Exact Multiple Sequence Alignment by Synchronized Decision Diagrams

Amin Hosseininasab and Willem-Jan van Hoeve
Tepper School of Business, Carnegie Mellon University, USA.
aminh@andrew.cmu.edu, vanhoeve@andrew.cmu.edu

This paper develops an exact solution algorithm for the Multiple Sequence Alignment (MSA) problem. In the first step, we design a dynamic programming model and use it to construct a novel Multi-valued Decision Diagrams (MDD) representation of all pairwise sequence alignments (PSA). PSA MDDs are then synchronized using side constraints to model the MSA problem as a Mixed-Integer Program (MIP), for the first time, in polynomial space complexity. Two bound-based filtering procedures are developed to reduce the size of the MDDs, and the resulting MIP is solved using logic-based Benders decomposition. For a more effective algorithm, we develop a two-phase solution approach. In the first phase we use optimistic filtering to quickly obtain a near-optimal bound, which we then use for exact filtering in the second phase to prove or obtain an optimal solution. Numerical results on benchmark instances show that our algorithm solves several instances to optimality for the first time, and in case optimality cannot be proven, considerably improves upon a state-of-the-art heuristic MSA solver. Comparison to an existing state-of-the-art exact MSA algorithm shows that our approach is more time efficient and yields significantly smaller optimality gaps.

Key words: Keywords: Multiple sequence alignment; Multi-valued decision diagrams; Decision diagram filtering; Optimistic filtering; Logic-based Benders decomposition.

1. Introduction

Sequence alignment is a fundamental problem in computational biology, but also finds application in other fields such as high frequency trading, speech recognition, and computer vision. The problem involves comparing a number of *sequences*, with the objective of finding their best *alignment*. A sequence s is defined as a string of characters $s := \left\langle c^1, \ldots, c^{|s|} \right\rangle$ with a meaningful order, for example, representing a DNA, RNA, or protein molecule in the context of computational biology. The optimal alignment of a set of sequences $\mathbb{S}$ is obtained by inserting gaps "$-$" into each sequence $s \in \mathbb{S}$, such that all sequences achieve the same length and their similar regions overlap.

The Pairwise Sequence Alignment (PSA) problem consists of finding the best alignment between a pair of sequences (s, s'), and as a generalization, the Multiple Sequence Alignment (MSA) problem involves aligning a set $\mathbb{S} \geq 3$ of sequences. The output of the sequence alignment problem is a matrix with $|\mathbb{S}|$ rows, where each row corresponds to a sequence $s \in \mathbb{S}$, and each column corresponds to a set of aligned characters. Two characters $c_s^i, c_{s'}^j$, or a character c_s^i and a gap symbol "$-$", are aligned if they are placed in the same column of the MSA matrix. An alignment is feasible if and only if it satisfies the following two requirements:

1. Order requirement: Ignoring the gap characters "$-$", the row associated with a sequence $s \in \mathbb{S}$ must be identical to sequence s, and preserve the order of its characters $\left\langle c_s^1, \ldots, c_s^{|s|} \right\rangle$.
2. Column alignment requirement: Each set of aligned characters belongs to exactly one column, and any column must contain at least one character $c_s^i \in s, s \in \mathbb{S}$.

A feasible alignment is optimal if it gives the highest objective value with respect to the *scoring function*. The scoring function assigns a reward for matching two characters, and penalizes alignments of characters to gaps. The reward of aligning different characters is determined by a predefined *substitution matrix*. Different substitution matrices result in different optimal solutions, and are used to accommodate various user objectives.

The penalty of aligning characters to gaps is primarily dependent on the user objective, and may be a constant, or a function of the number of consecutive gap alignments (Vingron and Waterman 1994). In affine or convex gap penalty functions, an opening gap alignment, i.e., a gap alignment immediately after an alignment of characters, is penalized by the *opening* penalty. An extension of a gap alignment is thereafter penalized by the *extension* penalty, which is a linear function in affine, or a convex function in convex gap penalties.

To obtain the overall alignment score for MSA, a popular technique is the *Sum-of-Pairs* objective template. In this template, the objective value is the sum of the rewards or penalties assigned to all pairs of aligned characters and gaps. Note that a pair of aligned gaps is not assigned any reward or penalty. Regardless of the scoring template, solving MSA is proven to be NP-complete (Wang and Jiang 1994), and is also challenging in practice. Thus, the majority of research in solving MSA is dedicated to heuristic algorithms (Katoh et al. 2017, Hung et al. 2015, Sievers et al. 2011, Chakraborty and Bandyopadhyay 2013, Magis et al. 2014). Although heuristic methods are fast and accommodate large-size problems, they fail to give any guarantee on the solution quality. In fact, the quality of heuristic solutions has been shown to be often far below the optimal solution on benchmark instances (Thompson et al. 2011, Nuin et al. 2006). As higher quality alignments offer significantly more insight into relationship of sequences, it is desirable to develop algorithms with higher accuracy. This motivates the development of algorithms that attempt to solve MSA exactly and guarantee optimality.

To the best of our knowledge, three exact solution approaches exist for MSA. Namely, the Dynamic Programming (DP) approach introduced by Needleman and Wunsch (1970); the Integer Programming (IP) approach of the Maximum Weight Trace (MWT) problem introduced by Kececioglu (1993) and solved exactly by Althaus et al. (2006); and the convex optimization approach of Yen et al. (2016). Although exact approaches may be slow compared to heuristics, they are guaranteed to reach the optimal solution eventually, even for large-size problems. However, a major bottleneck of existing exact MSA algorithms is their exponential space requirement, which is generally too high to allow the solution algorithm to fit in memory for large-size problems.

The DP algorithm models the MSA problem as a $|\mathbb{S}|$ dimensional cube of size $n = \underset{s \in \mathbb{S}}{max}\{|s|\}$, with a memory requirement of $O(n^{|S|})$ which is too space consuming to even model small sized instances. The convex optimization approach also has exponential space requirements, and is limited to solving small-sized sequences of length below 50 in practice. Similarly, the MWT approach has exponential space requirements, with a worst-case $O(|\mathbb{S}|n^{|\mathbb{S}|})$ space complexity.

The most successful exact approach for MSA is the branch-and-cut algorithm developed for the MWT problem (Althaus et al. 2006). Althaus et al. (2006) consider small to medium sized problems with 4 to 6 sequences and a total number of 300 to 600 characters. The branch-and-cut algorithm works well on benchmark instances with high degrees of similarity, and is able to solve those to optimality within a short amount of time. However, the algorithm does not converge for instances with lower degrees of similarity within 10 hours of computation, and terminates with 16-580% gaps to optimality. For such instances, heuristic solutions with good quality are generated by optimizing over a carefully chosen subset of the feasible solutions.

In this paper, we follow the Sum-of-Pairs template and develop an algorithm for global sequence alignment (Needleman and Wunsch 1970). As in most MSA algorithms, our algorithm may accommodate local alignment (Smith and Waterman 1981) or any objective template using straightforward modifications.

In the first step, we develop a DP model and use it to construct a novel representation of the MSA problem using Multi-valued Decision Diagrams (MDD). Decision diagrams were originally used for circuit design problems as a compact graphical representations of Boolean functions (Lee 1959, Bryant 1986). In recent years, decision diagrams have been successfully used to represent the set of feasible solutions of discrete optimization problems (Becker et al. 2005, Bergman et al. 2014, 2016, Cire and van Hoeve 2013, Darwiche and Marquis 2004, Bouquet and Jégou 1995, Cayrol et al. 1998).

An MDD approach to MSA possesses a number of advantages not available for existing DP, IP, or convex optimization approaches. Using the order preserved in MDDs, the structure of sequences may be exploited to remove a set of infeasible alignments prior to optimization. This is not case in the MWT algorithm for example, which identifies and removes such solutions iteratively by branch and cut on a linear program (Reinert et al. 1997, Althaus et al. 2006, Kececioglu et al. 2000). Using the MDD representation of PSA of all pairs $(s, s') : s, s' \in \mathbb{S}$, we model the MSA problem in worst-case $O(|\mathbb{S}|^2 n^3)$ space for affine gap penalty functions and $O(|\mathbb{S}|^2 n^4)$ for convex gap penalty functions, while existing exact approaches require a worst-case exponential space. This is a significant improvement, as it allows to model and potentially solve larger sized problems (in particular for affine penalty functions), even though they may not be solved to optimality in a reasonable amount of time.

We consider the MSA problem as the synchronized PSA of all pairs $(s, s') : s, s' \in \mathbb{S}$. More specifically, PSA MDDs are used as the underlying network structure to formulate the MSA problem as a Mixed-Integer Program (MIP), and their solution is synchronized using side constraints. Transforming DP models into an equivalent MIP formulation was introduced by Martin (1987), Martin et al. (1990), and shown to be an effective modeling approach. The resulting MIP in our paper is a collection of longest path problems with side constraints, and hard to solve in general.

To lower the solution difficulty, we first attempt to reduce the size of the feasible solution set by removing non-optimal solutions embedded in the PSA MDDs. We develop two bound-based filtering procedures using the Carrillo and Lipman (1988) lower bound, and an upper bound based on linear programming strengthened through additive bounding (Fischetti and Toth 1989). To the best of our knowledge, this is the first upper bounding procedure used to prune infeasible or non-optimal solutions in the MSA problem. We show that these filtering procedures complement each other, and in particular, additive bounding filtering is able to prune a significant number of solutions even after the Carrillo-Lipman filtering procedure is used. The MSA MIP is defined on the filtered PSA MDDs, and solved using a logic-based Benders decomposition algorithm (Hooker and Ottosson 2003).

For a more effective solution algorithm, we develop a two phase approach. In the first phase, we aggressively filter the PSA MDDs using an optimistic guess for the optimal solution objective. This allows us to optimize over a considerably smaller solution set and quickly find quality solutions (in most considered instances, optimal or near-optimal). Using the bound obtained by the optimal solution of the first phase, we exactly filter the PSA MDDs in the second phase to prove optimality of the first phase solution, or find an optimal solution.

We show that our algorithm is able to solve several benchmark instances to optimality (with respect to our considered objective functions). For instances not solved withing the imposed time limit, a primal heuristic embedded in the Benders decomposition algorithm generates feasible MSA solutions, which significantly improve alignment accuracy compared to a state-of-the-art MSA heuristic solver. The solution algorithm is compared to the best exact approach of the literature, and shown to optimally solve or reach lower gaps to optimality on almost all considered instances.

The rest of the paper is structured as follows. Section 2 models the PSA problem as a DP model, and uses that model to represent the PSA problem as an MDD. Section 3 shows how to synchronize PSA MDDs to model the MSA problem as an MIP, discusses the filtering procedures used to prune non-optimal solutions, and designs the two phase solution algorithm. In section 4, we develop the logic-based Benders decomposition and primal heuristic algorithms. Section 5 gives the numerical results on benchmark instances, and the paper is finally concluded in Section 6.

2. A DP model and MDD representation for the PSA problem

In this section, we develop a DP model for the PSA of sequences $s, s' \in \mathbb{S}$. Our model is similar to the matrix-format DP model of Needleman and Wunsch (1970) (in particular when gap penalties are linear); however, our DP model is better suited to develop an MDD representation of PSA. For clarity, we first define the DP model for the case of linear gap penalty functions, and thereafter expand the model to affine and convex settings.

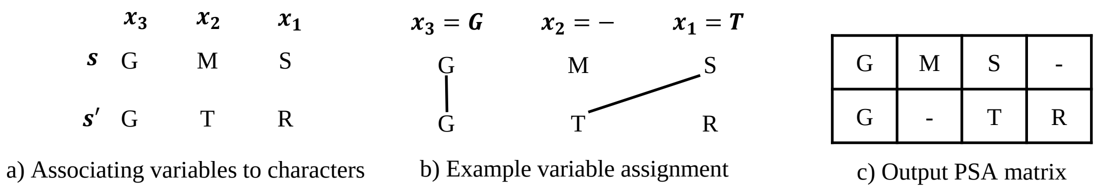
x_3 x_2 x_1
s G M S
s' G T R
a) Associating variables to characters
$x_3 = G$ $x_2 = -$ $x_1 = T$
G M S
G T R
b) Example variable assignment
G M S -
G - T R
c) Output PSA matrix

Figure 1 **Example variable association and assignment, and corresponding output PSA matrix.**

2.1. Dynamic programming model for the PSA problem

For a PSA alignment of sequences (s, s'), we define x as a tuple of discrete variables $(x_1, \ldots, x_i, \ldots, x_{|s|})$, one for each character $c_s^i : 1 \le i \le |s|$. Variables x_i are associated with finite domains D_i, defined as the possible alignment decisions for c_s^i, i.e., $D_i = \left\{c_{s'}^1, \ldots, c_{s'}^{|s'|}, -\right\}$. Assigning x_i to a value in D_i represents aligning character c_s^i to some character $c_{s'}^j, 1 \le j \le |s'|$, or to the gap symbol "$-$". For example, for the PSA of the sequences in Figure 1.a, we define three variables $x_1, \ldots, x_3$ with the domains $D_i = \{G, T, R, -\}, i = 1, \ldots, 3$. A PSA solution is thus represented by a feasible assignment to tuple x. For example, $\bar{x} = (\bar{x}_1, \bar{x}_2, \bar{x}_3) = (T, -, G)$ represents the alignment made in Figure 1.b, with the output alignment matrix shown in Figure 1.c.

To determine the optimal assignment for x, we solve the PSA problem using a DP with $i = 1, \ldots, |s|$ stages, where at each stage i, a decision is made for the value of x_i. The model is formulated by backwards induction, i.e., stages $i = 1, \ldots, |s|$ (consequently variables $x_1, \ldots, x_{|s|}$) correspond to the alignment decision for characters $c_s^{|s|}, \ldots, c_s^1$, respectively. The PSA DP model has the following elements:

- state spaces $\mathcal{S}_i$, for $i = 0, \ldots, |s| + 1$;
- transition functions $\mathcal{T}_i : \mathcal{S}_{i-1} \times D_i \to \mathcal{S}_i$, for $i = 1, \ldots, |s|$;
- transition reward functions $\mathcal{R}_i : \mathcal{S}_{i-1} \times \mathcal{S}_i \to \mathbb{R}$, $i = 1, \ldots, |s| + 1$;

and is written as:

$$\max_{z,x} \quad \sum_{i=1,\ldots,|s|+1} \mathcal{R}_i(z_{i-1}, z_i) \tag{1}$$

$$\text{s.t.} \quad z_i = \mathcal{T}_i(z_{i-1}, x_i) \qquad i = 1, \ldots, |s|, \tag{2}$$

$$z_0 = |s'| + 1, z_{|s|+1} = 0 \tag{3}$$

$$z_i \in \mathcal{S}_i, x_i \in D_i \qquad i = 1, \ldots, |s|. \tag{4}$$

The alignment variables $x_1, \ldots, x_{|s|}$ are regarded as controls, where a control x_i takes the DP system from state $z_{i-1} \in \mathcal{S}_{i-1}$ to state $z_i = \mathcal{T}_i(z_{i-1}, x_i)$ and accumulates a reward/penalty

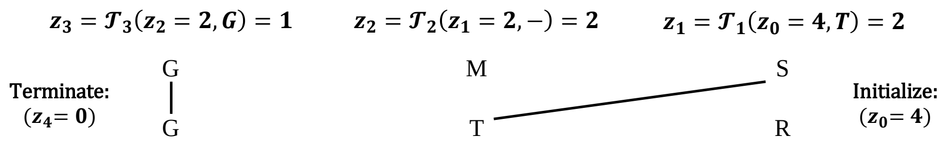
$z_3 = \mathcal{T}_3(z_2 = 2, G) = 1$
$z_2 = \mathcal{T}_2(z_1 = 2, -) = 2$
$z_1 = \mathcal{T}_1(z_0 = 4, T) = 2$
Terminate: ($z_4 = 0$)
G
G
M
T
S
R
Initialize: ($z_0 = 4$)

Figure 2 **Example DP transition function values.**

$\mathcal{R}_i(z_{i-1}, z_i)$. We define states z_i as the position j of the last non-gap alignment made to a character $c_{s'}^j$. That is, $z_i = j$, where $c_{s'}^j$ is the last non-gap alignment decision for a character $c_s^{i'}, i' \leq i$. Following this definition, the transition functions are defined as:

$$\mathcal{T}_i(z_{i-1}, x_i) = \begin{cases} j & \text{if } x_i = c_{s'}^j \text{, and} \\ z_{i-1} & \text{if } x_i = -. \end{cases}$$

The DP model is initiated from state $z_0 = |s'| + 1$ and terminated at state $z_{|s|+1} = 0$. For example, Figure 2 shows the transition values for the example PSA of Figure 1.b.

PSA feasibility is satisfied by ensuring that $z_i \leq z_{i-1}$ for any transition $z_i = \mathcal{T}_i(z_{i-1}, x_i)$, as proved in Lemma 1. State spaces $\mathcal{S}_i$ are defined as the set of possible state values for z_i at stage i of the DP model. Therefore, $\mathcal{S}_0 = \{|s'| + 1\}$, $\mathcal{S}_{|s|+1} = \{0\}$, and $\mathcal{S}_i = \{|s'| + 1, \ldots, 1\}$ for all $1 \leq i \leq |s|$.

Lemma 1 *A transition from a state z_{i-1} to a state z_i is feasible if and only if $z_i \leq z_{i-1}$.*

Proof The "only if" direction holds due to the well-known Markovian property of the DP solution for PSA. The "if" direction holds as any alignment such that $j = z_{i-1} > z_i = j'$ implies an alignment of character $c_s^{i'}, i' \leq i - 1$ to $c_{s'}^j$, and c_s^i to $c_{s'}^{j'}, j > j'$, and violates the order requirement. $\square$

The transition reward function $\mathcal{R}(z_{i-1}, z_i)$ is designed to accumulate a gap penalty if $z_i = z_{i-1}$. To model character alignment rewards, we first point out that the proposed DP model explicitly determines gap alignment decisions for characters c_s^i only, and it does not explicitly determine the gap alignment decisions for characters $c_{s'}^j$. Gap alignments for characters $c_{s'}^j$ are however implied by non-gap alignment decisions for characters c_s^i. For example, in Figure 3 the alignment of $c_s^4 = M$ implies that characters $c_{s'}^3, c_{s'}^4, c_{s'}^5$ are aligned with gap symbols. To capture the penalty of such implied gap alignments, $\mathcal{R}(z_{i-1}, z_i)$ is designed to accumulate a gap penalty of a length $z_i - z_{i-1} - 1$ gap, in addition to the explicit character alignment reward for characters c_s^i.

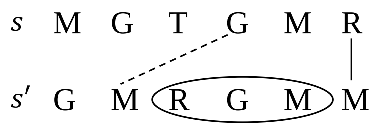

Figure 3 The alignment shown by the dashed arc implies gap alignments of characters $c_{s'}^{j}, j = 3, 4, 5$ (circled).

2.2. Extension to affine and convex gap penalty functions

The state definition z_i of the previous section is only sufficient for a linear gap penalty function. To accommodate more complex penalty functions such as affine or convex, we are required to define additional state values for variables z_i. In particular gap opening and extension penalties are unequal in such functions. Therefore, to correctly penalize gap alignments at stage i of the DP model, we are required to distinguish whether a gap or a non-gap alignment is made at stage $i-1$. In case a non-gap alignment is made at stage $i-1$, a gap alignment at stage i is penalized by the opening gap penalty. Similarly, if a gap alignment is made at stage $i-1$, an extension gap penalty is incurred for gap alignments at stage i.

To distinguish gap and non-gap alignments at each stage, we introduce negative state values $z_i < 0$, and associate them to gap alignments. That is, a negative state $z_i < 0$ denotes that a gap alignment is made at stage $i-1$. In this setting, a character alignment is modeled by a transition from a positive or negative state value z_{i-1}, to a positive state value $z_i > 0$. Opening gap alignments are modeled by a transition from a positive state value $z_{i-1} > 0$ to a negative state value $z_i < 0$; and gap extension alignments are modeled by transition between negative state values $z_i < 0, z_{i-1} < 0$. Similar to before, a transition $z_i = \mathcal{T}_i(z_{i-1}, x_i)$ is feasible (permitted) if and only if $0 \leq z_i < |z_{i-1}|$ for character alignments, or $z_i = -|z_{i-1}|$ for gap alignments.

The state spaces $\mathcal{S}_i$ for the DP model with affine penalty functions are defined as follows. For stage $i = 1$, we define $\mathcal{S}_1 = \{-(|s'|+1), 1, \ldots, |s'|\}$, where a state $1 \leq z_1 \leq |s'|$ indicates that $c_s^{|s|}$ is aligned to character $c_{s'}^{z_1}$; and state $z_1 = -(|s'|+1)$ indicates an opening gap alignment for $c_s^{|s|}$. Note that gap extension alignments are not possible at stage $i = 1$. For stages $i = 2, \ldots, |s|$, we define $\mathcal{S}_i = \{-(|s'|+1), -|s'|, \ldots, -1, 1, \ldots, |s'|\}$, with a negative state value $-|s'| \leq z_i \leq -1$ representing a gap alignment at stage $i-1$. Figure 4.a. shows example transitions (represented by arcs) between different state values z_i (represented by nodes) for affine penalty functions.

Unlike an affine penalty function where gap extension penalties incurred at stage i are equal regardless of the number of consecutive gap alignments prior to stage i, a convex

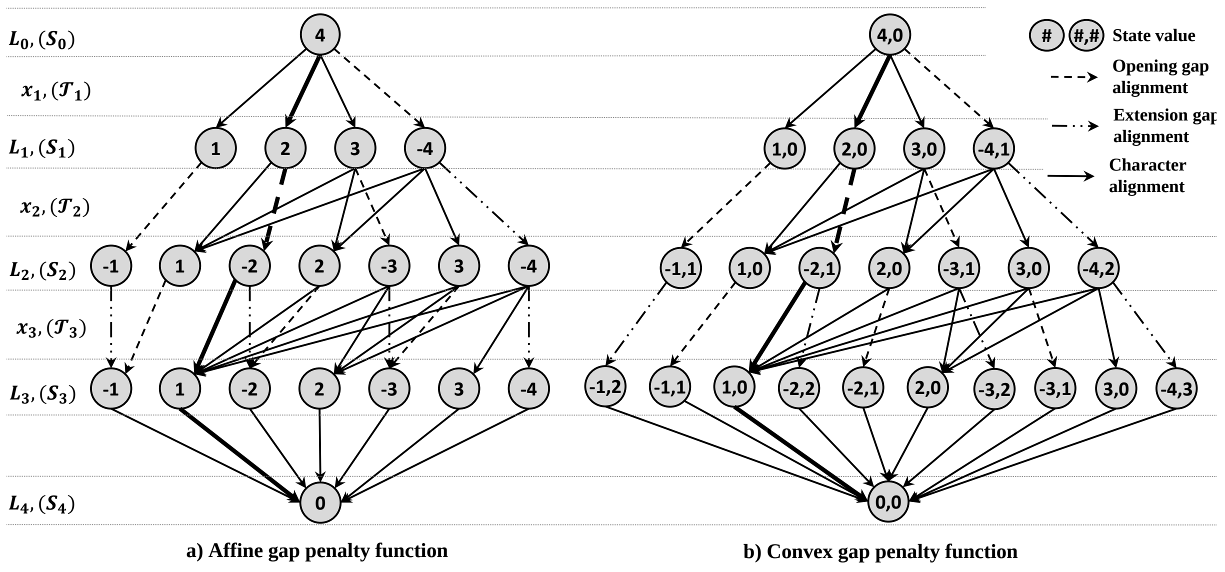

Figure 4 A PSA MDD with layers $\mathcal{L}_i$ corresponding to state spaces $\mathcal{S}_i$, and variables x_i (transitions $\mathcal{T}_i$) corresponding to arcs $(u, v) \in A^p$. The highlighted path corresponds to an alignment $(\bar{x}_1, \bar{x}_2, \bar{x}_3) = (c^1_{s'}, -, c^2_{s'})$

.

penalty function gives different penalties when extending gaps of different lengths. To accommodate a convex penalty function we must thus keep track of the number of consecutive gap alignments prior to stage i, in addition to representing gap states by negative values. This is done by redefining state variables z_i from single values j, to size-two tuples (j, g). The definition of values j is consistent with the case of affine penalty functions, and the newly introduced state gap length value g denotes the number of consecutive gap alignments made prior to state z_i. Using values (j, g) we can correctly calculate the cost of convex gap penalties in $\mathcal{R}_i(z_{i-1}, z_i)$.

To define the state spaces $\mathcal{S}_i$ for convex gap penalties, we first define $\mathcal{S}_i = \{(-|s'| - 1, i), (1, 0), \ldots, (|s'|, 0),\}, 1 \leq i \leq |s'|$, which denote character alignments by $z_i = (j, 0) : j > 0$, and the possibility of no character alignment (i consecutive gap alignments) before stage i by $z_i = (-|s'| - 1, i)$. To represent gap alignment states, we add state values $\{(-1, i'), \ldots, (-|s'|, i'),\}$ to $\mathcal{S}_i$ for every value of $i' \in \{1, \ldots, i\}$, as there are a possible i' number of possible values for g at stage i. Figure 4.b shows example transitions (represented by arcs) between different state values z_i (represented by nodes) for a convex penalty functions.

This concludes the definition of the proposed DP model for PSA. The next section uses the designed DP model to represent PSA as a MDD.

2.3. MDD representation for the PSA problem

This section uses the PSA DP model to represent the PSA of a pair of sequences $p = (s, s')$ as an MDD. For succinctness, we mainly discuss the MDD representation for the DP model with affine penalty functions. The extension to convex penalty functions is straightforward.

A PSA MDD $\mathcal{M}^p = (U^p, A^p)$ is a layered directed acyclic graph, where U^p is its set of nodes, and A^p is its set of arcs. Set U^p is partitioned into layers $(\mathcal{L}_0, \ldots, \mathcal{L}_{|s|+1})$, such that layers $\mathcal{L}_i : 1 \le i \le |s|$ correspond to stages $1 \le i \le |s|$ of the DP model, and contain one node per value of state spaces $\mathcal{S}_i$. For example, Layers $\mathcal{L}_0$ and $\mathcal{L}_{|s|+1}$ consist of single nodes, namely the root node $r \in \mathcal{L}_0$ $(z_0 = |s'| + 1)$, and the terminal node $t \in \mathcal{L}_{|s|+1}$ $(z_{|s|+1} = 0)$, which are used to initialize and terminate the PSA, respectively.

Nodes $u \in \mathcal{L}_i : 1 \le i \le |s|$, represent the possible values that z_i can take at stage i of the DP formulation. For example, for the DP formulation with affine penalty function, we create $|\mathcal{S}_1| = |s'| + 1$ nodes for layer $\mathcal{L}_1$, and $|\mathcal{S}_i| = 2|s'| + 1$ nodes for layers $\mathcal{L}_i : 2 \le i \le |s'|$. A node $u \in \mathcal{L}_i : 0 \le i \le |s| + 1$ thus represents a specific state value in $\mathcal{S}_i$, denote as $state(u)$. Transitions from states in $\mathcal{S}_{i-1}$, i.e., nodes $u \in \mathcal{L}_{i-1}$, to states in $\mathcal{S}_i$, i.e., nodes $v \in \mathcal{L}_i$, are represented by arcs $(u, v) \in A^p$. Arcs $(u, v) \in A^p$ model the assignment of a value in D_i to control variables x_i, and take the system from $state(u)$ to $state(v) = \mathcal{T}_i(state(u), x_i)$. As in the DP model, transitioning to a node $v : state(v) > 0$ at stage i, implies an alignment of c_s^i to character $c_{s'}^{state(v)}$. Transitioning from node u to a node $v : state(v) < 0$ implies an opening gap alignment if $state(u) > 0$, or an extension gap alignment if $state(u) < 0$.

Following the DP model, PSA feasibility is ensured by creating an arc (u, v) only if $state(v) < |state(u)|$ for character alignments, or $state(v) = -|state(u)|$ for gap alignments. By this representation, any path from r to t corresponds to a feasible PSA solution, as the satisfaction of the order and column alignment requirements are guaranteed by the definition of transition functions $state(v) = \mathcal{T}_i(state(u), x_i)$. Moreover, the MDD models all possible feasible PSA solutions, as its arcs model all feasible transitions of the DP model.

Figures 4.a, 4.b show the MDD representation of the example PSA in Figure 1, with the correspondence of layers $\mathcal{L}_i$ to state spaces $\mathcal{S}_i$, arcs (u, v) to transitions $state(v) = \mathcal{T}_i(state(u), x_i)$, and nodes $u \in \mathcal{L}_i$ to state values z_i, for affine and convex penalty functions respectively. The example alignment $x = (G, -, T)$ corresponds to the highlighted path from the root node r to the terminal node t in both MDDs. Note that the structure of any two PSA MDDs $\mathcal{M}^p, \mathcal{M}^{p'}$, respectively modeling the alignment of sequence pairs (s_1, s_2),

(s_3, s_4), is identical if $|s_1| = |s_3|, |s_2| = |s_4|$. In particular, Figure 4 shows the MDD structure for any two sequences (s, s') such that $|s| = 3, |s'| = 3$.

To implement the reward/penalty of alignments we use weighted MDDs, where arcs $(u, v) \in A^p$ are associated to values $w_{uv} = \mathcal{R}_i(state(u), state(v))$. The length of an r-t path is thus the objective value $\sum\limits_{i=1,\ldots,|s|+1} \mathcal{R}_i(z_{i-1}, z_i)$. Consequently, the longest r-t path corresponds to optimal PSA solution $\hat{x}$. Proposition 1 proves that the PSA MDD correctly models the PSA problem.

Proposition 1 *The PSA MDD is a valid representation for the PSA problem.*

Proof The PSA MDD mimics the DP model for the PSA problem by modeling all states by nodes $u \in A^p$, and all feasible transitions by arcs (u, v). The transition reward functions are also captured by weights w_{uv} on arcs $(u, v) \in A^p$. As the DP model is valid for the PSA problem, the PSA MDD is a valid representation of the PSA problem. $\square$

We next prove in Proposition 2 that our PSA MDDs are *reduced*. An MDD is reduced if there are no equivalent nodes in any of its layers $\mathcal{L}_i$ (Wegener 2000). Two nodes u, u' in a layer $\mathcal{L}_i$ are equivalent if there is a one to one equivalence between the set of u-t paths, and the set of u'-t paths. For equivalent nodes u, u', we can delete node u' and all its outgoing arcs $(u', v) : v \in \mathcal{L}_{i+1}$, and redirect its incoming arcs $(v, u') : v \in \mathcal{L}_{i-1}$ to node u, without altering the decision structure of the MDD.

Proposition 2 *The PSA MDD representation above is reduced by construction.*

Proof By construction, $state(u) \neq state(u')$ for any pair of nodes $u, u' \in \mathcal{L}_i, i = 0, \ldots, |s| + 1$. As arcs $(u, v) \in A^p$ are constructed based on $state(u)$, no two single nodes $u, u' \in \mathcal{L}_i$ have a common set of outgoing arcs (i.e., arcs representing the same decision and the same weights w_{uv}). Therefore, no two paths u-t, u'-t are equivalent for any nodes $u', u \in U^p$. $\square$

The current definition of a PSA MDD models all feasible PSA solutions in $O(n^3)$, $O(n^4)$ space complexity for affine and convex penalty functions, respectively. Although the entire PSA MDDs are required in order to model an exact solution approach for MSA (discussed in the next section), the space complexity of the algorithm may be reduced if we are interested in only finding an optimal PSA. To that end, we prove in Lemma 2, that for PSA the only required information at stage i of the DP model is the longest path values leading to nodes $u \in \mathcal{L}_{i-1}$.

Lemma 2 *Let $I^{\downarrow}(u)$ denote the value of a longest r-u path. Values $I^{\downarrow}(u) \in \mathcal{L}_i$ are the only information required to solve the PSA problem.*

Proof Proof is by induction on stages i of the DP model. The statement is trivial for stage $i = 1$. Assume the statement holds for stage $i > 1$. For any node $v \in \mathcal{L}_{i+1}$ we have by definition of the longest path, $I^{\downarrow}(v) = \underset{u:(u,v)\in A^p}{max} \left\{I^{\downarrow}(u) + w_{uv}\right\}$. By the principal of mathematical induction, the statement holds for any stage i. □

The DP model is thus only required to keep in memory $|\mathcal{L}_{i-1}| + |\mathcal{L}_i|$ number of values at each stage i, and update the partial optimal PSA at every step similar to the algorithm of Hirschberg (1975). This gives a space complexity of $O(n), O(n^2)$ for the PSA problem with affine or convex penalty gaps, which is consistent with the best complexity proven for the PSA problem with linear penalty costs in Hirschberg (1975).

3. Solving the MSA problem using synchronized PSA MDDs

3.1. Modeling the MSA problem as an MIP

The MSA problem may be viewed as the simultaneous alignment of all possible PSAs. In this section, we show how to synchronize all PSA MDDs for an exact solution to the MSA problem. We consider $1 \leq s \leq |\mathbb{S}| - 1$ and $s + 1 \leq s' \leq |\mathbb{S}|$, and let set $\mathbb{P}$ be the union set of all pairs $p = (s, s')$. We construct all pairwise MDDs $\mathcal{M}^p : p \in \mathbb{P}$, and define $\mathbb{M} = (U^c, A^c)$ as the union of all pairwise MDDs $\mathcal{M}^p$, where $U^c = \bigcup_{p\in\mathbb{P}} U^p, A^c = \bigcup_{p\in\mathbb{P}} A^p$. Observe that the worst-case space complexity of $\mathbb{M}$ is $O(|\mathbb{S}|^2 n^3), O(|\mathbb{S}|^2 n^4)$ for affine and convex penalty functions, respectively. To simultaneously solve the PSA MDDs, we first formulate the MIP model below, defined on set $\mathbb{M}$:

$$\max \quad \sum_{(u,v)\in A^c} w_{uv} y_{uv} \tag{5}$$

$$\text{s.t.} \quad \sum_{u:(r,u)\in A^p} y_{ru} = 1 \qquad \forall p \in \mathbb{P}, \tag{6}$$

$$\sum_{u:(u,t)\in A^p} y_{ut} = 1 \qquad \forall p \in \mathbb{P}, \tag{7}$$

$$\sum_{u:(u,v)\in A^p} y_{uv} - \sum_{u:(v,u)\in A^p} y_{vu} = 0 \qquad \forall p \in \mathbb{P}, \forall v \in U^p \setminus \{r, t\}, \tag{8}$$

$$y_{uv} \in \{0, 1\} \qquad \forall (u, v) \in A^c. \tag{9}$$

Here, y_{uv} is a binary variable defined for arcs $(u, v) \in A^c$, determining whether arc (u, v) is part of the optimal solution or not. The objective function (5) maximizes the reward of all pairs of character alignments (sum-of-pair score), and constraints (6)-(8) preserve the required flow in the longest path problems defined on $\mathbb{M}$, the union of disjoint directed acyclic graphs $\mathcal{M}^p$.

Solving the above MIP corresponds to finding the shortest paths in all PSA problems independently, and is likely infeasible with respect to the order requirement in MSA. To enforce the order requirement, we add additional continuous *column* variables $\pi_s^i \geq 0$, which denote the column placement of character c_s^i in the output MSA matrix. We enforce constraints (10) and (11), which ensure that any aligned characters $c_s^i, c_{s'}^j$ are placed into the same column of the MSA matrix (i.e., $\pi_s^i = \pi_{s'}^j$). Here, M denotes a big number, defined as $M = \sum\limits_{s \in \mathbb{S}} |s|$ in our computational experiments.

$$\pi_s^i \leq \pi_{s'}^j + M \left(1 - \sum_{\substack{(u,v) \in A^p : u \in \mathcal{L}_i^p, \\ state(v) = j}} y_{uv} \right) \qquad \forall p = (s, s'), \forall (i,j) \in (s, s'), \tag{10}$$

$$\pi_{s'}^j \leq \pi_s^i + M \left(1 - \sum_{\substack{(u,v) \in A^p : u \in \mathcal{L}_i^p, \\ state(v) = j}} y_{uv} \right) \qquad \forall p = (s, s'), \forall (i,j) \in (s, s'). \tag{11}$$

Finally, we add constraints (12), which enforce the order requirement between columns of the MSA matrix.

$$\pi_s^i \leq \pi_s^{i+1} - 1 \qquad \forall k, \forall i = \{1, \ldots, |s| - 1\} \tag{12}$$

This gives a constrained longest path formulation:

$$\mathbf{P:} \quad \{(5)|(6) - (12), \pi \geq 0\},$$

to solve MSA. Note that although variables π_s^i are continuous, in a solution for $\mathbf{P}$ they take integer values due to their integer upper and lower bounds. The worst-case space complexity of model $\mathbf{P}$ is consistent with the size of $\mathbb{M}$, which is lower than the worst-case exponential space requirements of all current exact MSA algorithms for $|\mathbb{S}| > 3$, in particular for linear or affine gap penalty functions. To prove correctness, Theorem 3 proves that constraints (6)-(12) are sufficient to enforce the feasibility requirements of MSA, and Corollary 1 proves that the solution of $\mathbf{P}$ is an exact MSA solution.

Proposition 3 *An MSA matrix satisfies the MSA feasibility requirements, if and only if constraints (6)-(12) are satisfied.*

Proof If the MSA matrix satisfies the MSA feasibility requirement, we can construct solution vector $\bar{y}$ by setting $\bar{\pi}_s^i = j, \forall s \in \mathbb{S}, \forall i \in s$, where character c_s^i is placed in column j of the MSA matrix. Solutions $\bar{y}, \bar{\pi}$ satisfy constraints (6)-(12), as all aligned characters are placed in the same column of the matrix, and the order of columns satisfy constraint (12). For the converse, we can construct a feasible MSA matrix using the ordering given by solution $\bar{\pi}$. All aligned characters have equal $\bar{\pi}$ and thus fall into the same column, and all columns satisfy the order requirement enforced by constraint (12). □

Corollary 1 *The solution of* ***P*** *is an exact MSA solution.*

Proof By Theorem 3, **P** optimizes over the set of all feasible MSA solutions. Thus, the optimal solution to **P** is an optimal MSA solution. □

Despite its polynomial model size, solving **P** to optimality is only practically feasible for small-sized instances when using a generic MIP solver. To increase the solution efficiency, we next introduce filtering algorithms to prune and reduce the size of $\mathbb{M}$ by removing arcs $(u,v) \in A^c$ (consequently variables y_{uv}) which cannot be part of an optimal MSA solution.

3.2. MDD filtering procedures for the MSA problem

Bound-based filtering is an effective method in reducing the size of large-scale problems (Detienne et al. 2016, 2015). For decision diagrams, filtering refers to the process of identifying and deleting arcs (u,v) which do not contribute to a feasible or optimal solution (Hoda et al. 2010, Andersen et al. 2007, Cayrol et al. 1998). Any r-t path using such arcs is either infeasible or not optimal. Therefore, we may delete arc (u,v) without removing any feasible or optimal solution. By construction, all arcs $(u,v) \in A^p$ may be part of a feasible MSA solution. Therefore, the the PSA MDDs are already filtered in that sense. However, not all arcs can participate in an optimal MSA solution. We use two bounds to filter $\mathbb{M}$ based on optimality, namely the Carrillo and Lipman (1988) lower bound, and the additive bounding Fischetti and Toth (1989) upper bound.

3.2.1. Carrillo-Lipman lower bound As the first method, we use the well-known Carrillo and Lipman (1988) lower bound to reduce the size of each pairwise MDD $\mathcal{M}^p$. The procedure generates a lower bound on the objective value of any $\mathcal{M}^p$, and removes any solution that violates that lower bound.

Let $\bar{y}$ be a feasible MSA solution, which may be obtained using any fast heuristic method, e.g., MUSCLE (Edgar 2004). Further, let $\hat{y}^p$ be the optimal PSA solution of any $\mathcal{M}^p$, and let $f(\bar{y})$ denote the objective value of solution $\bar{y}$. Carrillo and Lipman (1988) prove that $f(\bar{y}) - \sum_{p':p'\neq p} f(\bar{y}^{p'})$ is a valid lower bound for the PSA of pair p, and any PSA solution that violates this bound cannot be part of an optimal MSA solution. Formally,

$$f(\bar{y}^p) \geq f(\bar{y}) - \sum_{p':p'\neq p} f(\hat{y}^{p'}), \tag{13}$$

holds for all $p \in \mathbb{P}$. Now, let $I^{\downarrow}(u)$ denote the value of a longest path from root r to node $u \in U^p$. Similarly, let $I^{\uparrow}(u)$ denote the value of a longest path from a node $u \in U^p$ to terminal t. Using lower bound (13), we filter PSA MDDs $\mathcal{M}^p$ by deleting arcs (u,v) such that:

$$I^{\downarrow}(u) + w_{uv} + I^{\uparrow}(v) < f(\bar{y}) - \sum_{p':p'\neq p} f(\hat{y}^{p'}).$$

Lastly, we delete any node $u \in U^c$, which does not have any incoming arc $(v,u) \in \mathbb{M}$ or any outgoing arc $(u,v) \in \mathbb{M}$.

3.2.2. Additive bounding upper bound Although solving **P** is time-consuming, solving its Linear Programming (LP) relaxation is efficient. The LP relaxation of **P** is obtained by replacing the binary constraints (9), with $0 \leq y_{uv} \leq 1$. The solution of this LP relaxation may be used in filtering procedures for decision diagrams, using the additive bounding procedure (Fischetti and Toth 1989, Kinable et al. 2017).

Given an optimization problem **P**, additive bounding solves a series of relaxations $\tilde{P}_1, \dots \tilde{P}_k$ of P, such that $\tilde{P}_1$ is defined with respect to the original objective coefficients and $\tilde{P}_{k'}$ receives as objective coefficients the reduced costs obtained from $\tilde{P}_{k'-1}$, for $k' = 2, \dots, k$. Fischetti and Toth (1989) show that the sum of the bounds from $\tilde{P}_1$ to $\tilde{P}_k$ is a valid bound for **P**.

In the context of our problem, we combine the LP relaxation of **P** with the discrete relaxation defined by (5)-(9), i.e., the longest path formulation on the pairwise PSA MDDs. First, we solve the LP relaxation of **P**, and denote the resulting upper bound by $\tilde{f}(\mathbf{P})$, its optimal solution by $\tilde{y}$, and the reduced cost of variable y_{uv} by μ_{uv}. We then associate with each arc $(u,v) \in A$ a weight $\tilde{w}_{uv}$ defined as

$$\tilde{w}_{uv} = \begin{cases} \mu_{uv} & \text{if } \tilde{y}_{uv} = 0, \text{ and} \\ 0 & \text{otherwise.} \end{cases}$$

Observe that $\tilde{w}_{uv} \leq 0$ for all $(u,v) \in A$ since **P** is a maximization problem. Given a feasible solution $\bar{y}$ for **P** with objective value $f(\bar{y})$, the additive bound

$$\tilde{f}(P) + \sum_{(u,v)\in A} \tilde{w}_{uv} y_{uv}, \tag{14}$$

is a valid upper bound for **P**, i.e. $f(\bar{y}) \leq \tilde{f}(P) + \sum_{(u,v)\in A} \tilde{w}_{uv} y_{uv}$.

We can now define an MDD arc filtering rule as follows. Let $\tilde{I}^{\downarrow}(u)$ denote the value of a of a longest path, with respect to weights $\tilde{w}_{uv}$, from root node $r \in U^P$ to a node $u \in U^p$. Similarly, let $\tilde{I}^{\uparrow}(v)$ denote the value of a longest path, with respect to weights $\tilde{w}_{uv}$, from a node $v \in U^p$ to terminal node $t \in U^p$. Using the upper bound (14), we filter the MDD M^p by deleting any arcs (u,v) such that:

$$\tilde{I}^{\downarrow}(u) + \tilde{w}_{uv} + \tilde{I}^{\uparrow}(v) < f(\bar{y}) - \tilde{f}(P).$$

As before, we delete any node $u \in U^c$ that does not have any incoming arc $(v,u) \in \mathbb{M}$ or any outgoing arc $(u,v) \in \mathbb{M}$.

3.2.3. Filtering the MDD during construction It is more efficient (in both time and memory) to filter $\mathbb{M}$ during its construction, rather than build $\mathbb{M}$ and filter it thereafter. To that end, we can use the Carrillo-Lipman filtering procedure to filter the PSA MDDs during their construction, as its bound can be calculated before construction of the PSA MDDs. In particular, we can calculate the optimal PSA of all pairs $p = (s, s')$, and generate a heuristic solution $\bar{y}$ (e.g. using MUSCLE) to obtain the Carrilo-Lipman bound (13), without constructing the PSA MDDs. The only MDD-dependant step is calculating values $I^{\downarrow}(u), I^{\uparrow}(v)$ that are used to determine whether an arc (u,v) is constructed or not (filtered).

Values $I^{\downarrow}(u)$ can be calculated during construction, as the PSA MDDs are built incrementally from the root node r to the terminal node t. However, values $I^{\uparrow}(v)$ cannot be calculated from the MDDs before they are fully constructed. Fortunately, it is possible to calculate these values using a matrix-format PSA DP method, similar to that of Mott

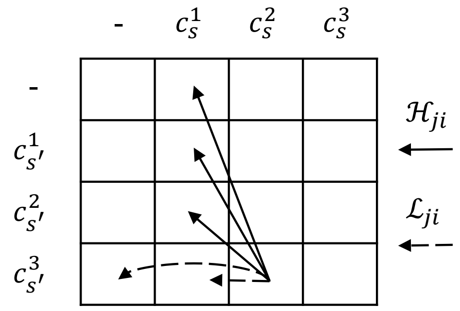

Figure 5 **Example matrix format for PSA, and update procedure for for cell $\mathcal{C}_{4,3}$.**

(1999) and Needleman and Wunsch (1970). The matrix-format DP method is suitable to calculate values $I^{\uparrow}(v)$ due to its forward induction on aligning characters $c_s^1, c_s^2, \ldots, c_s^{|s|}$ (recall that the PSA MDD follows a backwards induction). In the interest of space, we refer the interested reader to Mott (1999) and Needleman and Wunsch (1970) for thorough explanations of matrix-format PSA procedure, and briefly discuss the modified algorithm to calculate values $I^{\uparrow}(v)$ in our paper.

We create a $(|s|+1) \times (|s'|+1)$ alignment matrix, such as the example given in Figure 5 for $|s| = |s'| = 3$. In the matrix-format PSA, the value of cell $\mathcal{C}_{ji}$ at row j and column i of the matrix gives the objective of the best PSA alignment of the first $i-1$ characters $c_s^i \in s$ to the first $j-1$ characters $c_{s'}^j \in s'$. To accommodate affine and convex penalty functions, we additionally define g_{ji} as the number of consecutive gap alignments contributing to the value of cell $\mathcal{C}_{ji}$. Note that g_{ji} may be reduced to a binary value in affine penalty functions, as the penalty of extending a gap does not depend on the length of that gap. Lastly, we define function $\mathcal{G}(g, x)$ which gives a g length gap penalty cost plus an alignment reward based on the decision x.

The value of $\mathcal{C}_{00}$ is initialized to zero, with $g_{00} = 0$, and the values of the first row and column cells are calculated as consecutive gap alignments: $\mathcal{C}_{0i} = \mathcal{G}(i, -), g_{0i} = i$, $\mathcal{C}_{j0} = \mathcal{G}(j, -), g_{j0} = j$. The rest of the cells are calculated incrementally by a DP procedure as $\mathcal{C}_{ji} = \max\{F_{ji}, H_{ji}\}$, where $H_{ji} = \max\limits_{j'<j} \left\{\mathcal{C}_{j',i-1} + \mathcal{G}(j - j' - 1, c_{s'}^{j'})\right\}, g_{ji} = g_{j',i-1} + j - j' - 1$, and $F_{ji} = \max\limits_{i'<i} \{\mathcal{C}_{j,i'} + \mathcal{G}(i - i' - 1, -)\}, g_{ji} = g_{ji'} + i - i' - 1$. Values g_{ji} are updated according to whichever value F_{ji} or H_{ji} gives $\mathcal{C}_{ji}$. Figure 5 shows an example schema of determining these values for cell $\mathcal{C}_{43}$. Finally, cell $\mathcal{C}_{|s'|+1,|s|+1}$ gives the optimal PSA objective.

The calculated values of cells $\mathcal{C}_{ji}$ can be used to exactly determine values $I^{\uparrow}(v)$. By definition, $I^{\uparrow}(v), v \in \mathcal{L}_i, state(v) = j > 0$ is the value of the longest path from node v to terminal t. This path corresponds to the best PSA alignment of the first $i-1$ characters

$c_s^i \in s$ to the first $j-1$ characters $c_{s'}^j \in s'$, i.e. $\mathcal{C}_{ji}$. However, observe that the above procedure assumes no prior gaps leading to a cell $\mathcal{C}_{ji}$ when calculating its value. For example, when calculating C_{01}, the gap penalty is calculated according to a gap of length 1 in $\mathcal{G}(1,-)$, which gives an opening gap penalty. Although this is correct for $I^{\uparrow}(v), v \in \mathcal{L}_i, state(v) = j > 0$, it is an incorrect value for $I^{\uparrow}(v), v \in \mathcal{L}_i, state(v) = j < 0$. This is because extending a gap leading to node v by g additional gaps is not equal to opening a gap of length g in affine or convex penalty functions. In order to calculate values $I^{\uparrow}(v), v \in \mathcal{L}_i, state(v) = j < 0$, we need to build other matrices.

For affine penalty functions, it is sufficient to build an extra PSA matrix with the assumption that all cells C_{ji} proceed a gap of length one. The function $\mathcal{G}(g,x)$ thus calculates an extension gap penalty when calculating the value of cells $\mathcal{C}_{ji}$. The values calculated in this new matrix are used to determine values $I^{\uparrow}(v), v \in \mathcal{L}_i, state(v) = j < 0$.

For PSA with a convex penalty function, the extension gap penalty additionally depends on the number of consecutive gaps leading to nodes $v : state(v) < 0$. To exactly calculate values $I^{\downarrow}(u), u \in \mathcal{L}_i, state(u) = (j,g), j < 0$ we must solve $|s|$ number of PSA matrices, one per possible value of gap lengths g, which takes a relatively long time. A compromise between time and quality of our estimate for values $I^{\downarrow}(u)$ is to solve 2 PSA matrices. The original matrix assumes that $i > 0, g = 0$ to exactly calculate $I^{\downarrow}(u), u \in \mathcal{L}_i, state(u) = (j,0), j > 0$, and for the other matrix we assume $i < 0, g = |s| - i - 1$ to give an upper bound for the values of $I^{\downarrow}(u), u \in \mathcal{L}_i, state(u) = (j,g), j < 0$. That is, as $|s| - i - 1$ is the maximum possible value of g at stage i, all values of the second matrix are an upper bound to the exact values of $I^{\downarrow}(u), u \in \mathcal{L}_i, state(u) = (j,g'), j < 0, g' < g$ due to the convex property of the penalty function. Using the values of this second matrix, we estimate all values $I^{\uparrow}(v), v \in \mathcal{L}_i, state(v) = j < 0$ and filter the PSA MDD during construction. After a PSA MDD is constructed using the estimated values $I^{\uparrow}(v)$, we exactly calculate all values $I^{\downarrow}(u)$, and re-filter the MDD.

This concludes the procedure to filter the PSA MDDs during construction. The next section discussed how to use the filtering procedures for a more effective two phase solution for MSA.

3.2.4. Two phase solution algorithm for MSA based on optimistic filtering Initial numerical results showed that the filtering algorithms do not prune a significant number of arcs $(u,v) \in \mathbb{M}$, when using (commercial) heuristic solvers to provide the bound $f(\bar{y})$.

Unfortunately, obtaining solutions with higher quality quickly is not possible for large-size instances using any other method in the literature.

On the bright side, the discussed filtering procedures do not require the explicit alignments of an MSA solution $\bar{y}$, and operate only using its objective value $f(\bar{y})$. Therefore, it is not required to explicitly generate a solution to filter $\mathbb{M}$, and it is possible to use a guess for the value $f(\bar{y})$. A guess f^g is valid if $f^g \leq f(\hat{y})$, where $\hat{y}$ is an optimal MSA solution. A valid guess close to the optimal objective value $f(\hat{y})$ leads to considerable pruning of $\mathbb{M}$, and consequently a smaller problem to solve. On the other hand, if the guess is invalid, i.e., $f^g > f(\hat{y})$, the filtering procedures may delete the optimal solution $\hat{y}$ from $\mathbb{M}$. In case of an invalid guess, the solution of **P** may turn out to be a local optimal solution, or in the worst-case **P** becomes infeasible and does not contain any feasible MSA solutions.

The validity of f^g cannot be determined unless the optimal value $f(\hat{y})$ is known. However, it is possible to ensure that **P** is feasible after optimistic filtering, by ensuring that it contains at least one feasible MSA solution. A straightforward, yet time-consuming, approach to determine the feasibility of **P** after optimistic filtering is to directly solve **P** using an MIP solver, and observe the result. A much faster approach is to check whether the filtered $\mathbb{M}$ contains some feasible solution $\bar{y}$, which guarantees that **P** is feasible. This is done by ensuring all arcs (u, v) corresponding to variables $y_{uv} \in \bar{y} : \bar{y}_{uv} = 1$ remain in the filtered $\mathbb{M}$.

We consider checking whether $\mathbb{M}$ contains the best found feasible solution $\bar{y}$ (e.g., the solution generated by a heuristic solver). If $\mathbb{M}$ does not contain $\bar{y}$, we reduce the optimistic guess f^g by a step size *stp* (e.g., one which is chosen heuristically), and re-run the filtering procedures using the updated guess $f^g - stp$. This procedure is repeated until $\mathbb{M}$ contains solution $\bar{y}$, in which case the feasibility of **P** is guaranteed.

The optimal solution $\hat{y}^o$ of **P** on the resulting optimistically filtered $\mathbb{M}$, may be a local optimal if $f^g > f(\hat{y})$. To ensure global optimality, we solve the MSA problem in two phases. In the first phase, we perform optimistic filtering on $\mathbb{M}$ and solve **P** to generate a possibly local optimal solution $\hat{y}^o$. In the second phase, we check global optimality by using the first phase solution bound $f(\hat{y}^o)$ to filter $\mathbb{M}$. As $\hat{y}^o$ is a feasible MSA solution, the resulting filtered $\mathbb{M}$ is guaranteed to contain an optimal MSA solution $\hat{y}$. The solution of **P** in the second phase either proves optimality of $\hat{y}^o$, or generates an optimal solution $\hat{y}$ for the MSA problem.

The first phase optimal solution $\hat{y}^o$ is guaranteed to have a higher objective value $f(\hat{y}^o) \geq f(\bar{y})$, where $\bar{y}$ is generated by any heuristic MSA solver, proven in Lemma 3. In fact, it is noteworthy to say that our numerical results show that either $\hat{y} = \hat{y}^o$ or $f(\hat{y}^o)$ is in worst-case within 97% of the global optimal objective $f(\hat{y})$.

Lemma 3 *The solution $\hat{y}^o$ generated by solving $\boldsymbol{P}$ on an optimistically filtered $\mathbb{M}$ satisfies $f(\hat{y}^o) \geq f(\bar{y})$, where $\bar{y}$ is any feasible MSA solution.*

Proof In the first phase $\mathbb{M}$ is guaranteed to contain the best found feasible solution, e.g., $\bar{y}$. Therefore, solving $\mathbf{P}$ on $\mathbb{M}$ generates a solution $\hat{y}^o$ which is either $\hat{y}^o = \bar{y}$, or satisfies $f(\hat{y}^o) > f(\bar{y})$ due to the maximization objective (5). $\square$

This concludes the filtering procedures for $\mathbb{M}$. The filtered $\mathbb{M}$ is expected to be of smaller size and consequently leads to an easier problem to solve. However, solving $\mathbf{P}$ on the filtered $\mathbb{M}$ is still hard and time consuming using a commercial MIP solver. Our next step to reduce solution difficulty is to use Benders decomposition.

4. Logic-based Benders Decomposition for solving synchronized PSA MDDs

4.1. Logic-based Benders decomposition

In this section, we design a decomposition algorithm to solve model $\mathbf{P}$ more effectively than a generic MIP solver. Based on the structure of model $\mathbf{P}$, we propose using logic-based Benders decomposition (Hooker 2002). We refer the interested reader to Hooker and Ottosson (2003) for a thorough explanation of logic-based Benders decomposition.

As the first step, $\mathbf{P}$ is reformulated such that it only contains binary variables y_{uv}. This is done by relaxing constraints (10)-(12), and moving them to subproblem:

$$\textbf{SP:} \quad \max\left\{0^T\pi \mid (10)-(12), \pi \geq 0\right\}.$$

This gives the relaxed master problem:

$$\textbf{RMP:} \quad \max\{(5) \mid (6)-(9)\}.$$

The solution $\bar{y}$ of **RMP** may or may not satisfy the MSA order requirement, i.e., constraints (10)-(12). To check this, **DSP** the dual of **SP**, is solved at $\bar{y}$:

$$\min \quad M\left(1 - \sum_{\substack{(u,v)\in A^p: u\in\mathcal{L}_i^p,\\ state(v)=j}} \bar{y}_{uv}\right)\left(\sum_{(s,s')}\sum_{(i,j)\in(s,s')} \alpha_{(s,s')}^{ij} - \gamma_{(s,s')}^{ij}\right) + \sum_{s\in\mathbb{S}, i\in s} \beta_s^i \qquad (15)$$

$$\text{s.t.} \quad \sum_{s'\in\mathbb{S}:s'>s}\sum_{j\in s'}\alpha^{1j}_{(s,s')} - \sum_{s'\in\mathbb{S}:s'>s}\sum_{j\in s'}\gamma^{1j}_{(s,s')} + \beta^1_s \geq 0 \qquad \forall s\in\mathbb{S}, \tag{16}$$

$$\sum_{s'\in\mathbb{S}:s'>s}\sum_{j\in s'}\alpha^{ij}_{(s,s')} - \sum_{s'\in\mathbb{S}:s'>s}\sum_{j\in s'}\gamma^{ij}_{(s,s')} - \beta^i_s + \beta^{i+1}_s \geq 0$$

$$\forall s\in\mathbb{S}, \forall i\in s: 2\leq i\leq |s|-1, \tag{17}$$

$$\sum_{s'\in\mathbb{S}:s'>s}\sum_{j\in s'}\alpha^{|s|j}_{(s,s')} - \sum_{s'\in\mathbb{S}:s'>s}\sum_{j\in s'}\gamma^{|s|j}_{(s,s')} - \beta^{|s|}_s \geq 0 \qquad \forall s\in\mathbb{S}, \tag{18}$$

$$\alpha^{ij}_{(s,s')}, \gamma^{ij}_{(s,s')} \geq 0 \qquad \forall (s,s'), \forall (i,j)\in(s,s'), \tag{19}$$

$$\beta^i_s \geq 0 \qquad \forall s, \forall i\in s. \tag{20}$$

where variables $\alpha^{ij}_{(s,s')}, \gamma^{ij}_{(s,s')}, \beta^i_s$ correspond to the dual variables associated with constraints (10), (11), (12), respectively. If **DSP** is feasible, then $\bar{y}$ is a feasible solution for **P**. Otherwise, if **DSP** is unbounded, a Benders feasibility cut (21) is generated and added to **RMP** to remove the infeasible solution.

$$M\left(\sum_{(s,s')}\sum_{(i,j)\in(s,s')}\bar{\alpha}^{ij}_{(s,s')} - \bar{\gamma}^{ij}_{(s,s')}\right)\left(1 - \sum_{\substack{(u,v)\in A^p:u\in\mathcal{L}^p_i,\\ state(v)=j}} y_{uv}\right) \geq -\sum_{s\in\mathbb{S},i\in s}\bar{\beta}^i_s \tag{21}$$

We consider any feasible solution generated by the MIP solver when solving **RMP**, and add a corresponding cut (21) to **RMP** if that solution is an infeasible MSA solution. The Benders algorithm iterates until the solution of **RMP** leads to a feasible (not unbounded) solution for **DSP**, which indicates an optimal solution for **P**.

Solving **RMP** is significantly easier than solving **P**. However, the rate of convergence to the optimal solution is slow, and it takes many feasibility cuts (21) to reach the optimal solution. Cuts (21) tend to remove one infeasible solution per iteration of the algorithm, and given the difficulty of solving **RMP** at each iteration, this approach is not effective. To generate stronger cuts, we develop logic-based *column alignment* cuts, which are added to **RMP** together with constraints (21). Our column-alignment cuts are in fact a projection of the“generalized transitivity inequalities” used in Althaus et al. (2006) onto the MDD variable space.

Column alignment cuts are designed to enforce the alignment of a pair of characters $c^i_s, c^j_{s'}$, if those characters are both aligned with a third character $c^k_{s''}$. Although this alignment is guaranteed for **P** due to constraints (10), (11), it is not enforced when constraints

(10), (11) are relaxed, i.e., in **RMP**. Cuts enforcing this alignment can thus be used to increase the solution quality of **RMP** and aid in improving convergence. We model such cuts by considering a triple of characters $\left(c_s^i, c_{s'}^j, c_{s''}^k\right)$ of sequence pairs $p = (s, s'), p' = (s, s''), p'' = (s', s'')$, and imposing the following constraints:

$$\sum_{u:u\in\mathcal{L}_i^{p'}} \sum_{\substack{v:(u,v)\in A^{p'},\\ state(v)=k}} y_{uv} + \sum_{u:u\in\mathcal{L}_j^{p''}} \sum_{\substack{v:(u,v)\in A^{p''},\\ state(v)=k}} y_{uv} \leq 1 + \sum_{u:u\in\mathcal{L}_i^{p}} \sum_{\substack{v:(u,v)\in A^{p},\\ state(v)=j}} y_{uv} \tag{22}$$

Proposition 1 proves that the triple consideration of sequences (s, s', s'') is sufficient to ensure correct column alignment for any number of sequences above three.

Theorem 1 *It suffices to consider all $\binom{|\mathbb{S}|}{3}$ triples of sequences, to ensure correct column alignment in MSA, for any number of sequences.*

Proof Consider a graph with nodes denoting characters c and arcs denoting alignment of characters. A column in the MSA matrix corresponds to a set of connected nodes in the graph. Observe that aligned characters respect the column alignment requirement if the induced subgraph of their nodes forms a clique. If constraints (22) are satisfied, then for any connected component, all induced sub-components of size three are cliques. This implies that the component is itself a clique. □

It is possible to generate all possible cuts (22) a priori and add them to **RMP**; however, doing so makes **RMP** much harder to solve. In fact, adding all such cuts makes the LP relaxation of **RMP** unsolvable within 10 hours of computation, even for small instances. On the other hand, not all cuts (22) contribute to the solution of **RMP**, and most are non-binding for its optimal solution. We can thus only add a subset of these constraints to **RMP**, and achieve the same results in much lower computation time. The problem is that we do not know beforehand which cuts (22) are binding, and this is only observed given a solution to **RMP**.

We thus add cuts (22) to **RMP** as logic-based cuts within the Benders decomposition algorithm. At each iteration, in addition to Benders feasibility cuts (21), we generate violated cuts (22) and add them to **RMP**. The separation procedure to generate cuts (22) involves modeling the alignment determined by the solution of **RMP** as a graph (following the same design as in the proof of Theorem 1), and finding any triple of nodes which do not form a clique for any connected component. As proven in Althaus et al. (2006), the time

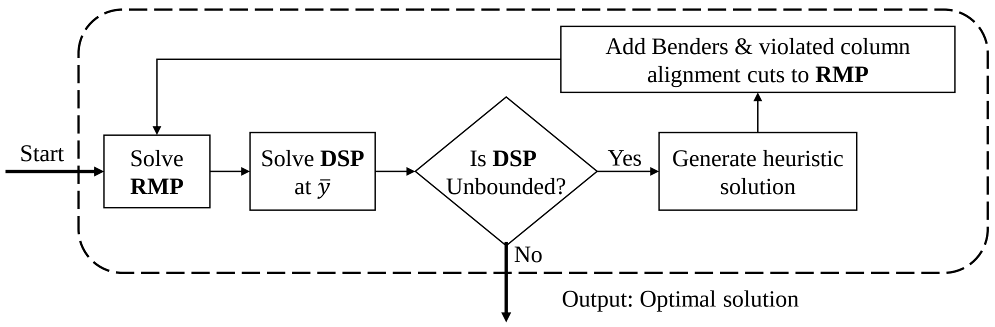
Add Benders & violated column alignment cuts to **RMP**
Start
Solve **RMP**
Solve **DSP** at $\bar{y}$
Is **DSP** Unbounded?
Yes
Generate heuristic solution
No
Output: Optimal solution

Figure 6 **Logic-based Benders decomposition algorithm.**

complexity of finding such cuts is bounded by $O(n^4)$. Figure 6 shows the overall Benders decomposition algorithm. Note that the heuristic procedure shown in Figure 6 is discussed in Section 4.3.

4.2. Warm-starting the logic-based Benders decomposition algorithm

The Benders decomposition algorithm solves **RMP** once per iteration. Repeatedly solving **RMP** is time consuming, and lowers the effectiveness of the algorithm. Moreover, cuts generated in early iterations of Benders decomposition tend to be of low quality, and not worth the computational effort of solving **RMP**. One approach to increase the effectiveness of Benders decomposition is to warm-start the relaxed master problem (McDaniel and Devine 1977). The warm-start algorithm is an iterative process that consists of solving the LP relaxation of **RMP** at each iteration, and adding to it violated column alignments cuts (22), and Benders feasibility cuts (21). The warm-start algorithm iterates until there is no improvement in the LP bound $\tilde{f}(\mathbf{P})$. Warm-starting enables fast generation of valid cuts, without the need to solve an MIP problem at each iteration. The resulting master problem has a tighter linear relaxation, and is used to start the Benders decomposition.

Warm-starting has additional benefits for our problem, as it provides opportunities to further reduce the size of $\mathbb{M}$ using the additive bounding filtering procedure. At each iteration of the warm-start algorithm, the LP relaxation of **RMP** gives an updated objective bound $\tilde{f}(\mathbf{P})$ and reduced costs μ_{uv}. These solutions can be used in the additive bounding filtering procedure to further filter $\mathbb{M}$. At each iteration, **RMP** is redefined to optimize over the newly filtered $\mathbb{M}$, which is smaller in size and easier to solve. Figure 7 gives the overall warm-start algorithm.

4.3. Primal heuristic procedure to generate feasible MSA solutions

As the last component of our solution framework, we develop a heuristic procedure that modifies an infeasible MSA solution $\tilde{y}$ derived from **RMP**, to generate a feasible MSA

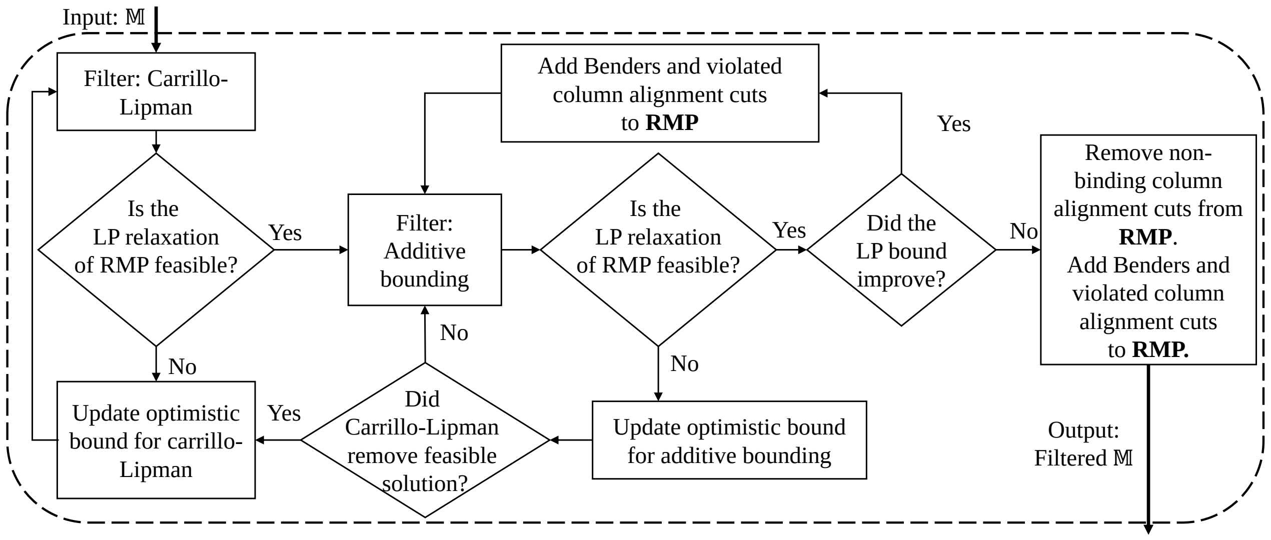
Input: 𝕄
Filter: Carrillo-Lipman
Is the LP relaxation of RMP feasible?
Yes
No
Update optimistic bound for carrillo-Lipman
Yes
Did Carrillo-Lipman remove feasible solution?
No
Filter: Additive bounding
Add Benders and violated column alignment cuts to **RMP**
Is the LP relaxation of RMP feasible?
Yes
No
Update optimistic bound for additive bounding
Did the LP bound improve?
Yes
No
Remove non-binding column alignment cuts from **RMP**. Add Benders and violated column alignment cuts to **RMP.**
Output: Filtered 𝕄

Figure 7 **Warm-start algorithm.**

solution $\bar{y}$. The heuristic is executed within the Benders decomposition, and used to possibly generate solutions $\bar{y}$ with higher quality than those generated by heuristic MSA solvers. Such solutions $\bar{y}$ are used to update the incumbent, and provide better bounds $f(\bar{y})$ for the MDD filtering algorithms.

The heuristic algorithm procedure consists of two steps. In the first step, the algorithm removes a set of alignments from the infeasible solution $\tilde{y}$, such that the resulting solution does not violate MSA requirements. To determine which set of alignments should be removed, we sort all made alignments $\tilde{y}_{uv} = 1$ in ascending order of their objective coefficient w_{uv}. We then remove alignments in the sorted order until the resulting solution does not violate MSA requirements.

In the second step, we use the feasible solution generated in the first step, and add to it any possible feasible alignments that improve the overall objective. This is done by feeding the solution to a heuristic solver, e.g MUSCLE (Edgar 2004), to refine the alignment.

The heuristic procedure is executed for every feasible solution generated by the MIP solver when solving **RMP**, in each iteration of the Benders algorithm. At any point, if the heuristic solution improves the bound $f(\bar{x})$ (or f^g in optimistic filtering) used in the filtering procedures, we re-filter 𝕄 using the new bound.

This concludes all procedures for the complete MSA solution algorithm, summarized in Figure 8. The second phase to check global optimality is executed after the first phase converges to an optimal solution. The second phase is identical to the first phase algorithm,

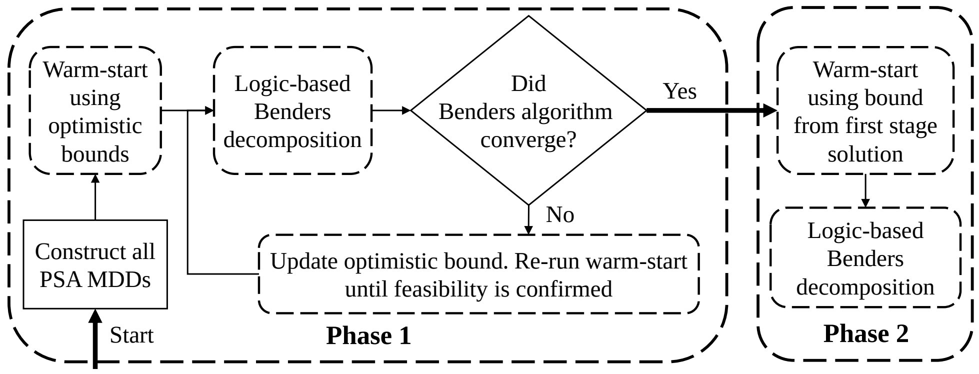
Warm-start using optimistic bounds
Logic-based Benders decomposition
Did Benders algorithm converge?
Yes
No
Construct all PSA MDDs
Update optimistic bound. Re-run warm-start until feasibility is confirmed
Start
Phase 1
Warm-start using bound from first stage solution
Logic-based Benders decomposition
Phase 2

Figure 8 **Overall algorithm for solving MSA by synchronized PSA MDDs.**

with the exception of using the generated solution of the first phase for exact filtering, instead of the guessed bounds used to optimistically filter the PSA MDDs.

5. Numerical Experiments

In this section, we describe two sets of numerical experiments on our MDD solution approach for MSA. In our first set of experiments, we consider an affine penalty function and analyze the performance of solving instances from BaliBASE benchmark version 4 (BaliBASE4 2017). We evaluate the effects of the filtering algorithms in reducing the size of $\mathbb{M}^c$, and compare the quality of our solutions to a state-of-the-art heuristic MSA solver. In our second set of numerical tests, we consider both convex and affine penalty functions and compare our algorithm to the best exact approach in the literature on instances from BaliBASE benchmark version 1.

5.1. Experimental setup

For alignment rewards we use the BLOSUM62 substitution matrix, and for our convex penalty function we use $8+2g+2\sqrt{g}$, where g is the length of a gap in the alignment. This function is adopted from Althaus et al. (2006), and shown to provide accurate alignments. For our tests using an affine penalty function, we use $12+2.24g$, where the opening and extension penalties are obtained from a least squares regression approximation of function $8+2g+2\sqrt{g}$, for gap lengths $g=1,...,50$.

For optimistic filtering, we heuristically choose a step size of $stp = \frac{\sum_p f(\hat{y}^p) - f(\bar{y})}{(|S|*75/4-50)}$, where $\bar{y}$ is the solution obtained from a heuristic MSA solver. We use two independent optimistic guesses for the Carrillo-Lipman (f^g_{CL}) and additive bounding (f^g_{AB}) filtering procedures. An initial value of $\sum_p f(\hat{y}^p) - stp$ is used for both guesses. At any time during the warm-start algorithm, if f^g_{AB} turns out to be greater than the value of the LP objective $\tilde{f}(\mathbf{P})$, we set $f^g_{AB} = \tilde{f}(\mathbf{P}) - stp$.

All algorithms are coded in C++, CPLEX 12.7.1 is used as the commercial MIP solver, and MUSCLE 3.8.31 (Edgar 2004) is used as the commercial MSA heuristic solver. MUSCLE was chosen as it consistently provided the best alignment for our objective setting compared to other heuristic solvers. All tests are executed on a PC with Intel Xeon E5345 2.33 GHz processor, with 24GB of memory on an Ubuntu 14.04.6 platform. Tests are limited to one thread of the CPU, and a 10 hour time limit is chosen for the overall computation time for each instance. Our algorithm is open-source and available to download[1].

5.2. Experiments on Balibase benchmark 4 and comparison to Muscle

For our first set of experiments, we use the latest BaliBASE benchmark, i.e. version 4 (BaliBASE4 2017), and initially limit our tests to instances with at most 1000 total number of characters, i.e., $\sum_{s \in S} |s| \leq 1000$. We then attempt to solve number of the largest size instance that can fit in our system memory, to investigate the limits of our algorithm in solving larger sized problems. All test in this section use an affine gap penalty function, as to the best of our knowledge no commercial heuristic solver accommodates convex penalty functions.

5.2.1. Effects of filtering algorithms To evaluate the effects of the filtering algorithm, we report in Figure 9 the relative size of the filtered $\mathbb{M}^c$ compared to its original size, in terms of number of arcs $|A^c|$ and nodes $|N^c|$. Results are reported both for optimistic filtering in the first phase, and exact filtering in the second phase (for instances that did not reach the time limit in the first phase).

Results show that the filtering algorithms filter a considerable number of nodes and arcs in most instances, in particular using the additive bounding filtering procedure. On average, the Carrillo-Lipman filtering procedure filters a considerable 94.5% of arcs, and 74.6% of nodes in $\mathbb{M}$ for optimistic filtering. The additive bounding filtering procedure is performed after the Carrillo-Lipman filtering, and filters on average 47.3% of the remaining arcs, and 31.1% of the remaining nodes in the filtered $\mathbb{M}$. In exact filtering for the second phase, the Carrillo-Lipman filtering procedure filters 75.4% of arcs, and 52.8% of nodes, with the additive bounding filtering procedure filtering 98.2%, 88.9% of the remaining arcs and

[1] https://github.com/aminhn/MSAMDD

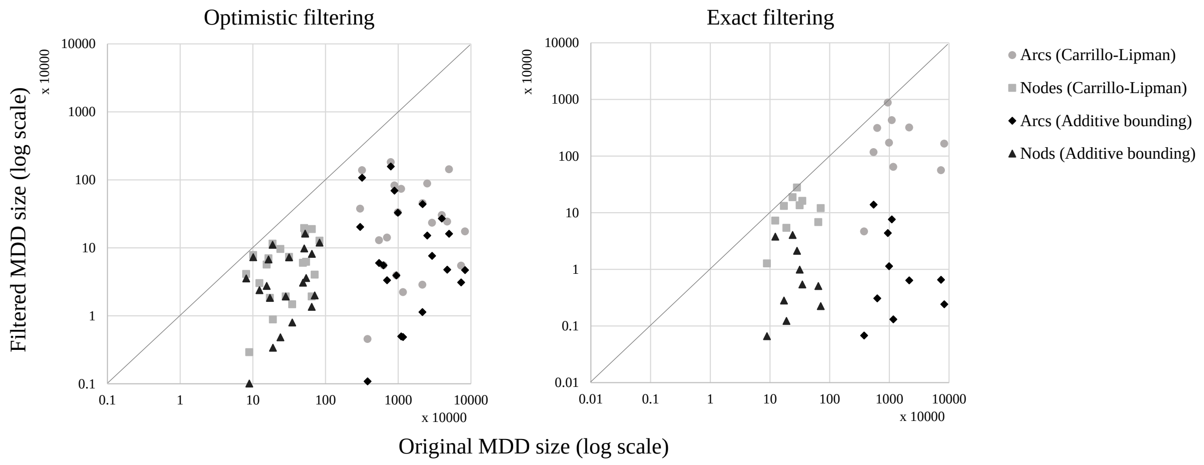

Figure 9 **Effects of filtering algorithms on the size of $\mathbb{M}^c$.**

nodes, respectively. It is noteworthy that in some instances the Carrillo-Lipman filtering procedure does not filter many nodes and arcs, but the additive bounding procedure is still able to considerably filter $\mathbb{M}$.

5.2.2. Comparison to MUSCLE Table 1 reports results on the CPU time, and accuracy of the proposed algorithm compared to the heuristic MSA solver MUSCLE. Instances are divided into two groups, with the first group reporting results for any instance in the benchmark with less than 1000 total characters, and the second group reporting results for a number of larger-sized instances, up to 1500 total number of characters. The "Tot-CPU" column reports the total CPU time (phase one plus phase two). The "Conv-Gap" column reports the Benders convergence gap of the first phase, calculated as $\frac{100(f(\mathbf{RMP})-f(\bar{y}))}{f(\bar{y})}$, where $\bar{y}$ is the best found solution. Note that $f(\mathbf{RMP})$ may not be the global upper bound in phase one due to optimistic filtering. Column "Impr-Mus" reports the percentage of improvement $\frac{100\left(f(\bar{y})-f(\bar{y}^h)\right)}{f(\bar{y}^h)}$ made by our algorithm compared to the initial heuristic solution $\bar{y}^h$ generated by MUSCLE. The final columns report the CPU time, global optimality gap, and improvement of solution $\bar{y}$ in the second phase compared to the optimal solution of the first phase. For the latter, a 0% improvement indicates that the first phase solution was in fact the global optimal MSA solution, and a dash indicates that the second phase was not executed, as the first phase reached the imposed time limit.

The majority of CPU-time was spent in the warm-start algorithm and solving the MIP formulation of **RMP**. On average, 50.5% of the time was consumed in the warm-start algorithm, and 42.8% of the total CPU time was spent solving **RMP**. The primal heuristic

Table 1 Results on Balibase benchmark 4, and comparison to MUSCLE. CPU time is given for the overall algorithm, phase one, and phase two. The convergence gap (Cnv-Gap) in phase one, and improvement of the best found solution compared to MUSCLE (Imp-Mus). The last columns give the CPU time and global optimality gap in phase two, and improvement of the best found solution compared to phase one (Impr-Ph1).

Instance	Tot-CPU	Phase one			Phase two		
		CPU	Cnv-Gap%	Imp-Mus%	CPU	Gap%	Imp-Ph1%
BB11001 (4/345)	7	4	**0**	285.8	3	**0**	0
BB11002 (8/784)	36,993	36,993	24.4	17.5	-	-	-
BB11009 (4/864)	36,553	36,553	13.0	0	-	-	-
BB11013 (5/358)	36,140	36,140	13.2	19	-	-	-
BB11021 (4/469)	36,176	36,176	35.8	0	-	-	-
BB11022 (4/472)	36,234	36,234	11.3	4	-	-	-
BB11025 (4/331)	36,106	36,106	11.0	16.4	-	-	-
BB11029 (4/408)	13,030	7,998	**0**	22.2	5,033	**0**	0
BB11035 (5/487)	36,118	36,118	39.0	0	-	-	-
BB12003 (8/570)	1,474	327	**0**	12.3	1,147	**0**	0
BB12006 (4/928)	490	384	**0**	9.2	106	**0**	0
BB12009 (5/563)	2,302	669	**0**	25.7	1,633	**0**	0.84
BB12014 (9/875)	36,645	36,645	6.7	16.2	-	-	-
BB12020 (4/502)	104	66	**0**	28.6	38	**0**	0
BB12021 (6/454)	378	105	**0**	33	272	**0**	0
BB12024 (4/973)	649	493	**0**	17.7	156	**0**	0
BB12025 (4/808)	36,268	36,268	28.0	0	-	-	-
BB12032 (9/595)	12,419	9,561	**0**	165	2,858	**0**	0
BB12040 (5/661)	261	123	**0**	20.2	138	**0**	0
BB12041 (7/795)	36,114	36,114	684.8	76.9	-	-	-
BB40010 (9/981)	38,258	38,258	**0**	0.9	-	-	-
BB11012 (4/1456)	5,329	4,806	**0**	21.4	523	**0**	0
BB11028 (10/1588)	36,215	36,215	46.0	0	-	-	-
BB12012 (4/1494)	38596	38596	20.1	0	-	-	-
BB12036 (7/1467)	3,873	1,558	**0**	6.8	2,315	**0**	0

procedure generally spends a few minutes, on average 3.9% of total CPU-time, and on average improves the solution generated by MUSCLE by 33.3%. Improvement is as high as 285.8% such as in instance BB11001, which was solved within 7 seconds.

Out of the 24 considered instances, the algorithm solves 12 instances to optimality. This is true for instances with as many as 9 sequences, or 1467 total characters. In 11 of the 12 instances solved, the solution found in phase one proved to be the global optimum, with the remaining instance within 99.16% of global optimality. To the best of our knowledge, our algorithm is the first to solve these Benchmarks to global optimality, for our considered affine objective function. The convergence gap is generally low for instances that did not converge within the 10 hour time limit, with the exception of instances BB12041. Our conjecture is that the upper bound in such instances is relatively tight, and the gap percentages are mainly due to lower quality incumbents solutions.

It is noteworthy that all instances with less than 1000 total characters used at most 4GB of memory, with the larger instances using up to 9GB of memory. Attempting to solve larger instances with 2000 total characters failed due to the algorithms exhausting the 24GB system memory.

5.3. Experiments on Balibase benchmark 1 and comparison to the MWT

For a comparison to the best exact approach in the literature, we compare to the MWT algorithm of Althaus et al. (2006). We do not compare to the DP approach as it does not scale to the size of our considered instances, or the convex optimization approach of Yen et al. (2016) which is limited to aligning sequences of length at most 50. We also do not report results of solving the original model **P** directly using a commercial solver (e.g. CPLEX), or a classical Benders decomposition algorithm without logic-based cuts 21. These algorithms performed poorly, and terminated with considerably high gaps to optimality within the imposed time limit, even smaller instances.

Unfortunately, Althaus et al. (2006) were not able to find and send us the MWT algorithm, and so we base our comparison on their reported results for the BaliBASE benchmark version 1. We remark that CPU time results of Althaus et al. (2006) were reported in 2006, and although their CPU has higher clock speed of 3.06GHz compared to our 2.33GHz (note that we limit our tests to one core of the CPU for fairer comparison), their results are still likely faster when executed on today's PCs. Moreover, for a fairer comparison, we analyzed the speedup from CPLEX 9.0 used in the MWT branch-and-cut algorithm to CPLEX 12.7.1 used in our algorithm.[2] This is measured by the speedup of solving the linear relaxation of our **RMP** for all considered instances. Note that the only relevant speedup involves the solution of linear programs, as "the running time for the branch-and-cut algorithm was dominated by the time required for solving the LPs" (Althaus et al. 2006), and

[2] We thank Ed Klotz for sharing the CPLEX 9.0 library.

Table 2 **Results on Balibase benchmark 1 and comparison to MWT. Results include the reported CPU time, adjusted CPU time due to CPLEX version (CPU-adj) and optimality gap for MWT, CPU time, convergence gap (phase one gap), and global optimality gap (phase two gap) for our MDD approach under affine and convex penalty functions. The last column (Imp-Aff) gives the improvement of the best found solution under the convex penalty function, over the best found solution under the affine penalty function evaluated by the convex function.**

Instance	MWT (convex)			MDD (affine)			MDD (convex)			
	CPU	CPU-adj	Gap%	CPU	Cnv-Gap%	Gap%	CPU	Cnv-Gap%	Gap%	Imp-Aff%
1aho (5/320)	194	[147, 165]	0	17	0	0	44	0	0	0.02
1csp (5/339)	2	[2, 2]	0	14	0	0	28	0	0	0
1dox (4/374)	428	[325, 364]	0	29	0	0	43	0	0	1.35
1fkj (5/517)	315	[239, 268]	0	64	0	0	186	0	0	0
1fmb (4/400)	2	[2, 2]	0	20	0	0	26	0	0	0
1krn (5/390)	17	[13, 14]	0	35	0	0	44	0	0	0
1plc (5/470)	34	[26, 29]	0	49	0	0	91	0	0	0
2fxb (5/287)	4	[3, 3]	0	5	0	0	15	0	0	0
2mhr (5/572)	30	[23, 26]	0	76	0	0	215	0	0	0
9rnt (5/499)	20	[15, 17]	0	40	0	0	78	0	0	0
1aab (4/291)	16	[12, 14]	0	2	0	0	8	0	0	0
1fjlA (6/398)	471	[358, 400]	0	67	0	0	91	0	0	0.32
1hfh (5/606)	1,782	[1,354, 1,515]	0	188	0	0	1,352	0	0	0.16
1hpi (4/293)	469	[356, 399]	0	9	0	0	19	0	0	0.94
1csy (5/510)	564	[429, 479]	0	120	0	0	430	0	0	0
1pfc (5/560)	2,438	[1,853, 2,072]	0	287	0	0	1,106	0	0	0.92
1tgxA (4/239)	128	[97, 109]	0	17	0	0	27	0	0	0.38
1ycc (4/426)	36,124	[27,454, 30,705]	14.1	505	**0**	**0**	1,584	**0**	**0**	0.59
3cyr (4/414)	415	[315, 353]	0	37	0	0	102	0	0	0.83
451c (5/400)	45,186	[34,341, 38,408]	582.7	21,482	**0**	**0**	37,184	88.3	-	-11.02
1aboA (5/297)	36,494	[27,735, 31,020]	38.1	36,002	18.8	-	36,003	39.3	-	-21.01
1idy (5/269)	37,402	[28,426, 31,792]	24.2	22,776	**0**	**0**	36,003	4.1	-	-2.01
1r69 (4/277)	36,375	[27,645, 30,919]	17.2	5,554	**0**	**0**	12,647	**0**	**0**	0
1tvxA (4/242)	37,713	[28,662, 32,056]	60.06	36,005	0	1.9	36,006	12.3	-	-10.06
1ubi (4/327)	37,851	[28,767, 32,173]	37.7	19,541	**0**	**0**	36,001	7.0	-	-7.52
1wit (5/484)	38,264	[29,081, 32,524]	16.2	1,098	**0**	**0**	12,281	**0**	**0**	1.27
2trx (4/362)	36,006	[27,365, 30,605]	50.05	158	**0**	**0**	479	**0**	**0**	0

no MIP is solved in any part of the MWT algorithm to the best of our understanding. Our results showed that the geometric mean of CPU time for CPLEX 12.7.1 is 15.4% to 24.0% faster than CPLEX 9. This range is obtained by considering all instances (15.4%), or only instances which had a solution time above 1 second (24%). We therefore adjust the CPU time of (Althaus et al. 2006) by a reporting a speedup range between 15.4% to 24% for their reported times.

Table 2 reports the comparison of the MDD and MWT approaches. Following the setting of Althaus et al. (2006), instances are divided into three groups, where the first group consists of similar sequences with identity $> 35\%$, group two sequences have $20-40\%$ identity, and group three consists of dissimilar sequences with $< 25\%$ identity. Results are evaluated based on the total CPU time, and gap to optimality (Gap%). We also report the convergence gap (Cnv-Gap) of our MDD approach in the first phase. We perform test under both an affine and convex penalty function. For the remainder of this section, we refer to the MDD algorithm under the convex or affine penalty function as the convex or affine MDD algorithm, respectively. Results of the convex MDD algorithm are used as comparison to the results of the MWT approach, and results of the affine MDD algorithm serve as an approximation to the optimal convex solution. To evaluate the quality of this approximation, the last column of Table 2 reports the improvement of the best found solution using the convex MDD algorithm $\bar{y}_{cnv}$, over the best found solution of the affine MDD algorithm $\bar{y}_{aff}$. This is done by calculating the objective of $\bar{y}_{aff}$ using the convex penalty function, and obtaining the percentage of improvement compared to $\bar{y}_{cnv}$.

Results show that our algorithm is competitive with the MWT approach in aligning the highly similar sequences of the first two group of instances. The MWT approach is effective for such instances, as the root node value of its branch-and-cut tree is near optimal (in some instances optimal), and not many cuts and branches are required to converge to the the optimal solution. For such instances, MUSCLE is also able to produce high quality solutions which are either optimal or near optimal. Using the initial heuristic solutions, The convex MDD approach is able to significantly filter $\mathbb{M}^c$, and also solves all instances within a few minutes.

The MWT approach is not effective for instances with low similarity, and does not solve such instances withing 10 hours of computation. The convex MDD algorithm solves 22 out of the 27 instances to optimality, and reaches relatively low optimality gaps in the remaining instances, except for instances 451c and 1aboA. In general, the MWT approach requires many cuts and branches to reach the optimal solution if its root node objective is not near optimal. For this reason, it may be the case that the results of the MWT algorithm do not improve significantly when solving harder instances, even using today's faster PCs.

The MDD approach performs well under an affine penalty function setting. The affine MDD algorithm is able to solve all but 2 of the instances, and reaches low gaps in the unsolved instances. The solution time is also considerably lower compared to the convex

MDD algorithm, while solution quality remains high. The solutions obtained from the affine MDD algorithm are in most cases within 99% of the optimal convex solution. In fact, for harder instances not solved to optimality by the convex MDD algorithm, the affine MDD algorithm generates better results evaluated under a convex penalty function.

In terms of memory requirements, the affine MDD algorithm uses at most 4GB for any instance solved using the affine penalty function. The memory requirement is much higher for the convex MDD algorithm, with the 451c instance consuming close to 22GB of memory. This shows that although our MDD approach has a worst-case polynomial space complexity, its initial memory consumption is much higher compared to the MWT algorithm.

To summarize the numerical experiments, our MDD approach to MSA gives favorable results compared to the best exact approach in the literature, and closes a number of benchmark instances for the first time under our considered convex and affine penalty functions. For instances not solved to optimality, our algorithm is able to generate solutions with much higher quality compared to a state-of-the-art heuristic MSA solver.

6. Conclusion

This paper developed an exact solution approach for the Multiple Sequence Alignment (MSA) problem. We considered MSA as the simultaneous solution to all Pairwise Sequence Alignments (PSA), which are modeled by dynamic programming and represented using Multi-valued Decision Diagrams (MDD). We used the collection of all PSA MDDs as the underlying graph structure to represent the MSA problem as a Mixed-Integer Program (MIP). The MIP problem synchronizes the solution to all PSA MDDs using side constraints, for the first time, in polynomial space complexity.

We designed two filtering procedures to reduce the size of the feasible set and lead to easier problems to solve. Using the filtering procedures, we increased the effectiveness of our algorithm by developing a two phase solution approach. In the first phase, the PSA MDDs are filtered aggressively using an optimistic guess for the bound used in the filtering procedures, resulting in a heuristic optimization. The advantage of optimistic filtering is that it filters a considerable size of the feasible solution set and enables a relatively fast algorithm to obtain near optimal solutions. On average, 96.3% of arcs, and 81.1% of nodes in our PSA MDDs were pruned using optimistic filtering, while not pruning the optimal solution in almost all solved instances. Global optimality is ensured by re-running the algorithm in a second phase with exact filtering using the bound from the first phase optimal solution.

Lastly, we designed a logic-based Benders decomposition algorithm to solve **P**. We developed a fast primal heuristic and integrated it into the Benders decomposition algorithm. On average, our heuristic solver improved the solution quality of a state-of-the-art MSA heuristic solver by 33.3%, and was able to improve the solution of 19 out of the 25 instances considered. The logic-based Benders algorithm was able to solve 37 out of 51 Benchmark instances for an affine penalty function, and close them for the first time.

Acknowledgment

This work has been supported in part by the Office of Naval Research under Grant No. N00014-18-1-2129.

References

Althaus E, Caprara A, Lenhof HP, Reinert K (2006) A branch-and-cut algorithm for multiple sequence alignment. *Mathematical Programming* 105(2-3):387–425.

Andersen HR, Hadzic T, Hooker JN, Tiedemann P (2007) A constraint store based on multivalued decision diagrams. *International Conference on Principles and Practice of Constraint Programming*, 118–132 (Springer).

BaliBASE4 (2017) Balibase 4. URL http://www.lbgi.fr/balibase/, online; accessed 5-November-2017.

Becker B, Behle M, Eisenbrand F, Wimmer R (2005) Bdds in a branch and cut framework. *International Workshop on Experimental and Efficient Algorithms*, 452–463 (Springer).

Bergman D, Cire AA, van Hoeve W (2014) Mdd propagation for sequence constraints. *Journal of Artificial Intelligence Research* 50:697–722.

Bergman D, Cire AA, van Hoeve WJ, Hooker JN (2016) Discrete optimization with decision diagrams. *INFORMS Journal on Computing* 28(1):47–66.

Bouquet F, Jégou P (1995) Solving over-constrained csp using weighted obdds. *International Workshop on Over-Constrained Systems*, 293–308 (Springer).

Bryant RE (1986) Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on* 100(8):677–691.

Carrillo H, Lipman D (1988) The multiple sequence alignment problem in biology. *SIAM Journal on Applied Mathematics* 48(5):1073–1082.

Cayrol C, Lagasquie-Schiex MC, Schiex T (1998) Nonmonotonic reasoning: from complexity to algorithms. *Annals of Mathematics and Artificial Intelligence* 22(3-4):207–236.

Chakraborty A, Bandyopadhyay S (2013) Fogsaa: Fast optimal global sequence alignment algorithm. *Scientific reports* 3.

Cire AA, van Hoeve WJ (2013) Multivalued decision diagrams for sequencing problems. *Operations Research* 61(6):1411–1428.

Darwiche A, Marquis P (2004) Compiling propositional weighted bases. *Artificial Intelligence* 157(1-2):81–113.

Detienne B, Sadykov R, Tanaka S (2015) The two-machine flowshop total completion time problem: A branch-and-bound based on network-flow formulation. *7th Multidisciplinary International Conference on Scheduling: Theory and Applications*, 635–637.

Detienne B, Sadykov R, Tanaka S (2016) The two-machine flowshop total completion time problem: branch-and-bound algorithms based on network-flow formulation. *European Journal of Operational Research* 252(3):750–760.

Edgar RC (2004) Muscle: multiple sequence alignment with high accuracy and high throughput. *Nucleic acids research* 32(5):1792–1797.

Fischetti M, Toth P (1989) An additive bounding procedure for combinatorial optimization problems. *Operations Research* 37(2):319–328.

Hirschberg DS (1975) A linear space algorithm for computing maximal common subsequences. *Communications of the ACM* 18(6):341–343.

Hoda S, Van Hoeve WJ, Hooker JN (2010) A systematic approach to mdd-based constraint programming. *International Conference on Principles and Practice of Constraint Programming*, 266–280 (Springer).

Hooker JN (2002) Logic, optimization, and constraint programming. *INFORMS Journal on Computing* 14(4):295–321.

Hooker JN, Ottosson G (2003) Logic-based benders decomposition. *Mathematical Programming* 96(1):33–60.

Hung CL, Lin YS, Lin CY, Chung YC, Chung YF (2015) Cuda clustalw: An efficient parallel algorithm for progressive multiple sequence alignment on multi-gpus. *Computational biology and chemistry* 58:62–68.

Katoh K, Rozewicki J, Yamada KD (2017) Mafft online service: multiple sequence alignment, interactive sequence choice and visualization. *Briefings in Bioinformatics* .

Kececioglu J (1993) The maximum weight trace problem in multiple sequence alignment. *Annual Symposium on Combinatorial Pattern Matching*, 106–119 (Springer).

Kececioglu JD, Lenhof HP, Mehlhorn K, Mutzel P, Reinert K, Vingron M (2000) A polyhedral approach to sequence alignment problems. *Discrete applied mathematics* 104(1):143–186.

Kinable J, Cire AA, van Hoeve WJ (2017) Hybrid optimization methods for time-dependent sequencing problems. *European Journal of Operational Research* 259(3):887–897.

Lee CY (1959) Representation of switching circuits by binary-decision programs. *Bell System Technical Journal* 38(4):985–999.

Magis C, Taly JF, Bussotti G, Chang JM, Di Tommaso P, Erb I, Espinosa-Carrasco J, Notredame C (2014) T-coffee: tree-based consistency objective function for alignment evaluation. *Multiple Sequence Alignment Methods* 117–129.

Martin RK (1987) Generating alternative mixed-integer programming models using variable redefinition. *Operations Research* 35(6):820–831.

Martin RK, Rardin RL, Campbell BA (1990) Polyhedral characterization of discrete dynamic programming. *Operations research* 38(1):127–138.

McDaniel D, Devine M (1977) A modified benders' partitioning algorithm for mixed integer programming. *Management Science* 24(3):312–319.

Mott R (1999) Local sequence alignments with monotonic gap penalties. *Bioinformatics (Oxford, England)* 15(6):455–462.

Needleman SB, Wunsch CD (1970) 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.

Nuin PA, Wang Z, Tillier ER (2006) The accuracy of several multiple sequence alignment programs for proteins. *BMC bioinformatics* 7(1):471.

Reinert K, Lenhof HP, Mutzel P, Mehlhorn K, Kececioglu JD (1997) A branch-and-cut algorithm for multiple sequence alignment. *Proceedings of the first annual international conference on Computational molecular biology*, 241–250 (ACM).

Sievers F, Wilm A, Dineen D, Gibson TJ, Karplus K, Li W, Lopez R, McWilliam H, Remmert M, Söding J, et al. (2011) Fast, scalable generation of high-quality protein multiple sequence alignments using clustal omega. *Molecular systems biology* 7(1):539.

Smith TF, Waterman MS (1981) Identification of common molecular subsequences. *Journal of molecular biology* 147(1):195–197.

Thompson JD, Linard B, Lecompte O, Poch O (2011) A comprehensive benchmark study of multiple sequence alignment methods: current challenges and future perspectives. *PloS one* 6(3):e18093.

Vingron M, Waterman MS (1994) Sequence alignment and penalty choice: Review of concepts, case studies and implications. *Journal of molecular biology* 235(1):1–12.

Wang L, Jiang T (1994) On the complexity of multiple sequence alignment. *Journal of computational biology* 1(4):337–348.

Wegener I (2000) *Branching programs and binary decision diagrams: theory and applications*, volume 4 (SIAM).

Yen IEH, Lin X, Zhang J, Ravikumar P, Dhillon I (2016) A convex atomic-norm approach to multiple sequence alignment and motif discovery. *International Conference on Machine Learning*, 2272–2280.