Available online at www.sciencedirect.com
ScienceDirect
Journal of Discrete Algorithms 5 (2007) 647–661
JOURNAL OF DISCRETE ALGORITHMS
www.elsevier.com/locate/jda

Regular expression constrained sequence alignment ☆

Abdullah N. Arslan

Department of Computer Science, The University of Vermont, Burlington, VT 05405, USA

Available online 12 February 2007

Abstract

We introduce *regular expression constrained sequence alignment* as the problem of finding the maximum alignment score between given strings S_1 and S_2 over all alignments such that in these alignments there exists a segment where some substring s_1 of S_1 is aligned to some substring s_2 of S_2, and both s_1 and s_2 match a given regular expression R, i.e. $s_1, s_2 \in L(R)$ where $L(R)$ is the regular language described by R. For complexity results we assume, without loss of generality, that $n = |S_1| \geqslant |m| = |S_2|$. A motivation for the problem is that protein sequences can be aligned in a way that known motifs guide the alignments. We present an $O(nmr)$ time algorithm for the regular expression constrained sequence alignment problem where $r = O(t^4)$, and t is the number of states of a nondeterministic finite automaton N that accepts $L(R)$. We use in our algorithm a nondeterministic weighted finite automaton M that we construct from N. M has $O(t^2)$ states where the transition-weights are obtained from the given costs of edit operations, and state-weights correspond to optimum alignment scores we compute using the underlying dynamic programming solution for sequence alignment. If we are given a deterministic finite automaton D accepting $L(R)$ with t_d states then our construction creates a deterministic finite automaton M_d with t_d^2 states. In this case, our algorithm takes $O(t_d^2 nm)$ time. Using M_d results in faster computation than using M when $t_d < t^2$. If we only want to compute the optimum score, the space required by our algorithm is $O(t^2 n)$ ($O(t_d^2 m)$ if we use a given M_d). If we also want to compute an optimal alignment then our algorithm uses $O(t^2 m + t^2|s_1||s_2|)$ space ($O(t_d^2 m + t_d^2|s_1||s_2|)$ space if we use a given M_d) where s_1 and s_2 are substrings of S_1 and S_2, respectively, $s_1, s_2 \in L(R)$, and s_1 and s_2 are aligned together in the optimal alignment that we construct. We also show that our method generalizes for the case of the problem with *affine gap penalties*, and for finding optimal regular expression constrained *local* sequence alignments.

Keywords: Regular expression; Sequence alignment; Dynamic programming; Pattern matching; Finite automaton

1. Introduction

We introduce *regular expression constrained sequence alignment* (*RECSA*) as the following problem: given strings S_1, S_2, and a regular expression R, find the maximum alignment score between S_1 and S_2 over all alignments that satisfy a given *regular expression constraint*. An alignment satisfies the constraint if it includes a segment in which a substring s_1 of S_1 is *aligned with* a substring s_2 of S_2, and both s_1 and s_2 match R where a string s is said to match a regular expression R if $s \in L(R)$, i.e. s is a string in the language described by R. We precisely explain

☆ Supported in part by NSF Award No. CCF-0514819. A preliminary version of this paper was presented in CPM 2005, Jeju Island, Korea, June 19–22, 2005.
E-mail address: aarslan@cs.uvm.edu.

1570-8667/$ – see front matter
doi:10.1016/j.jda.2007.01.003

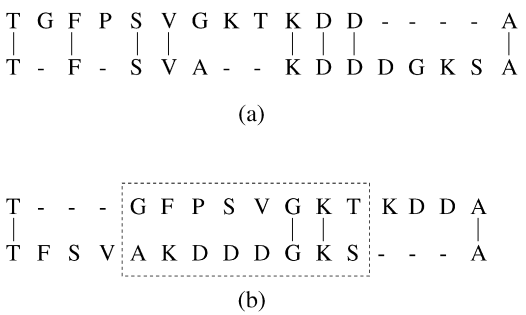

Fig. 1. For strings $S_1 = \texttt{TGFPSVGKTKDDA}$ and $S_2 = \texttt{TFSVAKDDDGKSA}$: (a) An alignment with maximum number of matches, 8. (b) An alignment in which substring $\texttt{GFPSVGKT}$ of S_1 is aligned with substring $\texttt{AKDDDGKS}$ of S_2, and both match $R = (\texttt{G}+\texttt{A})\Sigma\Sigma\Sigma\Sigma\texttt{GK}(\texttt{S}+\texttt{T})$ where Σ is a fixed alphabet on which the sequences are defined. This alignment has 4 matches, and it satisfies the regular expression constraint.

what we mean by "substring s_1 is aligned with substring s_2" when we define alignment paths in Section 3. In a simple case, if s_1, and s_2 are of the same length then we say that s_1 is aligned with s_2 if they appear in the same window of columns in the alignment matrix as shown in Fig. 1.

Fig. 1 illustrates an example in which sequences $S_1 = \texttt{TGFPSVGKTKDDA}$, and $S_2 = \texttt{TFSVAKDDDGKSA}$ are aligned in a way to maximize the number of matches (this is the *longest common subsequence* problem). An optimal alignment with 8 matches is shown in part (a). For the regular expression constrained sequence alignment problem with $R = (\texttt{G}+\texttt{A})\Sigma\Sigma\Sigma\Sigma\texttt{GK}(\texttt{S}+\texttt{T})$, where Σ denotes a fixed alphabet over which sequences are defined, the alignments sought change. The alignment in part (a) does not satisfy the regular expression constraint. Part (b) shows an alignment with which the constraint is satisfied. This alignment includes a region (shown with a rectangle drawn in dashed lines in the figure) where the substring $\texttt{GFPSVGKT}$ of S_1 is aligned with substring $\texttt{AKDDDGKS}$ of S_2, and both substrings match R. In this case, optimal number of matches achievable with the constraint decreases to 4.

The motivation for the problem is that when computing the homology of two protein sequences it may be important to take into account a common specific or putative structure. Family of similar protein sequences include a conserved region. Such conserved amino acid residues associated with a particular function is called a sequence motif. Typically, motifs span 10 to 30 amino acid residues. The notion of a motif was first explicitly introduced by Russell Doolittle in 1981 [6]. Discovery of sequence motifs related to a vast variety of enzymatic and binding activities of proteins has continued at a steady rate [2], and the motifs, in the form of amino acid patterns, were incorporated by Amos Bairoch in the PROSITE database. PROSITE (http://www.expasy.org/prosite) is maintained by Amos Bairoch and tightly integrated with SWISS-PROT [7]. For many years, PROSITE has been a collection of sequence motifs which were represented and stored as regular expressions. For example, the motif in Fig. 1 is the famous P-loop motif, first described in 1982 by John Walker and colleagues as "Motif A" and found later in many ATP- and GTP-binding proteins, corresponds to a flexible loop, sandwiched between a b-strand and an a-helix and interacting with b- and g-phosphates of ATP or GTP [14]. In PROSITE database it is represented as `[GA]-X(4)-G-K-[ST]` (ATP/GTP-binding site motif A (P-loop) (PS00017)) which means that the first position of the motif can be occupied by either Ala or Gly, the second, third, fourth, and fifth positions can be occupied by any amino acid residue, and the sixth and seventh positions have to be Gly and Lys, respectively, followed by either Ser or Thr.

The regular expression constraint can guide the alignments. As we observe in Fig. 1 the regular expression constraint changes the optimality of the alignments. If the sequences contain the same motif then it is biologically meaningful to seek an optimal alignment that contains the motif (i.e. that satisfies the corresponding regular expression constraint) because the motif should be part of the true alignment. In Fig. 1 strings S_1, and S_2 are not real protein sequences. We use them to present the effect of using a regular expression constraint in a simple setting with short strings.

In this paper we present an algorithm for the *RECSA* problem whose time complexity is $O(nmr)$, and $r = O(t^4)$, and t is the number of states of a nondeterministic finite automaton N that accepts the language $L(R)$ described by the given regular expression R. We use in our algorithm a nondeterministic weighted finite automaton M that we construct from N. M has $O(t^2)$ states where the transition-weights are obtained from the given costs of edit operations. If we are given a deterministic finite automaton D with t_d states accepting $L(R)$ then in our construction we can use D for N to create a weighted deterministic finite automaton M_d. If we use M_d for M in our algorithm then the resulting

time complexity is $O(t_d^2 nm)$. If $t_d < t^2$ then using M_d in our algorithm results in faster computation than using M. In general converting a deterministic finite automaton D (and minimizing the number of states afterward) takes in the worst case exponential time and space [9]. Therefore, we assume that we are given such D if one exists with $t_d < t^2$. M accepts alignments that satisfy the regular expression constraint where the weights of the states in M correspond to optimum constrained alignment scores. Our algorithm is based on a given dynamic programming formulation for sequence alignment. Instead of computing optimum scores, it uses the dynamic programming solution to compute weights for automaton M.

The outline of this paper is as follows: in Section 2, we summarize the previous related work, and results. In Section 3, we describe a framework for sequence alignment. In Section 4, we describe how we create the finite automaton that we use in our algorithm for the *RECSA* problem that we present in Section 5. In Section 6, we show how our techniques can be generalized for affine gap penalties, and computing regular expression constrained local alignment. We summarize our results in Section 7.

2. Previous related work

Given two sequences S_1 and S_2, the *pairwise sequence alignment* [15] problem is to compute the maximum score over all possible alignment matrices for these sequences. In an alignment matrix, a given scoring scheme assigns a score to each column corresponding to the symbols appearing in the column. A column contains symbols of S_1, and S_2, and it can also contain a special symbol $'-'$, but it cannot be composed entirely of $'-'$s. The score of an alignment matrix is the sum of its column-scores. In order to obtain an alignment matrix with the maximum score, we insert $'-'$s in S_1, and S_2, generating respectively, sequences S_1^*, and S_2^* with equal length such that S_1^* is the first, and S_2^* is the second row of this matrix. The *multiple sequence alignment* is the generalization of this problem for multiple sequences.

The constrained versions of the sequence alignment problems have been studied in the literature extensively [1,3–5,11–13].

Tang et al. [11] introduces the *constrained multiple sequence alignment* (*CMSA*) problem in which we are given k sequences $S_1, S_2, \ldots, S_k$ with maximum length n, and a pattern P with length r, and the solution of the problem is an alignment with optimal score such that there exists a sequence of columns $c_1 < c_2 < \cdots < c_r$ in the corresponding multiple sequence alignment matrix where column c_i is entirely composed of $P[i]$s. A motivation for the problem is the alignment of RNase sequences. Such sequences are all known to contain three active residues `His(H), Lyn(K), His(H)` that are essential for RNA degrading. Therefore it is natural to expect that in an alignment of RNA sequences, each of these residues should be aligned in the same column, i.e. alignment satisfies the constrained sequence `"HKH"`. the *CMSA* problem when $k = 2$ is called the *constrained pairwise sequence alignment* (*CPSA*) *problem* [3,11]. Solutions for the *CPSA* problem can be used to solve the *CMSA* problem. We can progressively align the sequences into a multiple alignment by using a minimum spanning tree obtained from the pairwise distance matrix of the sequences [3,11,13]. Tang et al. [11] introduces the *CPSA* problem, and presents an algorithm whose both time and space requirements are $O(rn^4)$ for sequences of length n. For the *CPSA* problem, Chin et al. [3], and Tang et al. [13] present improved algorithms with time complexity $O(nmr)$ where n, and m are the lengths of the sequences compared, and r is the length of the pattern P.

The longest common subsequence (*LCS*) problem for two strings is to find a common subsequence in both strings having maximum length. The *LCS* problem has many applications, and it has been studied extensively. Tsai [12] introduces the *constrained longest common subsequence* problem, and gives a dynamic programming solution whose time complexity is $O(rn^2m^2)$. For given strings S_1, S_2, and pattern P whose lengths are n, m, and r respectively, the constrained longest common subsequence problem is to find a longest common subsequence *lcs* of S_1 and S_2 such that P is a subsequence of this *lcs*. Chin et al. [4], and Arslan and Eğecioğlu [1] give different dynamic programming solutions for the constrained *LCS* problem with time complexity $O(nmr)$. Chin et al. [4] also shows that the constrained *LCS* problem is a special case of the *multiple sequence alignment* problem. Arslan and Eğecioğlu [1] introduces the *edit distance constrained LCS* problem as a generalization of the constrained *LCS* problem. The edit distance constrained *LCS* problem is, given strings S_1, S_2, P, and distance d, to find a longest common subsequence *lcs* of S_1 and S_2 such that this *lcs* has a subsequence whose simple edit distance from P is smaller than d. Simple edit distance between two strings is the minimum number of edit operations required to transform one string into the

other where the edit operations are insert, delete, and substitute. Arslan and Eğecioğlu [1] present an O($dnmr$)-time algorithm for the edit distance constrained *LCS* problem.

Using edit distances in the constraint is a step toward allowing alignments to contain patterns that may slightly differ in each sequence in the constrained sequence alignment problems. Another approach proposed by Comet and Henry [5] uses a method that rewards alignments containing motifs. From the motif database, the method first finds a known motif (or motifs) in each sequence separately to determine a common motif (or motifs). Next, it extends the dynamic programming solution for sequence alignment by reconsidering and rewarding in alignment each region where the motif appears in each sequence simultaneously. If the rewarding mechanism is properly set, the resulting alignment may contain the motif. If there are overlapping occurrences, it is difficult to choose the best one among them. In this paper we continue in the direction of using motifs as constraints in the alignments. Protein sequences contain motifs that are described in PROSITE format (http://www.expasy.org/txt/prosuser.txt) that can be translated into simple regular expressions. Our main contribution in this paper is that we introduce the regular expression constrained sequence alignment (*RECSA*) problem, and we present an algorithm for it. This makes it possible to constrain a desired score-optimal alignment of given protein sequences to contain a given motif.

Our algorithm is based on constructing a weighted finite automaton from the given regular expression constraint. We simulate copies of this automaton on alignments updating state weights as dictated by the underlying scoring scheme. Another problem in the literature that involves a regular expression in sequence alignment is the *approximate regular expression matching* which is studied by Myers and Miller [10]. This is the problem of finding a sequence S matching a given regular expression R (i.e. $S \in L(R)$) whose optimal alignment to a given sequence A has the highest score over all such sequences $S \in L(R)$. Myers and Miller construct an equivalent finite automaton from R, and use $|A|$ (where $|A|$ denotes the length of sequence A) copies of this automaton to create a graph $G_{A,R}$ where vertices in each copy are connected using possible alignment columns (i.e. edit operations) between the symbols of A and any sequence matching R. Then the problem becomes a path optimization problem in $G_{A,R}$ which can be solved in O($|A||R|$) time where $|R|$ is the length of regular expression R. Our automata construction is similar to the construction in this work in that we create a weighted finite automata (which can be represented as a directed graph) where the transitions (i.e. arcs) are on edit operations that have weights. However, in our case, these edit operations are between the given two sequences. We simulate the copies of the automaton we create from R on alignments as they are formed for these two sequences. In our case, (part of) an alignment we seek between two given sequences matches a sequence in $L(R)$. This is different than seeking a sequence S in $L(R)$ for a given regular expression R such that S and A have the highest possible score over all possible S in $L(R)$.

3. Framework

Given two strings S_1 and S_2, the global pairwise sequence alignment of S_1, and S_2 is to find an *alignment path* with the maximum score.

Given two strings $S_1[1..n]$ and $S_2[1..m]$ (for the complexity results we assume without loss of generality that $n \geqslant m$), we use the *alignment graph* G_{S_1,S_2} to analyze *alignments* between all substrings of S_1, and S_2. The alignment graph is a directed acyclic graph having $(n+1)(m+1)$ lattice points (u, v) as vertices for $0 \leqslant u \leqslant n$, and $0 \leqslant v \leqslant m$ (Fig. 2). An *alignment path* for substrings S_1 and S_2 is a directed path from the vertex $(0, 0)$ to (n, m) in G_{S_1,S_2}. To each vertex there is an incoming arc from each of its existing neighbors. Horizontal and vertical arcs correspond to insert and delete operations respectively. The diagonal arcs correspond to substitutions which are either matching (if the corresponding symbols are the same), or mismatching (otherwise). If we trace the arcs of an alignment path, and perform the indicated edit operations on S_1 in the order of the arcs in the alignment then we obtain S_2. Blocks of insertions and deletions are referred to as *gaps*.

The objective of sequence alignment is to quantify the similarity between S_1 and S_2 under a given *scoring scheme*. In *simple scoring scheme*, the arcs of G_{S_1,S_2} are assigned weights determined by some real function γ.

The following is the classical dynamic programming formulation [15] to compute the maximum global alignment score $\mathcal{H}_{i,j}$ achieved by an optimal alignment ending at each vertex (i, j):

$$\mathcal{H}_{i,j} = \max\{\mathcal{H}_{i-1,j} + \gamma(S_1[i] \to \epsilon), \mathcal{H}_{i-1,j-1} + \gamma(S_1[i] \to S_2[j]), \mathcal{H}_{i,j-1} + \gamma(\epsilon \to S_2[j])\} \tag{1}$$

for all i, j, $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant m$, with the boundary values $\mathcal{H}_{0,0} = 0$, $\mathcal{H}_{0,j} = H_{0,j-1} + \gamma(\epsilon \to S_2[j])$, and $\mathcal{H}_{i,0} = H_{i-1,0} + \gamma(S_1[i] \to \epsilon)$. Then $\mathcal{H}_{n,m}$ is the maximum global alignment score between S_1 and S_2. The maximum

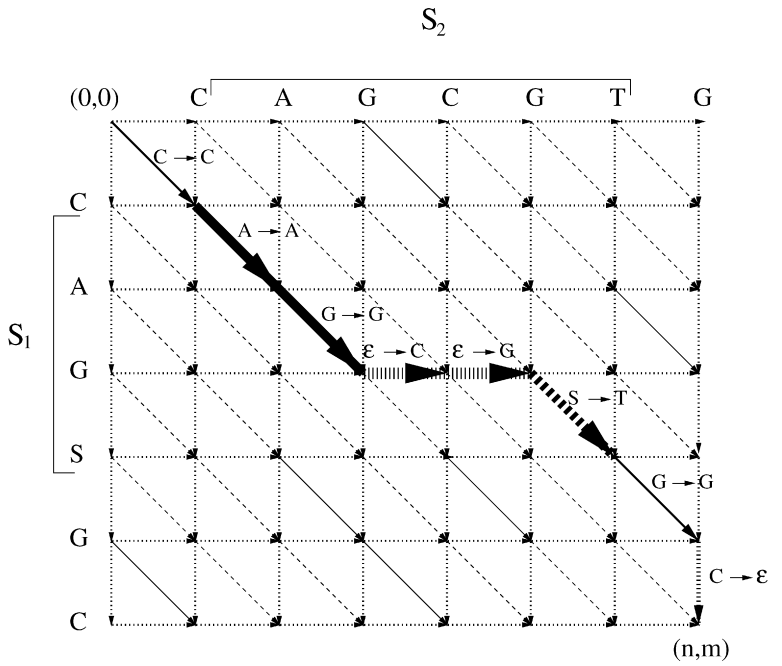

Fig. 2. Alignment graph G_{S_1,S_2} where $S_1 = \texttt{CAGSGC}$ and $S_2 = \texttt{CAGCGTG}$. Matching diagonal arcs are drawn as solid lines while mismatching diagonal arcs are shown by dashed lines. Dotted lines are used for horizontal and vertical arcs. An example alignment path is shown. Labels of the arcs on this path are the corresponding edit operations where ϵ denotes the null string.

global alignment score can be computed in time $O(nm)$ using $O(m)$ space because only $O(m)$ entries of the dynamic programming matrix need to be stored at any given time [15].

We say that substring s_1 of S_1 is *aligned with* substring s_2 of S_2 in a given alignment if there exists in the alignment a segment whose projection on S_1 is s_1, and whose projection on S_2 is s_2. In the alignment shown in Fig. 2, $s_1 = \texttt{AGS}$ is aligned with $s_2 = \texttt{AGCGT}$. The corresponding segment of the alignment is shown in thick lines in the figure.

In our algorithm for the regular expression constrained sequence alignment (*RECSA*) problem we use the dynamic programming formulation in (1), but instead of scores we compute weighted finite automata that we describe next.

4. Weighted finite automaton for regular expression constrained sequence alignment

We imagine alignments as strings of edit operations between two strings that are aligned, and we construct an automaton M that moves on edit operations. M changes states as the alignments are formed. M accepts those alignments in which the regular expression constraint is satisfied. An alignment satisfies a given regular expression constraint if M enters a final state after reading the edit operations in the alignment. That is, M must remember if the regular expression constraint is partially or completely satisfied by some substrings s_1 of S_1, and s_2 of S_2 that are aligned together. Since there may be many alignments accepted by M and we are interested in finding the maximum alignment score, we assign weights to the states, and as the alignments are formed the weights are updated after each move on an edit operation in M.

We construct M from a given regular expression R in several steps. We first construct a nondeterministic finite automaton A from R such that they are equivalent, i.e. $L(A) = L(R)$ [9]. Automaton A may have ϵ-moves. Then we construct an equivalent nondeterministic finite automaton $N = (Q, \Sigma, \delta, q_0, F)$ with no ϵ-moves as described in [9]. N has the same number of states as A. To summarize, N accepts the set of strings described by the regular expression R. We design automaton M to simulate two copies of N in parallel, one on S_1, and the other on S_2 as the alignments are formed.

We define a *weighted* $N \times N$ *automaton* as the finite automaton $M = (Q^M, W^M, \Sigma^M, q_0^M, F^M)$ which we construct as follows:

- $Q^M = Q \times Q$ is the set of states. Each state of M corresponds to a pair of states in N. M remembers in each state what part of the regular expression has been seen in S_1, and S_2.
- $\Sigma^M = (\Sigma \cup \{\epsilon\} \times \Sigma \cup \{\epsilon\}) - \{\epsilon \to \epsilon\}$. The alphabet for M is the set of edit operations which does not include $\epsilon \to \epsilon$. We note that since N has no ϵ-moves $\epsilon \to \epsilon$ is not a possible input for an $N \times N$ automaton.
- $q_0^M = (q_0, q_0)$ is the start state.
- $F^M = F \times F$ is the set of final states. If M is in a final state then M has processed an alignment that satisfies the regular expression constraint. That is, there are substrings s_1 of S_1 and s_2 of S_2 that are aligned together in an alignment, and both s_1, and s_2 take N to final states.
- $W^M : Q^M \to \mathbb{R}$ is a function that assigns real weights to each state in Q^M, and initially all weights are $-\infty$. We determine the active set of states of M by examining their weights. The active states of M have weights different than $-\infty$.
- $\delta^M : Q^M \times \Sigma^M \to Q^M$. M moves on edit operations as follows:
  - For all $x \to y \in \Sigma^M$, $\delta^M((q_0, q_0), x \to y)$ contains (q_0, q_0). That is, in the state diagram of M there is a loop on all possible inputs on state (q_0, q_0). This lets the regular expression matching start at any point during alignment formation.
  - For $x \neq \epsilon$, $\delta^M((p, q), x \to \epsilon) = \{(p', q) \mid p' \in \delta(p, x)\}$.
  - For $y \neq \epsilon$, $\delta^M((p, q), \epsilon \to y) = \{(p, q') \mid q' \in \delta(q, y)\}$.
  - For $x \neq \epsilon$, $y \neq \epsilon$, $\delta^M((p, q), x \to y) = \{(p', q') \mid p' \in \delta(p, x), q' \in \delta(q, y)\}$.
  - For all $x \to y \in \Sigma^M$, and $q_f \in F^M$, we also add q_f to $\delta^M(q_f, x \to y)$. That is, there is a loop on every final state on all possible inputs. Once an alignment satisfies the regular expression constraint, i.e. once a final state is reached in M, the rest of the alignment does not alter the satisfaction of the constraint. Therefore, M has the option of staying in a final state on any input after that final state is reached.

Fig. 3 includes an example weighted $N \times N$ automaton in part (b) for the finite automaton N shown in part (a) that is equivalent to regular expression $R = A(C + G)^*(S + T)$. For clarity, we choose as an example a simple regular expression, and we do not show the weights of the $N \times N$ automaton in part (b).

As M moves on edit operation $x \to y \in \Sigma^M$, the weights of its states are updated as described in the following two steps:

Step 1. For all $(p, q) \in Q^M$, if there exists (p', q') such that $(p, q) \in \delta^M((p', q'), x \to y)$ and $W^M(p', q') \neq -\infty$ then $W^M(p, q) = \max\{W^M(p', q') + \gamma(x \to y) \mid (p, q) \in \delta^M((p', q'), x \to y)\}$. New active states are those that are reachable from the active states on input $x \to y$. The weights of the active states are updated using the weight $\gamma(x \to y)$ of the edit operation $x \to y$, and the weights of the states through which new states are reached.

Step 2. For all $(p, q) \in Q^M$, if there does not exist (p', q') such that $(p, q) \in \delta^M((p', q'), x \to y)$ and $W^M(p', q') \neq -\infty$ then $W^M(p, q) = -\infty$. After the move some previously active states may become inactive. This may occur at some intermediate point (i, j) when a suffix of $S_1[1..i]$ (or $S_2[1..j]$) partially matching the regular expression R no longer partially matches R when the alignment ending at (i, j) is extended with the next edit operation $x \to y$. If a state is no longer active then its weight is reset to $-\infty$.

It is important that the state weights in M for each move are updated in these two steps, first Step 1, and then Step 2, because otherwise, the newly reachable states (new active states), and their weights may not be updated correctly.

For any given weighted $N \times N$ automaton M we denote by $M^{x \to y}$ for any $x \to y \in \Sigma^M$ a copy of the automaton after M finishes its move on $x \to y \in \Sigma^M$.

We will use multiple copies of the same weighted $N \times N$ automaton M. The weights will be updated as the alignment computations progress. At any given time the set of weights determine the current context in that copy of the automaton M. Otherwise all copies are identical.

Given two weighted $N \times N$ automata M_1 and M_2, we define a commutative and associative operation $\max_M$ such that $\max_M\{M_1, M_2\}$ is a weighted $N \times N$ automaton M with state weights calculated as follows:

$$\text{for all } (p, q) \in Q^M, \quad W^M(p, q) = \max\left\{W^{M_1}(p, q), W^{M_2}(p, q)\right\}. \tag{2}$$

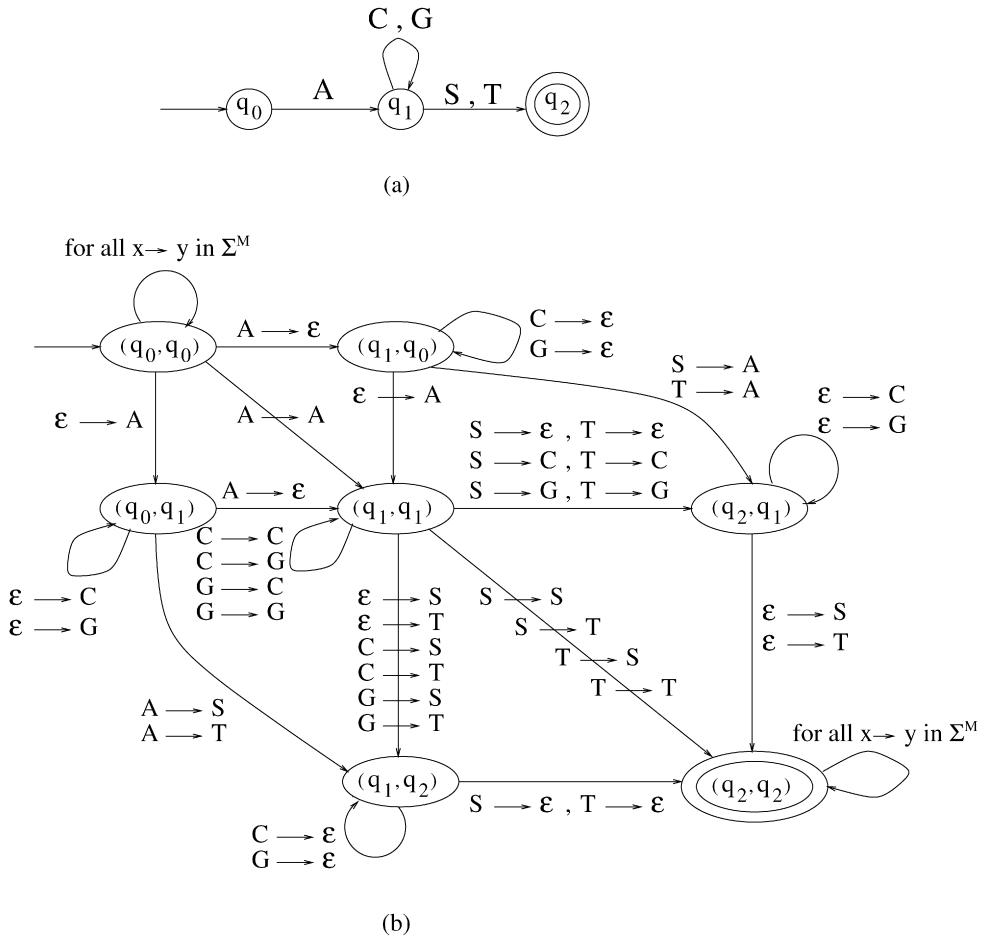

Fig. 3. (a) Finite automaton N equivalent to $R = A(C + G)^*(S + T)$. (b) Weighted $N \times N$ automaton M.

5. The algorithm

Let $|S_1| = n$, $|S_2| = m$ with $n \geqslant m$, and let N be a nondeterministic automaton with no ϵ-moves equivalent to regular expression R, and let M be a weighted $N \times N$ automaton constructed from N as we describe in Section 4.

We denote by $S[i..j]$ the substring of S from positions i to j, $i \leqslant j$. Let $S[i]$ denote the ith symbol of string S.

We say that a substring s matches a regular expression R if $s \in L(R)$.

Instead of optimal alignment scores in the classical dynamic programming solution we will compute *optimal finite automata*. A weighted $N \times N$ automaton $M_{i,j}$ is optimally weighted for $S[1..i]$, and $S_2[1..j]$ (or we simply say that $M_{i,j}$ is optimal since $S_1[1..i]$ and $S_2[1..j]$ are implied by the indices i and j in $M_{i,j}$) if the following two properties hold:

Property 1. *For all final states $(p, q) \in F^{M_{i,j}}$, $W^{M_{i,j}}(p, q)$ is the maximum alignment score between $S_1[1..i]$ and $S_2[1..j]$ over all alignments that include a region in which substring s_1 of $S_1[1..i]$ is aligned with substring s_2 of $S_2[1..j]$, and $s_1, s_2 \in L(R)$, i.e. N on input s_1 enters final state $p \in F$, and on input s_2 enters final state $q \in F$. If there do not exist such s_1 and s_2 then $W^{M_{i,j}}(p, q)$ is $-\infty$ (i.e. (p, q) is an inactive state in $M_{i,j}$).*

Property 2. *For all non-final states $(p, q) \in Q^M_{i,j} - F^{M_{i,j}}$, $W^{M_{i,j}}(p, q)$ is the maximum alignment score between $S_1[1..i]$ and $S_2[1..j]$ over all alignments that include a region in which s_1 is aligned with s_2, and s_1 is a suffix of $S_1[1..i]$, and s_2 is a suffix of $S_2[1..j]$, and N on input s_1 enters state $p \in Q$, and on input s_2 enters state $q \in Q$. If there do not exist such s_1 and s_2 then $W^{M_{i,j}}(p, q)$ is $-\infty$ (i.e. (p, q) is an inactive state in $M_{i,j}$).*

We compute all optimal $M_{i,j}$'s based on the dynamic programming solution in (1), and output the weight $\max\{W^{M_{n,m}}(p,q) \mid (p,q) \in F^{M_{n,m}}\}$. That is, the *maximum regular expression constrained alignment score* is the maximum weight of the final states in the optimal automaton $M_{n,m}$.

Our solution computes $M_{i,j}$ for all i, j by simulating automata $M_{i-1,j}$, $M_{i-1,j-1}$, and $M_{i,j-1}$ on the corresponding edit operations, and taking the maxima of the resulting optimal state-weights based on the dynamic programming solution in (1).

For all i, j, $0 \leqslant i \leqslant n$, $0 \leqslant j \leqslant m$, $M_{i,j}$'s are identical weighted $N \times N$ automata except that the weights can be different. In $M_{0,0}$ the state-weights are all $-\infty$ except that we set the weight of the start state (q_0, q_0) to 0, i.e. $W^{M_{0,0}}(q_0, q_0) = 0$.

On the boundary, $M[0, j]$ and $M[i, 0]$ are defined as follows:

- for all j, $1 \leqslant j \leqslant m$, the only possible alignment is obtained by extending an optimal alignment ending at $(0, j-1)$ by the edit operation $\epsilon \to S_2[j]$, therefore,

$$M_{0,j} = M_{0,j-1}^{\epsilon \to S_2[j]}, \tag{3}$$

- and for all i, $1 \leqslant j \leqslant n$, the only possible alignment is obtained by extending an optimal alignment ending at $(i-1, 0)$ by the edit operation $S_1[i] \to \epsilon$, therefore,

$$M_{i,0} = M_{i-1,0}^{S_1[i] \to \epsilon}. \tag{4}$$

We use Eq. (2) to compute the dynamic programming step in (1) as the following: for all i, j, $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant m$,

$$M_{i,j} = \max_M \left\{ M_{i-1,j}^{S_1[i] \to \epsilon}, M_{i-1,j-1}^{S_1[i] \to S_2[j]}, M_{i,j-1}^{\epsilon \to S_2[j]} \right\}. \tag{5}$$

In this step, we consider automaton $M_{i-1,j}$ after its move on edit operation $S_1[i] \to \epsilon$, and similarly automaton $M_{i-1,j-1}$ after its move on $S_1[i] \to S_2[j]$, and automaton $M_{i,j-1}$ after its move on $\epsilon \to S_2[j]$. In each of these resulting automata, weight of each state (p, q) is updated as we describe in Section 4. That is, each state (p, q) has the optimal weight obtainable by alignments ending with the corresponding edit operation. With $\max_M$ operation, we examine the weight of each (p, q) in all these automata, and the maximum weight is taken as the weight of (p, q) in automaton $M_{i,j}$.

Fig. 4 schematically describes the computations of $M_{i,j}$. We claim that for all i, j, $M_{i,j}$ computed in (5) is optimal. The correctness can be proved by induction on nodes (i, j). We consider an ordering for the nodes in which (i, j) comes after its neighbors $(i-1, j)$, $(i, j-1)$, and $(i-1, j-1)$ if they exist. This ordering can be generated by two nested loops: the outer loop $i = 0$ to n, and the inner loop $j = 0$ to m. The base case is when $i = 0$ for all j in which all weights are $-\infty$ in $M_{0,j}$, and the claim is true. Assuming that the claim is true for $M_{i-1,j}$, $M_{i,j-1}$, and $M_{i-1,j-1}$ we will show that each of the following automata is optimally weighted for $S_1[1..i]$, and $S_2[1..j]$ when the alignments are constrained to use the indicated arc:

1. $M_{i-1,j}^{S_1[i] \to \epsilon}$ when $((i-1, j), (i, j))$ is a required arc for the alignments,
2. $M_{i,j-1}^{\epsilon \to S_2[j]}$ when $((i, j-1), (i, j))$ is a required arc for the alignments,
3. $M_{i-1,j-1}^{S_1[i] \to S_2[j]}$ when $((i-1, j-1), (i, j))$ is a required arc for the alignments.

Optimality of $M_{i,j}$ will follow from these results since an optimal constrained alignment at node (i, j) uses one of these arcs, and we compute maximum scores for all possible optimal alignments (as state-weights) which partially or completely satisfy the regular expression constraint in the resulting optimal automaton in (5).

To show that $M_{i-1,j}^{S_1[i] \to \epsilon}$ is optimally weighted for $S_1[1..i]$ and $S_2[1..j]$ when the alignments are constrained to use the arc $((i-1, j), (i, j))$, we need to show that Properties 1 and 2 hold for $M_{i-1,j}^{S_1[i] \to \epsilon}$ with the given requirement. In this case, we consider only the alignments that include the arc $((i-1, j), (i, j))$. An optimal regular expression constrained score with this requirement is obtained from an optimal score obtained at node $(i-1, j)$ by adding to it the score $\gamma(S_1[i] \to \epsilon)$. For Property 1, for final states $(p, q) \in F_{M_{i,j}}$ there are two cases to consider: (1) If a final state (p, q) was already an active state in $M_{i-1,j}$ then the optimality of the weight $W^{M_{i,j}}$ is followed from Property 1

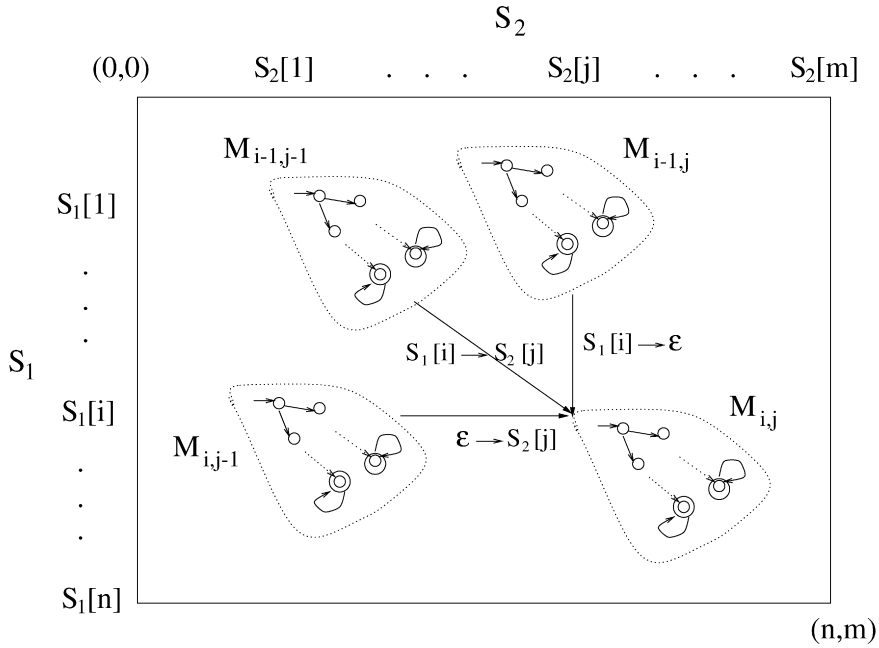

Fig. 4. Computations of weighted $N \times N$ automata.

of $M_{i-1,j}$, and the fact that all optimal alignment scores in this case include the score $\gamma(S_1[i] \to \epsilon)$. (2) If a final state (p, q) is entered newly, i.e. (p, q) is an active state in $M_{i,j}$ but not an active state in $M_{i-1,j}$ then the optimality is obtained in this case from the optimality of the weights of all non-final states in $M_{i-1,j}$ (Property 2), the fact that all optimal alignment scores in this case include the score $\gamma(S_1[i] \to \epsilon)$, and the regular expression match is obtained through one of these non-final states. For Property 2, the optimality is followed from the fact that all optimal alignments in this case use the same arc, $((i-1, j), (i, j))$.

Proving the case for $M_{i,j-1}^{S_1[i]\to\epsilon}$ with the constraint that the alignments are required to use the arc $((i, j-1), (i, j))$ is very similar to the case for $M_{i,j-1}^{S_1[i]\to\epsilon}$ and arc $((i-1, j), (i, j))$ because these two cases are symmetric.

The proof of the case for $M_{i-1,j-1}^{S_1[i]\to S_2[j]}$ with the required arc $((i-1, j-1), (i, j))$ is also similar. In this case, (1) if a final state (p, q) was already an active state in $M_{i-1,j-1}$ then the optimality of the weight $W^{M_{i,j}}$ is followed from Property 1 of $M_{i-1,j-1}$, and the fact that all optimal alignment scores in this case include the score $\gamma(S_1[i] \to S_2[j])$. (2) If a final state (p, q) is entered newly, i.e. (p, q) is an active state in $M_{i,j}$ but not in $M_{i-1,j-1}$ then the optimality is obtained in this case from the optimality of the weights of all non-final states in $M_{i-1,j-1}$ (Property 2), and the fact that all alignment scores in this case include the score $\gamma(S_1[i] \to S_2[j])$. Property 2 holds because of the fact that all optimal alignments in this case use the same arc, $((i-1, j-1), (i, j))$. This concludes the proof.

5.1. Implementation issues and complexity

Let t be the number of states in automaton N accepting the language $L(R)$. Automaton M we construct has $O(t^2)$ states.

We note that each $M_{i,j}$ is identical except for the state weights. Therefore, we can store for each $M_{i,j}$ only a list $A_{i,j}$ of active states with their weights. We can implement $A_{i,j}$ as an array such that $A_{i,j}[p]$ holds the weight of the state p which is active if it is not $-\infty$.

We maintain a single complete copy of M to access the transition function. For a given state and an edit operation the set of next states is stored in a list.

We can implement the two-step move of $M_{i,j}$ on any given input $x \to y \in \Sigma^M$ as follows: we initialize a new empty list A. For every $(p, w_p) \in A_{i,j}$ where p is an active state with weight w_p in $M_{i,j}$, if (p, w'_p) is not in A with any w'_p then we add $(p, w_p + \gamma(x \to y))$ to A; if (p, w'_p) is in A then we update p's weight in A to $w_p + \gamma(x \to y)$

if $w_p + \gamma(x \to y) > w'_p$. The resulting list A at the end of this process is the list of active states, and their weights in $M_{i,j}^{x \to y}$. Since there is no $\epsilon \to \epsilon$ transition in M, from any state $(p, q) \in Q^M$ on input $x \to y$ there are transitions to $O(t^2)$ states. Therefore, the result of the two-step move of $M_{i,j}$ on input $x \to y$ that we describe above can be computed in $O(t^4)$ time since $A_{i,j}$ contains $O(t^2)$ active states. The $\max_M$ operation in (2) can be performed in $O(t^2)$ time. Hence, our algorithm takes $O(t^4 nm)$ time.

If we are given a deterministic finite automaton D with t_d states that accepts the language $L(R)$ described by the given regular expression R then our construction for M creates a deterministic $N \times N$ automaton M_d from D when we use D for N in our construction that we describe in Section 4. It is easy to see that M_d is deterministic because in M_d for every state p on every input $x \to y \in \Sigma^M$, the next state is unique since D is deterministic, and there is no null-transition since $\epsilon \to \epsilon$ is not an edit operation. The number of states in M_d is $O(t_d^2)$. We note that in this case the two-step move of $M_{i,j}$ on input $x \to y$ can be computed in $O(t_d^2)$ time since from each active state on input $x \to y$ there is only one transition. In this case, our algorithm takes $O(t_d^2 nm)$ time. This is faster than using M if $t_d < t^2$. In general constructing an equivalent deterministic automaton D from a given nondeterministic finite automaton N (and minimizing the number of states afterward in D) takes in the worst case exponential time and space [9]. However, in some cases we can obtain D from N with simple modifications. For example, we can modify the automaton N shown in Fig. 3 part (a) and obtain a deterministic automaton D. In N, we add a trap state with a self-loop on all possible input, and transitions from all the states in N to this trap state on all possible input. The resulting automaton is deterministic, and it has only one more state (the trap state) than N.

For the complexity results, we assume without loss of generality that $n = |S_1| \geqslant m = |S_2|$. If we only want to compute the optimum score, then our algorithm can be implemented using $O(t^2 m)$ space (or $O(t_d^2 m)$ space if we use given M_d). In this case, the dynamic programming computations are done column by column (it can also be done row by row but this is more advantageous since $n \geqslant m$) and we only need to store the lists $A_{i,j}$'s of active states, and their weights for the previous and current columns when we compute $A_{i,j}$'s in the current column.

If we want to reconstruct an optimal alignment path, we first show a naive method to achieve this. We store all $A_{i,j}$'s, and for every active state in $A_{i,j}$, from which neighboring automaton (left, diagonal, or up neighbor, respectively, $M_{i,j-1}$, $M_{i-1,j-1}$, or $M_{i-1,j}$), and from which state in that neighbor the maximum score is obtained. We note that this information is enough for the reconstruction of an optimal alignment. We generate this information by modifying the maximum operation $\max_M$ in (2) such that along with the maximum score, the neighbor and state information (we call them source neighbor and *source* state) from which this score is obtained is also recorded. Then we can start with a final state p with maximum score at position (n, m). Next, we consider the neighboring source automaton-position recorded for p, and the source state q (not necessarily a final state) in that neighboring-automaton. If we repeat this process for all source automaton and state pairs until reaching position $(0, 0)$, and the start state in $M_{0,0}$, we can construct an optimal alignment in reverse. This method requires that we maintain for all (i, j), for all active states in $A_{i,j}$ neighboring source automaton position (left, diagonal, or up) and source state information along with the weights. Therefore, the space requirement increases to $O(t^2 nm)$ (or $O(t_d^2 nm)$ if we use given M_d). The additional work to reconstruct an optimal alignment after all $A_{i,j}$'s are computed is $O(n + m)$.

We propose another method that uses the linear space sequence alignment algorithm of Hirschberg [8], and the naive method we present above. We first use our algorithm only to compute the regular expression constrained sequence alignment score, and along with it in the alignment graph the start position (i_s, j_s) and the end position (i_e, j_e) in an optimal alignment path where the regular expression match occurs. We can do this by modifying the dynamic programming formulation so that in non-final states we carry along with the optimum score the start position of the regular expression match that occurs in an alignment this optimum score is obtained, and in every final state the regular expression match start position, and the first position this final state is reached (i.e. the end position of the regular expression match). This way, when the optimum score is computed using $O(t^2 m)$ space, we know the positions (i_s, j_s), and (i_e, j_e). Then we use Hirschberg's linear space algorithm for computing optimal unconstrained alignments between $(0, 0)$ and (i_s, j_s), and between (i_e, j_e) and (n, m). We can use the naive method that we describe above to find an optimal alignment between (i_s, j_s) and (i_e, j_e). This part takes $O(t^2 |s_1||s_2|)$ space (or $O(t_d^2 |s_1||s_2|)$ space if M_d is given and used) where s_1 and s_2 are the substrings, respectively, of S_1 and S_2, and both $s_1, s_2 \in L(R)$, and both appear in the alignment that we construct. Therefore, the total space requirement of this method is $O(t^2 m + t^2 |s_1||s_2|)$ (or $O(t_d^2 m + t_d^2 |s_1||s_2|)$ if M_d is given and used).

6. Affine gaps and local alignments

In sequence alignment, for presentation we usually use alignment matrices instead of edit paths. We can change our automata construction such that the automata we construct move on alignment columns instead of edit operations. In the remaining text for convenience we use $[a, b]$ to represent an alignment column containing a and b, i.e. $[a, b]^T$. We redefine the construction of $N \times N$ automaton by replacing moves (transitions) on edit operations by those on the corresponding alignment columns: we use $[S_1[i], -]$ for edit operation $S_1[i] \to \epsilon$, $[-, S_2[j]]$ for $\epsilon \to S_2[j]$, and $[S_1[i], S_2[j]]$ for $S_1[i] \to S_2[j]$. We show an example to such an automaton in Fig. 5. The weights of transitions on alignment columns are the same as those that are on the corresponding edit operations.

It is possible to modify our algorithm for scoring schemes other than the simple scoring scheme. For example, affine gap penalties is another common scoring scheme in which the total penalty for a gap of size k, i.e. a block of k insertions (or deletions), is $\alpha + (k-1)\mu$ where α is the gap open penalty, and μ is called the gap extension penalty. The dynamic programming formulation for computing the maximum alignment score in this case can be described as follows (see [15] for more details on affine gaps): let $\mathcal{H}_{0,0} = 0$, and for all j, $1 \leqslant j \leqslant m$, $\mathcal{H}_{0,j} = \mathcal{E}_{0,j} = \max\{\mathcal{H}_{0,j-1} - \alpha, \mathcal{E}_{0,j-1} - \mu\}$, and for all i, $1 \leqslant i \leqslant n$, $\mathcal{H}_{i,0} = \mathcal{F}_{i,0} = \max\{\mathcal{H}_{i-1,0} - \alpha, \mathcal{F}_{i-1,0} - \mu\}$, and define

$$\begin{aligned} &\mathcal{E}_{i,j} = \max\{\mathcal{H}_{i,j-1} - \alpha, \mathcal{E}_{i,j-1} - \mu\}, \qquad \mathcal{F}_{i,j} = \max\{\mathcal{H}_{i-1,j} - \alpha, \mathcal{F}_{i-1,j} - \mu\}, \\ &\mathcal{H}_{i,j} = \max\bigl\{\mathcal{H}_{i-1,j-1} + \gamma\bigl(\bigl[S_1[i], S_2[j]\bigr]\bigr), \mathcal{E}_{i,j}, \mathcal{F}_{i,j}\bigr\} \end{aligned} \tag{6}$$

where $\gamma([S_1[i], S_2[j]])$ is the *match score* (usually 1) if $S_1[i] = S_2[j]$, and the *mismatch penalty* (usually -1) otherwise. $\mathcal{E}$ and $\mathcal{F}$ keep track of optimum alignment scores that are obtained by alignments that end with an opened gap. $\mathcal{H}$ stores the maximum score of all possible alignments. Matrices $\mathcal{E}$ and $\mathcal{F}$ have the optimum scores that belong to alignments ending with a gap. In $\mathcal{H}$ we consider extensions of optimal alignment scores in $\mathcal{E}$ and $\mathcal{F}$, and optimum scores that can be obtained by alignments that do not end with a gap (that end with a substitution or a match). Affine gap penalties do not increase the asymptotic complexity of the alignment problem.

We can use our method to compute regular expression constrained sequence alignment for affine gap penalties. Using our redefinition of $N \times N$ automaton for alignment columns, we create three $N \times N$ automata. One of these automata is the automaton M which we describe in Section 4 but we redefine it by replacing in the construction,

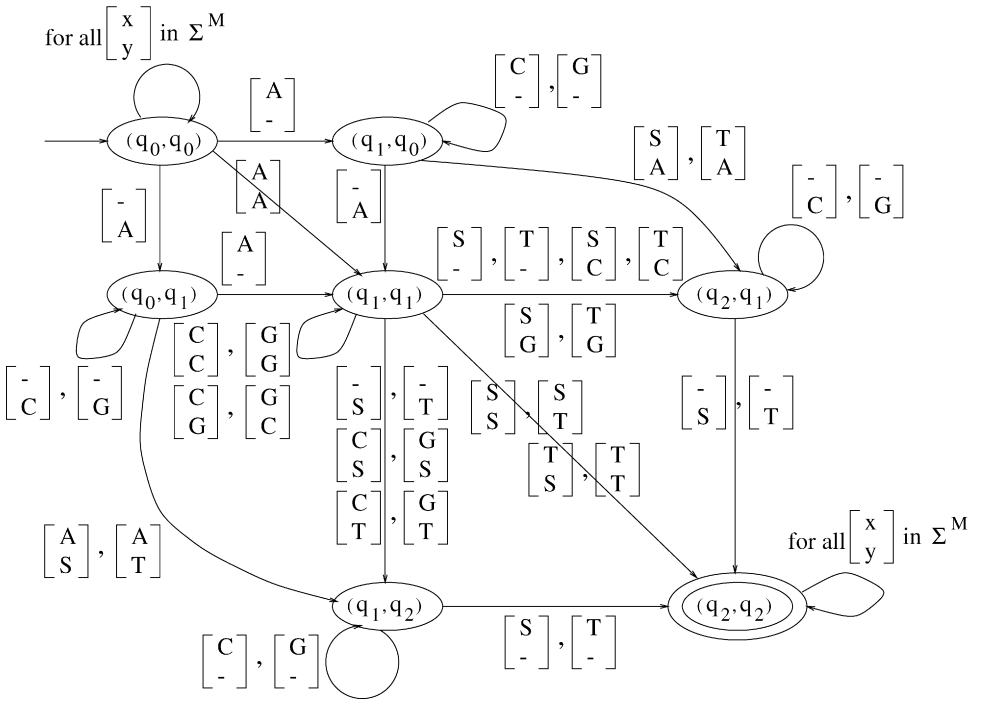

Fig. 5. $N \times N$ automaton that moves on alignment columns. This is equivalent to the automaton in Fig. 3(b).

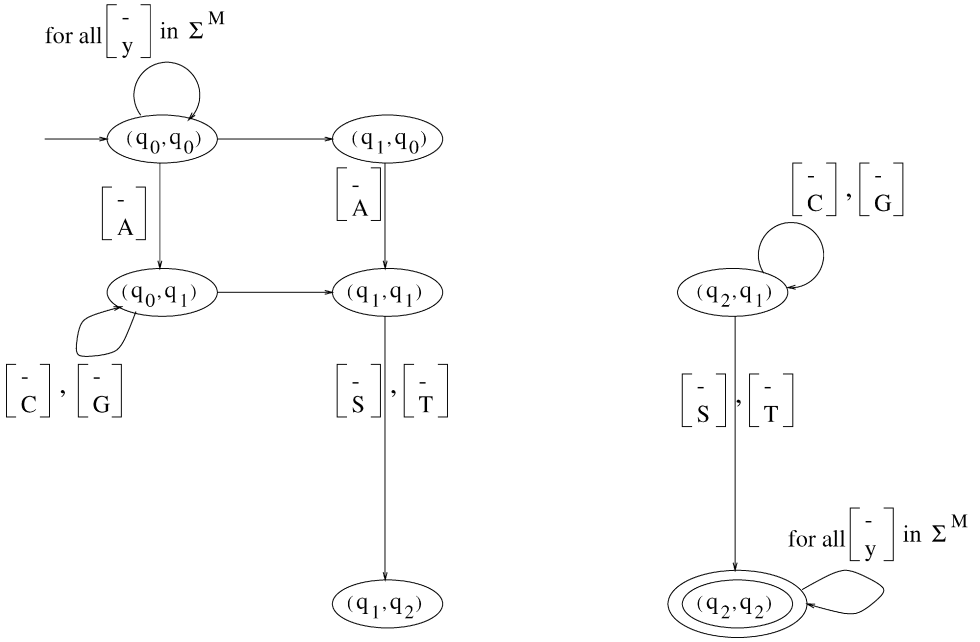

Fig. 6. $N \times N$ automaton E has the same states as the automaton M shown in Fig. 5, but E has only the transitions of M on alignment columns of type $[-, y]$. The weight of each transition in E is $-\mu$, whereas each of the corresponding transition in M is $-\alpha$.

every edit operation by the corresponding alignment column. We show this automaton, M, in Fig. 5. In addition we create two more $N \times N$ automata namely E and F which are obtained from M. We design these automata such that M in its state weights stores the optimum scores in matrix $\mathcal{H}$, and E and F in their state weights store the optimum scores in matrices $\mathcal{E}$ and $\mathcal{F}$ respectively. Both E and F have the same states as automaton M. Their set of transitions are subsets of that of M. E moves only on alignment columns of type $[-, S_2[j]]$, and F moves only on alignment columns of type $[S_1[i], -]$. We show automaton E in Fig. 6, and F in Fig. 7. A difference between automaton M, and automata E, and F is that the transition weights are not identical. Each move on E, and each move on F has score $-\mu$. The score of each of these same moves in M is $-\alpha$. M has transitions that are neither in E nor in F. These transitions are for substitutions, and matches, and their weights are, respectively, the given mismatch penalty and the given match score.

For all i, j, $0 \leqslant i \leqslant n$, $0 \leqslant j \leqslant m$, $M_{i,j}$'s are identical weighted $N \times N$ automata that move on alignment columns except that the weights can be different. This is true for $E_{i,j}$'s, and $F_{i,j}$'s. That is, except for state weights all $E_{i,j}$'s are identical, and except for state weights all $F_{i,j}$'s are identical. In $M_{0,0}$ the state-weights are all $-\infty$ except that we set the weight of the start state (q_0, q_0) to 0, i.e. $W^{M_{0,0}}(q_0, q_0) = 0$. In $E_{0,0}$ and $F_{0,0}$ the weights of all states are $-\infty$.

On the boundary, we define $M[0, j]$, $M[i, 0]$, $E[0, j]$, $E[i, 0]$, $F[0, j]$, and $F[i, 0]$ as follows:

– for all j, $1 \leqslant j \leqslant m$,

$$E_{0,j} = \max_M \left\{ M_{0,j-1}^{[-,S_2[j]]}, E_{0,j-1}^{[-,S_2[j]]} \right\}, \tag{7}$$

$M_{0,j}$ has the same state-weights as $E_{0,j}$. In this case, there is only one possible alignment: $S_2[1..j]$ is aligned to all $'-'$s (a gap of size j),

– and for all i, $1 \leqslant j \leqslant n$

$$F_{i,0} = \max_M \left\{ M_{i-1,0}^{[S_1[i],-]}, F_{i-1,0}^{[S_1[i],-]} \right\}, \tag{8}$$

$M_{i,0}$ has the same state-weighs as $F_{i,0}$. In this case, the only possible alignment is the one in which $S_1[1..i]$ is aligned to all $'-'$s (a gap of size i).

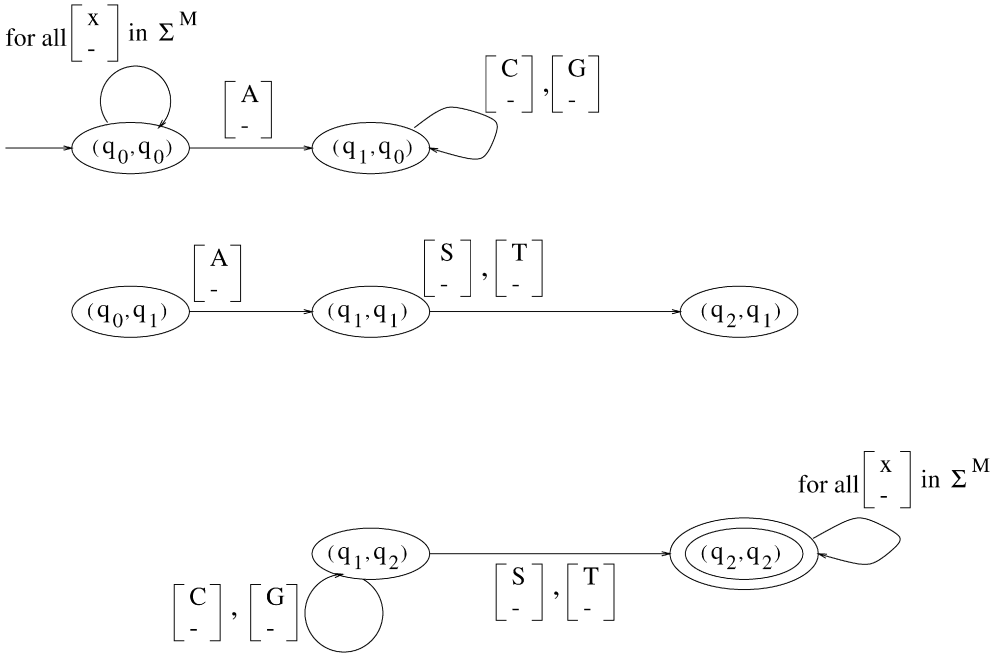

Fig. 7. $N \times N$ automaton F has the same states as the automaton M shown in Fig. 5, but F has only the transitions of M on alignment columns of type $[x, -]$. The weight of each transition in F is $-\mu$, whereas each of the corresponding transition in M is $-\alpha$.

We use Eq. (2) to compute the dynamic programming step in (6) as follows: for all i, j, $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant m$,

$$E_{i,j} = \max_M \bigl\{ M_{i,j-1}^{[-,S_2[j]]}, E_{i,j-1}^{[-,S_2[j]]} \bigr\}, \qquad F_{i,j} = \max_M \bigl\{ M_{i-1,j}^{[S_1[i],-]}, F_{i-1,j}^{[S_1[i],-]} \bigr\},$$
$$M_{i,j} = \max_M \bigl\{ M_{i-1,j-1}^{[S_1[i],S_2[j]]}, E_{i,j}, F_{i,j} \bigr\}. \tag{9}$$

In automata E and F, some states can be active because they are active in M even though these states are not reachable in E and F from their respective start states. We simulate automaton $M_{i,j-1}$ on $[-, S_2[j]]$, and $E_{i,j-1}$ on $[-, S_2[j]]$. These simulations update the state weights. In $M_{i,j-1}$, the weight of $[-, S_2[j]]$ is $-\alpha$, and a transition from an active state p to q defined on this alignment column adds this weight $-\alpha$ to the weight of state p, and if this sum is greater than the weight of state q then it becomes the new weight of state q. The weights of unreachable states are reset to $-\infty$. Similarly, in $E_{i,j-1}$ the weight of $[-, S_2[j]]$ is $-\mu$, and a transition from an active state p to q defined on this alignment column adds this weight $-\mu$ to the weight of state p, and this sum may become a new maximum weight at state q in which case the weight of state q is updated. The weights of unreachable states are reset to $-\infty$. We note that in $E_{i,j-1}$ since only a subset of transitions in $M_{i,j-1}$ are defined, normally there are many inactive states in $E_{i,j-1}$. Automaton $E_{i,j}$ receives in its states the maximum state weights obtained at the end of these simulations. We similarly, simulate automaton $M_{i-1,j}$ on $[S_1[i], -]$, and $F_{i-1,j}$ on $[S_1[i], -]$, and obtain automaton $F_{i,j}$. Finally, automaton $M_{i-1,j-1}$ is simulated on $[S_1[i], S_2[j]]$ in which each transition defined on this column from an active state has weight $-\alpha$. This maximum state weights obtained by this simulation, and the corresponding maximum state weights in $E_{i,j}$, and $F_{i,j}$ are compared, and the maximum weights become the state weights in automaton $M_{i,j}$.

Sometimes we are interested in finding local similarities. A local alignment can start at any position in sequences with initial score 0. The Smith–Waterman local alignment algorithm [15] modifies the dynamic programming solution for the global alignment such that at any given point (i, j) if extending existing alignments at (i, j) all result in non-positive scores then a new alignment starts with score 0. That is, this solution considers 0 as an argument to the maximum operation in Eqs. (1) (in the case of simple scoring scheme), and (6) (in the case the affine gap penalties). The maximum local alignment score is achieved at any point, therefore it is expressed as $\max_{i,j} \mathcal{H}_{i,j}$.

In the case of local alignment, we modify the simulation of automaton M such that the weight of the start state never becomes negative. That is, if a simulation of an edit operation leads to a non-positive weight in the start state then the weight of the start state becomes 0. This is because a local alignment can start at any position with initial score 0

if the extensions of all existing alignments would have non-positive weights at this position. To find the maximum regular expression constrained local alignment score we keep track of the running maximum weight in final states in $M_{i,j}$'s that we compute. We output the running maximum at the end of computing all $M_{i,j}$'s. These modifications are the same in both simple scoring, and affine gap penalties cases. In the case of affine gap penalties we do not make any changes in simulating automata E, and F.

Correctness in the local alignment case follows essentially from the correctness of our algorithm in computing an optimal global alignment that satisfies a given regular expression constraint since in the local alignment case with our modification we allow an alignment to start at any point with initial score 0. A noteworthy difference between our algorithm in this case and an ordinary local alignment algorithm is that we also let a local alignment have a zero or negative score in an intermediate point during the course of satisfying the given regular expression constraint. That is we extend an alignment with 0 or negative score if it partially satisfies (matches) the given regular expression R. Automaton M allows this because active states can have zero or negative weights. This is necessary because an optimal alignment we seek has to satisfy the given regular expression constraint, and it is possible that during this process it incurs zero or negative score.

More formally, let p be an alignment with the maximum score obtained over all alignments that satisfy the given regular expression constraint. We represent p by a sequence $(i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k)$ of points in the alignment graph where the alignment passes. This sequence identifies a sequence of edit operations. For example, we describe the alignment shown in Fig. 2 by the sequence $(0, 0), (1, 1), (2, 2), (3, 3), (3, 4), (3, 5), (4, 6), (5, 7), (6, 7)$. Any two consecutive points in this sequence uniquely identify an edit operation at a given position in the alignment. For example, from $(0, 0)$ and $(1, 1)$ we know that the first edit operation on the alignment is represented by the arc $((0, 0), (1, 1))$ corresponding to the match $S_1[1] \rightarrow S_2[1]$, and $((3, 3), (3, 4))$ corresponds to the edit operation $\epsilon \rightarrow S_2[4]$. A local alignment can start at any point in the alignment graph. We imagine the given optimal alignment p in three parts: part one is from the beginning to the point preceding where the regular expression match starts, part two is where the regular expression match occurs, and part three is the rest of the alignment. For the example alignment in Fig. 2, part one contains only $(0, 0)$, and part two is $(1, 1), (2, 2), (3, 3), (3, 4), (3, 5), (4, 6)$, and part three is $(5, 7), (6, 7)$. We claim that after M_{i_k,j_k} is computed, in M_{i_k,j_k} there is a final state whose weight is the score of the given optimal alignment p. To prove this we trace the computations on parts of the given optimal alignment p. Part two in p is not empty since the alignment satisfies the given regular expression constraint which we suppose, without loss of generality, is not empty (if it is empty then the problem becomes the ordinary local sequence alignment problem which we can solve using the Smith–Waterman local alignment algorithm [15]). Part one is not empty because of our representation of alignments. Part one contains at least the beginning point of the arc that corresponds to the first edit operation in part two. For the alignment shown in Fig. 2 part one only includes $(0, 0)$. Part three is empty if any extension of p starting at the end of its part two in the direction of increasing indices in the sequences does not increase the total score.

The points $(i, 0)$'s and $(0, j)$'s on the boundary in the alignment graph can only be part of part one of an optimal alignment p. In the corresponding automata $M_{i,0}$'s and $M_{0,j}$'s (formulas (3) and (4) in the case of simple scoring, and (7) and (8) in the case of affine gap penalties), the only active state is the start state whose weight in the maximum of 0, and the total score of the corresponding alignment of a block of insertions, or deletions (0, in general, because of the negative total score of any such block in a usual setting of the parameters). Therefore, any local alignment can start at any of these boundary points with initial score 0.

Let part two start with an edit operation from (i'_0, j'_0) to (i'_1, j'_1) that initiates the matching of the regular expression R. Let $M_{i'_2,j'_2}$ be the last automaton that is generated at the end of part two. It is easy to see that, in $M_{i'_0,j'_0}$, the weight of the start state is the score of part one because of the loop on the start state on all possible edit operations in Σ^M in all copies of M. Similarly, we can see that in $M_{i'_1,j'_1}$ there is an active final state f whose score is the total of the scores obtained in parts one and two. We note that in all states including f the weights are computed using the $\max_M$ operation that involves all neighboring automata, and the states from which f can be reached (formula (5) in the simple scoring scheme, and (9) in the case of the affine gap penalties). However, from the optimality of the given alignment f we know that the weight of f in $M_{i'_1,j'_1}$ is the total score of part one and two. We also note that, this weight can be negative. This happens because any optimal alignment is required to satisfy the given regular expression constraint although this process may add negative scores for the alignments. In one extreme example, an optimal regular expression constrained local alignment has empty part three, a single point in part one (essentially empty since this only leads to part two), and part two whose score is 0 or negative. If part three exists the weight of f is increased by the score of part three. In any case, after our algorithm computes M_{i_k,j_k}, the weight of one final state in M_{i_k,j_k} is

the score of p. The running maximum in the algorithm records this value, and returns it at the end. This concludes the correctness proof of our algorithm for the local alignment case.

The auxiliary automata E, and F, and their simulations in the case of affine gap penalties, and simple modifications in automata simulations in the case of local alignment do not increase the asymptotic complexity of the problem. Our algorithms share the same time and space complexities in these cases.

7. Conclusion

We introduce the regular expression constrained sequence alignment problem, and present an algorithm for it. Our algorithm is based on constructing an automaton from the regular expression in a given constraint. It simulates copies of this automaton updating the state weights as the underlying dynamic programming solution for sequence alignment dictates. This algorithm can be adapted for various scoring schemes, and it can also be used in computing local alignments. Our algorithm guides the alignments by forcing them to contain a pattern which is described as a regular expression. A very important application is the alignment of biological sequences that includes a given motif.

Acknowledgement

We thank an anonymous referee for bringing Ref. [10] into our attention.

References

[1] A.N. Arslan, Ö. Eğecioğlu, Algorithms for the constrained common sequence problem, Int. J. Found. Comput. Sci. 16 (6) (2005) 1099–1109.
[2] P. Bork, E.V. Koonin, Protein sequence motifs, Curr. Opin. Struct. Biol. 6 (1996) 366–376.
[3] F.Y.L. Chin, N.L. Ho, T.W. Lam, P.W.H. Wong, M.Y. Chan, Efficient constrained multiple sequence alignment with performance guarantee, in: Proc. IEEE Computational Systems Bioinformatics (CSB 2003), 2003, pp. 337–346.
[4] F.Y.L. Chin, A.D. Santis, A.L. Ferrara, N.L. Ho, S.K. Kim, A simple algorithm for the constrained sequence problems, Inform. Process. Lett. 90 (2004) 175–179.
[5] J.-P. Comet, J. Henry, Pairwise sequence alignment using a PROSITE pattern-derived similarity score, Computers and Chemistry 26 (2002) 421–436.
[6] R.F. Doolittle, Similar amino acid sequences: chance or common ancestry, Science 214 (1981) 149–159.
[7] L. Falquet, M. Pagni, P. Bucher, N. Hulo, C.J. Sigrist, K. Hofmann, A. Bairoch, The PROSITE database, its status in 2002, Nucleic Acids Res. 30 (2002) 235–238.
[8] D.S. Hirschberg, Algorithms for the longest common subsequence problem, J. ACM 24 (1977) 664–675.
[9] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 1979.
[10] E.W. Myers, W. Miller, Approximate matching of regular expressions, Bull. Math. Biol. 51 (1) (1989) 5–37.
[11] C.Y. Tang, C.L. Lu, M.D.-T. Chang, Y.-T. Tsai, Y.-J. Sun, K.-M. Chao, J.-M. Chang, Y.-H. Chiou, C.-M. Wu, H.-T. Chang, W.-I. Chou, Constrained multiple sequence alignment tool development and its applications to rnase family alignment, in: Proceedings of the 1st IEEE Computer Society Bioinformatics Conference (CSB 2002), 2002, pp. 127–137.
[12] Y.-T. Tsai, The constrained common sequence problem, Inform. Process. Lett. 88 (2003) 173–176.
[13] Y.-T. Tsai, C.L. Lu, C.T. Yu, Y.P. Huang, MuSiC: A tool for multiple sequence alignment with constraint, Bioinformatics 20 (14) (2004) 2309–2311.
[14] J.E. Walker, M. Saraste, M.J. Runswick, N.J. Gay, Distantly related sequences in the alpha- and beta-subunits of ATP synthase, myosin, kinases and other ATP-requiring enzymes and a common nucleotide binding fold, EMBO J. 1 (1982) 945–951.
[15] M.S. Waterman, Introduction to Computational Biology, Chapman & Hall, 1995.