Observations on the Feasibility of Exact Pareto Optimization with Applications to RNA folding

Cédric Saule and Robert Giegerich

Faculty of Technology and the Center for Biotechnology at Bielefeld University, Germany
cedric.saule@uni-bielefeld.de
robert@techfak.uni-bielefeld.de

Abstract. Pareto optimization combines independent objectives by computing the Pareto front of its search space, defined as the set of all candidates for which no other candidate scores better under both objectives. This gives, in a precise sense, better information than an artificial amalgamation of different scores into a single objective, but is more costly to compute.
We define a general Pareto product operator $*_{Par}$ on scoring schemes. Independent of a particular algorithm, we prove that for two scoring schemes A and B used in dynamic programming, the scoring scheme $A *_{Par} B$ correctly performs Pareto optimization over the same search space. We show that a "Pareto-eager" implementation of dynamic programming can achieve the same asymptotics as a single-objective optimization which computes the same number of results. For RNA structure prediction under the minimum free energy versus the maximum expected accuracy model, we show that the empirical size of the Pareto front remains within reasonable bounds. Without artificial amalgamation of objectives, and with no heuristics involved, Pareto optimization is faster than computing the same number of answers separately for each objective.

Keywords: Pareto optimization, dynamic programming, RNA structure

1 Introduction

In combinatorial optimization, we evaluate a search space X of solution candidates by means of an objective function ψ. Generated from some input data of size n, the search space X is typically discrete and has size $O(\alpha^n)$ for some α. Conceptually, as well as in practice, it is convenient to formulate the objective function as the composition of a *choice function* φ and a *scoring function* σ, $\psi = \varphi \circ \sigma$, computing $(\varphi \circ \sigma)(X) = \varphi(\{\sigma(x)|x \in X\})$ as the overall solution. The most common form of the objective function ψ is that σ evaluates the candidates to a score (or cost) value, and φ chooses the candidate which maximizes (or minimizes) this value. One or all optimal solutions can be returned, and with little difficulty, we can also define φ to compute all candidates within a threshold of optimality. This scenario is the prototypical case we will base our discussion on. However, it should not go unmentioned that there are other, useful types of "choice" functions besides maximization or minimization, such as computing score sums, full enumeration of the search space, or stochastic sampling from it.

DOI: 10.15455/CMSR.2014.0004

Multi-objective optimization arises when we have several criteria to evaluate our search space. Scanning the pizza space of our home town, we may be looking for the largest pizza, the cheapest, or the vegetarian pizza with the richest set of toppings. When we use these criteria in combination, the question arises exactly how we combine them.

Let us consider two objective functions $\psi_1 = \varphi_1 \circ \sigma_1$ and $\psi_2 = \varphi_2 \circ \sigma_2$ on the search space X and let us define different variants of an operator $*$ to designate particular techniques of combining the two objective functions.

- *Additive combination* $(\psi_1 *_+ \psi_2)$ optimizes over the sum of the candidates scores employed by φ_1 and φ_2. This is a natural thing to do when the two scores are of the same type, and optimization goes in the same direction, i.e. $\varphi_1 = \varphi_2 =: \varphi$; we define

$$\psi_1 *_+ \psi_2 = \varphi \circ (\sigma_1 + \sigma_2). \tag{1}$$

  In fact, this recasts the problem in the form of a single objective problem with a combined scoring function. This applies e.g. for real costs (money), that sum up in the end no matter where they come from. Gotoh's algorithm for sequence alignment under an affine gap model can be seen as an instance of this combination [6]. It minimizes the score sum of gap openings and gap extensions. However, often it is not clear how scores should be combined, and researchers resort to more general combinations.
- *Parametrized additive combination* $(\psi_1 *_{+\lambda} \psi_2)$ is defined as

$$\psi_1 *_{+\lambda} \psi_2 = \varphi \circ (\lambda\sigma_1 + (1-\lambda)\sigma_2). \tag{2}$$

  Here the extra parameter signals that there is something artificial in the additive combination of scores, and the λ is to be trained from data in different application scenarios, or left as a choice to the user of the approach. Such functions are often used in bioinformatics [12, 5, 8, 11, 19]. For example, the Sankoff algorithm scores joint RNA sequence and folding by a combination of base pairing (ψ_1) and sequence alignment (ψ_2) score [12]. RNAalifold scores consensus structures by a combination of free energy (σ_1) and covariance (σ_2) scores [9]. Covariance scores are converted into "pseudo-energies", and the parameter λ controls the relative influence of the two score components.
  This combination often works well in practice, but a pragmatic smell remains. Returning to our earlier pizza space example: It does not really make sense to add the number of toppings to the size of the pizza, or subtract it from the price, no matter how we choose λ. In a way, the factor λ manifests our discomfort with this situation.
- *Lexicographic combination* $(\psi_1 *_{lex} \psi_2)$ performs optimization on pairs of scores of potentially different type, not to be combined into a single score.

$$(\psi_1 *_{lex} \psi_2)(X) = (\varphi_1, \varphi_2)(\{(\sigma_1(x), \sigma_2(x)) | x \in X\}), \tag{3}$$

  where (φ_1, φ_2) optimizes lexicographically on the score pairs $(\sigma_1(x), \sigma_2(x))$. With the lexicographic combination, we define a primary and a secondary objective,

DOI: 10.15455/CMSR.2014.0004

seeking either the largest among the cheapest pizzas, or the cheapest among the largest – certainly with different outcomes. This is very useful, for example, when φ_1 produces a large number of co-optimal solutions. Having a secondary criterion choose from the co-optimals is preferable to returning an arbitrary optimal solution under the first objective, maybe even unaware that there were alternatives.

– *Pareto combination* $(\psi_1 *_{Par} \psi_2)$ must be used in the case when there is no meaningful way to combine or prioritize the two objectives. It may also be useful and more informative in the previous scenario, producing a set of "optima" and letting the user decide the balance between the two objectives *a posteriori*. The solution set it computes is the *Pareto front* of X. Taking φ_1 and φ_2 as maximization, the Pareto front operator **pf** is defined as

$$\mathbf{pf}(X) = \{(a,b) \in S | \nexists (a',b') \in X \setminus \{a,b\} \text{ with } a \leq a', b \leq b'\}. \quad (4)$$

In words: An element is in the Pareto front, if no other element scores strictly better in both dimensions. We define

$$(\psi_1 *_{Par} \psi_2)(X) = \mathbf{pf}\{(\sigma_1(x), \sigma_2(x)) | x \in X\}. \quad (5)$$

The combinations $*_+$ and $*_{+\lambda}$ are common practice and merit no theoretical investigation, as they reduce the problem to the single objective case. The lexicographic combination $*_{lex}$ has been studied in detail in [17].

Let us now turn to non-heuristic cases of Pareto optimization. We are aware of only a few cases where it has been advocated within a dynamic programming approach. It was used by Getachew et al. [2] to find the shortest path in a network given different time cost/functions, computing the Pareto front. The Pareto-Optimal Allocation problem was solved with dynamic programming by Sitarz [16]. In the field of bioinformatics, Schnattinger *et al.* [15, 14] advocated Pareto optimization for the Sankoff problem. Their algorithm computes $(\psi_1 *_{Par} \psi_2)$, where ψ_1 optimizes a sequence similarity score and ψ_2 optimizes base pair probabilities in the joint folding of two RNA sequences.

Pareto optimization in a dynamic programming approach brings about three specific problems:

(i) Does the Pareto combination of two objectives satisfy Bellman's principle of optimality, the prerequisite for all dynamic programming?
(ii) How to compute Pareto fronts both efficiently and incrementally, when proceeding from smaller to larger sub-problems?
(iii) What is the empirical size of the Pareto front, compared to its expected size?

Heretofore, these issues had to be solved ad-hoc with every approach employing Pareto optimization. Motivated by and generalizing on the work by Schnattinger *et al.*, we strive here for general insight in the use of Pareto optimization within dynamic programming algorithms.

The present article underlies the space constraints of a conference contribution. All proofs, algorithms, measurements, and further detail labeled "omitted" herein are available from the authors upon request. An extended manuscript including further experiments is in preparation.

DOI: 10.15455/CMSR.2014.0004

2 Background

We introduce Pareto sets together with some basic mathematical properties and algorithms. We restrict our discussion to Pareto optimization over value pairs, rather than vectors of arbitrary dimension, which can be defined in an analogous way. This is justified by the fact that what we are aiming at is a general mechanism of combining two objective functions in a Pareto fashion, used with dynamic programming.

2.1 Basic properties of Pareto optimization

We start from two sets A and B and their Cartesian product $C = A \times B$. A and B are totally ordered by relations $>_A$ and $>_B$, respectively . This induces a partial *domination* relation $\succ$ on C as follows. $(a,b) \succ (a',b')$ if $a >_A a'$ and $b \geq_B b'$, or $a \geq_A a'$ and $b >_B b'$. In words, the dominating element must be larger in one dimension, and not smaller in the other. In $X \subseteq C$, an element is *dominant* iff there is no other element in X that dominates it. We can now restate Eq. (4) in words as: The *Pareto front* of X, denoted $\mathbf{pf}(X)$, is the set of all dominant elements in X. The definition of **pf** actually depends on the underlying total orders, and we should write more precisely $\mathbf{pf}_{>_A,>_B}$, but for simplicity, we will suppress this detail until it becomes relevant.

It is easy to see that Pareto front is uniquely defined for all X, that the operator **pf** is idempotent,

$$\mathbf{pf}(\mathbf{pf}(X)) = \mathbf{pf}(X) \tag{6}$$

and that

$$\mathbf{pf}(X) \subseteq X \tag{7}$$

holds by definition. Note that **pf** is not monotone with respect to $\subseteq$.

A set $X \subseteq C$ is a *Pareto set* if $\mathbf{pf}(X) = X$. Algorithmically, we represent sets as lists, without duplicate elements. If these lists have no particular order, we still call them sets.

A *sorted list* is a subset of C sorted lexicographically in decreasing order. If a sorted list holds a Pareto set, we call it a *Pareto list*. Naturally, on sorted lists we can perform certain operations more efficiently, which must be balanced against the effort of keeping lists sorted.

Observation 1 A Pareto list sorted on the first dimension based on $>_A$ (i) is also sorted lexicographically in decreasing order, and at the same time (ii) is sorted based on $>_B$ in *increasing* order.

This is true because when the list l is a Pareto list and $(a,b) \in l$, there can be no other element (a,b') with $b \neq b'$. Therefore, (i) the overall lexicographic order is determined solely by $>_A$. and (ii) looking at the values in the second dimension alone, we find them in increasing order of $>_B$.

This implies an observation on the size of Pareto fronts over discrete intervals:

DOI: 10.15455/CMSR.2014.0004

Obervation 2 If A and B are discrete intervals of size N, then any Pareto set over $A \times B$ has $O(N)$ elements.

This is true because each decrease in the first dimension must come with an increase in the second component. On random sets, the expected size of the Pareto front of a set of size k follows the harmonic law, $H(k) = \sum_{i=1}^{k}(1/i)$ [7, 20].

By definition, the intersection of two Pareto sets is a Pareto set. This does not apply for Pareto set union, as elements in one Pareto set may be dominated by elements from the other. Therefore we define the *Pareto merge* operation

$$A \overset{p}{\vee} B := \mathbf{pf}(A \cup B) \tag{8}$$

Clearly, $\overset{p}{\vee}$ inherits commutativity from $\cup$.

Observation 3: Pareto merge associativity.

$$(A \overset{p}{\vee} B) \overset{p}{\vee} C = A \overset{p}{\vee} (B \overset{p}{\vee} C) \tag{9}$$

Proof is omitted. As a consequence, we can simply write $A \overset{p}{\vee} B \overset{p}{\vee} C$.

2.2 Pareto operation complexity

We consider operations on Pareto sets and lists, where k denotes their size. In dynamic programming, these operations are performed in the innermost loops of the algorithm, and hence they are crucial for overall efficiency. For algorithms implementing **pf** see Section 5.1.

2.3 Pareto merge in linear time

We now specify an implementation of the Pareto merge operation $\overset{p}{\vee}$ which makes use of the fact that its arguments are Pareto sets, represented as lists in decreasing order by the first component (and in increasing order by the second dimension).

$$[\,] \overset{p}{\vee} y = y$$
$$x \overset{p}{\vee} [\,] = x$$

DOI: 10.15455/CMSR.2014.0004

$$
\begin{aligned}
(a,b) : x \overset{p}{\vee} (c,d) : y \;=\; & \textbf{case}\ \ (a,b)?(c,d)\ \textbf{of} \\
& (>,>) \to (a,b) : (x \overset{p}{\vee} (dropWhile(\lambda(u,v).v \leq b), y)) \\
& (>,=) \to (a,b) : (x \overset{p}{\vee} y) \\
& (>,<) \to (a,b) : (x \overset{p}{\vee} ((c,d) : y)) \\
& (=,>) \to (a,b) : (x \overset{p}{\vee} (dropWhile(\lambda(u,v).v \leq b), y)) \\
& (=,=) \to (a,b) : (x \overset{p}{\vee} y) \\
& (=,<) \to (c,d) : ((dropWhile(\lambda(u,v).v \leq d), x) \overset{p}{\vee} y) \\
& (<,>) \to (c,d) : ((a,b) : x \overset{p}{\vee} y) \\
& (<,=) \to (c,d) : (x \overset{p}{\vee} y) \\
& (<,<) \to (c,d) : ((dropWhile(\lambda(u,v).v \leq d), x) \overset{p}{\vee} y)
\end{aligned}
$$

The function $dropWhile(p,l)$ walks down a list l until it finds an element that does not satisfy the predicate p. It returns this element and the remaining list in our case. We use it to eliminate elements smaller than b (resp. d) in the second dimension. At first glance, the combination of $\overset{p}{\vee}$ and $dropWhile$ reminds of an $O(n^2)$ algorithm, but this is not true. For input lists of length n_1 and n_2, where $n = n_1 + n_2$, the output list has at most length n. It requires at most $O(n)$ calls to $\overset{p}{\vee}$. $dropWhile$ requires $k+1$ calls when it deletes k elements, with $k \in O(n)$. However, each element deleted by $dropWhile$ safes a subsequent call to $\overset{p}{\vee}$. Overall, the number of steps remains within $O(n)$.

2.4 Pareto optimisation in dynamic programming

Dynamic Programming is governed by Bellman's principle of optimality, which the objective function $\psi = \varphi \circ \sigma$ must obey. Choice must distribute over scoring, which is computed incrementally from smaller to larger sub-solutions. For each *incremental* scoring function f that is used in the computation of σ,

$$\varphi(\{f(x), f(y))\}) = f(\varphi(\{x,y\}) \tag{10}$$

must hold. The above equation is formulated here for the simplest case: a unary function f and a choice function that returns a singleton result.

Dressing up Pareto optimization for dynamic programming, we must (i) formulate conditions under which it fulfills Bellman's principle, and (ii) show how the Pareto front of the overall solution can be computed incrementally and efficiently from Pareto fronts of sub-solutions. Heretofore, these issues had to be resolved with every dynamic programming algorithm that uses Pareto optimization, such as the one by Sitarz or Schnattinger *et al.* [16, 15]. Striving for general results for a whole class of algorithms, we resort to the *framework* of algebraic dynamic programming.

DOI: 10.15455/CMSR.2014.0004

3 Pareto optimization in ADP

Algebraic dynamic programming (ADP) is a framework for dynamic programming over sequential data [3]. Its declarative specifications achieve a perfect separation of the issues of search space construction, tabulation, and scoring, in sharp contrast to the traditional formulation of dynamic programming algorithms by matrix recurrences. Therefore, ADP lends itself to the investigation of Pareto optimization in dynamic programming in general, i.e. independent of a specific application domain.

Grammars and algebras describe ADP problems. See literature for theory and implementation [3, 13] . When grammar $\mathscr{G}$ describes the search space, and algebra A the scoring of candidates and the objective function h, an ADP algorithm is called in for input x in the form $\mathscr{G}(A,x)$, which is defined as

$$\mathscr{G}(A,x) = h(\{A(t) \mid t \in L(\mathscr{G}), yield(t) = x\}),$$

where $L(\mathscr{G})$ is the tree language generated by $\mathscr{G}$, for a candidate t, $yield(t)$ is the string of leaf symbols from the underlying input alphabet, and $A(t)$ is its score under algebra A.

3.1 Relation between Pareto and other products

As we show next, Pareto optimization can rightfully be considered the most general of the combinations discussed here. This holds strictly in the sense that from the Pareto front, the solutions according to the other combinations can be extracted.

Theorem 1. Pareto front subsumption theorem *For any grammar $\mathscr{G}$, scoring algebras A and B satisfying Bellman's principle, and input sequence x, consider the algebra combinations $A *_{+\lambda} B$, $A *_{lex} B$, and $A *_{Par} B$.*

1. $\mathscr{G}(A *_{+\lambda} B, x) = (\varphi_A *_{+\lambda} \varphi_B)(\mathscr{G}(A *_{Par} B, x))$
2. $\mathscr{G}(A *_{lex} B, x) = (\varphi_A *_{lex} \varphi_B)(\mathscr{G}(A *_{Par} B, x))$

The proof is omitted.

3.2 Preservation of Bellman's principle by the Pareto product

In this section we present our main theorem, showing that the Pareto product *always* preserves Bellman's principle. For the Pareto product to apply, we have the prerequisite that algebras A and B both maximize over a total order. In this situation, Bellman's principle specializes as follow:

Lemma 2. *If φ maximizes over a total order, it implies that all k-ary functions f for $k > 0$ are strictly monotone with respect to each argument.*

Theorem 3. The Pareto product preserves Bellman's principle.

Proof: *Dominated* solutions cannot become *dominant* by application of f. Details omitted.

DOI: 10.15455/CMSR.2014.0004

4 Implementation

The Pareto product can be implemented simply by providing the Pareto front operator $\varphi = \mathbf{pf}_{>_A>_B}$ as the choice function for the algebra product $(A *_{Par} B)$. This implementation can be improved by monitoring the status of intermediate results as lexicographically sorted lists.

We will describe these implementation issues by means of an example production which covers the relevant special cases. A tree grammar describing an ADP algorithm has an arbitrary number of productions, but their meaning is independent.

Let f, g, and h be a binary, a unary and a nullary scoring function from the underlying signature. A tree grammar rule such as

$$W \to f < X, Y > \mid g < Z > \mid h <>$$

specifies the computation of partial results for a subproblem of type (i.e. derived from) W from partial results already computed from subproblems of types X and Y, of type Z, or for an empty subproblem via a (constant) scoring function h. In general, signature functions may have arbitrary arity, and trees on the right-hand side can have arbitrary height. All this can be handled in analogy to what we do next.

We use the nonterminal symbols also as names for the list of subproblem solutions derived from them. Hence, we compute a list of answers $W = [w_1, w_2, ...]$ from $X = [x_1, x_2, ...]$ and so on. Note that h denotes a constant list, in most cases a singleton, but not necessarily so.

4.1 Standard implementation

Candidate lists are created by terminal grammar rules, by application of scoring functions to intermediate results, and by union of alternative answers from alternative rules for the same nonterminal.

We describe the standard implementation by three operators $\otimes, \oplus, \#$, named "build", "combine" and "select", which are defined as follows:

$$l \,\#\, \mathbf{pf} = \mathbf{pf}(l) \tag{11}$$

$$l_1 \oplus l_2 = l_1 +\!+ l_2 \tag{12}$$

$$\otimes(f, X, Y) = [f(x, y) | x \in X, y \in Y] \tag{13}$$

$$\otimes(g, X) = [g(x) | x \in X] \tag{14}$$

Hence, $\#$ simply applies the choice function (11), generally φ and **pf** in our specific case. $\oplus$ appends lists of solutions (12), and $\otimes$ builds solutions for bigger subproblems from smaller ones (13,14). Note that there is no requirement on the constant scoring function h. Typically, such a function generates a single element anyway. In general however, it may produce a list of alternative answers, and this need not be a Pareto list in the standard implementation.

Using this set of definitions, our example production describes the computation of

$$W = (\otimes(f, X, Y) \;\oplus\; \otimes(g, Z)) \;\oplus\; h) \,\#\, \mathbf{pf}$$

In section 5, we experiment with several implementations of **pf**, for the standard case.

DOI: 10.15455/CMSR.2014.0004

4.2 Pareto-eager implementation

The standard implementation applies a Pareto front operator after constructing a list of intermediate results. This list is built and combined from several sublists. By our main theorem, the Pareto front operation distributes over combinations of sublists, so we can integrate the # operator into the $\oplus$ operator. This has the potential that sizes of intermediate results are reduced as early as possible.

We define our operators as follows:

$$l \,\#\, \mathbf{pf} = l \tag{15}$$

$$l_1 \oplus l_2 = l_1 \overset{p}{\vee} l_2 \tag{16}$$

$$\otimes(f,X,Y) =$$

$$foldr \overset{p}{\vee} [] \; [[f(x,y)|x \in X]y \in Y] \tag{17}$$

$$\otimes(g,X) = [g(x)|x \in X] \tag{18}$$

$$h = \mathbf{pf}(h) \tag{19}$$

Now the # operator effectively skips the computation of the Pareto front, as this is already performed at all other places. The $\overset{p}{\vee}$ operation in Eq. (16) can assume argument lists to be Pareto lists already. In Eq. (18), the new list must be a Pareto list due to the strict monotonicity of g. In Eq. (17), the same holds for each intermediate list $[f(x,y)|x \in X]$ for each y, and we can merge them successively. Here, h must produce Pareto lists as inital answers (Eq. (19)).

In Section 2.3 we showed that $\overset{p}{\vee}$ can be implemented in $O(N)$, and therefore, each step in the Pareto-eager implementation takes linear time. This means that Pareto-optimization requires no intrinsic overhead, compared to a single optimization scheme which returns a comparable number of results. This is an encouraging insight, but leaves us an aspect to worry about: The size N of the Pareto front which is computed from an input sequence of length n.

4.3 Pareto front size

For a typical dynamic programming problem in sequence analysis, an input sequence of length n creates an exponential search space of $O(2^n)$. Still, by tabulation and reuse of intermediate subproblem solutions, dynamic programming manages to solve such a problem in polynomial time, say $O(n^r)$. The value of r depends on the nature of the problem, and when encoded in ADP, it is apparent as a property of the grammar which describes the problem decomposition [3]. We have $r = 2$ for simple sequence alignment, $r = 3$ for simple RNA structure prediction, $r = 4$ to $r = 6$ for RNA structures including various classes of pseudoknots, and so on. This all applies when a single, optimal result is returned.

For ADP algorithms returning the k best results, complexity must be stated more precisely as $O(n^r k^{r-1})$. As long as k is a constant, such as in k-best optimization, this does not change the asymptotics. However, computing all answers within p percent of

DOI: 10.15455/CMSR.2014.0004

the optimal score may well incur exponential growth of k. With Pareto optimization, the size k of the answer is not fixed in beforehand. The size of the Pareto front, for a set of size N, is expected to be $H(N)$ (cf. Section 2.1). Using $N \in O(2^n)$ and $H(N) \approx \log(N)$ [7], we can expect an effective size of the result sets in $O(n)$. Taking all things together, we can compute the Pareto front for an (algebraic) dynamic programming problem in $O(n^{2r-1})$ expected time, where n is input length and r reflects the complexity of the search space.

In applications, the size of the Pareto front need not follow expectation. We may achieve efficiency of $O(n^r k^{r-1})$ where $k << n$. Fortunately, in the application scenario of the next section, we find ourselves in this positive situation.

5 Applications

In this section, we report observations from different evaluations. (1) We use two real-world applications as black boxes, to measure runtime and space performance of Pareto optimization. (2) We use the same programs for assessing the empirical size of the Pareto front. Our test data consists of 331 RNA sequences of length 12 to 356 nucleotides, extracted from the full data set used in [1].

5.1 Algorithms implemented

We choose as test cases RNA folding algorithms, namely minimum free energy folding (MFE) and maximum expected accuracy folding (MEA). For each application, we can re-use grammars and algebras from the RNAshapes repository [10].

For the crucial operation **pf**, we tested several implementations. Let $k = |X|$.

- $\mathbf{pf}_{naive}(X)$ resembles insertion sort, but makes no effort to keep the result list sorted. The insertion phase adds an element to the end of the result list, if it is not dominated by any other element already in that list. This asymptotically in $O(k^2)$, and the worst case $\mathbf{pf}_{naive}$ actually occurs when X is already a sorted Pareto list!
- $\mathbf{pf}_{smooth}(X)$ computes a sorted result list. It requires $O(k)$ steps if X is sorted, and $O(k^2)$ steps in general.
- $\mathbf{pf}_{sort}$ also computes a sorted result list. It first sorts X lexicographically in $O(k \log k)$ steps, and from the result, obtains the Pareto front in $O(n)$, giving $O(k \log k)$ overall.
- $\mathbf{pf}_{isort}$ also computes a sorted list. In contrast to $\mathbf{pf}_{sort}$ it uses an insertion sort algorithm in $O(n^2)$, with the same worst case as $\mathbf{pf}_{naive}$.

5.2 Runtime and memory requirements.

We evaluate the performance of the Pareto front computation, using $\mathbf{pf}_{naive}(X)$, $\mathbf{pf}_{smooth}(X)$, $\mathbf{pf}_{sort}(X)$, and $\mathbf{pf}_{isort}(X)$. Note that all compute the same Pareto front, and hence have the same k in their asymptotics. For a fair comparison with two single-objective algorithms MFE and MEA, we use their versions $MFE(k)$ and $MEA(k)$, computing the k best structures. Here, k is set to the actual Pareto front size for the given input (which, of course, is only known because before we compute the Pareto front with the other

DOI: 10.15455/CMSR.2014.0004

algorithms). All programs are compiled by the Bellman's GAP compiler [13] using the same optimization options.

In Table 1 we show computation time and memory consumption, accumulated over all sequences and specifically for the longest sequence. These are our main observations:

Algebra	**Time (min)**	**Memory (GB)**	**Time (min)**	**Memory (GB)**
MFE(k) alone	71	163.68	5	1.16
MEA(k) alone	61	153.51	5	1.05
MFE(k) + MEA(k)	132(+)	163.68(max)	10	1.16 (max)
$(MFE *_{Par} MEA)$				
$\mathbf{pf}_{naive}$	8	197.28	0.22	1.28
$\mathbf{pf}_{smooth}$	9.5	192.79	0.5	1.28
$\mathbf{pf}_{sort}$	18	271.21	1	1.28
$\mathbf{pf}_{isort}$	32	250.21	3	2.11

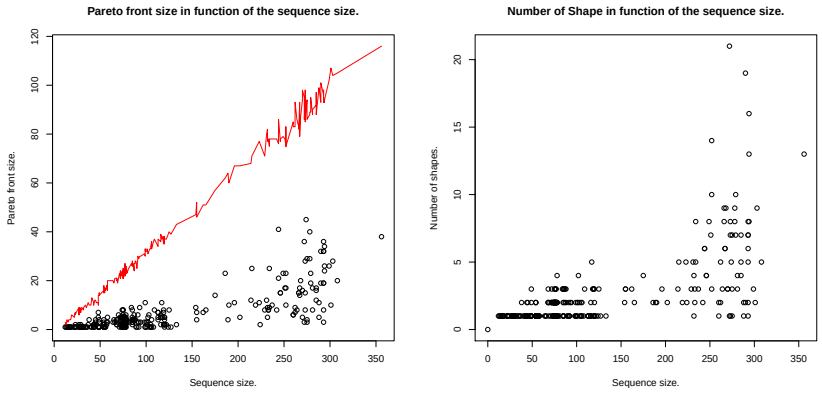
Table 1. Runtimes and memory requirements for $MFE(k)$, $MEA(k)$ (where k is the empirical Pareto front size for a given input), and their Pareto product $(MFE *_{Par} MEA)$, accumulated over 331 sequences (left) and for the longest sequence ($n = 356, k = 38$, right). The computations were performed by using Bellman's Gap.

(1) In terms of runtime, we find that the Pareto optimization performs not only better than the sum of the two independent optimizations, but also better than each of them individually. We attribute this to the fact that the Pareto algorithm adjusts itself to the size of the Pareto front, and this size tends[1] to be smaller than k for small subproblems. The search space itself, however, is exponentially larger than the Pareto front, and even on small sub-words it provides k near-optimals for $MFE(k)$ and $MEA(k)$ to spend computation on. This effect is strongest for our longest sequence, where $k = 38$ and the ratio of $(MFE(k) + MEA(k))/\mathbf{pf}_{naive} \approx 45$.

(2) The average case behaviour of $\mathbf{pf}_{naive}(X)$ is superior to all the sorting implementations of **pf**. This is an unexpected and interesting observation. We attribute this to a positive randomization effect. Comparing a new element to the extremal points of the Pareto front, maximal in one but minimal in the other dimension, is unlikely to establish domination. This what *always happens first* with sorted intermediate lists, and the element will walk along towards the middle of the list until it eventually is found to be dominated. In unsorted lists, a non-extremal element that dominates the new entry will, on average, be encountered earlier.

(3) Memory consumption of Pareto optimization is consistent over different implementations of **pf**. It is higher than either $MFE(k)$ or $MEA(k)$ alone, but clearly less than *the sum* of $MFE(k)$ and $MEA(k)$. This is better than expected, because after all, it solves both problems simultaneously.

[1] This is only a tendency – a final Pareto front of size k does not preclude intermediate results with Pareto fronts larger than k.



a b

Fig. 1. a) Empirical Pareto front size of $OverDangle(MFE *_{Par} MEA, x)$ as a function of $|x|$. The red line corresponds the $H(|X|)$, the expected Pareto front size according to the harmonic law, applied to the empirical value of $|X|$ for each x. b) Number of abstract RNA shapes [4] in the Pareto front, in function of the sequence size.

5.3 Pareto front size

The size of the Pareto front is of critical practical importance. Pareto front sizes in the hundreds, even for sequences of moderate length, would be prohibitive. This question can hardly be assessed analytically. Figure 1 shows our measurements. We observe the following:

- Much to our relief, Pareto front sizes are quite moderate, ranging round 10 for $n = 100$, 15 for $n = 200$, up to 45 for $n = 274$. Specifically, our longest sequence ($n = 356$) has a Pareto front of size 38.
- Variance is high (as expected) , and because of the strong variation, we did not fit a line through our measurement points. However, they are all dominated by the expected size of the Pareto front (red line).
- We did not smooth the graph for $H(|X|)$, such that it also demonstrates the variance in the search space sizes; just read the y-axis as a logarithmic scale for $e^{H(|X|)}$. The roughly linear behavior conforms with the theoretical analysis.

Abstract shape analysis is another established method (unrelated to Pareto optimization) to obtain an interesting set of near-optimal solutions from the search space [18]. We checked the ratio of the number of structures in the Pareto front, and the number of different abstract shapes they represent, but this ratio, ranging from 1 to ≈ 12, did no exhibit an obvious pattern. An experiment with $OverDangle(shape*(MFE *_{Par} MEA), x)$ is yet to be performed.

DOI: 10.15455/CMSR.2014.0004

6 Conclusion

We have shown that the exact Pareto front of two independent objectives can be computed by dynamic programming. The theoretical prerequisite for this is the preservation of Bellman's principle by the Pareto combination operator $*_{Par}$, established in our main theorem.

We have shown that by the Pareto-eager implementation, one can achieve Pareto optimization without an intrinsic penalty, compared to other optimizations which return a comparable number of results.

We have shown that empirically, for the case of RNA folding under different objectives, the size of the Pareto front remains within moderate bonds, clearly lower than theoretical expectation.

All in all, this says the Pareto optimization is practical for sequence analysis and moderate sequence sizes.

Acknowledgements

The authors would like to thank T. Schnattinger and H.A. Kestler for the discussions which inspired this generalization of their work. Thanks go to Stefan Janssen for help with the Bellman's GAP system, and to the anonymous CMSR reviewers for helpful feedback.

References

[1] M. Andronescu, A. Condon, H.H. Hoos, D.H. Mathews, and K.P. Murphy. Computational approaches for RNA energy parameter estimation. *RNA*, 16:2304–2316, 2010.

[2] T. Getachew, M. Kostreva, and L. Lancaster. A generalization of dynamic programming for Pareto optimization in dynamic networks. *Revue Française d'Automatique, d'Informatique et de Recherche opérationnelle. Recherche Opérationelle*, 34(1):27–47, 2000.

[3] R. Giegerich, C. Meyer, and P. Steffen. A discipline of dynamic programming over sequence data. *Science of Computer Programming*, 51(3):215–263, 2004.

[4] R. Giegerich, B. Voß, and M. Rehmsmeier. Abstract shapes of RNA. *Nucleic Acids Research*, 32(16):4843, 2004.

[5] J. Gorodkin, L.J. Heyer, and G.D. Stormo. Finding the most significant common sequence and structure motifs in a set of RNA sequences. *Nucleic Acid Research*, 25(18):3724–3732, 1997.

[6] O. Gotoh. An Improved Algorithm for Matching Biological Sequences. *Journal of Molecular Biology*, 162:705–708, 1982.

[7] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1994.

[8] J.H. Havgaard, R.B. Lyngsø, and J. Gorodkin. The FOLDALIGN web server for pairwise structural RNA alignment and mutual motif search. *Nucleic Acid Research*, 33:650–653, 2005.

[9] I.L Bernhart S.H.F. Hofacker and P.F. Stadler. Alignment of RNA base pairing probabilities matrices. *Bioinformatics*, 20(14):2222–2227, 2004.

DOI: 10.15455/CMSR.2014.0004

[10] Stefan Janssen, Christian Schudoma, Gerhard Steger, and Robert Giegerich. Lost in folding space? comparing four variants of the thermodynamic model for rna secondary structure prediction. *BMC Bioinformatics*, 12(1):429, 2011.

[11] D.H. Mathews. Predicting a set of minimal free energy RNA secondary structures common to two sequences. *Bioinformatics*, 21(10):2246–2253, 2005.

[12] D. Sankoff. Simultaneous solutions of the RNA folding, alignment and proto-sequences problems. *SIAM Journal on applied mathematics*, 45(5):810–825, 1985.

[13] Georg Sauthoff, Mathias Möhl, Stefan Janssen, and Robert Giegerich. Bellman's GAP – a language and compiler for dynamic programming in sequence analysis. *Bioinformatics*, 29(5):551–560, 2013.

[14] T. Schnattinger, U. Schoening, A. Marchfelder, and H.A. Kestler. RNA-Pareto: interactive analysis of Pareto-optimal RNA sequence-structure alignments. *Bioinformatics*, 29(23):3102–3104, 2013.

[15] T. Schnattinger, U. Schoening, A. Marchfelder, and H.A. Kestler. Structural RNA alignment by multi-objective optimization. *Bioinformatics*, 29(13):1607–1613, 2013.

[16] S. Sitarz. Pareto optimal allocation and dynamic programming. *Annals of Operational Research*, 172:203–219, 2009.

[17] P. Steffen and R. Giegerich. Versatile and declarative dynamic programming using pair algebras. *BMC Bioinformatics*, 6(1):224, 2005.

[18] Björn Voß, Robert Giegerich, and Marc Rehmsmeier. Complete probabilistic analysis of RNA shapes. *BMC Biology*, 4(1):5, 2006.

[19] Y. Wexler, C. Zilberstein, and M. Ziv-Ukelson. A study of accessible motifs and RNA folding complexity. *Journal of Computational Biology*, 14(6):856–872, 2007.

[20] M.A. Yukish. *Algorithms to identify Pareto points in Multi-dimensional data sets.* PhD thesis, The Pennsylvania State University, Graduate School, College of Engineering, 2004.

DOI: 10.15455/CMSR.2014.0004