Approximate Matching between a Context-Free Grammar and a Finite-State Automaton

Yo-Sub Han[1], Sang-Ki Ko[1], and Kai Salomaa[2]

[1] Department of Computer Science, Yonsei University
50, Yonsei-Ro, Seodaemun-Gu, Seoul 120-749, Republic of Korea
{emmous,narame7}@cs.yonsei.ac.kr
[2] School of Computing, Queen's University
Kingston, Ontario K7L 3N6, Canada
ksalomaa@cs.queensu.ca

Abstract. Given a context-free grammar (CFG) and a finite-state automaton (FA), we tackle the problem of computing the most similar pair of strings from two languages. We in particular consider three different gap cost models, linear, affine and concave models, that are crucial for finding a proper alignment between two bio sequences. We design efficient algorithms for computing the edit-distance between a CFG and an FA under these gap cost models. The time complexity of our algorithm for computing the linear or affine gap distance is polynomial and the time complexity for the concave gap distance is exponential.

Keywords: approximate matching, edit-distance, context-free grammars, finite-state automata.

1 Introduction

The string matching problem aims to find exact matches of a pattern w from an input text T and the approximate matching problem is to find similar occurrences of w that are within the distance k in T. Many researchers studied the approximate pattern matching problem that allows various types of mismatches [1,6,16,17,19,22]. For example, Aho and Peterson [1], and Lyon [16] introduced an $O(n^2m^3)$ algorithm for the problem of approximately matching a string of length n and a context-free language specified by a grammar of size m. They generalized Earley's algorithm [6] for parsing context-free languages and considered the edit-distance model [15] that has a unit-cost function. Myers [19] considered the variants of the problem under various gap costs such as linear, affine and concave gap costs; these gap cost models are very important to find proper alignment between two bio sequences in practices [20,21]. For the linear and affine gap costs, Myers designed $O(mn^2(n+\log m))$ algorithms and sketched an $O(m^5n^88^m)$ algorithm for the concave gap costs. His algorithm generalizes the Cocke-Younger-Kasami (CYK) algorithm [4,8,12].

The approximate matching problem is based on the edit-distance between two strings, or between a string and a language. This led researchers to examine the edit-distance between two formal languages. Mohri [18] proved that the

S. Konstantinidis (Ed.): CIAA 2013, LNCS 7982, pp. 146–157, 2013.

edit-distance between two context-free languages is undecidable and provided a quadratic algorithm for two regular languages. Choffrut and Pighizzini [3] considered the relative edit-distance between languages and defined the reflexivity of binary relations based on the definition. Recently, the authors [9] studied the problem of computing the Levenshtein distance [15] between a context-free language and a regular language given by a pushdown automaton (PDA) P and a finite-state automaton (FA) A, respectively. We constructed an *alignment PDA* that computes all possible alignments between $L(A)$ and $L(P)$, converted the alignment PDA into a CFG and found the optimal alignment from the resulting grammar. The overall runtime is $O((n_1 n_2) \cdot 2^{(m_1 m_2)^2})$, where m_1 is the number of states of A, m_2 is the number of states of P, n_1 is the number of transitions of A and n_2 is the number of transitions of P. We also showed that we can compute the optimal edit-distance value in $O((m_1 m_2)^4 \cdot (n_1 n_2))$ time. Note that the conversion from a PDA of size n into a CFG takes $O(n^3)$ time and the size of the resulting grammar is at most $O(n^3)$ [10]. If a context-free language is given by a CFG instead of a PDA, then we need to construct a PDA for an input CFG before computing the alignment PDA. This motivates us to design algorithms that compute the edit-distance between a CFG and an FA without constructing a PDA, and extend this problem to the approximate matching between a CFG and an FA. In other words, we calculate the minimum edit-distance and the optimal alignment between the most similar pair of strings generated by a CFG and an FA, respectively. We introduce algorithms for computing the various gap distances and the optimal alignments between a CFG and an FA. While the previous research [9,11,14,18] on computing the edit-distance of formal languages rely on variants of the Cartesian product, the proposed algorithms are based on the dynamic programming approach that are generalized from the CYK algorithm. Given an FA of size n and a CFG of size m, our algorithms compute linear and affine gap distances in $O(mn^2(n+\log m))$ time. Furthermore, the worst-case time complexity of our algorithm for computing the concave gap distance is $O(mn^8 8^m)$.

In Section 2, we give a basic notations and terminology used here. We present the definitions for the edit-distance model in Section 3. In Section 4, we introduce a dynamic programming algorithm for computing the edit-distance between a CFG and an FA. The following two sections extend the algorithm to the problems of computing affine and concave gap distance.

2 Preliminaries

Let Σ denote a finite alphabet of characters and Σ^* denote the set of all strings over Σ. The size $|\Sigma|$ of Σ is the number of characters in Σ. A language over Σ is any subset of Σ^*. Given a set X, 2^X denotes the power set of X.

The symbol $\emptyset$ denotes the empty language and the character λ denotes the null string. A finite-state automaton (FA) A is specified by a tuple $(Q, \Sigma, \delta, s, F)$, where Q is a finite set of states, Σ is an input alphabet, $\delta : Q \times \Sigma \to 2^Q$ is a multi-valued transition function, $s \in Q$ is the start state and $F \subseteq Q$ is a set of final states. If F consists of a single state f, we use f instead of $\{f\}$ for simplicity.

For a transition $q \in \delta(p, a)$ in A, we say that p has an *out-transition* and q has an *in-transition*. Furthermore, p is a *source state* of q and q is a *target state* of p. The transition function δ can be extended to a function $Q \times \Sigma^* \to 2^Q$ that reflects sequences of inputs. A string x over Σ is accepted by A if there is a labeled path from s to a state in F such that this path spells out the string x, namely, $\delta(s, x) \cap F \neq \emptyset$. The language $L(A)$ of an FA A is the set of all strings that are spelled out by paths from s to a final state in F.

A context-free grammar (CFG) G is specified by a tuple $G = (V, \Sigma, R, S)$, where V is a set of variables, $R \subseteq V \times (V \cup \Sigma)^*$ is a finite set of productions and $S \in V$ is the start symbol. Let $\alpha A \beta$ be a string over $V \cup \Sigma$ with A a variable and $A \to \gamma$ be a production of G. Then, we say that $\alpha A \beta \Rightarrow \alpha \gamma \beta$. The reflexive, transitive closure of $\Rightarrow$ is $\overset{*}{\Rightarrow}$. Then the context-free language defined by G is $L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$.

A CFG is in Chomsky normal form (CNF) if all of its production rules are of the form: $A \to BC$ or $A \to a$, where $A, B, C \in V$ and $a \in \Sigma$. Note that every context-free grammar can be converted into the CNF grammar with the size of $O(P^2)$ where P is the size of the original grammar. We consider the *pseudo-CNF* grammars that consist of the rules of the form $A \to BD$ or $A \to B$ or $A \to a$ where $A, B, C \in V$ and $a \in \Sigma$. We can transform every grammar into the pseudo-CNF grammar whose size is still $O(P)$ [19].

For more details on automata theory, we refer the reader to the books [10,23].

3 Edit-Distance

The edit-distance between two strings x and y is the smallest number of operations that transform x to y. People consider different edit operations depending on the applications. We consider three basic operations, insertion, deletion and substitution for simplicity. Given an alphabet Σ, let

$$\Omega = \{(a \to b) \mid a, b \in \Sigma \cup \{\lambda\}\}$$

be a set of edit operations. Namely, Ω is an alphabet of all edit operations for *deletions* $(a \to \lambda)$, *insertions* $(\lambda \to a)$ and *substitutions* $(a \to b)$. We call a string $\omega \in \Omega^*$ an *edit string* [11] or an *alignment* [18].

Let h be the morphism from Ω^* into $\Sigma^* \times \Sigma^*$ defined by setting

$$h((a_1 \to b_1) \cdots (a_n \to b_n)) = (a_1 \cdots a_n, b_1 \cdots b_n).$$

For example, a string $\omega = (a \to \lambda)(b \to b)(\lambda \to c)(c \to c)$ over Ω is an alignment of abc and bcc, and $h(\omega) = (abc, bcc)$. Thus, from an alignment ω of two strings x and y, we can retrieve x and y using h: $h(\omega) = (x, y)$.

Definition 1. *An edit string ω is a sequence of edit-operations transforming a string x into a string y if and only if $h(\omega) = (x, y)$.*

We associate a non-negative edit cost $c(\omega)$ to each edit operation $\omega \in \Omega$ where c is a function $\Omega \to \mathbb{R}_+$. We can extend the function to the cost $c(\omega)$ of an

alignment $\omega = \omega_1 \cdots \omega_n$ as follows:

$$c(\omega) = \sum_{i=1}^{n} c(\omega_i).$$

Definition 2. *The edit-distance $d(x, y)$ of two strings x and y over Σ is the minimal cost of an alignment ω between x and y:*

$$d(x, y) = \min\{c(\omega) \mid h(\omega) = (x, y)\}.$$

We say that ω is optimal if $d(x, y) = c(\omega)$.

We can extend the edit-distance definition to languages.

Definition 3. *The edit-distance $d(L, R)$ between two languages $L, R \subseteq \Sigma^*$ is the minimum edit-distance of two strings, one is from L and the other is from R:*

$$d(L, R) = \min\{d(x, y) \mid x \in L \text{ and } y \in R\}.$$

The edit-distance in Definition 3 is the distance between the closest pair of strings from L and R under the considered edit operations. In other words, the most similar pair of strings defines the edit-distance between L and R.

4 Algorithm

We compute the edit-distance between a CFG and an FA. We assume that an input CFG $G = (V, \Sigma, R, S)$ is in pseudo-CNF and an input FA $M = (Q, \Sigma, \delta, s, F)$ has no λ-production. We use a pseudo-CNF (instead of CNF) because an arbitrary grammar can be converted to a pseudo-CNF grammar with only constant increase in size. First, we define $\mathcal{C}(A, q, p)$ to be the minimum edit-distance between one string v derivable from a variable A and a string w that spells out a computation of M from q to p. We can compute the edit-distance between $L(G)$ and $L(M)$ by computing $\mathcal{C}$-values for all $A \in V$ and $q, p \in Q$. We formally define it as follows:

$$\mathcal{C}(A, q, p) = \min\{d(v, w) \mid v \in L(G_A) \text{ and } w \in L(M_{q,p})\},$$

where $G_A = (V, \Sigma, R, A)$ and $M_{q,p} = (Q, \Sigma, \delta, q, \{p\})$. Then, $\min\{\mathcal{C}(S, s, f) \mid f \in F\}$ is the edit-distance between $L(G)$ and $L(M)$. In Theorem 3, we provide a recurrence for computing the $\mathcal{C}$-values. For this purpose we first need to establish some preliminary properties and introduce notation.

First we establish the unsurprising property that among the strings $w \in L(M_{q,p})$ that take state q to state p, the string that minimizes the distance to an individual alphabet symbol uses a computation from q to p that does not repeat any loop. For states q and p, we define $L_{\text{one-cyclic}}(q, p)$ to consist of those strings w such that M has a computation on w from q to p that does not visit any state more than twice.

Lemma 1. *For any states of M, $q, p \in Q$ and $a \in \Sigma$,*

$$d(a, L(M_{q,p})) = d(a, L_{\text{one-cyclic}}(q, p)).$$

Note that, when the cost function is allowed to be arbitrary, a property analogous to Lemma 1 would not hold for strings that correspond to an acyclic computation of M from state q to p. If any string corresponding to an acyclic computation does not contain occurrences of the symbol a and the cost of deleting a is considerably larger than the costs of insertions of any other symbol, it is possible that the distance of a and $L(M_{q,p})$ cannot be minimized by a string that would not repeat any state in the computation from q to p.

Now corresponding to a variable $A \in V$ and states $q, p \in Q$ of M, we define the following sets:

(i) $X(A, q, p) = \{\mathcal{C}(B, q, r) + \mathcal{C}(D, r, p) \mid r \in Q,\ A \to BD \in R\}$.
(ii) $Y(A, q, p) = \{\mathcal{C}(B, q, p) \mid A \to B \in R\}$.
(iii) $Z(A, q, p) = \{d(a, L_{\text{one-cyclic}}(q, p)) \mid A \to a \in R,\ a \in \Sigma\}$.

Theorem 1. *For all $A \in V$ and $q, p \in Q$,*

$$\mathcal{C}(A, q, p) = \min[X(A, q, p) \cup Y(A, q, p) \cup Z(A, q, p)]. \tag{1}$$

Note that Equation (1) in Theorem 1 is an essential recurrence equation for computing $d(L(G), L(M))$ in bottom-up dynamic programming approach. First, we compute $\mathcal{C}(A, q, p)$ where the distance between two states q and p is 0. For convenience, we define the distance between two states q and p in the FA as the minimum number of transitions required to reach p from q and denote it by $\mathfrak{d}(p, q)$. For the basis, we start from when $\mathfrak{d}(p, q)$ is 0, thus, two states are the same. Let us assume that there is a cycle in M from q to q of length n. Since there can be a set of strings L_q accepted through the cycle including the self-loop, we should consider L_q for computing $\mathcal{C}(A, q, q)$. Therefore, we should compute all $\mathcal{C}$-values where the distance between two states is less than n to compute the basis. We denote the $\mathcal{C}$-values not considering the cycles in paths by $\mathcal{C}'$-values to avoid confusion.

Now we consider $\mathcal{C}(A, q, q)$, which is a basis for recursive definition of $\mathcal{C}$-values. First, for a variable $A \in V$ and a state $q \in Q$ of M, we define the following sets:

(i) $X(A, q, q) = \{\mathcal{C}'(B, q, r) + \mathcal{C}'(D, r, q) \mid r \in Q,\ A \to BD \in R\}$.
(ii) $Y(A, q, q) = \{\mathcal{C}'(B, q, q) \mid A \to B \in R\}$.

Then, for all $A \in V$ and $q, p \in Q$, we can establish another recursion for the basis of $\mathcal{C}$-values as follows:

$$\mathcal{C}(A, q, q) = \min[\mathcal{C}'(A, q, q) \cup X(A, q, q) \cup Y(A, q, q)].$$

Now it seems that we are ready to compute $\mathcal{C}$-values. However, we still have a problem to solve the recurrence step. Consider $Y(A, q, p)$ in Equation (1). We need to know $\mathcal{C}(B, q, p)$ to compute $\mathcal{C}(A, q, p)$ that is in the same level of recursion. Similarly, when r is the same state with q or p in the first term $X(A, q, p)$ of

the recurrence, we need to compute $\mathcal{C}(B, q, p)$ or $\mathcal{C}(D, q, p)$ to compute $\mathcal{C}(A, q, p)$. This problem also arises when we compute $\mathcal{C}'$-values. These dependencies between the recursive values in the same level prohibit us to compute the next level of recursion. Thus, we define an independent recursive definition for this problem. First, we define the following sets:

(i) $X(A, q, p) = \{\mathcal{C}(B, q, r) + \mathcal{C}(D, r, p) \mid r \in Q,\ A \to BD \in R\}$.
(ii) $Y(A, q, p) = \{d(a, L_{\mathrm{one-cyclic}}(q, p)) \mid A \to a \in R,\ a \in \Sigma\}$.

Here, r should not be q or p. Then, $\mathcal{K}$-values are defined as follows:

$$\mathcal{K}(A, q, p) = \min[X(A, q, p) \cup Y(A, q, p)].$$

Note that all $\mathcal{K}$-values can be computed by assuming that all $\mathcal{C}(A, q', p')$ are already computed where $\mathfrak{d}(q', p') < \mathfrak{d}(q, p)$. Now, we can redefine $\mathcal{C}(A, q, p)$ as the minimum of the following four values:

(i) $\mathcal{K}(A, q, p)$.
(ii) $\min_{A \to B} \mathcal{C}(B, q, p)$.
(iii) $\min_{A \to BD} \mathcal{C}(B, q, p) + \mathcal{C}(D, p, p)$.
(iv) $\min_{A \to BD} \mathcal{C}(B, q, q) + \mathcal{C}(D, q, p)$.

We can solve the dependencies between $\mathcal{C}$-values by the construction of a weighted graph, which has a vertex for each variable $A \in V$ and a special source vertex ϕ. Then, we connect ϕ to each vertex for a variable A with an edge whose weight is $\mathcal{K}(A, q, p)$. Also there are the edge of weight 0 from B to A if and only if $A \to B \in R$ and the edge of weight $\mathcal{C}(D, p, p)$ from B to A if and only if $A \to BD \in R$ or $A \to DB \in R$. Then, from the construction, $\mathcal{C}(A, q, p)$ becomes the shortest path from ϕ to A in the graph. Similarly, we can also solve the dependency problem for $\mathcal{C}'$-values. We give an algorithm for computing $d(L(G), L(M))$ in Algorithm 1.

Theorem 2. *Given a CFG $G = (V, \Sigma, R, S)$, an FA $M = (Q, \Sigma, \delta, s, F)$ and a non-negative cost function c, we can compute the edit-distance between $L(G)$ and $L(M)$ in $O(mn^2(n + \log m))$ worst-case time, where $m = |G|$ and $n = |Q|$.*

Lemma 2. *Given a CFG $G = (V, \Sigma, R, S)$, an FA $M = (Q, \Sigma, \delta, s, F)$ and an arbitrary cost function c, we can compute the edit-distance between $L(G)$ and $L(M)$ in $O(mn^2(n + m))$ worst-case time, where $m = |G|$ and $n = |Q|$.*

We can also observe that it is possible to retrieve the optimal alignment by backtracking the optimal path.

Lemma 3. *Given a CFG $G = (V, \Sigma, R, S)$ and an FA $M = (Q, \Sigma, \delta, s, F)$, we can compute the optimal alignment of length k between $L(G)$ and $L(M)$ in $O(mnk)$ worst-case time, where $m = |G|$ and $n = |Q|$.*

Algorithm 1. The algorithm for computing $d(L(G), L(M))$

Input: A CFG $G = (V, \Sigma, R, S)$ and an FA $M = (Q, \Sigma, \delta, s, F)$

for $q \in Q$ **do**
  for $d \leftarrow 1$ **to** $|Q| - 1$ **do**
    for $p \in Q$ **and** $\mathfrak{d}(p, q) = d$ **do**
      for $A \in V$ **do**
        $\mathcal{C}(A, q, p) \leftarrow \mathcal{K}(A, q, p)$
      end for
      $H \leftarrow$ heap of V (ordered by $\mathcal{C}(?, q, p)$)
      while $H \neq \emptyset$ **do**
        $A \leftarrow$ extract_min(H)
        for $A \in H$ **and** $(A \rightarrow BD \in R$ **or** $A \rightarrow DB \in R)$ **do**
          $\mathcal{C}(B, q, p) \leftarrow \min\{\mathcal{C}(A, q, p), \mathcal{C}(B, q, p) + \mathcal{C}(D, p, p)\}$, reheap($H, A$)
        end for
        for $A \in H$ **and** $A \rightarrow B \in R$ **do**
          $\mathcal{C}(A, q, p) \leftarrow \min\{\mathcal{C}(A, q, p), \mathcal{C}(B, q, p)\}$, reheap($H, A$)
        end for
      end while
    end for
  end for
end for
return $\min\{\mathcal{C}(S, s, f) \mid f \in F\}$

Output: $d(L(G), L(M))$

$$S_1 = A\ C\ T\ T\ A\ G\ T\ A\ G\ A\ T\ C\ C \qquad S_1 = A\ C\ T\ T\ A\ G\ T\ A\ G\ A\ T\ C\ C$$
$$\downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow \qquad \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow$$
$$S_2 = A\ C\ T\ T - G - A - - T\ C\ C \qquad S_2 = A\ C\ T\ T - - - - G\ A\ T\ C\ C$$

(a) (b)

Fig. 1. Two alignment examples that align S_2 to the target sequence S_1. The first alignment contains three short gaps while the second contains one long gap.

5 Affine Gap Distance

The approximate pattern matching problem is often used for the sequence alignment in bioinformatics [20,21]. A biological sequence alignment is a process of arranging the sequences of DNA, RNA or protein, and examining the similarities between the sequences. Consider the two alignments of sequences described in Fig. 1. Both have gaps of length four, which can be defined as deletion or insertion edit operations. However, the second alignment is biologically better since a deletion or insertion of four consecutive elements is more likely to occur than of three separated elements. Therefore, we need to give more penalty to the alignments containing many short gaps than few long gaps. Note that we can consider a sequence of consecutive deletion or insertion operations as a gap. Assume that an alignment ω consists of k consecutive insertions or deletions, in other words, a gap of the length k. Then, the cost of ω is linearly dependent

on $|\omega|$. Namely, $c(\omega) = g \cdot |\omega|$ where g is a constant. Instead of using this linear gap penalty function, we can use the *affine gap penalty* function to obtain biologically better alignments. The affine gap penalty function is defined as follows. Here the alphabet Ω of edit operations consists only of deletions $(a \to \lambda)$, insertions $(\lambda \to a)$ and trivial substitutions $(a \to a)$ that do not change the symbol. We denote $\Omega_{\text{del}} = \{(a \to \lambda) \mid a \in \Sigma\}$, $\Omega_{\text{ins}} = \{(\lambda \to a) \mid a \in \Sigma\}$ and $\Omega_{\text{triv}} = \{(a \to a) \mid a \in \Sigma\}$, and thus

$$\Omega = \Omega_{\text{del}} \cup \Omega_{\text{ins}} \cup \Omega_{\text{triv}}.$$

Let $\omega \in \Omega^+$ be a sequence of edit operations. The *(maximal) ID-decomposition of* ω (insertion–deletion decomposition of ω) is the tuple

$$\text{comp}_{\text{ID}}(\omega) = (\omega_1, \omega_2, \ldots, \omega_k)$$

where $\omega_i \in \Omega^+_{\text{del}} \cup \Omega^+_{\text{ins}} \cup \Omega^+_{\text{triv}}$, for $i = 1, \ldots, k$, and for any $1 \le j < k$ the strings ω_j and ω_{j+1} belong to different sets Ω^+_{del}, Ω^+_{ins} and Ω^+_{triv}.

The ID-decomposition of ω is obtained simply by subdividing ω into maximal substrings each consisting only of insertions, or only deletions, or only trivial substitutions and thus $\text{comp}_{\text{ID}}(\omega)$ is uniquely defined.

Now for a sequence consisting only of deletions or only of insertions, $\omega \in \Omega^+_{\text{del}} \cup \Omega^+_{\text{ins}}$, we define the affine gap cost of ω as $c_{\text{affine}}(\omega) = e + g \cdot |\omega|$, where e and g are constants. For a sequence consisting of trivial substitutions, $\omega \in \Omega^+_{\text{triv}}$, we set $c_{\text{affine}}(\omega) = 0$.

Now the *affine gap cost* of an arbitrary sequence of edit operations $\omega \in \Omega^+$, where $\text{comp}_{\text{ID}}(\omega) = (\omega_1, \omega_2, \ldots, \omega_k)$ is defined as

$$c_{\text{affine}}(\omega) = \sum_{i=1}^{k} c_{\text{affine}}(\omega_i).$$

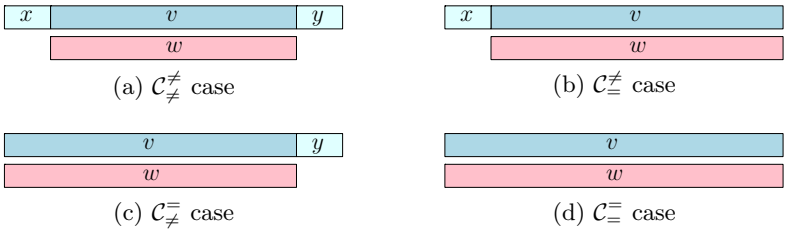
The affine gap cost gives, for a sequence of edit operations, a constant e penalty for each gap opening (consisting of consecutive insertions or consecutive deletions) and additionally a penalty that is linear in the length of the gap. The edit distance based on the affine gap cost function is called the *affine gap distance.*

We introduce an algorithm for computing the affine gap distance between a CFG and an FA. This is an extension of the previous algorithm, yet has the same time complexity. The key difference is that we define four types of $\mathcal{C}$-values as follows:

$$\mathcal{C}^{\lhd}_{\rhd}(A, q, p) = \min\{d(x, \lambda) + d(v, w) + d(y, \lambda) \mid A \overset{*}{\Rightarrow} xvy, |x| \lhd 0, |y| \rhd 0 \text{ and } p \in \delta(q, w)\},$$

where $\rhd, \lhd \in \{=, \neq\}$. We illustrate four cases in Fig. 2. The affine gap distance becomes $\min\{\mathcal{C}^{\lhd}_{\rhd}(S, s, f) \mid f \in F \text{ and } \rhd, \lhd \in \{=, \neq\}\}$.

Before introducing the recurrence for $\mathcal{C}^{\lhd}_{\rhd}$-values, corresponding to a variable $A \in V$ and states $q, p \in Q$ of M, we define the following sets:



(a) $\mathcal{C}_{\neq}^{\neq}$ case (b) $\mathcal{C}_{=}^{\neq}$ case

(c) $\mathcal{C}_{\neq}^{=}$ case (d) $\mathcal{C}_{=}^{=}$ case

Fig. 2. The pictorial representations of $\mathcal{C}_{\triangleright}^{\triangleleft}$-values where $\triangleright, \triangleleft \in \{=, \neq\}$

(i) $X(A,q,p) = \{\mathcal{C}_{\trianglerighteq}^{\trianglelefteq}(B,q,r) + \mathcal{C}_{\triangleright}^{\triangleleft}(D,r,p) - (h \textbf{ if } \trianglerighteq = \trianglelefteq = \text{`}\neq\text{'}) \mid r \in Q,\ A \to BD \in R\}$.
(ii) $Y(A,q,p) = \{\mathcal{C}_{\triangleright}^{\triangleleft}(B,q,p) \mid A \to B \in R\}$.
(iii) $Z(A,q,p) = \{\mathcal{I}_{\triangleright}^{\triangleleft}(a,w) \mid A \to a \in R,\ a \in \Sigma,\ w \in L_{\text{one-cyclic}}(q,p)\}$.

Here, $\mathcal{I}$-values are also defined as follows:

- $\mathcal{I}_{=}^{=}(a,w) = \min_{k\in[1,|w|]}\{c(a,w_k)+(|w|-1)\cdot g+(h \textbf{ if } k>1)+(h \textbf{ if } k<|w|)\}$,
- $\mathcal{I}_{\neq}^{=}(a,w) = \mathcal{I}_{=}^{\neq}(a,w) = (|w|+1)\cdot g + 2h$, and
- $\mathcal{I}_{\neq}^{\neq}(a,w) = g + h$.

Now we establish a recursive definition for $\mathcal{C}_{\triangleright}^{\triangleleft}$-values.

Theorem 3. *For all $A \in V$ and $q, p \in Q$,*

$$\mathcal{C}_{\triangleright}^{\triangleleft}(A,q,p) = \min[X(A,q,p) \cup Y(A,q,p) \cup Z(A,q,p)].$$

Note that the time complexity of this algorithm is still $O(mn^2(n+\log m))$, the same as in Theorem 2. Since we consider the four variations of $\mathcal{C}$-values, the time complexity increases to four times the runtime of Theorem 2.

Theorem 4. *Given a CFG $G = (V, \Sigma, R, S)$, an FA $M = (Q, \Sigma, \delta, s, F)$ and a non-negative cost function c, we can compute the affine gap distance between $L(G)$ and $L(M)$ in $O(mn^2(n + \log m))$ worst-case time, where $m = |G|$ and $n = |Q|$.*

Lemma 4. *Given a CFG $G = (V, \Sigma, R, S)$, an FA $M = (Q, \Sigma, \delta, s, F)$ and an arbitrary cost function c, we can compute the affine gap distance between $L(G)$ and $L(M)$ in $O(mn^2(n + m))$ worst-case time, where $m = |G|$ and $n = |Q|$.*

6 Concave Gap Distance

Many researchers consider non-linear gap penalty functions including the affine gap penalty function [13,17,22]. Although the affine gap penalty function prefers few longer gaps to many smaller gaps, the alignment results based on the affine gap penalty function are not practically the best. For example, assume that there are two alignments s_1 and s_2 aligning two sequences. Alignment s_1 contains two

gaps whose lengths are 99 and 100, respectively, while s_2 contains just one gap of length 240. Note that the remaining parts of s_1 and s_2 are perfectly matched. By employing the affine gap penalty function with $h = 5$ and $g = 1$, we obtain $c(s_1) = 99 + 100 + 5 \times 2 = 209$ and $c(s_2) = 240 + 5 = 245$. Even though the gap opening penalty is introduced in the affine gap distance, it may not be sufficient to consider some practical cases such as this example. This is why the *concave gap distance* is introduced and replaces other distances considering the linear or affine gap penalties. For a sequence consisting only of deletions or only of insertions, $\omega \in \Omega^+_{\mathrm{del}} \cup \Omega^+_{\mathrm{ins}}$, we define the concave gap cost of ω as

$$c_{\mathrm{concave}}(\omega) = e + g \cdot \log |\omega|,$$

where e and g are constants. For a sequence consisting of trivial substitutions, $\omega \in \Omega^+_{\mathrm{triv}}$, we set $c_{\mathrm{concave}}(\omega) = 0$. Now the *concave gap cost* of an arbitrary sequence of edit operations $\omega \in \Omega^+$, where $\mathrm{comp}_{\mathrm{ID}}(\omega) = (\omega_1, \omega_2, \ldots, \omega_k)$ is defined as

$$c_{\mathrm{concave}}(\omega) = \sum_{i=1}^{k} c_{\mathrm{concave}}(\omega_i).$$

Under this gap penalty function, the shape of the penalty score with respect to the length of the gap is concave in the sense that its forward differences are non-increasing. In other words, $\Delta c(\omega_1) \geq \Delta c(\omega_2) \geq \Delta c(\omega_3) \geq \cdots$ where $\Delta c(\omega_k) \equiv c(\omega_{k+1}) - c(\omega_k)$ and $|\omega_k| = k$. We define new $\mathcal{C}$-values for computing the concave gap distance as follows:

$$\mathcal{C}(A, q, p, i, j) = \min\{d(x, \lambda) + d(v, w) + d(y, \lambda) \mid A \stackrel{*}{\Rightarrow} xvy \neq \lambda\},$$

where $|x| = i, |y| = j$ and $p \in \delta(q, w)$. Here we use two additional parameters i and j for maintaining the lengths of gaps on both sides. We also define a set $\mathcal{V}(t)$ of variables that can derive strings of length t as follows:

$$\mathcal{V}(t) = \{A \mid A \in V, A \stackrel{*}{\Rightarrow} w \text{ and } |w| = t\}.$$

We can compute a set $\mathcal{V}(t)$ of variables as follows:

$$\bigcup_{k=1}^{t-1} \{A \mid A \to BD \in \mathcal{V}(k) \times \mathcal{V}(t-k)\} \cup \{A \mid A \to B \in \mathcal{V}(t)\} \cup \{A \mid A \to a\}.$$

Then, before introducing the recurrence for $\mathcal{C}$-values for the concave gap distance, we define the following sets corresponding to a variable $A \in V$ and states q, $p \in Q$ of M:

(i) $X(A, q, p) = \{\mathcal{C}(B, q, r, i, m) + \mathcal{C}(D, r, p, n, j) + g \cdot \log \frac{m+n}{mn} - h \mid r \in Q,\ A \to BD \in R\}$.
(ii) $Y(A, q, p) = \{\mathcal{C}(B, q, p, i, j - t) \mid A \to BD \in R,\ 1 \leq t \leq j,\ D \in \mathcal{V}(t)\}$.
(iii) $Z(A, q, p) = \{\mathcal{C}(D, q, p, i - t, j) \mid A \to BD \in R,\ 1 \leq t \leq i,\ B \in \mathcal{V}(t)\}$.
(iv) $U(A, q, p) = \{\mathcal{C}(B, q, p, i, j) \mid A \to B \in R\}$.

(v) $W(A,q,p) = \{\mathcal{I}(a,w,i,j) \mid A \to a,\ 0 \le i+j \le 1,\ w \in L_{\text{one-cyclic}}(q,p)\}$.

Here, $\mathcal{I}$-values are also defined as follows:

- $\mathcal{I}(a,w,0,0) = c(a,w_k) + 2h + \log(k-1)(|w|-k)$,
- $\mathcal{I}(a,w,0,1) = \mathcal{I}(a,w,1,0) = h + \log|w|$.

Now we establish a recurrence for computing the concave gap distance between a CFG and an FA.

Theorem 5. *For all $A \in V$, $q,p \in Q$ and $1 \le i,j \le |Q| \cdot 2^{\frac{h}{g}|V|}$,*

$$\mathcal{C}_{\triangleright}^{\triangleleft}(A,q,p) = \min[X(A,q,p) \cup Y(A,q,p) \cup Z(A,q,p) \cup U(A,q,p) \cup W(A,q,p)].$$

Based on the recurrence, we can compute the concave gap distance between $L(G)$ and $L(A)$ in exponential runtime.

Theorem 6. *Given a CFG $G = (V,\Sigma,R,S)$, an FA $M = (Q,\Sigma,\delta,s,F)$ and a non-negative cost function c, we can compute the concave gap distance between $L(G)$ and $L(A)$ in $O(mn^8 8^m)$ worst-case time, where $m = |G|$ and $n = |Q|$.*

7 Conclusions

We have considered the problem of approximately matching a context-free language specified by a CFG and a regular language specified by an FA. We have examined three types of gap cost functions that are used for approximate string matching: linear, affine and concave. Based on the dynamic programming approach, we have introduced algorithms for computing the linear, affine and concave gap distance between an FA and a CFG.

Given an FA of size n and a CFG of size m, we have presented algorithms for computing linear and affine gap distances in $O(nm^2(n+\log m))$ time under a non-negative cost function and $O(nm^2(n+m))$ time under an arbitrary cost function. We have also shown that computing the optimal alignment of length k takes $O(nmk)$ time by our algorithm when we consider linear or affine gap distance. Finally, we have proposed an $O(mn^8 8^m)$ time algorithm for computing the concave gap distance.

It will be interesting to see if we can compute the max-min distance between an FA and a CFG, or find a k optimal alignment between an FA and a CFG using a similar approach.

Acknowledgements. We wish to thank the referees for the careful reading of the paper and their constructive suggestions.

Han and Ko were supported by the Basic Science Research Program through NRF funded by MEST (2012R1A1A2044562) and Salomaa was supported by the Natural Sciences and Engineering Research Council of Canada Grant OGP0147224.

References

1. Aho, A., Peterson, T.: A minimum distance error-correcting parser for context-free languages. SIAM Journal on Computing 1(4), 305–312 (1972)
2. Bellman, R.: On a routing problem. Quarterly of Applied Mathematics 16, 87–90 (1958)
3. Choffrut, C., Pighizzini, G.: Distances between languages and reflexivity of relations. Theoretical Computer Science 286(1), 117–138 (2002)
4. Cocke, J.: Programming languages and their compilers: Preliminary notes. Courant Institute of Mathematical Sciences. New York University (1969)
5. Dijkstra, E.: A note on two problems in connexion with graphs. Numerische Mathematik 1, 269–271 (1959)
6. Earley, J.: An efficient context-free parsing algorithm. Communications of the ACM 13(2), 94–102 (1970)
7. Floyd, R.W.: Algorithm 97: Shortest path. Communications of the ACM 5(6), 345–348 (1962)
8. D.H.: Younger. Recognition and parsing of context-free languages in time n^3. Information and Control 10(2), 189–208 (1967)
9. Han, Y.-S., Ko, S.-K., Salomaa, K.: Computing the edit-distance between a regular language and a context-free language. In: Yen, H.-C., Ibarra, O.H. (eds.) DLT 2012. LNCS, vol. 7410, pp. 85–96. Springer, Heidelberg (2012)
10. Hopcroft, J., Ullman, J.: Introduction to Automata Theory, Languages, and Computation, 2nd edn. Addison-Wesley, Reading (1979)
11. Kari, L., Konstantinidis, S.: Descriptional complexity of error/edit systems. Journal of Automata, Languages and Combinatorics 9, 293–309 (2004)
12. Kasami, T.: An efficient recognition and syntax-analysis algorithm for context-free languages. Technical report, Air Force Cambridge Research Lab, Bedford, MA (1965)
13. Knight, J.R., Myers, E.W.: Approximate regular expression pattern matching with concave gap penalties. Algorithmica 14, 67–78 (1995)
14. Konstantinidis, S.: Computing the edit distance of a regular language. Information and Computation 205, 1307–1316 (2007)
15. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady 10(8), 707–710 (1966)
16. Lyon, G.: Syntax-directed least-errors analysis for context-free languages: a practical approach. Communications of the ACM 17(1), 3–14 (1974)
17. Miller, W., Myers, E.W.: Sequence comparison with concave weighting functions. Bulletin of Mathematical Biology 50(2), 97–120 (1988)
18. Mohri, M.: Edit-distance of weighted automata: General definitions and algorithms. International Journal of Foundations of Computer Science 14(6), 957–982 (2003)
19. Myers, G.: Approximately matching context-free languages. Information Processing Letters 54, 85–92 (1995)
20. Needleman, S.B., Wunsch, C.D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology 48(3), 443–453 (1970)
21. Sankoff, D., Kruskal, J.B.: Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison. Addison-Wesley (1983)
22. Waterman, M.S.: Efficient sequence alignment algorithms. Journal of Theoretical Biology 108, 333–337 (1984)
23. Wood, D.: Theory of Computation. Harper & Row (1987)