Improvements on Dynamic Programming for Bicriteria Pairwise Sequence Alignment

Maryam Abassi[1], Luís Paquete[1], and Arnaud Liefooghe[2]

[1]CISUC, Department of Informatics Engineering,
University of Coimbra, Portugal.
maryam/paquete@dei.uc.pt
[2]LIFL – CNRS – INRIA Lille-Nord Europe,
Université Lille 1, France.
arnaud.liefooghe@univ-lille1.fr

Abstract

In this article, we consider dynamic programming algorithms for solving two bicriteria formulations of the pairwise sequence alignment problem. A pruning technique based on the comparison of lower and upper bounds is introduced that reduces the number of states to be processed. The experimental results suggest that this technique leads to a pruning of 5% to 35% for the four problem variants considered in this article.

1 Introduction

Recently, there has been a growing interest on the multicriteria formulation of optimization problems that arise in computational biology (see an extensive review in Handl et al. [6]). However, exact solution approaches for most of these problems have not been thoroughly investigated. In this article we explore two bicriteria formulations of the pairwise sequence alignment problem as well as dynamic programming algorithms to solve them that extend early work of Roytberg et al. [9].

For an alignment φ of sequences $A := (a_1, \ldots, a_{n_1})$ and $B := (b_1, \ldots, b_{n_2})$, we denote by $\mathrm{m}(\varphi)$, $\mathrm{d}(\varphi)$ and $\mathrm{g}(\varphi)$, the number of matches, indels, and gaps of φ, respectively. The two following score vector functions are considered:

$$\mathrm{VMD}(\varphi) := (\mathrm{m}(\varphi), -\mathrm{d}(\varphi))$$

$$\mathrm{VMG}(\varphi) := (\mathrm{m}(\varphi), -\mathrm{g}(\varphi))$$

Two bicriteria problems that consist of finding the alignments that are "maximal" with respect to the score vector functions above are:

$$\arg\operatorname{vmax}\{\varphi : \mathrm{VMD}(\varphi),\ \varphi \in \Phi\} \tag{VMDP}$$

$$\arg\operatorname{vmax}\{\varphi : \mathrm{VMG}(\varphi),\ \varphi \in \Phi\} \tag{VMGP}$$

where Φ denotes the set of all feasible alignments. To give a proper meaning to the operator vmax in both problems, we introduce the following dominance relation between score vectors in Problem (VMDP): Given two alignments φ and φ', $\mathrm{VMD}(\varphi) > \mathrm{VMD}(\varphi')$ (φ *dominates* φ') if $\mathrm{m}(\varphi) \geq \mathrm{m}(\varphi')$, $\mathrm{d}(\varphi) \leq \mathrm{d}(\varphi')$, and $\mathrm{VMD}(\varphi) \neq \mathrm{VMD}(\varphi')$. An alignment φ is *efficient* if there exists no other alignment φ' such that $\mathrm{VMD}(\varphi') > \mathrm{VMD}(\varphi)$. The set of all efficient alignments is called *efficient alignment set*. The image of an efficient alignment in the score function space is a *nondominated score* and the set of all nondominated scores is called *nondominated score set*. The dominance relation and the above notation also apply to Problem (VMGP) with the necessary changes.

Our particular interest is to design algorithms that find the nondominated score set. Note that computing the efficient alignment set can be an intractable task: Consider the sequences $A := \mathtt{G}^{\mathtt{n}}$ and $B := \mathtt{T}(\mathtt{GT})^{\mathtt{2n}}$; then, there exists $\binom{2n}{n}$ efficient alignments that match $\mathtt{G}^{\mathtt{n}}$ in both sequences since there exists no other alignment with larger number of matches and lesser number of indels ($3n + 1$ indels). The example above also applies to the number of gaps (three gaps). However, the size of the nondominated score set is bounded by $\min(\ell, n_1 + n_2 - 2\ell)$, where ℓ is the size of longest common subsequence of A and B [9].

Interesting properties of this formulation in relation to parametric sequence alignment (see Gusfield et al. [2]) are discussed in Roytberg et al. [9]: An optimal alignment for the parametric score function with positive parameters is an efficient alignment, but there may exist efficient alignments that are not optimal for any parameter setting. Therefore, multicriteria sequence alignment brings advantages to the practitioner since it allows to get rid of parameters and to explore a tractable set of alignments that are not reachable by any other methods. However, to the knowledge of the authors, few work has been done on multicriteria sequence alignment [8–10].

In this note, we introduce a pruning technique that improves the performance of a dynamic programming algorithm for these problems [8,9], which is based on the comparison of lower and upper bounds [5] . In addition, we present numerical results on a number of randomly generated sequences.

2 Multicriteria dynamic programming

For the sake of clarity, we introduce the dynamic programming algorithm for Problem (VMDP) as proposed in Roytberg et al. [9], but with a different formulation. For a given alignment φ, we define a state $p := \mathrm{VMD}(\varphi)$. To compute the nondominated scores a matrix P is constructed where each entry $P(i,j)$, for $(i,j) \in \{1,\ldots,n_1\} \times \{1,\ldots,n_2\}$, will store the set of states corresponding to the efficient alignments of subsequences $(a_1,\ldots,a_i)$ and $(b_1,\ldots,b_j)$. The recurrence for $P(i,j)$ is as follows:

$$P(i,j) \quad := \quad \mathrm{vmax} \begin{cases} \{p + \sigma(i,j) \ : \ p \in P(i-1,j-1)\} \\ \{p + (0,-1) \ : \ p \in P(i-1,j)\} \\ \{p + (0,-1) \ : \ p \in P(i,j-1)\} \end{cases}$$

where $\sigma(i,j) := (1,0)$ if $a_i = b_j$ and $(0,0)$ otherwise, and with the bases cases $P(0,0) := \{(0,0)\}$, $P(i,0) := \{(0,-i)\}$, $P(0,j) := \{(0,-j)\}$, for $1 \leq i \leq n_1$ and $1 \leq j \leq n_2$. Operator vmax keeps only the nondominated states at entry $P(i,j)$. Roytberg et al. suggested the use of a loglinear algorithm for this operation [9]. However, this can be performed in linear time by extending the MERGE algorithm in Beier and Vöcking [1, pp. 380] for three sorted lists of nondominated scores. The overall time and space-complexity of the algorithm above is $O(n_1 \cdot n_2 \cdot (n_1 + n_2))$.

The dynamic programming algorithm for Problem (VMGP) is briefly discussed in Roytberg et al. [9]. We give a more detailed explanation of this approach, which extends the algorithm in Gusfield [3, pp. 244]. For a given alignment $\varphi := (A', B')$, we define a state $q := \mathrm{VMG}(\varphi)$. For computing the set of nondominated scores, we keep four dynamic programming matrices: Q, R, S, and T. For a given $(i,j) \in \{1,\ldots,n_1\} \times \{1,\ldots,n_2\}$, entry $R(i,j)$, $S(i,j)$ and $T(i,j)$ will store the set of states corresponding to efficient alignments of subsequences $(a_1,\ldots,a_i)$ and $(b_1,\ldots,b_j)$ that end with (a_i, b_j), $('-', b_j)$ and $(a_i, '-')$, respectively, where $'-'$ is a gap character. The entry $Q(i,j)$ will store the states corresponding to efficient alignments of subsequences $(a_1,\ldots,a_i)$ and $(b_1,\ldots,b_j)$. The recursion for the four matrices is as follows:

$$Q(i,j) \quad := \quad \mathrm{vmax} \begin{cases} R(i,j) \\ S(i,j) \\ T(i,j) \end{cases}$$

$$R(i,j) \quad := \quad \{q + \sigma(i,j) \ : \ q \in Q(i-1,j-1)\}$$

$$S(i,j) := \text{vmax}\begin{cases} S(i,j-1) \\ \{q+(0,-1) \ : \ q \in Q(i,j-1)\} \end{cases}$$

$$T(i,j) := \text{vmax}\begin{cases} T(i-1,j) \\ \{q+(0,-1) \ : \ q \in Q(i-1,j)\} \end{cases}$$

The bases cases of the matrices are: $Q(0,0) := \{(0,0)\}$, $Q(i,0) := S(i,0) := Q(0,j) := T(0,j) := \{(0,-1)\}$, for $1 \leq i \leq n_1$ and $1 \leq j \leq n_2$. Operation vmax takes also linear amount of time by using the same technique than for Problem (VMDP). Since the number of gaps is bounded from above by the number of indels in an alignment, the time and space-complexity is also $O(n_1 \cdot n_2 \cdot (n_1 + n_2))$.

3 Bounds in Problem (VMDP)

We describe a pruning technique for the dynamic programming algorithm for Problem (VMDP) that is able to reduce the number of states by comparing their upper bounds with a pre-computed lower bound set. In the following sections, we describe how these bounds are computed. The rationale is illustrated with an example.

3.1 Lower bound set

For the definition of lower bounds on the nondominated score set for Problem (VMDP), we introduce the notions of *lexicographic* and *scalarized* score functions.

We say that a vector $x \in \mathbb{R}^2$ is lexicographically larger than or equal to a vector $y \in \mathbb{R}^2$ ($x >_{\text{lex}} y$) if $x_1 > y_1$ or if $x_1 := y_1$ and $x_2 \geq y_2$. In this article, we consider two problems that consist of finding an alignment that is lexicographic maximal (lexmax) according to a given order of priority on the optimization of the two score function components:

$$\arg\text{lexmax}\left\{\varphi \ : \ (\text{m}(\varphi), -\text{d}(\varphi)) \, , \ \varphi \in \Phi\right\} \qquad \text{(LexMDP)}$$

$$\arg\text{lexmax}\left\{\varphi \ : \ (-\text{d}(\varphi), \text{m}(\varphi)) \, , \ \varphi \in \Phi\right\} \qquad \text{(LexDMP)}$$

The order of the function components indicates the priority that is considered among the criteria.

Let φ_m and φ_d be the lexicographic maximal alignments for Problems (LexMDP) and (LexDMP), respectively. Let MAX := VMD(φ_m) and MIN := VMD(φ_d). By the definition of optimality of Problem (VMDP), it

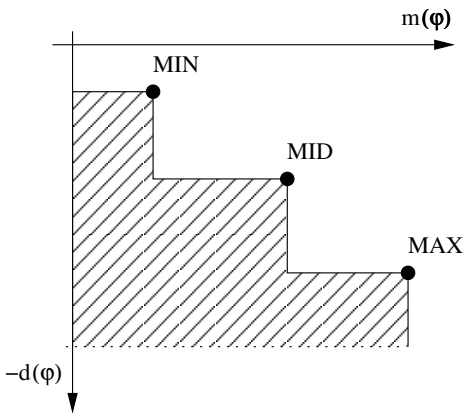

Figure 1: Illustration of the lower bound set.

holds that MAX and MIN belong to the nondominated score set [4]. Moreover, they indicate that there cannot exist an efficient alignment with more (less) matches and indels than given by the components of MAX (MIN). Hence, the two score vectors give a bound on the possible ranges of the nondominated score set. In fact, if MAX = MIN, then the nondominated score set contains only a single element and no further computation is required.

Another lower bound is given by the solution to a *scalarized* version of the bicriteria alignment problem. We consider the following weighted sum scalarization:

$$\mathrm{WMD}(\varphi) := w_m \cdot \mathrm{m}(\varphi) - w_d \cdot \mathrm{d}(\varphi)$$

where w_m and w_d are positive real weighting coefficients. The goal is to find the alignment that maximizes the scalarized score function as follows:

$$\arg\max \{\varphi \,:\, \mathrm{WMD}(\varphi),\, \varphi \in \Phi\} \qquad \text{(WMDP)}$$

Note that other scalarized functions are also possible [4]. In the particular case of the weighted sum function, the alignment that is optimal to Problem (WMDP) is also efficient to Problem (VMDP), although the opposite does not hold in general [4]. Let φ_w denote the optimal alignment for Problem (WMDP) for a given w_m and w_d and let MID := VMD(φ_w). Note that for some combinations of w_m and w_d, it may hold that MID = MIN or MID = MAX.

The score vectors MAX, MID and MIN allow to define a lower bound set on the nondominated score set of Problem (VMDP). Let $\mathcal{R}$ denote the

region

$$\mathcal{R} = \left\{ r \in \mathbb{R}_0^+ \times \mathbb{R}_0^- : \; b > r, \, b \in \{\text{MAX}, \text{MID}, \text{MIN}\} \right\}.$$

Fig. 1 illustrates the location of MAX, MID and MIN and definition of $\mathcal{R}$ (shaded area). Note that there may exist further efficient alignments whose score vectors are located in the complement of $\mathcal{R}$. However, any alignment whose score vector is in the interior of R cannot be efficient, since it would be dominated by an alignment whose score vector corresponds to MIN, MID or MAX.

The computation of the three score vectors can be performed with Needleman-Wunsch algorithm [7] by keeping the components separately in the dynamic programming matrix and choosing the state at each entry that maximizes the scalarized score function WMD, for the case of MID, or according to the lexicographic ordering for the case of MAX and MIN, respectively. Therefore, the three score vectors can be found in $O(n_1 \cdot n_2)$-time.

In the following, we only introduce the recurrence relation for computing the lexicographic maximal alignment for Problem (LexMDP), which gives the score vector MAX. We consider a dynamic programming matrix L, where the entry $L(i,j)$, for $(i,j) \in \{1,\ldots,n_1\} \times \{1,\ldots,n_2\}$, will store the state corresponding the lexicographic maximal alignment of subsequences $(a_1,\ldots,a_i)$ and $(b_1,\ldots,b_j)$. The elements of matrix L are calculated recursively by:

$$L(i,j) \quad := \quad \text{lexmax} \begin{cases} L(i-1,j-1) + \sigma(i,j) \\ L(i,j-1) + (0,-1) \\ L(i-1,j) + (0,-1) \end{cases}$$

with basis cases $L(0,0) := (0,0)$, $L(i,0) := (0,-i)$ and $L(0,j) := (0,-j)$, for $1 \leq i \leq n_1$ and $1 \leq j \leq n_2$. The operator lexmax keeps only the lexicographic maximum of the three states in the recursive step.

3.2 Upper bound

The pruning technique proposed in this note follows a branch-and-bound principle: If for a given state $s := (m,-d)$ at entry $P(i,j)$ for either $1 \leq i < n_1$ or $1 \leq j < n_2$, its upper bound $\text{ub}(s) := (m+u,-d-v)$ is located in the interior of $\mathcal{R}$, then s will not lead to any state that corresponds to a score vector of an efficient alignment. Therefore, state s can be discarded from entry $P(i,j)$.

We consider an upper bound of a state s in $P(i,j)$ that is given by the maximum number of matches u and minimum number of indels v that can be achieved from entry $P(i,j)$ to entry $P(n_1,n_2)$. The value of u can be computed by the size of the longest common subsequence of $(a_{i+1},\ldots,a_{n_1})$ and $(b_{j+1},\ldots,b_{n_2})$. This can be easily obtained for every entry in matrix P in a pre-processing step with the classical dynamic programming algorithm for computing the longest common subsequence in the reversed sequences. The minimum number of indels v is computed by the absolute difference between the sizes of two subsequences $(a_i,\ldots,a_{n_1})$ and $(b_j,\ldots,b_{n_2})$, i.e. $v := |(n_2-j)-(n_1-i)|$. Therefore, $\mathrm{ub}(s) := (m+u,-d-v)$ is a valid upper bound of state s. Note that this bound may not correspond to a feasible alignment.

Matrix L as well as the longest common subsequence for the reversed sequences can be computed in a pre-processing phase in $O(n_1 \cdot n_2)$-time. Hence, the upper bound at each matrix entry can be computed in a constant amount of time during the main phase of the algorithm.

3.3 Illustrative example

In the following, we describe an example that illustrates the pruning technique for Problem (VMDP). Let $A :=$ `AGGGCCTG` and $B :=$ `ACTAGGG`. Table 1 shows the dynamic programming matrix P where the nondominated score set is given at entry $P(8,7)$. The lower bounds are $\mathrm{MIN} := (1,0)$ and $\mathrm{MAX} := (4,-7)$; for any choice of w_m and w_g, we have that either $\mathrm{MID} := \mathrm{MIN}$ or $\mathrm{MID} := \mathrm{MAX}$ (which is due to the small-sized example). Table 2 gives the maximum number of matches and minimum number of indels for each entry. Matrix P with the pruning technique is shown in Table 3. For example, consider the state $s := (0,-6)$ at entry $P(6,0)$; then, the upper bound is $\mathrm{ub}(s) := (0+2,-6-5) := (2,-11)$, which is dominated by MAX. Therefore, this state can be pruned. In contrast, for the state $s := (1,-2)$ at entry $P(5,3)$ we have that $\mathrm{ub}(s) := (1+1,-2-1) := (2,-3)$, which is not dominated by any lower bound. Hence, in this case, state s cannot be pruned.

4 Bounds in Problem (VMGP)

For Problem (VMGP), the computation of lower and upper bounds follow the same reasoning as for Problem (VMDP). In the following, we will only give a brief explanation and highlight the main differences.

Table 1: Matrix P without pruning.

			A	C	T	A	G	G	G
		0	1	2	3	4	5	6	7
	0	(0,0)	(0,-1)	(0,-2)	(0,-3)	(0,-4)	(0,-5)	(0,-6)	(0,-7)
A	1	(0,-1)	(1,0)	(1,-1)	(1,-2)	(1,-3)	(1,-4)	(1,-5)	(1,-6)
G	2	(0,-2)	(1,-1)	(1,0)	(1,-1)	(1,-2)	(2,-3)	(2,-4)	(2,-5)
G	3	(0,-3)	(1,-2)	(1,-1)	(1,0)	(1,-1)	(2,-2)	(3,-3)	(3,-4)
G	4	(0,-4)	(1,-3)	(1,-2)	(1,-1)	(1,0)	(2,-1)	(3,-2)	(4,-3)
C	5	(0,-5)	(1,-4)	(2,-3)	(1,-2)	(1,-1)	(1,0)	(2,-1)	(3,-2)
					(2,-4)	(2,-5)	(2,-2)	(3,-3)	(4,-4)
C	6	(0,-6)	(1,-5)	(2,-4)	(2,-3)	(1,-2)	(1,-1)	(1,0)	(2,-1)
						(2,-4)	(2,-3)	(2,-2)	(3,-3)
								(3,-4)	(4,-5)
T	7	(0,-7)	(1,-6)	(2,-5)	(3,-4)	(2,-3)	(1,-2)	(1,-1)	(1,0)
						(3,-5)	(2,-4)	(2,-3)	(2,-2)
							(3,-6)	(3,-5)	(3,-4)
									(4,-6)
A	8	(0,-8)	(1,-7)	(2,-6)	(3,-5)	(4,-4)	(2,-3)	(1,-2)	(1,-1)
							(4,-5)	(2,-4)	(2,-3)
								(4,-6)	(3,-5)
									(4,-7)

4.1 Lower bound set

The lexicographic and scalarized problems described in Section 3.1 can also be formalized in terms of gaps. In this case, MAX and MIN correspond to the score vectors of the efficient alignments that maximize the number of matches and minimize the number of gaps, respectively. Also, MID corresponds to the score vector of the efficient alignment that maximizes a scalarized score function that takes into account the number of gaps (we will use w_g instead of w_d in the following to denote the weighted coefficient related to the number of gaps).

Score vectors MAX and MID can be computed in $O(n_1 \cdot n_2)$-time by using the algorithm described in Gusfield [3] with the necessary changes (see Section 3.1). However, score vector MIN can be computed faster. Note that the minimum number of gaps can only be one or zero, the latter case arising when $n_1 \neq n_2$. Assume w.l.o.g. that $n_1 < n_2$. Then, the computation of the

Table 2: Maximum number of matches and minimum number of indels for the example in Section 3.3.

			A	C	T	A	G	G	G
		0	1	2	3	4	5	6	7
	0	(4,1)	(4,2)	(4,3)	(4,4)	(3,5)	(2,6)	(1,7)	(0,8)
A	1	(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(2,5)	(1,6)	(0,7)
G	2	(3,1)	(3,0)	(2,1)	(2,2)	(2,3)	(2,4)	(1,5)	(0,6)
G	3	(3,2)	(3,1)	(2,0)	(1,1)	(1,2)	(1,3)	(1,4)	(0,5)
G	4	(3,3)	(3,2)	(2,1)	(1,0)	(0,1)	(0,2)	(0,3)	(0,4)
C	5	(3,4)	(3,3)	(2,2)	(1,1)	(0,0)	(0,1)	(0,2)	(0,3)
C	6	(2,5)	(2,4)	(2,3)	(1,2)	(0,1)	(0,0)	(0,1)	(0,2)
T	7	(1,6)	(1,5)	(1,4)	(1,3)	(0,2)	(0,1)	(0,0)	(0,1)
A	8	(0,7)	(0,6)	(0,5)	(0,4)	(0,3)	(0,2)	(0,1)	(0,0)

maximum number of matches that is possible for one gap can be performed by counting the number of matches for each of the $n_1 + 1$ possible locations of a gap. This can be performed in $O({n_2}^2)$-time in an incremental manner.

4.2 Upper bound

In this problem, the computation of the maximum number of matches that can be achieved in entry $Q(n_1, n_2)$ by a state s at entry $Q(i, j)$ follows the same procedure as explained in Section 3.2.

For the computation of the minimum number of gaps, we consider a partition of matrix Q into three sections; w.l.o.g, we assume that $n_1 < n_2$. Let $Q^D := Q(i - n_2 + n_1, i)$, for $n_2 - n_1 \leq i \leq n_2$, which corresponds to the diagonal in Q starting at $Q(0, n_2 - n_1)$ and ending at $Q(n_1, n_2)$. Let Q^A and Q^B denote the entries in matrix Q that are located above and below Q^D, respectively. From this partitioning of matrix Q, we can derive the following results for v, the minimum number of gaps, that is achieved at $Q(n_1, n_2)$ by a state s (for the sake of the explanation, we relate state s with a partial alignment $\varphi := (A', B')$):

i) If state $s \in Q^D$, then $v := 0$;

ii) If state $s \in Q^A$ (Q^B) and alignment φ ends with a gap in A' (B'), then $v := 0$;

Table 3: Matrix P with pruning.

			A	C	T	A	G	G	G
		0	1	2	3	4	5	6	7
	0	(0,0)	(0,-1)	(0,-2)	(0,-3)	-	-	-	-
A	1	(0,-1)	(1,0)	(1,-1)	(1,-2)	(1,-3)	-	-	-
G	2	(0,-2)	(1,-1)	(1,0)	(1,-1)	(1,-2)	(2,-3)	-	-
G	3	(0,-3)	(1,-2)	(1,-1)	(1,0)	(1,-1)	(2,-2)	(3,-3)	-
G	4	(0,-4)	(1,-3)	(1,-2)	(1,-1)	(1,0)	(2,-1)	(3,-2)	(4,-3)
C	5	-	(1,-4)	(2,-3)	(1,-2)	(1,-1)	(1,0)	(2,-1)	(3,-2)
					(2,-4)	(2,-5)	(2,-2)	(3,-3)	(4,-4)
C	6	-	-	(2,-4)	(2,-3)	(2,-4)	(1,-1)	(1,0)	(2,-1)
							(2,-3)	(2,-2)	(3,-3)
								(3,-4)	(4,-5)
T	7	-	-	-	(3,-4)	(2,-3)	(2,-4)	(1,-1)	(1,0)
								(2,-3)	(2,-2)
								(3,-5)	(3,-4)
									(4,-6)
A	8	-	-	-	-	(4,-4)	(2,-3)	(2,-4)	(1,-1)
							(4,-5)	(4,-6)	(2,-3)
									(3,-5)
									(4,-7)

iii) If state $s \in Q^A$ (Q^B) and alignment φ ends with two characters or a gap character in B' (A'), then $v := 1$.

Note that conditions ii and iii) can be determined by keeping an additional variable that stores whether state s was obtained from matrix R, S, or T. Therefore, the upper bound for the case of gaps can also be computed in constant amount of time.

5 Experimental analysis

The two algorithms were implemented, with and without the pruning technique (`Prune` and `NoPrune`, respectively). The implementations were run in a wide benchmark of randomly generated sequences. The size of sequences ranged from $n = 300$ to 2100. For realistic purpose, two alphabet sizes were

Table 4: Experimental results for Problem (VMDP) with $\alpha = 4$.

		NoPrune		Prune	
n	$\#nond$	$time$	$\#states$	$time$	$\%prun$
300	52.9	0.0	0.7×10^6	0.0	35.2%
600	104.5	0.3	5.6×10^6	0.3	30.3%
900	155.9	1.6	18.7×10^6	1.0	29.1%
1200	208.6	5.1	43.5×10^6	3.2	28.8%
1500	258.0	10.8	83.0×10^6	7.5	27.9%
1800	309.6	19.3	143.8×10^6	14.2	27.2%
2100	360.7	30.8	226.8×10^6	21.1	28.5%

Table 5: Experimental results for Problem (VMDP) with $\alpha = 20$.

		NoPrune		Prune	
n	$\#nond$	$time$	$\#states$	$time$	$\%prun$
300	64.0	0.1	1.1×10^6	0.0	21.9%
600	127.7	0.6	8.8×10^6	0.5	18.0%
900	191.2	3.0	29.0×10^6	2.5	17.6%
1200	256.7	8.5	68.1×10^6	7.1	16.6%
1500	316.5	16.9	132.4×10^6	13.9	18.3%
1800	382.2	30.7	228.8×10^6	26.1	17.2%
2100	443.6	47.8	360.7×10^6	40.8	17.4%

considered: $\alpha = 4$ and 20. A total of 30 instances were generated for each combination of values of n and α. The implementations were coded in `C++` and compiled with `g++` version 4.2.4 with the `-O3` compiler option, in a computer cluster with 18 nodes, each with an AMD Phenom II X6 processor with 3.2 GHz, 3 and 6 MB L2 and L3 Cache, respectively, and 12 GB DDR3 SDRAM, with operating system Ubuntu 8.04 LTS. Except of the compiler option, no other code optimization technique was used in the experiments.

Preliminary experiments indicated that three bounds were insufficient for obtaining good performance for Problem (VMGP). For this reason, several weighted sum problems were solved for different weights combinations in order to obtain a tighter lower bound set. A reasonable good trade-off between the time spent on the pre-processing phase and the time spent on comparing bounds was achieved with the number of bounds equal to 12, with

Table 6: Experimental results for Problem (VMGP) with $\alpha = 4$.

		NoPrune		Prune	
n	$\#nond$	$time$	$\#states$	$time$	$\%prun$
300	55.7	0.6	9.1×10^6	0.5	21.9%
600	111.2	6.8	71.4×10^6	5.5	20.5%
900	166.6	27.2	238.8×10^6	20.6	20.4%
1200	220.9	64.5	564.3×10^6	50.2	20.2%
1500	275.6	128.8	$1\,094.7\times10^6$	111.2	20.2%
1800	333.1	233.8	$1\,893.9\times10^6$	200.9	20.0%
2100	386.2	423.1	$3\,000.2\times10^6$	333.5	20.2%

Table 7: Experimental results for Problem (VMGP) with $\alpha = 20$.

		NoPrune		Prune	
n	$\#nond$	$time$	$\#states$	$time$	$\%prun$
300	56.7	0.6	9.3×10^6	0.6	7.1%
600	113.3	7.1	73.2×10^6	7.2	6.4%
900	171.5	27.0	247.7×10^6	27.0	6.0%
1200	228.6	63.6	587.8×10^6	63.3	6.0%
1500	282.9	127.4	$1\,139.2\times10^6$	137.7	6.0%
1800	341.5	230.6	$1\,977.4\times10^6$	250.0	6.0%
2100	398.9	418.3	$3\,134.1\times10^6$	416.4	5.7%

weights varying in the following manner: $w_m := i$, $w_g := 13-i$, $i = 1, \ldots, 12$.

Tables 4 and 5 present the experimental results for Problem (VMDP) with $\alpha = 4$ and 20, respectively, where $\#nond$ corresponds to the number of nondominated states, $time$ gives the CPU-time in seconds to terminate, $\#states$ is the total number of states that were generated by `NoPrune` version and $\%prun$ corresponds to the percentage of states that are pruned in the `Prune` version. The values are averaged over 30 instances of the same size.

The results show that the `Prune` version is able to prune between 17% and 35% of the states that are generated by the `NoPrune` version in Problem (VMDP). This has a direct consequence on CPU-time. It is noteworthy to mention that the number of nondominated states for sequences of size n is close to $n/5$.

Tables 6 and 7 give the results for Problem (VMDP). The difference

between the two versions is less noticeable in Problem (VMGP). Although the `Prune` version is able to prune approximately 20% of the states for instances with $\alpha = 4$, only 6% of pruning is possible to achieve for $\alpha = 20$. In fact, in latter case, the CPU-time of the `Prune` version is slightly larger than that of the `NoPrune` version, which suggests that this technique brings additional overhead when $\alpha = 20$ is considered. Similar results were observed for different number of bounds. A possible explanation for the lack of performance of the `Prune` version for Problem (VMGP) is given by the large number of efficient alignments that have the same score value, which results in a less effective pruning.

6 Concluding remarks

This article introduces a new technique to improve the performance of dynamic programming algorithms for bicriteria pairwise sequence alignment. This technique uses lower and upper bounds to discard states in early stages of the process. The experimental results indicate that is possible to obtain an improvement between 35% and 20% in terms of pruning when a score function of matches and indels is considered. In addition, an improvement of 20% can be obtained when gaps are considered and the alphabet size is small. However, the technique does not pay-off in terms of CPU-time in the latter case when the alphabet size is larger, which indicates that further investigation on more effective pruning is still needed.

This technique can be easily extended for three and four criteria case, where the number of matches, indels, gaps and mismatches are considered simultaneously in the score function. Other scenario where this pruning technique can be used is in the multicriteria multiple sequence alignment, possibly within some heuristic approach.

A final question of theoretical interest is whether it is possible to devise an algorithm for these two problems that is output-sensitive, that is, it takes $O(n_1 \cdot n_2 \cdot k)$, where k is the cardinality of the nondominated score set. Our approach fails to reach this bound since the number of elements at each entry is not monotonically decreasing in the dynamic programming matrix.

Acknowledgment The authors acknowledge Maria C. Dias for an early discussion on the main topic of this note. This work was supported by the Portuguese Foundation for Science and Technology under the project "*Multiobjective Sequence Alignment*" (PTDC/EIA-CCO/098674/2008).

References

[1] R. Beier and B. Vöcking. The knapsack Problem. In B. Vöcking et al. (eds), Algorithms Unplugged, Springer, pp. 375-381, 2011.

[2] D. Gusfield, K. Balasubramanian, D. Naor. Parametric optimization of sequence alignment. In Proceedings of the third annual ACM-SIAM Symposium on Discrete Algorithms, pages 432-439, 1992.

[3] D. Gusfield. Algorithms on strings, trees and sequences: Computer science and computational biology, Cambridge University Press, 1999.

[4] M. Ehrgott. Multicriteria optimization. Springer, 2005.

[5] M. Ehrgott, X. Gandibleux. Bound sets for biobjective combinatorial optimization problems. Computers & Operations Research, 34(9):2674–2694, 2007.

[6] J. Handl, D.B. Kell, and J. Knowles. Multiobjective optimization in bioinformatics and computational biology. IEEE/ACM Transactions on Biology and Bioinformatics, 4(2):279-292, 2007.

[7] B.S. Needleman, C.D. Wunsch, 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.

[8] L. Paquete, J.P.O. Almeida, Experiments with bicriteria sequence alignment, Cutting-Edge Research Topics on Multiple Criteria Decision Making, Communications in Computer and Information Science, 35, pp. 45-51, Springer, 2009.

[9] M.A. Roytberg, M.N. Semionenkov, O.Y. Tabolina. Pareto-optimal alignment of biological sequences. Biophysics, 44(4):565-577, 1999.

[10] A. Taneda, Multi-objective pairwise RNA sequence alignment. Bioinformatics, 26(19):2383-90, 2010.