). Because there exist thousands of distinct Chinese characters, it is not straightforward to enter Japanese text from keyboards, in contrast to English.

To enter text from keyboards, certain IMs are commonly used in Japanese (Tokunaga et al., 2011; Maeta and Mori, 2012; Okuno and Mori, 2012). In typical Japanese IMs, users first enter romanized Japanese text using a keyboard and then convert it into Japanese characters, which is composed of the four scripts. Because many-to-many mapping generally exists between Japanese text and its romanized form, the conversion is done by manually selecting the appropriate one from automatically-generated candidates.

The IMs in Japanese often cause misspellings that are difficult to detect by using edit distance (Table 2). In the first example, the user entered the romanized text ‘*kimetunoyaiba*’ with the intention of writing ‘鬼滅の刃 (*Demon Slayer*).’ Although the resulting candidate list includes the intended one, s/he inadvertently selected the wrong candidate ‘きめつのやいば’, which accidentally has the same romanized form as ‘鬼滅の刃 (*Demon Slayer*).’ In the second example, the user entered the wrong romanized text ‘*insat*’ (the correct romanized text is ‘*insatu*’). As a result, the candidate list no longer includes the intended one. Nevertheless, s/he accidentally selected the wrong candidate ‘いんさt’. In both examples, the misspellings, (*i.e.*, **IM Output**), and correct spellings, (*i.e.*, **Intended Text**), are dissimilar and do not share many characters in common. Unfortunately, many search engine users do not always type precisely, and such misspellings are abundant in search query logs.

Japanese IMs in which users enter *Katakana* forms rather than romanized forms are also popular. Although this work exclusively explores romanized forms because they are familiar to both native and non-native Japanese readers, the proposed lattice path edit distance can also be applied to *Katakana* forms straightforwardly.

2.2 Difficulty of estimating romanized forms

To deal with misspellings that are difficult to detect, this paper explores using romanized forms that are entered by IMs. As seen from the first two columns in Table 2, even if correct spellings and misspellings are dissimilar, their romanized forms are often similar or even identical: ‘鬼滅の刃’ and ‘きめつのやいば’ have exactly the same romanized forms ‘*kimetunoyaiba*,’ while ‘印刷’ and ‘いんさt’ have similar romanized forms, ‘*insatu*’ and

'*insat*.' Thus, it is reasonable to compute the edit distance of romanized forms rather than surface strings.

Using romanized forms is simple in theory but difficult to realize in Japanese because the following two ambiguities make it difficult to estimate romanized forms from surface strings.

Sense ambiguity Most Chinese characters have multiple senses, and each sense has a different pronunciation. Consequently, the characters can be romanized in different ways depending on the sense they represent in the context. For example, the character '行' is romanized as '*i*' when it means '*go*' and '*okona*' when it means '*do*' (Suzuki et al., 2009). This indicates that estimating romanized forms requires sense disambiguation, which is a difficult task.

Transliteration ambiguity The Japanese language has multiple romanization systems (*i.e.*, ways of transliterating Japanese characters into Latin alphabet) such as Hepburn romanization. Therefore, even characters other than Chinese characters can be romanized in multiple ways. For example, the *Hiragana* character 'し' can be romanized as either '*si*', '*shi*' or '*ci*' depending on the romanization system. Because we are unable to access the specific romanization systems used by the users, it is impossible to predict the exact romanized forms from surface strings.

Although existing tools such as transliterators and morphological analyzers can be used to address those ambiguities, they are not sufficient in practice. The experiment in Section 4 investigates baseline methods that make use of these tools, and the results demonstrate that they are suboptimal.

It is worth noting that improving edit distance by using representations of pronunciations, such as romanized forms, has been common in previous studies (Jurafsky and Martin, 2023). As a notable example, the GNU Aspell algorithm (Atkinson, 2019) uses simple rules (Philips, 1990) to convert input strings into representations of pronunciations, between which edit distance is computed. However, these studies primarily focused on English. Such approaches are not applicable to Japanese.

3 Lattice Path Edit Distance

This section introduces the proposed lattice path edit distance, which is aware of romanized forms of input strings.

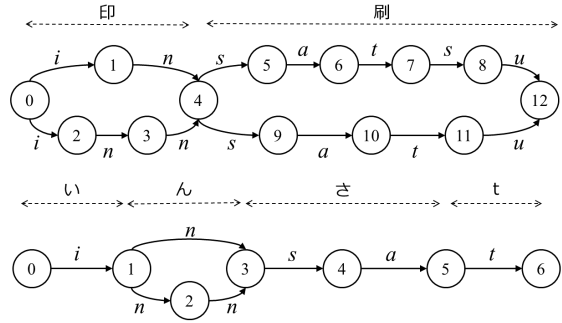

Figure 1: Romanization lattices for '印刷' (top) and 'いんさt' (bottom).

3.1 Formulation

To bypass the difficulty of estimating romanized forms, we explore a new edit distance that uses all possible romanized forms of input strings rather than uniquely determined romanized forms. Specifically, the new edit distance is defined as the minimum edit distance between all possible romanized forms of input strings:

$$d(x, y) = \min_{a \in R_x, b \in R_y} d_{\text{base}}(a, b), \quad (1)$$

where x and y are input strings, R_x and R_y represent sets of all possible romanized forms of x and y, respectively. We presume that the romanized forms are obtained by using a romanization dictionary. The function $d_{\text{base}}(a, b)$ is referred to as the *base edit distance*. The base edit distance is assumed to be the Levenshtein distance (Levenshtein, 1966) in the following discussion but can be extended straightforwardly to the Damerau-Levenshtein distance (Appendix A).

The brute-force computation of $d(x, y)$ is inefficient when $|R_x|$ and $|R_y|$ are large. To avoid this problem, we use *romanization lattices*, which implicitly encode all possible romanized forms of strings (Figure 1), as representations of R_x and R_y. In the romanization lattice, all edges are labeled with a single Latin letter, and every path from the start to the end node represents one romanized form. In what follows, we presume that the nodes are indexed by integers (starting from zero) in the topological order.

3.2 Distance computation

This subsection presents a DP algorithm for computing $d(x, y)$, which is hereafter referred to as *lattice path edit distance*. The algorithm is based

Algorithm 1 Distance computation

```
1: for i ← 0 to N do
2:     for j ← 0 to M do
3:         Compute D[i][j] by using Equation (3)
4:     end for
5: end for
6: return D[N][M]
```

on a DP table D defined as below:

$$D[i][j] = \min_{a \in R_x^i, b \in R_y^j} d(a, b), \qquad (2)$$

where R_x^i is a set of romanized forms in R_x ending with the node i. In Figure 1 (top), for example, $R^4_{\text{'印刷'}} = \{in, inn\}$, $R^{10}_{\text{'印刷'}} = \{insa, innsa\}$, etc. Note that we have $d(x, y) = D[N][M]$, where N and M are the indices of the end nodes of R_x and R_y, respectively.

$D[N][M]$, or equivalently $d(x, y)$, can be efficiently computed by using the following formula (Algorithm 1):

$$D[i][j] = \min \begin{cases} \min_{l \in P_y(j)} D[i][l] + 1, \\ \min_{k \in P_x(i)} D[k][j] + 1, \\ \min_{k \in P_x(i), l \in P_y(j)} D[k][l] + \delta_{\pi^x_{k:i}, \pi^y_{l:j}}, \end{cases} \qquad (3)$$

where $P_x(i)$ denotes a set of direct predecessors of the node i in R_x, and $\pi^x_{k:i}$ denotes the label (*i.e.*, Latin letter) of the edge going from the node k to i in R_x. $P_y(j)$ and $\pi^y_{l:j}$ are defined similarly. $\delta_{\cdot,\cdot}$ is the Kronecker delta. See Appendix B for relations to existing algorithms.

3.3 Neighborhood checking

Algorithm 1 can be accelerated by reducing the search space if it suffices to check whether $D[N][M]$ is equal to or less than a pre-defined threshold θ. Because the costs of the edit operations are non-negative, $D[i][j]$ is a monotonically increasing function of i and j. Therefore, once $D[i][j]$ exceeds θ, we can safely remove $D[i][j]$ from consideration to avoid unnecessary computation.

This results in Algorithm 2. The algorithm visits the node pair (i, j) that satisfies $D[i][j] \leq \theta$ in the topological order by using the priority queue Q, and updates D by using the following equations for all $k \in S_x(i)$ and $l \in S_y(j)$

Algorithm 2 Neighborhood checking

```
1: D[0][0] ← 0
2: Add (0, 0) to Q
3: while Q is not empty do
4:     (i, j) ← Q.pop()
5:     if θ < D[i][j] then
6:         continue
7:     end if
8:     if (i, j) = (N, M) then
9:         return TRUE
10:    end if
11:    Update D using Equations (4-6)
12:    Add updated node pairs to Q
13: end while
14: return FALSE
```

$$D[i][l] = \min\{D[i][l], D[i][j] + 1\}, \qquad (4)$$

$$D[k][j] = \min\{D[k][j], D[i][j] + 1\}, \qquad (5)$$

$$D[k][l] = \min\{D[k][l], D[i][j] + \delta_{\pi^x_{i:k}, \pi^y_{j:l}}\}, \qquad (6)$$

where $S_x(i)$ denotes a set of direct successors of i in R_x. The algorithm successfully terminates by returning TRUE (line 9) when the node pair (N, M) is visited and $D[N][M] \leq \theta$ is satisfied.

4 Experiment

Empirical experiments were conducted using Japanese search query logs to investigate the quality of the misspelling-correction pairs extracted by using the lattice path edit distance.

4.1 Task setting

When designing the experimental task, we considered a hypothetical use case in which edit distance is used to extract misspelling-correction pairs from query logs, with reference to (Hasan et al., 2015; Kuznetsov and Urdiales, 2021). Specifically, we considered two consecutive queries issued by the same users are extracted as misspelling-correction pairs, if the following conditions are all satisfied:

1. The two queries are issued within 60 seconds.
2. The number of unique users who issued the second query is more than five times larger than the first query.
3. A set of terms in one query does not subsume the other.

	Levenshtein		Damerau-Levenshtein	
	Precision	Recall	Precision	Recall
Base edit distance	70.5	43.2	71.2	44.7
Phonological edit distance	80.8	29.2	81.5	30.6
Romanization (kakasi)	88.2	60.3	88.3	61.2
Romanization (mecab+kakasi)	88.1	59.6	88.3	60.5
Lattice path edit distance	87.8	71.1	88.0	72.1

Table 3: Precision and recall for the misspelling-correction pair detection task.

4. The edit distance between the two queries is equal to or less than a predefined threshold.

We constructed an evaluation dataset that simulates such a use case. A total of 29,359 query pairs that satisfy the first three conditions described above were collected from the query logs of a Japanese Web search engine. The collected query pairs were then manually annotated by experts as to whether or not they are true misspelling-correction pairs. As the result, 1743 out of 29,359 were annotated as true misspelling-correction pairs.

Using this dataset, the goodness of edit distance was measured on the basis of misspelling-correction pair detection task in which two queries are regarded as true misspelling-correction pairs when their edit distance is equal to or less a predefined threshold. The result of this detection task represents the quality of the extracted misspelling-correction pairs in the abovementioned use case. The threshold was set to one considering the importance of the precision of the extraction results as training data.

The romanization dictionary was constructed from the UniDic dictionary (Version 3.1.0)[3].

4.2 Baseline methods

Both the Levenshtein and Damerau-Levenshtein distances were tested as the base edit distance. In both cases, the following baseline methods were implemented for comparison.

Base edit distance The base edit distance of the lattice path edit distance (*i.e.*, either Levenshtein or Damerau-Levenshtein distane) is used as is.

Phonological edit distance This method also uses the base edit distance, but allows only edit operations of phonograms (*i.e.*, Latin alphabet, *Hiragana*, and *Katakana* characters) to avoid excessive edits. This baseline is inspired by previous studies that successfully used edit distance for extracting Japanese spelling variants with a focus on *Katakana* words (Masuyama et al., 2004).

Romanization (kakasi) The input strings are deterministically romanized by using kakasi (version 2.2.1)[4], a transliteration library for Japanese, and then their base edit distance is computed.

Romanization (mecab+kakasi) The input strings are first processed by a Japanese morphological analyzer, mecab (version 0.996)[5], to estimate pronunciations (*i.e.*, *Katakana* forms), and then the results are romanized by kakasi. This method intends to take the advantage of the existent morphological analyzer to accurately estimate pronunciations.

Hereafter, the first two baseline methods are collectively referred to as *surface-level distances*, while the latter two and the lattice path edit distance are referred to as *romanization-aware distances*.

4.3 Results

Main results Table 3 shows the results of the misspelling-correction pair detection in the two settings (*i.e.*, Levenshtein and Damerau-Levenshtein distances used as the base edit distance). As shown, romanization-aware distances outperformed the surface-level ones, which demonstrates the importance of using romanized forms to detect misspellings in Japanese. In addition, the lattice path edit distance increased the recall by over 10 points at a negligible cost of precision, compared with the two romanization-aware baselines. This result suggests that methods based on uniquely determined romanized forms are subpotimal and it is a better strategy to consider all possible romanized forms. The existing transliterator and morphological analyzer (*i.e.*, kakasi and mecab) may not have been effective for the following reasons. First, they can

[3] https://unidic.ninjal.ac.jp

[4] https://github.com/miurahr/pykakasi

[5] https://taku910.github.io/mecab

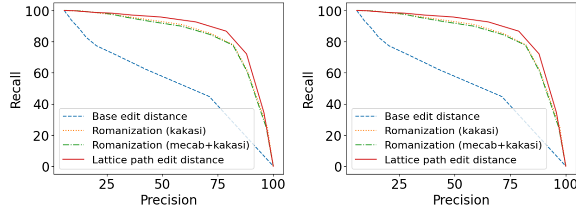

Figure 2: Precision-recall curves. **Left:** Levenshtein distance. **Right:** Damerau-Levenshtein distance.

address the sense ambiguity at least in principle but are incapable of addressing the transliteration ambiguity (*c.f.*, Section 2.2). Second, they are designed to process clean text rather than misspelled text.

Figures 2 illustrates the precision-recall curves with changing the threshold value. We can confirm that the lattice path edit distance was able to achieve better precision-recall trade-off compared with the other methods regardless of the choice of base edit distance.

Distance combination Thus far, the romanization-aware distances have demonstrated greater effectiveness over the surface-level ones. However, there exist misspellings that the surface-level distances can detect more effectively than romanization-aware distances (Appendix C), and therefore the two types of edit distances are considered complementary.

Thus, we investigate combining the lattice path edit distance and the surface-level distance by using a simple combination method of taking the minimum of two distances (Table 4). The result demonstrated that the combination with the phonological base distance improved the recall at the cost of a small decrease in precision. This simple combination method can be a good starting point to make the best use of the surface-level and romanization-aware distances. Meanwhile, the result of the combination with the base edit distance was not very promising. It achieved the highest recall, but the precision decreased significantly. This suggests that it remains challenging to achieve this level of recall without sacrificing precision. Future work should include exploring more sophisticated approaches.

Time efficiency A comparison between Algorithms 1 and 2 shows that Algorithm 2 achieved a 16 times speed-up when the threshold was set to one (Appendix D). This demonstrates the practical usefulness of Algorithm 2 as only checking neighbors, rather than computing the exact distance, is usually sufficient in practical use cases.

	Precision	Recall
Lattice path edit distance	88.0	72.1
+Base edit distance	79.9	81.0
+Phonological edit distance	87.4	76.7

Table 4: Results for the combination of the lattice path edit distance and the surface-level distances.

4.4 Example

Table 5 presents example misspellings that the `kakasi` baseline failed to detect but the lattice path edit distance succeeded. In the first example, not only '*aitikekorona*' but '*aitiiekorona*' are plausible romanized forms of '愛知家コロナ,' which is a meaningless string. In the second example, not only '*chadougu*' but '*tyadougu*' are correct romanized forms of '茶道具 (*tea-things*)' due to the transliteration ambiguity. In both cases, the only correct romanized forms do not exist. The `kakasi` baseline accidentally preferred the romanized forms that result in larger edit distance, thus failing to detect the two misspellings. Such detection errors are considered inevitable. On the other hand, the proposed lattice path edit distance successfully detected the two misspellings since it is able to consider all possible romanized forms.

4.5 Discussion

The threshold on the lattice path edit distance was set to one in the experiment. While we consider this threshold value is reasonable in practice, the recall in Table 3 suggests that the lattice path edit distance is still larger than one in a non-negligible percentage of cases.

One may suspect that most of those misspelling-correction pairs are long queries. To test this hypothesis, we investigated the query length[6] of the 1743 misspelling-correction pairs used in the experiment. The 1743 pairs were divided into two groups: the lattice path edit distance is less than or equal to one in one group, while more than one in the other. The result demonstrated that the average query length in the latter group is only slightly longer than the former (12.79 vs. 13.46), suggesting that the query length cannot sufficiently explain the difference of the lattice path edit distance.

[6]Specifically, the sum of the lengths of the two queries.

Misspelling	Correct spelling
愛知家コロナ (*aitikekorona*)	愛知県コロナ '*Aichi prefecture Covid*' (*aitikenkorona*)
chadougu	茶道具 '*tea-things*' (*chadougu*)

Table 5: Misspellings that the `kakasi` baseline failed to detect but the lattice path edit distance succeeded. The romanized forms that achieve the minimum edit distance are presented in the parentheses. In the second example, the user entered the correct romanized form but forgot to activate IM. This results in the misspelling '*chadougu*.'

Our manual investigation revealed that only prefix of the intended query is often entered when the lattice path edit distance is larger than one, *e.g.*, 'いんたーんし' and 'インターンシップ (*internship*).' Detection methods based on edit distance is not designed to handle such misspellings. Different approaches like query auto-completion (Kim, 2019) are considered desirable.

5 Related Work

Previous studies successfully used the Levenshtein distance to extract misspelling-correction pairs from GitHub's commit logs (Hagiwara and Mita, 2020) and Wikipedia's revision history (Tanaka et al., 2020). Although this may seem to contradict with our findings, these successes are reasonable because the text domains explored in those studies are substantially different from search query logs (Appendix E).

Some studies (Suzuki et al., 2009; Saito et al., 2017) investigated edit distance between representations of pronunciations as a clue for spelling variant extraction. Suzuki et al. (2009) deterministically converted input strings into romanized forms and then computed the edit distance between them. Their approach is essentially the same as the romanization baseline explored in our experiment.

Synthetically generating misspellings from correct spellings, rather than extracting misspelling-correction pairs from some linguistic resources (*e.g.*, query logs), is another common approach to addressing the scarcity of training data for spelling error correction. It is interesting to see that this line of attempts has also emphasized the importance of considering pronunciations in parallel to this work (Wang et al., 2018; Kakkar et al., 2023).

As discussed in Appendix B, the lattice path edit distance is closely related to finite-state automata. The contributions of this work compared to the previous studies on finite-state automata are two folds. First, we explored a new application of finite-state automata to the misspelling-correction pair detection in Japanese. Second, we introduced a simplified variant of the Mohri's algorithm (2003) that is tailored to the new application setting, where the input automata have lattice structures.

6 Conclusion and Future Work

We have introduced lattice path edit distance, a romanization-aware edit distance, with a focus on extracting misspelling-correction pairs from Japanese search query logs. A DP algorithm and its faster variant were proposed for the efficient computation of the lattice path edit distance. The empirical results demonstrated that the lattice path edit distance outperformed the standard edit distance even if an existing transliterator and morphological analyzer were employed together.

Although this work focused on Japanese, similar problems are considered to arise in other Asian languages that have their own IMs. For example, the Chinese language also has its own romanization system, *pinyin*, and language-specific IMs based on it. Application of the lattice path edit distance to such languages is a future direction worth exploring.

Acknowledgement

The author would like to thank his colleagues working on query spelling correction with him. His thanks also go to the anonymous reviewers, who provided insightful comments.

References

Kevin Atkinson. 2019. GNU Aspell. http://aspell.net.

Fred J. Damerau. 1964. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176.

Masato Hagiwara and Masato Mita. 2020. GitHub typo corpus: A large-scale multilingual dataset of misspellings and grammatical errors. In *Proceedings*

of the Twelfth Language Resources and Evaluation Conference, pages 6761–6768, Marseille, France. European Language Resources Association.

Saša Hasan, Carmen Heger, and Saab Mansour. 2015. Spelling correction of user search queries through statistical machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 451–460, Lisbon, Portugal. Association for Computational Linguistics.

Daniel Jurafsky and James H. Martin. 2023. *Speech and Language Processing*, third edition, chapter Spelling Correction and the Noisy Channel (`http://web.stanford.edu/~jurafsky/slp3/B.pdf`).

Vishal Kakkar, Chinmay Sharma, Madhura Pande, and Surender Kumar. 2023. Search query spell correction with weak supervision in E-commerce. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 5: Industry Track)*, pages 687–694, Toronto, Canada. Association for Computational Linguistics.

Gyuwan Kim. 2019. Subword language model for query auto-completion. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5022–5032, Hong Kong, China. Association for Computational Linguistics.

Alex Kuznetsov and Hector Urdiales. 2021. Spelling correction with denoising transformer. arXiv:2105.05977.

Vladimir I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics-Doklady*, 10(8):707–710.

Hirokuni Maeta and Shinsuke Mori. 2012. Statistical input method based on a phrase class n-gram model. In *Proceedings of the Second Workshop on Advances in Text Input Methods*, pages 1–14, Mumbai, India. The COLING 2012 Organizing Committee.

Takeshi Masuyama, Satoshi Sekine, and Hiroshi Nakagawa. 2004. Automatic construction of Japanese KATAKANA variant list from large corpus. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, pages 1214–1219, Geneva, Switzerland. COLING.

Mehryar Mohri. 2003. Edit-distance of weighted automata: General definitions and algorithms. *International Journal of Foundations of Computer Science*, 14(6):957–982.

Yoh Okuno and Shinsuke Mori. 2012. An ensemble model of word-based and character-based models for Japanese and Chinese input method. In *Proceedings of the Second Workshop on Advances in Text Input Methods*, pages 15–28, Mumbai, India. The COLING 2012 Organizing Committee.

Lawrence Philips. 1990. Hanging on the metaphone. *Computer Language Magazine*, 7(12):39–44.

Itsumi Saito, Kyosuke Nishida, Kugatsu Sadamitsu, Kuniko Saito, and Junji Tomita. 2017. Automatically extracting variant-normalization pairs for Japanese text normalization. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 937–946, Taipei, Taiwan. Asian Federation of Natural Language Processing.

Hisami Suzuki, Xiao Li, and Jianfeng Gao. 2009. Discovery of term variation in Japanese web search queries. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 1484–1492, Singapore. Association for Computational Linguistics.

Yu Tanaka, Yugo Murawaki, Daisuke Kawahara, and Sadao Kurohashi. 2020. Building a Japanese typo dataset from Wikipedia's revision history. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: Student Research Workshop*, pages 230–236, Online. Association for Computational Linguistics.

Hiroyuki Tokunaga, Daisuke Okanohara, and Shinsuke Mori. 2011. Discriminative method for Japanese kana-kanji input method. In *Proceedings of the Workshop on Advances in Text Input Methods (WTIM 2011)*, pages 10–18, Chiang Mai, Thailand. Asian Federation of Natural Language Processing.

Robert A. Wagner and Michael J. Fischer. 1974. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173.

Dingmin Wang, Yan Song, Jing Li, Jialong Han, and Haisong Zhang. 2018. A hybrid approach to automatic corpus generation for Chinese spelling check. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2517–2527, Brussels, Belgium. Association for Computational Linguistics.

Yang Zhang, Pilian He, Wei Xiang, and Mu Li. 2006. Discriminative reranking for spelling correction. In *Proceedings of the 20th Pacific Asia Conference on Language, Information and Computation*, pages 64–71, Huazhong Normal University, Wuhan, China. Tsinghua University Press.

Yingbo Zhou, Utkarsh Porwal, and Roberto Konow. 2019. Spelling correction as a foreign language. In *Proceedings of the SIGIR19 eCom workshop*.

A Extension to Damerau-Levenshtein Distance

Thus far, we have assumed that the base edit distance is the Levenshtein distance, but it is also possible to use the Damerau-Levenshtein distance (Damerau, 1964). Algorithm 1 can be extended

to the Damerau-Levenshtein distance by adding another term

$$\min_{\substack{k \in P_x(i), k' \in P_x(k) \\ l \in P_y(j), l' \in P_y(l) \\ s.t.\ \pi^x_{k':k} = \pi^y_{l:j} \wedge \pi^x_{k:i} = \pi^y_{l':l}}} D[k'][l'] + 1 \tag{7}$$

to Equation (3). A similar extension can also be made to Algorithm 2.

B Relation to Existing Algorithms

The proposed DP algorithm is closely related to existing algorithms such as the Wagner-Fischer algorithm for computing the Levenshtein distance (Wagner and Fischer., 1974). Because the proposed algorithm is reduced to the Wagner-Fischer algorithm when both romanization lattices have linear chain structures, it can be seen as an extension of the Wagner-Fischer algorithm from strings to lattices.

It is also worth considering the proposed algorithm from the viewpoint of a finite-state automaton. Note that the lattice is a special form of a finite-state automaton. Mohri (2003) argued that the minimum edit distance among strings accepted by two unweighted automata, A_1 and A_2, is equal to the shortest path distance of the weighted automaton $U = A_1 \circ T \circ A_2$, where T is an edit distance transducer and $\circ$ is the composition operation. See (Mohri, 2003) for detailed descriptions. Therefore, the lattice path distance can also be computed by the shortest path search over such an automaton.

An interesting observation here is that a bijection exists between the positions (i, j) in D and the states in U, and that $D[i][j]$ is equal to the shortest path distance to the state corresponding to (i, j) (Figure 3). Therefore, the proposed algorithm can be interpreted as performing the same shortest path search as in (Mohri, 2003) while eliminating the needs of the complex composition operations for constructing U. In this sense, the proposed algorithm is a simplified variant of Mohri's algorithm that is applicable when both A_1 and A_2 have lattice structures.

C Motivating Example for Distance Combination

Table 6 represents an example of a misspelling that is more easily detected by surface-level distances than by romanization-aware ones. In this example, the Damerau-Levenshtein distance between

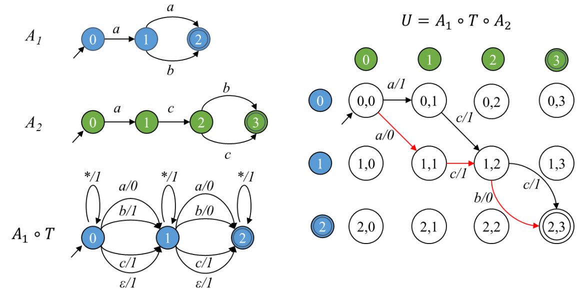

Figure 3: Mohri's algorithm (2003) for lattice-structured unweighted automata. **Left:** Lattice-structured unweighted automata, A_1 and A_2, defined over an alphabet $\{a, b, c\}$, and the weighted automaton $A_1 \circ T$. **Right:** the weighted automaton $U = A_1 \circ T \circ A_2$, which is obtained by composing (or intersecting) $A_1 \circ T$ and A_2 (only a fraction of the edges are illustrated for simplicity). Because states in U correspond to pairs of states in $A_1 \circ T$ and A_2, they are indexed by the corresponding integer pairs. The red edges represent the shortest path. Notice the similarity between the automaton U and the DP table D.

Correct spelling	**Misspelling**
マリトッツォ '*maritozzo*'	マトリッツォ
(*maritottso*)	(*matorittso*)

Table 6: Misspelling that is more easily detected by surface-level distances than by romanization-aware ones. The romanized forms are presented in the parentheses.

the surface strings is 1 because only one transposition operation is required to transform the correct spelling into the misspelling, while the distance between the romanized forms is 4. Such an example motivates us to combine surface-level and romanization-aware edit distances.

D Time Efficiency

Figure 4 compares the time in seconds required to process the evaluation data by Algorithms 1 and 2. For Algorithm 2, three threshold values (1, 2, and 3) were tested. The results revealed that Algorithm 2 attained a speed-up of up to 16.83 times compared with Algorithm 1. This demonstrates the practical usefulness of Algorithm 2 because only testing neighbors, rather than computing the exact distance, is usually sufficient in practical use cases.

E Comparison between Search Query Logs and GitHub's commit logs

Figure 5 compares the distributions of the normalized Levenshtein distances between misspellings

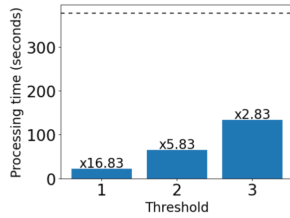

Figure 4: Processing times of Algorithms 2. The horizontal axis represents the threshold value. The dotted line represents Algorithm 1. The numbers above the bars represent the speed gains relative to Algorithm 1. All results were obtained by averaging over five independent runs.

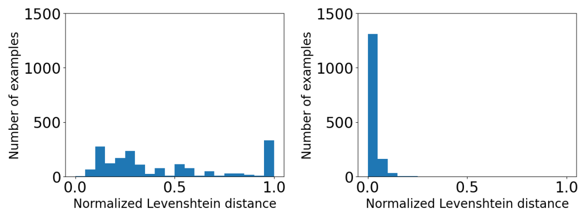

Figure 5: Distributions of the normalized Levenshtein distances between misspellings and corrections. **Left:** Search query logs. **Right:** GitHub's commit logs.

and corrections in the search query logs (*c.f.*, Section 4.1) and GitHub's commit logs[7]. As the figure shows, the types of spelling errors in the two datasets are different in nature; the misspellings and their corrections are quite similar in the commit logs but not in the search query logs. This difference is considered to be the reason that the Levenshtein distance was effective in previous studies but not in this work.

[7]The Japanese portion of GitHub Typo Corpus (version 1.0.0) was used.